Wigner GPU Lab

11. fejezet

Vizualizáció

Lectures on Modern Scientific Programming
Wigner RCP
23-25 November 2015

Tartalom

- Grafika ismétlés
- Grafikus API-k történelme
- Implicit/explicit API-k
- Megjelenítő kiszolgálók és ablakozó rendszerek
- OpenGL dióhéjban
- Hello triangle!
- Hello Nbody!

Grafika ismétlés

Ismétlés a tudás anyja. – ismeretlen bölcs

Vizsgaismétlés a tudás jó édes anyja! – ismeretlen egyetemista

Inkrementális kép szintézis

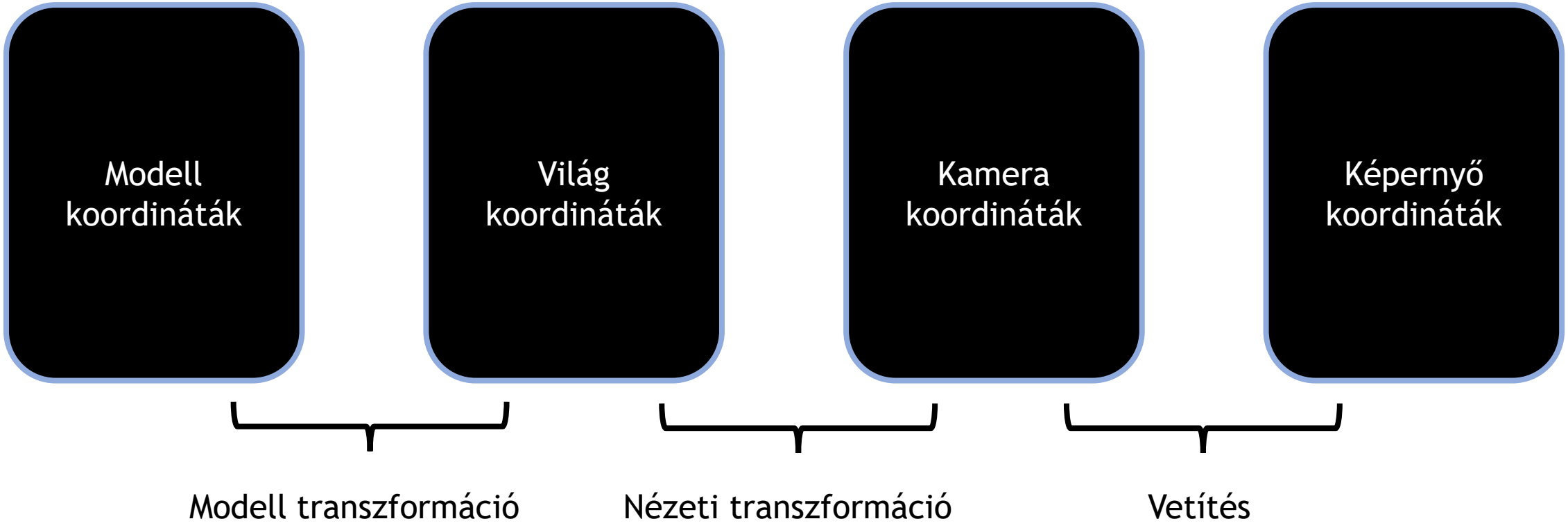
- Leginkább játékokban alkalmazott kép alkotási eljárás
- Célja: egy virtuális, 3 dimenziós színtér elemeit megjeleníteni a 2 dimenziós képernyőn
- A színtér elemeit háromszögekből (polygon) építjük fel
- A háromszögeket koordináta transzformációk sorozatával juttatjuk a képernyőre
- A transzformációk jellege közel azonos objektumról objektumra
- A háromszögeket transzformáció után textúrákkal kiszínezzük

Affin geometria

- A 3 dimenziós Euklideszi-teren ható lineáris transzformációk legtöbbje leírható 3×3-as mátrixokkal
  - Egyedül az elotlás nem
- Affin transzformációnak hívjuk a párhuzamosságot megtartó trafók összességét
- A 3 dimenziós Euklideszi-térhez tartozik egy 4 dimenziós homogén koordinátázott tér amiben minden számunkra lényeges trafó leírható 4×4-es mátrixokkal

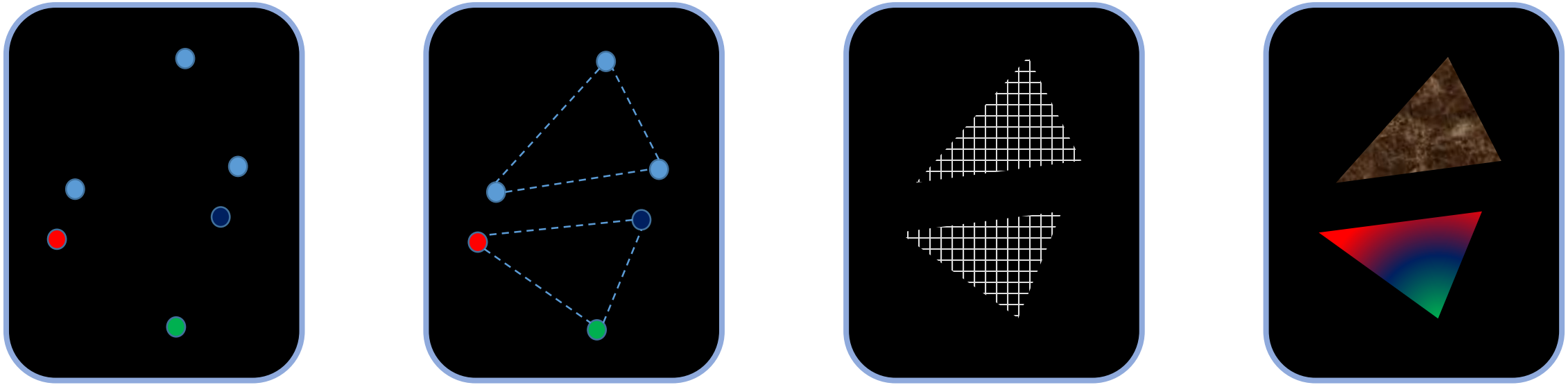
$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & b_{14} \\ a_{21} & a_{22} & a_{23} & b_{24} \\ a_{31} & a_{32} & a_{33} & b_{34} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix}$$

Csúcspont transzformációk



Minden egyes transzformációt egy 4x4-es mátrix ír le.

A raszterizálás menete



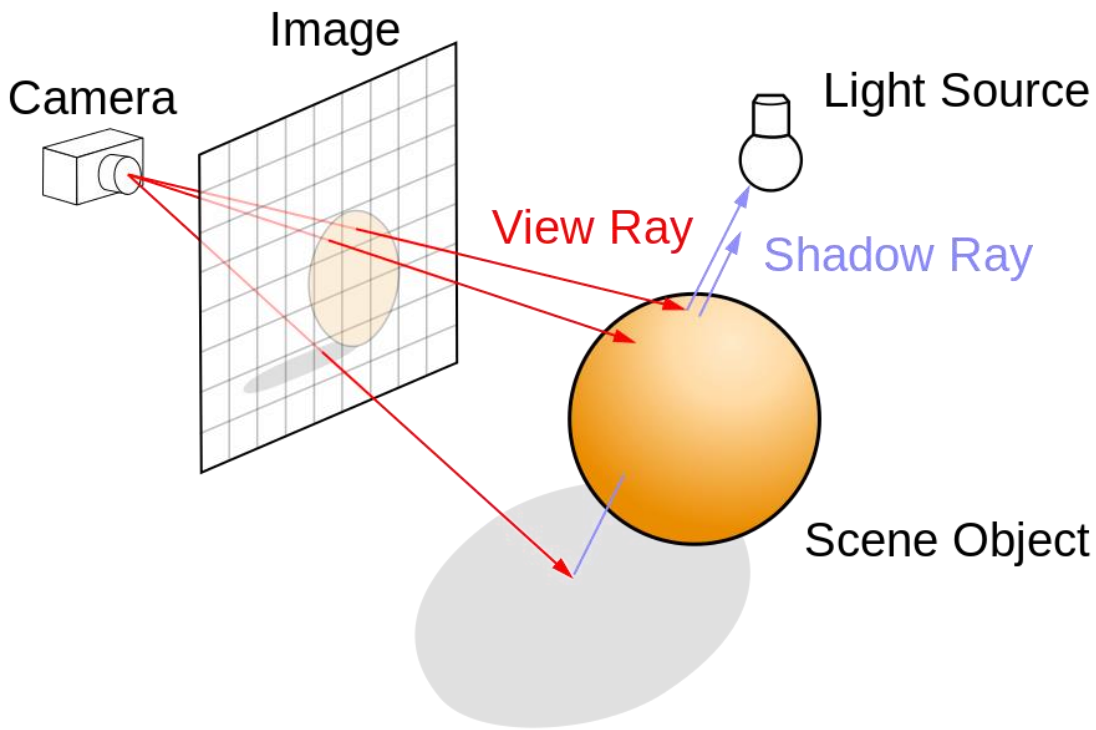
Primitívek összeszerelése

Raszterizálás

Interpolálás,
textúrázás,
színezés

Minden egyes lépés fix funkciós elemekkel támogatott.

Sugárvetés, sugárkövetés



- Animációs filmek kedvelt képalkotási eljárása
- A fénysugarak útját követi le a fényforrásoktól a képpontokig
  - A hatékonyság miatt pont a fordítottja történik
  - A képpontokból kiindulva keressük a fényforrásokat
- A sugarak felösszegzik a különböző felületkről származó színeket

Renderelési technikák

Inkrementális kép szintézis

- A render egyetlen közelítő megoldása
- Valós időben rajzolható
- Fix funkciós támogatás
  - Erre született

Sugárkövetés

- A render egyenlet pontos megoldása
- Tipikusan offline rajzolható
- Általános számolásnak minősül
  - GPU-n annyi sebből vérzik…

Grafikus API-k történelme

„He who controls the past commands the future, He who commands the future, conquers the past.” – Kane

GPU történelem

- Idővel változik, hogy mi a célja.
- A 80-as években, és a 90-es évek elején 2D (GUI)
- A 90-es években előtérbe kerül a 3D.
- Az első 3D-s grafikus gyorsítókat Az S3, az ATI és a Matrox gyártotta.
- A kártyákat API-kon keresztül programozták (!), amik között már ekkor dúlt a háború.
- Amelyik API elterjed, azok a kártyák, cégek maradnak fent.

A történelmi áttekintésért köszönet Valasek Gábornak.

1. Generáció (1996-1999)

- Nvidia Riva TNT2
- ATI Rage
- 3dfx Voodoo3

- Vertex transzformációkat a CPU végzi
- A kátrya csak textúráz és a Z-buffer kezelést végzi
- Így is rengeteg RAM sávszélesség szabadul fel CPU oldalon
- Az OpenGL az uralkodó grafikus API
- A Glide legalább ekkora támogatást élvez
- A Direct3D ilyenkor a „futottak még” kategóriába esik

2. Generáció (1999-2000)

- Nvidia GeForce 256
- ATI Radeon 7500

- Az OpenGL és a DirectX egyaránt támogatja a hardveresen gyorsított csúcspont transzformációt és az árnyalást
- Egyelőre az API-n keresztül konfigurálható, de nem programozható
- A teljes szerelőszalag a kártyán fut
- Multi-textúra támogatás (bump map, light map)

3. Generáció (2001)

- Nvidia GeForce 3, 4 Ti
- ATI Radeon 8500
- XBox

- A csúcspont transzformáció korlátozott mértékben programozható
- A pixelek árnyalása jobban konfigurálható, de továbbra sem programozható
- 3 dimenziós textúrák
- Többszörös mintavételezés élsimításhoz

4. Generáció (2002)

- Nvidia GeForce FX
- ATI Radeon 9700

- A csúcspont transzformáció és a pixel árnyalás is teljes mértékben programozhatóvá válik
- Magas szintű árnyalási nyelvek megjelenése
  - Nvidia Cg
  - Direct3D HLSL
  - OpenGL GLSL

5. Generáció (2004)

- Nvidia GeForce 6
- ATI Radeon X

- Több puffer egyidejű írása
- PCI-E busz megjelenése
- Hosszabb árnyaló kódok támogatása
- A pixel shaderben megjelenik az elágazás lehetősége
  - fizikai számításokra alkalmassá válik
- SLI (Scan Line Interleave) megjelenése

