Széchenyi István Egyetem

**Gépészmérnöki, Informatikai és Villamosmérnöki Kar**

**Videójáték készítés**

**Unity, Procedurális generálás, Marching Cubes algoritmus**

**Készítette:**

**Piltz Gergely  
Mérnökinformatika szak**

**2024**

Tartalom

[Bevezetés 3](#_Toc175664354)

[A videójáték ipar 3](#_Toc175664355)

[Játék Leírása 3](#_Toc175664356)

[Elméleti Háttér 4](#_Toc175664357)

[Voxel és voxeljáték definíció 4](#_Toc175664358)

[Kettő Dimenziós Voxelek (Terraria) 4](#_Toc175664359)

[Három Dimenziós Voxelek (Minecraft) 5](#_Toc175664360)

[Mesh (háló) Előállítás, Megjelenítés 5](#_Toc175664361)

[Vertex 6](#_Toc175664362)

[vertices 6](#_Toc175664363)

[triangles 6](#_Toc175664364)

[Flat Shading vs Smooth Shading 6](#_Toc175664365)

[miért a marching cubesra esett a váálasztás 7](#_Toc175664366)

[Marching Cubes Algoritmus 7](#_Toc175664367)

[Hol használatos a Marching Cubes algoritmus 8](#_Toc175664368)

[Orvosi Tudomány 8](#_Toc175664369)

[CAD és szimulációs szoftverek 8](#_Toc175664370)

[Az algoritmus működése 9](#_Toc175664371)

[Skalármező 9](#_Toc175664372)

[Felület meghatározása 9](#_Toc175664373)

[Interpoláció 10](#_Toc175664374)

[Háló elemeinek előállítása 10](#_Toc175664375)

[Háló összeállítás 10](#_Toc175664376)

[A Sokszöghalmaz Előállítása 11](#_Toc175664377)

[Lookup Table 11](#_Toc175664378)

[Edge Table 12](#_Toc175664379)

[Corner Table 12](#_Toc175664380)

[Követelmények 14](#_Toc175664381)

[implemetálás 14](#_Toc175664382)

[A Marching Cubes Algoritmus hátulütői 14](#_Toc175664383)

[Műveletigény 14](#_Toc175664384)

[Magas Vertex Szám 15](#_Toc175664385)

[Az algoritmus optimalizálása 15](#_Toc175664386)

[Kocka és kocka közötti optimalizálás 15](#_Toc175664387)

[Kockán belüli optimalizálás 17](#_Toc175664388)

[Teljes kód 18](#_Toc175664389)

[Az optimalizáció eredménye 19](#_Toc175664390)

[Pszeuorandom Generálás 20](#_Toc175664391)

[Hivatkozások 20](#_Toc175664392)

# Bevezetés

## A videójáték ipar

A videójáték ipar az informatika fejlődésével együtt rohamos lépésben halad előre. Az informatika és azon belül a videójáték fejlesztés is egy nagyon összetett iparág, amiben mindenki megtalálhatja a saját stílusának, érdeklődési körének megfelelő irányt. A videójáték fejlesztés emellett egy magas profitot generáló terület, kifejezetten az elmúlt években.

A globális videojáték-piac 2022-ben 217,06 milliárd dollárra volt becsülve és 2023 és 2030 között 13,4%-os összetett éves növekedési rátára lehet számítani. 2030-ra vélhetően meg fogja haladni a 580 milliárd dollárt. A piac bővülése köszönhető az online játékok térhódításának, amelyet a széles sávszélességű internetkapcsolat széleskörű elérhetősége tett lehetővé, emellett a mobiljátékok iránti kereslet is növekszik, miközben a 3D játékokra is folyamatosan nagy a kereslet. (Grand View Research, n.d.)

A gazdaság hullámvölgyei ellenére a videojáték-ipar figyelemre méltó rugalmasságot mutatott és évről évre tovább növekszik. Amíg van igény a szórakoztatásra, addig lesznek lehetőségek a játékfejlesztésben.

A videójáték készítés szenvedély és munka is lehet egyben. Egy olyan karrier választása, amit élvezünk nagyon fontos. A videójáték egy egyedülálló termék, amely egyesíti a művészetet, a történetmesélést és a technikai megvalósítást. A játékfejlesztés általában különböző készségekkel rendelkező fejlesztőkből álló csapatokat foglal magában, köztük művészeket, programozókat, írókat és tervezőket. Az ebben az iparágban való munkavégzés lehetőséget ad kreatív önkifejezésre, legyen szó karakterek tervezéséről, a videójáték világának létrehozásáról, vagy a narratívák megalkotásáról. A videojáték-ipar karrierutak széles skáláját kínálja, a játéktervezéstől és programozástól a marketingig és a közösségi menedzsmentig.

A videojátékok globális hatókörrel rendelkeznek, és emberek millióinak életét érinthetik szerte a világon. Legyen szó szórakoztatásról, társadalmi kapcsolatok ápolásáról, vagy akár az oktatás és az empátia előmozdításáról, a videojáték-iparban végzett munka lehetővé teszi, hogy jelentős hatást érjünk el nagy léptékben.

A videójátékok alapjaiban az informatika szokásos image-e jelenik meg, mely a problémamegoldást, programozást és a különböző funkcionalitások megvalósítását jelenti. A videójáték fejlesztés, mint terület nagyon dinamikus, a technológia folyamatosan fejlődik, melynek eredménye képpen újabb és újabb kihívásokkal állítja szembe a fejlesztőket. Ennek eredménye képpen a videójátékok is egyre összetettebbek és realisztikusabbak. Ám nem feltétlenük kell a realizmust és a komplexitást üldözni, hiszen egyszerűbb grafikával és játékmenettel rendelkező játékok is emberek millióinak nyújtanak elfoglaltságot nap mint nap.

## Játék Leírása

A játék megvalósítása Unity-ban történik. Ez több előnnyel is jár, mint például, hogy a játékmotort nem kell kifejleszteni. A játék világa Open World és egy random generált pályán játszódik.

Unity

A Unity a Unity Technologies által fejlesztett videójáték motor. A szoftverrel való videójáték készítés Windows-on, MacOS-en és Linuxon is lehetséges, a játékot futtatni pedig virtuálisan bármilyen rendszeren lehetséges, legyen az Xbox vagy PlayStation széria, Windows, Linux vagy Mac számítógép, vagy akár mobil. A Unity alkalmas 2D és 3D játékok készítésére is, valamint használják még egyéb interaktív tartalmak létrehozására, például építészeti látványtervek készítésére.

Open World

Az Open World játékok jellemzői, hogy a játékos nincs megkötve abból a szempontból, hogy a pályán hova mehet. Egy Open World játékban általában jelen van progresszió, amely végig vezeti a játékost sorban a világ különböző részein, de nincs megkötve a játékos a különböző területek közötti mozgásban. Az Open World játékok fontos eleme a fő történetszál mellett a világ szabad felfedezése, amely az irányított történet és progresszió mellett extra tartalmat szolgáltat a játékosnak és nagyon fontos a teljes játékélmény szempontjából.

Procedurális Generálás

A procedurális generálás a teljes pályára kiterjed. Mind maga a pálya és a pályán jelen lévő tartalom is minden új pálya létrehozásakor különböző, de egy adott „Seed”-hez kötött. A Seed a játék egy olyan érték, amely a pszeudorandom algoritmusoknak egy kezdőértéket szolgáltat. Ugyan arra a Seed-re a játék ugyan azt a pályát fogja visszaadni.

CÉL: EGY OLYAN VÁZ KÉSZÍTÉSE AMELY LEHETŐSÉGET AD EGY VOXEL ALAPÚ JÁTÉK KIALAKÍTÁSÁRA. VOXELEK ÉS SANDBOY RÉSZE MEG LEGYEN ALAPOZVA.

# Elméleti Háttér

## Voxel és voxeljáték definíció

A voxeljáték egy olyan videojáték, amelyben a környezet és a benne lévő tárgyak voxelekből állnak. A voxelek alapvetően térfogati pixelek, amelyek egy háromdimenziós rács értéket képviselnek. A hagyományos 2D pixelekkel ellentétben, amelyeknek csak szélessége és magassága van, a voxeleknek mélységük is van, ami lehetővé teszi teljesen háromdimenziós környezet létrehozását. A voxel és a voxeljáték definíciója elég laza és gyakran kéz a kézben jár a sandbox játékokkal.

A voxel-játékokban a játékosok gyakran úgy lépnek kapcsolatba ezekkel a környezetekkel, hogy módosítják vagy megsemmisítik a voxeleket, vagy létrehoznak új voxeleket, lehetővé téve a terep dinamikus deformációját. Mivel ez nem egy előre definiált folyamat, hanem a játékostól függ, hogy mit tesz, ezért egy ilyen játéknak egy olyan rendszert kell prezentálnia, amely előre definiált interakciókon lehetővé tesz „végtelen sok” konfigurációt. A voxeljátékok kettő jól ismert példája a Minecraft és Terraria.

## Kettő Dimenziós Voxelek (Terraria)

A Terraria egy 2011-ben kiadott 2D videójáték. A Re-Logic fejleszti, akik először Windows-ra adták ki, majd más PC- és konzolplatformokra is. A játék alapból sandbox jellegű, a felfedezésre és kreatív építésre inspirál első sorban egy procedurálisan generált pályán. A Terraria sok pozitív visszajelzést kapott a közösség felől, ennek köszönhetően 2022-ig több mint 44 millió példány kelt el belőle. A Terraria az eddigi egyik legkelendőbb videójáték.

