Dr. Mileff Péter

GRAFIKA PROGRAMOZÁSA

3D GRAFIKA I. RÉSZ

Miskolci Egyetem
Általános Informatikai Tanszék

HÁROMDIMENZIÓS MEGJELENÍTÉS ALAPJAI…

Általános áttekintés

- A mai számítógépes grafika legmagasabb színvonalát a háromdimenziós megjelenítés jelenti.
- A zászlóvivő szintén a számítógépes játékok ipara.
  - A grafikus és játékmotorok évről évre megújulnak,
  - beépítik a GPU fejlődése nyújtotta új lehetőségeket.
- **A háromdimenziós grafika:**
  - a térben elhelyezkedő három dimenziós objektumokkal, elemekkel dolgozik.
  - Megjelenítés során ezeket a két dimenziós képernyőre vetítik,
    - valamilyen típusú vetítési transzformáció segítségével
  - majd pixelekké alakítva jelenítik meg

Objektumok reprezentációja

- Több megoldás is kialakult az évek során
- **Legfontosabbak:**
  - **1. Voxel alapú (Volume) tárolás:**
    - az objektumot egy három dimenziós mátrix pontjai reprezentálják
    - Minden mátrixelem egy pontja az objektumnak reprezentálva annak színét.
  - **2. Poligon (Polygon) alapú tárolás:**
    - az objektumokat egy poligonhálóval írjuk le.
    - A hálóra textúrákat feszítünk
- Ma: poligon alapú vizualizáció a domináns
  - **Oka:**
    - A GPU-k a kezdetektől a poligon alapú raszterizációt támogatták
    - így az ipar, a szerkesztő szoftverek erre a megoldásra rendezkedtek be.

A megjelenítés

- A tárolási forma mellett a fontos terület **a képet előállító algoritmus típusa**.
- **Számos megközelítési mód került kidolgozásra:**
  - egy-egy szoftver akár többet is alkalmazhat a végső látvány kialakítása érdekében.
  - Bizonyos algoritmusok a jelenlegi számítási kapacitások mellett lehetővé teszik a valósidejű képalkotást,
  - mások csak speciális célhardverek segítségével teszik ezt lehetővé,
  - megint más algoritmusok pedig olyannyira időigényesek, hogy reménytelennek tűnik őket valós időben használni.
- **Raszterizációs algoritmusok**

Raszterizációs algoritmusok

- **Háromszög kifestés alapú megoldások:**
  - A mai grafikában domináns
  - A raszterizáció folyamatában a háromszögeket egyenként kifestik
    - Textúra és egyéb számított értékek (pl. fény) alapján
  - **Oka:** valós időben való alkalmazhatóság
    - pl. játékok, szimulációk, stb
- **Sugár alapú algoritmusok:**
  - Kibocsájtott sugarakkal dolgoznak és úgy állítják össze a képet.
    - Lassú folyamat
  - **Sugárvetés** (raycasting)
  - **Sugárkövetés** (raytracing, cone tracing, stb),

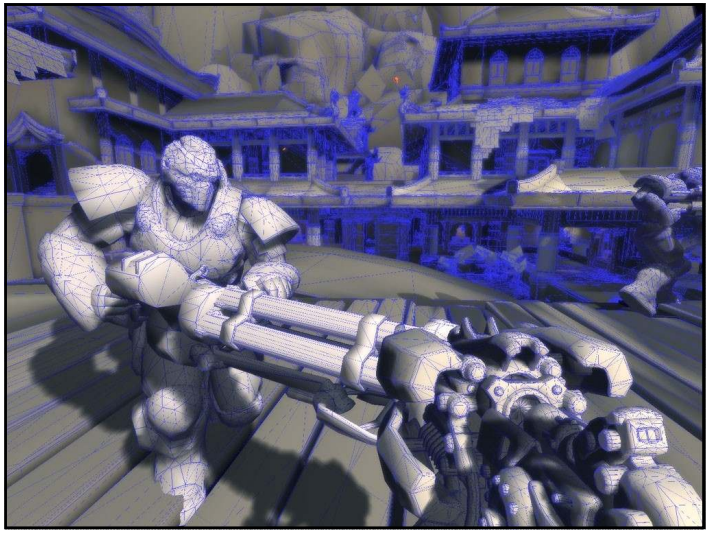
Raszterizációs algoritmusok

- Vannak úgynevezett hibrid technológiát alkalmazó megjelenítők
  - a gyakorlatban nem nagyon jellemző
- Ötvözik a különböző renderelési megoldásokat és reprezentációs formákat.
- **Példa:**
  - még csak elméletben vázolt és tervezett megoldása az ID Software cégnek
  - **Elképzelés:**
    - A játékban a látómezőtől bizonyos távolságra eső területet képét sugárvetéssel határoznák meg
    - speciális, Sparse-Voxel-Octree adatstruktúrát alkalmazva,
    - a közelebbi objektumok megjelenítése a megszokott módon történne.

