Jakub Maliszewski,

Mateusz Lewczak,

Krzysztof Blankiewicz,

Konrad Polański

# Raport z Projektu zespołowego

**Gra typu FPS(First-Person Shooter) w stylu gry Doom93**

## **Wstęp**

**Założenia projektowe**

Nasz projekt polega na stworzeniu pierwszoosobowej gry typu FPS (First Person Shooter) w perspektywie 2,5D inspirowaną stylem klasycznej gry Doom z 1993 roku. Doom jest jednym z legendarnych tytułów w historii gier wideo, często uznawanym za pioniera gatunku FPS, który zdefiniował wiele podstawowych mechanik, które stały się podstawą tego typu gier do dzisiaj. W grze zastosowana została perspektywa gracza 2,5D, która łączy dwuwymiarowe tekstury z trójwymiarową przestrzenią, tworząc iluzję głębi i realizmu przy zachowaniu prostoty graficznej – bez skomplikowanych modeli 3D.

Celem projektu jest nie tylko odtworzenie kultowej mechaniki rozgrywki, w tym najważniejsze aspekty rozgrywki, takie jak ruch gracza po mapie, walka z przeciwnikami oraz zbieranie przedmiotów i interakcje tych mechanik ze sobą. Projekt dał nam również możliwość zapoznania się z technicznymi aspektami tworzenia silników gier z perspektywą 2,5D. Projekt pozwoli nam także zgłębić zagadnienia takie jak optymalizacja kodu pod kątem wydajności, efektywne zarządzanie pamięcią, renderowanie grafiki w czasie rzeczywistym oraz implementacja podstawowej sztucznej inteligencji przeciwników. Gra będzie naśladować klimat i działanie oryginalnej gry – szybka akcja połączona z prostym sterowaniem, walka z przeciwnikami. Kluczowym elementem będzie zachowanie estetyki retro, z uwzględnieniem oryginalnej grafiki, jednak z pewnymi uproszczeniami na potrzeby projektu zespołowego.

Oryginalnego Dooma93 napisano w języku C, a w pewnych krytycznych aspektach, np. do obsługi grafiki oraz dźwięku wykorzystano assemblera, co zapewniło dużą wydajność na komputerach z tego okresu. My natomiast do skonstruowania naszego silnika gry sięgnęliśmy po język C++ w standardzie C++11, również ze względów wydajnościowych. Do utworzenia okna gry oraz rysowania elementów w grze użyliśmy popularnej i prostej w użytkowaniu biblioteki SFML 2.6.1. Do efektywnego zarządzania kodem źródłowym i umożliwienie wygodnej pracy zespołowej w projekcie wykorzystaliśmy system kontroli wersji Git, który pozwala na śledzenie zmian w kodzie i łatwe integrowanie pracy wszystkich członków zespołu.

W projekcie zastosowaliśmy wzorzec projektowy ECS (Entity-Component-System) do zarządzania logiką silnika gry. Wzorzec pozwoli na elastyczne tworzenie i zarządzanie obiektami gry poprzez podział na trzy główne typy elementów: Encje (Entities), Komponenty (Components) oraz Systemy (Systems). Takie podejście projektowe ułatwia dodawanie nowych elementów gry i modyfikację już istniejących co poprawia skalowalność i elastyczność projektu.

Proces rozwoju gry będzie obejmował regularnie przeprowadzane testy kolejnych implementowanych modułów, aby zapewnić stabilność i jakość tworzonej gry. Końcowym wynikiem projektu będzie gotowa, grywalna wersja demo, prezentująca kluczowe mechaniki gry, dostępna do gry na komputerach PC z przynajmniej jednym poziomem.

**Harmonogram i spis zadań**

Harmonogram projektu został rozpisany na dwa półsemestry, aby zapewnić wystarczająco dużo czasu na projektowanie, implementację, testowanie oraz dokończenie projektu. Daje to także margines na rozwiązanie problemów, które mogą pojawić się w trakcie realizacji projektu. Podczas przydziału zadań podzieliliśmy grę na odpowiednie moduły i systemy, tak aby w miarę możliwości każdy był odpowiedzialny za swoją autonomiczną część projektu, co na końcu pozwoli scalić wszystko w całość.

1. Przygotowanie środowiska programistycznego (Wiosna)
   1. Wybór narzędzi programistycznych (marzec):

* Decyzja o wyborze narzędzi programistycznych, takich jak kompilator, biblioteki, system kontroli wersji oraz środowiska IDE (cały zespół),
* Utworzenie repozytorium i konfiguracja systemu kontroli wersji dla całego zespołu (cały zespół),
  1. Instalacja i konfiguracja narzędzi (marzec – kwiecień)
* Instalacja wybranych narzędzi na komputerach roboczych każdego z członków zespołu (cały zespół),
* Konfiguracja środowiska programistycznego, ustawienia projektu w IDE (cały zespół),
* Stworzenie początkowej struktury katalogów w repozytorium (Mateusz Lewczak),
* Przetestowanie instalacji poprzez uruchomienie przykładowego projektu SFML (cały zespół)

1. Implementacja podstawowej struktury silnika (Wiosna)
   1. Struktura klasy GameEngine (kwiecień) (Jakub Maliszewski),
   2. Implementacja wzorca projektowego ECS (Entity – Component – System) (kwiecień – maj) (Mateusz Lewczak),
   3. Implementacja klasy WADLoader wczytującej dane z pliku o rozszerzeniu .WAD z poziomem (kwiecień – maj) (Mateusz Lewczak),
2. Implementacja głównych systemów logiki gry oraz potrzebnych komponentów (Lato) (do połowy sierpnia)
   1. System kontroli gracza i komponent stanu gracza (Jakub Maliszewski),
   2. System ruchu gracza i komponent ruchu (Jakub Maliszewski),
   3. System broni i komponent broni (Krzysztof Blankiewicz),
   4. System obrażeń i komponenty życia oraz obrażeń (Krzysztof Blankiewicz),
   5. System przeciwników i komponent przeciwnika (Jakub Maliszewski),
3. Implementacja systemów odpowiedzialnych za renderowanie grafiki (Lato)
   1. System renderowania minimapy i komponentu spritu minimapy (Mateusz Lewczak),
   2. System renderowania grafiki i komponentu spritu w grze (Mateusz Lewczak),
   3. System renderowania interfejsu gracza (Jakub Maliszewski).
4. Testowanie, optymalizacja i dokumentacja (Lato)
   1. Testowanie i optymalizacja (sierpień)
      * Przeprowadzenie pełnych testów funkcjonalnych i wydajnościowych, aby upewnić się, że gra działa zgodnie z założeniami projektowymi (cały zespół),
      * Poprawa ewentualnych błędów i optymalizacja kodu w krytycznych obszarach silnika (cały zespół),
   2. Opracowanie dokumentacji i końcowego raportu (sierpień – wrzesień)

* Poprawienie komentarzy w kodzie opisujących strukturę projektu (cały zespół),
* Sporządzenie raportu końcowego, podsumowującego wyniki projektu, napotkane problemy i ich rozwiązania oraz wnioski z realizacji projektu (cały zespół).

## Główne Osiągnięcia Projektu

### 1. Ładowanie Plików WAD

Jednym z fundamentów naszego projektu było umożliwienie pełnego wsparcia dla oryginalnych plików WAD, które zawierają wszystkie dane niezbędne do działania gry, takie jak mapy, tekstury, muzyka, i inne zasoby. Implementując funkcjonalność ładowania WAD-ów, udało nam się osiągnąć pełną kompatybilność z dowolnym poziomem z klasycznego *Doom*’a. Nasz system jest w stanie otworzyć każdy poziom, co było kluczowe do zachowania autentyczności i możliwości rozbudowy gry.

