



**Budapesti Műszaki és Gazdaságtudományi Egyetem**  
Villamosmérnöki és Informatikai Kar  
Irányítástechnika és Informatika Tanszék

Réti Ádám

# UNITY JÁTÉKFEJLESZTÉS

KONZULENS

Kárpáti Attila Ádám

BUDAPEST, 2024

# Tartalomjegyzék

Összefoglaló .....	4
1 Bevezetés .....	5
1.1 Motiváció .....	5
1.2 Játék logikája, játékmenet .....	5
1.2.1 UI (User interface) .....	5
1.2.2 Mozgó egységek és viselkedésük .....	5
1.2.3 Játékkör .....	6
1.2.4 Nyersanyagok .....	6
1.2.5 Biomok .....	6
1.3 Konklúzió .....	6
2 Használt technológiák, technikák .....	7
2.1 Unity Engine .....	7
2.1.1 GameObject .....	7
2.1.2 Rigidbody2D .....	7
2.1.3 BoxCollider2D .....	7
2.1.4 ScriptableObject .....	8
2.1.5 Coroutine .....	8
2.2 MeshRenderer .....	9
2.2.1 Geometry .....	9
2.2.2 Shader .....	10
2.2.3 Material .....	10
2.2.4 Mesh .....	11
2.3 ShaderGraph .....	13
2.3.1 Blueprint .....	13
2.3.2 SpriteOutliner .....	14
2.3.3 HexTexture .....	15
2.4 RayCast (sugárkövetés) .....	15
2.5 Noise (zaj) .....	16
2.5.1 NoisyEdge (zajos szélek) .....	17
2.6 A* útkereső algoritmus .....	18
2.6.1 Reroute, láthatatlan akadály .....	19

2.6.2 Pályageneráló algoritmus .....	19
3 Pálya felépítése, játék metrika .....	20
3.1 Geometria, orientáció.....	20
3.2 Elhelyezkedés .....	21
3.3 Koordináta rendszerek .....	21
3.3.1 Offset koordináta .....	21
3.3.2 Cube koordináta .....	22
3.3.3 Melyik koordináta rendszert érdemes használni.....	24
3.3.4 Koordináta rendszer konverzió .....	24
3.4 Szomszédos cellák .....	25
3.5 Távolság .....	25
3.6 Látható cellák (FOV) .....	25
4 Optimalizálás, Teljesítmény .....	27
5 Felhasznált források.....	28

# Összefoglaló

Az önálló laboratóriumi kutatásom célja a Unity Engine megismerése és alkalmazása egy stratégiai, körökre osztott társasjáték fejlesztése során, melynek neve HexWars. A játék többjátékos módot kínál, amely jelenleg csak ugyanazon az egy számítógépen belül érhető el. HexWars minden korosztály számára ajánlott, akik érdeklődnek a taktikai játékok iránt.

A játék egy HexGrid pályán zajlik, ahol a játékosok szabályos hatszögekből álló térképen mozognak és harcolnak. Ez az elrendezés számos tervezési lehetőséget biztosított számomra, mint fejlesztőnek. A játék célja, hogy a játékosok nyersanyagokat gyűjtsenek, védelmi rendszereket - mint például falak és tornyok - építsenek, valamint egységeket képezzenek ki. A nyersanyagokat különböző biomokból lehet megszerezni, amelyek különféle stratégiák alkalmazását teszik lehetővé. A játékosoknak az összegyűjtött erőforrások és egységek segítségével kell legyőzniük az ellenséges várakat és megvédeniük a sajátjukat.

A fejlesztési folyamat során számos új technológiát sajátítottam el, amelyek korábban ismeretlenek voltak számomra. Kiemelten foglalkoztam a Unity Engine beépített MeshRenderer komponensével, beleértve a Material, Mesh és Shader-ek használatát. Ezekhez kapcsolódóan RayCast (sugárkövetést) is használtam. Vizsgálataim során saját shadereket is létrehoztam a Unity ShaderGraph moduljának segítségével, melyeket vizuális scripting révén valósítottam meg. Továbbá, kisebb jártasságot szereztem a zajfüggvények (SimpleNoise, PerlinNoise) alkalmazásában is, amelyek, habár 2D-ben kevésbé látványosak, a jövőbeli projektjeimben 3D-ben is hasznosíthatók lesznek.

Az elkészült projekt alapjául szolgálhat a későbbi szakdolgozatomhoz, melyben tovább mélyítem az itt megszerzett tudást és tapasztalatokat. A HexWars fejlesztése során szerzett ismeretek és készségek jelentős mértékben hozzájárultak a szakmai fejlődésemhez és a Unity Engine lehetőségeinek megértéséhez.

# 1 Bevezetés

## 1.1 Motiváció

A játék elkészítésére való inspirációm a társas- és stratégiai játékok iránti szenvedélyem adta. Mivel első projektem során nem akartam rögtön 3D-ben dolgozni, fontosnak tartottam, hogy először alaposan megismerjem a Unity játékmotort, ezért döntöttem egy 2D-s játék fejlesztése mellett. A HexWars ötletét a Tribal Wars és a Catan telepesei játékok kombinációja ihlette, melyek mindketten HexGrid pályán játszódnak, ötvözve a stratégiai mélységet és a társasjátékok nyújtotta élményeket.

## 1.2 Játék logikája, játékmenet

### 1.2.1 UI (User interface)

A játék kezdetekor, miután a játékos kiválasztotta a pálya méretét a menüben, a kamera középpontja az induló játékos várára fókuszál. Az egyes cellákra kattintva a játékos kijelölheti azokat, aminek hatására a képernyő bal felső sarkában megjelennek az elérhető akciógombok. Ezek egyikére kattintva kiválaszthatja, milyen akciót szeretne végrehajtani, legyen az egységek létrehozása vagy mozgatása, illetve épületek építése. Mindig csak az adott cellán végrehajtható akciók gombjai jelennek meg. A jobb felső sarokban található egy gomb (End Turn), amellyel a játékos lezárhatja a körét. Az Escape billentyű megnyomásával egy beállítási menü hívható elő, ahol az animációk sebessége és a játék grafikai megjelenítése állítható.

### 1.2.2 Mozgó egységek és viselkedésük

A játékban létrejövő katonák fölött egy szám jelzi a sereg méretét. Ha ugyanazon játékos két serege találkozik, azok egyesülhetnek, összesítve az eredeti seregek haderejét. Ha viszont két különböző játékos seregei találkoznak, harcolni kezdenek, amíg egyikük teljesen megsemmisül. Egyenlő haderő esetén mindkét sereg megsemmisül, míg a nagyobb haderővel rendelkező sereg győztesen kerül ki, bár veszteségeket szenvedve. A katonák különböző biomok alapján választják meg mozgási útvonalukat, és minden cellára lépés különböző költséggel jár. Az ellenséges falak akadályozzák a mozgást, míg a saját épületeken való áthaladás költségmentes.

### **1.2.3 Játékkör**

Egy játékosnak csak limitált végrehajtható akció száma van, mely körönként frissül. Minden egyes végrehajtott lépés egyel csökkenti a maradék akciószámot. Egy kör lezárása után a kamera a következő játékos várására fókuszál. Minden játékos további nyersanyagokat kap a körök befejezése után, melyek mennyiségét az egyes biomokon álló egységek határozzák meg (a vár is termel egy kis mennyiséget).

### **1.2.4 Nyersanyagok**

Háromféle nyersanyag áll rendelkezésre: Fa, Élelem és Ásvány. Az Élelem az egységek létrehozásához szükséges, míg a Fa és Ásvány az épületek építéséhez. Minden biom termel mindhárom nyersanyagot, de eltérő mennyiségekben. A játék addig tart, amíg csak egy játékos vára marad állva. A közeljövőben, majd ezeket a nyersanyagokat másra is fel lehet használni, mint például új egységek/épületek létrehozására, vagy meglévők fejlesztésére.

### **1.2.5 Biomok**

A játékban négy fajta biom található meg: Hegy, Erdő, Part, Óceán. (Magasság szerint csökkenő sorrendben). Mindegyik biom máshogy befolyásolja a játékos által látott cellákat. A hegyek mögött lévő cellák például nem láthatóak a játékos számára, mivel magasak, ezért kitakarják őket az egységek elől.

