Térbeli adatstruktúrák

- A nagy teljesítmény elérésének gyakori akadálya a valós-idejű alkalmazásokban a geometriai áteresztőképesség
  - A színtér és a modellek sok ezer vertexből épülnek fel
  - Normálvektorok, textúra koordináták és más attribútumok kapcsolódnak hozzájuk
- A CPU-nak és a GPU-nak kell feldolgoznia
- Az adatok másolása grafikus hardver felé szintén szűk keresztmetszet lehet
- Az OpenGL számos lehetőséget biztosít
  - Gyors geometria áteresztő képesség
  - Adatok rugalmas és kényelmes kezelése

Display listák

Display listák

- Primitívek kötegei
  - `glBegin()/glEnd()` függvény párosok
  - Egyedi `glVertex()` hívások
- Ez a lehető legrosszabb módja a geometria továbbítására a GPU felé
  - A teljesítményt is figyelembevételekor

```
glBegin(GL_TRIANGLES);
  glNormal3f(x, y, z);
  glTexCoord2f(s, t);
  glVertex3f(x, y, z);
  glNormal3f(x, y, z);
  glTexCoord2f(s, t);
  glVertex3f(x, y, z);
  glNormal3f(x, y, z);
  glTexCoord2f(s, t);
  glVertex3f(x, y, z);
glEnd();
```

Display listák

- Egy háromszög létrehozásához
  - 11 függvényhívást kell elvégeznünk
  - Mindegyik függvény potenciálisan drága ellenőrző kódot tartalmaz az OpenGL meghajtóban
  - 24 különböző négy byte-os paramétert kell a veremre helyezni
  - Visszaadni a hívó függvénynek
- Elképzelhető, hogy a grafikus hardver a CPU-ra várakozik
  - Amíg összerakja és továbbítja a geometriai kötegeket

Display listák

- Használhatóak
  - Vektor-paraméterű függvényeket
    - Konszolidálni lehet a köteget
  - Háromszög sávokat és legyezőket
    - Redundáns transzformációk és másolások csökkentésére
- Szükség van több ezer nagyon kicsi, potenciálisan költséges művelet geometriai kötegekben való elküldésére

- A meghajtó program az OpenGL
  - A parancsokat „valamilyen” módon speciális hardver utasításokra vagy műveletekre alakítja át
  - A grafikus kártyára küldi azokat
- A parancsok nem hajtódnak végre egyből
  - Egy lokális pufferben gyűlnek össze
  - Amíg egy határt el nem érnek
    - Ekkor ezek a parancsok a hardverhez kerülnek/ürítődnek (flush)
- A grafikus hardverhez vezető út sok időt vesz igénybe

- A puffer küldése a grafikus hardverhez egy aszinkron művelet
  - A CPU egy másik feladatot kezdhet el
  - Nem kell várnia az elküldött kötegelt renderelési utasítások befejeződéséig
- A hardver egy adott parancshalmaz renderelése alatt
  - A CPU azzal van elfoglalva, hogy egy új grafikus képhez tartozó parancsokat dolgoz fel
- Hatékonyan működik a CPU és a grafikus hardver között

- Három esemény idéz elő ürítést az aktuális renderelő parancsok kötegénél
  1.) A meghajtó program parancs puffere tele van
     - A pufferhez nem férünk hozzá és nem módosíthatjuk annak méretét sem
  2.) Akkor is történik ürítés, amikor egy puffer cserét hajtunk végre
     - A művelet addig nem hajtódik addig, amíg a sorban álló parancsok mindegyike végre nem hajtódik
     - Az adott színtér létrehozása befejeződött és az elküldött parancsok eredményének meg kell jelennie a képernyőn
  3.) Manuálisan idézzük elő az ürítést
     - Egyszeres színpuffert használunk
     - Az OpenGL nem tudja azt, hogy mikor fejeztük be a parancsok küldését

- Néhány OpenGL parancs azonban nem pufferelt későbbi végrehajtás céljából
  - A függvények közvetlenül érik el a frame puffert
    - Direkt módon olvassák és írják
  - Sebesség csökkenést idéznek elő a csővezetéken való áthaladásban
    - Az aktuális sorban álló parancsokat először ki kell üríteni és végre kell hajtani azokat, mielőtt a színpuffert közvetlenül módosítanánk
    - Erőszakosan kiüríthetjük a parancs puffert és várhatunk arra, hogy a grafikus hardver befejezze az összes renderelési feladatát a `glFinish()` függvény meghívásával
    - Ezt a függvényt csak nagyon ritkán használjuk a gyakorlatban

- OpenGL parancsok hívásához kapcsolódó tevékenységek költségesek azonnali renderelési mód esetén
  - A parancsok lefordulnak és átalakítódnak alacsony szintű hardver utasításokká
- Gyakran a geometria vagy másik OpenGL adat nem változik képkockánként
  - Például csak a modellnézeti mátrix változik
- Megoldás
  - Elmentjük a parancs pufferben található, előre kiszámított adatdarabot, amely valamilyen ismételt renderelési műveletet hajt végre
  - Ezt az adatdarabot később bemásolhatjuk egyszerre a parancspufferbe
    - Ezzel sok függvényhívást és fordítási munkát takarítunk meg, amely az adatot létrehozza

- Előfeldolgozott parancsok létrehozására OpenGL-ben
  - Display listák
  - Az OpenGL primitíveket `glBegin`/`glEnd` utasításokkal határoljuk el
  - A display listákat `glNewList`/`glEndList` függvény hívásokkal különítjük el egymástól
  - Egy egész értékkel azonosítjuk

```
glNewList(<unsigned integer name>,GL_COMPILE);
// ...
// OpenGL függvényhívások
// ...
glEndList();
```

- A `GL_COMPILE` paraméter
    - Csak lefordítja a listát
    - Még ne hajtja azt végre
- Használhatjuk a `GL_COMPILE_AND_EXECUTE` értéket is
- Rendszerint a display listákat csak felépítjük a program inicializálási részében és csak a rendereléskor hajtjuk végre azokat

