SREDNJA ELEKTROTEHNIČKA ŠKOLA

SARAJEVO

MATURSKI RAD

TEMA: 3D renderer - C++

Mentor:

Adnan Delić dipl. el. ing.

Učenik:

Mubarak Appa Bugis

Sarajevo, April 2024

Sadržaj

.

1 Uvod

1.1 Pojam

3D renderiranje je tehnika obrade grafike koja se koristi za pretvaranje trodimenzionalnih modela u dvodimenzionalne slike, koje se zatim prikazuju na ekranima računara ili drugim uređajima. Ova tehnika je ključna u stvaranju vizualno impresivnih igara, filmova, simulacija i drugih digitalnih sadržaja.

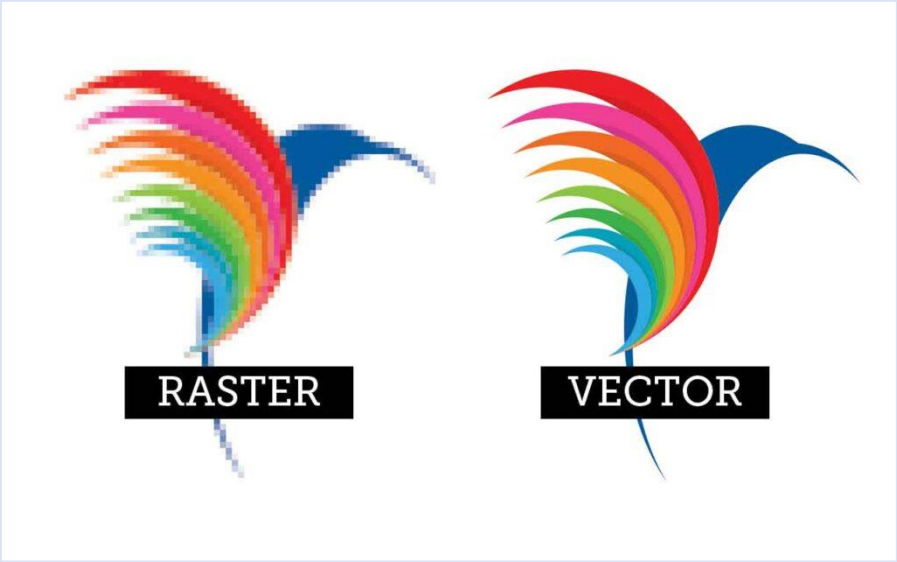
1.2 Historija

Historija 3D renderiranja seže u početke razvoja računarske grafike. Iako se koncepti 3D grafike razvijaju još od 1960-ih godina, prve primjene 3D renderiranja pojavile su se tokom 1970-ih i 1980-ih godina sa razvojem računarskih igara i filmske industrije. U to vrijeme, ograničeni resursi računara ograničavali su kompleksnost i kvalitetu 3D grafike.

Međutim, sa tehnološkim napretkom, posebno sa razvojem grafičkih kartica posvećenih obradi grafike (GPU), 3D renderiranje je ostvarilo značajan napredak. Razvoj tehnika poput teksturiranja, osvjetljenja, sjene i anti-aliasinga omogućio je stvaranje sve realističnijih 3D scena.

Danas, 3D renderiranje je osnova za stvaranje realističnih vizualnih iskustava u video igrama, filmskoj industriji, medicinskim simulacijama, arhitektonskom dizajnu, inženjeringu i mnogim drugim područjima. Neprestani tehnološki napredak, uključujući razvoj tehnika poput globalne iluminacije, fizikalno baziranog rendiranja i praćenja zraka, nastavlja poboljšavati kvalitetu i realizam 3D grafike, otvarajući vrata novim mogućnostima i aplikacijama.

1.3 Tipovi



Slika 1: Razlika između vektorske i rasterske slike

Dvije osnovne vrste 3D renderiranja su vektorsko i rastersko (ili "pikselizirano") renderiranje.

Vektorsko renderiranje se temelji na matematičkim definicijama geometrijskih oblika, poput linija, krivulja i poligona. Ovaj pristup omogućuje skaliranje slika bez gubitka kvaliteta, jer se objekti opisuju pomoću vektora, a ne piksela. Vektorsko renderiranje se često koristi u dizajnu logotipa, arhitektonskih planova i dijagrama, gdje preciznost i skalabilnost igraju ključnu ulogu. Međutim, vektorsko renderiranje nije prikladno za stvaranje kompleksnih vizualnih efekata ili realističnih prikaza, zbog čega se koristi kao dodatak rasterskom renderiranju.

Rastersko renderiranje, s druge strane, koristi piksele kao osnovnu jedinicu za stvaranje slike. Svaki piksel dobiva vrijednosti za boju, sjenu, osvjetljenje i druge karakteristike u skladu s pozicijom i svojstvima 3D objekata u sceni. Ova tehnika je dominantna u stvaranju realističnih 3D slika u video igrama, filmovima i drugim digitalnim medijima. Iako stvara statične slike s fiksnom rezolucijom, rastersko renderiranje omogućava bogatije vizualne efekte i detalje, što ga čini ključnim alatom za kreiranje vizualno impresivnih iskustava.

2 C++ programski jezik

2.1 Historija

C++ je programski jezik opće namjene i srednje razine s podrškom za objektno orijentirano programiranje. Prvobitno je razvijen u Bell Labs (laboratoriju telekomunikacijske tvrtke Bell) pod rukovodstvom Bjarnea Stroustrupa tokom 1980-ih, i to kao proširenje programskom jeziku C pa mu je originalno ime bilo "C with classes" (eng. C s klasama). Zbog velike potražnje za objektno orijentisanim jezicima te izrazitim sposobnostima istih, specifikacija programskog jezika C++ ratificirana je 1998. kao standard ISO/IEC 14882.

Bjarne Stroustrupov poticaj za stvaranje novog programskog jezika C++ proisteklo je iz njegovog rada na dokorskoj disertaciji u kojem se susreo s dilemom gdje je programski jezik Simula bio dobar za složene programske projekte, dok je programski jezik BCPL bio brz ali je bio na jako niskoj razini da bude praktičan za primjenu. U listopadu 1985. godine prva komercijalna distribucija jezika predstavljena je javnosti u knjizi The C++ Programming Language čiji je autor spomenuti Bjarne Stroustrup.

2.2 Objekti

C++ uvodi karakteristike objektno orijentisanog programiranja (OOP). Nudi klase koje obezbjeđuju dvije karakteristike uobičajeno prisutne u OOP (i nekim ne-OOP) jezicima: enkapsulaciju i nasljeđivanje.

2.3 Enkapsulacija

Enkapsulacija je skrivanje informacija kako bi se osiguralo da se strukture podataka i operatori koriste kako je predviđeno i kako bi se model upotrebe učinio očiglednijim programeru. C++ pruža mogućnost definiranja klasa i funkcija kao svojih primarnih mehanizama enkapsulacije. Unutar klase, članovi mogu biti deklarirani kao javni, zaštićeni ili privatni da bi se eksplicitno nametnula enkapsulacija. Javni član klase dostupan je bilo kojoj funkciji. Privatni član je dostupan samo funkcijama koje su članovi te klase i funkcijama i klasama kojima je klasa eksplicitno dodijelila dozvolu pristupa ("prijatelji"). Zaštićeni član je dostupan članovima klasa koje nasljeđuju klasu pored same klase i svih prijatelja.

