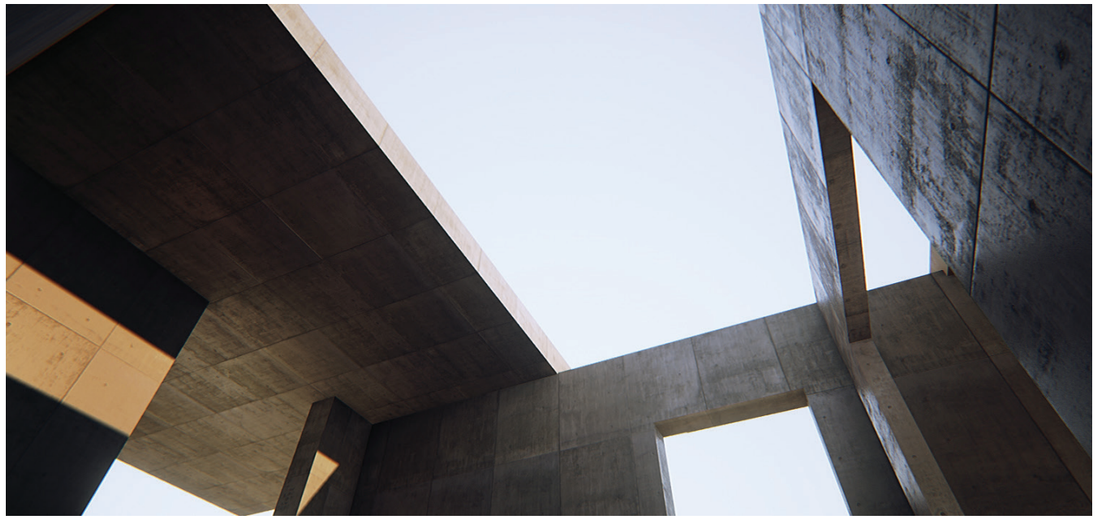


Fürtön Balázs és Háda Ádám

INTERAKTÍV ÉPÍTÉSZETI PREZENTÁCIÓ

Konzulens:
Szoboszlai mihály

MŰEGYETEM 1782

TDK dolgozat, építészeti ábrázolás tanszék
2014

TARTALOMJEGYZÉK

1. BEVEZETÉS

1.1. Gondolatok

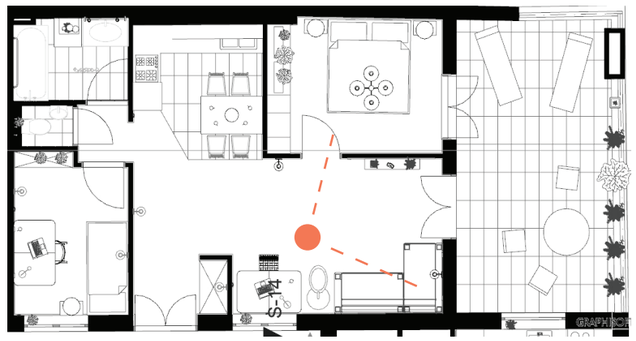
Építészként laikussal, nem építésszel építészetről beszélni nehéz. Absztrakt, elvont skiccek, valamint firkaként papírra vetett térbeli gondolatfoszlányok olvasásában, kuszának tűnő, zsúfolt műszaki rajzok értelmezésében jártas egyén, és avatatlanabb szemű partnere közötti párbeszéd ez, mely diskurzus célja sok esetben éppen az utóbbi igényeit kielégítő térstruktúra kialakítása. Ebből kifolyólag a félreértések számának minimalizására törekedni kell.

Sok esetben vettük észre, hogy akár axonometrikus nézeteken, akár perspektív vázlatokon keresztüli ötletbemutatásra érkező reakció mindössze bizonytalan egyetértés, mely egészen a térbeli modell megtekintéséig tart, mikor a téri viszonyok, arányok már mindenki számára érthetőek, és a félreértett, esetleg nem kommunikált részletek is láthatóvá válnak.

Sokat gondolkodtunk azon, hogy hogyan, milyen eszközökkel lehet áthidalni ezt a kommunikációs szakadékot, és arra jutottunk, hogy a precízen kidolgozott 3D modellen, majd az erről készített látványterveken keresztül sikerülhet a legbiztosabb módon.

A közelmúltban a számítógépes grafika elképesztő fejlődésen ment keresztül, eljutottunk arra a szintre, hogy a képernyőinken egyes esetekben lehetetlennek tűnő vállalkozás megkülönböztetni mi igazi, s mi csupán a gondolatok digitális leképeződése. Építészként úgy érezzük, hogy a technika ilyen irányú fejlődése hatalmas segítséget jelent a tervezés során, eddig nem látott módjai nyíltak az építészeti prezentációnak.

Egy-egy fotorealisztikus látványterv elkészítése köztudottan hosszadalmas, munkaigényes feladat, mely így jelentős költségekkel is jár. Ám legyen szó tervpályázatról, vagy megrendelői igényről, mára megkerülhetetlenné vállt a készítésük. Szépek, eladják a terveket, de nehézkesen módosíthatóak, ebből kifolyólag több változat prezentálására nem igazán alkalmasak, illetve roppant egyszerűen manipulálhatóak (a 3D szoftvereken belül pillanatok alatt elvégezhető kamerakorrekciók perspektív torzításai a tér arányát jelentős mértékben képesek változtatni), ez



1-3. ábra. A fenti illusztrációk ugyanahhoz a munkához készültek. Mindegyik rajz célközönsége más: a skicc saját magam felé, az alaprajz az építtetőnek, a látványterv pedig a potenciális vásárlók felé kommunikál. A vásárló döntését alapvetően a legutolsó perspektív kép befolyásolja, ez alapján képes megítélni a a tér milyenségét.

fals kép kialakulásához vezethet a megrendelőben. Ezeken felül a legszembeötlőbb hátrányuk: csak a kamera által látott részt képesek bemutatni, nem lehet végigsétálni, nézelődni a leképezett térrészben, másik nézet újabb rendert, ezáltal időt, munkaerőt, végső soron pénzt igényel.

Az egyes 3D szoftverek viewportjaiban már van lehetőség térbeli mozgásra, de a jelzésértékű árnyalás nem túl igényes, nem feltétlenül használható klienskommunikációra. Az egyes BIM szoftverek (Building Information Modelling, épületinformációs modell) mellé készülnek virtuális épületbejárást lehetővé tévő alkalmazások (például a BIMx az ArchiCAD mellé), melyekben már a szabad mozgás lehetséges.

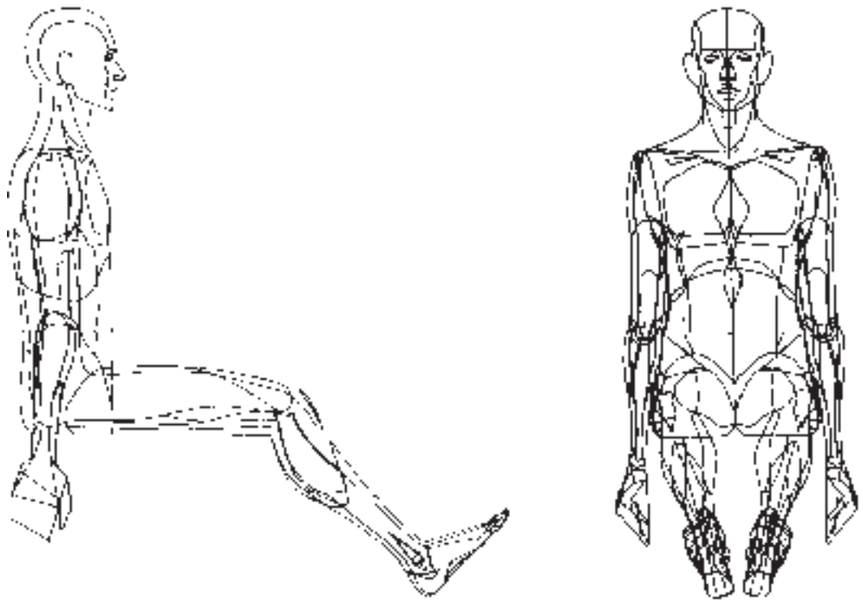
Jelen dolgozatban azonban ezekkel nem foglalkozunk, hanem a megjelenítésben nagyobb szabadságot biztosító megoldásoknak, azok használatának járunk utána, tehát a játékmotor építészeti vizualizáció, prezentáció céljára történő hasznosításának.

1.2. Vizuális prezentációs formák

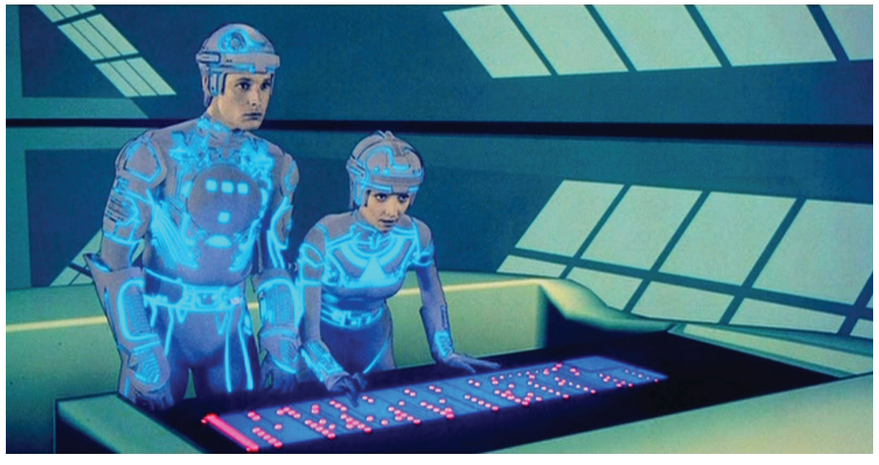
Minden épület mielőtt megépülne, valakinek a fejében megszületik, gondolataiban látja, hogy miként néz majd ki. Ezt leghatékonyabban rajz formájában tudja megosztani az emberekkel, ezért is tanítják a mai napig a kézi rajzolást. Viszont ha már egy terv egészen előrehaladott állapotban van, netán pontos alaprajzokkal, méretekkel rendelkezik, akkor már sokkal többféleképpen mutathatjuk meg a tervünket.

Régen, még a személyi számítógépek elterjedése előtt, a kézzel szerkesztett perspektíván kívül nem volt más lehetőség a rajzi bemutatásra. Azonban ezek elterjedése és fejlődése után, többek között a videojáték- és a filmiparnak köszönhetően, a számítógépek egészen szép grafikát voltak képesek előállítani.

A legelső számítógépes grafika 1961-ben készült William Fetter által, aki a “Boeing man”-t, egy emberalakot rajzolt ki. A technika együtt kellett fejlődjön a megjelnítő eszközökkel is, hiszen ekkor még a színes megjelenítés is gyerekcipőben volt. 1971-ben már 1024x1024 felbontású monitoron tudtak fényceruzával rajzolni az IBM 2250-es gépen. Piacra először csak műszaki rajz készítő programokat készítettek, az AutoCAD első verziója 1982-ban jelent meg. A 3D renderelő szoftverek népszerű kutatási témák voltak a ‘60-as, ‘70-es években, hasznosítása először a filmiparban jelent meg. A szintén 1982-es Tron című film már nagyban erre a technológiára épít, majd egyre többet használják a reklámokban, animációs filmek készülnek, a technológia pedig az egyre szebb végeredmény felé halad. A ‘90-es években jelenik meg az első modellező program kereskedelmi



4. ábra.William Fetter emberalakja, a Boeing man. 1961



5. ábra. Tron című film, 1982.

forgalomban, az Autodesk 3D Studio (1990). Innentől kerül az átlagember (és az építészek) kezébe ez az eszköz, ezáltal kerül kölcsönhatásba az építészettel. [historyofCGI]

Ha már a számítógép által renderelt látványról beszélünk, ezt is egészen sokféle formába lehet önteni. A legegyszerűbb, a renderelt perspektív kép. Ebből, ha sokat egymás után fűzünk, úgy hogy a kamerát mozgatjuk, kialakul a végigsétálós animáció. Ezeket nem nevezhetjük interaktívnak, hiszen semmit sem tudunk befolyásolni. Az interakció peremén állnak a renderelt 360°-os panorámaképek, ezekben már mi magunk tudunk körbenézni. A képet lehetséges ugyanabból a perspektívából külön rétegekre készíteni több variációval, és így biztosítani interaktivitási lehetőséget a képen belül. Ezt egy panorámával egybevéve, már egészen úgy érezhetjük, hogy beleszólhatunk a jelenetbe.

Sokkal nagyobb szabadságot jelent viszont, ha az elkészült látvány valós időben készül, a kamerát tetszőlegesen, minden szabadságfokban mozgathatjuk, így bejárhatjuk, felfedezhetjük az épület minden terét, részletét. Természetesen ennek az elkészítése jóval nehezebb, mint az előzőeknek. Nem csak a technika szempontjából, hanem azért is, mert nem játszhatjuk meg azt, hogy csak a bemutatott részeket dolgozzuk ki, hiszen bármit megtekinthet a szemlélődő. (Ez a metódus egyébként a játékok készítésekor igen elterjedt, miszerint ami nem látszik, az nem is létezik, majd a felhasználót korlátozzák a megmodellezett területekre.)

Technológia szempontjából a videó- és képkészítés - legyen szó akár egyszerűről akár panorámáról - egy előre elkészített, nem valós idejű renderer-rel készült kép. Az interaktív bejárás viszont valós időben készül, ezt más technikával lehet elérni, más tudás is szükséges az elkészítéséhez az álló- és mozgóképekhez : modellező-világító-anyagozó készségeken felül.

2. RENDER ENGINEK

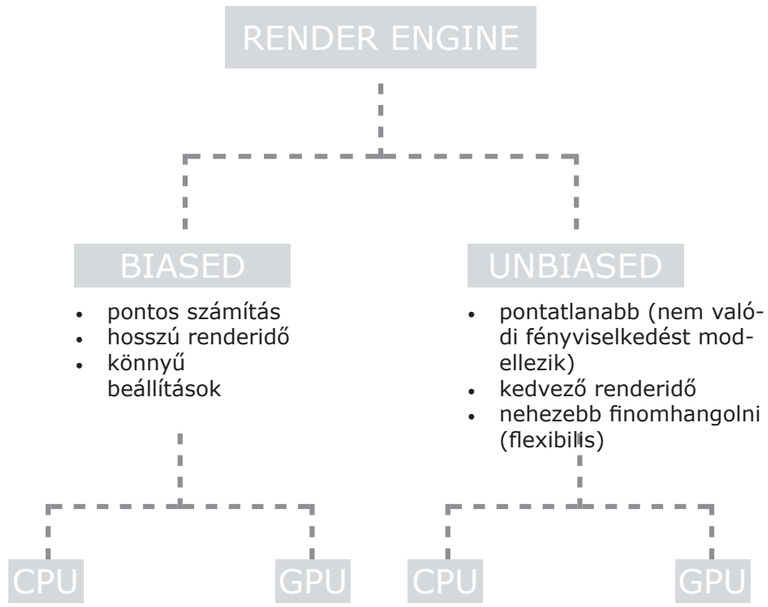
2.1. Nem valós idejű render engine-ek

A render engine-ek célja, hogy egy előre elkészített jelenetből, általában térbeli modellekből képet generáljanak. Ez a megfogalmazás elég általános, emiatt több különböző kategóriába lehet őket sorolni az alapján, hogy ezt a folyamatot milyen további célokkal egészítjük ki.

Nyilván a különböző célok eléréséhez különböző mennyiségű számítás kell, a számítások pedig véges erőforrással időbe kerülnek. Ez alapján lehet kettéválasztani őket valós idejű és nem valós idejű kategóriákba, az alapján, hogy mennyi idő alatt képesek egy képkockát renderelni: a valós idejű másodpercenként készít több képet (filmeknél, animációknál szokásos érték a 24 FPS (frame per second: képkocka másodpercenként), ami játékok esetén akár 60 FPS is lehet), ami ennél lassabb, az már a másik kategóriába tartozik, és nem a sebesség a cél, ezért nem sajnálnak akár órákat sem egy képre.

Az ilyen nagy számításigényű folyamatoknál a fő cél az, hogy az eredmény szép legyen. Talán a legtöbb számítást igénylő render engine-ek a fotorealisztikus célt kitűzőek, ezekből találjuk a legtöbbet. A valóságban a képek úgy készülnek, hogy egy érzékelő – legyen az kamera vagy szem – érzékeli a bejövő fénysugarakat. Ennek az analógiájára épülnek a programok, minden pixelben a beérkező fénysugarak színét és erősségét keressük.

A fotorealisztikusság egy szubjektív dolog, viszont több elmélet is született rá, de mind a fizika energia-megmaradás törvényén alapszik, ebből dolgozták ki a renderelési egyenletet. A renderelési egyenlet – ami egy integrálegyenlet – a felület egy pontjára értendő, ahol a kimenő sugárzás mértéke megegyezik a kibocsátott és visszavert fény összegével. Ezt egyszerre definiálta David Immel és James Kajiya 1986-ban. A realisztikus render engine-ek a mai napig ezt az problémát próbálják megoldani.



6. ábra. Render engine-ek csoportosítása.

Az egyenlet a következő:

$$L_o(\mathbf{x}, \omega_o, \lambda, t) = L_e(\mathbf{x}, \omega_o, \lambda, t) + \int_\Omega f_r(\mathbf{x}, \omega_i, \omega_o, \lambda, t)\, L_i(\mathbf{x}, \omega_i, \lambda, t)\, (\omega_i \cdot \mathbf{n})\, \mathrm{d}\,\omega_i$$

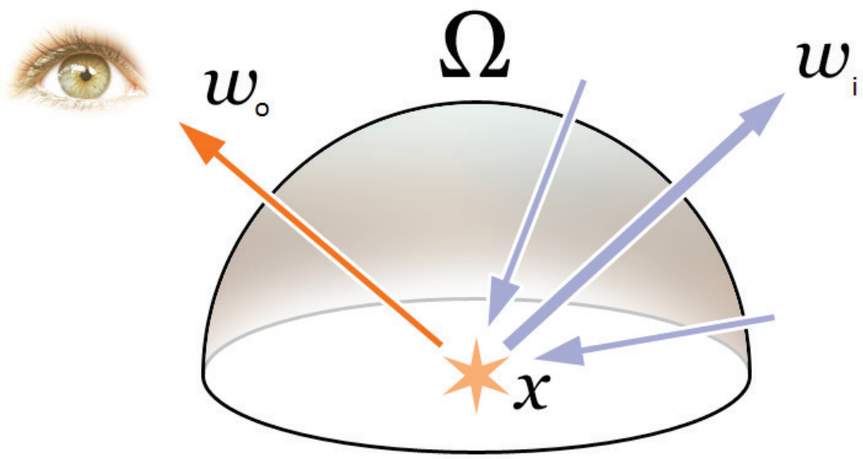
ahol:

- λ - a fény hullámhossza
- t - az idő
- x – hely a térben
- ω_o – a kimenő fény iránya
- ω_i – a kimenő fény irányának fordítottja
- $L_o(x, \omega_o, \lambda, t)$ – a teljes λ hullámhosszú sugárzás ω_o irányban x-ből t időben
- $L_e(x, \omega_o, \lambda, t)$ – a kibocsátott sugárzás
- Ω - egységnyi félgömb ami minden lehetséges ω_i értéket tartalmaz
- integrál(omega,. . . d ω_i) – az integrál a félgömbön
- $f_r(x_i, \omega_i, \omega_o, \lambda, t)$ – a kétirányú visszatükröződés eloszlási függvény (BRDF), az aránya a fénynek ami visszaverődik ω_i irányból ω_o irányba x pozícióban, t időben
- $L_i(x, \omega_i, \lambda, t)$ – a sugárzás λ hullámhosszal x-be ω_i iránnyal t időben
- $\omega_i * n$ – a gyengítő faktor a bejövő sugárzás erősségének, a beérkezési szög miatt, ahogy egy fluxus nagyobb területre érkezik, mint a vizsgált terület

Az egyenlet könnyen átrendezhető az egyszerű műveletek miatt. Érdemes megjegyezni az egyenlet spektrum- és időfüggőségét, L_o akár a látható fényen is túlléphet, illetve motion blur (ld. 2.2.1.) is előidézhető egy átlagolt, különböző t-kel számolt L_o -al. [renderEQ]

Viszont ez az egyenlet sem tökéletes, számos fényviselkedést nem vesz figyelembe, amit a renderelő programok már képesek szimulálni: például átlátszóság, fénytörés, subsurface scattering (felület alatti fényszóródás), és még szó sem esett a volumetrikus, nem felületi pontokról például köd, folyadékok.

A fotorealisztikusságra törekedő render programok tehát ezt próbálják megközelíteni, ez alapján lehet őket kettéválasztani biased és unbiased kategóriákba, amit magyarul ferdítettnek és ferdítetlennek lehetne fordítani. Az unbiased elég jól oldja meg a renderelési egyenletet, sima, nem zajos és valószínűsíthetően helyesnek tűnik.



7. ábra. Renderelési egyenlet. Leírja az x pontból adott irányban távozó fény nagyságát a beérkező és kibocsátott fények függvényében.

Unbiased renderelési eljárások a következők:

- Path tracing – gathering („gyűjtő”) sugárkövetés, Monte Carlo mintavételezéssel
- Light tracing – shooting („lövő”) sugárkövetés, Monte Carlo mintavételezéssel
- Bidirectional path tracing – kombinált gathering és shooting sugárkövetés, Monte Carlo mintavételezéssel
- Metropolis light transport – kevert gathering és shooting sugárkövetés, Metropolis–Hastings mintavételezéssel

Ha kevés mintát veszünk, a kép zajos maradhat, így nem számít unbiased-nek. [unbiased] [pathtracing]

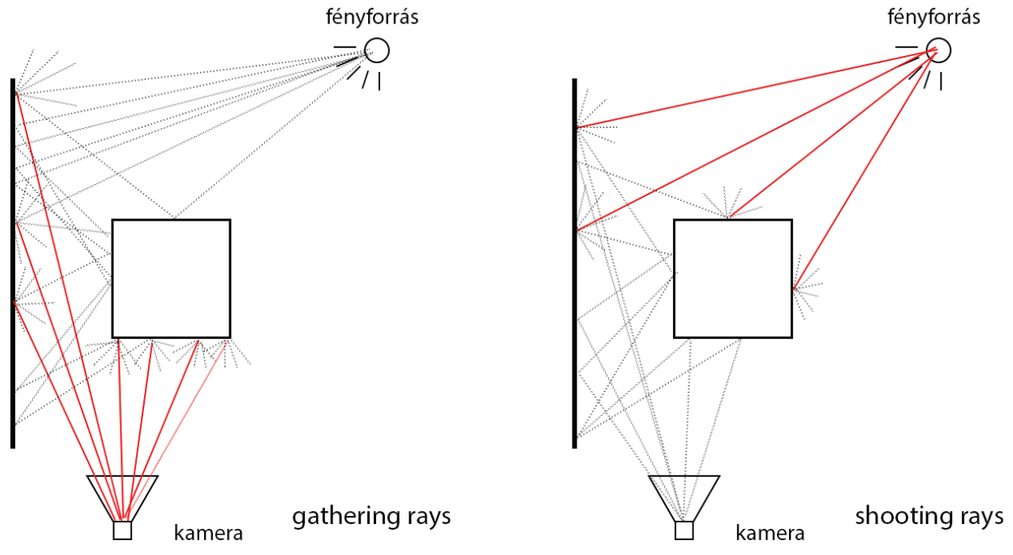
A Monte-Carlo módszerről: a fent említett módszerek nem raszterben, hanem véletlenszerűen elhelyezett mintákkal dolgoznak. A minták felvételénél alapvető szempont, hogy a minták számának növelésével gyorsan a valós eredményhez konvergáljunk. Szabályos raszterben történő felvétel esetén a dimenzió növelésével exponenciálisan nő a számításigény, a magyarázat abban keresendő, hogy a mintapontok közötti távolság nagy, így nem töltik ki megfelelő sűrűségben a teret. Véletlenszerűen felvett mintapontok esetében belátható, hogy ez is korrekt módon becsli a valós értéket. A becslés pontosítható a szórás alapján: ahol a szórás nagy, oda több mintapontot helyezünk, ezen eljárás neve fontosság szerinti mintavételezés (importance sampling) [SZIRMAY2009,208.o.].