- A display lista azonosító tetszőleges előjel nélküli egész lehet
  - Ha ugyanazt az értéket kétszer használjuk, akkor a második display lista felülírja az előzőt
    - `GLuint glGenLists(GLsizei range)`
    - Visszatérési értékként egy egyedi display lista azonosító sorozat első elemét kapjuk vissza
- Display lista felszabadítása
  - `glDeleteLists(GLuint list, GLsizei range)`
  - Felszabadítja
    - A display lista neveket
    - A listák számára lefoglalt memória területeket

- `glCallList(GLuint list)` függvényhívással tudjuk végrehajtani az előre lefordított OpenGL parancsokat tartalmazó listákat
- Display listákat tartalmazó tömbök esetén
  - `glCallLists(GLsizei n, GLenum type, const GLvoid *lists)` utasítás segítségével tudjuk lefuttatni
  - Az első paraméter a display listák száma
  - Második paramétere a tömb adat típusa
    - Rendszerint `GL_UNSIGNED_BYTE`
  - `lists` nevű tömb

- A display lista létrehozásának és végrehajtásának optimalizálása
  - Függ a megvalósítótól
- Akkor érdemes használni, amikor a lista állapotváltozásokat tartalmaz
  - Például fény ki- és bekapcsolása
- Egyes megvalósítások esetén a `glGenLists` utasítás használata szükséges a működéshez

- A display listában ne használjunk
  - `glReadPixels` függvény hívást
    - A frame puffert betölti egy memória területre mutató pointerbe
  - `glTexImage2D` függvény hívást
    - A textúra kétszer akkora memória területen lesz eltárolva
- A display lista nem tartalmazhat display lista létrehozást
  - Az egyik display listában meghívhatjuk a másik display listát

Vertextömbök

Vertextömbök

- A modell vertex adatait előre kiszámítva egy tömbben eltárolhatjuk

  Hátránya Végig kell menni a teljes tömbön, és vertexenként kell az adatokat az OpenGL-nek átadni

  Előnye A geometria változhat a műveletek során

- Mind a két megoldás jó tulajdonságát kihasználhatjuk vertextömbök használatával
- A CPU és GPU között lehet átvinni
  - Előre kiszámított vagy módosított geometriákat
  - Nagy mennyiségben
  - Egy időben

1.) Össze kell rakni a geometriához tartozó adatokat egy vagy több tömbben.
   - Ezt algoritmikusan, illetve egy állományból betöltve is el lehet végezni
2.) Meg kell mondani az OpenGL-nek, hogy hol van az adat
   - Renderelés során az OpenGL a vertex adatot a megadott tömbökből „húzza" be
3.) Világosan meg kell mondani, hogy mely tömböket használja az OpenGL
   - Elkülönített tömbökben tárolhatunk vertexeket, normálvektorokat, színeket, stb.
4.) Végre kell hajtani az OpenGL parancsokat
   - A megadott adatok alapján előállítják a megfelelő objektumot/objektumokat

- Modelljeinket tömbökben kell tárolni

```
// Vertex száma
GLuint VertCount = 100;

// Vertex pointer
GLfloat *pData = NULL;
// Normálvektor pointer
GLfloat *pNormals = NULL;
// ...

// Megfelelő méretű memória terület foglalása
  pData = malloc(sizeof(GLfloat) * VertCount * 3);
  pNormals = malloc(sizeof(GLfloat) * VertCount *
      3);

// ...
// Adatok feltöltése
// ...
```

- A `RenderScene` függvényben engedélyeznünk kell a vertexek és normálvektor tömbök használatát

```
glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_NORMAL_ARRAY);
```

- A letiltásra a `glDisableClientState(GLenum array)` függvényt használhatjuk
- Lehetséges paraméterek
  - `GL_VERTEX_ARRAY`, `GL_COLOR_ARRAY`, `GL_SECONDARY_COLOR_ARRAY`, `GL_NORMAL_ARRAY`, `GL_FOG_COORDINATE_ARRAY`, `GL_TEXURE_COORD_ARRAY` és `GL_EDGE_FLAG_ARRAY`

- Miért van szükség egy új függvényre?
  - A `glEnable`-t is használhatnánk erre a feladatra
- Az OpenGL kliens-szerver modell alapján van megtervezve

  Szerver A grafikus hardver

  Kliens A gazda CPU és memória
- Az engedélyezett/letiltott (enable/disable) állapotot a kliens oldali képre alkalmazzuk

- A vertex adatok használata előtt, meg kell mondani, hogy hol tároljuk az adatokat

```
glVertexPointer(2, GL_FLOAT, 0, pData);
```

- A többi vertextömb adattípus tartozó függvények

```
void glVertexPointer(GLint size, GLenum type,
GLsizei stride, const void *pointer);
void glColorPointer(GLint size, GLenum type,
GLsizei stride, const void *pointer);
void glTexCoordPointer(GLint size, GLenum type,
GLsizei stride, const void *pointer);
void glSecondaryColorPointer(GLint size, GLenum type,
GLsizei stride, const void *pointer);
void glNormalPointer(GLenum type,
GLsizei stride, const void *pData);
void glFogCoordPointer(GLenum type,
GLsizei stride, const void *pointer);
void glEdgeFlagPointer(GLenum type,
GLsizei stride, const void *pointer);
```

- type paraméter adja meg a tömb adattípusát

Parancs	Elemek	Érvényes adattípusok
glColorPointer	3, 4	GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_INT, GL_UNSIGNED_INT, GL_FLOAT, GL_DOUBLE
glEdgeFlagPointer	1	nem meghatározott (mindig GLboolean)
glFogCoordPointer	1	GL_FLOAT, GL_DOUBLE
glNormalPointer	3	GL_BYTE, GL_SHORT, GL_INT, GL_FLOAT, GL_DOUBLE
glSecondaryColorPointer	3	GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_INT, GL_UNSIGNED_INT, GL_FLOAT, GL_DOUBLE
glTexCoordPointer	1, 2, 3, 4	GL_SHORT, GL_INT, GL_FLOAT, GL_DOUBLE
glVertexPointer	2, 3, 4	GL_SHORT, GL_INT, GL_FLOAT, GL_DOUBLE

