

ESZTERHÁZY KÁROLY FŐISKOLA
MATEMATIKAI ÉS INFORMATIKAI INTÉZET

Körökre osztott táblás játék modellezése

Készítette:

Gregus Tamás
programtervező informatikus BSc

Témavezető:

Dr. Kovásznai Gergely
adjunktus

EGER, 2012

Tartalomjegyzék

1. Bevezetés	4
2. Játékfejlesztő cégek és játékaik	6
2.1. Indie játékfejlesztő cégek	6
2.1.1. A hobbifejlesztés egyik nagy sikere	7
2.2. AAA kategóriás játékokat fejlesztő cégek	8
2.2.1. Egy AAA játék bemutatása	9
3. Jó játék létrehozásának titka	12
3.1. A siker titka indie-ként	12
3.2. Hogyan legyünk jók nagy költségvetésből?	15
3.3. A marketing kérdése	19
3.4. Terméktámogatás	21
4. Blender	24
4.1. Modellezés	28
4.1.1. Subdivide	29
4.1.2. Decimate	29
4.1.3. Transzformációk	30
4.1.4. Boolean műveletek	31
4.1.5. Tükrözés	32
4.2. Digitális szobrászkodás	32
4.3. Textúrázás	34
4.4. Animálás	39
4.4.1. Kulcsképkockákkal történő animálás	39
4.4.2. Importált fájl segítségével történő animálás	41
4.5. Exportálás	45
5. Unity game engine	46
5.1. Ütközési zónák	48
5.2. Scriptek	50
5.3. Kapcsolók	52
6. Táblajáték létrehozása Blender és Unity segítségével	56
6.1. Tábla elkészítése	57
6.2. Bábu létrehozása	58
6.3. Bábuk és tábla összeillesztése	60

7. A játékfejlesztés jövője	73
Irodalomjegyzék	74

1. Bevezetés

Létezik egy ősi, a földön élő lényekkel egyidős tevékenység, melyet máig előszeretettel végeznek sokszor napi szinten az emberek és az állatok egyaránt. Ez a különös szabadidős tevékenység a *szórakozás*.

Szórakozni sokféleképpen lehet. Az állatok korai éveikben a játékot a tanulás részeként alkalmazzák, mely során játékos formában sajátítják el a túlélés törvényeit. Játékuk az emberek alakította környezetük hatására az évtizedek során változásokat idézett elő ugyan, de a játékuk módja az idő előrehaladtával mégis hasonló maradt.

Ezzel szemben az emberi szórakozás tárgya szinte évről-évre folyamatosan képes változni. Ennek oka alapvetően az, hogy a gyors technikai fejlődés, ipari forradalmak mind-mind újfajta játékokat és szórakozást tettek lehetővé a kor emberének.

Térjünk vissza az időben pár évvel. Akkoriban másabb és kevesebb szórakozási formák voltak jelen életünkben. A teljesség igénye nélkül következzen pár példa: régen a játék fogalma a társasjátékokat, labdajátékokat, és egyéb szabadtéri játékokat jelentette. Rossz idő esetén könyveket lehetett bújni. Később megjelentek az első rádiók, amelyek a hírek és riportok közvetítése mellett rádiójátékokat is nyújtottak a hallgatóik számára¹. A mozgókép elterjedésével az első filmek is megjelentek, melyek a történetmesélés egy új, izgalmas formáját mutatták be a közönségnek.

1972-ben egy presszóban egyik napról a másikra egy különös gép jelent meg. Nagy képernyője volt, és egy persely lógott az oldalán. Az egyik törzsvendég odament, hogy megnézze, mi ez a szerkezet. A képernyőn egy labdát pillantott meg, ami elvissza pattogott. Bedobott egy negyed dollárost, elkezdett játszani. Rájött, hogy van két kis gomb a gépen, ami az ütőket irányítja, aminek a segítségével sikerült eltalálnia a labdát. Az idő előrehaladtával egy kisebb tömeg kezdett kialakulni a gép előterében.

Ezt a gépet az *Atari* cég állította fel, melyet Nolan Bushnell 500\$-os tőkéből hozott létre, ezzel létrehozva az első videojátékot.

Az Atari sikerét látva más hasonló cégek is létrejöttek, egy új ipart teremtve. Megjelent az első otthoni játékkonzol is, a *Magnavox Odyssey*, és ezzel a videojátékok a mindennapok szórakozásává vált. Manapság a 8-10 évvel ezelőtti asztali számítógépekre megjelent kedvenc játékainkat okostelefonokon hordhatjuk magunkkal, akár a tömegközlekedési eszközökön lehet velünk egy-egy ismertebb nagy cím.

Természetesen a régebbi szórakozási formák is megmaradtak eredeti formájukban, illetve némelyeknek létrejött „modernebb” változata is, például az *e-book olvasók*.

¹ Például Orson Welles: Világok Harca

Jelenlegi gondterhelt, rohanó világunkban még nagyobb szükség van kikapcsolódásra. Sokszor az ember szinte menekülni szeretne a hétköznapiokból egy elképzelt, idealizált világba, és erre jó példa a videojátékok által nyújtott, manapság is újszerű élmény. Nem véletlen, hogy jelenleg ez a legnépszerűbb szórakoztatóipari termék a világon.

A témaválasztás számomra nem volt nehéz döntés, mivel 6-7 éves korom óta érdekelnek a videojátékok. Napi szinten figyelemmel követem a játékipar változását, és ha van időm, emellett a technikai hátteret is biztosították, élőben figyelem a nagyobb expókat¹. Egyfajta művészi kiteljesedésnek tartom a játékokat. Tetszik az bennük, hogy egy teljesen új világba, saját szabályrendszerrel tud bevezetni, és hogy gyakorlatilag minden, ember által valaha is tapasztalt dolgot le lehet vele élethűen szimulálni, melyben nem az Olvasóra bízunk a „képzeltet”, hanem saját magunk teremtünk egy új világot. Ha emellett megtalálhatóak bennük azok a dolgok, amik igazán egyedivé teszik őket², akkor sokáig emlegethetik, nem beszélve arról, hogy nagy profitot termelhetnek.

¹ itt játékbejelentő és ismertető rendezvényekre gondolok

² ennek részletezése a [3.](#) fejezetben

2. Játékfejlesztő cégek és játékaik

Ebben a fejezetben a teljesség igénye nélkül képet kaphatunk a jelentősebb cégekről, cégtípusokról, az adott cégtípusok céljairól, általuk létrehozott népszerű játékaikról, melyeket technikailag is megpróbálok felvázolni. Megpróbálom kiemelni, hogy mely tulajdonságaik jók, rosszak a cégnek és az adott játéknak egyaránt. Bemutatom, hogy egy gazdag és egy kis költségvetésű cég hogyan próbálja megállni helyét a piacon.

A játékok iparrá válása jelentősen hozzájárult ahhoz, hogy egyre több játékfejlesztő cég jöjjön létre. Ahogy a bevezetőben az Atari példáját látták mások, és valamilyen kreatív ötletük támadt, tőke híján saját projektekbe fogtak. Ezeket *garázsprojektekné*k hívjuk. A legtöbb hobbifejlesztő a mai napig mind innen indul, és ha a szerencséje egy jó ötlettel társul, akár milliomos is lehet.

Az internet feltörekvésével természetesen ma már az ismertséget könnyebb megszerezni, viszont az elismertségig vezető utat továbbra is akadályokkal teli, rögös út övezi. A következőkben a két jelentősebb cégtípust mutatom be:

2.1. Indie játékfejlesztő cégek

Az alfejezet címét látva az Olvasóban felmerülhet, mit is jelenthet az *indie* kifejezés. Az indie az angol independent szóból származik, jelentése független. A gyakorlatban ez azt jelenti, hogy az adott cég költségvetését nem fedezi egy kiadó, hanem a készülő terméket saját maguk adják ki. Az átlagosan legfeljebb egy tucat fejlesztőből álló csapatok akár évekig fejleszthetnek egy-egy játékot, de akár pár napos projektek is készülhetnek a cég berkein belül.

Ezeket a projekteket aránylag ritkán érint a szoftverkrízis, mert jóval kevesebb veszélyes tényező léphet fel a projekt életciklusának ideje alatt. Ezek kiküszöbölésére kockázatelemzést szoktak alkalmazni. A sikertelenség okai a következők lehetnek:

Nehézségek előre nem látása:

Előfordul, hogy a hobbifejlesztők szinte a nullával egyenlő programozási tapasztalattal rendelkeznek, és naívan azt hiszik, hogy kevés munkával nem is akármilyen, hanem jó játékot fognak „összedobni”. Lehet, hogy eljutnak a tervezésig, de látva a fokozatosan felmerülő problémákat, lehet, hogy a tényleges implementáció első fázisaiban befejeződik a fejlesztés folyamata, és sikertelen projektként lehet elkönyvelni a próbálkozást. Ennek megoldásaként a tervezést az ötletek megfogalmazása után folytathatjuk az erőforrások elemzésével, vagyis ki mennyi ideje programozik, illetve milyen nyelveken, milyen fejlesztői

környezetben tud otthonosan mozogni. Ezek megbeszélése jelentősen csökkentheti ezt a kockázatot.

Nézeteltérések:

Mivel nincs külső nyomás a fejlesztőkön és saját ötleteiket valósítják meg, nem a kiadó terveit kell leprogramozni, ezért a fejlesztők abszolút szabadjára engedhetik fantáziájukat a munka során. Mivel kis létszámú csapatról van szó, valamint lehet, hogy vannak makacs, művészelkű programozók a társaságban, akik önmaguk kiteljesedését látják a készülő projektben, előfordulhat az, hogy pár embernek ellentétes gondolatai támadnak a programot illetően. Amennyiben hasonló nézeteltéréseket tapasztalunk, két választásunk van: Közös projekt lévén vagy engedjük, hogy mindenki beletegyen valamit önmagából a játékba, vagy a kezdetektől egy elképzeléshez ragaszkodjunk, amit vagy vállal az egész csapat, vagy sem.

Hibás döntések:

Előfordulhat, hogy ami papíron jól néz ki, nem ad akkora játékélményt, vagy éppen frusztráló. Emiatt ajánlott már a legelső ötletet másokkal megvitatni, hogy fény derüljön, mennyire lehet életképes egy adott ötlet. Érdemes kis, tesztre szánt programokat létrehozni, amely segítségével a felhasználók kipróbálhatják, mennyire növeli az élvezeti faktort a szóban forgó játékelem. Ezt nevezik *bétatesztnek*.

Amennyiben belevágunk egy garázsprojektbe, gondoljunk arra, hogy kockáztathatunk. Megállapíthatjuk, hogy a fenti kockázatok kis költség esetén vállalhatóak, veszteni valónk gyakorlatilag a projektre fordított időn kívül csekély, és mivel nem feltétlenül pénzt akarunk a hobbifejlesztéssel keresni, akármilyen számunkra tetsző ötletet megvalósíthatunk. Kitalálhatunk *casual* (alkalmankénti) játékosoknak szóló játékokat, melyek rövid ideig tartó, úgymond elmélyülést nem igénylő, akár röpke 5-10 perces kikapcsolódást nyújthatnak, vagy épp komoly, a „nagyokat” megszégyenítő projektbe vágjuk a fejszénket. Csupán a fantázia szabhat határt a szabadidő korlátaival kívül, és lehet, hogy sikeresek is leszünk, felfigyelhet ránk az ipar egy-egy kreatív ötletünk folytán.

Az indie fejlesztők fejleszthetnek *egyedi*, illetve *dobozos* szoftvert is. Amíg előbbi egy adott kérésre lefejlesztett játék lehet, addig az utóbbi akárkinek szólhat, szélesebb közönségre tehet szert.

2.1.1. A hobbifejlesztés egyik nagy sikere

Íme egy példa hobbiprojektként indító, végül sikeres játékfejlesztő játékára:

Minecraft:

2009 májusában alakult meg a Mojang AB, melyet a svéd Markus "Notch" Persson alapított Stockholmban, és eddig egyetlen, ámde már-már hihetetlenül népszerűvé vált játékot fejlesztett. Ez a *Minecraft*. A Minecraft egy *sandbox* stílusú játék, emellett szerepjáték elemek is megtalálhatóak benne. A sandbox stílus nem véletlenül kapta a nevét. Jelentése ténylegesen a homokozóra utal, egyfajta olyan világra, ahol a játékos azt tehet, amit akar, mivel a játék rengeteg eszközt biztosít a szabad „garázdálkodásra” a játék világában. Magát a világot tekintve az egész kockákból áll, beleértve az élőlényeket és a karakterünket is. A játékos egy véletlenszerűen generált szigeten kezdi meg kalandjait, ezen a szigeten többféle éghajlat található meg, fenntartva a környezet változatosságát. A játékosnak elsősorban fát kell szereznie, ebből készíthet léceket a későbbi építéshez, fáklyákat, illetve munkapadot, mely segítségével amennyiben rendelkezünk sematikus rajzokkal és hozzávaló nyersanyagokkal kemencét, létrát, ládát, csapóajtót hozhat létre. Szerszámokat, ruhákat illetve fegyvereket is alkothatunk, amennyiben túl szeretnénk élni az éjszakákat ezen a rejtélyes helyen.

Az éjszakák ellenfeleket teremtenek, melyek próbálnak minket megölni. Ezért érdemes egy házat építeni nappal, hogy élve megússzuk a következő nappalig. Élőhalottak, óriáspók, íjász csontvázak próbálnak az életünkre törni. Emellett létezik egy négylábú, gyors zöld teremtmény is a játékban, a Creeper (ijesztgető), amely ha közel ér hozzánk, felrobban. Az Enderman pedig az egyik legveszélyesebb, mert gyakorlatilag sebezhetetlen, és tilos ránézni, ugyanis ekkor mögöttünk jelenik meg, ami egyet jelent a végzettel.

Multiplayer (többjátékos) lehetőséggel is rendelkezik a program, mely során vagy másokkal együtt próbáljuk az említett kalandokat túlélni, vagy végtelessen nyersanyag segítségével óriási, monumentális építmények munkálataiban vehetünk részt.

Jelenleg a játéknak 21 millió regisztrált felhasználója van, ebből közel 5 millióan vették meg az egyébként 20 euróba kerülő játékot.

2.2. AAA kategóriás játékokat fejlesztő cégek

Az AAA kategória, másnéven *Triple-A*, olyan játékokra utal, melyek magas költségvetésű, magas színvonalú fejlesztés eredményeképp jönnek létre. Az „A” az amerikai osztályzati rendszerből származik, mely a legjobb értékelést takarja.



1. ábra. Minecraft

Az AAA projektek mindig több évig készülnek, óriási, hollywoodi filmeket meg-szégényítő költségvetésből fejlesztik őket és sokszor több cég dolgozik rajtuk párhuzamosan, több száz programozóval. Előfordul, hogy egy külső fejlesztő cég bízta meg az adott céget azzal, hogy fejlesszen le valamilyen játékot, melynek paramétereit vagy a kiadó határozza meg, vagy csak és kizárólag pénzbeli támogatással próbálják a projektet sikerre vinni, melyből később részesednek.

A szoftverkrízis veszélye náluk jóval magasabb kockázati rátával bír, mint az indie cégeknél. Lássuk ennek az okait:

Nagy bukásveszély:

Egy nagy gond van ezekkel a cégekkel, mégpedig az óriási költségvetés. Mivel nem akarnak nagyot bukni, ezért igyekeznek már bejáratott sablon ötleteket alkalmazni, illetve saját, sikerre vitt címeiket egy-egy új résszel bővíteni. A bukás veszélye igazából csak egyetlen platformon tetőzik, a számítógépeken.

A *warezolás* (médiafájlok illegális letöltése, illetve használata) az utóbbi években az internet térhódításával és a sávszélesség jelentős növekedésével eddig soha nem látott csúcsokat ér el, amely veszélyezteti nem csak a játékegyesítőket, de a zeneszerzők, előadóművészek és a filmipar létjogosultságát is a piacon. A játékipar ezért a kreatív, drága ötleteiket *konzolokon* valósítja meg. Ezek közül a Playstation 3 illetve az Xbox 360 platformok rendelkeznek elég korszerű hardverrel ötleteik kiteljesedéséhez.

Az AAA kategóriájú játékokat fejlesztő cégek úgynevezett dobozos szoftvert fejlesztenek, mely széles közönségnek szól, így a fent említett formát betartva nagy eséllyel sikert tudnak elkönyvelni.

2.2.1. Egy AAA játék bemutatása

Most lássunk példát egy nagy költségvetésű, ámde igen sikeres játékra:

Portal:

1996 augusztus 24-én Gabe Newell és Mike Harrington megalapította a Valve Corporationt. Sikerük első játékkal, a *Half-Life* játékkal kezdődött 1998-ban, mely számos elismerést és díjat bezsebelt. 2003 szeptember 12-én létrehozták a Steam digitális disztribúciós platformot, melynek mára közel 40 millió aktív felhasználója van, ezenkívül 2012 januárjára 1504 játék vásárolható meg és tölthető le rajta keresztül.

2005-ben a Nuclear Monkey Software kiadta a Narbacular Drop című, *environmental puzzle* (környezeti elemeket felhasználó puzzle) típusú játékukat, melyben egy különös fantasy világban kellett No-Knees („térdrnélküli”) hercegnővel egy démon fogságából kiszabadulnunk egy Wally-nak nevezett elementál lény segítségével, melynek olyan különös képessége volt, hogy portálokat tudott létrehozni.

A Valve Corporation alkalmazta e játék létrehozóit egy saját, kísérleti projektjükhöz, a *Portal*hoz, melyet egyfajta jutalomjátékként szándékoztak kiadni két másik játékkal egyetemben, a Team Fortress 2 és a Half-Life 2 Episode Two játékokkal, az Orange Box pakk részeként.

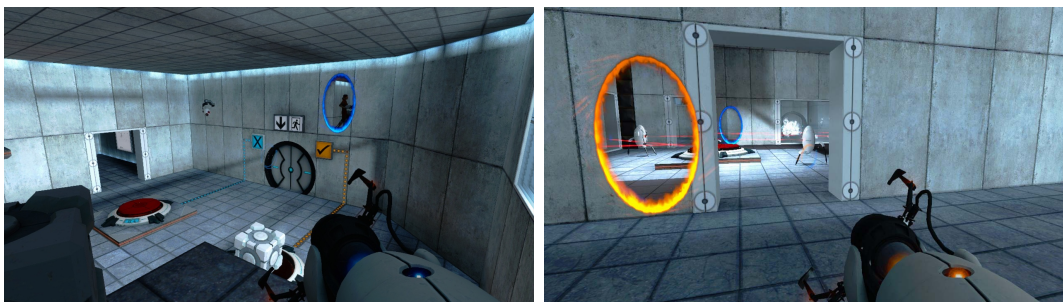
A játék sikerességét tükrözi a máig 4 millió eladott példány, a 85-90% feletti értékelések a szaksajtótól, valamint a számtalan *fan art* (rajongók által készített rajzok, replikák és egyéb tárgyak), melyek a játék egyes motívumait jelenítik meg. Népszerű fan artnak számít például Harry-UK nicknevű (becenevű), brit filmkészítő, szinkronszínész és zeneszerző által készített „This is Aperture” című nagysikerű videoklipp, melyet 16 nap alatt több, mint 1,7 millióan tekintettek meg a Youtube videómegosztó oldalon keresztül.

A Portal című játék egyik fő erőssége a története, erre csak alapvetően térnék ki, meghagyva az esetleges felfedezés lehetőségét az Olvasó számára. A játék egy fikcionális (kitalált) kísérleti laboratóriumban játszódik, az Aperture Laboratories-ban. A főszereplő egy tesztalany, Chell, akinek egy sor teszten kell átesnie. Ezeket a tesztek egy GlaDOS nevű mesterséges intelligencia követi figyelemmel, olykor szarkazmussal és cinizmussal teli kommentekkel (megjegyzésekkel) látva el Chellt.

A játékmenet alapvetően puzzle platformer, vagyis ugrálásból és rejtvénymegoldásokból áll a játék. Azonban a játék a portálokkal történő feladatmegoldásra helyezi a hangsúlyt. Egyetlen „fegyverünk” van, az ASHPD (Aperture Science Handheld Portal Device - Aperture Science Kézi Portáleszköz), melynek segítségével egy bemeneti és egy kimeneti „ajtót” hozhatunk létre. Benézve

az egyik „ablakon” láthatóvá válik az, amit a másik „ablakon” keresztül kellené látnunk. Ha az egyiken egy tárgy, személy, vagy akármi áthatol, akkor a másikon keresztül jut ki. A puzzle-eket persze emellett számtalan egyéb platformelem teszi érdekesebbé, például energiagömböket kell az ezeket felhasználó eszközökbe juttatnunk, stb.

Talán egyetlen negatívuma volt a játéknak, mégpedig a rövidejsége. 2011 áprilisában jelent meg a Portal 2, melyben rengeteg új platformelem és egyéb tartalom vált elérhetővé, jócskán felülmúlva a szintén sikeres első részt. Kooperatív (együttműködő) többjátékos mód saját történettel, több új játszható szereplő, még izgalmasabb és fordulatosabb történet fűszerezte a 90-95%-os értékeléseket begyűjtő második részt.



2. ábra. Portal

3. Jó játék létrehozásának titka

Ebben a fejezetben a jó és rossz játékok tulajdonságait taglalom, valamint követendő példákat állítok fel a 2. fejezetben említett kétfajta cégtípus számára, melyeket követve akár eredményesebbek lehetnek a piaci versenyben más cégekkel szemben.

Jó és rossz. Definíciójuk szerint mindkét fogalom meglehetősen relatív, de egy adott rendszer belsejében logikus és egyértelmű lehet a különbség közöttük.

Ezek a triviális különbségek fennállnak akkor is, amikor egy játékot tekintünk jónak avagy rossznak. Ugyanis cégek nagysikerű és bukott játékaiknak elemzésével világossá válhat számunkra, melyek lehetnek a pénzügyileg sikeres játék alapvető tulajdonságai, illetve mely leendő attribútumokat kerüljünk el lehetőleg minél messzebből.

a 2. fejezetben láthattuk, melyek a főbb cégtípusok és azok jellemzői, ez alapján, és sikeres illetve sikertelen projektek figyelembevételével könnyedén meghatározhatóak a fentebb említett tulajdonságok.

3.1. A siker titka indie-ként

Ha indie fejlesztésbe fogtunk bele, valószínűleg alacsony költségvetésből próbálunk alkotni valamit. Legtöbbször egy ötletet akarunk megvalósítani valamilyen játék formájában. Nyilván érdemes eleinte csak „szárnyakat próbálgatni”, mint nagy fába vágni a fejszénket, ugyanis közel se biztos, hogy megtérül próbálkozásunk. Erre nagyon jó példaként fel tudom hozni a 2008 októberében megjelent, Team Meat játékot, a Meat Boy-t, akik *flash* játék formájában adták ki ötletüket. Mivel meglehetősen sikeres lett, ezért 2 évvel később több platformon is kiadták az eredeti játékukhoz hasonló, de jócskán tökéletesített alkalmazásukat. 2012 januárjára az immáron fizetős játékukat több, mint egy millióan vásárolták meg.

Ebből a példából is jól látszik, hogy az ötlet sikeres bétatesztelését követően számtalan formában kiadhatjuk és nagyobb valószínűségű sikert könyvelhetünk el. Érdemes projektünket a már említett flash formában egy erre szakosodott oldalon publikálni (<http://www.newgrounds.com/>), esetleg egy már meglévő játék szerkesztőjét használatba véve modifikációt készíteni egy játékhoz. Példának okáért megtekinthetjük a Blizzard Entertainment Inc. által fejlesztett Warcraft III: Reign of Chaos modifikációját, melynek címe Defense of the Ancients. Ez a játék is valós idejű stratégia (RTS - Real-Time Strategy), akárcsak az ez alapján készülő teljes értékű folytatás, a DotA 2, melyet szintén a Valve Corporation fejleszt.

