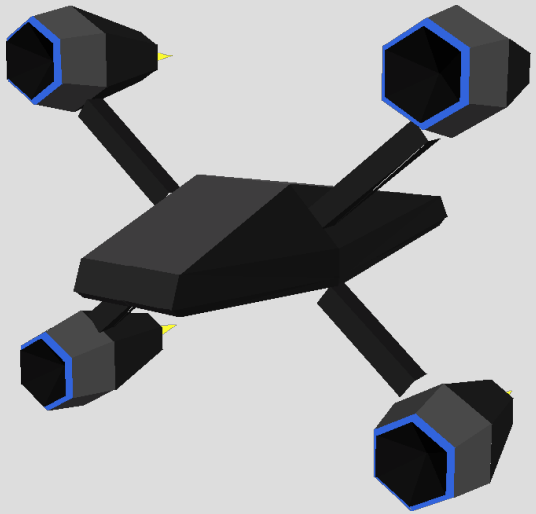


SDL_Universe

SDL, C++, 3D szoftver renderelő

Boros László,
harmadéves mérnökinformatikus

iamsemmu@gmail.com

I C what you did last summer
Programozói Konferencia 2014

http://progkonf.eet.bme.hu

SDL_Universe

- 3D szoftver renderelő függvénykönyvtár
- C++ nyelv → platformfüggetlen
- SDL alapok → platformfüggetlen
  (SDL1 és SDL2 kompatibilis)

Miért?

- Érdekel a 3D geometria illetve leképezés
- Régi vágyam egy ilyet “összedobni” (középsuliban fogalmazódott meg bennem az ötlet)
- Kihívás

“Szoftver renderelő”?

- A kép megjelenítéséhez nem használ GPU-t (videókártyát), hanem “kézzel” számolja ki a végeredményt a szoftver
  (olyan helyeken is lehet alkalmazni, ahol nincs GPU, bár manapság már ritkán találni ilyet)
- Színtiszta matek (3D koordinátageometria), erre épül a programom
- Ennek az algoritmusát kellett nekem megírnom

“Függvénykönyvtár”?

- “Kódcsomag”, adott probléma megoldására. Elsődleges szempont, hogy többször felhasználható legyen
  (ne kelljen mindig mindent megírni 0-ról)
- Általában könnyen beépíthetőek és használhatóak
  (a belső működésük el van rejtve, azzal nem kell törődnie a használónak)
- Ilyen az SDL_Universe
  (és ha már itt tartunk maga az SDL is!)

Hogy lesz 3D-ből 2D?

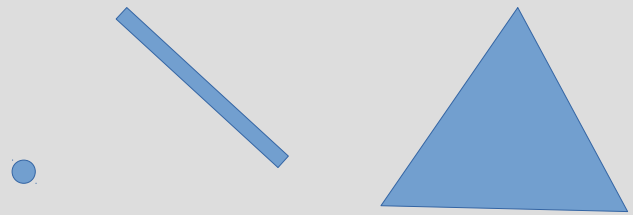
- Egy szóval: leképezés
- Bevezetés a Számításelméletbe: $N \rightarrow M$ dimenziós terek között lineáris leképezés mátrixa
- GPU-k hardverből tolják (ezért olyan gyorsak), de én nem ilyet akartam
- Érthetőbbet, amihez nem kell hardcore matek (emiatt picit lassabb a programom, de így is tök jól fut, illetve nem a sebességen van a hangsúly, elvégre szoftver renderelő)

Leképezés hozzávalók:

- Végy egy hagyományos 3D teret
- Helyezz el benne egy “szemet”
- Add meg, hogy merre néz a “szem”
- Illetve azt, hogy számára mi a felfele irány (ezért azért kell, hogy egyértelműen meghatározható legyen a nézőpont, enélkül szabadon lehetne forgatni cw ill. ccw)
- Végül pedig helyezd el a tárgyaidat a 3D térben

Milyen tárgyakat?