- `stride` paraméter byte-ban adja meg két egymást követő tömbelem közötti eltolás mértékét
  - Amennyiben ez az érték 0-val egyenlő, akkor ez azt jelenti, hogy az elemek a tömbben szorosan egymásután vannak elhelyezve
- Az utolsó paraméter, az adat tömbre mutató pointer
- Vertextömbök esetén a cél textúra egységet `glClientActiveTexture(GLenum texture)` függvény hívással lehet beállítani a `glTexCoordPointer` számára
  - A `target` paraméter `GL_TEXTURE0`, `GL_TEXTURE1` ... értékeket veheti fel

- Vertex adatok kinyerése

```
glBegin(GL_POINTS);
for(i = 0; i < VertCount; i++)
glArrayElement(i);
glEnd();
```

  - Veszi a megfelelő tömb adatokat azokból a tömbökből, amelyek a `glEnableClientState` függvénnyel engedélyezve lettek
  - Vertex, normál, szín, stb.

- Adott blokk teljes átküldése
    - `glDrawArrays(GLenum mode, GLint first, GLint count)`

`mode`	Milyen primitívet akarunk előállítani
`first`	A tömb melyik elemétől kezdve
`count`	Hány elemet akarunk a tömbökből kinyerni

```
glDrawArrays(GL_POINTS, 0, VertCount)
```

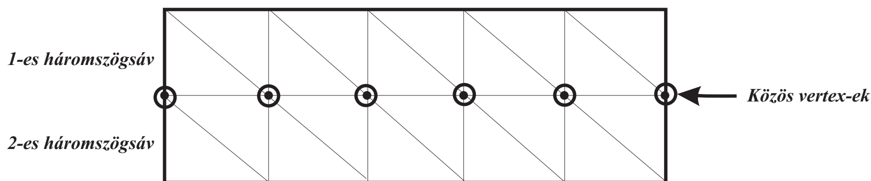
Indexelt vertextömbök

Indexelt vertextömbök

- A vertextömbhöz tartozik egy külön index értékeket tároló tömb
  - Megadja azt, hogy melyik vertexeket és milyen sorrendben kell felhasználni az adott objektum felépítésekor
- Memória területet takaríthatunk meg és csökkenthetjük a transzformációs költségeket is
- Kapcsolódó primitívek közös vertexekkel rendelkezhetnek, melyeket nem lehet egyszerűen háromszögsávok, háromszög-legyezők, négyszögsávok használatával megoldani

Indexelt vertextömbök

- Két szomszédos háromszögsáv esetén, nem létezik olyan módszer, amellyel vertexek halmazát meg lehetne osztani



Indexelt vertextömbök

- Amennyiben a vertexeket vagy a normálvektorokat újra felhasználjuk a vertextömbökben
  - Csökkenteni tudjuk a memória használatot
  - Csökkenteni lehet a transzformációk számát
  - A transzformációval töltött időt is jelentősen csökkenteni lehet

Indexelt vertextömbök

```
// ...
// A kocka nyolc sarka
static GLfloat sarkok[] =
// A kocka előlapja
    { -25.0f, 25.0f, 25.0f,
    25.0f, 25.0f, 25.0f,
    25.0f, -25.0f, 25.0f,
    -25.0f, -25.0f, 25.0f,
// A kocka hátlapja
    -25.0f, 25.0f, -25.0f,
    25.0f, 25.0f, -25.0f,
    25.0f, -25.0f, -25.0f,
    -25.0f, -25.0f, -25.0f };

// Az index tömb,
static GLubyte indexek[] =
    // Előlap
    { 0, 1, 2, 3,
    //Felső lap
    4, 5, 1, 0,
    Alsó lap
    3, 2, 6, 7,
    // Hátlap
    5, 4, 7, 6,
    // Jobb oldali lap
    1, 5, 6, 2,
    // Bal oldali lap
    4, 0, 3, 7 };

// ...
```

```
void RenderScene(void)
{
// ...

// A vertextömb engedélyezése és
    megadása
glEnableClientState(
    GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, 0,
    sarkok);

// Négyszögsávokkal való
    megjelenítés
glDrawElements(GL_QUADS, 24,
    GL_UNSIGNED_BYTE, indexek);

// ...
}
```

Indexelt vertextömbök

- `glDrawElements` függvény nagyon hasonlít a `glDrawArrays` függvényre
  - Az index tömböt is meg kell adni
  - Milyen sorrendben kell a vertextömböt tekinteni
- `glDrawRangeElements`
  - Az indexek mely részét kell felhasználni az objektum létrehozásához
- `glMultiDrawArrays`
  - Több index tömböt lehet elküldeni egyetlen függvényhívás segítségével
- `glInterleavedArrays`
  - Több tömböt egy összesített tömbben helyezzünk el
  - A tömbök memória szervezése javíthatja a teljesítményt néhány hardveres megvalósítás esetén

Vertex puffer objektumok

Vertex puffer objektumok

- A display listák egy gyors és könnyű módszert adnak az azonnali kódolási módban
  - Legrosszabb esetben a display lista egy előre lefordított OpenGL adatot fog tartalmazni
    - Gyorsan a parancs pufferbe lehet másolni és elküldeni a grafikus hardverhez
  - Legjobb esetben
    - Egy display listát a grafikus hardverbe másolhat
    - A hardver sávszélességét lényegében nullára csökkentve
  - Vertextömbök biztosítják számunkra az összes rugalmasságot
  - Az indexelt vertextömbök segítségével csökkenthetjük a vertex adatok mennyiségét

Vertex puffer objektumok

- Még egy lehetőséget biztosít a geometriai áteresztőképesség vezérlésére
  - Vertextömbök használatakor át lehet küldeni a CPU kliens oldaláról egyedi tömböket a grafikus hardverre
  - Vertex puffer objektum
    - A vertextömböket ugyanúgy használjuk és kezeljük, mint a textúra adatok betöltését és kezelését

