Projektowanie Wirtualnych Krajobrazów w grze Grave Out:

Deformacja Mesh z Jobs , Netcode for Gameobjects oraz Proceduralne Generowanie Świata z wykorzystaniem systemu Gridu

Klasy GraveRandom oraz RandomUtility

Klasa GraveRandom została stworzona jako rozbudowa standardowej klasy .NET System.Random, oferując znacznie większe możliwości w zakresie generowania liczb losowych, oraz stałość niezależnie od wyjściowej platformy. Jej głównym zadaniem jest możliwość posiadania jednego spójnego seedu pomiędzy zarówno klientem jak i serwerem, używamy jej na przykład w celu tworzenia losowych terenów czy wydarzeń w grze "Grave Out".

Problemem, z którym się borykałem, była niestabilność spowodowana prawdopodobnie różnicami między kompilatorami wyjściowymi Mono a kompilatorami bazowego .NET na Windowsie. W związku z tym, postanowiłem skupić się na wypróbowanych metodach haszowania, które gwarantowały generowanie spójnych wartości całkowitych (integers), które później były wykorzystywane jako ziarna (seeds) dla poszczególnych komórek (cells) tworzących grid gry. Na załączonych obrazach znajdziecie więcej informacji na ten temat.

Rozwiązaniem, które zostało wybrane do generowania spójnych ziaren niezależnie od platformy, jest wykorzystanie algorytmu haszującego SHA256, który przekształca dowolny ciąg znaków wejściowych w ustalony zestaw bajtów. Algorytm ten jest uniwersalny i niezawodny, co pozwala na jego stosowanie w różnorodnych środowiskach, takich jak Mac, Linux, Windows oraz potencjalnych przyszłych wersjach na konsole gier. Metoda **ComputeConsistentSeed** w statycznej klasie jest odpowiedzią na te potrzeby.

```
// Helper method to compute a consistent seed from a string input
Frequently called  1 usage
private static int ComputeConsistentSeed(string input)
{
    // Use UTF8 Encoding for consistent byte representation across platforms
    byte[] inputBytes = Encoding.UTF8.GetBytes(input);

    using (SHA256 sha256 = SHA256.Create())
    {
        byte[] hashBytes = sha256.ComputeHash(inputBytes);

        // Convert the first 4 bytes of the hash into an integer to use as a seed
        int seed = BitConverter.ToInt32(hashBytes, startIndex: 0);

        // Ensure the seed is positive
        return Math.Abs(seed);
    }
}
```

Z kolei metoda NextFullRangeInt() w klasie GraveRandom jest przeznaczona do generowania liczby całkowitej obejmującej pełny zakres wartości int. Proces ten rozpoczyna się od wygenerowania dolnych 31 bitów liczby, co osiągane jest poprzez użycie metody random.Next(int.MinValue, int.MaxValue). Następnie generowany jest bit znaku, który jest przesuwany o 31 miejsc w lewo, by stać się najstarszym bitem (MSB), a całość jest łączona za pomocą operacji OR, co umożliwia reprezentację pełnego zakresu wartości typu int, zarówno dodatnich, jak i ujemnych.

```
// Method to generate a random boolean with a specified probability of being true
public bool NextBoolean(float probabilityOfTrue = 0.5f)
{
    return NextFloat() < probabilityOfTrue;
}

// Method to retrieve a random integer across the full int range
// Corrected method to retrieve a random integer across the full int range
Frequently called  3 usages
public int NextFullRangeInt()
{
    // Generate the lower 31 bits
    int lowerBits = random.Next(int.MinValue, int.MaxValue);

    // Generate the sign bit
    int signBit = random.Next(0, 2) << 31; // Shift left to make it the MSB

    // Combine the two parts, taking advantage of bitwise OR to include the sign bit
    return lowerBits | signBit;
}
```

Wszystkie te rozwiązania zostały wzięte pod uwagę przy tworzeniu gry, aby zapewnić graczom jak najbliższe doświadczenie nieskończonych możliwości świata, podobnie jak w grze "Minecraft", gdzie generowana jest niemal nieograniczona liczba unikalnych światów. Chociaż finalnie pozostałem przy 4 294 967 296 wariantach światów, jestem przekonany, że i to rozwiązanie zapewnia szerokie możliwości i zanurzenie w tworzonym uniwersum.

Generacja Dolnych 31 Bitów

Pierwszy krok to wygenerowanie dolnych 31 bitów liczby całkowitej. Używamy metody random.Next(int.MinValue, int.MaxValue), która generuje losową liczbę całkowitą między minimalną a maksymalną wartością dla 31-bitowych liczb całkowitych (bez uwzględnienia najwyższego bitu znaku). Wybór tych granic pozwala na uzyskanie szerokiego zakresu możliwych wartości, które są podstawą do dalszych operacji.

Generacja Bitu Znaku

Następnie generujemy bit znaku, który decyduje o tym, czy wynikowa liczba będzie dodatnia czy ujemna. Robimy to poprzez losowanie wartości 0 lub 1, a następnie przesuwamy tę wartość o 31 bitów w lewo (<< 31), co umieszcza bit znaku w najwyższej pozycji w 32-bitowej liczbie całkowitej. Taki przesunięty bit znaku będzie albo 0 (dla liczby dodatniej), albo 1 (dla liczby ujemnej).

Łączenie Części i Bitu Znaku

Ostatni krok to połączenie wygenerowanych dolnych 31 bitów z bitem znaku za pomocą operacji OR (|). Dzięki temu dolne 31 bitów reprezentuje wartość bezwzględną liczby, natomiast najwyższy bit decyduje o jej znaku. To pozwala na wygenerowanie każdej możliwej wartości liczby całkowitej, zarówno dodatniej, jak i ujemnej, obejmującej cały zakres wartości.

Klasa RandomUtility ułatwia inicjalizację i użycie instancji GraveRandom z ziarnem pochodzącym z danego ciągu znaków. Jest to szczególnie przydatne do generowania spójnych światów gier lub elementów na podstawie danych wejściowych od gracza lub predefiniowanych ciągów. Zawiera ona również pomocniczych metod które znajdziecie we wszystkich standardowych klas .NET System.Random.

Procedural Generation Grid

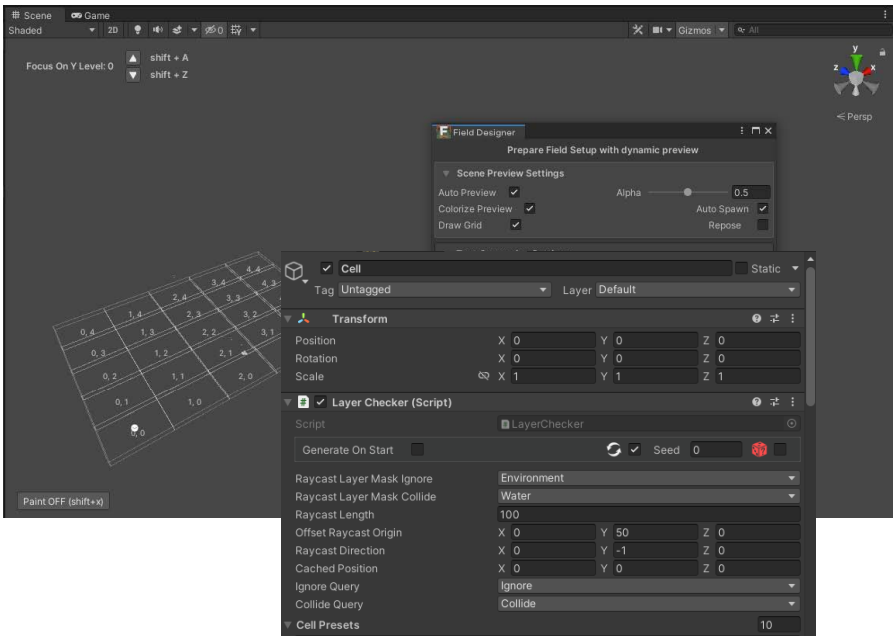
System Procedural Generation Grid (PGG) działa na zasadzie tworzenia świata gry za pomocą siatki komórek. Każda komórka jest generowana na podstawie określonych reguł, które można kontrolować za pomocą specjalnie przygotowanych prefabrykatów i skryptów, takich jak LayerChecker.

LayerChecker jest swoistym "opakowaniem" (wrapperem) klasy PGGGeneratorBase.
Większość logiki z LayerCheckera wraz z procesem powstawania całego systemu została przeniesiona do głównej klasy, którą jest "Cell Manager". O tej klasie opowiem zaraz więcej.

Główna idea generacji siatki polega na generowaniu świata w oparciu o zdefiniowane "rules", czyli zasady określające, jak poszczególne komórki mają zostać przekształcone w konkretne obiekty w świecie gry.

