SZAKDOLGOZAT

Almási Dávid

Debrecen

2011

Debreceni Egyetem

Informatika Kar

Gombavadászat

Témavezető:
Dr. Kósa Márk
egyetemi tanársegéd

Készítette:
Almási Dávid
Programtervező informatikus

Debrecen
2011

Bevezetés

Közel három éve támadt az az ötletem egy strand termál medencéjében üldögélve, hogy talán össze tudnék hozni egy kisebb játékot teljesen az alapoktól kiindulva, mellőzve mindenféle előre gyártott grafikus motort és hasonló programkomponenseket. Ezt azért tartom fontosnak, mert így bár lassabb a program fejlesztése, ugyanakkor lehetőség van mindent a legapróbb igények szerint a legoptimálisabban megírni, és közben sok olyan dolog működését és tervezését ismerheti meg az ember, aminek más nagyobb fejlesztőeszközökben már csak a jótékony hatását élvezheti. Ezért több a megjelenítést szolgáló elem nem csupán a megfelelő (keretrendszer által biztosított) metódushívásokból áll, hanem egy C-hez hasonló szintaktikájú kódban, HLSL-ben (High Level Shading Language) van megírva.

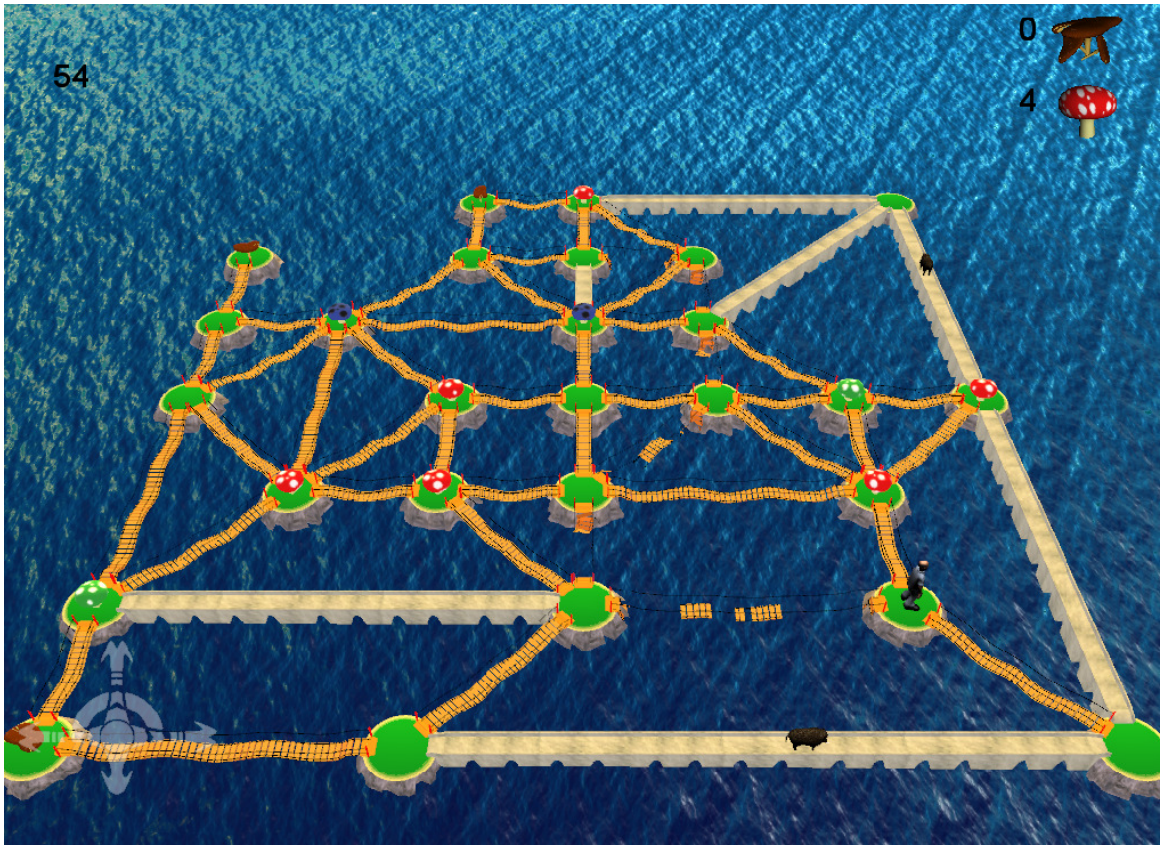
Sokat törtem a fejem azon, hogy milyen stílusban lenne érdemes a játékot elkészíteni. Végül egy a táblás és az ügyességi játékok ötvözetére esett a választásom, leginkább azért, mert ebből a kategóriából nem lehet annyit találni a piacon, mint például az FPS (First-Person Shooter) stílusú alkotásokból. Később, amikor konkretizáltam az igényeimet, kisült néhány alapvető követelmény, ami nagyvonalakban a fejlesztés útvonalát is meghatározta. Fontosnak tartottam, hogy három dimenzióban lehessen vele játszani, valamint hogy több apró játékmenetre lehessen feldarabolni az egészet. Ez alatt azt értem, hogy miután valakinek sikerült kiválasztani a neki megfelelő nehézségi szintet, ne kelljen órákat a monitor előtt görnyednie, hanem 1-1 menetet percek alatt végigjátszhasson.

Az is megfogalmazódott bennem, hogy a klasszikus táblás játékok többnyire unalmassá válhatnak akkor, ha az egyik játékos sokat gondolkozik a következő lépésén. Szerettem volna ezt a problémát olyan módon kiküszöbölni, ami inkább izgalmassá tesz egy játszmát, mintsem frusztrálttá a játékost. Így tehát a ketyegő és esetleg hangoskodó stopperóra ötletét gyorsan elvetettem, helyette úgy döntöttem, hogy kötetlenebbé és realisztikusabbá válhat a játszma, ha nem szabom meg, hogy ki mikor léphet.

A játék két személyes, ám nem ember játszik ember ellen, hanem ember a gép ellen próbál benne ügyeskedni. Szerettem volna, ha a gép a lépéseit több „stratégia” szerint is megtehetné, ezért több különböző hatékonyságú mesterséges intelligencia algoritmust is igyekeztem beültetni az alkalmazásba.

Röviden a játék menetéről (szabályok)

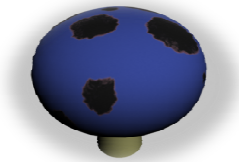
A játékban egy kissé túlsúlyos karaktert irányíthat a játékos, aki egy furcsa, elhagyatott szigetvilágra került, ahol néhány szigetre csak nehézségek árán érhet el. A szigeteken különböző gombák teremnek, ám a játékos sajnos nem gombaszakértő, ezért amit talál, azt meg kell, hogy egye. A játéknak akkor van vége, ha sikerült elfogyasztani a szigetvilágon található összes piros színű gombát.



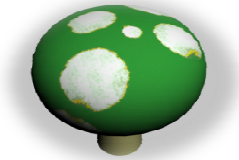
A szigetek különböző függőhidakkal vannak összekötve egymással, ezek készülhettek fából vagy kőből. Amennyiben a játékos egy kőhídon megy át, gond nélkül visszamehet a kiindulási pontra, ám ha egy fából készült függőhidat választ, akkor sietnie kell, mivel azok a hidak elég régiek, így eléggé labilisak is. Ezzel a rendszerrel sikerült elérnem, hogy a játéktér folyamatosan változzon mindkét játékos számára.

Két játékost említettem, amit ígérem, nemsokára pontosítani fogok, ám előbb leírom, a szigetcsoporton található gasztronómiai világot, és azok játékosra gyakorolt hatásait.

Piros gomba: Ízletes rágcsálni való, ebből kell az összeset megenni a játékosnak.



Kék gomba: Szintén ízletes lehet, ám rendkívül mérgező is egyben, ha ilyet sikerül a játékos gyomrába juttatni, a játéknak sajnos azonnal vége szakad.

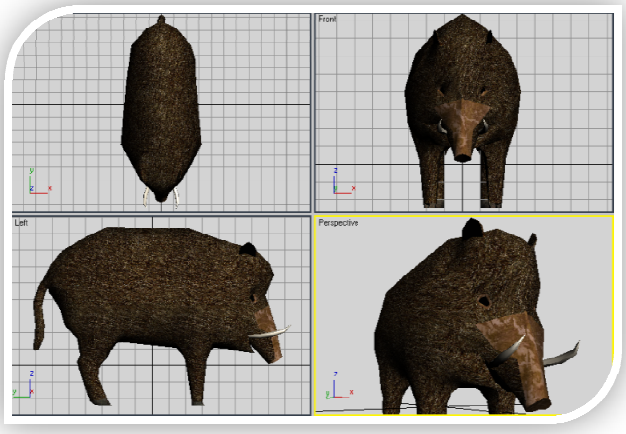


Zöld gomba: Jó hatással van a látásra, ám kevésbé az idegállapotra, hatását szerencsére csak rövid ideig fejti ki, ám azidőre a teljes játék és annak irányítása is a feje tetejére fordul.

Barna gomba: Az íze miatt sajnos teljesen ehetetlen, így bevált fogyókúra válhatna belőle, ezeket össze lehet gyűjteni vagy akár el is lehet dobni.

Az ellenség:

Azért, hogy a szigetcsoport mégse legyen teljesen elhagyatott (és hogy a mesterséges intelligencia is beszállhasson a dologba), a képen látható malacok is versenghetnek a gombák vagy a játékos



becserkészéséért. A gépnek tehát lehetősége van több ilyen karaktert is irányítani, akár egymástól függetlenül, akár kooperatívan. A történet szerint, mivel ezek a sziget régi lakói - és a vaddisznóknak amúgy is jó a szaglása - otthonosan mozognak keresztül-kasul a kiszemelt áldozat nyomában. Az elsődleges céljuk a barna színű gombák minél előbbi elpusztítása, ám ha arra már nincs lehetőség, akkor könnyen a játékos nyomába erednek. Kis termetük miatt bármelyik hídon akárhányszor átmehetnek anélkül, hogy az leomlana alattuk.

Elmélyedve a dolgozatban, látni fogjuk, hogy valójában két téma elegyedik benne, egyrészt a 3D-s grafikai megjelenítés, másrészt a mesterséges intelligencia. Mindkettőt fontosnak tartom kiemelni, ám a gyakoribb igénybevétel miatt az előbbi talán terjengősebb lesz, mivel abból a témakörből több elemet is fel kellett használnom, illetve összehangolnom a jelenleg használatos fejlesztő eszközökkel.

A program bemutatása

Áttekintés

Az alkalmazás több különböző komponensből (itt nem programkomponensre gondolok) tevődik össze, melyek biztosítják a teljes funkcionalitást. A komplexitás miatt itt röviden összefoglalom ezen komponensek egymással való kapcsolatát, majd részletesen ismertetem az egyes összetevőket.

A futtatható program három kisebb programból áll, melyek közül a legnagyobb a megjelenítést és az általános vezérlést biztosító rész. Ezt indíthatja el a felhasználó. Indítás után egy menürendszer jelenik meg, ahol paraméterezni lehet a játék nehézségét, valamint a megjelenítési beállításokat. A játék a kezdete előtt egy külső DLL-hez fordul a paraméternek megfelelő pálya generálásáért, valamint futás során egy másik DLL biztosítja a gép számára a lépésválasztó algoritmusok tárházát.
Ennek a felépítésnek a tesztelés során nagy hasznát élveztem, mivel az egyes elemek külön interfészekkel rendelkeznek, és emiatt a tesztekhez nem kellett a számítógép egyéb erőforrásait használnom, ami rendkívül felgyorsította a folyamatot. Tesztelés után egyszerű fájlmásolással lehet a DLL-ekbe befordított algoritmusokat használatba venni.
Futás során több multimédiás állományt is felhasznál a program a megjelenítéshez, ezek többnyire képek és 3D-s modellek, valamint azok animációs mintáit tartalmazó állományok. Továbbá kiegészítő állományokat is használ az alkalmazás, mint például tárolt HLSL (High Level Shading Language) kódok, amiket futási időben fordít le a keretrendszer. Valamint természetesen a beállításokat megőrző DAT fájlokat is kezelni kellett.
A játék megírása a fent említett modellek, illetve képek megalkotásával kezdődött, így ezzel kezdeném a bemutatást.

Képek, modellek és textúrák

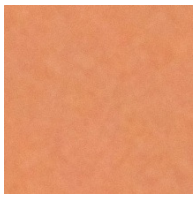
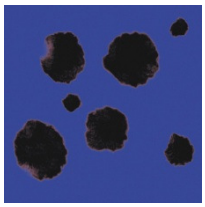
A programban felhasznált képek leginkább a játékos könnyű és gyors tájékoztatását segítik. Formátumukat tekintve tulajdonképpen majdnem mindegy, hogy JPG, BMP, TGA vagy egyéb. Azonban a lehető leggyorsabb kezelhetőség érdekében érdemes őket abban a felbontásban és színmodellben tárolni, amit a felhasználás leginkább megkíván.