Amennyiben *modifikációt* fejlesztünk, nem kérhetünk érte pénzt, legalábbis ez nem megszokott. Bár létezik olyan fizetési mód, hogy „pay what you want” (azaz fizess annyit, amennyit akarsz), de teljes értékű játékokért szokás pénzt kérni.

Valamennyire kevésbé kötött munkát tesz lehetővé, emellett nagyobb szabadságot ad, ha egy népszerű *game engine*-t (játék motort) használunk, abból a célból, hogy ötleteinknek formát öltünk. Célszerű lehet például a Unity-t használni erre a célra, mely bár nem minden funkciójában ingyenes, de ezek a fizetős tulajdonságok nem jelentenek akadályt a Unity-vel készült termékek publikálására. Jó példa a független fejlesztés és Unity kereszteződésére a Sauropod Studio által fejlesztés alatt álló Castle Story, amely gyakorlatilag egy stratégiai Minecraft játékra emlékeztet.

Itt célszerű megemlítenem, hogy lehet bizonyos ötleteket más játékokból venni, de azért próbáljunk egyediek lenni lehetőleg minél jobban. Ha ténylegesen olyan ötletet használunk fel, amely előfordulási gyakorisága elég ritka, *plágiumgyanúba* keveredhetünk, illetve be is perelhetnek minket (utóbbit valószínűleg nem mi nyerünk). Egy ilyen eset pedig teljesen alááshatja jelenlegi vagy leendő hírnevünket. Nagyon jó példa erre a Majestic Studios által fejlesztett Limbo of the Lost, amelyről kiderült, különféle játékokból összeollózott tartalmakkal rendelkezik.

Térjünk ki egy lehetséges kockázatra. Joggal hihetnénk azt, hogy a warezolás veszélyeztetheti projektünk létjogosultságát a játékiparban. Azonban gondoljunk arra, hogy kezdők vagyunk, nem várhatjuk el azt, hogy a játékosok felfigyeljenek egy ismeretlen cég semmitmondó termékére. Ergó biztosak lehetünk abban, hogy ha magas árat, vagy elsősorban a PC-t (személyi számítógép) célozzuk meg a játék platformjának, saját sikerességünket kockáztatjuk. Ezért érdemes lehet valamelyik konzolt választani elsősorban. Költségek csökkentése érdekében használhatjuk az adott konzol *online disztribúciós szolgáltatását* (Playstation 3 esetében PSN, XBox360 esetében pedig XBLA). Amennyiben a PC platformot részesítenénk előnyben, válasszuk a Valve Corporation által fejlesztett *Steam-et*, vagy az Electronic Arts készítette *Origin* szolgáltatást.

Akármelyik módot is választottuk korábban, a következő tulajdonságok segíthetnek jó és sikeres játék fejlesztésében:

Alkalmazzunk linearitást sandbox helyett:

Ugyanis egy sandbox játékot jóval nehezebb „emlékezetesen” kidolgozni, a linearitás viszont könnyebben lehetővé fogja tenni számunkra akár filmszerű ötleteink megvalósítását.

Többféle nehézségi mód:

Ügyeljünk arra, hogy a játékunkkal kezdő és *hard core* (profi) játékosok is

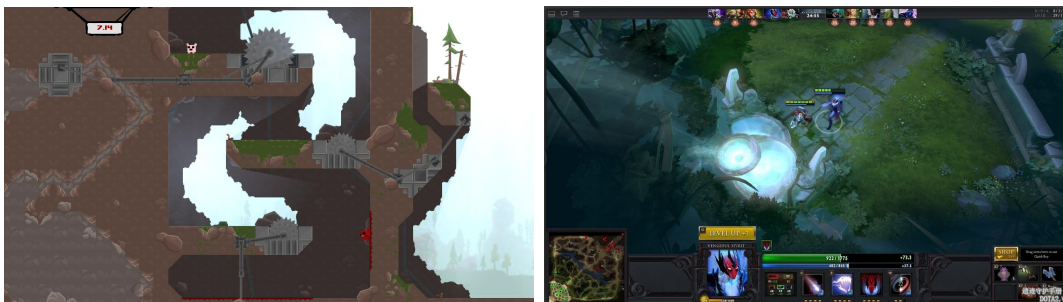
játszhatnak. Hogy kibővítsük lehetséges felhasználói közösségünket, ajánlott legalább három nehézségi fokozatot alkalmaznunk, ezek lehetnek a könnyű, közepes illetve nehéz szintek. A nehéz szintet bekonfigurálhatjuk úgy, hogy a gyakorlott játékosok számára kellő kihívást nyújtó élményt biztosítson. Amennyiben a végtelenül elszánt játékosokra is fókuszálni akarunk, külön legyen egy alig teljesíthető nehézségi szint is, amelyet ténylegesen csak napokig, akár hetekig tartó gyakorlással küzdhet végig a játékos.

Teljesítményértékelési rendszer használata:

Pszichológiailag kimutatható tényező, hogy az a rendszer, amely folyamatosan jutalmazza az emberek energiabefektetését, készíti a felhasználókat a folyamatos használatra. Ezen az elven működnek a szerepjátékok (RPG-k), illetve az online szerepjátékok (MMORPG-k) is. A szerepjátékok lényege, hogy egy adott karakternek a helyébe képzeljük magunkat, aki saját képmásunk (avatar) lehet egy virtuális világban, ahol kezdetben gyengébbek, sérülékenyebbek vagyunk. Ahogy a megbízóink (NPC) által kapott feladatainkat végrehajtjuk, tapasztalati pontokat szerzünk. Megfelelő mennyiségű *tapasztalati pont* (XP) segítségével szintet léphetünk, ezáltal újabb képességek, támadási formák birtokába juthatunk. Néhány játéknál, például több, mint 10 millió előfizető felhasználóval rendelkező World of Warcraft-ban - ami mellesleg Guinness rekordot tartja ezen a téren - ez a fajta fejlődési rendszer rengeteg egyéb dologgal, például fejlődési fával, szintenként elérhető, erre vonatkozó „fizetőszközzel” (talentpontokkal), stb. van megtoldva. Más játékokban még jelentős dolog lehet az, ha bizonyos területrészeket később érhetünk el (sandbox játékokban előny), például a Rockstar Games által fejlesztett Grand Theft Auto *franchise-ban* (sorozatban) láthatunk hasonló megoldást.

Változatos zene:

Előny, ha a zenénk a lehető legváltozatosabbak. Külön helyzetekre külön zenét alkalmazzunk, és ne törekedjünk más szerzők jogvédett zenéit beszerezni. A szakma jobban becsüli az egyedi zenékkal rendelkező játékokat. Ne erőltessük a manapság divatos, ámde élvezhetetlen, monoton zenéket. Helyette konyhanyelven a *dúdolhatóságra* törekedjünk. Minél több visszatérő momentum (refrén) van a zenénkben, annál jobbra fogják értékelni a kritikusok. További pozitívum lehet, ha zenekarral vesszük fel a megkomponált művet. Ez persze nem olcsó, így utóbbi tippet inkább az AAA kategóriás cégek fogják igénybe venni.



3. ábra. Super Meat Boy, Defense of the Ancients 2

3.2. Hogyan legyünk jók nagy költségvetésből?

Tegyük fel, hogy gazdag cég vagyunk, jó programozókkal, és nem félünk nagy tőkét rááldozni leendő projektünkre. Ekkor, mint már korábban, a 2. fejezetben is írtam, nagy kockázatot vállalunk a cég jövőjét illetően, ugyanis ha bizonyos dolgokra nem figyelünk, akár csődöt is jelenthet hirtelenjében cégünk. Most a teljesség igénye nélkül eltekintünk a részletes kockázatelemzéstől, csak a fejlesztés szempontjából fontos tényezőt ragadjuk ki, illetve erre ajánlunk valamilyen megoldási lehetőséget.

Amennyiben jó játéktervünk van, még mindig nem lehetünk biztosak a sikerességben. Ekkor ugyanis célszerű olyan platformra kiadni játékunkat, melyen kevésbé elterjedt a warezolás. Ennek a kockázati faktornak a leredukálására elsősorban meg kell akadályoznunk a *fejlesztői verziók* kiszivárogtatását. Ez az egyik legveszélyesebb dolog, ezért célszerű többszintű védelmet kialakítani. Ha ennek a kockázata elfogadható mértékűre csökkent, gondolnunk kell arra is, hogy ha egyszer megjelent a játék, akárki „ellophatja”, feltöltheti számos helyre a játékot, majd ezt *cracker* csapatok rövid idő alatt feltörhetik. Ennek elkerülésére sem ajánlott alkalmazni másolásvédelmet, mert az effajta lépések sokszor a legális felhasználók életét nehezíthetik meg, és a cracker csapatoknak ez nem lesz akadály. Következésképp felesleges a játék költségeit és fejlesztési idejét a védelemre való törekvéssel növelni. Ha PC-re fejlesztünk elsődlegesen, a lopás kockázata a maximális. Amennyiben mégis ezt a platformot választanánk, vegyük igénybe valamilyen *felhő alapú rendszer* szolgáltatásait, mint például az OnLive vagy Gaikai. Ennek segítségével a felhasználóknak nem kell relatíve erős gépet igénybe venniük, ha játékra szottyan kedvük, elég egy pár mb-os klienst letölteniük, regisztrálniuk OnLive esetén az <http://www.onlive.com/> oldalon, és ha engedélyezzük a fél órás kipróbálási lehetőséget, nagyobb eséllyel fogják megvenni termékünket. Arra viszont gondoljunk, hogy ez a platform a PC-n belül is csak egy kis szelet, tehát nem lehet csak erre bízni játékunk és cégünk jövőjét. Konzolok tekintetében a legbiztonságosabb platform a PlayStation 3. Volt olyan *firmware* (kisméretű program, mely egy adott elektronikai eszköz vezérlését végzi)

verziója, amely lehetővé tette például operációs rendszer telepítését a winchesterére, ezáltal manipulálhatóvá téve a rendszer *eredetiség-felismerő* komponenseit. Köszönhetően az állandóan érkező frissítéseknek, az efféle módosítások ki lettek küszöbölve, máig sikeresen ellehetetlenítve a kalózkodást ezen a platformon. Habár Xbox 360-on könnyűszerrel lehet kalózkodni, de még így is jóval kisebb arányban történik meg a Microsoft termékén, mint PC-n. A többi konzol, illetve platform *casual* játékokra lettek kitalálva, így azokra nagyjából az vonatkozik, mint a független fejlesztők játékaira.

Akármelyik platformot is választottuk, olyat ne tegyünk, hogy egy franchise különböző részeit különböző platformokra adjuk ki. Kivéve, ha ennek okai vannak, például ha a saját cégünk által fejlesztett konzolt akarjuk népszerűsíteni az adott játékkal. Az viszont biztos, hogy a játékosokban ez *ellenszenvet válthat ki*, mert ha valaki vett már egy erős konzolt az aktuális konzolgenerációban, nem fog egy teljesen másikra áttérni, még több játék miatt sem. Ilyen franchise-ra jó példa a *Condemned*, melynek első része még meglátogatta a PC platformot, a második része viszont már nem. Szemben a *The Darkness*-szel, aminek pont csak a második része jelent meg PC-re. De ide lehetne még sorolni a *Gears of War*, *Halo*, *Fable* franchise-okat is, utóbbinak az első és harmadik része jelent meg PC-re, a 2. rész nem.

Ezen dolgozat írásának pillanatában még nem, de pár hónap, esetleg egy év múlva már nagy valószínűséggel eljön a konzolok nyolcadik generációja. Ezekre a leírt kalózkodási és egyéb tulajdonságok figyelembevételével lesz érdemes fejleszteni.

Természetesen *nem garantálja* a sikert a platformválasztás, ugyanis PC-n is lehet sikeres egy játék, ha jó. Erre példaként a PC exkluzív, Microsoft Windows és Mac OSX operációs rendszerekre megjelent *Starcraft II: Wings of Liberty* című játékot lehet felhozni, mely miután kigördült a Blizzard Entertainment Inc. műhelyéből, 3 millió példányban kelt el világszerte.

Az indie játékokra vonatkozó tippek természetesen AAA kategóriásra tervezett játékoknál is érvényesek, viszont a kidolgozottságra lényegesen nagyobb hangsúlyt kell fektetnünk. Erre a sandbox játékoknál fokozottan kell ügyelnünk.

A következőkben kiemelek pár tippet az előzőek mellett, amiket érdemes alkalmazni:

Linearitás esetén filmszerű hatás elérése:

Sok scriptelt jelenet, hangszeres zene bejátszások, változatos, hirtelen jött események használata

Titokzatosság:

A figyelem folyamatos fenntartása miatt, a történet jó húzóereje lehet a játéknak. A játékos csak nehezen sejtthesse előre, hogy mi fog vele történni

Változatos játékmenet:

Minijátékokkal teletűzdelt játékot később lehet megunni, mint egy monoton, unalmas játékot

Változzon a zene a főszereplő hangulatának megfelelően:

Például pörgős eseményekhez illeszkedjen pörgős zene is

Gyűjthető tárgyak:

Fontos, hogy ne vigyük túlzásba a használatát és ezek a tárgyak aprók legyenek, rejtett helyeken helyezkedjenek el

Skálázható részletesség:

Ha PC-re (is) fejlesztettünk, támogassuk is a platformot, ne csak egy olcsó konzolportot kapjanak, mert ellenszenvet válthatunk ki a játékosokban

Újítások:

Ha egy franchise-t akarunk új taggal bővíteni, fontos, hogy valamit változtassunk. Persze ezt úgy tegyük meg, hogy a játékosok, illetve a szaksajtó által dícsért részekben ne változtassunk. Változtatás nélkül lehet, hogy páran „rókabőr” jelzővel fogják illetni termékünket és a kritikusok is esetleg leszólják művünket. Ezért ha fejlesztünk, nézzük meg egy hasonló játék értékelését, az alapján könnyebb lehet döntésünk.

Létező pályarészek ne ismétlődjenek:

Egy olcsó lépésről van szó, viszont szembeűnhet, ha több helyszín azonosan néz ki. Ezért ajánlott valamennyit mindig minden helyszínen változtatni.

Elveszettségérzés csökkentése:

Hacsak nem direkt ez a célunk, próbáljuk egyértelműen és átláthatóan, emellett ne zavaróan közölni a játékosal, mit tegyen, merre menjen, stb. Nem megfelelő közlésre példa a Bethesda Softworks által fejlesztett The Elder Scrolls III: Morrowind című RPG, melyben egy naplóban leírászerűen olvashatjuk feladatunk. Ez elősegíti a beleélést a játékba, viszont az angolul kevésbé tudóknak alig van lehetőségük kiásni a megfelelő ígét vagy főnevet, melyre szótárban utána tudnának keresni. Túlélőhorror játékoknál, illetve pszichológiai thrillerreknél *előnyt* jelenthet direkt előteremteni ezt az érzést. Hatásosságára jó példa

a feloszlott Team Silent által fejlesztett Silent Hill franchise, melynek inkább az első négy epizódjában látható sűrű ködben való barangolás elősegíti ezt az érzést.

Az „én szerep” érzésének növelése:

Ha nem az a célunk, hogy a játékunk filmszerű legyen, hanem a beleélésre, átérzésre törekszünk, érdemes a *cutscene-ek* (átvezető animációk) számát lecsökkenteni, akár megszüntetni. Történetet mesélni ezek nélkül is sikeresen lehetséges, scriptelt jelenetek segítségével. Fontos, hogy ne láthasson a játékos gyakran azonos scripteket, például FPS-ekben (első személyű lövöldözős játékokban) az elhalálozott karakterek ne előre legyártott animációban essenek össze, hanem *ragdoll* (rongybaba) technológiát alkalmazva, melynél a valós fizikának megfelelően esnek össze. Utóbbira jó példa a Valve Corporation által fejlesztett Half-Life 2 című nagysikerű játék, mely megjelenése korában a fizikai fejtörők terén jelentősnek számított. A Half-Life franchise-ra jellemző az is, hogy nincsenek benne átvezető animációk, legalábbis olyanok nem, amely során a főszereplőt kívülről láthatnánk. Az más kérdés, hogy a sorozat részeiben láthatjuk egy másik epizód főszereplőit, de a tipp szempontjából ez nem fontos.

Szimbolikusság:

Más néven karakteresség. Ez alatt azt kell érteni, hogy a játék világában tapasztalható dolgok kellően karakteresek, változatosak. A szimbolikusság marad meg hosszan a játékosban a játék végigjátszása után is. Ugyanezen okból kifelé is lesznek sikeresek bizonyos filmek illetve könyvek is. A szimbólum lehet akár a játékban elhangzott idézet (The Elder Scrolls V: Skyrim-ben az örök által mondott „I used to be an adventurer like you, then I took an arrow in the knee.”, vagyis „Hajdanán olyan kalandor voltam, mint te, de nyilat kaptam a térdembe.”, vagy a Portal című játékban elhangzó „The cake is a lie”, magyarul „A torta hazugság”) vagy egy nagyon egyedien kinéző karakter is (A Silent Hill 2. részében látható Pyramid Head, magyarul piramisfej).

MMO-val óvatosan:

Ez az a műfaj, amivel nagyon vigyázni kell, cégek mehetnek tőle tönkre, ugyanis nagyobb kidolgozottságra van szüksége, mint egy más stílusú játéknak. Ne bízzunk abban, hogy ha például a World of Warcrafttól nyúlunk le ötleteket, biztosan sikeresek leszünk. Ez nem így van. Jelentősen megnövelheti a költségeket, ha egy, már felhasznált *licenst* használunk, és bízhatunk abban, hogy a neve majd el fogja adni a játékot, de ez korántsem biztos. A felhasználó tábor

megtartásához folyamatosan új tartalmakkal kell bővíteni a világot, illetve ha a létszám jelentősen csökken, át kell térni *Free-to-play* modellre. Ez a marketingnek is jót tesz, mert többen fogják kipróbálni a játékot, ergó nagyobb az esély, hogy ha már valaki eljut egy bizonyos szintre, fizetni is fog valamilyen tartalomért. Célszerű ezt a modellt csak és kizárólag egy bizonyos létszám csökkenés után alkalmazni.



4. ábra. Starcraft II, Silent Hill 2

3.3. A marketing kérdése

A marketing a sikeresség nem elhanyagolható részlete. Fontos, hogy ne csak a marketingre bízunk mindent, de szánjunk rá a költségvetésből. Ennek mértéke különböző lehet, attól függően, milyen cégek vagyunk.

A készülő játék bejelentését célszerű elhúzni, fél évvel, vagy egy teljes évvel a tervezett megjelenés elé. Ennek az az oka, hogy ha túl hamar mutatják be a játékot, az még nem üt akkorát, amekkorát később üthetne, és ezzel a lehetséges felhasználók bizonyos részétől már meg is szabadulna a fejlesztőcsapat.

A bejelentést megelőző szakaszban készülhet *ARG* (alternate reality game) is. Ezt akkor célszerű alkalmazni, ha már jókora rajongói táborral rendelkezünk, ugyanis az ARG sokszor rejtvényekből, utalásokból áll, melyeket rendszerint nem is annyira egyszerű megfejteni. Az ARG a tényleges bejelentésig elég homályos utalásokat tartalmazzon, fokozva a későbbi felhasználók kíváncsiságát, ezzel együtt az elismerését a cég, vagy a még be sem jelentett játék iránt. Bejelentés előtti ARG-ra jó példa a 2K Marin által fejlesztett Bioshock 2 című játék, melynek a bejelentése előtt feltűnt <http://somethinginthesea.com/home.html> oldala fokozatosan átalakult a bejelentésig, mindig egyre több információt adva az akkoriban készülő játékról.

A bejelentésnek érdemes valamilyen nagyobb *expón* elhangoznia, mert azokat a legtöbb játékos figyelemmel követi. Ilyen nagy rendezvény lehet például az E3 (Electronic Entertainment Expo, melyet Los Angelesben rendeznek meg minden

évben), a PAX (Penny Arcade Expo, egyik fajtája a PAX East, Seattle-ben és Bostonban szokták megrendezni), a Németországban rendezett Gamescom, a GDC (Games Developer Conference, változó helyszínen zajlik le), illetve a japán TGS (Tokyo Game Show).



5. ábra. Bioshock 2 promóciós palack, Final Fantasy XIII óriásplakát

Amennyiben alacsony költségből szeretnénk reklámozni játékunkat, készítsünk videókat, melyeket majd elsősorban egy erre alkalmas videómegosztó oldalon publikálunk. Videóknak több fajtája lehet:

Teaser :

Ez a fajta videó tipikusan semmit nem mutat meg a készülő játékból, vagy ha igen, akkor se sokat. Az ilyen típusú videókat bejelentésre használják, figyelemfelkeltés céljából. Néha tartalmazhat rejtett utalásokat bizonyos tartalmakra. A *teaser* speciális trailernek számít.

Trailer :

A *trailer* jellemzően sokat mutat meg a játékból. Átlagosan két fajtája van: cinematic trailer és gameplay trailer. A cinematic trailer lehet egy CGI (computer-generated imagery, magyarul számítógéppel létrehozott kép) trailer is, amelyben vagy a játék motorját használatba véve csinálnak egy rövid, filmszerű, csattanóval fűszerezett hatásvadász videót, vagy a játékból emelik ki valamelyik cutscene-t. Alacsony költségvetésből általában nem készítünk cinematic trailereket. A gameplay trailerek kizárólag játékmenetet tartalmaznak, és a cinematic trailerekkel szemben nem csak pár percesek, hanem akár negyed órák is lehetnek. Ezeket olyan részekről készítik, melyek a lehető legkevesebb *spoilert* („elrontó”, amikor a játékból, filmből, könyvből elárulunk részleteket) lövik le a játékból, mégis felkelthetik a játékos figyelmét.

AAA kategóriás játék lévén hirdethetünk nagy népességű városokban plakátokon (például Los Angelesben a Square Enix által fejlesztett Final Fantasy XIII óriásplakát), reklámokon, palackpostával (Bioshock 2 játék esetében is ez történt, játékbeli plakátok vetődtek partra), üdítőn keresztül DLC-vel (letölthető tartalom) (az Electronic Arts 5 játékát népszerűsítette a Dr. Pepper üdítőitalhoz járó DLC-kkel, köztük a The Sims és Mass Effect 2 játékokat) is. Emellett még kirívó eset a Mass Effect 3 megjelenés előtt ürbe bocsájtott példányainak története is, melyek egy részét a szerencsések megszerezhették.