Amikor fotorealisztikus fénysugarakról beszélünk, a jelenet megvilágításának módszere a global illumination. Ilyen módszerek pl.:

- radiosity: iterációs, végeselem módszer, olyan jelenetekhez ahol diffúz felületek vannak (ld. lightmapping)
- ray-tracing: sugárkövetés
- beam tracking: hullámterjedés elvén működik
- ambient occlusion: szórt fényre való rálátás alapján számol
- photon mapping: fotonok szimulációja, lassú, de akár fénytöréseket is kezel
- kép alapú világítás (image based lighting - IBL): egy panorámakép alapján számol fényeket

Ezek a módszerek egymással kombinálhatóak, sok beállítási lehetőséggel. [globillum]

Nézzük meg az efféle render programok működési hátterét. A számítás mindig egy általános célú



8. ábra. Gyüjtő és lövő sugarak. A képen jól látható milyen irányból kezdeményezzük a fénysugarak útjainak lekövetését.

processzoron történik, ez legtöbb esetben a CPU, de manapság egyre tör előre a videokártyák efféle hasznosítása (GPGPU - general purpose computing on graphics processing units) pl. openCL, CUDA. A hardvergyártók egymással technológiai versenyt futnak, ebből következően sajnos ezek nem egységesek, így a program készítőjének el kell dönteni, hogy milyen technológiát részesít előnyben az által, hogy implementálja a rendererjében.

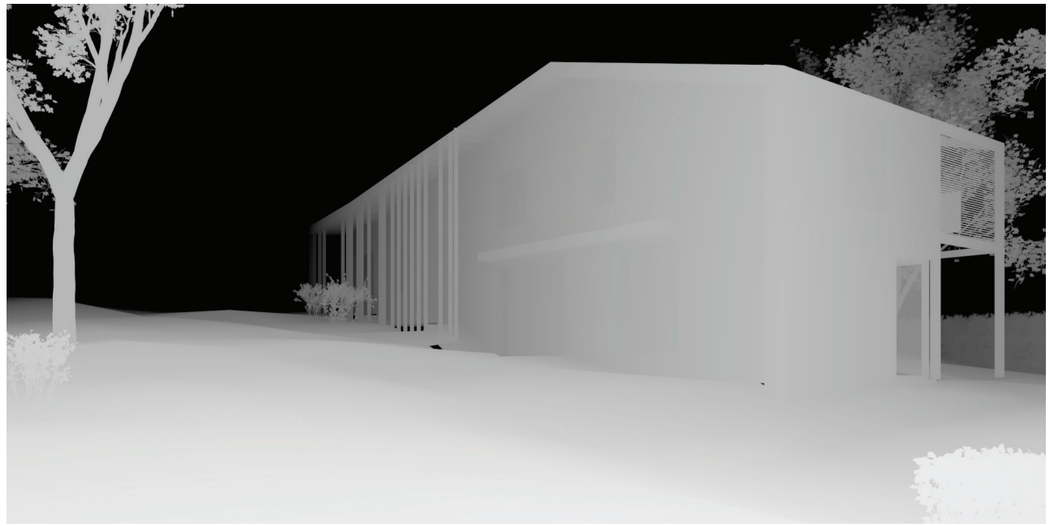
A program futása során a renderelendő felület pixeleinek különböző értékeit számolja, a fenti technikákban direktben csak a pixelek vannak kiszámolva, a jelentre vonatkozóan nincsenek számítások. A pixelek számításakor a program eléri a jelenet összes többi részét, tehát egy adott pixel számításakor az összes adat rendelkezésre áll. A renderelés során a pixelekhez értékek társulnak, általában a render motornál beállítható, hogy mely értékeket számolja ki. Az általános természetesen a pixel színe, de ezen kívül sok más részeredmény is letárolható, pl. kamerától való távolság, adott fény mértéke, felület normálvektora, stb. Ezek igen hasznosak a következő lépéshez.

A következő lépés, ami opcionális, az utómunka (post-processing). Ez már nem a fotorealisztikus rendereléshez tartozik, viszont nagyon gyakori lépés, hiszen a fényképezővel készült képeket is utómunkázzuk. Ez a lépés a hagyományostól annyiban tér el, hogy rendelkezésünkre áll(hat) extra adat, aminek a birtokában szép effektekkel módosíthatjuk a képet, hogy elérjük a kívánt hatást.

2.2. Valós idejű render engine-ek

Ezek a render engine-ek egészen komplex módon működhetnek. Ez azért van így, mert a megfelelő render sebesség (másodpercenként legalább 25, de megszokott a 60 FPS) eléréséhez, kevés idő áll rendelkezésre, azaz a számításokat limitálni kell. A valós idejű renderek célja az, hogy a virtuális világgal interakcióba tudjunk lépni. Ez legelőször és leggyakrabban videojátékoknál fordul elő. Az elkészült képeket gyakran hívjuk képkockának, angolul frame-nek.

Ebből szinte egyértelműen következik, hogy a fénysugarak szimulációjára nincs idő, bár az eredmény sokszor nagyon szép, de sok csalás van benne, így a biased kategóriába tartozik, a renderelési egyenlet csak egyszerű felületeknél van megközelítve. A sugárkövetési algoritmusok helyett a Z-bufferelt háromszög raszterizációt használjuk. Bár nagyon sok technika fejlődött ki ezidáig, de valamennyi erre vezethető vissza.



9. ábra. Z-buffer. A mélységi térkép alapján nem takarja el a közelebb levő részeket az utólag renderelt elem sem.

A valós idejű renderelés nem lenne lehetséges, ha nem készült volna hozzá célhardver, ez a videokártya. Ez az alkatrész párhuzamos munkavégzésre lett kifejlesztve, működése miatt beszélünk

egy úgy nevezett graphics pipelineról (grafikai csővezeték), ami egy szállítószalaghoz hasonlítható: a nagyszámú bemenő adatokat nagy sebességgel dolgozza fel, de nagyon hasonló műveleteket végez el. A „szállítószalag” egyes gépei különböző mértékben beállíthatóak, néhol csak műveletek elvégzését lehet ki-be kapcsolni pl. backface culling (hátlap eldobás, ld. 3.3.1), de néhány „gépnek” programot lehet megadni. [pipeline]

A renderelés teljesen máshogy történik, itt nem a pixeleket számoljuk egyesével, hanem a felületeket rajzoljuk ki egyesével, amihez több pixel is tartozhat. Később ezek egymást felül is írhatják, ha épp közelebb áll a kamerához egy felület és eltakarja a távolabbit. A technológia úgy lett kitalálva – a nagy sebesség elérése miatt – hogy az épp kirajzolódó felület nem éri el a többi felület adatait, csak azokat, amiket a felülethez hozzárendeltek, vagy globálisan elérhetőek (ezeket pedig igen alacsony számban kell tartani, mert lassúak).

Ez a nagy korlátozás viszont nem jelenti azt, hogy el kell búcsúznunk a szép fényhatásoktól, ezekre igen sok technika fejlődött ki. Számos adat nem igényli, hogy újraszámoljuk képkockánként, így ezeket előre is lehet számolni – bármi féle idő korlát nélkül – és tárolni egy fájlban. Ezek sokfélék lehetnek, de túlnyomó többségben textúrák, amiket a modellek felületéheihez rendelünk. Ily módszerrel a statikus fények, statikus környezeti elemek, kis számítással figyelembe vehetőek.

Fontos megjegyezni, hogy ez az először játékoknál használt technikákból ered, ahol a cél nem a tökéletes kép, hanem a szép, torz, de első ránézésre hihető megjelenés. Például előfordulhat, hogy egyes tükröződések nem egészen azt mutatják, mint ami valójában abból a szemszögből látszana, de mivel az eltérés nem nagy, elsőre nem feltűnő.

Egy egyszerű renderelés a grafikus kártyán röviden így történik, de ennél sokkal bonyolultabb is lehet:

I. Jelenet betöltése RAM-ba
II. Erőforrások betöltése a videokártyába: textúrák, ponthálók, shader-programok
III. Általános jeleneti konstansok beállítása pl.: napfény iránya, mértéke, kamera projekciós mátrix
IV. Képkocka renderelése videokártyán – ez történik meg másodpercenként sokszor
    a) Képkockára jellemző általános konstansok beállítása pl.: kamera pozíció, irány
    b) Ponthálók kirenderelése egyesével, bizonyos anyag technikájával (egy modell több ponthálóból is állhat, a technika többféle shader programok kombinációja)

a. konstansok beállítása: erőforrások (általában textúrák), modellre jellemzőek (pl. modell transzformációs mátrix), anyagtulajdonságok (pl. szín, csillogás mértéke)

b. a videokártya kiszámítja a pontháló háromszögeinek helyzetét a kamera szemszögéből

c. a videokártya raszterizálja a kiszámolt háromszögeket a rendertextúrába

c) A kész kép utómunkája adott technikával és beállításokkal

V. Erőforrások felszabadítása

Látható, hogy ez a célhardver speciális erőforrásokat igényel, amik már részben előre számoltak, olyanokat, amikhez további munka szükséges, mint ami elég lenne egy nem valós idejű renderkép elkészítéséhez. [3dgraf]

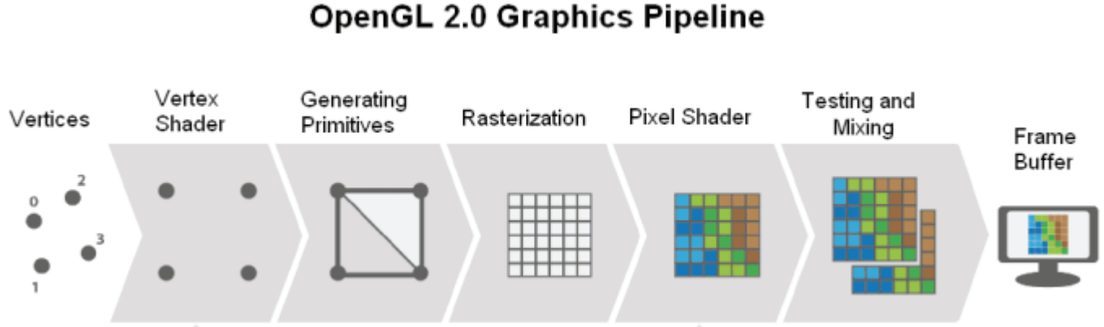
2.2.1. Grafikai elemek valós idejű megoldásai

Ebben a részben olyan technikákról lesz szó, amik a fény fizikai hatásaiból alapvetően következnek, így a nem valós idejű rendererek eredményeiben benne vannak, viszont nagy számításigényük miatt csak megfelelő trükkel fordulhatnak elő valós időben.

Viszont ezelőtt tisztázni érdemes a shaderek fogalmát. A shaderek számítógépes programok - ez esetben a videokártya számára – ami a jelenet megjelenítését szabja meg. Közös tulajdonságuk, hogy egy kép számítása során is sok ezerszer netán milliószor futnak le, mindezt másodpercenként többször, így fontos odafigyelni az optimalizációjukra. Több típusa van, sorrendjük viszont fix, ebben a sorrendben a következők:

2.2.1.1. Vertex shader

A vertex (magyarul csúcspont) shader a geometria csúcspontjaival foglalkozik, minden csúcspontra vonatkoztatva lefut. Ezekhez számolhat értékeket, pl. fényeket (ld. gouraud shading), UV koordinátákat, és még a csúcspont helyzetét is módosíthatja. Ez szinte mindig megtörténik, ugyanis a forgatások, mozgatások, átméretezések itt valósulnak meg sok esetben. Ez egy kötelező shader.



10. ábra. Példa egyszerű grafikus pipeline-ra.

2.2.1.2. Tesszalláció

Nem kötelező shader, itt nagy mennyiségű csúcspontokkal egészíthetjük ki a geometriát. A megkötés az, hogy előre definiált számú ponttal egészítünk ki minden pontot. Főleg felületek felosztására használják. Ez a kód is csúcspontonként fut.

2.2.1.3. Geometry shader

Ebben a fázisban a meglevő modellünket további csúcsokkal egészíthetjük ki. Ez a program minden csúcspontra lefut, még azokra is, amik a tesszalláció során születtek. A különbség ez és a tesszalláció között, hogy itt változó számú pontra cserélhetjük az előző pontot, azt is meg tehetjük, hogy nulla pontot adunk vissza. Ez is egy opcionális shader.

2.2.1.4. Pixel/Fragment shader

Talán a legfontosabb programrész, az utolsó a pipeline-ban és kötelező. Ez a háromszögek raszterizációjának egy fontos része, a program a háromszög pixeleire fut le, ez mondja meg, hogy milyen színt (értéket) fest a rendertextúrára. A textúrák és a fényhatások festése főképp itt történik meg.

Nézzünk néhány gyakori megvalósítást.

2.2.1.5. Előreszámolt fix fények - Lightmapping

Ezek felületekhez rendelt textúrák, amik a felület világosságát határozzák meg. Ezeket előre számolják valamilyen nem valós idejű módszerrel, népszerű megoldás erre a radiosity (2.1). A felület színe pedig a diffúz textúra és a lightmap textúra szorzata.

A lightmap erősségét utólag is lehet állítani, így ha például egy épületben két lightmapet készítünk a napfénynek és a mesterséges világításnak, akkor valós időben kapcsolhatjuk a villanyt fel-le. Úgy érjük ezt el, hogy egyszer csak a napfény hatásával dolgozunk, máskor a két lightmap összegével. Ez a techinka támogatja a színes fényeket is, illetve a HDR-t (erről később). Ez egy pixelenként

11. ábra. Tesszalláció. A felületek felosztásával érhetjük el a displacement technikát is. A directXbe csak a 11. verzióba került be.

számolt technika.

2.2.1.6. Csillogások, élfények - Specularity

A phong shading módszere ez, az egyenlet: szín = lightmap · diffuse + csillanás erőssége · csillanás színe · fény ereje. Ami az érdekes az egészben, az a csillanás erőssége, az ábra mutatja a működését. A vektorműveletek után kapott eredményt még hatványozzuk a csillanás keménységének függvényében, ezt szorozzuk a fény színével, és a fény erejével.

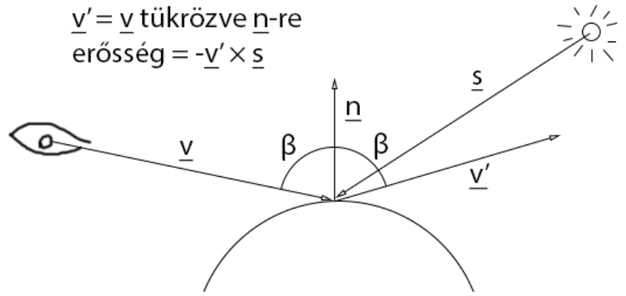
A felületre jutó fény ereje függ annak típusától, ha párhuzamos fény, akkor ez egy konstans, ha pontszerű fényforrás, akkor a távolság négyzetének fordítottjával szorzandó a fényforrás ereje. Ebben a technikában a normálvektorok interpolációja vertex alapon számolható, a többi pixel alapon.

2.2.1.7. Tükröződések - Environment mapping, igazi tükröződés

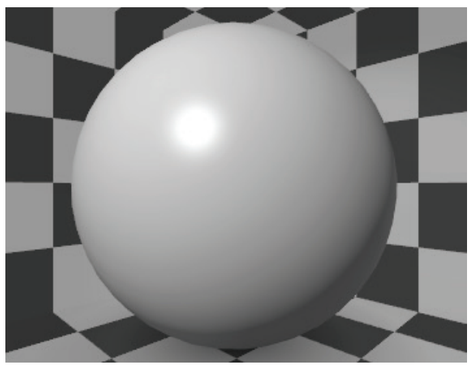
Az environmental mapping erőforrása egy panorámakép egy adott pontból. Gyakorlatban ez két féleképp fordul elő: gömb és kocka formájában. Ez a panorámakép tükröződés formájában jelenik meg a felületen. Viszont ezt fenntartással szabad használni, egy ilyen kép a valóságban egy pontra vonatkozik, viszont mi sokkal több helyen használjuk. Egy égboltnál nem probléma, mivel minimális különbség van két egymástól akár egy kilométerre levő pontban. Ezt a technikát viszont belső terekben is alkalmazzák, ahol már egy fél méteres táv is jelentős különbségeket okozhat. Ilyenkor több tucat panoráma is előfordulhat egy jelenetben, mindig a legközelebbit, vagy a legközelebbiek interpolációját használja. Nagy görbületű vagy nem sima felületeknél ez kevésbé feltűnő.

Néha vannak olyan felületek, amikre érdemes igazi tükröződést számolni valós időben, ilyen például egy vízfelület. Ez esetben nincs különösebb trükk, először készül egy renderkép úgy, hogy a kamerát tükrözzük a felületre. Ezt a képet használja fel majd a vízfelület pixel shaderje. Mivel ez esetben két kép is készül egy kép helyett, ez erősen megnöveli az egy képkockára jutó számításigényt. A hardverhasználat a csillogáséhoz hasonló.

12. ábra. Lightmap. A felületeken ezt az előre számolt fényértékeket mutatja a kép.



13 ábra. Csillogás. Egyszerű vektorműveletekkel számítható a csillanás mértéke.



14. ábra. Megcsillanó gömb. Az anyag szerkesztők gyakran egy gömbön mutatják az eredményt.

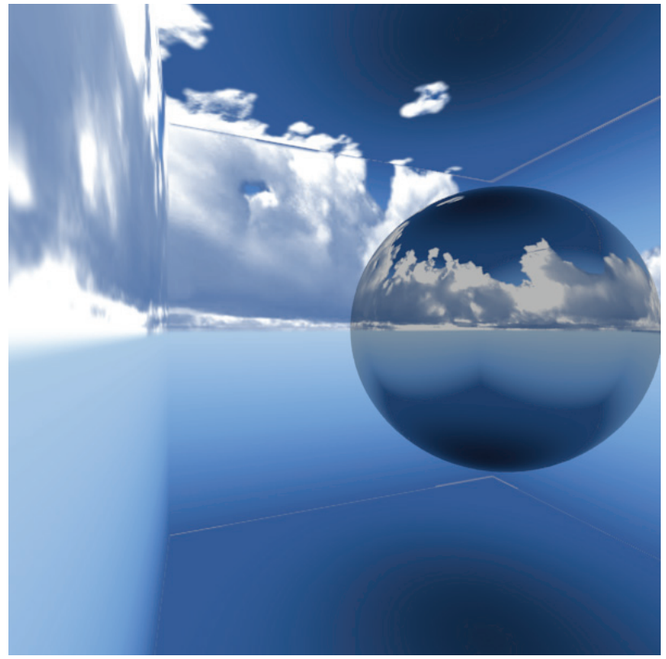
2.2.1.8. Domborodások, felületi részletek - Bump mapping, Displacement mapping

Igen sok tárgy, anyag felülete nem sima. Viszont, ha minden apró geometriai egyenletességet háromszögekkel képeznénk le, hamar milliárdos számok felé mennénk el. Erre nyújt egy nagyszerű megoldást a bump mapping. Azokat a technikákat soroljuk ide, amik a geometriát nem módosítják, csak a felület színét változtatják. Talán a legnépszerűbb bump mapping-technika a normal mapping. A normal map színeinek RGB komponensei az egységvektor XYZ koordinátakomponenseinek felelnek meg. Az RGB skála [0;1] tartománya az XYZ koordináták [-1;1] tartományának felelnek meg, tehát a semleges szürke (RGB(0.5,0.5,0.5)) felel meg a zérusvektornak. Fontos megjegyeznünk továbbá, hogy a normal map által meghatározott irányok tangens térben (a tangenstér a világ- és objectumtértől eltérően nem [kvázi] konstans koordinátarendszer [az objektumtér csak a kérdéses tárgy szintjén állandó], hanem poligon-szinten változik) értelmezendő. Ha egy tetszőleges háttérpixelt - olyat, amely nem tartozik egy poligonhoz sem - vizsgálunk, akkor a kapott szín kékeslila, jó esetben RGB(128,128,255), ami [0;1] intervallumon értelmezve XYZ(0,0,1), tehát Z irányú egységvektor, a poligon normálvektora. A normal map tehát egy textúra, ami színadatok helyett az adott pontban levő normálvektortól való eltérést tartalmazza. A fényhatásokban felhasznált normálvektor helyett ezt a módosított vektort felhasználva nagy részletesség jelenik meg. Feltűnőek ezek a technikák akkor, ha élben látjuk őket, hiszen csak a felület színét változtatják meg: az éle szögletes marad.

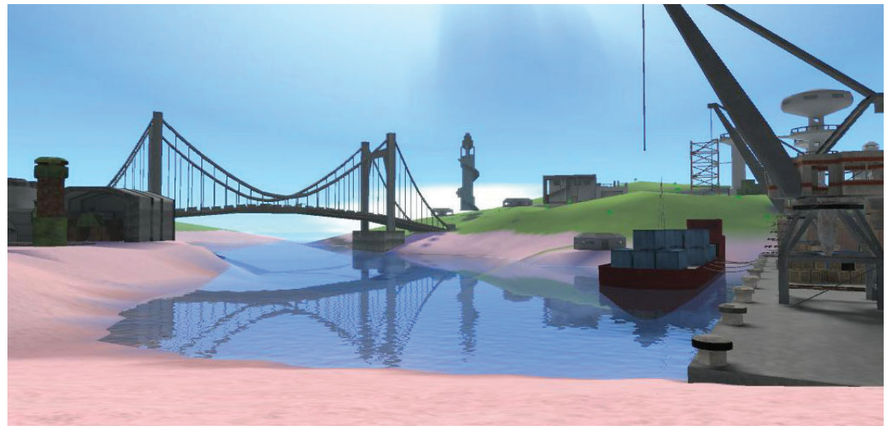
Gyakori technika még a parallax mapping, ahol párhuzamos egymástól kis mértékben eltolt maszkolt lapokkal növelik a domborodás látványát. Ez a technika is főképp pixel alapú, a normálvektor interpolációtól eltekintve.

A displacement mapping, szemben a bump mappingal, igenis módosítja a geometriát. A felhasznált textúra szürkeárnyalatos, a pixelértéktől függően tolja el pontokat. Sokszor tesszallációval együtt használják, a nagyobb sebesség és valós időben állítható felbontás miatt. Ez viszonylag új keletű technika – legalábbis valós időben – itt a tesszalláción és a vertex transzformáción van a hangsúly. Ez a technika, együtt is alkalmazható a normal mappinggal.

KÉP – displace.png: Normálmap és displacement. Látható, hogy az elsőn a kontúr kör marad, a másodikon a geometria térbeli.



15. ábra. Cubemap. Így lehet elképzelni a dobozra vetített égboltot, a tükörképen a hibák nem túlzottan zavaróak.



16. ábra. Tükröződés: Egy vízfelületen elengedhetetlen a tükröződés megjelenítése.

