Etapa 1 – Motoare de grafica 3D in timp real

Task-uri:

1. API 3D : OpenGL
2. Format Intern propriu de Mesh-uri pentru obiecte
    a. Triunghiuri
    b. Materiale
    c. Textura
    d. Diverse (normale, coordonate de textura, pozitii si culori)
3. Modul de 3D Math
4. Gestiune interactiune cu utilizatorul
    a. Input/Output Events
    b. Mouse
    c. Keyboard
5. Implementare Camera de tip FPS

Am ales folosirea API-ului grafic OpenGL cu framework-ul 3.3 Core (GLFW3) si Multi-Language Loader Generator (GLAD) ca librarie de incarcare pentru OpenGL.

Pentru crearea mesh-urilor am creat un shader care sa citeasca un `const char*` pentru datele din din fisierele shader de pe disk. Mai departe se citesc, se compileaza, se ataseaza unul la celalalt, se link-eaza la un `GLuint`, se compileaza ca shader finala si se sterg `GLuint`-urile pentru fostele componente ale shader-ului final.

Mai departe am creat un VAO si l-am legat la placa grafica.

Dupa aceasta am creat un VBO de dimensiunea unui vector de float-uri vertices[] si un EBO de dimensiunea unui vector de float-uri indices[].

Dupa aceasta, incarcat vertecsii in VBO-uri: Sursa VBO-urilor a fost vectorul vertices[] in care am pus coordonatele de pozitie, coordonatele de textura, normalele si culorile. In acest mod, fiecare vertex are 3 float-uri pentru pozitie (XYZ), 2 pentru coordonate de textura (UV), 3 pentru normale (XYZ) si 3 pentru culori (RGB). In total voi avea 11 float-uri pentru fiecare vertex. In concluzie, pentru un patrat voi avea 44 de float-uri.

Pentru EBO, am incarcat indicii in EBO-uri: In vectorul indices[] voi avea niste indici pentru fiecare vertex care specifica ordinea in care se vor desena triunghiurile mesh-ului. Deci, pentru un patrat voi avea 6 indici, pentru ca un patrat este format din 2 triunghiuri, iar un triunghi din 3 vertecsi.

In mod normal se poate desena si cu un VBO si VAO, dar EBO-ul este util pentru ca ii spune GPU-ului ce drepte sa omita de la denesare, daca s-au mai desenat o data. De exemplu, pentru un patrat cu indici 0, 1, 2, 3 se deseneaza 2 triunghiuri cu indicii 0, 1, 2 si 0, 3, 2. In acest caz se vor trasa drepte de la 0-1, 1-2, 0-2 si 0-3, 3-2, si 0-2, si se va umple continutul din interiorul acestui poligon inchis. Se observa ca 0-2 este comun la ambele triunghiuri; scopul EBO-ului este sa nu rasterizeze de 2 ori aceeasi dreapta.

Mai departe am legat VBO-ul la VAO in 4 locatii diferite corespunzatoare pozitie, coordonatelor de textura, normalelor si culorilor pe GPU. Diferentierea a facut-o functia `glVertexAttribPointer`

care are ca parametri locatia pe vertex shader, dimensiunea unui poligon inchis – 3 pentru triunghiuri, tipul de date folosite – float in cazul meu, daca datele sunt normalizate – la mine nu, pentru ca este important ca, coordonatele de pozitie sa fie si in spatiul negativ, stride-ul (care specifica din cati in cati octeti sa sara VBO-ul ca sa citeasca datele urmatorului vertex) – `11 * sizeof(float)`, adica 44 octeti la mine si pointer-ul de start (care reprezinta deplasamentul de la adresa vectorului pana unde este situata locatia de interes) – spre exemplu pentru normale este `(void*)(5 * sizeof(float))`.

Mai departe am dezlegat VAO, VBO si EBO de la GPU si le-am sters din memorie cand se inchide fereastra engine-ului.

In vertex shader pozitia se foloseste la calcularea gl_Position = vec4(vPos, 1.0), unde vPos sunt pozitiile vertecsilor primite de la VBO. Se trimite la fragment shader culoare ca uniforma sau pentru fiecare vertex. Se mai trimit si coordonatele de textura la fragment shader.

In fragment shader se calculeaza culoarea finala inmultind valorile din coordonatele de textura cu culoarea uniforma (sau culoare de vertecsi) sau folosind functie mix() intre textura si culoare/culori.

Revenind la initializare date, texturile sunt create, legate la GPU, iar mai apoi se genereaza mipmap-uri cu 2 functii supraincarcate de mine, in una se specifica modul de repetare a texturilor ca sa umple spatiul gol, in una se specifica o culoare, in cazul in care se doreste folosirea metodei de repetare `GL_CLAMP_TO_BORDER` care necesita pe deasupra si o culoare cu care sa deseneze marginea cu care va fi inconjurata textura.