3.4. Terméktámogatás

A rajongók szimpátiája szempontjából az egyik legfontosabb dolog a termék megjelenése utáni *terméktámogatás*. Ennek mára két formája alakult ki:

Patch :

Másnéven update. A *patch* magyarul foltozást, javítást jelent, mely a patch fogalom definíciójára is enged következtetni. A fejlesztők nem tudnak előre felkészülni az egyes konzolok firmware frissítéseire, a sokféle Windows, Macintosh és egyéb PC platformra, PC esetén a számtalan hardver kombinációra, ezért vagy még a bétateszt során, vagy megjelenés után gondoskodniuk kell a terméktámogatásról. Erre a cégek külön embereket alkalmaznak, például a Blizzard 4700 alkalmazottja közül csupán 750 fejlesztő dolgozik, az összes többi a terméktámogatásban segít. Például fogadják az elkeseredett felhasználók telefonhívásait, segítkeznek az adott játék hivatalos fórumán, más játékosokkal egyetemben, stb. A patcheknek, főleg a kritikus javításoknak minél hamarabb el kell készülniük, különben a termék veszíthet piaci értékéből. A nagyobb cégek több éven át támogatják elkészült termékeiket, javítgatják őket, elsősorban addig, amíg van rájuk igény a *visszajelzések* alapján. Ez egyébként minden más termékre igaz, nem csupán a játékokra. Ugyanakkor a patchek elkészítése olykor igen drága lehet, mivel olyan fejlesztésről van szó, amelyre elvileg nem kellene szükségszerűen, hogy sor kerüljön. Ezt a *költséget*, mely konzolok esetén elérheti a 40,000 dollárt is, a cégnek kell finanszíroznia. Viszont, ha kellően nagy a fan (rajongó) tábora a cégnek és a játéknak, érdemes lehetőséget adni a felhasználóknak, hogy támogathassák a céget ezeknek a költségek egy részének fedezésével. Ezzel a cég és a felhasználók is jól járnak, mivel biztosak lehetnek abban, hogy ténylegesen elkészül a javítás az ő igényeiknek megfelelően. Sajnálatos példa a kritikus patchek aránylag lassú érkezésére az Id Software LLC által fejlesztett *Rage* című játék terméktámogatása. John

Carmack, a cég egyik kulcsembere, kinek álma az volt, hogy az Id Tech 5 (a játék grafikus motorja) minden platformon 60 FPS-sel (frame per screen - képkocka/másodperc) lesz képes futni. A motor a MegaTexture technológia továbbfejlesztését, a *Virtual Texturing*-et alkalmazta, mely segítségével 20 Gb-nyi nagyfelbontású textúrákat (képeket) használunk a motor által hajtott játékban. Ezeket persze csak akkor tölti be a memóriába a motor, ha ténylegesen kellenek. Az optimális FPS szám eléréséhez pedig a motornak megfelelően kategorizálnia kellett volna a PC hardvereit a minél optimálisabb futási sebesség eléréséhez. A Virtual Texturing eredménye az volt, hogy ha fordultunk a játékban, folyamatosan láthattuk, ahogy a lebutított textúrák fokozatosan töltődnek be a memóriába és jelenítődnek meg nagy felbontásban. Eredményképp szépen lezuhant az FPS szám egy-egy hirtelen jött mozdulat során. A játék 2011 októberében jelent meg, melyhez még abban a hónapban jött is a javítás. Ez viszont sok PC-n nem oldotta meg a problémákat. A gondokat viszont Carmack driver hibáknak állította, és a fejlesztőit szidta le, amiért nem voltak képesek kellő körültekintéssel tesztelni a játékot különféle drivereken, videokártyákon. Ennek a jogosságába nem mennék bele, viszont ez a húzás a cég *reputációját* (elismerését) sok rajongó számára csökkentette.

DLC:

A *downloadable content* szóból származik, magyarul letölthető tartalom. Talán a legelső DLC az Earthworm Jim című játékhoz készült, melynek speciális kiadását a Shiny Entertainment fejlesztette. A DLC-k eredeti céljuknak megfelelően olyan támogatást nyújtottak volna, hogy a felhasználók megkaphassák azokat a tartalmakat, amelyeket a kiadó által sürgetett fejlesztés miatt nem volt lehetőségük a fejlesztőknek integrálniuk a kiadott játékba. Ez viszont legtöbbször nem így van. A DLC-k általában olyan hangsúlyos dolgok, mint külön pályák, helyszínek, küldetések, és csak ritkábban tárgyak egy-egy játékhoz, emiatt a fejlesztők általában pénzt kérnek értük. Emiatt előfordul, hogy a kész játékot kiadás előtt több részre darabolják fel, esetleg elszeparálnak pár dolgot a végleges játékból, hogy plusz pénzt kérhessenek értük. *Ingyenes* DLC-kre jó példa a Valve által fejlesztett Team Fortress 2, Portal 2 illetve Left 4 Dead 1 és 2 DLC-i. A Valve Corporation a Steam-el annyi bevételhez jut, hogy számukra nem okoz gondot ilyen apró bevételkiesés, inkább a reputációjukat szeretnék növelni, ezzel is hatva a jelenlegi vagy jövőbeli felhasználókra. Fizetős és meglehetősen *drága* DLC-kre példa az Activision által kiadott, számos fejlesztőcsapat által fejlesztett Call of Duty franchise legújabb tagjainak DLC-i. Például a Call of Duty Modern Warfare 2 című epizód Stimulus Package

névre keresztelt DLC-jének ára a megjelenésekor átszámítva 3250 Ft-ba került, mely a teljes játék árának körülbelül a 40%-a volt megjelenésének idején, és csak 5 multiplayer pályát tartalmazott, melyből 2 egy előző epizód térképeinek átdolgozása volt. Ennek ellenére 24 óra alatt 1 millióan megvették.

4. Blender

Az eddigiekben a jó és rossz játékok fogalmát taglalva megértettük, hogy mi kell egy jó és sikeres játék elkészítéséhez. Ezután bepillantást nyerhetünk a Blender és a Unity világába, mely programok ismeretében akár már neki is láthatunk első játékunknak.

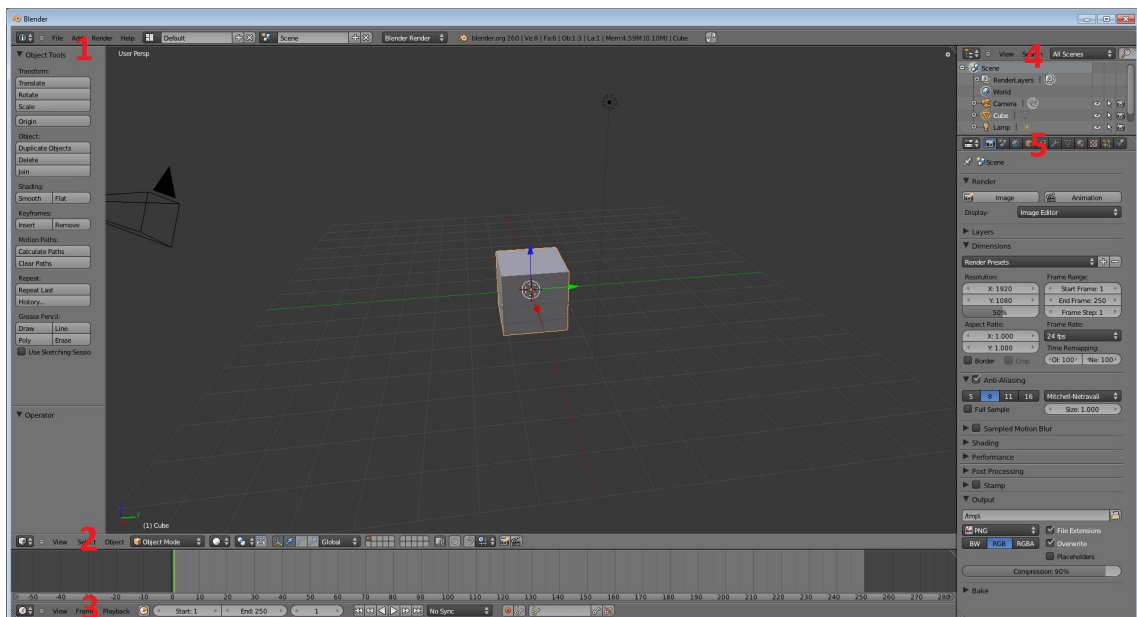
A Blender Foundation által fejlesztett *Blender* nevű 3D grafikai programot sokáig zártan, a cégen belül használták. Mára teljesen nyílt és népszerű szoftver lett. Lásd [1] és [2].

A Blender képességei hasonlóak más, kereskedelmi forgalomban kapható zárt forráskódú programokhoz, sőt, teljesen ingyenes és nyílt forráskódú, ennek ellenére rendszeresen fejleszti a Blender Foundation, és számos dologban kiemelkedő tulajdonságokkal rendelkezik, mint az UV kicsomagolásnál használt fejlett algoritmusok (lásd a 4.3. fejezetben).

Emellett a Blender hírhedt arról, hogy nehéz megtanulni. Ez leginkább abból adódik, hogy sokáig zárt szoftver volt egy cégen belül. Mióta nyílt lett, sokat foglalkoznak azzal, hogy logikusabbá, átláthatóbbá tegyék felületét és egyre több dokumentáció készül róla, mely egy vagy több funkcióját elemzi, taglalja működését.

Számos projektben használták, úgy, mint a Pókember 2 című filmben, ahol képes forgatókönyvet terveztek meg segítségével. Ezenkívül a Sintel, Big Buck Bunny és Elephants dream című animációs filmeket is a programnak köszönhetjük. Rendekért (képekért) lásd a [2, 500. oldaltól].

Indítás után a 6. ábrán látható képernyő fogad minket:



6. ábra. A Blender felülete

Mint a legtöbb program, úgy a Blender felülete is több kisebb ablakból áll. A különbség annyi, hogy a Blender felülete szabadon változtatható. Az „*action zone*”-okkal (vonalkázott rész) lehet új ablakokat felhúzni, illetve a meglévőket egybemosni a régiékkal. Az alapbeállítás szerint a kezdőképernyőn a képen látható módon helyezkednek el az ablakok, melyek meg vannak számozva a képen (egyéb ablakokhoz lásd a [2, 1. fejezet] és az [1, 2. fejezet]). Viszont ezek akármikor átválthatóak másik ablakra. Ez alapján az adott ablak funkciója:

1. Info

Gyakorlatilag a főmenü opciókat tartalmazza. Itt lehet menteni, betölteni, importálni, exportálni, új objektumot hozzáadni a projekthez, képet vagy animációt renderelni, ezenkívül lehetőségünk van bugot bejelenteni, manuált olvasni, elérni a honlapot, stb.

2. 3D View

Ez az ablak foglalja el a legnagyobb teret. Ezen keresztül jobb oldalon látjuk magát a színpadot vagy jelenetet egy adott kameraállásból. Bal oldalon helyezkednek el az objektum(oka)t valamilyen szempontból módosító eszközök, például (a gyorsbillentyűket nem számítva) itt tudunk transzformációkat végrehajtani, árnyalni, kulcsképkockákat (keyframe) beszúrni, mozgásirányt kiszámítani, stb. A tényleges színpad mellett jobb oldalon láthatunk egy kis „+” jelzést, amellyel további opciók nyithatók meg. Itt mindenféle koordinátákat láthatunk, ami az éppen aktuálisan kijelölt objektumra vonatkozik. A 3D View legalján a nézettel, kijelöléssel, illetve objektumokkal kapcsolatos attribútumokat állíthatunk be, melynek egy része a 3D View bal oldalán is állítható volt. Emellett váltható a 3D View módja is, az alábbi módok közül dolgozhatunk a számunkra kedvezőben: *Object Mode* alapvető objektumtranszformációkra, *Sculpt Mode* „faragásra”, *Weight Paint* „súlyozásra”, *Edit Mode* csúcsok transzformációjára. Emellett az árnyalást és a kijelölt objektum pivot pontját is áthelyezhetjük.

3. Timeline

Gyakorlatilag animációkezelést tudunk itt végezni. Láthatjuk, hol tartunk a képkockák között, hány képkockával dolgozunk az adott animációban, látjuk, hol vannak a kulcsképkockák, és akármikor ráléphetünk. Bekapcsolhatunk automatikus kulcsképkocka rögzítést, illetve *Keying Setet* választhatunk.

4. Outliner

Itt gyakorlatilag a hierarchiát láthatjuk, mely a felhelyezett objektumok ne-

veit jeleníti meg. Beállíthatjuk, melyik látszódjon, illetve melyik ne, illetve az adott objektum kijelölhető legyen-e, és alkalmazzunk-e rá valamilyen rendelési megkötést.

5. Properties

Ebben a részben a kijelölt objektum tulajdonságait állíthatjuk be. Rendelési, transzformációs attribútumok konfigurálhatók kedvünk szerint, ezenkívül megkötéseket, módosítókat illetve csúcs csoportokat is létrehozhatunk. Textúrákat kezelhetünk, anyagot hozhatunk létre, illetve fizikát is alkalmazhatunk az egyes objektumokon.

A programban számtalan *hotkey* (billentyűkombináció, gyorsbillentyű) található, mellyel a fentebb említett funkciókat gyorsabban is elérhetjük (a hotkey-kről további információhoz lásd a [2, 1. fejezet] és az [1, 2., 3. fejezettől kezdve]). Kezelésük módfüggő, így egy-egy dolgot csak bizonyos módból használhatunk. A teljesség igénye nélkül felsorolhatóak a legfontosabbak:

Object Mode-ból használható hotkey-k:

Kamerakezelő hotkey-k:

- 1: y tengellyel egyirányú kamera
- 3: x tengellyel egyirányú kamera
- 7: z tengellyel egyirányú kamera
- 4: kamerát balra forgatjuk az origó körül
- 6: kamerát jobbra forgatjuk az origó körül
- 2: kamerát lefelé forgatjuk az origó körül
- 8: kamerát felfelé forgatjuk az origó körül
- 5: perspektivikus nézet és párhuzamos vetítés között válthatunk
- 0: kamera perspektíva és felhasználói perspektíva között válthatunk. A kamera perspektívánál a jelenetben (scene-ben) létrehozott kamera objektum képét fogjuk látni
- egér görgőgombja: ha nyomva tartjuk, ennek segítségével szabadon forgathatjuk a kamerát
- egér görgőgombja+SHIFT: segítségével a képsíkot (kamera síkja) alkotó tengelyek mentén történő eltolás történik az egér mozgásának irányába

Objektumkezelő hotkey-k:

- G: eltolás (translation) az egérrel
- G+X G+Y G+Z: eltolás az adott tengely mentén

R: forgatás (rotation) az egérrel

R+X R+Y R+Z: forgatás az adott tengely mentén

S: skálázás (scale) az egérrel

S+X S+Y S+Z: skálázás az adott tengely mentén

CTRL+A: a transzformációs adatoknak megfelelően transzformációkat lehet végezni az objektumon, hogy az adott transzformáció szinkronba kerüljön az adattal. Ezt főleg exportálás előtt célszerű elvégezni.

CTRL+S: mentés

CTRL+J: egyesítés (join). Olyan, mintha boolean módosítót unió művelettel alkalmaztunk volna két objektumon, annyi különbséggel, hogy nem marad meg a kiválasztott objektum eredeti állapotában.

SHIFT+D: duplikáció. használata után az egérrel pozicionálhatjuk az objektumot, majd kattintással helyezhetjük el a kívánt pontban.

CTRL+Z: visszavonás.

CTRL+P: két objektum esetén van értelme használni, a másodikként kijelölt objektumot szülőnek beállítja az elsőként kijelölt objektumnak.

H: elrejtés. Az Outlinerből kikapcsolható.

Animációkezelő hotkey-k:

I: képkocka beszúrás végezhető el, ami az objektum különféle transzformációját tárolhatja el.

Rendereléses hotkey-k:

P: renderelés, ESC-el lehet kilépni belőle

Módváltó hotkey:

TAB: Object Mode és Edit Mode között váltogat

Kijelöléses hotkey-k:

A: mindent kijelölni/mindenről a kijelölést leszedni

B+BAL EGÉRGOMB: Lenyomása után kijelölhetünk poligonokat, éleket, illetve csúcsokat egy másik beállítástól függően.

B+GÖRGŐGOMB: Az egérrel kijelölt poligonokról, élekről, illetve csúcsokról leszedhetjük a kijelölést.

Edit Mode-ból használható hotkey-k:

Szerkesztéses hotkey-k:

E: A kijelölt dolog extrudálása az egér mentén. Kattintással rögzíthető az újonnan létrejött dolog helye. Extrudálható él, csúcs, illetve poligon is.

Kijelöléses hotkey-k:

ALT+SHIFT+JOBB KLIKK: Ha ezt a kattintást egy élre végezzük el, akkor a vele közel egy irányba mutató, folytatólagos (megosztják csúcsukat az előző éllel), párhuzamos élek kijelölődnek.

ALT+JOB BKKIKK: Az adott éllel párhuzamos, de nem folytatólagos élek jelölődnek ki (de egy tengely mentén).

CTRL+JOBB KLIKK vagy SHIFT+JOBB KLIKK: több él jelölhető ki a segítségükkel (poligonból és élből többet csak a SHIFT+JOBB KLIKKel lehet).

Az alap kijelölések gyorsbillentyűi azonosak az Object Mode-ban használttal.

Transzformációs hotkey-k:

Ezek azonosak az objektumkezelő hotkey-kenél megemlített transzformációs hotkey-ekkel.

Speciális hotkey-k:

W: Itt oszthatunk fel poligonokat, egyesíthetünk csúcsokat, poligon normálvektorának irányát válthatjuk illetve ha egy pontban több csúcs van, azokból csak 1-et hagyjunk meg.

Szeparációs hotkey-k:

P: A kijelölt poligonokat tudjuk egy új objektumba áthelyezni a segítségével.

A gyorsbillentyűk ismeretében könnyebben hozzá tudunk látni valamilyen művelethez. Megkötéseink ingyenes program lévén nem lesznek, tehát csak az erőforrásaink (idő, gép teljesítménye) szabhatnak határt kitűzött feladatunknak.

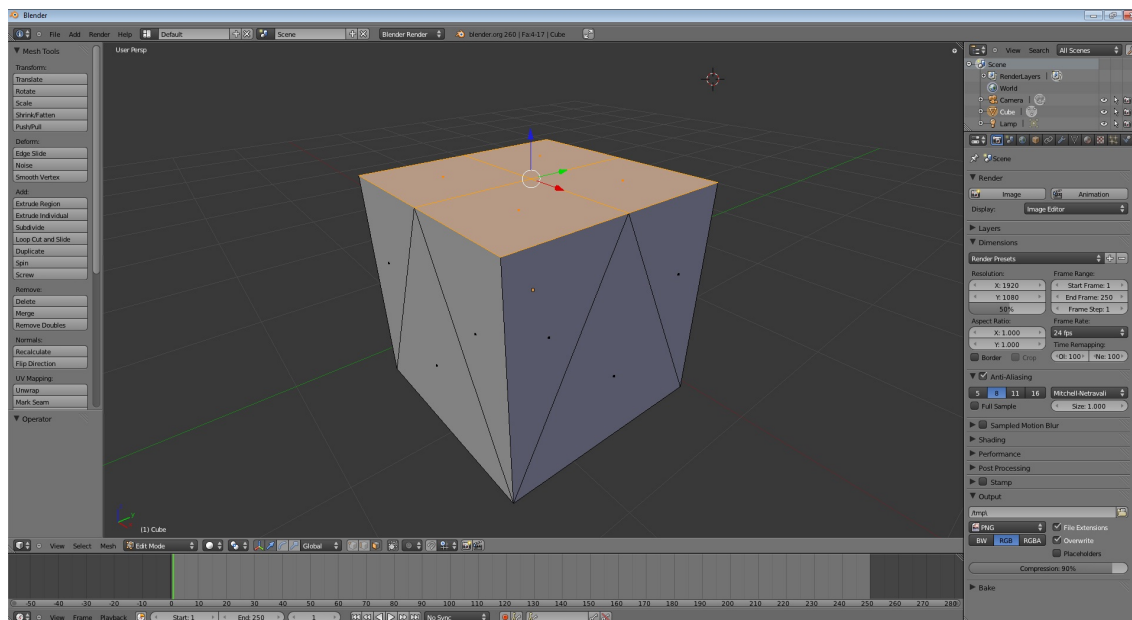
4.1. Modellezés

A *modellezés* hagyományos módja általában a következők szerint szokott zajlani. Általában modellezéskor már tervben van az, amit le szeretnénk modellezni. Ez vagy fejben elképzelt dolog, vagy egy előre elkészített rajzról, képről történhet. Az utóbbi segít bennünket a méretarányok meghatározásában is, amellet, hogy kevesebb korrekciót kell majd az elkészült modellen végrehajtani.

A kép/rajz, vagy egyéb dolog megfigyelése után (melyet egyébként akár egyenesen a Blenderbe is bevihetünk, jócskán megkönnyítve ezzel a modellezést) próbáljunk meg olyan alapértelmezett *mesh-eket* (hálót, felületet) keresni, amely az objektum egy adott részletéhez a leghasonlóbb. Ezt könnyedén megtehetjük az Info ablak Add opciója alatt. Az objektumot G billentyűvel könnyedén pozícionálhatjuk.

4.1.1. Subdivide

Amennyiben valamely poligonját részletesebbé akarjuk tenni, váltsunk Edit Mode-ba a TAB segítségével, majd kattintsunk a 3D View-ban Face Select Mode opcióra. A felosztani kívánt poligon, másnéven face kijelölése után nyomjunk W-t és válasszuk a *Subdivide* opciót. Így ha egy poligon volt kijelölve, akkor a középpontjából merőleges és függőleges tengely mentén lesz szétvágva a poligon, más szóval 4 negyedakkora poligon jön létre belőle. A vele szomszédos poligonok is módosulnak e szerint, hogy illeszkedni tudjanak hozzá. Lásd a [2, 2. fejezet] és az [1, 4. fejezet]. Hatása a 7. ábrán látható.



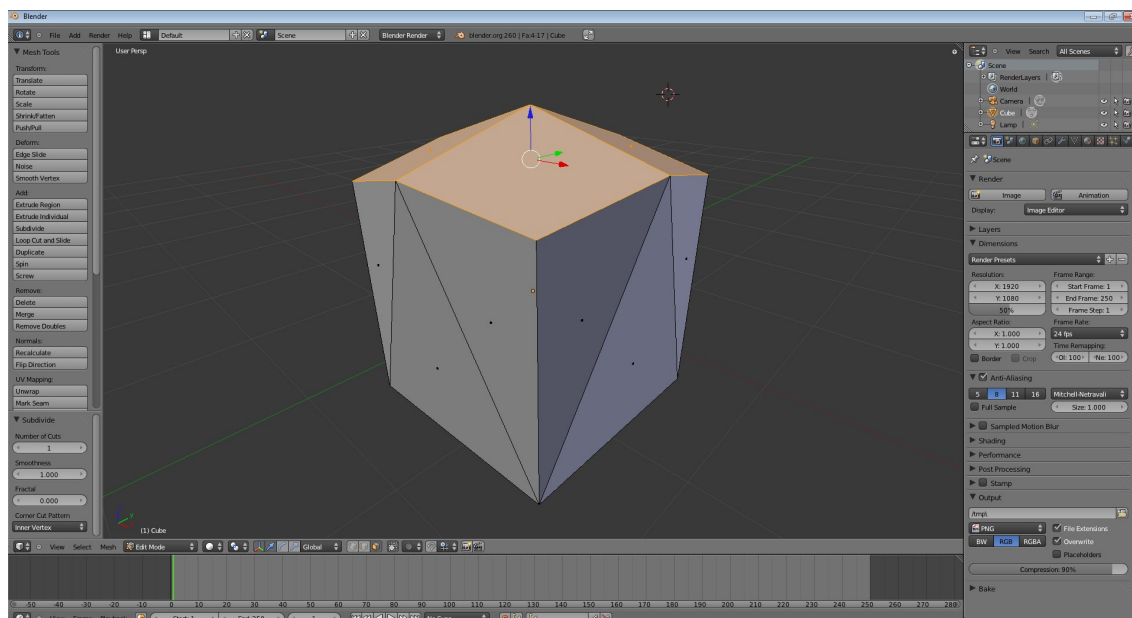
7. ábra. Subdivide után megfigyelhető a szomszédos poligonok felosztódása is

A *Subdivide Smooth* opcióval a felosztódás mellett a poligon által alkotott felület domborodik, tehát a létrejött felület középpontja lesz a legtávolabb a felület síkjához képest, és minél messzebb található egy csúcstól a középponttól, annál mélyebben fog elhelyezkedni a felületen. Végrehajtása után a 8. ábrán látható eredményt kapjuk.

A Subdivide ellentettje a Decimate.

4.1.2. Decimate

A *Decimate* valójában egy *módosító* (modifier). Alkalmazni akkor tudjuk, ha Object Mode-ra váltsunk, és az objektum kijelölése mellett megkeressük a Properties ablakot. Itt egy csavarkulcs kinézetű ábrát kell keresnünk, ahol hozzáadhatunk módosítókat



8. ábra. Subdivide Smooth hatása

az aktuálisan kijelölt objektumhoz. A legördülő listában láthatjuk a Decimate-et. Ezután a Ratio (arány) sávnál tudjuk a poligonok számának új arányát nyomon követni. A módosító alján látható, hogy az épp megadott arányban hány poligon vesz részt a mesh felépítésében. Az Apply segítségével fogadhatjuk el az objektum meshjének változásait.