2.2.1.9. Dinamikus, valós idejű árnyékok

Egyik legalapvetőbb fényjelenség, mégis elég nagy problémát okoz, mivel nagy a számításigénye. A legnagyobb probléma, amit meg kell kerülni, az az, hogy mikor egy háromszöget rajzol a videokártya, a működési elve miatt a jelenet többi geometriai eleméhez nem fér hozzá, így nem tudja, hogy mikor esik árnyékos helyre. Mivel dinamikus fényekről van szó, a totális előreszámolás nem működhet, a megoldás a pillanatnyi előreszámolás.

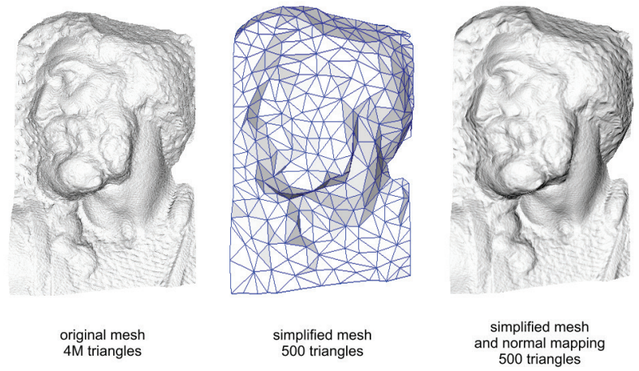
A fény „szemszögéből" készül egy renderkép, aminek egyedül a Z-buffere fontos, azaz a pixel kamerától vett távolsága. A fő kamera szemszögéből készülő kép közben minden esedékes pontnál kiszámoljuk, hogy a fény „szemszögéből" - szakszerűen a fény projekciós mátrixszával transzformálva - milyen távol van. Ha az árnyék textúrában levő távolság kisebb, mint az újonnan kiszámolt, akkor az a pont árnyékos, ellenkező esetben nem. Optimalizációs kérdés a textúra felbontása, hiszen minél nagyobb, annál részletesebb az árnyék, de a számítási igény is nő.

KÉP – shadow.png: Valós idejű árnyékvetés.

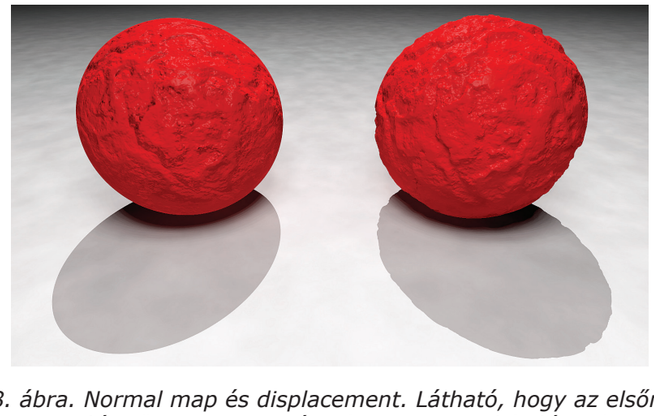
Persze a fény többféle lehet. A párhuzamos fény, mint a napfény, egy orthografikus vetítést kap, mivel a textúra véges, el kell dönteni, hogy milyen távolságig renderelünk árnyékokat. Másik egyszerű fényforrás a spotfény, itt is egy textúrába lehet renderelni, csak itt perspektív vetítéssel. A legbonyolultabb a pontszerű fény, itt nem elég egy kép, általában 6 térrészre bontják, hogy egy kockára lehessen ezeket a textúrákat renderelni. Az árnyékvetés sokszor utómunkával is megoldható, ha elegendő adatot tárolunk. Maga a technika főképp pixel szinten történik, bár a plusz render lépés az egész pipeline újbóli lefutását igényli.

2.2.1.10. Nagy dinamikatartományú (HDR: High Dynamic Range) kép

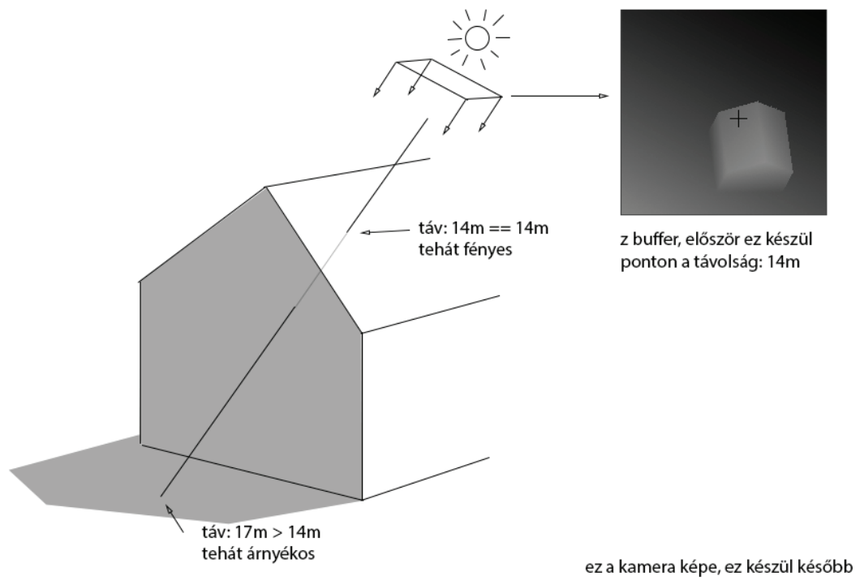
A valóságban a fény intenzitása igen nagy tartományban változik. Ha egy házra kívülről nézünk rá, belül sötétnek látjuk, de ha bemegyünk, a szemünk igazodik hozzá, és jól látunk. Erre nyújt megoldást a nagy dinamikatartomány. Ezt esetünkben több helyen alkalmazhatjuk. Az előreszámolt fények (lightmapek) készülhetnek HDR technikával, így egy külső és belső teret is tartalmazó jeleneten dinamikusabb fényhatások lehetnek. Másik igen szép alkalmazása az environment mapeknél jöhet elő, így egy vízfelületen megcsillanó nap is vakító lehet. Maga a HDR technika nem túlzottan bonyolult, gyakorlatilag kiterjesztjük az értékkészletet, a pixel értékek itt általában lebegőpontos számábrázolással vannak tárolva. Így az általános képformátumok nem is felelnek meg rá, de pl. a



17. ábra. Normal mapping. A felületi részleteket ha a textúrába visszük, nagyon hasonló eredményt érhetünk el kevesebb poligonnal.



18. ábra. Normal map és displacement. Látható, hogy az elsőn a kontúr kör marad, a másodikon a geometria térbeli.



19. ábra. Valós idejű árnyékvetés.

.HDR formátum pont erre lett kitalálva. Alap szintű pixel shaderekkel megoldható technika.

2.2.1.11. Utómunka - mélységélesség (DOF), bloom, motion blur, ambient occlusion

Az utómunka nem egy feltétlen szükséges munkafázis, viszont sokszor szebbé teszi a képet, illetve néhány hatás csak így oldható meg valós időben.

Gyakori utómunka a mélységélesség, angolul depth of field. Ezt az elkészül kép Z-bufferének segítségével lehet kiszámoltatni, egy kellően nagy sebességű elmosó algoritmussal (pl. fast-gaussian blur), a képet a kamerától való távolság függvényében mossuk el.

A bloom effekt azt a hatást próbálja utánozni, amikor olyan nagy fényesség különbség van két közel álló pixelen, hogy a fényes rész áthat a szomszédos területekre is. Ezt a jelenséget a gyakorlatban könnyen tudjuk közelíteni. Ha nagy dinamikatartományban dolgozunk, teljesen helyes eredményt kapunk, ellenben hamisat, de mindkét esetben szépet. Szeparáljuk a képnek a nagyon fényes részeit, azokat elmossuk, majd hozzáadjuk az eredetihez.

A jelenetben előfordulhatnak olyan elemek, amik nagy sebességgel mozognak a kamerához képest. Ebbe statikus modellek is beleértendőek, ha a kamera nagy sebességgel mozog vagy forog. A valós életben, ha egy fényképezőgépet gyors mozgás közben exponálunk, a kép elmosódik a mozgás irányába. Ez egy állóképen zavaró, viszont egy videót – vagy esetünkben animációt – valósághűbbé tesz. Ezt a jelenséget angolul motion blurnek hívják, magyarul mozgás elmosódásnak lehetne fordítani.

Ezt a technikát megvalósítani úgy lehet, hogy a renderelés során a pixelekhez hozzá számítjuk a helyben látszódó felületi pont sebességét is. Ebből egy irányított elmosással, a sebesség mértékétől függően szimulálhatjuk az adott hatást. Egy fokkal igénytelenebb módszer az előző képkockák lépcsőzetes elhalványítása, ez ritkán vezet szép eredményhez, viszont sokkal gyorsabb módszer.

Az ambient occlusion-ről már volt szó a nem valós idejű render engineknél. Létezik egy „olcsóbb”, nem tökéletes, de ehhez mérten szép változata, amit valós időben is ki lehet számítani. Ez a screen space ambient occlusion (SSAO), erőforrásként a Z-buffert használja, az adott pont környezetét vizsgálja, kernereléssel gyorsítva. [UDKmaterials][3dgraf]

20. ábra. HDR égbolt. Összehasonlítás a Half-Life 2: Lost Coast techdemóban.

21. ábra. Bloom. Látható, hogy a világos háttér fénye átterjed a sötétebb részekre.

2.3. Építészeti hasznosításuk

Nem valós idejű végeredményként álló- és mozgóképeket vagyunk képesek létrehozni, ezzel szemben a valós idejű megoldások interaktív tartalmak generálására lettek kifejlesztve. Egyfajta hierarchia is felállítható a végtermékek között, míg egy interaktívan bejárható virtuális térben tetszőlege térbeli görbe mentén készíthetünk animációt, majd az animáció tetszőleges képkockájából állóképet, ugyanez a fejlődés invertálva nem valósítható meg.

Valós idejű megoldásokat a BIM szoftverek mellé biztosít néhány fejlesztő, elég a BIMx-et megjegyeznünk, ami ArchiCAD modellek prezentálására készült. Ezek a programok jelenleg nem túl fotorealisztikusak, a műszaki dokumentáció és 3D információ összekapcsolását teszik lehetővé.

Valószínűleg a legegyszerűbb módja az épület virtuális sétára alkalmassá tételének ez. Azonban a vizuális megjelenítés finomhangolásának lehetősége nem adott. Rengeteg lehetőségtől fosztanánk meg magunkat, a témában szerzett ismereteink nagy része feleslegessé válna, ha csak ezek használatára szorítkoznánk.

Leginkább ez, és a kíváncsiság vezérelt minket, hogy ezt a feladatot ennyi munkával oldottuk meg. A mellékelt ábrákon látszik a különbség az ArchiCAD viewportjában látható kép, az ott renderelt kép, a BIMx-ben látható eredmény, és az Unreal Engine-ben séta közben látható eredmény között (az adott programban készült képek a következő oldalon tekinthetőek meg). Ez utóbbi az egyik tetszőlegesen választott játékmotor, amit a dolgozat írásakor, az általános vélemények olvasása, illetve demonstrációs videók megtekintése után választottunk. A döntés mögé pusztán esztétikai szempontok sorakoztathatóak fel, választhattunk volna más, széles körben használt engine-t, mint például a Crysis-t elkövető CryEngine.

Érdekes összehasonlítás, ha az interneten végzett kutatómunka során kiemelt, gyakorlatilag azonos kompozícióról készített két képet összehasonlítunk: az egyik öt éve készült, az építészeti vizualizációban gyakorlatilag napjainkban is leggyakrabban használt szoftverkombináció, a 3ds Max és Vray erejét demonstrálja, míg a másik idei alkotás: egy képernyőkép az általunk is használt játékmotorban elkészített animációból.

Az, hogy a legjobb valós időben bejárható munkák készítői eljutottak az öt évvel ezelőtti előrenderelt animációk szintjére, gyors terjedést prognosztizál a közeljövőben, habár a téma találgatása a jelenleg legelismertebb, elképesztő minőséget asztalra tévő művészek körében is heves viták tárgyát képezi [CORONAforum]. Az általános vélekedés az, hogy a befektetett munka mennyisége borzasztóan sok, nem éri meg foglalkozni megélhetés szintjén a témával, persze szórakozásnak remek. Nem szabad

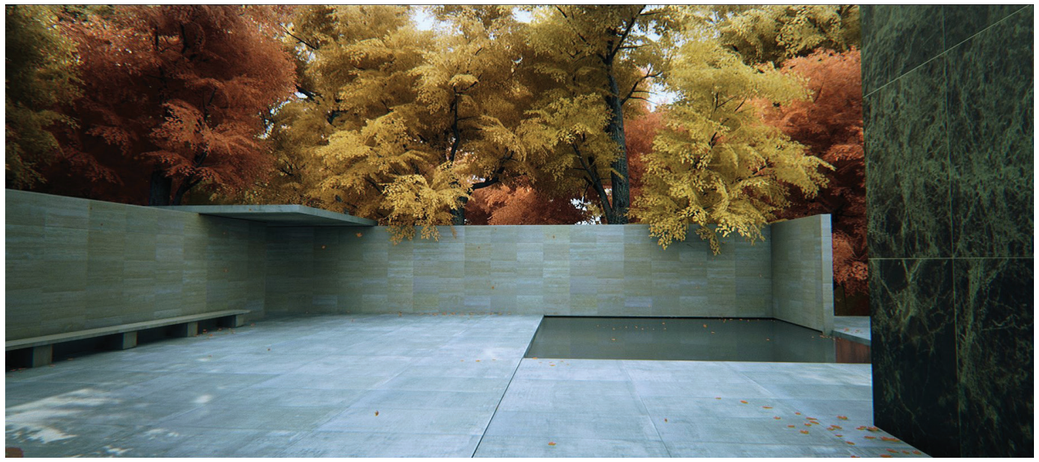
22. ábra. Betonfalról készített render. A projektjeinkről számos, atmoszférikus képet készíthetünk.

összetéveszteni az ő szempontrendszereiket a miénkkel, a tökéletesen fotorealisztikus, utolsó morzsáig kidolgozott jelenetek alatt bőven vannak még lehetőségek az elfogadható szinten felül, ráadásul ez a professzionális szcéna véleménye, akik mások tervei életre keltéséből élnek. A mi célunk megfogalmazható a megrendelő plusz szolgáltatásokkal megnyeréseként is.

Szóba került, hogy a munka mennyisége miatt nem foglalkoznak mélyebben a témával. Mi hát az a projektméret, ahol ez a többletenergia megtérül? Ismert példák híján a kérdés megválaszolására nem vállalkozunk, inkább felvetéseket teszünk. A technológia véleményünk szerint alapvetően a nagy, hosszú évekig előkészített beruházások esetén fordulhat elő gyakrabban a jövőben, mely projektek esetén a mindenki számára egyértelmű vizualizáció által okozott biztonság megtérülhet. Biztonság alatt itt a térformálást, anyaghasználatot illető WYSIWYG (what you see is what you get) jelenléte lehet meggyőző, ami kiegészül az interaktív megoldások (valós időben változtatható anyagok [UEinteraktiv], berendezési sémák) adta szabadsággal, tetszőleges számú variáció tekinthető meg egymás után ugyanazon térben állva. A videóban látható, hogy a kiadott produktum esetünkben egy különálló program, mely játékszoftverekre emlékeztet, így az adott programot futtatni képes hardver meglétén kívül nincsen további megkötés, például a megtekintés helyére vonatkozóan (ami a saját, zavartalan környezetben történő megvitatást, így felelősségteljes döntést elősegíti).

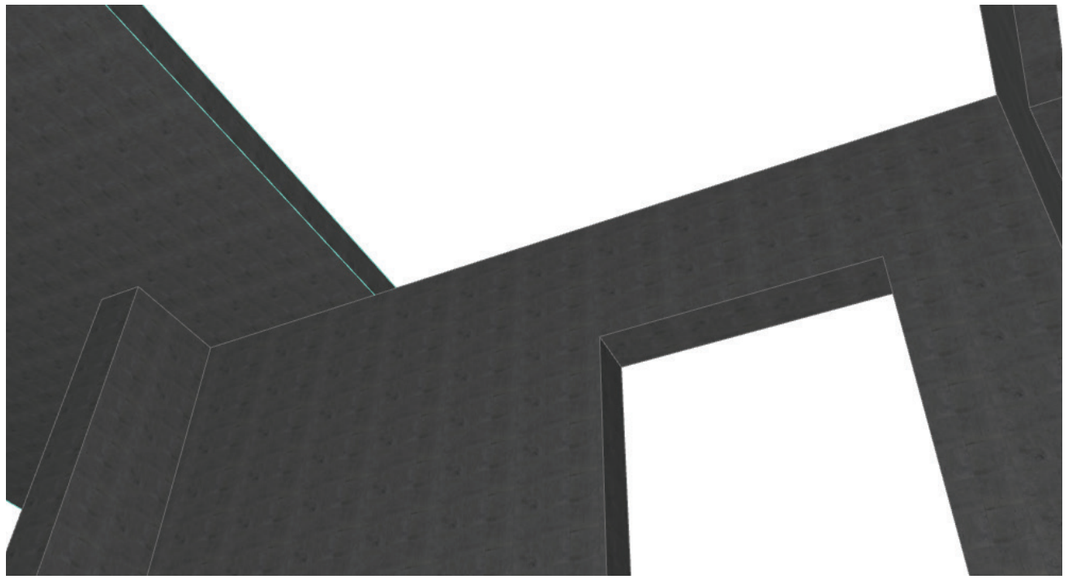
A terv kidolgozottsága szempontjából is vannak hatásai a valós idejű „látványtervek” alkalmazásának, leginkább a terv kidolgozottságában fedezhetjük fel a leghasznosabbat: egyszerűen nem adódik lehetőség elsiklani a részletek felett.

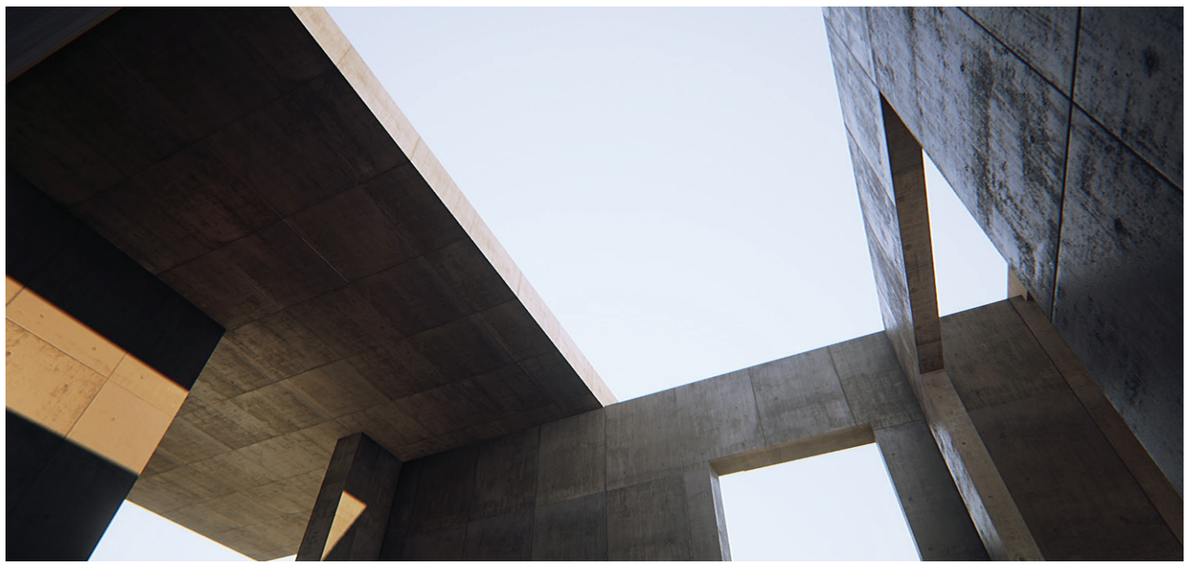
23. ábra. Részlet Alex Roman The Third & The Seventh című ikonikus művéből. Vray, 2009 [ROMAN3rd7th]

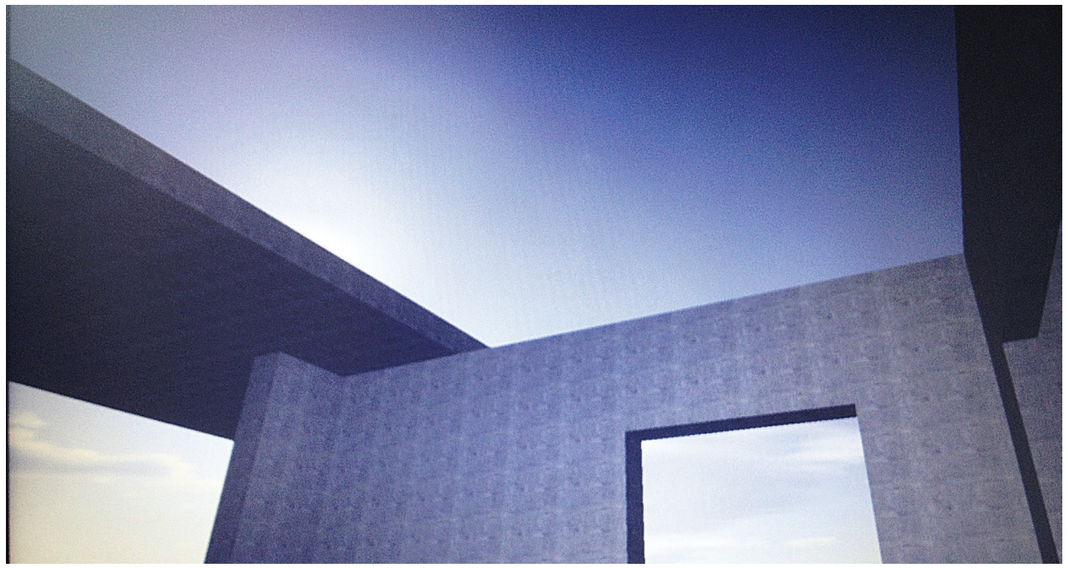


24. ábra. Részlet koola néven posztoló felhasználó alkotásából. Unreal Engine, 2014 [KOOLA2014]

24-27. ábra. Jobbra fent: ArchiCAD viewportból készített képernyőkép - állókép
Balra fent: ArchiCAD render (ArchiCAD 18, Cinerender physical renderer) - állókép
Jobbra lent: BIMx 3D nézet - mozgókép (interaktív). A monitort fényképeztem, a színek ennek tudhatóak be.
Balra lent: Unreal Engine 4 szerkesztőjének nézetablakáról készített képernyőkép - mozgókép (interaktív)





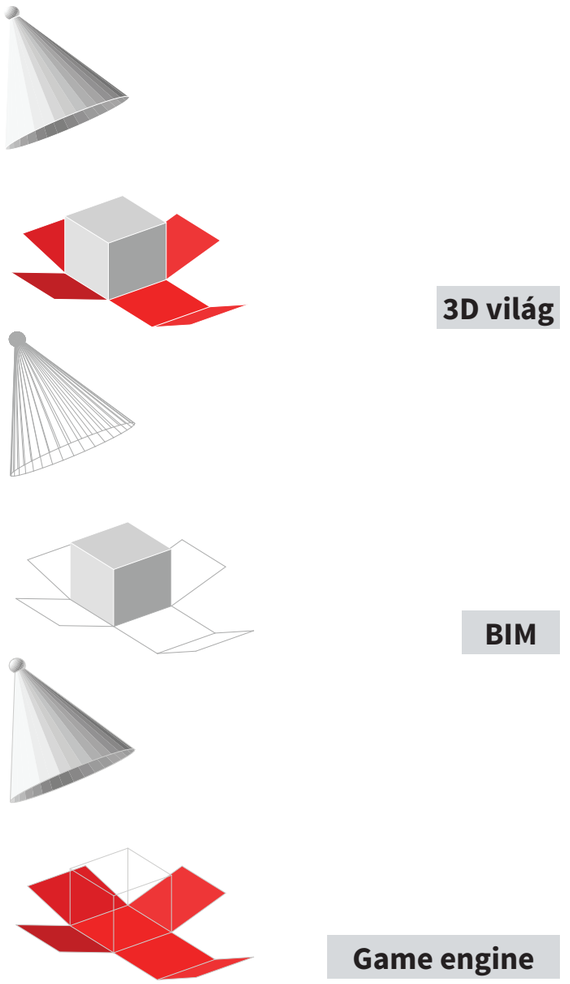


3. BIM / JÁTÉKMOTOROK ÖSSZEHASONLÍTÁSA

3.1. Az összehasonlítás alapja

Minden 3D modellező program használata során alapvetően háromfelé csoportosíthatjuk az elvégzendő feladatokat: létre kell hozni a geometriát, ehhez anyagokat kell rendelnünk, majd az egész jelenetet meg is kell világítani.