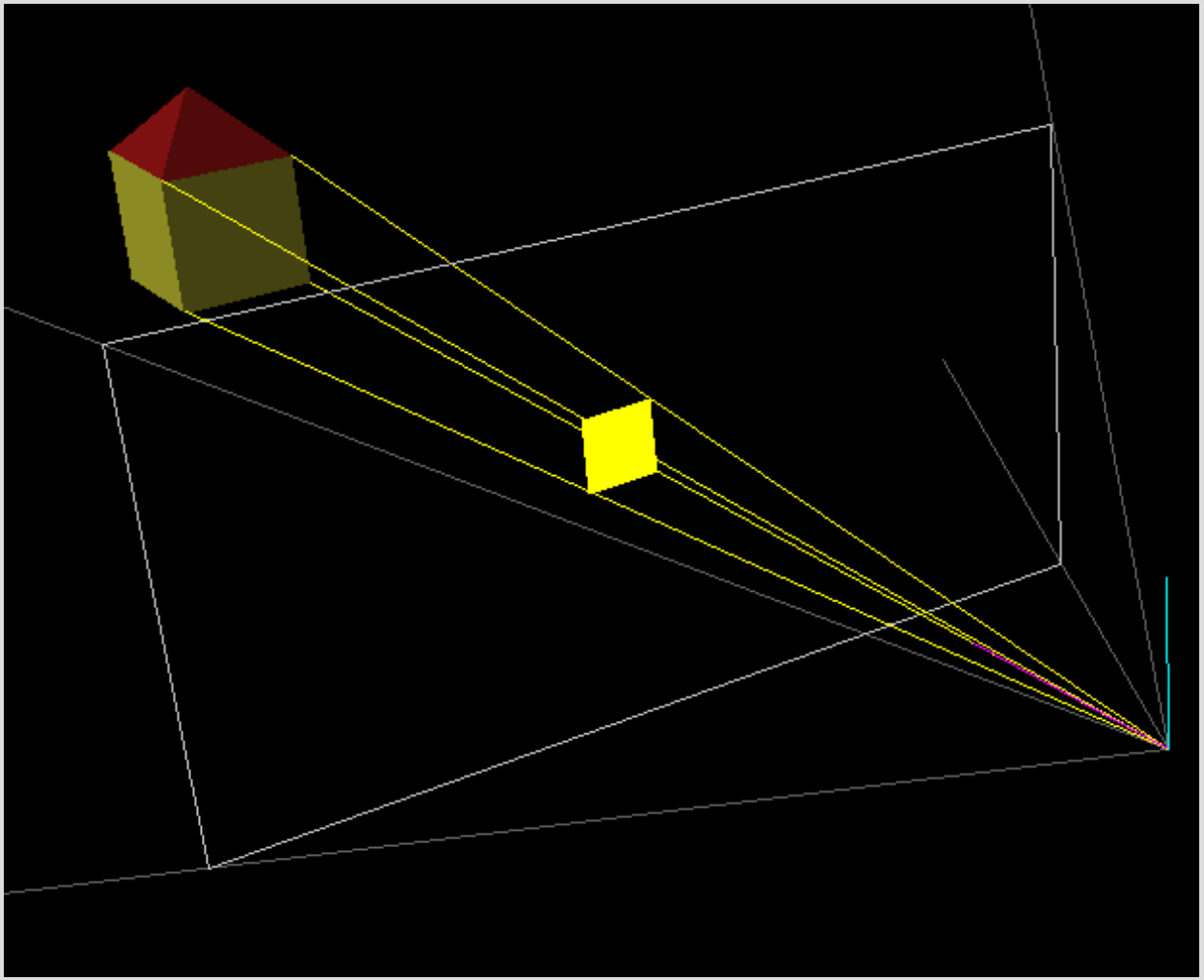
- Pontokat, szakaszokat, kitöltött háromszögeket
- = *térbeli primitívek*
- 1, 2, ill. 3 helyvektor határozza meg őket
- Ezekből ~mindent fel lehet építeni
  (sok kicsi háromszögből rakják össze a mai játékok is a testeket, illetve azokra húznak textúrát)



Na de mégis hogy történik a leképezés?

- Nagyon egyszerű: a térben megfeleltetünk egy síkot a számítógépünk monitorának síkjának
- Ezen a síkon egy bizonyos téglalap, ami pont a "szemmel" szemben van lesz a monitor síkja
- Minden egyes térbeli pontot összekütönk a "szemmel"
- Ha ezek az egyenesek döfik a monitor síkját, akkor sikeresen leképeztünk egy térbeli pontot a monitorra
- Ezt felhasználva pedig akárhány pontot és akármilyen sokszöget le tudunk képezni

Így



Azaz

- Térbeli pontnál csak 1 pontot kapunk
- Szakasznál a 2 végpontját képezzük le, majd azokat a monitoron kötjük össze
  (szóval nem kell a szakasz végtelen pontját leképezni, elég a 2 végpontját)
- Háromszögnél pedig szintén csak a 3 csúcsát képezzük le, majd a kapott pontokat kötjük össze a monitoron
  (illetve töltjük ki a háromszöget a színével, kb. mint Paint-ben a festékesdoboz)

Buktatók

- Mi van, ha egy szakasz döfi az egyik háromszöget?
- Mi van, ha két háromszög metszi egymást?

- Kézenfekvő megoldás: az összes ilyet megkeressük, majd a primitíveket daraboljuk, hogy ne metszék egymást (maximum érintsék)
- (sajnos ez még nincs implementálva)

Ez a 3D leképezés alapja

- Ezt csinálja az SDL_Universe
- Meg még egy kicsit többet!