6. Generáció (2007)

- Nvidia GeForce 7
- ATI Radeon HD2000

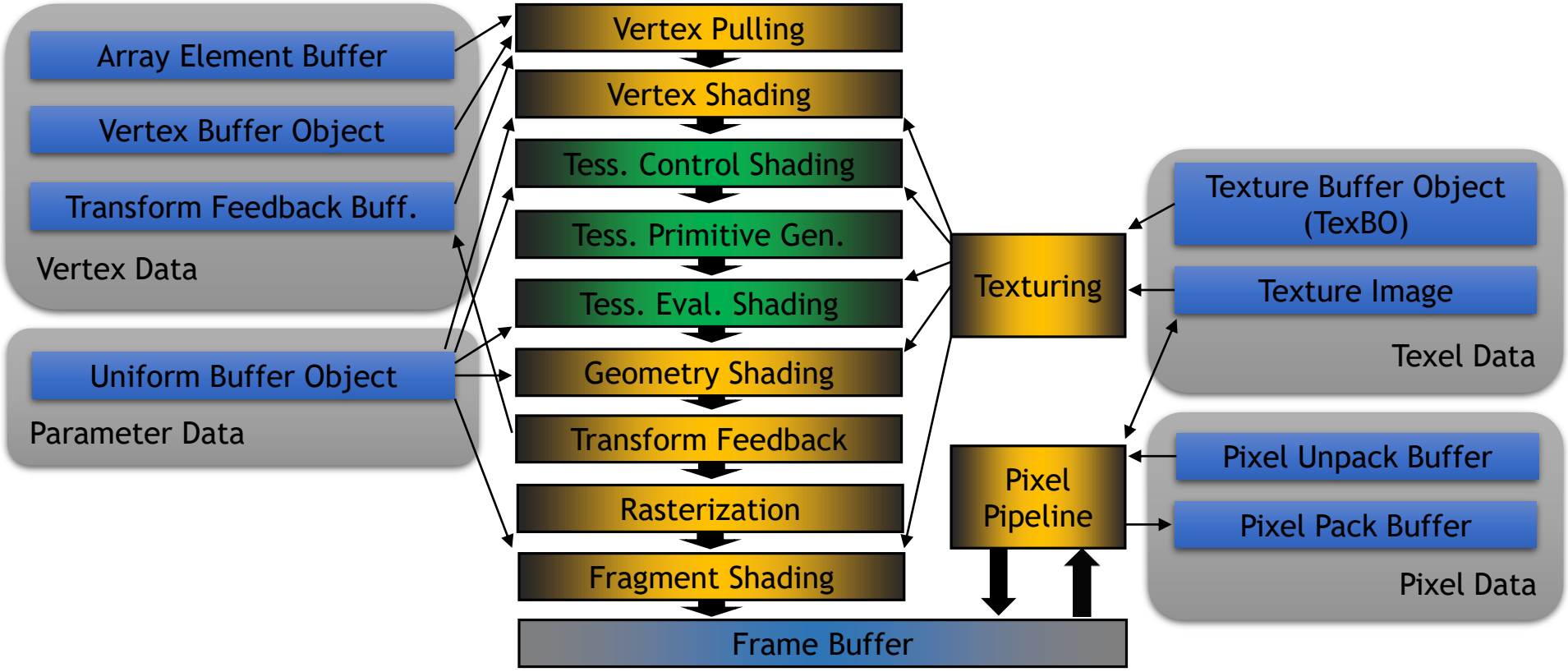
- DirectX 10
- Unified Shader Model megjelenése
- Geometry shader

7. Generáció (2009)

- Nvidia GeForce 400
- ATI Radeon HD5000

- DirectX 11, OpenGL 4.1
- Compute Shader
- Tesselation Shader
- Árnyaló kódok linkelhetősége

OpenGL 4.2 szerelőszalag



Kanonikus inkrementális képszintézishez alakított implicit API

8. Generáció (2016)

- Nvidia GeForce 900 series
- AMD Radeon 385/400
- Intel Gen. 9 (Skylake)

- DirectX 12 (FL_12_X), Vulkan 1.x, Mantle, Metal
- Explicit API-k
- Shader Model 6.0 (DXIL)
- Szabadabb szerelőszalag

Grafikus API-k szintjei

Implicit

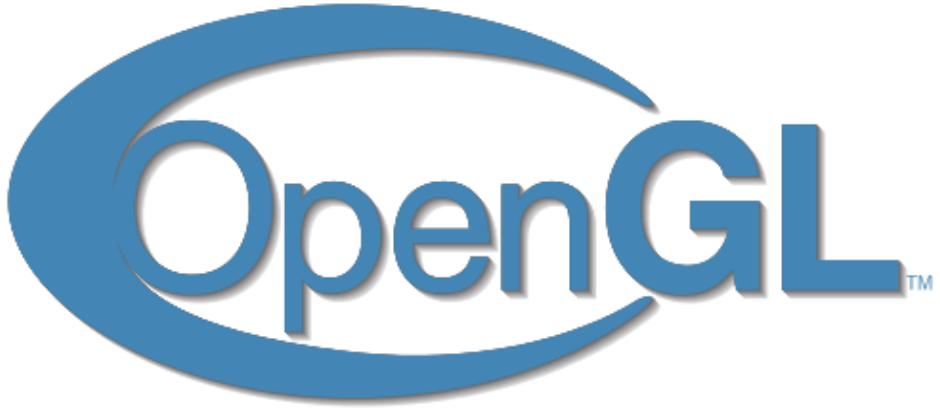
- Szerelőszalagos elrendezés
- Egyetlen támogatott sorrend
- Rengeteg ellenőrzés a meghajtókon belül
  - Késleltetett rajzolás
  - $\mathcal{O}(1000)$ rajzolási parancs
- Alkalmazások meghajtóból finomhangolhatók
  - Double-guessing

Explicit

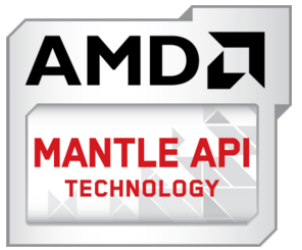
- Szerelőszalagos elrendezés
- „Tetszőleges” sorrend
- Nagyon kevés ellenőrzés a meghajtókon belül
  - Kisebb késleltetés
  - $\mathcal{O}(10000)$ rajzolási parancs
- Tipikusan nincs mód/szükség meghajtókból finomhangolni

Grafikus API-k szintjei

Implicit



Explicit



Rajzolás pseudo kód

```
auto it = std::cbegin(my_scene);

while (it != std::cend(my_scene))
{
  render_state_descriptor state = *(it).m_desc;

  auto next = std::partition(std::par, it, std::cend(m_scene),
    [&](const drawable& obj) { return obj.m_desc == state; });

  gfx::default_device().set_state(state);
  std::for_each(std::par, it, next, draw);

  it = next;
}
```

```
struct drawable {
  vbo m_vbo;
  ibo m_ibo;
  render_state_descriptor m_desc;};
```

Rajzolás pseudo kód

```
auto it = std::cbegin(my_scene);

while (it != std::cend(my_scene))
{
  render_state_descriptor state = *(it).m_desc;

  auto next = std::partition(std::par, it, std::cend(m_scene),
    [&](const drawable& obj) { return obj.m_desc == state; });

  gfx::default_device().set_state(state);
  std::for_each(std::par, it, next, draw);

  it = next;
}
```

```
struct drawable {
  vbo m_vbo;
  ibo m_ibo;
  render_state_descriptor m_desc;};
```

Kiválasztjuk a tömb első elemét, és ha ez nem az utolsó, elkezdünk… rajzolni.

Rajzolás pseudo kód

```
auto it = std::cbegin(my_scene);

while (it != std::cend(my_scene))
{
  render_state_descriptor state = *(it).m_desc;

  auto next = std::partition(std::par, it, std::cend(m_scene),
    [&](const drawable& obj) { return obj.m_desc == state; });

  gfx::default_device().set_state(state);
  std::for_each(std::par, it, next, draw);

  it = next;
}
```

```
struct drawable {
  vbo m_vbo;
  ibo m_ibo;
  render_state_descriptor m_desc;};
```

Kiválasztjuk az első elemhez tartozó rajzolási állapotot

Rajzolás pseudo kód

```
struct drawable {
  vbo m_vbo;
  ibo m_ibo;
  render_state_descriptor m_desc;};
```

```
auto it = std::cbegin(my_scene);

while (it != std::cend(my_scene))
{
  render_state_descriptor state = *(it).m_desc;

  auto next = std::partition(std::par, it, std::cend(m_scene),
    [&](const drawable& obj) { return obj.m_desc == state; });

  gfx::default_device().set_state(state);
  std::for_each(std::par, it, next, draw);

  it = next;
}
```

CPU párhuzamoson a jelenet elemeinek tömbjében az aktuális elem mellé rendezzük az azonos rajzolási állapotot igénylőket.

Rajzolás pseudo kód

```
auto it = std::cbegin(my_scene);

while (it != std::cend(my_scene))
{
  render_state_descriptor state = *(it).m_desc;

  auto next = std::partition(std::par, it, std::cend(m_scene),
    [&](const drawable& obj) { return obj.m_desc == state; });

  gfx::default_device().set_state(state);
  std::for_each(std::par, it, next, draw);

  it = next;
}
```

```
struct drawable {
  vbo m_vbo;
  ibo m_ibo;
  render_state_descriptor m_desc;};
```

Eztán beállítjuk a rajzoló eszköznek a kívánt állapotot. **EZ** a költséges!

Rajzolás pseudo kód

```
auto it = std::cbegin(my_scene);

while (it != std::cend(my_scene))
{
  render_state_descriptor state = *(it).m_desc;

  auto next = std::partition(std::par, it, std::cend(m_scene),
    [&](const drawable& obj) { return obj.m_desc == state; });

  gfx::default_device().set_state(state);
  std::for_each(std::par, it, next, draw);

  it = next;
}
```

```
struct drawable {
  vbo m_vbo;
  ibo m_ibo;
  render_state_descriptor m_desc;};
```

Majd a gépünk összes magját használva etetjük az eszközt rajzolási parancsokkal

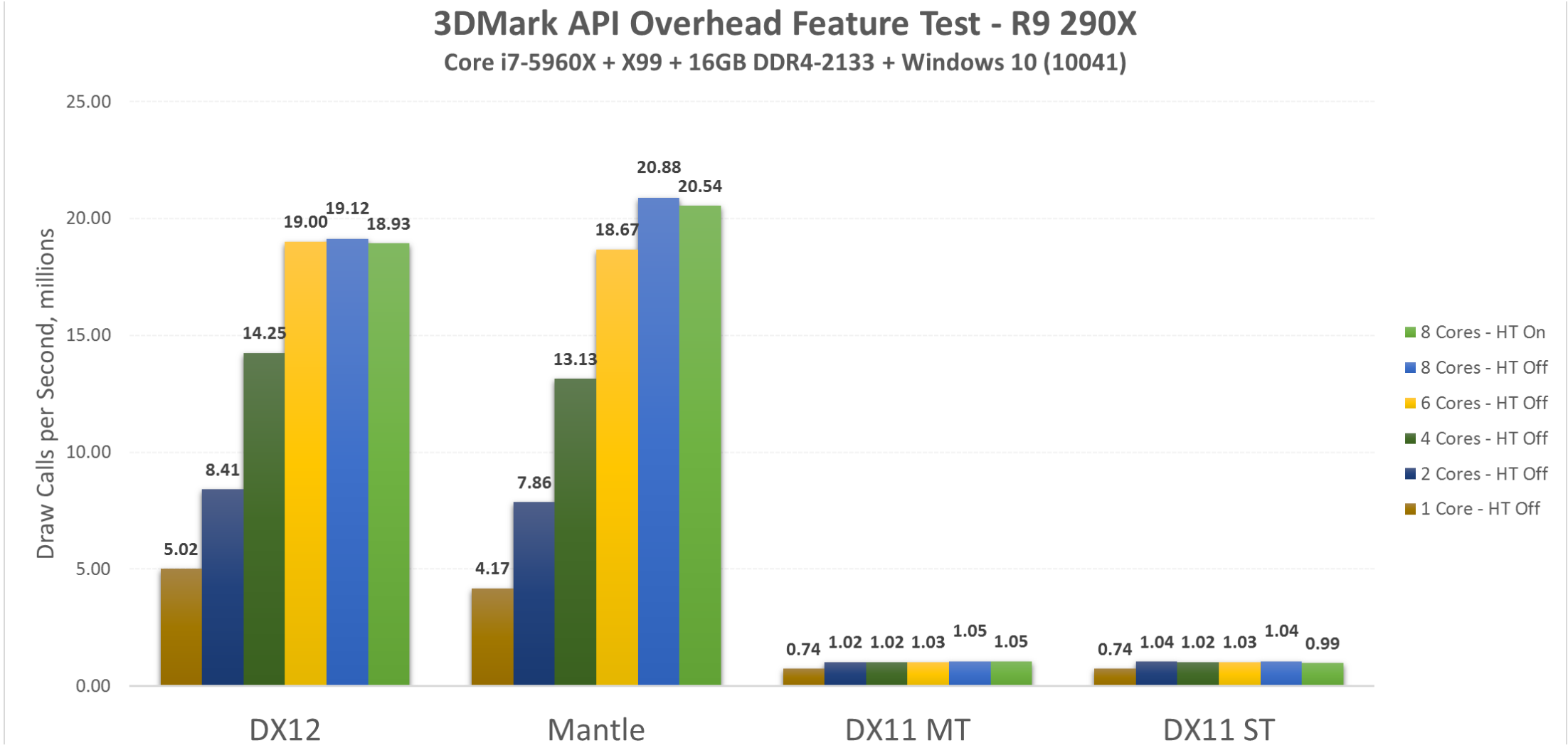
Rajzolás pseudo kód

```
auto it = std::cbegin(my_scene);

while (it != std::cend(my_scene))
{
  render_state_descriptor state = *(it).m_desc;

  auto next = std::partition(std::par, it, std::cend(m_scene),
    [&](const drawable& obj) { return obj.m_desc == state; });

  gfx::default_device().set_state(state);
  std::for_each(std::par, it, next, draw);

  it = next;
}
```

```
struct drawable {
  vbo m_vbo;
  ibo m_ibo;
  render_state_descriptor m_desc;};
```

Implicit/explicit API-k esetében eltérő mennyiségű ellenőrzést végeznek a meghajtók

Mennyi az annyi?



Forrás: https://www.pcper.com/image/view/60156?return=node%2F63528

Megjelenítő kiszolgálók és ablakozó rendszerek

…az utolsó szó jogán.

Mik is ezek pontosan?

- Az ablakozó rendszer vezérli az asztalon megjelenő tartalmat
  - Tudja mi az az ablak, tudja melyik fogad billentyű parancsokat, merre jár az egér, vannak beépített GUI elemek (csúszka, fejléc, X-gomb, stb.)
  - Ha egy ablakot kitakar egy másik, tudja hogy azt a részt nem kell kirajzolni; el is tud altatni programokat
- A megjelenítő kiszolgáló felelős biztosítani a kapcsolatot a videokártya meghajtója és az ablakozó között
  - Interfész, amibe az ablakozó tud bekapcsolódni
  - Meg tudja mondani hány monitor van a gépre kötve, azok milyen felbontásúak, hány DPI-sek (Dots Per Inch), (általában) ő továbbítja a bemeneti eszközök jeleit
- Ideális esetben az interfészek az egységek között úgy vannak kitalálva, hogy azok a lehető legjobban passzoljanak.