Adódik, hogy az egyes csoportok kidolgozottsága a végfelhasználás szempontjai szerint igencsak eltérhet: BIM szoftverek esetén a modell célja az épület geometriai viszonyainak rögzítése (ütközésvizsgálat, csomópontok kinyerése…), majd a geometriához további - elsősorban nem látható - információ rendelése. Más a helyzet akkor, ha a cél vizualizáció: a részletgazdag modelleken felül az anyagok kidolgozottsága is jelentős, a megvilágítás pedig meglehetősen komplex. Játékok esetén törekszünk alacsony poligonszámmal dolgozni, a részleteket textúrák segítségével megjeleníteni (magas poligonszámú modellekről készült textúrákat alacsony poligonszámú modellekre teszünk - texture bake), a megvilágítást pedig az adott pályaviszonyhoz mérten kidolgozni, ennek függvényében alkalmazhatunk statikus, vagy dinamikus fényforrásokat, ezeket később részletezzük.



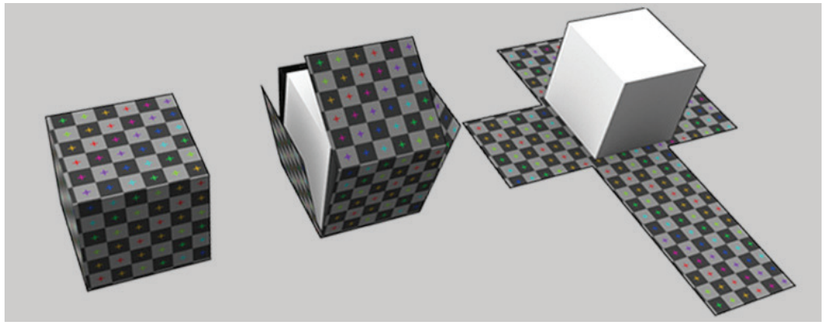
28 ábra. A fenti illusztráció a 3D világ 3 alappillérét, illetve azok egyes szoftverekben betöltött fontosságát hivatott szemléltetni. A teli kitöltésű illusztrációkra nagyobb hangsúly helyeződik, míg a drótvázas megjelenítés mellékes szerepet tölt be.

3.2. Megoldandó problémák

Amint azt láthatjuk, a két szoftvercsoport nem ugyanabban erős, ezért a BIM modell játékmotorokba migrálását számos nehézség övezi, amelyeket nem lehet kikerülni, hiszen egyből látható hatásuk van a végeredmény céljából. A legfontosabbak az alábbiak:

3.2.1. UV map-ek

Egyik legfontosabb feladat a megfelelően előkészített geometria biztosítása a játékmotor számára: mint azt a fentiekben már említettük, a játékok megjelenítésében fontos szerepe van az előre számított megvilágításnak (precomputed lightmaps). Hogy ezt kihasználhassuk, szükség van minden egyes objektum számára kettő darab UVW map-re (általában UV map-nek nevezik, mivel síkbeli textúrák használata az általános), melyek közül az egyik (nagyobb felbontású) a textúrák számára, míg a másik a lightmap fogadására készül. Az előbbiek számára nem szükséges (bár ajánlott) feltétel az összemetsződések elkerülése, a lightmapeknél ez kikerülhetetlen kritérium: egymásra fedő poligonok esetén a kapott megvilágítás furcsa, töredezett lesz, abszolút nem esztétikus végeredményt adva, bizonyos játékmotorok adott százalék átfedés felett ki sem számolja az adott testre vonatkoztatott lightmap-et, így megelőzve a hibás végeredményt.

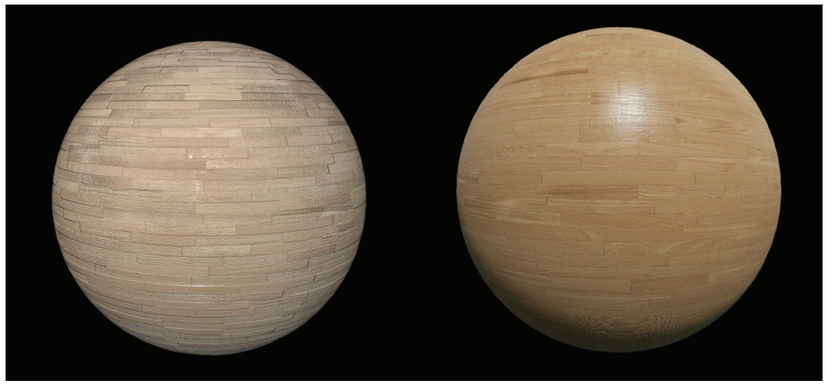


29 ábra. Kocka UV map-jének kiterítése. A középső képen a felvágások menti éleket nevezzük seam-nek. A talaj síkja az UV sík, amelyben a kockára vetítendő textúra fekszik.

3.2.2. Anyagok

További kérdések merülnek fel az anyagok kapcsán: minden egyes szoftver külön felfogással építi fel az anyagokat, többé-kevésbé a valóságot közlítve, de nagyon ritka a kompatibilitás, ami a következőt jelenti: hiába dolgozunk részletes anyagkönyvtárunkkal épületinformációs modellezés közben, ha a tárolt felületi információkat a következő lépcsőben már nem hasznosíthatjuk. Márpedig a legtöbb esetben nem is ezzel szembesülünk, hanem egyszínű modellekkel, melyek ezen felül általában nem túl használható texture mapping információkkal rendelkeznek: a geometriát alkotó vertex-ek térbeli helyzete alapján generált textúra koordináták állnak a rendlekezésünkre.

Még komplikáltabbá teszik a munkát a túlságosan sokféle anyagot tartalmazó modellek: material alapján a legtöbb 3D modellező szoftver remekül tud szűrni, az elaprózódás emiatt kerülendő.



30. ábra. Fa. Bal oldalon: Corona for Cinema 4D. Jobb oldalon: Unreal Engine 4.4. A textúra saját fotózás, ugyanabból a fotóból készült a diffuse, a normal, és reflection map. Látható, hogy hiába törekedtem hasonló eredményre (próbáltam egymásnak megfeleltetett csatornákat használni közel azonos értékekkel), a kapott renderek merőben eltérnek. Ennek oka egyrészt keresendő az említett okokban, a más anyagkezelésben, de a color mapping beállítások is szerepet játszanak az árnyalatbeli különbségekben.

3.2.3. Geometria

A geometria kapcsán jó kiindulási alapunk van, hiszen éppen ebben a legerősebb egy BIM alkalmazás a három kategória (modellezés-anyagozás-bevilágítás) közül. Nem jellemző (tapasztalataink szerint) hiányzó poligonok előfordulása sem (backface culling).

Az egyik, sokszor tapasztalt problémát a túlzott poligonszámú modellek jelentik, sok esetben előfordult a dolgozat készítése közben, hogy a használt játékmotor (Unreal Engine) a modell importálásakor lefagyott. Amennyiben nem fagy le, úgy a legelső pont okozhatja a legnagyobb fejfájást a nagy tárgyszám miatt: kisebb épületek (szobák) esetén is sok munka megfelelően UV-zni a modelljeinket, kézi erővel az ismereteink szerint rendelkezésre álló módszerekkel egy komplexebb építményt emberfeletti idő- és energiabefektetéssel lehetséges megcsinálni.

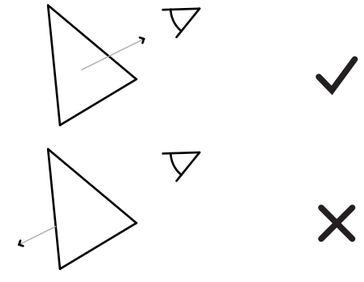
Hogyan célszerű tehát modellezni? Mindenféleképpen megfelelő LOD (Level Of Detail - a kidolgozottság mértéke) megválasztása szükséges, ebben a léptékfüggő parametrikus tárgyak jelenléte hatalmas segítséget jelent. Nem célszerű továbbá céltalanul magas poligonszámú berendezési tárgyakkal telezsúfolni az épületünket, részletgazdag székek, asztalok, ágyak, bútorok, függönyök rendelkezésre állnak más modellezőszoftvereken belül is, amelyek sokkal jobban kezelik a nagy poligonszámú jeleneteket. Bevett szokás ilyenkor proxy-k (egyszerűbb helyettesítő tárgyak) használata, a BIM-beli tárgyakra gondolhatunk ekként is. E megközelítés előnye, hogy általában már készen lévő, megfelelő UV mappel, anyagokkal rendelkező modelleket használunk (rengeteg ingyenesen letölthető található), így a BIM szoftverben történő fölösleges modellezés könnyen elkerülhető.

Leginkább ezen a területen, a poligonszámhoz való viszonyban mutatkozik nagy eltérés a játékfejlesztés és épületinformációs modellezés között: míg az előbbi célja a lehető legkevesebb poligonszámra való törekvés, egy épületmodell elkészítésénél ez az utolsó szempontok között van, hiába nem látszódik a szerkezet 3D séta közben, egy jól felvett metszettel az utolsó födémpalló körüreges kikönnyítése is bármikor láthatóvá tehető.

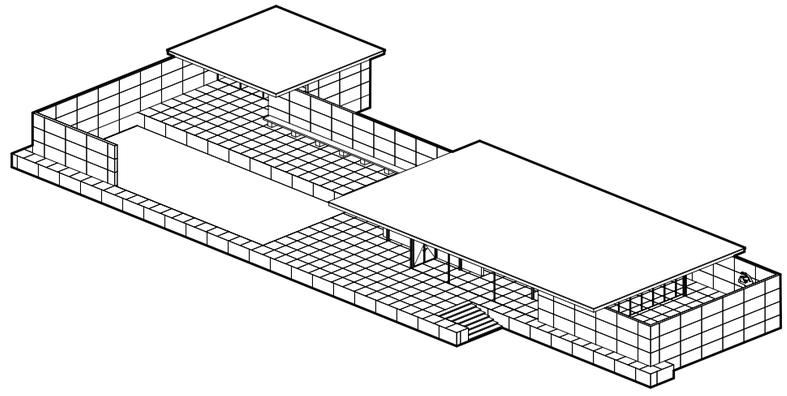
Nem alkalmazható tehát a jól bevált módszer, miszerint csak azt modellezzük, ami látszik, de helyette sokkal hatásosabb eszköz a léptékfüggő megjelenítés, mivel ez a 3D modell felbontását is nagyban befolyásolja (az íves tárgyak simaságának, tehát az ívek szegmensszámának megfelelő megválasztásával felesleges poligonok ezreit lehet eltüntetni), illetve kisebb-nagyobb részletek (ajtókilincs) szintén modellnézet-beállításokkal eltüntethetőek.

A modellezés a munka jelen esetben a kisebbik felét teszi ki, a próbák során az úgynevezett asset management sokkal több időt, gondolkodást igényelt, mint a geometria elkészítése.

Persze a gyakorlati tapasztalatok növekedésével az erre fordított idő jelentősen csökkent, de néhány gondolatot, tapasztalati összegzést szeretnék megosztani: a fentiekben már találkozhattunk egy ábrával, amit Mies van der Rohe Barcelona pavilonjáról készítettem. Végül ezt a modellt nem használtam föl a magas elemszám miatt, de remek példa vált belőle. A modellt egy az egyben nem



31. ábra. Backface culling. Amennyiben az adott poligon normálvektora nem a nézőpont felé mutat, úgy a poligon nem látható. A poligon körüljárási iránya (a vertexek lehelyezésének sorrendje) meghatározza a normálvektor irányát.



32. ábra. Példa a szükségesnél részletesebb modellre: a burkolatkiosztást célszerű lett volna textúrával (normal map) megol i az egyes burkolati elemek megmodellezése helyett.

tudtam áthúzni valós idejű környezetbe, a konkrét korlátot (fizikai méret, poligonszám, elemszám) nem tudom maximális biztonsággal meghatározni, valószínűleg az elemszám okozhatta az Unreal Engine lefagyását a modell importálásakor. Ennek oka a pavilon burkolatainak egyesével történt megmodellezésében keresendő. Megoldás erre, hogy a burkolati elemeket egy elemmé konvertáljuk, de ebben az esetben a modell frissítése fog problémát okozni (a példa esetében a frissítés úgy történik, hogy egyedi azonosítók alapján –object ID- generál elemtípuskonként mappákba rendezett tárgyakat a modellezőprogram, tehát az n tárgy egy objektummal való helyettesítése esetén a többi n-1 darab hiányzó ismételten be fog importálódni, így a csoportosítások-törlések ismételt manuális elvégzése szükséges), amiről következő fejezetben lesz részletesebben szó.

3.3. Általános elvek

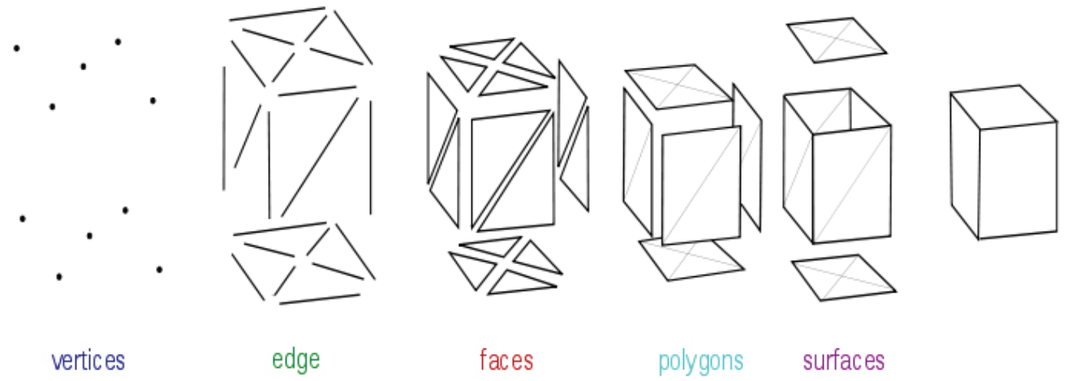
A szimultán programhasználat mellett nem szabad megfeledkeznünk a végcélról: valós időben mutatjuk be a tervünket az utolsó részletekig. Ehhez hasznosnak tartjuk, hogy általános érvényű gondolatokban lefektessük a modellezés-anyagozás-világítás vezérelveit.

3.3.1. Modellezés

BIM szoftveren belül adott építőelemekkel dolgozunk, ebből kifolyólag ritka, hogy poligonszinten kelljen bármit is felépítenünk, mégsem árt tisztában lenni az építőelemeinkkel: pontok, élek, poligonok ők (vertex, edge, polygon). A poligonok hálót (mesh) alkotnak, melyek tárgyakat (object) formáznak, a tárgyak pedig további csoportokat alkothatnak. Minden egyes poligon rendelkezik egy normálvektorral, mely az elő- és hátoldal megkülönböztetését segíti: a normálvektor irányába néző felület a poligon elülső oldala. A normálvektor a poligont súlypontjában döfi.

A mai modellező programokban egyre jobban elterjedt a három és négyszögek helyett a sokszög felületek használata. Ezek megkönnyítik a modellezést, viszont oda kell figyelni arra, hogy a videokárta csak háromszögeket tud renderelni. Érdemes még a game enginebe helyezés előtt háromszögesíteni, hogy lássuk, az automatikus háromszögesítés nem-e végez rossz munkát valahol.

A valós idejű renderelésben kiemelkedő fontosságú a felületek helyes normálvektorok meghatározása, ugyanis gyakran alkalmazzuk a hátlap eldobást (backface culling). Ez azt jelenti, hogy amerre a normálvektor áll, csak az az oldala lesz kirenderelve, tehát hátulról nézve nem látszik. A legtöbb modellező programban van automatikus felismerő algorimus, ami “rendbe rakja” a normálokat, ezek kitűnően működnek. Vannak olyan esetek is amikor egy lapos tárgyat egy kétoldalú lappal közelítünk. Ilyenkor vagy kikapcsoljuk a hátlap eldobást, vagy kétszerezzük a lapot és megfordítjuk



33. ábra. A 3D modellezés során alkalmazott terminológia szerinti építőelem-hierarchia.

az egyiket.

Jelen esetben a modellezés BIM szoftverben történik, így konkrét poligonmodellezéssel ritkán találkozunk, akkor is leginkább korrekciós feladatok formájában: normálvektor-irány változtatást, topológia-optimalizálást végzünk legtöbbször. A feladat így csupán annyi, hogy a 3 szoftver között biztosítsuk a geometria állandóságát, tehát a BIM-beli változásokat szinkronizáljuk a game engine-beli megfelelőjével.

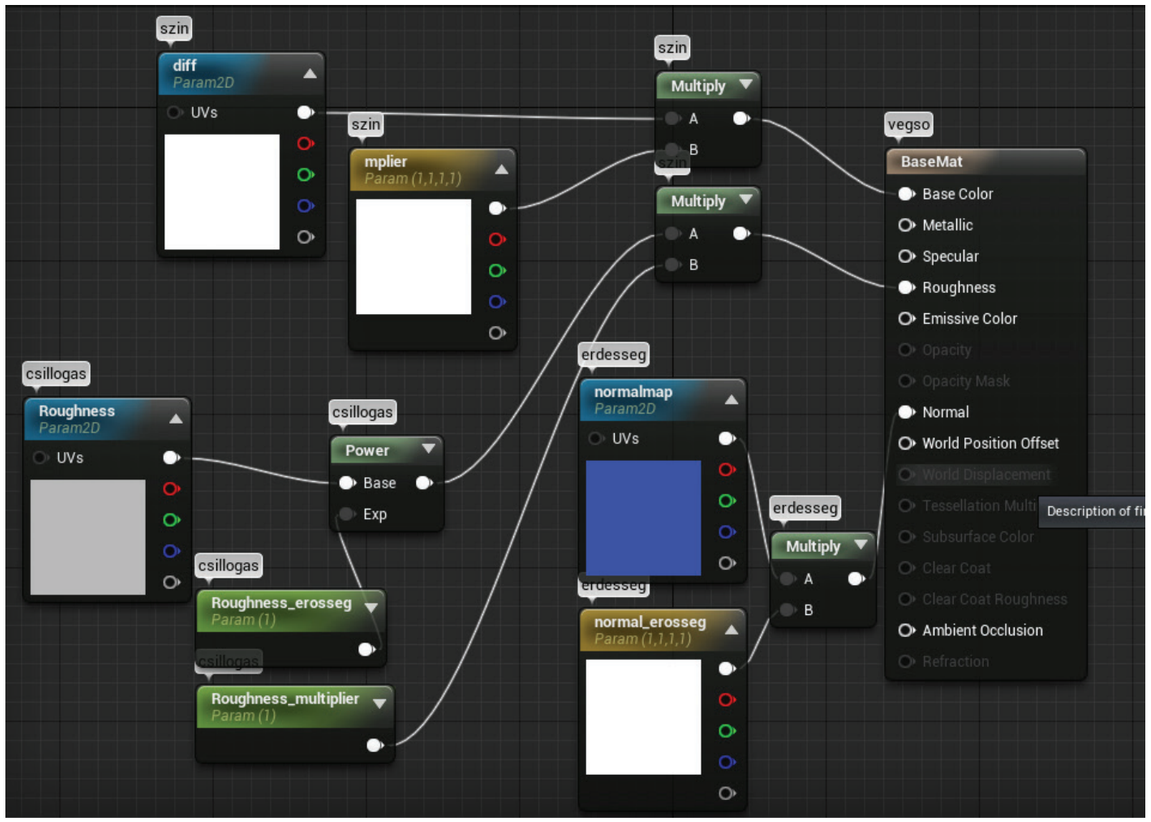
Rengeteg fájlformátum létezik, ezek között megtalálni a számunkra megfelelőt kihívás jelent. Olyanra van szükségünk, amely a geometrián túl anyagokra vonatkozó információkat is képes tárolni, ráadásul kedvezően frissíthető. Ez utóbbi kritérium sajnos nagyon szoftverspecifikussá redukálja a kérdést, melyet tapasztalataink alapján megválaszolva: ArchiCAD-ből .c4d formátumban Cinema 4D-be exportálva a modell anyaginformáció gyakorlatilag teljes mértékben átmennek, elemtípus szerint jól strukturált modell-hierarchiát kapunk, mely csoportosítás a game engine-en belül is megmarad, könnyű átanyagozási lehetőséget biztosítva. [C4Dmerge, C4Dupdate]

3.3.2. Anyagkészítés

Első ránézésre elrettentő, ahogyan minden szoftver máshogyan kezeli a valódi anyagokat, de az alábbi összefoglaló segíthet az univerzális elvek megértésében. Aki eltöltött némi időt látványtervezéssel, annak nehéz az őt körülvevő világot anélkül szemlélni, hogy ne bontaná az őt körülvevő felületeket a virtuális terminológia szerinti rétegekre. Milyen színe van? Mennyire tükröződik? Hogyan tükröződik? Mennyire átlátszó? Hogyan tori a fényt? Mennyire érdes? Majd ezekre a kérdésekre különféle mapek formájában választ is ad. Map alatt az egyes jellemzőket meghatározó textúrát értünk, ezek sokszor a diffuseból derivált, de néha procedúrális textúra is lehet. Map-ekre akkor van szükségünk, ha egy felületen belül változik az értéke az adott tulajdonságnak.

Az egyes kérdésekhez tartozó mapek:

- szín – diffuse map
- csillogás mértéke és színe - specularity map
- tükröződés mértéke – reflection map
- áttetszőség (átlátszóság) mértéke és színe – refraction map



34. ábra. Parametrikusan felépített alapanyag. A parametrizálás roppant hasznossá válik a későbbiekben, az Unreal Engine-el készített példákban az összes tömör anyag ennek az anyagnak az instance-e. A különféle anyagok megjelenítése érdekében a textúrákon kívül még néhány konstans paraméter szükséges a teli színek, illetve többszöröző értékek miatt. Némi utánajárással ennél lényegesen bonyolultabb anyagok is összeállíthatóak. természetesen az egész három node-al is működik, csak kevésbé lesznek testreszabhatóak az anyagaink.

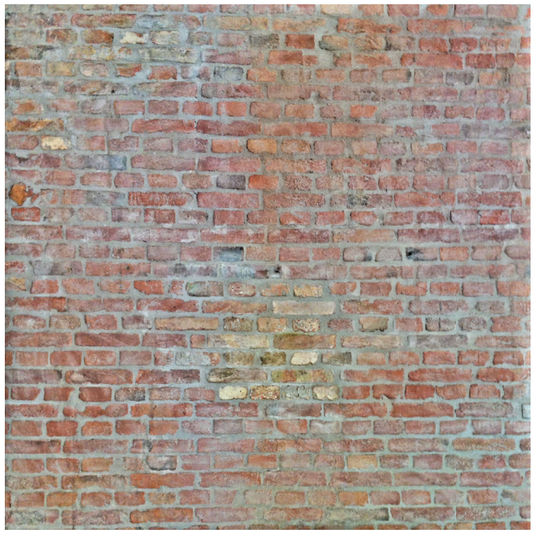
- érdesség – bump/normal map
- világítás mértéke és színe - emission map
- stb.