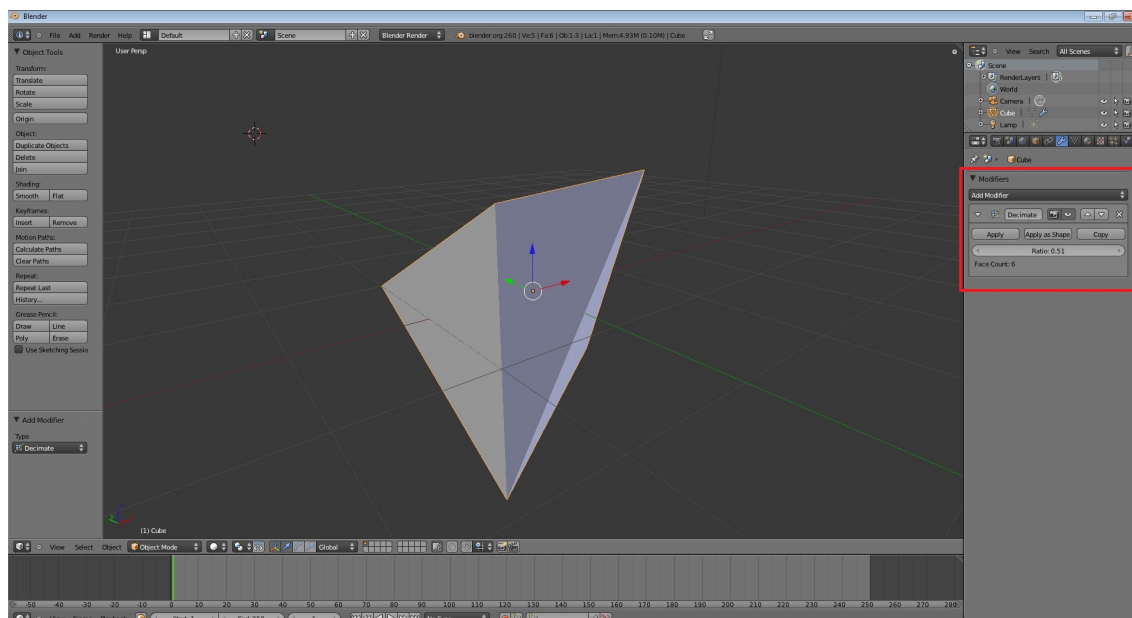
A Decimate használata viszont sajnálatos módon kerülendő, ugyanis okozhat előre nem várt dolgokat. Viszont ezeket aránylag könnyű megtalálni és kijavítani. A 9. ábrán a Decimate használat közben. Egyéb módosítókhoz lásd a [2, 2. fejezet] és az [1, 4. fejezet].

4.1.3. Transzformációk

Mint minden 3D modellező programban, így a Blenderben is transzformációk hajthatók végre egy adott objektumon, és/vagy a mesh felületében. Lásd az [1, 3. fejezet].

Az objektum Object Mode-ban skálázható az S billentyűvel, eltolható a G-vel és forgatható az R-rel. Ha mellette nyomva tartjuk például az X-et, akkor az X tengely mentén tudjuk például G nyomva tartása mellett eltolni az objektumot.

Az Edit Mode-ban az egy adott objektumot alkotó egységeket, úgy mint a csúcokat, éleket, poligonokat tudjuk transzformálni. A 10. ábrán látható az a három opció, melyek segítségével állítható, hogy mely komponense legyen kijelölhető az adott objektumnak.



9. ábra. Decimate

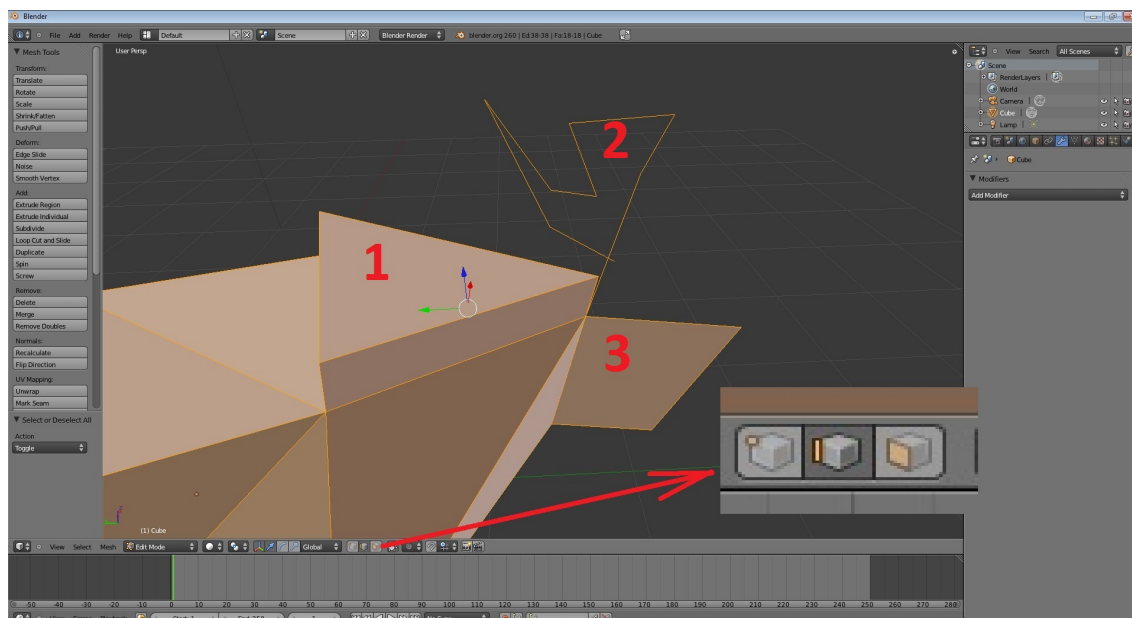
Ebben a módban használható még az E billentyű is, mely segítségével növesztetni (extrudálni) lehet az adott komponenst. Szintén a 10. ábrán 1-essel jelölve egy poligon *extrudációját* láthatjuk, mely során annyi új poligon jön létre, ahány éle volt a kijelölt poligonnak. 2-essel jelölve egy csúcs extrudálás látható, mely során új élek képződnek. A 2-es mellett 3-assal jelölve egy él extrudációja vehető ki, melynek mellékhatása pontosan egy poligon képződése. Bővebb információért lásd a [2, 2. fejezet] és az [1, 4. fejezet].

Mind az Object Mode-ban, mind az Edit Mode-ban használható a DEL. Ez előbbiben az objektum, utóbbiban a kijelölt komponens törlését jelenti. Ezáltal magával vonzhatja több komponens törlését is, bár kiválasztható, hogy pontosan mit szeretnénk a kijelölt részen belül törölni (ugyanis mondjuk ha már egy poligont jelöltünk ki, akkor párhuzamosan vele kijelölődik az őt alkotó élek és csúcsok halmaza is).

Viszont attól, hogy Edit Mode-ban a Delete menüben Allt választunk (mindent törölni akarunk), attól az objektum még létezni fog az Outlinerben, csak semmiből se fog állni. Ezért ha a teljes objektumot törölni akarjuk, váltsunk Object Mode-ra és onnan hajtsuk végre a törlést.

4.1.4. Boolean műveletek

Ezek gyakorlatilag az unió, különbség és metszet *halmazműveleteknek* felelnek meg. Azonos helyen található, mint a Decimate, ez is módosítóként érhető el, annyi különbséggel, hogy ki kell választanunk az aktuálisan kijelölt mellé egy objektumot,



10. ábra. Transzformációk és Extrude hatása.

amivel elvégezzük a műveletet. A modifier-en keresztül választott objektum a művelet elvégzése után az eredeti állapotában is megmarad. Két objektumot úgy jelölhetünk ki, ha a másodiknak kijelölt mellé a SHIFT-et nyomva tartjuk. A CTRL+J billentyűkombináció segítségével két kijelölt objektumot *join-olhatunk* (egyesíthetünk). Ennek azonos hatása lesz az unió műveletével, annyi különbséggel, hogy nem marad meg egyik objektum eredeti állapota sem (persze CTRL+Z-vel vagy szeparációval visszavonható a művelet).

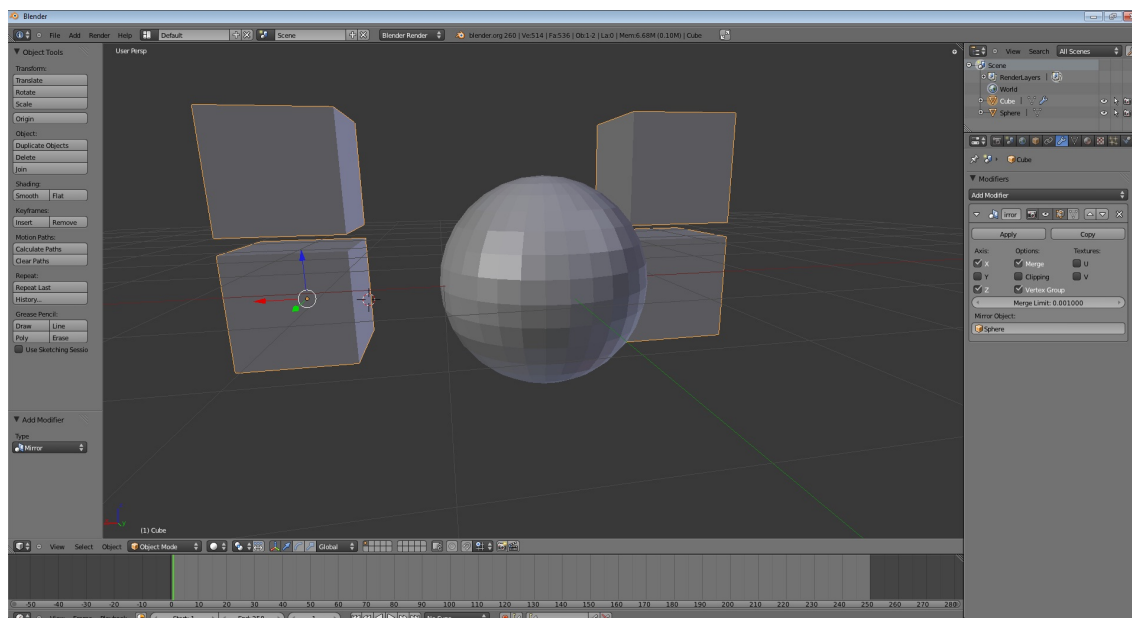
4.1.5. Tükrözés

A *mirror* (tükrözés) művelete is egy módosító. Használatához egy adott objektumot kell kiválasztani, majd mellé egy úgynevezett tükröző objektumot, aminek a középpontjára fog történni a tükrözés. Ha ezt kiválasztottuk, a *modifier Axis* részben azt a tengelyt választhatjuk ki, amelyik irányba akarjuk, hogy az aktuális objektum képe kerüljön. Ha még egy tengelyt választunk ki, akkor a már meglévő aktuális objektumot és kép(ek)et is tükrözi a módosító.

A 11. ábrán a sphere (gömb) a tükrözött objektum, és azt a cube-ot (kockát) tükröztük, amelyikben a tengelyek jelzése van feltüntetve.

4.2. Digitális szobrászkodás

A *sculptozás* (másnéven szobrászkodás) gyakorlatilag a modellezés részeként tekinthető, viszont egy sajátos technológiának fogható fel, direkt ilyen célra fejlesztett



11. ábra. Tükrözés X és Z tengelyekre.

programokkal. Ilyen például a ZBrush, illetve a Sculptris is, de számos más program is támogatja, mint a Blender, a 3D Studio Max, illetve a Maya.

Sculptozáskor egy poligonokból álló felületet tudunk könnyedén változtatni, legalábbis könnyebben, mintha külön kijelölgetnénk a poligonokat és transzformálnánk őket. A szobrászkodás segítségével szabályosabb eredményt vagyunk képesek elérni. Minél nagyobb felbontású egy felület, annál részletesebb, annál több részletet tudjuk megváltoztatni. A könnyítést az okozza, hogy gyakorlatilag egyszerre több *face* (poligon) pozícióját tudjuk módosítani.

Sculptozni Blenderben a Sculpt Mode-ban lehetséges (lásd az [1, 8. fejezet]). A módba belépéskor bal oldalon különféle opciók jelennek meg. Ezek közül a legfontosabbak:

Brush:

A *Radius* sávban állíthatjuk be az egér körül megjelenő kör méretét. Gyakorlatilag ez az ecsetünknek a mérete lesz, vagyis erre a területre fog hatni sculптоzásunk eredménye.

A *Strength*-ben határozható meg, hogy mekkora intenzitással változtassa a poligonokat egy egérekattintás. Ez viszont függ a választott ecsettípustól is.

Tool:

Ebben a részben a Sculpt Toolt (gyakorlatilag az ecsettípust) választhatjuk ki. Igen változatos eszközök közül választhatunk, a 12. ábrán a teljesség igénye nélkül láthatunk pár eszközt és a hatásait:

- 1: draw - add
- 2: draw - subtract
- 3: grab
- 4: inflate - deflate
- 5: snake hook
- 6: clay
- 7: pinch

Az eszközök hatása persze egy-egy kattintásra nem lesz látványos változás, ezért, és hogy szemléletesen látható legyen a poligonok térbeli helyzetváltozása, egy adott pozícióra többször kattintottunk. A *Draw*-nál minden kattintásra a *Strength*-be megadott értéktől függően többször történik változás az adott felületben. Viszont a *Grab*-nál semmilyen változás nem történik egy kattintásra, ott az egér pozícióját kell változtatni.

Curve:.

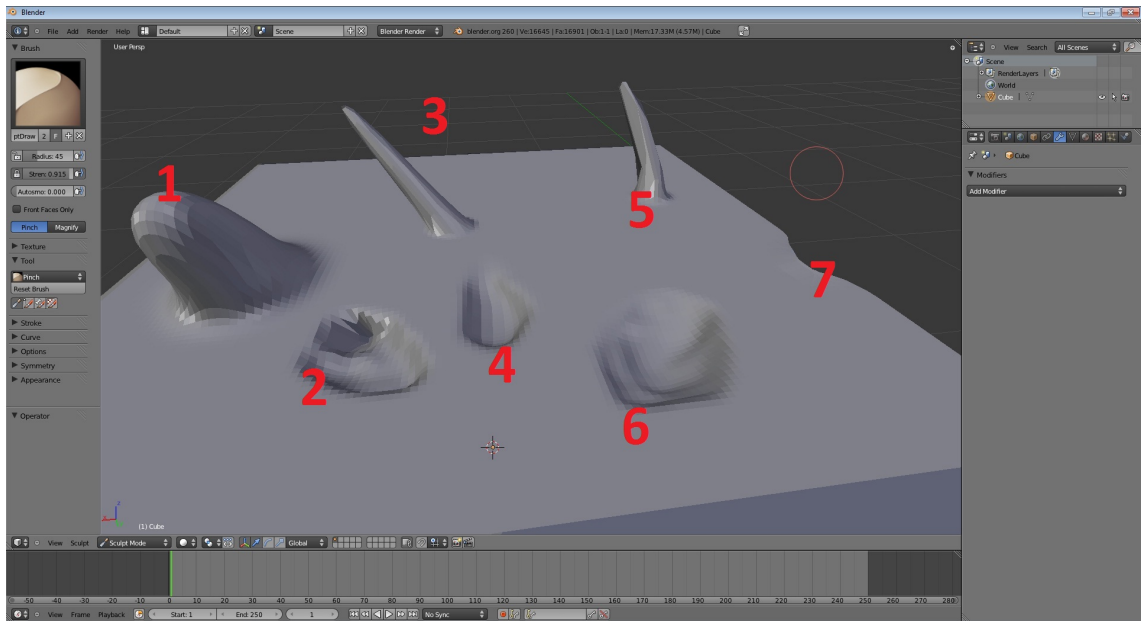
Ebben a részben állítható a sculптоzás görbéje. Gyakorlatilag a poligonok szobrászat utáni elrendeződését írjuk le a görbével. Változtatásának hatásai jól kivehetők a 13. ábrán. Itt a számokkal nem megjelölt részek a görbe alatti kis ikonok által kiválasztott alapértelmezett görbetípusok eredményei, míg az 1-essel illetve 2-essel jelölt alakzat a kép bal szélén látható görbealakkkal jött létre. A különbség közöttük, hogy az 1-es *Grabbel*, a 2-es *Draw*-val lett létrehozva. Megfigyelhető, hogy például ilyen görbe mellett a *Draw*-val történő „rajzolás” mennyivel másabb hatást kelt a módosítatlan *Draw*-hoz képest. Az alapértelmezett görbét könnyedén visszaállíthatjuk a *Reset Brush* opció segítségével, melyet a *Toolban* találunk.

Symmetry:.

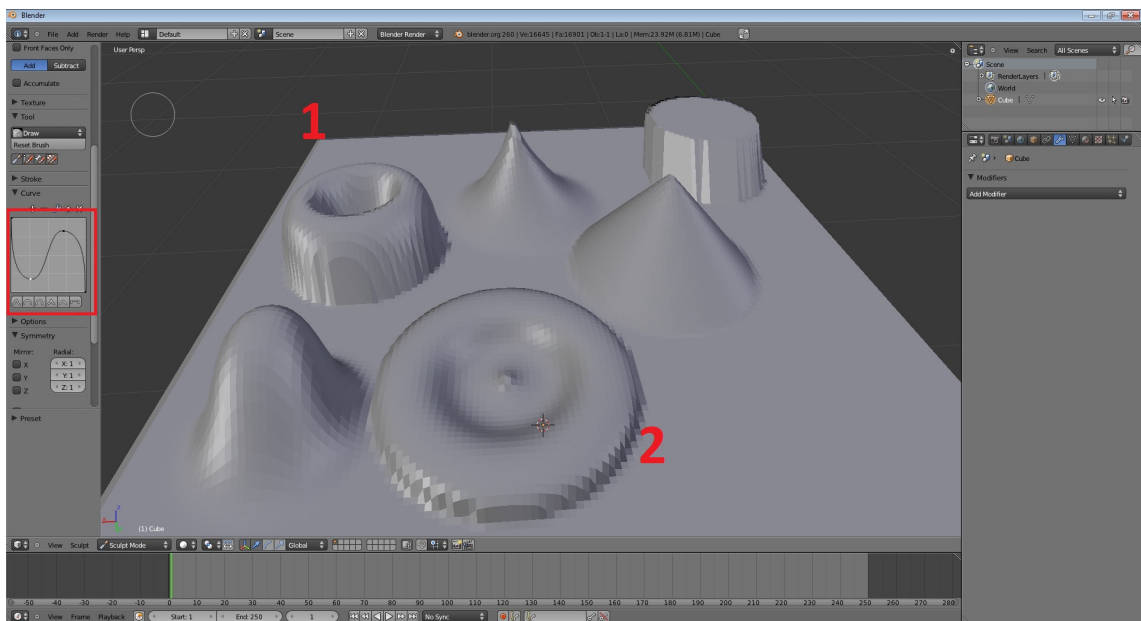
Ennek a segítségével a sculптоzott felületet biztosító objektum középpontja határozza meg a három tengely közös metszéspontját (origóját). Tehát ha egy kocka egyik oldalán szobrászkodunk és az *X* tengelyre tükrözzük rajzunkat, akkor a 14. ábra alapján fog kinézni az adott tengelyes tükrözés.

4.3. Textúrázás

Most tekintsük meg a Blenderben történő textúrázás folyamatát, mely *UV Mapping* segítségével fog történni (lásd az [1, 7. fejezet] és a [2, 3. fejezet]). Először *SHIFT+ALT+JOBB KLIKK* segítségével jelöljük körbe a textúrázni kívánt területet. Kattintsunk a *3D View* bal oldalán, az *UV Mapping* részben található *Mark*

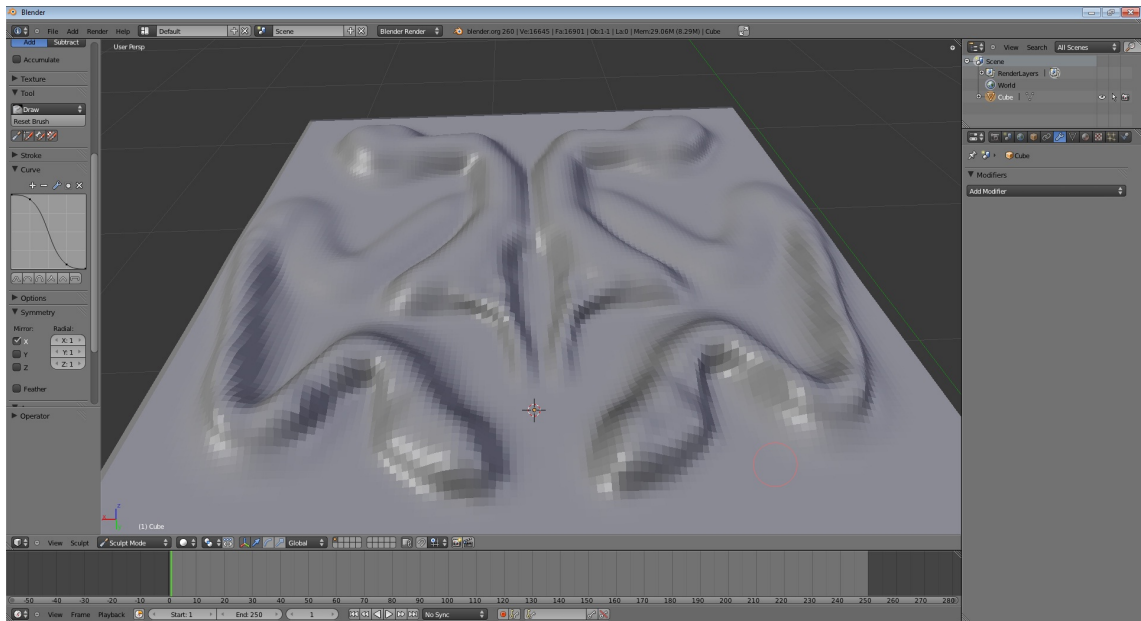


12. ábra. Sculpting Blenderben.

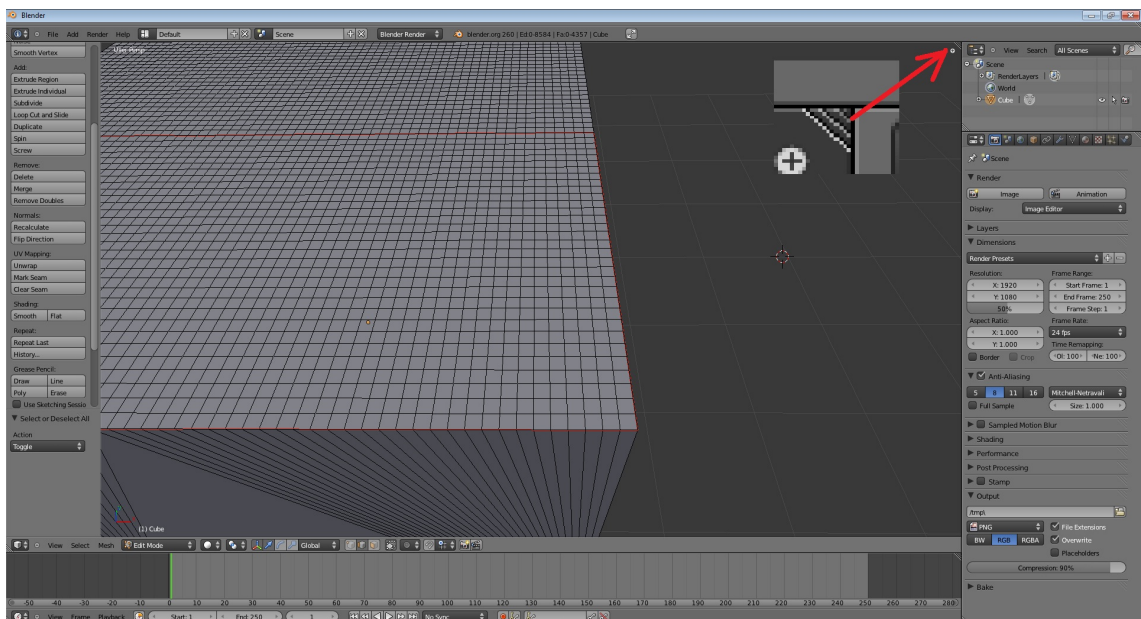


13. ábra. Curve változtatásának hatásai.