### 2. Renderowanie 3D

Największym wyzwaniem, ale zarazem największym osiągnięciem projektu, było zaimplementowanie pełnego systemu renderowania 3D, opartego na oryginalnych algorytmach. Nasze rozwiązanie obejmuje:

* **Tekstury:** System teksturowania, który pozwala na precyzyjne odwzorowanie powierzchni w grze, zapewniając autentyczny wygląd map.
* **Face Culling:** Optymalizacja poprzez wykluczanie niewidocznych powierzchni, co znacząco poprawia wydajność renderowania.
* **Wykluczenie segmentów poza widokiem:** Algorytm ten, bazujący na drzewie BSP, skutecznie eliminuje segmenty mapy, które znajdują się poza polem widzenia gracza, co redukuje obciążenie procesora graficznego.
* **System portali w BSP:** Implementacja portali, które umożliwiają renderowanie obszarów znajdujących się za ścianami, dodając głębi i realizmu do odwzorowania oryginalnych poziomów.
* **Renderowanie nieba:** Stworzyliśmy system renderowania tła nieba, który wiernie odwzorowuje estetykę *Doom*’a, z dynamicznym dopasowaniem do ruchu gracza.

### 3. Animacja Chodzenia

Aby oddać charakterystyczny dla *Doom*’a styl poruszania się, zaimplementowaliśmy efekt **head bobbing**, czyli animację chodzenia, która symuluje naturalne ruchy głowy postaci w trakcie przemieszczania się. Dodaje to warstwę realizmu i dynamiki do rozgrywki, sprawiając, że gracz czuje się bardziej zaangażowany w wirtualny świat.

### 4. System Broni

Ważnym elementem gry było odwzorowanie systemu broni, który odgrywa kluczową rolę w rozgrywce. Nasza implementacja obejmuje wszystkie klasyczne bronie z *Doom*’a, wraz z ich unikalnymi animacjami i efektami dźwiękowymi. Zapewnia to nie tylko estetyczne, ale także funkcjonalne wrażenia, które są bliskie oryginałowi.

### 5. System Debugowania

W celu usprawnienia procesu rozwoju, zaimplementowaliśmy zaawansowany **system debugowania**, który umożliwia:

* **Podgląd segmentów do renderu:** Wizualizowanie, które segmenty mapy są aktualnie renderowane, co pozwala na łatwe identyfikowanie potencjalnych problemów.
* **Podgląd kolejności rysowania segmentów:** Umożliwia sprawdzenie, w jakiej kolejności segmenty mapy są renderowane, co jest kluczowe przy optymalizacji.
* **Podgląd ostrosłupa widzenia:** Narzędzie to wizualizuje pole widzenia gracza, co pomaga w optymalizacji algorytmów związanych z wykluczaniem niewidocznych obiektów.

### 6. Sztuczna Inteligencja (AI)

Zaimplementowaliśmy również podstawowy system sztucznej inteligencji, który odpowiada za kontrolowanie zachowania wrogów. AI potrafi reagować na obecność gracza, poruszać się po mapie, a także prowadzić ataki, co dodaje grywalności i wyzwania.

### 7. System Walki i Zabijania Wrogów

Nasz system walki oparty na strzelaniu został zaimplementowany z uwzględnieniem oryginalnych mechanik *Doom*’a. Wrogowie mogą być eliminowani przez gracza, co prowadzi do dynamicznych starć, które są esencją rozgrywki w *Doom*’ie.

## Technologia i Narzędzia

Projekt został zaimplementowany w języku **C++**, co pozwoliło na osiągnięcie wysokiej wydajności i pełnej kontroli nad zasobami systemowymi. Do zarządzania procesem budowania wykorzystaliśmy **CMake**, co zapewniło nam elastyczność i możliwość pracy na różnych środowiskach programistycznych, od Visual Studio na Windowsie, po GCC na Linuxie. Każdy członek zespołu mógł wybrać środowisko, które najlepiej odpowiadało jego preferencjom, co znacznie ułatwiło współpracę.

# Teoretyczne wprowadzenie

## Format Pliku WAD w Grze Doom

### Przegląd

Plik WAD (ang. *Where’s All the Data?*), jest głównym formatem plików używanym w grze *Doom* oraz jej powiązanych tytułach do przechowywania danych gry. Pliki WAD działają jako kontenery, które zawierają różne rodzaje danych potrzebnych do działania gry, takie jak mapy, tekstury, efekty dźwiękowe i inne zasoby.

Pliki WAD dzielą się na dwie główne kategorie:

* **IWAD (Internal WAD):** Zawiera podstawowe dane gry, niezbędne do jej uruchomienia. Przykładami są doom.wad i doom2.wad.
* **PWAD (Patch WAD):** Używane do modyfikacji i dodatkowej zawartości tworzonej przez użytkowników, takich jak nowe poziomy czy niestandardowe grafiki.

### Struktura Pliku WAD

Plik WAD składa się z trzech głównych komponentów:

#### 1. **Nagłówek**

Nagłówek to pierwsza część pliku WAD i zawiera następujące pola:

* **Typ (4 bajty):** Określa, czy plik WAD jest IWAD-em, czy PWAD-em.
* **Liczba lumpów (4 bajty):** Wskazuje łączną liczbę lumpów (wpisów danych) zawartych w pliku WAD.
* **Offset katalogu (4 bajty):** Określa offset w bajtach, w którym zaczyna się katalog.

#### 2. **Katalog**

Katalog to tabela zawartości pliku WAD. Zaczyna się w miejscu określonym przez offset w nagłówku i zawiera jeden wpis dla każdego lumpa. Każdy wpis katalogu ma 16 bajtów i zawiera:

* **Offset (4 bajty):** Offset w pliku, w którym zaczynają się dane lumpa.
* **Rozmiar (4 bajty):** Rozmiar lumpa w bajtach.
* **Nazwa (8 znaków):** ASCII string (w razie potrzeby uzupełniony zerami), który nadaje nazwę lumpowi.

#### 3. **Lumpy**

Lumpy w pliku WAD są podstawowymi jednostkami danych, które przechowują różne elementy niezbędne do funkcjonowania gry *Doom*. Poniżej przedstawiam rozwinięty opis każdego z typowych lumpów znajdujących się w pliku WAD:

##### 1. **THINGS**

* **Opis:** Lump zawierający informacje o wszystkich obiektach umieszczonych na mapie, takich jak pozycje startowe graczy, przeciwnicy, przedmioty do zebrania itp.
* **Zawartość:** Każdy wpis w tym lumpie zawiera współrzędne X i Y obiektu, kąt startowy (jeśli dotyczy) oraz flagi, które określają właściwości obiektu (np. widoczność na poziomach trudności).

##### 2. **LINEDEFS**

* **Opis:** Lump definiujący linie, które tworzą struktury mapy, takie jak ściany.
* **Zawartość:** Zawiera informacje o początkowych i końcowych wierzchołkach linii, typ linii, tagi i powiązane SIDEDEFS (jeśli występują).

##### 3. **SIDEDEFS**

* **Opis:** Lump zawierający dane o teksturach przypisanych do boków linii (LINEDEFS) oraz ich wyrównanie.
* **Zawartość:** Zawiera tekstury, przesunięcia w osi X i Y oraz sektor, do którego dany SIDEDDEF należy.

##### 4. **VERTEXES**

* **Opis:** Lump zawierający współrzędne wszystkich wierzchołków użytych na mapie.
* **Zawartość:** Każdy wpis zawiera współrzędne X i Y danego wierzchołka.

##### 5. **SEGS**

* **Opis:** Lump definiujący segmenty linii (SEGS), które są używane do podziału mapy na podsektory.
* **Zawartość:** Informacje o początkowych i końcowych wierzchołkach segmentów, ich powiązaniu z LINEDEFS oraz SIDEDDEFS.

##### 6. **SSECTORS**

* **Opis:** Lump zawierający dane o podsektorach, które są grupami SEGS tworzących sektory mapy.
* **Zawartość:** Każdy wpis opisuje liczbę segmentów w podsektorze i odwołania do odpowiednich SEGS.

##### 7. **NODES**

* **Opis:** Lump zawierający drzewo BSP (Binary Space Partitioning), które pomaga w szybkim renderowaniu mapy, dzieląc ją na mniejsze podsektory.
* **Zawartość:** Drzewo BSP jest używane do optymalizacji widoku renderowanej sceny, określając, które części mapy są widoczne z danej pozycji gracza.

##### 8. **SECTORS**

* **Opis:** Lump definiujący właściwości sektorów mapy, takich jak wysokość podłogi i sufitu, poziom oświetlenia, tekstury oraz tagi.
* **Zawartość:** Każdy sektor jest opisany poprzez wysokości podłogi i sufitu, tekstury na powierzchniach oraz poziom oświetlenia.