Windows

- GUI-nehéz operációs rendszer, ez a része nagyon jól össze van rakva
- A megjelenítő kiszolgáló és az ablakozó rendszer egybe van gyúrva
  - Jó, mert nagyon jól illeszkedik a kettő (ugyanaz találta ki őket)
  - Rossz, mert nem lehet más GUI-t csinálni (node mégis ki csinálna?)
- Mindkettő funkcionalitását a WinAPI látja el. (OS API!)
- Az appok által használt WinRT (Windows Runtime) nem ad ilyen alacsony szintű elérést

Windows Display Driver Module

- A meghajtó és az OS API között a WDDM (Windows Display Driver Model) található. Ezt az interfészt kell a meghajtóknak implementálni.
- Az újabb verziói egyre okosabbak
  - WDDM 1.0 (Win Vista): az alapok támogatása (Alt+Tab támogatás, memória koherencia)
  - WDDM 1.1 (Win 7): 2D gyorsítás, több-meghajtó támogatás
  - WDDM 1.2 (Win 8): pre-emptive multi-task, gyors kontextus váltás
  - WDDM 1.3 (Win 8): Miracast
  - WDDM 2.0 (Win 10): DirectX 12, kevesebb kernel-mode
  - WDDM 2.1 (Anniversary Update): SM 6.0, HDR (High Dynamic Range)
  - WDDM 2.2 (Creators Update): VR/AR (Virtuális és kevert valóság)

Linux

- Hát… Linuxon nem ilyen rózsás a helyzet
- Akit humoros formában érdekel a tragikomikum az élet pingvines oldalán, annak ajánlom figyelmébe Bryan Lunduke munkásságát (Linux sucks)
- Linuxon annyi meghajtó létezik, mint égen a csillag
  - Zárt forrású, nyílt forrású, félig zárt, félig nyílt, félig meghajtó félig bit kupac…

Linux – megjelenítő kiszolgálók

- X
  - A dinoszauruszokkal karöltve is járta már a földet, csak sajnos nem pusztult ki velük együtt
  - Részben az ablakozó feladatait is ellátja (igen, rossz értelemben)
  - Kliens-kiszolgáló felépítéssel bír
- Wayland
  - A linuxos közösség úgy várja, akár a messiást
  - Modern felépítés
- MIR
  - A Canonical kezdeményezés (leállították)
  - Megjelenítő kiszolgáló telefonokba, tabletekbe, asztali gépekbe és HPC kiszolgálókba

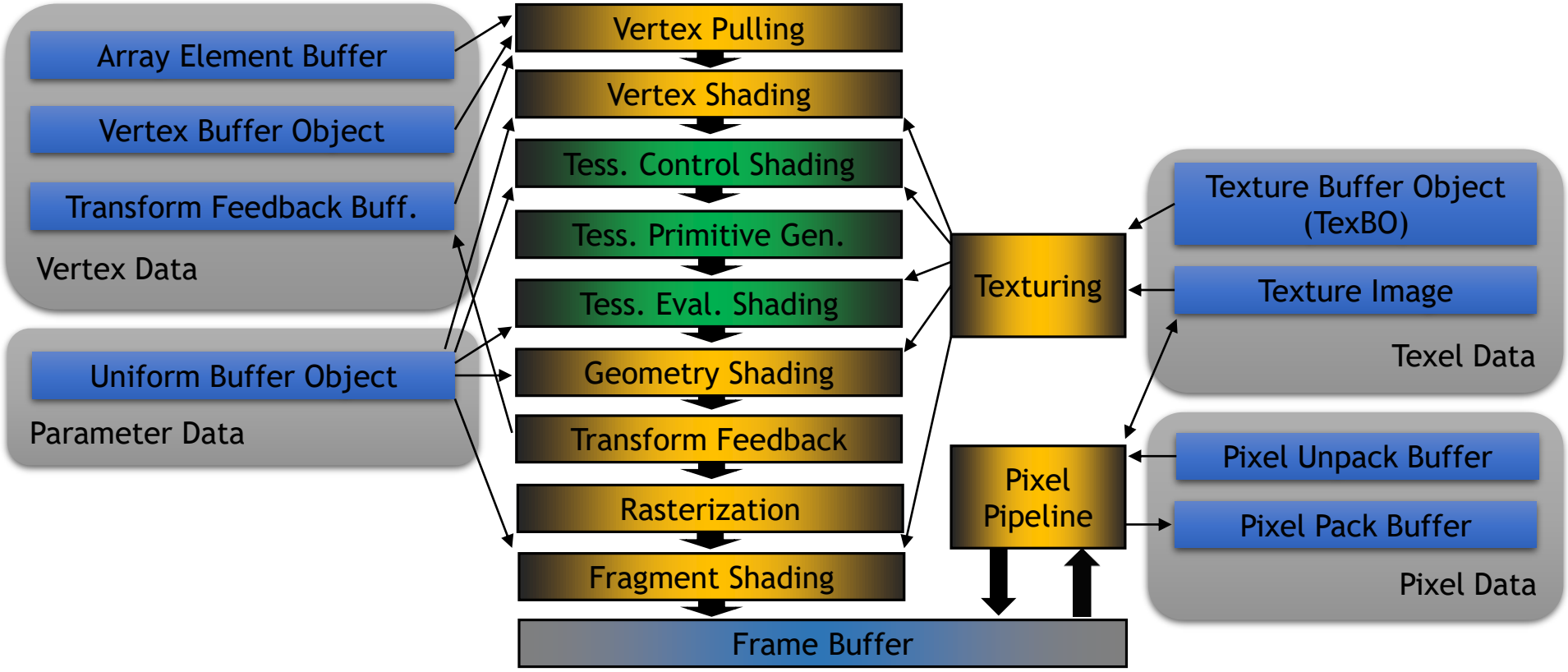
Linux – ablakozó rendszerek

- KDE, Gnome, Unity
  - Jelenleg X-re épülő ablakozó rendszerek
- Weston
  - A Wayland prototípus ablakozó rendszere
  - Elsősorban teszt célokat szolgál, nem valódi használatra termett
- Unity 8
  - A MIR-hez passzoló ablakozó rendszer

Linux – WDDM?

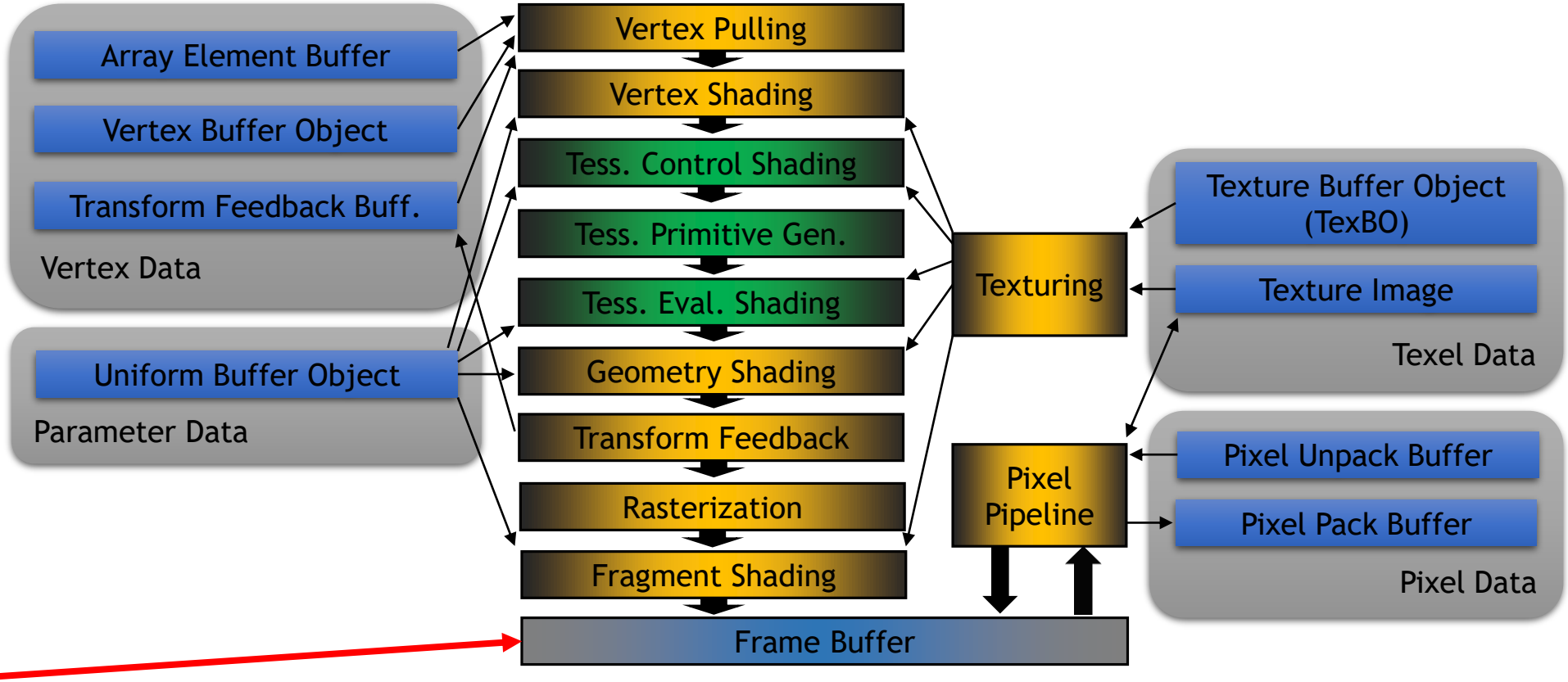
- A meghajtók direkten kötődnek be a megjelenítő kiszolgálókba
- Az interfész maga a kiszolgáló (nincs bináris kompatibilitás)
- Khronos EGL
  - Egységessé teszi a felhasználó felé az erőforrások enumerálását
  - Lehetőséget biztosít, hogy egy uniform felületen keresztül propagálja az erőforrásait a megjelenítő kiszolgáló felé
  - Eddig egyedül az Nvidia meghajtó támogatja

OpenGL 4.2 szerelőszalag



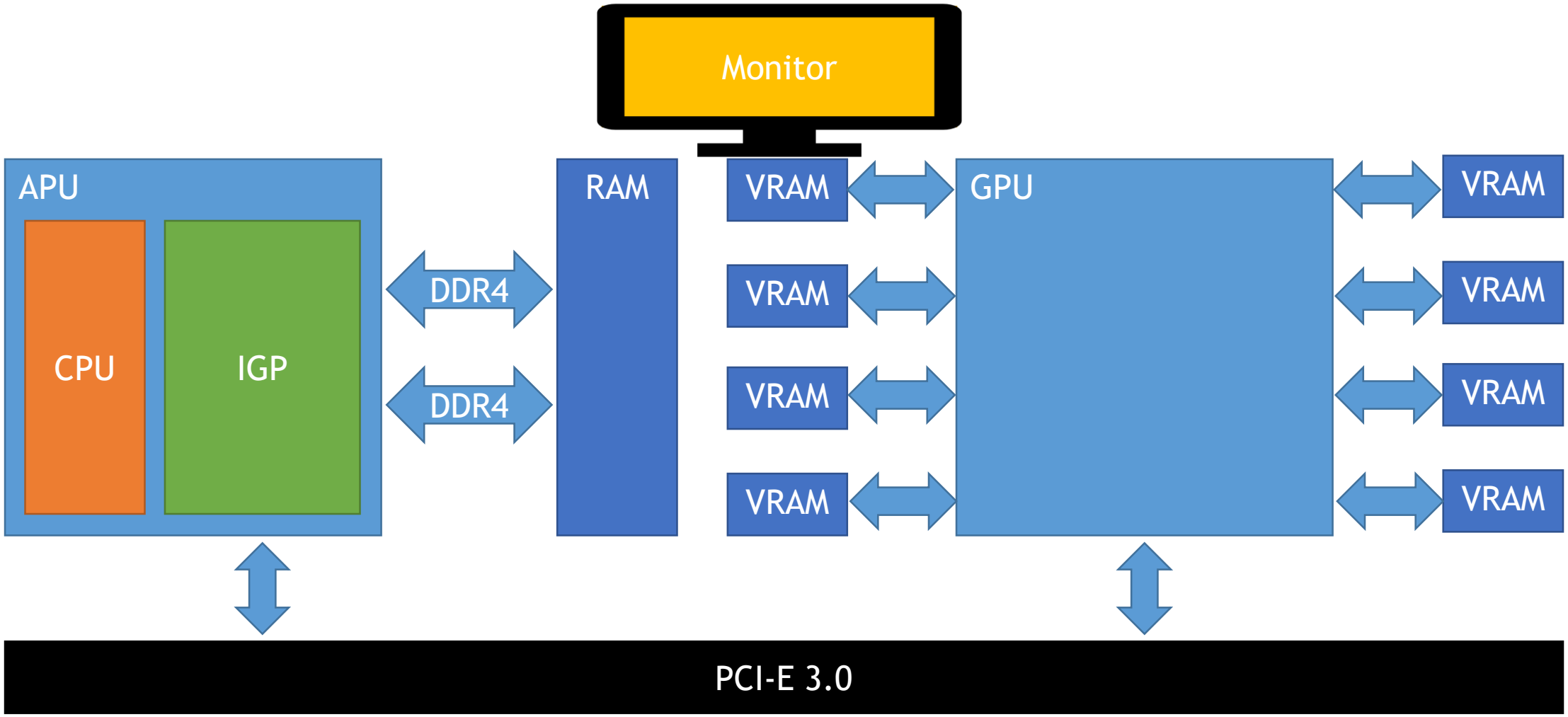
Kanonikus inkrementális képszintézishez alakított implicit API