Seam opcióra, lásd a 15. ábrán. Ezt követően a 3D View jobb felső sarkában található csíkozott részt húzzuk el az egérrel, hogy új ablakunk jöjjön létre. Eltűntetése egyébként úgy történik, hogy kiválasztjuk, mely ablakot szeretnénk eltüntetni, és a vele egy sorban/oszlopban lévő ablakban található hasonló csíkozott részt ragadjuk meg az egérrel, majd húzzuk rá az eltüntetni kívánt ablakra. A megjelenő nyíl és a világos ablak fogja jelezni, hogy az az ablak fog megmaradni és amelyikre mutat, az nem.



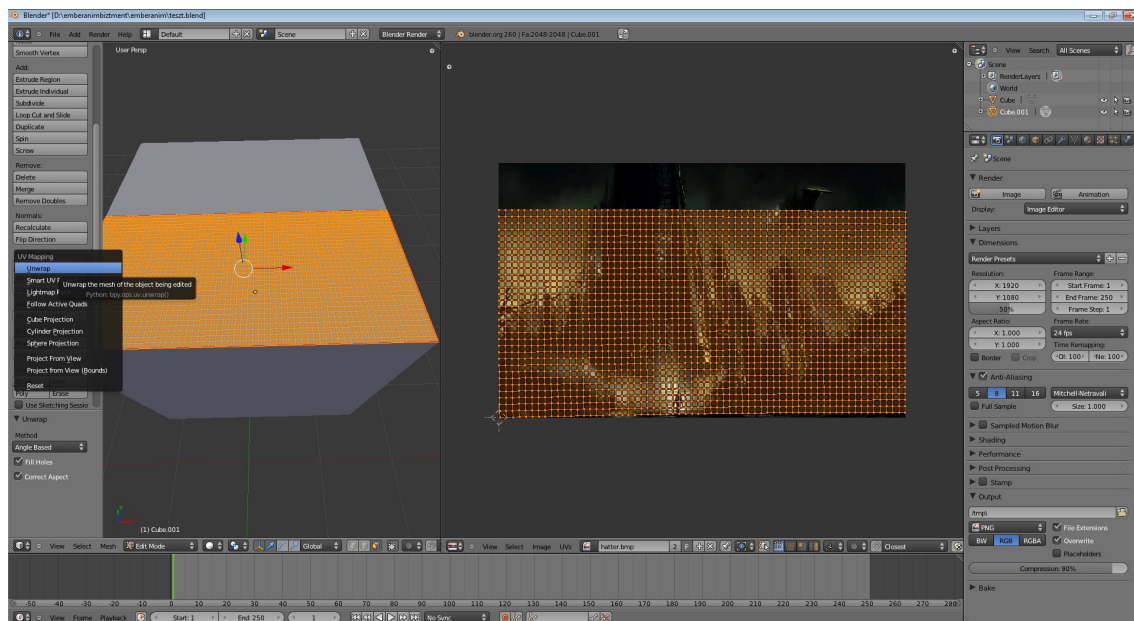
14. ábra. X tengelyre történő Draw tükrözés.



15. ábra. Seam megjelölés, új ablak nyitás

Az új ablakot váltsuk át *UV/Image Editor* módba. A 3D Viewban jelöljük ki a korábban körbekerített területet, majd a P-vel válasszuk a *Separate Selectiont*, ezzel egy új objektumot hozva létre. Válsunk ugyanitt Object Mode-ba és jelöljük ki az újonnan létrejött objektumot. Az UV/Image Editorban kattintsunk a New opcióra, ahol elég csak egy nevet megadni az új képnek, majd okézzuk le. Ekkor egy nagy feketeséget fogunk kapni. Ezután kattintsunk az Image opcióra, majd az Open Image-ben tallózzunk be egy képet, melyet textúráként szándékozunk használni. A

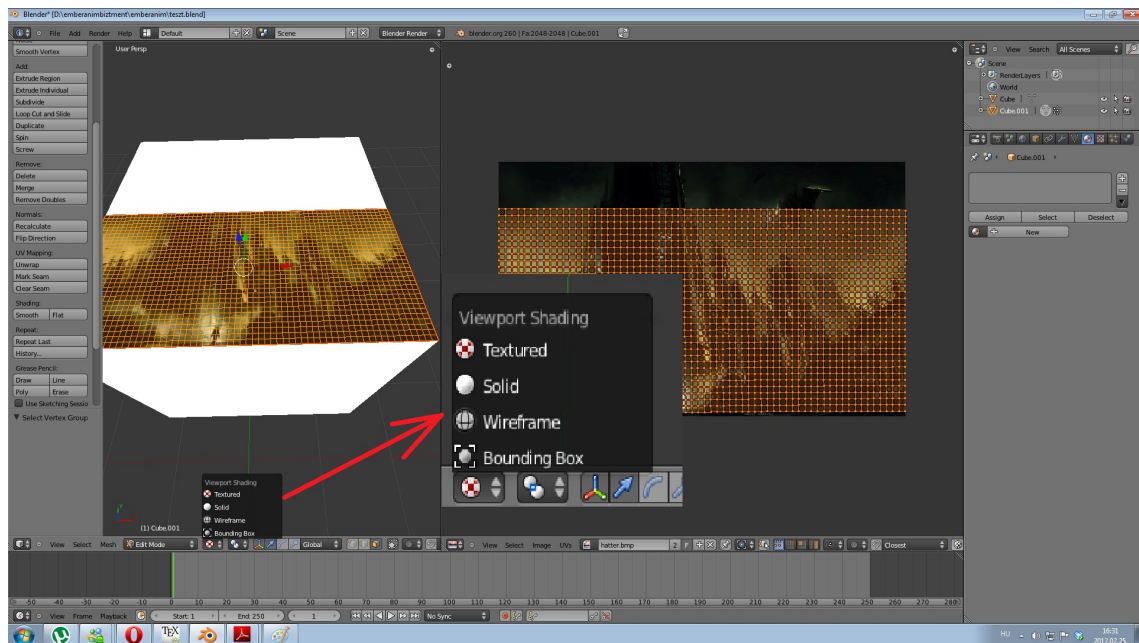
3D View bal szélén használjuk az UV Mapping *Unwrap* nevű opcióját (lásd a 16. ábrán), mellyel a mesht *kiterítve* láthatjuk az UV/Image Editor módban. Amennyiben a mesh nem sík felület volt, hanem egy gömb felülete, úgy inkább válasszuk a Sphere Projectiont. Hasonlóképp járjunk el más alakú felületeknél is, keressünk egy hozzá legjobban illő projectiont.



16. ábra. UV Mapping használata közben. A mesh kicsomagolását láthatjuk.

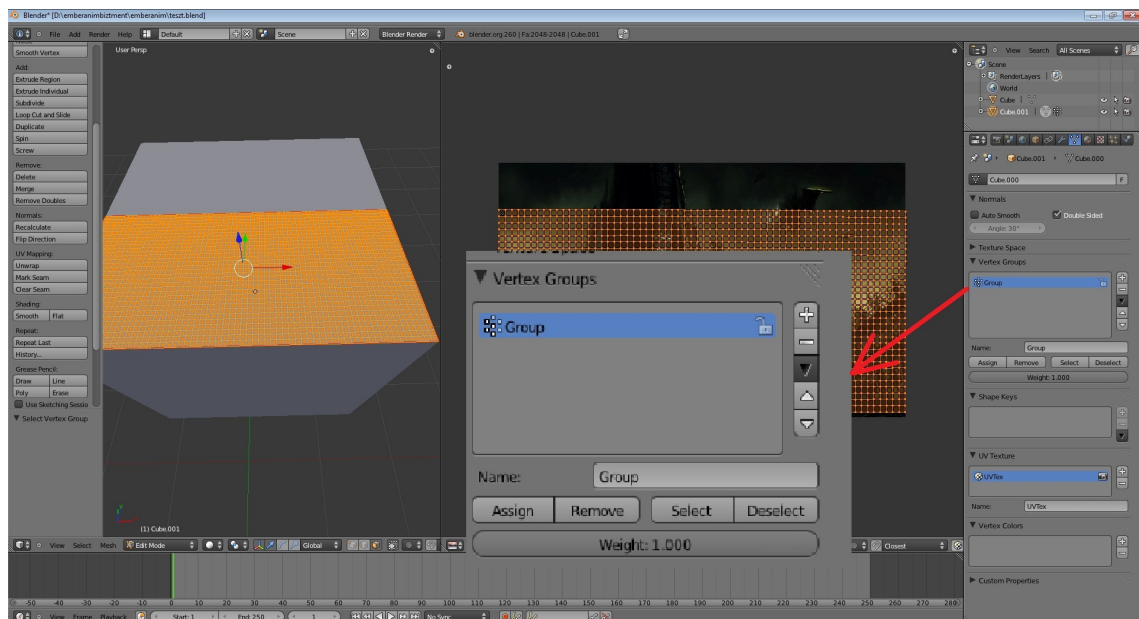
Ha ez megtörtént, a 17. ábra szerint a 3D View alján látható kis fehér gömbre kattintva válthatunk *Viewport Shadingek* között. Ez az általunk látott nézőpont árnyalása, gyakorlatilag annak a kamerának a képe, amelyből alapértelmezetten látjuk a jelenetünket. A fehér gömb a Solid árnyalást jelöli, vagyis szürke tónusú objektumokat látunk a segítségével. Ha viszont átváltunk Texturedre, láthatjuk, hogy az objektumunk mely részei vannak textúrázva, illetve milyen textúrát tartalmaznak objektumaink.

Amennyiben egy bonyolult felépítésű objektumot szeretnénk textúrázni és attól tartunk, elrontjuk a kijelölést, vagy csak egyszerűen könnyedén akarjuk kiterjeszteni a textúrát más poligonokra is, célszerű *vertex groupokat* (csúcs csoportokat) használnunk. Ehhez a Properties ablakban válasszuk az Object Data opciót, és a Vertex Groups résznél a + jel segítségével hozzunk létre egy csoportot. Ezt tetszőlegesen nevezzük át. Jelöljük ki a csoportosítani kívánt poligonokat, majd - mialatt a poligonok ki vannak jelölve - kattintsunk az Assign opcióra (lásd a 18. ábrán). Ezzel eltároltuk a csoportban a poligonokat. Select, illetve Deselect opciókkal tudjuk egyrészt csoportunk felépítését ellenőrizni, másrészt kijelölni a benne található



17. ábra. Viewport Shading átváltás Texturedre.

poligonokat, melyhez újakat is társíthatunk. Ekkor ne felejtünk el újra kattintani az *Assignra*.



18. ábra. Vertex Group

Lehetőségünk van a Properties ablakban *Materialokat* (anyagot) is létrehozni, melynek többféle textúrát adhatunk. Így ha valamelyik textúrát több helyen kívánjuk felhasználni, egyszerűen csak a számára szükséges anyagot elég kiválasztani. Viszont a textúrákkal gondunk lehet, ha exportáljuk az egész jelenetünket vagy csak

részleteit egy másik programba, így a plusz munka elkerülése végett elég csak szétválogatni az objektumokat, majd az új programban feltextúrázni őket. A materialok kezelése hasonló a vertex groupokéhoz. A legelső materiallal viszont nem szabad törődnünk, csak hozzuk létre. Ez az egész objektumfelületre fog mutatni, melynek felülete úgy csökken, ahogy az újabban létrehozott materialokban hivatkozunk ennek a felületnek az „alegységeire”, kisebb alkotórészeire. Fontos különbség közöttük továbbá az, hogy amíg a vertex groupok konkrétan vertexekre (csúcsokra) vonatkoznak, addig a materialok magukra a poligonokra, tehát két vertex közötti területet nem tudunk kihagyni a vertex groupoknál felületkijelöléskor, materialoknál viszont igen.

Unity-be történő exportálás során megmaradnak a materialok és textúrák, csak újra el kell helyeznünk őket az objektumon.

4.4. Animálás

A Blenderben képesek vagyunk animálni is (lásd a [2, 6., 7. fejezet] és az [1, 3. fejezet]). Ezt többféleképpen megtehetjük:

- Keyframe-eket (kulcsképkockákat) alkalmazva animálunk
- Valamilyen importált fájlból olvassuk ki az animációt és azt használjuk fel

Akármelyik módot is válasszuk, mintkét módszernek lesznek előnyei és hátrányai is.

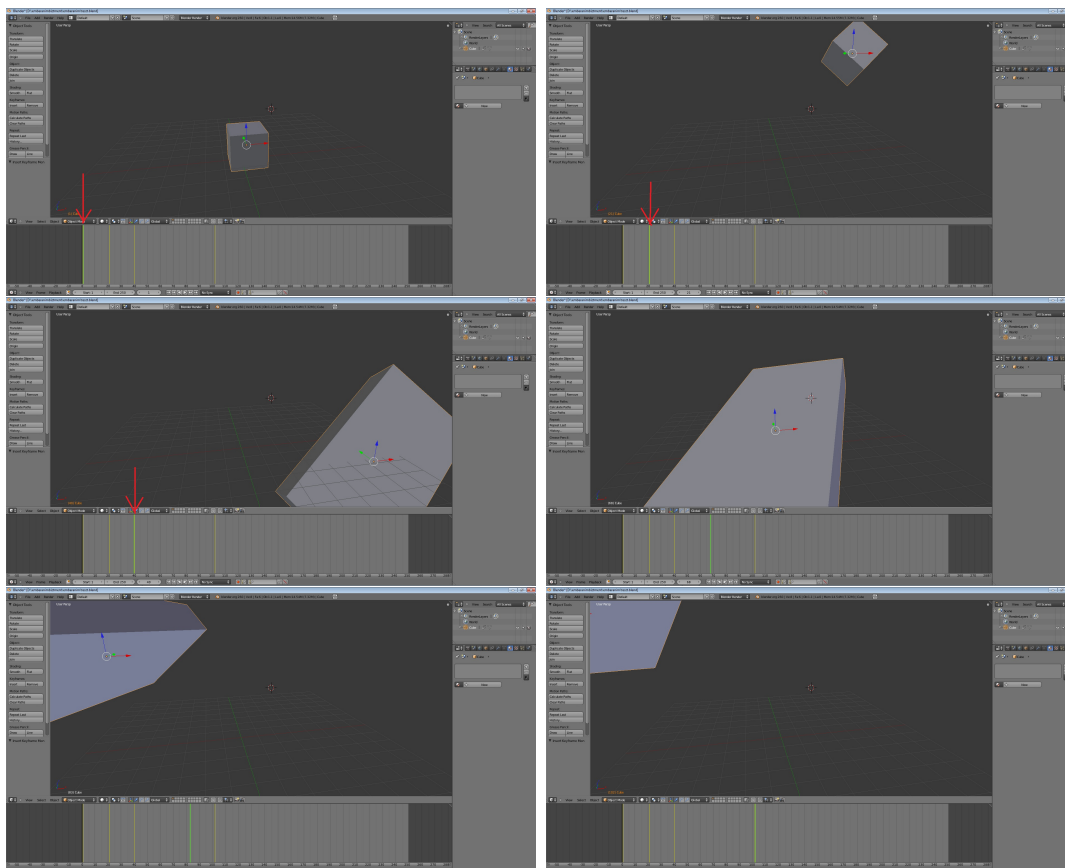
4.4.1. Kulcsképkockákkal történő animálás

- Kevésbé valóságghű animációkat hozhatunk létre vele
- Akármilyet létrehozhatunk
- Az animáció bonyolultságától függően sok időt elvehet a létrehozása

Jelöljük ki az animálandó objektumot. A *Timeline* ablakban álljunk a *x*. képkockára, ahol *x* az első és az utolsó képkocka közötti *frame* sorszáma. Ezután vigyük az egeret a 3D View felé, majd nyomjunk I-t kulcsképkocka létrehozásához, ahol kiválaszthatjuk, mely tulajdonságait rögzítsük az objektumnak az adott kulcsképkockára (nem véletlenül ez a neve, ugyanis a Blender a kulcsképkockák közötti frame-ekben saját maga számolja ki az eltéréseket, hogy egyenes arányban változzon meg az objektum minden paramétere, amíg az egyik kulcsképkockától a másikhoz ér a lejátszófej). Ha ez megtörtént, álljunk egy ezután következő képre, majd a 3D

Viewban transzformáljuk valamilyen módon az objektumot. A transzformációnak megfelelően rögzítsük a számunkra lényeges adatokat a képkockára. Egy ilyen létrehozott animáció képei láthatóak a 19. ábrán.

Miután ezt elvégeztük párszor, álljunk a legelső képkockára, majd a Timeline alján lévő lejátszás ikon segítségével megtekinthetjük animációnkat. Láthatjuk, hogy az így létrehozott animációnk messze van az élethű mozgástól, emellett ha ezzel a módszerrel egy bonyolultabb filmet szeretnénk létrehozni, igen sok munkánk lesz vele. Viszont könnyedén módosíthatjuk a már meglévő kulcsképkockáinkat is.

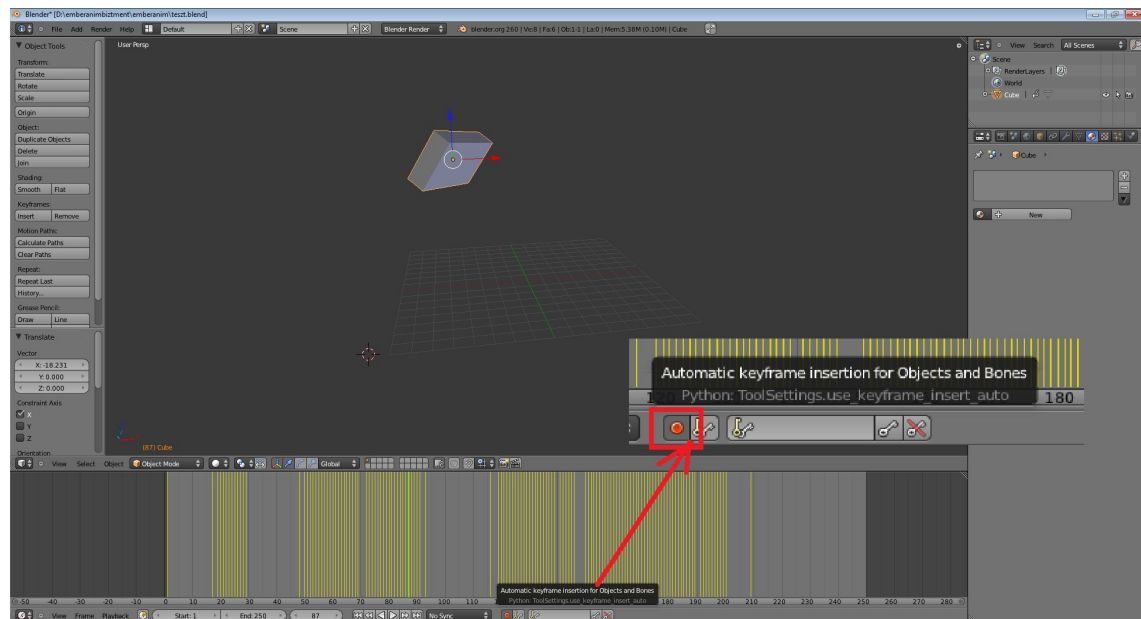


19. ábra. Keyframe Animation

Az adott *keyframe-et* el is tudjuk távolítani, ha a Timeline-on ráállunk a törölni kívántra, majd a 3D View bal szélén a Keyframes résznel a Remove-ra (törlésre) kattintunk.

Arra is van mód, hogy ne mi magunk hozzunk létre egyesével keyframe-eket, hanem ez mind automatikusan történjen. Ehhez válasszuk a Timeline alján lévő kerek és vörös *REC* gombot, majd indítsuk el az animációt. Ekkor ha transzformációkat végzünk az aktuálisan kijelölt objektum(ok)on, azok adatai rögzítésre kerülnek kulcsképkockaként arra a helyre, ahol épp a lejátszófej áll. Lásd a 20. ábrán.

Felhasználása például Unity-ben úgy fog történni, hogy az Inspector ablakban kell majd felosztanunk és megneveznünk a különféle animációk kezdő és befejező képkockájának számát.



20. ábra. Automatikus kulcsképkocka illesztés

4.4.2. Importált fájl segítségével történő animálás

Az Info ablakban a File-ban található Import opció segítségével importáljunk be egy *Motion Capture* (.bvh kiterjesztésű) fájlt. Ennek eredménye a jelenetben egy *armature* (váz), melyet egy hozzá hasonló alakú mesh-hez tudunk illeszteni (lásd a 21. ábrán).

Maga a *bvh* fájlkiterjesztés a Biovision Hierarchy rövidítése. Gyakorlatilag csontváz hierarchiáról tartalmaz információt (milyen csontok, hol és hogy helyezkednek el, stb.), illetve minden egyes csonthoz saját animációt kulcsképkockák százaival, ezreivel. Maga a fájl egy szöveges dokumentum, melynek részletei a következő formában néznek ki:

HIERARCHY

ROOT Hips

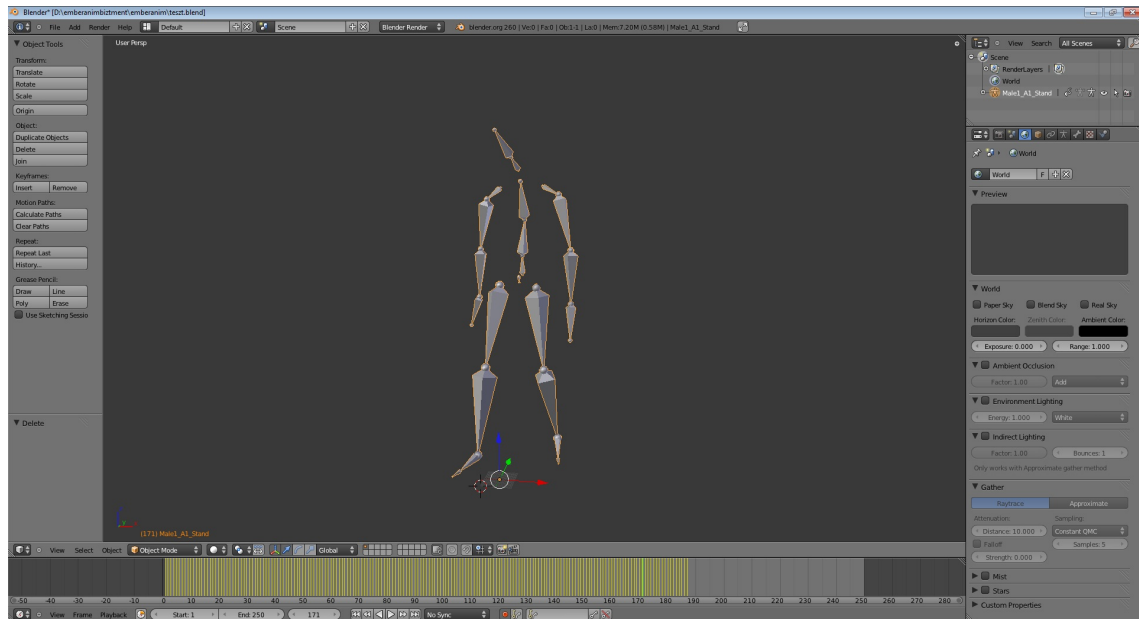
{

OFFSET 13.6774 106.653 -34.4427

CHANNELS 6 Xposition Yposition Zposition Zrotation

Xrotation Yrotation

JOINT ToSpine



21. ábra. Importált armature

```
{
  OFFSET -2.69724 7.43032 -0.144315
  CHANNELS 3 Zrotation Xrotation Yrotation
  JOINT Spine
}
...
}
MOTION
Frames: 187
Frame Time: 0.0333333
13.6653 106.685 -34.2172 -5.63641
...
```

Importálás után kattintsunk az armature-re, majd a Properties ablak Object data részben válasszuk a Rest Positiont, hogy az armature-t nyugalmi állapotba (emberalkatnál T-alak) állítsuk. Amennyiben nem T-alakú a Rest Position, valószínű, hogy másik bvh fájl kell beszereznünk. Ha ez megtörtént, létre kell hoznunk a korábban taglalt módszerekkel egy mesh-t, vagy szereznünk valahonnan, melynek alakja nagyjából hasonló az armature-höz. Úgy hajtunk végre transzformációkat a mesh-en és/vagy az armature-ön, hogy - például emberalaknál - a láb csontjai a lábban helyezkedjenek el, stb. Ezt elvégezve jelöljük ki először a mesh-t, majd SHIFT nyomva tartása mellett az armature-t is. Ekkor nyomjunk CTRL+P-t. Vá-

lasszuk a Set Parent to Armature Deform with Automatic Weights opciókat, mellyel az armature-t alkotó minden egyes csont a hozzá legközelebb eső mesh részletet (az azt alkotó csúcsokat) fogja a legnagyobb erővel mozgatni, míg a tőle egyre távolabb esőket a legkevésbé, vagy egyáltalán nem.