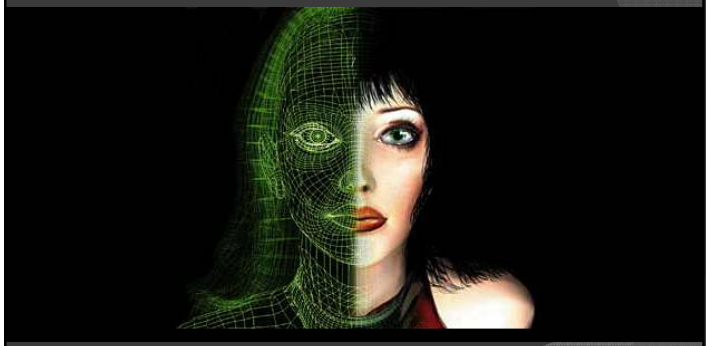
POLIGON ALAPÚ RASZTERIZÁCIÓ…

Poligon alapú vizualizáció

- **Jellemzője:**
  - A számítógépes grafika első „találmánya”.
  - A valósidejű megjelenítésben dominál főként
  - A mai grafikus kártyák mindegyike poligon alapú raszterizációt használ a kép előállítására.
    - egyszerűsített formában háromszög
- **A leképzésnek lényege:**
  - a megjelenítendő világ objektumait alapvetően poligonokból építik fel
  - A poligonok pontjait összekapcsolódó Vertex–ek alkotják.
  - A vertexek halmaza a modellek drótvázát alakítják ki.
  - Legegyszerűbb felület a háromszög
    - egy poligon legalább egy, általában több háromszögből tevődik össze.



Poligon alapú vizualizáció



SCANLINE ALAPÚ POLIGON KIFESTÉS…

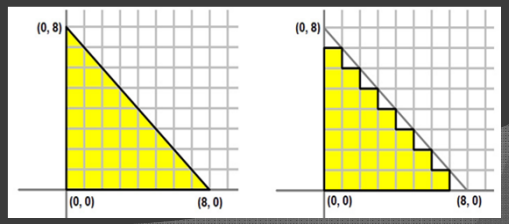
Kép előállítása

- A grafikus kártyák raszterizációja poligon kifestésen alapul
  - valamilyen módosított, optimalizált változatát valósítanak meg
- **A kép előállítása:**
  - Alapja az objektumok háromszög hálója
    - a megjelenítési API (OpenGL, DirectX) támogatásával a video memóriában tárolódik vagy jut el.
  - A GPU a kapott háromszög halmazon
    - először elvégzi a megfelelő transzformációkat (eltolás, nyújtás, elforgatás, kamera, stb),
    - majd egyenként kifesti azok két dimenzióra levetített képét.
  - Mi alapján fest?
    - Textúra, környezeti paraméterek (pl. fény), egyéb adatok

13

Diszkretizáció

- A kifestés alapegysége a **pixel**.
  - a képernyő egy gyűjtési pontja,
  - a háromszög pixelekből fog összetevődni.
- A folyamat lényegében egy **diszkretizálási eljárás**
  - a csúcspontokkal megadott háromszöget pixelekké alakítja át.



14

A poligon kifestésjellemzői

- **A diszkretizáció minősége függ:**
  - a kijelző felbontási képességétől,
  - a maximálisan megjeleníthető pixelek számától
- A modell sikerességét mindig a kifestési algoritmus gyorsasága határozza meg.
  - Erős optimalizációt igénylő feladat
- Több megoldás is kialakult az évek során
  - Pl. Scanline, Half-Space rasterization
- **Scanline:** ismertebb
  - a gyakorlatban leginkább alkalmazott kifestési algoritmus.