- Vertextömböket kell használnunk
- Puffer objektumokat kell létrehoznunk
  - `glGenBuffers` függvény meghívásával
    - Első paramétere a kért objektumoknak a számát adja meg
    - Második paraméter pedig egy olyan tömb, ami az új puffer objektumok neveivel van feltöltve
  - `glDeleteBuffers` utasítással lehet felszabadítani
- A `glBindBuffer` függvény összeköti az aktuális állapotot egy bizonyos puffer objektummal
  - A `target` paraméter határozza meg azt, hogy milyen típusú tömböt fogunk kijelölni
    - `GL_ARRAY_BUFFER` a vertex adatok esetén
    - `GL_ELEMENT_ARRAY_BUFFER` index tömbök számára

- Először hozzá kell csatolni a szóban forgó puffer objektumot
- Aztán kell meghívni a `glBufferData` függvényt

```
glBufferData(GLenum target, GLsizeiptr size,
 GLvoid *data, GLenum usage)
```

`target` GL_ARRAY_BUFFER vagy GL_ELEMENT_ARRAY_BUFFER értékeket kaphatja meg

`size` Adja meg a vertextömb méretét byte-ban

`data` Inicializálásként az adat tárolóba bemásolandó adat

`usage` A várható használati minta

Használati utasítás	Leírás
GL_DYNAMIC_DRAW	A puffer objektumban tárolt adat valószínűleg sokszor fog változni, de a változások között a kirajzolásra többször fogja használni a forrást. Ez a segítség a megvalósítás számára jelzi, hogy az adatot olyan helyen kell tárolni, melynek **időnkénti megváltoztatása nem okoz nagy gondot**.
GL_STATIC_DRAW	A puffer objektumban tárolt adat nem valószínű, hogy változni fog és sokszor ez lesz a forrása a rajzolásnak. Ez azt jelzi, hogy a megvalósításnak az adatot olyan helyen kell tárolnia, ami **gyorsan olvasható, de nem szükséges gyorsan frissíteni azt**.
GL_STREAM_DRAW	A pufferben tárolt adat gyakran változik és a változások között csak néhányszor lesz szükség a forrásra. Ez a segítség azt jelzi a megvalósításnak, hogy időben változó geometriai (például animált geometria) objektumot tárolunk, amit egyszer használunk és utána meg fog változni rögtön. Elengedhetetlen, hogy az ilyen jellegű adatot olyan helyen tároljuk, amelyet **gyorsan lehet frissíteni még a gyorsabb renderelés kárára is**.

- Hozzá kell rendelni az adott vertextömböt a renderelési módhoz mielőtt a vertex mutató függvényt meghívnánk
- Az aktuális tömbre mutató pointer ezentúl egy eltolás lesz a vertex puffer objektumban

```
glVertexPointer(3,
                GL_FLOAT,
                0, pVerts);
```

```
glBindBuffer(GL_ARRAY_BUFFER,
                    bufferObjects[0]);
glVertexPointer(3,
                GL_FLOAT,
                0, 0);
```

- Renderelési függvényhívások

```
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, bufferObjects[3])
    ;
glDrawElements(GL_TRIANGLES, nNumIndexes,
    GL_UNSIGNED_SHORT, 0);
```

Poligon technikák

Poligon technikák

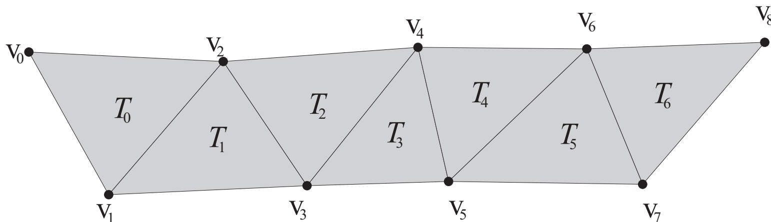
- A poligon ábrázolás általános célja a vizuális pontosság és a sebesség
- A pontosság mindig az adott környezettől függ
  - Egy modell adott pontossággal jelenik meg
  - Egy repülőgép-szimulátor esetén
    - A mérnök irányítani és pozicionálni akarja a modelleket valós időben és minden részletet minden időpillanatban látni akar a számítógépen
    - Egy játék esetén, ahol ha a képkockák sebessége elég magas, kisebb hibák vagy pontatlanságok az adott képkockán belül megengedettek

- Poligon felbontás
  - Az a folyamat, amikor a felületet poligonok halmazára bontjuk fel
- Poligon felületek felbontásával fogunk foglalkozni
- Háromszögek, mint elemi alkotó részek vesznek részt a renderelés során
  - Tetszőleges felületek előállíthatóak belőlük

- A renderelő lehet, hogy csak konvex poligonokat tud kezelni
- Előfordulhat, hogy a felületet kisebb részekre kell vágni azért,
  - Az árnyalás vagy a visszatükröződő fény megfelelően jelenjenek meg
- Poligon felbontására vonatkozó feltételek lehetnek
  - Egyetlen egy poligon se legyen nagyobb egy előre megadott területnél
  - Háromszögek esetén a háromszögek szögeinek nagyobbaknak kell lenniük egy előre megadott minimum szögnél
    - Nem-grafikai alkalmazások (pl. véges elem analízis) esetén megszokottak
    - Felület megjelenését is javítják
  - A hosszú, vékony háromszögeket jobb elkerülni
    - Különböző árnyalások esetén a vertexek közötti nagy távolságok miatt interpoláláskor a hibák jobban megjelennek

- Első lépés egy felület felbontása esetén
  - Egy 3D-s poligon esetén meghatározzuk azt, hogy melyik a legjobb vetítés 2D-re
  - A problémát és a probléma megoldásául szolgáló algoritmust leegyszerűsítsük
    - A poligont leképezzük az xy, yz és xz síkokra
    - Az a legjobb sík, amikor az adott poligon esetén a legnagyobb leképezett területet kapjuk
    - A legnagyobb nagyságú koordinátát hagyjuk el a poligon normálvektorában
  - A poligon normálvektor tesztjével nem mindig lehet meghatározni a legnagyobb területű vetületet