2.4 Nasljeđivanje

Nasljeđivanje omogućava jednom tipu podataka da stekne svojstva drugih tipova podataka. Nasljeđivanje iz osnovne klase može biti deklarirano kao javno, zaštićeno ili privatno. Ovaj specificator pristupa određuje da li nepovezane i izvedene klase mogu pristupiti naslijeđenim javnim i zaštićenim članovima osnovne klase. Samo javno nasljeđe odgovara onome što se obično podrazumijeva pod "nasljeđem". Druga dva oblika se mnogo rjeđe koriste. Ako je specifikacija pristupa izostavljena, "klasa" nasljeđuje privatno, dok "struktura" nasljeđuje javno. Bazne klase mogu biti deklarisane kao virtuelne; ovo se zove virtualno nasljeđivanje. Virtualno nasljeđivanje osigurava da samo jedna instanca bazne klase postoji u grafu nasljeđivanja, izbjegavajući neke od problema dvosmislenosti višestrukog nasljeđivanja.

2.5 Mingw-w64

Mingw-w64 je besplatno okruženje za razvoj softvera otvorenog koda za kreiranje (unakrsno kompajliranje) Microsoft Windows PE aplikacija. Razvijen je 2005–2010 od MinGW-a (Minimalistički GNU za Windows).

Mingw-w64 uključuje port zbirke GNU kompajlera (GCC), GNU Binutils za Windows (assembler, linker, arhivski menadžer), skup datoteka zaglavlja koje se slobodno distribuiraju za Windows i statične biblioteke za uvoz koje omogućavaju korištenje Windows API-ja, Windows izvorna verzija GNU Debuggera GNU projekta i razni uslužni programi. U projektu ima ulogu kao kompajler tj. da pretvori source kod u .exe fajl koji se može pokrenuti na Windows operativnom sistemu.

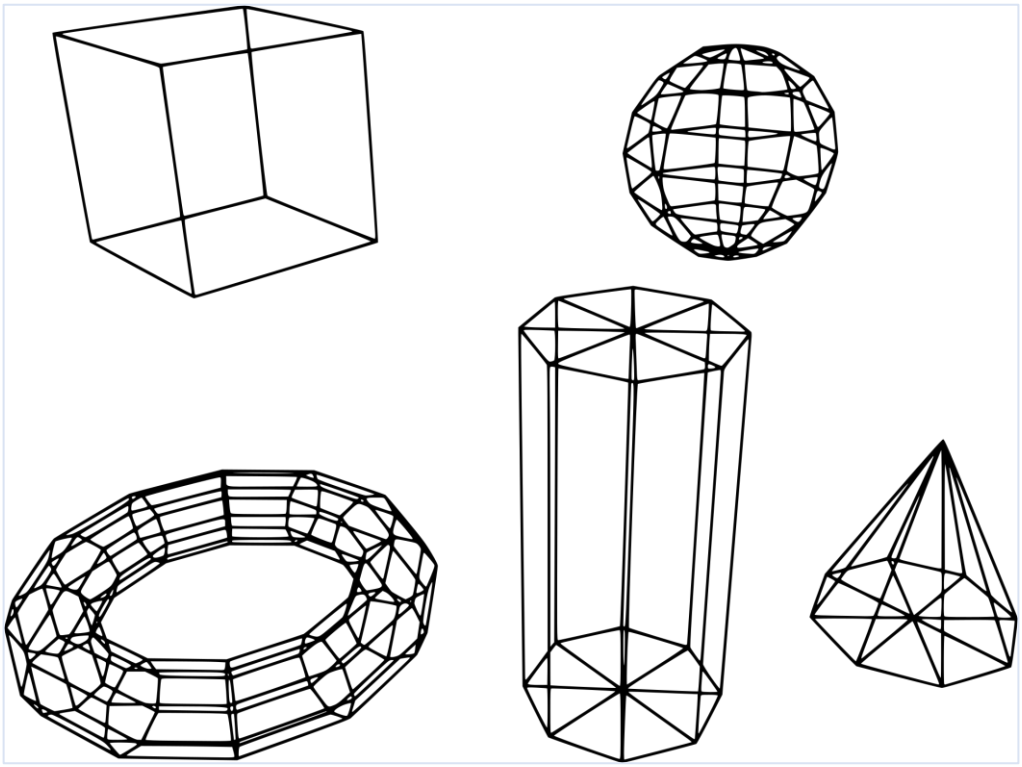
2.6 Windows.h header

Windows.h je datoteka zaglavlja specifična za Windows za programske jezike C i C++ koja sadrži deklaracije za sve funkcije u Windows API-ju, sve uobičajene makroe koje koriste Windows programeri i sve tipove podataka koje koriste različite funkcije i podsistemi. Definira veoma veliki broj Windows specifičnih funkcija koje se mogu koristiti u C-u. Win32 API se može dodati u C programski projekat uključivanjem datoteke zaglavlja <windows.h> i povezivanjem na odgovarajuće biblioteke. U projekat je uključen kako bi se mogla stvoriti API veza između Windows konzole i programa, te izvršiti manipulaciju konzole da bi se stvorio rasterizirani 3D pregled.