Proces generacji rozpoczyna się od utworzenia "Field Setup", który jest głównym elementem całego systemu. Zawiera on grupy modyfikatorów siatki, które odpowiadają za spawn (generowanie) lub usuwanie obiektów na siatce. Modyfikatory można grupować, co jest szczególnie przydatne w przypadku tworzenia bardziej złożonych struktur.

PGG nie generuje obiektów w trakcie sprawdzania komórek czy wykonania reguł spawnu. Generuje je na końcu procesu, kiedy wszystkie zasady z "Field Setup" są już obliczone. Dane dotyczące spawnu są przechowywane w dodatkowych klasach, które są wykorzystywane po zakończeniu obliczeń.



Layer Checker jest głównie kontenerem dla listy „Cell Presets” oraz zawiera parę metod pomocniczych takich jak RaycastWithParameters oraz RaycastCoroutine.

Klasa **CellPreset** przechowuje konfiguracje dla generowania komórek w środowisku proceduralnym gry, używając do tego celu różnorodnych ustawień i parametrów.

1. **PresetType**: Wyliczenie definiujące różne typy komórek (np. woda, trawa, rośliny, drzewa, budynki). To pozwala na łatwą rozbudowę o dodatkowe typy terenów czy obiektów, które mogą być potrzebne w grze.
2. **MinMaxDistance**: Struktura przechowująca informacje o minimalnej i maksymalnej odległości dla danego typu komórki, co może być użyte do określania, jak daleko od siebie mogą znajdować się obiekty danego typu.
3. **minMaxDistances**: Lista struktur **MinMaxDistance** służąca do przechowywania zakresów odległości dla poszczególnych typów komórek.
4. **probability**: Zmienna typu float określająca prawdopodobieństwo wygenerowania danej komórki w procentach.
5. **FieldPreset**: Pole typu **FieldSetup**, które może być przypisane do komórki, definiujące szczegółowe ustawienia generowania dla danego typu terenu.
6. **CalculationMethod**: Wyliczenie definiujące metodę obliczania, np. na podstawie dystansu od terenu.
7. **calculatedDistance**: Zmienna przechowująca obliczoną odległość, używaną do obliczeń w kontekście generowania.
8. **IsConstant**: Właściwość informująca, czy dana komórka jest stałym elementem (np. niezmieniająca się).
9. **raycastLayerMaskCheck**: Maski warstw używane w raycastingu, które mogą być przydatne do określenia, z których warstw należy odbijać promienie.
10. **distanceFrom**: Dystans od początku generowania.
11. **preventSize**: Wektor definiujący rozmiar obszaru, w którym nie będzie generowania.
12. **offset**: Wektor przesunięcia komórki.
13. **FieldSizeInCells**: Rozmiar pola generacji wyrażony w komórkach.
14. **provideDeformation**: Flaga określająca, czy komórka ma zapewniać deformacje terenu.
15. **deformationSettings**: Ustawienia deformacji, które mogą określać, jak teren ma być modelowany.

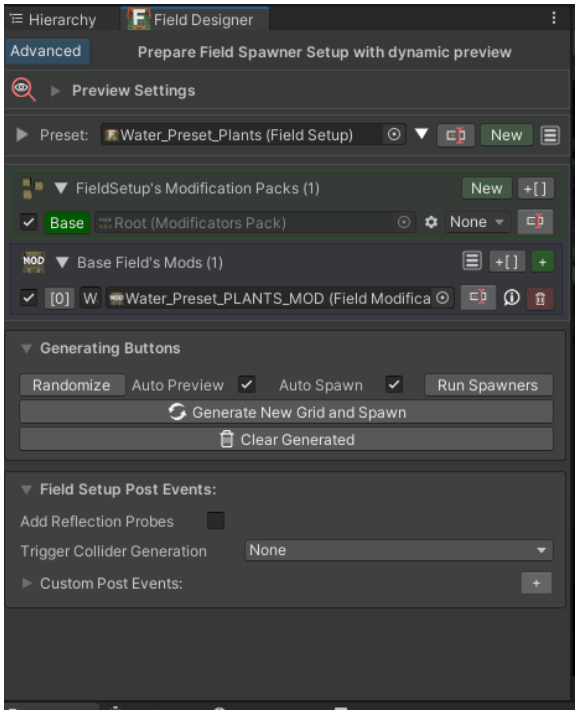
16. **CenterOrigin**: Flaga określająca, czy origin (środek) pola generacji ma być w centrum.
17. **isLimited**: Flaga określająca, czy istnieje limit generowania danej komórki.
18. **minLimitSpawnCount** i **maxLimitSpawnCount**: Minimalne i maksymalne ograniczenie liczby generowanych komórek.
19. **CalculatedValue**: Obliczona wartość na podstawie zdefiniowanego zakresu, służąca do generowania liczby komórek w danym obszarze.

Deformation Settings to klasa, który pozwala mi na precyzyjne kształtowanie terenu, a dzięki Unity Jobs mogę modyfikować mesh w sposób optymalizowany i wydajny. To zaawansowane podejście daje mi możliwość dokonywania skomplikowanych zmian terenu w czasie rzeczywistym, nie obciążając przy tym głównego wątku gry, co jest istotne dla zachowania płynności rozgrywki.

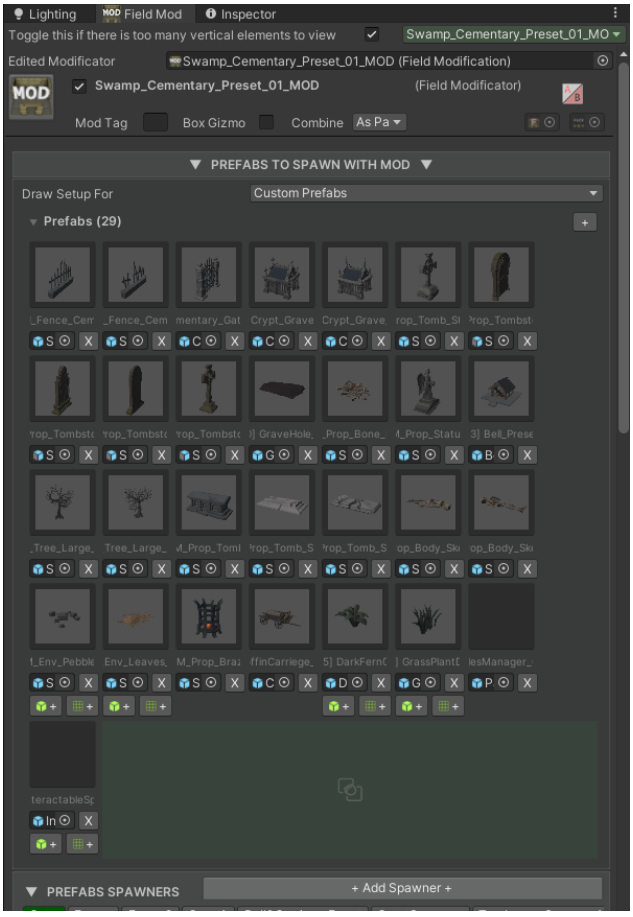
Courutine w Layer Checkerze sprawdza promieniem, czy warunki, które musi spełnić komórka, aby zespawnować swój "field setup", zostały spełnione. Przykładowe warunki to różnica wysokości terenu i wody, odstęp między monumentami etc.

Field setup i **Field Mod** to kontenery dla prefabów, spawnów i ich reguł lub eventów, koniecznych do wygenerowania prefaba w odpowiednim miejscu. Przykładowo prefab spawner krypt grobów posiada różne reguły. Na początku możemy określić, ile jednostek danego prefaba można zespawnować, sprawdzić pobliskie komórki i określić, czy muszą być w dopuszczalnym gridzie. Prevent spawns rule decyduje o tagach dla innych spawnerów i zabezpiecza możliwość spawnu w spawnerze krypty, aby nie doprowadzić do sytuacji, gdzie itemy "napiętrzą się" na siebie (patrz Rysunek 2).

