# Programowanie w API graficznych
## Finalny Raport: AimGL

Politechnika Śląska
Wydział Automatyki, Elektroniki i Informatyki Politechniki Śląskiej

| | |
|---|---|
| **Autor:** | Dawid Grobert |
| **Sekcja:** | VII (jednoosobowa) |
| **Stworzone:** | Styczeń 22, 2024 |
| **Prowadzący:** | dr inż. Damian Pęszor |

# 1 Założenia projektu

Założeniem projektu jest stworzenie narzędzia typu *Aim Trainer* w API graficznym *OpenGL*. Narzędzie tego typu służy do ćwiczenia pamięci mięśniowej w celu polepszenia umiejętności w grach typu *First-Person Shooter*.

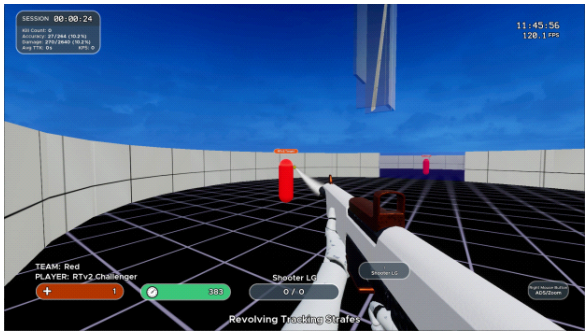




Figure 1: Gra „KovaaK's" stworzona na silniku Unity, która jest jednym z najpopularniejszych narzędzi typu *Aim Trainer*.

Pojęcie „pamięć mięśniowa" odnosi się do sytuacji, gdzie notorycznie powtarzana czynność zostaje „zapamiętana" przez mięśnie. W przypadku gier typu *First-Person Shooter* tyczy się to ruchu jaki należy wykonać myszką komputerową by przenieść celownik na cel (w domyśle innego gracza). Doświadczeni gracze komputerowi robią to w pełni automatycznie i niezwykle szybko. Dzieje się tak ponieważ nie muszą na bieżąco korygować położenia celownika, tylko znają dokładny ruch jaki trzeba wykonać by przenieść celownik na cel – inaczej mówiąc robią to bez zastanowienia, jednym sprawnym ruchem.

## 1.1 Ćwiczenia

Najczęściej spotykane rodzaje cwiczeń to:

- Ściana na której znajdują się rozrzucone tarcze strzeleckie. Rozbicie jednej tarczy powoduje pojawienie się jej w innym miejscu. Celem jest zestrzelenie największej ilości tarcz w jak najkrótszym czasie.
- Ściana na której na krótki czas pojawia się tarcza do zestrzelenia. Poza podstawowym treningiem pamięci mięśniowej pozwala to również ćwiczyć czas reakcji u trenującego.
- Poruszająca się w różnych kierunkach tarcza na której trzeba utrzymać celownik przez jak najdłuższy czas. Jest to trening śledzenia celu.
- Poruszające się we wszystkich kierunkach tarcze, której zestrzelenie powoduje pojawienie się jej w innym miejscu. Pozwala to ćwiczyć umiejętności strzeleckie na poruszającym się celu.

# 2 Narzędzia firm trzecich

W projekcie zostaną użyte następujące biblioteki:

- Biblioteka SFML – tworzenie okna, odtwarzanie dźwieku, pobieranie eventów z klawiatury i myszki. Bez użycia części modułu graficznego odpowiedzilnego za rysowanie obiektów 2D.
- Biblioteka glm – biblioteka matematyczna dla oprogramowania graficznego oparta na specyfikacji GLSL.
- Biblioteka stb – biblioteka służąca do łatwego ładowania obrazów.
- Biblioteka spdlog – biblioteka do loggowania (na przykład w konsoli).
- Biblioteka result – lekka alternatywa obsługi błędów dla wyjątków na styl Rust.
- Biblioteka glew – zapewnia wydajne mechanizmy run-time do określania, które rozszerzenia OpenGL są obsługiwane na platformie docelowej.
- Biblioteka minitrace – służaca do pomiaru czasu wykonywania się fragmentów kodu.
- Biblioteka imgui – służaca tylko w celach debugowych jako pomoc w developmencie.
- Biblioteka imgui-sfml – pośrednia biblioteka łącząca SFML z ImGui.
- Program gimp – program graficzny do edycji i tworzenia tekstur 2D.
- Program blender – program do edycji i tworzenia modeli 3D.

Projekt wraz z zewnętrznymi bibliotekami zostanie połączony w całość przy użyciu `CMake`, będącym narzędziem do zarządzania procesem kompilacji oprogramowania. Biblioteki będą pobierane i kompilowane w momencie uruchomienia projektu.

# 3 Analiza problemu

Aplikacja powinna zostać przygotowana z użyciem czystego `OpenGL API`. Wciąż mimo wszystko potrzebne są biblioteki, które pozwolą na:

- Stworzenie i zarządzanie oknami
- Obsługę zdarzeń wejściowych
- Obsługę Audio

W kontekście OpenGL jest kilka popularnych bibliotek, które umożliwiają tworzenie i zarządzanie oknami, jak również obsługe zdarzeń wejściowych. Są to między innymi:

- GLFW
- GLUT
- freeglut
- SDL (Simple DirectMedia Layer)
- SFML (Simple and Fast Multimedia Library)

`GLFW, GLUT` (lub jego rozszerzenie `freeglut`) raczej zawierają podstawowe funkcjonalności zarządzania oknami i wejściem. Są przy tym dosyć proste i niewielkie. W przypadku `SDL` czy `SFML` sytuacja wygląda trochę inaczej, ponieważ są to biblioteki pozwalające na więcej niż tylko tworzenie okna w kontekście OpenGL. Służą też do obsługi dźwięku, klawiatury, myszy, posiadają moduły sieciowe, czy klasy pozwalające załadować i wyświetlić elementy graficzne.

Moim faworytem zdecydowanie jest `SFML`, ponieważ posiada jeszcze wyższy posiom abstrakcji niż SDL i jest w pełni obiektowy. W projekcie chcąc trzymać się techniki `RAII (Resource Acquisition Is Initialization)` znacznie łatwiej jest mi to osiągnąć przy użyciu biblioteki SFML. Oprócz tego do projektu załączę też `GLEW` by móc swobodnie korzystać z OpenGL i rozszerzeń.

## 3.1 Moduły SFML

SFML składa się z pięciu modułów:

- Audio – dźwięki, streaming (muzyka lub niestandardowe źródła), nagrywanie, uprzestrzennianie.
- Graphics – moduł grafiki 2D: sprite'y, tekst, kształty.
- Network – pozwala na komunikację opartą na gniazdach, dostarcza narzędzia i protokoły sieciowe wyższego poziomu (HTTP, FTP).
- System – bazowy moduł SFML, definiujący różne narzędzia (zegar, czas, wektory).
- Window – okna oparte o OpenGL, abstrakcje zdarzen i obsługi wejścia.

Zamierzam wykorzystać tylko trzy z dostępnych pięciu modułów. Są to: `Audio, System, Window`. Moduł graficzny jest niepotrzebny, ponieważ zostanie napisany od zera na potrzeby tego projektu. Moduł sieciowy również nie zostanie wykorzystany, ponieważ projekt nie przewiduje wsparcia dla gry sieciowej.

## 3.2 Abstrakcja OpenGL

Chcąc trzymać się wzorca projektowego `RAII (Resource acquisition is initialization)` istnieje potrzeba owinięcia zawołań do `OpenGL` dodatkową warstwą abstrakcji. W tym celu potrzebne będą wrappery widoczne na figurze 2.

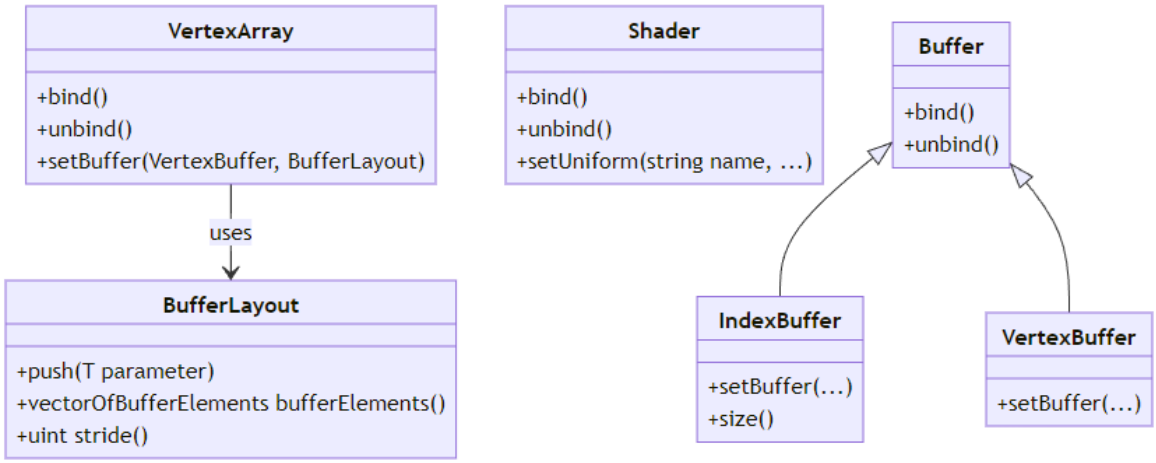



Figure 2: Propozycja abstrakcji OpenGL

- Obiekty widoczne na figurze 2 powinny zajmować się alokacją zasobów w momencie inicjalizacji i zwolnienia ich w momencie wołania destruktora obiektu.
- Obiekt `Shader` powinien być tworzony w oparciu o listę argumentów, gdzie jeden argument składa się z typu shadera (na przykład Fragment/Vertex) oraz z ścieżki do pliku shadera.
- `BufferLayout` opisuje sposób ułożenia danych w VertexBufferze.

Następnie powinny zostać utworzone klasy:

- Texture – klasa reprezentującą obraz żyjący na karcie graficznej.
- Rectangle2D – klasa rysująca prostokąt o podanych wymiarach, kolorze i przeźroczystości.
- Sprite2D – prosty obiekt 2D renderowany na ekranie niezależnie położenia kamery.
- Sprite3D – prosty obiekt 3D będący obiektem 2D renderowanym w przestrzeni 3D jako zwykły prostokąt z nałożoną na niego teksturą.
- Model – graficzna reprezentacja obiektu trójwymiarowego w świecie gry.

## 3.3 Pętla gry

Główna pętla gry jest zdecydowanie bardzo ważnym detalem w którym już na początku można popełnić sporo błędów. Przykładem jest kod załączony poniżej.

```
def gameLoop():
    while window.isOpen():
        updateGame()
        handleEvents()
        render()
```

Figure 3: Pseudokod najprostszej pętli gry

Brak czasu w aktualizacji gry świadczy o tym, że prędkość gry zależna jest od mocy obliczeniowej maszyny. Czas jest chociażby potrzebny w celu ustalenia dystansu przebytego przez gracza między dwoma klatkami.

```
def gameLoop():
    Clock clock
    time_elapsed = Time::Zero
    while window.isOpen():
        time_elapsed = clock.restart()
        updateGame(time_elapsed)
        handleEvents()
        render()
```

Figure 4: Pseudokod poprawionej pętli gry.