Belső adatszerkezetek

- SU::Vector
  - Helyvektor
    (X, Y, Z koordináták)
- SU::Line
  - Térbeli végtelen egyenes
    (van egy fix pontja és egy iránya)
- SU::Plane
  - Térbeli végtelen sík
    (van egy fix pontja és egy normálvektora)
- Ezek gyakorlatilag koordinátageometriai fogalmak

Belső adatszerkezetek

- SU::Point
  - Csak egy vektor
- SU::Segment
  - Véges szakasz, 2 vektor
- SU::Triangle
  - Kitöltött háromszög, 3 vektor
  - Ez árnyékolódhat is, be lehet állítani minden háromszögnek külön
- Ezek a *térbeli primitívek*! (Mindegyiknek van színe)

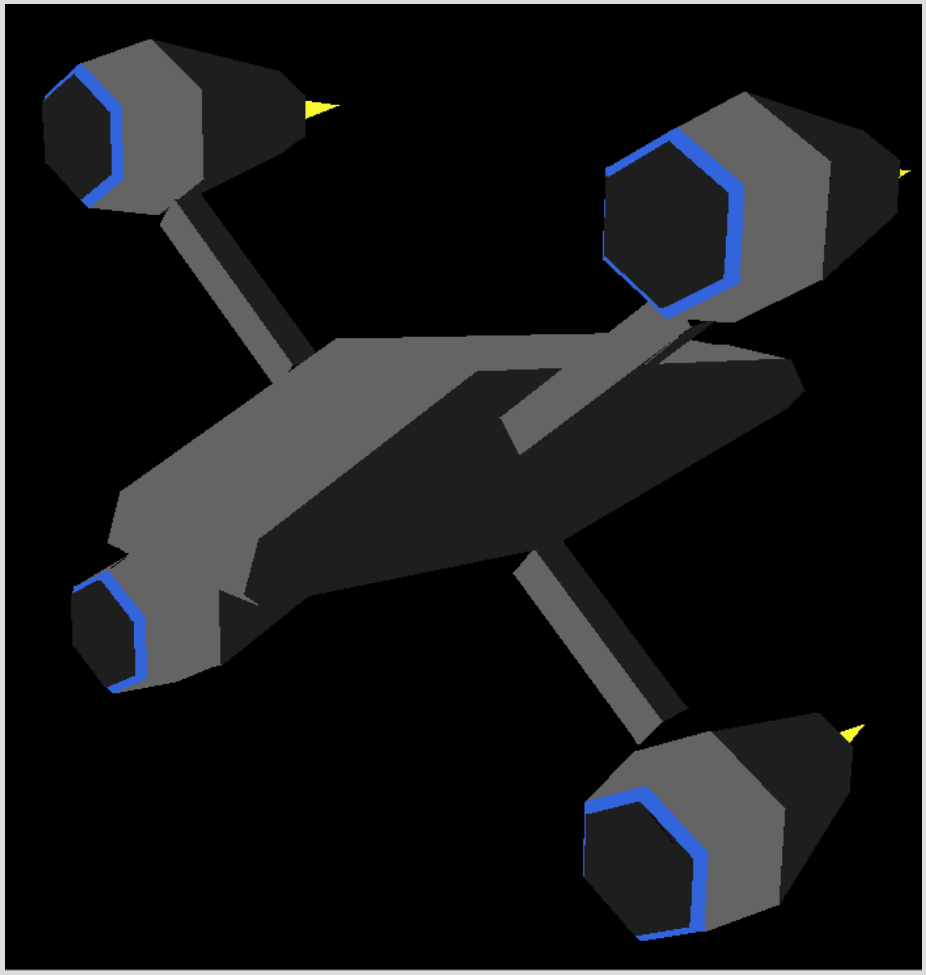
Árnyékolás

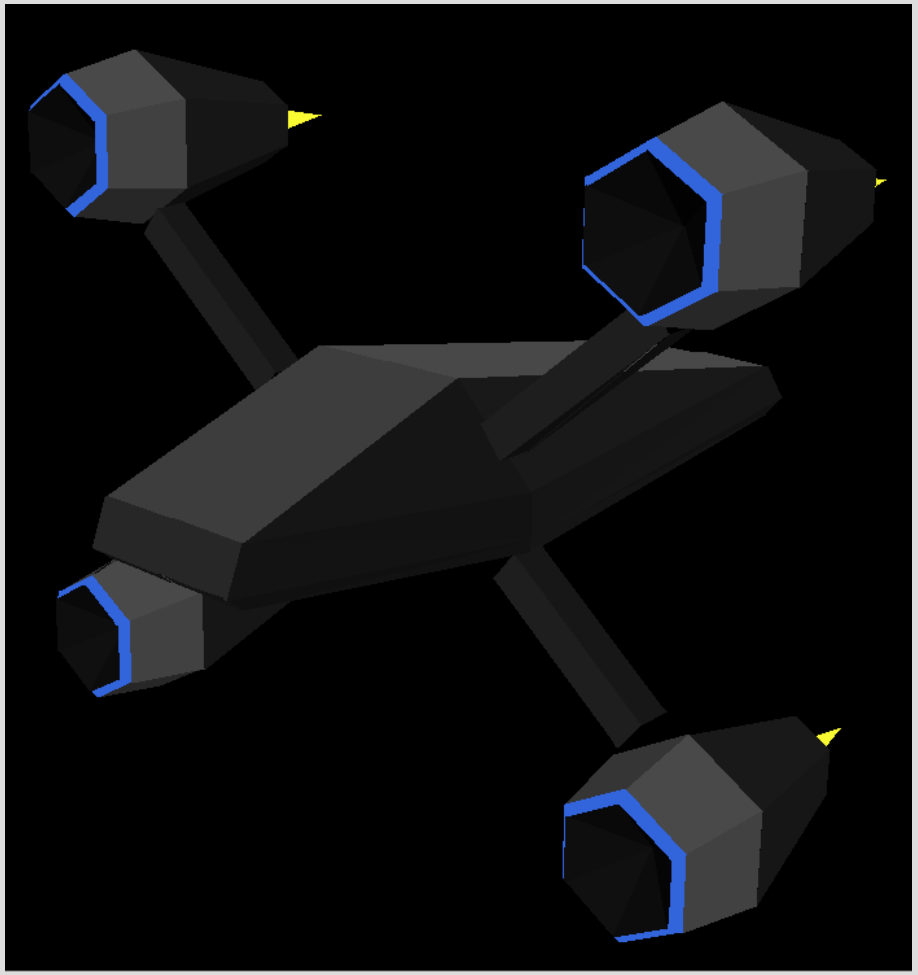
- Nagyon egyszerű, de effektív módszer: minden háromszög kirajzolásánál figyelembe vesszük, hogy merre néz → minél inkább fényforrás fele, annál jobban átszínezzük (matematikailag: a háromszög normálvektorának és a fény irányvektorának bezárt szögét vizsgáljuk)
- 2 fajta fényforrás van: globális és pontszerű.

Árnyékolás

- Globális:
  - Van színe, iránya
  - Úgy vesszük, hogy párhuzamosan jön belőle a "fény" (mint pl. Nap)
  - Van ambiens együtthatója is, ez olyan fény, ami mindig mindenhol ott van (mert a valóságban minden visszaveri valamennyire a fényt, szóródik)
- Pontszerű:
  - Van helye, színe, intenzítása (milyen távolságokra lévő háromszögekre van hatással)

