ДОДАТОК А

Графічний матеріал кваліфікаційної роботи

Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Кафедра ЕОМ

Кваліфікаційна робота на тему: «Модель процедурної генерації програмними засобами для спрощення розробки додатків»

Виконав: ст. гр. СПм-21-1 Кісь О.В.

Керівник: Ляшенко О.С.

Актуальність проблеми

З кожним роком все більше уваги приділяється ігровій індустрії. Буквально за кілька років, ігрова індустрія стала на стільки популярною, що, часом, деякі ігри перевершують по бюджету сучасні голлівудські фільми. Створення ігрових рівнів є багатогранною задачою, актуальною для більшості випущених ігор. Найчастіше створенням рівнів займається окрема людина - дизайнер рівнів. При розробці великих проектів створенням рівнів може займатися і команда людей. Дизайн рівнів як процес включає в себе спектр проблем. Одним з найбільш істотних з них є обсяг роботи, необхідний для створення всіх рівнів гри. Одним з методів вирішення данное проблеми є використання процедурної генерації. Процедурна генерація часто використовується для створення ігрових рівнів.

Постановка задачі

Мета роботи : Створення алгоритму з додатком, який дозволить контролювати наповнення контенту при генерації нових підземель roguelike стилю та доступу до них через згенерований ландшафт.

Для досягнення поставленої мети, треба виконати такі задачі:

- Провести аналіз існуючих алгоритмів генерації
- Програмна розробка алгоритмів генерації
- Тестування отриманої реалізації

3

Аналіз існуючих розробок

Гра «World of Warcraft: Shadowlands»

Гра «Valheim»

Гра «No Man's Sky»

4

Аналіз існуючих алгоритмів

Алгоритми генерації підземель:

- На основі BSP-дерева
- Клітинний автомат
- Drunkard walk

Алгоритми генерації ландшафту:

- Пагорбний
- Diamond-Square
- На основі Шуму Перліна

5

Алгоритм на основі Шуму Перліна

Шум Перліна реалізують як дво-, три-, чотиривимірна функція, при цьому доступна довільна кількість вимірів. Для виконання виділяють такі кроки:

- визначення сітки
- обчислення скалярного добудку градієнтних векторів.
- інтерполяція між цими значеннями.

Лінійна функція, для кінцевих точок на 0 та 1, зі значеннями a_0 та a_1:

$$f(x) = a_0\,(1 - x) + \ a_1 x$$

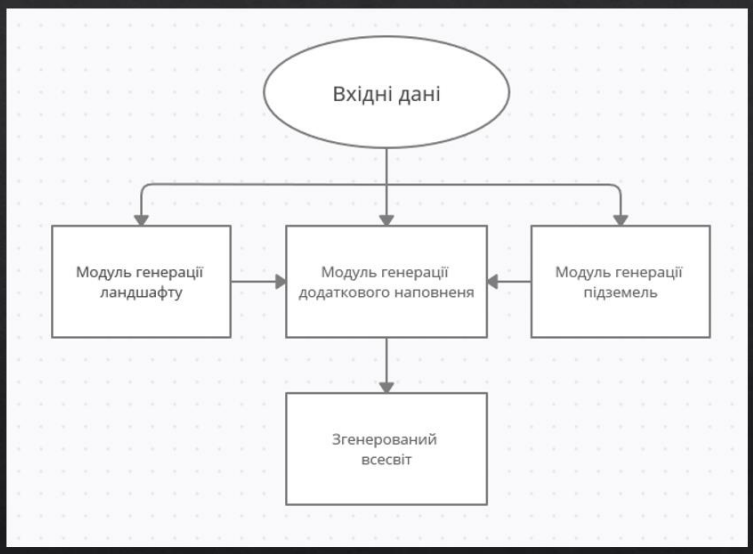
6

Недоліки та переваги алгоритмів

- Drunken Walk має стохастичну простоту. Але його можно використовувати і як метод Монте-Карло для оцінки значень, які занадто важко розрахувати іншими способами.
- Особливість і перевага алгоритму " Клітинний автомат" - це досить нескладна побудова рівнів вкрай незвичайних форм, в яких немає понять кімнат і коридорів, як в попередніх двох розглянутих. Структура рівня виходить згладженої і приємною на вигляд.
- За допомогою холмового алгоритму можна уявити досить великі простори. Але у нього є один істотний недолік - занадто багато описів для точок, а також, в деяких випадках, спостерігається надмірність даних.
- Генерація в основі шуму Перліна має більш органічний вигляд, оскільки він створює природно впорядковану («плавну») послідовність псевдовипадкових чисел.

