KUKATZ 3D

Török Attila, 2/14.A

2011/2012.

Tartalomjegyzék

Fejlesztői dokumentáció

1. Specifikáció

Szakdolgozatom témájaként egy térben játszható kukacos játék, egy úgynevezett 3D-s Snake megvalósítását választottam. Ennek, a finn gyártmányú mobiltelefonok gyárilag telepített alkalmazásai között régóta közismert játéknak az a célja, hogy egy megállás nélkül előre csúszó kukacot irányítva összegyűjtsük a pályán véletlenszerű helyre kerülő eledeleket, így pontokat szerezve, a kukacunk méretét növelve, és gyorsítva azt. Ha valamelyik falba, vagy saját magunkba ütközünk, a kukac, amivel játszunk, meghal. Több játékos esetén a rangsor a játék végéig elért pontszámok alapján alakul.

2. A projekt kezdete

Maga a projekt két éve kezdődött, egy ~~unalmas~~ átlagos programozásórán, amikor a feladatunk egy síkban játszható, egyszerűbb változat elkészítése volt WPF technológia használatával, C# nyelven. Ez nekem már nem tűnt kihívásnak, hiszen már egy évvel azelőtt is írtam hasonlót, igaz, még FreePascalban, karakteres kirajzolással, de nem volt kedvem ugyanazt a belső logikát még egyszer megírni. Gondoltam hát egyet, és jobban használható, szimpatikusabb, általam jobban ismert eszközökkel; a C++ programnyelvvel, az SFML könyvtárkészlettel, és OpenGL kirajzolási móddal elkezdtem egy térbeli kukacos játékot: ezt fejleszteni. Ám ez a játék nem csak olyan módon térbeli, ahogyan számtalan klónja, amelyekben csak a látvány 3D-s, de a játéktér változatlanul egy sík lap; hanem minden kukac 6 irányban is mozoghat egy kocka alakú pálya belsejében, mintha a földben ásnák magukat.

3. Fejlesztői eszközök

Mint azt már említettem az előző fejezetben, C++ nyelven fejlesztem a játékot. A Code::Blocks nyílt forráskódú IDE-ben kezdtem el a projektet, de idővel áttértem a CodeLite fejlesztői környezetre, mert ez utóbbi kisebb, könnyebb, gyorsabb, számomra jobban használható, és egységesebb a felülete. További nagy előnye, hogy a saját, XML formátumú projektfájljából szabványos Makefile-okat generál, és ezek használatával buildel, tehát nem szükséges a fejlesztés utáni, későbbi lefordításhoz a teljes fejlesztői környezet, elég a fordítóprogram és a forráskód mellé a generált Makefile és a make program, ami pedig minden platformon és operációs rendszeren megtalálható és használható.

A játék grafikus ablakának létrehozásához, a felhasználói interakcióhoz, és a jövőben esetleg hangokhoz, zenéhez, hálózati kommunikációhoz az SFML könyvtárkészletet használom. Azért erre esett a választásom, mert nyílt forráskódú, Windows, Linux és Macintosh operációs rendszereket is támogat, hiszen a széles körben ismert és elterjedt OpenGL grafikus rendszert használja. Jelenleg is aktívan fejlesztik, és könnyen használható, ez a könnyű használat azonban „nem veszi ki teljesen az irányítást a kezemből”. Gondolok ezzel arra, hogy, nincs saját, zárt eseménykezelő ciklusa (event loopja) (mint például a népszerű GLUT keretrendszernek, és számos GUI könyvtárrendszernek, mint a Gtk+, a Qt, vagy az FLTK), de az én rendelkezésemre bocsátja az ablak bemenetére érkező eventeket (eseményeket), és a segítségével akár aszinkron módon is lekérhetem a beviteli eszközök pillanatnyi állapotát.

Ez azért bír nagy jelentőséggel, mert a program a játéklogika frissítését fix időintervallumonként hajtja végre, a megjelenítési ütemtől (framerate) függetlenül. Így stabilabbak a számítások, kiszámíthatóbb az egész rendszer viselkedése, és a játékban zajló események sebessége nem függ attól, hogy a grafikus gyorsító milyen gyorsan képes megjeleníteni a képkockákat.

4. Megoldott problémák

4.1. Különböző kivételkezelési módok

Az általam használt SFML könyvtár Windows platformon több változatban érhető el. Az egyik változat a Microsoft Visual Studio fordítójával kompatibilis, a másik a MinGW-vel, ami a GCC fordítót használja. Én ez utóbbit használom fejlesztésre, így az ehhez megfelelő változatot kell használnom.

A windowsos fejlesztés korai szakaszában azt a problémát tapasztaltam, hogy az SFML-lel való linkelés során rengeteg „referencia nem található” hibaüzenetet kaptam. Rengeteg fejtörés, értetlenkedés és kutatás után rájöttem, hogy ezt az okozza, hogy az SFML DW2-es kivételkezelési móddal van fordítva, az általam letöltött MinGW-ben lévő fordító pedig az SJLJ (set jump, long jump) módszert használja. Így a kettő együtt képtelen együttműködni. Az egyik megoldás az lett volna, hogy az SFML-t fordítsam le újra az én fordítómmal, hiszen szabadon hozzáférhető a forráskódja, de ez sokkal nehézkesebb és bizonytalanabb lett volna, mint a másik, amely mellett végül is döntöttem, hogy egyszerűen letöltök egy olyan fordítócsomagot, ami DW2-t használ, és azzal fordítom le a programot. Így már minden rendeltetésszerűen működött.

Van viszont lehetőség arra is, hogy Linux rendszer alatt fordítsuk le úgy a játékot, hogy az majd Windowson fusson (cross-compiling). Ehhez persze speciálisan erre felkészített fordítóprogram kell, amelyből viszont nem találtam nekem megfelelőt előre elkészítve, így aztán nekem kellett azt a forrásából a kívánt tulajdonságokkal lefordítanom, nem volt más választásom.

4.2. Targetenkénti fordításkönyvtárak

Ez már elég nagy projekt volt ahhoz, hogy hosszadalmas legyen minden változtatás után kivárni az összes forrásfájl lefordulását, és bosszantó is, ha csak egyetlen fájlon végeztem egy apró módosítást. Erre szerencsére megoldást nyújt a

CodeLite által is használt make eszköz, amely csak akkor fordítja újra az adott forrásfájlt, ha az később volt módosítva, mint a belőle készült objektumkód.

Több fordítási célplatform esetén azonban az összes objektumkódot újra kell generálni, hiszen az egyik platform objektumfájljai nem használhatóak fel a többihez. Ezáltal ha a Linuxon való fejlesztés és tesztelés után Windowsra szerettem volna deployolni a programot, mindent újra kellett fordítani, azokat a forráskódokat is amelyeken nem változtattam. Ezzel ilyen esetekben egyszerre ki is dobtam a make minden előnyét.

Azonban ez a probléma csak akkor áll fent, ha ugyanabba a könyvtárba fordítom az összes targetet, hiszen ekkor van ütközés a fájlok között. Ha viszont én is alkalmazom azt a szokást, (mint a Visual Studio is), hogy minden célplatformnak egy saját, külön alkönyvtára van, ott mindig megmaradnak a korábbi fordításokból származó, megfelelő objektumkódok, így platformváltáskor sem kell a meg nem változott kódokat újra lefordítani.