OpenGL 4.2 szerelőszalag



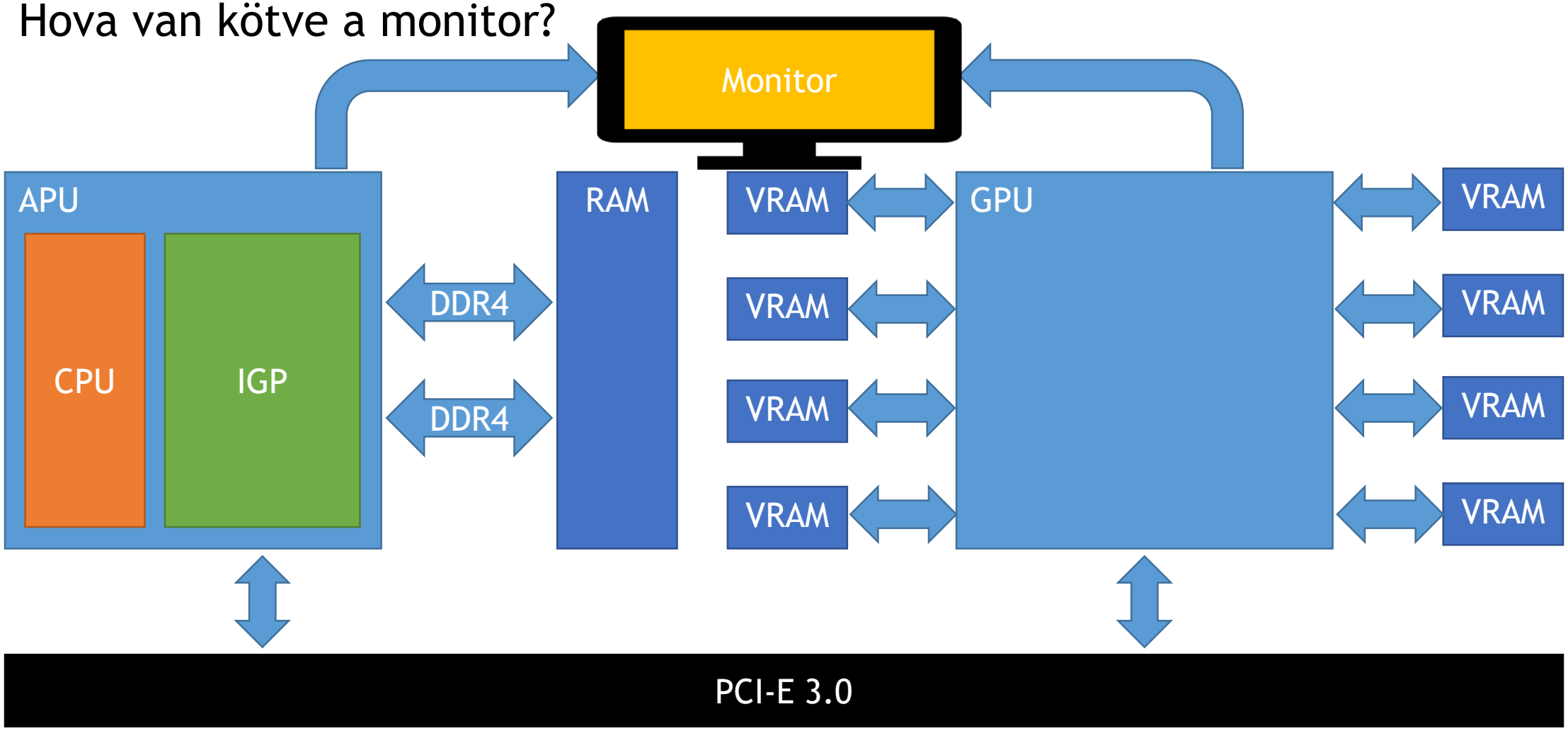
Az egész szakasz erről szólt. Ez a memória terület nem a mi programunkhoz tartozik, hanem kölcsön kapjuk az ablakozótól. Végső soron ő jeleníti meg.

Tipikus notebook

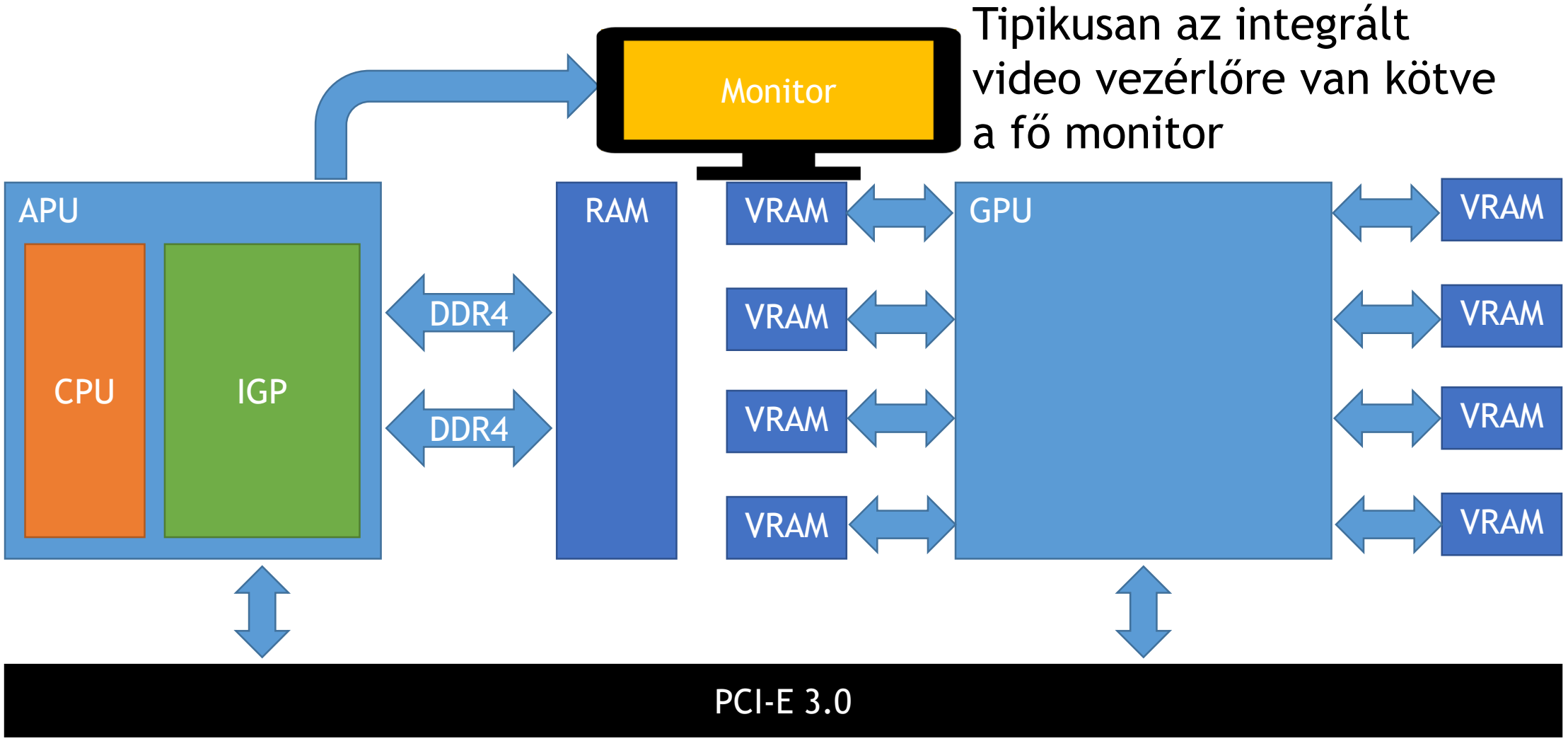


Tipikus notebook

Hova van kötve a monitor?

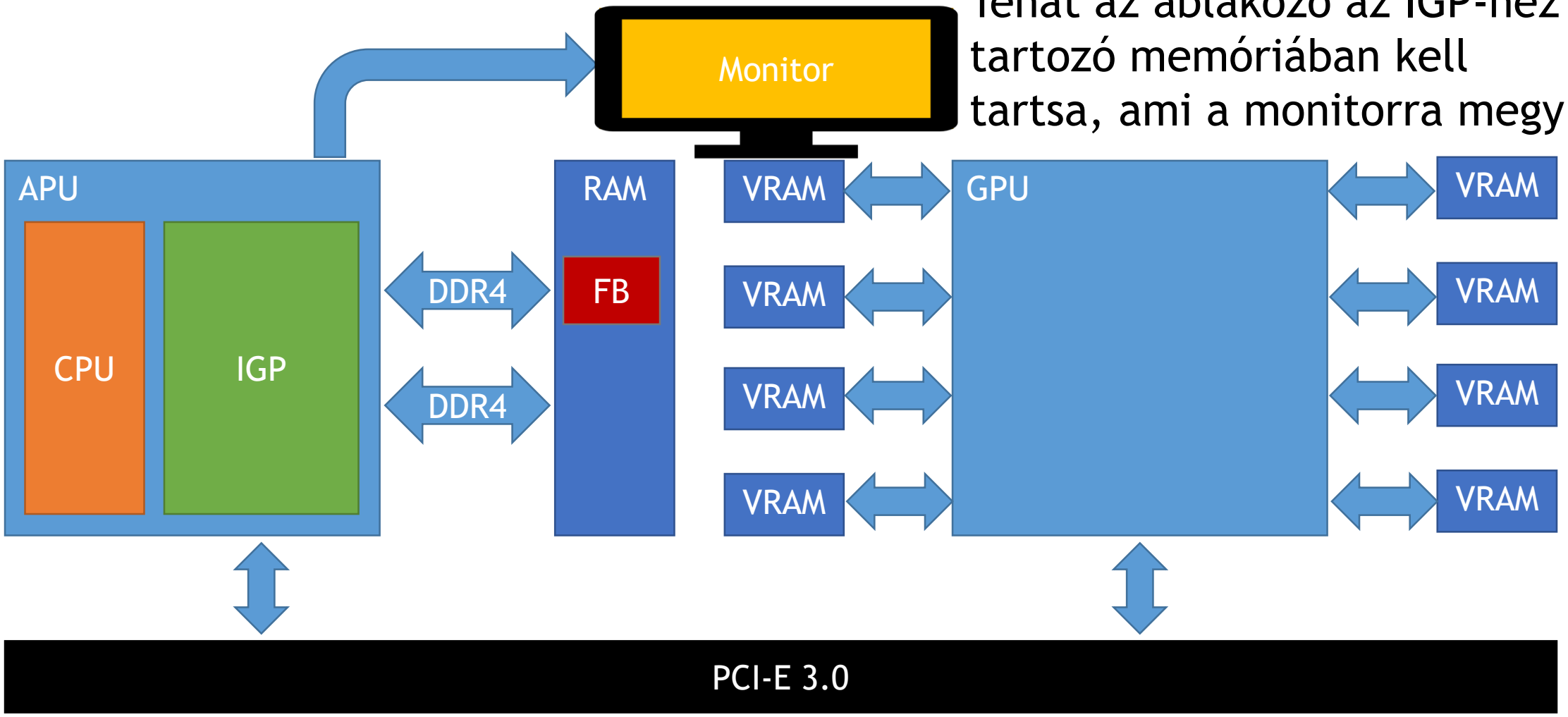


Tipikus notebook



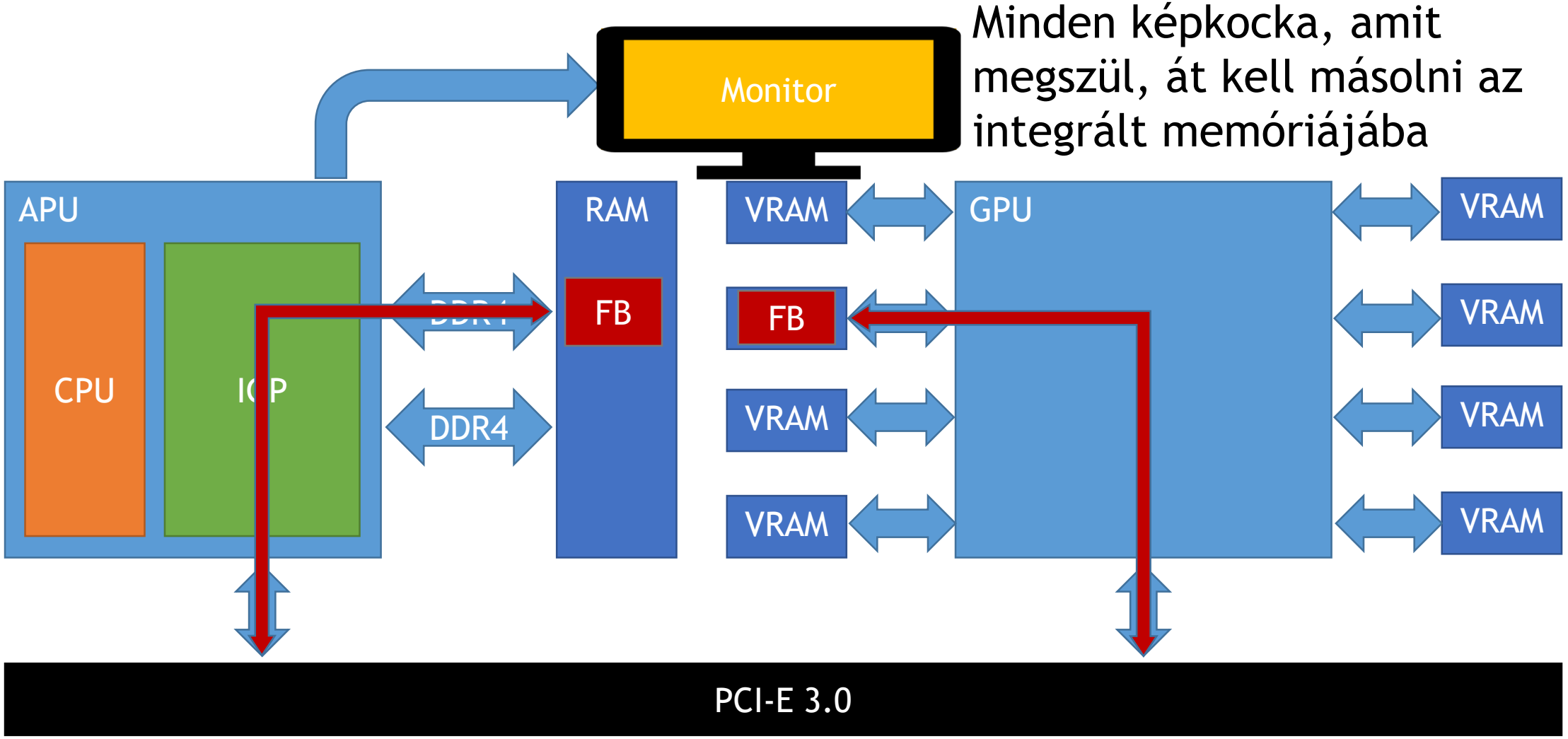
Tipikusan az integrált video vezérlőre van kötve a fő monitor