7

Модель процедурної генерації



Блок-схема моделі

8

Модуль генерації ландшафту

Етапи генерування:

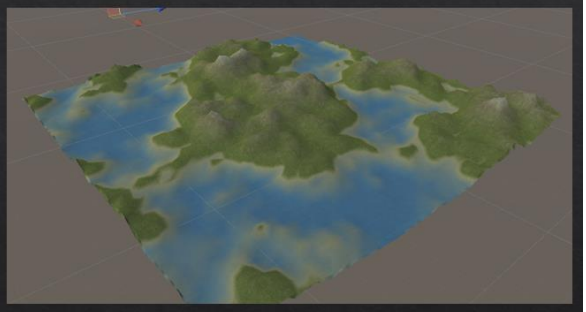
1)Визначення зміщення всіх шумових функцій.

2)Визначення допустимих координат.

3)Визначення середнього значення за індексом згідно до утворених координат.

4)Проекція утворених значень до графічного об’єкту.

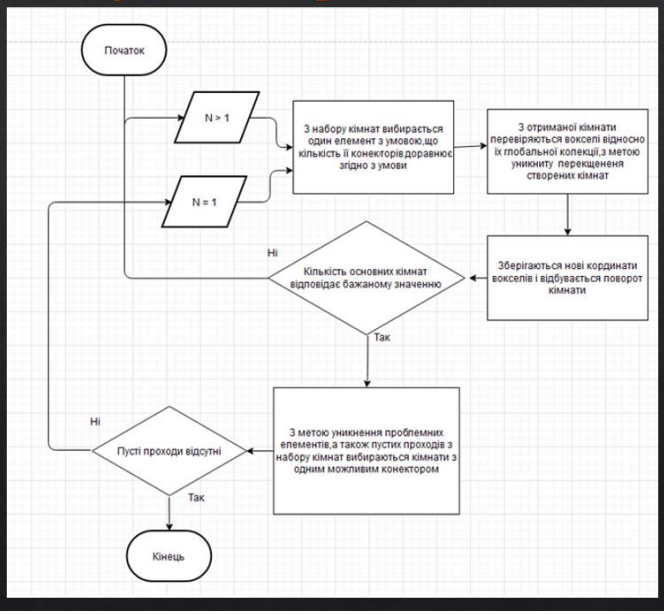
5)Визначення матеріалу відповідно до карти висот(надання кольору)



Приклад згенерованого світу

9

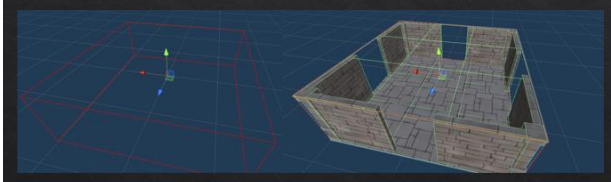
Модуль генерації підземель



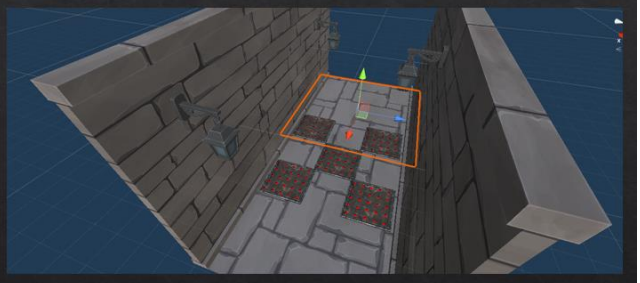
Блок-схема генерації кімнат

10

Графічні елементи модуля генерації підземель



Демонстрація вокселів



Приклад кімнати

11

Програмні елементи

Використані технології:

