AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE
WYDZIAŁ INFORMATYKI, ELEKTRONIKI I TELEKOMUNIKACJI

KATEDRA TELEKOMUNIKACJI

Projekt dyplomowy

Silnik do gier komputerowych 3D z edytorem graficznym
3D Game Engine with editor

Autorzy: Mateusz Rzeczyca
Kierunek studiów: Elektronika i Telekomunikacja
Opiekun pracy: dr. inż. Jarosław Bułat

Kraków, 2022

Uprzedzony o odpowiedzialności karnej na podstawie art. 115 ust. 1 i 2 ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t.j. Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.): „Kto przywłaszcza sobie autorstwo albo wprowadza w błąd co do autorstwa całości lub części cudzego utworu albo artystycznego wykonania, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 3. Tej samej karze podlega, kto rozpowszechnia bez podania nazwiska lub pseudonimu twórcy cudzy utwór w wersji oryginalnej albo w postaci opracowania, artystyczne wykonanie albo publicznie zniekształca taki utwór, artystyczne wykonanie, fonogram, wideogram lub nadanie.", a także uprzedzony o odpowiedzialności dyscyplinarnej na podstawie art. 211 ust. 1 ustawy z dnia 27 lipca 2005 r. Prawo o szkolnictwie wyższym (t.j. Dz. U. z 2012 r. poz. 572, z późn. zm.) „Za naruszenie przepisów obowiązujących w uczelni oraz za czyny uchybiające godności studenta student ponosi odpowiedzialność dyscyplinarną przed komisją dyscyplinarną albo przed sądem koleżeńskim samorządu studenckiego, zwanym dalej „sądem koleżeńskim", oświadczam, że niniejszą pracę dyplomową wykonałem osobiście i samodzielnie i że nie korzystałem ze źródeł innych niż wymienione w pracy.

Spis treści

1 Wprowadzenie

Nie tak dawno temu wiedza niezbędna do programowania gier wideo (ang. *Game Programming*) była dostępna tylko dla ograniczonej liczby osób, w szczególności weteranów branży gier. W minionej epoce nauka algorytmów, wzorców projektów oraz ogólnych zasad działania oprogramowania były podobne do uczenia się czarnej magii (stąd też tytuł książki Michaela Abrasha *Graphics Programming Black Book* [1]). W przypadku, gdy student chciał kontynuować formalną edukację w zakresie programowania gier i tworzenia silników graficznych (ang. *Game Engine*), możliwy wybór był ograniczony do niewielkiej liczby specjalistycznych szkół branżowych. Jednak w ciągu ostatnich kilkunastu lat edukacja dotycząca programowania gier wideo zmieniła się diametralnie. Najlepsze uniwersytety oferują kursy i stopnie naukowe związane z tym tematem. Mnóstwo młodych osób zafascynowanych branżą rozpoczyna kierunki związane z grafiką komputerową [2]. Zainteresowani tematem studenci AGH również znajdą satysfakcjonujące przedmioty na pierwszym stopniu Informatyki. Są to tematy wprowadzające w programowanie grafiki komputerowej pozwalające poznać temat podczas ostatnich semestrów studiów. Z kolei na drugim stopniu uczelnia oferuje bardziej zaawansowane przedmioty, które poruszają renderowanie w czasie rzeczywistym oraz algorytmiczną teorię gier.

Efektem ubocznym eksplozji zainteresowania nauki programowania gier wideo są zdecydowanie zwiększone wymagania wobec stażystów oraz juniorów. We wczesnych latach 2000, oczekiwania wobec osób wchodzących w branżę grafiki komputerowej były równoznaczne z solidnym wykształceniem informatycznym oraz pasją do tworzenia gier. Teraz, gdy formalna edukacja pozwala na dynamiczny rozwój dziedziny ówcześni seniorzy, doświadczeni deweloperzy stali się juniorami. Osoby niezainteresowane dalszym rozwojem pozostają w tyle w porównaniu do ludzi poświęconych tej działalności [2].

Spółki specjalizujące się w rozwoju silników graficznych rozszerzają swoją działalność na zupełnie nowe dziedziny. Dotychczas silniki graficzne były wykorzystywane jedynie w branży gier wideo. Błyskawiczny postęp tej dziedziny pozwolił spotęgować wykorzystanie technologii w wielu różnych dziedzinach, takich jak produkcja filmów i seriali telewizyjnych, architektura, motoryzacja i transport, transmisja z wydarzeń na żywo oraz symulacji w czasie rzeczywistym [3]. Często przez wysoki poziom zaawansowania oprogramowania ciężko dostrzec wykorzystanie silników, ponieważ komputerowe elementy graficzne wyglądają bardzo realistycznie.

Obecny próg wejścia w tworzenie silników graficznych jest postawiony bardzo wysoko, należy być osobą doświadczoną nie tylko w renderowaniu grafiki komputerowej, ale też w animacjach komputerowych, wzorcach architektonicznych, symulowaniu fizyki świata, odtwarzaniu audio, czy w systemach sztucznej inteligencji. Tworzenie tak złożonego oprogramowania jest zadaniem niezwykle złożonym i wymagającym, ale też pouczającym.

Celem pracy było zaimplementowanie oprogramowania, które następnie może otrzymać miano silnika do gier komputerowych. Aplikacja ma umożliwiać renderowanie obiektów oraz ich modyfikację z wykorzystaniem interfejsu graficznego w czasie rzeczywistym.

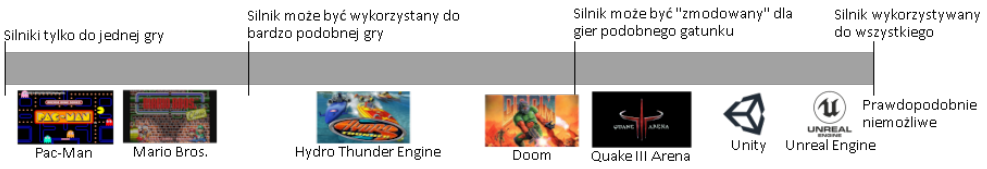
W pierwszym rozdziale opisano zagadnienia, które zapewnią podstawową wiedzę czytelnikowi, niezbędną w zrozumieniu dalszej części pracy. Drugi rozdział poświęcony jest implementacji oprogramowania. Opisano architekturę silnika graficznego jak i przedstawiono poszczególne komponenty aplikacji.

2 Teoria silników do gier komputerowych

W tym rozdziale sprecyzowano definicję silnika do gier komputerowych i porównano go do typowych silników używanych w grach wideo. Omówiono znane autorowi sposoby reprezentacji i manipulacji obiektów w przestrzeni $\mathbb{R}^3$, przedstawiono koncepty renderowania obrazu 3D celem późniejszego wyświetlenia go na ekranie 2D. Opisano również architekturę ECS, gdzie do określonej struktury mogą być przypisywane wybrane funkcje.

2.1 Pojęcie silnika graficznego

Termin "silnik gry" powstał w połowie lat 90, w odniesieniu do pierwszoosobowych gier akcji (ang. *FPS - First Person Shooter*), takich jak popularny *Doom* od studia *id Software*. *Doom* został zaprojektowany z dobrze zdefiniowaną separacją między głównymi komponentami oprogramowania, a logiką i zasobami artystycznymi gry. Główne komponenty oprogramowania (ang. *Core Software Components*) należy rozumieć jako trójwymiarowy system do renderowania sceny, system kolizji obiektów, bądź system audio. Natomiast logika i zasoby gry składają się na doświadczenia użytkownika rozgrywki [4]. Dotychczas gry komputerowe nie posiadały "silników", programiści pisali kod skierowany bezpośrednio pod konkretny projekt. Zasady gry włącznie z logiką były mocno zakodowane (ang. *hard-coded*), co oznacza, że wprowadzenie jakichkolwiek zmian wiązało się z refaktoryzacją aktualnej implementacji w celu dodania dodatkowej funkcjonalności. Wartość podziału wprowadzonego przez deweloperów gry *Doom* była znacząca, narodziła się społeczność ludzi, którzy hobbystycznie modyfikowali grę. Silnik posiadał niezależne narzędzia, dzięki którym modderzy (ang. *modders*) mogli wedle własnego uznania modyfikować istniejącą grę bez potrzeby zaglądania w kod źródłowy. Rozwój silników graficznych od ich początków do aktualnej sytuacji przedstawiony jest na rysunku 1.



Rysunek 1: Możliwość ponownego wykorzystania silników do gier z przykładami.

Zatem silnik do gier komputerowych możemy określić jako zbiór komponentów oprogramowania, systemów i bibliotek, które pozwolą w łatwy sposób kreować gry wideo. W skład głównych komponentów silnika do gier wideo wchodzą zazwyczaj:

- system renderujący (ang. *Renderer*),
- silnik fizyki lub detekcji kolizji (ang. *Physics Engine or Collision Detection*),
- system audio,
- język skryptowy (ang. *Scripting Programming Language*),
- system animacji,

- sztuczna inteligencja (ang. *Artificial Intelligence*),
- wielowątkowość (ang. *multithreading*),
- oraz wiele innych [4].

2.2 Grafika komputerowa 3D

Renderowanie w czasie rzeczywistym jest powiązane z energicznym tworzeniem kolekcji obrazów na komputerze. Jest to najbardziej interaktywna dziedzina grafiki komputerowej. Obraz pojawia się na ekranie i znika, widz w tym czasie obserwuje, działa i reaguje, a informacja zwrotna z reakcji wpływa na generowanie kolejnych obrazów. Wspomniany cykl reakcji i renderowania przebiega tak szybko, że użytkownik nie jest w stanie dostrzec pojedynczych obrazów, w pełni zanurza się w dynamicznym procesie ciągłego tworzenia obrazów - świecie wirtualnym [5].

Tempo wyświetlania się obrazów jest mierzone w ramkach / klatkach na sekundę (ang. *FPS - Frame Per Second*) lub w Hercach [Hz]. Gdy gra działa z renderowaniem jednej klatki na sekundę gracz nie ma poczucia imersji, nie zostaje wciągnięty do świata wirtualnego. Jest świadomy nadejścia kolejnych obrazów. Wraz z zwiększającą się liczbą FPS poczucie interaktywności zaczyna wzrastać. Obecnie standardem wśród gier komputerowych jest utrzymanie 30 lub 60 klatek na sekundę w zależności od wybranej rozdzielczości obrazu oraz możliwości posiadanego sprzętu.

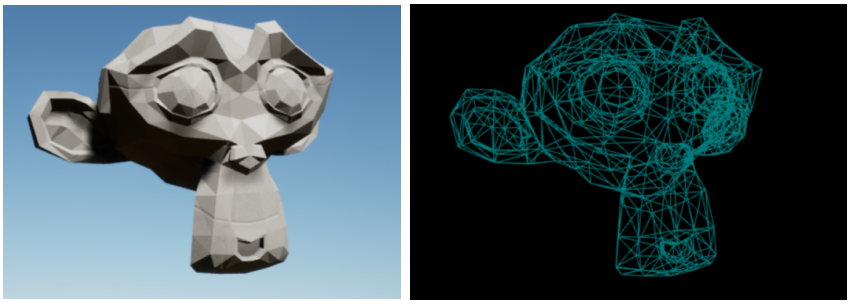
Pierwsze gry wideo w 3D wymagały od programisty implementacji swojego własnego oprogramowania renderującego. Oznacza to, że deweloper musiał zadbać chociażby o rysowanie linii na ekranie. Algorytmy niezbędne do rysowania obiektów 3D na ekranie 2D nazywa się rasteryzacją (ang. *software rasterization*). Nowoczesne komputery osobiste oraz konsole do gier posiadają sprzętową jednostkę specjalizującą się w grafice - kartę graficzną (ang. *GPU - Graphics Processing Unit*). Karta graficzna jest od samego początku zaprogramowana, w jaki sposób może narysować na ekranie takie obiekty jak punkty, linie oraz trójkąty. Programiści mogą wykorzystywać dostępną moc obliczeniową jednostek graficznych oraz mogą w pełni skupić się na programowaniu scen 3D z wykorzystaniem odpowiednich bibliotek oraz mikroprogramów personalizowanych pod karty graficzne (ang. *shaders*) [2].

2.2.1 Reprezentacja obiektów

Trójwymiarowe modele mogę być reprezentowane na wiele sposobów, jednak zdecydowana większość jest renderowania poprzez wykorzystanie wielokątów (ang. *polygon*) - szczególnie trójkątów [2]. Przykładowy model wygenerowany przy pomocy trójkątów ukazany jest na rysunku 2.

Trójkąty są optymalnym rozwiązaniem z kilku powodów:

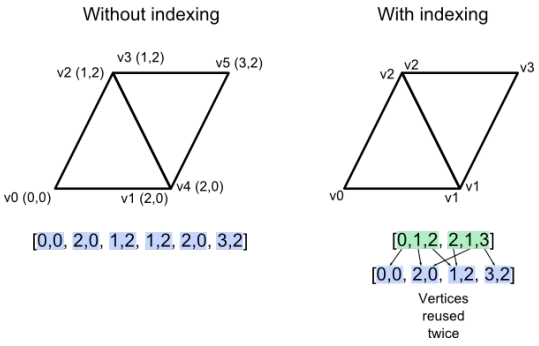
- trójkąt to najprostszy wielokąt, składa się tylko z 3 wierzchołków (ang. *vertices*),
- trójkąt zawsze leży na jednej płaszczyźnie (ang. *plane*),
- trójkąty z łatwością można podzielić na mniejsze obiekty nie pozostawiając dziur oraz deformacji (ang. *tessellation*) .



(a) Suzzane w standardowej formie. **(b)** Suzzane złożona z krawędzi.

Rysunek 2: Porównanie obiektu Suzzane w Unreal Engine 4.

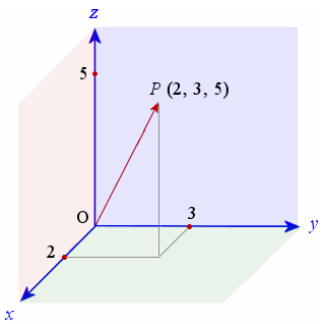
Pojedynczy model może składać się z ogromnej liczby trójkątów tworząc sporą siatkę (ang. *mesh*). W wyniku optymalizacji (zmniejszenia ilości wierzchołków) zaproponowano indeksowanie, czego rezultatem jest możliwość ponownego wykorzystania wierzchołka trójkąta do tworzenia następnych obiektów. Tak więc trójkąt jest definiowany przy pomocy trzech ponumerowanych wierzchołków [6] co zostało przedstawione na rysunku 3.



Rysunek 3: Ponowne wykorzystanie wierzchołków trójkąta [6].

2.2.2 Manipulacja obiektami

W programach graficznych punkt jest zdefiniowany poprzez wektor w przestrzeni $\mathbb{R}^3$ będący trójką skalarów (x, y, z). Warunkiem takiej definicji punktu jest założenie, że punkt zaczepienia wektora znajduje się na początku układu współrzędnych [4]. Przykładowy punkt $P = (2, 3, 5)$ w scenie 3D przedstawiono na rysunku 4.



Rysunek 4: Przedstawienie wektora w przestrzeni $\mathbb{R}^3$ [7]

Na tak zdefiniowanym punkcie możliwe jest zastosowanie operacji takich jak translacja, skalowanie, rotacja oraz kombinacja wymienionych manipulacji. Transformacje są zdefiniowane przy pomocy operacji macierzowych. Najprostszą macierzą wykorzystywaną w grafice 3D jest macierz jednostkowa (ang. *identity matrix*). Macierz jednostkowa pozostawia wektor w stanie nienaruszonym co potwierdza równanie (1) będące podstawą do wykonania następnych operacji [8]:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 1 * x \\ 1 * y \\ 1 * z \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \tag{1}$$

Wektor zawdzięcza niezmienne wartości jedynkom znajdującym się w macierzy jednostkowej, ponieważ każdy skalar jest mnożony przez jedynkę. Idąc tym tropem, zmiana liczb znajdujących się na przekątnej macierzy spowoduje też wydłużenie się bądź skrócenie wektora. Odpowiada to operacji skalowania (ang. *scaling*). Przy odpowiednio dużym zbiorze punktów prezentowany obiekt może stać zarówno większy jak i mniejszy. Efekt zawdzięczany jest czynnikom skalującym (ang. *scaling factors*), natomiast sama macierz nazywa się macierzą skalującą (ang. *scaling matrix*) [5]. Operację skalowania wektora przedstawiono na równaniu (2).

$$\begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & S_z \end{bmatrix} * \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} S_x * x \\ S_y * y \\ S_z * z \end{pmatrix} \tag{2}$$

Następną formą transformacji punktu jest rotacja. Rotacja w przestrzeni $\mathbb{R}^3$ wykonuje się przy pomocy zadanego kąta wokół podanej osi obrotu. Nowe wektory są rezultatem mnożenia podanego wektora i macierzy utworzonej z połączenia funkcji trygonometrycznych sinus i cosinus. Każda oś obrotu posiada własną spersonalizowaną macierz rotacji, której argumentem jest kąt obrotu [9]. Na równaniach (3), (4) oraz (5) zaprezentowano utworzone macierze kolejno dla osi x, y oraz z.

$$R_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & \sin\alpha \\ 0 & -\sin\alpha & \cos\alpha \end{bmatrix} \tag{3}$$

$$R_y(\alpha) = \begin{bmatrix} \cos\alpha & 0 & \sin\alpha \\ 0 & 1 & 0 \\ -\sin\alpha & 0 & \cos\alpha \end{bmatrix} \tag{4}$$

$$R_z(\alpha) = \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{5}$$

Kolejną formą manipulacji jest translacja, czyli zmiana pozycji w której miałby znajdować się punkt. Proces polega na dodaniu innego wektora, w tym przypadku wektora translacji (ang. *translation vector*) do już istniejącego, co przedstawia operacja $T+v$. Domyślnie translacji nie uzyskuje się poprzez mnożenie macierzowe. Celem zoptymalizowania

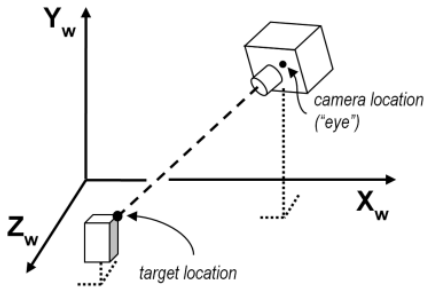
całego procesu zdecydowano się przejść na współrzędne jednorodne - wykorzystano macierze 4x4 oraz wektory w przestrzeni $\mathbb{R}^4$ z czwartym skalarem $w = 1$ - co z kolei pozwala na uzyskanie translacji poprzez mnożenie macierzowe tym samym skumulowanie wszystkich operacji w jedną macierz transformacji. Tak więc zamiast operacji $(T + v) * R * S$ uzyskano $(T*R*S)*v$ [5]. Przedstawiony sposób pozwoli na wykonanie wielu przekształceń afinicznych, czyli manipulacji obiektów w przestrzeni, za pomocą jednej zbudowanej macierzy 4x4 co umożliwia sporą optymalizację podczas renderowania grafiki 3D. Zatem operacja translacji przebiega zgodnie z równaniem (6).

$$\begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} T_x + x \\ T_y * y \\ T_z * z \\ 1 \end{pmatrix} \tag{6}$$

Wszystkie macierze transformacji zostały doprowadzone do postaci 4x4, dzięki czemu inżynierowie z prostotą mogą łączyć wiele operacji w jeden złożony iloczyn.