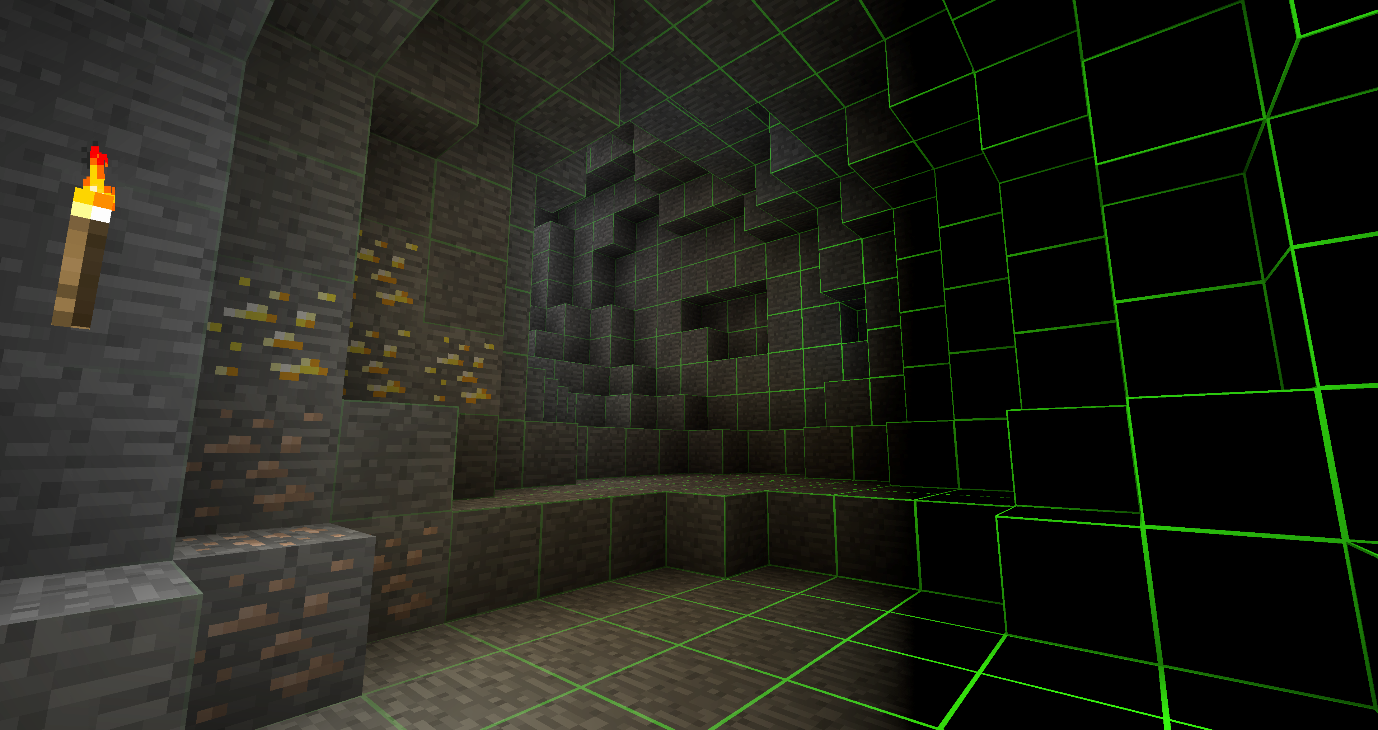
Bár a voxel egy térfogati pixelként van értelmezve, a Terraria is a voxeljátékok közé sorolható. A Terraria egy kétdimenziós rácsot vesz alapul, amelyen belül különböző blokkok helyezhetőek le. Az előre definiált interakciók blokk-blokk és blokk-entitás (pl. játékos) szinten vannak jelen, ez teszi lehetővé a világ dinamikus változtatását és különbözteti meg a játékot más, statikus pályákkal rendelkező hasonló játékoktól.

## Három Dimenziós Voxelek (Minecraft)

A Minecraft egy 2011-ben kiadott sandbox játék, a legnagyobb példányszámban eladott játék a világon. A Mojang Studios által fejlesztett játékból több mint 300 millió darabot adtak el eddig. Korai tesztelési verziókat követően először 2009 májusában hozta nyilvánosságra a játékot az eredeti fejlesztője, Markus "Notch" Persson. 2011. november 18-án adták ki hivatalosan amikor Notch lemondott és Jens "Jeb" Bergensten vette át a fejlesztést. Hamar népszerűvé vált és a világ legkelendőbb videójátéka lett, közel 140 millió havi aktív játékossal. A Minecraft-ból adtak ki verziókat mobil és konzol kompatibilitással is.

A Minecraft egy 3D kockás, pixeles, procedurálisan generált világban játszódik, gyakorlatilag végtelen tereppel. A Minecraft több szempontból is egy technikai csodának számít. A világ annak ellenre, hogy véletlenszerűen generált nagyon precízen illeszti össze a sok különböző tartalmat, miközben játékbeli méretekkel számolva több mint 3.6 milliárd négyzetkilométer. Referenciának, a föld, mint bolygó felszíne félmilliárd négyzetkilométerre becsülhető.

A Minecraft-ben a voxelek blokkoként vannak jelen. Egy blokk a tér egy kockányi területét definiálja egy háromdimenziós kockaháló mentén. Ezek a blokkok interaktálnak egymással és az entitásokkal, például a játékossal.



1. ábra Minecraft

A Voxel technológia lehetővé teszi a nagyméretű környezetek rugalmas és hatékony megjelenítését, ami hozzájárult a voxel alapú játékok népszerűségéhez. Ezen túlmenően a voxeljátékokhoz gyakran társított kockás esztétika ikonikussá és felismerhetővé vált a játékközösségben.

## Mesh (háló) Előállítás, Megjelenítés

A számítógépes grafikában használt mesh (háló) kifejezés, egy olyan adatstruktúrára utal, amely 3 dimenziós objektumok felszínét definiálja poliéder formájában. A poliéder egy olyan térbeli alakzat, amelyet pontok, élek és oldallapok definiálnak, vagy másszóval egy poliédert minden oldalról síkok határolnak. Poliéder például a kocka, de a gömb nem, hiszen a gömbnek nincs sík felszíne. A számítógépes grafikában íves felületeket is hálók segítségével jelenítenek meg, akkor azonban a felület le van egyszerűsítve síklapokra.

## Vertex

Vertex-nek nevezünk egy olyan pontot, ahol kettő vagy több él találkozik. Ezekből definiálhatóak az élek, amelyek körbezárnak egy adott poligont.

Egy háló használhat bármilyen N-csúcsú poligont a felszín definiálására, de a legelterjedtebb megoldás a háromszöget veszi alapul. Egy háromszög alapú hálóra Triangle Mesh-ként hivatkoznak.

Egy háló definiálása többfajta adatszerkezettel is lehetséges. Az ilyen adatszerkezetek rendelik hozzá a vertex-elket a poligonokhoz, Unity esetében háromszögekhez.

## vertices

A vertices (vertex többesszáma) tömb egydimenziós adatszerkezet, amelyen belül 3 dimenziós vektorok (Vector3) találhatók. Ezek a háló pontjainak térbeli elhelyezkedését határozzák meg. Tulajdonképpen egy hosszú felsorolás az összes vertex-ről. (Az ismétlődés meg van engedve)

Unity-n belül a maximális vertex szám egy hálón belül 16-bites buffer esetén 65535 vagy több mint 4 milliárd 32-bites buffer esetén. A 16-bites buffer az alapértelmezett, hiszen feleannyi memóriát használ, mint a 32-bites buffer.

## triangles

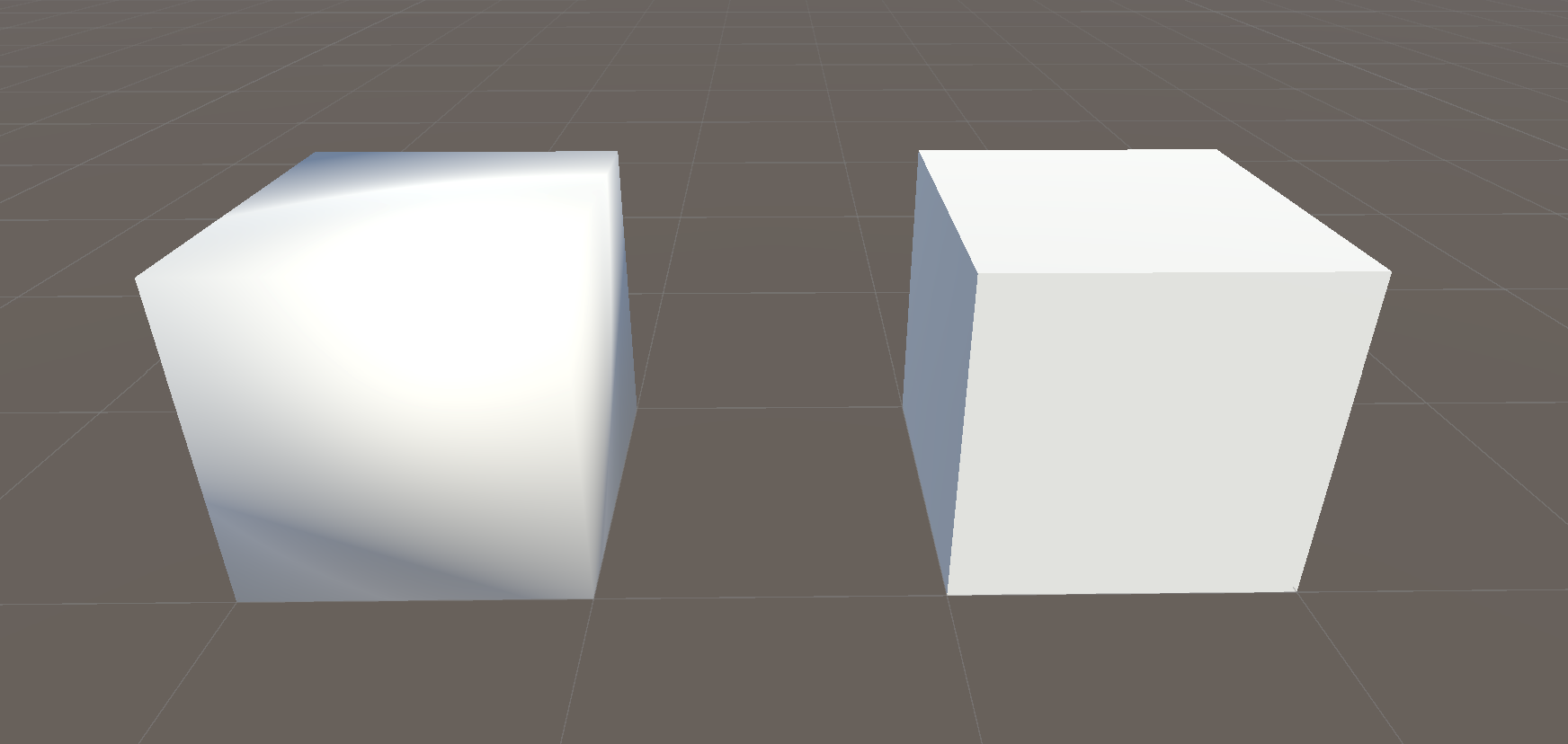
A triangles tömb szintén egydimenziós adatszerkezet, amely indexeket (Integer) tartalmaz, amelyek a vertices tömbre mutatnak. A tömb 3-as indexcsoportonkként van értelmezve. Minden hármas csoport egy háromszöghöz rendeli hozzá a háromszög csúcsainak a vertices tömbben definiált pozícióját indexelés segítségével. Ezekből következik, hogy a triangles tömb elemeinek száma mindig 3 többszöröse.