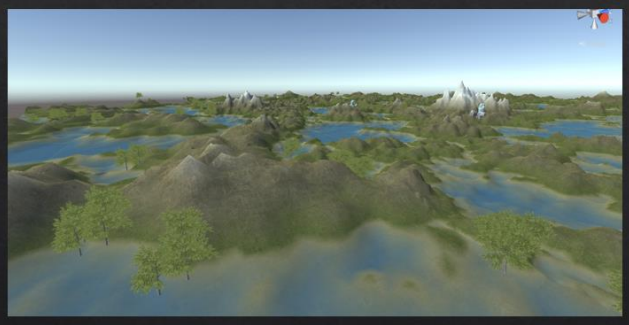
- .NET / C#
- Unity

Приклади розроблених скриптів при генерації:

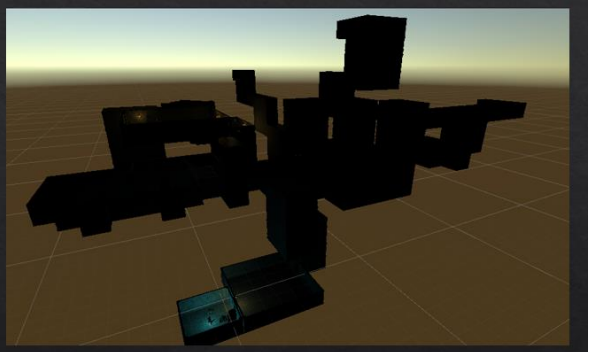
- Скрипт генерації ландшафту
- Скрипт генерації порталів
- Скрипт генерації підземелля
- Скрипт генерації дверей
- Скрипт випадкової зміни контенту у кімнатах

12

Результат реалізації та тестування



Результат генерації ландшафту



Результат генерації підземелля

13

Висновки

В рамках даної кваліфікаційної роботи були розглянуті проблеми при створенні віртуальних просторів з точок зору як ручного підходу, так і процедурного, коли структури даних, що задають цей простір, генеруються за допомогою програмних алгоритмів.

Робота спеціалізується на алгоритмах процедурної генерації рівнів, частина з яких використовується плиткове уявлення рівнів , на підставі аналізу трьох існуючих алгоритмів був створений новий, якому і присвячена основна частина роботи та один було реалізовано.

Тестування програми показало, що даний додаток має свої недоліки,які в майбутньому будут поправлені. Іноді можна спострігати проблеми з відкриттям дверей та неправильністю додаткової генерації об'єктів. В подальших версіях програми рекомендовано запобігти цієї недоробки.

Практичне застосування цієї програми у повсякденному житті можливе, але не кожному буде зручно грати у цей продукт, але у професійній сфері має велику перспективу, бо скорочує час розробки світу майже в 3 рази. Для більш комфортного застосування необхідно покращити программу за декількома пунктами, одним з яких буде можливість редактувати генерацію підземель та комбінувати згенерований ландшафт.

В ході роботи були :

- Проведений аналіз існуючих алгоритмів генерації;
- Розроблений алгоритм генерації підземель плиткового уявлення;
- Реалізовано генерацію ландшафту на основі шуму Перліна.