##### 9. **REJECT**

* **Opis:** Opcjonalny lump, który zawiera dane optymalizujące działanie gry poprzez określenie, które sektory są widoczne z danego sektora.
* **Zawartość:** Dane w tym lumpie pozwalają na pomijanie obliczeń dotyczących wrogów znajdujących się w niewidocznych sektorach, co przyspiesza działanie gry.

##### 10. **BLOCKMAP**

* **Opis:** Lump używany do detekcji kolizji między obiektami na mapie.
* **Zawartość:** Zawiera siatkę danych, która pozwala na szybkie określenie, które obiekty są w bezpośrednim kontakcie lub kolidują ze sobą.

##### 11. **PLAYPAL**

* **Opis:** Lump zawierający zestawy palet kolorów używanych w grze.
* **Zawartość:** Każda paleta składa się z 256 kolorów, gdzie każdy kolor jest reprezentowany przez trzy wartości (RGB).

##### 12. **COLORMAP**

* **Opis:** Lump mapujący wartości pikseli w celu dostosowania jasności, co pozwala na tworzenie efektów takich jak oświetlenie dynamiczne.
* **Zawartość:** Zawiera mapy kolorów, które są używane w celu redukcji jasności pikseli w różnych warunkach oświetleniowych.

##### 13. **TEXTURE1 i TEXTURE2**

* **Opis:** Lumpy definiujące tekstury ścian używane na mapach.
* **Zawartość:** Zawierają informacje o nazwach tekstur oraz ich rozmiarach, które są używane do renderowania powierzchni ścian na mapach.

##### 14. **PNAMES**

* **Opis:** Lump zawierający listę nazw wszystkich elementów składających się na tekstury zdefiniowane w TEXTURE1 i TEXTURE2.
* **Zawartość:** Każda nazwa w tym lumpie odwołuje się do części tekstur, które są później składane w większe, złożone tekstury.

##### Specjalne Lumpy i Markery

Niektóre lumpy pełnią specyficzne role, takie jak określenie, gdzie zaczynają się i kończą określone rodzaje danych: - **Markery:** Przykłady to S\_START i S\_END dla sprite’ów oraz F\_START i F\_END dla tekstur podłogi i sufitu (flats). - **Wirtualne Lumpy:** Są to symbole zastępcze, które mogą mieć rozmiar zero bajtów, często używane do oznaczania początku lub końca sekcji danych.

## Zasada Działania Algorytmu Binary Space Partitioning (BSP) w Grze *Doom*

### Wprowadzenie do BSP

Algorytm **Binary Space Partitioning** (BSP) został zaprojektowany, aby efektywnie zarządzać danymi dotyczącymi przestrzeni 3D, które są następnie używane do renderowania sceny. W kontekście gry *Doom*, BSP jest kluczowym narzędziem do organizowania i optymalizowania geometrii mapy w taki sposób, aby możliwe było szybkie obliczanie, które części sceny są widoczne z określonej pozycji gracza.

### Podstawowe Założenia BSP

BSP działa na zasadzie **rekursywnego podziału przestrzeni**. Cała mapa jest podzielona na mniejsze regiony, aż do momentu, gdy każdy region jest wystarczająco prosty, aby łatwo było określić jego widoczność. Wynikiem tego procesu jest struktura danych znana jako **drzewo BSP**.

### Budowa Drzewa BSP

Drzewo BSP jest strukturą danych złożoną z **węzłów** i **liści**:

* **Węzły wewnętrzne (Internal Nodes):** Każdy węzeł wewnętrzny reprezentuje podział przestrzeni na dwie części za pomocą płaszczyzny podziału (ang. *splitting plane*). W kontekście *Doom*’a, płaszczyzny te są w rzeczywistości liniami w przestrzeni 2D (ponieważ mapa gry jest zasadniczo dwuwymiarowa), które dzielą przestrzeń na dwie półprzestrzenie.
* **Liście (Leaf Nodes):** Każdy liść reprezentuje obszar, który nie jest już dzielony, i zawiera informacje o geometrii tego regionu. W grze *Doom* liście odpowiadają podsektorom, które są najmniejszymi jednostkami przestrzennymi mapy.

### Proces Tworzenia Drzewa BSP

1. **Wybór Płaszczyzny Podziału:** Proces zaczyna się od wyboru odpowiedniej linii podziału. Celem jest wybór takiej linii, która minimalizuje liczbę przecięć ścian, co z kolei minimalizuje złożoność podziału przestrzeni. W *Doom*’ie często używa się heurystyk, aby znaleźć optymalną linię podziału.
2. **Rekurencyjny Podział:** Przestrzeń jest dzielona na dwa podzbiory – jedną stronę linii podziału i drugą stronę. Następnie proces jest powtarzany rekurencyjnie dla każdej z tych podprzestrzeni, aż przestrzeń zostanie podzielona na wystarczająco małe regiony, które można uznać za liście drzewa.
3. **Generowanie Liści:** Gdy przestrzeń nie może być już dalej dzielona, generowany jest liść drzewa, który zawiera informacje o geometrii danego obszaru (np. które ściany i sektory znajdują się w tej części mapy).

### Struktura Drzewa BSP w Doom

* **Węzły:** Każdy węzeł przechowuje informacje o płaszczyźnie podziału oraz wskaźniki do swoich dzieci: lewego i prawego poddrzewa.
* **Liście:** Liście przechowują informacje o podsektorach, czyli regionach mapy, które nie zostały dalej podzielone. Każdy liść zawiera listę segmentów linii (*segs*), które tworzą podsektor.

### Obsługa Drzewa BSP

Podczas eksploracji mapy, algorytm BSP jest używany do szybkiego określenia, które podsektory są widoczne z danej pozycji gracza. Silnik gry przechodzi przez drzewo BSP, zaczynając od korzenia, i porównuje aktualną pozycję gracza z płaszczyznami podziału węzłów, aby ustalić, które liście (podsektory) są potencjalnie widoczne. Pozwala to na szybkie odrzucenie niewidocznych części mapy, co znacząco przyspiesza rendering i poprawia wydajność gry. ### Wykorzystanie BSP do Renderowania w grze *Doom*

## Wprowadzenie do Renderowania z użyciem BSP

Wykorzystanie algorytmu Binary Space Partitioning (BSP) w procesie renderowania było kluczowe dla wydajności gry *Doom*. Dzięki zastosowaniu drzewa BSP, silnik gry mógł szybko określić, które części sceny są widoczne z aktualnej pozycji gracza, co znacząco przyspieszało proces renderowania.

### Proces renderowania z użyciem BSP

1. **Przechodzenie przez Drzewo BSP:**
   * Gdy gracz przemieszcza się po mapie, silnik gry przechodzi przez drzewo BSP, zaczynając od korzenia.
   * W każdym węźle BSP, silnik porównuje pozycję gracza z płaszczyzną podziału tego węzła.
   * Na podstawie tej analizy, silnik decyduje, które poddrzewo (czyli która część przestrzeni) znajduje się “przed” gracza, a które “za” nim.
2. **Renderowanie Podsektorów:**
   * Podsektory, które znajdują się “przed” graczem (z punktu widzenia płaszczyzny podziału), są renderowane jako pierwsze.
   * Następnie, silnik przechodzi do renderowania kolejnych podsektorów, zgodnie z kolejnością ustaloną przez drzewo BSP.
   * W tym procesie silnik może od razu wykluczyć części sceny, które są niewidoczne z aktualnej pozycji gracza, ponieważ znajdują się “za” gracza w hierarchii drzewa BSP.
3. **Odrzucanie Niewidocznych Obszarów:**
   * Dzięki strukturze drzewa BSP, obszary mapy, które są zasłonięte przez inne obiekty (np. ściany) mogą być szybko odrzucone, co minimalizuje liczbę obiektów, które muszą być przetwarzane i renderowane.
   * To pozwala silnikowi skupić się tylko na tych fragmentach mapy, które są faktycznie widoczne, co znacznie przyspiesza renderowanie.