- A grafikus teljesítmény növelésének egy nagyon gyakori módja
    - Kevesebb vertexet küldünk át háromszögenként a grafikus csővezetéken
- A háromszögsávok és háromszöglegyezők esetén a közös vertexeket csak egyszer kell elküldeni



- Egy **szekvenciális háromszögsávot**, rendezett vertexek listájával definiálhatjuk
  - $\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_n$
    - Ahol a T_i az i-ik háromszöget $\triangle \mathbf{v}_i, \mathbf{v}_{i+1}, \mathbf{v}_{i+2}$ jelöli
    - $0 \leq i < n - 2$ esetén
  - A vertexeket a megadott sorrendben küldjük a GPU felé
- A szekvenciális háromszögsáv n vertexe $n - 2$ háromszöget határoz meg
- A vertexek v_a átlagos száma egy m hosszú szekvenciális háromszögsáv esetén a következőképpen fejezhető ki

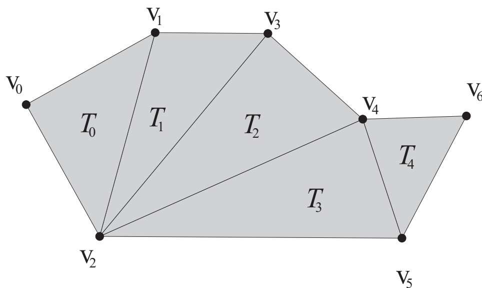
$$v_a = \frac{3 + (m - 1)}{m} = 1 + \frac{2}{m}$$

  - $m \to \infty$ esetén $v_a \to 1$

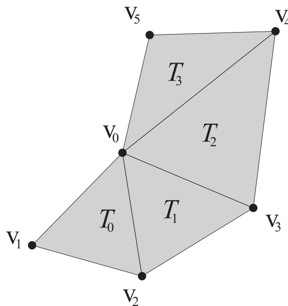
- Ha nem követeljük meg a szigorú szekvenciáját a háromszögeknek
  - Hosszabb és ezáltal hatékonyabb sávokat hozhatunk létre
  - **Általánosított háromszögsávok**
    - Szükség van vertex gyorsítótárra a grafikus kártyán
    - A transzformált és a megvilágított vertexeket tárolja
    - A vertexeket el lehet érni és ki lehet cserélni rövid bit kódok küldésével
    - Amikor a vertexek a gyorsítótárba kerülnek, akkor más háromszögek is felhasználhatják azokat elenyésző költséggel

- A szekvenciális háromszögek „általánosításához” a *csere* műveletet kell bevezetnünk
    - Megcseréli a két utolsó vertexnek a sorrendjét
    - OpenGL-ben és Direct3D-ben a csere parancsot egy vertex újra küldésével valósíthatjuk meg
        - Cserénként egy vertexnyi plusz költséget jelent
    - Olyan háromszöget hoz létre, melynek nincs területe
- Egy új háromszögsáv létrehozásának a költsége két vertex
- Az aktuális API hívások, melyek a sáv küldésért felelősek, további költségeket jelentenének

- Egy háromszögsáv, amely a $(\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3, \mathit{csere}, \mathbf{v}_4, \mathbf{v}_5, \mathbf{v}_6)$ vertexek átküldésére várakozik megvalósítható a következőképpen
  - $(\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3, \mathbf{v}_2, \mathbf{v}_4, \mathbf{v}_5, \mathbf{v}_6)$
    - Ahol a csere a $\mathbf{v}_2$ vertex újraküldésével lett megvalósítva



- A háromszög-legyező hasonló jó tulajdonsággal rendelkezik, mint a háromszögsáv
- A legtöbb alacsony szintű grafikus API támogatja a háromszög-legyezők létrehozását
- Egy általános konvex poligont könnyen lehet háromszög-legyezővé alakítani
  - Háromszögsávvá is könnyű alakítani



- A *középső vertexen* az összes háromszög osztozik
- Egy új háromszöget a középső vertex, az előzőleg elküldött vertex és az új vertex segítségével hozzuk létre
- Az n vertexből felépülő háromszög-legyezőt a $\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_{n-1}$ rendezett vertex listái alkotják
    - $\mathbf{v}_0$ a középső vertex
    - Az i-ik háromszög $\triangle \mathbf{v}_0, \mathbf{v}_{i+1}, \mathbf{v}_{i+2}$ jelöli, $0 \leq i < n-2$ esetén
- Bármelyik háromszög-legyező átalakítható háromszögsávvá
    - Sok cserét fog tartalmazni
    - Fordítva nem hajtható végre

- Általános háromszöghálót érdemes hatékonyan szétbontani háromszögsávokra
- Optimális háromszögsávok előállítására NP-teljes probléma
- Meg kell elégednünk heurisztikus módszerekkel
- Mindegyik háromszögsáv létrehozó algoritmus a poligon halmaz szomszédsági adat struktúrájának a létrehozásával kezdődik
  - Mindegyik poligonhoz tartozó él esetén szomszédos poligonra való hivatkozást is el kell tárolni
  - A poligonok szomszédjainak a számát *foknak* nevezzük
  - Egész értéke 0 és a poligon vertexeinek a száma között van