Mai apoi se dezleaga si textura si se sterge cand se termina programul.

Nu in ultimul rand, am creat `GLuint`-uri pe care le-am legat la locatiile uniformelor de pe shadere.

Stadiile principale ale programului sunt:

- Init
- Run
  - BeforeDrawing
  - Draw
  - AfterDrawing
- End

In Init sunt procesarile precizate anterior (crearea de bufferi, texturi si locatii uniforme).

In End se afla stergerea obiectelor si eliberarea memoriei.

In BeforeDrawing se sterge ecranul si buffer-ul de culoare.

In AfterDrawing se face buffering (se schimba buffer-ul din fata cu cel din spate) si se trateaza input-urile de la GLFW.

In Draw se deseneaza mesh-ul.

Toate aceste 3 metode din Run formeaza main loop-ul si se repeta la infinit cat timp fereastra este deschisa.

Pentru folosirea unui model 3D si nu 2D se schimba cateva detalii:

Se adauga vertecsii si indicii corespunzatori in vectorii vertices[] si indices[].

In BeforeDrawing, pe langa stergerea buffer-ului de culoare se sterge si buffer-ul de adancime (ca sa nu apara glitch-uri vizuale cauzate de faptul ca OpenGL nu stie care triunghi este randat in fata si care este in spate).

In Run se ruleaza `glEnable(GL_DEPTH_TEST)`.

Se calculeaza matricea de vizualizare, proiectie si matricea modelului.

Matricea modelului se foloseste ca sa converteasca coordonatele locale ale obiectului in coordonate globale. Eu am initializat-o ca matrice identitate, pentru ca am considerat ca obiectul se spawneaza in originea lumii.

Matricea de vizualizare se foloseste ca sa converteasca din coordonate globale in coordonate de vizualizare. Aici se tine cont si de rotatia camerei. Nu am vorbit inca despre camera, dar voi mentiona ulterior. Eu am calculat-o in functie de pozitia si orientarea camerei.

Matricea de proiecte se foloseste ca sa converteasca din coordonate de vizualizare in coordonate de clipping, adica se tine cont si de perspectiva obiectului, si de range-ul de clipping al camerei. Eu am calculat-o cu un camp de vizualizare de 45 grade si un range intre [0.1, 100] – in afara range-ului facandu-se clipping.

Trecerea din coordonate de clipping in coordonatele ferestrei se face automat de catre OpenGL.

Calcularea matricilor si trimiterea uniformelor la GPU se face in Draw, inainte de desenarea mesh-ului.

Folosind toate aceste detalii se ajunge la modelul tridimensional.

Camera am initializat-o la pozitia (0, 0 2) – se tine cont ca pe axa OZ, valorile negative sunt mai in fata, iar valorile pozitive sunt mai in spate. Deci camera se afla plasata la o distanta de 2 unitati in spatele obiectului. Orientarea camerei este (0, 0, -1), adica in fata (catre obiect). Camera se mai initializeaza cu 2 float-uri care configureaza valorile initiale pentru sensibilitatea camerei si viteza de deplasare. Camera se initializeaza cu aceleasi dimensiuni ca si fereastra.

Mai departe am definit tratarea input-urilor pentru camera folosind glfwGetKey, `GLFW_PRESS` si `GLFW_RELEASE`. Input-urile functioneaza clasic: WASD pentru deplasare, SPACE pentru deplasare in sus, LEFT CTRL pentru deplasare in jos, si miscari de mouse pentru rotire. Cat timp LEFT SHIFT este apasat camera se va deplasa cu o viteza mai mare, iar cand nu este apasat cu o viteza mai mica. Input-urile pentru camera se vor trata numai cand CLICK DREAPTA este apasat. Rotirea pe verticala va fi limitata intre 5 si 175 grade, pentru ca nu vrem ca, camera sa se dea peste cap. Cursorul se va forta in mijlocul ferestrei si se va ascunde cat timp se trateaza input-urile pentru camera, pentru ca miscarea lui contribuie la miscarea camerei.

Modulul de algebra liniara l-am implementat cu glm `glm.hpp`, `matrix_transform.hpp`, `type_ptr.hpp`, `rotate_vector.hpp` si `vector_angle.hpp`.

Toata aceasta logica este precedata si urmata de „motorul” motorului grafic – care trateaza initializarea GLFW si crearea ferestei, si la sfarsit distrugerea ferestrei si terminarea framework-ului GLFW.

Pentru toate componentele importante pe care le-am enuntat am creat cate o clasa: `Engine`, `World`, `Shader`, `VAO`, `VBO`, `EBO`, `Texture`, `Mesh` si `Camera`.