2.2.3 Kamera - obserwacja sceny z punktu

Kamera, nazywana też okiem, określa pozycję w której znajduje się obserwator oraz kierunek w który mierzy. Przy jej pomocy widz może poruszać się dookoła po trójwymiarowej scenie i dokładnie obejrzeć wszystkie wchodzące w jej skład obiekty. Wizualizację można obejrzeć na rysunku 5.



Rysunek 5: Przedstawienie wirtualnej kamery [10].

Kamerę definiujemy przy pomocy macierzy widoku (ang. *view matrix*). Na wejściu wymagane są trzy wektory - oko (ang. *eye*) będące pozycją widza, cel (ang. *target*) określający punkt w który mierzy kamera oraz wektor determinujący, gdzie znajduje się niebo (ang. *up*). Zazwyczaj do wygenerowania potrzebnych wektorów można wykorzystać iloczyn wektorowy (ang. *cross product*) oraz iloczyn skalarny (ang. *dot product*) [10]. Zależności (7), (8) i (9) pozwolą wyznaczyć macierz widoku (10) przy pomocy trzech punktów wejściowych.

$$\vec{fwd} = \text{normalize}(\text{eye} - \text{target}) \tag{7}$$

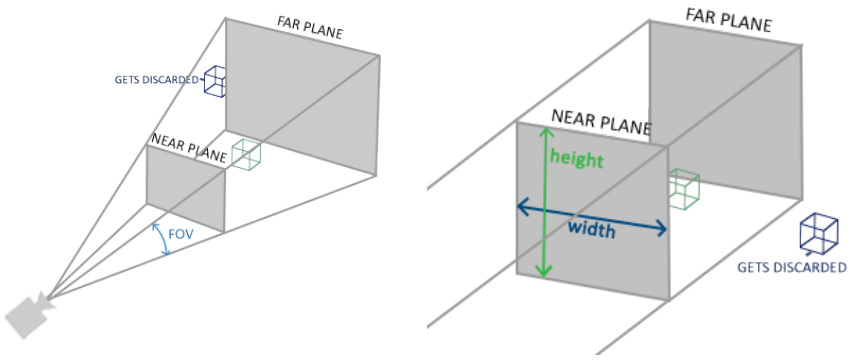
$$\vec{side} = \text{normalize}(-\vec{fwd} \times \text{up}) \tag{8}$$

$$\vec{up} = \text{normalize}(\vec{side} \times -\vec{fwd}) \tag{9}$$

$$V(e\vec{y}e, f\vec{w}d, s\vec{i}de, u\vec{p}) = \begin{bmatrix} s\vec{i}de_x & s\vec{i}de_y & s\vec{i}de_z & -(s\vec{i}de \cdot e\vec{y}e) \\ u\vec{p}_x & u\vec{p}_y & u\vec{p}_z & -(u\vec{p} \cdot e\vec{y}e) \\ f\vec{w}d_x & f\vec{w}d_y & f\vec{w}d_z & -(-f\vec{w}d \cdot e\vec{y}e) \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{10}$$

2.2.4 Projekcja Równoległa i Perspektywiczna

W danym momencie na ekranie wyświetlana jest jedynie cząstka świata. Renderowana część świata nazywana jest bryłą widzenia (ang. *viewing frustum / viewing volume*). Obiekty poza polem widzenia kamery wirtualnej nie ukazują się na ekranie, zatem w celu oszczędzenia zasobów nie są renderowane. Natomiast jeśli widoczna jest jedynie jakaś część obiektu - dochodzi do kolizji obiektu z bryłą widzenia - następuje proces przycinania (ang. *clipping*) niewidocznej części modelu. Wyświetlanie obiektów bądź ich kawałków odbywa się poprzez rzutowanie na płaszczyznę kamery [11]. Sposób wyświetlania się definiujemy poprzez projekcję, którą z kolei dzielimy na równoległą (2D) oraz perspektywiczną (3D). Porównanie generowanej sceny znajduje się na rysunku 6.



(a) 3D (perspektywiczna). **(b)** 2D (równoległa).

Rysunek 6: Porównanie typów projekcji w świecie wirtualnym [8].

Projekcja perspektywiczna (ang. *Perspective Projection*) próbuje sprawić, żeby obraz 2D wyglądał jak 3D wykorzystując koncepcję perspektywy naśladując widok jakim człowiek ogląda świat rzeczywisty. Można dostrzec, że obiekty usytuowane dalej stają się mniejsze [10]. Projekcja jest modelowana przez tzw. kamerę otworkową (ang. *Pinhole Camera*). Rzut perspektywiczny jest skonstruowany z punktu, w którym znajduje się abstrakcyjne "oko" bądź obiektyw kamery, oraz z utworzonej nieskończonej piramidy. Natomiast piramida jest pocięta przez dwie dodatkowe płaszczyzny, zwane dalej płaszczyzną bliską i daleką (ang. *near plane* oraz *far plane*) [11]. Przestrzeń znajdująca się pomiędzy dwiema płaszczyznami jest wyświetlana na ekranie. Macierz perspektywiczna (ang. *Perspective Matrix*) jest wykorzystywana do transformowania punktów w przestrzeni 3D do określonych pozycji, dzięki czemu obiekty mogą nabrać rzeczywistego kształtu. Zależności (11), (12), (13) i (14) posłużą do utworzenia macierzy perspektywicznej (15).

$$\mathrm{q}(\mathrm{fieldOfView}) = \frac{1}{\tan\frac{fieldOfView}{2}} \tag{11}$$

$$\mathrm{A}(\mathrm{q}, \mathrm{aspectRatio}) = \frac{\mathrm{q}}{aspectRatio} \tag{12}$$

$$\mathrm{B}(\mathrm{Z_{near}}, \mathrm{Z_{far}}) = \frac{Z_{near} + Z_{far}}{Z_{near} - Z_{far}} \tag{13}$$

$$\mathrm{C}(\mathrm{Z_{near}}, \mathrm{Z_{far}}) = \frac{2 * (Z_{near} * Z_{far})}{Z_{near} - Z_{far}} \tag{14}$$

$$\mathbb{P}(q, A, B, C) = \begin{bmatrix} A & 0 & 0 & 0 \\ 0 & q & 0 & 0 \\ 0 & 0 & B & C \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{15}$$

Natomiast projekcja równoległa (ang. *Orthographic Projection*) posiada pole widzenia zbliżone do sześcianu, wszystko poza nim jest ignorowane. Ten rodzaj projekcji wykorzystywany jest szczególnie przy tworzeniu gier dwuwymiarowych, przez efekt spłaszczenia obiektu [8]. Projekcja równoległa może musi zostać utworzona przy pomocy aż sześciu zmiennych wejściowych, gdzie należy zdefiniować L, R, T, B oraz Z_{near} i Z_{far}. Zmienne L, R, T i B i pochodzą od języka angielskiego, kolejno *left*, *right*, *top* oraz *bottom* [10]. Równanie (16) przedstawia macierz projekcji równoległej.

$$\mathbb{O}(L, R, T, B, Z_{near}, Z_{far}) = \begin{bmatrix} \frac{2}{R-L} & 0 & 0 & -\frac{R+L}{R-L} \\ 0 & \frac{2}{T-B} & 0 & -\frac{T+B}{T-B} \\ 0 & 0 & \frac{1}{Z_{far}-Z_{near}} & -\frac{Z_{near}}{Z_{far}-Z_{near}} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{16}$$

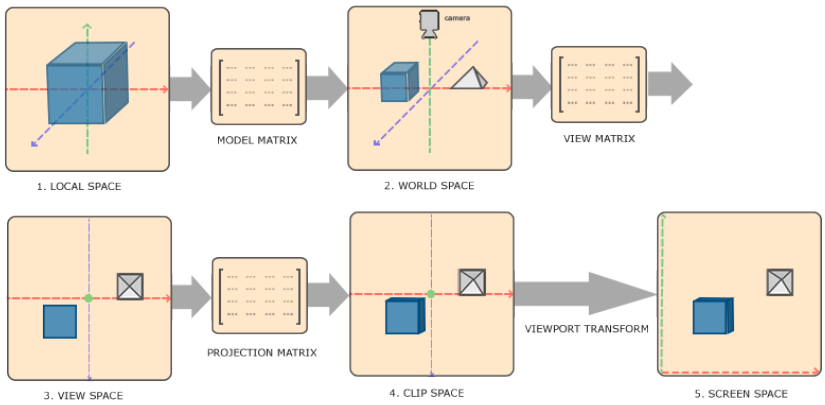
2.2.5 Przestrzeń współrzędnych

Najważniejszymi układami współrzędnych (ang. *coordinate systems*), zwanymi przestrzeniami (ang. *spaces*) są:

- przestrzeń modelu / lokalna (ang. *model / local space*) - współrzędne lokalne każdego obiektu, relatywnego do jego środka,
- przestrzeń świata (ang. *world space*) - koordynaty relatywne do środka świata, kilka obiektów jednocześnie może znajdować się w tej przestrzeni,
- przestrzeń widoku / kamery (ang. *view / camera space*) - współrzędne, które określają w jaki sposób obiekt jest widziany z perspektywy kamery bądź widza,
- przestrzeń projekcji (ang. *clip / projection space*) - przestrzeń determinująca, które wierzchołki zobaczymy na ekranie oraz sposób w jakim je może obserwować kamera,

- przestrzeń okna (ang. *window space*) - ostatnie współrzędne, które zastrzegają aktualne pozycje obiektów i przekształcają je na zakres określony przez bibliotekę do renderowania pozwalający na wyświetlenie obrazu na ekranie.

Każdy nowy rysowany obiekt zajmuje miejsce relatywne do jego początku układu współrzędnych, czyli znajduje się w przestrzeni lokalnej. W tej przestrzeni mogą nastąpić transformacje modelu względem jego własnego środka. Przejście siatki (ang. *mesh*) do każdej kolejnej przestrzeni jest rezultatem iloczynu wszystkich wierzchołków struktury i określonej macierzy. Zmiana przestrzeni lokalnej na przestrzeń świata wymaga macierzy modelu (ang. *model matrix*), która najczęściej jest macierzą jednostkową (patrz 2.2.2). Wykorzystując macierz jednostkową upewniamy się, że źródłem świata będzie punkt $(0, 0, 0)$. Następnym etapem jest wykorzystanie macierzy widoku (ang. *view matrix*), która pozwala ustalić pozycję i orientację punktu z którego widz będzie obserwował scenę (patrz 2.2.3). Ostatnią transformacją przestrzeni jest przejście z układu widoku do układu klipu. Wykorzystywana jest macierz projekcji (patrz 2.2.4). Na rysunku 7 znajduje się przejście przez wszystkie przestrzenie współrzędnych aż do wygenerowanego obrazu. Transformacja sceny trójwymiarowej do ekranu 2D jest zadaniem potoku graficznego.



Rysunek 7: Przestrzenie współrzędnych [8].

2.2.6 Potok Graficzny

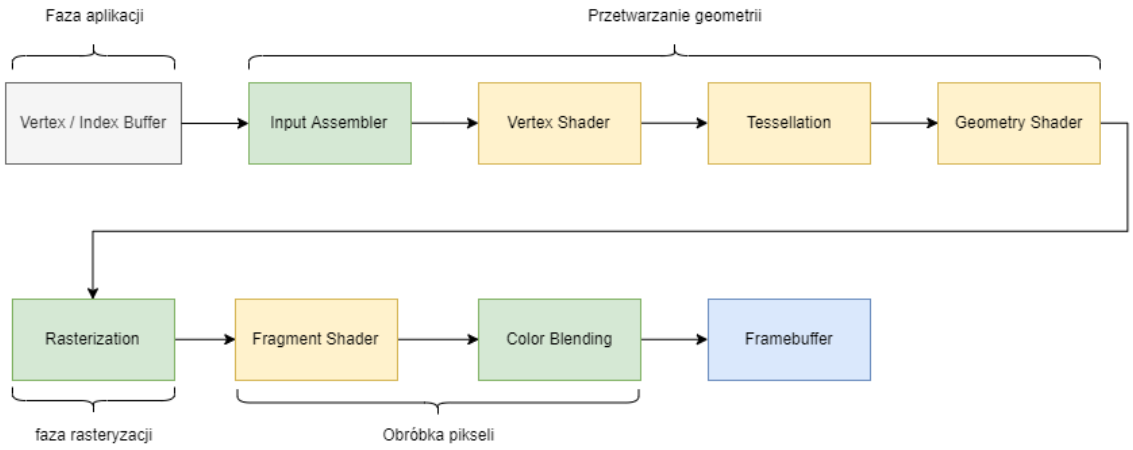
Potok graficzny (ang. *Graphics Pipeline*) jest sekwencją operacji, która na wejściu przyjmuje zbiór siatek obiektów 3D (patrz 2.2.1) oraz ich materiałów, a na wyjściu zwraca pokolorowane piksele, które mogą być wyświetlone na ekranie [12]. Zatem krótką definicją będzie transformacja sceny 3D w obraz 2D. Przystępna ilustracja ukazująca przykładowy potok graficzny znajduje się na rysunku 8.

Etapy potoku graficznego wykonują się na karcie graficznej, która jest przystosowana do tego rodzaju obliczeń. Jednostki umożliwiają deweloperowi programowanie określonym faz, w celu uzyskania indywidualnych rezultatów.

Oprogramowanie, które wykonuje się na GPU nazywa się shaderem. Moduł określa operacje, które są wykonywane dla każdego wierzchołka równolegle na określonych etapach potoku. W kodzie należy zdefiniować odgórnie zmienne wejściowe (ang. *Input Variables*) oraz wyjściowe (ang. *Output Variables*). Wspomniane wartości są transferowane kolejno przez wszystkie szczeble potoku renderingu [13].

Sam potok graficzny jest podzielony na cztery główne fazy: aplikacji, przetwarzania geometrii, rasteryzacji oraz obróbki pikseli. Ta struktura jest rdzeniem, który jest wykorzystywany w aplikacjach grafiki komputerowej. Idąc po kolei:

- faza aplikacji (ang. *Application Stage*) jest zarządzana przez program komputerowy, który jest zazwyczaj zaimplementowany na procesorze ogólnego celu (ang. general-purpose CPU). Jedyny etap potoku, który odbywa się na CPU. Niektóre z zadań, które są wykonywane na procesorze obejmują wykrywanie kolizji oraz symulację fizyki [5],
- przetwarzanie geometrii (ang. *Geometry Processing*), gdzie obliczane są transformacje oraz projekcje. Na tym etapie zapada decyzja, który obiekt zostaje narysowany, jak i gdzie powinien być narysowany. Przetwarzanie geometrii i etapy poniżej wykonują się na GPU [5],
- faza rasteryzacji (ang. *Rasterization Stage*), w przypadku wykorzystania trójkątów, przyjmuje na wejściu trzy wierzchołki i formuje z tego trójkąt. Następnie znajduje wszystkie piksele, które znajdują się wewnątrz utworzonego trójkąta [5],
- obróbka pikseli (ang. *Pixel Processing*) wykonuje obliczenia na każdy piksel w celu wyznaczenia koloru oraz może (ale nie musi) przeprowadzić badanie głębokości, czyli określenie pikseli widocznych bądź niewidocznych [5].



Rysunek 8: Uproszczony przegląd potoku graficznego w Vulkan. *Vertex/Index Buffer* znajdują się w etapie aplikacji. *Vertex Shader, Tesselation, Geometry Shader* są w fazie przetwarzania geometrii. *Rasterization* jak nazwa wskazuje do etapu rasteryzacji. Pozostałe, *Fragment Shader i Color Blending*, są obróbką pikseli. Etapy w żółtym kolorze są programowalne przez dewelopera, etapy zielone są predefiniowane [12].

2.3 System jednostek i komponentów

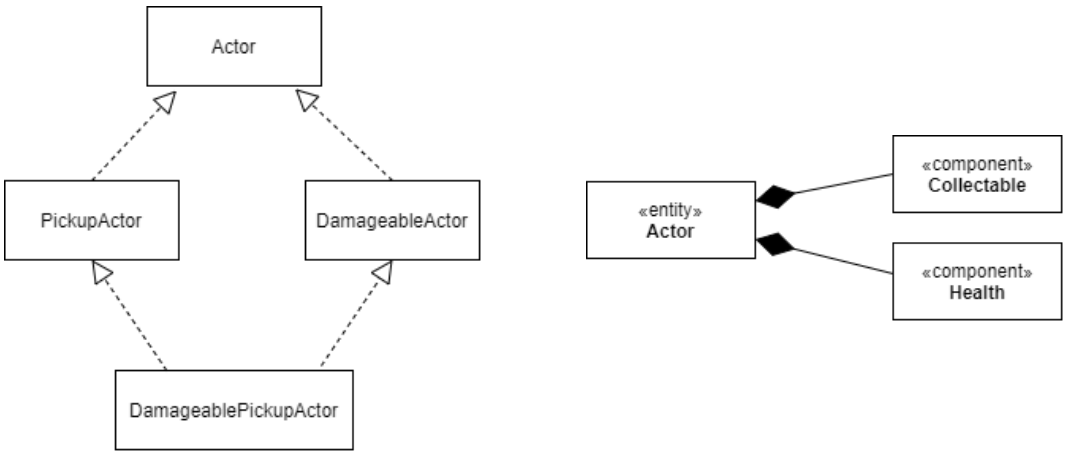
Pisząc zaawansowane oprogramowanie z wykorzystaniem języka zorientowanego obiektowo (ang. *Object-Oriented Programming Language*) dla jakości programu mają znaczenie interfejsy klas. Bardzo ważna jest wewnętrzna konstrukcja klas oraz ich implementacja, ponieważ czysty kod zapewni później prostotę przy pracy nad nowymi funkcjami [14].

Jednym z problemów podczas implementacji zaawansowanych systemów gier komputerowych jest dziedziczenie, które zazwyczaj jest wykorzystywane jako standardowy

sposób reprezentowania obiektów z coraz to nowymi funkcjami. Na początku programista decyduje się na interfejsy bardzo dobrze określające cel istnienia danego obiektu. Jednak wraz z postępem gry bądź silnika, który może być wykorzystany do innych projektów, rozszerza się hierarchia dziedziczenia i powstaje bardzo dużo klas pochodnych. Klasy pochodne mogą enkapsulować wewnątrz implementację nowych funkcji, bądź dziedziczyć ją od innej klasy bazowej. Kłopotem są narzucone nowe metody, które niekoniecznie będą wykorzystywane przez klasy pochodne co spowoduje dodatkowe wykorzystanie zasobów oraz powstanie długu technicznego [15]. Podczas wykorzystania dziedziczenia z notorycznie zwiększającą się liczbą nowych instancji trudnością będzie próba przestrzegania zasady podstawienia Liskov (ang. *LSP - Liskov Substitution Principle*). Barbara Liskov uzasadniła tezę, że po klasie bazowej należy dziedziczyć tylko i wyłącznie wtedy gdy, klasa pochodna faktycznie jest pewną jej szczególną odmianą. Innymi słowy, wszystkie procedury zdefiniowane w interfejsie powinny w klasach pochodnych mieć takie samo działanie [16].