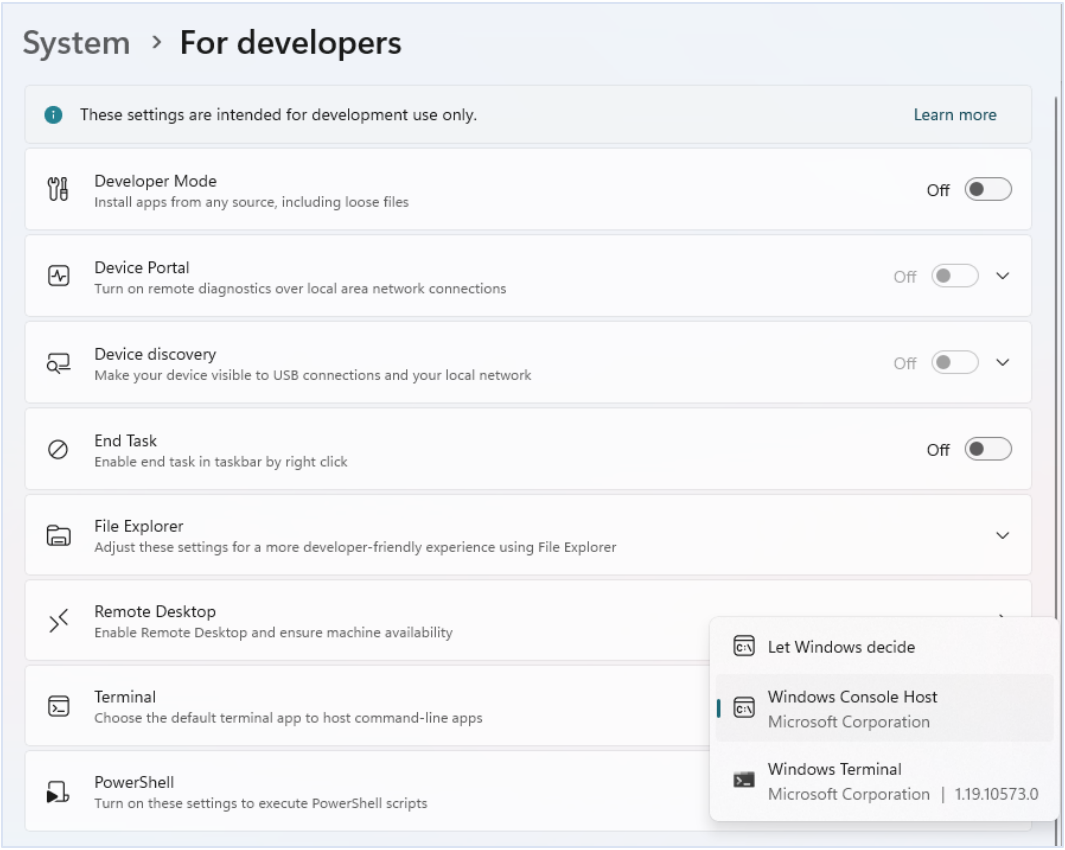
3 Projekat

3.1 Uvod

“3Drenderer” je jednostavan program koji može da prikaže **wire-frame model** (Slika 2) 3D objekta u Windows konzoli (Command prompt). Wire-frame model za razliku od standardnog rendera prikazuje samo stranice trougla, te površina trougla je izostavljena. Radi na principu **rasterizacije** i ima mogućnost prikazivanja Wavefront .obj fajlova. Za pokretanje je dovoljno pokrenuti prekompajlirani .exe fajl i napisati putanju .obj fajla. Treba se posvetiti pažnja da se promijeni default console host u Terminal Settings ako je host operativni system Windows 11 (Slika 3).



Slika 2: Wireframe model različitih geometrijskih oblika



Slika 3: Windows 11 Terminal Settings, neophodno je promijeniti terminal na "Windows Console Host"

3.2 Engine.h

"3Drenderer" je sastavljen od dva glavna fajla: Engine.h i 3Drenderer.cpp. Engine.h igra ulogu "Graphics Library", to jest ima najprostije funkcije i komponente kako bi se mogla nacrtati slika na ekranu. 3Drenderer.cpp je glavni program i on koristi funkcije definisane u Engine.h te matematičke transformacije kako bi mogao prikazati 3D objekte u Windows konzoli.

Engine.h sadrži "Engine" klasu sa sljedećim funkcijama:

1. Inicijalizatorska klasa **Engine()** – Pozove se kad se prvobitno kreira klasa u 3Drenderer.cpp
2. Deinicijalizatorska klasa **~Engine()** – Služi kako bi se bezbjedno mogla ukloniti klasa
3. **GetConsoleBufferDimensions()** – Vraća informaciju o dimenziji Windows konzole
4. **ScreenSize()** – Mijenja dimenziju Windows konzole i screen buffera
5. **FontSize()** – Mijenja veličinu teksta u Windows konzoli
6. **DrawCharacter()** – "Nacrta pixel" na određenoj lokaciji unutar screen buffera
7. **DrawLine()** – Vrši crtanje pixela na liniji između tačaka A i B u screen bufferu
8. **DrawTriangle()** – Vrši crtanje pixela na stranicama trougla ABC u screen bufferu
9. **DisplayFrame()** – Prekopira sadržaj screen buffera u Windows konzolu koja se prikazuje krajnjem korisniku

Također sadrži private varijable:

1. **outH** – Za povezivanje sa Windows konzolom
2. **csbi** – Čuva informacije o Windows konzoli
3. **screenBuffer** – Jednodimenzionalni niz koji sadrži trenutni frame koji će se prikazati na kraju. Procesor prvo sve renderuje u Screen Bufferu te nakon završetka ga prikazuje u Windows konzoli da bi se izbjegli nedovršeni frameovi.

i javne varijable **columns** i **rows** Windows konzole.

3.2.1 Engine(), ScreenSize(), FontSize(), GetConsoleBufferDimensions()

Kreiranjem Engine klase poziva se prvobitna funkcija Engine() sa parametrima fontW, fontH (veličina „piksela“ na ekranu), screenW i screenH (rezolucija ekrana po broju piksela). Prvo se dohvati output Windows konzole handle funkcijom GetStdHandle i sačuva u varijabli outH. Nakon toga se pozove funkcija FontSize() koja namjesti veličinu teksta prikazano u konzoli i ScreenSize() koja mijenja veličinu Windows konzole.

Funkcijom GetConsoleBufferDimensions() se osvježavaju varijable rows i columns sa broj redova i kolona trenutno prikazano u Windows konzoli.

SetConsoleActiveScreenBuffer() je funkcija Windows.h header fajla i ona postavlja outH handle kao aktivni screen buffer preko čega ćemo moći mijenjati frameove ekrana konzole.

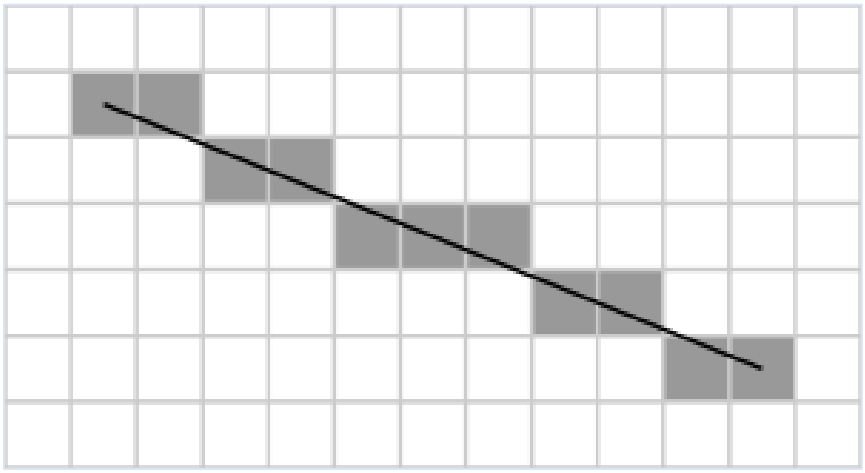
Na kraju se alocira memorija za screenBuffer u RAM i putem memset se elementi screenBuffera mijenjaju u znak Space.

```
1.      Engine(short fontW, short fontH, short screenW, short screenH) {
2.              // Handle initialization
3.              outH = GetStdHandle(STD_OUTPUT_HANDLE);
4.
5.              // Window tweaking
6.              FontSize(fontW, fontH);
7.              ScreenSize(screenW, screenH);
8.
9.              GetConsoleBufferDimensions();
10.             SetConsoleActiveScreenBuffer(outH);
11.
12.             // Memory allocation for the buffer
13.             screenBuffer = (char*)malloc(columns * rows);
14.             // Set buffer content to blank
15.             memset(screenBuffer, ' ', columns * rows);
16.
17.     };
18.
19.     // Set screen size of console
20.     void ScreenSize(short w, short h) {
21.             COORD size = { w, h };
22.             _SMALL_RECT window;
23.
24.             window.Top = 0;
25.             window.Left = 0;
26.             window.Right = w - 1;
27.             window.Bottom = h - 1;
28.
29.             SetConsoleWindowInfo(outH, true, &window);
30.             SetConsoleScreenBufferSize(outH, size);
31.
32.     }
33.
34.     // Set font size of console
35.     void FontSize(short w, short h) {
36.             CONSOLE_FONT_INFOEX cfi;
37.             cfi.cbSize = sizeof(cfi);
38.             cfi.dwFontSize.X = w;
39.             cfi.dwFontSize.Y = h;
40.
41.             SetCurrentConsoleFontEx(outH, FALSE, &cfi);
42.
43.     }
44.
45.     void GetConsoleBufferDimensions() {
46.             // Get info
47.             GetConsoleScreenBufferInfo(outH, &csbi);
48.             // Update
49.             columns = csbi.srWindow.Right - csbi.srWindow.Left + 1;
50.             rows = csbi.srWindow.Bottom - csbi.srWindow.Top + 1;
51.
52.     };
```