4.3. StateManager

Kezdetben a program elindításakor mindenféle menü nélkül maga a játék indult el. Ez természetesen nem járható út egy valamire is való játékprogram számára, hiszen nincs lehetőség semmilyen beállításra. Ahhoz azonban, hogy különböző „állomásokat”, „állapotokat” mutathasson a szoftver, ehhez kell egy rendszer, amellyel ezeket lehet tárolni, cserélgetni, egymás fölé helyezni és a legfelsőt frissíteni, mutatni.

Én ezeket az „állomásokat” State-eknek hívom. Mindegyik egy külön osztály, amelyik a `StateBase` absztrakt osztályból származik, és a `StateManager` az ezeket kezelő keretrendszer, ami egy `Singleton`. A `StateBase` és a `StateManager` is származik az `Updatable`, az `EventHandler` és a `Renderable` (szintén absztrakt) osztályokból, így ez a váza az egész játék működésének. A fő ciklus mindaddig fut, amíg legalább egy State van a `StateManager`ben, és nagyon egyszerű:

Ha vannak eventek (felhasználó által kiváltott események), amikre reagálni kell, meghívja a `StateManager` `handle_events` függvényét azok listájával, az pedig átadja őket a legfelső, aktív State-nek.

Továbbá a logika frissítéséhez fix időközönként meghívja az `update` függvényét a legutóbbi hívás óta eltelt időt paraméterként átadva neki, amely szintén továbbítódik.

A megjelenítéshez pedig a `render` függvényt hívja amilyen gyorsan csak tudja (persze egy ésszerű határig, például a monitor frissítési sebessége, körülbelül 50-60Hz), így kéri meg a legfelső State-et, hogy mutassa magát.

4.4. Erőforrások csomagolása, tárolása és betöltése

Az összes betűtípust, eledeltípust, textúrát, modellt és animációt egyetlen fájlból, a resources.pak-ból tölti be a program. Ennek formátuma többször megváltozott, ahogy fejlődött a program és egyre több funkcióval bővült, egyszer pedig a tömörítési eljárást is lecseréltem. Egy külön projekt, a resource_packer (ezt tartalmazza a játékom forráskódja) hozza létre ezt a fájlt, egy egyszerű szintaxisú szöveges listafájl, a resources.lst alapján. Lényegében csak mindent, amit ott felsorolunk, a megfelelő formába összepakolja, betömöríti a Google Snappy nevű könyvtárával, és a lemezre írja. Továbbá ez olvassa be a lassan értelmezhető, szöveges formátumú .obj fájlokat, rendezi a modellek adatait egy-egy nagy VBO-ba (vertex buffer object) és IBO-ba (index buffer object), ami megfelelő a GPU-nak, és adja meg, hogy melyik modell mettől meddig található az IBO-ban. Ezáltal elég a grafikus gyorsító memóriájába egyszer feltölteni azokat, így gyorsabb lesz a renderelés és a játék indulása is.

4.5. A betűk fekete kerete

Bár az SFML támogat szövegkirajzolást tetszőleges TrueType betűtípusfájl betöltésével, számos korlátozása miatt nekem nem volt megfelelő. Például, mint ahogy láthatjuk, a betűknek vastag fekete kerete van, ami nem színezhető át, a kitöltésük pedig tetszőleges színű lehet. Az SFML nem támogatja a betűk körvonallal való kiegészítését, én pedig mindenképp szerettem volna. Persze

részleges megoldásokat el tudtam érni hackelésekkel, de hosszú távon semmiképp sem lenne karban tartható egy ilyen kód. Továbbá az SFML alapjában véve síkbeli alkalmazásokra van tervezve és felkészítve, ezért a játékosok neveinek térbeli elhelyezéséhez is különböző kerülő megoldásokat kellett volna alkalmaznom.

Ezért arra kényszerültem, hogy saját szövegmegjelenítést írjak. Ez a forráskódban a `Font`, `String` és `GUIString` osztályokban található meg.

Először úgy terveztem, hogy fix szélességű karaktereket fogok alkalmazni, mert ehhez nem kell semmilyen egyéb információ, és a hosszadalmas, nehézkes keretrajzolástól eltekintve egyszerű lett volna előállítani a fontmapet is, viszont az azonos szélességű betűk nehezebben olvashatóak, és kevésbé szépek.

Hogy ezen segítsek, jobb megoldást kerestem, és rátaláltam a BMFont alkalmazásra, amely különféle, könnyen elhasználható fontmapeket tud generálni a TrueType betűtípusokból, akár változtatható vastagságú körvonalat is automatikusan hozzájuk adva, amely nekem alapkövetelmény volt. Továbbá biztosít a kép típusú fontmap mellé egy (akár bináris, akár szöveges formátumú) leírásfájlt (én az előbbit használtam, hisz gyorsabb a feldolgozása és kisebb is), amely különféle hasznos kiegészítő információ (méret, körvonal, kerning párok, képformátum, stb.) mellett azt tartalmazza, hogy melyik karakter hol található a textúrán, és mekkora.

Ezt a bináris leírást érintetlenül tömörítem bele a resources.pak fájlba, és a program az indulásakor értelmezi. A fontmapet pedig ugyanúgy töltöm be, mint bármelyik másik, átlátszósággal rendelkező textúrát.

Ezen adatok alapján némi utánanézéssel már egyszerűen megoldható volt a karakterek egymás után vágása, így szöveg alakítása. A játéktérbeli névkijelzéshez rendelkezésre áll a billboard változó, amelynek igaz értékekor a `String` osztály a kirajzolása előtt a modellnézeti mátrixot úgy alakítja, hogy a szöveg mindig a kamera síkjával párhuzamosan álljon, legyen bárhova is pozicionálva a világtérben, vagy bármekkorára méretezve, a `GUIString` osztály pedig egyesíti

a szövegkirajzolást a GUI alrendszerrel, hogy a menükben egyszerűen lehessen méretezni és igazítani a feliratokat.

4.6. A kukacok felépítése

Ahogyan a naiv programozó gondolná (és gondoltam én is anno), hogy a kukacok testelemeinek adatait (a helyét és a hozzá tartozó modellt/animációt) egy tömbben kell tárolni. A tömbök pedig folytonosak a memóriában, ami növelheti a rajta való végigiterálás sebességét a cache-barát elrendezés miatt, problémát okoz akkor, ha módosítani szeretnénk a kukacot. Márpedig minden lépéskor szeretnénk egy elemet hozzáadni az elejéhez (mert előre csúszott a kukac), és ha épp nem kell növekednie, egy egységet eltávolítani a végéből. Hogy az előbbit megtegyük, egy átlagos tömbben minden lépéskor egyel arrébb kell másolni az (majdnem) összes testrész adatait a tömbben, az pedig egy hosszabb kukac esetében túl sok művelet ahhoz, hogy ilyen gyakran végrehajtsuk.

Alternatíva a tárolásra a láncolt lista. Ekkor ugyan a memóriában szétszóród(hat)nak az elemek, így megszaporodnak a cache miss-ek, ami iteráláskor sebességcsökkenést eredményezhet, viszont a léptetés, és egyéb módosítás konstans komplexitásúvá válik, ami - főleg hosszabb kukacok esetében - többszörösen megtéríti az időveszteséget.

A test felépítéséhez szükséges animációk meghatározására az adott testrész, illetve az eggyel, kettővel előtte, és eggyel utána lévő részek pozícióját vizsgálom, ezek egymáshoz való viszonyából állapítom meg, valamint hogy épp egyenes vagy kanyar-e a jelenlegi testrész, hogy az előtte lévő kanyar lesz-e, és hogy mekkora az általuk bezárt szög.