Néhány ezek közül:



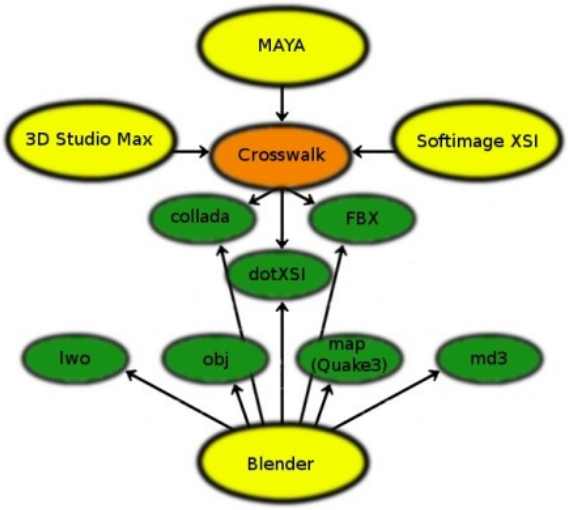
A textúrák már kényesebbek a normál képeknél, mivel ezeknél a vizuális minőség is sokat számít. A felhasználási igényeknek megfelelően kell őket elkészíteni vagy összegyűjteni. A lent látható jobb oldali textúra például egyáltalán nem alkalmas nagyobb felületen sorozatos megjelenítésre, még a bal oldali kisebb méretekben megfelelhet. Azokat a textúrákat, melyek felhasználhatók nagyobb felületek lefedésére azáltal, hogy sorozatosan egymás mellé/alá helyezzük őket tiling textúráknak is hívják.


Az ilyen jellegű textúráknál fokozottan ügyelni kell a felbontásra és ezzel összefüggésben arra, hogy a játékost milyen közel engedjük a textúrához. A sokszorosítás miatt a nagyobb felbontású textúrák tárolása és mintavételezése sok időt emészthet fel, valamint a mintavételezési hibák miatt akár zavaró (vibráló) látványt is nyújthatnak. Túl kicsi felbontás esetén pedig a közeli jeleneteknél láthatunk az alkalmazott szűrőtől függő elmosódásokat.
Néhány felhasznált textúra:

			
ez a karakter bőrfelületének élethűbbé varázsolásához nyújtott segítséget	a fentebb emlegetett kék gomba majdnem kék textúrája	a szigetek felső felületének textúrája, melyet kör alakban húztam rá a szélekre	ez a karakter ruházatát tartalmazó textúra

A modellek elkészítéséhez az Autodesk® 3ds Max® 2008 programot használtam. Ez egy elég komplex fejlesztő eszköz, mellyel akár teljes animációs filmeket is lehet gyártani.

Hasonló fejlesztések során problémát okozhat a fejlesztőeszközök kompatibilitási hiánya, ami ebben az esetben azt jelenti, hogy nem képesek egymás fájlformátumait megfelelően (bár gyakran egyáltalán nem) kezelni. Néha még az egyes verziók között is erős eltérések lehetnek. Az ábra utat mutat az átjárhatóságra az egyes fejlesztőeszközök között.



A testmodellezés terén az idők során számos technika alakult ki, mint például a CSG (Constructive Solid Geometry), Volumetric modelling Soft-Object modelling, ám a célnak leginkább megfelelő eljárás (és talán a legszélesebb körben használt) a poligonmodellezés volt.

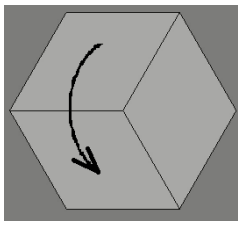
A poligonmodellezés alatt a leírandó objektumot pontjaival, éleivel, lapjaival határozzuk meg. Az élek és a közöttük értelmezett sokszögek – polygonok – többé-kevésbé tökéletesen leírhatnak minden formát. A felületek leírásához az alábbi elemek használatosak:

- Vertex: kiterjedés nélküli pont. Ez az alapvető építőelem.

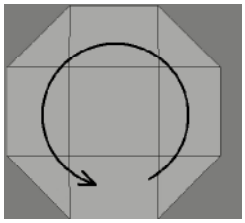
- Edge: a vertexek között húzódó él. Természetesen nem minden vertex között van edge, de – általában – nincsen olyan pont, amelyik ne lenne egy élnek valamelyik végpontja.
- Face: egy poligon – azaz sokszög – melyet élek határolnak körbe. A legtöbb program megengedi a háromnál több oldalú lapokat is, bár ezek a „sokoldalú” lapok fura szerzetek. Mivel több mint három vertex definiál egy ilyen poligont, semmi nem garantálja, hogy ezen pontok egy síkban vannak.

A programok általában a megjelenítés előtt az elfajult lapokat felosztják apróbb háromszögekre, melyek természetesen valóban síkbeliek lesznek. Mivel ez a felosztás nem egyértelmű, célszerű többségében három-, ill. négyszögekkel dolgozni.

A poligonmodellezés fő célja olyan négyszögháló (és az edge-loop-ok) kialakítása, melyeken a subdivision módszerek is megfelelően alkalmazhatóak.

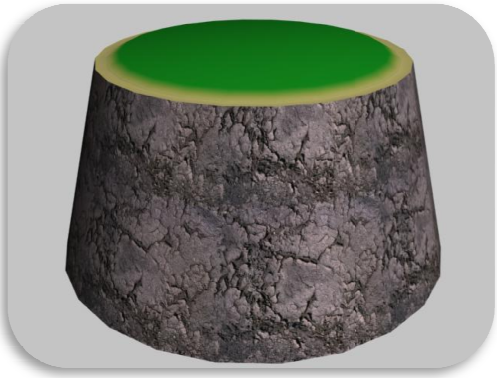


A négyszögek legalapvetőbb kapcsolódása az, ha egy pontban négy négyszög találkozik. Egy csúcspontban három négyszög találkozása megfordítja az edge-loop (face-loop) irányát.



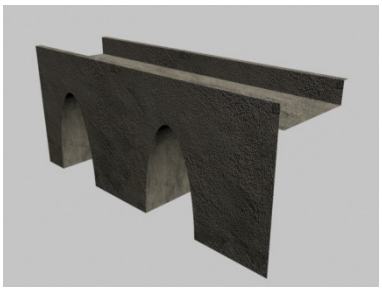
Néhány az elkészített modellek közül:

Sziget:



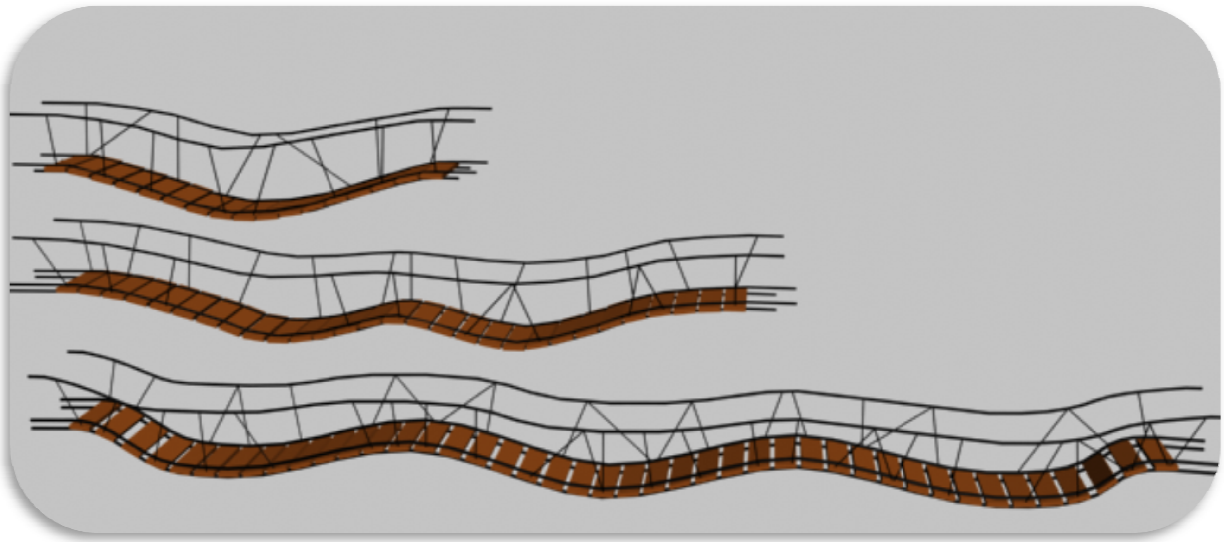
Szigetből csak egy modell készült, mivel így csak egyszer kell betölteni a memóriába. Betöltés után, egy függvény a peremen lévő csúcspontokat véletlenszerű transzlációknak veti alá, és az eredmény egy tömbben tárolódik. Miután minden szigethez hozzákapcsolódik egy ilyen transzlációs tömb, biztosított, hogy egyrészt ne legyenek ennyire szabályos alakúak, másrészt mindegyik egy „kicsit” másképpen néz ki a randomizáció miatt.

Kőhíd:

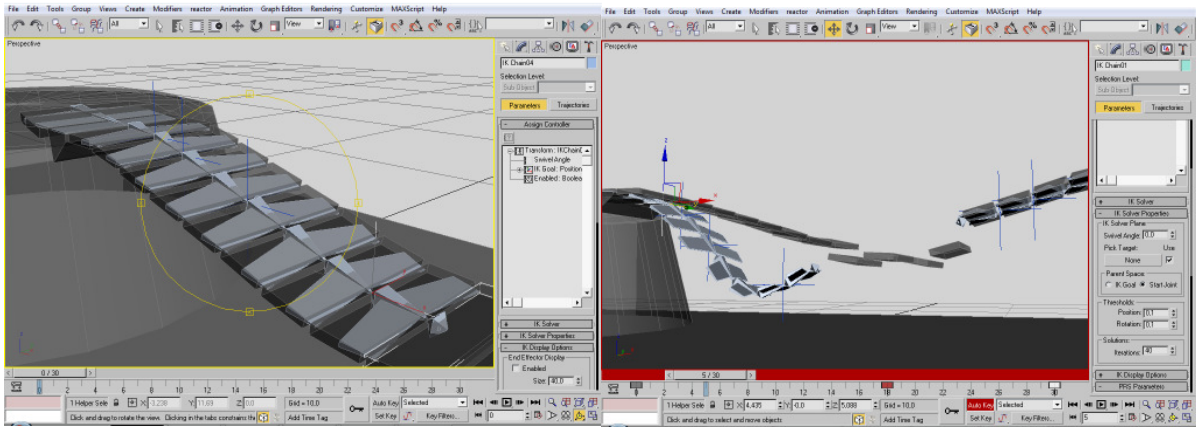




Mint látható, több részhídról van szó. Attól függően, hogy milyen irányba, és milyen hosszú hidat kell építeni, a program egy hídelemeket tartalmazó készletből választja ki az összerakáshoz szükséges elemeket, és azokkal végzi a kirajzolást.

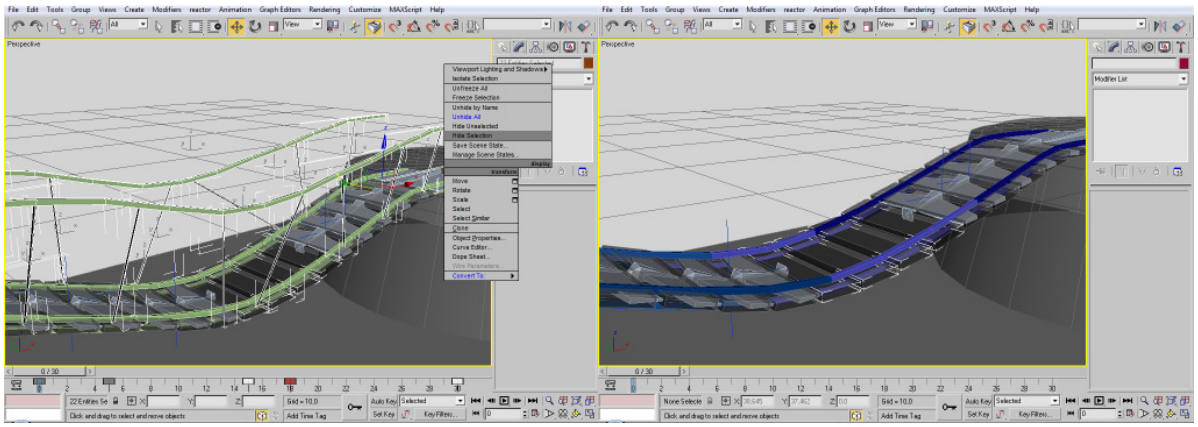
Függőhíd:



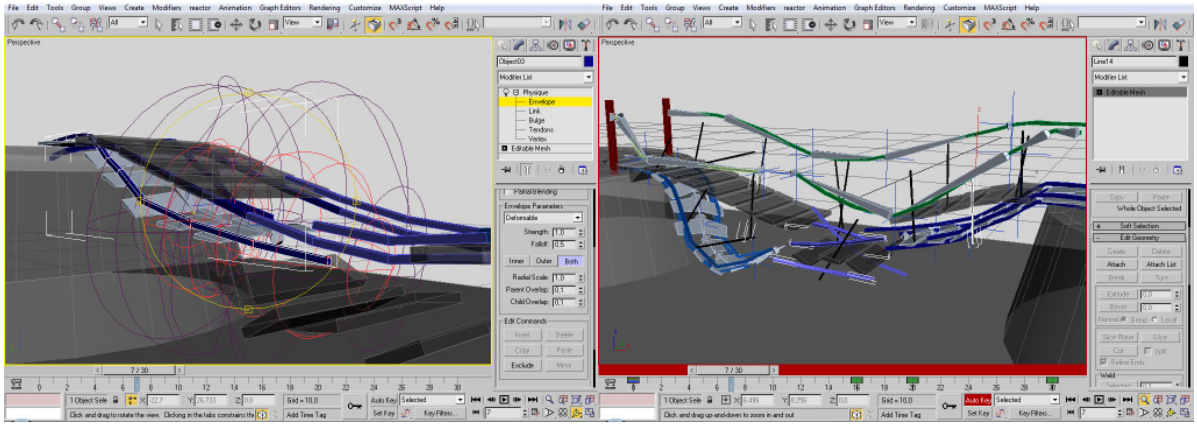
Ebből három különböző hosszúságú is készült, amikhez egy-egy animáció is tartozik. Azért készítettem animációkat a leomláshoz, mert a programból vezérelt fizikai erőhatás modellezés túlságosan számításigényes a komplexebb modelleken és gyakran pontatlan (még a profi játékokban is), így alkalmazása meghaladja a program és a cél platform kereteit. Az animáció készítésének néhány fontosabb lépését az alábbi ábrák szemléltetik:



A szürkés félig áttetsző elemek a kimerevített modellek. Az első képen egy egyszerű csontváz modell látható kék színű keresztekkel jelölt kontrollpontokkal. A második képen az alsó időzítő segítségével rögzíteni lehet a jelenet adott időpillanatban kívánt állását, mely ebben az esetben a kontroll pontok pontos koordinátáit, valamint a közöttük alkalmazandó függvény megadását jelenti.