3.2.2 DrawCharacter(), DrawLine(), DrawTriangle()

Funkcija DrawCharacter() prima karakter i poziciju na ekranu. Console buffer i prozor Windows konzole je suštinski dvodimenzijonalni niz, pa je moguće nacrtati jedan piksel u obliku slova na bilo kom mjestu na ekranu.

DrawLine() funkcija prima dvije tačke p0 i p1, inkorporira DrawCharacter() funkciju za crtanje piksela i koristi modificirani Bresenhamov algoritam za rastersko crtanje linija na dvodimenzionalnom nizu da nacrta bilo koju liniju u Screen Bufferu.



Slika 4: Bresenhamov algoritam za crtanje linija na bitmap ekranu

Na kraju, DrawTriangle() funkcija poziva tri DrawLine() funkcije sa tačkama p0, p1 i p2 da nacrta trougao u Screen Bufferu.

```
1.      void DrawCharacter(char charType, COORD pos) {
2.              // Out of bounds
3.              if (pos.X < 0 || pos.X > columns || pos.Y < 0 || pos.Y > rows)
4.                      return;
5.              // Turn 2d coordinates into 1d array position
6.              int bufferIndex = (pos.Y * columns) + pos.X;
7.
8.              if (bufferIndex < 0 || bufferIndex > columns * rows)
9.                      return;
10.
11.             screenBuffer[bufferIndex] = charType;
12.     };
13.
14.     // Bresenham's line algorithm
15.     void DrawLine(char charType, COORD p0, COORD p1) {
16.             int dx = abs(p1.X - p0.X);
17.             int dy = -1 * abs(p1.Y - p0.Y);
18.
19.             int sx = (p0.X < p1.X) ? 1 : -1;
20.             int sy = (p0.Y < p1.Y) ? 1 : -1;
21.
22.             int err = dx + dy;
```

```
23.
24.                 while (true) {
25.                         DrawCharacter(charType, p0); // Draw character
26.
27.                         if (p0.X == p1.X && p0.Y == p1.Y) // End case
28.                                 break;
29.
30.                         int e2 = 2 * err; // Simplifies the comparison
31.
32.                         if (e2 >= dy) { // Moves by x axis since dx is positive
33.                                 if (p0.X == p1.X) // Edge case
34.                                         break;
35.                                 err += dy;
36.                                 p0.X += sx; // Changes x position
37.                         }
38.                         if (e2 <= dx) {
39.                                 if (p0.Y == p1.Y)
40.                                         break;
41.                                 err += dx;
42.                                 p0.Y += sy;
43.                         }
44.                 }
45.         };
46.
47.         void DrawTriangle(char charType, COORD p0, COORD p1, COORD p2) {
48.                 DrawLine(charType, p0, p1);
49.                 DrawLine(charType, p1, p2);
50.                 DrawLine(charType, p2, p0);
51.         }
```

3.2.3 ~Engine(), DisplayFrame()

DisplayFrame() funkcija prvo pozove SetConsoleCursorPosition() da postavi kursor konzole na nultoj poziciji (u konzoli se upisuje od pozicije kursora), te pozivom WriteConsoleA() upisuje sadržaj našeg Screen Buffera u Windows konzoli. Na kraju se Screen Buffer resetuje na prvobitno stanje (svi postaju karakteri Space).

~Engine() funkcija se zove na kraju života i on briše Screen Buffer da se izbjegne memory leak.

```
 1.         void DisplayFrame() {
 2.
 3.                 DWORD numChars; // Not needed but not optional
 4.                 COORD origin = { 0, 0 };
 5.
 6.                 // Set cursor position to origin and write contents of screen buffer to console
 7.                 SetConsoleCursorPosition(outH, origin);
 8.                 WriteConsoleA(outH, screenBuffer, columns * rows, &numChars, NULL);
 9.
10.                 // Reset screen buffer
11.                 memset(screenBuffer, ' ', columns * rows);
12.         };
13.
14.         ~Engine() {
15.                 // Destroy screen buffer
16.                 delete[] screenBuffer;
17.         };
```

3.3 3Drenderer.cpp

3Drenderer.cpp je glavni program projekta u kojem je definisan kod za unos putanje .obj fajla i njegovog parsiranja, matrične matematičke operacije za rotiranje i projektiranje iz 3D u 2D površinu Windows konzole te korektno skaliranje i offsetovanje 3D modela za bolju preglednost.

3.3.1 Unos .obj fajla

3Drenderer jedino podržava unos Wavefront .obj fajlova. .obj fajlovi sadržavaju informacije o geometriji 3D modela naime: pozicija svakog vrha, UV pozicija svakog vrha koordinata teksture, normale verteksa i lica koja čine svaki poligon definisan kao lista vrhova i vrhova teksture. Vrhovi se po defaultu pohranjuju u smjeru suprotnom od kazaljke na satu. Jedine bitne informacije za naš program su pozicije vrhova/vertex (oznaka "v") u 3D geometrijskoj ravni i definicije trouglova/face (oznaka "f") sa indeksima vrhova prethodno navedeni.

```
# Primjer rudimentarnog Wavefront .obj fajla (isječak)
v 4.350000 0.180000 3.000000
v -4.350000 0.180000 3.000000
v -4.350000 -0.180000 3.000000
v 0.000000 -0.700000 3.000000
v -0.720000 -0.120000 -1.400000
v 0.720000 -0.120000 -1.400000
v 0.720000 0.120000 -1.400000
s off
f 21 52 12
f 6 13 8
f 5 23 1
f 7 1 3
f 4 6 8
f 4 12 10
f 17 20 10
f 20 4 10
```

Potrebno je prvenstveno u programu napraviti strukturu podataka koja će čuvati informaciju parsirano iz .obj fajla. Navedene strukture su point (tačka u geometrijskom prostoru), triangle (trougao sačinjen od point struktura) i mesh (sačinjeno od triangle struktura, ovo ce čuvati cjelokupni 3D model koji ćemo prikazati u konzoli).

```
struct point
{
	float x, y, z;
};

struct triangle
{
	point p[3];
};

struct mesh
{
	vector<triangle> tris;
};
```