Niestety nawet takie rozwiązanie nie jest idealne. Takie rozwiązanie może prowadzić do problemów, gdzie gracz "przeniknie" przez przeszkodę. Zakładając, że przerwa między dwoma klatkami będzie na tyle duża by dystans poruszania się był większy niż grubość ściany to do wykrycia kolizji nigdy nie dojdzie. Dlatego zamierzam zastosować pętlę gry przedstawioną na figurze 5.

```
def gameLoop():
    Clock clock
    Clock fixedUpdateClock
    Time timeSinceLastFixedUpdate = Time::Zero
    time_elapsed = Time::Zero
    while window.isOpen():
        time_elapsed = clock.restart()
        updateGame(time_elapsed)

        timeSinceLastFixedUpdate += fixedUpdateClock.restart()
        if timeSinceLastFixedUpdate > TIME_PER_FIXED_UPDATE:
            do:
                timeSinceLastFixedUpdate -= TIME_PER_FIXED_UPDATE
                fixedUpdate(TIME_PER_FIXED_UPDATE)
            while timeSinceLastFixedUpdate > TIME_PER_FIXED_UPDATE

        handleEvents()
        render()
```

Figure 5: Pseudokod pętli z dodatkową funkcją dla obliczeń fizyki.

Pętla widoczna w figurze 5 nadrabia utracone aktualizacje. Przez co długi odstęp czasowy między dwoma klatkami nie przekłada się na nagłe przeskoczenie w świecie gry. Taka pętla jest osobno używana na przykład do obliczeń fizyki.

## 3.4 Stos stanów

Stos stanów powinien pozwalać wykonać diagram maszyny stanowej przedstawiony w figurze 6. Przy czym istotne jest, że po powrocie ze stanu pauzy, stan gry powinien być kontynuowany od poprzedniego momentu. Nie powinien zostać tworzony na nowo.


Figure 6: Diagram maszyny stanowej przedstawiający przepływ gry.

Zachowanie stosu stanów powinno więc działać w sposób pokazany na figurze 7.

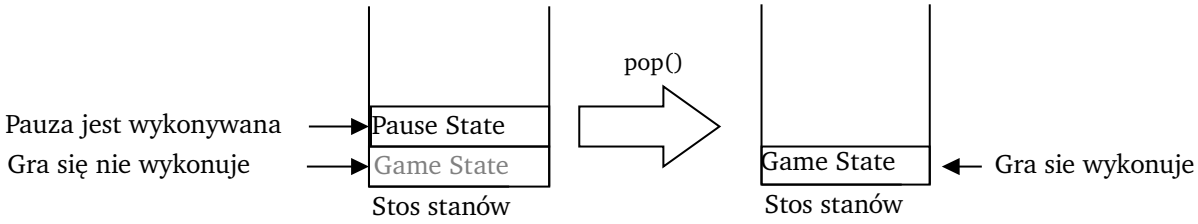



Figure 7: Zachowanie stosu stanów z więcej niż jednym stanem.

Co warto jednak zaznaczyć to, że w przypadku figury 7 nie chcemy aktualizować stanu gry. Mimo wszystko wciąż oczekiwane jest, że w momencie pauzy będzie widoczny zatrzymany stan gry. Dlatego stos stanów powinien zaczynając od spodu stosu, zacząć rysować wszystkie stany aż do stanu na górze stosu. Kiedy jednak aktualizowana jest logika gry, stos stanów powinien zaktualizować stan na górze stosu i pozwolić mu decydować, czy może zaktualizować stany znajdujące się pod nim. Jest to bardzo elastyczny system pozwalający na stworzenie wiele niestandardowych przepływów.

## 3.5 Fizyka

By zaimplementować fizykę potrzebny będzie rejestr colliderów. Rejestr mając dostęp do wszystkich colliderów na scenie będzie mógł spróbować sprawdzić kolizję między wszystkimi colliderami.

W przypadku kolizji między colliderami wewnątrz rejestru, powinien być wołany callback na każdym kolidującym colliderze. Taka wiadomość powinna też z sobą nieść dodatkową informacje z czym kolidował collider.

```
def updateAllCollisions():
    for collider1, collider2 in colliders:
        if collider1.collidesWith(collider2):
            collider1.executeCallback(collider2)
            collider2.executeCallback(collider1)
```

Figure 8: Pseudokod wewnątrz rejestratora colliderów sprawdzający kolizje między colliderami i wołający ich callbacki w przypadku kolizji.

W przypadku kolizji między graczem a światem założenie jest dosyć proste. W przypadku gracza wystarczy do każdej osi na której się porusza z osobna dodać prędkość w danej iteracji. Następnie sprawdzić czy gracz z czymkolwiek koliduje. Jeżeli gracz koliduje przywracamy pozycję z przed kolizji i zerujemy prędkość gracza w danej osi. W takim przypadku rejestr colliderów będzie też potrzebował funkcji na sprawdzenie kolizji między colliderami z wnętrza rejestru a colliderem podanym w argumencie funkcji.

```
def updatePosition():
    for position, velocity in each axis:
        position += velocity
        if colliderRegister.collideWith(this):
            position -= velocity
            velocity = 0
```

Figure 9: Funkcja aktualizująca pozycję gracza zależnie od wykrycia kolizji.

Zakładając, że nie każdy collider musi blokować przemieszczanie się gracza, można dodać tag. Takim tagiem mógłby byc na przykład tag `Solid` i tylko na te collidery reagowałby gracz w kwestii blokady przemieszczania się.

# 4 Specyfikacja wewnętrzna

## 4.1 Główna architektura gry

Główna architektura gry opiera się o klasę `Game` oraz `StateStack`.

- Game – zapewnia główną pętlę gry, oraz okno gry.
- StateStack – zapewnia złożony mechanizm stosu pozwalający na tworzenie stanów (nawet z poziomu znajdujacego się na nim stanu) i usuwanie stanów ze stosu. Pozwala znajdującym się na nim stanach decydować o kontynuowaniu wykonywania się funkcji w dół stosu.

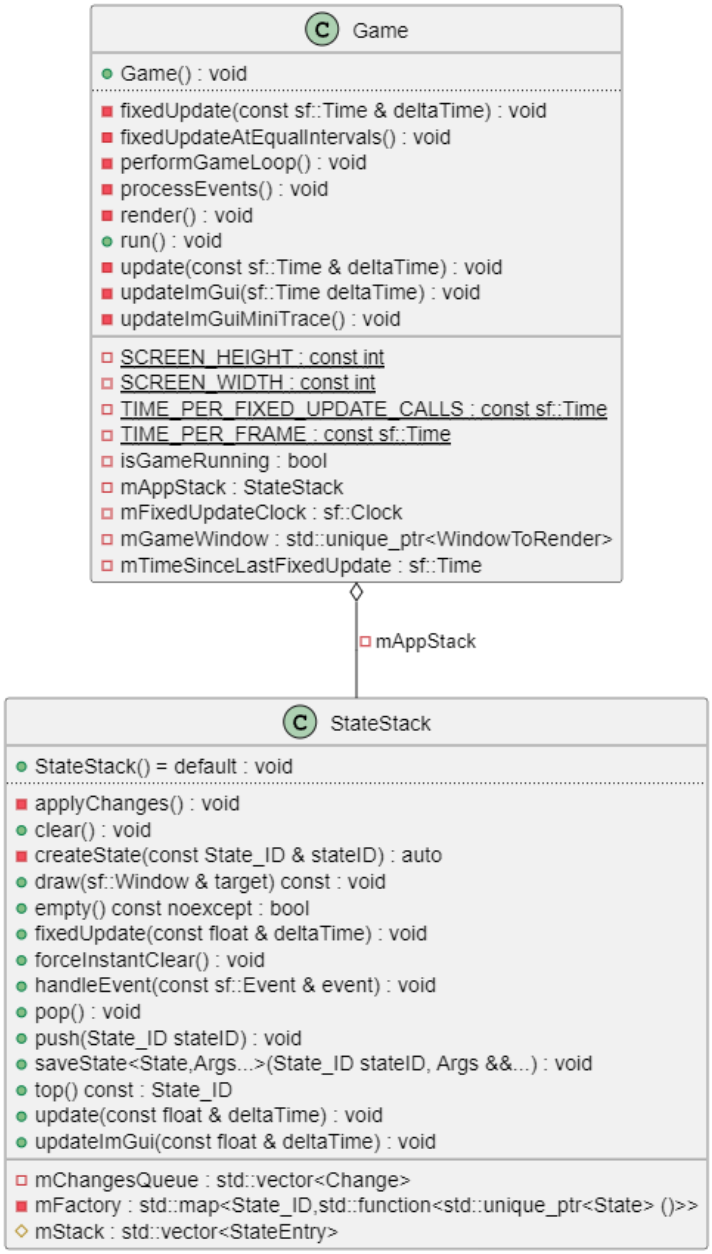



Figure 10: Diagram klas przedstawiający główną klasę gry wraz z stanem stosu.

### 4.1.1 Stany

Stany na stosie dziedziczą po klasie bazowej stanu (`State`). Każdy stan ma 5 podstawowych funkcji, które może nadpisać a które wywoływane są przez `StateStack`.

- draw – rysuje obiekty na ekran.
- fixedUpdate – zapewnia stałą liczbę zawołań funkcji. Na przykład 60 w ciągu sekundy. W przypadku gdy gra „nie wyrabia" to zwalnia by nadrobić brakujące zawołania.
- update – funkcja aktualizująca wywołująca się raz na iteracje pętli gry. Zawiera czas od ostatniej iteracji pętli gry.
- handleEvent – obsługuje zdarzenia wejścia takie jak mysz, czy klawiatura.
- updateImGui – aktualizuje dane związane z debugowym frameworkiem ImGui.