15

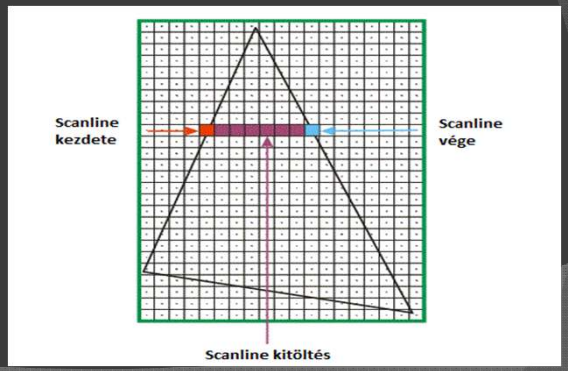
Scanline alapú poligon kifestés

Az algoritmus:

- a háromszöget úgynevezett scanline-okból építi fel
  - egy háromszögön belüli vízszintes vonalat reprezentál.
- A kifestés során gyakran fentről haladnak lefelé.
  - Az y irányú lépték 1 pixel.
  - **Ehhez ki kell számolni:**
    - az adott y értékű sorhoz mekkora x szélességű scanline tartozik,
    - a scanline széleit,
    - azaz a háromszög aktuális két oldalának x koordinátáit.
- Lineáris interpolációt használ
  - kiszámolva azt, hogy 1 pixelnyi y érték változás milyen mértékű x irányú növekményt eredményez.

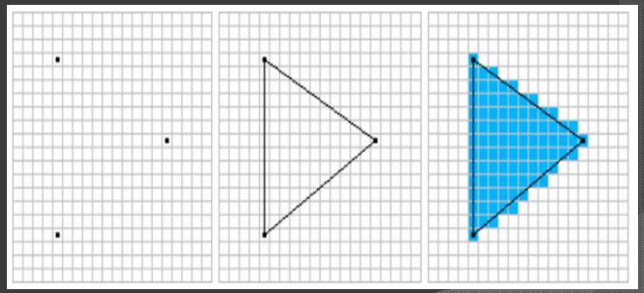
16

Scanline alapú kifestés (általánosan)



17

Scanline alapú kifestés (általánosan)



18

A kifestés jellemzői

- A kifestésnek számos paramétert kell figyelembe vennie
  - Az igényeknek megfelelve
- **Egy mai játékszoftver jellemzői:**
  - legalább egy, de sokszor több textúra
  - Sok effekt: pl. köd, átlátszó felületek, bump-, parallax mapping
  - Több fényforrás: minimum 2-3
  - Árnyék(ok): általában Shadow Map
  - egyéb vizuális élményt növelő trükkök: pl. depth of field, postprocessing
- **Különböző bufferek megvalósítása**
  - Egy grafikai motor több buffert támogat
    - pl. Színbuffer- framebuffer
    - pl. Z buffert, amely a z érték szerinti rajzolási sorrendet segíti.

19

A kifestés jellemzői

- **A kifestési algoritmus ezért kifejezetten számításigényes!**
  - Ezért lett ez a folyamat először hardveresen támogatott a GPU-kban.
- **Miért jó ha szoftveres renderer-ként implementáljuk?**
  - Az algoritmus saját kezű implementálása sok ismerethez juttat
  - Kiváló tanuló példa a grafika alapjait is megismerni akarók számára
    - legtöbbször ilyen szoftveres scanline algoritmust készítenek egy-egy nagyobb gyakorlati feladatként,
  - **De nehéz feladat:** a módszer és az egyéb feladatok megvalósítása komplex ismeretek igényel.
    - pl. Perspektíva helyes textúra leképzés, forráskód optimalizálás, stb)

20

A kifestés jellemzői

- **A raszterizációnak vannak határai:**
  - Az eljárás hatékonysága egyenes arányban csökken a jelenetben használt poligonszám növekedésével.
  - A jobb minőségért egyre több és kisebb háromszögeket fog alkalmazni az ipar.
  - Ez egyre több memóriát és számítási kapacitás fog igényelni.
- A távoli jövőben a **sugárkövetés** és más eljárások előnyét is eredményezheti
  - A sok és kis méretű háromszögek esetén nem biztos, hogy a kifestés marad a leghatékonyabb megoldásnak.

21

FÉLTÉR ALAPÚ POLIGON KIFESTÉS…

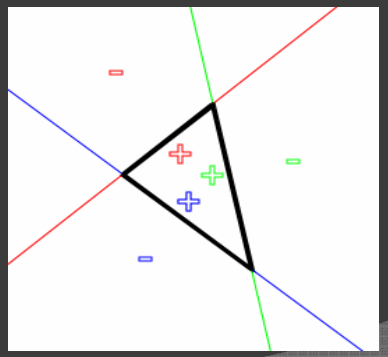
22

A féltér alapú kifestés

- További, bár kevésbé közismert módszer a kifestés megvalósítására
  - **Half Space rasterization**
- **Az algoritmus lényege:**
  - a háromszögnek három oldala van, és mindegyik oldal egy síkot reprezentál
  - Melyik pixel tartozik a háromszög belsejébe?
    - a háromszög csúcspontjai alapján meghatározzuk minden oldal szakaszának egyenletét,
    - megnézzük, hogy az éppen vizsgált pixel a szakasz melyik oldalán van.
    - Ha az aktuális vizsgált pontot minden oldal egyenletébe helyettesítve pozitív eredményt kapunk, akkor a pont a háromszög része.