Árnyékolás





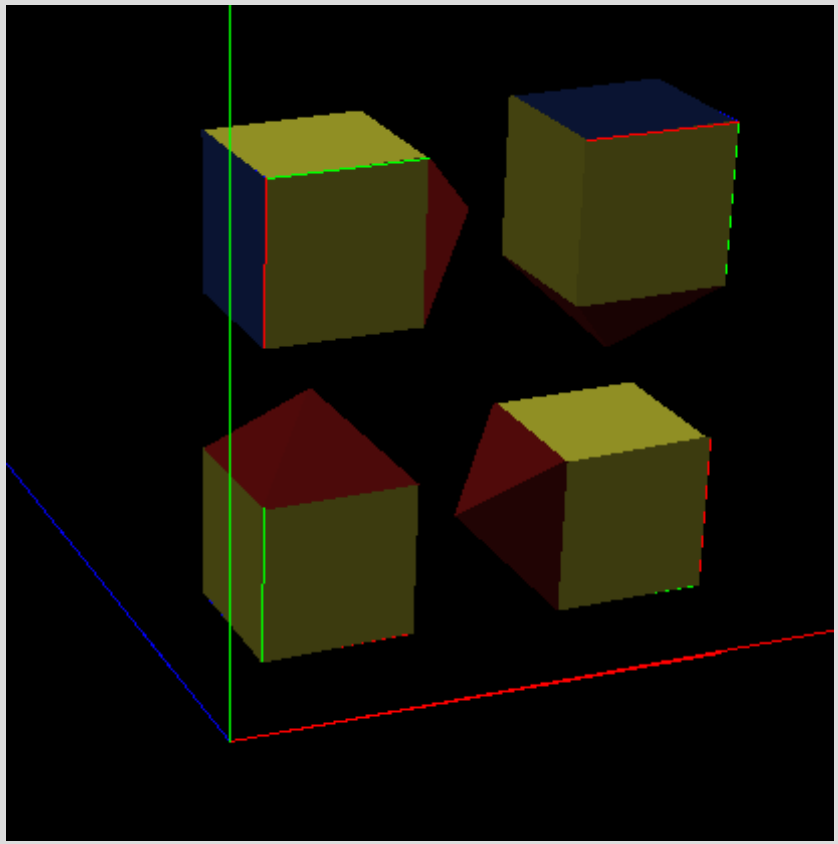
Belső adatszerkezetek

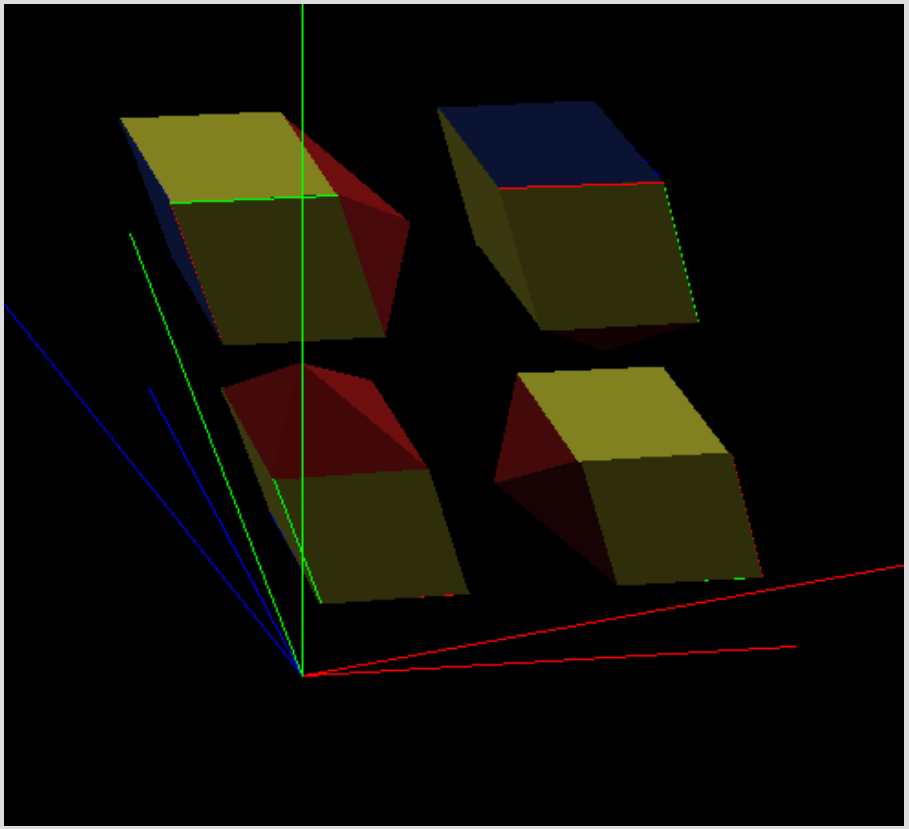
- SU::Model
  - *Primitíveket* foglal egybe
  - Statikus, utólag nem túl könnyen módosítható
  - Cél: egy adott tárgyat/testet ne kelljen többször elkészíteni/lemodellezni. Ez az adatstruktúra többször felhasználható.
- SU::Object
  - Ez az SU::Model "példányosítása"
  - Ez jelenik meg, ezt lehet transzformálni
  - Több ilyen SU::Object-nek lehet ugyanaz a hozzárendelt Model-je, így lehet őket többször felhasználni
    (pl. van 1 ház model, amivel egy egész utcát létrehozhatunk)
  - Illetve lehetnek "gyermek" Object-jei, így lehet többet csoportosítani (és együtt transzformálni)

Transzformálás

- Minden SU::Object tartalmaz 4 vektort: X, Y, Z, pozíció
- A pozícióvektorral pozícionálhatjuk, a többivel pedig deformálhatjuk (forgathatjuk, nyújthatjuk, tükrözhetjük, stb.)
- Olyan, mintha minden objektumnak saját koordinátarendszere lenne, amit ha mi eldeformálunk, akkor a benne lévő tárgy is eldeformálódik