4.7. Animációk

Az új fajta kukacfelépítéshez elengedhetetlen animációk megvalósítási módját több lehetséges módszer közül választottam ki. Az egyik a csontváz alapú (skeleton) animáció. Ezt, bár újabb hardveren lehetséges, az én megcélzott platformomon nem lenne célszerű a videokártyán számítani, továbbá nem is felel meg teljesen az igényeimnek.

A választásom egy másik (régebbről eredő) elterjedt módszerre, a kulcs alapú, úgynevezett keyframe animációra esett. Ennek az a lényege, hogy több, ugyanolyan topológiájú modell között alkalmazunk (jelen esetben teljesen lineáris) csúcspontonkénti interpolációt az idő függvényében, ezzel mozgatva azt. Ezt sokkal egyszerűbb megvalósítani egy pofonegyszerű vertex shaderrel is. Persze nem lenne ilyen pofonegyszerű az a shader, ha végezne ezen kívül csúcspontonkénti világítást (színszámítást) is, de jelen esetben nem hátrány számomra, hogy nem végez, hisz épp ezt a hatást próbáltam már korábban is elérni az alapértelmezett shaderrel, különböző fényforrásokkal. Ehhez így már nem kell külön erőfeszítéseket tenni, mert egész egyszerűen kikerült a világítás a renderelésből.

A trükk abból adódik, hogy egy kernel egyszeri futáskor csak egy csúcspont adatait kaphatja meg, egy másikat nem csatolhatok mellé. Persze egy kis trükközéssel dehogynem. A trükk az, hogy van lehetőség egyéb attribútumokat megadni minden vertexhez, ráadásul akár olyan tömbökből is, mint amilyen az általam használt VBO.

Ehhez a `glEnableVertexAttribArray[ARB]` és a `glVertexAttribPointer[ARB]` függvényeket kellett használnom, hogy ugyanazon meglévő buffert, amelyik az interpoláció első, kezdő modelljének adatait szolgáltatja, szolgáltasson még 2 egyéb attribútumot, amik pedig a következő keymesh csúcspontjainak pozíciói és textúrakoordinátái, hiszen a normálokra nincs szükség. Hogy összepárosítsam az egymás után következő kulcsmodelleket, az első modell első IBO-beli bájtjától a második modell első bájtjáig számolt különbséget, azaz offsetet kellett kiszámolnom és az OpenGL-lel közölnöm. Ez Linux rendszeren tökéletesen működött abban az esetben is, ha a cél modell van előbb a bufferben, mint a forrás. Windowson viszont ilyenkor felvetődött az a probléma, hogy az offset csak pozitív lehet. Ehhez alkalmazkodhatok úgy, hogy minden animáció minden keymeshét az időbeli sorrendjében töltök be az IBO-ba, viszont az ismétlődő animációk (mint a Pacman

típusú fej) esetében ez nem lehetséges, hiszen egy modell kétszer szerepel benne. Ekkor kerülő megoldásként nem tehettem mást, mint azt a modellt a bufferben is duplán szerepeltetem, amely ugyan (igaz, meglehetősen csekély) memóriapazarlás, de sajnos nincs más megoldás.

Ez a megoldás egy olyan korlátozást is bevezet, hogy a kívánt eredmény elérése érdekében az animáció minden kulcsmodelljének azonos számú, sorrendű és topológiájú vertexből kell állnia. Szerencsére az általam használt Blender program Wavefront .obj fájlexporter és -importer scriptje biztosít lehetőséget arra, hogy megtartsa a modelleken belüli sorrendet.

Az azonos számú és összeköttetésű csúcspontok korlátja pedig nekem nem okoz gondot, hiszen ennek minden animációm megfelel, és nem is éreztem szükségét ettől eltérőek megalkotásának.

4.8. Fejlécsorrend-konfliktus a GLee és az SFML között

A helyes működéséhez a GLee fejlécének mindenképpen az OpenGL fejlécek előtt kell szerepelnie. Azonban ha az előtt már beágyaztuk (include-oltunk) egy SFML fejlécet, már nem tud működni, mivel az tartalmazza az OpenGL-t is. A sokrétű osztályszerkezet és fájlhierarchia miatt nehézkes volt ezt megoldani, így szükségesnek láttam az opengl.hpp nevű, saját headerem megalkotását, amelyet minden SFML fejléc előtt be kell ágyazni, így az teljesíteni tud minden sorrendbeli követelményt, és definiálni tudja a linuxon a GLee helyett szükséges `GL_GLEXT_PROTOTYPES` makrót.

4.9. Magyar ékezetes betűk támogatása

Számos szoftvernek nehézséget okoz a nem angol (ékezetes, több bájton ábrázolható) karakterek megfelelő kezelése, megjelenítése. Én, mivel magyar nyelvű a szoftver, ezt semmiképp nem engedhettem meg magamnak, alapvető követelménynek tartottam nyelvünk minden betűjét egyenlőnek tekinteni.

Tehát a régi karakterkódolási módok (ASCII, ANSI, és az ISO változatok) semmiképpen nem voltak opciók, helyettük a modern, „mindent” tartalmazó Unicode táblát használom. Ezen belül is a többféle ábrázolási módszer közül a

legegyszerűbbet, az UTF-32-t választottam a fájlban és a memóriában is a szövegek tárolására. Persze az UTF-8 általában kevesebb helyet foglal, meggyűlhetett volna a bajom a változó hosszúságú karakterkódokkal, ezt pedig el akartam kerülni, hisz itt nincs szó olyan hosszú karakterláncokról, amelyeknél jelentős helymegtakarítást hozna az UTF-8 kódolás.

Rengeteget segített az SML beépített, könnyen használható Unicode támogatása, amellyel nagyon egyszerűen alakíthatom a szövegeket a különböző ábrázolási módok között.

A játék mindenhol használt betűtípusa, az interneten talált, szabadon felhasználható, „#44v2” nevű font azonban nem tartalmazta az „ő”, sem az „ű” betűformákat, ezeket nekem kellett utólag, a FontForge programmal hozzáadnom, majd ebből a kiegészített változatból legenerálnom a fontmapet. Szerencsémre a BMFont is akadálytalanul elboldogul az Unicode kódtáblával.

4.10. Befordított 3D kamera

A 3D renderelés megvalósításához a két szemnek szánt különböző kép előállításához „hagyományos” kamerát egyrészt el kell tolni oldalirányba, másrészt oly módon kell a látómezejét kialakítani, hogy egy meghatározott „képernyősíkban” egyáltalán ne legyen eltolódás (ezt érzékeljük majd úgy a képzetes térben, mintha a valóságban a képernyőnk helyén lenne), előtte és mögötte pedig ellentétes irányú legyen.

Az első, naiv megvalósításban ezt egy egyszerű, függőleges tengely körüli elfordítással tettem meg. Ez, bár alapvetően jó közelítés, számos zavaró pontatlanságot okoz, hiszen az elforgatás miatt a két szem számára előállított két külön kameraállapotban nem esnek egybe a képernyősíkok, sem a közeli- és távoli vágósíkok, hanem azok is kissé el vannak fordítva.