## Flat Shading vs Smooth Shading

A Shading a Mesh megvilágítására vonatkozó kifejezés. Grafikai megjelenítéskor a megvilágítás a vertex-ek normáljaiból vannak kiszámítva. Egy vertex normálja az élektől függ, amelyeknek tagja. Ha egy vertex csak egy háromszöghöz tartozik az azt jelenti, hogy csak kettő élnek a végpontja, amelyekből egyértelműen kiszámítható a normálvektor. Ha egy vertex több háromszögnek, vagyis kettőnél több élnek a végpontja, akkor a normált az adott háromszögekhez tartozó élekből kiszámított normáloknak az átlagolásával lehet megkapni.

Flat Shading-nek nevezzük amikor az éleken és sarkokon találkozó háromszögek külön vertex-ekkel rendelkeznek, ezért a megvilágításuk is határozottan különböző, vagyis az él/sarok tisztán kivehető.

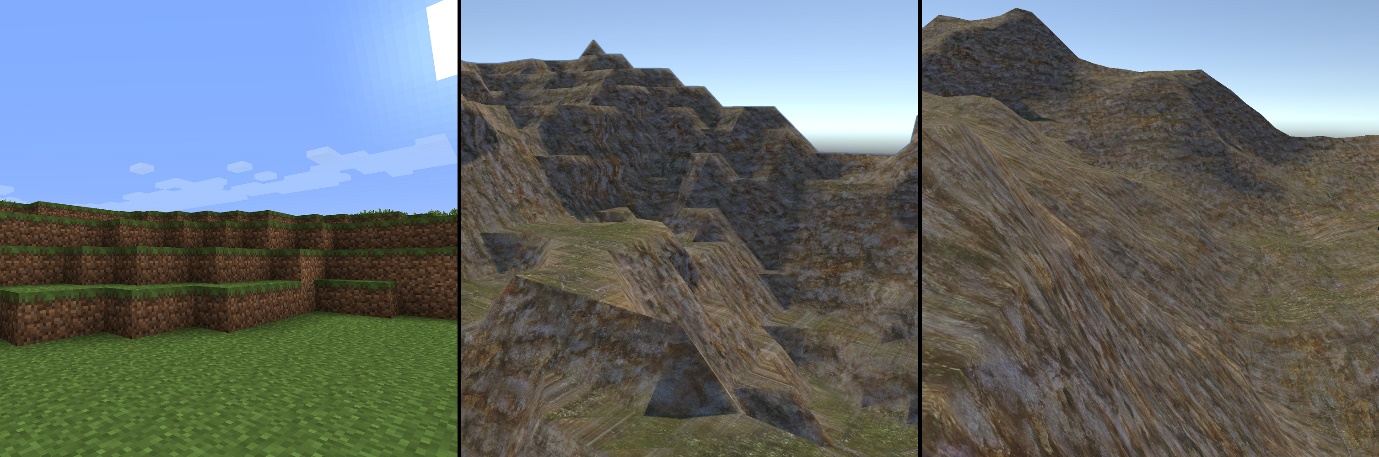
Smooth Shading-nek nevezzük amikor az éleken és sarkokon találkozó háromszögek ugyan az a vertex-et használják, tehát a felületek megvilágítása folytonosan változik oldalról oldalra, ezért az élek és sarkok nehezebben kivehetők.



2. ábra Smooth Shading (bal) vs Flat Shading (jobb)

## miért a marching cubesra esett a váálasztás

A Marching Cubes algoritmus a Minecraft-al ellentétben a kockák szintje alá megy precizitás szempontjából. Míg a Minecraft-ban egy blokk vagy jelen van egy koordinátán, vagy nem, addig a Marching Cubes egy kockához 256 konfiguráció közül társít egyet. Ez lehetővé tesz a talaj lényegesen nagyobb felbontásban való megjelenítését. Ez a precizitás két lépcsőben is jelen van.



3. ábra Kockás (bal) Durva (közép) és Simított felszín (jobb)

## Marching Cubes Algoritmus

A Marching Cubes algoritmus a számítógépes grafikában használt módszer, amellyel háromdimenziós skalármezőből poligon hálót lehet létrehozni. Különösen hasznos „isosurface” renderelésére. Az algoritmust William E. Lorensen és Harvey E. Cline fejlesztette ki az 1980-as években az amerikai General Electric számára folytatott kutatásuk közben. A kutatás célja az MRI és CT scan-ek effektív számítógépes megjelenítése volt. (Lorensen & Cline, 1987)

A háromdimenziós skalármező alatt egy térbeli négyzetrácsot kell elképzelni, amelynek minden pontja képvisel egy adott értéket. Ezeknek a térbeli pontoknak az értelmezésével lehet felületeket létrehozni, például sűrűségre hivatkozva. Például úgy, hogy a pontok, amelyek egy alakzat belsejében helyezkednek el magas konstans értékkel rendelkeznek, míg amik az alakzaton kívül vannak, alacsony értéket vesznek fel.

Az isosurface egy olyan felület, amely állandó értékű pontokat (pl. nyomás, hőmérséklet, sűrűség) képvisel egy térfogaton belül; más szóval, ez egy olyan folytonos függvény szinthalmaza, amelynek tartománya a 3 dimenziós tér. A pontok általában 3 dimenziós négyzetrács mentén helyezkednek el és a felvett értékeik határozzák meg a megjelenített alakzat felszínét.

## Hol használatos a Marching Cubes algoritmus

A Marching Cubes algoritmust széles körben használják olyan alkalmazásokban, mint az orvosi képalkotás, a tudományos vizualizáció és a számítógéppel segített tervezés (CAD), összetett háromdimenziós struktúrák skaláris adatokból történő megjelenítésére.

## Orvosi Tudomány

A Marching Cubes algoritmus egyik elterjedt felhasználási területe az orvosi tudomány. Az MRI (Magnetic resonance imaging) és CT (computed tomography, vagy CAT, computed axial tomography) scan-ek a technológiájából adódóan e algoritmus használatához megfelelő adatokat generálnak.

Az ilyen scan-ek általában 2 dimenziós képekként vannak megjelenítve szeletenként, ahol egy kép egy adott pozícióban lévő metszetet szimbolizál szürkeárnyalatos módon. A különböző árnyalatok az adott metszet adott pontjainak az intenzitását mutatja. A CT és MRI szkennerek különböző szövetekről különböző intenzitású jeleket olvasnak le. Ezeknek a scan-eknek a precizitása nagyon magas, ezért könnyen különbséget tudnak tenni akár erek és izomszövet közt is. (National Institute of Biomedical Imaging and Bioengineering, 2022)

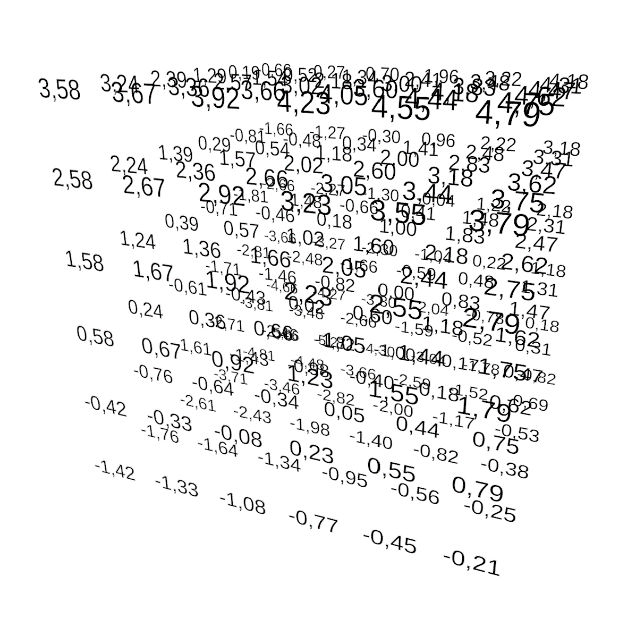
Ezeknek az értékeknek a megjelenítése lehetséges a Marching Cubes algoritmussal, úgy, hogy ha ismertek a különböző szövetekhez tartozó jellemző konstansokat, akkor a scan értékeiből és térbeli pozíciójukból kiszámítja a különböző alakzatok felületét, például a csontoknak az alakját.

## CAD és szimulációs szoftverek

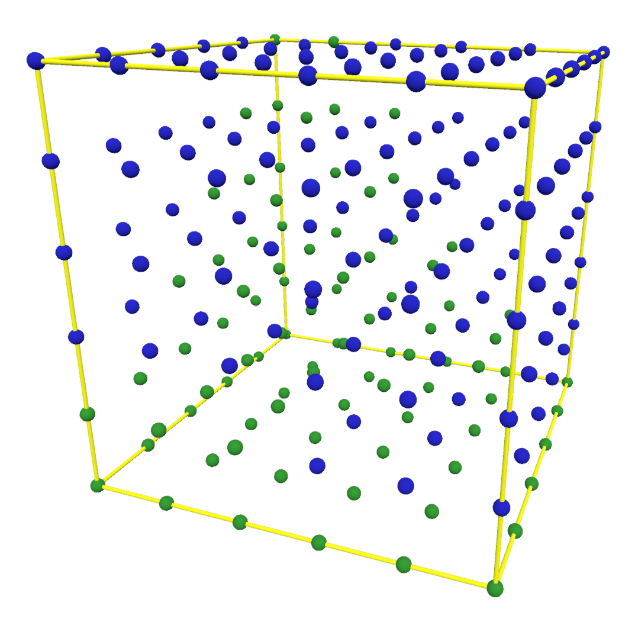
A Marching Cubes algoritmust CAD (számítógéppel segített tervezés) szoftverek is alkalmazzák, ahol a felhasználó saját maga készíti el az alakzatot, amelyet a számítógép megjelenít. A Marching Cubes algoritmus szimulációk kiértékelésére is használatos, például folyadékdinamikai szimulációk vizualizálására. Egy Marching Cubes által megalkotott isosurface reprezentálhat repülőgép propeller által keltett lökéshullámokat.

Érdemes megjegyezni, hogy bár a Marching Cubes úttörő volt a bevezetésekor, az algoritmust később változtatták és fejlesztették olyan problémák megoldására érdekében, mint a topológiai helyesség és a háló minősége.