A következő lépés az összes olyan modell kialakítása, mely részt közvetlenül részt vesz az animációban, majd a csontváz csontjait hozzákapcsolhatjuk a modellekhez vagy azok bizonyos részeihez. A modellek elkészítése során fokozott figyelmet kell szentelni az élek megfelelő kialakításának. Azoknál a pontoknál, melyeknél hajlítás történik, több élt kell alkalmazni és legjobb, ha ezek közel merőlegesek tudnak maradni a hajlítás irányára valamint egymással közel párhuzamosak. A jobb oldali ábrán a hidat összekapcsoló elemek különböző színűek, ez azért van, mert mivel az egy egységet képező modellek nem választhatóak el egymástól (nem törhetőek el).

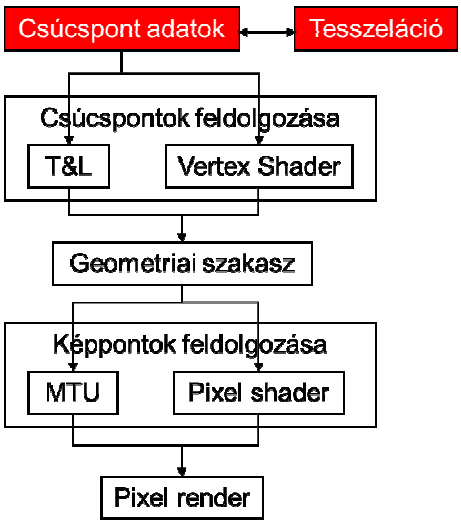


Legutoljára pedig a finomhangolás marad, a csontok néha olyan csúcspontokat is magukkal húznak, melyeket nem kellene, vagy kellene, de csak kisebb súllyal. Ezeket mindenképpen egyenként vagy nagyon kis csoportokban, kézzel érdemes kiküszöbölni, és közben újra és újra lejátszani a jelenetet.

Megjelenítés

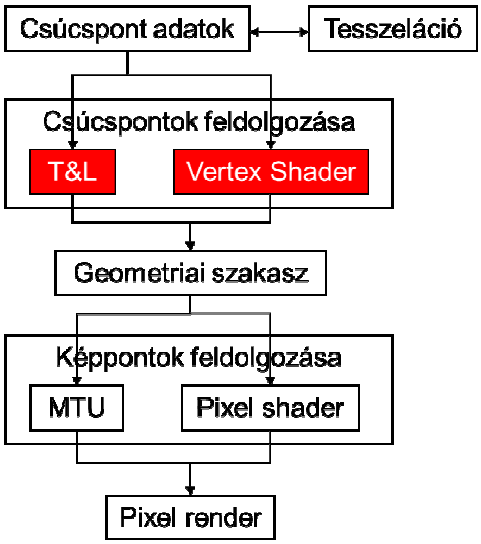
A grafikai elemek megjelenítéséhez a DirectX csatolófelületet használtam, mely remekül képes kihasználni a legkorszerűbb hardverek nyújtotta lehetőségeket is. A program írásának kezdetekor ennek a 9.0c verziója volt a leginkább elterjedve mind az oktatásban, mind az otthoni számítógépek világában, ezért én is ezt a verziót választottam. A DirectX segítségével a képszintézis minden lépését rendkívüli precizitással lehet befolyásolni, így a kód elég nagy részét a különböző grafikát szabályozó függvényhívások, valamint matematikai egyenleteket megvalósító parancssorozatok teszik ki.

A képszintézis alaplépéseit az alábbi ábrák szemléltetik:



Az első lépés a kirajzolandó objektumokat reprezentáló adatok összegyűjtése és pufferekbe rendezése. Ekkor még lehetőség van különböző algoritmusok segítségével az adatok méretét csökkenteni vagy növelni. Előbbi akkor lehet indokolt, ha az objektum távol esik a szemlélőtől, ezáltal nem ront a látott képen az, ha bizonyos csúcspontokat eltávolítunk a modellből. Természetesen lehetőség van a meglévő adathalmaz

finomítására is, ezeket különböző subdivision technikák segítségével a modernebb (DirectX10 – geometry shader) rendszerekben könnyedén elvégezhetjük futás időben is. Az ilyen algoritmusokat közös gyűjtőnevükön: LOD (Level Of Detail) emlegetik. Ilyen például az N-patch vagy az RT-patch. Az adatok összegyűjtése után pontosan meg kell mondani a rendszernek, hogy hova töltse azokat, hogyan kezelje, és legfőképpen hogyan értelmezze őket. Az én programomban erre egy példa a vertex buffer, amit a VGA memóriájába kell tölteni, és csúcspont koordinátanégyes, normál koordinátanégyes, valamint textúra koordinátapáros módon kell értelmezni.



A következő lépés a képszintézis során a megadott csúcspont adatok feldolgozása a homogén 4d-s koordináta rendszerben értelmezve. A DirectX 9-ben ez két eltérő módon történhet. A régi módszer a T&L (Transform and Lighting) igénybevétele volt, ahol „leegyszerűsítve” csupán meg kellett mondani a rendszernek, hogy hova, mit és milyen fényforrásokat, valamint anyagmintákat használva rajzoljon. Ez egy nagyon egyszerű megoldás volt, azonban a hátránya az volt, hogy komplex, egyedi csúcspontfeldolgozásra nem adott lehetőség, mivel például 8 fényforráson kívül nem lehetett többet használni. A Vertex Shader segítségével saját kódot lehet írni, mely minden bejövő csúcsponton lefut, és a megfelelő helyre transzformálja a világ koordináta rendszerben, valamint a kívánt egyéb számértékeket (tipikusan 4d-s vektorok) rendeli a csúcspontokhoz.

A legalapvetőbb dolog szerintem mindenképpen az objektumok transzformációinak megvalósítása, hiszen e nélkül minden egybecsúszna a világ (közös) koordináta rendszerben, és így a számítógép monitorán is. A mátrixok a grafikában és más keretrendszerekben (pl. OpenGL) jól ismert mátrixokkal egyeznek meg:

Eltolás:

$$T(\vec{v}) = \vec{v} \cdot M_t = \begin{pmatrix} v_x & v_y & v_z & v_w \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{pmatrix}$$

Skálázás:

$$T(\vec{v}) = \vec{v} \cdot M_s = \begin{pmatrix} v_x & v_y & v_z & v_w \end{pmatrix} \cdot \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Forgatás (a pozitív forgatási irány a tengely csúcsából az origó felé nézve megegyezik az óramutató járásának irányával):

X	Y	Z
$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & \sin\alpha & 0 \\ 0 & -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} \cos\alpha & 0 & -\sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ \sin\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} \cos\alpha & \sin\alpha & 0 & 0 \\ -\sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

A komplexebb forgatások megvalósíthatóak a csavarás-billenés-fordulás (Z-X-Y) hármassal is, ami tulajdonképpen a megfelelő mátrixok szorzatát jelenti: $R_{3D}(\vec{v}) = \vec{v} \cdot R_{3DZ} \cdot R_{3DX} \cdot R_{3DY}$

Ez a módszer néha kimondottan hasznos, ám sajnos szenved az úgynevezett gimbal lock jelenségtől, valamint a sorrendi kötöttségtől. Lehetőség van azonban tetszőleges tengely körül is forgatni, melynek mátrixa:

$$R_{3DA} = \begin{pmatrix} \cos\alpha + x^2(1-\cos\alpha) & xy(1-\cos\alpha) + z\cdot\sin\alpha & xz(1-\cos\alpha) - y\cdot\sin\alpha & 0 \\ xy(1-\cos\alpha) - z\cdot\sin\alpha & \cos\alpha + y^2(1-\cos\alpha) & yz(1-\cos\alpha) + x\cdot\sin\alpha & 0 \\ xz(1-\cos\alpha) + y\cdot\sin\alpha & yz(1-\cos\alpha) - x\cdot\sin\alpha & \cos\alpha + z^2(1-\cos\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Harmadik lehetőségként kínálkozik még a kvaterniós forgatás, melynek segédfüggvénye már régóta megtalálható a DirectX-ben.

Az ezeket megvalósító „mindennapos” parancsok az alábbiak:

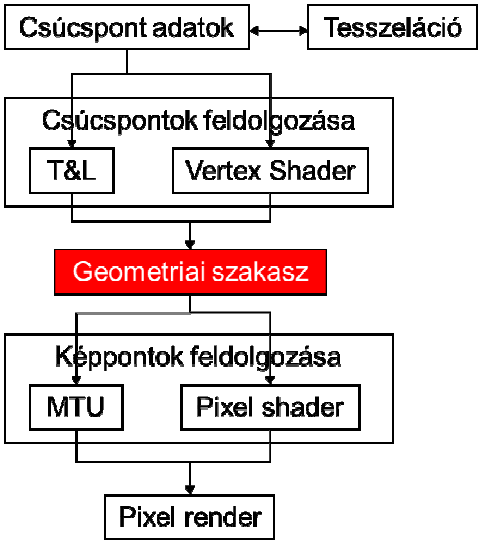
```
D3DXMatrixRotationX;
D3DXMatrixRotationY;
D3DXMatrixRotationZ;
D3DXMatrixRotationYawPitchRoll;
D3DXMatrixRotationAxis;
D3DXMatrixTranslation;
D3DXMatrixRotationQuaternion;
D3DXMatrixScaling;
```

A parancsoknak ugyanakkor nincsenek „shaderes” megfelelőik, ott a mátrixok és a vektorok megfelelő komponenseinek manipulálásával lehet elérni a kívánt hatásokat. Természetesen a shader bemenő adatai között ott lehetnek az akár már teljesen elkészített forgatómátrixok is. Egy ilyen kódra példa:

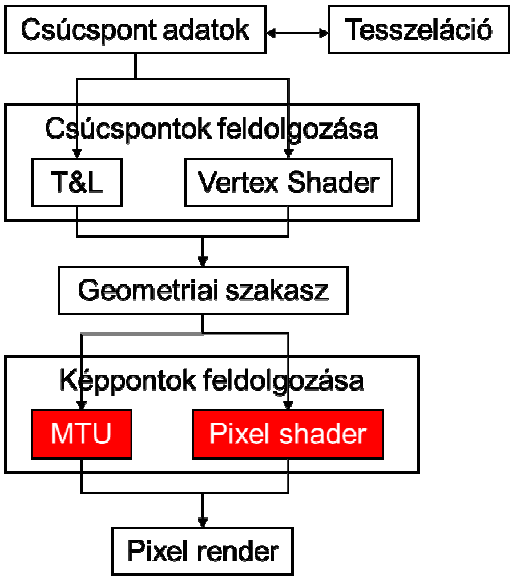
```
VS_OUTPUT vs_main (float3 posLocal_in : POSITION0,
                           float3 normalLocal_in : NORMAL0)
{
        VS_OUTPUT outVS = (VS_OUTPUT)0;
        float3 normalWorld = mul(float4(normalLocal_in, 0.0f), gWorld).xyz;
        normalWorld = normalize(normalWorld);
        float3 posW = mul(float4(posLocal_in, 1.0f), gWorld).xyz;
        float3 lightVecW = normalize( gLightPos – posW );
        float i_diffuse = max(dot(lightVecW, normalWorld), 0.0f);
        float3 diffcolor = i_diffuse *  (gDiffuseMaterialColor *
gLightColor).rgb;
        outVS.VertexColor.rgb = gAmbientMaterialColor + diffcolor;
        outVS.VertexColor.a   = gDiffuseMaterialColor.a;
        outVS.PositionProj = mul(float4(posLocal_in, 1.0f),
gWorldViewProjection);
return outVS;
}
```

Az utolsó előtti sorban látható, hogy a teljes local koordináta rendszerből a projekciós csonkagúlába vetítés megtörténhet a vertex shaderben.

Az ilyen programot közvetlen a GPU hajtja végre. Egyetlen egy dolgot nem lehet megtenni, az pedig az új csúcspontok létrehozása. Amennyiben ilyenre lenne szükség, akkor azt vagy a tesszelációs szakaszban kell megtenni, vagy Shader Model 3.0 vagy újabb verziót kell igénybe venni.



A képszintézis következő lépése a Geometriai szakasz, aminek fő funkciója az általában perspektivikus hatást biztosító csonkagúlában (prejekciós koordináta rendszer) lévő vektorgrafikus adatok raszterizációja. Tehát itt már normalizált ábrázolási térben van minden. A részletek verzió és implementáció függők, de az aktuális működés mindig megtalálható a keretrendszer dokumentációjában. A cél általában a csúcspontok csoportjainak háromszögekre bontása, valamint ezen háromszögekből a fragmentek kialakítása. Amit ezen a ponton kiemelnék, az a bacface culling, ami a normálvektorok és a szemlélő vektora által bezárt szögekből határozza meg, hogy mely háromszögeket lehet eldobni, miáltal akár 40%-al is javulhat a teljesítmény. A kilógó részek metszése (kivágása), valamint a csúcsok közötti lineáris interpoláció is ebben a szakaszban történik. A folyamat végén úgynevezett fragmentek keletkeznek, amik potenciális képpontként tekinthetők.