Nakon toga funkcija LoadObjFile() otvara .obj fajl kroz **ifstream** i postepeno kopira 3D model u globalnu mesh varijablu **objModel**, koja će kasnije biti korištena za prikaz u Windows konzoli**:**

```
bool LoadObjFile() {
	vector<point> pointBuffer;

	string dir;

	cout << "Read directory: ";
	cin >> dir;

	ifstream objfile(dir);

	if (!objfile.is_open()) {
		cout << "Error could not open file" << endl;
		return false;
	}


	string line;
	while (!objfile.eof())
	{

		char fileline[200];
		objfile.getline(fileline, 200);

		strstream ss;
		ss << fileline;

		char temp;

		// Vertices data
		if (fileline[0] == 'v') {
			point v;
			ss >> temp >> v.x >> v.y >> v.z;
			pointBuffer.push_back(v);
		}
		// Triangle data
		if (fileline[0] == 'f') {
			int f[3];
			ss >> temp >> f[0] >> f[1] >> f[2];
			objModel.tris.push_back({ pointBuffer[f[0] - 1], pointBuffer[f[1] - 1],
pointBuffer[f[2] - 1] }); // .obj format of triangle faces starts at 1
		}

	}

	return true;

};
```

3.3.2 MultiplyMatrixVector()

Za transformacije tačaka koriste se 4x4 matrice projekcije i rotacije prema X i Z osi. Stoga nam je potrebna funkcija koja množi jedan vektor (u geometrijskom smislu) sa matricom i izbacuje transformisani drugi vektor. MultiplyMatrixVector() u ovom slučaju obavlja tu funkciju i potrebna su mu 3 parametra: pokazivač prema input vektoru (point& i), pokazivač prema output vektoru (point& o) i pokazivač prema matrici s kojom vršimo transformaciju (mat4x4& m). Važno je napomenuti da je mat4x4 struct koja inicijalizira samo 4D dimenzionalni niz:

```
1. struct mat4x4
2. {
3.         float m[4][4] = { 0 };
4. };
```

Za transformaciju se koristi matrična multiplikacija vektora i matrice pa rezultirajući kod izgleda kao:

```
 1. void MultiplyMatrixVector(point& i, point& o, mat4x4& m)
 2. {
 3.       o.x = i.x * m.m[0][0] + i.y * m.m[1][0] + i.z * m.m[2][0] + m.m[3][0];
 4.       o.y = i.x * m.m[0][1] + i.y * m.m[1][1] + i.z * m.m[2][1] + m.m[3][1];
 5.       o.z = i.x * m.m[0][2] + i.y * m.m[1][2] + i.z * m.m[2][2] + m.m[3][2];
 6.       float w = i.x * m.m[0][3] + i.y * m.m[1][3] + i.z * m.m[2][3] + m.m[3][3];
 8.       if (w != 0.0f)
 9.       {
10.                o.x /= w; o.y /= w; o.z /= w;
11.       }
12. };
```

Pošto 3x3 matrice ne mogu raditi translaciju vektora (potrebno je za projekciju), potrebna nam je 4x4 matrica pa je također dodata dimenzija vektoru sa oznakom "w". Matrične operacije također vršimo i na dodatnu dimenziju, a na kraju sve članove vektora podijelimo sa w (izuzev ako je 0) da bi mogli naš izlazni vektor pretvoriti nazad u Kartezijev koordinatni sistem.

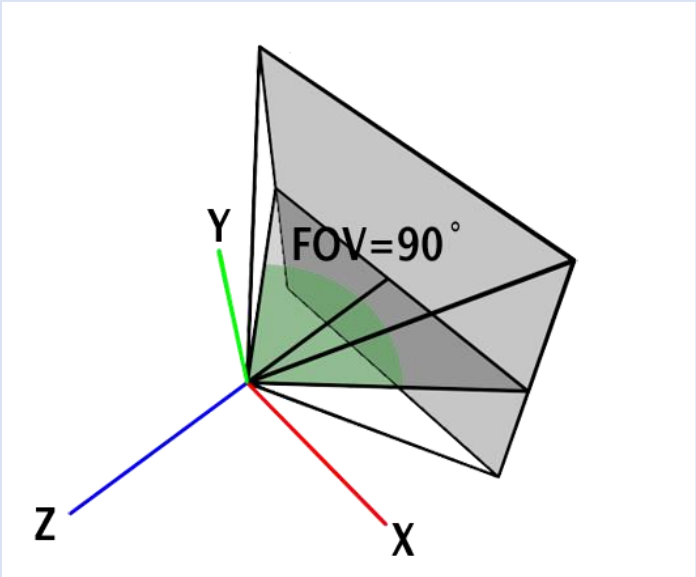
3.3.3 Projekcijska matrica

Za transformaciju 3D geometrijskog prostora u kojoj se nalazi naš model u 2D viewport prozor Windows konzole iskorištena je perspektivna projekcijska matrica.

$$\begin{bmatrix} \text{fAspectRatio} * \text{fFovRad} & 0 & 0 & 0 \\ 0 & \text{fFovRad} & 0 & 0 \\ 0 & 0 & \frac{fFar}{fFar - fNear} & 1 \\ 0 & 0 & \frac{-fFar * fNear}{fFar - fNear} & 0 \end{bmatrix}$$

fAspectRatio je omjer veličine ekrana Windows konzole mjeren kao količnik visine i duzine. Njegov efekat u računu je normaliziranje X osu tačke u Windows konzoli.

fFov (vidno polje) je ugaoni opseg geometrijskog prostora koje je vidljivo u Windows konzoli. Definisan je kao 90° u projektu. Da bismo korektno implementirali naše vidno polje u projektu potrebno je fFov pretvoriti u radijane i kalkulisati njegov arctg. Taj rezultat na kraju dijelimo sa faktorom 2, pa je formula: $fFovRad = \tan^{-1}\frac{fFov*\pi}{360}$



Slika 5: FOV kontroliše koliko se scene vidi

fFar i fNear su varijable koje služe za normalizaciju Z ose (clipping plane). fNear i fFar opisuju opseg daljine koju hoćemo da mi renderujemo u našoj konzoli (fNear = 1, fFar = 1000). Normalizacija se vrši jednačinom $\frac{fFar}{fFar-fNear}$, i jednačinom $\frac{-fFar*fNear}{fFar-fNear}$.

Na kraju izvučemo iz vektora Z i podijelimo sve koordinate sa Z, pa je rezultirajući vektor vraćen nazad u Kartezijski koordinatni sistem.