### Dlaczego renderowanie grafiki z wykorzystaniem BSP jest szybkie?

1. **Efektywne Odrzucanie Niewidocznych Obszarów:**
   * Drzewo BSP pozwala na szybkie odrzucenie części sceny, które nie są widoczne. Zamiast przetwarzać całą mapę, silnik gry analizuje tylko te obszary, które są rzeczywiście widoczne z aktualnej pozycji gracza.
2. **Zoptymalizowane Przetwarzanie Geometrii:**
   * Dzięki BSP, silnik gry przetwarza geometrię w odpowiedniej kolejności, co zmniejsza konieczność przerysowywania obiektów. W przypadku *Doom*’a, który działał na sprzęcie o ograniczonych zasobach, taka optymalizacja była kluczowa.
3. **Rekurencyjne Renderowanie:**
   * Rekurencyjna natura algorytmu BSP umożliwia łatwe przetwarzanie hierarchiczne, gdzie najpierw renderowane są obiekty bliższe graczowi, a następnie te dalsze. To zapewnia, że bliższe obiekty zawsze są renderowane na wierzchu dalszych, co eliminuje problemy z widocznością (ang. *overdraw*).

## Wykorzystanie BSP do Szybkiej Detekcji Kolizji

### Wprowadzenie

Algorytm Binary Space Partitioning (BSP) może być również wykorzystany do efektywnej detekcji kolizji w grach wideo, takich jak *Doom*. Dzięki swojej strukturze, BSP pozwala na szybkie ograniczenie obszaru, w którym może wystąpić kolizja, co znacznie przyspiesza proces wykrywania zderzeń.

### Zasada Działania

1. **Podział Przestrzeni:**
   * Wstępnie podzielona przestrzeń za pomocą drzewa BSP pozwala na szybkie określenie, w którym podregionie znajduje się dany obiekt lub obiekty. Dzięki temu, zamiast sprawdzać kolizje w całej przestrzeni gry, algorytm koncentruje się na konkretnym regionie.
2. **Rekursywne Przeszukiwanie:**
   * Proces wykrywania kolizji zaczyna się od korzenia drzewa BSP. Każdy węzeł określa, po której stronie płaszczyzny podziału znajduje się obiekt.
   * Silnik porusza się w dół drzewa, sprawdzając, które poddrzewo zawiera obiekt i jego potencjalnych kolidujących sąsiadów.
3. **Redukcja Liczby Porównań:**
   * Zamiast sprawdzać kolizje ze wszystkimi obiektami na mapie, drzewo BSP pozwala na szybkie ograniczenie liczby porównań tylko do tych obiektów, które znajdują się w tym samym lub sąsiednich podsektorach.

### Proces Detekcji Kolizji

1. **Lokalizacja Obiektu:**
   * Obiekt, którego kolizje chcemy wykryć, jest lokalizowany w drzewie BSP poprzez przechodzenie przez kolejne węzły, aż do dotarcia do odpowiedniego liścia.
2. **Sprawdzanie Sąsiednich Podsektorów:**
   * Po zlokalizowaniu obiektu w odpowiednim liściu, sprawdzane są kolizje tylko z obiektami w tym samym liściu oraz ewentualnie w sąsiadujących liściach, jeśli obiekt przecina granice podsektora.
3. **Efektywne Przetwarzanie:**
   * Dzięki temu, że algorytm BSP skupia się tylko na małym podzbiorze przestrzeni, sprawdzenie kolizji odbywa się szybciej, co jest kluczowe w grach działających w czasie rzeczywistym, takich jak *Doom*.

### Zalety Wykorzystania BSP do Detekcji Kolizji

* **Redukcja Złożoności:** BSP znacząco zmniejsza złożoność problemu, ponieważ kolizje są sprawdzane tylko w niewielkich, lokalnych regionach zamiast w całej przestrzeni gry.
* **Szybkość:** Dzięki efektywnemu podziałowi przestrzeni, algorytm BSP pozwala na szybkie wykluczenie dużej liczby potencjalnych kolizji, co zwiększa ogólną wydajność gry.
* **Skalowalność:** Algorytm doskonale sprawdza się zarówno w mniejszych, jak i większych scenach, co czyni go uniwersalnym rozwiązaniem do detekcji kolizji w grach 3D.

## Zasada Działania Wzorca Entity Component System (ECS)

**Entity Component System (ECS)** to wzorzec projektowy szeroko stosowany w projektowaniu systemów gier oraz innych aplikacji, które wymagają elastycznego zarządzania obiektami i ich zachowaniami. ECS oferuje modularne podejście, które ułatwia dodawanie, modyfikowanie oraz zarządzanie różnorodnymi funkcjonalnościami obiektów w grze.

### Podstawowe Elementy ECS

ECS składa się z trzech głównych komponentów:

1. **Encje (Entities)**
2. **Komponenty (Components)**
3. **Systemy (Systems)**

Każdy z tych elementów odgrywa kluczową rolę w architekturze ECS.

#### 1. Encje (Entities)

* **Opis:** Encje są abstrakcyjnymi identyfikatorami, które reprezentują obiekty w świecie gry. Mogą to być postacie graczy, przedmioty, wrogowie, pociski, i inne.
* **Struktura:** Encja sama w sobie nie zawiera żadnej logiki ani danych. Jest to jedynie unikalny identyfikator, zazwyczaj liczba całkowita lub wskaźnik.
* **Rola:** Rolą encji jest łączenie komponentów, które definiują jej właściwości i zachowanie.

#### 2. Komponenty (Components)

* **Opis:** Komponenty przechowują dane dotyczące właściwości lub stanu encji. Są to podstawowe “cegiełki”, które definiują cechy i możliwości encji.
* **Struktura:** Każdy komponent zawiera wyłącznie dane, bez logiki. Może to być np. pozycja w świecie (PositionComponent), prędkość (VelocityComponent), zdrowie (HealthComponent), itp.
* **Rola:** Komponenty są przypisywane do encji, definiując w ten sposób, jakie cechy i możliwości encja posiada.

#### 3. Systemy (Systems)

* **Opis:** Systemy zawierają logikę, która operuje na komponentach. Systemy przetwarzają dane zawarte w komponentach przypisanych do encji, realizując określone zadania.
* **Struktura:** Każdy system jest odpowiedzialny za określony aspekt działania gry, np. system fizyki (PhysicsSystem), system renderowania (RenderingSystem), system AI (AISystem).
* **Rola:** Systemy iterują po encjach, które mają przypisane określone zestawy komponentów, i wykonują na nich odpowiednie operacje.

### Przykład Działania ECS

Załóżmy, że mamy grę, w której występuje postać gracza i przeciwnik. Obie te encje mogą być reprezentowane w systemie ECS w następujący sposób:

1. **Encje:** PlayerEntity, EnemyEntity
2. **Komponenty:**
   * PositionComponent: przechowuje współrzędne x, y postaci.
   * VelocityComponent: przechowuje prędkość poruszania się.
   * HealthComponent: przechowuje stan zdrowia.
   * RenderComponent: przechowuje dane potrzebne do renderowania.
3. **Systemy:**
   * MovementSystem: przetwarza PositionComponent i VelocityComponent, aktualizując pozycję encji.
   * HealthSystem: przetwarza HealthComponent, sprawdzając stan zdrowia encji.
   * RenderSystem: przetwarza RenderComponent, rysując encję na ekranie.

#### Przebieg Operacji

1. **Tworzenie encji:**
   * Tworzymy encję PlayerEntity i przypisujemy jej komponenty PositionComponent, VelocityComponent, HealthComponent, i RenderComponent.
   * Tworzymy encję EnemyEntity i przypisujemy jej komponenty PositionComponent, HealthComponent, i RenderComponent.
2. **Systemy działają na encjach:**
   * MovementSystem iteruje po wszystkich encjach, które posiadają PositionComponent i VelocityComponent, aktualizując ich pozycję.
   * HealthSystem iteruje po encjach z HealthComponent, sprawdzając, czy encje nie straciły całego zdrowia.
   * RenderSystem iteruje po encjach z RenderComponent, renderując je na ekranie.

#### Zaletą takiego podejścia jest modularność i elastyczność:

* **Łatwość rozszerzania:** Dodawanie nowych komponentów lub systemów nie wpływa na istniejącą architekturę. Możemy na przykład łatwo dodać AIComponent i AISystem, aby wzbogacić przeciwników o sztuczną inteligencję.
* **Reużywalność:** Komponenty mogą być używane przez różne encje, co zwiększa reużywalność kodu.
* **Separacja danych i logiki:** Dzięki oddzieleniu danych (komponentów) od logiki (systemów), łatwiej jest zarządzać złożonością kodu.