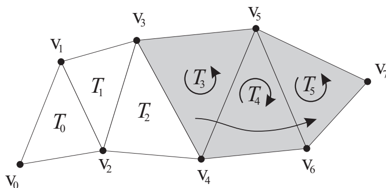
- *Euler* síkbeli kapcsolódó gráfokra vonatkozó tétele: $v - e + f - 2g = 2$
  - v a vertexek száma
  - f a lapok száma
  - g az objektumban lévő lyukak száma
- Meghatározhatjuk a vertexek átlagos számát, amit a csővezetékbe küldünk
  - A $2e \geq 3f$ mindig teljesül
    - Kapcsolódó gráfok esetén minden él mentén két lap helyezkedik el
    - Minden lapnak legalább három éle van
  - *Euler tételbe* behelyettesítve
  - Egyszerűsítés után $f \leq 2v - 4$
  - Ha mindegyik lap háromszög, akkor $2e = 3f \Rightarrow f = 2v - 4$
- A háromszögek száma kisebb vagy egyenlő a vertexek számának a kétszeresénél a háromszögekre való felbontáskor
  - Minden vertexet (átlagosan) legalább kétszer kell elküldeni a szekvenciális háromszögsáv használatakor

- Ez az algoritmus csak olyan modellek esetén működik, amelyek teljesen háromszögekből épülnek fel
- Lokális optimumok alapján dönt
  - Mindig azt a kezdő háromszöget választja, amelyiknek a legkisebb a foka (legkevesebb szomszédos oldala van)
- **SGI háromszögsáv-képző algoritmus**

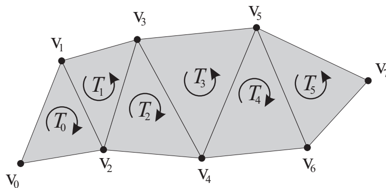
1.) Válasszuk ki a kezdő háromszöget.
2.) Építsünk 3 különböző háromszögsávot a háromszög minden éle mentén.
3.) Terjesszük ki ezeket a háromszögsávokat az háromszögsáv első elemétől az ellenkező irányba.
4.) Válasszuk ki a három közül a leghosszabb háromszögsávot és töröljük a többit.
5.) Ismételjük meg a folyamatot az 1-es lépéstől addig, amíg az összes háromszög be nem került a sávba.

- Az első lépésben az algoritmus a legkisebb fokszámú háromszöget választja ki
    - Több ilyen megegyező fokszámú háromszög esetén szomszédos háromszögek szomszédjainak a fokszáma alapján dönt
    - Ha ezek után még mindig nem egyértelmű a háromszög kiválasztása, akkor tetszőlegesen kiválaszt egyet
- A sávot a háromszögsáv kezdő és végső háromszögének az irányba terjesztjük ki
    - Az elkülönített háromszögek nem szerepelnek egyetlen egy háromszögsávban sem
    - Ezeknek a háromszögeknek a számát minimalizálja
- Lineáris idejű algoritmus megvalósítható
    - Hash táblák segítségével szomszédsági adatokat tárolására
    - Prioritási sorokkal, amelyeket mindegyik új sáv kezdő háromszögének keresésére használunk fel
- Tetszőleges háromszöget választva az algoritmus során ugyan olyan jó eredményt kaphatunk

- Praktikus szempont
  - A háromszögek irányítottságát meg kellene őrizni
  - A háromszögsáv első háromszöge határozza meg a háromszögek körbejárását
- Mi történik abban az esetben, ha a körbejárás megváltozik a kiterjesztés során?

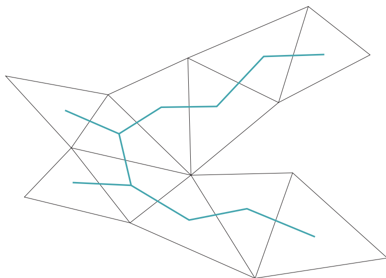


T_3 háromszög órajárással ellentétes körbejárású



T_0 háromszög órajárással megegyező körbejárású

- A háromszög-háló duális gráfjának a használata
- A gráf élei a szomszédos lapok középpontjait összekötő élei lesznek
- Ezen élek gráfját *feszítőfának* nevezzük
    - A jó háromszögsávok keresésére használhatunk fel

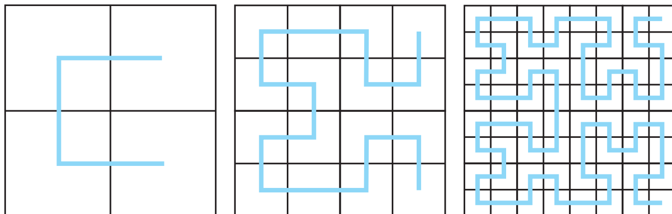


- Vertexek listájából és körvonalak halmazából állnak
- Minden vertex pozíció és további adatokat tartalmaz
    - Diffúz és spekuláris színeket
    - Árnyalási normálvektort
    - Textúra-koordinátákat
- Mindegyik háromszög körvonal egész értékű indexek listájával rendelkezik
- Ezek az indexek a vertexekre mutatnak a listában

- Tételezzük fel, hogy a háromszöghálókat háromszögsávokkal hoztuk létre
  - A grafikus kártyák többségének van vertex puffere
  - A csővezetéken átküldendő sávok sorrendjei különböző vertex puffer teljesítményt fognak mutatni
    - A gyorsítók különböző tármérettel rendelkeznek
- Előfeldolgozási művelettel javíthatjuk a sorrendet
  1.) Vegyünk egy háromszögsávot, melynek a vertexeit a vertex gyorsítótár (FIFO) egy szoftveres szimulációjában helyezzük el
  2.) A fennmaradó háromszögsávok közül kiválasztjuk azt, amelyik a legjobban használja ki a tár tartalmát
  3.) Az eljárást addig ismételjük, amíg az összes háromszögsávot fel nem dolgoztuk
- Az NVIDIA NVTriStrip függvénykönyvtár is pontosan ezeket a lépéseket követi
  - http://developer.nvidia.com/object/nvtristrip_library.html

- Adott egy indexelt háromszög háló
    - Amely hatékonyan renderelhető egy primitív vertex gyorsítótárral rendelkező grafikus hardver segítségével
- A kérdés továbbra is az indexek megfelelő sorrendjének meghatározása
    - Különböző hardverek különböző méretű vertex gyorsítótárral rendelkeznek
    - Mindegyik méretre más és más módon állítjuk elő az indexek sorrendjét?!?
- Az igazi megoldás egy *univerzális* index sorozat előállítása
    - Az összes lehetséges vertex gyorsítótár méret esetén jól viselkedik

