GRAFIKA DOKUMENTÁCIÓ

by Blázsovics Bence

OGL Háttér (Főleg Lacinak, mert tudom hogy érdekel az Ogl 3.3+) a projekthez nem kell, skip ha sietsz

Mindent megtalálsz a learnopengl.com-on részletesebben elmagyarázva, de talán ettől kedvet kapsz megint belenézni :3 – pár sor kód innen lett másolva, pl a Shaderek kezelése.

Minden VAO-kkal van rajzolva. VAO = vertex array object, egy tömb a vidkártyán, amiben megvannak a modelled vertexei. Ennek sok paramétere van, pl hogy mekkora az egész byteokban, mennyi float van benne, hogy olvassa ki a pontokat stb.

pl:

```
glGenVertexArrays(1, &vao);
glGenBuffers(1, &vbo);

glBindVertexArray(vao);

glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER, sizeof(data) * num_floats, data, GL_STATIC_DRAW);

glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 5 * sizeof(float), (void*)0);
```

pozíció x,y,z 3 float ^

```
glEnableVertexAttribArray(0);

glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 5 * sizeof(float), (void*)(3 * sizeof(float)));
```

textúra az 1-es attrib-ban, 2 float ^ így az egész vertex sablon 5 float nagy, 3mat kell az elejétől kihagyni

```
glEnableVertexAttribArray(1);
```

ebben csinálsz egy vao-t, a vao-hoz kell egy VBO vertex buffer object.

a vao-ban lehet egyéb is, ezért kell ez, ami megmondja hogy ebbe most pontok mennek~

glBufferData mondja meg, hogy mennyi adat, és hol az adat.

glVertexAttribPointer mondja meg hogy a rengeteg floatot amit kap, hogy kell olvasni.

pl pozíciót mindig kap, de kaphat még textúra koordinátákat, és irányvektorokat megvilágításhoz – itt mondod meg, hogy milyen sorrendben jönnek ezek, és hogy mennyi van belőlük.

paraméterek sorrendben: attrib id; mennyi érték ebből egy vertexben; milyen típus(float); kell-e a GPU-nak ezt normalizálnia, az egész feltötlendő vertex sablon mérete; és az egész sablon elejétől hány byteot kell kihagyni (stride) – hol kezdődik ez az attribútum.

Shader – innentől ezek kellenek, ha nélkülem akarsz valami extrát rajzolni

A shader az ami megmondja hogy mégis hogy jöjjön ki a szín a képernyőre abból amit megetettünk a GPU-val. Ennek programozására van egy külön C-szerű nyelv (GL shading language), ami erősen tánogatja a vektorműveleteket. Ezt látni a shader mappában 2 kis txt-ben. A programon belül csak „waterShader" névre hallgat. Külön van geometry, vertex, és fragment shader.

A geometry teljesen opcionális, új vertexeket csinál adott adatból – a GPU-n generál adott kód szerint. Ez elég advanced dolog, de ezen szoktak pl subdivide-olni.

A vertex shader azért felel, hogy az adott vertexed jó helyen legyen, megeszi az MVP mátrixot, és beszorozza vele a pozíciókat.

A fragment shader a színért felel – a waterShaderben pl megírtam hogy ha egy pixel túl sötét, legyen átlátszó, de emelett elfogad színezést (tint) és custom átlátszóságot – ezzel működik a menü, és ezért kell csak 1 player textúra.

Háttér meta & math lite

A matek a glm (open gl mathematics) könyvtár által működik, mivel ennek nyelvezete erősen hasonlít a GLSL-hez.

A pozíciót a model M mátrix határozza meg – ezt lehet

> vektorral eltolni M=glm::translate(M, vec3);
>
> átméretezni M=glm::scale(M, vec3); - minden tengelyre
>
> vagy elforgatni M=glm::rotate(M, glm::radians(360-fok), glm::vec3) – ebben a vektor mindenképp bázis és normál.

A 3D renderhez szükséges az MVP mátrix = model-view-projection; ezeket fordítva szorozzuk össze, mert a műveletek jobbról jönnek sorrendben. MVP = projection * view * model.

A projection konstans, ez felel a prespektívikus projekcióért. Ferdíti a messzebb lévő vonalakat – pl egy sínbe nézve azok a végtelenben találkoznak.

A view a kamera mátrixa - egyfajta inverz model mátrix – mindent az ellenkező irányba tol el, mint amerre az egeret mozdítjuk. (Nem a „kamerát forgatjuk", kamera csak virtuál-virtuálisan létezik :D – hanem az egész világ fordul az ellenkező irányba). Ezt framenként frissíteni kell.

vektor definíció: glm::vec3 vektor; - vektor 3 floatból

könnyű komponens elérés vektor.x = vektor.y stb

NOTE: a pozíciókban a Z-t békén hagyjuk! azzal a dolgok átmennek a kamerán vagy be a map mögé

> Z = 0.0f;

a playerek pozíciója a glm::vec3 global_player_positions[3]; tömbhöz van kötve, amiből a 0 mindig a saját, és az én kódomban a kamera mozgásához van kötve. ez könnyen átírható, csak szóljatok egymásnak.

Fő függvények lényegében az én részem

void drawPlayer(unsigned id, VAO &vao, float wave, float size); - id szerinti player rajzolás, működik 0 local playerre is

void drawLocalPlayer(VAO &vao, float wave, float size); - csak saját player rajzolása. főleg teszteléshez

void drawMenu(VAO &_screen, VAO &_play_button, VAO &_options_button, VAO &_options_screen, VAO &_volume_slider); - magáért beszél

void drawMap(VAO &_floor, VAO &_water, VAO &_grass, VAO &_sky); - magáért beszél

void drawPickup(VAO &_pickup, glm::vec3 _pos); - felvehető objektum (3szög) rajzolása

void drawVictory(VAO &_vic0, VAO &_vic1, VAO &_vic2, unsigned id); - magáért beszél

void drawDisconnect(VAO &_disc); - hálózati hiba esetére :3

Egyéb

A menü állapotát egy karakter tartja számon, amelynek 4 állapota van.:

P --play gomb

O -- options gomb

V – volume slider

W – other slider – a csúszkáknak ilyen néven saját float értékeik vannak 0.0 – 1.0