14

ДОДАТОК Б

Лістинг скриптів

Б.1 Скрипт кімнати

```
public class Room : MonoBehaviour
{
    public List<DoorContainer> doors;

    public DoorContainer GetRandomDoor(Random random)
    {
        doors.Shuffle(random);
        return GetFirst();
    }

    public bool IsOpen()=>
       doors.Any(x=>x.open);

    public DoorStorage GetFirst()=>
          doors.FirstOrDefault(x=>x.open);
}
```

Б.2 Скрипт переміщення

```
using UnityEngine;

[RequireComponent(typeof(Rigidbody))]
[RequireComponent(typeof(CapsuleCollider))]
public class Movement : MonoBehaviour
{
    [SerializeField] float _speed = 0.3f;
    [SerializeField] float _jumpForce = 1f;
    [SerializeField] Transform _camera;
    [SerializeField] float _turnSmoothVelocity;
    [SerializeField] LayerMask _groundLayer = 1;

    private Rigidbody _rb;
    private CapsuleCollider _collider;

    private bool _isGrounded
    {
        get
        {
            var bottomCenterPoint = new
Vector3(_collider.bounds.center.x, _collider.bounds.min.y,
_collider.bounds.center.z);
```

```
            return Physics.CheckCapsule(_collider.bounds.center,
bottomCenterPoint, _collider.bounds.size.x / 2 * 0.9f,
GroundLayer);

        }
    }

    private Vector3 _movementVector
    {
        get
        {
            var horizontal = Input.GetAxis("Horizontal");
            var vertical = Input.GetAxis("Vertical");

            return new Vector3(horizontal, 0.0f,
vertical).normalized;
        }
    }

    void Start()
    {
        _rb = GetComponent<Rigidbody>();
        _collider = GetComponent<CapsuleCollider>();
        _rb.constraints = RigidbodyConstraints.FreezeRotationX |
RigidbodyConstraints.FreezeRotationZ  |
RigidbodyConstraints.FreezeRotationY ;
    }

    void FixedUpdate()
    {
        JumpLogic();
        MoveLogic();
    }

    private void MoveLogic()
    {
        if (_movementVector != Vector3.zero)
        {

            _rb.transform.rotation =
Quaternion.AngleAxis(Mathf.SmoothDampAngle(_rb.transform.eulerAn
gles.y,
                (Mathf.Atan2(_movementVector.x,
_movementVector.z) * Mathf.Rad2Deg + _camera.eulerAngles.y),
                ref _turnSmoothVelocity, 0), Vector3.up);

            _rb.AddForce(_rb.transform.forward * _speed,
ForceMode.Impulse);
        }
    }

    private void JumpLogic()
    {
```

```
        if (_isGrounded && Input.GetAxis("Jump") > 0)
            _rb.AddForce(Vector3.up * _jumpForce,
ForceMode.Impulse);
    }
}
```

Б.3 Скрипт стану дверей

```
namespace DungeonRaider
{
    public class Door: MonoBehaviour
    {
        [SerializeField] bool _isOpen = true;
        [SerializeField] GameObject _voxelOwner;
        [SerializeField] DoorContainer _container;
        [SerializeField] DoorView _view;
    }
}
```