## **1.3 Konklúzió**

A HexWars játék dinamikája és stratégiai mélysége a különböző biomokból származó nyersanyagok gyűjtésére, valamint a védelmi és támadási stratégiák kialakítására helyezi a hangsúlyt, biztosítva ezzel a kihívást és szórakozást minden játékos számára. A hexagonális rács alapú pálya és a változatos biomok lehetővé teszik, hogy a játékosok különböző taktikai megközelítéseket alkalmazzanak. A körökre osztott játékmenet gondolkodásra és tervezésre ösztönzi a résztvevőket, miközben a többjátékos lehetőség fokozza a versengést és a közösségi élményt. Az intuitív felhasználói felület és a testreszabható beállítások tovább növelik a játék élvezhetőségét.

## **2 Használt technológiák, technikák**

### **2.1 Unity Engine**

A Unity Engine egy népszerű, több platformos játékkészítő eszköz, amelyet játékfejlesztők használnak interaktív 2D és 3D játékok és egyéb tartalmak létrehozására. Lehetővé teszi a fejlesztők számára, hogy könnyedén kezeljék a játékok különböző elemeit, például a grafikát, a fizikát és a hangot, miközben támogatja a különböző platformokra való exportálást, mint például a PC, konzolok, mobil eszközök és VR.

#### **2.1.1 GameObject**

A GameObject a Unity alapvető építőeleme, amely bármilyen objektumot képviselhet a játékban, legyen az karakter, kamera vagy akár egy láthatatlan pont. Minden GameObject rendelkezik egy pozícióval, rotációval és mérettel a játék világában, melyet a GameObject Transform-jának hívunk. A GameObject-ek különböző komponensekkel bővíthetők, hogy specifikus funkciókat és viselkedéseket biztosítsanak.

#### **2.1.2 Rigidbody2D**

A Rigidbody2D egy fizikai komponens a Unity-ben, amely lehetővé teszi, hogy a 2D objektumok fizikai tulajdonságokkal rendelkezzenek, például tömeggel, gravitációval és erőkkel való interakcióval. Ez a komponens szükséges ahhoz, hogy a GameObject-ek fizikailag valósághűen mozogjanak és ütközzenek egymással a 2D térben. Mivel nálam a játékban, nincs szükség semmilyen fizikai szimulációra, ezért én csak az ütközés érzékelésére használom fel ezt az elemet.

#### **2.1.3 BoxCollider2D**

A BoxCollider2D egy olyan komponens, amely egy téglalap alakú ütközési mezőt definiál a GameObject számára. Ez lehetővé teszi, hogy az objektumok ütközzenek és reagáljanak egymásra a játék világában. A BoxCollider2D-t gyakran használják a 2D platformjátékokban, hogy meghatározzák a karakterek és a környezet közötti interakciókat. Ennek az elem segítségével detektálom a tényleges ütközést, aminek segítségével a játékban lévő katonák erejét össze tudom forrasztani egygé, vagy a két különböző játékoshoz tartozó hadsereg harcát szimulálom.

### 2.1.4 ScriptableObject

A ScriptableObject a Unity egy speciális típusú objektuma, amely lehetővé teszi, hogy az adatokat különálló eszközként (asset) lehessen tárolni. Ezek az objektumok nem jelennek meg közvetlenül a jelenetben (scene), mint a GameObject-ek, hanem inkább adatstruktúrák kezelésére szolgálnak, amelyek könnyen újrafelhasználhatók és szerkeszthetők. Használatuk különösen hasznos konfigurációs adatok, beállítások vagy egyéb olyan információk tárolására, amelyek több helyen is felhasználhatók a játékban. Játékomban számtalan adat lista tárolására használok ezt az objektumot, legyen az színlista, vagy környezet elemeinek vizuális listája.

```
[CreateAssetMenu(menuName = "Data/Building")]  
Unity Script | 2 references  
public class BuildingData : ScriptableObject  
{  
    public List<Building> list = new List<Building>();  
}
```

### 2.1.5 Coroutine

A Coroutine egy Unity-specifikus programozási mechanizmus, amely lehetővé teszi, hogy megszakítsuk és folytassuk a függvények végrehajtását több kereten keresztül. Ez hasznos, ha időbeli késleltetéseket, animációkat vagy hosszú műveleteket szeretnénk végrehajtani anélkül, hogy blokkolnánk a fő szálát és ezáltal a játék futását. A Coroutine-ok használata lehetővé teszi, hogy aszinkron műveleteket hajtsunk végre egyszerűen és hatékonyan. Ezt össze lehet keverni a szálkezeléssel azzal, hogy a coroutine új szálát indít, de itt nem ez a helyzet, hiszen a coroutine nem indít új szálát. A coroutinet a seregek mozgásának késleltetéséhez használok, mely addig tart, míg el nem érték a cél cellát, vagy el nem akadtak (olyan cellánál, ahova nem lehet eljutni). Coroutine-ok kezelik még a pop-up szövegeket, melyek a játékos számára jelzik, hogy mennyi nyersanyagot szerzett/költött el egy körben. Ezeknek a pop-up szövegeknek tulajdonsága, hogy míg él a coroutine, addig folyamatosan csökkenti a szöveg láthatóságát (text alpha értékét csökkenti). Ha újabb coroutine hívás történik, miközben él az eredeti, akkor azt megszakítom, és újat indítok, az előző pop-up text értékeinek frissítésével.

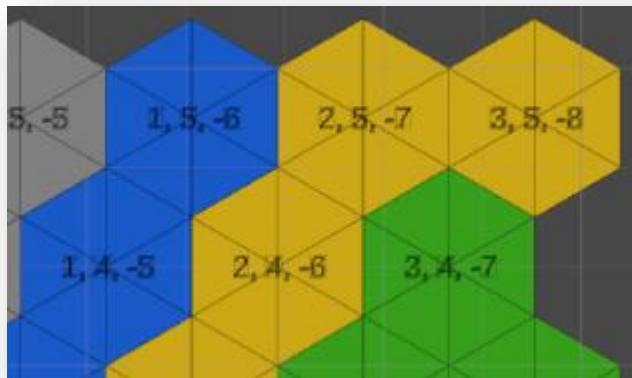


## 2.2 MeshRenderer

A Mesh Renderer a Unity-ben egy komponens, amely felelős egy objektum hálójának (mesh) megjelenítéséért a játék világában. Ez a komponens biztosítja, hogy a háló a megfelelő anyagokkal és textúrákkal renderelve látható legyen a képernyőn. A Mesh Renderer-t általában egy GameObject-re helyezik, amely tartalmaz egy Mesh Filter-t is, ami meghatározza, melyik hálót kell megjeleníteni. Nálam a katonák, környezeti elemek és épületek kivételével minden Mesh segítségével van kirajzolva. Ilyen például maga az egész pálya kirajzolása, ahol egy cellát a HexGeometry segítségével rajzolok ki, ami 6 db Tri-ből (háromszög) áll.

### 2.2.1 Geometry

A hexagonal geometry a hatszögletű rácsokkal és struktúrákkal foglalkozik. Ezeket gyakran használják játékokban és térképekben, különösen azokban, amelyek hexagonális (hatszögletű) csempéket (tile-okat) alkalmaznak, mint például a stratégiai játékok és bizonyos típusú térképes játékok. A hexagonális csempék előnye, hogy a szomszédos cellák egyenlő távolságra vannak egymástól, ami természetesebb mozgást és elrendezést tesz lehetővé bizonyos játékstílusokban. Fontos megjegyezni, hogy a Mesh létrehozásához ismerni kell a pontokat (vertice), amelyek majd felépítik azt. Én ezt 7 db ponttal hozom létre. Nálam ez jelenti a HexGeometry-t.



*Mesh HexGeometry-vel kirajzolva és Cube koordinátákkal feltüntetve*

## **2.2.2 Shader**

A shader egy speciális program, amely meghatározza, hogyan jelenjen meg a grafikus objektum a képernyőn a számítógépes grafika terén. Ezek a programok általában a GPU-n (Graphics Processing Unit) futnak, és különféle vizuális effektusokat, textúrákat, árnyékolást, fényeket és egyéb grafikákat számolnak ki valós időben.

### **2.2.2.1 Vertex Shader**

Feldolgozza az egyes csúcsokat (vertex) a modellben, meghatározva azok pozícióját, színét és textúra koordinátáit a renderelési folyamat során.