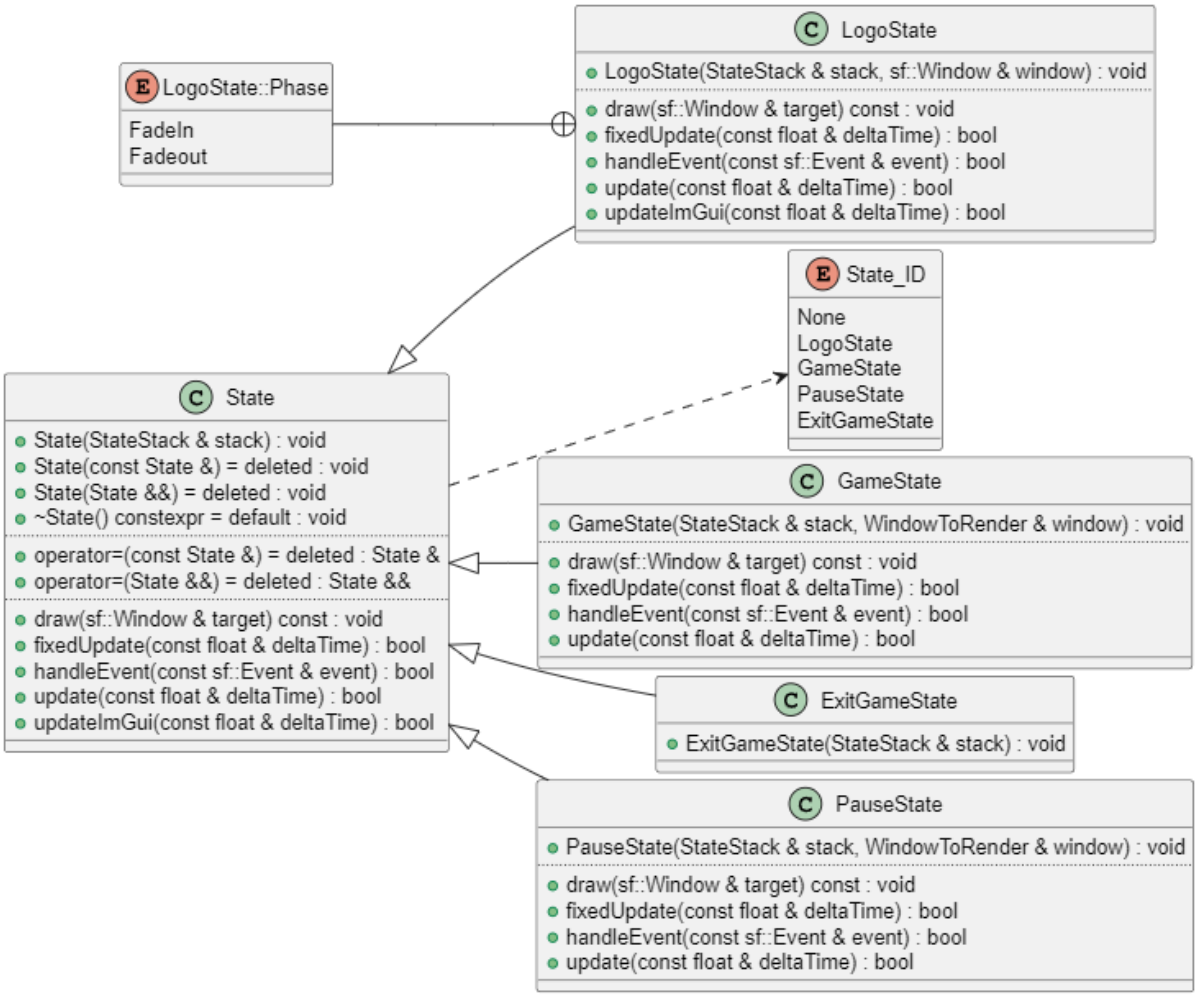



Figure 11: Wszystkie stany dostępne w grze.

W grze znajdują się 4 stany gry:

- LogoState – wyświetla logo gry, które stopniowo się pojawia i po chwili znika. Po tym stanie następuje przejście do stanu gry.
- GameState – stan gry w którym gracz ma możliwość poruszania się, strzelania i trenowania.
- PauseState – stan pauzy, który blokuje aktualizacje do poniższych warstw stosu.
- ExitGameState – stan wyłączenia gry, który wrzucony na stos kończy działanie programu.

## 4.2 Struktura projektu

Struktura projektu rozkłada się na cztery główne foldery:

- `resources` – wszelkie zasoby potrzebne do działania do gry jak modele, shadery czy dźwięki.
- `src` – kod źródłowy aplikacji.
- `tests` – testy aplikacji (niestety raczej nieużytkowane)
- `vendor` – zewnętrzne biblioteki.

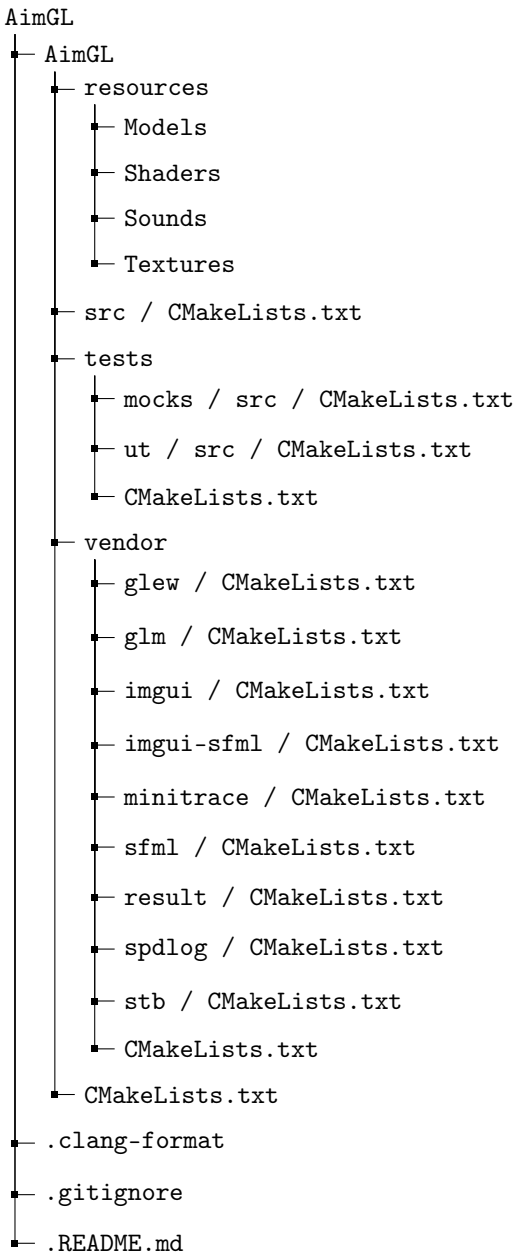



Figure 12: Struktura plików w projekcie.

## 4.3 Stos stanów

Każdy stan (`State`) na stosie ma możliwość poprosić o wyczyszczenie stosu, usunięcie siebie z stosu, bądź wrzucenie nowego stanu na stos. Takie zmiany są aplikowane na początku każdej z funkcji aktualizujących/rysujących wewnątrz klasy `StateStack`.

Istotnym jest tutaj fakt, że `StateStack` jest tutaj pewnego rodzaju fabryką. Najpierw rejestrowane są na nim stany – przekazywane są argumenty potrzebne do utworzenia stanu wraz z odpowiadającym mu identyfikatorem (enumem). Dzięki temu żaden z stanów „wypychając" nowy stan na stos nie musi uzupełniać argumentów jego konstruktora. Wystarczy tylko identyfikator stanu.

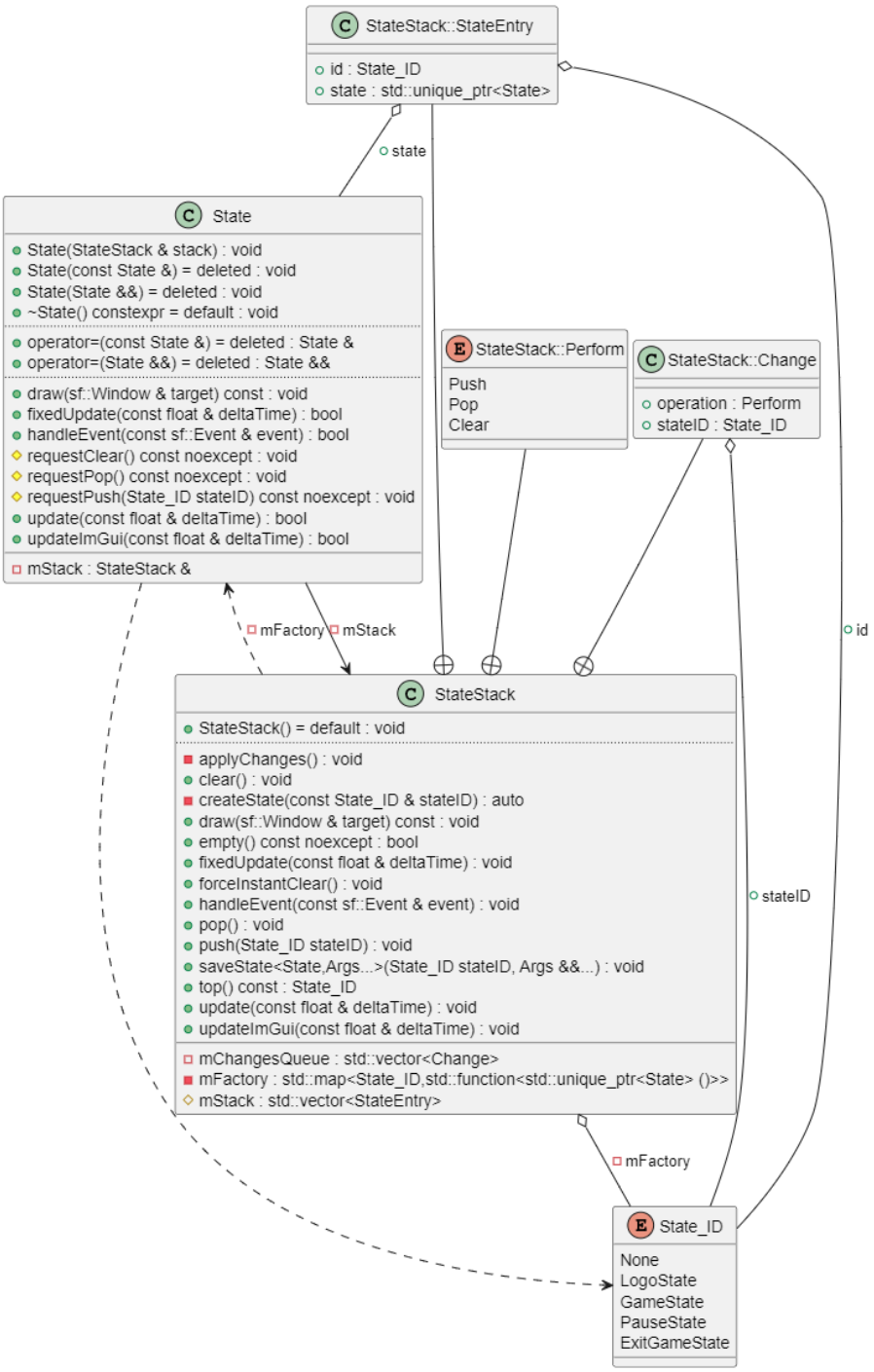



Figure 13: Architektura stosu stanów.

Na samym początku w klasie `Game` zapisywane są stany, a następnie wypychany jest pierwszy stan na stos.

```
mAppStack.saveState<LogoState>(State_ID::LogoState, *mGameWindow);
mAppStack.saveState<ExitGameState>(State_ID::ExitGameState);
mAppStack.saveState<GameState>(State_ID::GameState, *mGameWindow);
mAppStack.saveState<PauseState>(State_ID::PauseState, *mGameWindow);

mAppStack.push(State_ID::LogoState);
```

Figure 14: Kod zapisujący stany do StateStacka i następnie wypychający LogoState na stos.