Ezt végrehajtva kattintsunk a mesh-re, és a Properties ablak Object data opciójában láthatjuk, hogy sok vertex csoportunk jött létre. Ezeknek nevei meg fognak egyezni a csontokkal. Figyeljük meg, hogy minden csonthoz tartozik egy vertex csoport.

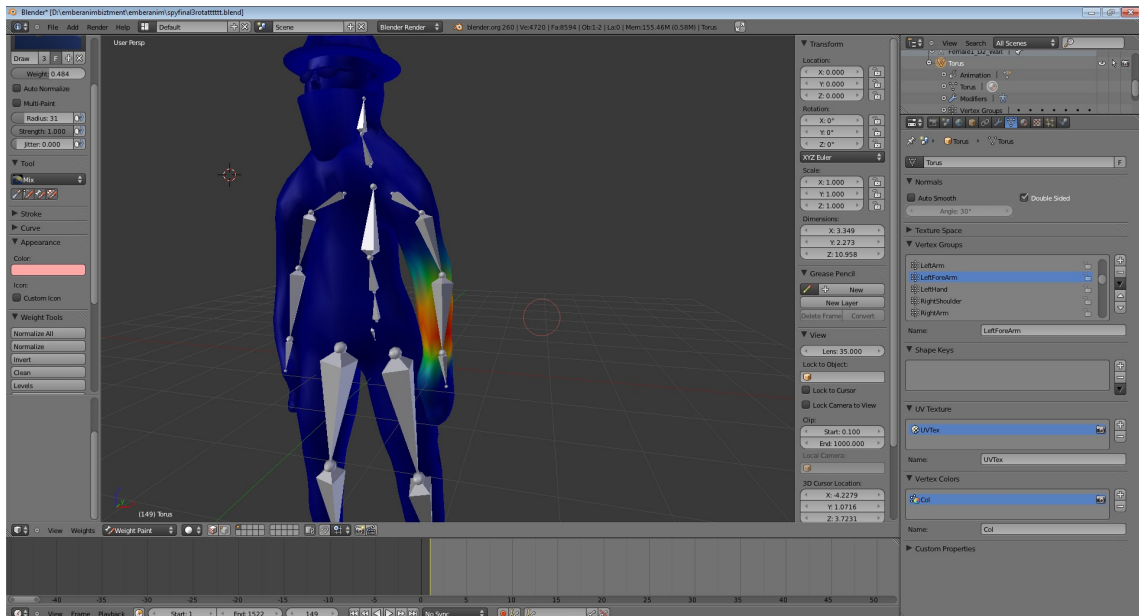
Váltsunk át a 3D View alján a *Weight Paint* opcióra. Ekkor mesh-ünk átvált egy elég tarka színű felületté. Ahogy a vertex csoportokon végigmegyünk és sorban jelöljük ki őket, láthatjuk, hogy a csontokhoz tartozó vertex (csúcs) csoportok sorban vörösdnek el a mesh felületén.

Ennek egyszerű oka van. Weight Paint módban láthatóvá válik, mely csúcsokra hatnak a legjobban az adott csont helyzetének változásai. Vörössel vannak jelölve azok a részek, melyekre a leginkább hat, melyek együtt mozognak az azonos nevű csonttal, illetve narancssárga, sárga színnel vannak jelölve azok a csúcsok, melyekre már csak fele olyan erővel hat a csont mozgása. A kézzel jelölt csúcsokra egyáltalán nincs hatással a változás.

Amennyiben valamelyik vertex csoport nem megfelelően néz ki és furcsa dolgokat művelnek azok a csúcsok, melyek a csont közelében helyezkednek el (például az egész objektum elindul, de egy-egy csúcs hátul marad, vagy más testrésszel együtt mozog), lehetőségünk van korrigálni manuálisan a csúcsok súlyozását. Weight Paint módban a 3D View bal oldalán hasonló opciók jelennek meg, mint Sculpt mód esetén. A Tool opciót hagyjuk változatlanul, helyette egyszerűen Weight, Radius és Strength opciókat változtassuk. Weighttel gyakorlatilag a festés színét állítjuk be, másnéven a súly erősségét. Radiusban az ecsetünk nagyságát, vagyis hogy egyszerre mekkora felületet tudunk festeni. A Strength opció segítségével a festés gyorsaságát, hirtelenségét állíthatjuk, hogy egy kattintásra milyen sebességgel változzon a felület színe. Hatását lásd a 22. ábrán.

Ha új animációt akarunk importálni egy már meglévő armature-höz, abban az esetben hozzunk létre egy új Blender jelenetet. Ehhez indítsuk el újra a Blendert és importáljuk be a *bvh* fílet, ezután mentsük el. Az aktuális jelenetünkben jelöljük ki az armature-t, majd az Info ablak File opciójának Append részében válasszuk ki az elmentett jelenetet (a *.blend* fílet), melyben az Action mappában megtaláljuk ezt az *action fílet*.

Amennyiben ez az *append* (csatolás) megtörtént, a Properties ablakot váltsuk át *Dopesheet* ablakra, amiben láthatjuk a csontok kulcsértékeit elmentve különféle

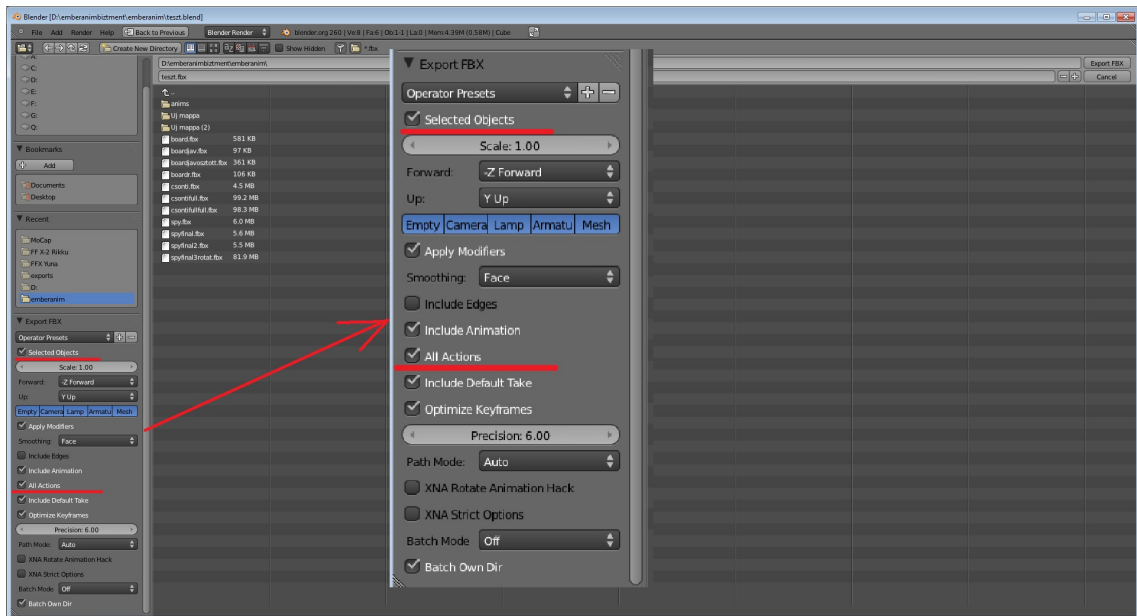


Node-okként (csomópontokként). Az ablak alján váltsunk Action Editor módba, ahol a [23.](#) ábrán 1-essel jelölt ikonra kattintva tudunk az aktuálisan linkelt action-ök között váltani. Átváltás után viszont a 2-essel jelölt F betűre kattintva mentsük le az adatblokkot, a gomb segítségével ez akkor is meg fog történni, ha nincs felhasználója a linkelt animációnak (tehát ha épp nincs hozzákötve egy armature-höz sem). Ha az F mellett bal oldalon 2-es vagy nagyobb szám áll, biztosak lehetünk abban, hogy van felhasználója az adatblokknak, létezik rá hivatkozás jelenetünkben.

4.5. Exportálás

Exportálás előtt ne felejtsük el az objektumon elvégzett transzformációkat a tárolt adatokkal egyenlővé tenni a CTRL+A opción belül található transzformációk segítségével, máskülönben az új programban anomáliákat tapasztalhatunk.

Exportáláskor számtalan formátumba menthetjük ki jelenetünket (vagy bizonyos részeit). A leghasznosabb formátum az Autodesk FBX, melyet választva majdnem biztosak lehetünk abban, hogy sikeres lesz az exportálás és az új programba történő importálás. Figyeljünk arra, hogy az All actions, illetve a Selected Objects be legyenek pipálva. Lásd a [2, 17. fejezet] és a 24. ábrán.



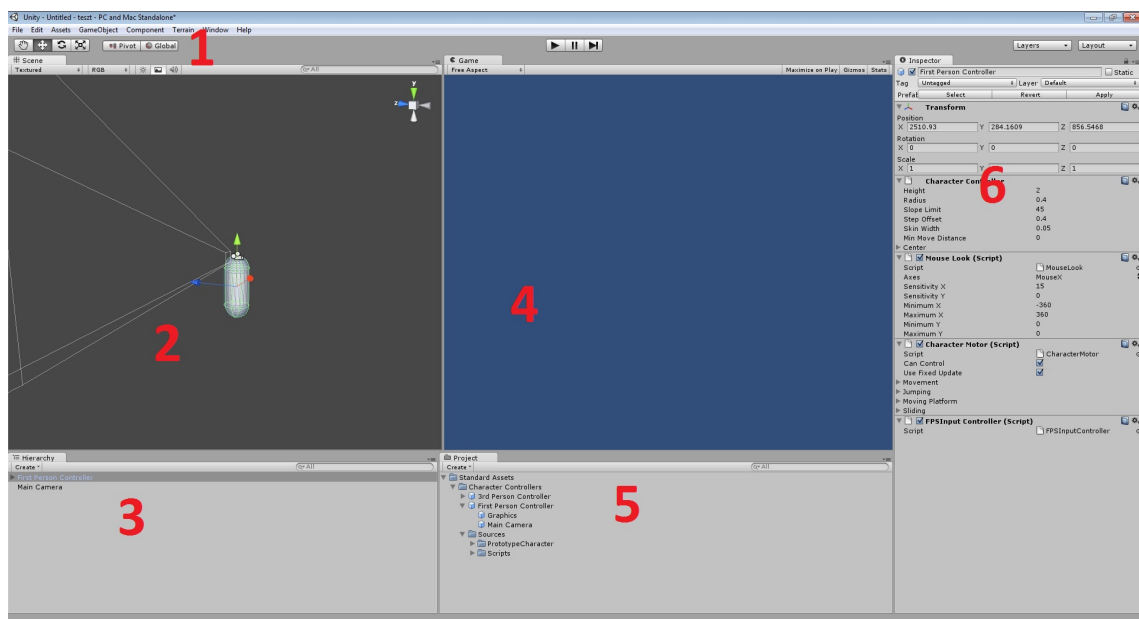
24. ábra. FBX Exportálás

5. Unity game engine

A Blender után betekintést nyerhetünk a Unity Technologies által fejlesztett *Unity game engine*-be (játékmotorba), mely segítségével a Blenderben megszerkesztett dolgainkat összerakhatjuk egy igényes, új játékká. Ez egy olyan eszköz a kezünkben, amivel esélyesen nyithatunk a hobbifejlesztők körében és jó belépőnk lehet a játékfejlesztői iparágba. Lásd [3].

A Unity (ejtsd: juniti) valójában egy integrált szerzői eszköz, mint maga a Blender, melynek a segítségével videójátékokat, valósídejű 3D animációkat és építészeti látványterveket is létrehozhatunk. Magában foglal egy szerkesztőt fejlesztésre, tervezésre és egy játékmotort, amivel az elkészült terméket futtathatjuk. A Unity ezen kívül számos díjat zsebelt be, például a Wall Street Journal 2010-es Technology Innovation Award díjat szoftver kategóriában. Számos platformra lehet vele játékot írni, úgy mint Playstation 3, Xbox 360, illetve fejlesztés alatt van a Playstation Vita támogatás is.

A Unity képernyője a következőképpen néz ki (lásd a [3, 1. fejezet]):



25. ábra. A Unity felülete

A 25. ábrán látható képernyő számozott részei a következőket jelentik:

1. Vezérléssel kapcsolatos opciók, transzformációk:

A File részben a jelenettel kapcsolatos beállításokat találjuk, amely a szokásos mentés, betöltés opciókon túl a célplatformra történő exe létrehozás képességét is magában foglalja.

Az Edit (szerkesztés) részben található megszokott gyorsbillentyűk mellett (másolás, kijelölés, duplikáció, stb.) láthatjuk, hogy a projekt beállításában lehetőség van az irányítással kapcsolatos (Input), hangért felelős (Audio) és minőséggel foglalkozó (Quality) opciókat is szerkeszteniük.

A Quality opcióban nagyon fontos, hogy a beállított minőségben belül a Pixel Light Count értékét például 100-nak adjuk meg, ugyanis ha ez az alapértelmezetten van, könnyen lehet, hogy bizonyos fényforrások nem lesznek hatással a poligonok színének változására.

Az Assets részben előre legyártott illetve saját vagy mások által készített package-eket (csomag) importálhatunk, illetve exportálhatunk, valamint *assetekkel* (dolgokkal) is megtehetjük ugyanezt. Az asset egy csomag része lehet. Tartalmazhat textúrákat, objektumokat, animációs file-okat, stb.

A GameObject részben *gameobjecteket* (játékelemeket, objektumokat) hozhatunk létre. Ezek előre legyártott és saját objektumok is lehetnek, illetve lehetnek üresek is. Ezek hasonlóak, mint a Blender Add opciójának Mesh részében található elemei, de létrehozhatunk 3D Textet illetve GUITexture-t is. Utóbbi jó például a kamera mozgását követve élet vagy lőszer kijelzőként is.

A Component részben érdemes megemlíteni a Physics és Audio opciókat. Az Audio-ban Audio listener tulajdonsággal ruházhatunk fel egy gameobject-et, a Physicsből különféle colliderekkel (ütközési zónákkal) ruházhatjuk fel a kijelölt objektumot, valamint rápakolhatunk olyan *Joint-okat* (illesztési helyeket), melyekkel két vagy több objektumot rögzíthetünk egymáshoz.

A Terrain részben létrehozhatunk egy olyan síkot, melyet könnyedén formálhatunk (gyakorlatilag egy sculптоzható felület, de pályarészek kialakítására van kitalálva). A Window részben *Lightmappingot* (fénytérképet) hozhatunk létre, gyakorlatilag a fényforrásoktól árnyékolt felületek lenyomatait bake-elik (rásütik) egy képre, mely később térképként szolgál és optimálisabb megoldást nyújt, mintha valós időben kellene a grafikus motornak kiszámítania, mely felületrészek vannak árnyékban és melyek nem.

A Help rész általános segítségnyújtást, információkat nyújt a felhasználó számára. A menüsáv és eszköztár alatti részben bal oldalon a kamera mozgatására szolgáló kéz ikont láthatjuk (melyre GÖRGÖGOMB segítségével gyorsan válthatunk), valamint az alapvető transzformációkat láthatjuk, balról jobbra eltolás, forgatás illetve skálázás.

2. Scene:

Ez az ablak gyakorlatilag egyenértékű a Blender 3D View ablakával.

3. Hierarchy:

A Blender Outliner ablakához hasonlít. Itt viszont semmi nem állítható, csak kijelölhető.

4. Game:

Az aktív kamerából ezt a képet látjuk játékon belül. Ha fel van helyezve egy *First Person Controllerünk* a jelenetre, akkor az abban lévő kamera lesz az aktív kamera és azt láthatjuk a Game ablakban, amit ő lát. Ezenkívül itt nem láthatók a hivatalosan *Gizmo-knak* (bigyóknak) elnevezett jelölések sem, mint a fényforrást jelző kis nap képe, vagy a kamera ikonja.

5. Project:

Itt találhatóak azok a package-ek, textúrák, materialok, objektumok (és mindenféle asset) illetve scriptek („forgatókönyv”, valójában forráskód), melyeket felhasználhatóvá tettünk projektünkben.

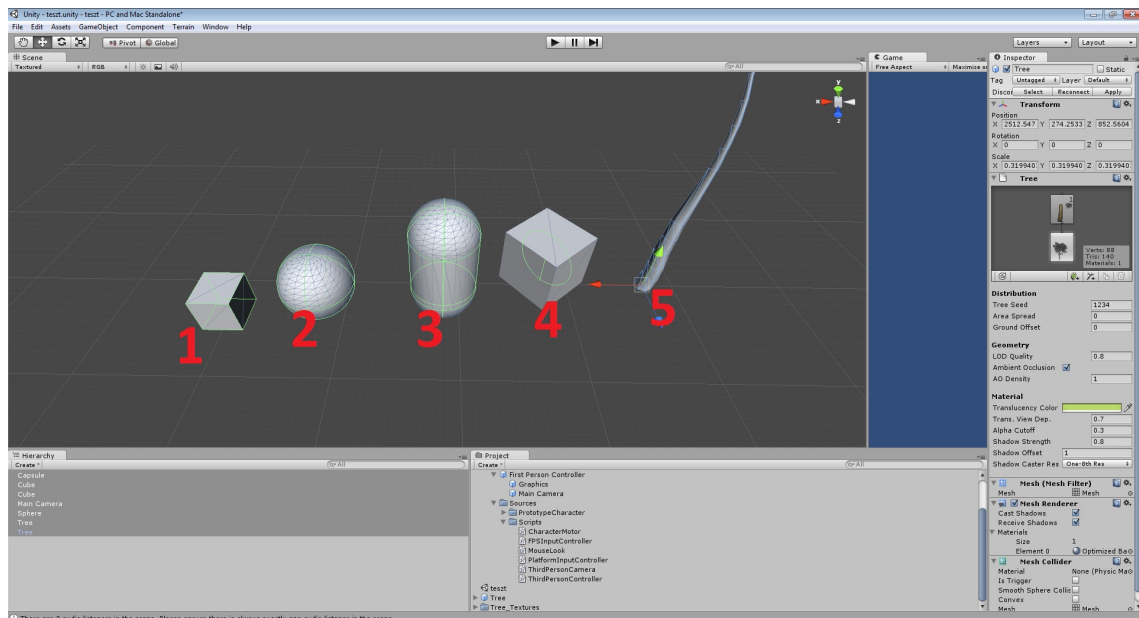
6. Inspector:

Az aktuálisan kijelölt objektumról kaphatunk részletes információkat. Az ablak felső részében az objektum neve mellett láthatóvá illetve láthatatlanná tehetjük. címkét tehetünk rá, illetve ráhelyezhetjük egy *layerre* (rétegre). Előbbi scripteknél az objektumra hivatkozásnál, utóbbi például fényforrások szeparációjánál jelentős. Alatta transzformációs értékeket láthatunk (ezek lehetnek helyi koordináták is, nem csak világkoordináták, amennyiben ez egy gyerekobjektuma egy szülőnek, vagyis ha a hierarchiában valamelyiben helyezkedik el). Ezalatt az objektumra húzott különféle scriptek publikus mezőit állíthatjuk be, illetve egyéb tulajdonságok lehetnek, saját opciókkal.

Ezek ismeretében rá tudunk térni a program használatára.

5.1. Ütközési zónák

Másnéven *colliderek*. Az ütközési zónák segítenek abban, hogy ha egy objektumra hat a gravitáció (Inspectoron belül láthatjuk, hogy a *Rigidbody*, másnéven merev test komponensében a Use Gravity be van pipálva), akkor ne zuhanjon a végtelenségig, ne essen át a pályán, hanem álljon meg, ütközzön neki valaminek (lásd a [3, 1. fejezet]).



26. ábra. Különféle colliderek

Ütközési zónából többféle van, de alakja, elhelyezkedése függhet attól, hogy egy impoltált objektumot az exportfájl készítő programban hogy transzformáltunk. Ugyanis előfordulhat, hogy az ütközési zóna transzformált lesz, ekkor a szülő programban kell transzformációkat végrehajtanunk. Alapvetően hatféle collider érhető el Unityben, ahogy a 26. ábrán is látható, a *character controller* (3), amely egy speciális collider, *box* (1), *sphere* (2), *capsule* (3), *mesh* (5) és *wheel collider* (4). Ezeket akkor láthatjuk, ha kijelölünk egy objektumot. Amennyiben nincs rajta semelyik collider, rátehetünk egyet vagy többet is. Ehhez válasszuk a Component opcióból a Physics részt, melyben megtaláljuk a különféle colliderek listáját. Ha már van egy collider az adott objektumon, a Unity rá fog kérdezni, hogy szeretnénk-e helyettesíteni az előzőt az új colliderrel vagy hozzá szeretnénk adni az előző(ek) mellé.

A Character Controller olyan collider, melynek kapszula alakja van. Lényegében ez egy capsule collider. A character controller a First Person Controller (első személyű vezérlő) és Third Person Controller (harmadik személyű vezérlő) colliderje. Mivel mindkét dolog egy emberalakot feltételez, emiatt kapszula alakúak.

A Capsule collider és a character controller colliderek között viszont alakjuk megegyezése mellett van egy nagy különbség, mégpedig az, hogy más opcióik vannak. Mindkettőnél állíthatjuk a magasságot és sugarat (szélességet), viszont amíg a character controllernél különféle mozgással kapcsolatos opciókat találunk (az irányíthatóságból adódóan), addig a capsule collidernél meghatározhatjuk azt, hogy trigger-e (kapcsoló). A triggerrekről a későbbiekben lesz szó.

A Box, Sphere, Mesh és Wheel colliderek esetében is létezik a trigger jelenlétére irányuló opció. A box collider egy téglatestet alkot, amelyen belül fog elhelyezkedni az objektum. Ezenkívül x, y illetve z koordinátában adhatóak meg méretei.

A Sphere collider egy gömbalakot formál. Sugarát lehetséges egyedül szabályozni. Szélessége miatt, illetve keskenység esetén előforduló, objektumból kilógó részek miatt csak gömb alakú objektumokra érdemes használni.

A Mesh collider az objektum alakját veszi fel, illetve pont annyi poligonból fog állni, mint amennyiből az objektum áll. Célszerű bonyolult felépítésű, animációnélküli objektumokhoz, viszont amelyek valamilyen mozgást hajtanak végre, azoknál nem ajánlott, ugyanis az objektum alapállapotának a képét fogja felvenni a collider és nem fog az objektummal együtt változni.