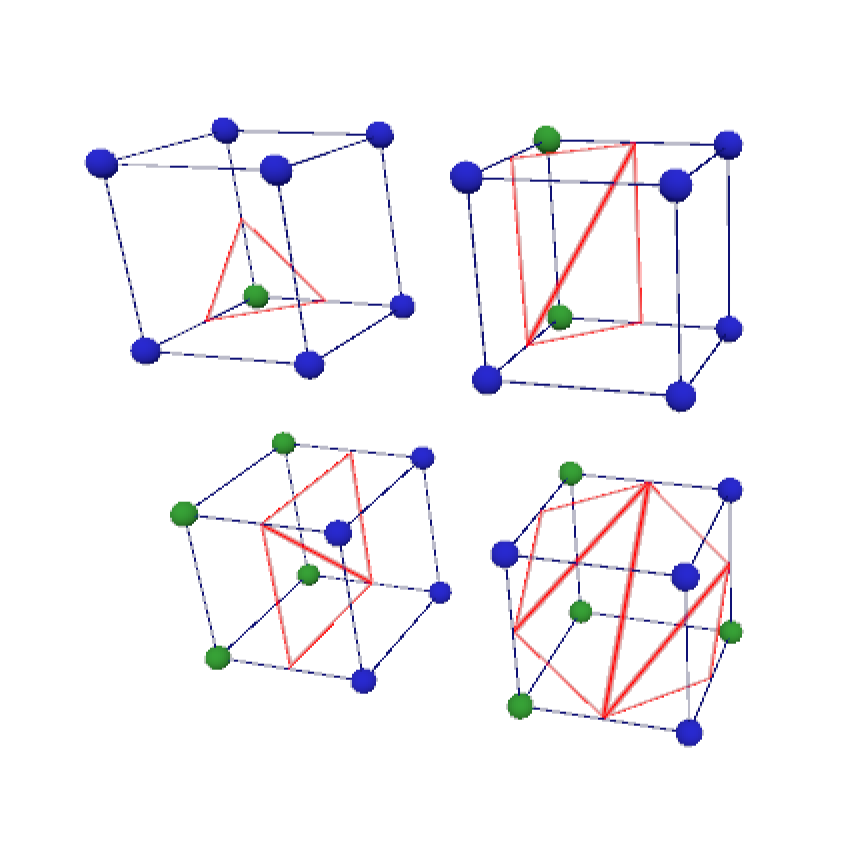
## Az algoritmus működése



4. ábra Skalármező



5. ábra Kiértékelt skalármező



6. ábra Példa négy konfigurációra

Skalármező: Az első lépés egy háromdimenziós skalármező meghatározása, ahol a térben egy kockaháló mentén minden pont egy skalárértéknek felel meg. Ezek a skaláris értékek különféle tulajdonságokat jelenthetnek, például sűrűséget, hőmérsékletet vagy mesterségesen generált értékek is lehetnek. Lásd 4. ábra Skalármező

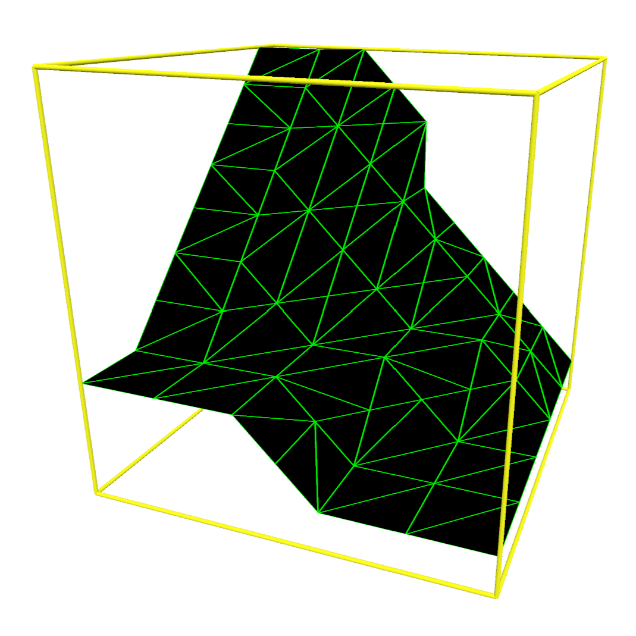
Egy kockahálóban minden pont nyolc kockának a sarka egyszerre. Ebből fakad, hogy minden skalárérték egyszerre nyolc kockához tartozik. (Kivéve, ha a háló valamely oldalán vagy élén helyezkedik el)

Felület meghatározása: Az algoritmus minden kockánál kiértékeli a skaláris értékeket a nyolc csúcsnál és meghatározza, hogy a felület metszi-e a kockát.

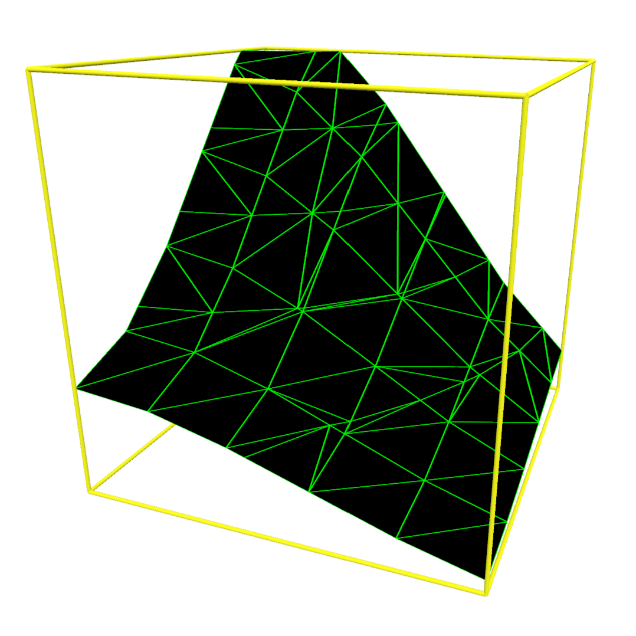
Ez úgy történik, hogy a skaláris értékeket egy előre meghatározott konstanssal hasonlítja össze, amely a vizsgált felületet reprezentáló állandó érték. A konstansnál kisebb értékkel rendelkező pontok például a felület alatt, míg a konstansnál nagyobb értékek a felület felett helyezkednek el. Lásd 5. ábra Kiértékelt skalármező.

Ez egy bináris döntés a kocka mind a nyolc csúcsán. Ezen összehasonlítások alapján egy kocka 28 = 256 lehetséges konfiguráció egyikébe eshet.

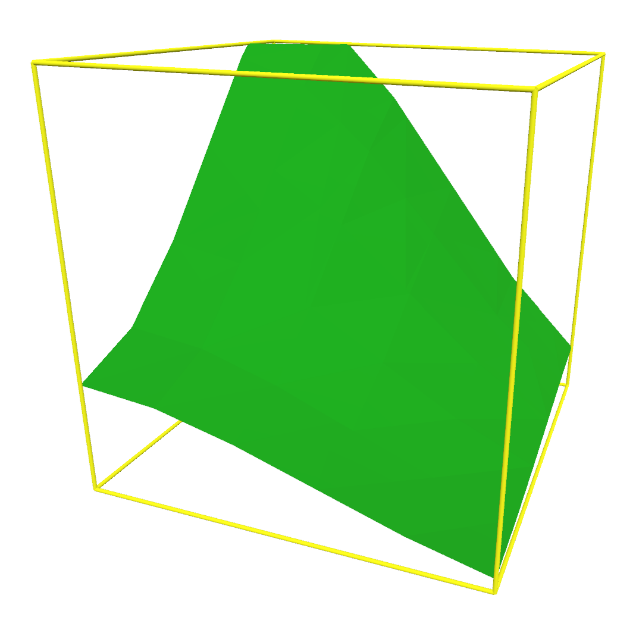
A kocka konfigurációja határozza meg, hogy a felület azon részét képező háromszögek pontjai mely éleken helyezkednek el.   
Lásd 6. ábra Példa négy konfigurációra



7. ábra Simítatlan felület



8. ábra Simított felület



9. ábra Renderelt Mesh

Interpoláció: Ha a kocka konfigurációja ismert, az algoritmus meghatározza a felület metszéspontjait a kocka éleivel.

A legegyszerűbb megoldás veszi a felezőpontot a két csúcs között és ott halad át a felület. Ez összeállítás után durva és pontatlanabb felszínt eredményez. Lásd 7. ábra Simítatlan felület

A matematikailag korrekt megoldás a metszéspontokat a kocka csúcsainak skalárértékeiből interpolálja oly módon, hogy az értékek közötti különbség fogja meghatározni a felület metszéspontját és a kocka élén való elhelyezkedését. Ez összeállítás után simább és pontosabb felszínt eredményez. Lásd 8. ábra Simított felület

Háló elemeinek előállítása: Az előző lépésben kiszámított metszéspontok felhasználásával az algoritmus háromszögeket generál a kockán belül, hogy megközelítse a felület azon részét, amely metszi a kockát. A kocka minden konfigurációja egy adott háromszöghalmaznak felel meg.

Háló összeállítás: Végül az egyes kockákhoz létrehozott halmazokat összeilleszti, hogy egy folytonos hálót képezzenek, amely a felületet képviseli. Ez a háló ezután hagyományos grafikai technikákkal megjeleníthető. Lásd 9. ábra Renderelt Mesh

## A Sokszöghalmaz Előállítása

Hogy a folyamat könnyebben értelmezhető legyen, az algoritmus három előre definiált segédhalmazt használ. Ezek a halmazok konstans értékeket tartalmaznak, amelyek irányítják az algoritmust, hogy elérjen egy adott konfigurációig, amelyet már használni lehet.

Táblázatok forrásai:

* Marching Cubes Project on Github by Sebastian Lague
  + <https://github.com/SebLague/Marching-Cubes/blob/master/Assets/Scripts/Compute/Includes/MarchTables.compute>
* Paul Bourke (1994)
  + <https://paulbourke.net/geometry/polygonise/>
* Marching Cubes Project on Github by b3agz
  + <https://github.com/b3agz/how-to-make-7-days-to-die-in-unity/blob/master/01-marching-cubes/Marching.cs>

## Lookup Table

A Marching Cubes algoritmusban a kockák limitálva vannak 256 különböző konfigurációra oly értelemben, hogy mely éleken helyezkednek el a háromszögek csúcsai, amelyek az adott konckán belüli felületet reprezentálják. Ezek a konfigurációk előre ismertek, ezért fölösleges lenne őket újra kiszámolni minden egyes alkalommal. Erre az algoritmus egy „Lookup Table”-t használ, amely 256 sorral rendelkezik.

public static readonly int[,] TriangleTable = new int[,] {

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},

{0, 8, 3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},

{0, 1, 9, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},

...