Б.4 Скрипт генерації підземелля

```
public class DungGenerator : MonoBehaviour
{
    public DungeonController data;
    private GameObject startRoom;
    public int targetRooms = 10;
    public List<Room> rooms = new List<Room>();
    public List<DoorView> doors = new List<DoorView>();

    private float voxelPixelSize = 10f;

    public List<Room> openSet = new List<Room>();
    public Dictionary<Vector3, GameObject> globalVoxels = new
Dictionary<Vector3, GameObject>();
    public List<GameObject> doorVoxelsTest = new
List<GameObject>();



    void Start()
    {
        random = new Random();
        roomsCount = 0;
        globalVoxels = new Dictionary<Vector3, GameObject>();
        dungeonSet = Random.Range(0, data.sets.Count - 1);

        StartGeneration();
    }

    private void StartGeneration()
```

```
    {
        rooms = new List<Room>();
        doors = new List<DoorView>();

        var spawn = Random.Range(0,
data.sets[dungeonSet].spawns.Count - 1);
        var room =
Instantiate(data.sets[dungeonSet].spawns[spawn].gameObject);
        startRoom = room;
        rooms.Add(room.GetComponent<Room>());
        room.transform.parent = gameObject.transform;
        openSet.Add(room.GetComponent<Room>());
        room.GetComponent<Volume>().RecalculateBounds();
        AddGlobalVoxels(room.GetComponent<Volume>().voxels);
        roomsCount++;

        while (openSet.Count > 0)
        {
            GenerateNextRoom();
        }


        for (int i = 0; i < rooms.Count; i++) {
            for (int j = 0; j < rooms[i].doors.Count; j++) {
                if (rooms[i].doors[j].door == null) {
                    var d =
(Instantiate(data.sets[dungeonSet].doors[0].gameObject)).GetComp
onent<DoorView>();
                    doors.Add(d);
                    rooms[i].doors[j].door = d;
                    rooms[i].doors[j].sharedDoor.door = d;

                    d.gameObject.transform.position =
rooms[i].doors[j].transform.position;
                    d.gameObject.transform.rotation =
rooms[i].doors[j].transform.rotation;
                    d.gameObject.transform.parent =
gameObject.transform;
                }
            }
        }
    }

    private void AddGlobalVoxels(List<GameObject> voxels) {
        for (int i = 0; i < voxels.Count; i++) {
var position =
RoundVec3ToInt(voxels[i].gameObject.transform.position);
            if (!globalVoxels.ContainsKey(position))
                globalVoxels.Add(position, voxels[i]);
        }
    }

    private Vector3 RoundVec3ToInt(Vector3 v) {
```

```
        return new Vector3(Mathf.RoundToInt(v.x),
Mathf.RoundToInt(v.y), Mathf.RoundToInt(v.z));
    }

     private void GenerateNextRoom() {
    var lastRoom = startRoom.GetComponent<Room>();
    if (openSet.Count > 0) lastRoom = openSet[0];

    var possibleRooms = new List<Room>();
    for (int i = 0; i <
data.sets[dungeonSet].roomTemplates.Count; i++) {

possibleRooms.Add(data.sets[dungeonSet].roomTemplates[i]);
    }
    possibleRooms.Shuffle(random);

    GameObject newRoom;
    DoorStorage door;
    var roomIsGood = false;


    do {
        for (int i = 0; i < doorVoxelsTest.Count; i++) {
            DestroyImmediate(doorVoxelsTest[i]);
        }
        doorVoxelsTest.Clear();

        if (roomsCount > targetRooms)
            possibleRooms =
GetAllRoomsWithOneDoor(possibleRooms);


        var doors = 0;
        var tryAgain = false;
        GameObject roomToTry;
        var r = Random.Range(0, possibleRooms.Count - 1);
        roomToTry = possibleRooms[r].gameObject;
        doors = roomToTry.GetComponent<Room>().doors.Count;

        if(doors == 1 && possibleRooms.Count > 1)
        {
            var chance = 1f - Mathf.Sqrt(((float)roomsCount /
(targetRooms));
            var randomValue = (float)random.NextDouble();
            if (randomValue < chance) {
                r = Random.Range(0, possibleRooms.Count - 1);
                roomToTry = possibleRooms[r].gameObject;

                doors =
roomToTry.GetComponent<Room>().doors.Count;
                if (doors == 1 && possibleRooms.Count > 1) {
                   var chance2 = 1f -
Mathf.Sqrt(((float)roomsCount / targetRooms));
```

```
                    var randomValue2 =
(float)random.NextDouble();
                    if (randomValue2 < chance2) {
                        r = Random.Range(0, possibleRooms.Count
- 1);
                        roomToTry = possibleRooms[r].gameObject;
                    }
                }
            }
        }
        possibleRooms.RemoveAt(r);

        newRoom = Instantiate(roomToTry);
        newRoom.transform.parent = gameObject.transform;
        door = ConnectRooms(lastRoom,
newRoom.GetComponent<Room>());


        var v = newRoom.GetComponent<Volume>();
        var ro = newRoom.GetComponent<Room>();
        var overlap = false;
        for (int i = 0; i < v.voxels.Count; i++) {
            if