Hogy ezt kiküszöböljem, aszimmetrikus gúla formátumú vetületet kellett képeznem, amelyre az addig használt `gluProjection` függvény nem alkalmas, helyette a `glFrustum` függvényt kellett használnom, de ez utóbbi teljesen más paramétereket vár, így ahhoz, hogy elérjek egy, a `gluProjection`

eredményével egyenértékűt, további aritmetika szükséges, de az így kapott értékeket már egyszerűen korrigálhatom úgy, hogy azok már a kért, pontosan megfelelő vetületet eredményezzék mindkét szem számára.

4.11. A két szem képének egyesítése

Miután az anaglif módszer egyik (a legelterjedtebb) változatához a cián és a vörös színt (amik épp egymás komplementerei, és az RGB ábrázolásban szélsőségek) kell kivonni a két külön képből, az első megoldásban egyszerűen a `glColorMask` függvénnyel beállíthattam, hogy az egyik kép rajzolásakor csak a piros komponens kerüljön a képernyőbufferbe, majd a képet nem törölve, a másik képet is kirajzoltam, annak csak a zöld és kék komponenseit meghagyva.

Ez megfelelően működött, de mivel egy elavult megoldás, nem volt megfelelő a sebessége, és az a hatalmas hátránya is volt, hogy nem lehetett volna finoman bekalibrálni a kivont színeket az adott színszűrő szemüveghez, hanem csak a teljesen piros és teljesen cián volt használható. Persze választhattuk volna a zöldet vagy a kéket, és komplementerét is, de az még így sem teljes finomhangolás, csak alapvető színválasztás.

Később pedig, mikor megismerkedtem a képkockabufferek (FBO – frame buffer object) és fragment shaderek használatával, adta magát a jelenlegi megoldás: A két képet két külön FBO-ba rajzolom ki (és hogy közben egy kis memóriát takarítsak meg, egy közös Z-buffert alkalmazok, amelyet a két fázis között kiürítek), majd ezeket samplerként átadom a megfelelő shadernek, hogy a végleges, képernyőre kerülő képet az alkossa meg belőlük. Ehhez szükség van egy teljes képernyős négyszög kirajzolásához, pusztán azért, hogy a shader az összes fragmenten lefusson. Így már szinte végtelenek a kombinálási lehetőségek, mindegyikhez csak egy külön shadert kell írni, de alapvetően a képalkotás menete megegyezik minden (akár jövőbeni) megjelenítési módban.

Ezzel lehetőségem nyílt az anaglif módban a kivont színek tetszőleges megválasztására (bár ehhez felhasználói felület még nem készült), és a képek

egymás mellé helyezéséhez sem szükséges a viewportot módosítani, mindent megold maga a shader.

4.12. Láthatóságvizsgálat

Mivel a játéktéren a rengeteg objektumnak a kamerák mozgásából adódóan legtöbbször csak egy igen kicsi részhalmaza látszik, rengeteg időbe telik azon objektumok kirajzolásának megkísérlése is, amelyek egyáltalán nem látszódnak majd a végső képen. Bár a grafikus gyorsító természetesen kiszűri ezen elemeket, de egy későbbi fázisban, így az addig való eljutás (függvényhívások, OpenGL parancsok) is veszteség. Nem beszélve arról, hogy az poligonszinten végzi az ellenőrzést, ami sokkal nagyobb igénybevételt jelent.

Gyorsítható a renderelés azzal, hogy ismerve a kamera állását és paramétereit, valamint az összes objektum méretét és helyzetét, már a processzoron megvizsgáljuk, hogy van-e esély arra, hogy a pillanatnyi látótérbe kerül az adott objektum, és ha nem, nem is próbáljuk már elküldeni sem a GPU-nak.

Ehhez először egy nagyon lassú megoldást implementáltam, ami megkereste az objektumok AABB-jét (axis aligned bounding boxát – tengelyre igazított befoglaló dobozát), majd áttranszformálta annak mind a 8 sarkát ablaktérbe a `gluProject` függvény használatával, amihez a mátrix paramétereket a `glGet` függvénnyel kértem le. Ez alapján, ha csak egyetlen sarok is láthatónak bizonyult, megkísérelte a kirajzolást a program.

Ez a megoldás azon kívül, hogy rengeteg fölösleges `glGet` hívást és mátrixszal való vektortranszformációt vett igénybe így lassabbá tette a program futását ahelyett, hogy gyorsította volna, hibás is volt, mert a kamera valódi látómezején kívül egy másik, a képernyő síkjára tükrözött csonka gúlán belül is láthatóságot érzékelt.

Az újabb, jelenlegi, jobb megvalósításban a resource_packer a resources.pak fájlba a modell lokális koordinátarendszerbeli legkisebb köré írható gömbének adatait írja bele, amelyet a Miniball nevű fejléc használatával számol ki, hisz ez

nem olyan triviális, mint az AABB megkeresése. Dolgozni vele viszont sokkal gyorsabb, mert nem kell 8 sarkot transzformálni, csak a középpontot és a sugarat (egy sugár hosszú, egyik tengely felé irányított vektor transzformálás utáni hosszának meghatározásával, így az csak egyenletes méretezés után helyes), azokat is csak a „világtérbe”, vagyis a globális koordinátarendszerbe. És mivel már nem is transzformálok semmit teljesen ablak-koordinátarendszerbe, nincs is szükségem az OpenGL mátrixaira, tehát elkerülhetem a rengeteg lassú `glGet` és `gluProject` hívást.

Helyette az adott kamera világtérbeli adataiból meghatározom a látóterének 8 körülhatároló síkját (beleszámítva a 3D megjelenítéshez szükséges esetleges aszimmetrikus vetületet). Ezután megvizsgálom, hogy mindegyik síkon legalább egy ponton belelóg a globális térbe transzformált befoglaló kör a látótérbe, és csak akkor rajzolom ki a modellt/animációt, ha igen.

5. Amit megtanultam

Ez volt az első nagyobb projektem, és az első játékom, amely – mondhatni – ténylegesen elkészült. Rengeteg ismerettel gazdagodtam a dolgozatom elkészítse során. Megtanultam, hogyan lehet mások által elkészített könyvtárakat beépíteni a programomba, hiszen erre szükség volt az erőforrások tömörítésnél, az OpenGL kiterjesztéseinek használatánál, és a modellek köré írható befoglaló gömbök megkeresésénél.

A C++ programnyelvvel kapcsolatos tudásom is sokat bővült. Először használtam ilyen összetett és szerteágazó osztályhierarchiát, virtuális metódusokat, template-eket, a `static_cast` és a `dynamic_cast` operátort.

Az OpenGL számos funkciójával is most ismerkedtem meg. Ezek például a textúrázás, a vertex buffer objektumok (VBO, IBO) használata, képkocka buffer objektumokba (FBO) való renderelés, ezeknek egy későbbi fázisban textúraként való elhasználása (többfázisú renderelés), valamint vertex- és fragment shaderek (GLSL) írása, használata, alkalmazása.

Egyáltalán ennyire interaktív szoftvert is most fejlesztettem először, valós idejű felhasználói beavatkozási lehetőséggel, összetett belső logikával.

6. Ami mégsem került bele a programba

6.1. Hálózati játék

Korai terveim között szerepelt a hálózati játék lehetőségének beépítése a játékba, mind helyi hálózaton, mind interneten. Ehhez egy másik projektet, a netframework nevű hálózati könyvtárat kezdtem el fejleszteni, amiből szintén sokat tanultam, és egész használható állapotra jutott, de erre most nem térek ki részletesen. Ez a projekt szintén bárki számára szabadon elérhető a https://bitbucket.org/torokati44/netframework címen egy Mercurial tárolóban.