### **2.2.2.2 Fragment (Pixel) Shader**

Meghatározza az egyes pixelek színét, textúráját és fényviszonyait, lehetővé téve részletesebb és valósághűbb képek megjelenítését.

### **2.2.2.3 Geometry Shader**

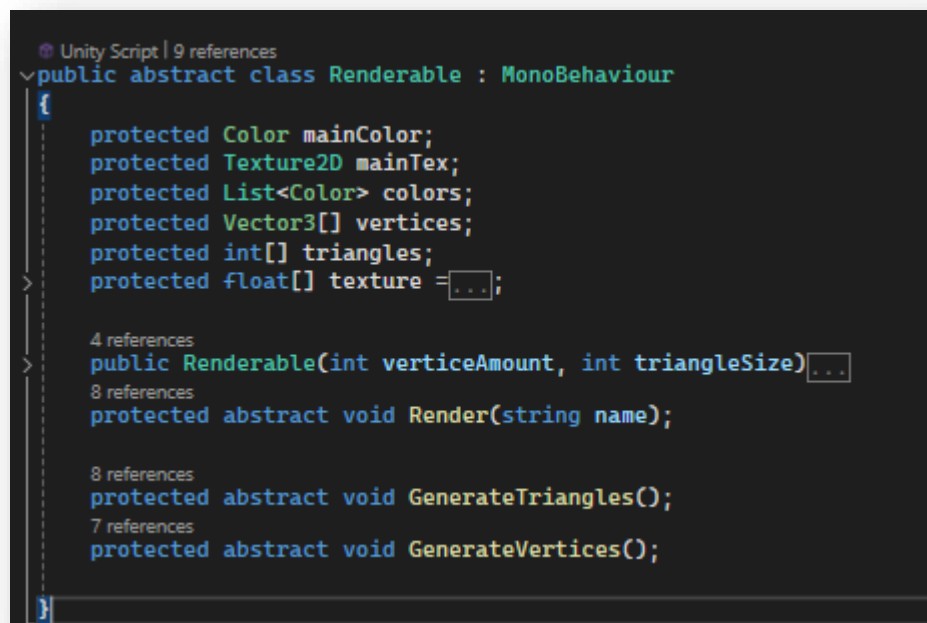
Opcionálisan bővíti a vertex shader adatait további geometriával, például új csúcsokkal vagy primitívekkel. (HexGeometry).

## **2.2.3 Material**

A material (anyag) a Unity-ben egy olyan erőforrás (asset), amely meghatározza, hogyan jelenjen meg egy objektum felülete a játék világában. A materialok különféle tulajdonságokat és beállításokat tartalmaznak, például színeket, textúrákat és shader-eket, amelyek segítségével az objektumok megjelenését testre szabhatjuk. Számos saját materiált és shadert készítettem, melyet majd a 2.3 ShaderGraph pontban be is mutatok.

## 2.2.4 Mesh

A mesh a Unity-ben és általában a számítógépes grafikában egy olyan adatstruktúra, amely egy objektum geometriai információit tartalmazza. Ez az információ tartalmazza a pontok (vertexek) koordinátáit, amelyek meghatározzák az objektum alakját, valamint a poligonokat (általában háromszögeket), amelyek ezen pontok összekapcsolásával határozzák meg az objektum felületét. Fontos megjegyezni, hogy amikor definiáljuk a Mesh háromszögeit, akkor 2D-ben például nem mindegy, hogy milyen sorrendben adjuk meg a vertexeket. Többször is olyan problémába ütköztem, hogy a vertexek sorrendje miatt 2D-ben a háromszög másik oldala volt renderelve emiatt, ami 3D-ben jól látszódott, hiszen ott van egy 3. fő tengely is, de 2D-ben csak egy átlátszó „semmiként” jelent meg. Tudomásom szerint best practise, ha a vertexek sorrendjét óramutató, vagy valami hasonló sorrend szerint adjuk meg. Minden GameObject, amit én hozok létre Mesh segítségével egy absztrakt „Renderable” osztályból származik le, aminek a következő három függvényét kell felüldefiniálni:

A screenshot of the Unity Script editor showing the definition of the 'Renderable' class. The class is public, abstract, and inherits from MonoBehaviour. It contains several protected fields: Color mainColor, Texture2D mainTex, List<Color> colors, Vector3[] vertices, int[] triangles, and float[] texture. There are three methods: a public constructor 'Renderable(int verticeAmount, int triangleSize)', and two protected abstract methods 'Render(string name)' and 'GenerateTriangles()'. The 'GenerateVertices()' method is also protected and abstract. Reference counts are shown for some methods: 4 for the constructor, 8 for Render, 8 for GenerateTriangles, and 7 for GenerateVertices.

```
Unity Script | 9 references
public abstract class Renderable : MonoBehaviour
{
    protected Color mainColor;
    protected Texture2D mainTex;
    protected List<Color> colors;
    protected Vector3[] vertices;
    protected int[] triangles;
    protected float[] texture = [...];

    4 references
    public Renderable(int verticeAmount, int triangleSize) {...}
    8 references
    protected abstract void Render(string name);

    8 references
    protected abstract void GenerateTriangles();
    7 references
    protected abstract void GenerateVertices();
}
```

Egy példa erre a HexCell létrehozása, amiből épül fel a pálya:

```
2 references
protected override void Render(string name)
{
    Mesh mesh = new Mesh();
    mesh.name = name;
    // Define UVs
    Vector2[] uv = new Vector2[texture.Length / 2];
    for (int i = 0; i < uv.Length; i++)
    {
        uv[i] = new Vector2(texture[i * 2], texture[i * 2 + 1]);
    }

    // Assign vertices, UVs, colors, and triangles to the mesh
    mesh.vertices = vertices;
    mesh.uv = uv;
    mesh.colors = colors.ToArray();
    mesh.triangles = triangles;

    MeshFilter meshFilter = GetComponent<MeshFilter>();
    meshFilter.mesh = mesh;

    // Assign the mesh to the MeshCollider
    MeshCollider meshCollider = GetComponent<MeshCollider>();
    meshCollider.sharedMesh = mesh;
}

2 references
protected override void GenerateVertices()
{
    for (int i = 0; i < HexMetrics.corners.Length; i++)
    {
        vertices[i] = HexMetrics.corners[i];
    }
    vertices[HexMetrics.corners.Length] = HexMetrics.GetHexagonCenter();
}

2 references
protected override void GenerateTriangles()
{
    for (int i = 0; i < HexMetrics.corners.Length; i++)
    {
        int index = i * 3;
        triangles[index] = i;
        triangles[index + 1] = (i + 1) % HexMetrics.corners.Length;
        triangles[index + 2] = HexMetrics.corners.Length;
    }
}
```

HexMetrics statikus osztály corners tömbjének tartalma, ami tartalmazza a vertice-ket.

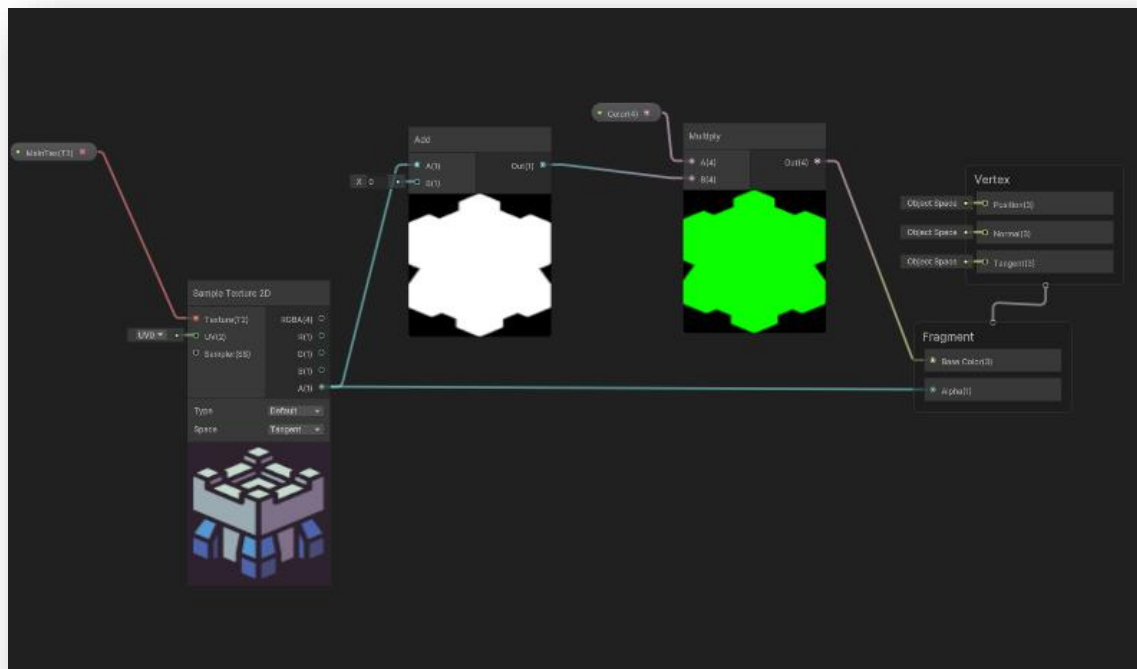
Lásd később 3. szekció, pálya felépítése, játék metrika.

```
public static Vector3[] corners = {
    new Vector3(0f, outerRadius, 0f),
    new Vector3(innerRadius, 0.5f * outerRadius, 0f),
    new Vector3(innerRadius, -0.5f * outerRadius, 0f),
    new Vector3(0f, -outerRadius, 0f),
    new Vector3(-innerRadius, -0.5f * outerRadius, 0f),
    new Vector3(-innerRadius, 0.5f * outerRadius, 0f),
};
```