Metoda `saveState` nie tworzy żadnych obiektów samych w sobie – to byłoby zbyt obciążające dla aplikacji. Jest to tylko informacja jak stworzyć dany stan, kiedy wołana jest metoda `requestPush`. Dzięki temu, że bazowa klasa stanu posiada metody takie jak `requestPush`, czy `requestPop` bez potrzeby zapewnienia argumentów konstruktora stanu to każdy aktywny stan ma możliwość zarządzania stanem stosu.

```
case Phase::Fadeout:
    mLogo.setOpacity(2.f - mClock.getElapsedTime().asSeconds() / 2.f);
    if (mClock.getElapsedTime().asSeconds() > 4.f)
    {
        requestPop();
        requestPush(State_ID::GameState);
    }
    break;
```

Figure 15: Część kodu LogoState odpowiedzialna za usunięcie się ze stosu i wypchnięcie stanu gry.

Następnie takie zapytania są przechowywane w kolejce i aplikowane na początku którejkolwiek funkcji aktualizującej, czy rysującej (patrz Fig. 16).

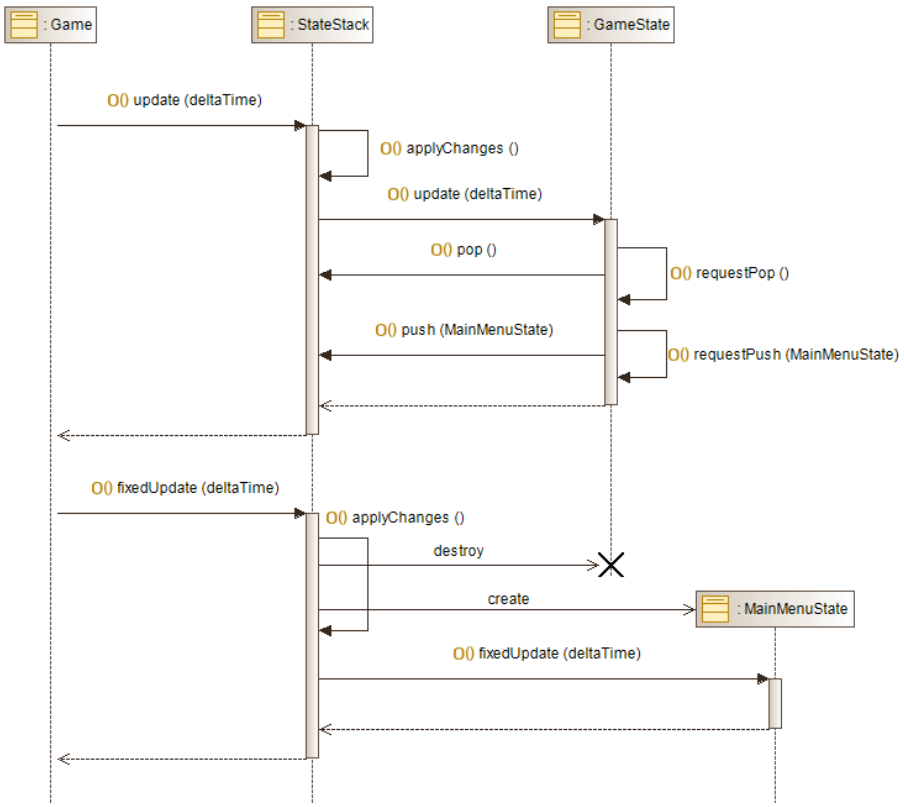



Figure 16: Diagram sekwencji przedstawiającą aktualizację stanu stosu na podstawie kolejki zmian.

## 4.4 Fizyka

Fizyka w głównej mierze opiera się na korzyściach polimorfizmu. Główne skrzypce gra rejestr colliderów (`ColliderRegister`) i bazowa klasa `Collider`.

- `ColliderRegister` – zbiera w sobie collidery, na których przeprowadzana jest detekcja kolizji.
- `Collider` – bazowa klasa potrzebująca w konstruktorze `ColliderRegister` do poprawnego działania. Przy tworzeniu wpisywana jest w rejestr, a przy usuwaniu jest usuwana z rejestru.
- `ColliderTag` – określa typ collidera. Jest to dodatkowa informacja, którą collider z sobą niesie. Jest to informacja czy jest to na przykład pocisk, czy może ciało stałe.

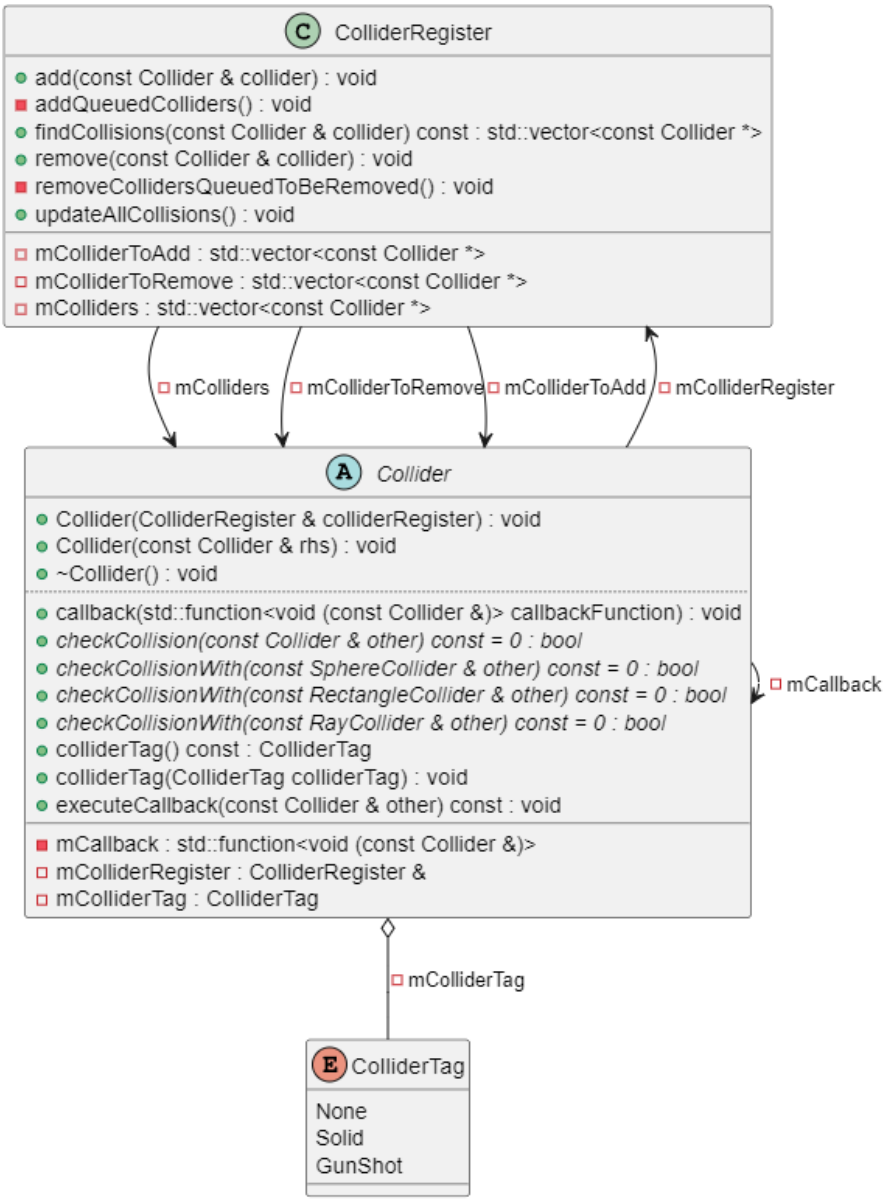



Figure 17: Architektura rejestratora colliderów.

`ColliderRegister` wyznaczoną ilość razy w sekundzie wykonuje funkcję aktualizującą kolizje w swoim rejestrze. Jest ona dosyć prosta i polega na stopniowym rozpakowywaniu typów korzystając z mechanimzu późnego wiązania (polimorfizmu dynamicznego). Rejestr w tym celu przechowuje wskaźniki do klasy bazowej kolizji.

```
std::vector<const Collider*> mColliders;
```

Figure 18: Kontener przechowujący wskaźniki do colliderów wewnątrz rejestra colliderów.

Następnie sprawdza kolizje między dwoma `collider`ami i w przypadku wystąpienia kolizji wywołuje na nich `callback`.

```
void ColliderRegister::updateAllCollisions()
{
    addQueuedColliders();
    removeCollidersQueuedToBeRemoved();
    for (size_t i = 0; i < mColliders.size(); ++i)
    {
        for (size_t j = i + 1; j < mColliders.size(); ++j)
        {
            if (mColliders[i]->checkCollision(*mColliders[j]))
            {
                mColliders[i]->executeCallback(*mColliders[j]);
                mColliders[j]->executeCallback(*mColliders[i]);
            }
        }
    }
}
```

Figure 19: Sprawdzanie kolizji wewnątrz rejestra colliderów. Wołanie callbacków na colliderach z którymi wystąpiła kolizja.

Samo sprawdzanie kolizji najpierw po `vpt`'rze obiektu ląduje w `v-table` a przy jego pomocy dynamicznie wybiera odpowiednią implementacje. W ten sposób w figurze 20 trafia do `RayCollider::checkCollision`. Na drugim obiekcie, który jest referencją do klasy bazowej również wołana jest funkcja sprawdzająca kolizję. Ta już jednak jako argument dostaje rozpakowany typ Collidera. Proces się powtarza, ale finalnie wykona się funkcja w której znane są już oba typy.

```
bool RayCollider::checkCollision(const Collider& other) const
{
    return other.checkCollisionWith(*this);
}
```

Figure 20: Funkcja collidera promienia sprawdzająca kolizje z innym obiektem. Robi to przez wołanie sprawdzenia kolizji tego collidera z wiadomym już typem collidera – `RayCollider`. Korzysta w ten sposób z dynamicznego polimorfizmu.

Każda kolejna implementacja klasy bazowej `Collider` wymaga dopisania metody `checkCollisionWith` w klasie abstrakcyjnej, oraz w klasach pochodnych. Na przykład sfera musi wiedzieć jak sprawdzać kolizję z AABB (Axis-Aligned Bounding Box), ale też osobno musi to umieć robić z promieniem. Powiązania te widać na figurze 21.