- A *terület-kitöltő görbe* fogalma
  - Egy egyszerű, folytonos görbe
  - Nem metszi önmagát
  - Kitölt egy olyan négyzetet vagy téglalapot, amely uniform négyzetrácsait csak egyszer érint annak bejárása során
- A jó terület-kitöltő görbe jó térbeli összefüggőséggel rendelkezik
  - Amikor bejárjuk azt mindig az előzőleg meglátogatott pontok közelében maradunk
  - A Hilbert görbe egy példa a terület-kitöltő görbére



- A háromszög háló esetén a vertex gyorsítótárban nagy találati arányra számíthatunk terület-kitöltő görbe használatakor
- Egy rekurzív algoritmussal jó index sorozatot lehet létrehozni a háromszöghálókra
- Alapötlet
    - Szétvágjuk a hálót megközelítőleg két egyforma méretű hálóra
    - Majd az egyik illetve a másik hálót rendereljük le
    - Ezt ismételjük rekurzívan minkét hálóra
- A háló szétvágását előfeldolgozási lépésként hajtjuk végre kiegyensúlyozott él-vágás algoritmussal
    - Minimalizálja az él vágások számát
- Az algoritmus komplexitása lineáris a hálóban lévő háromszögek számára nézve

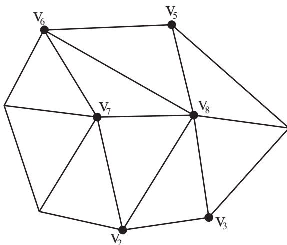
1.) Hozzuk létre a háromszögháló duális gráfját.
2.) Ezután a kiegyensúlyozott él-vágás algoritmussal eltávolítjuk a minimális élek halmazát a duális gráfban
    - A hálót két különálló, nagyjából megegyező méretű hálóra vágjuk szét
3.) A jó gyorsítótár teljesítmény eléréséhez ajánlott, hogy az utolsó háromszög az első hálóban közel legyen a második háló első háromszögéhez.
    - Az első háló utolsó háromszöge és a második háló első háromszöge esetén megengedjük, hogy a duális gráfban egy élen osztozzanak, amelyet átvágtunk

- A *háló egyszerűsítéssel adat csökkentésként* vagy *decimálásként* is találkozhatunk
  - Egy részletes modell poligonjainak a számának csökkentése
  - Megpróbálja megőrizni annak megjelenését
- Valósidejű munka során ez az eljárás a csővezetéken átküldendő vertexek számát csökkenti
  - Fontos lehet az adott alkalmazás régebbi számítógépeken való futtatásakor
- Modellek redundánsak lehetnek egy elfogadható megjelenítés esetén is

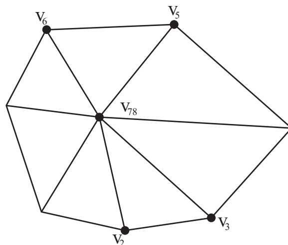
- Poligon egyszerűsítési technikát
  1.) Statikus
     - Elkülönített részletességi szinteket (Level of Detail, röviden LOD) hozunk létre még a renderelés előtt és a megjelenítő ezek közül választ
  2.) Dinamikus
     - Folytonos spektruma az LOD modellek néhány diszkrét modelljével szemben
  3.) Nézőpont-függő
     - Például egy terep renderelésekor a közeli területeket részletesebben, míg a távolabbiakat a távolságtól függően kisebb részletességgel jelenítjük meg

- Az LOD algoritmusoknak 3 fő része van
  - Generálás
    - Egyszerűsítéssel
    - Kézzel
  - Kiválasztás
    - Valamilyen szempont alapján az LOD modell kiválasztása
    - A képernyőn lévő becsült terület alapján például
  - Váltás
    - LOD szintek közötti váltás

- Az egyik módszer a poligonok számának csökkentésére az élek összevonása művelet
- Két vertex egybe olvasztásával lehet elvégezni



A $\mathbf{v}_6\mathbf{v}_7$ és $\mathbf{v}_6\mathbf{v}_8$ illetve a $\mathbf{v}_2\mathbf{v}_7$ és $\mathbf{v}_2\mathbf{v}_8$ élek összevonása



Eltűnik a $\mathbf{v}_7\mathbf{v}_8$ él, a $\triangle\mathbf{v}_6\mathbf{v}_7\mathbf{v}_8$ és $\triangle\mathbf{v}_7\mathbf{v}_2\mathbf{v}_8$ háromszögek

- Az élek összevonása művelet visszafordítható
- Él összevonásokat rendezve, az egyszerűsített modellből kiindulva visszaállíthatjuk az eredeti modellt
- A $\mathbf{v}_7$ és $\mathbf{v}_8$ vertexeket összeolvasztottuk $\mathbf{v}_{78} = \mathbf{v}_7$ vertexbe
  - A másik vertexbe való olvasztás is ($\mathbf{v}_{78} = \mathbf{v}_8$) elfogadható lett volna
- Ha korlátozzuk a lehetőségek számát, akkor értelemszerűen kódolhatjuk az aktuálisan végrehajtott választást

- Az *optimális elhelyezés* eléréshez több lehetőséget kell megvizsgálni
- Ahelyett, hogy az egyik vertexet a másikba olvasztanánk az élen lévő mindkét vertexet egy új pozícióban húzunk össze
  - Jobb minőségű háló jön így létre
  - Hátránya, hogy több műveletet kell végrehajtani és több memóriát is kell felhasználni a nagyobb területen való elhelyezési lehetőségek kiválasztására
- Bizonyos vertex összeolvasztásokat a költségekre való tekintet nélkül el kell kerülni
  - Például konkáv alakzatok esetén a vertex összeolvasztás után egy új él a poligonon kívülre kerül
    - A szomszédos poligonok normálvektorának az iránya megfordul-e az összeolvasztás következtében