Végül időhiány miatt ez a funkció nem került beépítésre.

6.2. Instancing

Játék közben rengetegszer kell ugyanazt kirajzolnia a programnak, hiszen az eledelek, valamint a kukacok egységnyi méretű alkotóelemei típusonként mind-mind egyazon modell-textúra párosból állnak össze, csak más helyen és állásban (tehát eltérő transzformációval) kerülnek megjelenítésre.

Hasonló helyzetekre az OpenGL 3.0-s verziója támogatja az instanced renderinget, amikor is csak egyszer kell a grafikus gyorsítónak betáplálni a modell csúcspontjainak adatait, majd tetszőleges számú transzformációs mátrixot lehet küldeni neki, hogy mindegyiket alkalmazva rajzolja ki az adott objektumot, ezzel jelentős busz- és memória-sávszélességet lehet megtakarítani, a megjelenítést így gyorsítani.

Bár magától értetődőnek tűnne ennek alkalmazása, jelen esetben nem feltétlenül van rá szükség, hiszen nem túl összetett a jelenet, nincs benne túl sok objektum, ráadásul már a processzor kiszűri a nem látható objektumokat, így talán nem is lenne túl nagy teljesítménybeli javulás, ha alkalmaztam volna.

Hasznos lett volna megtanulni a használatát, de számos hátrány is származhat belőle. Például a jelenleg 2.1-es minimum OpenGL verziószámot (ami – ha nem is mindenhol, de – a legtöbb helyen már támogatott) 3.0-sra emelné, így

jelentősen csökkenne a játék futtatására alkalmas számítógépek száma. Épp ezen okból lehetőségem is kevés lett volna ennek a funkciónak az implementációjára és tesztelésére, hiszen a saját számítógépeim közül egy sem lenne alkalmas erre.

6.3. Végtelenített tér

A játék klasszikus változatában ha a pálya egyik szélébe irányítjuk a kukacot, akkor az az ellentétes oldalon fog felbukkanni, tehát effektíve végtelen a játéktér. Persze ezt ott, felülnézetből nem látjuk, csak a kukacok viselkedéséből érzékelhetjük.

Mivel jelen esetben a játékteret teljes térbeli valójában láthatjuk, meg kellett volna oldani ennek a különös fizikai viselkedésnek a látványbeli megmutatkozását is. Ezt olyan módon lehetett volna véghez vinni, hogy a játéktér falát el kellett volna távolítani, és a benne látható elemeket (kukacokat és eledeleket) még egyszer ki kellett volna rajzolni a játéktér mind a 6 oldalára, 12 élére és 8 sarkára. Ezzel összesen meghuszonhétszereződött volna a renderelési idő, és bár talán az instancing segíthetett volna ezen, fentebb tárgyalt okokból nem alkalmaztam azt.

Ezen felül a kísérleti próbaváltozat okozott gondokat a láthatóságvizsgálatnál, több megkérdezett tesztalany pedig egyenesen zavarónak találta.

7. A projekt jövője

A játékomat koránt sem tartom befejezettnek, késznek vagy teljesnek, de a célt, ami egy használható (vagy legalábbis vállalható) állapot elérése volt, úgy vélem, elértem vele. Nem is beszélve arról a rengeteg szerteágazó ismeretről és szakmai gyakorlatról, amit eddigi munkám során szereztem.

Ahogyan az időm engedi, továbbra is szeretném fejlesztgetni a programot, hogy még több tapasztalatot szerezhessek ezen a téren. Tervezem továbbá megvalósítani a hálózati játék lehetőségét mind helyi hálózaton, mind az interneten keresztül.

A belső, technikai fejlődéseken kívül szeretném bővíteni a csemegék választékát, a velük elérhető speciális hatások lehetőségeit, a választható kukacfejek skáláját.

Szeretném működésbe hozni az egyébként már létező mechanizmust akadályok kezelésére, hogy különböző akadályok, például kövek, eldobott szemét és miegymás akadályozza a szabad mozgást a játéktérben. Ehhez azonban a grafikai munkák még hiányoznak.

Szükségét érzem még egy átfogó grafikai felújításnak is, amelynek keretében átdolgoznám szinte az összes menüt, textúrát, modellt és animációt.

Célszerű lenne további térbeli megjelenítési módokat is hozzáadni, hogy a különféle technikákat alkalmazó, modern 3D tévék által feldolgozható bemeneti formátumok minél nagyobb részét legyen képes előállítani a program.

A pillanatnyi (kisebb-nagyobb) teendők listája mindenkor elérhető a forráskódtárolóban a TODO nevű szöveges fájlban.

Mivel a szöveges forráskódfájlok licenceként a nyíltnak számító és sokat megengedő zlib/libpng licencet választottam, ennek megfelelően a teljes forrást és minden tartalmat bárki számára ingyenesen és szabadon elérhetővé teszem az eddig is használt Mercurial tárolójában a BitBucket weboldalon, a https://bitbucket.org/torokati44/kukatz-3d címen.

Felhasználói dokumentáció

1. A menürendszer

Minden menüben piros betűszín jelöli a jelenleg kiválasztott menüpontot. A kívánt pontot a fel és le nyíl gombokkal egyesével lépkedve választhatjuk ki, majd az enter gombbal léphetünk bele, vagy aktiválhatjuk azt. A menükből az escape gombbal léphetünk vissza. Ahol látható egy vagy két oldalra mutató nyíl, ott a balra és jobbra nyíl gombokkal változtathatjuk a menüponton belül jelenleg kiválasztott elemet. Egeret vagy egyéb mutatóeszközt sehol nem használhatunk a menük, sem a játék vezérlésére. Az ablak bezárásakor bármikor szabályosan kilép a játék.

1.1. A Főmenü

Amint a játékot elindítjuk, a Főmenüt láthatjuk, amely a címmet és a következő menüpontokat tartalmazza: Új játék, Játékosok, Toplista, Beállítások, KreditZ, Kilépés. Mindegyik értelemszerűen működik. Az escape billentyű megnyomása kilép a játékból, bármelyik menüpont is legyen kiválasztva.

1.2. Az Új játék menü

Az Új játék menüben állíthatjuk be azt, hogy mely játékosok vegyenek részt a játékban, hány eledel legyen először a játéktérben, és legyenek-e egyáltalán, vagy milyen ütemben legyenek pótolva ahogy fogynak. A két felső beállítást a már leírt módon, a balra és jobbra nyíl gombok megnyomásával változtathatjuk.

A játékosok két oszlopba vannak rendezve. A bal oldali, „Elérhetőek:” című oszlopban a létrehozott profilok láthatóak, amiket még nem választottunk ki a jelenlegi játékhoz. A „Kiválasztottak:” alatt, a jobb oldalon pedig a már kiválasztott leendő résztvevők. Egy játékban legfeljebb 6 (legyen akár gépi vagy emberi irányítású) játékos vehet részt. A két oszlop között a nyíl gombokkal mozoghatunk, és amennyiben ez lehetséges, az enter billentyű megnyomásával helyezhetjük át a játékosokat a másik oszlopba. Addig nem indíthatjuk el a játékot, amíg legalább egy játékost nem választottunk ki játékra.