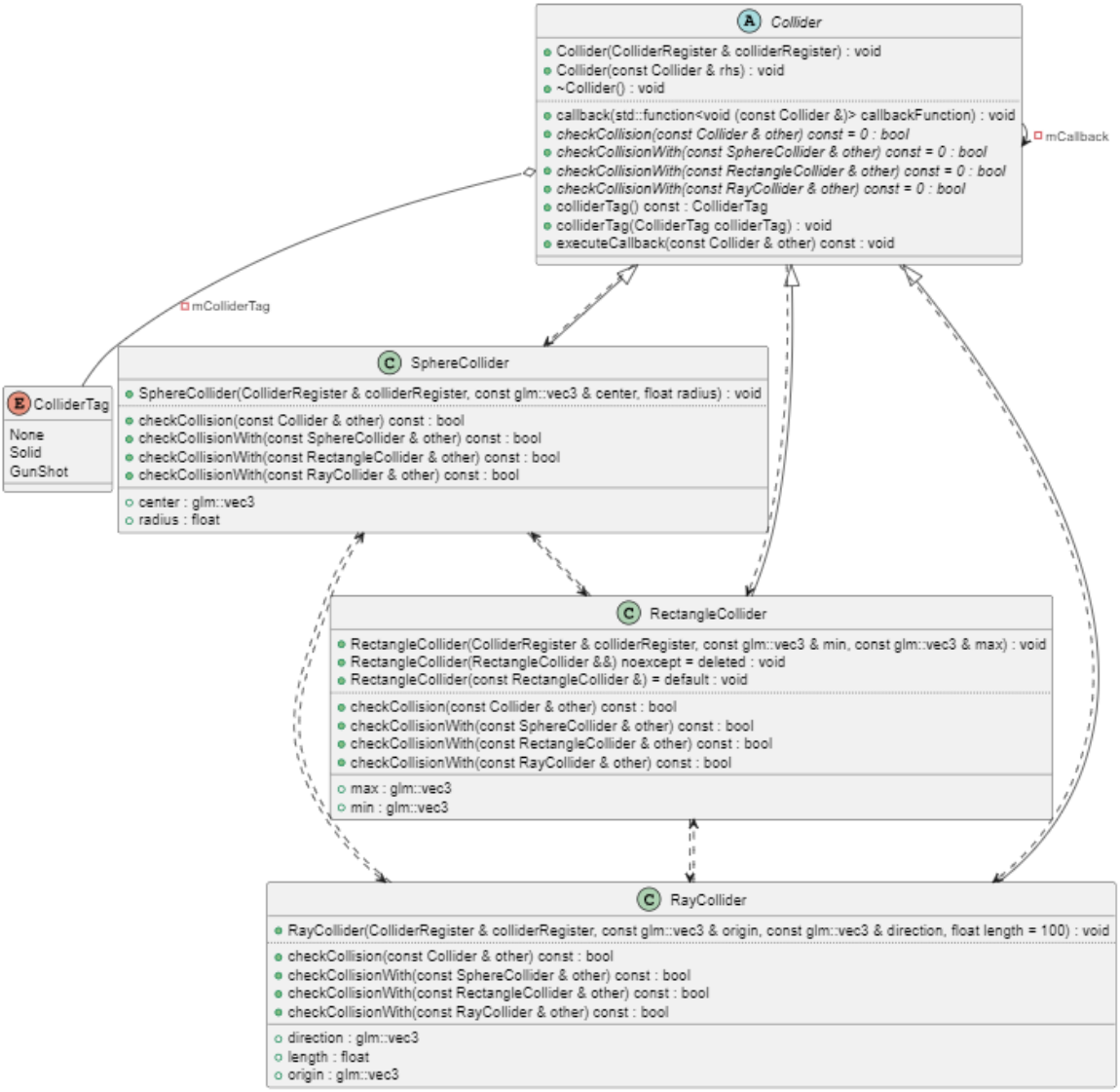



Figure 21: Podchodne abstrakcyjnej klasy `Collider`.

Z klas kolizji korzysta też abstrakcyjna klasa `DrawableCollider`, która pozwala dodatkowo zwizualizować kolizję. W tym przypadku obsługiwane są tylko dwie: `AABB`, oraz `Ray`.

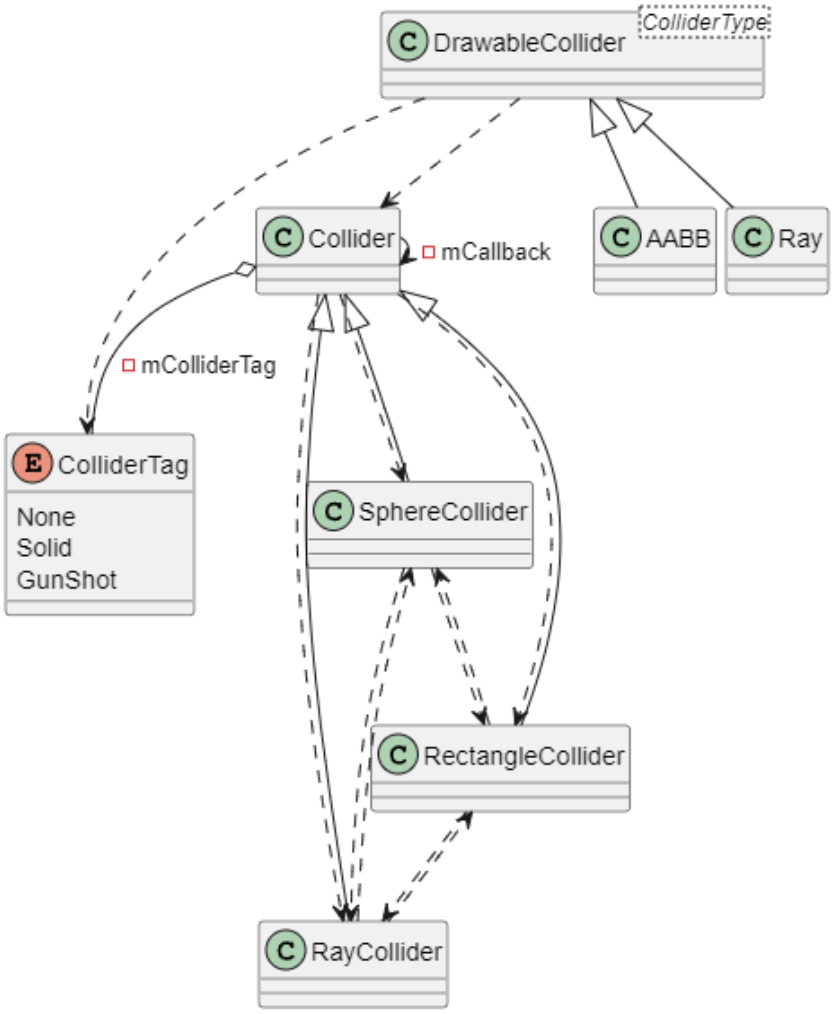



Figure 22: Prosty diagram klas przedstawiający zależności między colliderami (w tym tymi z wizualizacją graficzną).

## 4.5 Abstrakcja OpenGL

Wszystkie rzeczy związane z `OpenGL` znalazły się w folderze `Renderer`. Tam wytworzył się podział na:

- `Core` – Niskopoziome rzeczy w OpenGL (Buffery, Shadery).
- `Graphics` – Większe obiekty graficzne (Modele, tekstury, sprite'y).

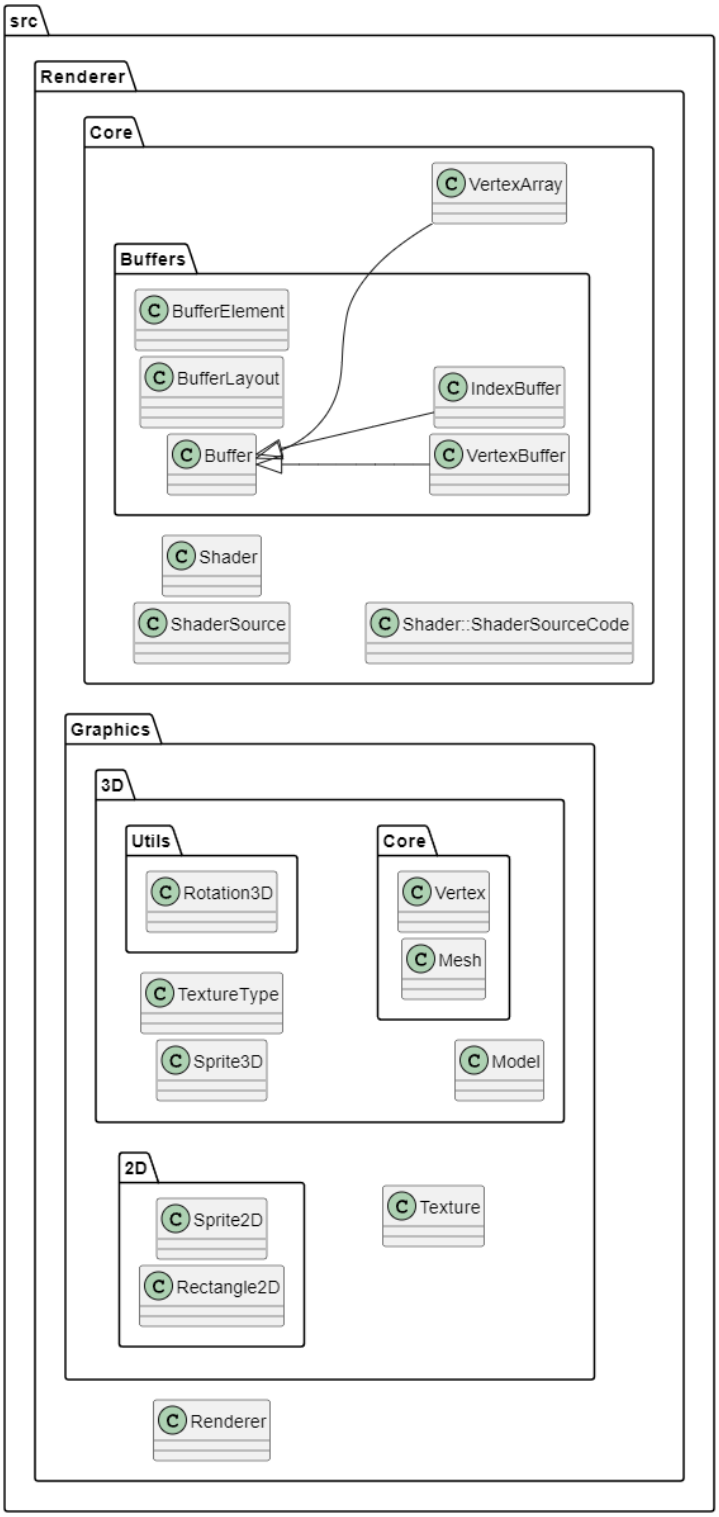



Figure 23: Diagram pakietów przedstawiający strukture folderów `Renderer`'a.

Najbardziej podstawowym elementem całej paczki są oczywiście elementy w folderze `Core`. Znajdują się tutaj `Buffer`'y oraz `VertexArray` (na swój sposób też poniekąd buffer).