# System Współpracy w Grupie

W naszym projekcie zastosowaliśmy kilka narzędzi i metodologii, które wspierały naszą efektywną współpracę i zarządzanie zadaniami. Dzięki odpowiedniemu podziałowi zadań oraz wykorzystaniu sprawdzonych narzędzi mogliśmy bezproblemowo realizować kolejne etapy projektu, minimalizując konflikty oraz zapewniając spójność pracy. W niniejszym opisie przedstawiamy szczegółowo nasze podejście.

## Zarządzanie Zadaniami

### Backlog na Discordzie

Do zarządzania zadaniami wykorzystaliśmy prostą listę zadań (Backlog), którą zamieszczaliśmy na naszym kanale na Discordzie. Backlog jest podstawowym narzędziem w metodykach zwinnych (Agile) i służy do przechowywania listy wszystkich zadań, które muszą zostać wykonane w projekcie. Nasza lista zadań zawierała zarówno zadania bieżące, jak i te planowane na przyszłość, a także zadania już zakończone.

**Zalety korzystania z Backlogu na Discordzie:**

* **Centralizacja informacji:** Wszystkie zadania były dostępne dla całego zespołu w jednym miejscu.
* **Elastyczność:** Każdy z członków zespołu mógł swobodnie dodawać nowe zadania lub aktualizować istniejące, co umożliwiało szybkie reagowanie na zmieniające się potrzeby projektu.
* **Przejrzystość:** Wszystkie aktualizacje i postępy były widoczne dla całego zespołu, co zapewniało transparentność działań.

## Wspólna Praca nad Kodem

### Git i GitFlow

Do pracy nad kodem korzystaliśmy z systemu kontroli wersji Git oraz modelu rozwoju GitFlow. **Git** to system kontroli wersji, który pozwala na śledzenie zmian w kodzie źródłowym oraz na zarządzanie pracą nad projektem w zespole. **GitFlow** jest to konkretne podejście do zarządzania gałęziami (branchami) w Git, które ułatwia organizację procesu rozwoju oprogramowania.

**Podstawowe założenia GitFlow:**

* **Gałąź master:** Służy do przechowywania stabilnej wersji projektu, która jest gotowa do wydania.
* **Gałąź develop:** To główna gałąź, na której prowadzony jest bieżący rozwój projektu. Wszystkie nowe funkcjonalności trafiają do niej po zakończeniu prac.
* **Feature branches:** Każda nowa funkcjonalność (feature) jest rozwijana na osobnej gałęzi, wywodzącej się z gałęzi develop. Po zakończeniu prac nad funkcjonalnością gałąź ta jest łączona (merge) z gałęzią develop.
* **Release branches:** Przed wydaniem nowej wersji projektu, tworzona jest gałąź release, która pozwala na przygotowanie wersji do wydania, w tym na testy i wprowadzenie drobnych poprawek.
* **Hotfix branches:** Jeśli w stabilnej wersji na gałęzi master pojawią się błędy wymagające natychmiastowej naprawy, tworzy się gałąź hotfix, która jest później łączona zarówno z master, jak i develop.

**Zalety podejścia GitFlow:**

* **Brak konfliktów:** Dzięki wyraźnemu podziałowi na gałęzie, każdy członek zespołu mógł pracować nad swoimi modułami bez obawy o konflikty w kodzie.
* **Bezpieczeństwo kodu:** Główne gałęzie (master i develop) były chronione przed bezpośrednimi zmianami, co zapobiegało przypadkowym błędom.
* **Łatwość wydawania nowych wersji:** Proces przygotowania nowej wersji był zorganizowany i przewidywalny, dzięki czemu mogliśmy łatwo zarządzać wydaniami.

## Współpraca i Komunikacja

### Komunikacja Asynchroniczna

W naszym zespole dużą wagę przykładaliśmy do komunikacji asynchronicznej, co oznaczało, że każdy mógł pracować w swoim tempie, a decyzje i informacje były udostępniane w taki sposób, aby każdy miał do nich dostęp wtedy, gdy to dla niego wygodne. Discord oraz Git odegrały tu kluczową rolę, umożliwiając przechowywanie i przeglądanie historii komunikacji oraz zmian w kodzie w dowolnym momencie.

**Zalety komunikacji asynchronicznej:**

* **Elastyczność:** Członkowie zespołu mogli pracować w różnych godzinach, dostosowując harmonogram pracy do swoich potrzeb.
* **Dokumentacja:** Każda rozmowa i każda zmiana w kodzie były automatycznie dokumentowane, co ułatwiało powrót do wcześniejszych ustaleń i rozwiązań.

## Wykorzystanie CMake

### Elastyczność i Indywidualność Środowiska Programistycznego

W naszym projekcie skorzystaliśmy z CMake jako narzędzia do budowania, co pozwoliło nam na dużą elastyczność w doborze środowiska programistycznego. CMake jest narzędziem open-source, które generuje pliki konfiguracyjne dla różnych systemów budowania (takich jak Makefile czy projekty Visual Studio), w oparciu o wspólną konfigurację. Dzięki CMake, członkowie zespołu mogli pracować na różnych systemach operacyjnych i korzystać z różnych kompilatorów.

**Różnorodność środowisk programistycznych:**

* **Windows + Visual Studio:** Część zespołu korzystała z Windowsa i Visual Studio, co ułatwiało integrację z innymi narzędziami Microsoftu oraz zapewniało wygodne środowisko z zaawansowanym debugerem.
* **Windows + GCC:** Inni członkowie zespołu preferowali korzystanie z Windowsa w połączeniu z GCC, co pozwalało im na korzystanie z narzędzi typowych dla ekosystemu GNU.
* **Linux:** Część zespołu pracowała na systemach Linux, korzystając z narzędzi takich jak GCC czy Clang. Dzięki CMake mogli oni bez problemu zbudować projekt na swoich systemach.

**Zalety korzystania z CMake:**

* **Cross-platformowość:** CMake umożliwia budowanie projektu na różnych platformach bez potrzeby modyfikowania kodu, co jest szczególnie ważne w zespołach, gdzie członkowie pracują na różnych systemach operacyjnych.
* **Elastyczność:** CMake umożliwia łatwą konfigurację projektu, dostosowanie flag kompilatora, zależności bibliotek i innych ustawień specyficznych dla środowiska.
* **Modularność:** Dzięki CMake mogliśmy efektywnie zarządzać modularną strukturą naszego projektu, co ułatwiało integrację różnych części kodu rozwijanych przez różne osoby.

# Struktura Katalogów w Naszym Projekcie

Poniżej znajduje się opis struktury katalogów naszego projektu, który został zaprojektowany z myślą o przejrzystości i modularności, co ułatwia pracę nad kodem, zarządzanie zasobami oraz organizację dokumentacji.

## Główna Struktura Katalogów

### 1. Pliki Konfiguracji

* **CMakeLists.txt**
  + Główny plik konfiguracyjny dla CMake, zawierający definicje kompilacji projektu. Zawiera instrukcje dotyczące tego, jakie pliki źródłowe i nagłówkowe powinny być kompilowane, jakie biblioteki są potrzebne oraz inne parametry konfiguracyjne.
* **CMakeSettings.json**
  + Plik ustawień CMake, używany głównie przez środowiska programistyczne, takie jak Visual Studio, do definiowania różnych konfiguracji budowania, np. dla wersji debug i release.
* **Makefile**
  + Plik używany przez make do budowania projektu. Jest to alternatywna metoda budowania, zwłaszcza na systemach Linux, gdzie make jest powszechnie używany.
* **README.md**
  + Plik dokumentacji, zawierający ogólne informacje o projekcie, instrukcje dotyczące kompilacji, uruchomienia oraz inne ważne informacje dla deweloperów i użytkowników.