- Field Setup

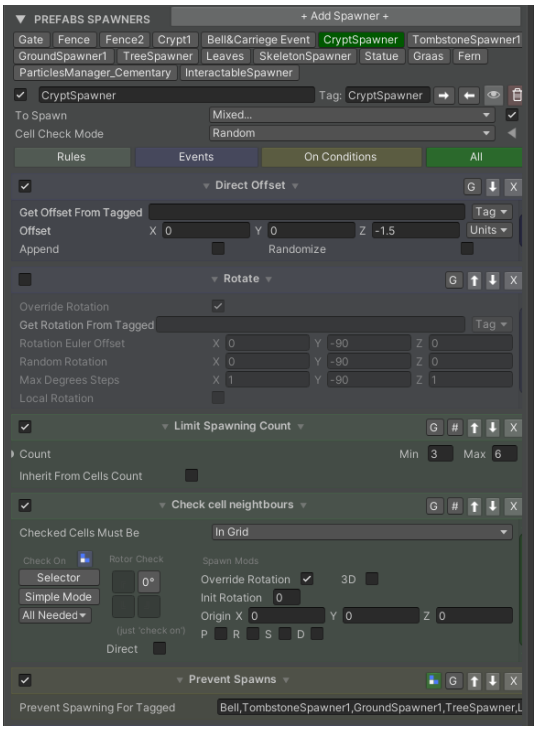


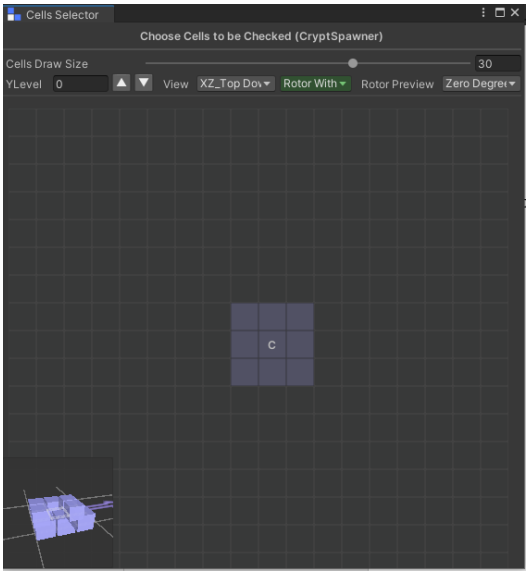
- FieldMod

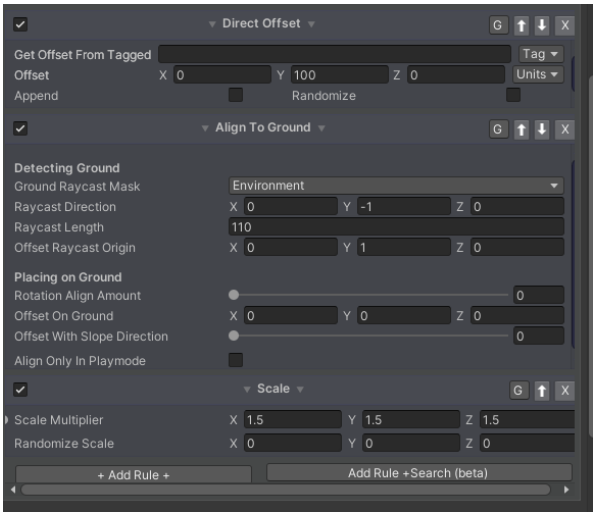


Direct offset np. zakłada, że podczas generacji gdy będziemy może mieć wysokość maksymalnie 100 jednostek w y, więc unosi go na wysokość 100 jednostek w osi y, aby mieć dobrą widoczność na wszystkie layery, które promień będzie sprawdzał (patrz Rysunek 3).

Następnie wchodzi do gry jedna z najważniejszych reguł - Align to Ground Rysunek 1, w której uzależniamy dopasowanie spawnowanego prefaba do konkretnego layera, określając slope, offset względem layera i inne pomocnicze wartości żeby wizualnie dostosować prefab który spawnimy do Terenu który zostaje zespawniony wcześniej przed innymi Obiektami jako podstawa.





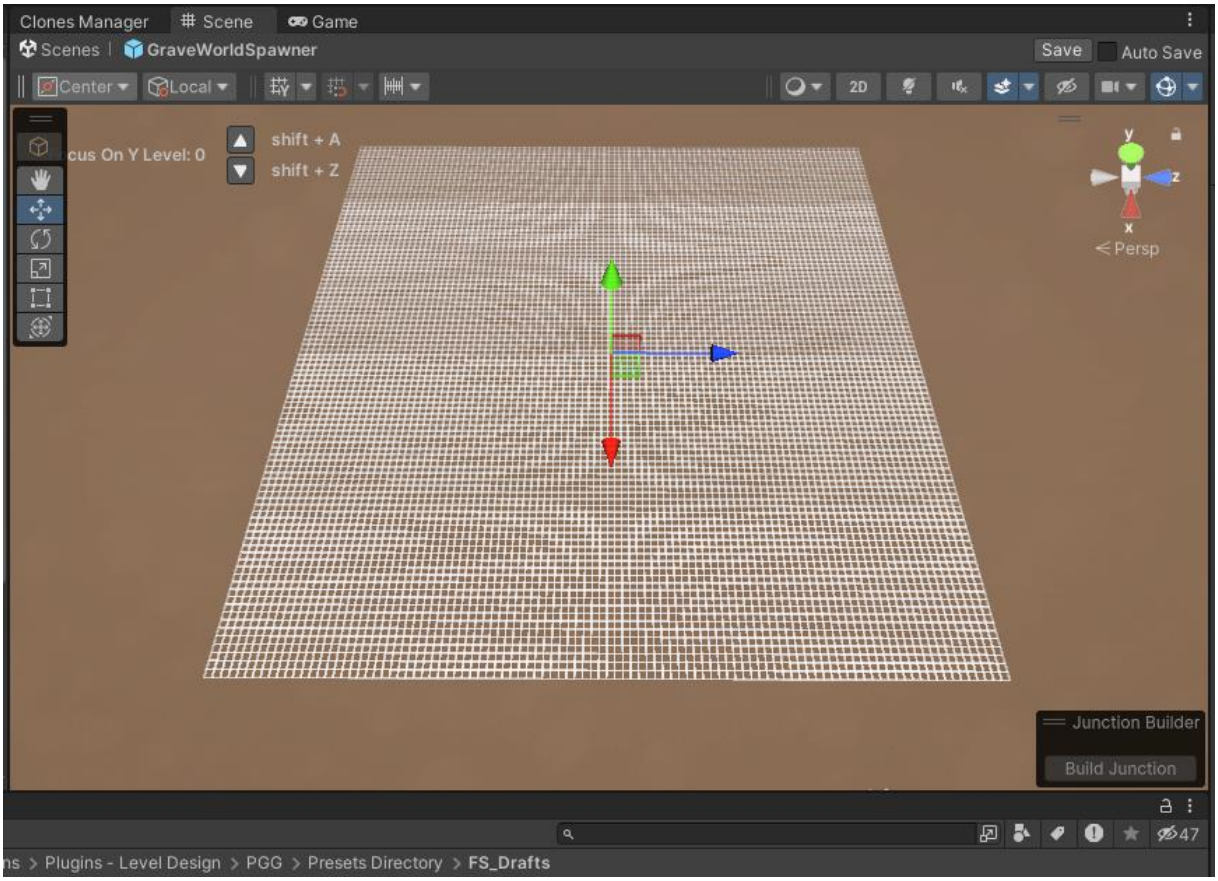


Klasy dla wstępnej Generacji

Klasa **AwakeField_Generator**, którą widzimy , jest istotną częścią systemu generowania proceduralnego, integrując różne rodzaje generatorów – terenu, wody oraz segmentów (chunków). Każdy generator, określony przez enum **GeneratorType**, odgrywa specyficzną rolę: generator terenu tworzy podwaliny dla świata gry, generator wody zajmuje się tworzeniem obszarów wodnych, a generator chunków tworzy zsegmentowane bloki terenu, które mogą być później wypełnione różnorodnymi prefabrykatami takimi jak drzewa czy budynki.

```
internal class AwakeField_Generator : MonoBehaviour
{
    public GeneratorContainer[] _generators;  Serializable

    Event function
    private void Start()
    {
        StartCoroutine( routine: WaitForServerAndCellManager());
    }

    Frequently called  1 usage
    private IEnumerator WaitForServerAndCellManager()
    {
        yield return new WaitUntil(() => CellManager.Instance != null && CellManager.Instance.IsSpawned);

        CellManager.Instance.Generators = _generators;

        if (NetworkManager.Singleton.IsServer) CellManager.Instance.StartGenerationProces();
    }


    public virtual void EnableSpawningForGeneratorInstance(SimpleFieldGenerator_GenImplemented generatorInstance)
    {
        foreach (var generatorContainer in _generators)
            if (generatorContainer.GeneratorInstance == generatorInstance)
            {
                generatorContainer.canSpawn = true;
                Debug.Log( message: $"Enabled spawning for {generatorContainer.Type}");
                break;
            }
    }
```

W skrypcie **GeneratorContainer** definiowany jest typ generatora i jego instancja, a także dostęp do ustawień pola **FieldPreset**, które określają, jakie prefabrykaty będą spawnione w danym segmencie świata. To podejście pozwala na modułową i elastyczną konstrukcję świata, umożliwiając generowanie struktur według zdefiniowanych parametrów. Na przykład, na ten moment są generowane chunki o wielkości 20 na 20 jednostek, każdy zawierający własny wrapper klasy „Layer Chacker” : PGGGeneratorBase, więc w sumie używamy tego samego systemu po to żeby dyktować własną wielkość domyślną świata i mieć wgląd na każdy pojedynczy chunk jako obszar do późniejszych odniesień dla różnych skryptów które będziemy używać . Zakładając że takich komórek będziemy Instantiate 100 x100 co w skali obecnego biomu gry jakim jest Bagno może pokryć obszar około 2.5 km².