Minden sorban, felsorolás szerűen jelen vannak az éleknek az indexei, amelyeken egy háromszög csúcs jelen van. Egy kockán belül maximum 5 háromszög lehet jelen, ezért a Lookup Table egy 256 x 15-ös (vagy 16) kétdimenziós tömbként értelmezhető. A sorok indexei jelentik az adott konfigurációt, a sorokon belül pedig hármas csoportokban jelen vannak a háromszögek csúcsaihoz tartozó indexek. A kocka éleinek a pozíciója, vagyis a kezdő és végpontja ismert, tehát ha tudjuk melyik élen van az adott pont, amit a TriangleTable indexekkel megad az adott konfiguráción belül, akkor az él két csúcsa között csak interpolálni kell a pontnak a pozícióját.

Megjegyzés:

A TriangleTable -1 értékei azt jelölik, hogy nincs több index az adott sorban, tehát a sornak a bejárását meg lehet szakítani. A sorok a példában 16 elemet tartalmaznak, azonban az utolsó mindig -1. Ez mindössze preferencia kérdése. Ha FOR ciklust használunk a sorok bejárására, az mindig megáll a 15. elemnél, ha viszont WHILE ciklust használnánk annak a -1 jelezné, hogy ki kell lépni a ciklusból.

Az 1. és 256. sor csak -1-et tartalmaz, mivel azokban a konfigurációkban a felület nem halad át a kockán, mert vagy felette vagy alatta halad el. A legtöbb implementáció, ha a konfigurációra az elsőt vagy az utolsót kapja, akkor azonnal a következő kockára ugrik. Ez nem kötelező, viszont megspórol pár műveletet.

if (configIndex == 0 || configIndex == 255) continue;

## Edge Table

Ez a táblázat mindössze a kocka éleit definiálja úgy, hogy az algoritmust átirányítja a kettő megfelelő sarokhoz, amelyek az él kezdő és végpontjai. Az átirányítás a Corner Table-ben történik indexelés segítségével.

public static readonly int[,] EdgeTable = new int[12, 2] {

{0, 1},

{1, 2},

{3, 2},

...

## Corner Table

Ebben a táblázatban a kocka csúcsai vannak jelen 3 dimenziós vektorok formájában. Ezeket a vektorokat hozzáadva a kockának a (0, 0, 0) csúcsához mind a nyolc csúcsot el lehet érni.

public static readonly Vector3Int[] CornerTable = new Vector3Int[8] {

new Vector3Int(0, 0, 0),

new Vector3Int(1, 0, 0),

new Vector3Int(1, 1, 0),

new Vector3Int(0, 1, 0),

new Vector3Int(0, 0, 1),

new Vector3Int(1, 0, 1),

new Vector3Int(1, 1, 1),

new Vector3Int(0, 1, 1)

};

Az Edge Table és a Corner Table használata nélkül is meg lehetne írni az algoritmust, ám mivel ezek az értékek konstansok és fontos, hogy mindig ugyan olyan sorrendbe következzenek a konfiguráció kiértékelése során, ezért egyértelmű, hogy ha kiemeljük őket konstans tömbökbe, az lecsökkenti a hibalehetőségek számát és kompaktabbá, jobban olvashatóvá teszi a kódot. Ennek a folyamatnak elhanyagolható műveletigénye van ahhoz képest, hogy mennyivel olvashatóbbá teszi a kódot.

Egy példa:

Az alább látható FOR ciklusban a kocka sarkai vannak kiértékelve. Ha a tömbön végigmegyünk a 0-ás indextől kezdve, akkor mindig ugyan abban a sorrendben fognak következni a csúcsok, amelyeket egy indexel elérünk, nem kell minden csúcsnak a háromdimenziós eltolását kiírni a (0, 0, 0) csúcstól.

int GetCubeCongif(float[] cube)

{

int configIndex = 0;

for (int i = 0; i < 8; i++)

if (cube[i] > surfaceHeight)

configIndex |= 1 << i; //bitshift

return configIndex;

}

Ezeknek a tömböknek a „konstanssága” nagyon fontos, hiszen az algoritmus mindig ugyan abban a sorrendben kell, hogy kiértékelje az adott konfigurációt, főleg, hogy a kód szimplifikálása érdekében a konfiguráció meghatározásakor bitshift-et használ. A bitshift csak akkor fogja a megfelelő konfigurációt visszaadni, hogyha a konfigurációk is sorban vannak és a csúcsok kiértékelése is minden egyes alkalommal ugyan abban a sorrendben történik.

# Követelmények

|  |  |  |
| --- | --- | --- |
| **Alap követelmények** | | |
| **Megnevezés** | **Leírás** | **Prioritás** |
| Futtatható állomány | Alkalmazás indítása egy futtatható állományból | Magas |
| Főmenü | Az alkalmazás indítás után a főmenüt jeleníti meg | Magas |
| Játék indítása főmenüből | A főmenüből lehet indítani a játékot | Magas |
| Játék megállítása kilépés nélkül | Játék közben ESC megnyomására a játék ideiglenesen megáll amíg újra nem indítják | Közepes |
| Játék közbeni főmenübe kilépés | A játékból lehetőség van a főmenübe visszalépni | Magas |
| Alkalmazásból való kilépés főmenün keresztül | A főmenüben lehetőség van kilépni az alkalmazásból | Magas |
| Alkalmazásból való kilépés PAUSE menüből | A játék ESC gombbal való megállítását követően lehetőség van kilépni az alkalmazásból | Alacsony |
| PAUSE menü | Játék közben ESC gombbal való megállításkor menü megjelenítése | Közepes |

**Játékmenet követelmények**

|  |  |  |
| --- | --- | --- |
| **Mozgás** | | |
| **Megnevezés** | **Leírás** | **Prioritás** |
| Kamera irányítása egérrel | Egér mozgatásával a kamera mozog a függőleges és vízszintes tengely mentén is | Magas |
| Előre mozgás – W | W megnyomására a karakter előre mozog  (horizontális síkon kameraszöghöz képest relatív) | Magas |
| Hátra mozgás – S | S megnyomására a karakter hátra mozog  (horizontális síkon kameraszöghöz képest relatív) | Magas |
| Jobbra mozgás – D | D megnyomására a karakter jobbra mozog  (horizontális síkon kameraszöghöz képest relatív) | Magas |
| Balra mozgás – A | A megnyomására a karakter balra mozog  (horizontális síkon kameraszöghöz képest relatív) | Magas |
| Ugrás – Space | A Space megnyomására a karakter 1 blokknyit ugrik felfele | Közepes |
| Guggolás – Bal CTRL | A bal-CTRL megnyomására a karakter magassága lecsökken és sebessége lassabb lesz | Alacsony |
| Futás – Bal Shift | A bal-shift megnyomására a karakter gyorsabb lesz | Alacsony |

|  |  |  |
| --- | --- | --- |
| **Környezeti interakció** | | |
| **Megnevezés** | **Leírás** | **Prioritás** |
| Ütközés környezeti elemekkel | A karakter a környezetet alkotó objektumokkal (talaj, növényzet…) ütközik, vagyis nem tud rajtuk keresztül menni/esni | Magas |
| Játékfizika alkalmazása | Lefele ható gravitáció, talajjal való súrlódás | Magas |
| Blokk kiütése – Bal egérgomb | Bal egérgombbal a blokk, amelyre a játékos néz megsemmisül, ha a blokk elérhető távolságon belül van | Közepes |
| Blokk lerakása – Jobb egérgomb | Jobb egérgomb megnyomásával egy blokk jelenik meg a környezet felszínén, ha a felszín elérhető távolságban van. | Közepes |

ÁLLAPOTGÉP

abztrakt -> specifikus

vízesés modell. miért jó az ide.

követelmények. legyen futtathat -> mit lehet benne játszeni

osztálydiagram

movement system

implementáláshoz mehet a movement systemből is különböző state-ek…

# implemetálás

## A Marching Cubes Algoritmus hátulütői

## Műveletigény

és

Ahol:

Feltételezve, hogy a skaláris értékekkel rendelkező kockaháló dimenziói megegyeznek (vagyis a tér kocka alakú) és a dimenziókat n-el jelölve szemléletesebb a műveletigény:

és

Ahol:

A Marching Cubes algoritmus, mivel 3 dimenziós térben dolgozik n3 műveletigénnyel rendelkezik. A konfigurációk megállapításához mind a nyolc csúcsot ki kell értékelni minden n-edik kockánál. Ez nem változik, vagyis a „térfogat nyolcszorosával” megegyező műveletet mindig végre kell hajtani. Ez 10 x 10 x 10-es nagyságú tér esetén 1000 x 8, de 100 x 100 x 100-as nagyságú tér esetén már 1.000.000 x 8, ami 8.000.000 (nyolcmillió).

A konfigurációk kiértékeléséhez minden n-edik kockánál végre kel hajtani legalább 1 nem teljes értékű műveletet vagy legfeljebb 15 teljes értékű műveletet. Az első (0.) vagy utolsó (255.) konfiguráció csak egy, a többi konfigurációhoz képest nem teljes értékű műveletet igényel, ami megállapítja, hogy a felület nem halad át a kockán. A többi konfiguráció esetében 3, 6, 9, 12 vagy 15-ször ki kell számolni a háromszögek csúcsainak pozícióit.

## Magas Vertex Szám

Az algoritmus alapból minden kocka és kocka között és kockán belül is megkülönbözteti kettő háromszögnek ugyanazon pontját. Ez nem minden alkalmazási területen probléma, sőt néhol még preferált is. Ahol az úgynevezett flat shading szükséges, ott a vertex-ek nem lehetnek közösek. (Ez általánosságban igaz, manuálisan meg lehet kerülni a problémát)

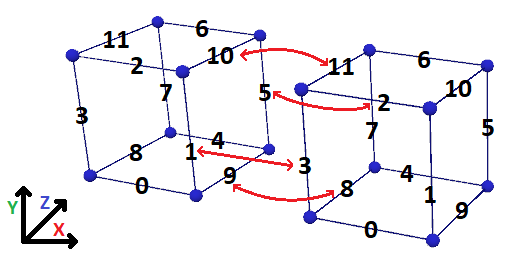
Videójátékok esetében viszont, ahol a terep dinamikus változtatása a cél a magas vertex szám és az ahhoz tartozó magas műveletigény megnehezíti a frissítési ráta tartását. Minden változtatáskor újra kell számolni az egész mesh-t, ami időbe telik. Emellett magas vertex számú mesh megjelenítése magasabb memóriahasználattal jár.

## Az algoritmus optimalizálása

Az algoritmus optimalizálatlan működése abból fakad, hogy a kockákat és a kockák konfigurációin belüli háromszögeket is külön kezeli, annak ellenére, hogy egy összefüggő hálót alkot belőlük.