A Wheel collider kör alakú, és a sugara mellett megadhatjuk, hogy mennyi legyen a suspension distance (kerékfelfüggesztési távolság), a suspension spring (felfüggesztés rugalmassága), amin belül a rugalmasság, a csillapítás és a felfüggesztés nyugalmi állapotának a helyzetét adhatjuk meg. Ezenkívül szabályozhatjuk a kerék tömegét, illetve a súrlódás milyenségét. Ezekből az attribútumokból következtethető, hogy a wheel collider tipikusan előnyös autók kerekeinél, mely egy könnyen módosítható felületet biztosít például autószimulátorok fejlesztéséhez.

5.2. Scriptek

A scriptek, másnéven forgatókönyvek gyakorlatilag események leírására szolgálnak, ezért hívják így őket. A scriptek programkódok, ebből adódóan megkövetelik a programozói tudást. Programoktól függően számtalan programozási nyelv segítségével hozhatunk létre scripteket, melyek az adott programon belül, a programhoz adott *API-t* (application programming interface - alkalmazásprogramozási felület), vagyis az előre megírt metódusokat, függvényeket felhasználhatják, illetve újakat tartalmazhatnak. Megírásuk egy, a kód programozási nyelvéhez optimalizált *IDE-ben* (integrated development environment - integrált fejlesztői környezet) történik.

Az IDE-k főbb tulajdonságai a következők lehetnek:

Code completion:.

Kódkiegészítés.

Code folding:.

Kód blokkok egységbe zárása.

Code browser:.

Segítségével meg lehet nézni, milyen osztályok, metódusok, property-k vannak projektünkben.

Code intantation:.

Sorok behúzása.

Search & replace:.

Segítségével bizonyos szavakra, szórészletekre rákereshetünk a kódra, illetve változók, metódusok neveit lecserélhetjük.

Debugger:.

A kód soronkénti futtatását teszi lehetővé, segítségével figyelni lehet a változásokat.

Kommentezés:.

Kiegészíthetjük kódunkat magyarázatokkal.

Syntax highlight:.

Kódszínezés az átláthatóság érdekében.

Code snippet:.

Rövid (mondjuk kétbetűs) szavak kóddá kiegészítése.

A Unity IDE-je a Scintilla nevű ingyenes kódszerkesztő Unity-re módosított változata, a *UniSciTe*. Segítségével írhatunk a Unity-s projektünkhöz C# (Unityscript), Java, Boo nyelveken is scripteket. Természetesen ezekhez a scriptekhez az adott nyelvhez alkalmasabb IDE-t is választhatunk (C# script esetén például a Microsoft Visual Studiot, mely egy igen fejlett IDE-ként van számontartva), a Unity csupán egy alternatívát kínál a kódok írásához (lásd a [3, 1. fejezet]).

A UniScite tulajdonságait tekintve tartalmaz code completion-t, code folding-ot, code intantationt, search & replace-t, kommentezést, syntax highlightot is.

Scriptet könnyűszerrel létrehozhatunk, ha a Project ablakban jobb egérgombbal kattintunk és kiválasztjuk a Create opció belül a C Sharp scriptet (vagy a másik kettő közül valamelyiket).

Magukat a scripteket tekintve a Unityscriptek alapértelmezetten a következők szerint épülnek fel:

```
using UnityEngine;
/*namespace, amely a Unity API-ját tartalmazza*/

using System.Collections;
```

```

/*különbféle adatszerkezeteket tartalmazó névtér*/

public class ScriptNeve : MonoBehaviour {
    /*publikos osztály, melyet a MonoBehaviour osztályból
    származtattunk le*/

    void Update()
    {
        /*minden egyes képkocka kirajzolásakor
        lefutó metódus. Ciklusnak fogható
        fel. Emiatt nem érdemes egyéb
        ciklusokat beleírni, mert az az FPS
        (frame per screen – képkocka per
        másodperc) érték csökkenését
        eredményezheti*/
    }

    void Start()
    {
        /*a script lefutásának legelején fut
        le, itt változódeklarációkat (változók létrehozását)
        végezhetünk.*/
    }

}

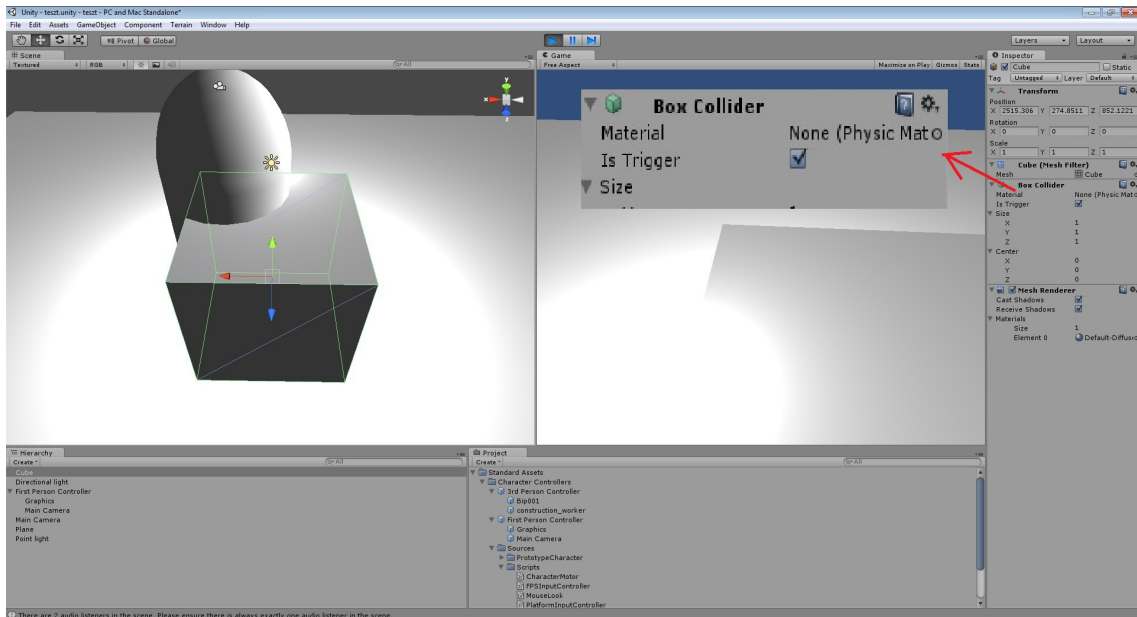
```

Az így létrehozott script egy szöveges állomány lesz, melynek (C# script esetén) „cs” kiterjesztése van. Természetesen szabadon módosítható IDE nélkül is, viszont akkor kevésbé fogjuk tudni átláthatóan szerkeszteni illetve hibákat keresni benne.

5.3. Kapcsolók

A *trigger*ek (kapcsolók) olyan colliderok, amelyeken áthatolhat egy másik collider vagy trigger. Erre az áthatolásra viszont már tudunk valamilyen, az objektumra aggatott scripten belül hivatkozni (a hozzárendeléshez elég egyszerűen az objektumra húzni a megfelelő scriptet). Segítségével elérhető az, hogy a behatoló objektumnak a környezet számára elérhető (publikus) tulajdonságaira hivatkozhatunk, máshogy megfogalmazva megváltoztathassuk a behatoló vagy más objektumok belső állapotát. Könnyebben elképzelhető, ha úgy tekintünk rá, mint egy konkrét kapcsolóra, melyet ha átkapcsolunk, valamilyen eseményt vagy eseményeket indíthatunk el (mint egy villanykapcsoló).

Jelentősége akkora, hogy gyakorlatilag új és régi játékokban egyaránt használják. Triggerekkel valósítható meg egy mozgásérzékelős ajtó szimulációja, egy *scriptelt event* (esemény) elindítása, ebből adódóan mondhatjuk azt, hogy a legtöbb, valamilyen történéshez köthető, ezáltal létrejövő esemény (mely akár látható, akár láthatatlan, háttérben zajló is lehet) triggerelt esemény, ugyanis valaminek hatására hajtódik végre feltételesen. Lásd a [3, 4. fejezet] és a 27. ábrán.



27. ábra. Trigger

Működésének konkrét bemutatásához tekintsük meg a következő scriptet. Ezt egyszerűen drag & drop módszerrel (bal egérgombot nyomva tartva elengedni a kívánt objektum felett) húzzuk arra az objektumra, amely a triggerünk.

```
using UnityEngine;
using System.Collections;

public class Script1 : MonoBehaviour {

    public bool b=true;
    /*egy b bool típusú változónk van, mely igaz értéken áll*/
    void OnTriggerEnter(Collider other){
        if(other.name=="First_Person_Controller"){

            Script2 s=other.gameObject.GetComponent<Script2>();
            s.b=this.b;
        }
    }
    /*
    Ez a metódus akkor fut le, amikor
```

```

        egy tetszőleges colliderrel
        rendelkező objektum behatol
        a scriptet tartalmazó objektum
        colliderébe. Itt a megfelelő kezelés
        érdekében szükséges megvizsgálnunk,
        hogy az az objektum hatolt-e be,
        amelyikre a feladatunk
        szempontjából figyelnünk kell.
        Ekkor meghívjuk azt a Script2-t,
        ami a behatoló objektumon van,
        majd megváltoztatjuk a b nevű
        attribútumát (tulajdonságát) a
        saját értékünkre
        */
    }
}

void OnTriggerExit(Collider other){
    if(other.name=="First_Person_Controller"){

        Script2 s=other.gameObject.GetComponent<Script2>();
        s.b!=this.b;
        /*
        Ez értelemszerűen egy behatoló
        objektum elhagyásakor fut le,
        mely lehet, hogy épp az előzőleg
        behatoló volt, de
        az is lehet, hogy egy régebben
        behatolt objektum. Ezért
        fontos itt is megvizsgálni, hogy
        az hagyta-e el
        az objektumot, amelyikre mi
        gondolunk.
        Ezután az előzőleg megváltoztatott
        tulajdonságát ellentétes
        értékűre állítottuk.
        */
    }
}
}

```

A behatoló objektumnak a trigger szempontjából teljesen mindegy a felépítése, csupán az a fontos számára, hogy a feltételvizsgálatkor a neve megegyezzen a vizsgált névvel, valamint számára elérhető (publikus) legyen az az attribútum, melynek

értékét át szeretné állítani a sajátjára. Ezáltal az egyetlen, működéshez szükséges dolgot fog példánkban tartalmazni ez a kód:

```
using UnityEngine;
using System.Collections;

public class Script2: MonoBehaviour {
    public bool b;
    /*a mező, melynek értéke behatoláskor
    megkapja a trigger értékét, majd
    kilépéskor az ellentétes értéket kapja
    meg
    */
}
```

6. Táblajáték létrehozása Blender és Unity segítségével

Az eszközök bemutatásán túl vagyunk. Most készítsünk egy *táblajátékot*.

A feladat legyen a következő:

Hozzunk létre egy $n \times n$ -es táblát, melynek a két oldalán két *sereg* álljon. Az egyik a saját bábuinkból, a másik a számítógép bábuiból épüljön fel. Tartalmazzon a tábla *üres* mezőket, melyekre a bábuk léphetnek, illetve *tavat*, melyre nem. A tó a pálya középső részén helyezkedjen el. A tábla ez alapján így fog kinézni, ahol „E” jelöli az ellenséget, „U” az üres területet, „T” a tavat, „S” pedig a saját bábuk helyét:

E	E	E	E	E	E
E	E	E	E	E	E
U	U	T	T	U	U
U	U	T	T	U	U
S	S	S	S	S	S
S	S	S	S	S	S

A sereget a következőképp lehet felosztani, a *bábuk* rangja alapján növekvő sorrendben (zárójelben a hozzá tartozó rangszám):

- Kém (0)
- Marsall (1)
- Tábornok (2)
- Ezredes (3)
- Hadnagy (4)
- Aknaszedő (5)
- Felderítő (6)

Emellett van két, a helyzetüket nem változtató bábu, a bomba és a zászló. Kezdetben nem ismerjük az ellenséges karakterek *rangsámát*, így „vaktában tapogatózva” saját szerencsénkben bízva kell előretörnünk.

A játék lényege, hogy megszerezzük az ellenség zászlaját. Ennek eléréséhez át kell a saját karaktereinknek verekedni magukat az ellenség erőin. Ezt úgy tehetik meg, ha minden körben vagy oldalirányba, vagy előre, hátra lép egyet. Amennyiben az ellenség egyik kiszemelt karakterére lép, kiderül az ellenséges karakter rangszáma, mely összehasonlítódik a saját bábunkéval. 4 féle eset kifejtése fontos:

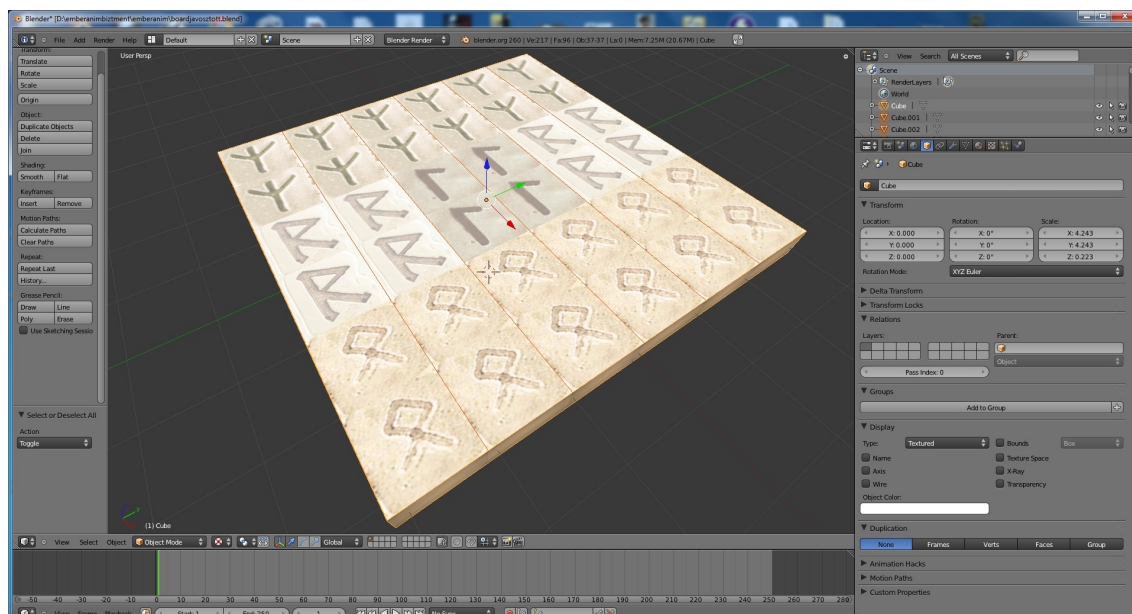
- Amennyiben az ellenfél bábujának rangja magasabb a miénknél, úgy a karakterünk a helyére kerül, az ellenfél bábuja viszont alulról fogja szagolni az ibolyát (fordított helyzetben hasonló történik)
- Ha azonos a két bábu rangszáma, akkor mindkét karakter lekerül a tábláról
- A marsall és a kém ütik egymást, amelyik előbb támadja meg a másikat, az nyer
- A bomba megtámadásakor maga a bomba mindenféleképp a játékon kívülre kerül, viszont amíg az aknaszedő a bomba megtámadásakor életben marad, addig a közönséges bábu kiesik a játékból

6.1. Tábla elkészítése

A feladatunkat végigolvasva először is *modellezzük* le a táblánkat Blender segítségével, mely egy egyik tengely irányával leskálázott kocka primitívből fog állni, lásd a [28.](#) ábrán. Ezután egyik lapját Subdivide segítségével osszuk fel. Minden egyes felosztáskor a tábla egyik oldalát figyelembe véve az eddig 1 négyzetet tartalmazó oldal két négyzetet (ami jelen esetben a mező lesz) fog tartalmazni, ezáltal megállapíthatjuk, hogy nem tudjuk szabályosan a táblánkat felosztani. Emiatt osszuk fel úgy, hogy több mező essen az oldalaira, de szabályos felosztása maradjon. Ezután, mivel szabályos maradt a felosztása a táblánknak, egyszerűen csak vágjuk le belőle a felesleget, majd a vágás után keletkezett üres részeket töltjük ki poligonokkal.

A tábla *textúrázásához* célszerű lehet a mezőknek valamilyen módon jelentést adni. Szövegesen nem érdemes a mezőket jelölni, mert lehet, hogy idegen nyelvűek is használni akarják játékunkat, és nem akarjuk, hogy hátrányos megkülönböztetésben részesüljenek. Emiatt maradjunk a vizuálisabb információközlés területén.

A legcélszerűbb dolognak az tűnik, hogy az üres mezőket szimpla talaj textúrával lássuk el, a tavat pedig valamiféle víznek látszódo dologgal borítsuk be. A gond viszont az, hogy ezt Unity-ben kellene megtennünk, ott pedig az ingyenes változatban egy elég valótlanul kinéző „víz” objektum használható erre a célra. Ettől komolyabb víznek tűnő dolog lemodellezése időigényes és gyakorlatilag felesleges feladat, annak tükrében, hogy nem ez a feladatunk.



28. ábra. A táblajáték táblája

Emiatt egy olyan megoldás lehet praktikus a jelenlegi helyzetben, amely egyszerre információközlő és figyelemfelkeltő, emellett kellően egyedi ötlet ahhoz, hogy egy játékba való integrálást megérjen. Én a feladathoz az *ősi Futhark rúnáírást* vettem alapul, melyben a játékhoz illően minden mezőnek egyedi rúnát lehet választani. A rúnák képeit körülbelül azonos méretűre nagyítsuk fel és sárgásítsuk el az „ősi”, poros, elsárgult hatás érdekében. A felhasznált rúnák és jelentésük a 29. ábrán látható.

A tábla mezőit szeparáljuk el a táblától, hogy bábuinkat könnyedén a mezők közepére lehessen a későbbiekben pozicionálni. A tábla többi részét vonjuk be egyszerű textúrákkal, hisz a játékos azokkal a részekkel kevésbé fog törődni, jó eséllyel észre se fogja venni.

A mezőkre hozzunk létre *materialt* a Unity-ben történő textúrázáshoz.

Táblánkat gyakorlatilag késznek lehet tekinteni, egyszerűen exportálásra kerülhet fbx formátumban. A tábla slálázásával nem kell törődnünk, végleges méretet ugyanis Unity-ben fog kapni.

6.2. Bábu létrehozása

Egy igazán jól kinéző, nagy felbontású modell létrehozása hosszú és időigényes feladat, emellett nem csekély gyakorlottságot és kezűgyességet is igényelhet. Arról nem is beszélve, hogy ha sok ilyen karaktert kell a gépnek egyszerre megjelenítenie, igen csak le fog esni az fps szám (másodpercenkénti képkockaszám). Mivel még nem vagyunk kellő gyakorlottsági szinten, ezért egy sculptró programot, a *Sculptrist*

	Elhaz (Algiz), védelem, az ellenséges védelmi vonalat szimbolizálja
	Laguz, a tavat szimbolizálja A járhatatlan mezőket mutatja
	Othila, az örökség, birtok szimbóluma A saját területet jelenti
	Raido, az utazás rúnája, a járható, léphető mezőket jelenti

29. ábra. A játékban használt rúnák jelentése

(ZBrush ingyenes változatát) fogjuk segítségül hívni, mely segítségével aránylag rövid idő alatt egy sematikus emberalakot létre tudunk hozni. Amennyiben nem boldogulnánk, használhatjuk a program előre elkészített embersémáját, melyet tetszés szerint tovább formálhatunk akár a Sculptrisban, akár a Blenderben.

Ezt meg is kell tennünk, ugyanis a rangszámok szemléletes ábrázolásához szükségünk lesz majd valamiféle vizuális különbségre is.

A Sculptrisből történő exportálás, majd Blenderbe beolvasás után viszont az első dolgunk a poligonszám drasztikus *lecsökkentése* lesz, ugyanis ez például 29160 poligonból fog állni, nekünk viszont jócskán elég maximum 7-8 ezer poligonból álló karakter is. A rengeteg poligon miatt használjuk a Decimate-et a poligonszám csökkentésre. A bábu arc részét viszont érdemes valamelyest felosztani, hogy például orrot tudjunk neki formázni, vagy egy egyszerű arcformát.

Ahhoz, hogy eljuttassuk bábuinkat egyik helyről a másikra, valamilyen mozgást kellene használnunk. Ezt megoldhatjuk többféle módon, ahogy korábban láthattuk, viszont most a minél élethűbb mozgásra törekedjünk. Az interneten számos, ingyenesen felhasználható bvh formátumú fájlt találhatunk erre a célra. Ekkor arra ügyeljünk, hogy az animációk *Rest Position*-je egy T alakban álló emberre emlékeztessenek, amennyiben nem így van, úgy kevés eséllyel fogjuk tudni kezelni az animációt. Fontos, hogy ne módosítsuk a csontjainak és a *pivot* pontjának (origójának) helyzetét, különben az animációban mindenféle furcsa mozgás lesz.

Méretezzük körülbelül akkorára, hogy látszólag az *armature* (váz) pont a mesh csontvázának tűnjön, ezután párosítsuk a két objektumot (CTRL-P) automatikus súlyozással.

Előfordulhat, hogy ezután a *Pose Position* elég kitekert formát ölt, ekkor a mesh módosítói közül töröljük ki a második *armature* módosítót.

A materialok helyét előre állítsuk be, Unity-ban erre már nem lesz lehetőségünk, de könnyedén megváltoztathatjuk az eredetit, ha egyszerűen a Unity-be importált assetet felülírjuk az új fbx fájlal. Ez persze az összes többi beállításra is a felülíró fájl beállításai fognak érvényesülni, nem csak a materialokra.

Hajtsuk végre az adat és a tényleges transzformációk közötti szinkront (CTRL-A).

Figyeljünk arra, hogy a normálvektorok milyen irányba mutatnak. Ehhez még Blenderben a

Properties ablakban a Shading alatt aktiváljuk a *Textured Solid* beállítást, itt látjuk a feltextúrázott területek aktuális kinézetét is.

Ha újabban lemodellezett dolgokat aggatunk a bábura, figyeljünk a súlyozásra, hogy mi mivel mozogjon egyszerre. Fontos, hogy animáláskor minden a lehető legjobban nézzen ki, ne csússzanak egymásba a poligonok, stb.

Ha rendes fizikára reagáló ruhát szeretnénk megvalósítani, akkor készüljünk fel, hogy rengeteg munkánk lesz vele. Ugyanis az *Interactive Cloth*-ot valamilyen colliderre lehet aggatni. Egyetlen collider jöhetne szóba, a mesh collider, viszont annak egyfajta pozíciója lesz, melyen legfeljebb transzformációkat hajthatunk végre, de mozgatni nem fogjuk tudni. Emiatt sokszor a ruha mozgását külön animálják, hogy a karakter különféle mozdulatsoraival szinkronban mozogjon.

Exportáláskor figyeljünk arra, hogy Pose Position-be legyen állítva az *armature*, különben az animációk *nem fognak* exportálásra kerülni.

A feladatomban az alábbi módon oldottam meg a karakterek külsejét. A 30. ábrán látható módon néznek ki.