A játékosok nevei mellett balra a kis képek azt jelzik, hogy a játékos gépi (áramkör ikon), vagy emberi (bábu ikon) irányítású-e.

Ha a bal oldali oszlop mellett balra látunk kis lefele és/vagy felfele mutató nyilat, akkor a nyíl irányába tovább gördíthetjük az elérhető játékosok listáját.

A fenti két kép közül a bal oldalin láthatjuk, hogy jelenleg 3 játékosprofil létezik, de még egyet sem választottunk be a mostani játékba. A jobb oldalin a két emberi játékos, Béla és Panni a „Kiválasztottak” oszlopa kerültek, így már a játék résztvevői lesznek.

Ha a két oszlop helyett egy nagy, vörös „Nincsenek játékosok” feliratot látunk, lépjünk vissza a Főmenübe az escape billentyű megnyomásával, és a Játékosok menüben hozzunk létre profilokat igény szerint. Ennek módja a következő fejezetben van tárgyalva.

A játékot a legalsó, „Játék indítása” menüpontot kiválasztva indíthatjuk el.

1.3. A Játékosok menü

Ebben a menüben a jelenleg az adatbázisban tárolt játékosprofilokat láthatjuk felsorolva. Értelemszerűen az Új játékos menüpontot kiválasztva létrehozhatunk egy új profilt. Egy játékosprofilt kiválasztva a delete gombbal törölhetjük azt, az enterrel pedig módosíthatjuk. (Fontos megjegyezni, hogy a játékos profiljának ezen adatbázisból való törlése nem jár a Toplistán elért eredményeinek törlésével, sem semmilyen módosulásával.) Mint ahogy az Új játék menüben, itt is egy kis ikon jelzi a játékos irányításának típusát (gépi vagy emberi), és bal oldalon kis függőleges nyilak jelzik, ha tovább görgethetjük a listát valamelyik irányba.

1.4. Játékosprofilok szerkesztése

A játékosprofilok adatait rögtön a létrehozásuk után bevihetjük, később pedig módosíthatjuk ezzel a menüvel, ami a következőképp néz ki:

A legfelső sorban a játékos nevét írhatjuk be. A backspace billentyűvel karakterenként törölhetünk a név végéből. Magyar ékezeteket is gond nélkül használhatunk, és a kis- és nagybetűk között nem látszik különbség. A választott név legfeljebb 10 karakter hosszú lehet.

A második sorban azt választhatjuk ki, hogy a játékos által irányított kukacnak milyen feje legyen. Három féle stílusból választhatunk, ezek a Pacman, a Katona és az Öcsi. Közülük a Pacman fej animált, vagyis mozgás közben tátog.

A harmadik sorral állíthatjuk be, hogy ez egy gépi játékos legyen, vagy ember irányítsa. Ennek megfelelően alakul a menü alsó fele.

Ha emberi irányítást választunk, akkor a négy irányítóbillentyűt konfigurálhatjuk. Ennek a módja, hogy a beállítandó billentyűt kiválasztva nyomjuk meg az entert, ekkor a gomb megnevezése a képernyőn négy kérdőjelre változik. Az ez után megnyomott egyetlen billentyű lesz a Játékos adott irányítóbillentyűje, és ezt láthatjuk is a jobb oldalra kiírva.

Ha pedig gépit, akkor a „mesterséges intelligencia” ügyességét szabhatjuk meg egy 0-tól 10-ig terjedő skálán. A 0-s szintű gépi játékos szinte soha nem tud egyetlen élelmet sem megenni, és valós esélye van a falnak, vagy másoknak, esetleg magának ütközésre, míg a 10-es erősségű majdnem tökéletesen játszik: hamar magas pontszámot ér el, miközben igen nehéz keresztbe tenni neki. A többi érték egyenletes átmenetet képez a két szélsőség között, érdemes kísérletezgetéssel, próbálgatással beállítani a megfelelő kihívást nyújtó, de a játék élvezetét lehetetlenné még nem tevő szintet ellenfeleinknek.

Amennyiben a menüből az escape billentyű használatával lépünk ki, akkor az újonnan létrehozott játékos adatai elvesznek, és nem jön létre a játékos, ha pedig meglévő játékost módosítunk, akkor nem lépnek érvénybe a változtatások, minden marad a régiben.

Hogy az adatokat rögzítsük, a „Mentés” menüpontot kiválasztva kell az enter gombot megnyomni. Ezzel a mentés után ki is lépünk a menüből.

A „Játékos törlése” menüpont értelemszerűen törli a profilt az adatbázisból, de mint ahogy fentebb már leírtam, ez sem törli azokat a Toplistabeli bejegyzéseket, amiket ezzel a profillal értek el.

1.5. A Toplista

A Toplista egy nagyon egyszerű menü, egyáltalán nem interaktív. A mindenkori legfeljebb 10 legjobb emberi játékosok által elért pontszámot láthatjuk nagyság szerint csökkenő sorrendbe rendezve, és persze mellette azt, hogy milyen nevű játékos érte el. Gépi játékosok nem kerülhetnek fel a Toplistára.

A menüből az enter vagy az escape billentyű megnyomásával léphetünk ki. Ennél több funkciója nincs, a listát tényleges játékeredmények elérésén kívül más módon megváltoztatni nem lehet.

1.6. Beállítások

Ebben a menüben szabhatjuk testre a program működésének néhány fontosabb beállítását. A beállítások mentéséhez és alkalmazásához az enter billentyűt kell leütni, bármelyik menüpont is legyen kiválasztva. Ezzel azonban

még nem lépünk ki a menüből, azt az escape gomb megnyomásával tehetjük meg, szintén függetlenül az épp kiválasztott menüponttól. Vigyázzunk, hogy ha kiválasztottuk a megfelelő értékeket, nehogy mentés nélkül lépjünk ki a menüből, hiszen úgy nem lépnek érvénybe.

A felső sorban a képernyő felbontását (és nem teljes képernyős módban ezzel együtt effektíve az ablak méretét is) választhatjuk ki. Azonban ha nem kapcsoljuk be a teljes képernyős módot, a játék ablakát magunk is szabadon átméretezhetjük az egérrel, a játék alkalmazkodni fog hozzá, és a kívánt méretben fog kirajzolódni.

A második sorban beállíthatjuk, hogy a játék teljes képernyős módban, ablakkeretek nélkül, minden más ablak fölött fusson, vagy egy tetszőleges méretű, teljesen átlagos ablakban jelenjen meg.

A „3D” beállítással a manapság nagyon divatos térbeli megjelenítés módját konfigurálhatjuk. Ez persze nincs hatással a játéktér térbeli mivoltjára, csupán a vizuális élményre.

A „kikapcsolva” állapot egyszerű, hagyományos megjelenítést jelent.

A „kancsal” módban egymás mellé rajzolódik ki a bal és a jobb szemünknek szánt kép. A térbeli hatás eléréséhez egy, a monitornál közelebb lévő pontot mindkét szemünkkel „célba véve” kancsalítanunk kell, hogy a bal szemünk a jobb oldali képfelet lássa, és fordítva. Így a kettő középen egy harmadik, térbelinek látszó képpé egyesül. Szükséges lehet némi gyakorlás, mire sikerül, addig is célszerű a megszokottnál egy kicsit távolabbról nézni a monitort, így könnyebb

ráfókuszálni. Ez a módszer egyeseknél hosszabb alkalmazás után fejfájást okozhat, és a kép is kisebb, mint 3D mód nélkül (alul és fölül van egy-egy fekete sáv), viszont nem kell hozzá semmilyen más, különleges eszköz.