23

A féltér alapú kifestés



24

A féltér alapú kifestés

- **A módszer előnye:**
  - alapesetben gyorsabb, mint a scanline alapú megközelítés
  - Egyszerűbb modell
  - Könnyebben párhuzamosítható
  - Hatékony optimalizáció:
    - Pl. Fix pontos matematikával
- **Hátránya:**
  - nagy háromszögek esetén lassabb működést eredményezhet.
  - **Oka:**
    - a befoglaló doboz miatt sok felesleges pontot kell átvizsgálni

25

Példa algoritmus

```
// Befoglaló téglalap kiszámítása
int minx = (int)min<float>(x1, x2, x3);
int maxx = (int)max<float>(x1, x2, x3);
int miny = (int)min<float>(y1, y2, y3);
int maxy = (int)max<float>(y1, y2, y3);

// A háromszög befoglaló dobozának bejárása
for(int y = miny; y < maxy; y++) {
    for(int x = minx; x < maxx; x++) {
        // Egyenletek kiszámítása
        float aa = (x1 - x2) * (y - y1) - (y1 - y2) * (x - x1);
        float bb = (x2 - x3) * (y - y2) - (y2 - y3) * (x - x2);
        float cc = (x3 - x1) * (y - y3) - (y3 - y1) * (x - x3);
        // Ha minden érték pozitív, a pixel a háromszög része.
        if (aa < 0 && bb < 0 && cc < 0){
            frameBuffer->SetPixel(x,y,color);
        }
    }
}
```

26

A féltér alapú kifestés

- Az algoritmus ebben a formában sebességileg nem kielégítő.
- **Számtalan javított megoldása létezik:**
  - főleg a fix pontos matematikával,
  - valamint az SSE instrukciócsalád kiterjesztéssel dolgozók a legsikeresebbek.
- **Érdekes kitekintés: (kommerciális)**
  - **Pixomatic renderer:** Unreal Tournament 2004 játékprogramban kipróbálható
  - **Swiftshader:** próba változata ingyenesen beszerezhető.
  - Érdemes megtekinteni milyen teljesítményt nyújtanak egy mai gyors, többmagos számítógépen

27

TIPIKUS (STATIKUS) MODELL REPREZENTÁCIÓ…

28

Modellek reprezentációja

- A modellek grafikus szerkesztő szoftverekben készülnek
  - A háromdimenziós modellek valamilyen tárolási struktúrával rendelkeznek
    - mind a memóriában: cél a rugalmasság
    - mind pedig a háttértárolón: cél a kompaktság
- A grafikus motorok/játékszoftverek is hasonló elven építik fel a saját reprezentációikat
- **Célszerű:** egy egységes, általános felépítési struktúrát értelmezni

29

Modellek reprezentációja (alapfogalmak)

- A háromdimenziós objektum reprezentációját **modell**nek nevezzük.
  - Egy összefoglaló adatstruktúra
  - Feladata: egységbe zárja a modellt felépítő elemeket.
    - **Tehát nem csak vertex-ek egyszerű halmaza!**
- **Elemek?**
  - Fontos az elnevezés, mert többféle adatról lehet szó.
    - pl. material-ok, objektumok, face-ek, stb
- A modell egy, vagy több **objektum**ból tevődik össze.
  - az irodalom gyakori elnevezés a **mesh**

30

Modellek reprezentációja (alapfogalmak)

- Egy-egy objektum lényegében **vertexek halmaza**.
- **Fő tulajdonsága:**
  - önállóan megjelenítendő/kezelendő egységet képvisel.
  - A material/effekt általában objektumonként változhat.
- **A logikai szétbontás lényege:**
  - egy-egy bonyolultabb modellt nem célszerű egy nagy vertex halmazként megrajzolni
  - inkább azt kisebb logikai egységekre kell bontani.
  - A szerkesztő szoftverekben (és a játékokban is) így könnyebb az egyes összetartozó, de mégis külön egységet képviselő elemek kezelése
    - pl. lecserélés, átrajzolás, stb

31

Modellek reprezentációja (alapfogalmak)

Példa: Egy autó modell.