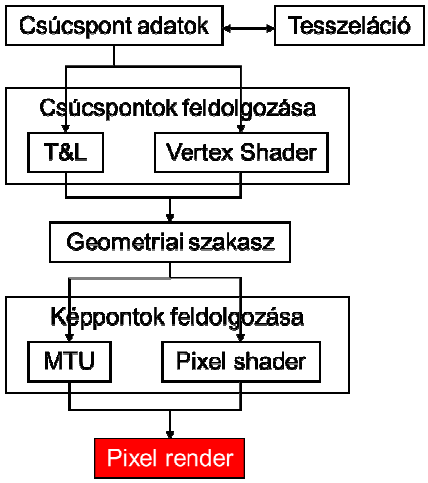
A geometriai szakaszban keletkező fragmenteket a rendszer újból feldolgozza, és ezen a ponton ismét két lehetőség kínálkozik. A régi, merev csővezetékhez tartozó az MTU (Multitexturing Unit) valamint a dinamikus, GPU-nak szóló kódolást lehetővé tevő Pixel Shader igénybe vétele. Funkciólyát tekintve, itt a csúcspontok színértékei, valamint a textúrák mintavételezéséből kapott értékek interpolálásával keletkezik egy majdnem véglegesnek tekintendő pixel színérték. Ha a dinamikus lehetőséget választjuk, akkor lehetőség van olyan megvilágítási modell implementálására is, mely minden képpontra külön számolja ki a fényhatásokat (Phong ármyalás). A talán legegyszerűbb pixelshader program az alábbi sorokban olvasható:

```
float4 ps_main (float4 color_in : COLOR0) : COLOR
{
    return color_in;
}
```

A bemeneti paraméter csupán egy 4 elemű vektor, a szolgáltatott színérték pedig pontosan ezen vektor által reprezentált színnel fog megegyezni.

Egy második, megvilágítást megvalósító példa:

```
float4 ps_main (float3 normalW_in : TEXCOORD0,
                        float3 camObjDirW_in : TEXCOORD1,
                        float3 lightVecW_in : TEXCOORD2) : COLOR
{
        float3 normalWorld = normalize(normalW_in);
        float3 viewWorld = normalize(camObjDirW_in);
        float3 lightWorld = normalize(lightVecW_in);
        float i = max(dot(lightWorld, normalWorld), 0.0f);
        float3 diffuseColor = i*( gDiffuseMaterialColor * gLightColor).rgb;
        float3 r = reflect(-lightWorld, normalWorld);
        float i_spec = pow(max(dot(r,viewWorld),0.0f), gSpecularParam);
        float3 specColor = i_spec*( gSpecularMaterialColor *
gLightColor).rg
        return float4 (gAmbientMaterialColor + diffuseColor + specColor,
gDiffuseMaterialColor.a);
}
```

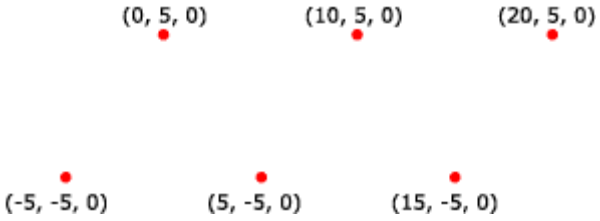


Végezetül pedig, az utolsó szakasz a képpontok megjelenítését célozza. A megjelenítés előtt azonban még számos teszten kell átesniük a fragmenteknek: mélységteszt (Z-test), alfateszt, stencilteszt alfakeverés, ködhatás. Végezetül, ha az összes előírt teszten sikerrel átesik a fragment, akkor egy utolsó, nézeti->fizikai eszközkoordináta rendszer transzformáció következik. Ekkor alakul ki a tényleges pixel.

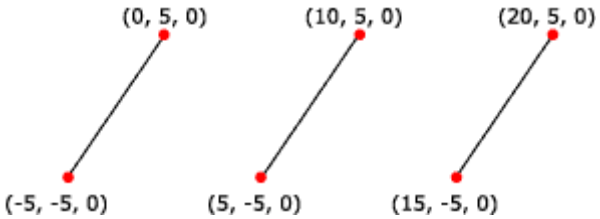
A 2005 után megjelent videokártyák többsége már rendelkezik programozható csővezetékkel. A rögzített csővezeték (T&L, MTU) azonban kompatibilitási okokból még a ma gyártott eszközökben is benne van, azonban ezt a programozott csővezetéken keresztül emuláljája.

A DirectX 9-ben, mint a minden más hasonló keretrendszerben, vannak alapból támogatott geometriai primitívek, melyekből komplexebb alakzatokat lehet felépíteni. Ezek az alábbiak:

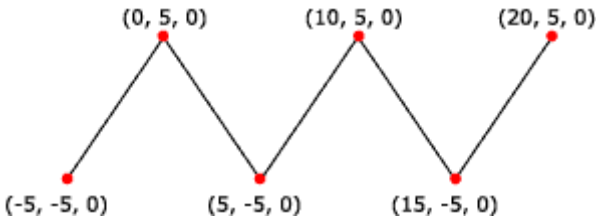
D3DPT_POINTLIST



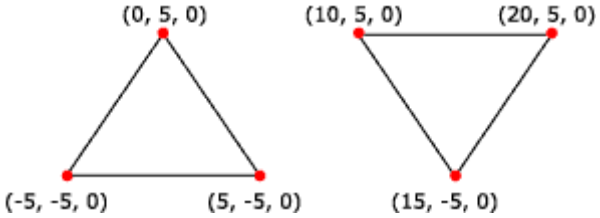
D3DPT_LINELIST



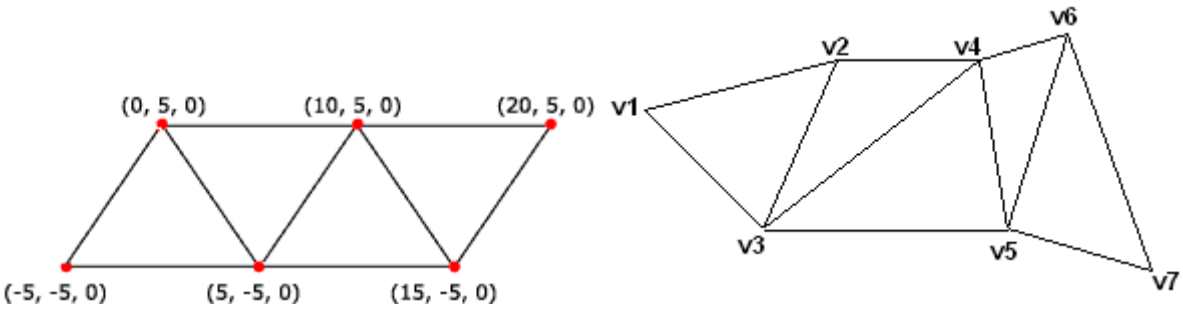
D3DPT_LINESTRIP



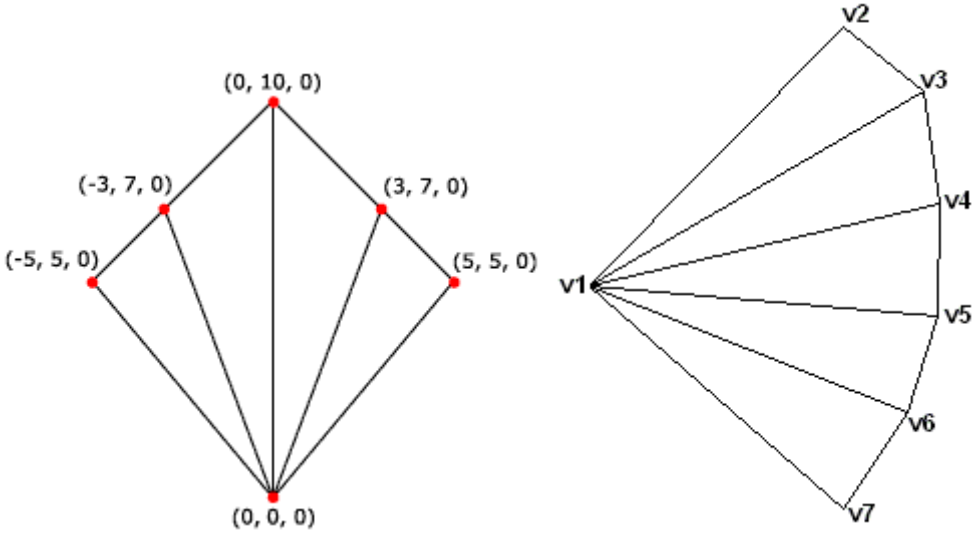
D3DPT_TRIANGLELIST



D3DPT_TRIANGLESTRIP



D3DPT_TRIANGLEFAN



Érdekességként megjegyzem, hogy a DirectX 10-ből kivették a trianglefan primitívet, viszont a többihez (a pointlist-en kívül) létrehoztak egy, a pontok szomszédjait is tartalmazó primitívtípust.

A megjelenítésről talán elég ennyit megemlíteni, a többi rész könnyen párhuzamba vonható az egyetemen is tanított hasonló keretrendszerek (pl.: OpenGL) vonásaival. A matematikai leírás, a mátrixok levezetése valamint az ezeket tükröző parancsok megtalálhatóak a felsorolt szakirodalmakban, valamint a DirectX SDK-hoz járó dokumentációkban.

Kódolás

A program oroszlánrészét C++ kódban írtam, ami az idő során tetemes (körülbelül 5200 sor) méretre hízott, ezért a szakdolgozat írott részében csak nagyon apró, összefüggéstelen részletek találhatóak meg belőle. A kódsorok között a könnyebb érthetőség kedvéért, kommentek formájában további magyarázatok találhatóak. Érdekességként megemlítem, hogy a program mesterséges intelligenciáját képező kódsorok száma csupán mintegy ezres nagyságrendet képvisel.

A fordítás idején a DirectX SDK 2010. júniusi változatát használtam. A további felhasznált függvénykönyvtárak listája megtalálható a projektállományban.

Több osztályból épül fel a program, néhány csak segítséget nyújt a többi osztálynak bizonyos adatok lekérdezésében, néhány viszont (illetve annak példányai) 1-1 egész funkcionalitást képvisel.

A következőkben a legfontosabb osztályokat fogom felsorolni, valamint a hozzájuk tartozó funkcionalitást leírni.

AllocateHierarchy:

Az animált modelleket speciálisan kell beolvasni. Modellezés közben a testhez egy csontvázat rendelünk, majd a csontokhoz készíti el a program a megfelelő transzformációs mátrixokat. A csontváz felfűzhető egy speciális gráffá, fává (modellhierarchia), melynek a csomópontjaihoz lehet rendelni a különböző forgatómátrixokat, a gyökér elemhez pedig a modell transzlációs mátrixát. A 3D Studio Max 2008 sajnos nem biztosított beépített eszköz az animált modellek .x fájlba történő mentéséhez. A DirectX 9 pedig csak a .x modellekhez biztosít beépített beolvasó és kezelő függvényeket. Megoldásként a Pandasoft Panda Directx Exporter x64 (5.2008.67.0) for 3DS Max 2008 (64 bit) plugin-jét használtam fel, mely ingyenesen letölthető az irodalomjegyzékben megtalálható weboldalról. Az így kapott fájlok egy részével – Elek.X – dolgozik az említett osztály, felépíti a transzformációs fát, betölti a modelleket és a textúrákat, valamint ha már nincs szükség a modellre, gondoskodik a lefoglalt memóriaterület felszabadításáról. Magát a modell betöltését a D3DXLoadMeshHierarchyFromX függvényre bízhatjuk.

```
HRESULT  D3DXLoadMeshHierarchyFromX(
  __in   LPCSTR Filename,
        //X-állomány neve
```

```
  __in   DWORD MeshOptions,
        //Mesh létrehozási opciói
  __in   LPDIRECT3DDEVICE9 pDevice,
        //érvényes IDirect3DDevice9 pointer
  __in   LPD3DXALLOCATEHIERARCHY pAlloc,
        //ID3DXAllocateHierarchy pointer
  __in   LPD3DXLOADUSERDATA pUserDataLoader,
        //felhasználói adatokra mutató pointer
  __out  LPD3DXFRAME *ppFrameHierarchy,
        //pointer a D3DXFRAME hierarchiára
  __out  LPD3DXANIMATIONCONTROLLER *ppAnimController
        //animációvezérlő
);
```

A `ppFrameHierarchy` paraméter mutat a modellhierarchia gyökérszegmensére. A `ppAnimController` pedig a .X állományba exportált animációs vezérlő pointere – amennyiben az állomány tartalmaz ilyet. Az animációs vezérlővel strukturálni és időzíteni lehet az animációs mintákat. Lehetőség van egy adott minta elindítására több sebességgel is, vagy akár visszafele lejátszani azt. Magát az osztályt az `pAlloc` mutató miatt kellett létrehozni. A DirectX ugyanis csupán interfészt – itt absztrakt osztályt – biztosít az animációkhoz. Maga az osztály leginkább a szegmentálásért felelős a hivatalos leírás szerint. Az `ID3DXAllocateHierarchy` osztályból a `d3dx9anim.h` headerben van deklarálva.