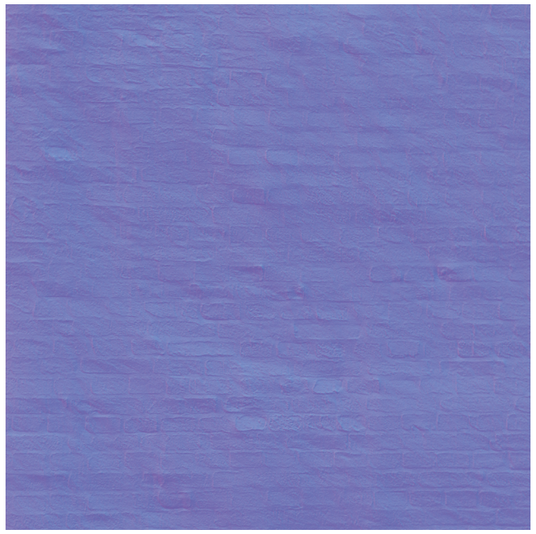
Mivel esetünkben a cél az interakció ezért alapvetően a szín, tükröződés, illetve az érdesség definiálásával már elfogadható eredményeket kaphatunk, persze minél összetettebb egy anyag (több csatornát határozunk meg), annál realisztikusabb lesz a végeredmény, itt is egyéni mérlegelés kérdése a kidolgozottság mértéke. Ez a három csatorna egy bitmapből kiindulva, abból alterációkat készítve betölthető: a diffuse mapből egy fekete-fehér képet készítve a tükröződés, egy normal mapet generálva pedig az érdesség is letudható (viszonylag jó eredményeket tud felmutatni az ingyenes nVidia Photoshop plugin). De léteznek erre az egy feladatra célszoftverek is, ilyen például a CrazyBump.

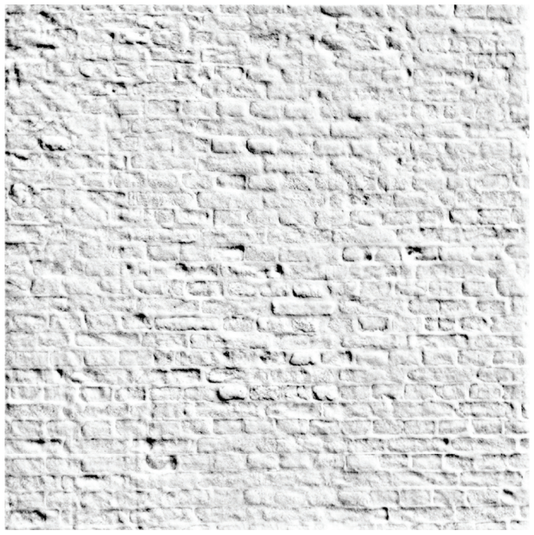
Jártunkban gyakorlatilag bármilyen felületet lefotózva (a célnak a mai okostelefonok is megfelelnek, így egy-egy gyártó kiállítótermében járva , vagy katalógusból befotózva egy-egy specifikus termék) annak anyaga pillanatok alatt elkészíthető. Fontos megjegyezni, hogy a nyers fotó nem alkalmas diffuse mapként történő használatra, mivel a színinformáción kívül további értékeket is tartalmaz: tükröződések, csúcsfények (specularity), árnyékok, bevetett árnyékok (ambient occlusion) egyaránt megtalálhatóak, melyektől meg kell szabadulni, különben nem kapunk helyes eredményt, miután a lightmapet kiszámoltattuk és alkalmaztuk a felületen. Ez helytelen fényességű pixelek formájában fog megjelenni a végeredményben.

Az anyagkészítésben új keletű hozzáállás a PBR metódus, mely betűszó a Physically Based Rendering elnevezésnek felel meg. Ez újragondolt material rendszerekben (Cinema 4D R16 rétegalapú standard material-je), gyakorlatiasabbá váló anyagparaméter-elnevezések (roughness), illetve az animációsfilm-készítés zászlóshajóinak (például a Disney) kísérletein, mérésein alapuló újabb és újabb táblázatokban, melyek egy anyagtípus alapszínét, IOR értékét, stb. gyűjtik össze. Például a valóságban a csillogás és a tükröződés ugyan az a fényjelenség, viszont a videokártyán ezek más módszerekkel közelíthetőek, a hagyományos módszerrel ezek mértékét külön külön kell megadni, de a PBR módszer ezt egyben kezeli. Bővebben a [DISNEYpbs] alatt tájékozódhatunk a PBR metódusról, azonban az idézett dolgozat messze túlmutat jelen írásunkon.

Valós időben támogatottak a számításokat optimalizáló megoldások, nem kivétel ez alól az anyagrendszer sem, ezen a téren material instance-k formájában jelenik meg a törekvés. A háttérben ezt azt jelenti, hogy ugyan az a shaderprogram más paraméterekkel teljesen más felületet képes kirajzolni. Ez lehetővé teszi, hogy egy jól felépített, paraméterezett material néhány változójának (főként az imént említett három csatorna, a szín, tükröződés, érdesség értékeit befolyásoló képek)







35-37. ábra. téglatextúrát a volt Honvéd Főparancsnokság épületében fotóztam, a kiállítótér nyers téglafala. Fent látszik az utómunkált diffúz textúra, mellette jobbra az ebből generált normal map, a szöveg mellett jobbra pedig a szintén generált occlusion map.

megváltoztatásával (képek cserélése) pillanatok alatt új anyagtípust hozzunk létre.

Az alábbiakban egy ilyen alaptípus felépítése látható, természetesen lehet sokkal részletesebb, de egyszerűbb is a hierarchia.

3.3.3. Unwrapping

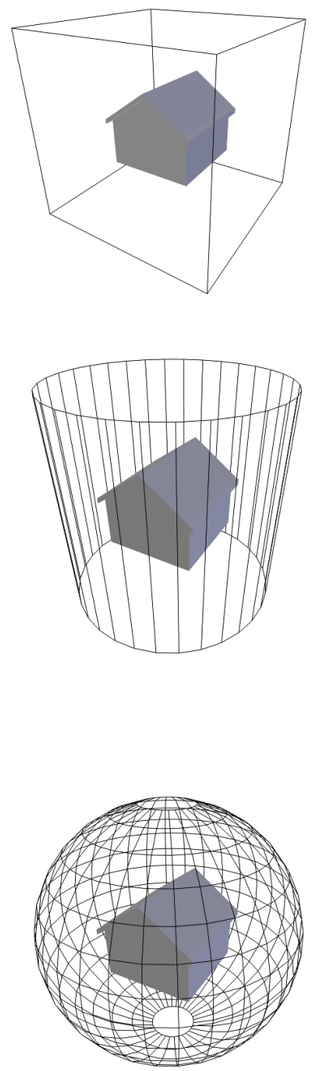
Az előbbiekben már volt szó arról, hogy BIM szoftverekből sokszor haszontalan vagy hiányos mapping információjú geometria exportálódik, ami textúrázás használatakor szerencsétlen. A probléma könnyen megoldható: másik szoftvert használunk a textúrák felvetítésének meghatározására.

A vetítési módok a következőek:

- flat (planar)
- box (cubic)
- cylindrical
- spherical
- camera mapping
- ...

Az esetek nagy részében ezek remekül használhatóak, hiszen a legtöbb építőelem geometriája ezek valamelyikével közelíthető (falak: általában téglatest -> cubic mapping, födémek: sík lemez -> planar mapping, oszlopok: hengerfelület -> cylindrical mapping, stb.). A vetítés ugyanis úgy történik, hogy a test körül a vetítési mód által meghatározott virtuális alaptestet veszünk fel, mely alapmérete vagy meghatározott, vagy a testünk befoglaló kockáját (bounding box) érinti – ennek módja az általunk használt szoftvertől függ, automatikusan megtörténik. A primitívek térbeli orientációja a világ-koordinátarendszerben meghatározható, a további módosítások (ezekre sokszor PSR-ként hivatkoznak, mely betűszó a position-térbeli helyzet, scale-nagyítás, rotation-forgatás szavakból ered) UV szerkesztő módban megtehetőek. A textúrák a virtuális primitív lapjairól (palástjáról, felületéről) vetítődnek, így a PSR módosítások tulajdonképpen a textúránk méretét, torzítását, valamint irányát határozzák meg. A vetítések természetesen textúránként is meghatározhatóak, ha szükséges.

Belátható, hogy a fenti módszerek a projektáló testekből származtatható geometria esetén



38-40. ábra. Cubic, cylindrical és spherical mapping vizualizációja.

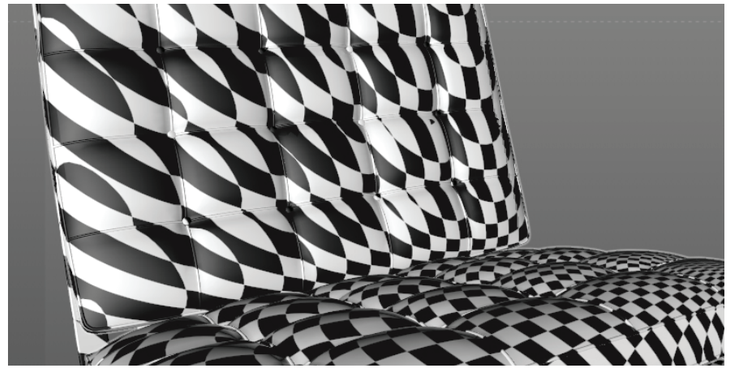
működnek a legjobban, eltérő esetben jelentős (esetünkben: nem hihető) torzulást okoznak. A torzulásmentes projekció geometriai feltétele, hogy a térbeli poligonra az adott textúrarészlet a poligon normálvekorával párhuzamos parallel vetítéssel kerüljön felvetítésre. (Más, a UV unwrapping szempontjából kézenfekvőbb megfogalmazással: textúra síkjába transzformált poligon a térbeli poligonnal egybevágó maradjon.)

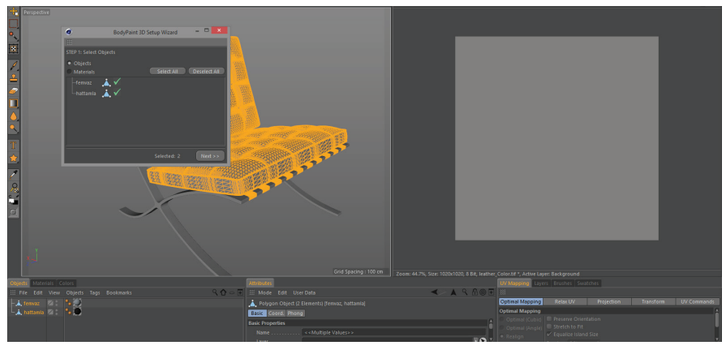
Ez a feltétel legegyszerűbben úgy teljesíthető, ha a 2D textúrára ráfektetjük a testünk síkbafejtett felületét: ábrázoló geometriát tanultak számára ismerős tézis, hogy nem minden test fejthető torzulásmentesen síkba, elég gömbfelületekre gondolnunk, az egyéb, alaptestekből nem származtatható testekről már nem is beszélve. Szerencsére itt minden görbe felület is szögletes modellhez van közelítve, így könnyebb darabokra vágni az élek mentén. Ez az eljárás a 3D modellezésben használt terminológia szerint a UV unwrapping: a világ térbeli koordináta rendszerében élő felülethálót a textúra UV koordinátarendszerébe fordítjuk.

Az alapvető eljárás úgy működik, hogy él kiválasztásokon keresztül meghatározzuk a vágási éleket, melyek mentén a szoftver szétválaszthatja a felületet, majd az így keletkezett bevágott meshen (háló) elvégzi a transzformációt. Amit célszerű szem előtt tartani: az élek száma, és a textúra torzulása közötti összefüggés: kevés él – torzabb vetítés, több él – torzulásmentesebb vetítés. Ezek alapján érdemes lehetne minél több darabba vágni a felületet, de az így megjelenő illesztési vonalak tömkelege ettől gyorsan elveheti a kedvet. Mivel a cél „csupán” a fotorealisztikus megjelenítés, ezért az élek számát javasolt érzésre belőni, míg a helyzetüket a testen a test jelenetben elfoglalt térbeli pozíciója határozza meg: minél kevésbé van szem előtt egy-egy seam (a vágási éleket nevezik így), annál tökéletesebbnek tűnik a vetítés.

Léteznek teljes mértékben automatizált UV kiterítő algoritmusok is, amik többé kevésbé jó eredményeket adnak. A legnagyobb probléma velük általában az, hogy túl sok kihasználatlan helyet hagynak a textúrán, olyan pixelek lesznek amihez nem tartozik felület. Ez csak akkor probléma, ha a textúra a modell számára van készítve. Főképp tárgyaknál és bonyolult alakú, egyedi textúrát igénylő modelleknél van erre szükség.

Ismétlődő textúráknál más a helyzet, a legtöbb építőanyag ebbe a kategóriába tartozik, itt nem számít milyen mértékben van kihasználva a textúra, a cél csak hogy a textúra léptéke, kiosztása jó legyen a felületen. Itt egy másik probléma jöhet elő, a textúra látványosan és csúnyán ismétlődik a felületen. Ez esetben egy nagyobb méretű textúrát szokás alkalmazni, de léteznek más technikák. Ilyenre példa a detail textúra-technika, aminek a lényege az, hogy gyakorlatilag két textúrával dolgozunk. Az egyik textúra léptéke 5-10x erese a másiknak, és a két textúra egymásra van blendelve. Az UV koordinátákat ilyenkor a nagyobb textúra számára adjuk meg, a részleteket adó textúra UV-ját valós





41-42. ábra. Fent az ArchiCAD-ből importált tárgy látszik a nem túl hasznos, vertexpozíció alapján generált textúra koordinátáival. Alatta az automatikus UV unwrapping-et elvégző varázsló kezdőablaka.

időbe szorozzák be a léptékváltásnak megfelelően.

Az alábbiakban egy egyszerű példán keresztül ennek gyakorlati lépéseit tekinthetjük át (az illuszrációk Cinema 4D-ben készültek, bármelyik nagyobb szoftverben a lépések analóg módon elvégezhetőek):

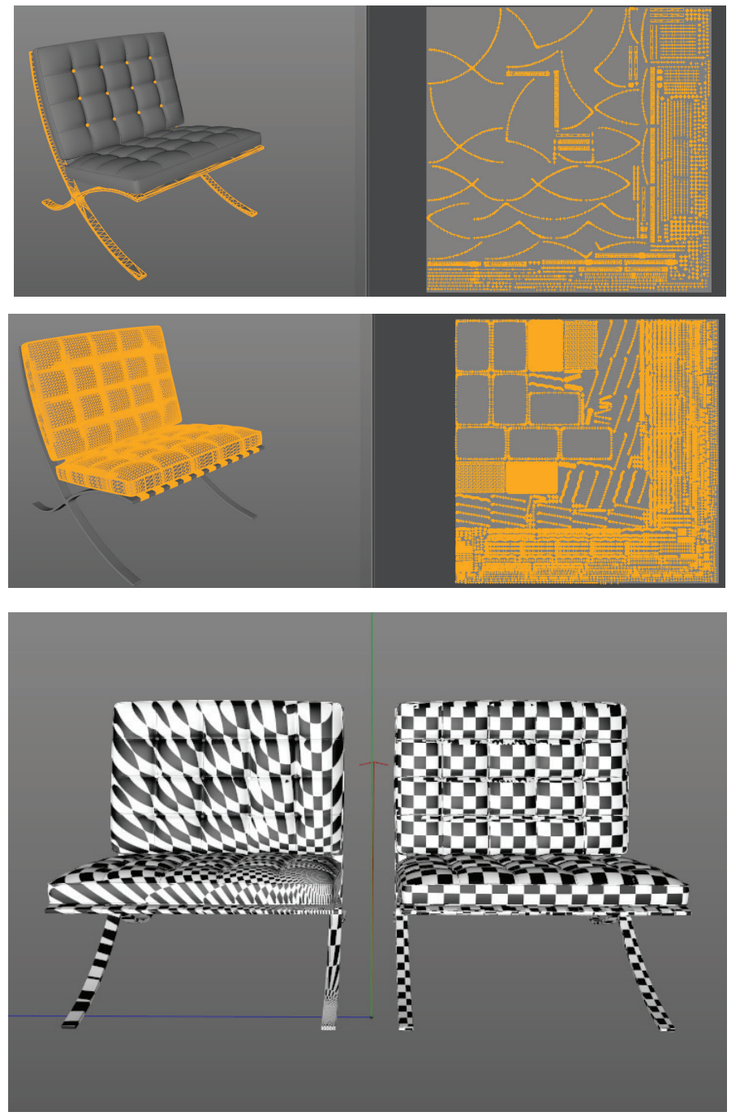
1. Az első képen a modellezőprogramba importált modellt láthatjuk egy generált, sakktábla-mintás textúrával. A fekete-fehér négyzetek a nagy kontraszt miatt jól elkülönülnek egymástól, az arányaik pedig a szemünk által megszokott, tehát a torzulások jól láthatóak lesznek. Mint látható, jelen textúra felvetítése nem helyes, vertexpozíció által generált. Mivel procedurális vetítések a szék formáját nem követnék le, ezért unwrappelünk. Lehetne manuálisan, nagyon tiszta munkát végezni, de automatikusan gyorsabb, meglátjuk milyen minőséget produkál.

2. Miután a programot UV szerkesztő módba kapcsoltuk (Layout -> BP UV Edit), a BodyPaint varázslóban az UV-zni kívánt tárgyakat kiválasztva a feladat automatikusan elvégzésre kerül.

3-4. Még az 1. pont előtt két darab, a későbbi anyagok alapján szétválaszott részre bontottam a széket, a fém- (3.) illetve bőr (4.) részek kiválasztva látszanak a bal oldalon, míg a jobb oldalon a textúra síkjába fektetett poligonok.

5. Összehasolítás: UV szerkesztés előtt (balra), illetve után (jobbra).

Ezek után a kapott UVW tag-et az adott tárgyon duplikálva exportálásra kész modellhez jutunk (ajánlott az FBX formátum használata, az FBX pipeline fejlesztői részről támogatott



43-45. ábra. Automatikusan generált UV layout, alatta az előtte-utána kép. Érdemes rászánni az időt.

3.3.4. Világítás

A fényviselkedés szimulációjának megoldásairól már szóltunk, ahogyan a global illumination illetve lightmap fogalmai is elhangzottak. Arról nem beszéltünk, hogy a fényforrásokat hogyan lehet csoportosítani, melyek a szimuláció bemeneti adatait biztosítják.

Megkülönböztethetünk természetes, illetve mesterséges fényforrásokat, de a 3D világában ennek a csoportosításnak túlságosan sok értelme nincsen. Méret, illetve alak szerinti csoportosításnak viszont annál inkább van jelentősége, hiszen a fényforrásainkhoz kapcsolódó paraméterek leginkább ezeket az értékeket befolyásolják.

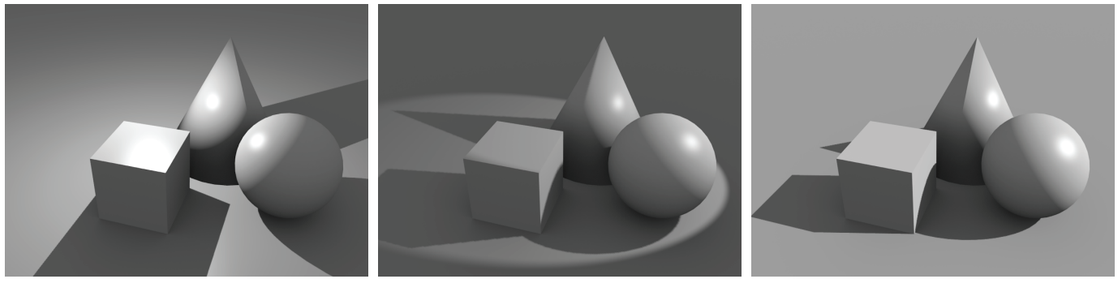
A legnagyobb fényforrások a természetes világítást modellező fények: két csoportba oszthatóak, a diffúz megvilágításért, az általános fényérzetért felelős égboltra (sky), illetve határozott árnyékokat, csúcsfényeket biztosító napra (sun). Előbbi többféleképpen hozható létre, alapvetően a jelentünk körül elhelyezkedő, végtelen sugarú gömb (félgömb) belső textúrájáról van szó, mely

a világítás intezitását, illetve színét határozza meg. A színértékeket vagy procedurálisan vagy egy panorámakép alapján vesszük fel. Egy épületnél realisztikus fényhatásokat jelenthet, ha a terepen készül panorámaképpel dolgozunk. Manapság a legtöbb render engine rendelkezik dedikált, saját algoritmusai szerint optimalizált megoldásokkal, sok esetben sebességbeli megfontolásokból ezeket célszerűbb használni.

Egyedi megvilágításra adnak lehetőséget a nagy dinamikatartományú (HDR, High Dynamic Range) képek. A tárolt színmélység, információmennyiségalkalmassá teszi őket fényforrásokként való alkalmazásra is: a mindennapokban használt képek általában 8 bitesek, ami azt jelenti, hogy színenként -RGB színtérben ezek a piros, zöld és kék- 256 árnyalatot vagyunk képesek megjeleníteni, ami összesen 16 777 216 színre ad lehetőséget. 32 bites HDR képek használata során a megjelenített színeken felül az egyes pixelek világítás-információit is tároljuk lebegőpontos számok formájában (exponenciális alakban tárolt számok). Így lehet azt megtenni, hogy a napkorong fényessége sokszorosa a kép többi részének, ezzel dominálva a jelenet fényviszonyait.

Egy jelenetben viszont nem csak természetes, hanem mesterséges fényforrások is gyakran jelen vannak. Ezek tulajdonságai igen változóak lehetnek, alapvetően csoportosíthatjuk pontszerű, és felületből származó fényekre. Valós időben csak pontszerű fényekkel tudunk megszokott számításokat végezni, de a pontszerű fények is kétfelé bonthatóak: szpot és egyszerű pontfényekre. Minden fénynél meg kell határozni a fény erősségét és színét. A szpotfényeknél pedig a világítás szögét, és lehetőség nyílik ezeknek a fényeknek a formájának módosítására is - ezt gyakorlatban egy textúrával határozzuk meg.

Valós időben gondolkodva osztályozásra ad még lehetőséget a lámpa interaktivitásra való képessége: attól függően, hogy előtte elmozogva kapunk-e reakciót, beszélhetünk dinamikus, illetve statikus fényforrásokról.



46-48. ábra. Példa pontszerű fényforrás (bal szélső kép), spotlámpa (középső), illetve napfény okozta megvilágításra.

3.4. Az összekapcsolás lehetőségei

3.4.1. Általános munkafolyamat

Az alábbi folyamatábra, illetve a következő gondolatmenet segítségével felvázoljuk, hogy a workflow lehetőségei milyen módon használhatóak ki a lehető legteljesebb módon. A BIM folyamat pont erre lett kitalálva, hogy a szakmák közötti átjárást segítse, esetünkben ez az építész, megrendelő párost.

Az építész már a tervezés kezdetétől bevonhatja a megrendelőt érdemi párbeszédbe, hiszen a BIM modell már a kezdeti, műszaki szempontból részletezetlen állapotában is alkalmas a prezentálásra. A megrendelőt nem feltétlenül a gépészeti rendszerek, szerkezeti részletek izgatják, de a tervezés előrehaladtával ezek is belekerülhetnek a bejárható modellbe.