Az „anaglif” mód mindkét képből kivon egy színt (vöröset illetve ciánkéket), majd összekeveri őket. A megtekintéshez szükséges egy olcsó, piros/cián színszűrő lencséjű szemüveg. Ennek a módszernek előnye, hogy nem megerőltető, és teljes méretű képet mutat, viszont a színek kissé eltorzulnak, és esetenként egy kényelmetlen érzést, úgynevezett „retinaversengést” okozhat, mert teljesen különböző színeket lát a két szem.

A negyedik sorban azt állíthatjuk be, hogy a gépi játékosok szemszögéből is lássunk-e kameraképet, mint ahogyan a sajátunkból is látunk. Ha ez a lehetőség ki is van kapcsolva, de a játékban csak gépi játékosok játszanak, akkor is látni fogjuk a kameraképüket.

Az utolsó sorban megadhatjuk, hogy a játéktérben lássuk-e a többi kukac feje fölött az őket irányító játékos nevét.

1.7. KreditZ

Ez egy statikus menü, amelyben megemlítem a program elkészítéséhez felhasznált eszközöket, programokat, könyvtárakat és szerzőiket, valamint köszönetet mondok azoknak, akik segítségükkel jelentősen hozzájárultak az én és a játék fejlődéséhez.

Az escape vagy az enter billentyű megnyomásával léphetünk vissza a főmenübe.

2. A játékmenet

A játéktér egy 41x41x41 egységkocka méretű nagy kocka. Úgy tekinthetünk rá, mint egy föld alatti térfogat, ahol szabadon furkálhatnak a kukacok, de a talaj nem látszódik, hiszen akkor semmit nem látnánk. Nincs fel vagy le, sem semmilyen kitüntetett irány. Mindenki mehet amerre lát, egészen a „falakig”, amik a játéktér határai.

A kukacok a kocka alakú játéktér 6 lapjának közepei közül véletlenszerű helyről indulnak, véletlenszerű fejállással. Kezdetben minden kukac 4 egység hosszú, 4 kocka/másodperc sebességgel halad.

Minden játékos a saját profiljában beállított gombokkal irányíthatja az állatát. Csak a hat „kijelölt” irányba lehet haladni, minden gombnyomással a következő rácsponton egy derékszöget elfordulva.

A cél minél több étel elfogyasztásával (a kukac fejének nekivezetésével) a legtöbb pontot összegyűjteni.

A játékot a „P” gomb megnyomásával bármikor szüneteltethetjük, majd ugyanígy folytathatjuk. A szünet ideje alatt minden játékos kameráján középen a „SZÜNET” felirat látható nagy, fehér betűkkel.

2.1. A kamerák képernyőn való elrendezése

A kamerák elrendezése a képernyőn attól függ, hogy hányan játszanak egyszerre, valamint hogy a gépi játékosok kamerája látszik-e. Tehát lényegében attól, hogy hány kamera képét kell megjeleníteni.

- Ha csak egy megjelenítendő kamera van, az természetesen kitölti az egész ablakot.
- Ha kettő, akkor egymás mellett fognak elhelyezkedni, ha a játék ablaka szélesebb, mint amilyen magas, és egymás fölött, ha fordítva.
- Ha négy, akkor egyszerűen négy egyforma negyedre oszlik a képernyő.
- Ha hat, akkor három oszlopba és két sorba rendeződnek, ha az ablak szélesebb, mint amilyen magas, két oszlopba és három sorba, ha fordítva.

A felsorolásból kimaradt a három és az öt kamerás eset. Ilyenkor egy úgynevezett „kiegészítő” kamera kerül be utolsó helyre, hogy egyenlő részekre lehessen osztani a játék ablakát. Ez a kamera mindig a jobb alsó sarokban van, és rajta az összefoglaló felületen egy táblázatban láthatjuk soronként az össze játékos nevét, valamint minden adatát, amit a játékosok saját felülete is közöl, ebben a sorrendben: pontszám, hosszúság, sebesség. A név színe itt is piros színnel jelzi, ha a kukac még életben van, és szürkével, ha már meghalt.

Ez után a 4, illetve 6 kamerás felosztási mód lép érvénybe.

A kamerák elhelyezése (beleértve a szélességhez és magassághoz kötött eloszlást) az ablak játék közbeni átméretezésével frissülnek. Ilyenkor érdemes

megállítani a játékot, hogy a felületen látható név alapján minden játékos újra megtalálja, hogy hol láthat a saját kukacának szemszögéből.

2.2. A játékosfelület elemei

Minden játékos kamerájának bal felső sarkában látunk egy lekerekített, elmosott fekete keretű, félig átlátszó, szürke téglalapot. Ebben a következő információkat olvashatjuk: a játékos neve, jelenlegi pontszáma, kukacának hosszúsága, valamint sebessége.

A játékos neve piros, ha még életben van a kukaca, és szürke, ha már nem.

Amikor megeszünk egy csemegét, amitől megnő a kukacunk, a hosszúságot két részletben láthatjuk, például valahogy így: 4+ 16. Az első szám azt jelzi, hogy hány egységet fog még ezután nőni, és a második a pillanatnyi hossz. Ilyenkor a kukac farka vége egy helyben marad, és ahogy a feje csúszik előre, úgy nő. Erre azért van szükség, mert ha azonnal szeretnénk növelni a fej mozgását nem befolyásolva, a farkat kellene hátratolni, de nem tudhatjuk, hogy erre van-e elég hely a kukac háta mögött, vagy nincs-e ott valamilyen akadály.

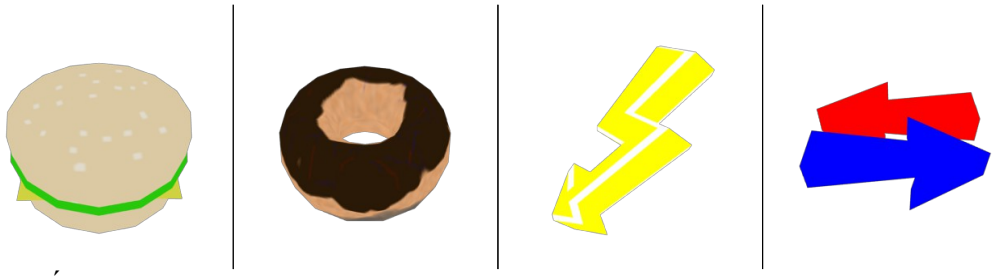
2.3. A kiegészítő kamera mozgása és összefoglaló felülete

A kiegészítő kamerára – amint azt már említettem – akkor van szükség, ha az egyéb kamerák száma 3 vagy 5. Ez a kamera a játékban részt vevő kukacok fejeinek pozíciója alapján mozog, úgy, hogy egy átfogó képet próbál mutatni a játékmenetről. Ennek technikai részleteit most nem ismertetem részletesen, de mindig 3 kukacot választ ki véletlenszerűen, és ezeket próbálja a képernyőn tartani, attól egyenlő távolságra.

A rajta látható úgynevezett összefoglaló felület öt, fekete vonallal elválasztott sorra van osztva. Itt láthatóak a játékosok, pillanatnyi pontszámuk szerint csökkenő sorrendbe rendezve. Minden ilyen sorban egy-egy kukac adatait láthatjuk. Ezeken belül középen felül a játékos neve, alul pedig bal oldalt a jelenlegi pontszáma, középen kukacának hossza, jobb oldalt pedig sebessége látható. A játékos nevének színjelölése és a hosszúság megjelentési módja megegyezik a játékos felületéével.