Transzformálás



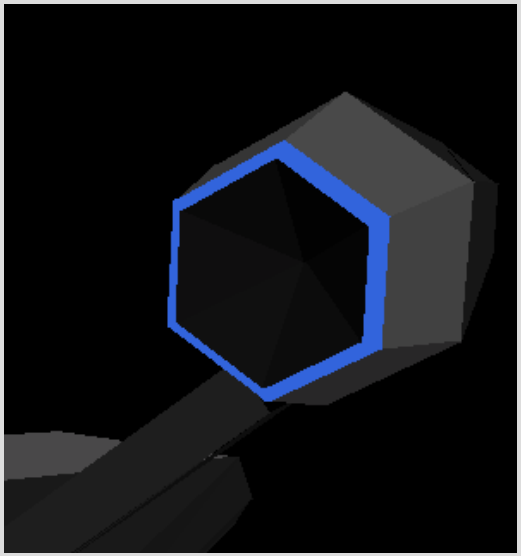


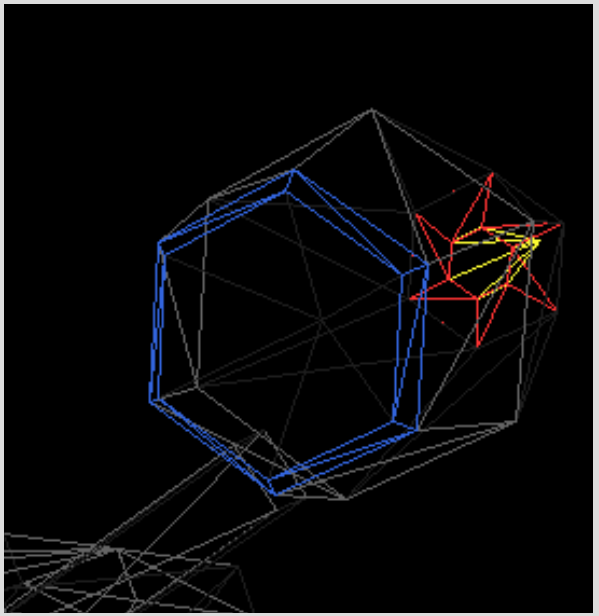
Belső adatszerkezetek

- SU::Camera
  - A “szem” fontosabb vektorait tartalmazza
  - Pozíció, nézési irány, felfele irány, FOV
- SU::Flags
  - Kapcsolók az egyes funkcionalitásokhoz
  - Nagyban befolyásolják a végeredményt

DEBUG_WIREFRAMING

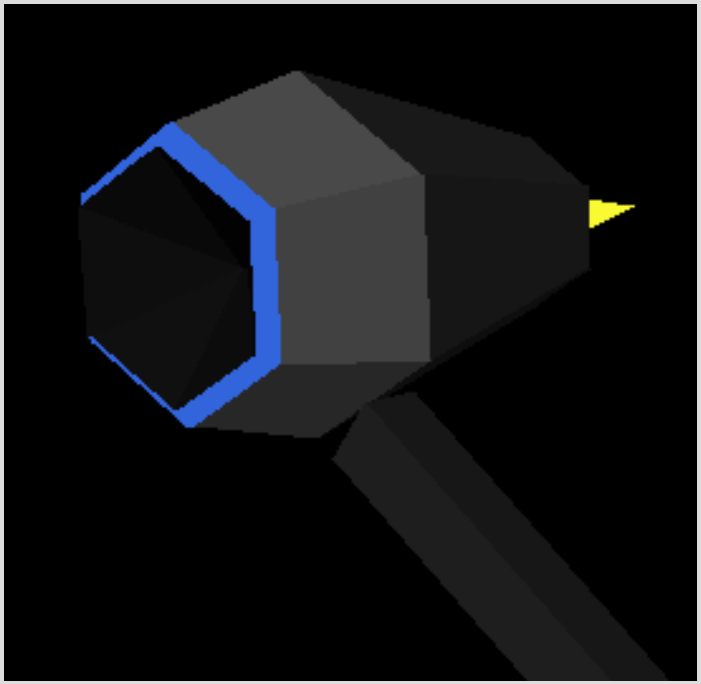
- Ezt bekapcsolva a kitöltött háromszögek helyett 3 szakasz jelenik meg
- A “tömör” testek dróthálóját láthatjuk
- Illetve az egyes háromszögeket könnyebben megkülönböztethetjük

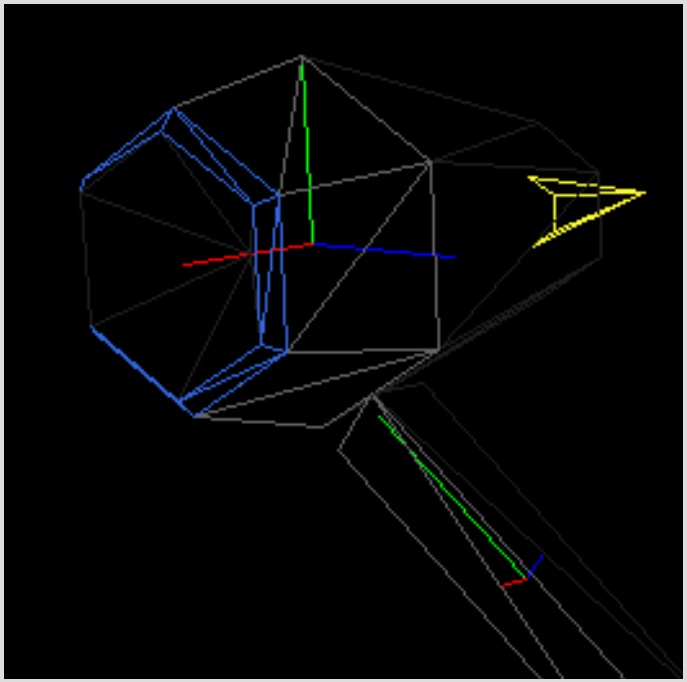




DEBUG_TRANSFORMATIONS

- Az SU::Object-ek transzformációs vektorait teszi láthatóvá (megjelenik minden objektum X, Y, Z vektora is)