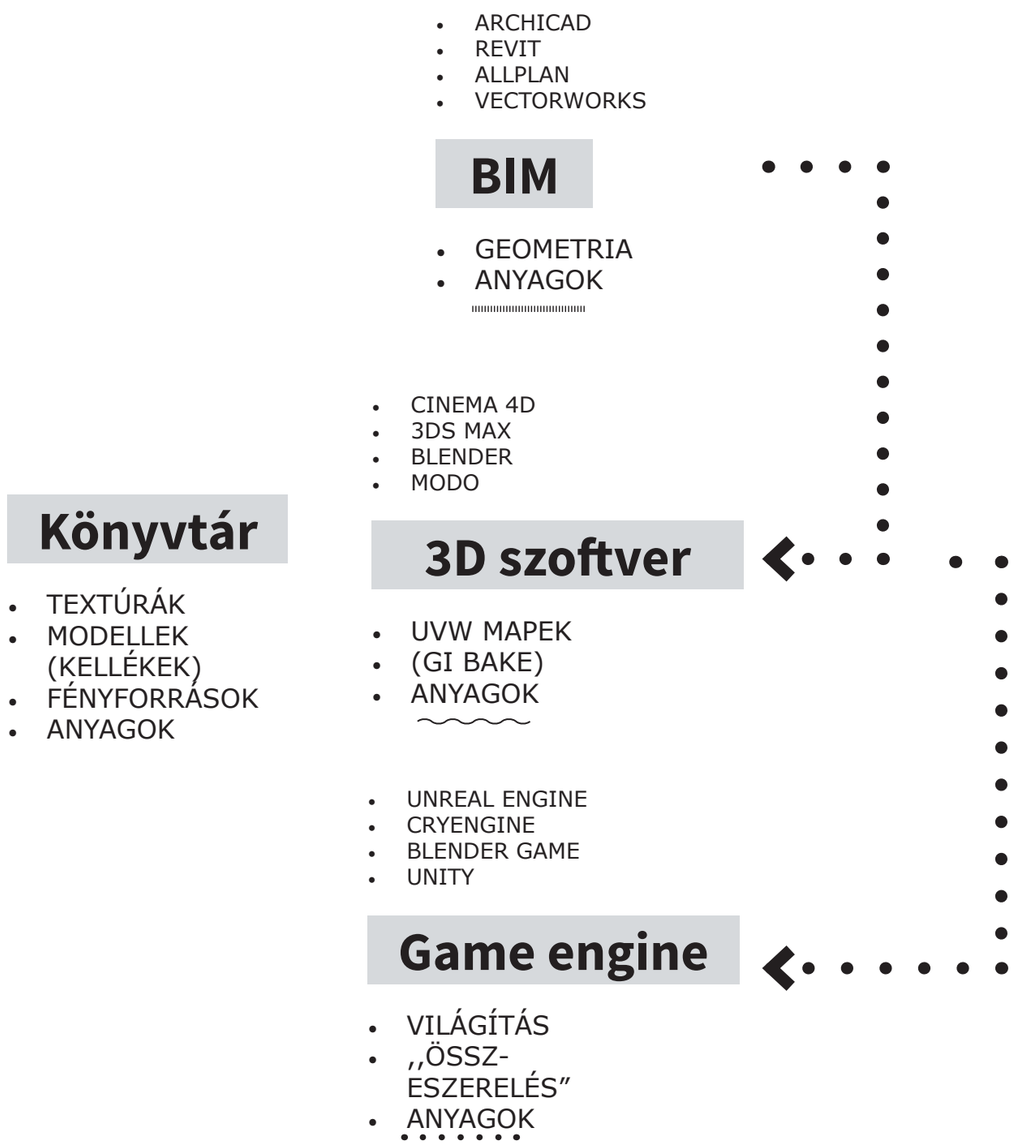
A BIM programból megfelelő kidolgozottságú (léptékű) 3D nézetből generált modell kerül bemutatásra a valós idejű renderer segítségével, mely a megrendelőhöz különálló alkalmazás formájában jut el. A tervezés során bekövetkezett változtatások a későbbiekben kevesebb munkával átvihetőek a rendszeren, ennek főbb lépéseit tekintjük át a következőekben.

3.4.2. Közbenső 3D szoftver közbeiktatása

Az BIM rendszerek, illetve játékmotorok összehasonlításakor láthattuk, hogy számos megoldandó probléma áll előttünk, így a dolgozat írásakor közvetlen kapcsolat BIM szoftver és játékmotor között tulajdonképpen nem lehetséges. A szükséges munka elvégezhető bármelyik, komolyabb modellezőszoftverben. Ha közbeiktatott 3D programot használunk, akkor célszerű úgy választani, hogy a modell gyakorlatilag interaktívan frissíthető legyen, ebből kifolyólag a nagy szoftverfejlesztő cégek-cégcsoportok termékeit érdemes lehet együtt használni (Revit - 3ds Max, ArchiCAD - Cinema 4D, stb.) egyébként plugin használata válhat szükségessé (pl. Din3D plugin ArchiCAD és 3ds Max közötti interakcióhoz).

A 3D programunkban a megkapott geometriai, anyagozási és más rendelkezésünkre információkat kell általalakítanunk, szűrnünk és feldolgoznunk ahhoz, hogy ez a game engine számára megfeleljen. A geometriát ponthálókká kell alakítanunk, bár ez legtöbbször már adott. Az anyagtulajdsonságok alapján shadereket kell választanunk, azok paramétereit beállítani. Sokszor előfordul, hogy a hozott textúra nem elég nagy felbontású, csúnya, vagy csak olyan anyagot használnánk, amihez további mapek tartoznak. Ilyenkor ezeket is cserélni kell. Az anyag a geometriához viszonyított léptékének, és a felületek síkba való kiterítésének információi sajnos sokszor rosszul vagy hiányosan érkeznek a BIM szoftverből, ezeket javítani vagy pótolni kell. A jelenetnek a fényelését szinte mindig a 3D programban állítjuk be, ugyanis itt megfelelő részletességű eszközök állnak rendelkezésre.

Ha a jelenet össze információja meg is van, sok fényhatást csak előre számolva tudunk a jelenetben bemutatni. Két eset van: a jobb esetben a game enginehez kapott segédszoftver ezt automatikusan



49. ábra. Flowchart a munkafolyamatról. Célszerű egy központi könyvtárból dolgozni, hogy az összes programban ugyanazokból az egységekből (textúrák stb.) építkezhessünk.

elvégzi helyettünk, rosszabbik esetben magunknak kézzel kell megcsinálni.

Ha ezek a lépések mind megvannak, ekkor jutunk csak el a valós idejű megjelenítéshez. Az interaktivitás beépítése egy kényes dolog. Ehhez vagy a játékmotor segédprogramjában kell összerakni a lehetőségeket és a hatásokat vagy egy meglevő programrészletet használni hozzá, vagy végső esetben le kell programozni. Mivel ez a bemutató a játékok által használt technológiák kitaposott ösvényén megy, az interaktivitás lehetőségei, kellő képzettséggel határtalanok. A mai legújabb game enginek rendelkeznek saját visual scripting rendszerrel, melyekkel minden eddiginél könnyebben készíthetünk interaktív elemeket. Gyakran egy meglevő objektum kis módosításával megfelelő elemet kapunk, pl. egy nyitható ajtó működését átmásolhatjuk a saját ajtónkéra.

Ezzel az általános módszerrel bár számos szoftver felhasználói környezete között kell váltogatni, a módszer legfőbb előnye, hogy általában programozási ismeretek nélkül is megfelelő minőségű eredmények érhetőek el.

3.4.3. Direkt hasznosítás

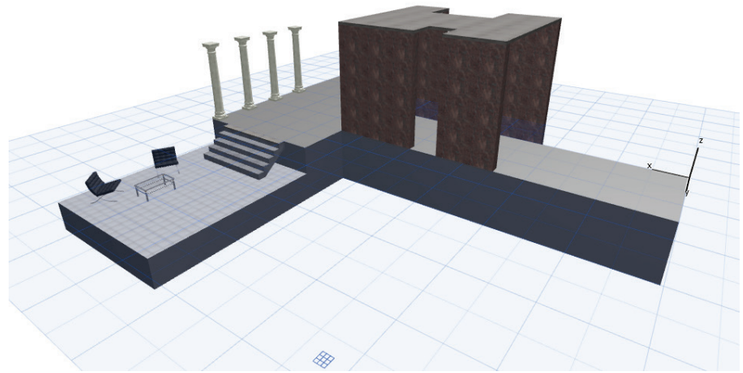
Elméletileg az egész áttérési probléma nem létezne, ha nem lenne külön véve a game engine a BIM szoftvertől. Gyakorlatilag, mikor a szerkesztő programban a 3D nézetet nézzük az is egy grafikus kártya által renderelt valós idejű grafika, tehát már biztosít a szoftver egyféle grafikus megjelenítést. A szoftver fejlesztőinek lehetőségében állna, hogy ezt a grafikai megjelenítést felfejlessze, közelebb vigye a valós látványhoz. Nyilván az épület elemeinek szerkesztése közben semmi szükség ilyen megjelenésre. Viszont egy külön programrészben, ami egy valós idejű épület bejárós bemutatót készítene elő, igenis helye lenne egy szebb grafikájú verziónak. A hangsúlyt arra szeretném helyezni, hogy mivel a program már tartalmazza a geometriák kiszámolásának és megjelenítésének lehetőségét, viszonylag kevesebb munkával lehetne egy ilyen programrészt készíteni, de sajnos erre még nem láttunk példát.

3.4.4. Script

A számítógép az automatizálásra lett kitalálva, hogy megkönnyítse az ember feladatait. Ma sok munka már a számítógépre alapszik, a dolgozat témái mind ilyenek. A népszerűbb programok készítői profitorientált cégek, amik főképp csak olyan dolgokra nyújtanak megoldást, amire tömeges igény van. Szerencsénkre felismerésre lelt, hogy az emberek még ezekkel a programokkal is sokszor

repetitív munkát végeznek, így alakultak ki a makrózási és a scriptelési lehetőségek.

Dolgozatunk fő problémája a CAD szoftverekből való áttérés game enginekbe, felismertük, hogy ez egy olyan munkafolyamat, ami mindig hasonlóan zajlik, és számos repetitív munkafázist tartalmaz. Ezért egy script készítéséhez álltam, ami Archicad modellekből segít eljutnunk egy Blender game engineben levő jelenetbe. A szoftverválasztás önkényes alapú: ezeket ismerem legjobban. Fontos, hogy bár működőképes a script, a célja mégsem egy totális megoldás kínálása, inkább egy kísérlet, amin megmutathatjuk a felmerülő problémákat, és az ezeket lefedni próbáló megoldásokat. A nagyobb szoftverek – főleg ha egy gyártóttól származnak – általában már tartalmaznak áttérési lehetőségeket, azonban fontos megjegyezni, hogy sokszor a hivatalosan kiadott programok sem működnek tökéletesen.



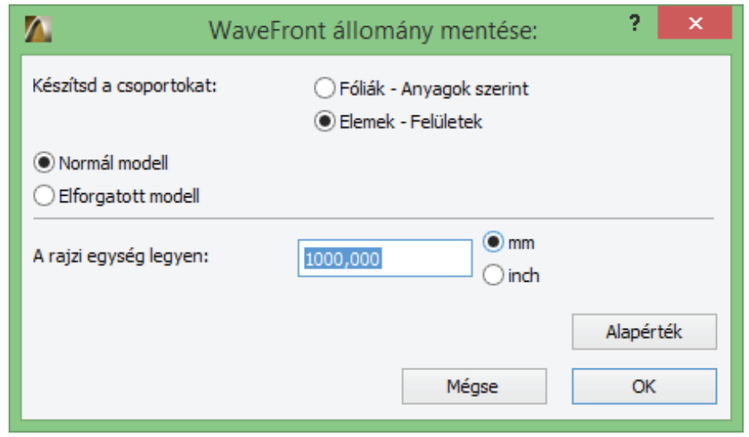
50. ábra ArchiCAD 17 jelenet. Fal, födém, lépcső és objektumok, minél több anyaggal.

3.4.4.1. Mit csinál a script?

A Blender egy igen általános program, sok célra használható, de mind a 3D modellek köré csoportosul. A része egy saját game engine is, amit viszonylag egyszerűbb tartalommal megtölteni a program többi funkciójával. A szoftvernek van egy beépülő scriptnyelve, a python, aminek a segítségével plusz funkciókat építhetünk be. Érdemes megjegyezni, hogy a Blender teljesen ingyenes és hozzáférhető program.

A megoldásom két fő részből áll. Az első egy Blender addon script, ezt beregisztrálva az addonok közé (kiegészítők), új műveletek lesznek elérhetők, sőt saját kezelőfelületet is nyújt. A második fő része egy kiinduló fájl, ahol számos eleme előre be van állítva a jelenetnek, mint pl. a kamera irányíthatósága.

Végeredményben egy valós időben bejárható jelenetet kapunk, amiben a fények előre vannak számolva, és felületek phong shadinggel vannak árnyalva. Az irányítás a megszokottnak tekinthető WSAD + egér, ctrl és space pedig a süllyedés, emelkedés.



52. ábra. Wavefront exportáló párbeszédpanel.

3.4.4.2. Hogyan használjuk?

A képen látható az ArchiCAD kiindulási fájl.

Az áttérés az ArchiCAD-ben kezdődik. Az ArchiCAD 3D-s nézetéből exportáljuk a megjelenő tartalmat, így érdemes még ott beállítani a felületek anyagát, hiszen az fog átkerülni. Tehát 3D-s nézetben a Mentés másként opcióval elmentjük Wavefront OBJ (.obj) formátumba, egységnek a métert vesszük, a csoportok készítését „Elemek- Felületek” opcióra tesszük. Így a program generál egy .obj modellfájlt és egy .mtl anyaginformációs fájlt.

Ezután a Blender következik. Megjegyzendő, hogy a működés a 2.72-es verziót megköveteli. Első sorban regisztrálnunk és aktiválnunk kell az addont. Ez a File>User preferences ablakból érhető el, ott is az Addons fülön. Az ablak alján találunk egy „Install from File...” feliratú gombot, erre kattintsunk, és keressük meg a mellékelt addon.py fájlt, és válasszuk ki. Az ablakban ezután meg jelenik szürkén a „System: Archicad to game engine” sor, a mellette levő dobozt pipáljuk ki, így aktiválva. Utolsó lépésként az ablak alján levő „Save User Settings” gombbal mentjük a beállításokat, ez után bezárhatjuk az ablakot.

Következő lépésben nyissuk meg a mellékelt kiindulási fájlt (starter.blend), a File>Open menüponttal. A fájlban át vannak a panelek rendezve, így az elrendezés egyezni fog a képeken láthatóval.

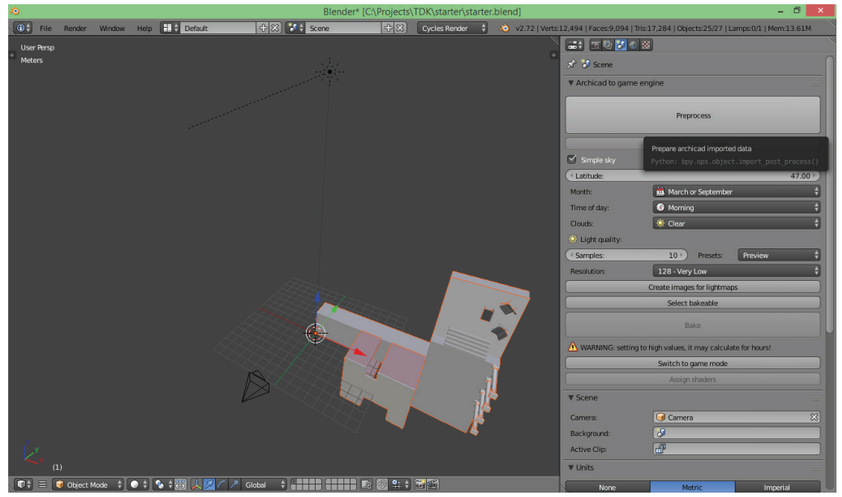
Az elkövetkező lépés a modell importálása lesz ezt a File>Import>Wavefront (.obj) menüponton érhetjük el, mint ahogy a képen is látható.

Ezt követően kezdjük használni a jobb oldalon látható „Archicad to game engine” addon panelját.

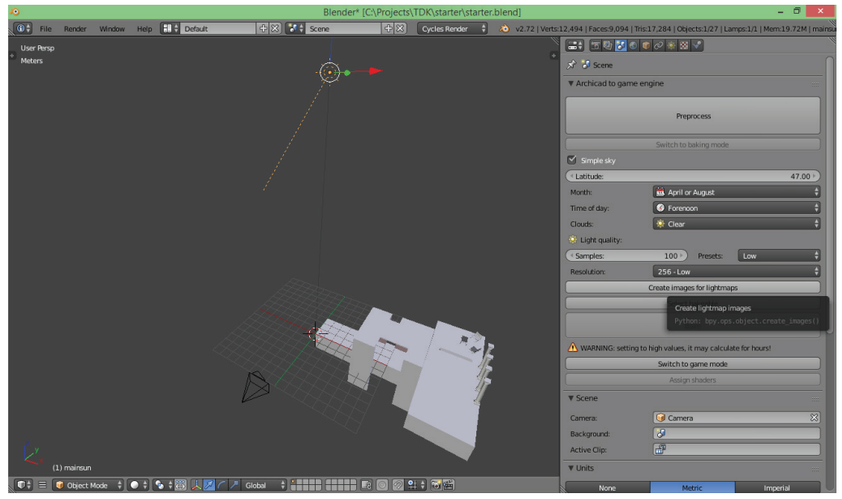
Első lépésben a preprocess gombot nyomjuk meg, ilyenkor több dolog is történik. Először is át forgatja megfelelő koordináta rendszerbe az importált geometriát. Az beépített importáló már feldolgozta az anyagokat, de nekünk ezt tovább kell formálnunk. Minden objektum különböző lightmap textúrát kap, emiatt, minden objektumhoz külön anyagot kell társítani. Tehát azokat az anyagokat, amiket több objektum is használ, le kell másolni, hogy mindhez külön legyen. A harmadik lépés, amit csinál, az az anyagok konvertálása, ugyanis mi a Cycles render motort akarjuk használni, a beépített importáló a régebbi Blender internal rendererhez készíti az anyagot. Végül minden objektumhoz készítünk egy külön UV réteget a lightmap számára, amire a „smart UV mapping” algoritmussal kiterítjük a geometriát.

KÉP – tutor3.png: Fények beállítása. A jobb oldali menüsáv az addon amit telepítettünk.

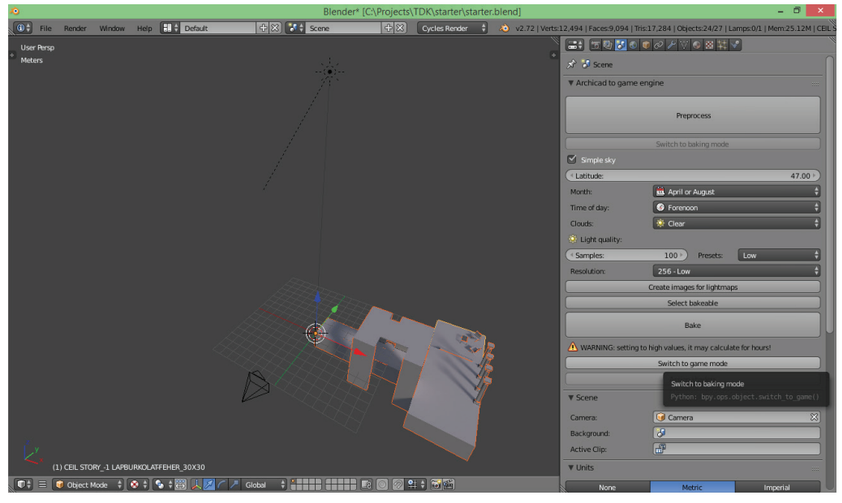
A következő fontos lépés a jelent fény beállítása. Erre egy fokkal nagyobb hangsúlyt fektettem, így



53. ábra. Blender feldolgozatlan adatokkal. Láthatjuk, hogy a két program más koordináta rendszert használ.



54. ábraFények beállítása. A jobb oldali menüsáv az addon amit telepítettünk.



55. ábra. Bake-lés elötti állapot. Minden készen áll, hogy elkezdje a számításokat a gép.

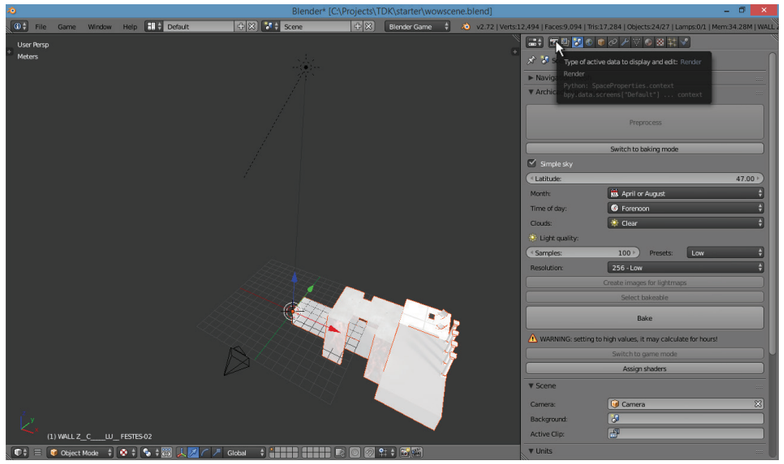
beállíthatjuk a földrajzi szélességet, a hónapot, a nap közbeni időt, és a felhőzöttség mértékét. Ez alapján a script a nap állását, színét és erősségét illetve az ég átmenetes színét és erősségét állítja be, hogy a bakelt fények minél szebbek legyenek. A színeket Kelvin fokban, színhőmérséklet formájában tápláltam be, majd konvertáltam RGB-re.

Nagy hiányosság az, hogy az exportálás-importálás során az Archicadben levő fényforrások elvesznek, így azokat manuálisan lehet pótolni.

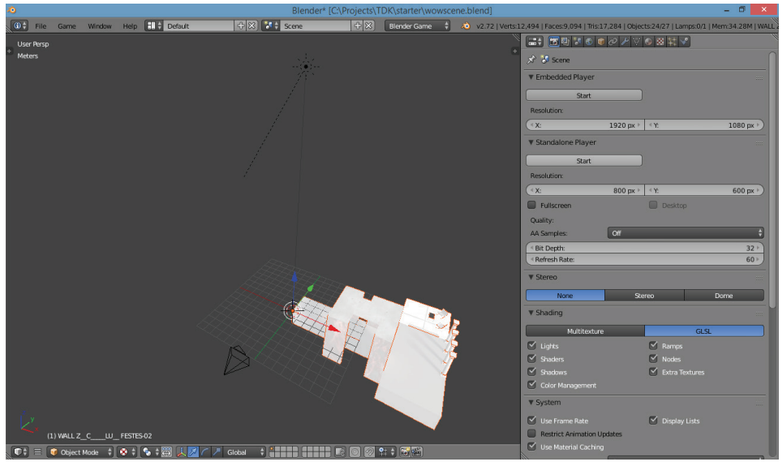
A jeleneten elegendő beállítást végeztünk ahhoz, hogy a lightmapek kalkulációjához fogjunk. Be kell állítani tehát a textúrák felbontását és a bakelés minőségét. A számok mellé tettem preseteket, hogy tudjuk, mi számít magas illetve alacsony minőségnek, de itt is leírom, hogy a jó minőségű fények számolása akár órákat is igénybe vehet.