Boar, Boars:

Ezek az osztályok közösen kezelik a vaddisznók megjelenítését valamint mozgatását. Szintén itt történik a kommunikáció a karaktereket irányító lépésajánló osztállyal. A használni kívánt MI osztályt a felhasználó választhatja ki még a program menüjében. A `Boar` osztály kizárólag egy malac megjelenítéséért felelős. A `Boars` osztály pedig egyfajta kollekciónak tekintendő, melyben az aktuális helyzetnek megfelelően számos `Boar` található. A megjelenítést biztosító osztály rendelkezik az animációs vezérlést biztosító függvényekkel is, ám ezek nagy részétét sajnos nem tudtam kiaknázni, mivel nem sikerült jól használható animációs mintákat gyűjteni. A vaddisznók lépéseinek folyamatos karbantartásáért az alábbi kis kódrészlet felelős:

```
for (int instance=0; instance<cBoars->size(); ++instance) {
       pActBoar=(*cBoars)[instance];
       if (IsElekReached()) GCORE->GetGame()->KillElek(1);
       if (!pActBoar->Doing && pActBoar->act) {
              int step = GetOffer(pActBoar->sn);
              if (step == -2) {
                     pActBoar->act=0;
                     pActBoar->SetUpAnimRender(dt);
```

```
                        Model::AnimRender ();
                        continue;
                }
                if (step == -1) {
                        pActBoar->SetUpAnimRender(dt);
                        Model::AnimRender ();
                        continue;
                }
                pActBoar->MoveTo(step);

        }
        pActBoar->SetUpAnimRender(dt);
        Model::AnimRender ();
}
```

Mint látható, a lépésajánlás előtt mindig lefut egy ellenőrzés, melynek célja annak megállapítása, hogy a kijelölt vaddisznó elérte-e a játékos karakterét. Ha igen, akkor vége szakad a játéknak, és a játékos veszít.

Az ajánlható lépések közül van pár kakukktojás, mely nem egy adott iránnyal egyezik meg, hanem inkább vezérlő utasításnak tekintendő. Az egyik a -1, mely arra utal, hogy jelenleg nem sikerült lépést ajánlani. Ez előfordulhat olyan esetben, amikor a vaddisznó a jelenlegi állások szerint a lehető legjobb pozícióban van. A második ilyen eset a -2-vel való visszatérése az MI algoritmusoknak. Ez arra utal, hogy a vaddisznó annyira rossz helyzetbe került, hogy már nem képes fordítani a játék menetén, ezért a jövőben már nem kell lépést ajánlani neki. Utóbbi csupán az optimalizáció miatt vált szükségessé.

A kódban még feltűnhet, hogy geometriai kirajzolás nem a `Boar` egy példányából, hanem egyenesen a `Model` osztály egy függvényéből történik. Ez azért van, mert a modellek kategorizálva vannak a megjelenítéshez szükséges igényeik szerint, és ezen kategóriákhoz van rendelve egy-egy általánosabb megjelenítő függvényeket meghívó függvény. Magában az aktuális modellt képviselő osztályokban az említett függvények paramétereinek előállítása történik. Ilyen paraméter például a World, View mátrixok előállítása, valamint ezek kombinálása a `Camera` osztályból lekérhető projekciós mátrixszal. Az ilyen jellegű csoportosításokkal szintén a program sebességének javulását próbáltam biztosítani.

BridgeModel:

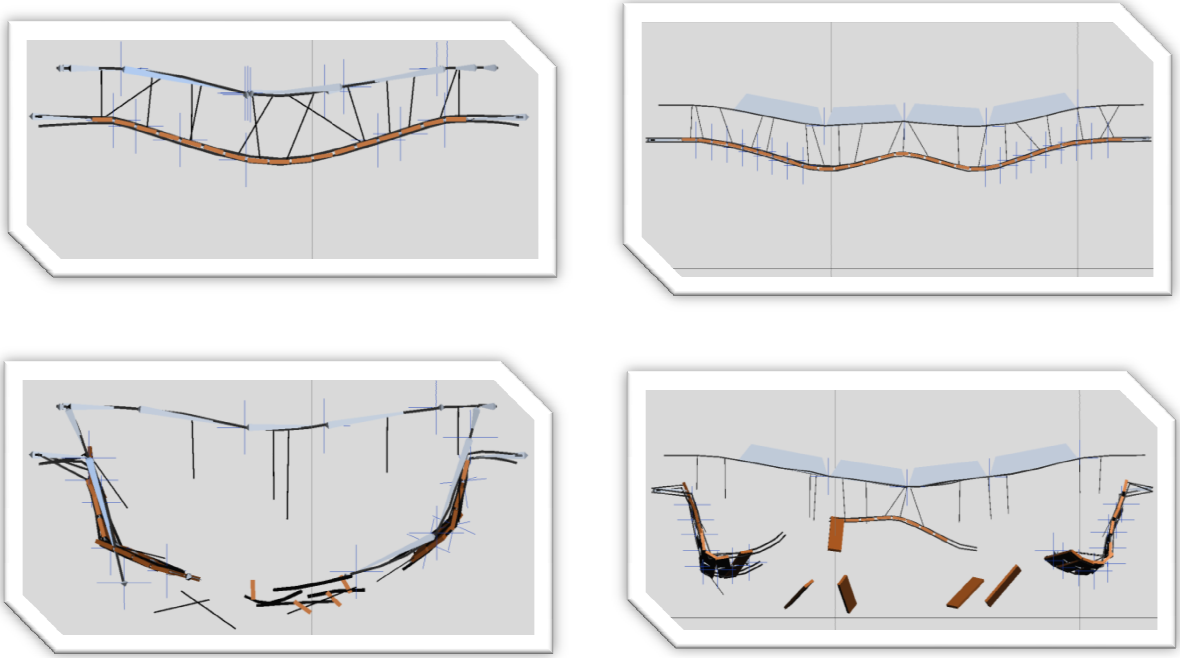
Ezen a ponton még nehéz lenne pontosan kifejteni ennek az osztálynak a jelentőségét. Röviden talán annyit mindenképp érdemes róla megemlítenem, hogy szintén a fentebb

említett animációs vezérlővel, illetve mintákkal dolgozik a modell csúcspontadatait felhasználva.

Az itteni modell csontváza inkább csak egy gerincoszlopra hasonlít, ám az élethűbb mozgatáshoz itt az animációba kellett foglalni egyedi, a modell egyes vertexeire vonatkozó transzformációkat is. Ezzel lehetett megvalósítani az egyes alkotóelemek vízbe hullását.

Három különböző függőhíd készült, három különböző animációs mintával, melyek akkor játszódnak le, ha a modell leomlik. A leomlás valós időben történő fizikai modellezését azért vetettem el, mivel amikor a program még ebben a fázisban tartott, csak a meglehetősen drága számítógépekkel lehetett volna megvalósítani hasonló leomlás-szimulációt.

A tesztelések során felmerült, hogy esetleg a leomlás után bizonyos, a hidat alkotó elemek a levegőben maradnak. Ez azonban nem történik meg. A probléma abból ered, hogy azokon a gépeken, melyeken kisebb felbontásban, vagy nagy felbontásban, de kisebb monitoron futott a program, a leomlás után fennmaradó kötelek elég vékonyak lettek ahhoz, hogy szabad szemmel olyan másfél méteres távolságból már ne látszódjanak. A nagyobb darabok természetesen jól láthatóak voltak ekkor is. Az alábbi képekben megpróbálom szemléltetni a függőhidak végállapotait, valamint a felépítésüket.



A hidak omlása a `StartCrumble` illetve a `StopCrumble` nevű rövid függvényekkel vezérelhető. Ezek a függvények azért lettek 1-2 sorosak csupán, mivel kihasználtam, a játékszabályok adta optimalizálási lehetőségeket. Jelen esetben arról van szó, hogy egy híd csak a játékos alatt képes leomlani és az omlást éppen befejezi, mire a játékos eléri a célbavett szigetet.
A későbbi fejlesztések során (esetleg el lehetne menni több játékos irányába is) a leomló hidakat, illetve azok animáció vezérlőinek aktuális állapotát gondosan nyilván kellene tartani.

Camera:

Ebben az osztályban a nézőpont vezérlését biztosító függvények kaptak helyet, valamint itt történik a képszintézis kezdeti paramétereinek beállítása is, mint például a nézeti csonka gúla. A nézeti csonka gúla határozza meg a világ kamera által látható „szeletét”: minden ami ezen kívül esik, a képszintézis geometriai szakaszában kivágásra kerül.
A szokványos megjelenítőeszközök még ma sem képesek háromdimenziós kép előállítására, ezért a programban jelen lévő teret a monitor síkjára kell transzformálni. Az ilyen és ehhez hasonló transzformációkat síkgeometriai vetítésnek vagy projektív transzformációnak nevezik. Ez a funkció a DirectX-ben két lépében valósítandó meg, az egyik a kamera nézeti koordinátarendszer beállítása a `D3DXMatrixLookAtLH` függvénnyel, a másik pedig a projekciós transzformáció beállítása a `D3DXMatrixPerspectiveFovLH` függvénnyel.
Az első függvény célja a nézeti transzformációs mátrix előállítása a kamera világbeli helyzetéből, a nézési irányból valamint a felfele irányt mutató vektorokból. A transzformáció az alábbi lépésekkel valósítható meg:
Először a teljes színteret eltoljuk úgy, hogy a kamera az origóba kerüljön.

$$T = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -k_x & -k_y & -k_z & 1 \end{pmatrix}$$

A következő lépésben a világot kellene a kamera koordinátarendszerébe transzformálni, mivel a jelenetet a kamera szemszögéből szeretnénk látni. Ez egy inverz transzformáció - tehát a kameraobjektum világ-transzformációjának inverzére van szükségünk. Ismert a

kamera koordinátarendszerének oc origója és a bázisvektorai a világ-koordináta rendszerben. A W_{kamera} világ-transzformációs mátrix felírható a következő alakban:

$$W_{kamera} = \begin{pmatrix} uc_x & uc_y & uc_z & 0 \\ vc_x & vc_y & vc_z & 0 \\ nc_x & nc_y & nc_z & 0 \\ oc_x & oc_y & oc_z & 1 \end{pmatrix}$$

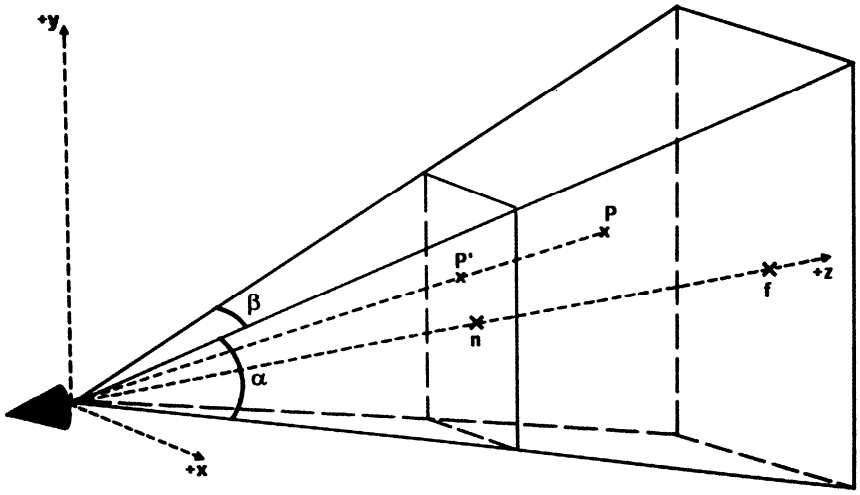
ahol uc a vízszintes, vc a függőleges és nc a kameranézeti irányba mutató vektora. Az uc a másik két vektor keresztszorzataként kapható. Azonban az is megadható a DirectX-ben, hogy jobb vagy balsodrású koordinátarendszert szeretnénk kapni, ezért az lehet az említett vektor -1 szerese is. Mivel azonban a kamera úgyis az origóba van eltolva a T mátrix szerint, ezért az utolsó sor lehet a (0,0,0,1) vektor is. Ekkor egy ortogonális mátrixot kapunk, melynek inverze megegyezik a transzponáltjával. A két mátrix szorzata adja a nézetre alakítás mátrixát.

$$V = T \cdot W_{kamera}^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -k_x & -k_y & -k_z & 1 \end{pmatrix} \cdot \begin{pmatrix} uc_x & vc_x & nc_x & 0 \\ uc_y & vc_y & nc_y & 0 \\ uc_z & vc_z & nc_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} =$$

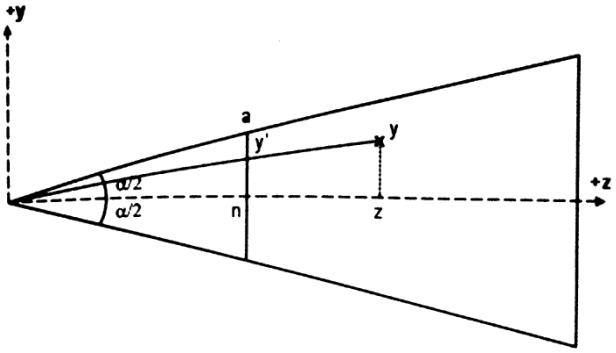
$$= \begin{pmatrix} uc_x & vc_x & nc_x & 0 \\ uc_y & vc_y & nc_y & 0 \\ uc_z & vc_z & nc_z & 0 \\ -\underline{k} \cdot \underline{u_c} & -\underline{k} \cdot \underline{v_c} & -\underline{k} \cdot \underline{n} & 1 \end{pmatrix}$$

A következő lépés a már említett vetületi transzformációs mátrix kidolgozása, mellyel a nézeti csonka gúlába foglalt csúcspontok koordinátáit normalizált eszközkoordinátákká kell alakítani úgy, hogy a mélységinformáció megmaradjon. Itt is egy mátrix formájában keressük a megoldást, melyet a következő lépésekkel kaphatunk meg:

- a nézeti tér pontjait a közeli vágósíkra vetítjük úgy, hogy a látható pontok x és y koordinátái a [-1, 1] intervallumba essenek;
- a nézeti tér szakaszai a normalizált eszköztérben is szakaszokra kell leképeződjenek; pontok z koordinátáit – mélység – a [0, 1] intervallumra kell képezni;