## Kocka és kocka közötti optimalizálás

Ennek megoldása egy olyan adatszerkezet bevezetése, amely számontartja mely értékeket számolta már ki az algoritmus egy előző iterációban és hogy az értékek hol találhatóak meg. Ezek az értékek a kiszámolt vertex-ek (3 dimenziós vektorok). Minden vertex-el van tárolva a vertices listában, ezeknek az indexeit pedig be kell illeszteni a triangles listába. Ahhoz, hogy az algoritmus össze tudja kötni a kiszámolandó vertex-et az előzőleg kiszámított párjával vagy párjaival, egy 4 dimenziós tömböt használ. Az első három dimenzió az adott kocka térbeli pozícióját határozza meg, a negyedik dimenzió a kocka 12 élét tartalmazza.



int[,,,] vertexIndexArray = new int[X, Y, Z, 12];

A tömbben az algoritmus eltárolja azt az indexet, ahol az adott vertex található a vertices listában. Később, ha újra szükség van egy vertex-re, akkor azt nem kell kiszámolni, hanem a tömbben visszanézve indexe segítségével referálni lehet. A tömb -1 -el van inicializálva, ha valahol -1 szerepel, az azt jelenti, hogy nincs rendelkezésre álló index, vagyis nincs kiszámított vertex amire referálni lehetne. Például:

int index = vertexIndexArray[x - 1, y, z, 1];

if (index != -1)

{

vertexIndexArray[x, y, z, 3] = index;

triangles.Add(index);

}  
else ...

A fenti példában az adott kocka 3-as számú élén lévő vertex-et akarjuk megkeresni. Ehhez vissza kell lépni egyet az X tengelyen, majd ellenőrizni, hogy létezik-e már korábban kiszámolt vertex a jelenlegi 3. számú éllel megegyező, az előző kockán 1. számú élhez. Ha igen, vagyis, ha az index nem egyenlő -1 -el, akkor azt az indexet tovább lehet hozni az adott kocka 3. élére és be lehet illeszteni a triangles listába, különben ki kéne számolni.

Azonban ahhoz, hogy ezt a folyamatot automatizálni lehessen szükség van egy pár feltételre. A fenti példa feltételezi, hogy létezik az X tengelyen előző kocka és tudja, hogy a 3. élnek a párja az 1. él. Algoritmikus módon ezeket a feltételeket mindig ki kell értékelni és irányítani kell az algoritmust, hogy a megfelelő éleket válassza egy listából, ahol az összes él közötti kapcsolat fel van tüntetve.

Az élek közötti kapcsolatot egy 3 dimenziós vektorokból álló tömb tartja számon:

public static readonly Vector3Int[] redirect = new Vector3Int[]

{

// X Y Z

new Vector3Int(-1, 2, 4), // 0

new Vector3Int(-1, -1, 5), // 1

new Vector3Int(-1, -1, 6), // 2

new Vector3Int( 1, -1, 7), // 3

new Vector3Int(-1, 6, -1), // 4

new Vector3Int(-1, -1, -1), // 5

new Vector3Int(-1, -1, -1), // 6

new Vector3Int( 5, -1, -1), // 7

new Vector3Int( 9, 11, -1), // 8

new Vector3Int(-1, 10, -1), // 9

new Vector3Int(-1, -1, -1), // 10

new Vector3Int(10, -1, -1) // 11

}

A tömb indexei jelentik az adott élnek a sorszámát, aminek a párját keressük az előző kockában. A vektorok X, Y és Z elemei jelentik, hogy az adott tengelyen van-e párja az élnek és ha igen, mi annak a sorszáma az előző kockában. Például, a 0. élnek kettő párja is van, az Y és Z tengelyen. Az X komponens értéke -1, vagyis a 0. élhez nem tartozik pár az X tengelyen. Ha tudjuk, hogy az Y tengelyen van előző kocka, vissza lehet nézni annak a kockának a 2. élét és megvizsgálni, hogy van-e ott érvényes index eltárolva. Ha nincs, vagy ha nincs az Y tengelyen előző kocka, akkor még tovább lehet lépni a Z tengelyre, ha ott létezik előző kocka. A Z tengelyen a 0. él párja a 4. él az előző kockából. Ha ott van eltárolva érvényes index akkor azt fel lehet használni.

A tömb adataiból látszik, hogy a 5., 6. és 10. élen nincs lehetőség előző értékeket visszanézni, mert nincsenek előző értékek. Ez abból fakad, hogy az algoritmus nullától felfele, mind a három tengelyen pozitív irányba járja be a teret. Ezek az élek mindig elől vannak, pár nélkül, mert a következő kocka még nincs kiszámolva. Ezen a három élen mindig ki kell számolni a vertex-eket.

Ez elől lévő élekből következik, hogy vannak hátul lévő élek is, amelyeket hátra kell hagyni, hiszen a rajtuk kiszámolt vertex-ek már nem kapcsolódnak újonnan kiszámolt vertex-ekhez. Ezek az élek a 0., 3. és 8. élek. Ezekre az élekre nem hivatkozik a tömb sehol, hiszen ezek minden lépésben hátra vannak hagyva.

Feltételek a megfelelő működéshez:

1. Létezzen az adott tengelyen előző kocka, amire hivatkozhatunk.
   * A ciklus nem az első iterációban van
2. Létezzen olyan él az előző kockában, amely az adott éllel megegyezik.
   * A redirect tömb adja meg
3. Létezzen a hivatkozott élen index, amely már kiszámolt értékre mutat.
   * Az algoritmus felépítéséből adódóan mindig igaz \*

if (edgeIndex == -1) break;

...

... if (x > 0 /\* <- 1. \*/ && Tables.redirect[edgeIndex].x != -1 /\* <- 2. \*/)

{

int index = vertexIndexArray[x - 1, y, z, Tables.redirect[edgeIndex].x];

vertexIndexArray[x, y, z, edgeIndex] = index;

triangles.Add(index);

}

else if ...

A fenti kódrészlet az X tengelyre vonatkozó rész. Nem tartalmazza az Y és Z tengelyeket, hiszen a kód majdnem ugyan az, csak más tengelyeken vizsgál feltételt és más tengelyen lép vissza.

\*Megjegyzés:

A 3. feltétel mindig igaz, ha túljut az algoritmus az if (edgeIndex == -1) break; kifejezésen. A lehetséges szomszédos konfigurációk és a háló folytonossága miatt az előző kockában a kapcsolódó élen (ha van ilyen) biztos van egy már kiszámított érték. Ha nem lenne, akkor a szomszédos kockák konfigurációit lehetetlen lenne összeilleszteni annak ellenére, hogy négy sarkuk, vagyis négy skaláris értékük megegyezik és ugyan arra a négy bináris értékre jutott kiértékeléskor az algoritmus a sarkokat illetően.

## Kockán belüli optimalizálás

Kockán belül is van olyan eset, akár többször is, amikor egy élen kettő, három, de lehet, hogy öt háromszögcsúcs helyezkedik el, természetesen ugyan abban a pozícióban. Ezeket újra és újra kiszámolni szintén felesleges. Ennek a problémának a megoldása az előzőekhez képest nagyon egyszerű. Mindössze annyit kell tenni, hogy az adott pozícióban leellenőrizzük, hogy nincs-e már eltárolt index, amit felhasználhatunk.

if (vertexIndexArray[x, y, z, edgeIndex] != -1)

{

triangles.Add(vertexIndexArray[x, y, z, edgeIndex]);

} ...

## Teljes kód

// Tripla egymásba ágyazott for ciklus a tér bejárásához

for (int x = 0; x < cubesX; x++) // X tengely

for (int y = 0; y < cubesY; y++) // Y tengely

for (int z = 0; z < cubesZ; z++) // Z tengely

{

Vector3Int position = new(x, y, z);

// Kocka nyolc sarkán lévő skaláris értékek kiolvasása

float[] cube = new float[8];

for (int i = 0; i < 8; i++)

cube[i] = SampleTerrain(position + Tables.CornerTable[i]);

// Skaláris értékek alapján a konfiguráció meghatározása

int configIndex = GetCubeCongif(cube);

// Első és utolsó konfiguráció üres

if (configIndex == 0 || configIndex == 255) continue;

// Ciklus a konfigurációban található élekhez (max 15 darab vertex)

for (int vertexCounter = 0; vertexCounter < 15; vertexCounter++)

{

// Él kiolvasása a konfiguráció alapján

int edgeIndex = Tables.TriangleTable[configIndex, vertexCounter];

// Ha nincs több él amin vertex van befejeződik a ciklus

if (edgeIndex == -1) break;

// Ha az adott élre már számoltunk ki vertex-et

int redirect, index;

if ((index = vertexIndexArray[x, y, z, edgeIndex]) != -1)

{

// Index beillesztése

triangles.Add(index);

}

// Ha az X tengelyen van előző kocka ÉS van az adott élnek párja

else if (x > 0 && (redirect = Tables.redirect[edgeIndex].x) != -1)

{

// Index kiolvasása (X tengely)

index = vertexIndexArray[x - 1, y, z, redirect];

// Index bemásolása az adott pozícióba

vertexIndexArray[x, y, z, edgeIndex] = index;

// Index beillesztése az adott háromszög egyik csúcsának

triangles.Add(index);

}

// Ha az Y tengelyen van előző kocka ÉS van az adott élnek párja

else if (y > 0 && (redirect = Tables.redirect[edgeIndex].y) != -1)

{

index = vertexIndexArray[x, y - 1, z, redirect];

vertexIndexArray[x, y, z, edgeIndex] = index;

triangles.Add(index);

}

// Ha az Z tengelyen van előző kocka ÉS van az adott élnek párja

else if (z > 0 && (redirect = Tables.redirect[edgeIndex].z) != -1)

{

index = vertexIndexArray[x, y, z - 1, redirect];

vertexIndexArray[x, y, z, edgeIndex] = index;

triangles.Add(index);

}

else // Ha nem volt előzőleg kiszámolt érték

{

// A metódus kiszámolja majd beilleszti a vertex-et és  
 // visszaadja az indexét

vertexIndexArray[x, y, z, edgeIndex] =  
 CalculateVertex(position, edgeIndex, cube);

}

}

}

## Compute Shader Használata

A Compute Shader egy specializált Shader, amelyet arra terveztek, hogy általános célú számításokat végezzen a videókártyán. A videókártya általában grafikai műveletek számítását végzi, amelyekből közvetlenül megjelenített kép lesz. Általános célú számítás alatt bármilyen olyan műveletet lehet érteni, amelynek a kimenete nem megjeleníthető kép, vagy amelynek kimenete nem a képernyőre van irányítva, hanem még visszatér egy ciklusban a CPU oldalra.

A Compute Shader-eknek több kulcsfontosságú funkciója van, amelyet CPU oldalon nem lehet megvalósítani, vagy nem olyan effektív.