Inicjalizacja w **AwakeField_Generator** odbywa się poprzez procedurę, która czeka na serwer i „Cell Manager”, co sugeruje, że Grave Out to gra sieciowa, gdzie generacja terenu musi być zsynchronizowana między serwerem a klientami. Kiedy warunki są spełnione, generatorów można używać do rozpoczęcia procesu generowania, co jest kluczowe dla utrzymania spójności świata w grze wieloosobowej.

Każdy generator ma flagę **canSpawn**, która decyduje, czy dany generator może być aktywowany do generowania obiektów. To pozwala na kontrolowane uruchamianie procesu generowania na serwerze, zapewniając, że każdy aspekt świata jest gotowy do interakcji z graczami, a całość procesu odbywa się w sposób zorganizowany i zoptymalizowany pod kątem wydajności.

IGeneration

Zrozumienie procesu, w jaki system Procedural Grid Generation (PGG) wykorzystuje niezbędne komponenty do tworzenia świata gry na podstawie gotowych presetów , jest kluczowe dla skutecznego projektowania i generowania w mojej grze. W tym kontekście, kluczową rolę odgrywa skrypt "IGeneration", będący częścią pakietu "Procedural Grid Generation" stworzonego przez FMposible. Mimo że nie jest to moje autorskie dzieło (nie znajdziecie jej w repozytorium), ta częściowo statyczna klasa stanowi istotne narzędzie umożliwiające głębsze zrozumienie całego procesu i mechanizmów pracy systemu. Skrypt "IGeneration" ułatwia proces spawnienia obiektów w oparciu o zdefiniowane reguły oraz analizę zależności określonych presetów , co pozwala na efektywne generowanie listy obiektów gotowych do użycia . Jej kluczową funkcjonalnością jest tworzenie listy obiektów zaimplementowanych interfejsem IGenerating, przygotowanych na podstawie zdefiniowanych wcześniej reguł i wyzwalaczy, zapisanych w serializowanych „field presetach” oraz zbiorach modyfikacji terenu „ field mods” .

Zbudowana lista, zawierająca wszystkie niezbędne dane, jest następnie przekazywana jako callback do klasy "PrefabAssetStaticData", którą w pełni opracowałem. Wykorzystuje ona najlepsze praktyki po głębszej analizie, takie jak zasoby adresowalne (Addressable Assets) oraz wzorzec Singleton dla skryptowalnych obiektów jednoznacznych (Scriptable Singleton), co znacznie ułatwia zarządzanie zasobami prefabrykatów i umożliwia szybki dostęp do potrzebnych instancji obiektu.

Następnie celem bloku kodu który widzicie poniżej jest przekazanie przygotowanych obiektów i wywołanie funkcji instancjonowania prefabrykatów z określonymi parametrami, takimi jak obiekt do spawnowania (spawn), docelowy kontener (target container), który jest odniesieniem do zserializowanych wartości. Po otrzymaniu zwrotnego wywołania callback, obiekt jest w pełni instancjonowany i zapisany, co pozwala na przejście do dalszej analizy i wykorzystania klasy "PrefabAssetStaticData". Dodatkowo, macierz używana w tym procesie umożliwia precyzyjne umiejscowienie spawnowanych obiektów w przestrzeni gry, co jest kluczowe dla utrzymania spójności generacji z uwzględnieniem poprzednio określonych zasad.

```
if (spawned == null)
{
    if (Application.isPlaying)
    {
        PrefabAssetStaticData.Instance.InstantiatePrefab(
            prefab: spawn.Prefab,
            callback: (GameObject instantiatedObject) =>
            {
                spawned = instantiatedObject;
            },
            spawn: spawn,
            targetContainer: targetContainer,
            generatorsCollected: generatorsCollected,
            cell: cell,
            preset: preset,
            listToFillWithSpawns: listToFillWithSpawns,
            gatheredToCombine: gatheredToCombine,
            gatheredToStaticCombine: gatheredToStaticCombine,
            matrix: matrix
        );
    }
```

PrefabAssetStaticData: Skrypt do optymalizacji generowania świata

PrefabAssetStaticData to klasa który wykorzystuje zaawansowane funkcje zarządzania zasobami Unity, w tym system Addressables, do dynamicznego ładowania i instancjonowania obiektów w świecie gry. Celem skryptu jest optymalizacja generowania rozległych światów, takich jak ten w projekcie Grave Out gdzie bazowo w zamyśle powierzchnia ma obejmować kilka biomów podobnie jak w grze „Valheim".

W kontekście finalnego rozwoju świata gry Grave Out, który powinien osiągnąć wielkość około 10 km^2, konieczność rozwoju tej klasy wynikała z niedostatecznej optymalizacji i wysokiego zużycia zasobów poprzednich rozwiązań. Spowodowałoby to długie czasy ładowania świata oraz niestabilność aplikacji. Kolejnym problemem przy standardowym wywołaniu instancji poprzez po prostu Instantiate jest zanieczyszczenie pamięci, gdzie korzystając z Addressable Assets, możemy je dynamicznie zwalniać i usuwać z pamięci, co pozwala na lepsze jej zarządzanie.

Skrypt ten składa się na takie kluczowe elementy jak:

- **Enum PrefabCategory:** Definiuje kategorie prefabów używanych w grze (np. teren, drzewa).
- **Klasa PrefabAssetGroup:** Grupuje prefabrykaty na podstawie kategorii i zawiera listę referencji do prefabów (PrefabAssetReference).
- **Klasa PrefabAssetReference:** Przechowuje informacje o pojedynczym prefabie, w tym referencję do AssetReferenceGameObject, wykorzystywaną do dynamicznego ładowania.
- **Klasa PrefabInstanceInfo:** Śledzi informacje o instancji prefabu (np. liczbę instancji, GUID zasobu).
- **Singleton PrefabAssetStaticData:** Przechowuje i zarządza danymi wszystkich grup prefabów. Posiada listę PrefabAssetGroups i metody do ich inicjalizacji i zarządzania.

Metoda InstantiatePrefab: Główna metoda instancjonowania prefabów. Używa słownika prefabAssetDictionary do znalezienia referencji prefabu i Addressables.InstantiateAsync do asynchronicznego ładowania i instancjonowania.


```
public void InstantiatePrefab(GameObject prefab, Action<GameObject> callback, SpawnData spawn,
    Transform targetContainer, List<IGenerating> generatorsCollected, FieldCell cell, FieldSetup preset,
    List<GameObject> listToFillWithSpawns, List<GameObject> gatheredToCombine,
    List<GameObject> gatheredToStaticCombine, Matrix4x4 matrix)
{
    if (prefabAssetDictionary.TryGetValue(prefab.name, out var prefabAssetReference))
    {
        var assetReference = prefabAssetReference.assetReference;

        if (Application.isPlaying)
            Addressables.InstantiateAsync(assetReference).Completed += handle =>
            {
                if (handle.Status == AsyncOperationStatus.Succeeded)
                {
                    var newObject = handle.Result;

                    // Call ProvideElse with the instantiated object and other parameters
                    ProvideAdressables(
                        newObject,
                        spawn,
                        targetContainer,
                        generatorsCollected,
                        cell,
                        preset,
                        listToFillWithSpawns,
                        gatheredToCombine,
                        gatheredToStaticCombine,
                        matrix
                    );
                    callback?.Invoke(newObject); // Pass result to callback

                    // Save only if the 'save' flag is true
                    if (prefabAssetReference.save)
                        // Always perform save logic
                        SaveKeyAndObject(newObject, assetReference);

                    // Additional logic if spawnNetworkObject is also true
                    if (prefabAssetReference.spawnNetworkObject)
                    {
                        // Check if the newObject has a NetworkObject component
                        var networkObjectComponent = newObject.GetComponent<NetworkObject>();
                        if (networkObjectComponent != null)
                        {
```

Metody zapisu i ładowania stanu obiektów: SaveKeyAndObject, SaveInstanceInfo, LoadInstanceInfo i ApplySavedStatesToInstantiatedObjects pozwalają na zapisywanie i odtwarzanie stanu obiektów.

Metoda ReparentAndSpawnWithCallback: Umożliwia zmianę rodzica obiektu sieciowego i odpowiednie spawnowanie.

Metoda ProvideAdressables: Statyczna metoda wywoływana po instancjonowaniu prefabu przez InstantiatePrefab. Przyjmuje załadowany prefab i dane spawnowania (pozycja, obrót), co pozwala na elastyczne umieszczanie obiektów.