## 2.3 ShaderGraph

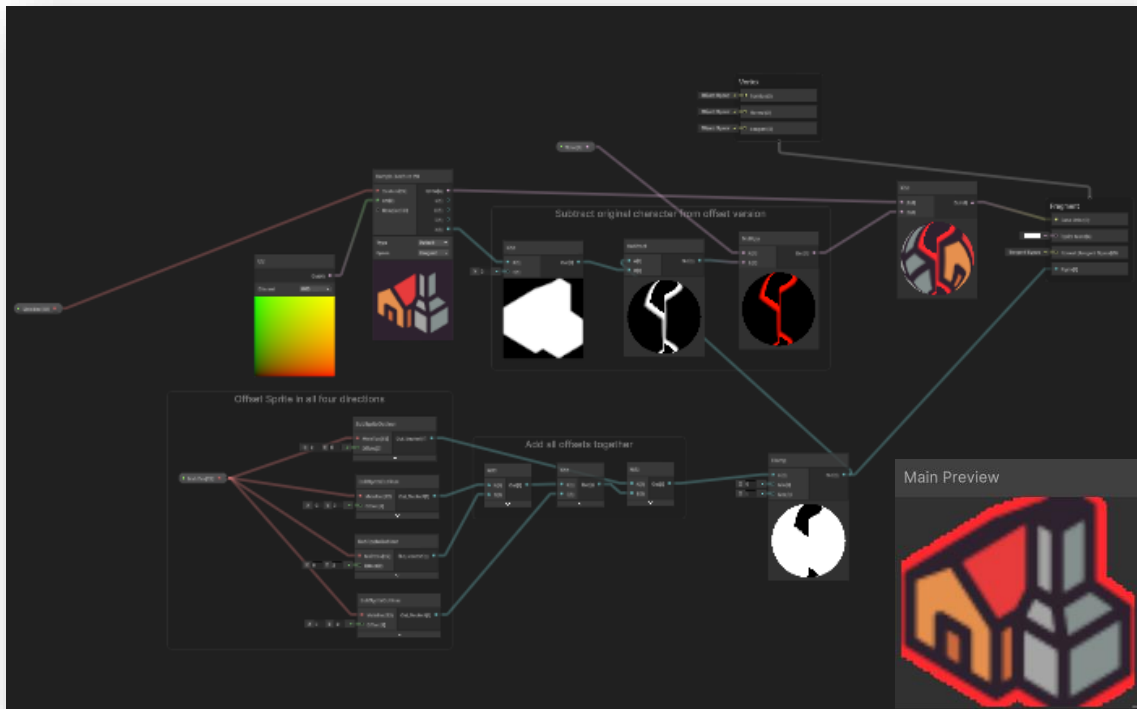
A Shader Graph a Unity-ben egy vizuális eszköz, amely lehetővé teszi shader-ek létrehozását és szerkesztését grafikus felhasználói felületen keresztül, kódolás nélkül (visual scripting). A Shader Graph segítségével a fejlesztők és művészek node-alapú rendszerben dolgozhatnak, ahol különböző funkciókat és effektusokat képviselő blokkokat (node-okat) húznak be és kapcsolnak össze. Ez a módszer intuitívabbá és gyorsabbá teszi a shader fejlesztést, különösen azok számára, akik nem rendelkeznek mély programozási ismeretekkel. A következő alpontokban bemutatom, hogy milyen shaderek/materiálokat hoztam létre:

### 2.3.1 Blueprint



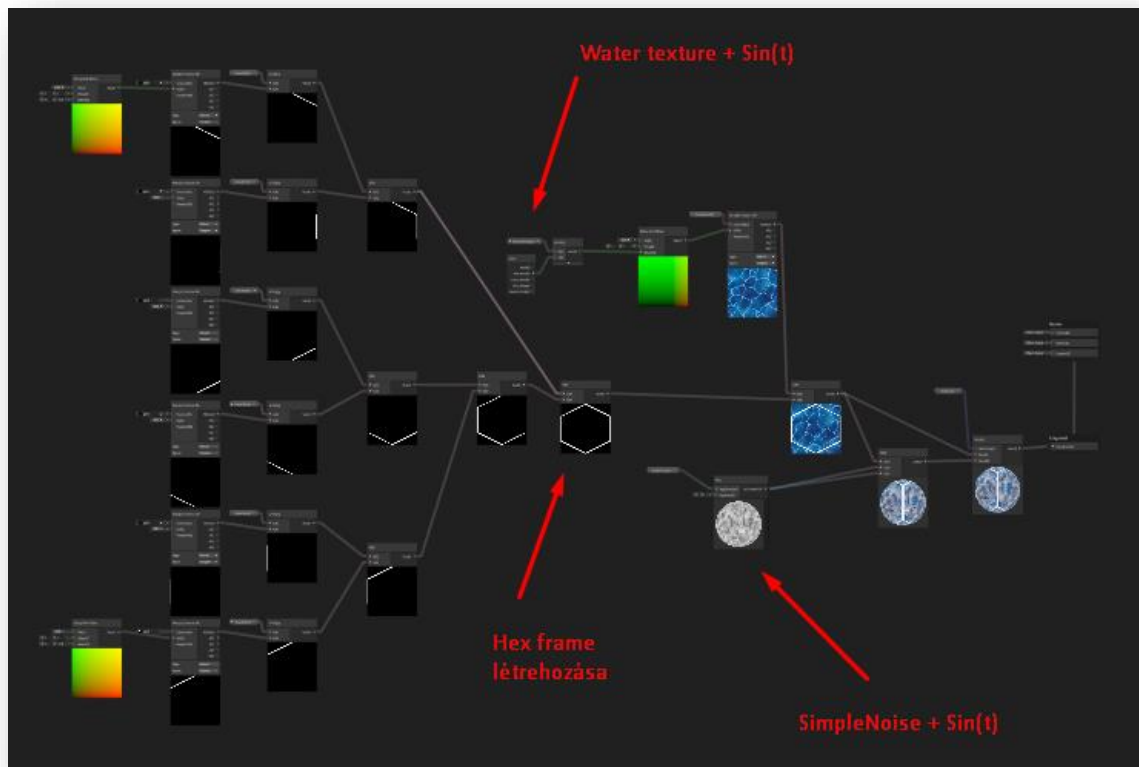
A blueprint shader, veszi az összes olyan pixelt, ahol van színeknek értéke, és kiszínezi az összeset egy adott színűre, amelyet a „\_Color” uniform változón keresztül tudok elérni. Ennek a shader használatával előállított materiált használom, az épületek építésénél. Ezzel jelzem a felhasználónak, hogy hova tud/nem szabad építeni tornyokat/falakat.