### 2. Katalog build

* **CMakeCache.txt, CMakeFiles, cmake\_install.cmake, compile\_commands.json**
  + Pliki wygenerowane przez CMake, przechowujące informacje o konfiguracji projektu, kompilowanych plikach oraz komendach kompilacji.
* **bin/**
  + Katalog, w którym znajdują się skompilowane pliki binarne, czyli wykonane pliki programu.
* **data/**
  + W tej podkategorii przechowywane są zasoby gry wykorzystywane w trakcie budowania i testowania projektu.
* **libs/**
  + Zawiera skompilowane biblioteki wykorzystywane przez projekt.

### 3. Katalog data

* **assets/**
  + Zasoby gry takie jak modele 3D, tekstury, dźwięki, które są używane bezpośrednio w grze.
* **font/**
  + Pliki czcionek używane w grze, np. do wyświetlania tekstu w menu lub HUDzie.
* **maps/**
  + Pliki map, które definiują poziomy gry. Są to zwykle pliki .WAD lub inne formaty map, które są ładowane podczas gry.
* **textures/**
  + Tekstury używane w grze do nakładania na powierzchnie obiektów, ścian, podłóg itp.

### 4. Katalog doc

* **Gra\_FPS\_wstępne\_założenia.docx, Raport.docx**
  + Dokumenty związane z projektem, zawierające założenia projektowe oraz raport z postępów prac. Przechowują one dokumentację techniczną i założenia projektowe, które były fundamentem prac.

### 5. Katalog include

* **core/**
  + Zawiera pliki nagłówkowe związane z rdzeniem silnika gry, czyli podstawowe klasy i funkcje, które są wspólne dla całego projektu.
* **game/**
  + Pliki nagłówkowe specyficzne dla logiki gry, np. systemy zarządzające AI, bronią, czy interakcjami w grze.

### 6. Katalog libs

* **ext/**
  + Zewnętrzne biblioteki używane w projekcie. Mogą to być biblioteki open-source lub inne zależności, które są dołączane do projektu.

### 7. Katalog src

* **core/**
  + Zawiera pliki źródłowe implementujące podstawowe funkcjonalności silnika gry.
* **game/**
  + Pliki źródłowe implementujące logikę gry, takie jak mechaniki rozgrywki, AI, obsługa gracza, itp.
* **main.cpp**
  + Główny plik źródłowy, który zawiera punkt wejścia do aplikacji. To tutaj inicjalizowane są podstawowe systemy gry i rozpoczyna się główna pętla gry.

# Zasada Działania WAD Loader’a

## Wprowadzenie

W naszym projekcie implementacja WAD Loader’a odgrywa kluczową rolę w ładowaniu i interpretowaniu danych z plików WAD, które są integralną częścią gry *Doom*. Dzięki tej implementacji, jesteśmy w stanie załadować każdy poziom z oryginalnej gry i odwzorować go w naszym nowoczesnym silniku, który bazuje na klasycznych algorytmach renderowania i detekcji kolizji.

## Struktura WAD Loader’a

### Plik WADLoader.h

Plik nagłówkowy WADLoader.h definiuje podstawowe struktury danych oraz funkcje, które są niezbędne do ładowania plików WAD:

* **Struktury danych:**
  + wadinfo\_t: Przechowuje podstawowe informacje o pliku WAD, takie jak liczba lumpów (danych) oraz offset do tablicy informacyjnej.
  + filelump\_t: Reprezentuje pojedynczy lump w pliku WAD, przechowując informacje o jego pozycji w pliku oraz rozmiarze.
  + wadfile\_t: Struktura przechowująca informacje o pliku WAD.
* **Klasa WADLoader:**
  + Klasa ta jest odpowiedzialna za ładowanie i interpretowanie danych z plików WAD.
  + Zawiera metody takie jak getLumpTypeFromName, która identyfikuje typ lumpa na podstawie jego nazwy, oraz loadFromFile, która ładuje poziom gry na podstawie podanego pliku WAD i nazwy mapy.
* **Szablonowa funkcja loadDefinitionsFromLump:**
  + Funkcja ta jest generyczna i wykorzystuje szablony C++ do ładowania różnych typów danych z lumpów. Dzięki temu możemy w prosty sposób załadować różne struktury danych z plików WAD, bez potrzeby pisania osobnych funkcji dla każdego typu danych.
  + Funkcja przyjmuje wskaźnik do danych (defsData) oraz rozmiar tych danych (defsSize). Następnie przekształca te dane w wektor obiektów typu T.

### Plik WADLoader.cpp

Plik implementacyjny WADLoader.cpp zawiera szczegółową implementację funkcji zdefiniowanych w WADLoader.h:

* **Konstruktor WADLoader:**
  + Konstruktor klasy WADLoader jest odpowiedzialny za inicjalizację obiektu. W obecnej implementacji nie wymaga on specjalnych działań, dlatego jest pusty.
* **Funkcja getLumpTypeFromName:**
  + Funkcja ta interpretuje nazwę lumpa i zwraca odpowiedni typ, np. eThings, eLineDefs, eVertexes, itp. Jest to istotne przy ładowaniu różnych rodzajów danych, które są kluczowe dla poprawnego odwzorowania poziomu.
* **Funkcja loadFromFile:**
  + Jest to główna funkcja odpowiedzialna za załadowanie poziomu gry. Otwiera plik WAD, odczytuje informacje o lumpach, a następnie za pomocą funkcji loadDefinitionsFromLump ładuje konkretne dane (np. informacje o rzeczach, liniach, sektorach).
  + Funkcja wykorzystuje wcześniej zdefiniowaną funkcję getLumpTypeFromName, aby określić, jaki rodzaj danych należy załadować z danego lumpa.

## Wykorzystanie Szablonów C++ w WAD Loaderze

Decyzja o wykorzystaniu szablonów C++ w funkcji loadDefinitionsFromLump była świadoma i wynikała z potrzeby elastycznego i efektywnego ładowania różnorodnych struktur danych z plików WAD.

### Dlaczego Szablony?

* **Elastyczność:** Szablony pozwalają na tworzenie jednej funkcji, która może działać na wielu typach danych. W przypadku WAD Loader’a oznacza to, że możemy ładować różne typy lumpów (np. mapsegs\_t, mapnodes\_t, mapsectors\_t) za pomocą jednej funkcji, co znacząco upraszcza kod i redukuje powtarzalność.
* **Typ Bezpieczeństwo:** Szablony w C++ oferują statyczne typowanie, co oznacza, że typy danych są sprawdzane podczas kompilacji. To zmniejsza ryzyko błędów związanych z nieprawidłowym przekształcaniem danych.
* **Oszczędność Czasu i Kodu:** Dzięki szablonom, unikamy konieczności pisania osobnych funkcji dla każdego rodzaju danych, co przyspiesza rozwój i upraszcza zarządzanie kodem.

# Zasada Działania Systemu Entity Component System (ECS)

## Wprowadzenie

W naszym projekcie implementacja systemu **Entity Component System (ECS)** stanowiła fundament architektury gry, umożliwiając modularne, elastyczne i wydajne zarządzanie logiką gry. ECS pozwala na rozdzielenie danych (komponentów) od logiki (systemów), co znacząco upraszcza zarządzanie złożonymi obiektami i interakcjami w grze. Nasza implementacja ECS została zrealizowana z wykorzystaniem języka C++ i obejmuje kluczowe komponenty, takie jak zarządzanie encjami, komponentami i systemami.

## Struktura Systemu ECS

### Plik ComponentArray.h

Plik ComponentArray.h definiuje sposób przechowywania i zarządzania komponentami dla różnych encji w grze:

* **Interfejs IComponentArray:**
  + Bazowy interfejs dla wszystkich tablic komponentów. Zawiera metodę entityDestroyed, która jest wywoływana, gdy encja jest niszczona, aby usunąć powiązane z nią komponenty.
* **Szablon ComponentArray:**
  + Szablonowa klasa ComponentArray umożliwia przechowywanie komponentów dowolnego typu. Wykorzystanie szablonów C++ pozwala na tworzenie efektywnych i typowo bezpiecznych tablic komponentów.
  + Metody takie jak insertData, removeData oraz getData zarządzają dodawaniem, usuwaniem i dostępem do komponentów przypisanych do konkretnych encji.
  + Szablony zostały wybrane, aby zminimalizować powtarzalność kodu i umożliwić łatwe dodawanie nowych typów komponentów bez potrzeby duplikacji logiki zarządzania danymi.