6.3. Bábuk és tábla összeillesztése

Unity-be történő beimportálás után helyezzük el a táblát, valamint az *anonym* (felfedetlen) karaktertípust a mezőkön, hisz tudjuk, hogy a játék egyik tulajdonsága pont az egymás csapatának felfedése lesz. Ahhoz, hogy a karakterek pontosan középen álljanak meg a mezőkön, húzzuk be a bábukat egy-egy mezőbe, hogy azoknak a „gyerek” legyen. Ekkor a karakter pozíciójának koordinátái nem a *világkoordináta-rendszer* szerinti koordinátákat, hanem a *helyi koordináta-rendszer* koordinátáit fogják tar-



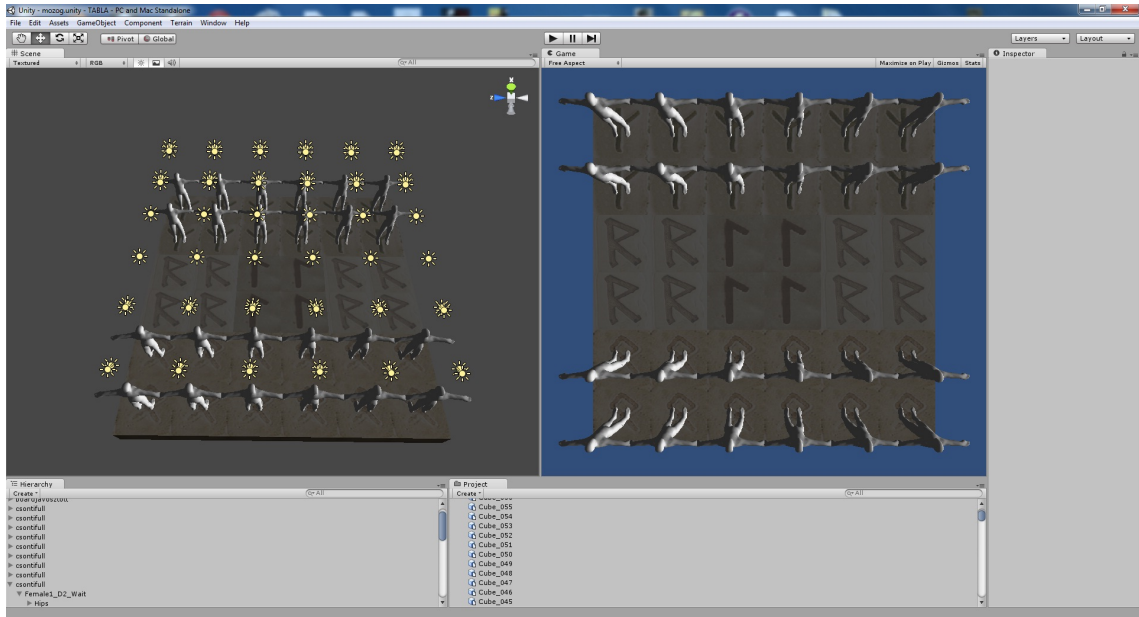
30. ábra. Játékbeli bábufajták

talmazni, tehát az origó pontosan az adott mező közepe lesz. Ezáltal egyszerűen csak 0 értékeket kell megadnunk a pozíció háromelemű vektorának, lásd a 31. ábrán.

A kamerát érdemes úgy beállítanunk, hogy a lehető legjobban belássuk a táblát, tehát az a legjobb, ha merőlegesen a pályára néz a kameránk. Ha van egy beállított *Main Camera* a Scene-ünkben, akkor azt kapcsoljuk ki a neve mellett bal oldalon található checkbox segítségével, így a Main Camera szerepét egy másik kamera veszi át. Ha nincs aktiválva az eredeti Main Camera, akkor a legutóbb létrehozott kamera képét fogjuk látni a játékban.

Ezután érdemes megvalósítanunk azokat a *scripteket*, melyek segítségével az utasítandó karakterünk kijelölhető, mozgítható, valamint támadásra buzdítható. Hozunk létre egy mappát például Gamescripts néven, és hozzunk létre benne egy C Sharp scriptet, mondjuk Select néven. Arra ügyeljünk, hogy a benne lévő class (osztály) neve megegyezzen a script nevével, különben hibát fog jelezni a compiler (fordító).

A kijelölés scriptje (a mi seregünkre vonatkozó) és a hozzá tartozó scriptek a következőképp nézhetnek ki:



31. ábra. Bábuk elhelyezkedése a táblán

```

using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class SelectedObject : MonoBehaviour {

    public static GameObject selectedobject;
    public static List<Light> environment=new List<Light>();
    static RaycastHit hit;
    public static string selectedfield(){
        bool b=Physics.Raycast(selectedobject.transform.position,
            selectedobject.transform.up,out hit,100f);

        Debug.DrawRay(selectedobject.transform.position,
            selectedobject.transform.up*100f,Color.green);

        if(!b){
            Debug.Log("nincs_talalat");
        }
        return hit.transform.parent.gameObject.name;
    }
}

```

A kijelölt karaktert érdemes valamilyen módon megjelölni. Ez történhet például egy *spotlighttal* történő erős megvilágítással, melynek színei más-más jelentéssel bírhatnak.

A script használatához először is minden karakter belsejébe (a Hierarchy-n belül) érdemes elhelyezni egy spotlightot, melyet fel tudunk később kapcsolni a megfelelő színnel.

A fenti scriptben a mezők az aktuálisan kijelölt objektumot és a lámpák környezetét fogják eltárolni. A *sugárkirajzoló* metódus csak a tesztelés megkönnyítése miatt van. A `selectedfield` függvény, amennyiben mezőt jelöltünk ki, visszaadja a kijelölt mező nevét.

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class Select : MonoBehaviour {

    void OnMouseDown() {

        if (SelectedObject.environment != null &&
            SelectedObject.environment.Count > 0) {

            for (int i = 0; i < SelectedObject.environment.Count; i++)
            {
                SelectedObject.environment[i].color = Color.white;
                SelectedObject.environment[i].enabled = false;
            }
        }
        SelectedObject.environment = new List<Light>();
    }
}
```

A script valójában a kijelölés scriptjének egy részlete. Itt, amennyiben már volt egy kijelölt objektum korábban, és újat jelölünk ki, ekkor az előzőleg felvett fények színét alapértelmezetten fehér színűre állítjuk, majd lekapcsoljuk, és ürítjük a fények listáját.

```
bool r;
Debug.Log("kezd");
Light script;
if (SelectedObject.selectedobject != null)
{ //ha már volt egy, akkor kikapcsoljuk
    script =
        SelectedObject.selectedobject.GetComponentInChildren<Light>();
    script.enabled = false;
}

script = this.gameObject.GetComponentInChildren<Light>();
```

```
script.enabled=true;
SelectedObject.selectedobject=this.gameObject;
```

```
Debug.Log(SelectedObject.selectedfield());
```

```
RaycastHit hit;
Light s;
```

A fenti scriptrészletben a régebben kijelölt objektum megvilágítását kapcsoljuk le, felkapcsoljuk az aktuálisan kijelölt objektum lámpáját, valamint a kiválasztott mezőt kiíratjuk a Unity logjába. Ez a Unity ablakának legalsó részén elhelyezkedő sáv, mely egy kattintással megnyitható, és információt közölhet a scriptek szemantikai bugjairól (elírásból származó hibáiról), típuskompatibilitási és bármilyen egyéb hibáról, melyet az információ alapján a fejlesztő ki tud javítani.

//—————*előre*—————

```
r=Physics.Raycast(transform.position,transform.forward,out hit,120f);
Debug.DrawRay(transform.position,transform.forward*120f,Color.green);
if(r){
    s=hit.transform.gameObject.GetComponentInChildren<Light>();
    if(s==null)//tó van arra
    {
        s=hit.transform.parent.gameObject.GetComponent<Light>();
    }
    SelectedObject.environment.Add(s);
    if(hit.collider is CapsuleCollider ||
    hit.collider is BoxCollider){

        if(hit.collider is CapsuleCollider){
            Side2 valt=
            hit.transform.gameObject.GetComponentInChildren<Side2>();
            if(valt!=null){
                s.color=Color.yellow;
            }
            else s.color=Color.red;
        }
        else s.color=Color.red;
    }
    else if(hit.collider is SphereCollider){
        s.color=Color.green;
    }
    s.enabled=true;
```

Ebben a scriptrészletben változókba vesszük a *Light* scriptet, melyeket egy sugár kibocsátásával teszünk meg. Amennyiben objektum van elől, legyen az ellenséges bábu, saját bábu vagy üres mező, akkor megtörténik egyféleképp az értékadás, különben tó esetén máshogy kell értéket adnunk. Az előttünk lévő mező lámpáját eltároljuk. Attól függően, hogy a sugár előttünk milyen collidert talált el, úgy színezzük a mező feletti fényt. Vörös mező esetén vagy saját bábu vagy tó van az adott mezőn, sárga esetén ellenséges bábu, zöld esetén üres mező. A szín kiválasztása után felkapcsoljuk a fényforrást.

Ugyanezt elvégezzük a többi mezőn is, így a szomszédos mezőkre vonatkozó sugár koordinátái a következő scriptrészletben tekinthetők meg.

```
//—————hátra—————
r=Physics.Raycast(transform.position,-transform.forward,out hit,120f);
...
//—————jobbra—————
r=Physics.Raycast(transform.position,transform.right,out hit,120f);
...
//—————balra—————
r=Physics.Raycast(transform.position,-transform.right,out hit,120f);
...
```

A mezők tartalmaznak egy Select2 névre keresztelt scriptet, mely nem a kijelölésért, hanem a már korábban kijelölt bábu *átmozgatásáért* felelős. A Select2 scriptet is több részletben fejtjük ki hosszára való tekintettel, az átláthatóság érdekében.

```
using UnityEngine;
using System.Collections;

public class Select2 : MonoBehaviour {

    bool b=true;
    Vector3 pos;
    Transform j;

    bool el=false;
    bool ha=false;
    bool jo=false;
    bool ba=false;

    void OnMouseDown(){
        Debug.Log("lefutok1");
        if(SelectedObject.selectedobject!=null){
```

```

        Debug.Log("lefutok2");
        if(GetComponentInChildren<Light>().color==
        Color.green){
            Debug.Log("zöld");
        }
    }
}

```

Az előbbi kódrészletben bool logikai változókbán tároljuk el, hogy a kijelölt karakterhez képest előre, hátra, jobbra vagy balra jelöltük ki a célmezőt. Az *OnMouseDown* metódus akkor hajtódik végre, ha arra az objektumra kattintottunk, amelyen ez a script rajta van. Amennyiben volt korábban kiválasztott mező és arra zöld fény vetült, vagyis léphető, úgy a Unity logjában a fejlesztőt értesítik ezek az üzenetek.

```

float xtav=
Mathf.Abs(SelectedObject.selectedobject.transform.position.x-
GetComponentInChildren<Light>().transform.position.x);

```

```

float ytav=
Mathf.Abs(SelectedObject.selectedobject.transform.position.y-
GetComponentInChildren<Light>().transform.position.y/2);
//mert magasabban van, mint a karakter

```

```

float ztav=
Mathf.Abs(SelectedObject.selectedobject.transform.position.z-
GetComponentInChildren<Light>().transform.position.z);
if(xtav>ytav && xtav>ztav)
{
    if(GetComponentInChildren<Light>().transform.position.x>
    SelectedObject.selectedobject.transform.position.x){
        Debug.Log("előre_kattintottunk");
        el=true;
    }
    else{
        Debug.Log("hátra_kattintottunk");
        ha=true;
    }
}
if(ztav>xtav && ztav>ytav)
{
    if(GetComponentInChildren<Light>().transform.position.z<
    SelectedObject.selectedobject.transform.position.z){
        Debug.Log("jobbra_kattintottunk");
    }
}

```

```

        jo=true;
    }
    else {
        Debug.Log("balra_kattintottunk");
        ba=true;
    }
}

}

```

A fentebb látható kódban meghatározódnak az x,y és z távolságok, melyet a mező feletti fényhez viszonyítunk (ugyanis az a közös minden mezőben, hogy van hozzárendelt fényforrás). Két koordinátájukat azonos módon kell vizsgálni az objektuméhoz képest, egyedül a magasságuk térhet el (az y koordináta). Ezt már csak azért is elég kettővel *elosztanunk*, mert az így létrejött koordináta is mindenképp olyan tartományban marad, hogy nem zavarhatja össze az iránymeghatározást, mégis pontosabb így vizsgálni, mint osztás nélkül (a lámpa magassága miatt). Az objektum és a kijelölt mező koordinátáinak viszonyának vizsgálatakor egyértelműen megállapítható, milyen *irányba* kattintottunk. Itt viszont csak a fejlesztő számára küldünk egy üzenetet a Unity logjában, hogy megfelelően működnek-e a feltételvizsgálatok.

```

void Update(){
    float fok;//138
    if(el || ha || jo || ba){

        Debug.Log(el+"ha"+jo+"ba");
        SelectedObject.selectedobject.transform.eulerAngles=
        new Vector3(0,-135F,0);
            //úgy forgatjuk a rest position-t,
            //hogy a pose mode az eredeti rest mode
            //iránnyal egyirányba nézzon

        if(ha){
            SelectedObject.selectedobject.transform.eulerAngles=
            new Vector3(0,45F,0);
        }
        if(jo){
            SelectedObject.selectedobject.transform.eulerAngles=
            new Vector3(0,-45F,0);
        }
        if(ba){
            SelectedObject.selectedobject.transform.eulerAngles=
            new Vector3(0,135F,0);
        }
    }
}

```

Itt vizsgáljuk meg, hogy történt-e bármilyen irányban célmező kijelölés. Ha igen, úgy forgatjuk a bábút, hogy majd a mozgás animációjának elindításakor pont arra induljon el, amerre el kell indulnia. Ugyanis előfordulhat, hogy a rest positionben a bábu *más irányba néz*, mint valamelyik animáció iránya. Tehát ezzel hozzuk gyakorlatilag szinkronba a kettőt.

Az elforgatás szöve természetesen *eltérhet* más-más bvh fájlok alkalmazásánál.

```
j=SelectedObject.selectedobject;
transform.Find("Female1_D2_Wait/Hips/Jel");

if(b){
    if(SelectedObject.selectedobject.animation.
        IsPlaying("Female1_A01_Stand")){
        SelectedObject.selectedobject.animation.Stop();
    }
    Debug.Log("en_is_lefutok");
    SelectedObject.selectedobject.animation.
        Play("Female1_B03_Walk1");

    b=false;
}
if(!b && !SelectedObject.selectedobject.animation.
    IsPlaying("Female1_B03_Walk1")){

    SelectedObject.selectedobject.animation.
        Play("Female1_A01_Stand");

    b=true;
}
```

A kódban a „j” változóba tároljuk el a *Jel* objektumot. Ez lesz az az objektum, amelyet a spotlight mellett elhelyezünk a bábukban. Létjogosultsága lehet egyrészt az, hogy ha hosszabb területeket kellene bejárnia a bábunak, a mozgás animációjának végén az átpozicionálást könnyítené meg (hogy onnan kezdjen járni, ahol félbehagyta), itt viszont az, hogy figyeljük vele később az objektum aktuális helyzetét. A „b” logikai változó itt azt figyeli, hogy a bábu épp mozog, vagy sem. Amennyiben állt a bábu, úgy megállítjuk, és lejátszuk a járás animációját. Amennyiben már lefutott, úgy átváltunk a megállásra.

```
if((el &&
j.position.x>GetComponentInChildren<Light>().transform.position.x) ||
(ha &&
```

```

j.position.x<=GetComponentInChildren<Light>().transform.position.x) ||
(jo &&
j.position.z<=GetComponentInChildren<Light>().transform.position.z) ||
(ba &&
j.position.z>=GetComponentInChildren<Light>().transform.position.z)){

    SelectedObject.selectedobject.animation.Stop();

    SelectedObject.selectedobject.transform.position=
new Vector3(GetComponentInChildren<Light>().transform.position.x,
    SelectedObject.selectedobject.transform.position.y,
    GetComponentInChildren<Light>().transform.position.z);

    SelectedObject.selectedobject.transform.Find("Female1_D2_Wait/Hips")
    .transform.localPosition=
new Vector3(-0.312901f,0.9842569f,0.169629f);

    SelectedObject.selectedobject.transform.eulerAngles=
new Vector3(0,-270F,0);
    SelectedObject.selectedobject.animation.Play("Female1_A01_Stand");

    if(el){
        el=false;
    }
    if(ha){
        ha=false;
    }
    if(jo){
        jo=false;
    }
    if(ba){
        ba=false;
    }
}

```

Ebben a scriptrészletben, amennyiben valamelyik irányba ment bábunk, és az ott elterülő mező fényforrása alatt épp áthaladt (vagyis a mező közepén), ekkor teljes mértékben „normál állapotba” állítjuk vissza, majd újra elindítjuk az állás animációját. Ebből látszik, hogy a forgatás nem az aktuális forgatásértékhez képest történő forgatás, hanem egy *adott* forgásértéknek a meghatározása, mely független a jelenlegi forgatástól.

Ahhoz, hogy a bábuk rangját eltároljuk, célszerű a rangot magát ráaggatni valahogy a bábukra. Viszont ahhoz, hogy minden egyes bábu tulajdonságait beállítsuk,

ezért az nem járható út, hogy minden azonos rangszámú bábura más-más scriptet helyezünk el, ez nem volna túl általános megoldás.

Ezért ezt érdemes egy olyan, *property-kből* (virtuális mezőkből) álló script segítségével megvalósítani, melynek mezőinek értékét majd külső kódból fogjuk tudni egyszer beállítani, viszont ezt az értéket bármikor olvashatjuk.

Ennek tudatában ez a rövid, ámde lényeges kód a következőképpen nézhet ki:

```
protected int _rang;  
  
protected bool _lathato;  
public bool lathato  
{  
    get { return _lathato; }  
}  
public int rang  
{  
    get { return _rang; }  
}  
  
public Babu(int rang)  
{  
    _rang = rang;  
}
```

A bábu konstruktorában kell megadnunk a rangot, mely megadás után beépül az objektum belső állapotába és a későbbiekben nem változik. Ennek értékét később nem módosíthatjuk, de lekérhetjük az értékét. Tartalmaz még egy „lathato” mezőt, melyből lekérhetjük, hogy a bábu felfedte-e már magát, vagy sem.

```
public void LathatovaTesz()  
{  
    _lathato = true;  
    Vector3 pos=this.gameObject.transform.position;  
    Vector3 rot=this.gameObject.transform.eulerAngles;  
    Vector3 sc=this.gameObject.transform.localScale;  
    Destroy(this.gameObject);  
    GameObject g=null;  
  
    switch(Math.Abs(rang)){  
        case 1:  
            g=Instantiate(Resources.Load("bomba"), pos,  
                Quaternion.identity) as GameObject;  
            break;
```

```

    case 2:
        ...
    }
    g.transform.eulerAngles=new Vector3(0F,-90F,0F);
}

}

```

Ebben a kódban a `LathatovaTesz` metódus annyit tesz, hogy elmenti a bábu transzformációs értékeit, majd *elpusztítja* az objektumot. Ezután a rangszámnak megfelelően létrehozuk a helyére, teljesen azonos transzformációval az új objektumot, ami tulajdonságait tekintve azonos lesz, csak a külseje fog eltérést mutatni.

Ahhoz viszont, hogy ezt megtegyük, előtte egy `Resources` mappát kell létrehozunk Unity-ben a `Project` ablakban, amibe *prefabokat* hozunk létre. A *prefabok* olyan előre elkészített objektumok, melyeket már előreláthatólag nem kell tovább módosítani, lezártnak tekinthetjük. Másszóval egy olyan *prototípust* jelent, amelyből újakat hozhatunk létre. Ha egy bábut például elkészültnek tekintettünk, akkor egy ilyen prefabra húzzuk rá. A scriptben „g”-nek értéket adva egy ilyen prefabot, a „bomba” nevűt példányosítjuk, „g” ezt a prefab példányt kapja meg.

Az ellenséges bábu felfedése után, következhet a magasabb rangszámú bábu elpusztítása, mely ugyanúgy fog történni, ahogy a felfedetlen objektum *felfedését* végeztük el (`Destroy(objektumneve)`).

Felfedéskor azért szükséges ezt a drasztikus módszert tenni, ugyanis egy animált objektum külsejének (mesh-jének) átcseréléséhez az *armature-jének* (vázának) értékeit is át kell másolni, vagyis azt, hogy melyik csont melyikkel kapcsolódik, stb. Így az elpusztítás-példányosítás ennek tudatában annyira nem is értelmetlen.

A játék persze úgy nem az igazi, ha csak a mi eseményeinket kezeli, ezért el kell látnunk valamilyen *mesterséges intelligenciával*, mely reagál tevékenységeinkre. A mesterséges intelligencia implementációja viszont elég sokrétű lehet, nem kizárólag a játék típusa és a feladat, hanem a játék nehézsége is befolyásolja. Megtörténhet az is, hogy a használt mesterséges intelligencia megvalósítása miatt az eddigi kódunkat is át kell írni, esetleg ki kell egészíteni.

Mivel az integráció igényelhet egy kisebb átervezést is, ezért nem térünk ki külön az effajta kód beépítésére.

Én ezt a feladatot Tajti Tamással csináltam együtt, Ő írta meg a mesterséges intelligenciát a játékhoz. A két projekt összeillesztése folyamatban van, melyet *MVC* (Model-View-Controller) programtervezési mintával valósítok meg, hogy a kódjának változásából ne következzen az én munkám átírása. Ezáltal a jövőben más, továbbfejlesztett kódokkal is egyszerűen lehet illeszteni a játékot.

Egyéb információk, tippek, ötletek a Unity hivatalos oldalán (<http://unity3d.com/>), azon belül is a Support linken belül találhatóak, ahol a Unity engine API-ja (alkalmazásprogramozási felülete) is helyet foglal (<http://unity3d.com/support/documentation/ScriptReference/index.html>). Ezenkívül rendelkezésre áll a számos irodalom és oldalak mellett a gazdag látogatottsággal bíró IRC (internet relay chat) egyik channelje (csatornája), ahol a világ minden tájáról, sok különböző tudással rendelkező fejlesztő, még a hivatalos fejlesztők is rendelkezésre állnak, hogy segíthessenek ([#unity3d](http://irc.freenode.net) csatornája).

Blenderrel kapcsolatos terméktámogatást a hivatalos oldal (<http://www.blender.org/>) mellett több IRC csatorna nyújt segítséget ([#blender](http://irc.freenode.net), [#blenderQA](http://irc.freenode.net), illetve <http://www.blenderartists.org/forum/>).

7. A játékfejlesztés jövője

Ahogy a technika *fejlődik*, úgy a szórakozásunk is más-más eszközökkel fog bővülni. Játékunk régi tárgyai lassacskán feledésbe merülnek, kikopnak, teret adva az újabb lehetőségeket nyújtó modern szórakozás felé.

Az évente megrendezett expók egyre növekvő népszerűsége - melyet nem kizárólag a látogatók, de a *live-streamelők*, másnéven az élő adást az interneten keresztül megtekintők számából is lehet következtetni - előremutatja ezt az exponenciálisan, robbanászerűen *növekvő* érdeklődést a játékok iránt.

A felhő alapú rendszerek előrevetítik azt az elképzelést, miszerint pár éven belül a több különböző platform *eggyé* fog válni, egy olyan eszközzé, melyet majd az akkori szemmel a maiakkal szemben ténylegesen multimédiás rendszerként tekinthetünk.

A virtuális valóság nem egy apró képernyőre fog fókuszálni, hanem *körül fog venni minket*, mindenféle értelemben. Az ezt előteremtő, realitásérzetet növelő technológiák fognak elterjedni, mellyel sajnos párhuzamban az emberek játékok iránti *addiktivitása* is korábban sosem látott méreteket fog ölteni. Ezzel egyidejűleg remélhetőleg egyre több videojáték-elvonó klinika fog nyílni a jövőben.

A játékok egyre változatosabbak lesznek az újabb felnövő indie cégek kreatív termékei és a verseny miatt, illetve a másolásvédelem tényleg kedvezni fog a legális vásárlóknak.

Milyen lesz élni a videojátékok újabb generációinak idején? Ennek a kérdésnek a megválaszolását az Olvasóra bízuk.

Irodalomjegyzék

- [1] Roland Hess, Blender Foundations: *The Essential Guide to Learning Blender 2.5*, Elsevier, Inc., 2010.
- [2] Tony Mullen: *Introducing Character Animation With Blender*, Wiley Publishing, Inc., 2007.
- [3] Will Goldstone: *Unity Game Development Essentials*, Packt Publishing Ltd., 2009.