Rozwiązaniem będzie system jednostek i komponentów (ang. *Entity Component System*) będący wzorcem architektonicznym (ang. *Architectural Pattern*) bardzo często wykorzystywanym przy tworzeniu gier komputerowych. Przestrzega on zasady kompozycji ponad dziedziczeniem (ang. *Composition over Inheritance*), aby zapewnić większą elastyczność w definiowaniu jednostek poprzez składanie spójnych obiektów z poszczególnych części, które można mieszać i dopasowywać. Koncepcją systemu są jednostki, będące kontenerami zazwyczaj hierarchicznymi do których można przypisywać moduły oraz komponenty reprezentujące określone funkcje, takie jak wygląd postaci, zachowania czy inne dane [17].

W ten sposób rozwiązywane są problemy dziedziczenia. Przykładem może być niejednoznaczność, gdzie potrzebna jest implementacja aktorów sceny *Actor* oraz jednostek reagujących na obrażenia *Damageable*. Programista zadaje sobie pytanie, czy hierarchia powinna wyglądać jako *Actor* <- *DamageableActor*, czy *Damageable* <- *Actor*. Następnie problemem będzie rozbudowa klas, która powinna być zduplikowana przykładowo jako *PickupActors* lub *DamageablePickupActors*. Wykorzystując system jednostek i komponentów możemy utworzyć jednostkę jako *Actor* i dodać do niej komponenty *Collectable* oraz *Health* [17]. Na ilustracji 9 zaprezentowano porównanie obydwu rozwiązań.



Rysunek 9: Porównanie wariantu dziedziczenia (a), gdzie występuje problem diamentu (ang. *diamond problem*) z systemem ECS (b).

3 Implementacja silnika graficznego

W tym rozdziale omówiono implementację projektu dyplomowego. Poruszono tematy takie jak wybór języka programowania, struktury projektu oraz jego funkcji. Opisano szczegóły dotyczące kodu źródłowego wzorca ECS, biblioteki matematycznej. Ponadto obszernie omówiono procedurę renderowania 3D i przedstawiono interfejs graficzny użytkownika.

3.1 Język programowania

Głównym celem inżynierii oprogramowania jest wytwarzanie software'u wysokiej jakości z punktu widzenia użytkowników, którzy oczekują szybkich i łatwych programów, oraz deweloperów, dla których ważna jest czystość kodu i modularność. Ponieważ silnik do gier komputerowych jest potężnym i skomplikowanym systemem należy dobrze rozważyć wybór języka programowania.

Zaczynając od reguł językowych - decydując się na imperatywny język programowania (np. C) - tworzenie gier komputerowych może być całkiem skomplikowanym procesem, ponieważ gry często są związane z manipulacją stanów obiektów bądź ich instancji. Przykładowo Pac-Man z 1980 jest bardzo prostą grą, gdzie implementacja opiera się na zdefiniowaniu kilku zmiennych globalnych (wynik, poziom, pozycje duchów itp.) oraz na funkcjach zmieniających wartość tych zmiennych. Jest to jak najbardziej realne, ponieważ logika gry z perspektywy dzisiejszych możliwości jest bardzo prosta. Niestety wątpliwe jest napisanie nowoczesnej i złożonej gry (tj. polski Cyberpunk 2077) wykorzystując jedynie rozwiązania funkcji i zmiennych globalnych. Rozbudowane projekty korzystają z możliwości zorientowanego-obiektowo języka programowania, dlatego że łatwiej jest zarządzać kodem z konceptem OOP w systemach o długości co najmniej kilkunastu milionów linii kodu.

Obiektowe języki programowania posiadają zasady, które są szeroko wykorzystane podczas wytwarzania gier komputerowych. Wśród nich znaleźć można pojęcia takie jak:

- klasy (ang. *classes*) będące zbiorami atrybutów (danych) i określonych zachowań (metod), które razem pozwalają uformować dobry i sensowny kod,
- enkapsulację (ang. *encapsulation*) dającą dostęp do wybranych zasobów utworzonych klas,
- dziedziczenie (ang. *inheritance*) rozszerzające istniejące już klasy. Oczywiście niektóre języki wspierają funkcjonalność wielokrotnego dziedziczenia mówiącego o klasie pochodnej posiadającej więcej niż jednego rodzica,
- polimorfizm (ang. *polymorphism*) jako właściwość języka programowania pozwalająca na manipulację kolekcji obiektów o innych typach danych przez pojedynczy wspólny interfejs.

Cechą idącą w parze z konceptem OOP jest kompozycja oraz agregacja, pojęcia zostały wyjaśnione w podrozdziale 2.3. Wykorzystując obiektowy język programowania zapewnimy wspomnianą w pierwszym akapicie czystość kodu i modularność.

Następnie należy rozważyć wykorzystanie języka pod względem wizji software'u przez użytkownika końcowego, któremu zależy na szybkim i intuicyjnym oprogramowaniu. Autorzy pracy *Energy Efficiency across Programming Languages* [18] przeanalizowali wiele

istniejących języków programowania pod względem kombinacji takich jak czas egzekucji, zużycie pamięci oraz wykorzystanie energii. Z rezultatów wynikło, że najlepszymi językami obiektowymi (posiadającymi koncepty wymienione powyżej) są C++ oraz Rust. Języki są kompilowane bezpośrednio na kod maszynowy, obsługują wskaźniki i bezpośredni dostęp do pamięci, wykorzystywane są na podobnych płaszczyznach (aplikacje IoT, programowanie hardware'u, sterowniki mikro kontrolerów itd.). Pomimo konkurencji na tej samej arenie C++ ma zdecydowanie silniejsze podstawy przez swoją długą żywotność - istnieje od 1979 roku, kiedy to Bjarne Stroustrup rozpoczął prace przy projekcie "C z klasami" - podczas, gdy Rust jest nową propozycją wśród programistów - Mozilla zaczęła sponsorować projekt w 2009, natomiast pierwszy kompilator powstał w 2011. I z tego też powodu C++ ma znacząco większą społeczność posiadającą doświadczenie w programowaniu. W skład C++ wchodzi bardzo dużo framework'ów, bibliotek i modułów. Dodatkowo w wytwarzaniu gier komputerowych C++ jest najczęściej wykorzystywanym językiem (Unity, Unreal Engine bądź CryEngine są napisane w C++).

Podsumowując zdecydowano się na C++ jako główny język programowania. Wybór ma pozwolić na prostotę implementacji, odpowiednią abstrakcję danych, dołączenie do ogromnej społeczności twórców gier komputerowych oraz bezpośredni dostęp do zasobów sprzętowych.

3.2 Struktura projektu

Żeby zachować niezależność silnika od pliku binarnego zdecydowano się rozdzielić rdzeń silnika (kompilować go jako bibliotekę statyczną) od formatu wykonywalnego (który będzie uruchamiany przez użytkownika). Uproszczona struktura katalogów silnika graficznego znajduje się na rysunku 10.

Główny korzeń (ang. *The Root*) jest miejscem w którym znajdują się pliki konfiguracyjne, dokumentacji bądź licencji oraz wszystkie pozostałe katalogi,

build/ - miejsce do którego trafiają skompilowane pliki i formaty wykonywalne.

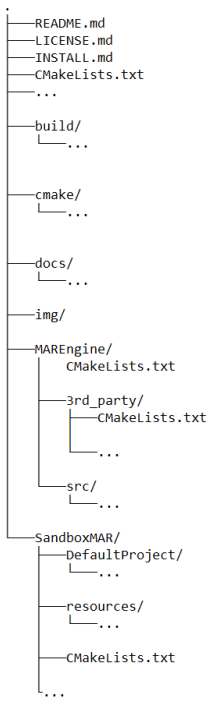
cmake/ - folder, gdzie znajdują się pliki pomagające znaleźć zewnętrzne biblioteki podczas procesu budowania projektu.

docs/ - katalog zawierający zupełnie oddzielne repozytorium, które odpowiada za dokumentację projektu oraz generowanie materiałów na podstawie plików źródłowych systemu.

resources/ - katalog zasobów statycznych, tutaj można znaleźć potrzebne zasoby.

MAREngine/ - miejsce wszystkich plików źródłowych rdzenia silnika graficznego (*src/*) oraz jego zależności (*3rd_party/*). To co znajduję się w tym katalogu będzie kompilowane do biblioteki statycznej.

SandboxMAR/ - folder posiadający pliki, które posłużą do zbudowania pliku binarnego. Ponadto zawarto w nim katalogi *DefaultProject/*, gdzie znajduje się domyślny projekt utworzony w silniku, oraz *resources/*, gdzie umieszczono pliku źródłowe shaderów oraz zasoby interfejsu graficznego.

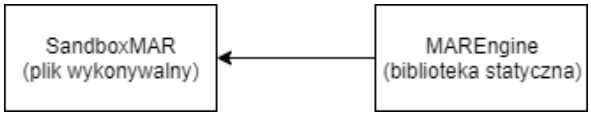


Rysunek 10: Przybliżone drzewko projektu.

3.3 Kompilacja silnika

Zdecydowano się poruszyć temat budowania projektu, ponieważ kreując duże oprogramowanie zależy nam na łatwości zarządzania kodem. Dla programisty duże znaczenie mają takie aspekty jak uniknięcie pisania bezwzględnych ścieżek dostępu, wykorzystanie bibliotek zewnętrznych, kompilacja kodu na wielu urządzeniach czy logiczna strukturą projektu. Wybierając narzędzie umożliwiające proste budowanie silnika graficznego należy wziąć pod uwagę wybrany język programowania (w tym przypadku C++), systemy budowania, czy wsparcie w IDE (ang. *Integrated Development Environment*) [19].

Dlatego też zdecydowano się na wieloplatformowe narzędzie do kompilacji, jakim jest CMake. Posiada ono natywne wsparcie dla języka C++, które jest wręcz dla niego stworzone. Przy jego pomocy można wykorzystać takie systemy budowania jak Clang i Visual Studio. Deweloperzy bibliotek zewnętrznych często dostarczają pliki źródłowe CMake'a, umożliwiające łatwą integrację z kodem oraz środowiska wytwarzania oprogramowania posiadają wsparcie CMake'a. Wykorzystując takie rozwiązanie uproszczono wiele procesów i umożliwiono łatwe zarządzanie kodem i budowanie silnika graficznego na wielu urządzeniach.

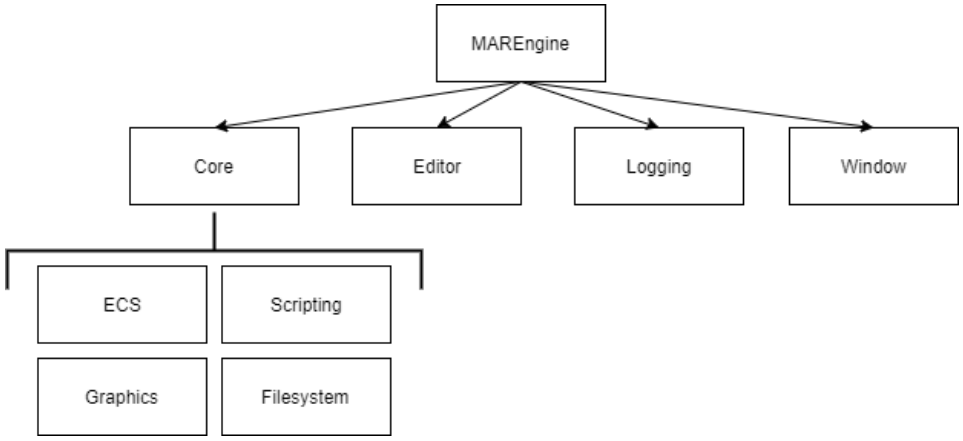


Rysunek 11: Struktura projektu jako biblioteka i plik binarny.

Na rysunku 10 z strukturą projektu specjalnie umieszczono pliki CMakeLists.txt, które odpowiadają za dołączenie określonych katalogów do projektu. W głównym korzeniu znajduje się najważniejszy plik, który zarządza całością i dołącza pozostałe moduły. Plik w katalogu *MAREngine/* jest odpowiedzialny za utworzenie biblioteki statycznej ze wszystkimi głównymi funkcjami silnika. Z tego powodu, że w projekcie są wykorzystane biblioteki zewnętrzne celem uniknięcia błędów linkera oraz łatwością operowania na tych bibliotekach utworzono kolejny plik w ścieżce *MAREngine/3rdparty*, który jest odpowiada za dołączenie wszystkich kodów zewnętrznych do silnika. Finalnym plikiem źródłowym CMake'a jest ten, który znajduje się w folderze *SandboxMAR/*. Opisano w nim budowę pliku wykonywalnego projektu, który jest uruchamiany przez użytkownika. Ilustracja 11 przedstawia wspomniany podział kodu na bibliotekę statyczną oraz plik binarny.

3.4 Funkcje oprogramowania

Zakres pracy obejmuje implementacje poszczególnych komponentów, które wchodzą w skład silnika do gier komputerowych i pełnią określone funkcje. Na ilustracji 12 przedstawiono konkretne części oprogramowania. Określone systemy należy traktować jako osobne projekty, które wykorzystane razem umożliwią napisanie projektu inżynierskiego.



Rysunek 12: Lista funkcji silnika MAREngine.

System składa się z czterech głównych komponentów:

- **Core** - najważniejsze ogniwo projektu, rozszerzeniami przynależącymi do rdzenia są moduły ECS (Entity-Component-System), system do renderowania grafiki 3D, system skryptowy pozwalający na opisywanie zachowań podczas rozgrywki oraz moduł systemu plików umożliwiający zapis / odczyt dokumentów,
- **Editor** - oddzielny komponent pełniący rolę graficznego interfejsu użytkownika, wykorzystując GUI użytkownik może manipulować powstającym produktem w czasie rzeczywistym,
- **Logging** - podsystem logowania umożliwiający debugowanie wydarzeń w kodzie.
- **Window** - moduł okienkowy, renderowana treść znajduje się nowo otwartym oknie poświęconemu silnikowi.

3.5 Realizacja biblioteki matematycznej

Osobny podrozdział postanowiono poświęcić implementacji biblioteki matematycznej. Renderowanie obiektów w 3D opiera się przede wszystkim na wykonywaniu wiele operacji matematycznych na każdym poszczególnym wierzchołku siatki. Moduł matematyczny nazwano zgodnie z ideą *MAREngine*, czyli *MARMaths*. Biblioteka składa się z kilku plików nagłówkowych .h oraz adekwatnych plików źródłowych o rozszerzeniu .cpp. Zdecydowano się na umieszczenie komponentu matematycznego w katalogu *3rdparty* wśród innych zewnętrznych bibliotek, ponieważ *MARMaths* zostało napisane w sposób pozwalający na podpięcie jej w innych silnikach. *MARMaths* nie jest modułem personalizowanym pod silnik graficzny *MAREngine*.

marmaths

Rysunek 13: Logo biblioteki matematycznej *MARMaths*.

Zgodnie z teorią dot. reprezentacji obiektów 3D (patrz 2.2.1) oraz sposobami manipulacji obiektów (patrz 2.2.2) każdy wierzchołek modelu będący punktem w przestrzeni trójwymiarowej jest zdefiniowany poprzez wektor w przestrzeni $\mathbb{R}^3$. Tak więc należy zacząć od struktury *vec3* przedstawionej na listingu 1. Najważniejszymi składowymi struktury są trzy zmienne typu *float* (liczby zmiennoprzecinkowe). Dodatkowo należało umożliwić programiście wykonywanie obliczeń poprzez implementację podstawowych własności jakimi są między innymi dodawanie (*add()*), odejmowanie (*substract()*), mnożenie (*multiply()*) i dzielenie (*divide()*) zarówno skalarów i innych wektorów. Grafika komputerowa wymaga też implementacji iloczynów skalarnego i wektorowego, wyznaczenia długości wektora, czy normalizacje.

```
struct vec3 {

        float x, y, z;

        vec3 add(float f) const;
        vec3 subtract(float f) const;
        vec3 multiply(float f) const;
        vec3 divide(float f) const;

        vec3 add(vec3 other) const;
        vec3 subtract(vec3 other) const;
        vec3 multiply(vec3 other) const;
        vec3 divide(vec3 other) const;

        vec3 cross(vec3 other) const;
        static vec3 cross(vec3 x, vec3 y);

        float dot(vec3 other) const;
        static float dot(vec3 left, vec3 right);

        float length() const;
        static float length(vec3 v);

        vec3 normalize() const;
        static vec3 normalize(vec3 other);

};
```

Listing 1: Przybliżona deklaracja struktury *vec3*.

Tak jak zostało wspomniane w części dot. przestrzeni współrzędnych (patrz 2.2.5) każde przejście do nowego układu współrzędnych jest wykonywane poprzez mnożenie wierzchołków modelu z macierzami pozwalającymi na takie przejście. Ponadto sama manipulacja obiektami opiera się na mnożeniu siatki z macierzami translacji, rotacji bądź skali. Dlatego też kolejnym ważnym krokiem była implementacja macierzy czterowymiarowych. Na listingu 2 znajduje się przybliżona deklaracja struktury *mat4*. W skład klasy wchodzi szesnaście zmiennych typu *float*, ponieważ macierz jest 4x4. Ważnymi metodami, które programista z pewnością będzie wykorzystywał są tworzenie macierzy jednostkowej i operacja mnożenia z innymi macierzami, wektorami w przestrzeni $\mathbb{R}^4$ bądź skalarami. Tak więc zaimplementowano metodę *identity()* oraz poszczególne warianty mnożenia *multiply()*. Zgodnie z sekcją dot. kamery 2.2.3 była potrzeba implementacji macierzy *lookAt*. Ponadto w części związanej z projekcjami (równoległej 2D jak i perspektywicznej

3D w rozdziale 2.2.4) zaprezentowane wzory, które również odnalazły się w kodzie pod deklaracjami *orthographic()* oraz *perspective()*. Ostatnimi macierzami wchodzącymi w skład struktury *mat4* są te związane z manipulacją obiektami w przestrzeni lokalnej, czyli macierze translacji *translation()*, rotacji *rotation()* i skali *scale()*.

```
1  struct mat4 {
2
3      float elements[4 * 4];
4
5      static mat4 identity();
6
7      mat4 multiply(const mat4& other) const;
8      vec4 multiply(const vec4& other) const;
9      mat4 multiply(float other) const;
10
11     static mat4 orthographic(float left, float right,
12                              float top, float bottom, float near, float far);
13     static mat4 perspective(float fov, float aspectRatio, float near, float far);
14
15     static mat4 lookAt(vec3 eye, vec3 center, vec3 y);
16
17     static mat4 translation(vec3 trans);
18     static mat4 rotation(float angle, vec3 axis);
19     static mat4 scale(vec3 scal);
20
21 };
```

Listing 2: Przybliżona deklaracja struktury *mat4*.

Powstanie struktury *mat4* wymaga utworzenia nowej struktury jaką jest *vec4*. Funkcjonalności implementacji wektora w przestrzeni $\mathbb{R}^4$ nie różnią się w porównaniu do napisanego kodu klasy *vec3*. Również zawarte są podstawowe operacje takie jak dodawanie, odejmowanie, mnożenie oraz dzielenie, dodatkowo przydatnymi metoda struktury są też normalizacja wartości wektora i jego długość. Celem zapewnienia integracji pomiędzy strukturami *vec3* i *vec4* dodano specjalne konstruktory, które przepiszą wartości z pierwszej instancji struktury do drugiej. Przykładowe deklaracje konstruktorów znajdują się na listingu 3.

```
1  struct vec4 {
2
3    vec4(float _x, float _y, float _z, float _w);
4    vec4(vec3 _v, float _w);
5
6  };
7
8  struct vec3 {
9
10   vec3(float _x, float _y, float _z);
11   vec3(const vec4& _v);
12
13 };
```

Listing 3: Porównanie deklaracji konstruktorów konwertujących w strukturach *vec3* i *vec4*.