### 2.3.2 SpriteOutliner



A Spriteoutliner shader, a Blueprinthez hasonlóan kiszínezi és kiegészítve, el is tolja a színes pixeleket mind a 4 irányba pár pixel távolságra, majd veszi az eredeti Sprite textúráját, és hozzáadja az egy színűre színezett Sprite-hoz. Ezzel egy olyan hatást hozok létre, ami körvonalazza a Spritokat. Ezzel könnyen elkerülhető, hogy az összes Sprite-ot egyenként módosítsuk képszerkesztő program segítségével. Ezt minden játékos által létrehozott épület/egység Sprite-jánál használom, hogy kiemeljem őket egy külön színnel, hogy mi melyik játékoshoz tartozik.

### 2.3.3 HexTexture



A HexTexture a játékomban leggyakrabban használt Shader. Segítségével létre tudok hozni egy olyan keretet a celláknak, melynek minden oldalának színét külön tudom állítani. Ez a shader kezeli külön a fog of war és víz folyásának effektjét. A víz textúrájának textúra koordinátát változtatgatom a  $\sin(t)$  függvény alapján, hogy egy vízfolyás effektust jelenítsek meg a cellákon belül. A fog of war effektet teljesen ugyanígy működik, azzal a különbséggel, hogy itt maga a Textúra is random generált (SimpleNoise segítségével). Ezeket a „grafikai animációkat”, mind uniform változókon keresztül el tudom érni és ki/be tudom kapcsolni. Ennek segítségével hoztam létre a beállítások menüpontot, melyet az **1.2.1 UI** szekcióban említettem korábban.

### 2.4 RayCast (sugárkövetés)

A raycast egy olyan módszer a számítógépes grafikában és játékfejlesztésben, amelynek segítségével egy "sugarat" (ray) indítunk egy adott pontból és irányba a térben, és megvizsgáljuk, hogy ez a sugár metszi-e valamilyen objektumot. Ez lehetővé teszi például az ütközésdetekció megvalósítását, az egér vagy egyéb irányító eszközök kattintásainak érzékelését, valamint a látótávolság ellenőrzését. A raycast egy hatékony

eszköz a játékfejlesztők számára, amely segítségével interakciókat és fizikai viselkedéseket valósíthatnak meg a játékokban. Ezt arra használtam, hogy detektáljam, hogy a játékos egy cellára kattintott és kijelölte azt. Fejlesztésem során olyan problémába ütköztem, hogy amikor RayCast-oltam, és a UI elemekre kattintottam, akkor is kilőtte a sugarakat és közben ki is jelölte a UI elemek mögötti cellákat. Én ezt nem akartam, de szerencsére a Unity EventSystem objektuma tudja ezt jelezni, ha UI elemen van az egerünk:

```
private void Update()
{
    if (!EventSystem.current.IsPointerOverGameObject())
    {
        if (Input.GetMouseButton(0))
        {
            Ray inputRay = Camera.main.ScreenPointToRay(Input.mousePosition);
            RaycastHit hit;
            if (Physics.Raycast(inputRay, out hit))
            {
                SelectCell(hit.point);
            }
        }
    }
}
```

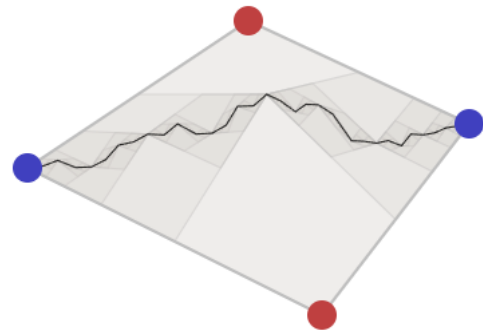
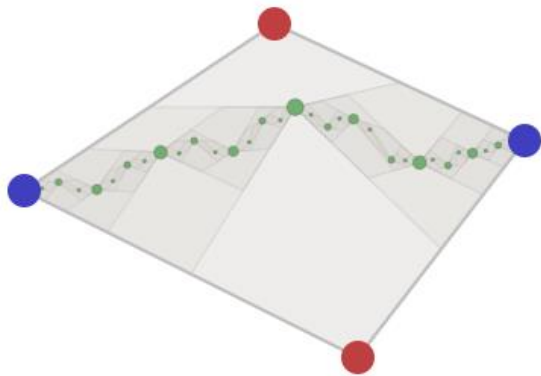
## 2.5 Noise (zaj)

A zajfüggvény egy matematikai függvény, amely véletlenszerű értékeket ad vissza egy adott tartományban vagy térben. Ezeket a függvényeket gyakran használják a számítógépes grafikában és játékfejlesztésben, például textúrák generálásához, térbeli zajok szimulálásához vagy realiztikus véletlenszerűség létrehozásához. A zajfüggvények különféle típusokban létezhetnek, például Perlin zaj, simplex zaj vagy White zaj, és különböző tulajdonságokkal rendelkeznek, például simaság, részletesség vagy térbeli változatosság. Ezeket a zajfüggvényeket gyakran alkalmazzák például tereptextúrák, animációk, procedurális generáció és egyéb grafikai feladatok során a valósághűség és a változatosság növelése érdekében. Játékomban én csak SimpleNoise-t használtam, melyet a **2.3.3 HexTexture** szekcióban már említettem. A közeljövőben terveztem, maga a pályagenerálási algoritmusnál is használni zajfüggvényeket. Habár ez nem lesz olyan látványos, mint 3D-ben, de a különböző zajok által generált magasságokat, különböző biomok (**1.2.5 Biomok**) formában tudom reprezentálni.



### 2.5.1 NoisyEdge (zajos szélek)

Ezzel a rekurzív algoritmussal generálok a celláknak széleket, mely extra vizuális élményt nyújt a játékos számára. A lényeg az, hogy mindig a létező pontok felénél felveszek további pontokat, melyeket egy véletlen generált amplitúdóval eltolok a szélek irányába. Minél többször iterálok végig a pontokon, annál részletesebb szélet tudok kapni.



levels =  5

subdivide into   $2^5 = 32$  segments  
amplitude =  0.5



*Ezzel a technikával „elmostam” a cellák szabályos felépítését, amivel tovább fokoztam a pálya felépítésének bonyolultságát.*

## 2.6 A\* útkereső algoritmus

Útkereső algoritmusoknál fontos, hogy hogyan reprezentáljuk az adatot, jelen esetben a pályát. Már létezik a HexGrid, és a cellák egymáshoz való pozíciója. Ezt kihasználom, hogy a cellák közepének helyzetét „Node” -oknak, a pontok közötti távolságot, pedig „Edge” -knek tekintem. Ezzel egy gráfot reprezentálok, ahol minden cella ismeri, hogy ki a szomszédja. Sok útkereső algoritmus fut gráfokon, pl.BFS, Dijkstra algoritmus, de ezeket most nem használtam. Helyettük a Dijkstra algoritmusának továbbfejlesztett változatát az A\* keresést használtam.

Az A\* keresés egy hatékony keresési algoritmus, amely a gráfok vagy térképek bejárására és a legrövidebb út keresésére használatos. Az A\* keresés a heurisztikus keresési algoritmusok egyik típusa, amely az adott probléma specifikus jellemzőit kihasználva keresi meg a legrövidebb útvonalat egy kiindulópontból egy célpontig egy gráfban vagy térképen.

Az A\* algoritmus képes hatékonyan navigálni a térképeken vagy gráfokon, mivel a heurisztikus függvény segítségével becsüli meg a legrövidebb út hosszát a cél felé. Ezáltal az A\* keresés kevésbé eredménytelen "vak" kereséssel kezdődik szemben az egyszerű Breadth First Search (szélességi keresés) vagy Depth First Search (mélységi keresés) algoritmusokkal.

Az A\* algoritmus lényege az, hogy minden pontnak egyfajta "költséget" vagy "prioritást" rendel egy adott állapottól való távolság alapján, amely magában foglalja az eddig megtett távolságot és egy becsült maradék távolságot a célhoz. Ezután az algoritmus folyamatosan a legkisebb összköltségű pontokat választja ki és vizsgálja meg, így haladva előre a kiindulópontból a célig. Az A\* keresés hatékonyan találja meg a legrövidebb utat, miközben minimalizálja a vizsgált pontok számát, így ideális választás lehet olyan problémák megoldására, ahol a kiindulópont és a cél közötti út megtalálása a feladat. Ezzel a megoldással módosítom a katonák útvonalát a játékban.