Ha beállítottuk a dolgokat, akkor a Create images for lightmaps gombra kattintunk, ekkor létrehozza a képeket a program, és hozzá csatolja az anyagokhoz, ez után a Select bakeable gomb segítségébel kijelöljük a megfelelő objektumokat (pl. az üveg felületekre nem számol, így azt nem jelöli ki). Végül a Bake gombot aktiváljuk, és megvárjuk, míg elkészülnek a textúrák. A folyamat jelző a program tetején található.

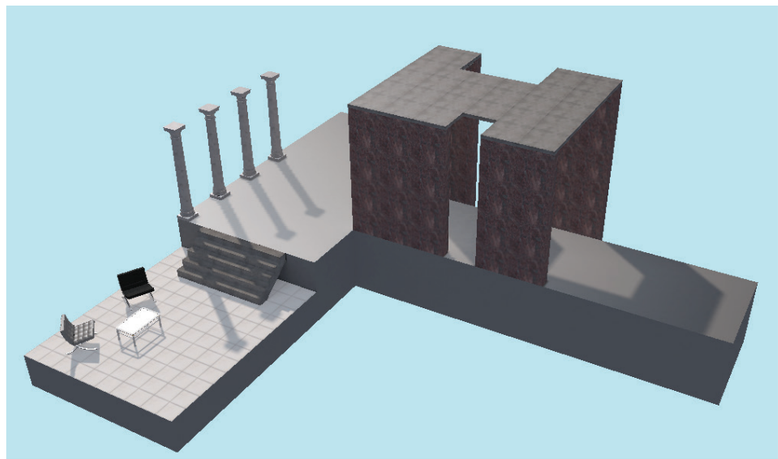
A texture baking után a Switch to game modera bökünk, ilyenkor elmenti a képeket a program, és átvált Blender game enginere. Végül az Assign shaders gomb a valós idejű shadereket hozzárendeli a modellekhez, a textúrákkal együtt, így készen áll a jelenet.



56. ábra. Panel átváltása renderre. Ezt követően a Standalone Player részen beállíthatjuk a felbontást, és egyéb renderelési beállításokat.



57. ábra. Játék render beállításai. Beállíthatunk akár multisamplinget is.



58. ábra. Valós idejű renderkép. Láthatóak az árnyékok és a csillanások is, miközben szabadon mozgunk.

Hogy elindítsuk a játékot, még egy két beállítást kell megejteni. Váltsuk át a jobb oldali menüs részt render módba a tetején levő kis fényképező ikonnal

A Start gombbal pedig elindíthatjuk, és láthatjuk mit készítettünk el.

KÉP – gameengine.png: Valós idejű renderkép. Láthatóak az árnyékok és a csillanások is, miközben szabadon mozgunk.

4. Alkalmazási lehetőségek

A bemutatott technológiák építészeti alkalmazásáról már esett szó a [2.3] pont alatt, az alábbiakban összefoglaljuk őket, illetve további alkalmazási lehetőségeket vetünk fel.

4.1. Mindennapi használat

Tisztában vagyunk vele, hogy a jelenleg a szakmabeliek nagy része 3D modellezéshez kapcsolódó ismeretekkel nem, vagy felületesen rendelkezik, ezért a technikák implementálása nehézkes, a majdnem automatizált felhasználás pedig egyenesen utopisztikus elgondolás. Mindenesetre nem tartjuk kizártnak, hogy éveken belül változni fog a helyzet, hiszen a vizuális minőség-befektetett munka arányában elérkezett egy olyan fordulópont, ami gyors terjedést prognosztizál. A dolgozat írásának kezdetekor a fórumokon elvétve lehetett találni egy-egy építészeti vizualizációt, most –a vége felé közeledve- pedig nem túl nehéz újabb lenyűgöző példákba botlani.

A modellben való virtuális séta során jobban körvonalazódhatnak a megrendelő igényei, és egyszerűbben elmagyarázhatók a változások hatásai, így gyorsabban juthat konszenzusra a megrendelő és a tervező. Egy BIM modell segíthet a megrendelő igényeinek leginkább megfelelő koncepció kialakításában.

A végtermék egy program, amit a megrendelőnek át lehet nyújtani, otthon minden részletében át tudják gondolni saját számítógépen futtatva a leendő otthonukat (legyen szó bármilyen építményről), ezáltal sokkal közvetlenebb, konstruktívabb párbeszéd alakítható ki bárkivel. Ez az elgondolás a legnagyobb érték, ami írásra késztetett minket.

A modellben való virtuális séta során a megrendelő pontosabb képet kaphat az igényeiről, és egyszerűbben követni tudja a változtatásokat illetve azok hatásait, ami végső soron a kommunikáció felgyorsulását, így idő- és anyagi megtakarításokat eredményezhet.

59-60. ábra. A mindennapos használatra néhány példa. Benapozás vizsgálatát, felületek, anyagok tesztelését egyaránt elvégezhetjük.

Mivel a játékmotorok tetszőlegesen programozhatóak –a saját visual scripting rendszerük is teljes körű lehetőséget biztosít rá, ezért bármilyen változtatás interaktívan elérhetővé tehető. Számos példa létezik dinamikus anyagváltások megvalósítására, amely különféle festési-anyaghasználati kérdésekben jelenthet döntési támpontot. Ugyanezen elv alapján nem csupán dinamikus anyag példányok készíthetőek, de tárgyakat is cserélhetünk, ami egyazon terv számos berendezési variációjának egyben, interaktívan történő bemutását teszi lehetővé.

Mindezt saját grafikus felülettel megtoldva prezentálhatjuk, mely például lehetővé teszi a fenti variációk listába gyűjtését, majd exportálását: a legjobban tetsző színeket-anyagokat-berendezéseket saját maguk között alaposan megvitatva, majd az adott elrendezést exportálva (akár egy szövegfájlként) a tervezés gyorsítható, könnyebbé tehető.

A fenti példákon felül gyakorlatilag bármi, ami eszünkbe jut, programozható. A scriptnyelvek enginenként elég változóak, így egy általános programozási tudás szükséges a használtaukhoz, de ha van beépített vizuális scripting rendszer, ezek segítségével minimális programozási ismeretekkel lehet a fentieket elkészíteni.

61. ábra. Az előzőeken túl dinamikusan, valós időben változtathatjuk a felületeken használt anyagokat.

4.2. Akadémiai felhasználás

Nem csak a versenyszféra eszköztárába kerülhet be a technológia, hanem oktatási környezetbe is. Itt sokkal valószínűbbnek tartjuk az elterjedést, lévén a ráfordítható humánerőforrás nagyobb jelenléte miatt, illetve a relatíve friss technológiát a jelenlegi egyetemista korosztályon, és a fiatalabbnak tekinthető építészeken kívül az idősebb korosztály nehezen, kis hajlandósággal tanulja meg, sajnos erős kisebbségben érezzük a téma iránt nyitott, azt értékelni képes réteget az egyetemi oktatók között is.

Építészettörténeti, urbanisztikai vonatkozásai lehetnek a legérdekesebbek, kivitelezésben legfeljebb mankóként tudná segíteni a meglévő szoftvereket, melyek kifejezetten a műszaki tartalom 3D prezentációjára készültek (BIMx, Navisworks), tekintve a látvány-centrikusságot.

Régi épületek, illetve városok feltárásának-felméréseinek építészek által megalkotott, szakmai alapokon nyugvó, interaktívan szemlélhető dokumentációja viszont hasznos segédanyag lehet nem csak a szakmához kapcsolódó területet hallgató egyetemisták, de történelmet tanulni kezdő kisiskolások, illetve középiskolások számára is.

4.3. Virtuális valóság

A számítógépes játékok között manapság kezd újra megjelenni a virtual reality (VR) technológiája. Ezek a virtuális számítógépes világok nagyobb fokú átélését próbálják meg elérni, sztereó kép renderelésével, közvetlenül a szem elé téve, manapság egyre nagyobb teret nyerő név az Oculus Rift. Ezek az eszközök közelebb hozzák az emberekhez a virtualitást, így a kommunikáció egy fokkal könnyebb a megrendelő és a tervező között.

Érdekes lehet eljátszani a gondolattal, hogy milyen alkalmazási lehetőségei nyílnak meg, ha összekapcsoljuk a technológiát a 3D szkenneléssel. A szkennelés során meglévő épületekről készíthetünk nagy sűrűségű pontfelhőket, melyeket háromszögelve, letisztítva poligonhálóhoz juthatunk, amit az ismertetett technikákkal fel tudunk dolgozni. Épületek felmérésére napjainkban már drónokat is használnak, amik nagy beépítettségű területeken nélkülözhetetlenek, hiszen nem feltétlenül biztosított a hely a felmérőcsapat- és felmérő eszközök számára. A kisméretű, távolról irányítható eszközök rengeteg képet készítenek a felmérendő objektumról (ami akár vár-méretű is lehet, elég megnézni a Pix4D csapatának chilloni kastély felméréséhez kapcsolódó projektjét), a fotókból ezek után fotogrammetriai elvek mentén meghatározzák kellően sok felületi pont (kellően sok: a kastély esetén 95 000 000 pontból álló pontfelhőt generáltak) térbeli helyzetét, az ebből álló pontfelhő poligonhálóvá alakítása után a készített fotók alapján textúrázható, a mellékelt képek szerint tetszetős eredményeket adva. Az elkészített modell akár egy tervezési helyszín is lehet, ami az épületünk valós idejű bejárásához remek hátteret szolgáltatna. Gyakorlatilag a fotóba illesztett látványtervek körbesétálható, 3D megoldásáról van szó. Mindezt az említett virtual reality technológiákkal kombinálva fizikai mozgással bejárható térben prezentálhatunk építészeti gondolatokat fotorealisztikus módon.

A Zeiss cinemizer® szemüveg is egy példa rá, hogy a nagy cégek is látnak fantáziát a virtuális valóság és építészeti látványtervezés egy tető alá hozásában [BIMx][ZEISS].

62. ábra. A chilloni kastély drónok által készített fotók alapján elkészített 3D modellje.

5. TOVÁBBI KUTATÁSI LEHETŐSÉGEK

Dolgozatunk jelenlegi formájában áttekintésre használható, azokat a vezérelveket tartalmazza, amelyek mentén a munkát folytattuk. Nem szól részletesen az egyes munkafolyamatok lépéseiről, az egyes lépések mögött álló döntéseink okáról, részben idő-, részben terjedelembeli korlátokból fakadóan, illetve célunk sem egy részletes oktatókönyv írása volt, oktatóvideók milliói találhatóak az interneten, az egyes szoftverek kézikönyvei (szintén online dokumentálva), illetve beépített súgói mind elég támpontot adnak a témában való első lépésekhez.

A dolgozatot lehetne folytatni az egyes interakciót biztosító eszközök leírásával: mit kell tenni, hogy a padlón az E gomb megnyomására megváltozzon az anyag? Érdekesebb lehet egy konkrét példa feldolgozása, ahol ezek a kérdések konstruktívan, a feldolgozást elősegítendő merülnek fel, nem pedig egy programozási példatárra emlékeztetően következnek egymás után. Egy bonyolultabb formákkal bíró, építészettörténeti vonatkozásaiban fontos, nem többszörösen dokumentált épület a célra kiváló lehet. Egy ilyen példán keresztül mind BIM-beli, mind pedig e dolgozat vizsgálta vonatkozásaiban mélyebben dokumentálható lehet a témakör, részletezve modellezési, modell-menedzselési kérdéseket is.

Elképzelhető a az elkészített alkalmazás pontosítása, további eszközökkel történő bővítése is.

6. ÖSSZEGZÉS

A dolgozat főbb tanulságai a következőek: a bemutatott technika kétségkívül hasznos, bár a hasznosság kétségkívül a befektetett idő/energia függvénye. Ezt elsősorban a személyes, 3D modellezés, látványtervezés terén szerzett képességek befolyásolják. Mivel először csináltunk ilyet, ezért sok volt az első alkalmakra jellemző, később már ritkán elkövetett hiba: a megfelelő exportálási-importálási beállítások megtalálása, a megfelelő munkasorrend megtalálása nem ment könnyen. Nem voltak kész, felhasználható anyagaink, így ezek elkészítése is időbe telt. Ezek mérlegelésével alapvetően kijelentetjük, hogy aki érez magában érdeklődést, illetve némi előismerettel rendelkezik fotorealizmusra törekvő látványtervek készítését illetően, annak érdemes belevetnie magát a valós idejű látványtervek világába. Az elején, a terv összerakásakor elvesztegetettnek hitt idő a hagyományos állóképek renderelésének fázisában már azzal a borzasztó előnnyel bírt, hogy bármilyen kameraállást „renderelni” egyet jelentett a képernyőkép készítése gombra kattintással. Nem kellett órákat, vagy akár napokat várni a kép elkészítésére.

Grafikai szempontból a végeredmény véleményünk szerint a meglévő valós idejű megoldásokhoz képest magasabb minőséget képvisel, de a kapott dinamizmus, térbeli mozgás szabadsága az, ami a legfontosabb szempontként a legnagyobb érv a technológia implementálása mellett. A szabad mozgás, a dinamikusan változtatható modell az, ami újszerűségével, közérthetőségével hatni tud. A technológia gyakorlatilag a szemünk előtt fejlődik, folyamatosan egyszerűsítve a számítógép segítségével dolgozó építészek életét. Í