Jeśli chodzi o optymalizację procesu tworzenia i zarządzania rozległym światem Grave Out, kluczowe znaczenie miało tutaj efektywne wykorzystanie systemu Addressable Assets. Dzięki temu systemowi udało się dynamicznie ładować i instancjonować prefabrykaty, znacząco zmniejszając zarówno czas ładowania, jak i zużycie pamięci, co jest szczególnie ważne w projektach o dużej skali do których urósł projekt nad którym pracuje.

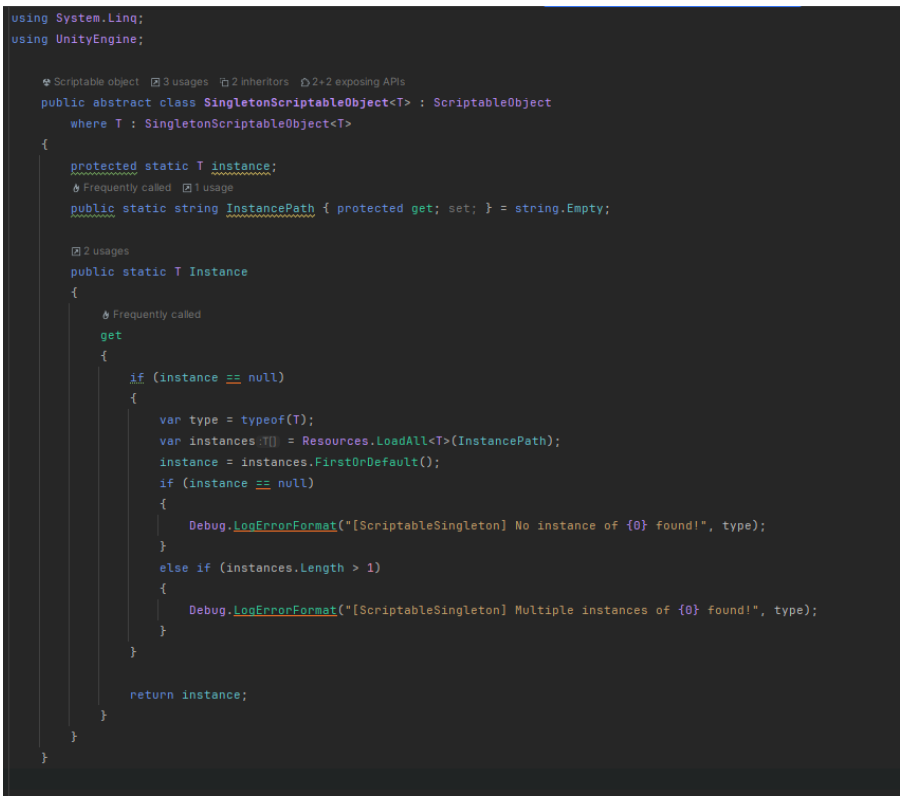
Wykorzystując Addressable Assets, zyskałem kontrolę nad procesem ładowania zasobów, co umożliwia opóźnione ładowanie tylko tych elementów, które są aktualnie potrzebne w danym momencie gry. To z kolei przekłada się na płynniejszą rozgrywkę i lepszą wydajność, nawet na urządzeniach o ograniczonej pamięci(docelowo, chyba jak każdy twórca chciałbym żeby moja gra była zoptymalizowana pod kątem odbiorców ze słabszą specyfikacją).

System ten pozwala także na bardziej zorganizowane zarządzanie zasobami. Prefabrykaty mogą być grupowane w logiczne kategorie, co ułatwia zarządzanie nimi i ponowne wykorzystanie w różnych częściach projektu. Dodatkowo, dzięki wykorzystaniu patternu singleton w PrefabAssetStaticData, mamy globalny dostęp do zarządzania zasobami prefabów, co jeszcze bardziej ułatwia proces tworzenia skomplikowanych scen i świata gry.

Dzięki asynchronicznemu ładowaniu w PrefabAssetStaticData prefabrykatów terenu, drzew, roślin i innych elementów w zależności od pozycji gracza, możliwe jest stworzenie wciągającego i rozległego świata gry, który jest zarówno bogaty w detale, jak i wydajny pod względem wykorzystania teraźniejszych i przyszłych zasobów które planuje dodawać wraz z rozwojem tego projektu.

SingletonScriptableObject

SingletonScriptableObject implementuje wzorzec projektowy singleton dla obiektów skryptowalnych (ScriptableObject) w Unity, umożliwiając globalny dostęp do pojedynczej instancji danej klasy przez cały cykl życia aplikacji. Dzięki wykorzystaniu Resources.LoadAll i filtrowaniu za pomocą Linq, skrypt dynamicznie wyszukuje i przypisuje instancję obiektu skryptowalnego, zapewniając, że tylko jedna instancja jest dostępna w projekcie. Użycie singletona w kontekście obiektów skryptowalnych jest korzystne, ponieważ zapewnia łatwy dostęp do współdzielonych zasobów i konfiguracji bez konieczności wielokrotnego ładowania tych samych zasobów, co mogłoby prowadzić do niepotrzebnego zużycia pamięci i spadków wydajności. Rozwiązanie to eliminuje również ryzyko powstania wielu instancji tego samego obiektu, co mogłoby prowadzić do niespójności danych w różnych częściach aplikacji.



```
using System.Linq;
using UnityEngine;

    Scriptable object  3 usages  2 inheritors  2+2 exposing APIs
public abstract class SingletonScriptableObject<T> : ScriptableObject
    where T : SingletonScriptableObject<T>
{
    protected static T instance;
    Frequently called  1 usage
    public static string InstancePath { protected get; set; } = string.Empty;

    2 usages
    public static T Instance
    {
        Frequently called
        get
        {
            if (instance == null)
            {
                var type = typeof(T);
                var instances = Resources.LoadAll<T>(InstancePath);
                instance = instances.FirstOrDefault();
                if (instance == null)
                {
                    Debug.LogErrorFormat("[ScriptableSingleton] No instance of {0} found!", type);
                }
                else if (instances.Length > 1)
                {
                    Debug.LogErrorFormat("[ScriptableSingleton] Multiple instances of {0} found!", type);
                }
            }

            return instance;
        }
    }
}
```

Cell Manager

Skrypt Cell Managera pełni kluczową rolę w dynamicznej analizie i interakcji z otoczeniem gry, zarządzając wszystkimi LayerCheckerami. Te sensory środowiskowe wykorzystują raycasting do zbierania danych o terenie i obiektach, identyfikując odpowiednie miejsca do generowania obiektów i wyzwalając deformację terenu w określonych warunkach.

Na podstawie analizy, LayerCheckery mogą tworzyć obiekty w świecie gry, kierując się zdefiniowanymi w CellPresetach regułami i warunkami. Te predefiniowane zestawy zawierają kryteria analizy, instrukcje dotyczące pojawiania się obiektów i ustawienia deformacji, stanowiąc swego rodzaju "zbiór reguł" dla LayerCheckerów.

Konfiguracja Raycastingu w skrypcie Cell Managera obejmuje parametry, takie jak raycastLayerMaskIgnore i raycastDirection, precyzujące interakcje raycastingu z otoczeniem.

Metoda GenerateObjects odpowiada za generowanie obiektów na podstawie analizy i wytycznych CellPresetów.

Proces generowania przebiega następująco:

1. Inicjalizacja: LayerCheckery są dodawane do Cell Managera, który zarządza całym procesem.
2. Raycasting i analiza: LayerCheckery zbierają dane o środowisku za pomocą raycastingu, a następnie analizują je w odniesieniu do warunków CellPresetów.
3. Wykonywanie akcji: W przypadku spełnienia warunków, LayerCheckery generują obiekty lub wyzwalają deformację terenu zgodnie z wytycznymi CellPresetów.

CellPresety wprowadzają element losowości do generowania, wykorzystując współczynnik prawdopodobieństwa, który wpływa na to, czy LayerChecker podejmie działanie na podstawie presetu, nawet gdy warunki zostaną spełnione. Presety oznaczone jako provideDeformation integrują się z systemami deformacji terenu, modyfikując go zgodnie z określonymi ustawieniami.

Współpraca LayerCheckerów i CellPresetów stanowi fundament proceduralnego generowania świata w Cell Managerze. Precyzyjne definicje w CellPresetach, w połączeniu z analizą danych środowiskowych i dynamicznymi akcjami LayerCheckerów,

Zarządzanie Konfiguracją i Instancją

- cellManagerData**:** Ta właściwość zapewnia dostęp do danych konfiguracyjnych Cell Managera z zewnętrznego pliku zasobów ("CellManagerData").
- Instance**:** Właściwość z wzorcem singleton, pozwalająca na jedyne aktywne wystąpienie Cell Managera wewnątrz sceny.
- OnAllCheckersProcessed**:** Delegat (rodzaj przekazanej funkcji) uruchamiany po przetworzeniu wszystkich LayerCheckerów.
- spawnPositionData**:** Zmienna przechowująca dane lokalizacji spawnu.
- Awake()**:** Metoda wywoływana przy tworzeniu obiektu, w tym przypadku:
    - Wykonuje kontrolę unikatowości instancji Cell Managera za pomocą wzorca singleton.
    - Zapewnia trwałość obiektu Cell Managera poprzez DontDestroyOnLoad.