- `Buffer` – abstrakcyjna klasa bazowa wszystkich bufferów.
- `VertexBuffer` oraz `VertexArray` – zachowują swoje oryginalne zachowanie (buffer przechowywujący dane, które są przetwarzane przez GPU, oraz buffer przechowywujący indeksy wierzchołków).
- `VertexArray` przyjmuje `VertexBuffer` oraz opisujący jego strukturę `BufferLayout`. `Renderer` przy rysowaniu używa go razem z `IndexBuffer`.

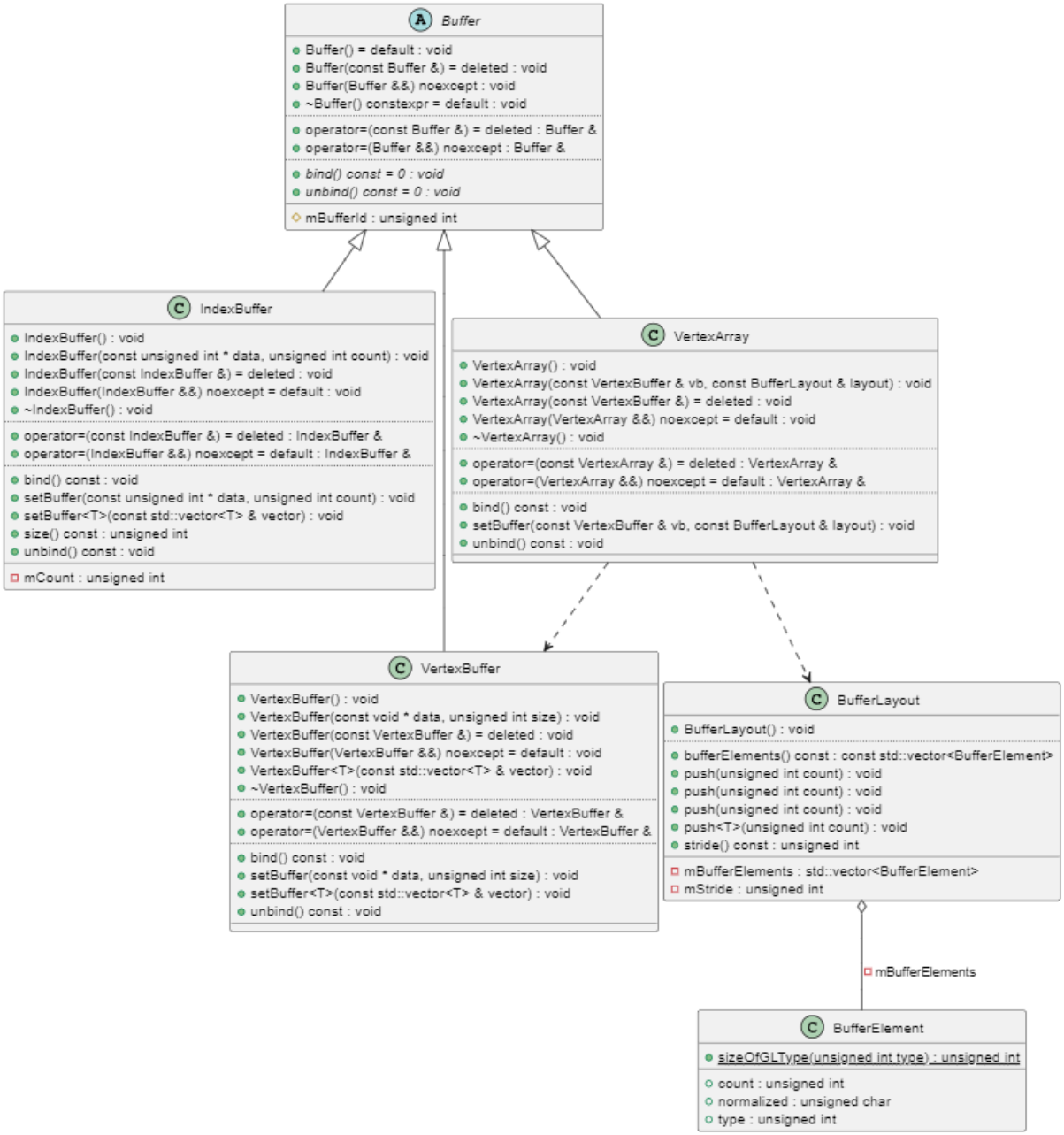



Figure 24: Diagram klas przedstawiający abstrakcję bufferów OpenGL'a.

`Shader` może być delikatnie mylącą nazwą ponieważ jeden obiekt klasy `Shader` może się składać z kilku tworzących go `Shader`ów. Zamysł jest taki, że tworząc `shader` rysujący coś w specyficzny sposób, chcemy móc sprecyzować całą ścieżkę począwszy od *shadera wierzchołków* a kończąc na *shaderze fragmentów*. Dlatego jeden obiekt klasy *Shader* potrafi przyjąć kilka różnych typów. Służy do tego konstruktor *initializer_list* bądź *vector* obiektów *ShaderSource* na które składa sie ścieżka do pliku i typ *shader*a.


Figure 25: Diagram klas przedstawiający implementację shadera.

Przykład tworzenia obiektu klasy `Shader` można zobaczyć w figurze 26.

```
Model::Model(const std::string& objFilePath, const std::vector<TextureType>& texturesToLoad)
    : mShader{{ShaderType::VertexShader, "resources/Shaders/Graphics/Model/ModelTextured.vs"},
              {ShaderType::FragmentShader, "resources/Shaders/Graphics/Model/ModelTextured.fs"}}
```

Figure 26: Sposób konstruktowania obiektu klasy `Shader`.

Następnie `shader` jest bindowany przy ustawianiu `uniform` oraz przed rysowaniem (patrz fig. 27, 28, 29).

```
void Model::draw(const Renderer& target, const Camera& camera) const
{
    mShader.bind();
    mShader.setUniform("model", mLastCalculatedModel);
    mShader.unbind();
    mMesh->draw(target, camera, mShader);
}
```

Figure 27: Ustawianie wartości `uniform` wewnątrz shadera.

```
void Mesh::draw(const Renderer& target, const Camera& camera, const Shader& shader) const
{
    shader.bind();
    setDefaultValuesToMaterialUniforms(shader);
    for (auto i = 0; i < textures.size(); ++i)
    {
        auto& texture = textures[i];
        setTextureToShaderUniform(shader, texture);
        texture.bind(i);
    }
    target.draw3D(mVAO, mEBO, shader, camera);
}
```

Figure 28: Rysowanie siatki. Bindowanie shadera w celu ustawienia domyślnych wartości materiałów.

```
void Renderer::draw3D(const VertexArray& va, const IndexBuffer& ib, const Shader& shader,
                      const Camera& camera, const DrawMode& drawMode) const
{
    shader.bind();
    va.bind();
    ib.bind();
    // ...
}
```

Figure 29: Sposób użycia shadera (bindowanie) chwilę przed rysowaniem.

# 5 Specyfikacja zewnętrzna

W celu zaprezentowania gry przygotowany został zwiastun, który można wyświetlić skanując kod QR, bądź klikając na niego myszką. Są w nim pokazane najważniejsze funkcje gry.

Wideo

Zwiastun gry.

## 5.1 Wymagania sprzętowe

Gra buduje się tylko na systemie `Windows`. `Linux` ani `MacOS` nie są wspierane. Zalecany jest `Windows 10` lub nowszy. Konfiguracja sprzętowa na której testowany był projekt:

- Procesor: Intel Core i5 13600k
- Płyta główna: Gigabyte Z690 Gaming X DDR4
- RAM: 48 GB DDR4
- Karta graficzna: AMD Radeon RX 6700 XT

Testując nie zauważyłem żadnych problemów z wydajnością.

## 5.2 Procedura uruchomienia

### 5.2.1 Wstępnie zbudowany plik wykonywalny

Jeżeli projekt został dostarczony już z wersji zbudowanej z dostępnym plikiem wykonywalnym to wystarczy tylko go uruchomić klikając dwukrotnie na plik wykonywalny `.exe`.

### 5.2.2 Budowanie projektu przy użyciu CLion 2023.3 EAP (Nova)

By zbudować projekt potrzebny jest CMake w wersji przynajmniej 3.24, połączenie do internetu (ponieważ zewnętrzne biblioteki pobierane są podczas budowania), oraz kompilator MSVC obsługujący standard C++20.

1. Uruchom CLion i wybierz opcję "open file or project" (patrz figura 30).

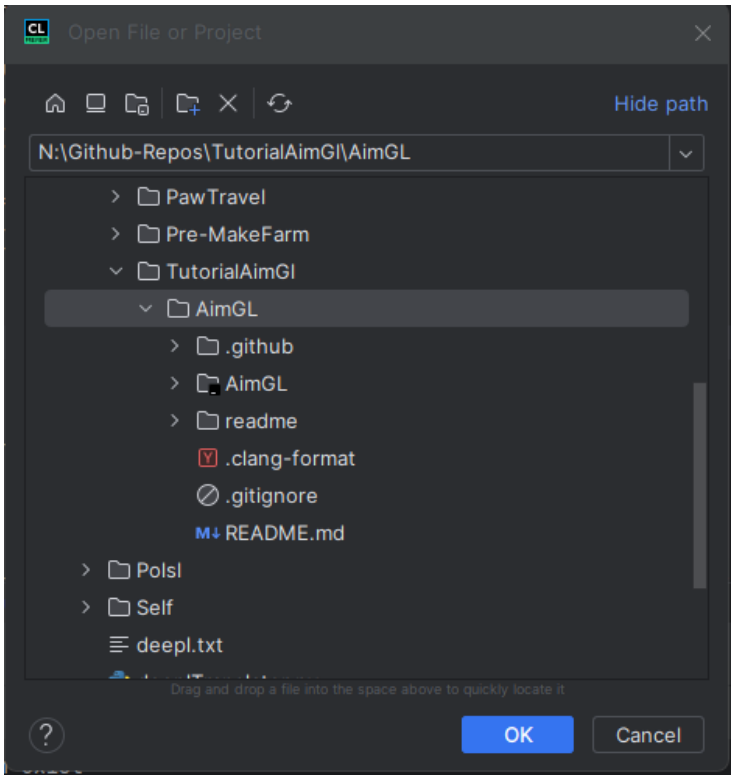



Figure 30: Okno otwarcia istniejącego projektu, zakładka „Open File or Project”

2. Odnajdź CMakeLists.txt w głównym folderze AimGL (gdzie znajduje się folder src). Kliknij prawym przyciskiem myszy na CMakeLists.txt i wybierz opcję „Load CMake Project” (patrz figura 31).