### Plik ComponentManager.h

Plik ComponentManager.h zarządza wszystkimi tablicami komponentów w grze:

* **Klasa ComponentManager:**
  + Zarządza kolekcją tablic komponentów, gdzie każda tablica przechowuje komponenty określonego typu.
  + Kluczowe funkcje to registerComponent (rejestracja nowego typu komponentu) oraz getComponentArray (uzyskiwanie dostępu do tablicy komponentów konkretnego typu).

### Plik Entity.h

Plik Entity.h definiuje typ encji jako alias na liczbę całkowitą, co pozwala na łatwe zarządzanie identyfikatorami encji w grze:

* **Typ Entity:**
  + Encje są po prostu identyfikatorami, które mogą być używane do wiązania różnych komponentów w systemie ECS.

### Plik EntityManager.h i EntityManager.cpp

Te pliki zarządzają tworzeniem, niszczeniem oraz zarządzaniem sygnaturami encji:

* **Klasa EntityManager:**
  + Umożliwia tworzenie nowych encji poprzez zarządzanie pulą dostępnych identyfikatorów encji.
  + Funkcje createEntity oraz destroyEntity zarządzają cyklem życia encji, natomiast setSignature i getSignature pozwalają na przypisywanie i odczytywanie sygnatur encji.

### Plik System.h i SystemManager.h

Te pliki definiują i zarządzają systemami, które przetwarzają logikę gry:

* **Klasa System:**
  + Bazowa klasa dla wszystkich systemów w grze. Każdy system zawiera listę encji, które przetwarza, oraz implementuje logikę, która działa na przypisanych encjach.
* **Klasa SystemManager:**
  + Zarządza wszystkimi systemami w grze, umożliwiając ich rejestrację oraz aktualizację na podstawie zmian sygnatur encji.
  + Odpowiada za dodawanie encji do odpowiednich systemów, kiedy ich sygnatury pasują do sygnatur systemów.

### Plik ECSManager.h

Ten plik pełni rolę centralnego menedżera ECS, łącząc zarządzanie encjami, komponentami i systemami:

* **Klasa ECSManager:**
  + Łączy wszystkie elementy systemu ECS, zapewniając centralne miejsce zarządzania encjami, komponentami i systemami.
  + Pozwala na tworzenie encji, dodawanie do nich komponentów oraz zarządzanie systemami, które przetwarzają logikę gry.

### Componenty Zaimplementowane w Game Engine

1. **GameDrawableComponent:** Komponent odpowiedzialny za rysowanie obiektów gry na ekranie.
2. **HealthComponent:** Przechowuje informacje o stanie zdrowia encji, co jest kluczowe dla mechanik związanych z walką.
3. **PlayerStateComponent:** Przechowuje stan gracza, takie jak aktywna broń, aktualny poziom życia, itp.
4. **TransformComponent:** Zawiera informacje o pozycji, rotacji i skali encji w świecie gry.
5. **WeaponComponent:** Komponent odpowiedzialny za przechowywanie informacji o broni używanej przez gracza lub przeciwnika.
6. **DamageComponent:** Przechowuje informacje o obrażeniach, które encja zadaje lub otrzymuje.

### Systemy Zaimplementowane w Game Engine

1. **PlayerControllSystem:** Odpowiada za przetwarzanie wejść od gracza i sterowanie postacią w grze.
2. **PlayerMovementSystem:** Zarządza ruchem gracza, aktualizując jego pozycję w świecie gry.
3. **EnemySystem:** Kontroluje zachowanie przeciwników, w tym ich AI oraz interakcje z graczem.
4. **WeaponSystem:** Zarządza mechanikami związanymi z używaniem broni przez gracza i przeciwników.
5. **DamageSystem:** Przetwarza informacje o obrażeniach zadawanych i otrzymywanych przez encje, aktualizując odpowiednie komponenty.
6. **EnviromentDamageSystem:** Zarządza obrażeniami, które encje otrzymują od środowiska, np. pułapki, toksyczne substancje.
7. **CollectableSystem:** Obsługuje zbieranie przedmiotów przez gracza, takie jak amunicja, zdrowie czy inne power-upy.
8. **GameRenderingSystem:** Odpowiada za renderowanie całej sceny gry, w tym wszystkich obiektów na ekranie.
9. **HUDRenderingSystem:** Renderuje interfejs użytkownika, w tym paski zdrowia, amunicję, informacje o stanie gracza.
10. **MinimapRenderingSystem:** Odpowiada za renderowanie minimapy, która pokazuje graczowi orientację na poziomie.

## Wykorzystanie Szablonów C++ w ECS

Decyzja o wykorzystaniu szablonów C++ była kluczowa dla stworzenia elastycznego i wydajnego systemu ECS:

### Dlaczego Szablony?

* **Elastyczność:** Szablony pozwalają na definiowanie uniwersalnych struktur danych i funkcji, które mogą działać na dowolnych typach. W przypadku ECS oznacza to, że możemy zarządzać różnymi typami komponentów bez potrzeby tworzenia zduplikowanego kodu dla każdego typu.
* **Typ Bezpieczeństwo:** Dzięki statycznemu typowaniu w C++, szablony zapewniają, że błędy związane z niezgodnością typów są wychwytywane na etapie kompilacji, co zmniejsza ryzyko błędów w czasie działania.
* **Oszczędność Czasu i Kodu:** Wykorzystanie szablonów pozwala na ponowne wykorzystanie tego samego kodu dla różnych typów danych, co znacznie zmniejsza ilość kodu i ułatwia jego utrzymanie.

# Zasada Działania Game Engine’a

## Wprowadzenie

Game Engine w naszym projekcie pełni rolę centralnego zarządcy, który łączy wszystkie elementy gry, takie jak zarządzanie stanami gry, renderowanie, przetwarzanie logiki gry, oraz interakcje użytkownika. Jego głównym zadaniem jest inicjalizacja, aktualizacja i zarządzanie wszystkimi aspektami rozgrywki w czasie rzeczywistym.

## Struktura Game Engine’a

### Plik GameEngine.h

Plik nagłówkowy GameEngine.h definiuje interfejs oraz kluczowe składniki Game Engine’a, które są niezbędne do jego działania:

* **Enum GameEngineState:**
  + Definiuje różne stany gry, takie jak eInitialized, eMainMenu, eInGameMenu, eGame, i eGameMinimap. Te stany pozwalają na zarządzanie różnymi fazami gry, od menu głównego po aktywną rozgrywkę.
* **Struktura InitSettings:**
  + Zawiera ustawienia inicjalizacyjne, takie jak rozdzielczość okna, tytuł, tryb pełnoekranowy, poziom trudności, oraz początkowy stan gry. Ta struktura pozwala na elastyczne dostosowanie parametrów uruchomienia gry.
* **Klasa GameEngine:**
  + Jest to główna klasa zarządzająca, która kontroluje cały cykl życia gry.
  + Kluczowe funkcje:
    - processEvents(): Odpowiada za przetwarzanie zdarzeń użytkownika, takich jak kliknięcia myszką czy naciśnięcia klawiszy.
    - run(): Główna pętla gry, która wywołuje aktualizacje i renderowanie w regularnych odstępach czasu.
    - init(): Inicjalizuje wszystkie komponenty, systemy oraz inne elementy niezbędne do uruchomienia gry.
    - update(sf::Time deltaTime): Aktualizuje stan gry na podstawie upływu czasu, synchronizując logikę gry z działaniami gracza oraz AI.
  + Klasa ta zarządza również komponentami, systemami oraz stanem gry.

### Plik GameEngine.cpp

Plik implementacyjny GameEngine.cpp zawiera szczegółową implementację funkcji zdefiniowanych w GameEngine.h:

* **Funkcja setupComponents():**
  + Odpowiada za inicjalizację wszystkich komponentów, które są wymagane w grze. Dzięki temu każdy komponent zostaje przypisany do odpowiednich encji na początku gry.
* **Funkcja setupSystems():**
  + Rejestruje oraz inicjalizuje wszystkie systemy, które są odpowiedzialne za logikę gry. Każdy system jest przypisany do encji, które spełniają określone kryteria (sygnatury), co pozwala na efektywne przetwarzanie logiki gry.
* **Funkcja handleTabToggle():**
  + Zarządza przełączaniem między widokiem gry a minimapą za pomocą klawisza Tab. Zmienia stan gry na eGameMinimap lub z powrotem na eGame, w zależności od aktualnego stanu.