Inicjalizacja w Sieci

- OnNetworkSpawn()**:** Metoda wywoływana przy sieciowym "narodzinach" obiektu, wykonywana jedynie na kliencie posiadającym kontrolę (IsOwner):
    - Pobiera główne ustawienia generatora ziarna losowego (seed) z cellManagerData.
    - Odczytuje z zasobów dane pozycji spawnu.
    - Inicjalizuje listy grobów i krypt.
    - Inicjalizuje generator liczb losowych z ustawionym ziarnem (seed).

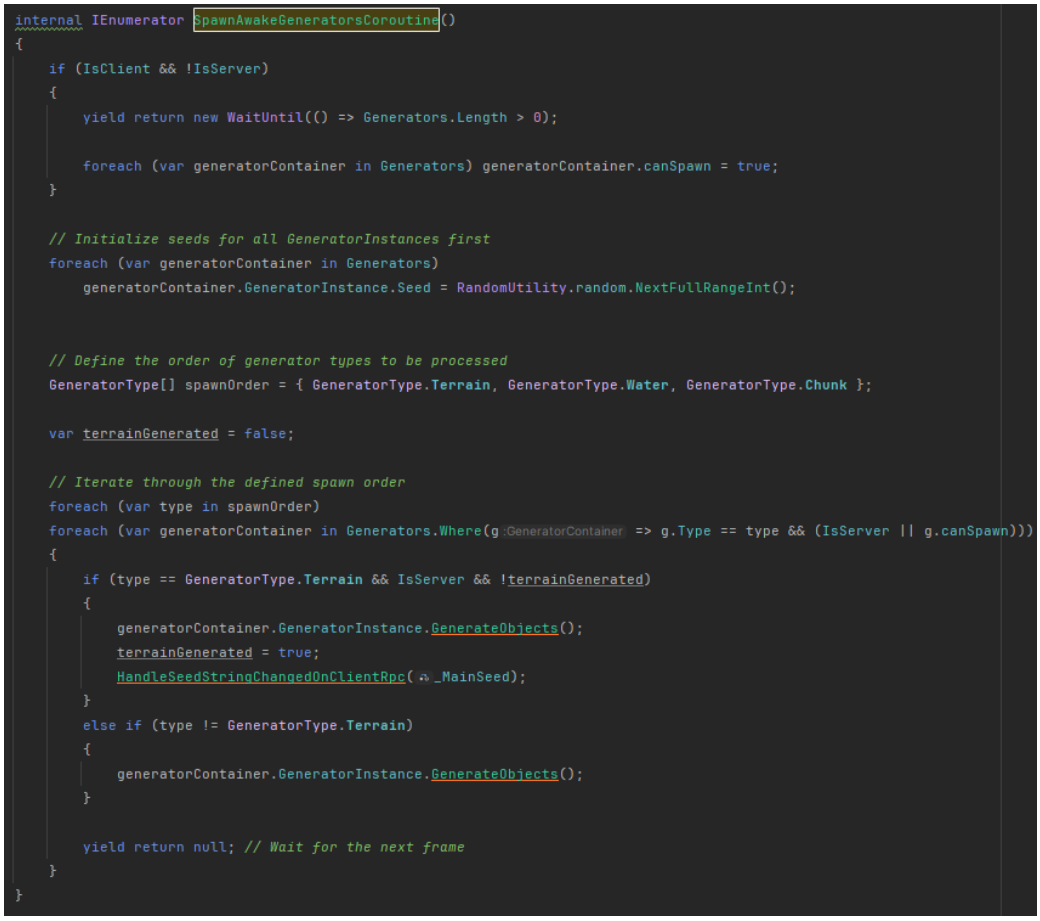
Proces Generacji

- StartGenerationProces()**:** Uruchamia sekwencję generacji świata uruchamianą jako coroutine (SpawnAwakeGeneratorsCoroutine).
- AddChecker()**:** Dodaje przekazany w parametrze LayerChecker do listy. Jeśli lista Checkerów osiągnie zadaną wielkość, wywoływane jest OnAllCheckersAdded.
- OnAllCheckersAdded()**:** Ta coroutine zarządza kolejnymi etapami generowania świata po dodaniu wszystkich Checkerów:
    - Czeka na zakończenie generowania początkowych obiektów (GenerateObjectSeeds).
    - Rozpoczyna proces dobierania Checkerów do modyfikacji terenu (SpawnPresetsCoroutine).
    - Przetwarza Checkery w grupach (ProcessCheckersInBatches).
    - Wywołuje delegat OnAllCheckersProcessed jako sygnał zakończenia pracy.
    - Tworzy nagrobki (SpawnCoffins).

```
public virtual void StartGenerationProces()
{
    StartCoroutine( routine: SpawnAwakeGeneratorsCoroutine());
}

1 usage
public virtual void AddChecker(LayerChecker checker)
{
    // Check if the checker is not already in the list
    if (!_layerCheckers.Contains(checker))
    {
        _layerCheckers.Add(checker);
        checkerCount++;

        if (checkerCount >= CellsChunkSize)
        {
            allCheckersAdded = true;
            StartCoroutine( routine: OnAllCheckersAdded());
        }
    }
}

Frequently called  1 usage
private IEnumerator OnAllCheckersAdded()
{
    // Wait until GenerateObjectSeeds is completed
    yield return StartCoroutine( routine: GenerateObjectSeeds());

    // Select checkers for deformation
    yield return StartCoroutine( routine: SpawnPresetsCoroutine());

    yield return StartCoroutine( routine: ProcessCheckersInBatches());

    OnAllCheckersProcessed?.Invoke();
```

Generowanie Obiektów Startowych

- SpawnAwakeGeneratorsCoroutine()**:** Coroutine odpowiedzialna za zarządzanie generowaniem początkowych obiektów w świecie.
    - Dla klientów (nie serwera): Oczekuje, aż lista generatorów zostanie wypełniona.
    - Inicjalizuje ziarna losowe dla wszystkich generatorów.
    - Definiuje kolejność generowania: Teren -> Woda -> Chunk.
    - Wywołuje GenerateObjects na każdym generatorze z uwzględnieniem kolejności i trybu (server/client).



```
internal IEnumerator SpawnAwakeGeneratorsCoroutine()
{
    if (IsClient && !IsServer)
    {
        yield return new WaitUntil(() => Generators.Length > 0);

        foreach (var generatorContainer in Generators) generatorContainer.canSpawn = true;
    }

    // Initialize seeds for all GeneratorInstances first
    foreach (var generatorContainer in Generators)
        generatorContainer.GeneratorInstance.Seed = RandomUtility.random.NextFullRangeInt();


    // Define the order of generator types to be processed
    GeneratorType[] spawnOrder = { GeneratorType.Terrain, GeneratorType.Water, GeneratorType.Chunk };

    var terrainGenerated = false;

    // Iterate through the defined spawn order
    foreach (var type in spawnOrder)
    foreach (var generatorContainer in Generators.Where(g GeneratorContainer => g.Type == type && (IsServer || g.canSpawn)))
    {
        if (type == GeneratorType.Terrain && IsServer && !terrainGenerated)
        {
            generatorContainer.GeneratorInstance.GenerateObjects();
            terrainGenerated = true;
            HandleSeedStringChangedOnClientRpc( _MainSeed);
        }
        else if (type != GeneratorType.Terrain)
        {
            generatorContainer.GeneratorInstance.GenerateObjects();
        }

        yield return null; // Wait for the next frame
    }
}
```

Obsługa Zmiany Stanu (Deformacja)

- HandleStateChange()**:** Zapewne obsługuje komunikację dotyczącą deformacji terenu w modelu sieciowym. Wykonywana tylko na serwerze.