Na kraju je matrica u programu definisana kodom:

```
1.        matProj.m[0][0] = fAspectRatio * fFovRad;
2.        matProj.m[1][1] = fFovRad;
3.        matProj.m[2][2] = fFar / (fFar - fNear);
4.        matProj.m[3][2] = (-fFar * fNear) / (fFar - fNear);
5.        matProj.m[2][3] = 1.0f;
6.        matProj.m[3][3] = 0.0f;
```

Sa varijablama definisane kao:

```
1. float fNear = 0.1f;
2. float fFar = 1000.0f;
3. float fFov = 90.0f;
4. float fAspectRatio = (float)height / (float)width;
5. float fFovRad = 1.0f / tanf(fFov * 0.5f / 180.0f * 3.14159f);
```

3.3.4 Rotacione matrice X i Z

Rotacione matrice u projektu služe da rotiraju tačke 3D modela po X osi i Z osi za ugao θ (theta). Rotacione matrice u sve tri ose su definisane sljedeće:

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$$

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$$

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Slika 6: Rotacione matrice za X, Y i Z ose

U programu su definisane kodom:

```
// Rotation Z
matRotZ.m[0][0] = cosf(fTheta);
matRotZ.m[0][1] = sinf(fTheta);
matRotZ.m[1][0] = -sinf(fTheta);
matRotZ.m[1][1] = cosf(fTheta);
matRotZ.m[2][2] = 1;
matRotZ.m[3][3] = 1;

// Rotation X
matRotX.m[0][0] = 1;
matRotX.m[1][1] = cosf(fTheta * 0.5f);
matRotX.m[1][2] = sinf(fTheta * 0.5f);
matRotX.m[2][1] = -sinf(fTheta * 0.5f);
matRotX.m[2][2] = cosf(fTheta * 0.5f);
matRotX.m[3][3] = 1;
```

3.3.5 int main() – Startup

Program na startup zahtjeva od korisnika za putanju prema .obj fajlu. Nakon što korisnik upiše putanju onda se .obj file učitava u memoriju programa preko LoadObjFile() funkcije:

```
1.        bool loaded = false;
2.        while (!loaded) {
3.                  loaded = LoadObjFile();
4.        }
```

Nakon toga se inicijalizira Engine klasa sa statičnim varijablama fSizeX, fSizeY, width i height,.pa Onda se inicijaliziraju rotacione matrice matRotZ i matRotX na nulu a projekciona matrica matProj se odmah definiše:

```
 1.       bool loaded = false;
 2.       while (!loaded) {
 3.                 loaded = LoadObjFile();
 4.       }
 5.
 6.       Engine render3D(fSizeX, fSizeY, width, height);
 7.
 8.       // Projection Matrix
 9.
10.       mat4x4 matRotZ, matRotX;
11.
12.       mat4x4 matProj;
13.
14.       matProj.m[0][0] = fAspectRatio * fFovRad;
15.       matProj.m[1][1] = fFovRad;
16.       matProj.m[2][2] = fFar / (fFar - fNear);
17.       matProj.m[3][2] = (-fFar * fNear) / (fFar - fNear);
18.       matProj.m[2][3] = 1.0f;
19.       matProj.m[3][3] = 0.0f;
20.
21.       float fTheta = 0.0;
22.
```

Također se kreira varijabla fTheta koja ce služiti za inkrementiranje ugla 3D modela.

3.3.6 int main() – Programska petlja

Nakon što se startup sekvenca završi, program prelazi u beskonačnu petlju koja rotira, transformiše, projektira te skalira 3D model u Windows konzoli.

Program prvo inkrementira fTheta varijablu za 0.025 radijana, te na osnovu novog ugla osvježava rotacione matrice X i Z:

```
1. while (true) {
2.       // Render step
3.       fTheta += 0.025;
4.
5.       // Rotation Z
6.       matRotZ.m[0][0] = cosf(fTheta);
```

```
 7.       matRotZ.m[0][1] = sinf(fTheta);
 8.       matRotZ.m[1][0] = -sinf(fTheta);
 9.       matRotZ.m[1][1] = cosf(fTheta);
10.       matRotZ.m[2][2] = 1;
11.       matRotZ.m[3][3] = 1;
12.
13.       // Rotation X
14.       matRotX.m[0][0] = 1;
15.       matRotX.m[1][1] = cosf(fTheta * 0.5f);
16.       matRotX.m[1][2] = sinf(fTheta * 0.5f);
17.       matRotX.m[2][1] = -sinf(fTheta * 0.5f);
18.       matRotX.m[2][2] = cosf(fTheta * 0.5f);
19.       matRotX.m[3][3] = 1;
20.
```

Nakon toga će program za svaki trokut definisan u objModel rotirati za Z i X osu:

```
 1. // Draw Triangles
 2. for (auto tri : objModel.tris)
 3. {
 4.       triangle triProjected, triTranslated, triRotatedZ, triRotatedZX;
 5.
 6.       // Rotate in Z-Axis
 7.       MultiplyMatrixVector(tri.p[0], triRotatedZ.p[0], matRotZ);
 8.       MultiplyMatrixVector(tri.p[1], triRotatedZ.p[1], matRotZ);
 9.       MultiplyMatrixVector(tri.p[2], triRotatedZ.p[2], matRotZ);
10.
11.       // Rotate in X-Axis
12.       MultiplyMatrixVector(triRotatedZ.p[0], triRotatedZX.p[0], matRotX);
13.       MultiplyMatrixVector(triRotatedZ.p[1], triRotatedZX.p[1], matRotX);
14.       MultiplyMatrixVector(triRotatedZ.p[2], triRotatedZX.p[2], matRotX);
15.
```

Pošto bi model bio preblizu ekrana, radi se offset:

```
1.       // Offset into the screen
2.       triTranslated = triRotatedZX;
3.       triTranslated.p[0].z = triRotatedZX.p[0].z + 8.0f;
4.       triTranslated.p[1].z = triRotatedZX.p[1].z + 8.0f;
5.       triTranslated.p[2].z = triRotatedZX.p[2].z + 8.0f;
6.
```

Transformisani trokuti se projektiraju iz 3D u 2D prostor:

```
1.       // Project triangles from 3D --> 2D
2.       MultiplyMatrixVector(triTranslated.p[0], triProjected.p[0], matProj);
3.       MultiplyMatrixVector(triTranslated.p[1], triProjected.p[1], matProj);
4.       MultiplyMatrixVector(triTranslated.p[2], triProjected.p[2], matProj);
5.
```