### 2.6.1 Reroute, láthatatlan akadály

Algoritmusomban figyelembe van véve az egyes cellákról másik cellára lépés költsége. Ez a költségfüggvény még figyelembe veszi a látható akadályok is, mint például az ellenséges falakat, de játékomban - mivel van FOW (Fog of War, háború köde) és FOV (Field of View, Látótávolság) -, ezért nem feltétlenül látható minden akadály (falak). Ilyen esetben az útkereső algoritmus úgy számolja a költségeket, mintha nem is lenne ott semmilyen úttorlasz, pedig valójában van. Amint láthatóvá válik a korábban láthatatlan akadály, az A\* keresés újra számolja az optimális útvonalat, figyelembe véve az előző próbálkozás óta felfedezett akadályokat. Ezt mindaddig folytatja, míg rá nem jön arra, hogy „unreachable node” a célpont, vagy míg el nem éri a célpontot. Mivel itt Coroutinokat használok, újratervezéskor a korábbi Coroutine megszűnik és egy új indul a frissített adatokkal.

### 2.6.2 Pályageneráló algoritmus

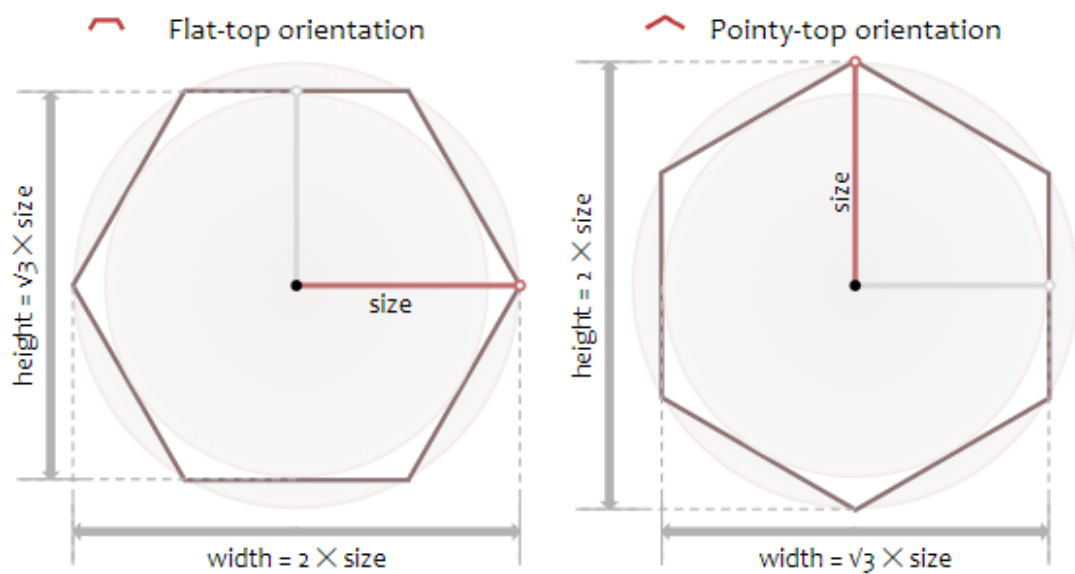
Korábban említettem, (2.5 Noise) hogy hogyan fejleszteném tovább a pályageneráló algoritmusomat., pl. Perlin zaj segítségével. Most jelenleg egy véletlen generáló pálya algoritmus hozza létre a pályát (Minden futtatáskor más a pálya kinézete).

Maga az algoritmus egyszerű automatákra épül: Véletlen kiválaszt X darab cellát a HexGrid-nek, amiket „szigetközéppontnak” nevezek. Ezekből a középpontokból elkezdek A\* keresés segítségével közeledni egy véletlen választott másik sziget fele. Ezt a műveletet a pálya méretének függvényében, fix mennyiségű iterációval hajtom végre. Így van lehetőségem egy fő kontinens létrehozására, és néhány szerencsés esetben kisebb szigetek létrehozására is.

## 3 Pálya felépítése, játék metrika

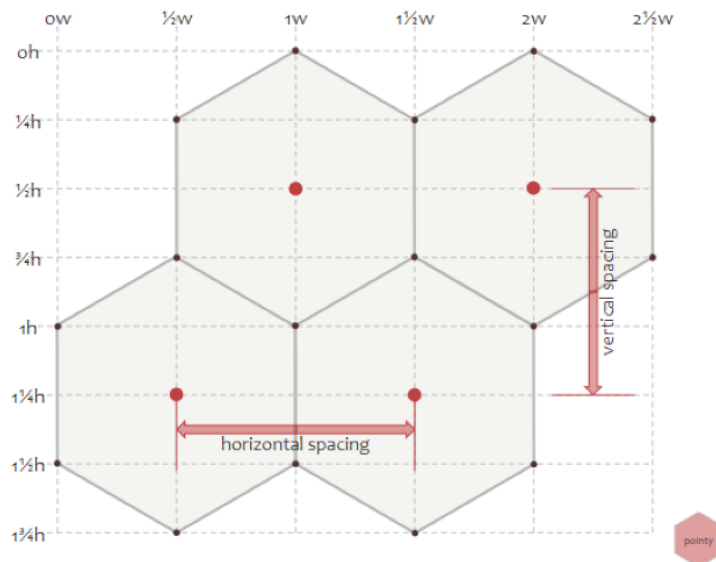
### 3.1 Geometria, orientáció

Hexagonoknak nevezzük azokat a poligonokat, melyeknek hat oldaluk van. A szabályos hexagonok oldalainak hossza azonos. A hexagon méretét a belső kör vagy az összeérő élekhez érő külső kör átmérőjével lehet leírni. A hexagon szélességét és magasságát a két kör átmérőjének méretei alapján határozzák meg. Ez a képlet eltérő a hexagonok orientációja alapján. Játékomban a Pointy-top orientációt használom.



## 3.2 Elhelyezkedés

Ha több cellát szeretnénk egymáshoz illeszteni, újra figyelembe kell venni az orientációt. Az általam használt orientáció esetén a szomszédos hatszög középpontjainak vízszintes távolsága  $\text{horiz} = \text{width} = \sqrt{3} * \text{size}$ . A függőleges távolság pedig  $\text{vert} = 3/4 * \text{height} = 3/2 * \text{size}$ . Ezek a távolságok pontos meghatározása fontos a cellák összeillesztésekor, miven ezen számítások segítségével készül el a HexGrid.

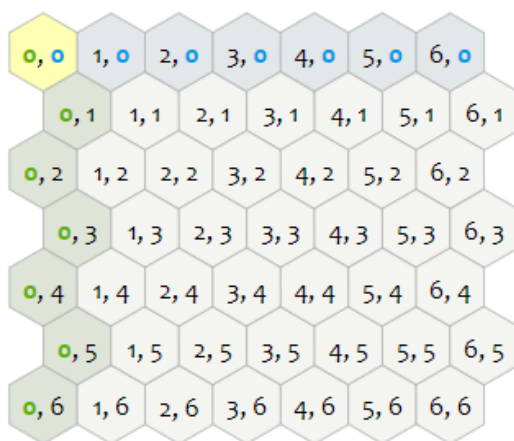


## 3.3 Koordináta rendszerek

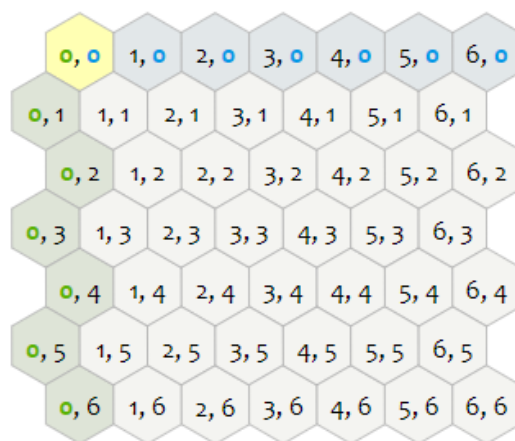
Általában, amikor egy „Grid felépítésű” pályát készítünk, akkor azt érdemes egy  $N * M$ -es (2 Dimenziós) tömbben eltárolni. Ezt a hexagonokkal is megtehetjük, de ha simán csak „Offset koordinátákat alkalmazunk”, akkor később problémákba ütközünk.