2.4. Eledeltípusok

Kukacainkkal négyféle eledelre vadászhatunk. Ezek a hamburger, a fánk, a villám és a fordító.



Általános tulajdonsága mindnek, hogy pontokat kapunk megevésükért, egy adott egységhosszal megnövelik kukacunk testét, és valamennyi kocka/másodperc sebességet hozzáadnak a gyorsaságához, hogy egyre nehezebb legyen a testek között navigálni és a kukacokat pontosan irányítani. Mindegyik étel különböző relatív gyakorisággal fordul elő, más-más pontszámot ér, és eltérő tulajdonságokkal rendelkezik. Ezek a következők:

Név	Gyakoriság	Pontszám	Növelés	Gyorsítás
Hamburger	1	10	4	0.1
Fánk	2	8	5	0
Villám	0.5	20	0	0.5
Fordító	0.1	5	0	0

Mint láthatjuk, a fordítótól se gyorsabb, se hosszabb nem lesz a kukacunk. Van viszont egy különleges, egyedi tulajdonsága: Ha megesszük, az összes többi kukac azonnal megfordul, hirtelen hátraarcot csinál.

2.5. A játék vége

A játék akkor ér véget, ha a pályáról az összes eledel elfogyott, vagy ha az összes kukac meghalt. Persze mindig befejezhetjük a játékot az escape gomb megnyomásával.

Ilyenkor a következő képernyőt láthatjuk:

Nem látható rajta más, mint a játékban részt vevők neve és a játék alatt összesen elért pontszámuk, ez utóbbi alapján csökkenő sorba rendezve.

Ha volt olyan emberi játékos, aki felkerült a Toplistára, akkor az enter vagy az escape gomb megnyomására a Toplista menübe kerülünk, ha nem volt, akkor a főmenübe.

Felhasznált irodalom

- A C++ programozási nyelv I.-II. kötet
  (Bjarne Stroustrup, ISBN 9639301191)
- OpenGL röviden
  (Paul Martz, ISBN 9789639637252)
- View Frustum Culling » Lighthouse3d.com
  http://www.lighthouse3d.com/tutorials/view-frustum-culling/
- Efficient View Frustum Culling
  http://www.cescg.org/CESCG-2002/DSykoraJJelinek/index.html
- Point-Plane Distance -- from Wolfram MathWorld
  http://mathworld.wolfram.com/Point-PlaneDistance.html
- Sam's Blog – OpenGL VBO Tutorial
  http://sdickinson.com/wordpress/?p=122
- SFML – Simple and Fast Multimedia Library
  http://sfml-dev.org/documentation/1.6/
- GLSL Tutorial
  http://zach.in.tu-clausthal.de/teaching/cg_literatur/glsl_tutorial/index.html
- OpenGL @ Lighthouse 3D – View Frustum Culling Tutorial
  http://zach.in.tu-clausthal.de/teaching/cg_literatur/lighthouse3d_view_frustum_culling/index.html
- View frustum culling « The ryg blog
  http://fgiesen.wordpress.com/2010/10/17/view-frustum-culling/
- NeHe Productions: GLSL: An Introduction
  http://nehe.gamedev.net/article/glsl_an_introduction/25007/
- OpenGL Transformation
  http://www.songho.ca/opengl/gl_transform.html
- Tutorials
  http://www.opengl.org/sdk/docs/tutorials/ClockworkCoders/attributes.php
- Common Mistakes – OpenGL.org
  http://www.opengl.org/wiki/Common_Mistakes
- Anaglyphs
  http://paulbourke.net/texture_colour/anaglyph/
- Frame Buffer Objects (FBO) | OpenTK
  http://www.opentk.com/doc/graphics/frame-buffer-objects
- Image processing with FBO and GLSL - OpenGL - GameDev.net
  http://www.gamedev.net/topic/601413-image-processing-with-fbo-and-glsl/
- Reading FBO-Attached Depth Texture in Fragment Shader – OpenGL – GameDev.net
  http://www.gamedev.net/topic/560589-reading-fbo-attached-depth-texture-in-fragment-shader/
- General OpenGL: Transformations – OpenGL.org
  http://www.opengl.org/wiki/General_OpenGL:_Transformations#How_do_I_draw_a_full-screen_quad.3F
- flipcode – Frustum Culling
  http://www.flipcode.com/archives/Frustum_Culling.shtml

- GLSL Sampler – OpenGL.org
  http://www.opengl.org/wiki/GLSL_Sampler#Binding_textures_to_samplers
- The graphics blog » Simple FBO rendering
  http://www.flashbang.se/archives/48
- Framebuffer Object – OpenGL.org
  http://www.opengl.org/wiki/Framebuffer_Object
- OpenGL Frame Buffer Object (FBO)
  http://www.songho.ca/opengl/gl_fbo.html
- View Culling
  http://www.cbloom.com/3d/techdocs/culling.txt
- Game Development – Stack Exchange
  http://gamedev.stackexchange.com/
- c++ - Frustum Culling with VBOs - Game Development - Stack Exchange
  http://gamedev.stackexchange.com/questions/2217/frustum-culling-with-vbos
- opengl - Manual GL_REPEAT in GLSL - Stack Overflow
  http://stackoverflow.com/questions/4221963/manual-gl-repeat-in-glsl
- OpenGL Toolkits (e.g. GLUT, GLFW, GLM) - OpenGL.org Discussion and Help Forums
  http://www.opengl.org/discussion_boards/ubbthreads.php?ubb=postlist&Board=10&page=1
- glGet Alternative? - OpenGL.org Discussion and Help Forums
  http://www.opengl.org/discussion_boards/ubbthreads.php?ubb=showflat&Number=238616
- About glGet() - OpenGL - GameDev.net
  http://www.gamedev.net/topic/503012-about-glget/
- glsl – How do you get the modelview and projection matrices in OpenGL - Stack Overflow
  http://stackoverflow.com/questions/4202456/how-do-you-get-the-modelview-and-projection-matrices-in-opengl
- The Normal Matrix » Lighthouse3d.com
  http://www.lighthouse3d.com/tutorials/glsl-tutorial/the-normal-matrix/
- GLM_GTC_matrix_transform: Matrix transform functions.
  http://glm.g-truc.net/api-0.9.2/a00245.html
- Building a Cross compiler for Windows on Linux
  http://silmor.de/qtstuff.mingw.php
- OGRE: OgreSingleton.h Source File - OGRE Documentation
  http://www.ogre3d.org/docs/api/html/OgreSingleton_8h_source.html
- BMFont - AngelCode.com
  http://www.angelcode.com/products/bmfont/
- BMFont OpenGL implementation | SourceForge.net
  http://sourceforge.net/projects/oglbmfont/
- Bernd Gärtner – Smallest Enclosing Ball Code
  http://www.inf.ethz.ch/personal/gaertner/miniball.html
- Wavefront .obj fájl – Wikipedia, the free encyclopedia
  http://en.wikipedia.org/wiki/Wavefront_.obj_file
- Object Files (.obj)
  http://www.martinreddy.net/gfx/3d/OBJ.spec

A webes hivatkozások utoljára 2012. 04. 20-án voltak ellenőrizve.