(globalVoxels.ContainsKey(RoundVec3ToInt(v.voxels[i].gameObject.
transform.position))) {

                overlap = true;
                continue;
            }

            for (int j = 0; j < openSet.Count; j++) {
                for (int k = 0; k < openSet[j].doors.Count; k++)
{
                    if(!openSet[j].doors[k].open) continue;
                    if (openSet[j].doors[k] == door) continue;
                    var rot =
NormalizeAngle(Mathf.RoundToInt(openSet[j].doors[k].transform.ro
tation.eulerAngles.y));
                    var direction = new Vector3();
                    if (rot == 0)
                        direction = new Vector3(1f, 0f, 0f);
                    else if (rot == 90)
                        direction = new Vector3(0f, 0f, -1f);
                     else if (rot == 180)
                        direction = new Vector3(-1f, 0f, 0f);
                    else if (rot == 270)
                        direction = new Vector3(0f, 0f, 1f);

                    var g =
GameObject.CreatePrimitive(PrimitiveType.Sphere);
                    g.transform.position =
openSet[j].doors[k].voxelOwner.transform.position + (direction *
v.voxelScale);
```

```
                    doorVoxelsTest.Add(g);
                    if
(RoundVec3ToInt(v.voxels[i].gameObject.transform.position) ==
RoundVec3ToInt(openSet[j].doors[k].voxelOwner.transform.position
+ (direction * v.voxelScale))) {
                        overlap = true;
                    }
                }
            }
        }

        bool hasSpace = true;
        if (!overlap) {
            for (int i = 0; i < ro.doors.Count; i++) {
                if (!ro.doors[i].open) continue;
                if (ro.doors[i] ==
newRoom.GetComponent<Room>().GetFirstOpenDoor()) continue;
                                float rot =
NormalizeAngle(Mathf.RoundToInt(ro.doors[i].transform.rotation.e
ulerAngles.y));
                Vector3 direction = new Vector3();
                if (rot == 0)
                   direction = new Vector3(1f, 0f, 0f);
                 else if (rot == 180)
                    direction = new Vector3(-1f, 0f, 0f);
                else if (rot == 90)
                    direction = new Vector3(0f, 0f, -1f);
                else if(rot == 270)
                    direction = new Vector3(0f, 0f, 1f);

                var g =
GameObject.CreatePrimitive(PrimitiveType.Sphere);
                g.transform.position =
ro.doors[i].voxelOwner.transform.position + (direction *
v.voxelScale);
                doorVoxelsTest.Add(g);

                if
(globalVoxels.ContainsKey(RoundVec3ToInt(ro.doors[i].voxelOwner.
transform.position + (direction * v.voxelScale)))) {

                    hasSpace = false;
                    break;
                }
                    for (int j = 0; j < openSet.Count; j++) {
                        for (int k = 0; k <
openSet[j].doors.Count; k++) {
                            if (!openSet[j].doors[k].open)
continue;
                            var rot2 =
NormalizeAngle(Mathf.RoundToInt(openSet[j].doors[k].transform.ro
tation.eulerAngles.y));
                 Vector3 direction2 = new Vector3();
```

```
                        if (rot2 == 0)
                        direction2 = new Vector3(1f, 0f, 0f);
                        else if (rot2 == 180)
                        direction2 = new Vector3(-1f, 0f, 0f);
                        else if (rot2 == 90)
                        direction2 = new Vector3(0f, 0f, -1f);
                        else if (rot2 == 270)
                        direction2 = new Vector3(0f, 0f, 1f);
 if (RoundVec3ToInt(ro.doors[i].voxelOwner.transform.position +
(direction*v.voxelScale)) ==
RoundVec3ToInt(openSet[j].doors[k].voxelOwner.transform.position
+ (direction2*v.voxelScale))) {
                                    hasSpace = false;

                                    break;
                                }
                            }
                            if (!hasSpace) break;
                        }
                    }
                }
            }

            if (!overlap && hasSpace)
                roomIsGood = true;
             else
                DestroyImmediate(newRoom);

        } while (possibleRooms.Count > 0 && !roomIsGood);

        if (roomIsGood) {

            var otherDoor =
newRoom.GetComponent<Room>().GetFirstOpenDoor();
            door.sharedDoor = otherDoor;
            otherDoor.sharedDoor = door;

            door.open = false;
            newRoom.GetComponent<Room>().GetFirstOpenDoor().open =
false;

            rooms.Add(newRoom.GetComponent<Room>());

            AddGlobalVoxels(newRoom.GetComponent<Volume>().voxels);
            if (!lastRoom.hasOpenDoors()) openSet.Remove(lastRoom);
            if (newRoom.GetComponent<Room>().hasOpenDoors())
openSet.Add(newRoom.GetComponent<Room>());
            roomsCount++;


        }
```