### 3.3.1 Offset koordináta

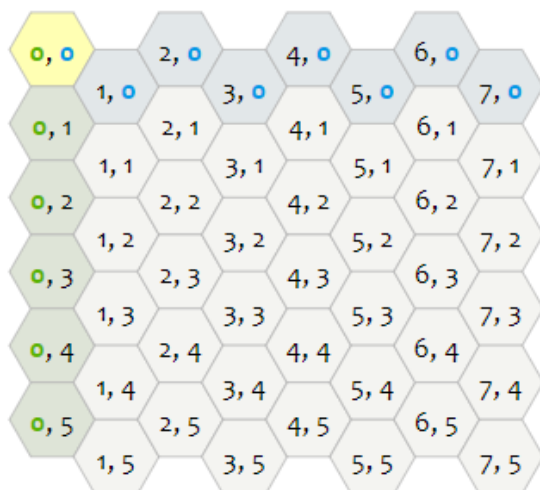
A leggyakoribb megközelítés az, hogy minden második oszlopot vagy sort eltolunk. Az oszlopokat  $\text{col}$  ( $q$ )-val jelöljük, míg a sorokat  $\text{row}$  ( $r$ )-rel. Eltolhatjuk a páratlan vagy a páros oszlopokat/sorokat, így a vízszintes és függőleges hexagonoknak két változata van. Ez az eltolásos módszer segít abban, hogy a hexagonok szabályosan és hézagmentesen illeszkedjenek egymáshoz, például hexagonális rácsok létrehozása során. Ezt a koordinátarendszert használok a HexGrid inicializálásakor. Én az „odd-r horizontal layout”-ot használok. Később áttérek a Cube koordinátákra, amelyek megkönnyítettek az életemet számos dologban.



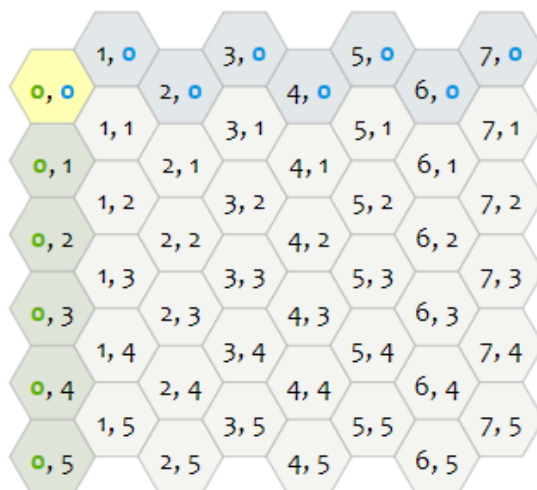
“odd-r” horizontal layout  
shoves odd rows right



“even-r” horizontal layout  
shoves even rows right



“odd-q” vertical layout  
shoves odd columns down



“even-q” vertical layout  
shoves even columns down

### Offset koordináták variánsai

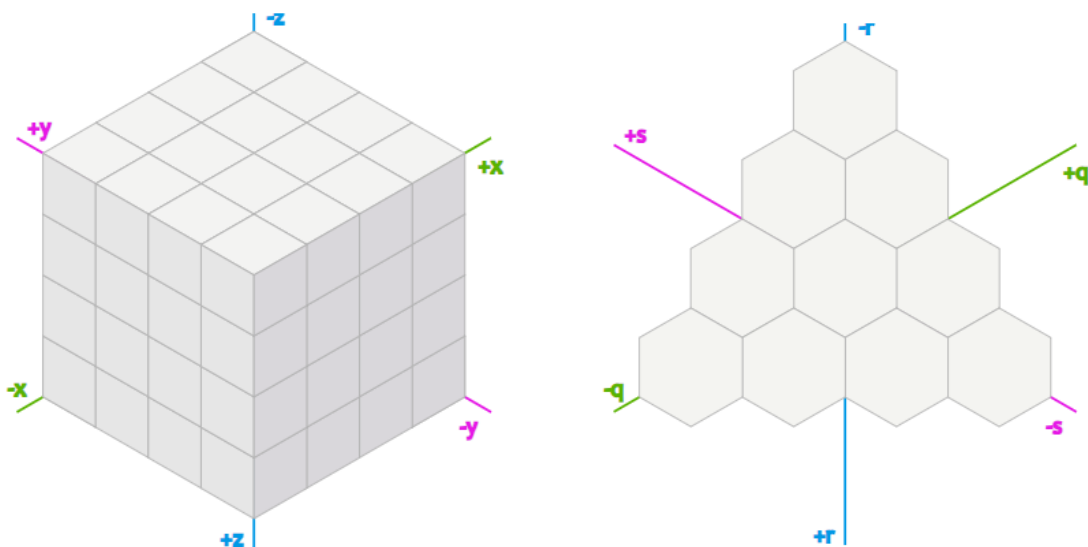
#### 3.3.2 Cube koordináta

Egy másik megközelítés a hexagonális rácsok vizsgálatánál az az, hogy felismerjük, hogy három elsődleges tengely van, ellentétben a négyzetrácsok két tengelyével. Ennek elegáns szimmetriája van.

Kell venni egy kockarácsot, amiből ki kell vágni egy átlós síkot az  $x + y + z = 0$  egyenlettel. Ez furcsának tűnhet, de segít a hexagonális rácsok algoritmusainak megértésében:

A 3D Descartes-koordináták követik a szokásos vektorműveleteket: hozzáadhatjuk és kivonhatjuk a koordinátákat, szorzatot és osztást is végezhetünk egy skalárral stb. Az eltolásos (offset) koordináták nem támogatják ezeket a műveleteket. A 3D Descartes-koordinátáknak léteznek meglévő algoritmusai, mint például távolságmérés, forgatás, tükrözés, vonalrajzolás, átalakítás képernyőkoordinátákra és vissza stb. Ezeket az algoritmusok közül én csak néhányat adaptáltam (Vonalrajzolás, távolságmérés és képernyőkoordinátákra alakítás). Minden irány a kockarácson megfelel egy vonalnak a hexagonális rácson. Például, az északnyugat a hexagonális rácson az  $+s$  és  $-r$  között helyezkedik el, így minden lépés északnyugat felé azt jelenti, hogy 1-et hozzáadunk  $s$ -hez és 1-et kivonunk  $r$ -ből. Ezt a tulajdonságot használom majd a szomszédos cellák szekciójában. A Cube koordináták ésszerű választásnak bizonyultak a hexagonális rács koordinátarendszereként. A korlátozás az, hogy  $q + r + s = 0$ , így az algoritmusoknak ezt meg kell őrizniük. Ez a korlátozás biztosítja azt is, hogy minden hexagonhoz egy kanonikus koordináta tartozik.

A hexagonális rácsokat tehát hatékonyan lehet kezelni a Cube koordináták segítségével, mivel ezek lehetővé teszik a szabványos vektorműveletek alkalmazását és a meglévő 3D algoritmusok adaptálását. A három tengely szimmetriája és a koordináták kombinációjának tulajdonságai elegáns és praktikus megoldásokat nyújtanak a hexagonális rácsokkal kapcsolatos problémákra. Vannak még ezeknek a koordinátarendszereknek variánsaik, melyeket most nem mutatok be (Nagyon hasonlóak).



### 3.3.3 Melyik koordináta rendszert érdemes használni

Egyszerűbb projektekhez, elég egy „alacsonyabb rendű” koordinátarendszer használata, mint például. az offset, de én egy elég komplex logikával rendelkező játékot készítettem, ahol mindkét korábban említett (offset/cube) koordinátarendszert használtam.

	Offset	Doubled	Axial	Cube
Pointy rotation	evenr, oddr	doublewidth	axial	cube
Flat rotation	evenq, oddq	doubleheight		
Other rotations	no		yes	
Vector operations (add, subtract, scale)	no	yes	yes	yes
Array storage	rectangular	no <sup>*</sup>	rhombus <sup>*</sup>	no <sup>*</sup>
Hash storage	any shape		any shape	
Hexagonal symmetry	no	no	no	yes
Easy algorithms	few	some	most	most

### 3.3.4 Koordináta rendszer konverzió

Projektemben az Offset/Cube koordinátarendszerek közötti konverziót használok. Ez oda vissza működik. Korábban említett képlet ( $q + r + s = 0$ ) kielégítésével lehet Offset-ből Cube koordinátákba átmenni, ahol „q” és „r” ismert Offset koordinátáknál, „s” pedig kiszámolható az egyenlet kielégítésével. Visszafele alakításnál pedig csak a „q” és „r” koordinátákra van szükségünk.