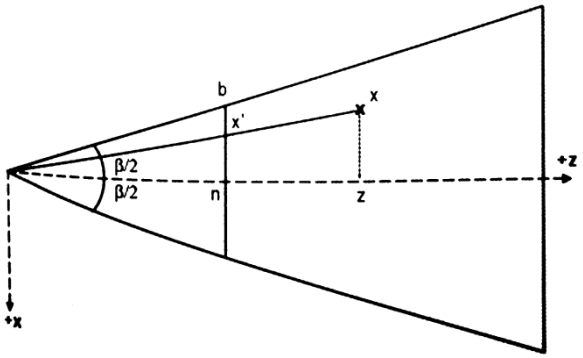
Természetesen ezen a ponton is segítségül hívható a megfelelő segédfüggvény: D3DXMatrixPerspectiveFovLH aminek számos paramétere van. A számításokhoz nélkülözhetetlen információk az ablak szélesség/magasság aránya (aspect ratio), az egyik látószög (általában a függőleges), valamint a transzformálandó P pont nézeti koordinátái. A vetítés síkokra bontható a könnyebb láthatóság kedvéért:



Az ábrán két hasonló háromszög látható, így felírható a következő összefüggés:

$$\frac{y'}{n} = \frac{y}{z} \Rightarrow y' = \frac{n \cdot y}{z}$$

Hasonló gondolatmenettel kapható x' koordinátája is:



$$x' = \frac{n \cdot x}{z}$$

Az ábrákból látható, hogy fennáll: $-b \leq x' \leq +b$ valamint $-a \leq y' \leq +a$

Tehát: $x_{norm} = \frac{x'}{b} = \frac{n \cdot x}{z \cdot b}$, illetve $y_{norm} = \frac{y'}{a} = \frac{n \cdot y}{z \cdot a}$

illetve érvényes: $\tan\frac{\beta}{2} = \frac{b}{n}$, valamint $\tan\frac{\alpha}{2} = \frac{a}{n}$, így kapjuk:

$$\boldsymbol{x_{norm}} = \frac{x'}{b} = \frac{n \cdot x}{z \cdot b} = \frac{x}{z \cdot \tan\frac{\beta}{2}} = \frac{x}{z \cdot \frac{\frac{b}{a} \cdot a}{n}} = \frac{x}{z \cdot r \cdot \frac{a}{n}} = \frac{x}{z \cdot r \cdot \tan\frac{\alpha}{2}}$$

$$\boldsymbol{y_{norm}} = \frac{y'}{a} = \frac{n \cdot y}{z \cdot a} = \frac{y}{z \cdot \tan\frac{\alpha}{2}}$$

ahol r a fent említett képarányt jelenti.

Az egyenletek rendre beszorozhatóak z-vel, akkor a nevezőből z kiesik, úgy pedig könnyebb visszavezetni mátrixszorzásra az egyenleteket. Persze utólag a korrekt értéket szeretnénk visszakapni, amit egy z-vel való osztással valósíthatunk meg. Azonban itt kihasználható, hogy minden korszerű grafikus rendszer homogén koordinátás alakkal dolgozik. Ezt a stratégiát figyelembe kell venni a z_{norm} meghatározásakor is, melyet ugyanúgy felszoroz majd a rendszer z-vel, aminek az eredménye a következő formában fog megjelenni: $z \cdot z_{norm} = u \cdot z + v$. Azaz a függvényt a következő formában keressük: $z_{norm} = u + \frac{v}{z}$

A közeli és a távoli vágósíkban ismertek a kívánt koordináták, ezért felállítható egy egyenletrendszer:

$$u + \frac{v}{n} = z_n = 0 \Rightarrow v = -u \cdot n$$

$$u + \frac{v}{f} = z_f = 1 \Rightarrow 1 = \frac{u(f-n)}{f} \Rightarrow u = \frac{f}{f-n}$$

$$v = -u \cdot n = -\frac{f \cdot n}{f-n}$$

$$\Longrightarrow \boldsymbol{z_{norm}} = u + \frac{v}{z} = \frac{f}{f-n} - \frac{f \cdot n}{z(f-n)}$$

A három képletből 1 mátrixot írhatunk fel, de közben figyelembe kell venni, hogy a z-vel való osztást a homogén koordinátás osztás során szeretnénk viszont látni, tehát az (x,y,z,w) koordinátanégyesben w:=z.

$$P = \begin{pmatrix} \frac{x}{r \cdot \tan\frac{\alpha}{2}} & 0 & 0 & 0 \\ 0 & \frac{y}{\tan\frac{\alpha}{2}} & 0 & 0 \\ 0 & 0 & \frac{f}{f-n} & 1 \\ 0 & 0 & -\frac{f \cdot n}{f-n} & 0 \end{pmatrix}$$

A fenti levezetésből két dolog következik:

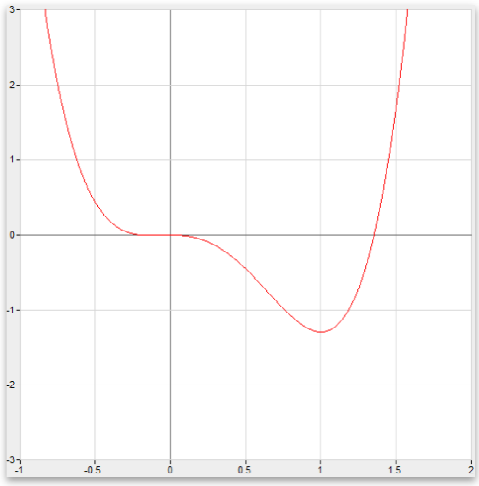
Gyakran megpróbálják 0-ra vagy nagyon 0 közelire állítani a közeli vágósíkot, abban a reményben, hogy úgy majd a legközelebbi pontok is kirajzolódnak. Ez a kerekítési hibákkal is együtt számolva, 0-val való osztást eredményezhet, ami hibás megjelenítéshez vezet. A következő probléma az, hogy azon z értékek melyek a távoli vágósíkhoz esnek közel, ritkábban fognak elhelyezkedni a kapott z tengelyen, ezért van az, hogy a sokak által ismert „Z-fighting” jelenség leginkább a távolabbi objektumokon jelenik meg.

Ebben a játékban a fennmaradó kamerakezelési funkciók a két forgatás funkció biztosítását, valamint a zöld gombák esetén bekövetkező átbillenést szolgálják. A forgatás az idő függvényében történik, méghozzá úgy, hogy a kezdeti sebesség nagy legyen, majd folyamatosan csökkenjen. A generátorfüggvény a négyzetgyök volt.

Elek:

Elek a `Model` osztályból származik, és leginkább az animációt biztosító funkciókat veszi igénybe. Amikor a játékos elindul valamelyik irányba egy függőhídon, akkor a játékos tengerszinttől való magasságát is ki kell számolni, hogy a karakter valóban a híd felületén mozoghasson. Próbáltam itt is az optimalizálásra törekedni akkor, amikor – kihasználva, hogy a hidak görbülete fajtánként ugyanolyan – előre kiszámoltam azt a függvényt, ami az aktuális görbület magasságát közelíti az idő függvényében. Ezeket a függvényeket a numerikus módszerek tárgyon is tanított interpolációs eljárással konstruáltam, az első deriváltak és bizonyos helyek ismert pozíciói alapján. Az egyik híd ilyen függvénye például:

y=-1.6527*x*x+0.5384*x*x*(x-0.33)+3.053*x*x*(x-0.33)*(x-1)



Mivel az összes híd szimmetrikus, ezért a függvény értékes értelmezési tartománya csupán a 0-1-ig vett zárt intervallum.

Ebben az osztályban történik még a játékost leíró adatstruktúra karbantartása is, illetve bármilyen mozgást követően a megfelelő naplózó függvények meghívása, valamint a mesterséges intelligenciát biztosító osztály értesítése a változásokról. Az állapottér-reprezentációt úgy alakítottam ki, hogy egy időben lehessen tárolni a játékos tartózkodási helyét, valamint a következő lépés utáni pozícióját. Ez megkönnyítette a gépi játékosok azon képességének programozását, amikor megállnak egy szigeten, és egyszerűen bevárják, hogy a játékos odaérjen.

Game:

A programban két fő ciklus van, az egyik a menürendszerben tartja a játékost, a másik pedig a tényleges játékmenet során gondoskodik a folyamatos élményről. A második kapott ebben az osztályban helyet, valamint a futáshoz szükséges objektuminicializáló hívások is itt történnek. Futás közben nem lenne jó, ha a játékos csalhatna, ezért itt megszakad a kapcsolat a Windows üzenetkezelőjével, és a DirectX alacsony szintű billentyűzetkezelője kapja meg a további lehetőségeket a leütött billentyűkre való reagálásra. A billentyűzet kezelése egyszerű bitmaszkolással történik, melyből egyértelműen következik, hogy a meghívás pillanatában le van-e nyomva adott billentyű vagy sem. Ennek két előnye van: az első, hogy így általános esetekben letiltható a Windows, a Ctrl+Alt+Del vagy egyéb a program futását megszakítani akaró események. Természetesen az Esc billentyűvel bármikor ki lehet lépni. A másik, még nagyobb előnye az alacsony szintű kezelésnek az, hogy nem szenved annyi késleltetést a vezérlés, illetve nincs az a DOS-ból átörökített jellegzetes reagálási séma, mint amikor megnyomnak egy billentyűt hosszan, eljut a jel a programig, majd nyomva tartás után a többi, gyakoribb jel csak pár tizedmásodperces késéssel érkezik meg. Ez a viselkedés szöveg gépelésénél rendkívül kényelmes - a menü vezérlésére is ezt használtam - de valós idejű alkalmazásokat sajnos képtelenség rendesen vezérelni vele.

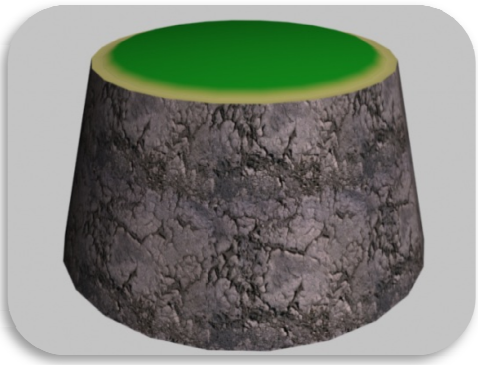
Az MI inicializálásának második fele is itt történik, mivel ezen a ponton már elérhető a pályát reprezentáló gráf, valamint ismert a játékosok pozíciói. Az inicializálás első fele még a főmenü betöltése közben, valamint a stratégia módosításakor történik. Ennek az osztálynak a legfőbb függvénye a `GameRender` függvény, melyben a kirajzoló függvények hívása történik. A kirajzolás az OpenGL-hez hasonló módon, a back bufferbe írással kezdődik, majd ha elkészült egy teljes kép, akkor a back buffer és a front buffer szerepet cserélnek.

Mint minden játék, vagy egyéb valós idejű alkalmazásnál, az idő precíz nyilvántartása itt is elengedhetetlen volt, hiszen az animáló, valamint az MI-s függvények számára is az értelmezési tartomány az idő tengely, továbbá az idő az egyetlen olyan tényező, amely nem függ a különböző hardverkonfigurációktól. Szerencsére a kicsit is újabb Windowsokban (Windows NT vagy 2000) a hagyományos `DWORD timeGetTime()` függvény helyett (ami körülbelül 49.7 nap után újrainicializálódik), kínálkozik egy még pontosabb lehetőség az idő mérésére a „performance timer” által.

A fennmaradó függvények a változások naplózásának elindítását, valamint egyéb adminisztratív szolgáltatásokat nyújtanak.

IslModel:

Ez gyakorlatilag egy darab szigetnek a modelljét kezeli (természetesen példányonként). Azért emeltem ki, mivel a megvalósítás eltér a többi modellhez képest. A játékban fontos volt számomra, hogy az egyes pályák és helyzetek minél kevésbé hasonlítsanak egymásra. Az egyik problémát a szigetek jelentették, mivel nagyon sokat kellett volna belőlük készíteni ahhoz, hogy folyamatosan érezhető legyen egy kis változatosság, ez azonban feleslegesen megnövelte volna a program forrásainak méretét. Helyette a program még a szigetek kihelyezése előtt létrehoz egy tömböt, melyben viszonylag rövid, de véletlen irányba mutató vektorok vannak. Ezt a tömböt minden pálya generálásánál csak egyszer kell feltölteni. Később, a kirajzoláskor a vertex shader lehetőséget biztosít a csúcspontok transzformálására, és az alkalmazásból átadott adatokon is tud dolgozni.



```
if (posLocal_in.y <= 1.0f) {
        float3 normPosLocal = normalize (posLocal_in);
        int i =
acos(dot(normPosLocal,float3(1,normPosLocal.y,0)))*2/radians(15);
        posLocal_in = posLocal_in + float3(gTransf[i],0.0f,gTransf[i+1]);
}
```

Map:

Ez az osztály felelős az állapottér reprezentáció tényleges megjelenítéséért, valamint karbantartásáért. Már az előző részben is említettem, hogy mindenhol próbáltam a változatosságra törekedni, így a pályák sincsenek eltárolva sehol. Az osztály inicializációs része a `MapMaker.dll`-hez fordul egy, a kívánt nehézségi szintnek megfelelő pálya „legyártásáért”. Miután megtörtént a pálya reprezentációs gráfjának összeállítása, az egy pointer képében adódik tovább a Map osztálynak, ahol aztán egy átvizsgálás történik, mely során kiderül, hogy mely objektumok forrását kell betölteni a memóriába. A játékban

eseménynek számít például az, ha az egyik játékos elindul valahonnan, vagy megérkezik valahova. Az eseményeket a `Journalize...` függvények naplózzák, melyek szükség esetén elindítanak egy-egy függvényt, mely akár a játék végét is jelentheti. A lényeg azonban az, hogy a gráf mindig konzisztens maradjon. A reprezentációs gráf pontos szerkezetét a MapMaker részletes tárgyalásánál adom meg.