## Párhuzamos futtatás

Egy compute shader sok, egymással párhuzamosan futó szálon tud végrehajtást végezni egyszerre kihasználva a GPU-ban jelen lévő sok magot, amelyekből lényegesen több áll rendelkezésre, mint egy CPU-ban.

Ez lehetővé teszi, hogy minden egyes kocka egy számára dedikált magon fusson le.

## Direkt memória hozzáférés

Egy compute shader hozzáféréssel rendelkezik a dedikált videómemóriához (VRAM). Ez elengedhetetlen a feldolgozási sebesség növeléséhez. A videómemóriába adatokat lehet bemásolni a CPU oldalról ComputeBuffer-ek segítségével.

## Azonalli végrehajtás

A hagyományos shader-ekkel ellentétben, amelyek a megjelenítés bizonyos fázisaiban futnak le, a compute shader manuálisan van indítva script oldalról. Ez lehetővé teszi a fejlesztők számára, hogy az alkalmazás bármelyik fázisában műveletek végezzenek a videókártya segítségével.

## Munkacsoportok

A compute shader munkacsoportokon alapszik. Minden munkacsoport szálak kollekciója, amelyek az adatok egy adott részén dolgoznak. A compute shader munkacsoportjainak száma és a munkacsoportokban lévő szálak száma is a fejlesztőre van bízva.

A compute shader munkacsportjainak száma és a munkacsoportok szálainak száma is 3 dimenzión van elosztva.

## Dispatch

A Dispath hívással lehet végrehajtani egy compute shadert, ahol a 3 bemenő adat a munkacsoportok számát határozza meg a 3 dimenzió mentén.

Dispatch(int kernelIndex, int groupsX, int groupsY, int groupsZ);

## Kernel

Minden futtatható compute shader tartalmaz legalább egy úgynevezett Kernel-t. A kernel a compute shader belépési pontja. Több kernel is lehet egy compute shader-ben. A kernek fölötti numtreads mező határozza meg, hogy egy munkacsoporton belül hány szál van jelen a 3 dimenzión.

[numthreads(4, 4, 4)]

void GetTriangles (int3 id : SV\_DispatchThreadID) {

## Chunk loading system

A játék világa úgynevezett Chunk-okra van felosztva. A chunk a videójátékok körében, azon belül is az open world és/vagy sandbox játékok esetében gyakran emlegetett kifejezés. Jelentése egy független terület a játék világán belül, legyen az 2 vagy 3 dimenziós. A hasonló játékok világa sok ilyen, kiterjedésükben egyforma chunk-ból áll. Ez részben elengedhetetlen, részben pedig sok előnnyel is jár.

A világ chunk-okra való felosztása mindenképpen szükséges, a legtöbb ilyen játék világa nem férne bele a memóriába egyszerre és erre nem is lenne szükség. A látótávolság nagyon sok esetben mesterségesen limitált. Ezt a limitet állítani is lehet, amely a különböző számítási kapacitással rendelkező számítógépeknek is esélyt ad a játék megfelelő frissítési rátával való futtatására.

Természetesen a realizmust, ha nem is végtelen, de nagyon nagy látótávolsággal lehetne megközelíteni. Ez azonban szükségtelen, hiszen akkora távolságban semmi érdemleges információt nem szolgáltatnának a megjelenített elemek, hiszen nem lennének kivehetőek.

A chunk-ok dimamikus betöltése és kezelése azonban nem egy egyszerű logikai folyamat. Bár leggyakrabban csak egy faktort kell figyelembe venni, a játékos pozíciójához viszonyított látótávolságot, megállapítani, hogy mely chunk-ok vannak látótávolságon belül és kívül egyedi problémákat eredményez, amelyekre többfajta megoldás létezik.

Minden lejjebb említett megoldás tartalmaz minimális és maximális látótávolságot. A minimális látótávolságon belül mindenhol léteznek chunk-ok, a maximális látótávolságon kívül pedig sehol sem. Erre azért van szükség, hogy lehetőséget adjunk a játékosnak arra, hogy egy kis területen mozogjon anélkül, hogy a játékot új chunk-ok betöltésére kényszerítse. Ezt úgy éri el, hogy a minimális látótávolságon kívül eső chunk-okat nem rögtön törli, vagyis, ha a játékos visszafele halad kis távolságon, akkor még be vannak töltve a chunk-ok, amiket látnia kell.

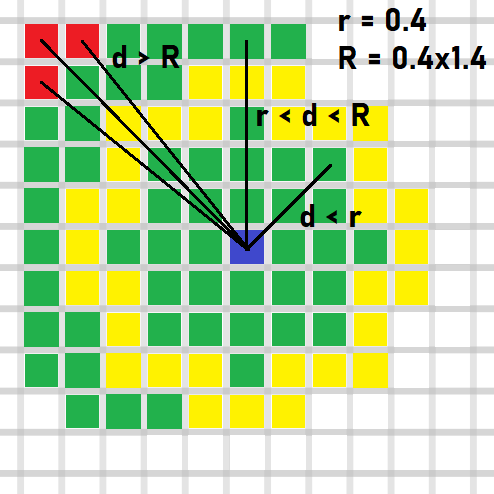
## Távolság alapú ellenőrzés és pozíció alapú létrehozás

Ebben az esetben a chunk-ok egy Dictionary-ben vannak eltárolva objektum-kulcs páronként. Az objektum maga a chunk, a kulcs pedig az chunk pozíciója.

Ez a módszer a játékos pozíciójához képest számítja ki, hogy melyek azok a koordináták, ahol egy chunk-nak jelen kell lennie, hogy a látótávolságon belül minden látható legyen. A matematikai műveletek elvégzésén egy előre meghatározott koordináta halmaz segít. Ez a koordináta halmaz a (0,0,0) koordinátához képest megjelöli azokat a koordinátákat, ahol egy chunk-nak már mindenképpen jelen kell lennie. Ezeket a koordinátákat csak hozzá kell adni annak a chunk-nak a koordinátájához, amiben a játékos tartózkodik.

A koordináta halmaz 2 dimenziós ábrázolásban egy gyűrű. A gyűrűn belüli chunk-okat nem kell létrehozni vagy ellenőrizni, hogy léteznek-e, hiszen ahogy a játékos „viszi magával a gyűrűt” a gyűrűn nem jut át olyan koordináta, amely nem lett volna ellenőrizve legalább egyszer.

Ez a módszer a távoli chunk-ok megszüntetéséhez vektor alapú távolság ellenőrzést alkalmaz. A játékos pozíciójához képest leméri, hogy milyen messze vannak a chunk-ok és ha a látótávolságon kívül esik egy chunk, azt megszünteti.



Jelmagyarázat: Kék – Játékos, Zöld – Létező chunk, Sárga – Létrehozáshoz ellenőrizendő pozíció, Piros – Törlendő chunk, Fehér – Üres pozíció, r – Látótávolság, R – Törlési távolság

Előnyök:

* Viszonylag egyszerű implementálni. Egy ciklus ellenőrzi az összes létező chunk-ot hiányzó koordinátát keresve, egy cilus pedig ellenőrzi a létező chunk-ok játékostól való távolságát.
* Egy dimenziós adatszerkezetet használ, amely egy dictionary. A dictionary keresésre jól optimalizált.

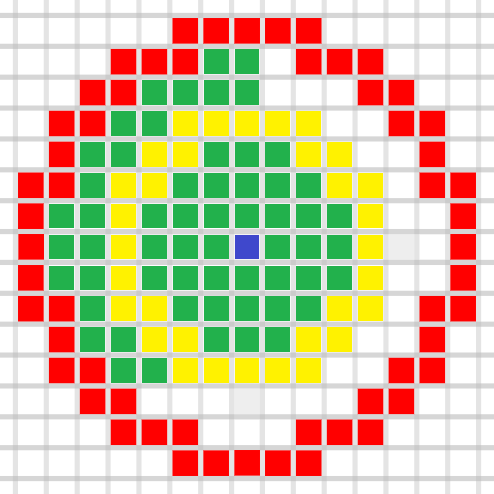
Hátrányok:

* Minden létező chunk – létrehozni kívánt chunk kombinációt meg kell vizsgálni, hogy elkerüljük a duplikált chunk-okat. Ez legoptimalizáltalanabb esetben dupla for ciklust jelent, kis optimalizálással valamilyen keresőalgoritmust, amelyhez viszont rendezetten kellene tartani az adatszerkezetet, ami az adatok dinamikussága miatt nehezen kivitelezhető.
* Keresés komplexitása: O.
* Minden már létrehozott chunk esetében ki kell számítani a távolságot a játékoshoz képest majd össze kell hasonlítani a látótávolsággal. Ez vektorok távolságának kiszámítását jelenti, amely nagy mennyiségben lelassítja a játék futását.
* Távolság számítások: .

## Pozíció alapú ellenőrzés és létrehozás

Ez a módszer is objektum – kulcs páronként tárolja a chunk-okat, ahol az objektum a chunk, a kulcs pedig a pozíciója.

Ebben az esetben mind a létrehozás és a megszüntetés is egy előre meghatározott koordináta halmaz alapján történik. A külső halmaz koordinátáit hozzáadva a játékos chunk-jának koordinátájához kiszámolható, hogy mely chunk-okat kell megszüntetni, a belső halmaz segítségével pedig kiszámolható, hogy mely chunk-okat kell létrehozni. Ezek után a már létező chunk-ok között keresni kell az adott koordinátára és a keresés eredménye alapján vagy létrehozni, ha még nem létezik, vagy megszüntetni, ha már létezik.



Jelmagyarázat: Kék – Játékos, Zöld – Létező chunk, Sárga – Létrehozáshoz ellenőrizendő pozíció, Piros – Törléshez ellenőrizendő pozíció, Fehér – Üres pozíció

Ez a megoldás kettő O keresést hajt végre, amely időkomplexitásának alsó határa esetenként jobb lehet.

## Memória struktúra alapú rendszer

A memória struktúra alapú rendszer nagyon hasonló a pozíció alapú ellenőrzés és létrehozás módszeréhez, viszont itt a chunk-ok nem egy rendezetlen adatszerkezetben vannak eltárolva és nem is objektum-kulcs páronként vannak jelen.

Az adatszerkezet ebben az esetben egy négyzetes tömb. A tömbben chunk objektumok mutatói szerepelnek. A négyzetes tömb dimenziói páratlanok és a tömb közepén mindig annak a chunk-nak a mutatója szerepel, amelyben a játékos tartózkodik. A négyzetes tömb mindig akkora területet foglal el a memóriában, hogy be tudja fogadni az összes elméletileg létezhető chunk-nak a mutatóját az adott látótávolság mellett. A többi chunk objektum mutatója a játékvilágban a játékos chunk-jának pozíciójához viszonyított, egész chunk-okban mért távolság alapján foglal helyet a tömb közepéhez képest.