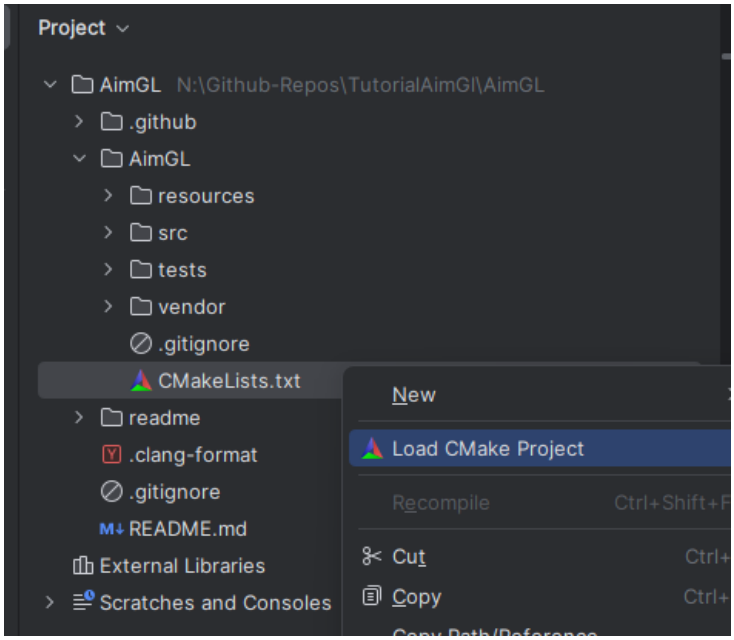



Figure 31: Wczytywanie projektu AimGL poprzez odczytanie głównego pliku projetkowego CMake’a

3. Rozpocznie się pobieranie zewnętrznych bibliotek (patrz figura 32).

```
-- SFML Fetched!
-- Fetching ImGui...
-- ImGui Fetched!
-- Fetching ImGui-SFML...
-- Found ImGui v1.89.9 in N:/Github-Repos/TutorialAimGl/AimGL/AimGL/cmake-build-debug/_deps/imgui-src
-- ImGui-SFML Fetched!
-- Fetching GLM...
...
```

Figure 32: Przykładowy tekst wyświetlający się w zakładce „CMake” gdy projekt się konfiguruje.

4. Konfiguracja powinna zakończyć się napisem: *[Finished]* (patrz figura 33).

```
...
-- Found Python: C:/Users/dawid/AppData/Local/Programs/Python/Python312/python.exe...
-- Configuring done (48.3s)
-- Generating done (0.1s)
-- Build files have been written to: N:/Github-Repos/TutorialAimGl/AimGL/AimGL/cmake-build-debug

[Finished]
```

Figure 33: Informacja o poprawnie skonfigurowanym projekcie

5. Wybierz target „AimGLApp” z listy dostępnych konfiguracji i kliknij przycisk uruchomienia budowania (patrz figura 34).

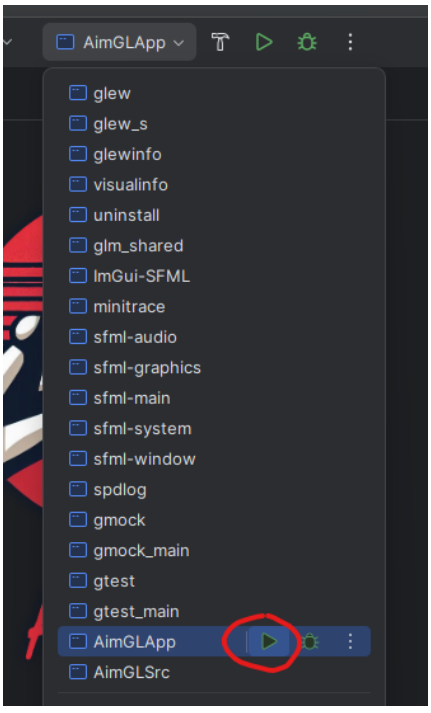



Figure 34: Lista dostepnych konfiguracji do budowania.

6. Gdy kompilacja zostanie zakończona, okno z grą powinno się wyświetlić. W przypadku problemów sprawdź proszę ustawienia wewnątrz *CLion* w zakładce *Build, Execution, Deployment > CMake* upewniając się, że Toolchain ustawiony jest na *Visual Studio*.

## 5.3 Instrukcja użytkownika

### 5.3.1 Pierwsze uruchomienie gry

Grę należy uruchomić poprzez uruchomienie pliku wykonywalnego `.exe`. Powinno pojawić się okno gry i użytkownik powinien zobaczyć logo gry (patrz figura 35).


Figure 35: Logo gry pojawiające się po uruchomieniu aplikacji

Po chwili logo powinno zniknąć i gracz powinien zostać przeniesiony do gry. W tym miejscu pojawia się ekran powitalny z informacją, że pole z treningami znajduje się za graczem. (patrz figura 36). Do różnych trybów treningowych prowadzą strzałki rozmieszczone na mapie (patrz fig. 37)

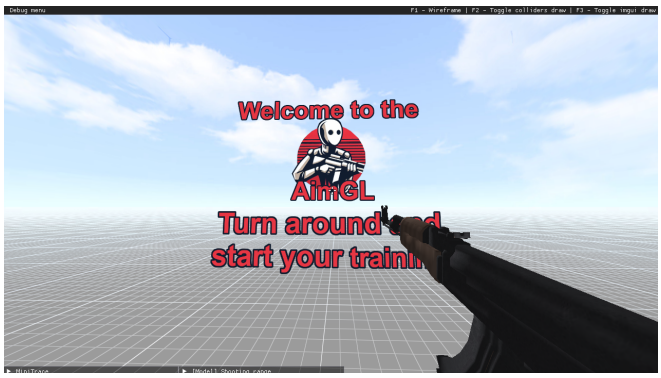



Figure 36: Moment rozpoczęcia gry

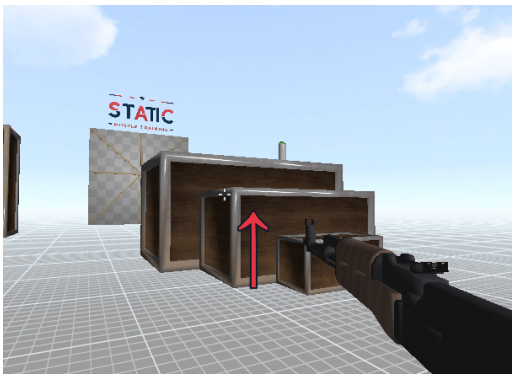



Figure 37: Strzałka wskazująca na tryb treningowy.

Sterowanie jest intuicyjne i przypomina każdy znany dzisiaj *FPS (First Person Shooter)*:

- Gracz porusza się przy użyciu klawiszy **W, A, S, D**.
- Gracz rozgląda się po świecie gry przy użyciu myszki.
- By strzelić z broni należy kliknąć **lewy przycisk myszy**.
- Żeby skoczyć należy wcisnąć **spację**.

### 5.3.2 Tryby treningu

W grze dostępne są dwa tryby treningowe.

#### Trening statycznych celów

W tym trybie na specjalnej tarczy pojawiają się cele. Utrzymują się na niej przez 0.75 sekundy i pojawiają się na nowo w innym miejscu.

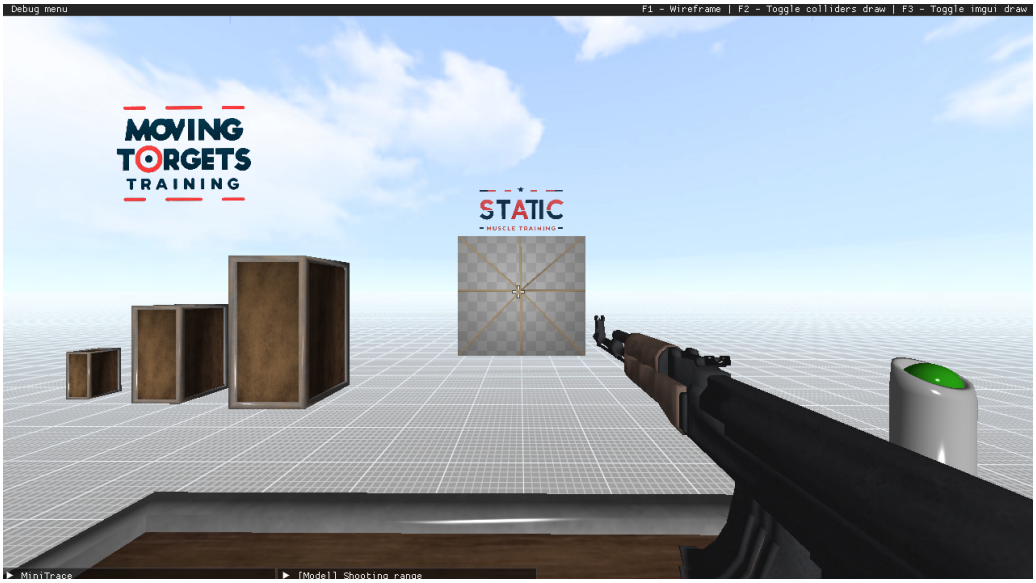



Figure 38: Trening statycznych celów

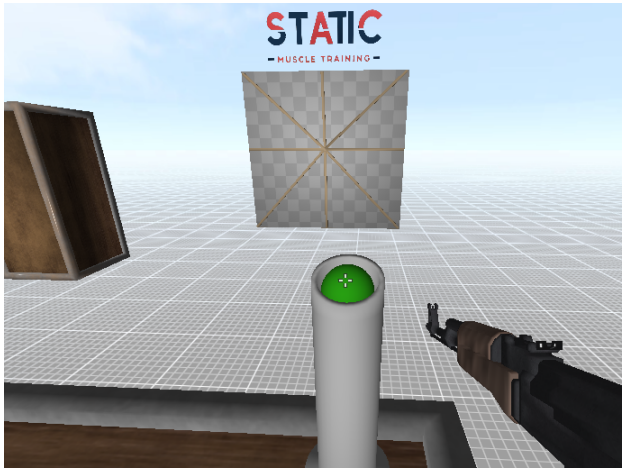
#### Tryb poruszających się celów

W tym trybie cele pojawiają się na różnej wysokości i przemieszczają się w lewo. Jeżeli oddalą się wystarczająco daleko to znikają.

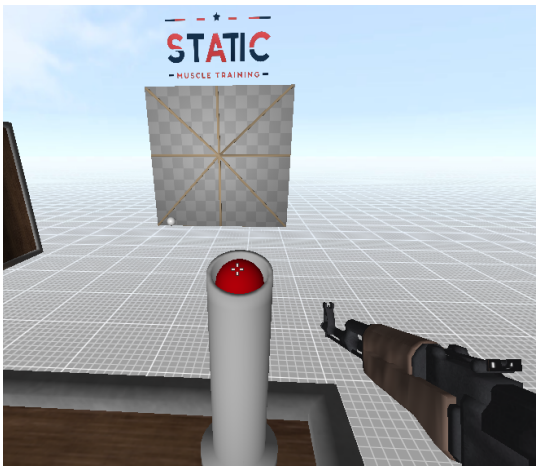

Figure 39: Trening poruszających się celów

### 5.3.3 Uruchomienie i zatrzymanie treningu

Trening można uruchomić strzelając w przycisk umieszczony przed strefą treningową. By zatrzymać trening wystarczy ponownie strzelić w przycisk umieszczony przed strefą treningową.