- Célszerű az autó kerekeit külön objektumként megtervezni
  - Esetleg mozoghat, foroghat.
- Az ilyen logikai egységeket névvel és egyéb tulajdonságokkal láthatjuk el.
  - A játékból hivatkozni lehet rá.

32

Példa modell struktúra

```
struct t3DModel {
   char m_pModelName[255];              // Name of the model
   char m_pFileName[255];               // Model filename

   int numOfObjects;                    // The number of objects in the model
   int numOfMaterials;                  // The number of materials for the model

  vector<tMaterialInfo> pMaterials;     // The list of material information)
   vector<t3DObject> pObject;           // The object list for our model

   CBoundingBox* boundingBox;                   // Bounding box of the Model
   CBoundingBox* transFormedBoundingBox;  // transformed model level BB
   struct sBoundingSphere boundingSphere;    // Bounding sphere of the Model
};
```

33

Modellek reprezentációja (alapfogalmak)

- A modell struktúra láthatóan a magas szintű elemeket tárolja.
- Ezek közül a legfontosabb az **objektumok** és a **material**-ok listája.
- Ezek mellett járulékos információk még **a befoglaló testek** és az **elnevezések**.
- **Az elnevezés:**
  - nem azonosítja egyértelműen a modellt és objektumokat
  - de a szerkesztő szoftverekben előszeretettel használják
  - a szöveges leírás könnyebben megjegyezhető az ember számára, mint egy szám.

34

(Egyszerű) Objektum felépítése

```
struct t3DObject {
   char strName[255];               // The name of the object
   int objectID;                    // ID of the object
   int  numOfVerts;                 // The number of verts in the model
   int  numOfFaces;                 // The number of faces in the model
   int  numTexVertex;               // The number of texture coordinates
   int  materialID;                 // The texture ID to use,
   bool bHasTexture;                // Has the object texture map
   CVector3  *pVerts;               // The object's vertices
   CVector3  *pNormals;             // The object's normals
   CVector2  *pTexVerts;            // The texture's UV coordinates
   tFace *pFaces;                   // The faces information of the object
   CBoundingBox* boundingBox;                   // Bounding box of the object
   CBoundingBox* transFormedBoundingBox;  // Bounding Boxes
   struct sBoundingSphere boundingSphere;    // Bounding sphere of the object
};
```

35

(Egyszerű) Objektum felépítése

- Az objektum reprezentáció már kicsit bonyolultabb
  - Bár a bemutatott példa csak egy egyszerűbb objektumformára mutat példát
- **Fő elemei:**
  - a vertex-ek: az objektumot alkotó összes pont listája (*pVerts)*
  - azok normálisai: *pNormals* tömb tárolja a vertex-ekhez tartozó normálisokat
  - textúra koordináták a textúrázáshoz: *pTexVerts tömb*
  - azok face leíró információi: háromszög leíró információk (pFaces)
  - Befoglaló doboz információk
  - ID: objektum azonosító - szám
  - Név: objektum elnevezése

36

(Egyszerű) Objektum felépítése

- Az objektumok tárolása általában „**tömörített**” formában történik.
- **Jelentése:**
  - a háromszögekből felépülő objektumban több háromszög osztozik ugyanazon a vertex-en.
  - **A vertex-eket nem duplikálják!**
    - a vertex-ek nem jelennek meg külön a *pVerts* tömbben
  - Ezért szükség van egy objektum háromszögeit leíró külön struktúrára
    - összerendeli a háromszöghöz tartozó vertex-et és a textúra koordinátákat a fő tömbökből
    - pVerts, pTexVerts

37

Face struktúra

```
struct tFace {
   int vertIndex[3];        // A háromszöget alkotó vertex-ek indexei
   int coordIndex[3];       // A háromszöget alkotó vertex-ek textúra
koordinátáinak indexei
   CVector3 normal;         // Face normál vektora
};
```

- Egy háromszög vertex-einek meghatározása ilyenkor:

```
face = &obj->pFaces[i];          //  Current Face pointer

CVector3 v0 = obj->pVerts[face->vertIndex[0]];
CVector3 v1 = obj->pVerts[face->vertIndex[1]];
CVector3 v2 = obj->pVerts[face->vertIndex[2]];
```

38

MATERIAL-OK REPREZENTÁCIÓJA…

39

Material-ok tárolása

- A modell második központi eleme az objektumokat beborító material-ok.
  - Textúra és egyéb társított jellemzők halmaza
  - Egy modell tetszőleges számú material-lal lehet ellátva
- Szintén külön struktúrával kell rendelkezzen a társított tulajdonságok miatt

```
struct tMaterialInfo {
   char  strName[255];      // The texture name
   char  strFile[255];      // The texture file name
   int   texureId;          // the texture ID (OpenGL, DirectX unique ID)
   struct tColor color;     // The color of the Material
   float opacity;           // opacity of the material
};
```