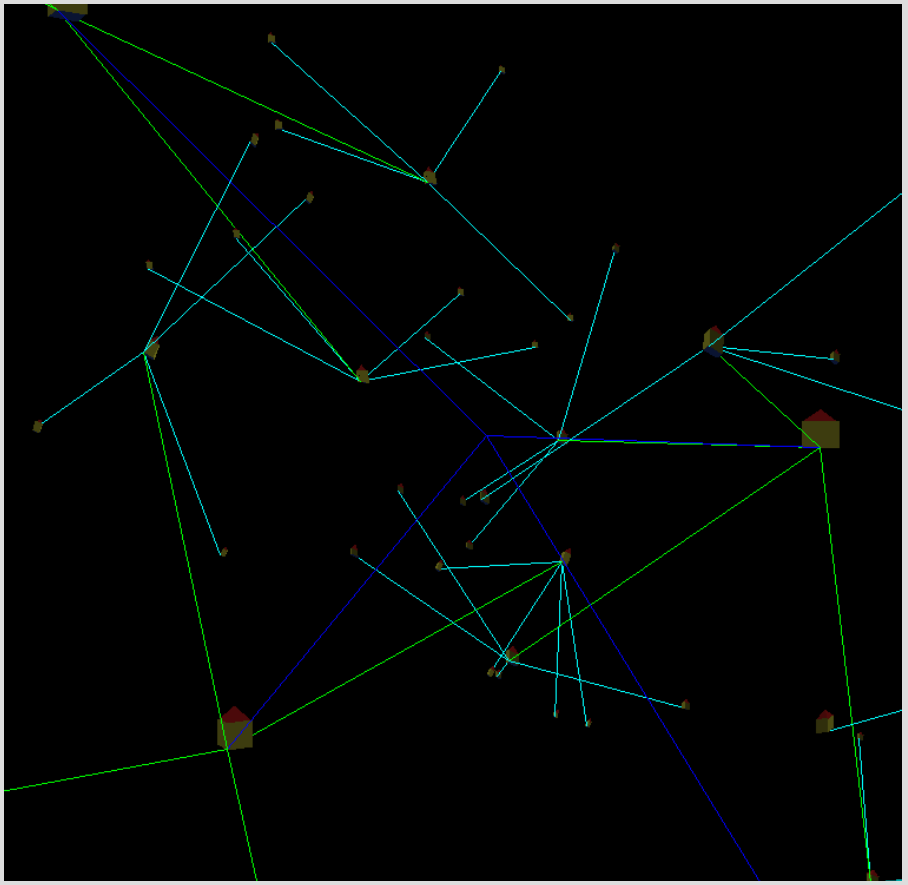




(a piros+kék+zöld tengely a lényeg, a wireframing csak láthatósági okokból van bekapcsolva)