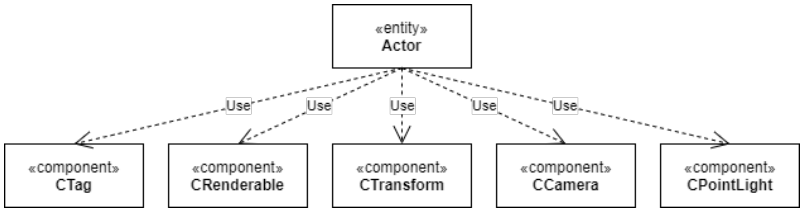
3.6 Wzorzec Entity Component System

Tak jak zostało wspomniane w podrozdziale 2.3 w ramach definicji jednostek istniejących w grach zostanie wykorzystany wzorzec architektoniczny Entity Component System. Gry komputerowe opisują jednostkę bardzo podobnie - jednostka (ang. *entity*) jest obiektem istniejącym w świecie, który zazwyczaj jest widoczny oraz może się przemieszczać. Przykładowymi jednostkami są samochód, kamienie, bronie wykorzystywane przez gracza itp. Żeby obiekt był widoczny musi mieć przypisaną siatkę modelu, odpowiednie tekstury należące do modelu, dodatkowo należy opisać jego zachowanie podczas rozgrywki. Tym samym wprowadzono pojęcie komponentu będącego informacją dot. właściwości określonej jednostki. Żeby jednostka przyjęła określone funkcje należy przypisać do niej wybrany komponent. Tak więc zdefiniowano:

- jednostkę (ang. *entity*) jako obiekt istniejący w świecie gry będący kontenerem komponentów,
- komponent (ang. *component*) jako strukturę posiadającą surowe informacje oraz parametry, które następnie mogą zostać przypisane do jednostek,
- system jako sposób obsługi wszystkich komponentów należących do jednostek.

W celu prostej implementacji modułu ECS wykorzystano bibliotekę entt, która w prosty i przystępny sposób realizuje wiele wymaganych funkcjonalności. Ponadto jest wykorzystywana w znanych produkcjach, co potwierdza jej stabilność na przestrzeni lat.

Na początek zdecydowano się zaimplementować komponenty celem określenia wymagań dla modułu do renderowania oraz interfejsu graficznemu. Przyjęto konwencje dot. nazw komponentów, tzn. pierwsza litera nazwy będzie "C" celem rozpoznania struktur będących komponentami dynamicznie. Tak więc podstawową informacją posiadaną przez każdą jednostkę jest nazwa wymagana podczas edycji sceny w interfejsie graficznym. Zatem dodano komponent *CTag* zawierający łańcuch znaków któremu użytkownik może nadać tag. Oprócz tego potrzebną strukturą jest *CRenderable* posiadające informacje w jaki sposób obiekt powinien być renderowany. Jej składowe pozwalają wyciągnąć dane dot. modelu, przypisanych tekstur oraz grupy renderującej (ang. *batch*, zostaną omówione później w podrozdziale 3.7.5). Aktualna pozycja, rotacja i skala modelu również są wymagane ze strony jednostki, zatem dodano strukturę *CTransform*. Scena bez jakiegokolwiek źródła światła byłaby cała czarna, dlatego też dodano komponent *CPointLight* reprezentujący typ światła punktowego. Ponadto jedna jednostka powinna być obiektem reprezentującym kamerę, tak więc dodano komponent *CCamera* posiadający składowe pozwalające na utworzenie projekcji zarówno perspektywicznej jak i równoległej. Zbiór wszystkich komponentów zaprezentowano na rysunku 14.



Rysunek 14: Prezentacja wszystkich komponentów przypisanych do jednostki.

Później zaimplementowano klasę *Entity* reprezentującą każdy istniejący w silniku obiekt do którego można przypisać określone funkcje, tj. przypisać komponenty. Zgodnie z założeniami wzorca ECS zadeklarowano możliwość dodania struktury poprzez metodę *addComponent()*, podmiany istniejącego za wykorzystaniem *replaceComponent()*, bądź usunięcia z jednostki określonych funkcji poprzez *removeComponent()*. Wykorzystano do tego możliwości języka C++ - szablony, czyli generyczne kreowanie funkcji podczas kompilacji projektu. Tym samym udostępniono możliwość dodania każdej struktury do jednostki. Wartymi uwagi są składowe klasy, gdzie *m_entityHandle* jest unikalnym identyfikatorem pozwalającym na definicję oryginalnych jednostek (*entt::entity* to typ zdefiniowany przez bibliotekę entt za którym kryje się unsigned integer 32-bitowy) oraz *m_pSceneRegistry* będące rejestrem wszystkich utworzonych jednostek (w entt do komponentów odwołujemy się poprzez rejestr stąd wymagany wskaźnik na rejestr). Deklaracja klasy *Entity* znajduje się na listingu 4.

```
class Entity {
public:

  Entity() = delete;
  Entity(entt::registry* pSceneRegistry);

  MAR_NO_DISCARD bool isValid() const;
  void destroyYourself() const;

  static void fillEntityWithBasicComponents(const Entity& entity);

  template<typename TComp> MAR_NO_DISCARD bool hasComponent() const;
  template<typename TComp, typename... Args> TComp& addComponent(Args&&... args) const;
  template<typename TComp> MAR_NO_DISCARD TComp& getComponent() const;
  template<typename TComp> TComp& replaceComponent(const TComp& other) const;
  template<typename TComp> void removeComponent() const;

private:

  entt::registry* m_pSceneRegistry{ nullptr };
  entt::entity m_entityHandle{ entt::null };

};
```

Listing 4: Przybliżona deklaracja klasy *Entity*.

Silnik do gier komputerowych nie opiera się jedynie na klasie jednostki *Entity*, wymaga również implementacji sceny, która mówiąc abstrakcyjnie, jest poziomem, miejscem w którym aktualnie przebywa gracz. Scenę należy interpretować jako kompletny etap gry, jako kontener posiadający rejestr do jednostek, tak więc scena jest odpowiedzialna za kontrolowanie życia wszystkich istniejących obiektów. Tak więc zaimplementowano klasę *Scene* posiadającą metody takie jak *createEntity()* i *destroyEntity()* mogące tworzyć i niszczyć jednostki w świecie gry. Dodano też tworzenie podstawowej sceny *createEmptyScene()* której odpowiedzialnością jest utworzenie dwóch jednostek - kamery oraz punktowego światła. Tym samym użytkownik operuje wszystkimi istniejącymi obiektami na scenie. Przybliżona implementacja deklaracji klasy *Scene* znajduje się na listingu 5.

```cpp
class Scene {
public:

  Scene() = delete;
  explicit Scene(std::string name);

  MAR_NO_DISCARD static Scene createEmptyScene(std::string sceneName);
  void close();

  MAR_NO_DISCARD const Entity& createEntity();
  void destroyEntity(const Entity& entity);
  MAR_NO_DISCARD const FEntityArray& getEntities() const;

  MAR_NO_DISCARD entt::registry* getRegistry();
  template<typename TComponent> MAR_NO_DISCARD auto getView();
};
```

Listing 5: Przybliżona deklaracja klasy *Scene*.

Ostatnią wartą wspomnienia klasą jest menedżer zarządzający sceną, czyli *FSceneManagerEditor*. Klasa odpowiedzialna jest za działanie i aktualizowanie sceny podczas działania silnika. Posiada metody, gdzie pierwsza przygotowuje moduł ECS do startu przy pomocy *initialize()* oraz drugą, która kończy jego pracę *close()*. Jednak jak nazwa *FSceneManagerEditor* wskazuje, klasa jest upoważniona do zarządzania sceną podczas trybu edytora. Zaimplementowano takie metody jak *update()*, która powinna być uruchomiana w pętli podczas renderowania wszystkich ramek celem aktualizowania sceny. Ponadto posiada funkcję ustawienia trybu gry bądź edytora (*setEditorMode()* i *setPlayMode()*) oraz przejrzenia sceny wykorzystując kamerę edytora albo kamerę gry (*useEditorCamera()* i *useGameCamera()*). Listing 6 przedstawia deklarację klasy.

```cpp
class FSceneManagerEditor {
public:

  void initialize(Scene* pScene, FBatchManager* pBatchManager, FMeshManager* pMeshManager);
  void update();
  void close();

  void setEditorMode();
  MAR_NO_DISCARD bool isEditorMode() const;
  void setPlayMode();
  MAR_NO_DISCARD bool isPlayMode() const;

  void useEditorCamera();
  MAR_NO_DISCARD bool usingEditorCamera() const;

  void useGameCamera();
  MAR_NO_DISCARD bool usingGameCamera() const;

private:

  Scene* m_pScene{ nullptr };

};
```

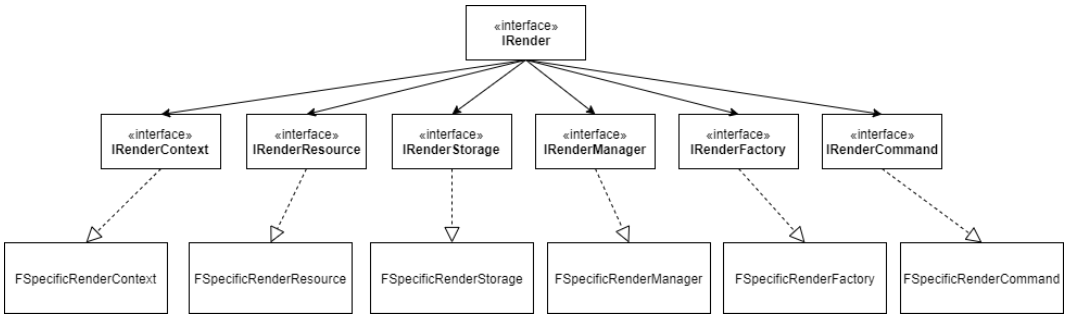
Listing 6: Przybliżona deklaracja klasy *FSceneManagerEditor*.

3.7 Renderowanie sceny 3D

Generowanie obrazu trójwymiarowego, następnie wyświetlanie rezultatu na wyświetlaczu dwuwymiarowym jest zadaniem niezwykle trudnym. Bardzo wiele zależy od podejścia do tematu oraz początkowych założeń działania systemu.

3.7.1 Abstrakcja modułu graficznego

Podczas implementacji systemu graficznego, który ma wyrenderować scenę 3D, zdecydowano się na rozdzielenie faktycznych klas z kodem źródłowym od klas wykorzystywanych przez użytkownika. Dzięki takiemu rozwiązaniu uzyskano czystość kodu oraz jego przejrzystość. Z łatwością można zapoznać się z metodami abstrakcyjnymi wymagającymi implementacji. Na rysunku 15 znajduje się przykładowe drzewko dziedziczenia systemu graficznego. Poniżej opisano też konkretne interfejsy, które później wymagały swojej implementacji.



Rysunek 15: Schemat dziedziczenia z interfejsu *IRender* do poszczególnych implementacji.

- *IRender* - rodzic całego drzewka dziedziczenia modułu generującego obraz,
- *IRenderContext* - główny interfejs opisujący implementację potencjalnej biblioteki graficznej jakimi są OpenGL oraz Vulkan, poszczególne implementacje muszą w adekwatny sposób zarządzać kontekstem wybranej biblioteki,
- *IRenderResource* - abstrakcja opisująca wszystkie zasoby potrzebne do działania silnika oraz gry komputerowej, są to na przykład bufory danych, załadowane materiały, tekstury bądź potoki graficzne,
- *IRenderStorage* - abstrakcyjny magazyn wszystkich kontenerów zasobów, *IRenderResource* powinny być składowane w takich magazynach i dostęp jest tylko poprzez odpowiednie metody,
- *IRenderManager* - interfejs menadżera, klasy pochodne powinny zapewniać odpowiednie sposoby zarządzania instancjami klas pochodnych *IRenderResource*,
- *IRenderFactory* - abstrakcyjna fabryka, odpowiedzialna za tworzenie instancji konkretnych klas pochodnych *IRenderResource*,
- *IRenderCommand* - interfejs komend, które może wykonywać system renderujący, są to przykładowo *drawSquare()* bądź *drawModel()* odpowiedzialne za rysowanie siatek na ekranie.

3.7.2 Kontekst biblioteki graficznej - OpenGL vs. Vulkan vs. DirectX

Gdy zaznajomiono się z abstrakcją systemu graficznego należy przystąpić do implementacji wszystkich dependencji, których owy system wymaga, oraz należy zdecydować się na konkretną bibliotekę graficzną. Do dyspozycji są OpenGL, Vulkan i DirectX. Wspomniane biblioteki są tak zwanymi pośrednikami udzielającymi dostęp do kontroli graficznego systemu na wybranej karcie graficznej. Biblioteki różnią się zdecydowanie swoją żywotnością i wykorzystaniem w aplikacjach graficznych. OpenGL zostało wydane jeszcze w 1992 roku i przez wiele lat było standardem dla branży graficznej [20]. Natomiast Vulkan na rynku jest od 2016 roku, przez co jest mniej materiałów dot. technologii co za tym idzie moduł jest rzadziej wykorzystana w aplikacjach komercyjnych. Vulkan jest często określanym jako explicit API, czyli każda procedura jaką wykonuje CPU musi być odgórnie napisana w kodzie. Programista jest odpowiedzialny za prawie wszystkie operacje. Sterownik jest oprogramowaniem przyjmującym komendy oraz dane celem przetłumaczenia ich w takich sposób, żeby jednostki sprzętu komputerowego mogły go zrozumieć. Dla porównania, w OpenGL sterowniki śledzą stan obiektów, zarządzają pamięcią i synchronizacją oraz podczas działania aplikacji poszukują nowych błędów, tym samym wiele operacji jest wykonywanych bez wiedzy programisty. Tak więc Vulkan jest modułem wymagającym ogromu pracy, żeby uzyskać cokolwiek działającego [21]. Z kolei DirectX istnieje od 1995 roku, a oprogramowanie należy do firmy Microsoft. Dlatego też wielkim minusem biblioteki jest brak wieloplatformowości, producent zapewnia wsparcie tylko dla systemów Windows oraz konsol Xbox. Ponadto język shaderów się różni w porównaniu do poprzedników - w przypadku DirectX propozycją jest HLSL, OpenGL i Vulkan wykorzystują GLSL. Celem zapewnienia w przyszłości wsparcia wielu platform przez silnik zrezygnowano z DirectX. Porównano implementację kodu, który pozwala na praktyczne generowanie prostego trójkąta na ekranie co jest podstawą każdego silnika graficznego. Kod źródłowy wykorzystujący OpenGL miał zaledwie 150 linijek, natomiast wersja Vulkana zmieściła się w ok. 1000 liniach.

W silniku *MAREngine* zdecydowano się na OpenGL celem sprawniejszej realizacji modułu graficznego i szybszego efektu uzyskania działającego kodu, który pozwoli uzyskać zadowalające efekty. Powstała zatem klasa *FRenderContextOpenGL* na podstawie której zaimplementowano następnie pozostałe. Na listingu 7 znajduje się aktualny kod źródłowy klasy. Zgodnie z drzewkiem dziedziczenia z ilustracji 15 klasa *FRenderContextOpenGL* będąca "sprecyzowaną" implementacją dziedziczy po klasie bazowej *FRenderContext*. Posiada metody takie jak:

- *create(FWindow* pWindow)* - odpowiada za utworzenie nowego kontekstu graficznego i adaptację okna do konkretnej biblioteki,
- *close()* - zamyka kontekst, tym samym uniemożliwia dalsze generowanie grafiki,
- *prepareFrame()* - przygotowuje nową ramkę obrazu,
- *endFrame()* - kończy wyrenderowaną ramkę obrazu,
- *metody getBufferFactory(), getShadersFactory(), getPipelineFactory(), getFramebufferFactory(), getMaterialFactory()* - zwracają fabryki kolejno buforów danych, shaderów, potoków graficznych, buforów ramek oraz materiałów,

- *metody getBufferStorage(), getShadersStorage(), getPipelineStorage(), getFramebufferStorage(), getMaterialStorage()* - zwracają magazyny kolejno buforów danych, shaderów, potoków graficznych, buforów ramek oraz materiałów.

Warto zwrócić uwagę na fakt, że składowymi klasy *FRenderContextOpenGL* są konkretne implementacje poszczególnych klas, ponieważ wymagają one kontekstu biblioteki OpenGL. Zwracany jest wskaźnik do klasy bazowej, pozwalając tym samym na wykorzystywanie klas bazowych. Dzięki takiemu rozwiązaniu uzyskano efekt ukrytej implementacji, programista wykorzystujący moduły nie musi znać implementacji, potrzebuje jedynie znajomości sposobu wykorzystania klas bazowych.

```
1  class FRenderContextOpenGL : public FRenderContext {
2  public:
3
4    bool create(FWindow* pWindow) final;
5    void close() final;
6
7    void prepareFrame() final;
8    void endFrame() final;
9
10   MAR_NO_DISCARD ERenderContextType getType() const final;
11
12   MAR_NO_DISCARD FBufferStorage* getBufferStorage() const final;
13   MAR_NO_DISCARD FShadersStorage* getShadersStorage() const final;
14   MAR_NO_DISCARD FPipelineStorage* getPipelineStorage() const final;
15   MAR_NO_DISCARD FFramebufferStorage* getFramebufferStorage() const final;
16   MAR_NO_DISCARD FMaterialStorage* getMaterialStorage() const final;
17
18   MAR_NO_DISCARD FBufferFactory* getBufferFactory() const final;
19   MAR_NO_DISCARD FShadersFactory* getShadersFactory() const final;
20   MAR_NO_DISCARD FPipelineFactory* getPipelineFactory() const final;
21   MAR_NO_DISCARD FFramebufferFactory* getFramebufferFactory() const final;
22   MAR_NO_DISCARD FMaterialFactory* getMaterialFactory() const final;
23
24 private:
25
26   FPipelineFactoryOpenGL m_pipelineFactory;
27   FShadersFactoryOpenGL m_shadersFactory;
28   FBufferFactoryOpenGL m_bufferFactory;
29   FFramebufferFactoryOpenGL m_framebufferFactory;
30   FMaterialFactoryOpenGL m_materialFactory;
31   FWindow* m_pWindow{ nullptr };
32
33 };
```

Listing 7: Przybliżona deklaracja klasy *FRenderContextOpenGL*.