40

Material-ok fontosabb tulajdonságai

- Tárolni kell a material nevét és a fájlnevét is.
  - Későbbi azonosításhoz szükségesek
- A legfontosabb a ***textureId*** mező:
  - a grafikus API textúra létrehozás általi **egyedi** azonosítót tárolja.
- **Célszerű tárolni még:**
  - egy szín információt, amennyiben a textúra színét módosítani szeretnénk,
  - és egy úgynevezett átlátszósági értéket.
    - Nem feltétlenül fontos, hiszen az RGBA textúrákban önmagában benne van az átlátszóság,
    - a gyakorlati megvalósításokban mindig hasznos mezőnek bizonyult.

41

Material-ok fontosabb tulajdonságai

- Tárolni kell a material nevét és a fájlnevét is.
  - Későbbi azonosításhoz szükségesek
- A legfontosabb a ***textureId*** mező:
  - a grafikus API textúra létrehozás általi **egyedi** azonosítót tárolja.
- **Célszerű tárolni még:**
  - egy szín információt, amennyiben a textúra színét módosítani szeretnénk,
  - és egy úgynevezett átlátszósági értéket.
    - Nem feltétlenül fontos, hiszen az RGBA textúrákban önmagában benne van az átlátszóság,
    - a gyakorlati megvalósításokban mindig hasznos mezőnek bizonyult.

42

Material-ok fontosabb tulajdonságai

A textúra színt biztosító struktúra:

```
struct tColor {
    float ambient[4];       // Ambient Color
    float diffuse[4];       // Diffuse Color
    float specular[4];      // Specular Color
    float specular_exp;     // Specular Light factor
};
```

43

Korlátok

- A példa reprezentáció célja:
  - az alapokat lefektető modell és objektum struktúra bemutatása
  - segítségével egy egyszerű modellfelépítés, a grafikus motor alapjai már megvalósítható.
- **Korlátok:**
  - minden objektumhoz a struktúrában 1 material tartozik
  - Egy minőségileg magasabb színvonalú játékszoftver esetén ez már nem állja meg a helyét.
  - A különböző modern effektek alkalmazása megköveteli a több textúra együttes tárolását és alkalmazását
    - pl. Bump mapping, Parallax mapping, stb
  - Erre bemutató példa nem tér ki

44

MODELLEK TÁROLÁSA A HÁTTÉRTÁRON…

Modellek tárolása a háttértáron

- A modellek reprezentációjának központi kérdése:
  - **Hogyan tároljuk a háttértárolón az adatokat?**
- **Fontos szabály!**
  - Célszerű megnézni az ismert modell formátumokat, azok felépítését!
    - Formájuk tükrözi a több éves kialakult tapasztalatot.
  - Nem rögtön saját formátumot kitalálni!
  - Ismertebb formátumok:

ASE, 3DS, OBJ, X, COLLADA, MD5, BLEND, B3D, stb.

Modellek tárolása a háttértáron

- Egy saját grafikus motorhoz előbb-utóbb saját modell struktúrát is kell kidolgozni.
- **Célszerű két formában is kialakítani:**
  - Szöveges formában
  - Bináris formában

Modellek tárolása a háttértáron

- **1. Szöveges forma jellemzői:**
  - kiválóan alkalmas kisebb modellek tárolására, gyors módosításokra
    - az esetleges vertex, face, vagy textúra leírási hibák könnyebben felderíthetők
  - nélkülözhetetlen a bináris forma kialakításához a fejlesztésben
  - egy próba modell akár kézzel is létrehozható
  - **Hátránya**: nagyobb modellek esetén sokat foglal a háttértáron

Modellek tárolása a háttértáron

- **1. Szöveges forma jellemzői (folytatás):**
  - betöltésük lényegesen több időt vesz igénybe még optimalizált szöveges betöltők esetén is
  - Kiemelendő az **XML** formátum: előszeretettel használják a gyakorlatban
- **2. Bináris forma jellemzői:**
  - előnye a kompaktsága és a nagy adathalmaz gyors feldolgozhatósága
  - A játékszoftverek és egyéb modellező alkalmazások főként bináris állományokat alkalmaznak