Tipikus notebook



Tehát az ablakozó az IGP-hez tartozó memóriában kell tartsa, ami a monitorra megy

Tipikus notebook



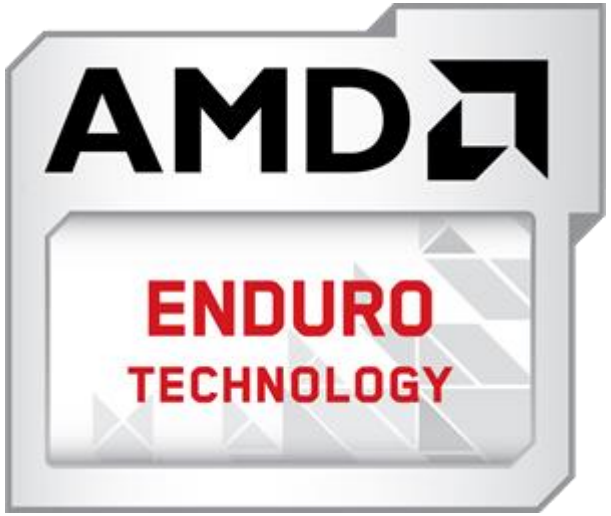
Minden képkocka, amit megszül, át kell másolni az integrált memóriájába

Szoftveres váltás

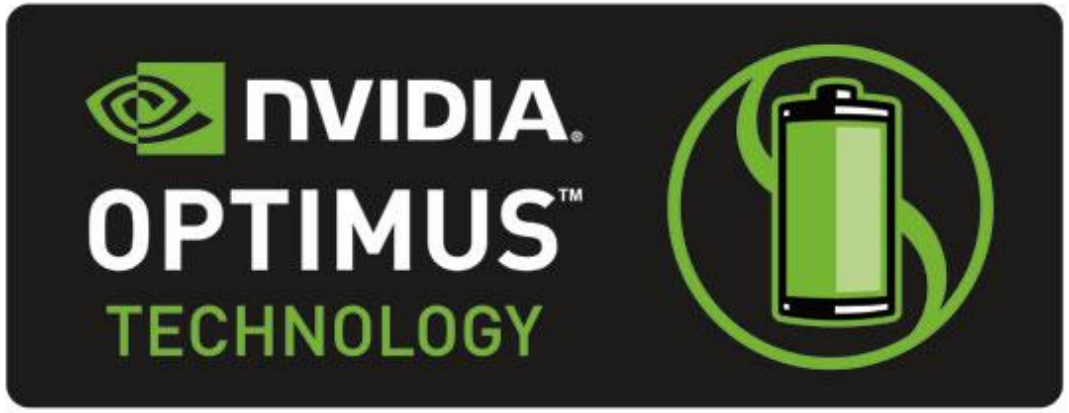
- Az integrált és dedikált video vezérlők között nem csak az elkészült kép kockát kell átmásolni
- Esetenként a program teljes video memóriáját át kell tudni migrálni
  - Táp csatlakozóról játszunk, és kihúzzuk a konnektorból a laptopot
  - Energia séma alapján átsorolja a meghajtó a játékunkat az integrált vezérlőre, hogy energiát takarítson meg
  - A játék nem áll meg, csak lelassul

Szoftveres váltás

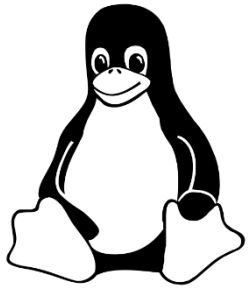
Intel/AMD+AMD



Intel+Nvidia



Mi rossz történhet?

NVIDIA OPTIMUS™ TECHNOLOGY +  + NVIDIA® CUDA®

OpenCL

OpenGL dióhéjban

Open Graphics Library - Miért jó?

- Hordozható grafikus könyvtár
- Tudományos életben mondhatni az egyetlen elfogadott
- Szerelőszalag (pipeline) felépítésű
- Egy adott képalkotási módszert szolgál ki
  - Cserébe az egyszerű és széleskörűen dokumentált
- Képes együttműködni több compute API-val
  - CUDA, OpenCL, C++AMP

Open Graphics Library - Miért rossz?

- Visszafele kompatibilis a legrégebbi verziókig
- Archaikus felépítésű
- Önmaga miatt Google-immunis
- Nincs saját köztes nyelve
- Aluldefiniált az árnyaló nyelve
- Aluldefiniált a gazda oldali API

Véges állapotú gép

- Az OpenCL-ből megszokott context entitás itt is létezik
- Minden OpenGL utasítás akkor érvényes, ha van aktív context a szálban
- Az OpenGL-nek nincs saját context típusa
  - Létrehozni csak az ablakkezelő tudja
  - Aktiválni is
- Egy szálban egyszerre csak egy context lehet aktív
- Egy context egyszerre csak egy szálban lehet aktív
- A hiba kód egyetlen rejtett globális változó
- Dokumentáció

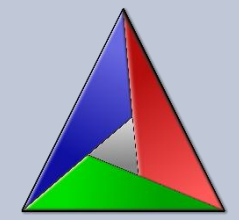
OpenGL madártávlatból

- A state machine-jelleg miatt tipikus használati sémák alakulnak ki
- Bind-modell használata
  - context.setActive(true);
  - Do some OpenGL stuff
  - context.setActive(false);
- A hibák csakis és kizárólag az elkövetésükkor detektálhatók.
  - GLint glGetError();
- Alapvetően C API, és abból is NAGYON gyengén típusos
  - A beépített típusokon túl egyetlen új sincs
  - Minden handle GLint

Hello triangle!

Rajzoljunk ki egyetlen színes háromszöget!

<Subroutine Name=„Cmake”/>



- CMake = Cross-platform Make
- Egy „meta” makefile, amiből számos más, natív makefile generálható
- Több forrásnyelvet támogat: Fortran, C, C++ beépítve, MINDEN más kiegészítés
- Függőségeket automatikusan megtalálja
- Társ szoftverek kiegészítik a képességeit:
  - CTest, amivel unit testek írhatók
  - CPack, amivel platform-specifikus telepítők készíthetők (RPM, DEB, EXE, stb.)
  - CDash, amivel a fordítás/tesztelés/telepítés helyessége tesztelhető
- A script nyelve nem a legkényelmesebb (bár human readable), de hosszú távon kifizetődő.

Qt

- Átfogó megoldás cross-platform programok fejlesztésére
- C++ könyvtár nyelvi kiegészítésekkel és kódgenerálással
- Fordítását „megkönnyítendő” saját make nyelve is van (QMake)
- Nem kötelező használni.  támogatás is van.
- Kis projektekhez „overkill” tud lenni
- Az egyszerű dolgokat sem feltétlen könnyű megcsinálni
- Cserébe bónuszként az ember megkap RENGETEG más mindent is ingyen
- Piaci értéke van a Qt ismeretnek

Simple Fast Multimedia Lib.

- Egyszerű ablakkezelő, input, audio és OpenGL context kezeléssel
- Számtalan nyelvhez létezik API, többek között C++ is
- Fordításhoz biztosítottak  scriptek
- Kis projektekhez ideális tud lenni
- Már csak OpenGL viszonylatban sem átfogó megoldás
  - Más könyvtáraktól is függni kell
  - Cserébe azok olyanok lehetnek, amik nekünk szimpatikusak

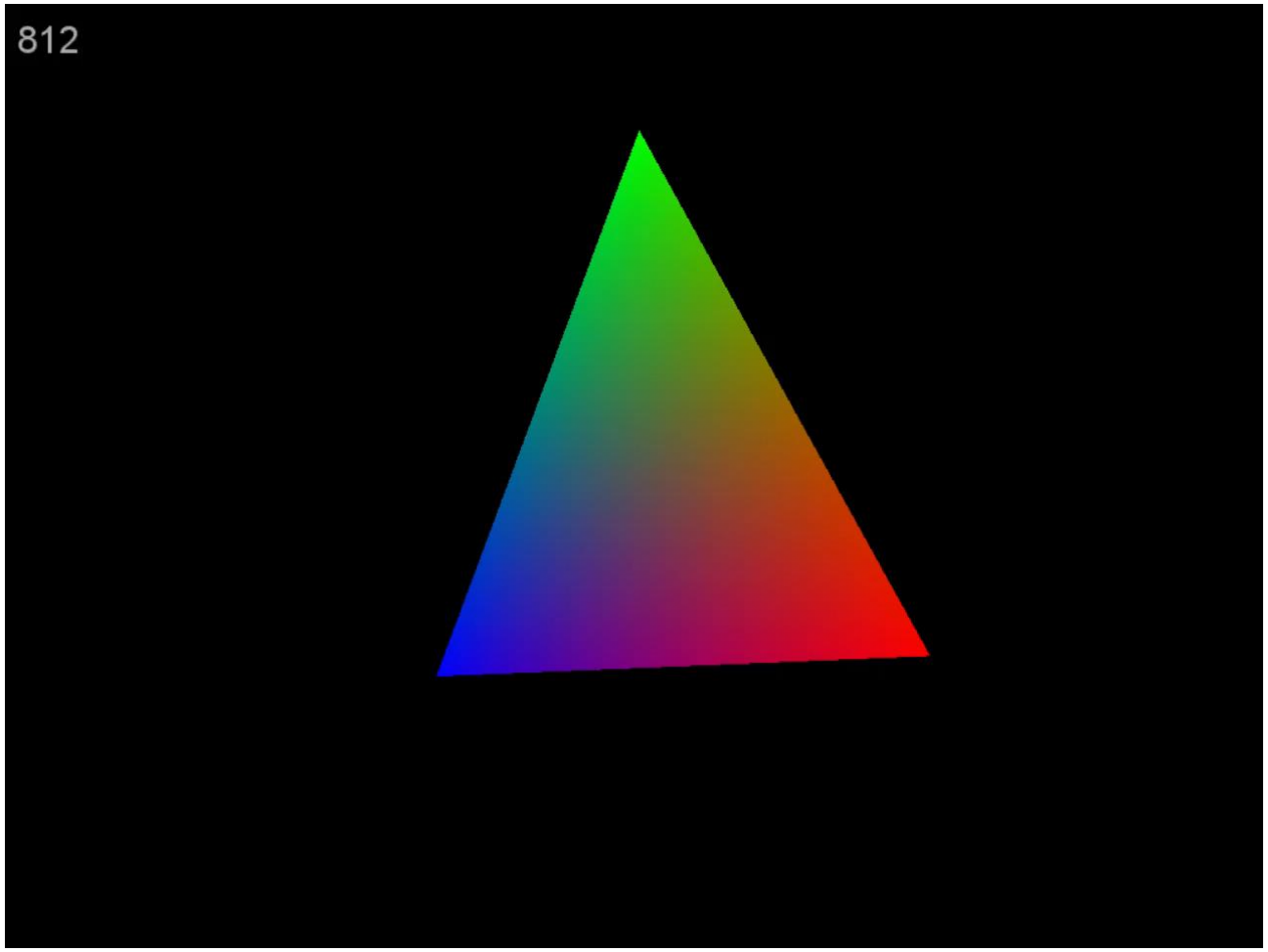
Simple Directmedia Layer

- Egyszerű ablakkezelő, input, audio és OpenGL / DirectX context kezeléssel
- Számtalan nyelvhez létezik API, többek között C++ is
- Fordításhoz biztosítottak  scriptek
- Kis projektekhez ideális tud lenni
- Már csak OpenGL viszonylatban sem átfogó megoldás
  - Más könyvtáraktól is függni kell
  - Cserébe azok olyanok lehetnek, amik nekünk szimpatikusak

GL Mathematics Library

- GLSL-ben az ember használhat mátrixokat és vektorokat
- Gazda oldalon azonban semmilyen hasonló nincsen
- C++ sablon könyvtár (header only)
- Pokolian gyors
- Fordításhoz biztosítottak  scriptek
- Kis projektekhez ideális tud lenni
- Ez az egyik olyan rész könyvtár, amiből megéri ezt választani

Hello, triangle!



- OpenGL 3.3 Core
  - Vertex árnyaló
  - Fragment árnyaló
- SFML

GUI = eseményvezérlés

- Konzolos programnál megszoktuk, hogy van belépési pont, és egy vagy több helyen véget tud érni a programunk
  - A vezérlés szekvenciálisan halad előre
- Grafikus alkalmazásnál, viszont *reagálni* szeretnénk bemeneti eszközökre, akár miközben a program éli saját kis életét
  - Időszakosan kiváltódnak események, amik megváltoztatják a vezérlés menetét
- A C/C++, mint nyelv kevéssé alkalmas az ilyen struktúrák átlátható megfogalmazására
  - Szerencsére mi nem operációs rendszert írunk, csak egy darab ablakot szeretnénk, amibe rajzolunk

Madártávlatból

```
int main() {
  // Init

  while (window.isOpen()) // Application loop
  {
    sf::Event event;
    while (window.pollEvent(event)) {
      // Handle all event types
    }

    // Update scene and draw
  }
  // Cleanup
  return EXIT_SUCCESS;
}
```

Madártávlatból

Minden OpenGL erőforrást itt fogunk létrehozni, az árnyalókat is itt fogjuk lefordítani.

```
int main() {
  // Init

  while (window.isOpen()) // Application loop
  {
    sf::Event event;
    while (window.pollEvent(event)) {
      // Handle all event types
    }

    // Update scene and draw
  }
  // Cleanup
  return EXIT_SUCCESS;
}
```

Madártávlatból

```
int main() {
  // Init

  while (window.isOpen()) // Application loop
  {
    sf::Event event;
    while (window.pollEvent(event)) {
      // Handle all event types
    }

    // Update scene and draw
  }
  // Cleanup
  return EXIT_SUCCESS;
}
```

Ez az ablakunk esemény ciklusa: amíg a fő ablak nincs bezárva, addig nem szabadul a program innen.

Több ablakozó API-ban ez a „végtelen” ciklus el van rejtve a programozó elől, és pl. csak egy virtuális függvényt kap, amit implementálva lekezelheti az érkező eseményeket.