3.7.3 Bufory danych

Następnym etapem utworzenia modułu graficznego było napisanie kodów źródłowych dla wspomnianych wyżej fabryk oraz magazynów. Rozpoczęto od buforów danych. W OpenGL bufory są obszarami pamięci, które mogą być wykorzystywane na wiele sposobów. Są reprezentowane przez określone nazwy, gdzie domyślny typ danych dla nazw to *GLuint* - odpowiednik typu *unsigned integer* 32-bitowego. Dopiero wtedy, gdy nazwa zo-

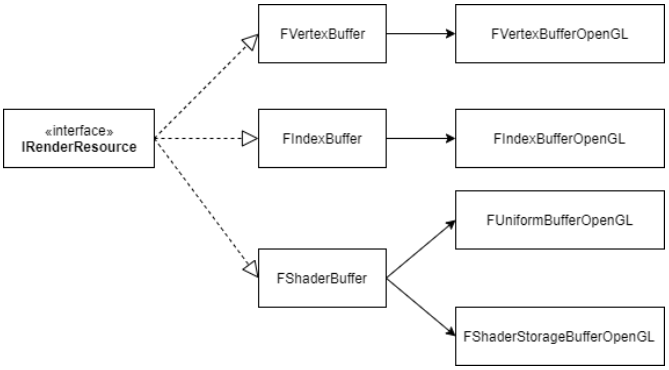
stała przydzielona, można alokować pamięć oraz umieścić w niej dane [22]. Przykładowe tworzenie buforu znajduje się na listingu 8.

```
// Typ danych wykorzystywany jako nazwy w OpenGL to GLuint
GLuint buffer;
// Generowanie nazwy dla buforu
glGenBuffers(1, &buffer);
// Wiązanie buforu z określonym typem wiążącym, w tym przypadku GL_ARRAY_BUFFER
glBindBuffer(GL_ARRAY_BUFFER, buffer);
// Specyfikacja jak wiele pamięci należy zarezerwować pod przyszłe wykorzystanie
glBufferData(GL_ARRAY_BUFFER, 1024 * 1024, nullptr, GL_STATIC_DRAW);
```

Listing 8: Tworzenie buforu danych w OpenGL z alokacją wskaźnika o wartości *nullptr*.

OpenGL jako biblioteka pracuje w postaci maszyny stanów ciągle aktualizującej zawartość danych. Dlatego też nie posiada określonych typów danych, w tym przypadku buforów danych. Są nimi jedynie typy wiążące (ang. *binding points*) określające w jaki sposób powinien zachowywać się określony bufor. Wykorzystując ten fakt postanowiono wykorzystać je na korzyść utworzenia konkretnych typów, które pozwolą w łatwiejszy sposób realizować co niektóre operacje. Najważniejszymi typami wiążącymi są:

- *GL_ARRAY_BUFFER* - wykorzystywany jako źródło dla danych wszystkich wierzchołków załadowanych siatek obiektów, w grafice komputerowej nazywany jako VBO - Vertex Buffer Object,
- *GL_ELEMENT_ARRAY_BUFFER* - indeksuje wszystkie wierzchołki w VBO, wykorzystywane celem optymalizacji (patrz 2.2.1), znane jako IBO - Index Buffer Object, czasem spotykane EBO - Element Buffer Object,
- *GL_UNIFORM_BUFFER* - składuje dane dla shaderów (takie jak macierze bądź określone tekstury), zwany potocznie UBO - Uniform Buffer Object,
- *GL_SHADER_STORAGE_BUFFER* - tak jak UBO, składuje dane dla shaderów, jednak SSBO (Shader Storage Buffer Object) może wykorzystać zdecydowanie więcej pamięci (16KB vs. 128MB) oraz może mieć zmienną liczbę danych, podczas gdy UBO wymaga odgórnie wybranej alokacji [23].



Rysunek 16: Schemat dziedziczenia klas implementujących bufory danych.

Tym samym zaimplementowano adekwatne klasy, które odpowiadają za bufory danych, widoczne na rysunku 16. Natomiast na listingu 9 umieszczono deklarację klasy *FVertexBufferOpenGL*. Pozostałe nie różnią się od siebie poza typem wiążącym.

```
class FVertexBufferOpenGL : public FVertexBuffer {
public:

  void create(int64 memoryToAllocate) final;
  void free() final;
  void destroy() final;

  void bind() const final;
  void update(const FVertexArray& vertices) const final;
  void update(const float* data, uint32 offset, uint32 sizeOfData) const final;
  void update(const int32* data, uint32 offset, uint32 sizeOfData) const final;
  void update(const uint32* data, uint32 offset, uint32 sizeOfData) const final;

private:

  static constexpr GLenum m_glBufferType{ GL_ARRAY_BUFFER };
  uint32 m_id{ 0 };

};
```

Listing 9: Deklaracja klasy *FVertexBufferOpenGL*.

3.7.4 Programowalne etapy potoku graficznego - shadery

Kolejnym etapem kreowania modułu graficznego jest zaprogramowanie shaderów, czyli tak jak zostało wspomniane w podrozdziale 2.2.6, programów wykonujących komendy zdefiniowane przez użytkownika na określonych etapach potoku graficznego. Potok na kilku fazach jest programowalny, gdzie każda faza reprezentuje określony typ procesowania danych. Potok graficzny biblioteki OpenGL definiuje poniższe etapy jako programowalne:

- *Vertex Shader (GL_VERTEX_SHADER)* - wykonuje zdefiniowane komendy na każdy pojedynczy wierzchołek siatki modelu,
- *Tesselation Control (GL_TESS_CONTROL_SHADER)* - kontroluje ilość teselacji jaką otrzyma dany fragment potoku,
- *Evaluation Shaders (GL_TESS_EVALUTION_SHADER)* - na podstawie rezultatów oblicza ich interpolowane pozycje i inne dane dotyczące wierzchołków,
- *Geometry Shader (GL_GEOMETRY_SHADER)* - reguluje przetwarzanie prymitywów (czyli trójkątów),
- *Fragment Shader (GL_FRAGMENT_SHADER)* - przetwarza fragment wygenerowany przez rasteryzację na zestaw kolorów oraz wartość głębi (ang. *depth value*),
- *Compute Shader (GL_COMPUTE_SHADER)* - używany wyłącznie do obliczania arbitralnych informacji. Może wykonywać renderowanie, jednak generalnie nie powinien być wykorzystywany do zadań związanych z rysowaniem trójkątów.

W OpenGL shadery tworzymy poprzez tak zwane Obiekty GLSL, które enkapsulują skompilowane i zlinkowane shadery. Zgodnie z ideą programu oraz maszyną stanów OpenGL opisano pseudokod umożliwiający konkretną implementację widoczny na listingu 10.

```
// Tworzenie pustego programu
GLuint shaderPipelineID = glCreateProgram();
// Tworzenie pustego shadera (w tym przypadku Vertex Shader)
GLuint shaderID = glCreateShader(GL_VERTEX_SHADER);
// Ustawianie kodu źródłowego załadowanego shadera z pliku
glShaderSource(shaderID, 1, &sourceCode, nullptr);
// Kompilacja shadera
glCompileShader(shaderID);
// Dołączanie skompilowanego shadera do utworzonego programu
glAttachShader(shaderPipelineID, shaderID);
// Linkowanie programu
glLinkProgram(shaderPipelineID);
// Walidowanie programu
glValidateProgram(shaderPipelineID);
```

Listing 10: Pseudokod opisujący tworzenie programu shaderów w OpenGL.

Część związana z shaderami jest taka sama dla każdego shadera, jedyną różnicą jest parametr dot. wybranego typu shadera i załadowanego kodu źródłowego, dlatego też zdecydowano się na implementację jednej klasy *FShadersOpenGL*. Klasa wymaga podania ścieżek do określonych shaderów, które następnie zostaną skompilowane i wykorzystane podczas realizacji potoku graficznego. Oczywiście nie muszą być wykorzystane wszystkie shadery.

```
class FShadersOpenGL : public FShaders {
public:

  void passShader(EShaderType type, const char* path) final;

  void compile() final;
  void close() final;
  void bind() final;

};
```

Listing 11: Przybliżona deklaracja klasy *FShadersOpenGL*.

Opisując klasę *FShadersOpenGL* nie można nie pokazać chociażby jednego kodu źródłowego wykorzystywanego shadera w silniku, zatem na listingu 12 znajduje się Vertex Shader. Tłumacząc po kolei, aktualny kod shadera zaczyna się od podania wersji na której programista bazuje, w tym przypadku jest to wersja 4.5.0 GLSL (OpenGL Shading Language). Następnie są podawane wszystkie zmienne wejściowe, później wyjściowe. Kolejnym etapem jest definicja wszystkich UBO bądź SSBO. Ostatnią częścią kodu źródłowego shadera jest funkcja main(), która wykonuje się na każdym wierzchołku. Oczywiście w kodzie może definiować wiele innych funkcji, w tym przypadku nie ma takiej potrzeby. Sam kod jest bardzo podobny do C/C++ pod względem składni, jednak ma odrobinę inne właściwości.

```
#version 450

// Podawanie zmiennych wejściowych, wykorzystując słowo kluczowe "in"
layout(location = 0) in vec3 position;
layout(location = 1) in vec3 lightNormal;
layout(location = 2) in vec2 texCoord;
layout(location = 3) in float shapeIndex;

// Podawanie zmiennych wyjściowych, wykorzystując słowo kluczowe "out"
layout(location = 0) out vec3 v_Position;
layout(location = 1) out vec3 v_lightNormal;
layout(location = 2) out vec2 v_texCoords2D;
layout(location = 4) out flat int v_shapeIndex;

// Definicja SSBO obiektu Kamery, posiadającego jedną macierz 4x4
layout(std430, binding = 0) buffer CameraSSBO {
        mat4 MVP;
} Camera;

// Definicja SSBO obiektu Transformacji, posiadającego 32 macierze 4x4
layout(std430, binding = 5) buffer TransformSSBO {
        mat4 Transform[32];
} Transforms;

// Funkcja main(), która się wykonuje na każdym wierzchołku
void main() {
        // Kalkulacja nowej pozycji dla wierzchołka wykonując iloczyn
        // pozycja = MVP * Transform * lokalna_pozycja
        int intShapeIndex = int(shapeIndex);
        vec4 vertexComputed = Transforms.Transform[intShapeIndex] * vec4(position, 1.f);
        gl_Position = Camera.MVP * vertexComputed;

        // Przekazywanie parametrów do dalszych etapów potoku graficznego
        v_Position = vertexComputed.xyz;
        v_lightNormal = lightNormal;
        v_texCoords2D = texCoord;
        v_shapeIndex = intShapeIndex;

}
```

Listing 12: Vertex Shader wykorzystywany w silniku MAREngine.

3.7.5 Zarządzanie siatkami modeli

Kod źródłowy opisujący abstrakcję modeli musi pozwolić na wykorzystanie obiektów podstawowych (takich jak kostka bądź piramida) tak więc zaimplementowano klasę *FMeshProxy*, która ma reprezentować rozwiązanie "pośrednika". Następnie dodano klasy podstawowe posiadające wierzchołki z ich indeksami - *FMeshCube* (kostka), *FMeshPyramid* (piramida) oraz *FMeshSurface* (powierzchnia). Wspomniane klasy odgórnie posiadają zdefiniowane dane pozwalające je zidentyfikować w interfejsie graficznym jak i przesłać je do renderowania. Silnik graficzny jednak nie może opierać się tylko i wyłącznie na podstawowych typach obiektów, bardzo ważne jest umożliwienie ładowanie modeli z zewnątrz. Tak więc deweloperzy mogą kreować wszelakiego rodzaju gry komputerowe. Dodano zatem klasę *FMeshExternal* posiadającą możliwość ładowania obiektów z lokalnych plików. Listing 13 przedstawia deklaracje wszystkich klas reprezentujących siatki modeli.

```
class FMeshProxy : public IMeshProxy {
public:

  MAR_NO_DISCARD const FVertexArray& getVertices() const final;
  MAR_NO_DISCARD const FIndicesArray& getIndices() const final;
  MAR_NO_DISCARD EMeshType getType() const final;

};

class FMeshExternal : public FMeshProxy {
public:

    void load(const std::string& path);
    MAR_NO_DISCARD const FMeshExternalInfo& getInfo() const;
    MAR_NO_DISCARD const char* getName() const final;

};

class FMeshCube : public FMeshProxy {
public:
    FMeshCube();
    MAR_NO_DISCARD const char* getName() const final;
};

class FMeshPyramid : public FMeshProxy {
public:
    FMeshPyramid();
    MAR_NO_DISCARD const char* getName() const final;
};

class FMeshSurface : public FMeshProxy {
public:
    FMeshSurface();
    MAR_NO_DISCARD const char* getName() const final;
};
```

Listing 13: Deklaracje klas *FMeshProxy*, *FMeshExternal*, *FMeshCube*, *FMeshPyramid*, *FMeshSurface*.

W module graficznym silnika *MAREngine* wykorzystano grupowanie obiektów. Zatem możliwe jest renderowanie kilku modeli równolegle co zapewnia oszczędność czasu jak i zasobów, ponieważ rzadziej uruchamiamy jest proces rysowania (tj. przejścia przez wszystkie etapy potoku graficznego). Na listingu 15 przedstawiono różnicę pomiędzy renderowaniem obiektów pojedynczo oraz w grupie. Dodano więc klasę *FMeshBatch* będącą klasą bazową podsystemu grupowania. Pochodne deklaracje reprezentują określone typy modeli z odpowiadającymi materiałami, tj. modele z przydzielonym jedynie kolorem mogą być wspólną renderowaną serią (klasa *FMeshBatchStaticColor*) oraz modele z przypisaną teksturą załadowaną z pliku (*FMeshBatchStaticTex2D*). W ten sposób zaprojektowano podsystem ze względu na różniące się kody źródłowe shaderów (jeden specjalny dla kolorów, drugi dla tekstur). Klasy muszą posiadać zaimplementowane metody:

- *shouldBeBatched(entity)* określającą, czy podana jednostka powinna być przechowywana w tej sekcji,
- *canBeBatched(entity)* sprawdzającą, czy konkretna instancja klasy może przechowywać podaną jednostkę,

- *submitToBatch(entity)* wysyłającą jednostkę do instancji klasy, dzięki czemu komponent *CRenderable* będzie magazynowany w tej porcji siatek modeli.

Pozostałe funkcjonalności mają pozwolić na aktualizowanie grup modeli w czasie rzeczywistym oraz wyciągnięcie informacji z wybranej instancji. Deklarację klasy zaprezentowano na listingu 14.

```
class FMeshBatch : public IMeshBatch {
public:

  void reset() override;

  MAR_NO_DISCARD bool shouldBeBatched(const Entity& entity) const override;
  MAR_NO_DISCARD bool canBeBatched(const Entity& entity) const override;
  void submitToBatch(const Entity& entity) override;

  MAR_NO_DISCARD const FVertexArray& getVertices() const final;
  MAR_NO_DISCARD const FIndicesArray& getIndices() const final;
  MAR_NO_DISCARD const FTransformsArray& getTransforms() const final;

  void updateVertices(const Entity& entity) final;
  void updateIndices(const Entity& entity) final;
  void updateTransform(const Entity& entity) final;

};
```

Listing 14: Przybliżona deklaracja klasy *FMeshBatch*.

```
// Renderowanie pojedynczo modeli
FMeshExternal mesh1;
mesh1.load("resources/monkey.obj");
FMeshCube mesh2;
FMeshPyramid mesh3;

while(true) {
    renderCommand.draw(mesh1);
    renderCommand.draw(mesh2);
    renderCommand.draw(mesh3);
}
```

```
// Renderowanie grupy
FMeshExternal mesh1;
mesh1.load("resources/monkey.obj");
FMeshCube mesh2;
FMeshPyramid mesh3;

FMeshBatchStaticColor batch;
batch.submitToBatch(mesh1);
batch.submitToBatch(mesh2);
batch.submitToBatch(mesh3);

while (true) {
    renderCommand.draw(batch);
}
```

Listing 15: Porównanie kodu źródłowego z renderowaniem pojedynczego obiektu (trzeba wywoływać rysowanie z osobna na każdą siatkę, niekiedy dokładnie tą samą w innej pozycji) oraz grupy modeli (jednorazowe rysowanie wielu siatek równolegle).

3.7.6 Potok graficzny

Przyszedł czas na implementacją potoku graficznego. Jak wiadomo z poprzednich rozdziałów potok graficzny odpowiada za generowanie sceny trójwymiarowej na ekrany dwuwymiarowe. Taka definicja jest jednak abstrakcyjna, ponieważ renderowanie opiera się na

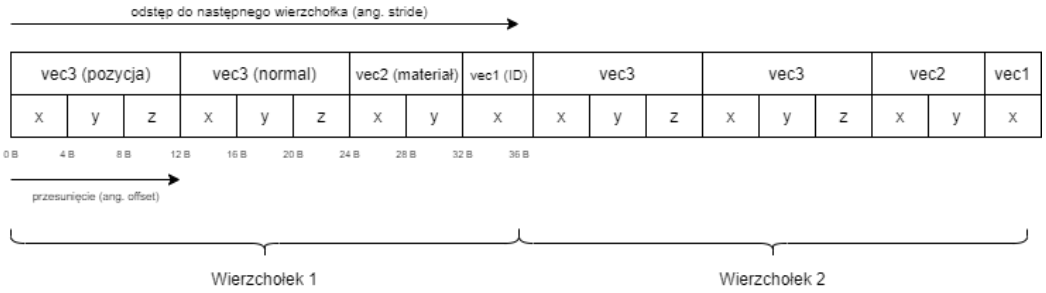
przejściu przez wszystkie etapy potoku. W podrozdziale 2.2.6 dot. potoku zostało powiedziane, że istnieją cztery zdefiniowane fazy gdzie tylko jedna uruchamiana jest na procesorze CPU (faza aplikacji), pozostałe na jednostce graficznej poprzez shadery omówione w podrozdziale 3.7.4. Implementacje potoku graficznego w silniku znajdują się w fazie aplikacji i odpowiada za to klasa *FPipeline* oraz jej pochodne wersje - zajmujące się przechowywaniem informacji o wszystkich zależnościach jakie są potrzebne do poprawnego uruchomienia potoku. Wymaganiami są:

- magazyny:
    1. *FBufferStorage* - przechowuje bufory w których przechowywane są wszystkie informacje dot. siatek modeli, ich pozycji w przestrzeni bądź skali obiektów oraz dane kamery (czyli obliczona macierz Model-View-Projection MVP),
    2. *FShadersStorage* - magazynuje skompilowane programy GLSL, czyli załadowane shadery,
    3. *FMaterialStorage* - zbiera wszystkie wczytane tekstury z plików,
- identyfikatory składowych:
    1. buforów wierzchołków *VertexBuffer*, indeksów *IndexBuffer*, transformacji obiektów *TransformSSBO*, kamery *CameraSSBO*, światła punktowego *PointLightSSBO*, materiałów *MaterialSSBO*,
    2. wybranego zestawu shaderów *FShaders*.

Pozyskane dane posłużą klasie *FPipeline* podczas generowania obrazu. Jednak to nie wszystko. Biblioteki graficzne wymagają zdefiniowania z góry w jaki sposób konkretne bufory mają być interpretowane. Czyli w jaki sposób procesor karty graficznej GPU powinien czytać wszystkie przekazane dane z buforu wierzchołków. Czemu zatem tak ważny jest *Vertex Buffer*? Otóż wierzchołek w silnikach gry nie jest zdefiniowany tylko poprzez posiadanie trójki skalarów określających jego pozycję w przestrzeni $\mathbb{R}^3$. Wierzchołek może być strukturą znacznie bardziej rozbudowaną i taką też jest w silniku *MAREngine*. Każdy punkt oczywiście posiada informacje dot. jego pozycji w formie trzech skalarów (klasa *vec3*), oprócz tego wymagane są jeszcze 3 dane. Dlatego też na ilustracji 17 zamieszczono jak wygląda pełny wierzchołek oraz poniżej zdefiniowano co za co odpowiada:

- pozycja wierzchołka (ang. *Vertex Position*),
- wektor normalny (ang. *Normal Vector*) - wektor 3D prostopadły do wybranej płaszczyzny (w tym przypadku do trójkąta do którego należy wierzchołek),
- koordynaty tekstur (ang. *Texture Coordinates*) - wektor 2D mówiący w jaki sposób tekstura powinna zostać nałożona na obiekt,
- identyfikator obiektu w grupie *Batch Index* - indeks pozwalający wyciągnąć przypisaną macierz transformacji oraz materiał do obiektu w grupie.