49

Modellek tárolása a háttértáron

- **Honnan jönnek az adatok a modellfájlba?**
- A válasz egyszerű: szerkesztő szoftverekből.
- **Módja:**
  - A mai vezető szerkesztő szoftverek rendelkeznek **programozható interfésszel**
  - lehetőséget nyújt saját exporter készítésére akár többféle nyelven is.
  - Az exporter segítségével a modellek struktúráját a saját formátumunkra konvertálhatjuk.
    - A szerkesztőben tárolt adatokat képes elérni és azt fájlba menteni.

pl. 3D Studio, Blender, Maya, stb

50

Modellek tárolása a háttértáron

- Saját modell formátum készítésekor ne feledkezzünk meg a továbbfejleszthetőségről sem.
- **A fejlesztések korai fázisában:**
  - **A grafikus motor** biztosan nem fogja kihasználni a GPU-k nyújtotta mai színvonalat
  - Ezért célszerű a formátumot úgy megtervezni, hogy könnyen továbbfejleszthető legyen
  - Nem gond például ha bizonyos részeket még nem használ a motor.

51

Modellek tárolása a háttértáron

Néhány gondolat mikre érdemes figyelni:

- A modell objektumok halmaza
- Egy objektumon több(féle) textúra (material) alkalmazása.
- Material-ok jellegzetességei.
  - Pl. Szín, átlátszóság
- Effektek kezelése.
  - Pl. bump/parallax mapping, stb
- Egyéb, modell/objektum specifikus paraméterek.
  - Pl. elforgatás.

52

MODELLEK TÁROLÁSA FUTÁSIDŐBEN…

Modellek tárolása futásidőben

- A középhaladó programozók:
  - akik már a komplexebb, több ezer poligont megjelenítő alkalmazások készítésével próbálkoznak
- **Gyakori kérdések:**
  - *Miért lassú az általam készített alkalmazás?*
  - *Hogyan csinálja más?*
- A válasz a szoftver implementációjától függően szerteágazó lehet
- Legtöbbször azonban a **tárolás mikéntjé**vel van a probléma!

Modellek tárolása futásidőben

- Az OpenGL többféle tárolási és megjelenítési lehetőséget kínál a programozónak.
- A világhálón fellelhető segédletek sokszor megtévesztők
  - Céljuk az egyszerűbb érthetőség és bemutatás
  - Emiatt a legegyszerűbb rajzolási megoldáson keresztül mutatják be a kérdéses területet.
- Ez pedig a ***glBegin/glEnd*** páros

```
glBegin(GL_TRIANGLES);
…
vertex-ek megadása
....
glEnd();
```

Modellek tárolása futásidőben

- A közismert példa a lehető leglassabb rajzolást teszi lehetővé.
- **Több oka is van:**
  - **1. Adatmozgatás:** folyamatos mozgatás
    - a modell vertex adatai a központi memóriában vannak
    - Minden megjelenítési fázisban az itt megadott vertex-eket mozgatni kell a GPU memóriájába
    - Ez korlátozva van a busz (PCIe) átviteli sebessége által.
  - **2. Adategység:** nincs egység
    - az adatok nem foglalódnak és tárolódnak egy egységes OpenGL struktúrába,
    - minden megjelenítési fázisban ismét megadásra kerülnek.

Modellek tárolása futásidőben

- Ezért ezt a rajzolási/tárolási módot csak példák bemutatásánál célszerű alkalmazni
- Amikor a sebesség nem kritikus.
- **Az OpenGL 3.0 verziótól a megközelítési mód már nem támogatott**
  - valamint az OpenGL ES sohasem tartalmazta ezt a rajzolási módot.

57

Modellek tárolása futásidőben

- Akkor mi a megfelelő formája a tárolásnak?
- **Az OpenGL többféle megoldás támogat:**
  - Display List (Opengl 3.2-től nem támogatott),
  - Vertex Array,
  - Vertex Buffer Object - VBO,
  - Vertex Array Object - VAO

58

MODELLEK TÁROLÁSA FUTÁSIDŐBEN (VBO ÁTTEKINTÉS)…

59

Vertex Buffer Object

- Egy OpenGL kiterjesztés: a vázolt adatmozgatási problémára jelent megoldást.
  - A szabvány 1.5 verziójától támogatott
- **Legfontosabb előnyei:**
  - lehetővé teszi, hogy a vertex adatokat a grafikus kártya gyors memóriájában tároljuk (szerver oldal)
    - „buffer objektumok” formájában.
  - Az objektumok a vertex attribútumokat tárolják,
    - elhelyezkedhetnek akár indexelt formában is.
  - A megoldás jelenleg a leggyorsabb megjelenítési módot jelenti

60

Vertex Buffer Object

- **Legfontosabb előnyei (folytatás):**
  - Kiküszöböli a folyamatos adatmozgatást:
    - központi memóriából a videokártya memóriájába.
  - A vertex adatok elérésének gyorsasága:
    - nem lassabb, mint egy tömb adatainak elérése.
  - Memóriamenedzser:
    - gondoskodik arról, hogy az adatok a memória megfelelő helyén helyezkedjenek el.
    - Pl. a gyakrabban érintett területek fontosabbak.