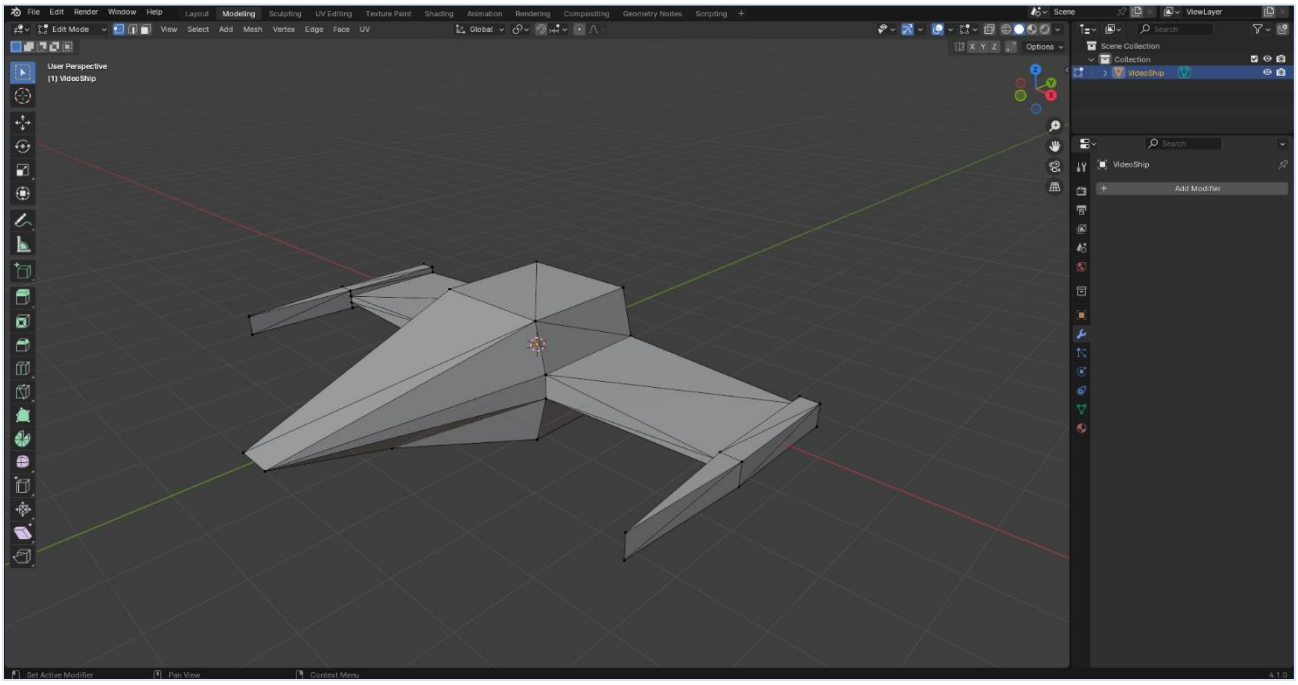
Nakon projektiranja se vrši skaliranje 3D modela da stane pregledno u Windows konzoli:

```
 1.                       // Scale into view
 2.                       triProjected.p[0].x += 1.0f; triProjected.p[0].y += 1.0f;
 3.                       triProjected.p[1].x += 1.0f; triProjected.p[1].y += 1.0f;
 4.                       triProjected.p[2].x += 1.0f; triProjected.p[2].y += 1.0f;
 5.
 6.                       triProjected.p[0].x *= 0.5f * (float)width;
 7.                       triProjected.p[0].y *= 0.5f * (float)height;
 8.
 9.                       triProjected.p[1].x *= 0.5f * (float)width;
10.                       triProjected.p[1].y *= 0.5f * (float)height;
11.
12.                       triProjected.p[2].x *= 0.5f * (float)width;
13.                       triProjected.p[2].y *= 0.5f * (float)height;
```

Na kraju se svi trokuti iscrtaju u Screen Buffer i potom su prikazani u Windows konzoli u 60fps.

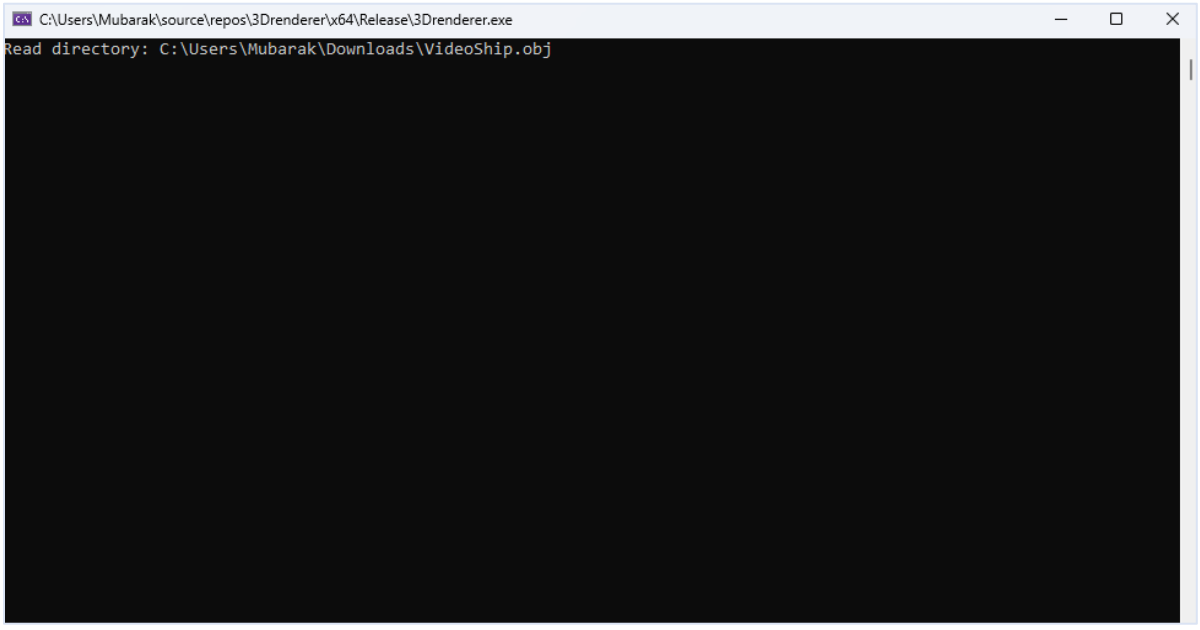
```
1.                          // Rasterize triangle
2.                          render3D.DrawTriangle(219,                  ToCoord(triProjected.p[0].x,
triProjected.p[0].y),
3.                                      ToCoord(triProjected.p[1].x, triProjected.p[1].y),
4.                                      ToCoord(triProjected.p[2].x, triProjected.p[2].y));
5.
6.                  }
7.                  render3D.DisplayFrame();
8.                  Sleep(1 / 60.0f * 1000.0f);
9.          }
```

3.4 Primjer rada



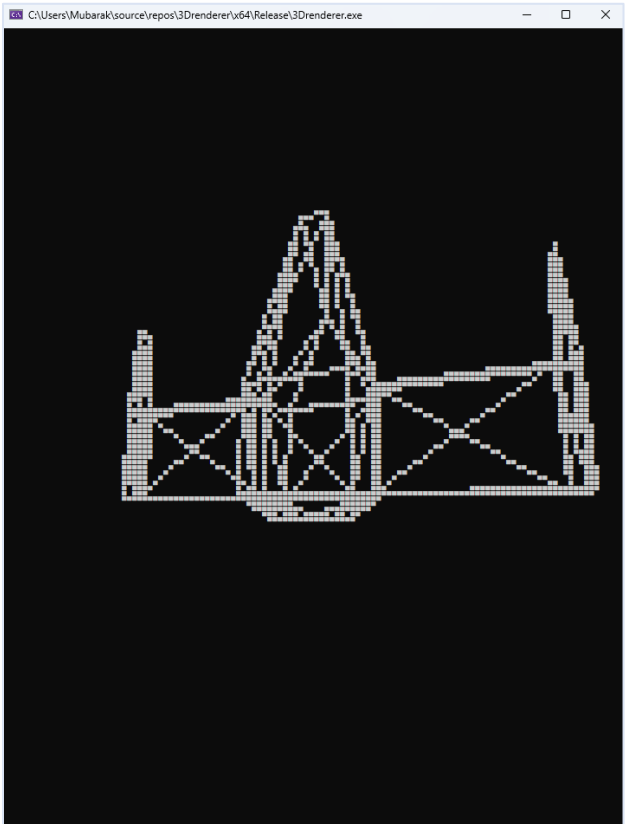
Slika 7: VideoShip.obj otvoren u Blender programu