Madártávlatból

```
int main() {
  // Init

  while (window.isOpen()) // Application loop
  {
    sf::Event event;
    while (window.pollEvent(event)) {
      // Handle all event types
    }

    // Update scene and draw
  }
  // Cleanup
  return EXIT_SUCCESS;
}
```

A két rajzolás között beérkezett eseményeket itt kezeljük le.

Itt tipikusan OpenGL erőforrásokat forgunk módosítani (más különben nem látnánk a hatásukat), pl. egér mozgásra módosítjuk a kamera nézeti mátrixát.

Madártávlatból

```
int main() {
  // Init

  while (window.isOpen()) // Application loop
  {
    sf::Event event;
    while (window.pollEvent(event)) {
      // Handle all event types
    }

    // Update scene and draw
  }
  // Cleanup
  return EXIT_SUCCESS;
}
```

Tipikusan itt rajzol az ember és kiszámol mindent, ami ehhez (vagy a következő) képkockához kell.

Madártávlatból

Erőforrások felszabadítása. Egy valamire való RAII csomagolóanyag feleslegessé teszi ezt a részt.

```
int main() {
  // Init

  while (window.isOpen()) // Application loop
  {
    sf::Event event;
    while (window.pollEvent(event)) {
      // Handle all event types
    }

    // Update scene and draw
  }
  // Cleanup
  return EXIT_SUCCESS;
}
```

Ablak és kontextus létrehozás

```
sf::RenderWindow window(sf::VideoMode(1024, 768),
                        "SFML / OpenGL test",
                        sf::Style::Default,
                        sf::ContextSettings(32));

// OpenGL kontext verzio ellenorzese
if ( sf::Uint32(window.getSettings().majorVersion * 10 +
                window.getSettings().majorVersion) < 33 )
{
  std::cerr << "Highest OpenGL version is " <<
    window.getSettings().majorVersion << "." <<
    window.getSettings().minorVersion << " Exiting..." <<
    std::endl;
  std::exit(EXIT_FAILURE);
}
```

Ablak és kontextus létrehozás

Az ablak felbontása

```
sf::RenderWindow window(sf::VideoMode(1024, 768),
                        "SFML / OpenGL test",
                        sf::Style::Default,
                        sf::ContextSettings(32));

// OpenGL kontext verzio ellenorzese
if ( sf::Uint32(window.getSettings().majorVersion * 10 +
                window.getSettings().majorVersion) < 33 )
{
  std::cerr << "Highest OpenGL version is " <<
    window.getSettings().majorVersion << "." <<
    window.getSettings().minorVersion << " Exiting..." <<
    std::endl;
  std::exit(EXIT_FAILURE);
}
```

Ablak és kontextus létrehozás

Az ablak címe

```
sf::RenderWindow window(sf::VideoMode(1024, 768),
                        "SFML / OpenGL test",
                        sf::Style::Default,
                        sf::ContextSettings(32));

// OpenGL kontext verzio ellenorzese
if ( sf::Uint32(window.getSettings().majorVersion * 10 +
                window.getSettings().majorVersion) < 33 )
{
  std::cerr << "Highest OpenGL version is " <<
    window.getSettings().majorVersion << "." <<
    window.getSettings().minorVersion << " Exiting..." <<
    std::endl;
  std::exit(EXIT_FAILURE);
}
```

Ablak és kontextus létrehozás

Az fejlécének stílusa

```
sf::RenderWindow window(sf::VideoMode(1024, 768),
                        "SFML / OpenGL test",
                        sf::Style::Default,
                        sf::ContextSettings(32));

// OpenGL kontext verzio ellenorzese
if ( sf::Uint32(window.getSettings().majorVersion * 10 +
                window.getSettings().majorVersion) < 33 )
{
  std::cerr << "Highest OpenGL version is " <<
    window.getSettings().majorVersion << "." <<
    window.getSettings().minorVersion << " Exiting..." <<
    std::endl;
  std::exit(EXIT_FAILURE);
}
```

Ablak és kontextus létrehozás

```
sf::RenderWindow window(sf::VideoMode(1024, 768),
                        "SFML / OpenGL test",
                        sf::Style::Default,
                        sf::ContextSettings(32));

// OpenGL kontext verzio ellenorzese
if ( sf::Uint32(window.getSettings().majorVersion * 10 +
                window.getSettings().majorVersion) < 33 )
{
  std::cerr << "Highest OpenGL version is " <<
    window.getSettings().majorVersion << "." <<
    window.getSettings().minorVersion << " Exiting..." <<
    std::endl;
  std::exit(EXIT_FAILURE);
}
```

Itt állítható a mélységi- avagy Z-buffer, a stencil buffer bit szélessége, az anti-aliasing módja és a kért OpenGL verzió.

Ablak és kontextus létrehozás

Legalább OpenGL 3.3-as kontextust akarunk kell

```
sf::RenderWindow window(sf::VideoMode(1024, 768),
                        "SFML / OpenGL test",
                        sf::Style::Default,
                        sf::ContextSettings(32));

// OpenGL kontext verzio ellenorzese
if ( sf::Uint32(window.getSettings().majorVersion * 10 +
                window.getSettings().majorVersion) < 33 )
{
  std::cerr << "Highest OpenGL version is " <<
    window.getSettings().majorVersion << "." <<
    window.getSettings().minorVersion << " Exiting..." <<
    std::endl;
  std::exit(EXIT_FAILURE);
}
```

Ablak és kontextus létrehozás

```
// GLEW inicializalas
if (glewInit() != GLEW_OK) std::exit(EXIT_FAILURE);

// Arnyalok betoltese fajlbol, forditasa es linkelese
loadShaders();

// Program hasznalata
glUseProgram(glProgram); checkError("glUseProgram");
```

Ablak és kontextus létrehozás

OpenGL-ben futásidőben kell betölteni az összes API függvényt. Ezt teszi meg helyettünk a GLEW (GL Extension Wrangler)

```
// GLEW inicializalas
if (glewInit() != GLEW_OK) std::exit(EXIT_FAILURE);

// Arnyalok betoltese fajlbol, forditasa es linkelese
loadShaders();

// Program hasznalata
glUseProgram(glProgram); checkError("glUseProgram");
```

Ablak és kontextus létrehozás

Az árnyaló kódokat ugyanúgy futásidőben töltjük be és fordítjuk le, mint OpenCL esetében a kerneleket.

```
// GLEW inicializalas
if (glewInit() != GLEW_OK) std::exit(EXIT_FAILURE);

// Arnyalok betoltese fajlbol, forditasa es linkelese
loadShaders();

// Program hasznalata
glUseProgram(glProgram); checkError("glUseProgram");
```

Ablak és kontextus létrehozás

Az OpenGL bind-modelljének megfelelően jelezzük, hogy eztán az előbb betöltött és lefordított árnyalókat fogjuk használni. (Csak ez az egy lesz)

```
// GLEW inicializalas
if (glewInit() != GLEW_OK) std::exit(EXIT_FAILURE);

// Arnyalok betoltese fajlbol, forditasa es linkelese
loadShaders();

// Program hasznalata
glUseProgram(glProgram); checkError("glUseProgram");
```

Árnyalók betöltése

```
std::basic_ifstream<GLchar> vs_file(VERTEX_SHADER_PATH);

std::basic_string<GLchar> vs_string(
  std::istreambuf_iterator<GLchar>(vs_file),
 (std::istreambuf_iterator<GLchar>()));
std::vector<const GLchar*> vs_c_strings{ vs_string.c_str() };

vertexShaderObj = glCreateShader(GL_VERTEX_SHADER);

glShaderSource(vertexShaderObj,
               (GLsizei)vs_c_strings.size(),
               vs_c_strings.data(), NULL);
glCompileShader(vertexShaderObj);
glGetShaderiv(vertexShaderObj, GL_COMPILE_STATUS, &GL_err);
```

Árnyalók betöltése

```
std::basic_ifstream<GLchar> vs_file(VERTEX_SHADER_PATH);

std::basic_string<GLchar> vs_string(
  std::istreambuf_iterator<GLchar>(vs_file),
 (std::istreambuf_iterator<GLchar>()));
std::vector<const GLchar*> vs_c_strings{ vs_string.c_str() };

vertexShaderObj = glCreateShader(GL_VERTEX_SHADER);

glShaderSource(vertexShaderObj,
               (GLsizei)vs_c_strings.size(),
               vs_c_strings.data(), NULL);
glCompileShader(vertexShaderObj);
glGetShaderiv(vertexShaderObj, GL_COMPILE_STATUS, &GL_err);
```

Az API-nak saját karakter típusa van. Castolgatás helyett inkább használjuk. (Igazából typedef char;)

Árnyalók betöltése

```
std::basic_ifstream<GLchar> vs_file(VERTEX_SHADER_PATH);

std::basic_string<GLchar> vs_string(
  std::istreambuf_iterator<GLchar>(vs_file),
 (std::istreambuf_iterator<GLchar>()));
std::vector<const GLchar*> vs_c_strings{ vs_string.c_str() };

vertexShaderObj = glCreateShader(GL_VERTEX_SHADER);

glShaderSource(vertexShaderObj,
               (GLsizei)vs_c_strings.size(),
               vs_c_strings.data(), NULL);
glCompileShader(vertexShaderObj);
glGetShaderiv(vertexShaderObj, GL_COMPILE_STATUS, &GL_err);
```

String iterátor CTOR. Extra zárójel most vexing parse miatt kell.

C++11 óta uniform initialazition syntax használata javasolt.

Árnyalók betöltése

```
std::basic_ifstream<GLchar> vs_file(VERTEX_SHADER_PATH);

std::basic_string<GLchar> vs_string(
  std::istreambuf_iterator<GLchar>(vs_file),
 (std::istreambuf_iterator<GLchar>()));
std::vector<const GLchar*> vs_c_strings{ vs_string.c_str() };

vertexShaderObj = glCreateShader(GL_VERTEX_SHADER);

glShaderSource(vertexShaderObj,
               (GLsizei)vs_c_strings.size(),
               vs_c_strings.data(), NULL);
glCompileShader(vertexShaderObj);
glGetShaderiv(vertexShaderObj, GL_COMPILE_STATUS, &GL_err);
```

Beállítjuk a forráskódot, fordítunk, és lekérdezzük sikerült-e…

Árnyalók betöltése

…ha nem sikerült…

```
if (!GL_err)
{
  GLint log_size;
  glGetShaderiv(vertexShaderObj, GL_INFO_LOG_LENGTH, &log_size);
  std::basic_string<GLchar> log(log_size, ' ');
  glGetShaderInfoLog(vertexShaderObj, log_size, NULL,
                     &(*log.begin()));

  std::cout << "Failed to compile shader: " << std::endl <<
    log << std::endl;
  std::exit(EXIT_FAILURE);
}

// Fragment shader detto
```

Árnyalók betöltése

Lekérdezzük mekkora a fordítás hibaüzenete…

```
if (!GL_err)
{
  GLint log_size;
  glGetShaderiv(vertexShaderObj, GL_INFO_LOG_LENGTH, &log_size);
  std::basic_string<GLchar> log(log_size, ' ');
  glGetShaderInfoLog(vertexShaderObj, log_size, NULL,
                     &(*log.begin()));

  std::cout << "Failed to compile shader: " << std::endl <<
    log << std::endl;
  std::exit(EXIT_FAILURE);
}

// Fragment shader detto
```

Árnyalók betöltése

Lefoglalunk egy akkor stringet, amibe pont belefér

```
if (!GL_err)
{
  GLint log_size;
  glGetShaderiv(vertexShaderObj, GL_INFO_LOG_LENGTH, &log_size);
  std::basic_string<GLchar> log(log_size, ' ');
  glGetShaderInfoLog(vertexShaderObj, log_size, NULL,
                     &(*log.begin()));

  std::cout << "Failed to compile shader: " << std::endl <<
    log << std::endl;
  std::exit(EXIT_FAILURE);
}

// Fragment shader detto
```

Árnyalók betöltése

Beleírjuk a hibaüzenetet.

```
if (!GL_err)
{
  GLint log_size;
  glGetShaderiv(vertexShaderObj, GL_INFO_LOG_LENGTH, &log_size);
  std::basic_string<GLchar> log(log_size, ' ');
  glGetShaderInfoLog(vertexShaderObj, log_size, NULL,
                     &(*log.begin()));

  std::cout << "Failed to compile shader: " << std::endl <<
    log << std::endl;
  std::exit(EXIT_FAILURE);
}

// Fragment shader detto
```

Árnyalók betöltése

```
if (!GL_err)
{
  GLint log_size;
  glGetShaderiv(vertexShaderObj, GL_INFO_LOG_LENGTH, &log_size);
  std::basic_string<GLchar> log(log_size, ' ');
  glGetShaderInfoLog(vertexShaderObj, log_size, NULL,
                     &(*log.begin()));

  std::cout << "Failed to compile shader: " << std::endl <<
    log << std::endl;
  std::exit(EXIT_FAILURE);
}

// Fragment shader detto
```

Sajnos a std::basic_string<GLchar>::data() metódusa const GLchar*-ot ad.

Itt kihasználjuk, hogy a string folytonosan van memóriában letárolva és az első elem memória címét kivarázsoljuk.

Árnyalók betöltése

Létrehozunk egy programot

```
glProgram = glCreateProgram();

glAttachShader(glProgram, vertexShaderObj);
glAttachShader(glProgram, fragmentShaderObj);

glLinkProgram(glProgram);
glGetProgramiv(glProgram, GL_LINK_STATUS, &GL_err);

if (!GL_err)
{
  // Print link log
}
```

Árnyalók betöltése

Adott sorrendben belepakoljuk a lefordított árnyalókat. Itt hozzuk létre a szerelőszalagot.