Б.5 Скрипт генерації висот ландшафту

```
public static class HeightMapGenerator
{
     public static HeightMap GenerateHeightMap(int width, int
height, HeightMapSettings settings, Vector2 sampleCentre)
     {
          var values = Noise.GenerateNoiseMap (width, height,
settings.noiseSettings, sampleCentre);

          var heightCurve_threadsafe = new AnimationCurve
(settings.heightCurve.keys);

          var minValue = float.MaxValue;
          var maxValue = float.MinValue;

          for (var i = 0; i < width; i++)
          {
               for (var j = 0; j < height; j++)
               {
                    values [i, j] *=
heightCurve_threadsafe.Evaluate (values [i, j]) *
settings.heightMultiplier;

                    if (values [i, j] > maxValue)

                         maxValue = values [i, j];

                    if (values [i, j] < minValue)
                         minValue = values [i, j];

               }
          }

          return new HeightMap (values, minValue, maxValue);
     }

}

public struct HeightMap
{
     public readonly float[,] values;
     public readonly float minValue;
     public readonly float maxValue;

     public HeightMap (float[,] values, float minValue, float
maxValue)
     {
          this.values = values;
          this.minValue = minValue;
          this.maxValue = maxValue;
     }
```

```
}
```

Б.6 Скрипт текстур для матеріалу ландшафту

```
[CreateAssetMenu()]
public class TextureData : UpdatableData {

     const int textureSize = 512;
     const TextureFormat textureFormat = TextureFormat.RGB565;

     public Layer[] layers;

     float savedMinHeight;
     float savedMaxHeight;

     public void ApplyToMaterial(Material material) {

          material.SetInt ("layerCount", layers.Length);
          material.SetColorArray ("baseColours", layers.Select(x
=> x.tint).ToArray());
          material.SetFloatArray ("baseStartHeights",
layers.Select(x => x.startHeight).ToArray());
          material.SetFloatArray ("baseBlends", layers.Select(x
=> x.blendStrength).ToArray());
          material.SetFloatArray ("baseColourStrength",
layers.Select(x => x.tintStrength).ToArray());
          material.SetFloatArray ("baseTextureScales",
layers.Select(x => x.textureScale).ToArray());
          Texture2DArray texturesArray = GenerateTextureArray
(layers.Select (x => x.texture).ToArray ());
          material.SetTexture ("baseTextures", texturesArray);

          UpdateMeshHeights (material, savedMinHeight,
savedMaxHeight);
     }
     public void UpdateMeshHeights(Material material, float
minHeight, float maxHeight) {
          savedMinHeight = minHeight;
          savedMaxHeight = maxHeight;
          material.SetFloat ("minHeight", minHeight);
          material.SetFloat ("maxHeight", maxHeight);
     }
     Texture2DArray GenerateTextureArray(Texture2D[] textures) {
          Texture2DArray textureArray = new Texture2DArray
(textureSize, textureSize, textures.Length, textureFormat,
true);
          for (int i = 0; i < textures.Length; i++) {
               textureArray.SetPixels (textures [i].GetPixels
(), i);
          }
          textureArray.Apply ();
```

```
        return textureArray;
    }
    [System.Serializable]
    public class Layer {
        public Texture2D texture;
        public Color tint;
        [Range(0,1)]
        public float tintStrength;
        [Range(0,1)]
        public float startHeight;
        [Range(0,1)]
        public float blendStrength;
        public float textureScale;
```