Nakon što pokrenemo program od nas se zahtjeva putanja .obj fajla kojeg želimo prikazati:



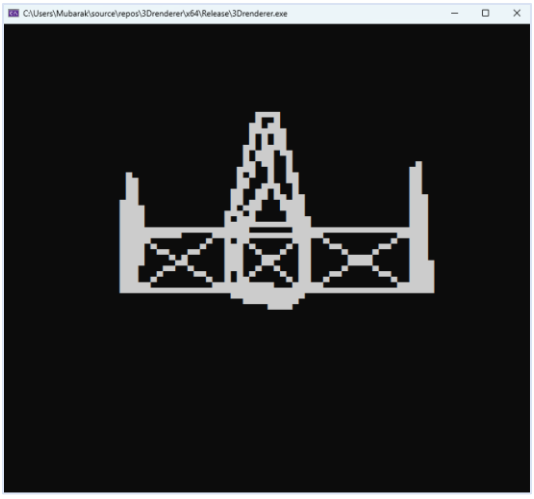
Slika 8: Početak programa, unesena je putanja .obj fajla

Nakon što unesemo putanju i kliknemo Enter, prikazan nam je naš 3D model kako rotira po X i Z osi:



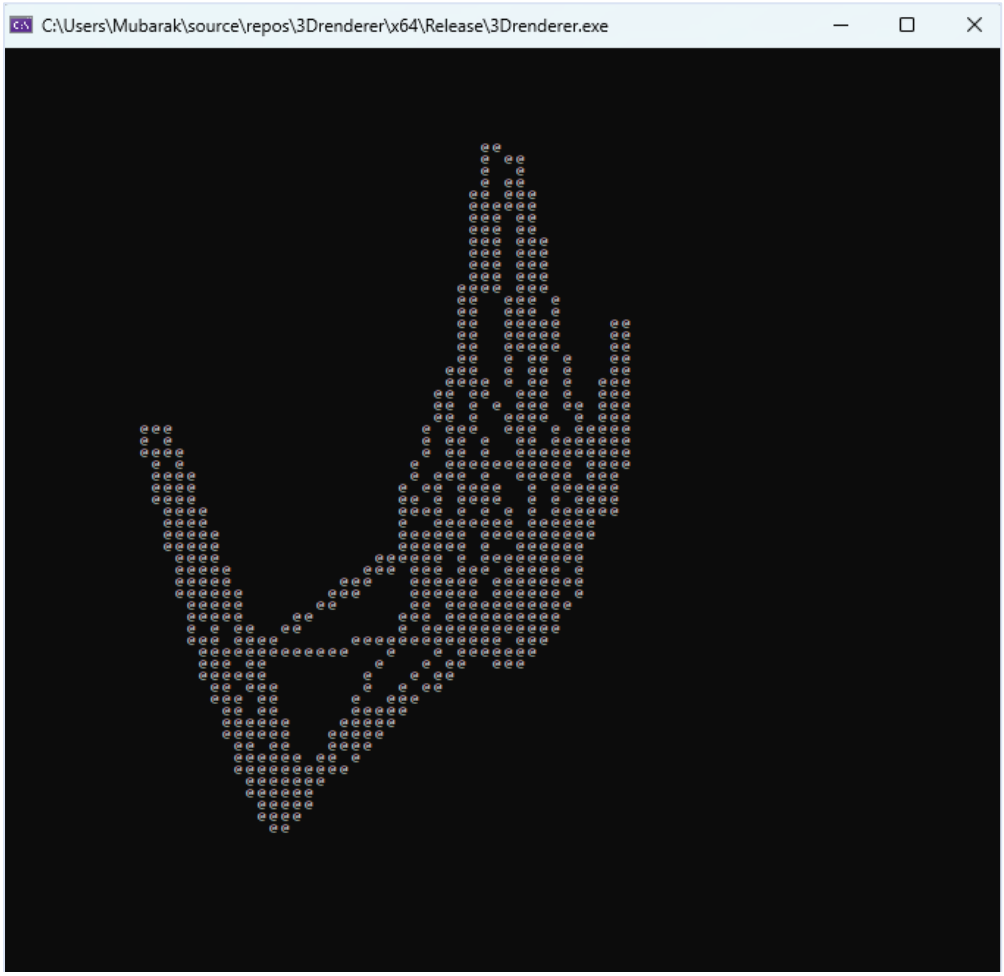
Slika 9: Krajnji rezultat

Mijenjanjem varijabla fSizeX, fSizeY, width i height moguće je promjeniti rezoluciju prikaza:



Slika 10: Prikaz modela sa rezolucijom od 86x86

Također je moguće promijeniti koji karakter se koristi za prikaz 3D modela:



Slika 11: Prikaz modela sa karakterom „@“

ZAKLJUČAK

3D rendering predstavlja ključni vid tehnološke inovacije. Proces prikazivanja trodimenzionalnih objekata na dvodimenzionalnu površinu je složen proces, zahtijevajući različite matematičke i računarske operacije. Ovaj projekat pomoću C++ nam je prikazao principe rada jednog jednostavnog 3D rasterizera. Za bilo kakvo prikazivanje prvenstveno nam je potreban Graphics Library. U industriji se danas koristi OpenGL, Vulkan i DirectX, ali za naš projekat smo napravili za sebe jedan prostiji Graphics Library koji ima funkcionalnost crtanja na ekranu. Nakon toga su objašnjeni koncepti poput projekcione matrice i pojmovi Aspect ratio, Field of view, Z normalizacija/clipping i Screen buffer, nešto što se često koristi danas za složenije operacije. Funkcionalnosti poput Normals, Textures, Depth buffering i Culling se mogu dodati da se ovaj početnički program pretvori u potpuni 3D Renderer.

MIŠLJENJE O RADU

Potpis mentora:

_____

Adnan Delić dipl.el. ing.

LITERATURA

- https://www.techtarget.com/whatis/definition/vector-graphics
- https://guides.lib.umich.edu/c.php?g=282942&p=1885352
- https://isocpp.org/
- https://www.scratchapixel.com/lessons/3d-basic-rendering/perspective-and-orthographic-projection-matrix/building-basic-perspective-projection-matrix.html
- http://members.chello.at/~easyfilter/Bresenham.pdf
- https://www.youtube.com/watch?v=ih20l3pJoeU
- https://www.cuemath.com/algebra/transformation-matrix/
- https://mathinsight.org/matrix_vector_multiplication
- https://mathworld.wolfram.com/RotationMatrix.html