Faza Deformacji: SpawnPresetsCoroutine()

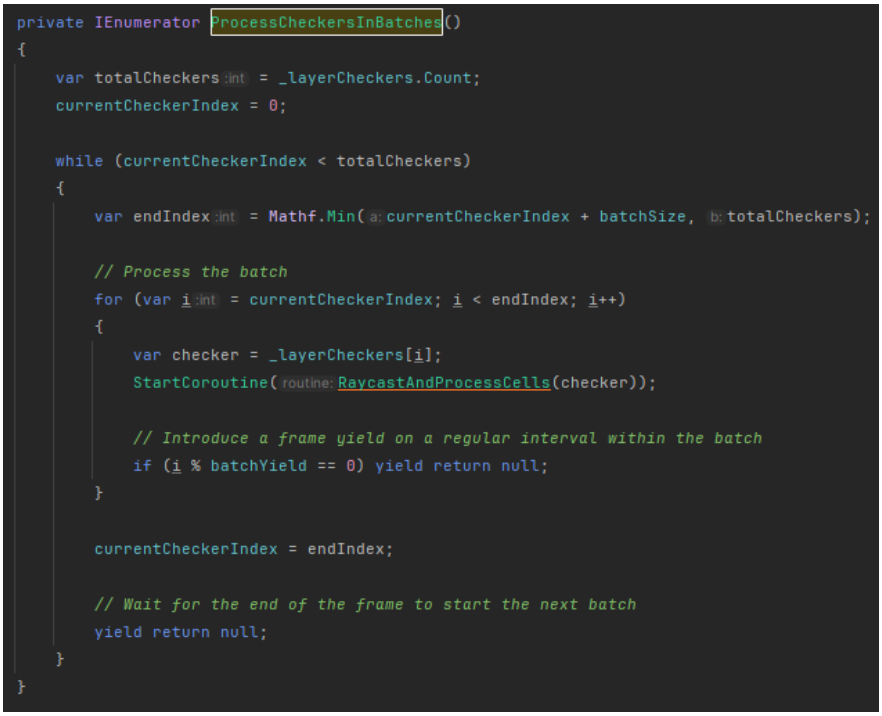
1. **Losowy Wybór Checkera:** Najpierw losowo wybiera się LayerChecker (LayerCheckerUtility.GetRandomChecker) z całej puli.
2. **Ograniczone Deformacje:** Jeśli wybrany checker istnieje:
    - Inicjalizuje liczbę pozostałych deformacji dla każdego CellPresetu z właściwościami isLimited i provideDeformation. Liczba ta jest pobierana z CalculatedValue.

3. **Przetwarzanie Checkerów:** Pętla przetwarza wszystkie LayerCheckery używając wcześniej losowo ustalonej kolejności (LayerCheckerUtility.GetRandomizedCheckers).
4. **Przetwarzanie Presetów:** Dla każdego LayerCheckera i jego CellPresetu spełniającego warunki deformacji i ograniczenia, uruchamia się coroutine RaycastAndProcessDeformationCells
5. **Warunki Deformacji:** Jeśli liczba deformacji dla nazwy (FieldPreset.name) danego presetu jest większa od 0:
    - Sprawdza wynik analizy terenu (RaycastAndProcessDeformationCells).
    - Jeśli analiza dała pozytywny wynik (raycastResults.Any(result => result.MatchingDistanceFound)):
    - Aktualizuje ustawienia deformacji (UpdateDeformationSettingsWorldOffset).
    - Wywołuje deformację (RaycastDeformer.Instance.Deform).
    - Oczekuje na zakończenie deformacji (WaitUntil(() => RaycastDeformer.Instance._State)).
    - Zmniejsza liczbę pozostałych deformacji dla presetu.
    - Generuje obiekty (checker.GenerateObjects).
    - Czyści dane analizy raycastu (raycastResults.Remove).

```
private IEnumerator SpawnPresetsCoroutine()
{
    var selectedChecker = LayerCheckerUtility.GetRandomChecker(_layerCheckers);
    var totalSpawnCounts = new Dictionary<string, int>();
    // Initialize spawn counts if a selected checker is available
    if (selectedChecker != null)
        foreach (var preset in selectedChecker.cellPresets.Where(p :CellPreset => p.isLimited && p.provideDeformation))
        {
            preset.InitializeCalculatedValue();
            totalSpawnCounts.Add(preset.FieldPreset.name, preset.CalculatedValue);
        }

    var randomizedCheckers :IEnumerable<LayerChecker> = LayerCheckerUtility.GetRandomizedCheckers(_layerCheckers);

    foreach (var checker in randomizedCheckers)
    foreach (var preset in checker.cellPresets.Where(p :CellPreset => p.isLimited && p.provideDeformation))
        if (totalSpawnCounts.TryGetValue(preset.FieldPreset.name, out var remainingSpawnCount :int) &&
            remainingSpawnCount > 0)
        {
            yield return StartCoroutine( routine: RaycastAndProcessDeformationCells(checker, preset));

            if (raycastResults.Any(result => result.MatchingDistanceFound))
            {
                var matchingResult = raycastResults.First(result => result.MatchingDistanceFound);
                // Perform actions based on the result of the first matching result
                matchingResult.Checker.UpdateDeformationSettingsWorldOffset(matchingResult.Preset);
                RaycastDeformer.Instance.Deform(matchingResult.Preset.deformationSettings, notifyStateChange: false);
                yield return new WaitUntil(() => RaycastDeformer.Instance._State);

                // Decrement the spawn count and generate objects
                totalSpawnCounts[preset.FieldPreset.name]--;
                checker.GenerateObjects(matchingResult.Preset.FieldPreset, matchingResult.Preset);
                raycastResults.Remove(matchingResult);
            }
        }
}
```

Rozdzielanie Pracy: ProcessCheckersInBatches()

- **Zarządzanie Wydajnością:** Metoda ta dzieli zadania na grupy (batchSize) unikając przeciążeń w jednej klatce.
- **Iteracja Checkerów:** Przetwarza LayerCheckery od currentCheckerIndex do (currentCheckerIndex + batchSize).
- **Analiza Raycast:** Dla każdego LayerCheckera uruchamia RaycastAndProcessCells jako coroutine.
- **Kontrolowane Oczekiwanie:** Wprowadza opóźnienie (yield return null) co batchYield obiektów aby pozwolić na obsługę innych procesów w grze.



```
private IEnumerator ProcessCheckersInBatches()
{
    var totalCheckers :int = _layerCheckers.Count;
    currentCheckerIndex = 0;

    while (currentCheckerIndex < totalCheckers)
    {
        var endIndex :int = Mathf.Min( a: currentCheckerIndex + batchSize,  b: totalCheckers);

        // Process the batch
        for (var i :int = currentCheckerIndex; i < endIndex; i++)
        {
            var checker = _layerCheckers[i];
            StartCoroutine( routine: RaycastAndProcessCells(checker));

            // Introduce a frame yield on a regular interval within the batch
            if (i % batchYield == 0) yield return null;
        }

        currentCheckerIndex = endIndex;

        // Wait for the end of the frame to start the next batch
        yield return null;
    }
}
```

Szczegółowa Analiza: RaycastAndProcessDeformationCells()

1. **Raycast Checkera:** Oczekuje na wynik RaycastCoroutine uruchomionej w LayerCheckerze.
2. **Podwójny Raycast:** Pobiera informacje z raycastów (GetIgnoreHit, GetColideHit).
3. **Warunek Analizy:** Przetwarza dalej tylko jeśli oba raycasty trafiły.
4. **Sprawdzenie Odległości:** Wywołuje CheckMatchingDistance aby ustalić czy odległość odpowiada kryteriom danego CellPresetu.
5. **Dane Wyniku:** Tworzy RaycastResult przechowujący wynik analizy i dodaje do raycastResults.

```
private IEnumerator RaycastAndProcessDeformationCells(LayerChecker checker, CellPreset specificPreset)
{
    yield return checker.StartCoroutine(routine: checker.RaycastCoroutine());

    var hit1 = checker.GetIgnoreHit();
    var hit2 = checker.GetColideHit();

    // Process only if both raycasts hit
    if (hit1.collider != null && hit2.collider != null)
    {
        // Check if the checker's distance matches the specific preset's criteria
        var matchingDistanceFound bool = CheckMatchingDistance(checker, specificPreset, hit1, hit2);

        // Create a new RaycastResult instance and add it to the list
        var result = new RaycastResult
        {
            MatchingDistanceFound = matchingDistanceFound,
            Checker = checker,
            Preset = specificPreset
        };

        checker._FieldSpawned = true;
        raycastResults.Add(result);
    }
    else
    {
        // Create a new RaycastResult instance and add it to the list
        var result = new RaycastResult
        {
            MatchingDistanceFound = false,
            Checker = checker,
            Preset = specificPreset
        };
        raycastResults.Add(result);
    }
}
```

Analiza i Generacja: RaycastAndProcessCells()