### 3.4 Szomszédos cellák

Egy hexagonális koordináta-rendszerben egy mezővel való elmozdulás azt jelenti, hogy a három Cube koordináta egyikét +1-gyel változtatjuk meg, míg egy másikat -1-gyel (az összegnek 0-nak kell maradnia). Három lehetséges koordináta van, amelyet +1-gyel lehet változtatni, és a maradék két koordinátából egyet -1-gyel. Ez hat lehetséges változást eredményez. Mindegyik egy-egy hexagonális iránynak felel meg. A legegyszerűbb és leggyorsabb megközelítés az, hogy előre kiszámítjuk ezeket a permutációkat, és egy tömbbe helyezzük őket Cube(dq, dr, ds) formában.

A Cubekoordináta-rendszerrel könnyen tárolható két koordináta közötti különbségek (mint "vektor"), majd ezeket a különbségeket hozzáadhatjuk egy koordináta-hoz, hogy egy másik koordinátát kapjunk. Az Offset koordináták ezt nem támogatják, hiszen „2D-s vektorról beszélünk”.

Ez a módszer különösen hasznos, mivel lehetővé teszi a gyors és hatékony navigációt a hexagonális rácson, megkönnyítve a mozgás és az irányok kezelését. Az előre kiszámított permutációk táblázata segít az irányok egyszerű és gyors meghatározásában, ami különösen hasznos lehet játékok és más interaktív alkalmazások fejlesztésénél.

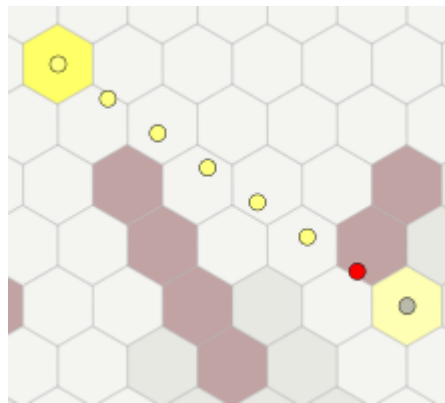
### 3.5 Távolság

Mivel a Cube koordináták a 3D-s kocka koordinátáin alapulnak, a távolság számítását is adaptálhatjuk a hexagonális rácsokra. Minden hexagon egy kockának felel meg a 3D térben. A szomszédos hexagonok távolsága 1 a hexagonális rácson, de 2 a kockarácson. Minden 2 lépés a kockarácson, csak 1 lépés a hexagonális rácson. A 3D kockarácson a Manhattan-távolság:  $\text{abs}(dx) + \text{abs}(dy) + \text{abs}(dz)$ . A távolság a hexagonális rácson ennek a fele. Ez a képlet segít meghatározni a két hexagon közötti távolságot, figyelembe véve a hexagonális koordináta-rendszer sajátosságait. Ezt a távolságot gyakran használtam játékomban a FOV (Field of View) kiszámításához.

### 3.6 Látható cellák (FOV)

Ahhoz, hogy meghatározzuk azt, hogy a katonák melyik cellákat látják, ahhoz először is meg kell határozni a FOV-t. Ezt a **3.5 Távolság** segítségével, le tudjuk szűkíteni néhány cellára, nem kell az összes cellát vizsgálni (Rekurzívan végig lehet iterálni egy cella szomszédjain). A legtávolabbi cellák és a katona cellája között, egyenlő közönként,

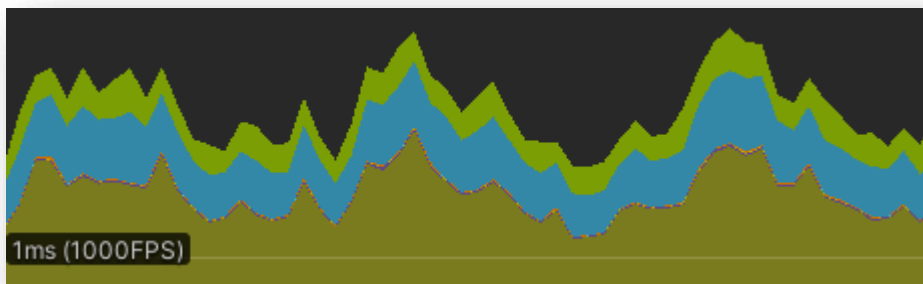
mintavételezünk pontokat (Lerp, linear interpolation). Ezeket a pontokat képernyőkoordináták alapján vissza tudjuk alakítani, hogy melyik cellát érinti. Az így elállított pontsorozat fogja megadni, hogy milyen messze lát a katona. Ha az egyik pont ütközést detektál, egy akadállyal (Hegy biommal), akkor a további pontok vizsgálata, már nem érdemes, hiszen azt nem fogja látni a katona. Ha nem érzékelünk ütközést hegygel, akkor sikeresen végig lehet iterálni az pontsorozat teljes hosszán, ami azt jelenti, hogy a katona a teljes FOV értékével egyenlő távolságra lát el adott irányba. Nyilván FOV mértékének függvényében nő az irányok/iterációk száma.



*FOV (Field of View)*

## 4 Optimalizálás, Teljesítmény

A játék tervezésekor sok problémába ütköztem, ami korábban ismeretlen tudás volt számomra. Nem gondoltam át rendesen, hogy mennyire terheli le CPU/GPU-t a programkód, amit írok. Szembesültem egy problémával, amikor pl. Update (ami minden frame-ben hívodik meg) függvényen belül minden cellán végig iteráltam. Ez nem egy optimális megoldás, hiszen nagy pályák esetén, ahol nem csak 1000 cella, van, hanem mondjuk 10000, nagyon sok függvény hívást igényel, ami le tudja terhelni a rendszert. Ezt sikerült elkerülni, constraint-ek (korlátozások) segítségével, ami szabályozza, hogy ne minden frame-ben kerüljön meghívásra, vagy ne az összes cellán keljen végig iterálni. Ezzel a technikával nagyobb pályák esetén átlagos ~50Fps-ről sikerült ~400-700Fps-t elérni. Optimalizálás keretein belül megismerkedtem a Unity beépített Profilerével, amivel ki tudtam deríteni, mi okozta a problémákat.



Statistics	
<b>Audio:</b>	
Level: -74.8 dB	DSP load: 0.1%
Clipping: 0.0%	Stream load: 0.0%
<b>Graphics:</b>	
585.9 FPS (1.7ms)	
CPU: main 1.7ms render thread 0.8ms	
Batches: 282 Saved by batching: 140	
Tris: 3.9k Verts: 4.4k	
Screen: 2560x1440 - 42.2 MB	
SetPass calls: 21 Shadow casters: 0	
Visible skinned meshes: 0	
Animation components playing: 0	
Animator components playing: 0	

## 5 Felhasznált források

<https://catlikecoding.com/unity/tutorials/mesh-basics/>

<https://catlikecoding.com/unity/tutorials/hex-map/>

<https://www.toptal.com/unity-unity3d/2d-camera-in-unity>

<https://discussions.unity.com/t/detect-cursor-on-edge-of-screen/70650/2>

<https://thingonitsown.itch.io/heros-hour/devlog/226750/random-map-generation-w-symmetry-and-adventure-objects>

<https://www.redblobgames.com/pathfinding/a-star/introduction.html>

<https://www.redblobgames.com/maps/noisy-edges/>

<https://www.redblobgames.com/grids/line-drawing/>

<http://www-cs-students.stanford.edu/~amitp/game-programming/polygon-map-generation/>

<https://www.redblobgames.com/grids/hexagons/>

<https://www.youtube.com/watch?v=VsUK9K6UbY4&list=PLzDRvYVwl53tpvp6CP6e-Mrl6dmxs9uhx>

<https://www.youtube.com/watch?v=mRDG5sQYjdo>

<https://www.youtube.com/watch?v=jVNC0Z2p9qw>

[https://www.youtube.com/watch?v=MG-v6H\\_Acx8](https://www.youtube.com/watch?v=MG-v6H_Acx8)

[https://www.youtube.com/watch?v=6mNj3M1il\\_c](https://www.youtube.com/watch?v=6mNj3M1il_c)

<https://www.youtube.com/watch?v=QG5i6DL7-to>

<https://www.youtube.com/watch?v=WiUUW9RSa5Y>