W OpenGL definiowanie wierzchołka opiera się na obiekcie VAO - *Vertex Array Object*. VAO wie w jaki sposób odczytać format podanych buforów, jednak najpierw należy te dane przekazać. Te operacje odbywają się w klasach *FPipelineOpenGL*. Na rysunku



Rysunek 17: Atrybuty wierzchołka w silniku *MAREngine*.

17 celowo zaznaczono takie elementy jak odstęp pomiędzy wierzchołkami (ang. *stride*), przesunięcie (ang. *offset*) bądź rozmiar określonych wartości w bajtach, ponieważ te dane wymagane są od rozczytywania wierzchołków na karcie graficznej. Listing 17 przedstawia tworzenie VAO oraz przekazanie wszystkich parametrów. Utworzone zostały klasy pomocnicze *EInputType*, *FVertexInputVariableInfo* oraz *FVertexInputDescription* pomagające w łatwiejszy i zautomatyzowany sposób okiełznać problem.

Wykorzystując wszystkie informacje zamieszczone powyżej zaimplementowano klasy *FPipeline*, które mogą zostać utworzone przy pomocy dwóch metod - *pass()* oraz *create()* - gdzie reprezentacje pierwszej funkcji muszą być uruchomione najpierw celem przekazania wszystkich zależności, a następnie wywołując metodę *create()* tworzymy obiekt potoku graficznego. Tym samym przed wywołaniem operacji generowania na ekranie programista musi tylko zadeklarować wykorzystanie określonego potoku graficznego i nic więcej, implementacja jest schowana i nie musi być znana. Na listingu 16 zamieszczono przybliżoną implementację klasy *FPipeline*.

```
1  class FPipeline : public IPipeline {
2  public:
3
4    void create() final;
5    void close() final;
6    void bind() const final;
7
8    void passBufferStorage(FBufferStorage* pBufferStorage) final;
9    void passShadersStorage(FShadersStorage* pShadersStorage) final;
10   void passMaterialStorage(FMaterialStorage* pMaterialStorage) final;
11   void passVertexBuffer(int32 i) final;
12   void passIndexBuffer(int32 i) final;
13   void passTransformSSBO(int32 i) final;
14   void passCameraSSBO(int32 i) final;
15   void passPointLightSSBO(int32 i) final;
16   void passShaderPipeline(int32 i) final;
17
18 };
```

Listing 16: Przybliżona deklaracja klasy *FPipeline*.

```cpp
enum class EInputType { // typy parametrów wejściowych do shaderów
  NONE, FLOAT, INT, VEC4, VEC3, VEC2, MAT4
};

struct FVertexInputVariableInfo { // informacje dot. pojedynczego parametru wierzchołka
  EInputType inputType{ EInputType::NONE }; // typ danych
  uint32 location{ 0 };                     // indeks atrybutu
  uint32 offset{ 0 };                       // offset parametru
};

struct FVertexInputDescription { // informacje dot. opisu całego wierzchołka
  std::vector<FVertexInputVariableInfo> inputVariables; // tablica parametrów wejściowych
  uint32 binding{ 0 };                                  // wiązanie
  uint32 stride{ 0 };                                   // rozmiar jednego wierzchołka
};

GLuint vao;
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);

uint32 i = 0;
for(const auto& inputVar : inputDesc.inputVariables) {
  glVertexAttribPointer(i,                                        // indeks atrybutu
                        sizeof(inputVar.inputType),               // rozmiar typu danych
                        inputVar.inputType,                       // typ danych
                        GL_FALSE,                                 // czy parametr jest znormalizowany
                        inputDescription.stride,                  // rozmiar wierzchołka
                        (const void*)inputVariable.offset) ); // offset do następnego atrybutu
  glEnableVertexAttribArray(i);
  i++;
}
```

Listing 17: Generowanie obiektu VAO wraz z uzupełniem danych dot. wierzchołków w bibliotece OpenGL.

3.7.7 Przechowywanie wyrenderowanej sceny

Zależności pozwalające na generowanie obrazu napisano oraz zaprezentowano w poprzednich podrozdziałach. Następnie należy skupić się na sposobie przetrzymywania wyrenderowanej sceny 3D, którą później będzie można wyświetlić w graficznym interfejsie użytkownika. Poprawną drogą będzie wykorzystanie tzw. buforu ramek (ang. *Framebuffer*). W grafice komputerowej takie obiekty nazwano FBO - *Framebuffer Object* - mające stanowić część kontekstu biblioteki graficznej i zazwyczaj powinny reprezentować okno bądź urządzenie gotowe do wyświetlenia obrazu.

Bufor ramek specjalnie został rozdzielony od poprzednich buforów zamieszczonych w podrozdziale 3.7.3 ze względu na fakt, że ten przyjmuje zdecydowanie inne właściwości oraz operuje się nim przy pomocy innych funkcji OpenGL. W tym celu utworzono klasę *FFramebuffer*, która ma posłużyć jako bufor wyrenderowanych scen. To właśnie do instancji klasy *FFramebuffer* będą generowane wszystkie obrazy. Bufor ramek z technicznego punktu widzenia jest zbiorem poszczególnych punktów powiązań, gdzie w silniku *MAREngine* zaimplementowano dwa najważniejsze:

- *GL_COLOR_ATTACHMENT0* - w tym punkcie w formie tekstury zapisywany jest kolor obrazu,

- *GL_DEPTH_ATTACHMENT* - tutaj zawarto informacje dot. głębokości obiektów zawartych w scenie, przy pomocy tego punktu powiązań dodaje się uczucie głębi.

Oprócz klasy *FFramebuffer* dodano strukturę *FFramebufferSpecification* pozwalającą na utworzenie określonego buforu ramek. Programista jest w stanie zarządzać renderowanym obrazem w czasie rzeczywistym poprzez aktualizowanie specyfikacji. Na listingu 18 zamieszczono deklarację *FFramebufferOpenGL*, standardowo dodano metody mogące utworzyć instancje jak i ukończyć ich pracę poprzez metody *create()* oraz *destroy()*. Dodano też jak na OpenGL przystało funkcje *bind()* i *unbind()*, które muszą być wywołane podczas operacji rysowania do buforu. W ten sposób biblioteka aktualizuje sobie stan buforów ramek i wie do którego aktualnie powinien zostać wygenerowany obraz. Warto wspomnieć o metodzie *clear()*, która powinna być wywoływana po każdym wyświetleniu ramki celem wyczyszczenia buforu, oraz *resize()* zmieniającej rozdzielczość renderowanego obrazu.

```
1  class FFramebufferOpenGL : public FFramebuffer {
2  public:
3
4    void create(const FFramebufferSpecification& specs) final;
5    void destroy() final;
6
7    void bind() const final;
8    void unbind() const final;
9    void clear() const final;
10   void resize(uint32 width, uint32 height) final;
11
12 };
```

Listing 18: Deklaracja klasy *FFramebufferOpenGL*.

3.7.8 Obserwacja wygenerowanych obiektów

Przyglądanie się wyrenderowanej scenie nie byłoby możliwe bez instancji kamery. W silnikach graficznych kamera jest okiem, które określa sposób w jaki użytkownik obserwuje scenerię. Zgodnie z rozdziałami 2.2.3 oraz 2.2.4 dotyczącymi macierzy widoku oraz projekcji, zaimplementowano klasę *FRenderCamera* odpowiedzialną za obliczenie wszystkich wymaganych macierzy oraz zwrócenie rezultatu w postaci *Model-View-Projection Matrix* - macierzy MVP. Zaimplementowano metody takie jak *calculatePerspective()* oraz *calculateOrthographic()* odpowiadające za wyliczenie macierzy projekcji w wybrany sposób - 2D bądź 3D. Dodano wyliczenie macierzy widoku poprzez metodę *calculateView()* oraz pełne przetwarzanie macierzy MVP po wykonanych wcześniej wyliczeniach przy pomocy metody *recalculateMVP()*. Nie zapomniano o systemie ECS (patrz 3.6), gdzie zaproponowano komponenty *CTransform* i *CCamera*. Celem utworzenia kamery gry dodano funkcję przyjmującą za argumenty podane dwa komponenty *calculateCameraTransforms(CTransform, CCamera)*, dzięki czemu w ręce programisty zostaje oddane zarządzanie kamerą podczas gry. Na listingu 19 zaprezentowano deklarację klasy *FRenderCamera* ze wszystkimi opisanymi wyżej funkcjami.

```
class FRenderCamera {
public:

  void calculatePerspective(float zoom, float aspectRatio, float nearPlane, float farPlane);
  void calculateOrthographic(float left, float right, float top, float bottom,
                             float nearPlane, float farPlane);
  void calculateView(vec3 position, vec3 lookAt, vec3 up);
  void calculateModel(vec3 arg);
  void recalculateMVP();

  void calculateCameraTransforms(const CTransform& transform, const CCamera& camera);

  MAR_NO_DISCARD const mat4& getProjection() const;
  MAR_NO_DISCARD const mat4& getView() const;
  MAR_NO_DISCARD const mat4& getModel() const;
  MAR_NO_DISCARD const mat4& getMVP() const;
  MAR_NO_DISCARD const vec3& getPosition() const;

private:

  mat4 m_model;
  mat4 m_view;
  mat4 m_projection;
  mat4 m_mvp;
  vec3 m_position;

};
```

Listing 19: Przybliżona deklaracja klasy *FRenderCamera*.

3.7.9 Działanie systemu graficznego

Zaprezentowano wszystkie potrzebne klasy, które w mniejszym lub większym stopniu przyczyniają się do renderowania obiektów w czasie rzeczywistym w przestrzeni $\mathbb{R}^3$. Ostatnim etapem jest wykorzystanie napisanego kodu i zebranie go w jedną spójną całość.

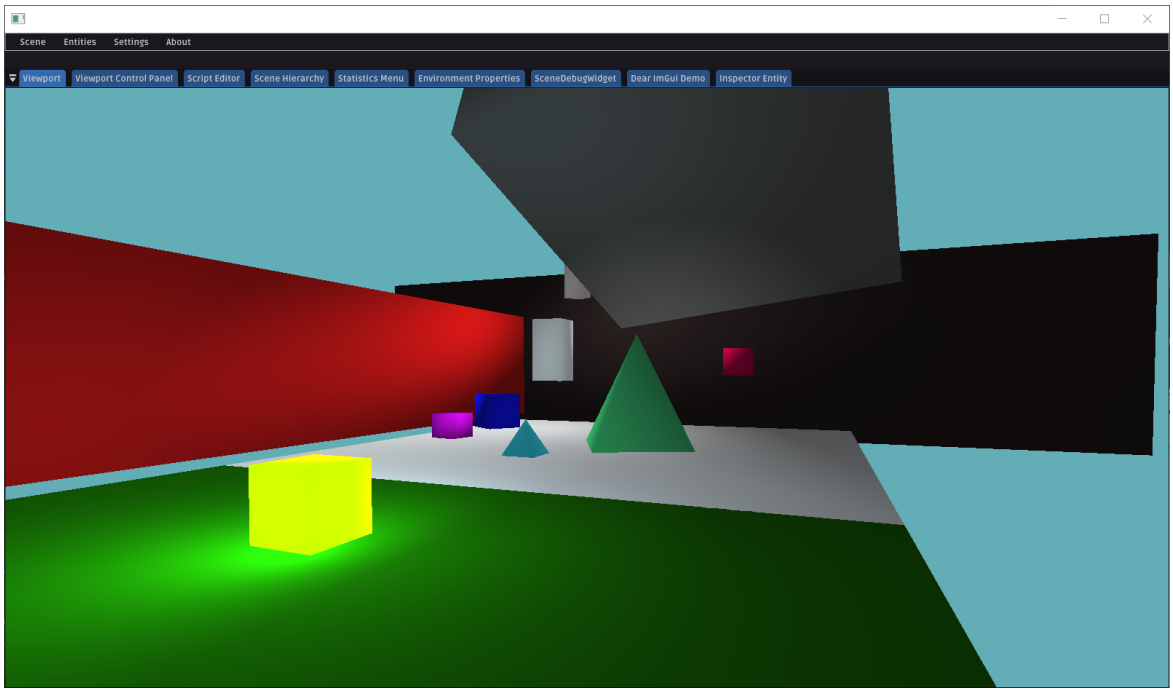
Na początek utworzono instancje klas zarządzających oraz zainicjalizowano stan silnika. W przygotowanie silnika do pracy wchodzi utworzenie kontekstu biblioteki OpenGL, utworzenie wszystkich buforów oraz ich alokacja poprzez załadowane wszystkie wierzchołki i przypisane do nich indeksy, załadowanie wszystkich tekstur, przekazanie informacji o siatkach i materiałach do menadżera grup modeli i utworzenie buforu ramek do którego będzie renderowana scena. Gdy wszystkie procesy będą gotowe, dodano pętlę podczas której silnik będzie operował na wszystkich utworzonych składowych. Tutaj najważniejsze będą iteracje, gdzie należy aktualizować stan wszystkich jednostek oraz przetwarzać cały potok graficzny. Na początek trzeba zadbać o wyczyszczenie buforu ramek i przygotowanie go na kolejne obrazy. Następnie dodano pętlę, która iteruje przez wszystkie grupy siatek oraz rysuje ich zawartość na ekranie. Na listingu 20 zaprezentowano całą procedurę pomijając moduły, które nie należą do systemu renderującego.

```
// Startowanie okna i pozostałych bibliotek...

auto renderContext{ createRenderContextType<ERenderContextType::OPENGL>() };
auto renderCommands{ createRenderCommandsType<ERenderContextType::OPENGL>() };

renderContext.create(&window);
renderManager.create(&renderContext);
materialManager.create(&renderContext);
batchManager.create(&renderManager, meshManager.getStorage(), materialManager.getStorage());
renderCommands.create(&renderStatistics);

FPipelineStorage* pPipelineStorage{ renderContext.getPipelineStorage() };

FFramebuffer* pFramebufferViewport{ renderManager.getViewportFramebuffer() };
pFramebufferViewport->setClearColor(pScene->getBackground());

// Startowanie GUI i innych modułów silnika...

while(/* Sprawdzenie, czy należy zamknąć silnik... */) {
    renderStatistics.reset();
    pFramebufferViewport->clear();

    for(int32 i = 0; i < (int32)pPipelineStorage->getCountColorMesh(); i++) {
        renderCommands.draw(pFramebufferViewport, pPipelineStorage->getColorMesh(i));
    }

    // Aktualizacje pozostałych modułów...
}

// Zamykanie wszystkich instancji i silnika...
```

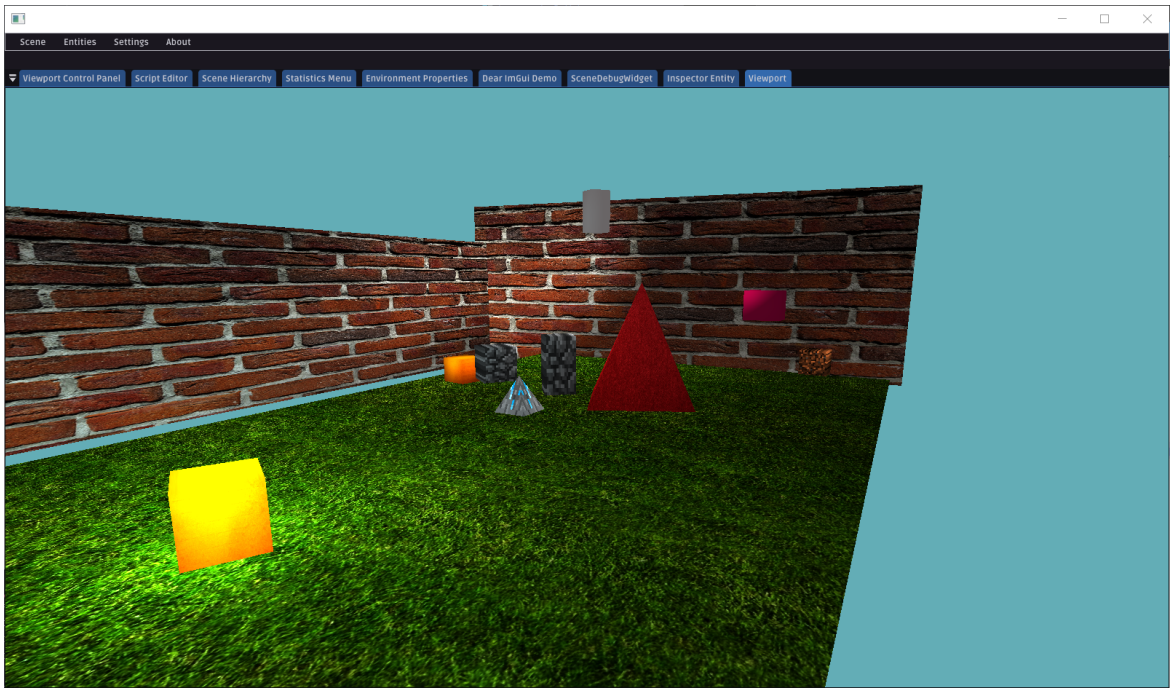
Listing 20: Przybliżona funkcjonalność modułu renderującego.

Przetestowano działanie systemu graficznego na kilku płaszczyznach celem sprawdzenia zaimplementowanej funkcjonalności, gdzie moduł renderuje kilka przykładowych scen. Efekty testów zostały przedstawione na ilustracjach i polegały na utworzeniu:

- sceny posiadającej podstawowe obiekty z przypisanymi tylko kolorami (rys. 18),
- sceny z załadowanymi teksturami z plików o rozszerzeniu jpg (rys. 19),
- sceny z obiektami wczytanymi z plików zewnętrznych o rozszerzeniu obj i słabym oświetleniem (rys. 20),
- oraz sceny z dobrym oświetleniem punktowym (rys. 21).



Rysunek 18: Wyrenderowana scena kolorowych obiektów w silniku *MAREngine*.



Rysunek 19: Wyrenderowana scena obiektów z przypisanymi teksturami w silniku *MAREngine*.

Rysunek 20: Wyrenderowana scena obiektów z przypisanymi kolorami i teksturami przy słabym oświetleniu w silniku *MAREngine*.

Rysunek 21: Wyrenderowana scena obiektów z przypisanymi kolorami i teksturami z silnym oświetleniem w silniku *MAREngine*.

3.8 Interfejs graficzny użytkownika

3.8.1 Abstrakcja modułu GUI

Podczas definiowania hierarchii klas interfejsu graficznego użytkownika należy skupić się na dobrze zdefiniowanej abstrakcji, która umożliwi podmianę biblioteki GUI na inną, pozwoli na proste dodawanie nowych okienek i ich pełną niezależność. Tym samym spodziewanym efektem jest czysty i przejrzysty kod.