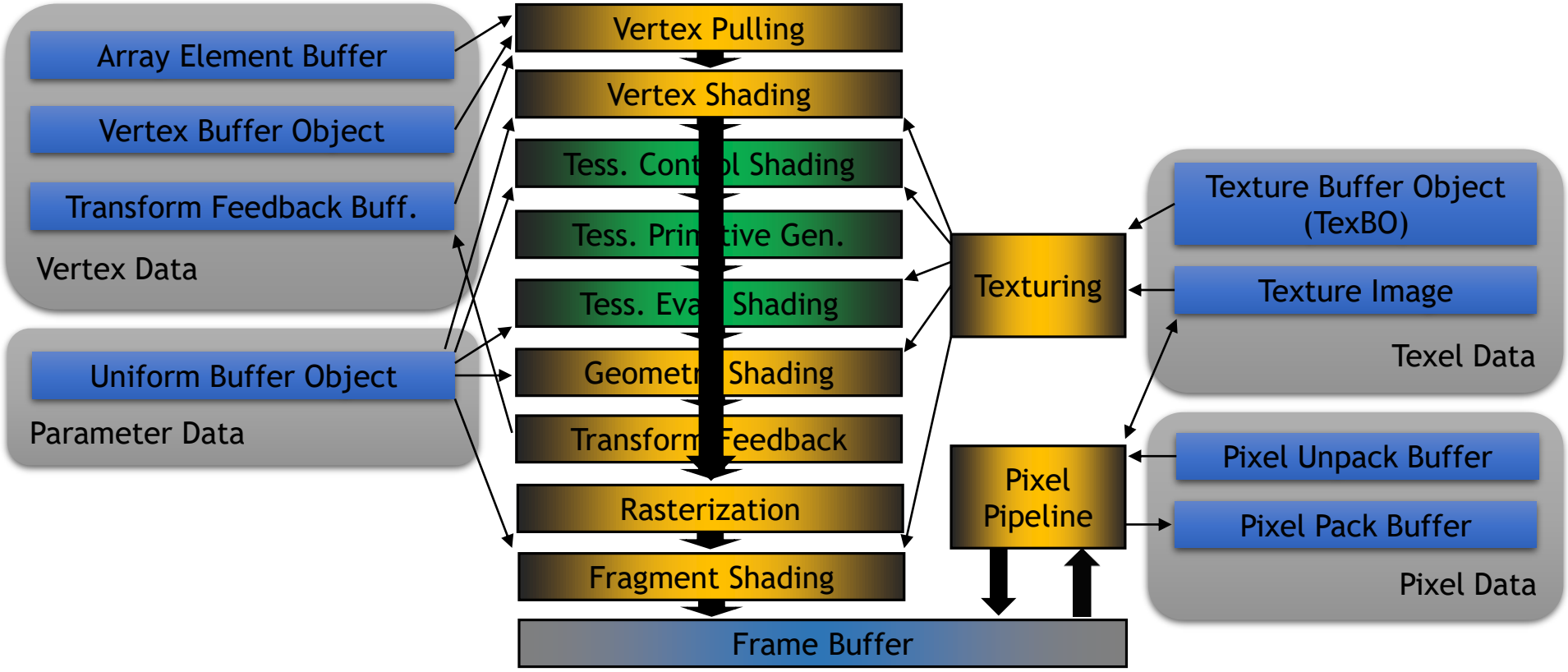
```
glProgram = glCreateProgram();

glAttachShader(glProgram, vertexShaderObj);
glAttachShader(glProgram, fragmentShaderObj);

glLinkProgram(glProgram);
glGetProgramiv(glProgram, GL_LINK_STATUS, &GL_err);

if (!GL_err)
{
  // Print link log
}
```

Árnyalók betöltése

Linkelünk, és ha nem sikerült kiírjuk a naplót.

```
glProgram = glCreateProgram();

glAttachShader(glProgram, vertexShaderObj);
glAttachShader(glProgram, fragmentShaderObj);

glLinkProgram(glProgram);
glGetProgramiv(glProgram, GL_LINK_STATUS, &GL_err);

if (!GL_err)
{
  // Print link log
}
```

OpenGL 4.2 szerelőszalag



Kanonikus inkrementális képszintézishez alakított implicit API

Gazda oldali memória inicializálás

```
                                                    struct Vertex
                                                    {
                                                      glm::vec3 p;
                                                      glm::vec3 c;
                                                    };

// geometria kialakitasa
Vertex geom[] =
{
//               p.x                        p.y                        p.z
//               c.R   c.G    c.B
  { glm::vec3( cos(0.f   * PI/180), sin(0.f   * PI/180), 0.0f),
    glm::vec3(1.0f, 0.0f, 0.0f) },
  { glm::vec3( cos(120.f * PI/180), sin(120.f * PI/180), 0.0f),
    glm::vec3(0.0f, 1.0f, 0.0f) },
  { glm::vec3( cos(240.f * PI/180), sin(240.f * PI/180), 0.0f),
    glm::vec3(0.0f, 0.0f, 1.0f) },
};
```

Gazda oldali memória inicializálás

Ez egy háromszög az x-y síkban. A tömb elemei vertexek, amelyek koordináta-szín párokból állnak.

```
struct Vertex
{
  glm::vec3 p;
  glm::vec3 c;
};
```

```
// geometria kialakitasa
Vertex geom[] =
{
//               p.x                        p.y                        p.z
//               c.R  c.G   c.B
  { glm::vec3( cos(0.f   * PI/180), sin(0.f   * PI/180), 0.0f),
    glm::vec3(1.0f, 0.0f, 0.0f) },
  { glm::vec3( cos(120.f * PI/180), sin(120.f * PI/180), 0.0f),
    glm::vec3(0.0f, 1.0f, 0.0f) },
  { glm::vec3( cos(240.f * PI/180), sin(240.f * PI/180), 0.0f),
    glm::vec3(0.0f, 0.0f, 1.0f) },
};
```

Gazda oldali memória inicializálás

A glm::vec3 három darab floatot tartalmaz, és nem analóg a cl_float3-mal. Ez TÉNYLEG három darab float.

```
struct Vertex
{
  glm::vec3 p;
  glm::vec3 c;
};
```

```
// geometria kialakitasa
Vertex geom[] =
{
//               p.x                    p.y                    p.z
//               c.R  c.G   c.B
  { glm::vec3( cos(0.f   * PI/180), sin(0.f   * PI/180), 0.0f),
    glm::vec3(1.0f, 0.0f, 0.0f) },
  { glm::vec3( cos(120.f * PI/180), sin(120.f * PI/180), 0.0f),
    glm::vec3(0.0f, 1.0f, 0.0f) },
  { glm::vec3( cos(240.f * PI/180), sin(240.f * PI/180), 0.0f),
    glm::vec3(0.0f, 0.0f, 1.0f) },
};
```

Eszköz oldali memória inicializálás

```
// Generaljunk egy buffer objektumot
glGenBuffers(1, &m_vbo); checkError("glGenBuffers(m_vbo)");

// Buffer objektum hasznalatba vetele
glBindBuffer(GL_ARRAY_BUFFER, m_vbo); checkError("glBindBuffer(m_vbo)");

// Memoria lefoglalasa, de meg ne masoljunk bele.
glBufferData(GL_ARRAY_BUFFER, sizeof(geom), NULL, GL_STATIC_DRAW);
checkError("glBufferData(m_vbo)");

// Toltsuk fel a buffer (jelen esetben egeszet) a geometriaval
glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(geom), geom);
checkError("glBufferSubData(m_vbo)");

// Buffer feltoltese utan hasznalat vege
glBindBuffer(GL_ARRAY_BUFFER, 0); checkError("glBindBuffer(0)");
```

Eszköz oldali memória inicializálás

```
// Generaljunk egy buffer objektumot
glGenBuffers(1, &m_vbo); checkError("glGenBuffers(m_vbo)");

// Buffer objektum hasznalatba vetele
glBindBuffer(GL_ARRAY_BUFFER, m_vbo); checkError("glBindBuffer(m_vbo)");

// Memoria lefoglalasa, de meg ne masoljunk bele.
glBufferData(GL_ARRAY_BUFFER, sizeof(geom), NULL, GL_STATIC_DRAW);
checkError("glBufferData(m_vbo)");

// Toltsuk fel a buffer (jelen esetben egeszet) a geometriaval
glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(geom), geom);
checkError("glBufferSubData(m_vbo)");

// Buffer feltoltese utan hasznalat vege
glBindBuffer(GL_ARRAY_BUFFER, 0); checkError("glBindBuffer(0)");
```

Eszköz oldali memória inicializálás

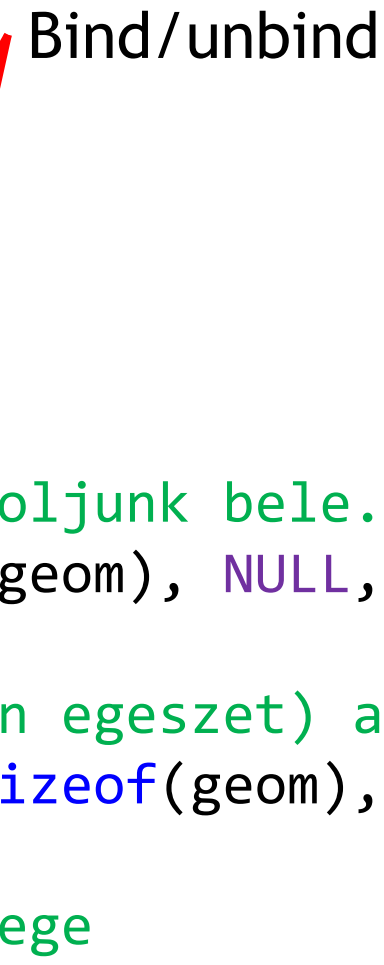
```
// Generaljunk egy buffer objektumot
glGenBuffers(1, &m_vbo);

// Buffer objektum hasznalatba vetele
glBindBuffer(GL_ARRAY_BUFFER, m_vbo);

// Memoria lefoglalasa, de meg ne masoljunk bele.
glBufferData(GL_ARRAY_BUFFER, sizeof(geom), NULL, GL_STATIC_DRAW);

// Toltsuk fel a buffer (jelen esetben egeszet) a geometriaval
glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(geom), geom);

// Buffer feltoltese utan hasznalat vege
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

Eszköz oldali memória inicializálás

```
// Generaljunk egy buffer objektumot
glGenBuffers(1, &m_vbo);

// Buffer objektum hasznalatba vetele
glBindBuffer(GL_ARRAY_BUFFER, m_vbo);

// Memoria lefoglalasa, de meg ne masoljunk bele.
glBufferData(GL_ARRAY_BUFFER, sizeof(geom), NULL, GL_STATIC_DRAW);

// Toltsuk fel a buffer (jelen esetben egeszet) a geometriaval
glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(geom), geom);

// Buffer feltoltese utan hasznalat vege
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

Buffer handle létrehozása. m_vbo típusa GLint. Semmi sem akadályoz meg bennünket abban, hogy legközelebb m_vbo-t glUseProgram-ban használjuk.

Eszköz oldali memória inicializálás

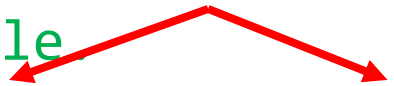



```
// Generaljunk egy buffer objektumot
glGenBuffers(1, &m_vbo);

// Buffer objektum hasznalatba vetele
glBindBuffer(GL_ARRAY_BUFFER, m_vbo);

// Memoria lefoglalasa, de meg ne masoljunk bele.
glBufferData(GL_ARRAY_BUFFER, sizeof(geom), NULL, GL_STATIC_DRAW);

// Toltsuk fel a buffer (jelen esetben egeszet) a geometriaval
glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(geom), geom);

// Buffer feltoltese utan hasznalat vege
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

Eszköz oldali memória inicializálás

```
// Generaljunk egy buffer objektumot
glGenBuffers(1, &m_vbo);

// Buffer objektum hasznalatba vetele
glBindBuffer(GL_ARRAY_BUFFER, m_vbo);

// Memoria lefoglalasa, de meg ne masoljunk bele
glBufferData(GL_ARRAY_BUFFER, sizeof(geom), NULL, GL_STATIC_DRAW);

// Toltsuk fel a buffer (jelen esetben egeszet) a geometriaval
glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(geom), geom);

// Buffer feltoltese utan hasznalat vege
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

Buffer méretének meghatározása, de az adat pointere még NULL, nem ezzel a paranccsal töltjük fel. Varázs define azt jelenti, ritkán (vagy sose) fog változni a tartalma.