61

Vertex Buffer Object

VBO létrehozásának lépései:

1. Új buffer objektum generálása - glGenBuffers()
2. Buffer kötése - glBindBufferARB()
3. Vertex adatok másolása a bufferbe - glBufferData()

62

Vertex Buffer Object

```
GLuint vboID;                                   // ID of VBO
GLfloat* vertices = new GLfloat[vertexCount*3]; // Create Vertex Array
...
glGenBuffers(1, &vboID);    // generate a new VBO and get the associated ID
glBindBuffer(GL_ARRAY_BUFFER, vboID); // bind VBO in order to use

// upload data to VBO
glBufferData(GL_ARRAY_BUFFER dataSize, vertices, GL_STATIC_DRAW);

delete [] vertices;     // it is safe to delete after copying data to VBO
...
glDeleteBuffersARB(1, &vbold); // delete VBO when program terminated
```

63

Vertex Buffer Object

- A VBO létrehozása nagyon egyszerű.
- **Lépések:**
  - A *glGenBuffers()* függvénnyel létrehozunk egy új, üres VBO buffert, amelynek visszakapjuk az azonosítóját
  - A buffer kötése és típusának megadása következik a *glBindBuffer()* függvénnyel.
    - A GL_ARRAY_BUFFER flag a normál buffer típust jelzi, míg index buffer esetén GL_ELEMENT_ARRAY_BUFFER kerülne a kódba.
  - Majd a *glBufferData()* függvénnyel áttöltjük a vertex adatokat a bufferbe.
  - **Az utolsó paraméter fontossága:**
    - Ez jelzi az adatok tárolási típusát.
    - Pl. módosítható, ritkán, gyakran módosuló, stb.
    - Jelen példában a GL_STATIC_DRAW azt jelenti, hogy az adatok nem módosulnak.

64

Vertex Buffer Object

- **Megjegyzés:**
  - A kód csak a vertex adatokra mutat példát
  - A VBO alkalmazható ugyanúgy a vertex-ekhez tartozó textúra koordináták és egyéb, más adatok tárolására is!

65

A VBO KIRAJZOLÁSA…

66

Vertex Buffer Object

- **Példa:**
  - két dimenziós textúra kirajzolása
  - Két háromszög, 6 vertex és textúra koordináta van tárolva a tömbökben
- **Rajzolás folyamata:**

```
glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_TEXTURE_COORD_ARRAY);
```

- Az fenti sorok aktiválják a vertex és textúra tömböket.
  - Jelzik, hogy használni szeretnénk őket.

67

Vertex Buffer Object

```
glBindBuffer(GL_ARRAY_BUFFER, vhVBOVerticesID);
glVertexPointer(2, GL_FLOAT, 0, (char *) NULL);
```

- Megtörténik a vertex buffer kötése és a tömb definíciós leírása.
- A glVertexPointer() függvény segítségével adhatjuk meg a tömb felépítésének leírását.
- A jelenlegi nagyon egyszerű.
  - Vertex-enként két lebegőpontos típusú koordináta,
  - az elemek a tömb elejétől kezdődnek (NULL)
  - és nincs offset az elemek között.

68

Vertex Buffer Object

```
// Enable client side texturing
glClientActiveTexture(GL_TEXTURE0);

glActiveTexture(GL_TEXTURE0);       // Activate the 0 texture unit
glEnable(GL_TEXTURE_2D);            // Enable texturing
glBindTexture(GL_TEXTURE_2D, textureID);      // Bind texture

// Bind Texture coords buffer
glBindBuffer(GL_ARRAY_BUFFER, vhVBOTexcoordsID);
// Define the texture coordinates array
glTexCoordPointer(2, GL_FLOAT, 0, (char *) NULL);
```

69

Vertex Buffer Object

Végül kirajzoljuk a tömböt, majd lezárjuk azok használatát

```
glDrawArrays(GL_TRIANGLES, 0, 6);           // Draw Arrays

glDisableClientState(GL_VERTEX_ARRAY);
glDisableClientState(GL_TEXTURE_COORD_ARRAY);
```

70