Na początek zdefiniowano klasę abstrakcyjną *IWidget*, która ma być wzorem dla każdego zaimplementowanego widżetu przy pomocy silnika. Na listingu 21 pokazano szablon każdego widżetu. Specjalnie nie dodano metody *create()*, ponieważ każdy widżet może potrzebować swoich własnych argumentów umożliwiających poprawne działanie.

```
1  class IWidget {
2  public:
3
4    virtual void destroy() { }
5
6    virtual void beginFrame() { }
7    virtual void updateFrame() { }
8    virtual void endFrame() { }
9
10   virtual void onCreation() const { }
11   virtual void onDestruction() const { }
12
13   virtual void onBeginFrame() const { }
14   virtual void onUpdateFrame() const { }
15   virtual void onEndFrame() const { }
16
17   virtual bool isHovered() const { return false; }
18   virtual bool isFocused() const { return false; }
19
20   virtual void onKeyboardButtonPressed() const { }
21   virtual void onMouseButtonPressed() const { }
22
23   virtual void onDragDetected() const { }
24   virtual void onDrop() const { }
25
26 };
```

Listing 21: Definicja abstrakcyjnej klasy *IWidget*.

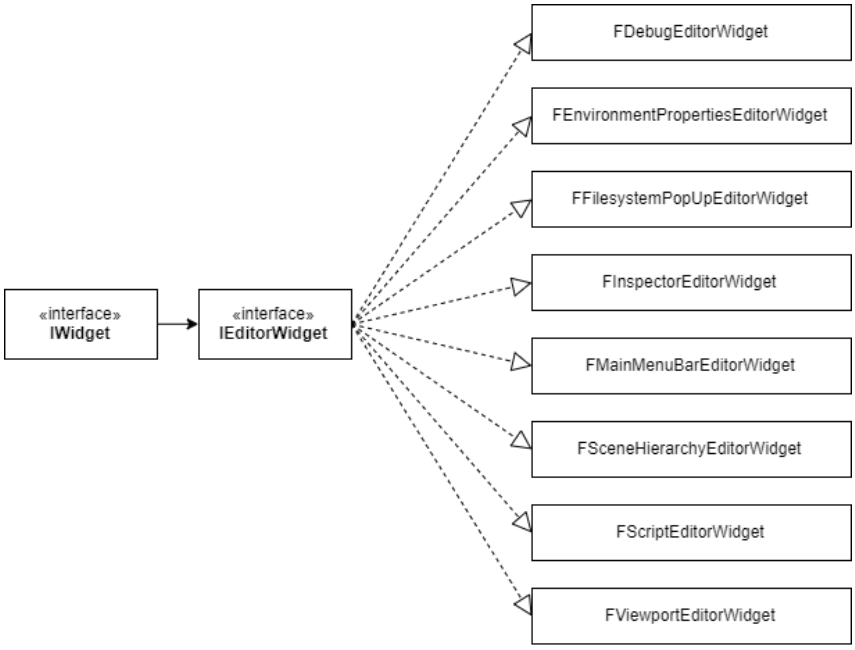
Kolejno dodano klasę *IEditorWidget* dziedziczącą po *IWidget* z tylko jedną metodą *create(FServiceLocatorEditor*)*. Niektóre widżety są od siebie zależne, wymagają informacji o stanie pozostałych okien tym samym aktualizując swój status bądź są okienkami wyskakującymi (ang. *PopUp Widget*). W tym przypadku rozważano nad dwoma rozwiązaniami - pierwsze, gdzie zostałby wykorzystany wzorzec architektoniczny Wstrzykiwanie Zależności (ang. *Dependency Injection*) oraz drugi, gdzie każdy widżet przyjąłby własności singletona i byłby dostępny w każdym miejscu w kodzie poprzez statyczną metodę. Szybko zrezygnowano z drugiej opcji, tj. singletona dostępnego wszędzie, ponieważ w tej sytuacji programista znając tylko i wyłącznie deklarację klasy nie jest w stanie określić jej wymagań. Bardzo często może zdarzyć się sytuacja, gdy uruchomiona instancja klasy kończy swoje działanie z błędem krytycznym przez brak inicjalizacji innego singletona

(czyli innego zależnego widżetu). Rozwiązanie pierwsze, wzorzec *Dependency Injection*, pod tym względem jest lepszym sposobem proponującym "wstrzykiwanie" wszystkich zależności do pozostałych widżetów. Podczas implementacji tego wzorca konstruktory i metody *create()* były całkiem potężnymi komendami z uwagi na fakt, że lista argumentów była zbyt długa co czyniło ją zdecydowanie niezrozumiałą. Robert C. Martin (ps. Uncle Bob, autor książki *Clean Code* [24]) uważa, że akceptowalną liczbą argumentów jest trzy. Dlatego też próbowano znaleźć lepsze rozwiązanie problemu. Zaproponowano zatem wykorzystanie wzorca *Service Locator* z implementacją klasy *FServiceLocatorEditor*, która powinna posiadać informacje dot. wszystkich zainicjalizowanych widżetów. W ten sposób klasa wzorca jest odpowiedzialna za dbanie o stan posiadanych obiektów, a wszystkie widżety traktują wydobyte serwisy jako gotowe do użytku.

Tak więc wszystkie okna oraz funkcje GUI podczas startu są rejestrowane przez klasę *FServiceLocatorEditor*, która później pozwala zlokalizować instancje i umożliwić poprawne działanie interfejsu graficznego użytkownika.

W następnym etapie sprecyzowano okna, które z pewnością będą wymagane podczas edycji sceny w czasie rzeczywistym. Sprawdzono widżety wchodzące w skład innych silników graficznych oraz po rozmowach z deweloperami gier komputerowych zapisano faktyczne potrzeby interfejsu graficznego. Na rysunku 22 zaprezentowano pełną hierarchię klas GUI. W skład hierarchii wchodzą:

- *FDebugEditorWidget* - zawiera wszystkie informacje potrzebne do debugowania jednostek i komponentów w czasie rzeczywistym,
- *FEnvironmentPropertiesEditorWidget* - określa właściwości środowiska otaczającego scenę,
- *FFilesystemPopUpEditorWidget* - wyskakujące okno (*Pop Up*) potrzebne podczas ładowania plików zewnętrznych (modeli, tekstur, skryptów) oraz operacji na plikach sceny projektu,
- *FInspectorEditorWidget* - widżet pełniący rolę inspektora wybranej jednostki w *FSceneHierarchyEditorWidget*, panel zarządzający wszystkimi komponentami przypisanymi do jednostki,
- *FMainMenuBarEditorWidget* - główny pasek u samej góry okna silnika pozwalający na dostosowanie ustawień okna, załadowanie wybranej sceny bądź wyjście z programu,
- *FSceneHierarchyEditorWidget* - lista wszystkich jednostek znajdujących się w załadowanej scenie, pozwala dodać nowe jednostki bądź usunąć już istniejące,
- *FScriptEditorWidget* - okno spersonalizowane do edycji kodu źródłowego Pythona w którym użytkownik może pracować nad skryptami opisującymi logikę gry komputerowej,
- *FViewportEditorWidget* - najważniejsze okno - poglądowe (ang. *Viewport*) - gdzie wyświetla się scena na której użytkownik pracuje, scena renderowana jest w czasie rzeczywistym i pozwala na modyfikację obiektów wykorzystując gadżet zwany chwytakiem (ang. *Guizmo*).



Rysunek 22: Hierarchia klas interfejsu graficznego użytkownika w silniku.

3.8.2 Wybór biblioteki

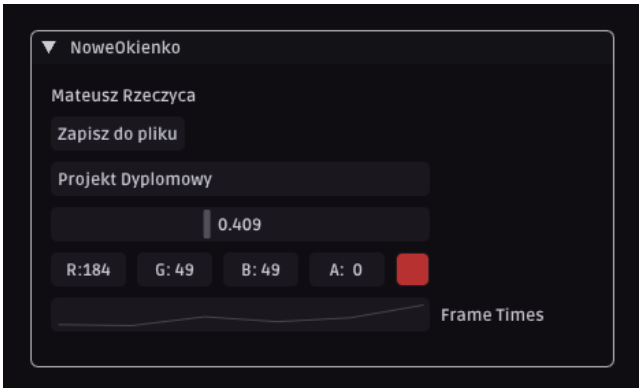
Bardzo ważną decyzją podczas implementacji modułu edytora był wybór biblioteki interfejsu graficznego użytkownika. W tym przypadku nie szukano interfejsu z bezpośrednim dostępem do karty graficznej (tak jak OpenGL bądź Vulkan), rozglądano się za systemem proponującym gotowe rozwiązania z gotowymi klasami takimi jak *Button* lub *Checkbox*. Przekonującymi bibliotekami są Qt oraz Dear ImGui. Rozważono pierwszą propozycję. Sugerowanym narzędziem do pracy z modułami Qt jest ich autorski program Qt Creator posiadający specjalizowany sposób budowania projektów, domyślnie w Qt wykorzystywany jest QMake. W innym przypadku wymagana jest personalizacja systemu do kompilacji projektu. Ponadto napisanie autorskiego modułu renderującego spotka się z drobnymi problemami, ponieważ proponowanym i najpewniejszym sposobem napisania systemu graficznego jest wykorzystanie API wykreowanego przez Qt. Oprócz tego lista nowych gier bądź silników wydanych w poprzednich kilku latach z tym interfejsem jest całkiem mała co prowadzi do niskiego zainteresowania przez programistów silników [25] [26]. Ogromnym benefitem Qt z pewnością jest obszerna dokumentacja, mało kto posiada ją tak zorganizowaną. Z kolei analizując drugą propozycję - Dear ImGui - znaleziono więcej plusów. Biblioteka jest kompletnie niezależna od systemu renderującego, do rozpoczęcia pracy wymaga przekazania kontekstu modułu graficznego (natywnego kontekstu OpenGL lub innego API). Integracja ImGui opiera się na dodaniu kilku plików źródłowych i skompilowaniem ich razem z projektem co czyni ją bardzo elastyczną biblioteką. Ponadto wsparcie jest ogromne, interfejs ma sporą listę sponsorów w którą wchodzą znane marki z branży produkcji gier wideo [27] oraz wykorzystywany jest w wielu nowoczesnych produktach (takich jak chociażby Cyberpunk 2077 studia CD Project Red) [28]. Ostatnią zaletą jest uznanie modułu wśród społeczności branży gier wideo, na wielu forach można znaleźć personalizowane silniki graficzne wykorzystujące Dear ImGui. Dlatego na podstawie powyższych zalet wybrano bibliotekę Dear ImGui.

3.8.3 Opis zaimplementowanych okienek

Ostatnim etapem tworzenia GUI była implementacja wszystkich okienek omówionych w podrozdziale 3.8.1. Obsługa Dear ImGui jest intuicyjna, co zaprezentowano na listingu 22 oraz ilustracji 23, przedstawiających kolejno kod źródłowy oraz jego rezultaty. Zaprezentowano kilka funkcjonalności biblioteki, które z pewnością zostaną wykorzystane podczas implementacji interfejsu graficznego silnika.

```
static char buffer[256];
static float value{ 0.f };
static float v4[4]{ 0.f, 0.f, 0.f, 0.f };
constexpr std::array<float, 6> floatValues{ 0.2f, 0.1f, 1.0f, 0.5f, 0.9f, 2.2f };

ImGui::Begin("NoweOkienko");            // utworzenie okna
ImGui::Text("Mateusz Rzeczyca");        // wyświetlanie tekstu
if (ImGui::Button("Zapisz do pliku")) { // klikalny przycisk
    // Wywołanie funkcji zapisujących do pliku
}
// panel wejściowy dla łańcucha znaków
ImGui::InputText("##text_input1", buffer, sizeof(buffer)/sizeof(buffer[0]));
// suwak umożliwiający zmianę wartości zmiennej
ImGui::SliderFloat("##float_slider1", &value, 0.f, 1.f);
// edytor wartości tablicy czterech zmiennych typu float (będących kolorem RGBA)
ImGui::ColorEdit4("##color_edit", v4);
// rysowanie wykresu
ImGui::PlotLines("Frame Times", floatValues.data(), floatValues.size());
ImGui::End();
```

Listing 22: Kod źródłowy przykładowego okienka napisany wykorzystując bibliotekę Dear ImGui.

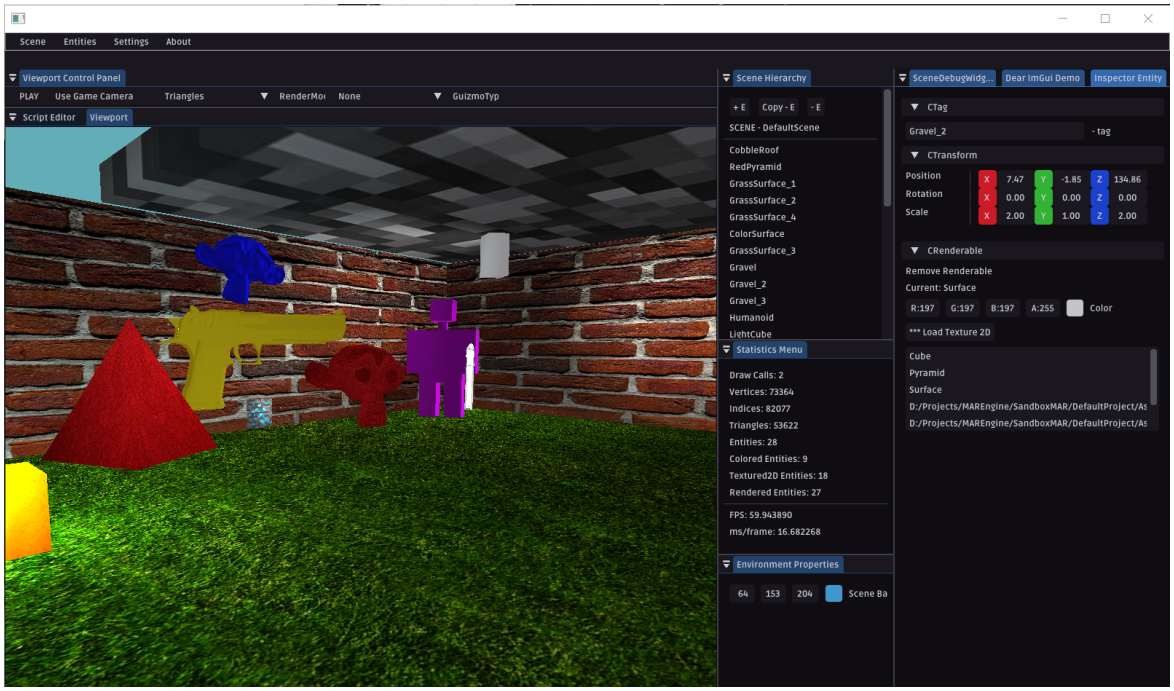


Rysunek 23: Przykładowe okienko wyrenderowane przy pomocy Dear ImGui.

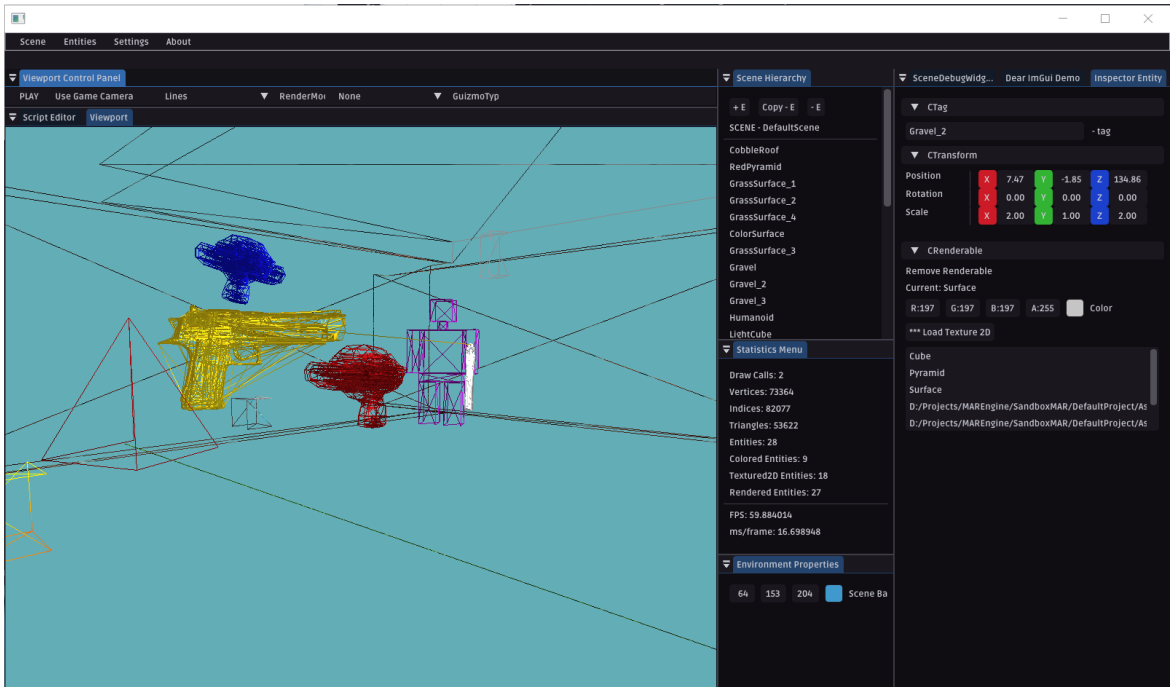
Utworzono zatem specjalne klasy pochodne z dopiskiem nazwy biblioteki (przykładowo *FInspectorWidgetImGui*) implementujące określone funkcjonalności klas bazowych. Podczas doboru interfejsu wzorowano się na innych istniejących edytorach graficznych celem dostosowania się do znanego już interfejsu wielu programistom gier komputerowych. Skonfigurowano czarny motyw, ponieważ zaobserwowano jego wykorzystanie przez znaczącą część deweloperów.