Eszköz oldali memória inicializálás

```
// Generaljunk egy buffer objektumot
glGenBuffers(1, &m_vbo);

// Buffer objektum hasznalatba vetele
glBindBuffer(GL_ARRAY_BUFFER, m_vbo);

// Memoria lefoglalasa, de meg ne masoljunk bele.
glBufferData(GL_ARRAY_BUFFER, sizeof(geom), NULL, GL_STATIC_DRAW);

// Toltsuk fel a buffer (jelen esetben egeszet) a geometriaval
glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(geom), geom);

// Buffer feltoltese utan hasznalat vege
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

Buffer feltöltése a szokásos módon. Offset, méret, pointer. clEnqueueWriteBuffer-rel analóg.

Rajzolás pseudo kód

```
auto it = std::cbegin(my_scene);

while (it != std::cend(my_scene))
{
  render_state_descriptor state = *(it).m_desc;

  auto next = std::partition(std::par, it, std::cend(m_scene),
    [&](const drawable& obj) { return obj.m_desc == state; });

  gfx::default_device().set_state(state);
  std::for_each(std::par, it, next, draw);

  it = next;
}
```

```
struct drawable {
  vbo m_vbo;
  ibo m_ibo;
  render_state_descriptor m_desc;};
```

Rajzolás pseudo kód

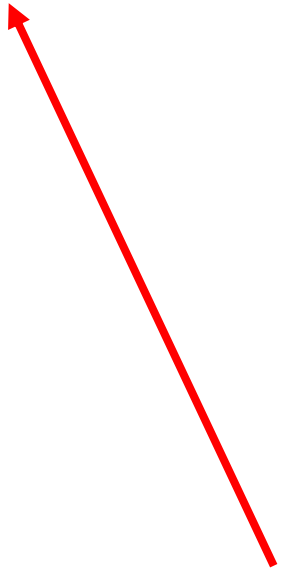
```
auto it = std::cbegin(my_scene);

while (it != std::cend(my_scene))
{
  render_state_descriptor state = *(it).m_desc;

  auto next = std::partition(std::par, it, std::cend(m_scene),
    [&](const drawable& obj) { return obj.m_desc == state; });

  gfx::default_device().set_state(state);
  std::for_each(std::par, it, next, draw);

  it = next;
}
```

```
struct drawable {
  vbo m_vbo;
  ibo m_ibo;
  render_state_descriptor m_desc;};
```

Emlékezzünk vissza erre az állatra. Ebben az objektumban van benne minden információ, ami az objektum kirajzolásához kell. Most ilyesmit fogunk látni.

Parancsok mentése, visszajátszása

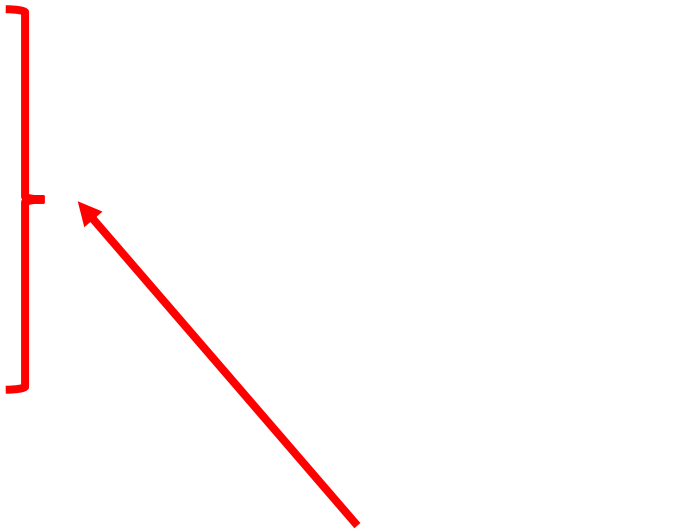
```
// Osszefogo VAO letrehozasa es hasznalatba vetele
glGenVertexArrays(1, &m_vao);

glBindVertexArray(m_vao);
{
   ...
}
glBindVertexArray(0);
```



Vertex Array Object: egy objektum, amely vertex array-ekhez tartozó csatolásokat/állapotot reprezentál.
A valóságban: visszajátsza azokat a parancsokat, amiket tanítottak neki / megjegyzi az állapotot, amibe állították.

Parancsok mentése, visszajátszása

```
// Osszefogo VAO letrehozasa es hasznalatba vetele
glGenVertexArrays(1, &m_vao);

glBindVertexArray(m_vao);
{
   ...
}
glBindVertexArray(0);
```



Csatol, beállít, lecsatol

Vertex attribútumok beállítása

- A Vertex Buffer Objectbe sorfolytonosan beírtuk a pozíció-szín értékeket
- Ezt egyedül mi tudjuk, az OpenGL-nek dunsztja sincs arról, mit kell csinálni ezekkel az értékekkel
- Az árnyaló kódban, szeretnénk majd névvel illetni ezeket a mezőket, tehát valahogy csatolni kell a nevet az értékekhez
  - A név egyedül az árnyaló kódban lesz megnevezve
  - Gazda oldalon egy indexhez kötjük az értéket
  - Az árnyaló kódja fog az index értékhez nevet rendelni
- Nagyon hasonlít a folyamat a clSetKernelArg parancshoz
  - Gazda oldalon index értékhez (kernel argumentum index) buffert rendeltünk
  - Itt gátlástalanul használunk aliasingot, OpenCL-ben kicsit trükkösebb

Vertex attribútumok beállítása

```
// Belso buffer aktivalasa
glBindBuffer(GL_ARRAY_BUFFER, m_vbo);
glVertexAttribPointer(0,                // A modositando tulajdonsaghoz tartozo index
                      3,                // Hany darab adatot olvassunk
                      GL_FLOAT,         // Adat amit olvasunk
                      GL_FALSE,         // Arnyalo olvasaskor automatikus normalizalas
                      sizeof(Vertex),// Mekkora kozokkel szerepelnek az adatok
                      (GLvoid *)0);  // Elso elem pointere (nem gazda oldali pointer!)

glVertexAttribPointer(1,                // A modositando tulajdonsaghoz tartozo index
                      3,                // Hany darab adatot olvassunk
                      GL_FLOAT,         // Adat amit olvasunk
                      GL_FALSE,         // Arnyalo olvasaskor automatikus normalizalas
                      sizeof(Vertex),// Mekkora kozokkel szerepelnek az adatok
                      (GLvoid *)(0 + sizeof(glm::vec3))); // Elso elem pointere

// Vertex attributum indexek aktivalasa
glEnableVertexAttribArray(0);
glEnableVertexAttribArray(1);
```

Dragons be here!

Wigner GPU Lab

glBufferSubData

- Paraméterek
  - GLenum target,
  - GLintptr offset,
  - GLsizeiptr size,
  - const GLvoid * data

glVertexAttribPointer

- Paraméterek
  - GLuint index,
  - GLint size,
  - GLenum type,
  - GLboolean normalized,
  - GLsizei stride,
  - const GLvoid * pointer

Dragons be here!

glBufferSubData

- Paraméterek
  - GLenum target,
  - GLintptr offset,
  - GLsizeiptr size,
  - const GLvoid * data

Gazda oldali pointer, ahonnan olvasunk

glVertexAttribPointer

- Paraméterek
  - GLuint index,
  - GLint size,
  - GLenum type,
  - GLboolean normalized,
  - GLsizei stride,
  - const GLvoid * pointer

Eszköz oldali „pointer”, ami az első elemre mutat. A buffer a 0 pointer értéknél kezdődik.

Dragons be here!

glBufferSubData

- Paraméterek
  - GLenum target,
  - GLintptr offset,
  - GLsizeiptr size,
  - const GLvoid * data

Előjeles 32/64-bites integer

glVertexAttribPointer

- Paraméterek
  - GLuint index,
  - GLint size,
  - GLenum type,
  - GLboolean normalized,
  - GLsizei stride,
  - const GLvoid * pointer

Előjeles 32-bites integer

Dragons be here!

glBufferSubData

- Paraméterek
  - GLenum target,
  - GLintptr offset,
  - GLsizeiptr size,
  - const GLvoid * data

Előjeles 32/64-bites integer

glVertexAttribPointer

- Paraméterek
  - GLuint index,
  - GLint size,
  - GLenum type,
  - GLboolean normalized,
  - GLsizei stride,
  - const GLvoid * pointer

Előjeles 32-bites integer

A méret (ami mindig nem-negatív) előjeles, mondván „32-bites rendszeren sosem fog kelleni 2GB-nál nagyobb buffer”.

Let them eat cake!

„640 kB is enough” - ?

Model-View-Projection mátrixok

```
m_vecEye = glm::vec3(0.f, 0.f, 3.f);

// Matrixok beallitasa
m_matWorld = glm::mat4(1.0f); // Modell&vilag koordinatak azonosak

m_matView = glm::lookAt(m_vecEye,                       // honnan
                        glm::vec3(0.f, 0.f, 0.f),   // hova
                        glm::vec3(0.f, 1.f, 0.f));  // fel vektor

m_matProj = glm::perspective(45.0f,  // 90 fokos nyilasszog
                             ((float)window.getSize().x) /
                                 window.getSize().y,
                                     // nezeti arany
                             0.01f,  // Kozeli vagosik
                             100.0f);// Tavoli vagosik
```

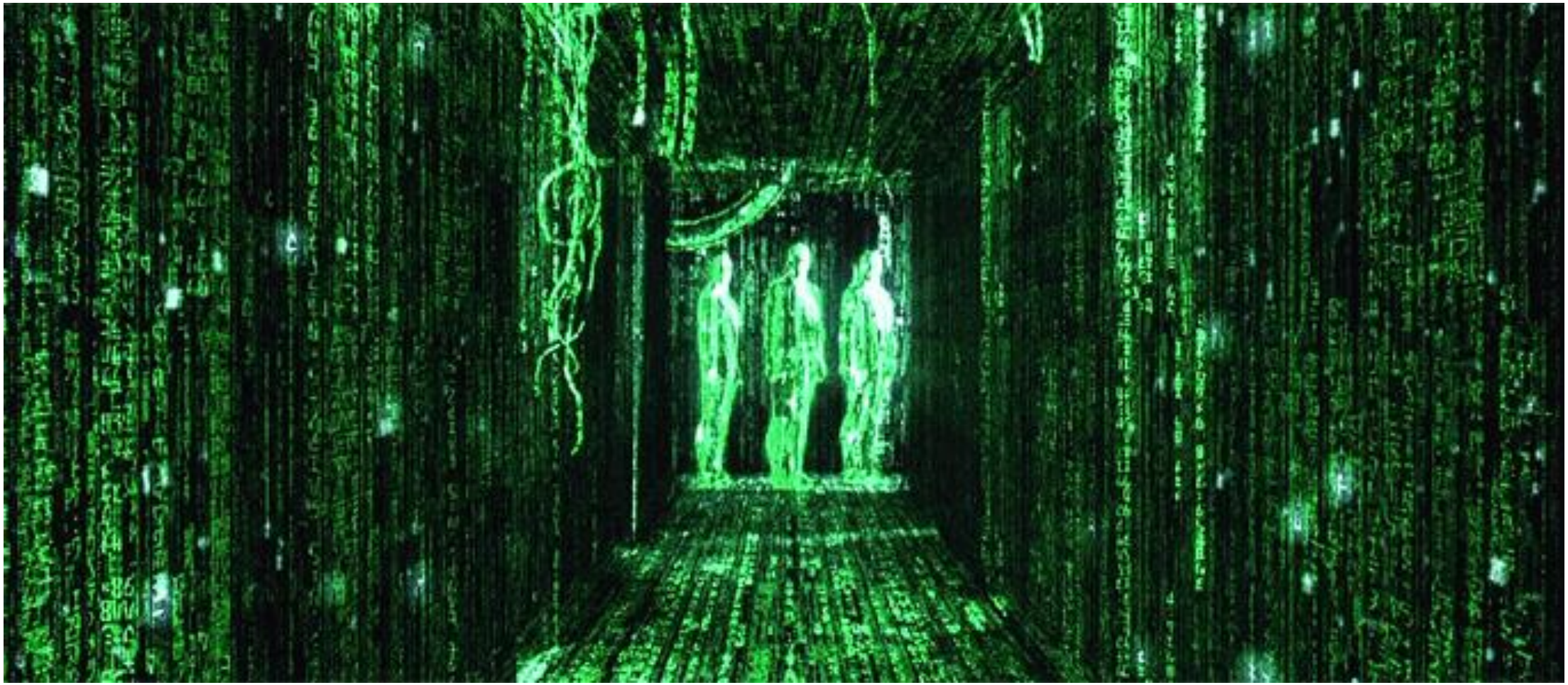
Uniformis változók beállítása

- OpenCL-ben a bufferek (tömbök) és az egyszerű típus példányok is kernel argumentumok voltak
  - Az argumentum indexekhez rendeltünk értéket gazda oldalról
- OpenGL-ben csak a bufferek lesznek indexhez kötve; az egyszerű típus példányok globális változók lesznek névvel
  - A globális változóknak nincs indexük, csak nevük.
  - Gazda oldalon lekérdezzük, létezik-e adott nevű globális (uniform) változó
  - Az így kapott handle segítségével rendelhetünk hozzá értéket
  - Uniform, mert nem változók vertexről vertexre az értéke

Model-View-Projection mátrixok

```
// Arnyalo uniformis valtozoinak gazda oldali leiroinak letrehozasa
worldMatrixLocation = glGetUniformLocation(glProgram, "matWorld");
viewMatrixLocation = glGetUniformLocation(glProgram, "matView");
projectionMatrixLocation = glGetUniformLocation(glProgram, "matProj");

// Matrixok beallitasa az arnyalokban
glUniformMatrix4fv(worldMatrixLocation, 1, GL_FALSE,
                   &m_matWorld[0][0]);
glUniformMatrix4fv(viewMatrixLocation, 1, GL_FALSE,
                   &m_matView[0][0]);
glUniformMatrix4fv(projectionMatrixLocation, 1, GL_FALSE,
                   &m_matProj[0][0]);
```

Lássuk már azt az árnyaló kódot



Vertex árnyaló

```
#version 330

// VS locations
#define POSITION 0

layout(location = POSITION) in vec3 in_Position;

out block
{
    vec4 Position;
    vec4 Color;
} VS_Out;

uniform mat4 matWorld;
uniform mat4 matView;
uniform mat4 matProj;

void main() {
    gl_Position = matProj * matView * matWorld * vec4(in_Position, 1.0);

    VS_Out.Position = matWorld * vec4(in_Position, 1.0);
    VS_Out.Color = vec4(in_Color, 1.0);
}
```

Fragment árnyaló

```
#version 330

in block
{
    vec4 Position;
    vec4 Color;
} FS_In;

void main()
{
    gl_FragColor = FS_In.Color;
}
```

Hello interop!

OpenGL Application Binary Interface

Wigner GPU Lab

OpenGL

- Egyidejűleg egy implementáció a gépen
- Implementáció a libGL.so fájlban
- A gyártók kiterjesztéseit külön be kell tölteni
- Az ablakkezelők kiterjesztéseit is

OpenCL

- Egyidejűleg több implementáció a gépen
- Implementáció külön dll-ben, ICD tölti be
- Gyártó kiterjesztéseket az ICD kezeli
- Nem hat direkten kölcsön az ablakkezelővel

OpenCL-OpenGL interop

```
// Render
glFinish();
// Compute
clFinish();
```

- Minden OpenGL erőforrásnak léteznie kell, mire az OpenCL contextet létrehozzuk, amivel interoppolni akarunk!
- Aktívnak kell lenni az OpenGL contextnek
- A cl_context_properties tömbön keresztül fogjuk megadni, mely OpenGL contexttel szeretnénk kölcsönhatni
- Az OpenCL interop buffereket az OpenGL-es megfelelőikből fogjuk létrehozni
  - Osztoznak az erőforráson
  - Használatukhoz kölcsön kell kérni őket az OpenGL contexttől
- Bufferek, textúrák, és eventek oszthatók meg