OSD:

Itt a 2D-s grafika kapott helyet. Ez az összeszedett gombák számának kiíratását, valamint a bal alsó sarokban látható iránytűt jeleni. Az iránytű funkcionalitásban akkor kap szerepet, amikor a játékos belefut egy zöld gombába, ekkor ugyanis nem a kameranézeti iránynak megfelelően történik a mozgás, hanem egy adott, fixen rögzített irányzékhoz viszonyítva, amit az iránytű jelöl ki. A 2D-s grafikák megjelenítését a DirectX dokumentációban bővebben tárgyalt „sprite”-ok segítségével valósítottam meg.

VGACaps:

A VGACaps osztály a video grafikus adapter képességeit igyekszik felmérni. DirectX-ben kétféle objektumot kell létrehozni ahhoz, hogy használható dolgokat jeleníthessen meg az ember. Az első és legfontosabb az `IDirect3DDevice9` interfészpointer, mely lényegében a megjelenítő grafikus hardvert (vagy emulált hardvert) reprezentálja. Ezen az interfészen érhetőek el a DirectX szolgáltatásai, melyek tovább kommunikálnak a driverrel. Az eszközt a `CreateDevice` hívással lehet létrehozni, amihez egy másik interfészpointer kell, ez pedig az `IDirect3D9` típusból származó valamely típus, pl.: `LPDIRECT3D9`.

A rendszert azért találták ki így, mert a megjelenítő eszköz létrehozásához szükségesek a megjelenítési paraméterek is: melyik VGA legyen kiválasztva, milyen DirectX támogatás érvényes (hardveres/szoftveres), Windows ablakkezelő és a `D3DPRESENT_PARAMETERS` struktúra, mely a hátsó pufferrel kapcsolatban is tárol információt. Azonban annak a struktúrának a feltöltéséhez elengedhetetlen lekérdezni, hogy például a rendszer, illetve adott VGA + monitor támogatja-e például az 1920*1080-as felbontást 75Hz-es frekvenciával. Az ilyen, és ehhez hasonló lekérdező függvényeket tartalmaz az `IDirect3D9` interfész, mely létrehozásához csupán az alkalmazott DirectX SDK verzió kell, amit a

`D3D_SDK_VERSION` makrokonstans biztosít. Ha sikerült létrehozni ezt az interfészt, akkor egy ciklussal végigkérdezgetem az összes támogatható videómódot, majd a támogatottak paramétereit egy körkörös láncolt listában gyűjtöm össze. A beállítások menüben ennek a listának az elemei láthatóak, amik között a jobb/bal nyíl billentyűkkel lehet váltogatni. A rendszer ezen verziója sajnos nem biztosít lehetőséget a dinamikus változtatásra, így ha új eszközt szeretnék létrehozni új beállításokkal, akkor az összes előzőleg létrehozott erőforrást (textúrák, vertex pufferek) törölni kell, majd újból felépíteni őket.

Pályakészítés

Szerintem vitathatatlan, hogy egy jó játék alapját képezik a megfelelő részletességgel kidolgozott pályák. Ebben a játékban azonban nem nagyon lehet a részletekben elveszni, mint ahogy az ehhez hasonlóakban sem. Az általános megközelítés sajnos az, hogy a fejlesztők kézzel „megépítenek” pár (néhány tíz) pályát, majd ezeken játszhat a játékos. Szerintem ettől unalmassá válhat egy játék, arról nem is beszélve, hogy kitartó próbálkozással meg lehet sejteni az a lépéssorozatot (taktikát), ami sikerre viszi a játékost. A saját programomban ezért alapkövetelménynek tűztem ki, hogy nem legyenek konkrét, „beégetett” pályák, hanem minden alkalommal új játéktéren próbálhassa ki magát a játékos. Ugyanakkor persze szerettem volna bizonyos korlátok közé szorítani a változatosságot, hiszen az nem olyan jó, amikor folyton olyan problémákat kell megoldania az embernek, amik esetleg már inkább fejfájósak, mint szórakoztatóak. Ezért aztán a játék nehézségi szintekre van osztva, ami a játék előre haladtával egyre bonyolultabb kihívások elé állítják a játékost. A nehézségi szint leginkább a szigetek számával fogható meg a programban. Például a 10-es érték azt jelenti, hogy 10, 11, 12 vagy 13 sziget lesz. A szigetek számából pedig következik a kihelyezett akadályok száma. Természetesen az akadályokat is rangsorolni kellett. Nyilván, a statikus akadálymezőt képező gombák előbb jelennek meg az alkalmazásban, mint a vaddisznók. Nagyon nagyszámú szigetvilág esetén pedig már lehet kombinálni az akadályokat, illetve növelni a disznók számát. Mindent egybevetve, a végeredménynek egy olyan adathalmaznak kell lennie, amit képes megérteni a kirajzoló, valamint a vaddisznókat vezérlő program és ezen túl még megfelelő kihívást is jelent a játékos számára. Az egész probléma megoldását a MapMaker függvénycsoport biztosítja,

mely dll formájában került a szakdolgozatba. Ennek az az előnye, hogy bármikor lecserélhető a teljes program újrafordítása nélkül.

A probléma megoldása a véletlenszám-generálásokkal indul. A szigetek helyzete azonban nem mindegy, egyrészt nagyjából középen kell lenniük, másrészt nem lehetnek túl messze egymástól, hiszen ekkor csak kőhidakkal lehetne összekapcsolni őket, ami nagyban leegyszerűsítené a piros gombák összegyűjtögetését. Így a szigeteket egy függvény nagyjából kör alakban próbálja meg elrendezni a középpont körül, néhol kihagyva egy-egy helyet, néhol pedig csúsztatva valamelyik irányba. Ez után – mivel azért semmit se érdemes teljes egészében a véletlenre bízni – megszámolja a kihelyezett szigetek szomszédjait, majd minden lehetséges irányból összeköti őket – ügyelve természetesen a kereszteződések elkerülésére. Ha ez megvan, akkor további hidakat töröl, részben ügyelve arra, hogy maradjanak úgynevezett forgalmas csomópontok, részben pedig véletlenszerűen. Az egész végén még egyszer lefut egy ellenőrzés, ami azt is megvizsgálja, hogy valóban bejárható-e az összes csomópont. Eddig a pontig a kézenfekvő reprezentációnak a 7*7-es 2 dimenziós tömb viszonyult. Ezután viszont át kell konvertálni a meglévő pályacsonkot olyan reprezentációvá, amiben megfelelő gyorsasággal lehet keresni, illetve ami egy megfelelő és bővíthető feldolgozást tesz lehetővé.

Nekem valahogy adta magát a probléma megoldása, mivel valójában a pálya egy gráfnak tekinthető, ezért a reprezentáció is egy gráf formájában történt.

A gráf csomópontjai a szigetet leíró adatstruktúrák, élei pedig az adatstruktúrákhoz csatlakozó egyéb – kisebb – adatstruktúrák közötti pointerekkel megvalósított kapcsolatok.

Egy csomópont reprezentációja:

```
typedef struct MapPoint {
        short sn;
        short x;
        short y;
        unsigned short features;
        int anything;
        short lcnt;
        MapPoint *nextPoint;
        MapPointLink MPLs[8];
};
```

Fontos megjegyeznem, hogy a felsorolt mezők száma több lehet a kelleténél, ez azért van, mert így könyebb lesz a későbbiekben továbbfejleszteni a programot, ha esetleg valamiért szeretném, hiszen nem kell új elemeket felvennem. A másik ok pedig az, hogy több

különböző algoritmus is dolgozik a reprezentáción, így több szemszögből kellett ugyanazt az információt elhelyeznem, ami bizonyos értelemben redundanciához vezet. Ezt a redundanciát könnyen kézben lehetett tartani a fejlesztés során, valamint ez biztosította azt, hogy optimálisan lehet keresni is a gráfban, valamint szintén optimálisan lehet kirajzolni is a reprezentált elemeket.

A mezők az alábbiakat jelentik:

`sn:`

A SerialNumber rövidítésből jött, ez azonosítja a szigetet, méghozzá balról-jobbra, fentről-lefele haladva. A számozás (mint a C nyelvbeli tömböknél is) nullától indul.

`x:`

A sziget x (vízszintes) koordinátája.

`y:`

A sziget y (függőleges) koordinátája.

`features:`

Ez egy flag regiszterhez hasonló mező, bitjei számtalan különböző jellemzőt azonosítanak. A felsorolást az LSB bittől kezdem:

1. bit:

piros gomba van a szigeten

2. bit:

zöld gomba van a szigeten

3. bit:

kék gomba van a szigeten

4. bit:

barna gomba van a szigeten

5. bit:

a sziget zsákutca

6-9. bit:

fenntartott

10. bit:

startpont, illetve rajta volt vagy van vagy lesz a játékos

11. bit:

a vaddisznók kezdőhelye, illetve jelzi, ha van rajta vaddisznó

12-16. bit:

fenntartott

`anything:`

bármely algoritmus felhasználhatja tetszőleges célra. A legrövidebb út kereső például ezzel jelzi, hogy kiterjesztette-e a csúcsot.

`lcnt:`

kapcsolódó szigetek száma

`nextPoint:`

a szigetek mint láncolt listák is egymásra vannak fűzve, ez a következő láncszemet jeleni.

`MPLs:`

további adatstruktúra, mely a szomszédos szigetekkel biztosít kapcsolatot.

```
struct MapPointLink {
        short type;
        short length;
        unsigned int smell;
        struct MapPoint *nextMapPoint;
};
```

`type:`

Szintén egy flag regiszterhez hasonló mező, bitjei számtalan különböző jellemzőt azonosítanak. A felsorolást az LSB bittől kezdem:

1. bit:

jelzi, hogy a híd fából van

2. bit:

jelzi, hogy a híd kőből van

3. bit:

a híd éppen omlik

4. bit:

a híd leomlott

5. bit:

innen indult Elek

6. bit:

ide tart Elek

7-16. bit:

fenntartott

`length`:

a híd hossza

`smell`:

szagminta jelzésére használt érték, mely szintén 3 részre oszlik:

1-8. bit:

a barna gomba szagmintájának erőssége

9-16. bit:

Ekel szagmintájának erőssége

16-32. bit:

a malacok szagmintájának erőssége

`nextMapPoint`:

a következő csomópontra mutató pointer

Az átalakítás ezen adatszerkezet felépítésével kezdődik. A következő lépés a gombák elhelyezése. Az akadályt képező gombák részben véletlenszerűen, részben pedig aszerint súlyozottan kerülnek elhelyezésre, hogy mely szigethez mennyi szomszéd csatlakozik. A játékos általában távol a kamerától kerül start helyzetbe, a malacok pedig tőle elegendően távol ahhoz, hogy még időben meg lehessen lépni előlük. A barna gombák kihelyezésekor az algoritmusok figyelembe veszik a malacok, valamint a játékos kezdő pozícióját is, ám

leginkább az számít, hogy minél inkább „eldugott” helyre kerüljön az ilyen gomba, ami szintén a szomszédos csomópontokat vizsgálva deríthető fel.

A szomszédos szigetek sorszámozása az irányzékukkal van kapcsolatban, amit a következő táblázat mutat be:

0	1	2	3	4	5	6	7
→	←	↑	↓	↗	↙	↖	↘
→	↗	↑	↖	←	↙	↓	↘

A második sor a kétdimenziós tömbös reprezentációban mutatja az irányokat, a harmadik pedig a gráfos adatszerkezetben.

Természetesen az adatszerkezet felszabadításáról is gondoskodni kell, ez a rész a `DeleteMap` függvényben lett megírva.

Mesterséges intelligencia

Minden, a lépésajánlással, azaz a mesterséges intelligenciával kapcsolatos programrész az AI.dll-be került, melynek a MapMaker-hez hasonlóan, létezik önálló projekt változata is, ami rendkívül jó szolgálatot tett a hosszas tesztelések közben.

A programba egy egészen egyszerű „viselkedésmód” is bekerül, amit amolyan vak keresőnek lehet felfogni. Az alábbi rövid kis kódrész lényege, hogy azok közül a lépések közül, melyeket meg a szabályoknak megfelelően meg lehet tenni, sorsol egyet, majd visszatér az ajánlattal.

```
static class RandomStep {
public:
        static int GetOffer(int sn_in) {
                if (!MPList[Boars[sn_in].pos]->lcnt) return -2;
                int i=(int)((double)rand()/(RAND_MAX+1)*8);
                while (!(MPList[Boars[sn_in].pos]->MPLs[i].nextMapPoint ||
                        MPList[Boars[sn_in].pos]->MPLs[i].type & 0x0C))
                         i=++i%8;
                Boars[sn_in].pos =
                   MPList[Boars[sn_in].pos]->MPLs[i].nextMapPoint->sn;
                return i;
        }
};
```

Ennek ugyan a tényleges működés szempontjából nem sok értelme van (bár azért örültem neki, amikor először működött), ám most talán mégiscsak hasznosnak fog bizonyulni.

Ugyanis jól szembetűnik rajta az a séma, amit a többi algoritmus is követni igyekszik. Az egyes, elkülönülő algoritmusok külön osztályokban kaptak helyet, melyekben néha egyedi, csakis az algoritmustól függő adatok is helyet kaptak. Az ajánlott lépésnek minden esetben `int` típusúnak kell lennie, ami a fent vázolt táblázat harmadik sorának megfelelő jelentéssel bír. Ezeken az értékeken kívül még a -1 és a -2 lehetséges. Az előbbi azt jelenti, hogy jelen pillanatban nem sikerült lépést ajánlani a támogatott játékosnak. Utóbbi pedig azt a sajnálatos esetet közli a hívó féllel, hogy olyan rossz helyzetbe került játék közben, melyből már nem érdemes bárhova is lépnie. Egy ilyen eset lehet például az, amikor az egyik játékos elzárja magát a még teljesítendő céljaitól – például, olyan szigetcsoportra téved, melyből már nem elérhető a játékos. Ekkor a visszatérési érték hatására a program deaktiválja a pórul járt vaddisznót.