Przycisk przed postrzeleniem




Aktywowany przycisk (po postrzeleniu)

### 5.3.4 Pauzowanie gry

Grę można zapauzować wciskając przycisk **Escape (ESC)** (patrz fig. 40).


Figure 40: Stan pauzy

### 5.3.5 Opcje debugowe

**Uwaga:** Uwaga, opcje debugowe są dostępne tylko przy zbudowaniu gry w trybie `DEBUG`. Tryb `RELEASE` nie posiada tych opcji.

#### Siatka

Wciskając przycisk **F1**, bądź wybierając taką opcje z rozwijanej listy (patrz 44. Opcje debugowe wewnątrz panelu imgui.) można uruchomić tryb siatki (wireframe).

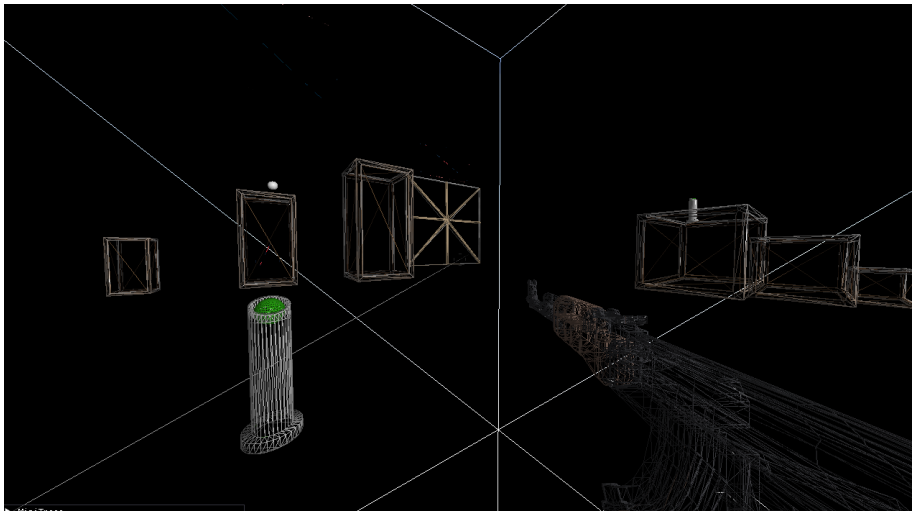


Figure 41: Włączony tryb siatki (wireframe).

#### Wizualizacja colliderów

Wciskając przycisk **F2**, bądź wybierając taką opcje z rozwijanej listy (patrz 44. Opcje debugowe wewnątrz panelu imgui.) można uruchomić tryb wizualizacji colliderów.

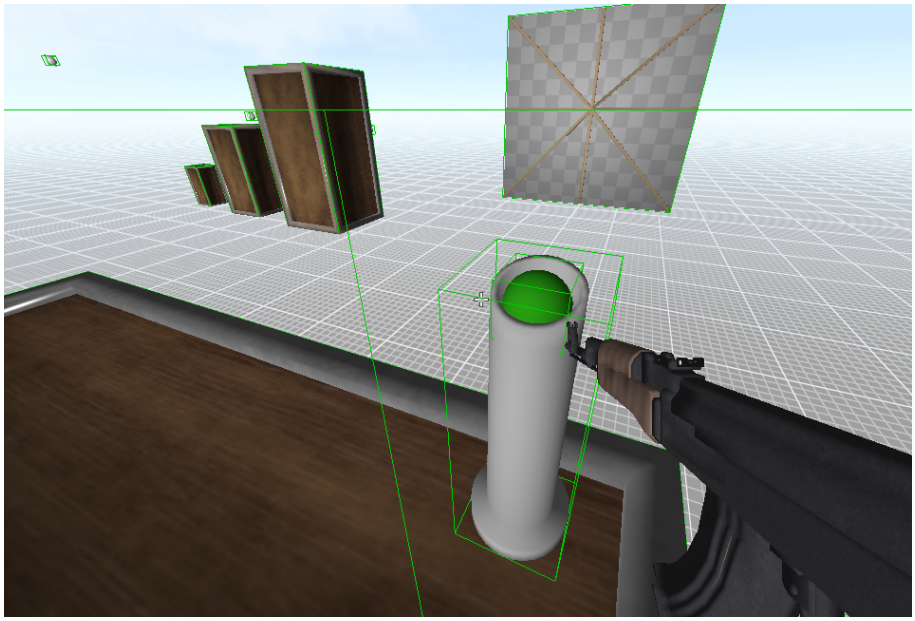


Figure 42: Włączony tryb wizualizacji colliderów.

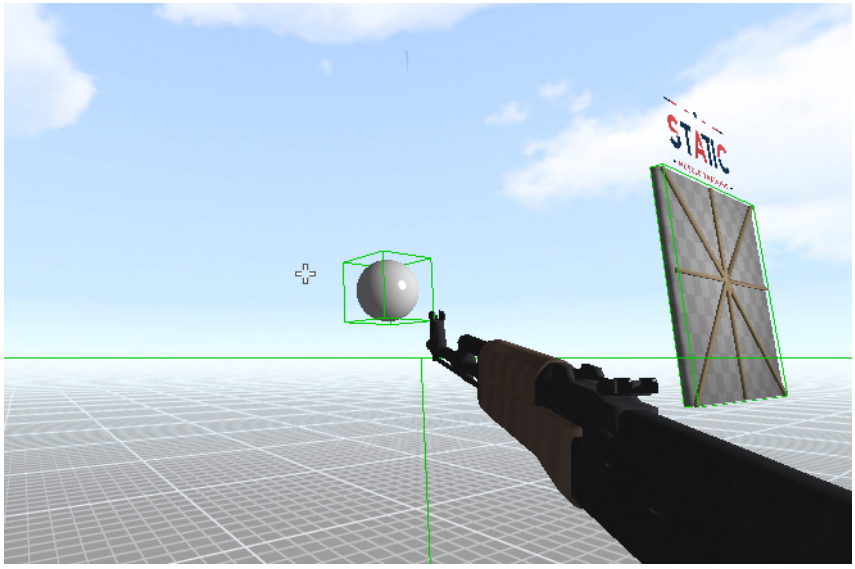


Figure 43: Włączony tryb wizualizacji colliderów

**Panel ImGUI**

Lista rozwijana debugowego panelu ImGui.

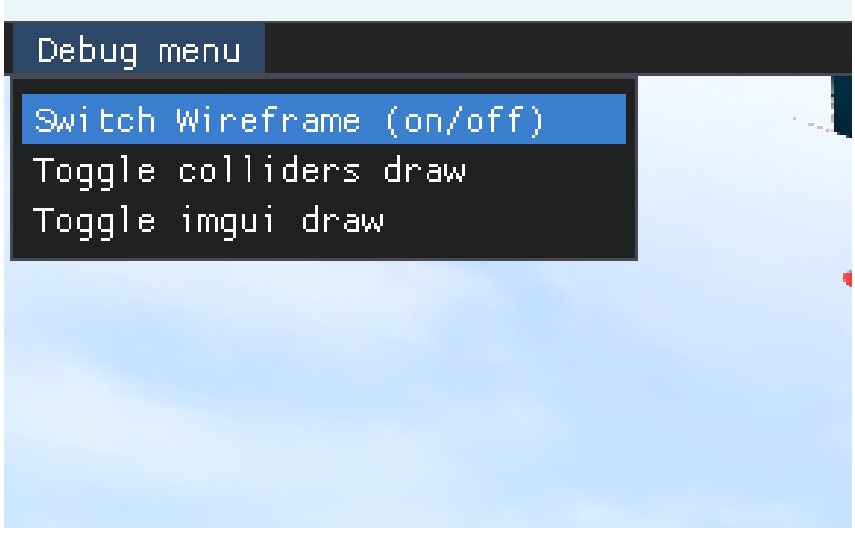



Figure 44: Opcje debugowe wewnątrz panelu imgui.

# 6 Opis procesu testowania

## 6.1 Testy manualne

W głównej mierze aplikacja była testowana przy użyciu testów manualnych. Jest to dość skuteczna metoda przy tworzeniu gier komputerowych, czego dowodem są ogromne ilości testerów zatrudnionych w branży gamedev. W tym projekcie w większości większość funkcjonalności było sprawdzanych ręcznie. Z czasem rozwoju aplikacji stare funkcjonalności również były sprawdzane pod kątem poprawnego działania.

## 6.2 Testy jednostkowe

Projekt został dostosowany pod dodawanie testów jednostkowych z użyciem biblioteki `gmock` i `gtest`. W praktyce niestety żadne testy nie zostały napisane (jest tylko parę testów dla `Statestacka`). Całkiem łatwo można je jednak dodać i są one uruchamiane w ciągłej integracji.

## 6.3 Ciągła integracja

Ciągła integracja wbudowana w Github (Github Actions) okazała się być bardzo przydatna. W tym projekcie każda zmiana w momencie wypychania na główną gałąź repozytorium była sprawdzana pod kątem:

- Budowania projektu.
- Przechodzenia wszystkich testów jednostkowych.
- Formatowania kodu.

Spójne formatowanie kodu ma zapobiec niepotrzebnym zmianom w linijkach tylko z powodu zmienionego formatowania. Zaś budowanie projektu pozwalało sprawdzić czy aplikacja wciąż buduje sie poprawnie. Czasami błąd nie musiał być widoczny lokalnie, gdzie wszystkie dependencje był już pobrane i projekt byłpoprawnie ustawiony. Dlatego budowanie na `CI`'u od zera okazywało się być bardzo przydatne w celu sprawdzenia czy cały proces wciąż działa.

## 6.4 Testowanie wydajności

Dzięki integracji biblioteki `minitrace` można było w prosty sposób prześledzić czas wykonywania się poszczególnych funkcji w celu znalezienia wąskiego gardła wydajnościowego.

## 6.5 Uruchamianie

Uruchamianie aplikacji jest szczegółowo opisane w specyfikacji zewnętrznej (patrz 5. Specyfikacja zewnętrzna).

# 7 Wnioski

To zdecydowanie był mój ulubiony przedmiot w tym semestrze. Sporo swobody w działaniu i jestem całkiem zadowolony z efektów. Szczęśliwie sięgnięcie do niskopoziomowego API nie jest takie trudne i bardzo łatwo jest zastąpić jego przestarzałość odpowiednimi wrapperami wokół zawołań do OpenGLa. Znacznie przyśpiesza to wtedy pracę i zapomina się, że ma sie do czynienia z niskopoziomowym API. Na temat OpenGL są ogromne zasoby wiedzy dostępne w internecie. Jest to twór zdecydowanie niezbyt nowy i na ilość dostępnych materiałów na pewno nie można narzekać. Wiedza zdobyta na wykładach i laboratoriach okazała się bardzo przydatna, a jej braki trzeba było koniecznie nadrobić. Myślę, że dalej będe rozwijał projekt, który tutaj rozpocząłem.