- Garland és Heckbert alap célfüggvénye
  - Egy adott vertexhez megadhatjuk azon háromszögek halmazát, amelyeknek eleme ez a vertex és mindegyik háromszöghöz adott a sík egyenlete
  - A célfüggvény egy mozgó vertexre a síkok és az új pozíció távolságainak négyzet összegei

$$c(\mathbf{v}) = \sum_{i=1}^{m} (\mathbf{n}_i \cdot \mathbf{v} + d_i)^2$$

  - A **v** az új pozíció
  - Az **n** az adott sík normálvektora
  - *d* az eredeti pozícióhoz viszonyított eltolási értéke

1.) Képzeljünk el két háromszöget, amelyek egy közös éllel rendelkeznek.
- Ez az él egy nagyon éles él, amely pl. egy turbina lapát része lehet.
- A célfüggvény értéke a vertex összeolvasztás esetén ebben az esetben alacsony
  - Az egyik háromszögön csúszó pont nem kerül távol a másik háromszög síkjától
- Egy olyan síkot veszünk hozzá az objektumhoz, amely tartalmazza az élet és az él normálvektora a két háromszög normálisának az átlaga
  - Azoknál a vertexeknél, amelyek nagyon eltávolodnak ettől az éltől, nagyobb költség függvény értéket fogunk kapni

2.) A költség függvény másik fajta kiterjesztése másfajta felületi tulajdonságok megőrzésére szolgál.

- Például a modell gyűrődési és határ élei fontosak a megjelenítésben, ezért kisebb mértékben szabad csak azokat módosítani
- Érdemes más felületi tulajdonság esetén is megőrizni a pozíciókat
    - Változik az anyag
    - Textúra élek vannak és változik a vertexenkénti színek száma

3.) Leginkább azokat az éleket érdemes összevonni, amelyek a legkisebb észlelhető változást eredményezik.

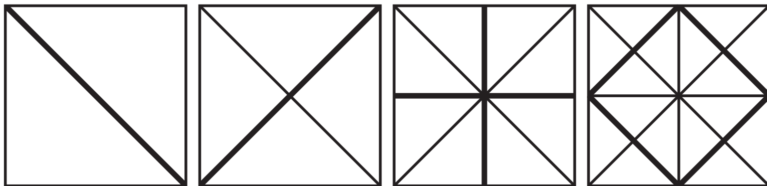
- Kép-vezérelt függvényt használjunk ilyen esetek kezelésére
  - Az eredeti modellből különböző nézetből (mondjuk 20), legyártjuk a képeket
  - Ezután minden potenciális élre kipróbáljuk az összevonásokat az adott modell esetén és előállítjuk a képeket, amelyeket összehasonlítjuk az eredetivel
  - Azt az élet vonjuk össze, amelyik vizuálisan a legkisebb különbséget adja
- Ennek a célfüggvény értékének a kiszámítása igen drága és természetesen nem valósítható meg valós-időben
  - Az egyszerűsítési műveletet el lehet végezni előzetesen, amelyet később fel lehet használni

- Komoly problémát jelent az egyszerűsítési algoritmusoknál az, hogy gyakran a textúrák észrevehetően eltérnek az eredeti megjelenésüktől
  - Ahogy az élek eltűnnek, a felület mögött lévő textúrázási leképezés eltorzulhat
- A poligon csökkentési technikák hasznosak lehetnek
  - Egy tehetséges modellező létrehozhat egy alacsony poligon számú modellt, amely minőségben sokkal jobb, mint az automatikus eljárással előállított
  - A legtöbb redukáló algoritmus nem tud semmit a vizuálisan fontos elemekről vagy a szimmetriáról
    - Például a szemek és az orr a legfontosabb része az arcnak
    - Egy naiv algoritmus elsimítja ezeket a területeket, mivel lényegtelenek

- A terep az egyik olyan modell típus, amelyik egyedi tulajdonságokkal rendelkezik
- Az adatokat rendszerint egyenletes rácson megadott magassági értékekként tárolják el
- A nézőpont-függő módszerek általában valamilyen feltételben megadott kritérium határértékéig folytatják az egyszerűsítést
- Élösszevonás kiegészítve egy célfüggvénnyel, ami a nézőpontot is figyelembe veszi
    - A terepet nem egy egyszerű hálóként kell ilyen esetben kezelni, hanem kisebb részterületekre bontva

- Algoritmusok másik osztálya
  - A magasságmező rácsból származtatott hierarchikus adatstruktúrát használ
  - Egy hierarchikus struktúrát építünk fel az adatok felhasználásával és kiértékeléskor pedig csak a bonyolultság szintjének megfelelően állítjuk elő a terep felszínt
- Hierarchikus struktúra lehet a bináris háromszögfa
  - Egy nagy, jobb háromszöggel kezdődik a magasságmező darab sarkaiban lévő vertexekkel
  - Ez a háromszög felosztható az átfogón lévő középpont és a szemközti sarokpont összekötésével
  - A felosztást addig folytathatjuk, amíg el nem érjük a magasságmező rácsának a részletességét

- Bináris háromszögfa létrehozása
  - A baloldalon a magasságmező két háromszöggel van közelítve
  - A következő szinteken, mindegyik háromszög újból ketté van osztva
  - Az osztásokat a vastag szakaszok jelölik



- Mindegyik háromszöghöz előállítunk egy hibahatárt
  - Ez a hibahatár fejezi ki azt a maximális mennyiséget, amivel a magasságmező eltérhet a háromszögekkel kialakított síktól
  - A hibahatár és a háromszög együtt határozzák meg egy torta-alakú szeletét a térnek
    - Tartalmazza a teljes terepet, amely kapcsolatban áll ezzel a háromszöggel
  - A futás alatt ezeket a hibahatárokat leképezzük a nézősíkra és kiértékeljük a megjelenítésre kifejtett hatásukat

Összefoglalás

Témakörök

- Display Listák
  - Állapot változások
- Vertex tömbök
  - Változó vertex pozíciók
- Vertex pufferek
- Poligon technikák
  - Háromszögsávok és hálók
  - Poligon egyszerűsítési technikák