A bemeneti paraméter szintén egy egész típus, mely a segítendő játékost azonosítja. Az összes algoritmus arra lett felkészítve, hogy a malacokat kell támogatnia, ezért egy kisebb tömböt tart fenn működés közben, melybe közvetlen a pálya létrehozása után kerülnek be a lehetséges kliensek.

Ezek a kliensek bizonyos célokat is kijelölhetnek, amiket a gyors elérés igénye miatt szintén betehetnek egy gyorsító tömbbe, ahonnan bármikor, keresés nélkül elérhető az aktuális helyzetük. Erre egy tipikus példa például annak a szigetnek az azonosítója, melyen Elek tartózkodik.

Az első algoritmus egyből fel is használja a fent vázolt adatokat. Egy legrövidebb út keresőről van szó, mely az egyik legnehezebb ellenfélnek bizonyult abban az esetben, ha a színtéren már legalább két disznó volt játékban. Az algoritmus természetesen ki lett egészítve néhány elemmel. Például minden híváskor igény szerint megvizsgálja, hogy a céltárgy létezik-e még, és ha nem, akkor igyekszik egy újabb céltárgyat megjelölni a játékosnak, figyelembe véve, hogy más vaddisznónak ne legyen ugyanaz a tárgy a célja (amennyiben van elég szabad kijelölendő cél).

A folyamatos játszmák közben kiderült, hogy bár ennek az algoritmusnak a hatékonysága meglepően jó, a kinézete hagy némi kivetnivalót maga után. A probléma az vele, hogy mivel a legrövidebb utat kell megtalálni, előfordul az az eset, amikor a játékoson kívül már nem marad több kijelölendő céltárgy, és ekkor egy idő után az összes vaddisznó elkezd ugyanazon

az úton haladni a játékos felé, ami miatt annyira összecsúszhatnak, hogy képtelenség megállapítani, hányan is vannak. A játékosok szerint – köztük szerintem is – néha picit lehet rontani a hatékonyságát akár a vizuális élmény javára is, ha játékról van szó. A többi algoritmusnál ezt már figyelembe vettem.

A következő algoritmus alapötletét a mesterséges intelligencia alapjai tanórán sajátítottam el. Az említett algoritmus a minimax algoritmusnak az alfa-béta vágásos változata, ami ugye gyorsabban járja be a játékfát, mint a közönséges minimax vagy negamax algoritmusok. Amikor ezt az algoritmust választottam, leginkább a kíváncsiság vezérelt. Az előadásokon leginkább kétszemélyes, diszkrét, véges, teljes információjú, determinisztikus, zérusösszegű stratégiai játékokról volt szó. Ez a játék azonban nem teljesen ilyen. Bizonyos szemszögből tekinthető kétszemélyesnek, vagy akár diszkrétnek is, bizonyos megengedő szabályok azonban rendkívül megnehezítették az algoritmus implementálását. A legproblémásabb az volt, hogy a játékfát sajnos mindig újra kell építeni, mivel nem igazán mondhatóak az egyes játékosok lépéslehetőségei végesnek. Igaz, csak 8 irányba lehet lépni, vagy helyben maradni, de sajnos helyben maradás esetén nem lehet tudni, hogy melyik pillanatban fogja megtenni a játékos a következő lépését. A játékkal kapcsolatos összes információ sem áll a játékosok rendelkezésére, mint például, bizonyos távolságból már nem tudják a vaddisznók, hogy pontosan hol is van a játékos. Azonban a pálya megtervezése után a véletlennek nincs szerepe a játékban, így az determinisztikusnak tekinthető.
Mindezeket egybevetve, szerettem volna látni, hogy hogy teljesít egy amúgy elég jó algoritmus ezen a terepen.
Összességében az a véleményem, hogy nem valami jól. Ennek a legfőbb oka szerintem a mélységkorlátban keresendő, hiszen a játékfa a legrosszabb esetben már csupán 6 mélységben is 531441 elemet tartalmaz minden egyes támogatni kívánt játékosnál. Szerencsére az általános eset azért ennyire nem durva, hiszen a már leomlott hidakon átvágó lépésirányokra nem kell felépíteni a fát, azonban azt végig szem előtt kellett tartanom, hogy több malacnál is, az algoritmusnak észrevétlenül kell lefutnia egy közönséges notebookon is, hiszen egy játékban se jó, ha megszakad a folyamatosság.

Az éles teszteknél a lépéskorlát végül a már említett 6 mélységig terjedt, ám valamivel korszerűbb PC-n ezt nyugodtan lehet kettővel növelni.

Érdekes volt megfigyelni, hogy a kis lépéskorlát miatt a támogatott játékosok viselkedése a határozatlan emberekéhez hasonlított leginkább.

A stratégiához természetesen heurisztika is kellett, ez végül – optimalizációs okok miatt – az egyszerű euklideszi távolság lett.

Végezetül pedig, az utolsó algoritmus alapötlete a természetből fakadt, hiszen az állatok jó része nagyban támaszkodik a kifinomult szaglására. A környezet is olyan, hogy a préda akárhova is meneküljön, nyomokat hagy maga után, mely egy ragadozónak vagy egy nyomkereső kutyának leginkább a különböző szagmintákat jelenti. Így hát, követve a természetet, a reprezentációs gráf minden egyes sarkalatos pontjához hozzáadtam egy olyan mezőt, mely kombináltan képes tárolni néhány szagmintát. Fontosnak tartottam, hogy a barna gombák, a malacok, valamint a játékos mintáit meg lehessen különböztetni, míg a malacok szagmintája már nem különül el, habár erre lenne még elég hely az adatszerkezetben. A legfőbb adatforrás a hidakhoz kapcsolódó `smell` mező, mely három részre van felosztva. Az alsó 8 bit a barna színű gombák mintáinak erősségét jelzi. A rá következő 8 bit pedig a játékos mintájának erősségét.

A mintákat természetesen karban kell tartani, ez nem fix időközönként, hanem az egyes lépésajánlatok előtt történik meg. Amennyiben mondjuk eltűnik az egyik barna színű gomba valamelyik szigetről, akkor a szagminta is lassan elvész, ám remekül látszik, hogy amíg a minta valamilyen erősségben jelen van, addig ez elég jól össze tudja zavarni az ellenséges malacokat.

A játékosra ugyanez vonatkozik. A követhetőséget maga a játékos idézi elő saját mozgásával, mivel amelyik szigeten éppen jelen van, onnan indít egy rekurzív függvényt, mely elterjeszti a környező szigetekre – bizonyos mértékben természetesen csökkenő erősséggel – a szagmintát.

```
static void SpreadElekSmell(int pos, int strong) {
	int as;
	for (int i=0; i<8; ++i) {
		if ((MPList[pos]->MPLs[i].smell & 0x0000FF00) < strong) {
			MPList[pos]->MPLs[i].smell &= 0xFFFF00FF;
			MPList[pos]->MPLs[i].smell |= strong;}
```

```
                if (MPList[pos]->MPLs[i].length && !(MPList[pos]-
>MPLs[i].type & 0x0C)) {
                        if ((MPList[pos]->MPLs[i].nextMapPoint-
>MPLs[(i+4)%8].smell & 0x0000FF00) < strong) {
                                MPList[pos]->MPLs[i].nextMapPoint-
>MPLs[(i+4)%8].smell &= 0xFFFF00FF;
                                MPList[pos]->MPLs[i].nextMapPoint-
>MPLs[(i+4)%8].smell |= strong;}
                                if (strong-33280 > 0)
SpreadElekSmell(MPList[pos]->MPLs[i].nextMapPoint->sn,strong-33280);
                        }
                }
        }
```

A végén az algoritmusnak nincs más dolga, mint összegyűjteni az aktuális szigeten lévő információkat, értékelni azt, majd megtenni a szükséges lépéseket. A természethez hűen, természetesen itt sem terjednek a végtelenségig az egyes minták. A pontos részletek a tesztelés során folyamatosan kerülnek kidolgozásra, de jelenleg mindössze 3 szigeten belül érezhető egy minta 1 terjedés esetén. Amennyiben nem sikerül lépést ajánlani a mintákra támaszkodva, akkor egy véletlenszerűsített, cirkáló jellegű mozgásba kezd az ellenfél, abban a reményben, hogy előbb-utóbb talál valami neki tetsző mintát.

Az algoritmusnak létezik kooperáló változata is. Ez abban nyilvánul meg, hogy a malacok egymást próbálják elkerülni. Ez leginkább a saját szagmintájuk terjesztésével valósítható meg, azaz ha egy adott irányba létezik elég erős szagminta, akkor az algoritmus mérlegel, hogy az említett minta, valamint a cél szagmintáját figyelembe véve vajon megéri-e ugyanabba az irányba menni, vagy esetleg a második legjobb út fog a legjobb választásnak bizonyulni a végén.

Az utóbbi két témát lefedő, dll-be forduló algoritmusok normál projekt formájában is megtalálhatóak a mellékelt CD-n. Ezek a változatok önmagukban is futtatható konzolos programok, melyek főként tesztelési, valamint hibaelemzési célból készültek.

Összefoglalás

A diplomamunkát olvasva, bizonyára jócskán maradtak homályos részek a program részleteit tekintve. A program bizonyos részei már nekem is homályosnak tűnnek így három év távlatából nézve, hiszen az első sorokat még az 1. szemeszter legvégén írtam, a források nagy részét pedig a 2. szemeszter végén, valamint az azt követő nyáron készítettem. Mindent összegezve, örülök, hogy végül sikerült befejeznem a programot, mivel régi vágyam volt már egy játékprogram megírása.

Szerencsére időközben nagyon sok dolgot sikerült tanulnom, részben az egyetemi órák alatt, részben pedig a szakdolgozat írása közben. Legjobban a C++ nyelvből szerzett ismereteimnek örülök, mivel úgy látom, hogy ez egy rendkívül rugalmas nyelv, amiben bár könnyű hibázni, és annak nem egyszer durvább következményei voltak, mint mondjuk C#-ban, ugyanakkor rendkívül hatékony és gyors kódot lehet benne írni, ami szerintem a grafikus és a valós idejű alkalmazásoknál aranyat ér. Sikerült talán az egyik legnagyobb modellező programot is megismernem és az alapjait elsajátítanom, valamint egy másik, korszerű és újabban feltörekvőben lévő hasonló funkcionalitású szoftverrel is volt némi dolgom. Megismertem egy ingyenesen használható, kimondottan shaderek fejlesztésére készített programot is, az FX Compsert, melyben a HLSL vagy CgFX effektek is kényelmesen és gyorsan programozhatóak.

Ebbe a szakdolgozatba szándékosan nem írtam bele annak a sok-sok DirectX specifikus függvénynek a szignatúráját, valamint a funkcionalitását, amiket a kódolás során nap mint nap felhasználtam, hiszen azok megtalálhatóak a dokumentációban sokkal részletesebben, mint ahogy én itt tudtam volna őket prezentálni.

A részletek szerintem mindig is a forráskódban lakoznak, azt olvasgatva rengeteg lehetőség és hiba fedezhető fel valamennyi programban. A sarkalatosabb részeket igyekeztem kommentekkel érthetőbbé tenni, valamint ahol csak lehetett, „beszédes” változóneveket használtam. Az algoritmusok leírásának egy része kézírásos dokumentum, valamint rajzok formájában van jelen.

A mesterséges intelligenciával kapcsolatos részekkel az volt a tapasztalatom, hogy nem elég egy algoritmusnak a működését vagy esetleg a pszeudokódját ismerni. Ahhoz, hogy hatékonyan lehessen implementálni egy képességet a programba, tisztában kell lenni vele,

hogy milyen környezetben fog működni a program, és kik fogják azt használni. A játékfás algoritmus például jobban tudna teljesíteni a többinél, ha mélyebbre áshatna, ám ekkor akadozottá válna a játék, így erre nincs lehetőség. Továbbá szerintem rendkívüli hangsúlyt kell fektetni az algoritmusok finomhangolására is, mivel néhány kis érték megváltoztatása akár teljesen használhatatlanná is tehetné a szimulációs algoritmust. Tapasztalataim szerint ezeknek a legmegfelelőbb finom kis értékeknek a megtalálásához viszont rendkívül sok idő szükséges, mivel a teszteléssel lehet leginkább felderíteni a beállítások hatásait, valamint a további igényeket. Ugyanakkor kétszer ugyanazt meg nem lehet tesztelni, hiszen az már nem bizonyulna éles helyzetnek a játékszabályok szerint.

A kitűzött követelmények nagy részét szerintem sikerült megvalósítanom. Olyan hanghatásokat, zenét sajnos nem sikerült az alkalmazásba ültetni, amit a legelején elképzeltem.

Remélem, hogy a program grafikai része esetleg megfelelő segédprogramot biztosít majd a jövőben másoknak is saját lépésajánló jellegű algoritmusaik fejlesztésében, szemléltetésében.

Az elkövetkező években szeretném tovább bővíteni a számítógépes grafikával, és a mesterséges intelligenciával kapcsolatos ismereteimet.

Irodalomjegyzék

Futó Iván: Mesterséges intelligencia

Nyisztor Károly: Grafika és játékprogramozás DirectX-szel

Nyisztor Károly: Shaderprogramozás - Grafika és játékfejlesztés DirectX-szel

Windows DirectX Graphics Documentation (June 2010)

Köszönetnyilvánítás:

Köszönetet szeretnék mondani témavezetőmnek, Dr. Kósa Márknak.