7. SCRIPT FORRÁSKÓD

```
###################
# addon.py
# Blender addon script
# Konvertáló műveletek az Archicadből exportál .obj modellből
# a blender game engineben való megjelenítésre,
# és lightmap kalkulációra

bl_info = {
    „name”: „Archicad to game engine”,
    „author”: „Adam Háda <hada.adam@gmail.com>”,
    „version”: (0, 7),
    „blender”: (2, 72, 0),
    „location”: „Properties > Scene”,
    „description”: „Converting operations from archicad exported .obj, to display in game engine”,
    „warning”: „”,
    „category”: „System”,
}

import bpy
import mathutils
import math
from bpy.props import *
import os


def clamp(x,min,max):
    if x<min:
        return min
    if x>max:
        return max
    return x

# színhőmérséklet konvertálása RGB színtérbe
# from gist.github.com/paulkaplan/5184275
def colorTemperatureToRGB(kelvin):
    temp = kelvin/100
    red = 0
    green = 0
    blue = 0

    if temp<=66:
        red = 255
        green = temp
        green = 99.4708025861 * math.log(green) - 161.1195681661

        if temp<=19:
            blue = 0
        else:
            blue = temp-10;
            blue = 138.5177312231 * math.log(blue) - 305.0447927307

    else:
        red = temp - 60
        red = 329.698727446 * math.pow(red, -0.1332047592)

        green = temp - 60
        green = 288.1221695283 * math.pow(green, -0.0755148492 )

        blue = 255

 return(clamp(red,0,255)/255,clamp(green,0,255)/255,clamp(blue,0,255)/255,1)

# A napfény amit a kezelőfelületen állítunk,
# itt dolgozza fel az adatokat, és állítja be a napfény színét, irányát, intenzitását
# illetve az égbolt átmenetének színeit, irányát és intenzitását
def updateSimpleSky(self, context):
    sun = None
try:
    sun = bpy.data.objects[‚mainsun’]
    bpy.ops.object.select_all(action=’DESELECT’)
    sun.select = True
    bpy.ops.object.delete()
except:
    print („No sun”)


bpy.ops.object.lamp_add(type=’SUN’, location=(0,0,25))
sun = bpy.context.active_object
sun.name=’mainsun’

bpy.context.scene.objects.active = sun

# nap irányának számítása
# a szélességi fok, naptári és órai időpont alapján
zenit = self.Latitude

if self.Month == „December”:
    zenit+= 23.43
if self.Month == „Januar”:
    zenit+= 23.43*2/3
if self.Month == „February”:
    zenit+= 23.43*1/3
if self.Month == „April”:
    zenit-= 23.43*1/3
if self.Month == „May”:
    zenit-= 23.43*2/3
if self.Month == „June”:
    zenit-= 23.43

bpy.ops.transform.rotate(value=(zenit/180*3.1415), axis=(1.0, 0.0, 0.0))

# presetek alapján a színek, intenzitások kiválasztása
    angle = 0
    suntemp = 5000
    fronttemp = 25000
    backtemp = 10000
    backint = 1
    frontint = 0.65
    backint = 0.55
    sunint = 1

    if self.SkyTime == „Dawn”:
        angle = 100
        backtemp = 27000
        fronttemp = 5000
        sunint = 0
        frontint = 0.85
        backint = 0.3

if self.SkyTime == „Morning”:
    angle = 70
    suntemp = 3000
    backtemp = 20000
    fronttemp = 7500
    frontint = 0.68
    backint = 0.49

if self.SkyTime == „Forenoon”:
    angle = 40
    suntemp = 4500
    backtemp = 15000
    fronttemp = 20000
    frontint = 0.65
    backint = 0.5

if self.SkyTime == „Afternoon”:
    angle = -40
    suntemp = 4500
    backtemp = 15000
    fronttemp = 20000
    frontint = 0.65
```

```
        backint = 0.5

    if self.SkyTime == „Sunset”:
        angle = -70
        suntemp = 2000
        backtemp = 25000
        fronttemp = 3000
        frontint = 0.75
        backint = 0.24

    if self.SkyTime == „Evening”:
        angle = -100
        backtemp = 27000
        fronttemp = 3000
        frontint = 0.54
        backint = 0.33

    if self.SkyTime == „Night”:
        angle = 180
        sunint = 0
        frontint = 0.1
        backint = 0.1
        backtemp = 27000
        fronttemp = 27000

# szögek beállítása a napra és az ég anyagára vonatkozóan
    bpy.ops.transform.rotate(value=(angle/180*math.pi), axis=(0.0, math.cos(zenit/180*math.pi), math.sin(zenit/180*math.pi)))

    self.world.node_tree.nodes[„Mapping”].rotation[1] = (90 - zenit)/180*math.pi
    self.world.node_tree.nodes[„Mapping”].rotation[2] = (90 - angle)/180*math.pi

# a kiszámított értékek módosítása a felhőzöttség mértékétől függően
# az ég két végének színének részleges átlagolása
    if self.Clouds == „Partly cloudy”:
        sunint = 0.8
        frontint *= 0.95
        backint *= 1
        t1 = fronttemp
        t2 = backtemp
        fronttemp = 0.9*t1 + 0.1*t2
        backtemp = 0.9*t2 + 0.1*t1

    if self.Clouds == „Cloudy”:
        sunint = 0.25
        frontint *= 0.8
        backint *= 0.9
        t1 = fronttemp
        t2 = backtemp
        fronttemp = 0.8*t1 + 0.2*t2
        backtemp = 0.8*t2 + 0.2*t1

    if self.Clouds == „Stormy”:
        sunint = 0
        frontint *= 0.7
        backint *= 0.7
        t1 = fronttemp
        t2 = backtemp
        fronttemp = 0.6*t1 + 0.4*t2
        backtemp = 0.6*t2 + 0.4*t1


    sun.data.node_tree.nodes[,Emission’].inputs[0].default_value = colorTemperatureToRGB(suntemp)
    sun.data.node_tree.nodes[,Emission’].inputs[1].default_value = sunint*1.5


    self.world.node_tree.nodes[„ColorRamp”].color_ramp.elements[0].color = colorTemperatureToRGB(fronttemp)
    self.world.node_tree.nodes[„ColorRamp”].color_ramp.elements[1].color = colorTemperatureToRGB(backtemp)
    self.world.node_tree.nodes[„ColorRampInt”].color_ramp.elements[0].color = (frontint,frontint,frontint,1)
    self.world.node_tree.nodes[„ColorRampInt”].color_ramp.elements[1].color = (backint,backint,backint,1)


# az technika ki-be kapcsolása
def toggleSimpleSky(self, context):
    if self.SimpleSky:
        updateSimpleSky(self,context)
    else:
        try:
            sun = bpy.data.objects[,mainsun’]
            bpy.ops.object.select_all(action=’DESELECT’)
            sun.select = True
            bpy.ops.object.delete()
        except:
            print(„no sun”)


# UI elem: ég ki-be kapcsolása
bpy.types.Scene.SimpleSky = BoolProperty(
    name = „Simple sky”,
    description = „Use only a sun as the main lightsource of the scene”,
    update=toggleSimpleSky)

# UI elem: nap közbeni időpont választása
bpy.types.Scene.SkyTime = EnumProperty(
    items = [(,Dawn’, ,Dawn’, ,’),
             (,Morning’, ,Morning’, ,’),
             (,Forenoon’, ,Forenoon’, ,’),
             (,Noon’, ,Noon’, ,’),
             (,Afternoon’, ,Afternoon’, ,’),
             (,Sunset’, ,Sunset’, ,’),
             (,Evening’, ,Evening’, ,’),
             (,Night’, ,Night’, ,’)],
    name = „Time of day”,
    update=updateSimpleSky)

# UI elem: hónap kiválasztó menü
bpy.types.Scene.Month = EnumProperty(
    items = [(,December’, ,December’, ,’),
             (,Januar’, ,Januar or November’, ,’),
             (,February’, ,February of October’, ,’),
             (,March’, ,March or September’, ,’),
             (,April’, ,April or August’, ,’),
             (,May’, ,May of July’, ,’),
             (,June’, ,June’, ,’)],
    name = „Month”,

    update=updateSimpleSky)

# UI elem: felhőzöttség kiválasztó elem
bpy.types.Scene.Clouds = EnumProperty(
    items = [(,Clear’, ,Clear’, ,’),
             (,Partly cloudy’, ,Partly cloudy’, ,’),
             (,Cloudy’, ,Cloudy’, ,’),
             (,Stormy’, ,Stormy’, ,’)],
    name = „Clouds”,
    update=updateSimpleSky)

# UI elem: szélességi fok beállító menü
bpy.types.Scene.Latitude = FloatProperty(
    name = „Latitude”,
    description = „Circle of latitude”,
    default = 47,
    min = -90,
    max = 90,
    update=updateSimpleSky)

# sample frissítő függvény
def samplesetter(self, context):
    self.cycles.samples = int(self.QualityPreset)

# a preprocess művelet
class PreProcess(bpy.types.Operator):
    „””Prepare archicad imported data”””
    bl_idname = „object.import_post_process”
    bl_label = „Preprocess”

    @classmethod
    def poll(cls, context):
        return context.scene.render.engine == ,CYCLES’

    def execute(self, context):
```

```
        bpy.ops.object.select_all(action='DESELECT')

        for obj in bpy.data.objects:
            if obj.type == ,MESH':
                obj.select = True
        # minden objektum elforgatása -90 fokkal X tengelyen
        bpy.ops.transform.rotate(value=-1.5708, axis=(1, 0, 0))

            # anyag másolatok készítése az össze objektumhoz, a másolatok
hozzákacsolása
        for obj in bpy.data.objects:
            if obj.type == ,MESH':
                if obj.material_slots[0].material.users>1:
                    context.scene.objects.active = obj
                    basemat = obj.material_slots[0].material
                    newmat = bpy.data.materials.new(basemat.name)

                    obj.active_material = newmat
                    obj.active_material_index = 0
                    bpy.ops.object.material_slot_assign()



                    i = 0
                    for tex in basemat.texture_slots:
                        if tex != None and tex.texture != None:
                            newmat.active_texture_index = i
                                newtex = bpy.data.textures.new(tex.name,tex.
texture.type)
                            newtex.image = tex.texture.image

                            mtex = newmat.texture_slots.add()
                            mtex.texture_coords = ,UV'
                            mtex.texture = newtex

                        i+=1


        #az összes anyag cycles renderer számára konvertálása

        for mat in bpy.data.materials:

                print(„mat: „+mat.name)

                tree = mat.node_tree
                mat.use_nodes = True

                # cycles nodefa készítése a bakinghez (ha nincs)
                if not type(tree) is bpy.types.ShaderNodeTree:

                    mat.use_nodes = True

                tree = mat.node_tree
                links = tree.links

                tree.nodes.clear()

                # üveg anyag, ez az átlátszó és a csillogó BRDF keveréke
                if (not mat.name.find(,UVEG')):
                    n1 = tree.nodes.new(,ShaderNodeOutputMaterial')
                    n2 = tree.nodes.new(,ShaderNodeMixShader')
                    n3 = tree.nodes.new(,ShaderNodeBsdfTransparent')
                    n4 = tree.nodes.new(,ShaderNodeBsdfGlossy')

                    links.new(n3.outputs[0],n2.inputs[1])
                    links.new(n4.outputs[0],n2.inputs[2])
                    links.new(n2.outputs[0],n1.inputs[0])

                    n2.inputs[0].default_value = 0.05
                    n4.inputs[1].default_value = 0

                    n1.location = 200,0
                    n3.location = -200,0
                    n4.location = -200,-100
                    print („glass")

                #nem átlátszó anyag - teljesen diffúz
                else:
                    n1 = tree.nodes.new(,ShaderNodeBsdfDiffuse')
                    n2 = tree.nodes.new(,ShaderNodeTexImage')
                    n3 = tree.nodes.new(,ShaderNodeOutputMaterial')
                    links.new(n1.outputs[0],n3.inputs[0])
                    n2.location = -200,0
                    n3.location = 200,0
                    print („opaque")

        # automatikus smart UV kiterítés
        for obj in bpy.data.objects:
            if obj.type == ,MESH':
                print(„mesh: „+obj.name)
                bpy.ops.object.select_all(action='DESELECT')
                obj.select = True
                context.scene.objects.active = obj

                lmuv = obj.data.uv_textures.new()

                lmuv.name = ,lmuv'
                lmuv.active = True
                bpy.ops.uv.smart_project(island_margin=0.08)

                mat = obj.material_slots[0].material

                return {,FINISHED'}


        class UseShaders(bpy.types.Operator):
            „""Converting materials and assinging shaders to objects by material"""
            bl_idname = „object.assign_shaders"
            bl_label = „Assign shaders"

            @classmethod
            def poll(cls, context):
                return context.scene.render.engine == ,BLENDER_GAME'

            def execute(self, context):
                for obj in bpy.data.objects:
                    if obj.type == ,MESH':
                        context.scene.objects.active = obj

                        bpy.ops.logic.sensor_add(type='ALWAYS')
                        bpy.ops.logic.controller_add(type='PYTHON')



                        mat = obj.material_slots[0].material# shader hozzárendelés,
3 féle shader
                # textúrázott phong - általános
                # textúrázatlan phong - textúra nélküli felületek
                # üveg - az 'UVEG' stringet tartalmazó nevű materialok számára
                        if not mat.name.find(,UVEG'):
                            obj.game.controllers[0].text = bpy.data.texts[„glassshader.
py"]
                        elif not type(mat.texture_slots[0]) is bpy.types.
MaterialTextureSlot:
                            obj.game.controllers[0].text = bpy.data.texts[„solidshader.
py"]
                        else:
                            obj.game.controllers[0].text = bpy.data.texts[„shader.py"]

                        obj.game.sensors[0].link(obj.game.controllers[0])

                        mat.use_nodes = False
                for mat in bpy.data.materials:
                    print(„mat:"+mat.name)
                    mat.use_nodes = False
                # textúrák shaderekhez csatolása
                # a megfelelő  UV csatorna használatával
                        if  mat.name.find(,UVEG'):
                            if type(mat.texture_slots[1]) is bpy.types.MaterialTextureSlot:
                                if type(mat.texture_slots[0]) is bpy.types.
MaterialTextureSlot:
                                    mat.texture_slots[0].uv_layer = ,UVMap'

                                newtex = bpy.data.textures.new(„lm"+mat.name,"IMAGE")
                                newtex.image = mat.node_tree.nodes[„Image Texture"].
image
                                mat.texture_slots[1].texture = newtex
```

```
                    mat.texture_slots[1].uv_layer = ‚lmuv’
                    print(„multi”)
                elif not type(mat.texture_slots[0]) is bpy.types.
MaterialTextureSlot:
                    newtex = bpy.data.textures.new(„lm”+mat.name,”IMAGE”)
                newtex.image = mat.node_tree.nodes[„Image Texture”].image
                    mat.texture_slots.create(0)
                    mat.texture_slots[0].texture = newtex
                    mat.texture_slots[0].uv_layer = ‚lmuv’
                    print(„single”)
        return {‚FINISHED’}

# cycles módba váltás, és az anyagok node módjának aktiválása
class SwitchToBaking(bpy.types.Operator):
    „””Switch to baking mode”””
    bl_idname = „object.switch_to_baking”
    bl_label = „Switch to baking mode”

    @classmethod
    def poll(cls, context):
        return context.scene.render.engine == ‚BLENDER_GAME’

    def execute(self, context):
        for mat in bpy.data.materials:
            mat.use_nodes = True
        context.scene.render.engine = ‚CYCLES’
        return {‚FINISHED’}

# Game modeba váltás, az anyagok node módjának tiltása,
# és a renderelt lightmapek lementése
class SwitchToGame(bpy.types.Operator):
    „””Switch to baking mode”””
    bl_idname = „object.switch_to_game”
    bl_label = „Switch to game mode”

    @classmethod
    def poll(cls, context):
        return context.scene.render.engine == ‚CYCLES’

    def execute(self, context):
        for mat in bpy.data.materials:
            mat.use_nodes = False
        for image in bpy.data.images:
            if image.is_dirty:
                image.save()
        context.scene.render.engine = ‚BLENDER_GAME’
        return {‚FINISHED’}

# a lightmapekhez kapcsolódó textúrák készítése, és lementése fájlokba
# képméret választás a menüben kiválasztott méret szerint
class CreateImages(bpy.types.Operator):
    „””Create lightmap images”””
    bl_idname = „object.create_images”
    bl_label = „Create images for lightmaps”

    @classmethod
    def poll(cls, context):
        return context.scene.render.engine == ‚CYCLES’

    def execute(self, context):
        filepath = bpy.data.filepath
        directory = os.path.dirname(filepath)
#sajnos az area contextjét meg kell változtatni, hogy a poll ne dobja vissza
a mentést
        area = context.area
        old_type = area.type
        area.type = ‚IMAGE_EDITOR’
        for mat in bpy.data.materials:

            if mat.name.find(‚UVEG’):
                print(„mat:”+mat.name+str(int(context.scene.ResPreset)))
                bpy.ops.image.new(name=”lm”+mat.name,width=int(context.scene.
ResPreset),height=int(context.scene.ResPreset),alpha=False)
                lm = bpy.data.images[„lm”+mat.name]
                area.spaces[0].image = lm
                bpy.ops.image.save_as(filepath=os.path.join( directory ,
„lm”+mat.name+”.png”))
                mat.node_tree.nodes[„Image Texture”].image = lm
        area.type = old_type
        return {‚FINISHED’}

#meshek kiválasztása, kivéve az üvegeket
class SelectBakeable(bpy.types.Operator):
    „””Select everything bakeable”””
    bl_idname = „object.select_bakeable”
    bl_label = „Select bakeable”

    @classmethod
    def poll(cls, context):
        return context.scene.render.engine == ‚CYCLES’

    def execute(self, context):
        bpy.ops.object.select_all(action=’DESELECT’)
        for obj in context.scene.objects:
            if obj.type == ‚MESH’ and len(obj.material_slots)>0 and obj.
material_slots[0].material.name.find(‚UVEG’):
                obj.select = True
                context.scene.objects.active = obj

        return {‚FINISHED’}

#Felhasználói felület meghatározása
class ArchicadConvertPanel(bpy.types.Panel):
    „””Archicad to game engine”””
    bl_label = „Archicad to game engine”
    bl_idname = „SCENE_PT_layout”
    bl_space_type = ‚PROPERTIES’
    bl_region_type = ‚WINDOW’
    bl_context = „scene”

    def draw(self, context):
        layout = self.layout

        scene = context.scene
        row = layout.row()
        row.scale_y = 3.0
        row.operator(„object.import_post_process”)

        layout.operator(„object.switch_to_baking”)
        layout.prop(scene, „SimpleSky”)
        layout.prop(scene, „Latitude”, icon = ‚MESH_UVSPHERE’)
        layout.prop(scene, „Month”, icon = ‚SORTTIME’)
        layout.prop(scene, „SkyTime”, icon = ‚TIME’)
        layout.prop(scene, „Clouds”, icon = ‚LAMP_SUN’)
        layout.label(text=”Light quality:”, icon=’LAMP_SUN’)
        row = layout.row()

        row.prop(scene.cycles, „samples”, text=”Samples”)
        row.prop(scene, „QualityPreset”, text=”Presets”)
        layout.prop(scene, „ResPreset”)
        layout.operator(„object.create_images”)
        layout.operator(„object.select_bakeable”)
        row = layout.row()
        row.scale_y = 2.0
        row.operator(„object.bake”)
        layout.label(icon=’ERROR’, text=”WARNING: setting to high values,
it may calculate for hours!”)

        layout.operator(„object.switch_to_game”)
        layout.operator(„object.assign_shaders”)


# render minőség presetek enumerációja
bpy.types.Scene.QualityPreset = EnumProperty(
items = [(‚10’, ‚Preview’, ‚’),
         (‚50’, ‚Very Low’, ‚’),
         (‚100’, ‚Low’, ‚’),
         (‚200’, ‚Normal’, ‚’),
         (‚500’, ‚High’, ‚’),
         (‚1000’, ‚Very High’, ‚’)],
name = „Light quality presets”,
description=”Quality of shadows and refractions”,
update=samplesetter)
```

```
# lightmap felbontás presetek enumerációja
bpy.types.Scene.ResPreset = EnumProperty(
items = [(‚128’, ‚128 - Very Low’, ‚’),
         (‚256’, ‚256 - Low’, ‚’),
         (‚512’, ‚512 - Normal’, ‚’),
         (‚1024’, ‚1024 - High’, ‚’)],
description=”Resolution of the lightmap”,
name = „Resolution”)



#az osztályok beregisztrálása az addon betölésekor
def register():
    bpy.utils.register_class(ArchicadConvertPanel)
    bpy.utils.register_class(SelectBakeable)
    bpy.utils.register_class(CreateImages)
    bpy.utils.register_class(SwitchToBaking)
    bpy.utils.register_class(SwitchToGame)
    bpy.utils.register_class(PreProcess)
    bpy.utils.register_class(UseShaders)

# és a kiregisztrálás
def unregister():
    bpy.utils.unregister_class(ArchicadConvertPanel)
    bpy.utils.unregister_class(SelectBakeable)
    bpy.utils.unregister_class(CreateImages)
    bpy.utils.unregister_class(SwitchToBaking)
    bpy.utils.unregister_class(SwitchToGame)
    bpy.utils.unregister_class(PreProcess)
    bpy.utils.unregister_class(UseShaders)

if __name__ == „__main__”:
    register()

#program vége

############################
# shader.py
# a textúrázott, lightmapelt phong technika shaderkódjai
# pythonba ágyazott GLSL kód

import bge

cont = bge.logic.getCurrentController()

VertexShader = „””

   varying vec2 uv1;
   varying vec2 uv2;
   varying vec4 color;

void main()
{
   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
   uv1 = gl_MultiTexCoord0.xy;
   uv2 = gl_MultiTexCoord1.xy;

   vec3 normalDirection =
            normalize(gl_NormalMatrix * gl_Normal);
   vec3 viewDirection =
            -normalize(vec3(gl_ModelViewMatrix * gl_Vertex));

   vec3 lightDirection = normalize(vec3(gl_LightSource[0].position));

   vec3 specularReflection;
   if (dot(normalDirection, lightDirection) < 0.0)
   {
      specularReflection = vec3(0.0, 0.0, 0.0);
   }
   else
   {
      specularReflection = vec3(gl_LightSource[0].specular)
       * vec3(gl_FrontMaterial.specular)
       * pow(max(0.0, dot(reflect(-lightDirection,
       normalDirection), viewDirection)),
       gl_FrontMaterial.shininess);
      }

      color = vec4(specularReflection, 1.0);


   }
„””

FragmentShader = „””

   varying vec2 uv1;
   varying vec2 uv2;
   varying vec4 color;
   uniform sampler2D difftex;
   uniform sampler2D lmtex;

   void main()
   {
      gl_FragColor = texture2D(difftex, uv1) *
                        texture2D(lmtex, uv2) + color;

   }
„””
# a textúra források megadása
mesh = cont.owner.meshes[0]
for mat in mesh.materials:
    shader = mat.getShader()
    if shader != None:
        if not shader.isValid():
            shader.setSource(VertexShader, FragmentShader, 1)
            shader.setSampler(‚difftex’,0)
            shader.setSampler(‚lmtex’,1)

####################
# solidshader.py
# a textúra nélküli, lightmapelt phong technika shaderjei
# pythonba ágyazott GLSL kód
import bge

cont = bge.logic.getCurrentController()

VertexShader = „””

   varying vec2 uv2;
   varying vec4 color;

   void main()
   {
      gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
      uv2 = gl_MultiTexCoord0.xy;

vec3 normalDirection =
         normalize(gl_NormalMatrix * gl_Normal);
vec3 viewDirection =
         -normalize(vec3(gl_ModelViewMatrix * gl_Vertex));

vec3 lightDirection = normalize(vec3(gl_LightSource[0].position));

vec3 specularReflection;
if (dot(normalDirection, lightDirection) < 0.0)
{
   specularReflection = vec3(0.0, 0.0, 0.0);
}
else
{
   specularReflection = vec3(gl_LightSource[0].specular)
    * vec3(gl_FrontMaterial.specular)
    * pow(max(0.0, dot(reflect(-lightDirection,
    normalDirection), viewDirection)),
    gl_FrontMaterial.shininess);
}

color = vec4(specularReflection, 1.0);
```

```
   }
„”””

FragmentShader = „”””

   varying vec2 uv2;
   varying vec4 color;
   uniform sampler2D lmtex;

   void main()
   {
      gl_FragColor =  gl_FrontMaterial.diffuse * texture2D(lmtex, uv2) +
color;

   }
„”””
# a lightmap textúra hozzákapcsolása a samplerhez
mesh = cont.owner.meshes[0]
for mat in mesh.materials:
    shader = mat.getShader()
    if shader != None:
        if not shader.isValid():
            shader.setSource(VertexShader, FragmentShader, 1)
            shader.setSampler(‚lmtex’,0)

##############################
# glassshader.py
# az üveg technika shaderjei GLSL nyelven
# csillogás és átlátszóság
import bge

cont = bge.logic.getCurrentController()

VertexShader = „”””

   varying vec2 uv1;
   varying vec2 uv2;
   varying vec4 color;

   void main()
   {
      gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
      uv1 = gl_MultiTexCoord0.xy;
      uv2 = gl_MultiTexCoord1.xy;

      vec3 normalDirection =
               normalize(gl_NormalMatrix * gl_Normal);
      vec3 viewDirection =
               -normalize(vec3(gl_ModelViewMatrix * gl_Vertex));
      vec3 lightDirection = normalize(vec3(gl_LightSource[0].position));

      vec3 specularReflection;
      if (dot(normalDirection, lightDirection) < 0.0)
      {
         specularReflection = vec3(0.0, 0.0, 0.0);
      }
      else
      {
         specularReflection = vec3(gl_LightSource[0].specular)
          * vec3(gl_FrontMaterial.specular)
          * pow(max(0.0, dot(reflect(-lightDirection,
          normalDirection), viewDirection)),
          gl_FrontMaterial.shininess);
      }

      color = vec4(specularReflection, 1.0);


   }
„”””

FragmentShader = „”””

   varying vec2 uv1;
   varying vec2 uv2;
   varying vec4 color;

   void main()
   {
      gl_FragColor = vec4(1,1,1,color.x);

   }
„”””
# ennél nincs textúra
mesh = cont.owner.meshes[0]
for mat in mesh.materials:
    shader = mat.getShader()
    if shader != None:
        if not shader.isValid():
            shader.setSource(VertexShader, FragmentShader, 1)
```

SZÓMAGYARÁZAT

Fontos egy - a fontosabb kifejezéseket kifejtő - szómagyarázat melléklése a dolgozat mellé, hiszen a témánk jellegéből fakadóan hemzseg az angol szavaktól.

biased rendering: a renderelési egyenletet szándékos egyszerűsítésekkel megoldó renderelő motorokat soroljuk ide. A végeredmény minőségében nem, de fizikai értelemben vett helyességében megmutatkozik.

CPU: central processing unit - a számítógép fő, általános célú processzora

diffuse szín, textúra: a felület fehér fényben megvilágított színe, textúrája

FPS: frame per secundum - képkocka per másodperc, az animáció, vagy videó képsebessége

frame: képkocka, egy kép a videóból vagy a renderelt animációból

game engine, játékmotor: játékok készítéséhez használt szoftverek angol elnevezése. Segítségükkel interaktív tartalmakat készíthetünk változatos célokkal, nem feltétlenül játékokra korlátozódik a használatuk.

global illumination: valós fényviselkedést közelítő mód, mely során nem csupán a direkt megvilágítás, hanem a környező felületekről visszaverődő fény is befolyásolja a vizsgált pont megvilágítottságát. Fotorealisztikus eredményekhez nélkülözhetetlen a használata, a fényviselkedés szimulációja miatt a renderidőt jelentősen növeli.

gouraud shading: gömbölyítettnek, simának ható árnyalás, de a fényértékek csak a csúcspontokra vannak számolva, a felületen ezek értékei interpolálva jelennek meg.

jelenet, scene: a virtuális térben található objektumok - geometriák, fények, egyéb elemek - adott elrendezése

normálvektor: egy felületi pontra merőlegen néző vektor

orthografikus: rövidülés nélküli

pipeline: olyan feldolgozási folyamatok sora, ahol az egyik kimenete a másik bemenete, grafikai pipeline: olyan folyamat sorozat amik egy 2D képet készítenek egy 3D jelenetből

poligon: másnéven sokszög, valós időben csak háromszöget értünk rajta

pontháló, mesh: csúcspontok, élek és lapok gyüjteménye, amik egy testet alkotnak

raszterizáció: egy síkbeli geometriai elem pixelekre való konvertálása, textúrába festése

rendering: képszintézis. A renderelés során a megadott adatok

render engine, rendermotor, renderer: szoftver, mely a képszintézist végzi.

rendertextúra: az a textúra, amire a videokártya a renderképet készíti

shader: rövid programok a videokártya számára, ld. 2.2.1.

techdemó: technológiai demonstráció - főleg videojátékoknál elterjedt rövidítés

texture baking: eljárás, mely során a renderelt textúra-, illetve anyaginformációkat véglegesen hozzárendeljük a geometriához. Játékmotorok használata során szükség lehet kettő (vagy akár több) UV map meglétére, egyik tartalmazza az anyagot, a másik pedig a megvilágítási információkat (light map, ez utóbbi az úgynevezett GI bake, természetesen global illumination használatakor).

unbiased rendering: olyan render eljárás, mely nem teljesítni a renderelési egyenletet, fénytanilag fals eredmény hoz létre

utómunka, post-processing: a renderelt kép módosítása 2D információkból

UV koordináta: egy háromszög csúcspontjához rendelt 2D koordináta, ami a textúrázás UV síkján értendő

UV unwrapping: térbeli felületek síkba terítése (a 3D térben használt koordinátarendszerek általában XYZ tengellyel jelöltek, ezért a textúrákat az UV síkba fejtjük).

valós idő, valós idejű: olyan sebességű képsorozat, amin rögtön, egyből érezhető és látszik a beavatkozás

vertex: egy geometria ponthálójának egy csúcsa

videokártya, grafikus kártya: a számítógépnek azon alkatrésze amely a vizuális megjelenítésért felel

viewport: programokban található 3D szerkesztő felület

Z-buffer, depth-buffer: renderelés során készített olyan textúra, amiben az egyes pixeleken található felületi pont kamerától való távolsága van tárolva

FORRÁSOK

IRODALOMJEGYZÉK,HIVATKOZÁSOK

3DGRAF:
Dr. Szirmay-Kalos László, Antal György, Csonka Ferenc: Háromdimenziós grafika, animáció és játékfejlesztés

BBMETAL:
http://bertrand-benoit.com/blog/2013/05/26/materialism-1-5-rough-metal/

BIMX:
http://www.graphisoft.hu/bimx/

C4DDOC:
http://www.maxon.net/support/documentation.html

C4DMERGE:
http://www.c4dcafe.com/ipb/topic/54451-merging-two-scene-files/

C4DUPDATE:
http://www.youtube.com/watch?v=Dm00BtiE5NM

CGARCH:
http://forums.cgarchitect.com/75832-unreal-engine-4-archviz.html

COL_UE:
http://www.youtube.com/watch?v=OpbAX75e3Ak

CORONAFORUM:
https://corona-renderer.com/forum/index.php?topic=5416.0

DISNEYPBS:
http://disney-animation.s3.amazonaws.com/library/s2012_pbs_disney_brdf_notes_v2.pdf

GLOBILLUM:
Global illumination - Wikipedia - 16 October 2014, http://en.wikipedia.org/wiki/Global_illumination

HISTORYOFCGI:
Wayne Carlson: A Critical History of Computer Graphics and Animation, http://design.osu.edu/carlson/history/

KOOLA2014:
https://forums.unrealengine.com/showthread.php?28163-ArchViz-Lighting
pathtracing: Path tracing - Wikipedia - 11 July 2014, http://en.wikipedia.org/wiki/Path_tracing

PBR:
http://www.marmoset.co/toolbag/learn/pbr-theory

PIPELINE: Graphics pipeline - Wikipedia - 23 September 2014, http://en.wikipedia.org/wiki/Graphics_pipeline

RENDEREQ: Rendering equation - Wikipedia - 18 September 2014, http://en.wikipedia.org/wiki/Rendering_equation

ROMAN2009: http://www.youtube.com/watch?v=PSGx4bBU9Qc

ROMAN3RD7TH: Alex Roman: From Bits To The Lens. Fordította: Laura F. Farhall. The Third & The Seventh S.L., 2013

SZIRMAY1999: Dr. Szirmay-Kalos László: Számítógépes grafika, 1999 http://sirkan.iit.bme.hu/~szirmay/grafika/graf.pdf

UELIGHT:
https://docs.unrealengine.com/latest/INT/Engine/Rendering/LightingAndShadows/LightTypes/index.html

UEFORUM_FBX1: https://answers.unrealengine.com/questions/39885/materials-missing-on-fbx-import.html

UEFORUM_FBX2: https://docs.unrealengine.com/latest/INT/Engine/Content/FBX/ImportOptions/index.html

UEFORUM_FBX3: https://docs.unrealengine.com/latest/INT/Engine/Content/FBX/StaticMeshes/index.html

UEFORUM_UVMAP: https://forums.unrealengine.com/showthread.php?1861-Tutorial-How-to-proper-export-a-FBX-scene-from-Cinema-4D-to-U4-with-2-UV-channels

UEINTERAKTIV: http://www.youtube.com/watch?v=eTt7AGIpV2I

UDKMATERIALS: UDK material examples - http://udn.epicgames.com/Three/MaterialExamples.html

UNBIASED: Unbiased rendering - Wikipedia - 15 June 2014, http://en.wikipedia.org/wiki/Unbiased_rendering

ZEISS: http://www.zeiss.com/cinemizer-oled/en_de/home.html

Ábrajegyzék

Minden további, nem hivatkozott ábra saját készítésű.

Fürtön Balázs és Háda Ádám, Budapest, 2014.