DEBUG_TRANSLATIONS

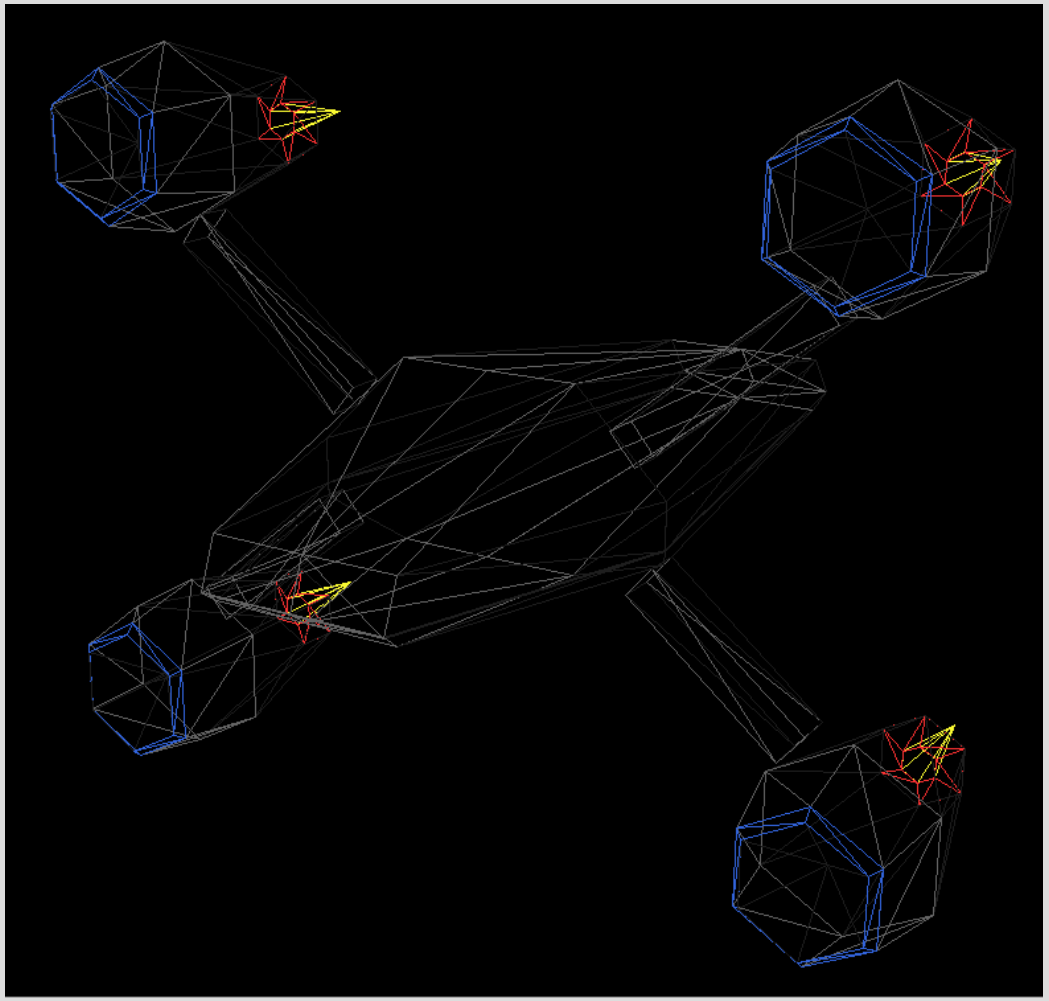
- Az előzőhöz hasonló, csak épp a pozícióvektorokat jeleníti meg



ONLY_FACING_TRIANGLES

- Optimalizáció, sokat gyorsít
- Csak azokat a háromszögeket rajzoljuk ki, amik a kamera fele néznek
  (hisz általában “tömör” testeket jelenítünk meg, ami azt jelenti, hogy az a háromszög, ami nem a kamera fele néz, a test hátulja, úgyhogy nagy valószínűséggel úgy is eltakarja valami)

ONLY_FACING_TRIANGLES



DEPTH_SORT

- Távolsági rendezés
- Ha ki van kapcsolva, mindent olyan sorrendben rajzolunk ki, amilyen sorrendben létrejöttek/hozzá lettek adva a világhoz
- Bekapcsolt esetben kirajzolás előtt mindent rendezünk a kamerához viszonyított pozíciók alapján
- Nem tökéletes, de gyors és relatív keveset hibázik

Z_BUFFER

- Az előző rendezés hibáit javítja
- Minden pixel kirajzolása előtt megnézzük, hogy a jelenleg oda rajzolt dolog milyen távol van → csak akkor rajzolunk, ha a mostani közelebb esik a kamerához
- "Tökéletes", hiba mentes képet kapunk
- (sajnos még nem implementáltam)

LIGHTING

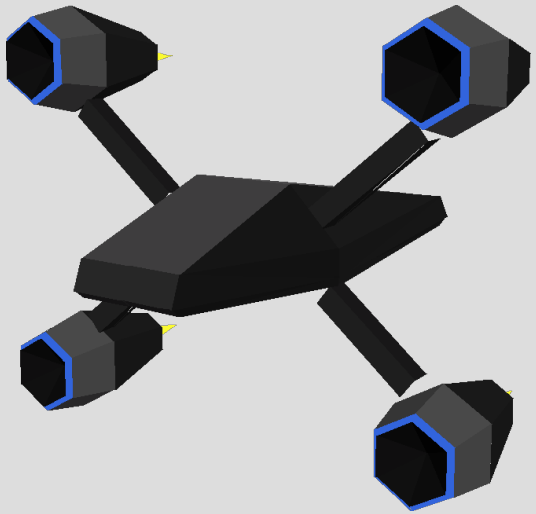
- Az árnyékolást kapcsolja be/ki
- Ennek a hatását már láttátok

Egyéb tervben lévő funckiók

- Félátlátszó színek
- Dinamikus részletesség
- Magasabb szintű testek, futás idejű tesszelláció

<demo>

Kérdések?



Köszönöm a figyelmet!

Boros László,
harmadéves mérnökinformatikus

iamsemmu@gmail.com

I C what you did last summer
Programozói Konferencia 2014

http://progkonf.eet.bme.hu