Ennek a rendszernek a funkcionális követelménye, hogy minden egyes alkalommal, amikor a játékos átlép egyik chunk-ból a másikba, akkor a haladási irányával ellentétesen el kell mozgatni a tömbben az összes mutatót egyel.

Ez a tömb méretével megegyező olvasási és írási műveletet jelent. A tömb külső határán mindig NULL értékek szerepelnek ezért mindig van hely egyel elmozgatni a mutatókat. Ezt követően minden egyes mozgatás után le kell ellenőrizni az előre meghatározott memóriacímeket a külső és belső gyűrű mentén:

* Ha egy olyan memóriacímbe másolódott be egy mutató, amely törlendő chunk pozícióját tartja fent, akkor a mutató végén lévő chunk-o törölni kell. Ez egyszerű, *ha NEM NULL akkor* feltétel.
* Ha egy olyan memóriacímen nincs érvényes mutató, ahol egy chunk-nak léteznie kelljen, hogy a minimális látótávolságot megtartsuk, akkor az adott pozíció alapján létre kell hozni egy chunk-ot és muttatóját be kell másolni arra a pozícióra. ez egy egyszerű, *ha NULL akkor* feltétel.

Műveletigény:

* Mozgatás: olvasás és írás ()
* Ellenőrzés: létrehozáshoz és törléshez, ahol az olyan címek száma, ahol chunk-nak léteznie kell és az olyan címek száma, ahol a chunk-okat törölni kell.
* Mivel ezért eddig ennek a legkisebb a komplexitása és nem is használ bonyolult számításokat és kereséseket.

Az alábbi kódrészlet másolja ás a memóriacímeket a tömbben egyel arrébb a tengelyek mentén. Csak a kettő vízszintes tengely mentén vannak az elemek mozgatva, mivel a chunk-okat a világ tetejétől az aljáig egy adott oszlopban mindig betölti a játék.

// Check if the player moved chunks

playerChunk = Vector2FloorToNearestMultipleOf(ToVector2FromXZ(playerTransform.position), CubicChunk.cubesPerAxis);

if (playerChunk == oldPos) return;

if (playerChunk.x > oldPos.x)

for (int x = 0; x < outerDiameter - 1; x++)

for (int z = 0; z < outerDiameter; z++)

for (int y = 0; y < verticalChunks; y++)

renderedChunks[x, y, z] = renderedChunks[x + 1, y, z];

else if (playerChunk.x < oldPos.x)

for (int x = outerDiameter - 2; x >= 0; x--)

for (int z = 0; z < outerDiameter; z++)

for (int y = 0; y < verticalChunks; y++)

renderedChunks[x + 1, y, z] = renderedChunks[x, y, z];

if (playerChunk.y > oldPos.y)

for (int x = 0; x < outerDiameter; x++)

for (int z = 0; z < outerDiameter - 1; z++)

for (int y = 0; y < verticalChunks; y++)

renderedChunks[x, y, z] = renderedChunks[x, y, z + 1];

else if (playerChunk.y < oldPos.y)

for (int x = 0; x < outerDiameter; x++)

for (int z = outerDiameter - 2; z >= 0; z--)

for (int y = 0; y < verticalChunks; y++)

renderedChunks[x, y, z + 1] = renderedChunks[x, y, z];

oldPos = playerChunk;

Az alábbi kód a koordináták létrehozásáért felel egy kör mentén, amelyeket a minimális és maximális látótávolság meghatározásához használ.

private List<Vector2Int> CalculateRing(int radius, Vector2Int center, int gridSize)

{

List<Vector2Int> ring = new();

for (int x = 0; x < gridSize; x++)

for (int y = 0; y < gridSize; y++)

{

Vector2Int pos = new(x, y);

float dist = Vector2Int.Distance(center, pos);

if (dist < radius + 1 && dist >= radius - 1)

ring.Add(pos);

}

return ring;

}

Ezek után a kód csak le ellenőrzi a megadott koordinátákat.

foreach (Vector2Int offset in innerRing)

{

if (renderedChunks[offset.x, 0, offset.y] == null)

{

Vector2Int horizontalPos = playerChunk + (offset - center2) \* CubicChunk.cubesPerAxis;

for (int y = 0; y < verticalChunks; y++)

{

CubicChunk c = GetFromPool();

renderedChunks[offset.x, y, offset.y] = c;

Vector3Int pos = new(horizontalPos.x, y \* CubicChunk.cubesPerAxis, horizontalPos.y);

c.Move(pos);

chunksToBuild.Add(c);

}

}

}

foreach (Vector2Int offset in outerRing)

{

if (renderedChunks[offset.x, 0, offset.y] != null)

{

for (int y = 0; y < verticalChunks; y++)

{

ReturnToPool(renderedChunks[offset.x, y, offset.y]);

renderedChunks[offset.x, y, offset.y] = null;

}

}

}

## Chunk Pooling System

Pooling system sok játékban jelen van, ahol azonos vagy hasonló dinamikus objektumokból sokat kell létrehozni a program futása során előre nem meghatározott időpontokban. Egy objektum létrehozása idő és erőforrás igényes folyamat. Le kell foglalni a memóriaterületet és azt fel kell tölteni adatokkal. Ez sokszor a frissítési ráta lecsökkenéséhez vezethet egy intenzív szituációban.

A pooling system alapja, hogy a létrehozott objektumokat újrahasznosítja. Ezt úgy éri el, hogy egy pool-ban tárolja a már létrehozott objektumokat és ebből a pool-ból kér le példányokat a játék amikor szükség van rájuk. Amikor egy objektum már nem szükséges, akkor megsemmisítés helyet visszaküldi a pool-ba, hogy később ismét fel lehessen használni.

Egy Object Pool-nak három alapfunkciója van.

Tárolás:

A pool általában egy dinamikus méretű adatszerkezet. C#-ban egy Lista alkalmas erre. Egy listában könnyen el lehet érni az elemeket és könnyen ki is lehet törölni azokat. Hozzáadni a listához is könnyű.

private List<CubicChunk> pool = new();

private int inPool = 0;

private int inWorld = 0;

Bár a lista osztálynak van elérhető length tulajdonsága, amely visszaadja a listában szereplő elemek számát, ennek a használata nem a leg effektívebb módszer. Ha lokális változókkal is könnyen számon lehet tartani az elemek számát, akkor az a módszer minimálisan, de hatékonyabb.

Lekérés:

Amikor a játéknak szüksége van egy objektumra a megadott típusból, akkor meghív egy metódust, amely visszaad egy megfelelő példányt. A metódus le ellenőrzi, hogy a pool-ban van-e elég példány és ha igen kiválasztja az elsőt. Ha nincs akkor kénytelen létrehozni egyet. A legtöbb pool lekezeli az esetet amikor üres állapotban kérnek le belőle példányt. Létezik olyan pool is, amely ezt nem kezeli le, de abban az esetben az egy időpillanatban aktív példányok számát meg kell kötni.

private CubicChunk GetFromPool()

{

CubicChunk c;

if (inPool > 0)

{

inPool--;

c = pool[0];

pool.RemoveAt(0);

c.SetActive(true);

}

else

c = new(worldTransform);

inWorld++;

return c;

}

Visszahelyezés:

Amikor egy példányra már nincs szükség, akkor az vissza lesz helyezve a pool-ba. Ez teszi lehetővé, hogy a későbbiekben ismét le lehessen kérni.

private void ReturnToPool(CubicChunk c)

{

inPool++;

if (inWorld > 0)

inWorld--;

c.SetActive(false);

pool.Add(c);

}

A pool lehet előre feltöltött is, ami azt jelenti, hogy egy meghatározott számú objektum alapból jelen van a pool-ban későbbi felhasználásra, vagy indulhat a pool 0 elemmel is. Ha kezdetben nulla eleme van a pool-nak, akkor amíg objektum nem kerül visszaküldésre a pool-ba addig minden alkalommal adott időpillanatban kell létrehozni az objektumot, de a későbbiekben újra lehet fel lehet használni azokat. A pool indulhat nem nulla elemszámmal is. Ekkor az első lekérésre is már létező objektumot fog visszaadni a pool addig amíg ki nem fogy.

# Tervezés

## Az optimalizáció eredménye

Az alábbi értékek mind átlagos esetekből származnak, amikor az alakzat felszíne se nem a 0, se nem az elméleti maximum felé nem konvergál erőteljesen.

A fenti grafikonon látható, hogy a tér méretének növekedésével drasztikusan nő a vertex-ek száma optimalizálás nélkül. Olyan mértékben, hogy 64x64x64-es nagyságú térnél már több mint 45 ezer vertex van jelen. Ehhez képest, megosztott vertex-eknél mindössze csak 7-8 ezerrel kell számolni. Ez azért is fontos, mert nagyon könnyű átlépni az alapértelmezett 16 bites buffer maximumát, ami 65535.

Olyan eset elméleti szintem sem szabad, hogy fennálljon, amikor a meghatározott területet reprezentáló mesh túllépi a maximum vertex számot. Igaz, a maximum vertex szám mesterségesen is ritka, hogy előforduljon.

A vertex szám egyenesen összefügg a mesh memóriafelhasználásával. A 200, de még az 1200 KiB sem sok, de a tér nem csak egy egységből áll, így a gyorsan összeadódnak a MiB-ok és lehet, hogy gyorsan elérjük a 100 MiB-ot is.

## Pszeuorandom Generálás

A talaj megjelenítése után a következő lépés a talajnak a megalkotása. Erre sok különböző pszeudorandom algoritmus áll rendelkezésre. A választás a Perlin Noise algoritmusra esett egy nagyon egyszerű indok miatt. A Perlin Noise a Unity Mathf könyvtárának egy beépített funkciója és tökéletesen alkalmas talajmagasság meghatározására.

# Hivatkozások

Cline, H. E., & Lorensen, W. E. (1987). *United States of America Szabadalom száma: US4710876A.*

Grand View Research. (dátum nélk.). *Video Game Market*. Forrás: Grand View Research: https://www.grandviewresearch.com/industry-analysis/video-game-market#:~:text=b.-,The%20global%20video%20game%20market%20size%20was%20estimated%20at%20USD,USD%20583.69%20billion%20by%202030

Lorensen, W. E., & Cline, H. E. (1987). Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *SIGGRAPH*, (old.: 163-169). Anaheim.

National Institute of Biomedical Imaging and Bioengineering. (2022. Június). *Computed Tomography (CT)*. Forrás: Nibih: https://www.nibib.nih.gov/science-education/science-topics/computed-tomography-ct