Na ilustracjach 24, 25, 26, 27, 28, 29 widocznych poniżej zaprezentowano kolejno widżety:

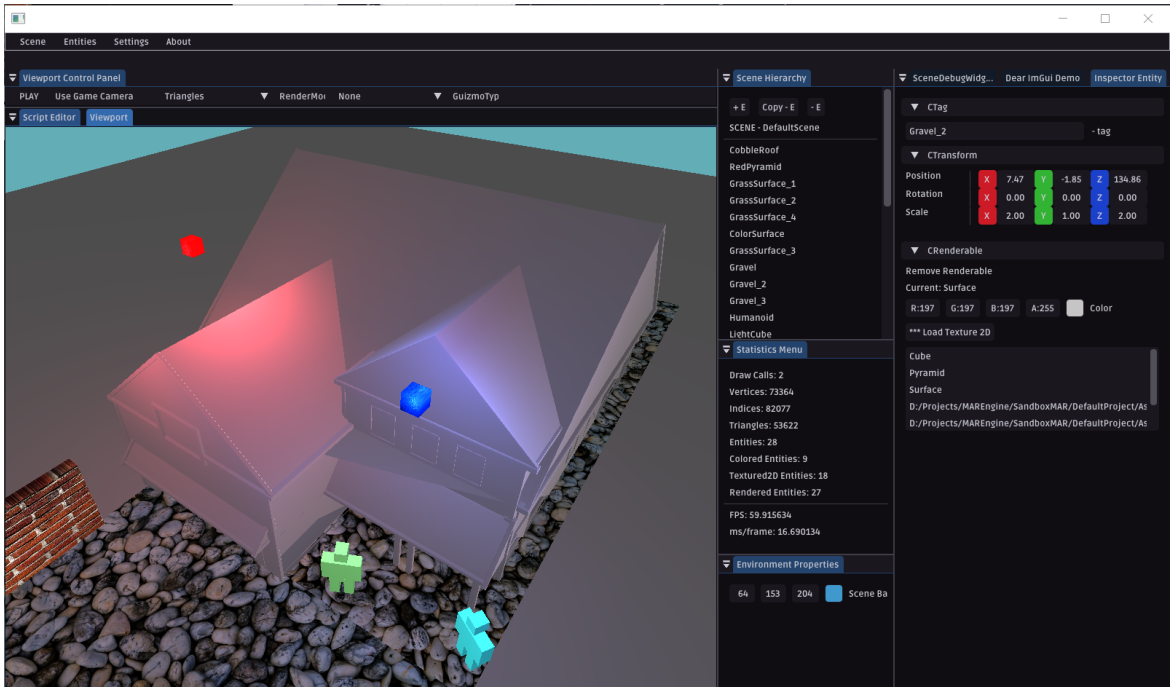
- Main Menu Bar (*FMainMenuBarEditorImGui*) - górny panel główny na którym znajdują się zakładki *Scene*, *Entities*, *Settings* oraz *About* pozwalające na operacje na scenach (zapisywanie i ładowanie), funkcje na jednostkach, personalizowanie ustawień silnika oraz informacje dot. silnika jak i jego autora, widoczny na wszystkich rysunkach,
- Viewport Control Panel (*FViewportEditorImGui*) - panel w którym znajdują się aktualnie cztery opcje - przetestowania gry po wciśnięciu przycisku "Play", sprawdzenia kamery przypisanej do wybranej jednostki (będącej kamerą gry), zmiany trybu renderowania (do wyboru są trójkąty oraz krawędzie) i zmiany typu chwytaka (możliwymi opcjami są translacja, rotacja, skala), widoczny na wszystkich ilustracjach,
- Viewport (*FViewportEditorImGui*) - okno w którym znajduje się wyrenderowana scena, widoczny na czterech pierwszych rysunkach,
- Scene Hierarchy (*FSceneHierarchyEditorImGui*) - widżet listujący wszystkie istniejące jednostki w scenie, mający możliwość dodania nowej lub usunięcia jednostki, widoczny na wszystkich ilustracjach,
- Statistics Menu (*FDebugEditorImGui*) - okno ze statystykami sceny takimi jak liczba wywołań rysowania, liczby wierzchołków, ich indeksów, trójkątów, jednostek oddzielnie pokolorowanych i z przypisanymi teksturami, widoczny na wszystkich rysunkach,
- Environment Properties (*FEnvironmentPropertiesEditorImGui*) - posiada możliwość zmiany koloru tła otaczającego scenę, widoczny na wszystkich ilustracjach, widoczny na wszystkich rysunkach,
- Inspector Entity (*FInspectorEditorImGui*) - widżet pozwalający na modyfikację przypisanych komponentów do wybranej jednostki, na ilustracjach widoczne są komponenty *CTag*, *CTransform* oraz *CRenderable*, rozwijając nagłówek otwiera się możliwość zmian określonych parametrów wchodzących w skład komponentu, widoczny na czterech pierwszych rysunkach,
- Open File Pop Up (*FFilesystemPopUpEditorImGui*) - wyskakujące okienko otwierające pliki o spersonalizowanym rozszerzeniu, na ilustracji 28 widoczne wyszukiwanie tekstur (czyli plików o formacie .jpg),
- Script Editor (*FScriptEditorImGui*) - edytor skryptów napisanych w Pythonie przypisanych do jednostek, ilustracja 29,
- SceneDebugWidget (*FDebugEditorImGui*) - okno ze wszystkimi szczegółowymi informacjami dot. projektu, sceny i jednostek, ilustracja 29.



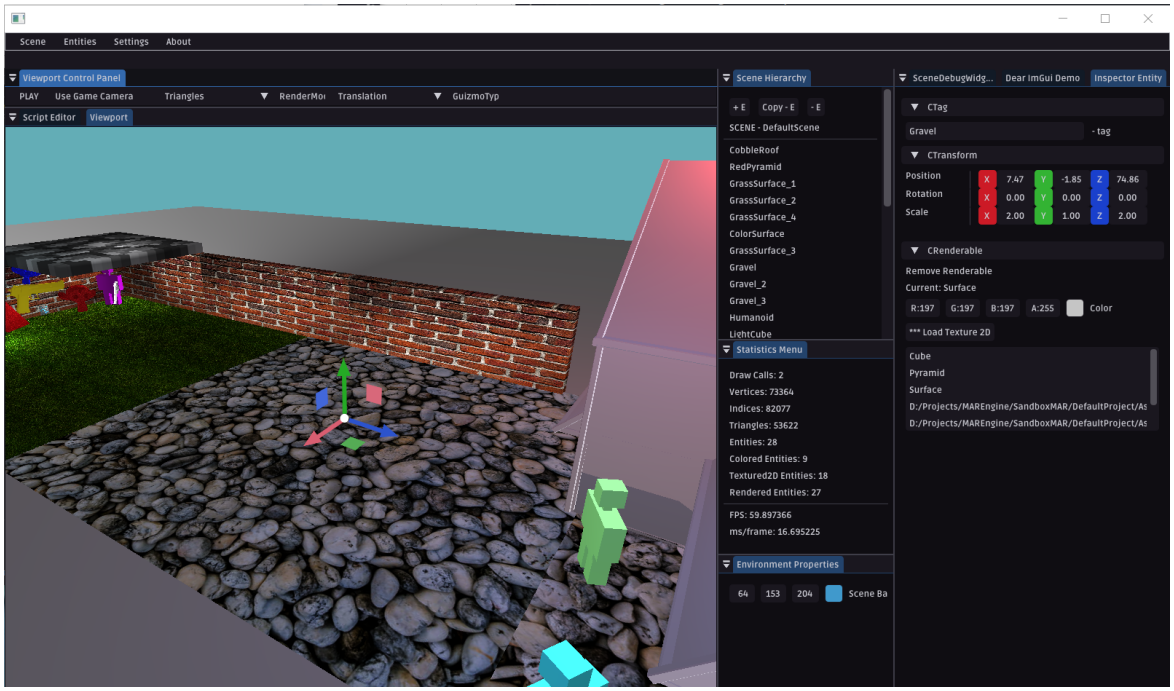
Rysunek 24: Scena wyrenderowana z trójkątów z edytorem graficznym.



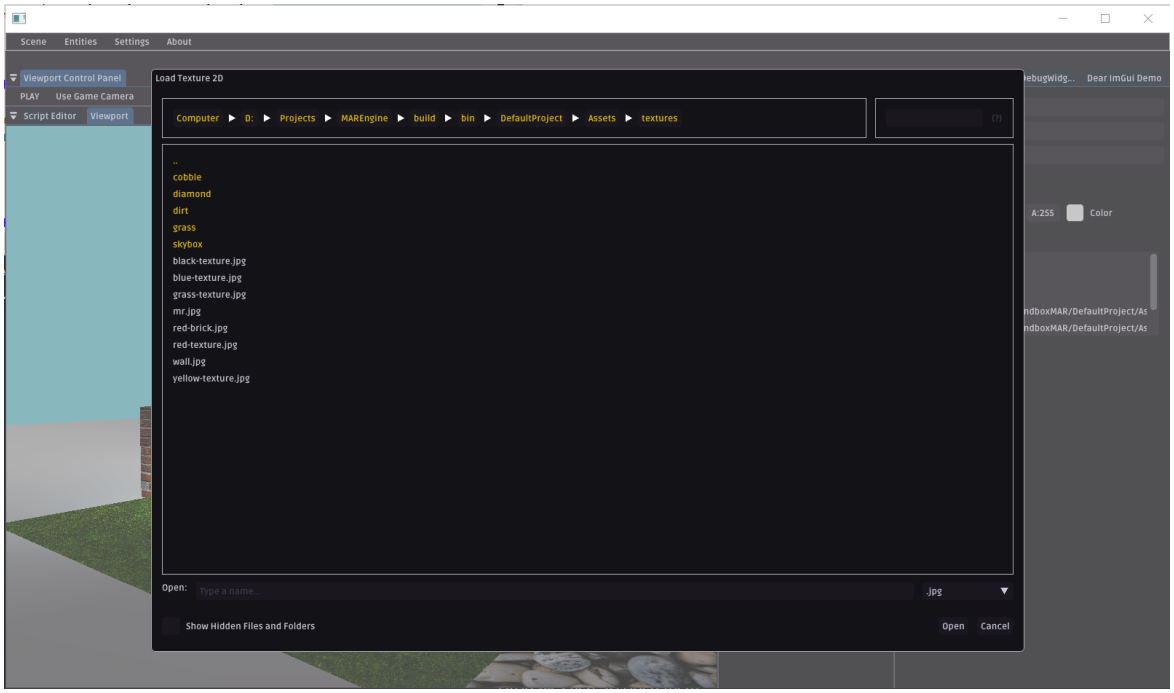
Rysunek 25: Scena wyrenderowana z krawędzi z edytorem graficznym.



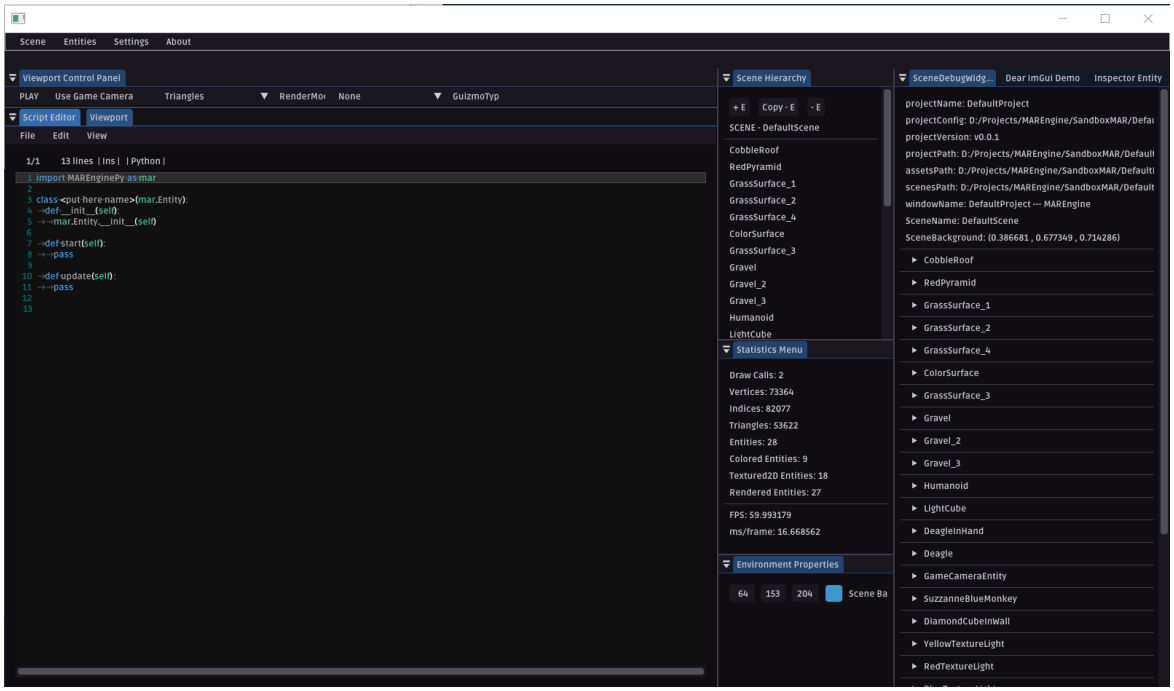
Rysunek 26: Przykładowa wyrenderowana scena z edytorem graficznym.



Rysunek 27: Przykład wykorzystania chwytaka modyfikującego pozycję obiektu w edytorze graficznym.



Rysunek 28: Przykład okienka z ładowaniem tekstury z pliku w edytorze graficznym.



Rysunek 29: Przykład domyślnego okienka z edycją skryptów Pythona oraz okna debugującego w edytorze graficznym.

4 Podsumowanie

Realizacja projektu silnika do gier komputerowych wraz z edytorem graficznym jest zadaniem bardzo trudnym. Silnik jest złożonym oprogramowaniem będącym zbiorem wielu modułów i systemów, nie tylko graficznych. Od programisty wymagana jest wiedza nie tylko ta dotycząca znajomości grafiki komputerowej, technik renderowania i procesowania obiektów. Deweloper musi być obeznany również w wzorcach architektonicznych pozwalających na przejrzystą implementację i napisanie czystego kodu, który w przyszłości będzie mógł być rozwijanym przez innych. Musi posiadać wiedzę na temat projektowania architektury w taki sposób, aby później mogła zostać udostępniona do języka skryptowego bądź edytora graficznego. Ponadto, to szczególnie wielkie wyzwanie dla pojedynczego programisty, ponieważ nie istnieje dywersyfikacja zadań pomiędzy członków zespołu. Deweloper musi mądrze zarządzać czasem, z łatwością nadawać wysoki priorytet zadaniom tego wymagającym.

Podsumowując projekt dyplomowy, udało się stworzyć kompletny silnik umożliwiający renderowanie wielu obiektów przy technice oszczędzenia zasobów i zmniejszenia liczby wywołań funkcji rysowania (poprzez grupowanie obiektów). Zaprojektowano kompletny system ECS pozwalający na dodawanie do jednostek wielu komponentów przypisujących jednostkom konkretne akcje. Dodano interfejs graficzny użytkownika uzupełniający wszystkie dostępne podsystemy poprzez ich wykorzystanie w czasie rzeczywistym przy pomocy edytora graficznego. Obsłużono system plików dzięki czemu użytkownik może ładować utworzone sceny z plików oraz wczytywać modele i tekstury z plików zewnętrznych. W skład silnika wchodzi:

- 21 498 autorskich linii kodu w silniku graficznym,
- 196 klas,
- 41 struktur

Utworzono silnik do gier komputerowych z edytorem graficznych o nazwie *MAREngine* i logiem widocznym na ilustracji 30.

Rysunek 30: Logo silnika *MAREngine*.

Pomimo osiągnięcia sukcesów w spełnieniu założeń przy tworzeniu projektu opisanego w pracy, wiele funkcjonalności może zostać dodanych. System do renderowania można urozmaicić o mapowanie cieni bądź renderowanie PBR (Physically-Based-Rendering). Ponadto aktualne ustawienia projektu nie wspierają możliwości wyboru rozdzielczości renderowania, szczegółowości obiektów bądź tekstur, tzw. poziom detali LOD (ang. *Level Of Detail*). Interfejs graficzny może być rozszerzony o nowe okna i widżety lub personalizację motywów zgodnie z upodobaniem użytkownika.

Bibliografia

[1] Michael Abrash. *Graphics Programming Black Book Special Edition*. Coriolis Group,U.S., 1997.

[2] Sanjay Madhav. *Game Programming Algorithms and Techniques: A Platform-Agnostic Approach (Game Design) 1st Edition*. Addison-Wesley, 2014.

[3] Epic Games Inc. The most powerful real-time 3D creation platform - Unreal Engine. `https://www.unrealengine.com/en-US/`, 2004 (wykorzystano 26 września 2021).

[4] Jason Gregory. *Game Engine Architecture, Third Edition*. CRC Press, 2019.

[5] Tomas Akenine-Möller Eric Haines Naty Hoffman Angelo Pesce Michał Iwanicki and Sébastien Hillaire. *Real-Time Rendering, Fourth Edition*. A K Peters/ CRC Press, 2018.

[6] opengl tutorial.org. OpenGL-Tutorial. `http://www.opengl-tutorial.org/`, 2015 (wykorzystano 27 września 2021).

[7] Murray Bourne. Interactive Mathematics Learn math while you play with it. `https://www.intmath.com/`, 2021 (wykorzystano 28 września 2021).

[8] Joey de Vries. *Learn OpenGL: Learn modern OpenGL graphics programming in a step-by-step fashion.* KW Kendal and Welling, 2020.

[9] Khan Academy. Matrix Transformations | Linear Algebra | Math | Khan Academy. `https://www.khanacademy.org/math/linear-algebra/matrix-transformations`, 2021 (wykorzystano 28 września 2021).

[10] John L. Clevenger V. Scott Gordon. *Computer Graphics Programming in OpenGL with C++)*. Mercury Learning and Information, 2019.

[11] David H. Eberly. *3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics (The Morgan Kaufmann Series in Interactive 3d Technology)*. Morgan Kauffman Publishers, 2007.

[12] Alexander Overvoorde. Vulkan Tutorial. `https://vulkan-tutorial.com/`, 2020 (wykorzystano 27 września 2021).

[13] The Khronos® Vulkan Working Group. Vulkan® 1.2.193 - A Specification (with all registered Vulkan extensions). `https://www.khronos.org/registry/vulkan/specs/1.2-extensions/html/vkspec.html`, 2021 (wykorzystano 27 września 2021).

[14] Steve McConnell. *Code Complete: A Practical Handbook of Software Construction, Second Edition*. O'Reilly Media, Inc., 2004.

[15] Mick West. Evolve Your Hierarchy. `https://cowboyprogramming.com/2007/01/05/evolve-your-heirachy/`, 2007 (wykorzystano 6 października 2021).

[16] Barbara Liskov. *Data Abstraction and Hierarchy*. SIGPLAN Notices, 1988.

[17] tmachine. Entity Systems Wiki. `http://entity-systems.wikidot.com/`, 2010 (wykorzystano 6 października 2021).

[18] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, João Saraiva. Energy efficiency across programming languages. In *How Do Energy, Time, and Memory Relate?*, 2017.

[19] CLIUtils. ModernCMake. `https://cliutils.gitlab.io/modern-cmake/`, 2021 (wykorzystano 3 listopada 2021).

[20] Khronos Group. History of OpenGL. `https://www.khronos.org/opengl/wiki/History_of_OpenGL`, 2020 (wykorzystano 19 grudnia 2021).

[21] Graham Sellers. *Vulkan Programming Guide: The Official Guide to Learning Vulkan*. Addison-Wesley, 2017.

[22] Jr Nicholas Haemel Graham Sellers, Richard S. Wright. *OpenGL SuperBible: Comprehensive Tutorial and Reference*. Addison-Wesley, 2013.

[23] Khronos Group. Shader Storage Buffer Object. `https://www.khronos.org/opengl/wiki/Shader_Storage_Buffer_Object`, 2020 (wykorzystano 19 grudnia 2021).

[24] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson Education (US), 2009.

[25] The Qt Company. Qt Based Games. `https://wiki.qt.io/Qt_Based_Games`, 2021 (wykorzystano 30 grudnia 2021).

[26] The Qt Company. Using 3D engines with Qt. `https://wiki.qt.io/Using_3D_engines_with_Qt`, 2020 (wykorzystano 30 grudnia 2021).

[27] Omar Cornut. Sponsors of Dear ImGui. `https://github.com/ocornut/imgui/wiki/Sponsors`, 2021 (wykorzystano 30 grudnia 2021).

[28] Omar Cornut. Software using dear imgui. `https://github.com/ocornut/imgui/wiki/Software-using-dear-imgui`, 2021 (wykorzystano 30 grudnia 2021).