1. **Oczekiwanie Na Raycast:** Oczekuje na wynik RaycastCoroutine
2. **Podwójny Raycast:** Pobiera informacje z raycastów.
3. **Brak Trafień:** Kończy przetwarzanie Checkera jeśli raycasty nie trafiły w cel.
4. **Przetwarzanie Presetów:** Przechodzi przez CellPresety Checkera, które nie są isLimited i nie były jeszcze użyte (!checker._FieldSpawned).
5. **Losowa Szansa:** Używa prawdopodobieństwa (preset.probability) aby ustalić czy aktywować presetu
6. **Sprawdzenie Odległości:** Wywołuje CheckMatchingDistance
7. **Generowanie:** Jeśli warunki odległości i szansy zostały spełnione, wywołuje checker.GenerateObjects.

```
private IEnumerator RaycastAndProcessCells(LayerChecker checker)
{
    // Begin the raycast coroutine and wait for it to finish.
    yield return checker.StartCoroutine(routine: checker.RaycastCoroutine());

    var hit1 = checker.GetIgnoreHit();
    var hit2 = checker.GetColideHit();

    // If either raycast did not hit, skip processing.
    if (hit1.collider == null || hit2.collider == null) yield break;

    // Process hits for each preset.
    foreach (var preset in checker.cellPresets.Where(p :CellPreset => !p.isLimited && !checker._FieldSpawned))
    {
        // Roll the dice for each preset
        float roll = RandomUtility.random.Next(0, 100); // Random value between 0 and 100
        if (roll <= preset.probability && CheckMatchingDistance(checker, preset, hit1, hit2))
            // If the roll is within the probability range and matches distance, process this preset
            checker.GenerateObjects(preset.FieldPreset, preset);
    }
}
```

Wpływ stosowania GraveRandom

```
private IEnumerator GenerateObjectSeeds()
{
    foreach (var checker in _layerCheckers)
        if (checker != null)
            checker.Seed = RandomUtility.random.NextFullRangeInt();

    yield break;
}
```

Metoda GenerateObjectSeeds() odgrywa kluczową rolę, gdyż tutaj następuje ustawienie indywidualnych seedów dla każdego LayerCheckera. Zastosowanie GraveRandom w połączeniu z RandomUtility.InitializeRandomWithSeed ma na celu zapewnienie dwóch rzeczy.

Gdy podamy ten sam łańcuch wejściowy (seedString) do InitializeRandomWithSeed, uzyskamy identyczną sekwencję liczb losowych. Pozwala to na odtworzenie całego przebiegu generacji.

Z użyciem algorytmu hashującego SHA256 z ComputeConsistentSeed, możemy zagwarantować, że ten sam seed (ziarno) wygeneruje te same liczby losowe niezależnie od platformy sprzętowej, na której działa gra.

Wewnątrz metody GenerateObjectSeeds(), pętla przechodzi przez listę LayerCheckerów. Dla każdego LayerCheckera:

Pobierany jest losowy seed używając random.NextFullRangeInt() z klasy GraveRandom. Funkcja ta wykorzystuje wcześniej zainicjalizowany obiekt GraveRandom, pozwalając na wygenerowanie pełnego zakresu liczb losowych typu integer (int).

Przypisany seed jest przechowywany we właściwości Seed danego LayerCheckera.

Różne seedy dla LayerCheckerów skutkują różnorodnością w ich analizie terenu i późniejszych działaniach.

W skrócie : *Jeśli zapamiętasz główny string użyty w InitializeRandomWithSeed (a w konsekwencji cały zestaw unikalnych seedów dla LayerCheckerów), możesz dokładnie odtworzyć dany wygenerowany świat.*

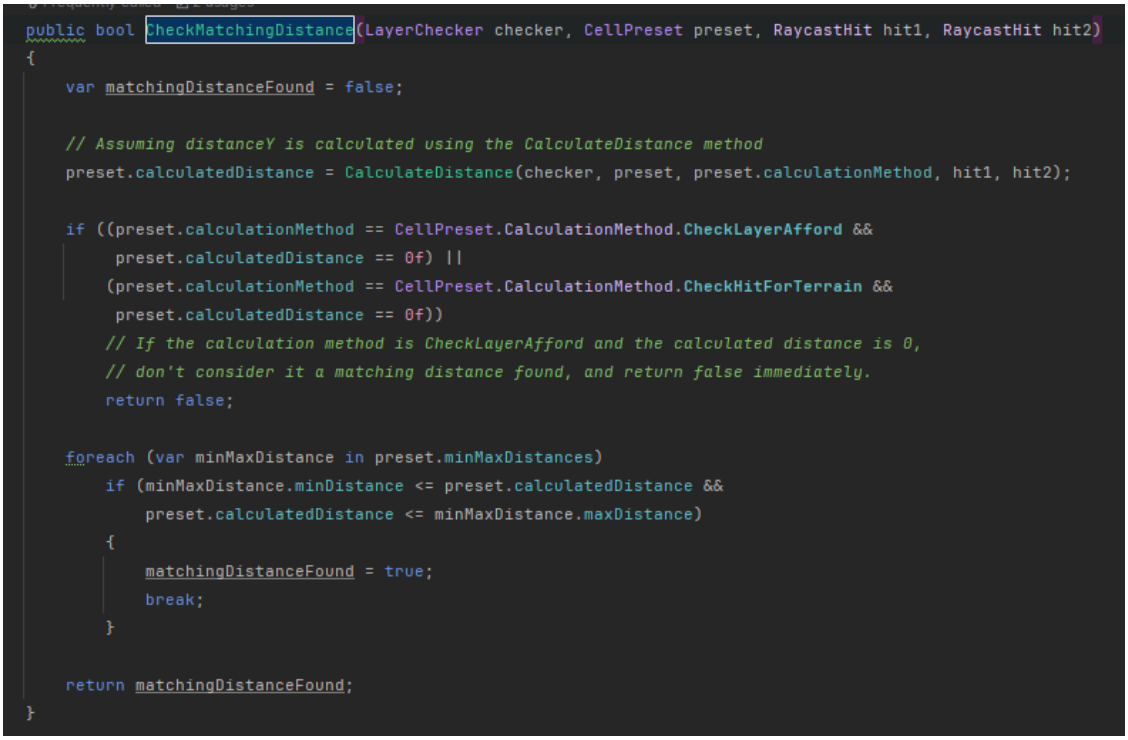
```
public static void InitializeRandomWithSeed(string seedString)
{
    int seed = ComputeConsistentSeed(seedString);
    random = new GraveRandom(seed); // Initialize the GraveRandom instance with the consistent seed
}
```

Seed każdego LayerCheckera zasadniczo wpływa na **cały** proces decyzyjny i wyniki, które następują później. Oto jak:

Metoda RaycastCoroutine() w LayerCheckerach zależna jest od liczb losowych do określenia kierunku, długości raycastów itd. Różne seedy oznaczają odmienne raycasty.

```
public IEnumerator RaycastCoroutine()
{
    var hitSomething1 bool =
        RaycastWithParameters(ray1, out hit1, raycastLength, raycastLayerMaskIgnore, ignoreQuery);

    // Early return if the first raycast doesn't hit anything
    if (!hitSomething1) yield break;


    var hitSomething2 bool =
        RaycastWithParameters(ray2, out hit2, raycastLength, raycastLayerMaskCollide, collideQuery);


    // Early return if the second raycast doesn't hit anything
    if (!hitSomething2) yield break;
}
```

CheckMatchingDistance() może porównywać wartości losowe z wartościami obliczanymi dla danego terenu.



```
public bool CheckMatchingDistance(LayerChecker checker, CellPreset preset, RaycastHit hit1, RaycastHit hit2)
{
    var matchingDistanceFound = false;

    // Assuming distanceY is calculated using the CalculateDistance method
    preset.calculatedDistance = CalculateDistance(checker, preset, preset.calculationMethod, hit1, hit2);

    if ((preset.calculationMethod == CellPreset.CalculationMethod.CheckLayerAfford &&
         preset.calculatedDistance == 0f) ||
        (preset.calculationMethod == CellPreset.CalculationMethod.CheckHitForTerrain &&
         preset.calculatedDistance == 0f))
        // If the calculation method is CheckLayerAfford and the calculated distance is 0,
        // don't consider it a matching distance found, and return false immediately.
        return false;

    foreach (var minMaxDistance in preset.minMaxDistances)
        if (minMaxDistance.minDistance <= preset.calculatedDistance &&
            preset.calculatedDistance <= minMaxDistance.maxDistance)
        {
            matchingDistanceFound = true;
            break;
        }

    return matchingDistanceFound;
}
```

Prawdopodobieństwo (preset.probability) określa, czy LayerChecker podejmie akcję na podstawie presetu (nawet jeśli inne kryteria odległości itd. są spełnione).

GenerateObjectSeeds() wraz z GraveRandom odgrywa kluczową rolę w zapewnieniu, aby generacja proceduralna w CellManagerze była zarówno **zróżnicowana** i **deterministyczna**.

Unikalne seedy przypisane LayerCheckerom mają kaskadowy wpływ na ich indywidualne analizy, decyzje oraz generowanie obiektów, ostateczne kształtując świat w moim projekcie.

Raycast Deformer

In Progress