## Rola Game Engine’a

Game Engine w naszym projekcie pełni funkcję centralnego zarządcy, który integruje wszystkie komponenty gry oraz zarządza nimi w czasie rzeczywistym. Jest odpowiedzialny za:

* **Zarządzanie Stanem Gry:** Umożliwia płynne przechodzenie między różnymi stanami gry, takimi jak menu główne, aktywna rozgrywka czy widok minimapy.
* **Renderowanie:** Kontroluje proces renderowania, w tym aktualizowanie obrazu gry na ekranie, obsługę rysowania HUD-u (Head-Up Display) oraz innych elementów graficznych.
* **Przetwarzanie Logiki Gry:** Umożliwia przetwarzanie logiki gry, w tym interakcji gracza, zachowań przeciwników oraz systemów związanych z walką, poruszaniem się postaci i innymi aspektami rozgrywki.
* **Interakcje Użytkownika:** Odbiera i przetwarza wejścia od użytkownika, takie jak ruchy myszy, naciśnięcia klawiszy, które mają bezpośredni wpływ na stan gry. # Zasada Działania Core’owych Elementów Silnika

## Wprowadzenie

Core’owe elementy silnika stanowią fundament, na którym zbudowane są wszystkie inne systemy i funkcjonalności naszego projektu. Obejmują one kluczowe funkcje, struktury danych, zarządzanie poziomami gry, logowanie, a także różnorodne narzędzia pomocnicze. Te elementy zapewniają stabilność, spójność oraz wydajność działania całego systemu, pełniąc funkcję fundamentu, na którym opiera się cała logika gry.

## Struktura Core’owych Elementów

### Plik core.h

Plik core.h zawiera definicje podstawowych stałych, makr oraz funkcji pomocniczych, które są używane w całym projekcie:

* **Stałe i Makra:**
  + M\_PI: Stała definiująca wartość liczby pi.
  + FOV, H\_FOV, FOV\_ENEMIES, FOV\_LENGTH: Stałe definiujące pole widzenia (field of view) oraz inne parametry związane z renderowaniem i detekcją.
  + WIDTH, HEIGHT: Definiują rozdzielczość ekranu.
  + DEG2RAD(x), RAD2DEG(x): Makra konwertujące kąty między stopniami a radianami.
* **Funkcje pomocnicze:**
  + getRandomColor(): Funkcja generująca losowy kolor, która może być używana do debugowania lub innych celów wizualnych.
* **Struktura DebugSettings:**
  + Przechowuje ustawienia debugowania, takie jak wyświetlanie pola widzenia, widocznych węzłów czy segmentów.

### Plik GameLevel.h

Plik GameLevel.h zarządza wszystkimi aspektami związanymi z poziomami gry:

* **Struktury danych:**
  + color\_t, name\_t, patchheader\_t: Struktury reprezentujące dane dotyczące kolorów, nazw oraz nagłówków plików graficznych.
  + maptexture\_t, maptexturelump\_t, pnames\_t: Struktury przechowujące dane dotyczące tekstur używanych na poziomach gry.
  + mapthings\_t: Struktura reprezentująca obiekty umieszczone na mapie (np. przeciwników, przedmioty do zebrania).
* **Funkcjonalności:**
  + Plik ten pełni kluczową rolę w zarządzaniu zasobami związanymi z poziomami gry, umożliwiając ładowanie, interpretację oraz manipulację danymi map i tekstur.

### Plik Logger.h i Logger.cpp

Te pliki odpowiadają za system logowania, który umożliwia monitorowanie działania silnika gry oraz śledzenie błędów:

* **Klasa Logger:**
  + Odpowiedzialna za rejestrowanie komunikatów, błędów oraz innych informacji diagnostycznych, które są kluczowe podczas rozwoju i debugowania gry.
  + Metody logInfo, logWarning, logError umożliwiają rejestrowanie wiadomości o różnym poziomie ważności, co ułatwia identyfikację i rozwiązywanie problemów.

### Plik utilities.h i utilities.cpp

Te pliki zawierają zestaw funkcji pomocniczych, które wspierają różnorodne operacje w grze:

* **Funkcje matematyczne:**
  + distance(): Funkcja obliczająca odległość między dwoma punktami w przestrzeni 2D.
  + vectorLength(), normalize(): Funkcje operujące na wektorach, służące do obliczania długości wektora oraz jego normalizacji.
  + segmentEnd(): Oblicza końcowy punkt segmentu na podstawie punktu początkowego, kąta i długości.
* **Inne narzędzia:**
  + projectVectorOntoLine(): Projekcja wektora na linię, używana w różnych kontekstach, takich jak fizyka czy detekcja kolizji.
  + negMod(): Funkcja obliczająca wartość modułu, obsługująca ujemne liczby.

### Plik InputManager.cpp

Plik ten odpowiada za zarządzanie wejściem od użytkownika:

* **Klasa InputManager:**
  + Zarządza wszystkimi interakcjami użytkownika z grą, w tym ruchami myszy, naciśnięciami klawiszy oraz innymi formami wejścia.
  + Umożliwia rejestrowanie i przetwarzanie sygnałów wejściowych w czasie rzeczywistym, co jest kluczowe dla interaktywności gry. # Zasada Działania Klasy BSP

## Wprowadzenie

Klasa **BSP (Binary Space Partitioning)** w naszym projekcie odgrywa kluczową rolę w zarządzaniu i przetwarzaniu przestrzeni gry, zgodnie z oryginalnymi algorytmami *Doom*’a. BSP jest strukturą danych, która umożliwia efektywne podzielenie przestrzeni na mniejsze segmenty, co jest kluczowe zarówno dla renderowania, jak i detekcji kolizji. Klasa BSP jest odpowiedzialna za przechowywanie oraz operowanie na tych danych, umożliwiając szybkie określenie, w której części mapy znajduje się gracz, oraz na której stronie danej płaszczyzny się znajduje.

## Struktura Klasy BSP

### Plik BSP.h

Plik BSP.h definiuje klasę BSP oraz jej główne funkcje i atrybuty:

* **Stałe:**
  + SSECTOR\_ID: Stała reprezentująca identyfikator subsektora, używana do odróżnienia węzłów BSP od liści (subsektorów) w drzewie BSP.
* **Klasa BSP:**
  + Jest to kluczowa klasa, która zarządza operacjami związanymi z Binary Space Partitioning. Posiada ona kilka kluczowych funkcji:
    - **Konstruktor BSP(GameLevel \*gameLevel):**
      * Konstruktor inicjalizuje obiekt BSP, ustawiając m\_rootNodeID na ID korzenia drzewa BSP oraz przechowując wskaźnik do poziomu gry (GameLevel), z którego pobierane są dane.
    - **Funkcja isPlayerOnBackSide(float playerX, float playerY, int16\_t nodeID):**
      * Funkcja ta określa, czy gracz znajduje się po “tylnej” stronie danej płaszczyzny w węźle BSP. Używa do tego celu iloczynu wektorowego (cross product), aby określić położenie gracza względem płaszczyzny podziału.
    - **Funkcja getCurrentSubsectorID(float playerX, float playerY):**
      * Funkcja ta zwraca identyfikator aktualnego subsektora, w którym znajduje się gracz. Przechodzi przez drzewo BSP, zaczynając od korzenia, aż dotrze do liścia, który odpowiada aktualnej pozycji gracza.
    - **Funkcja getSubSectorHeight(float playerX, float playerY):**
      * Ta funkcja zwraca wysokość podłogi w subsektorze, w którym znajduje się gracz. Jest to kluczowe dla określenia pozycji gracza względem terenu i potencjalnych kolizji.
  + **Prywatne atrybuty:**
    - m\_gameLevel: Wskaźnik na obiekt GameLevel, który zawiera dane poziomu gry, takie jak węzły BSP, sektory i segmenty.
    - m\_rootNodeID: ID korzenia drzewa BSP, od którego rozpoczyna się przeszukiwanie.

## Funkcjonalność Klasy BSP

Klasa BSP jest odpowiedzialna za zarządzanie i przetwarzanie struktury BSP, która dzieli przestrzeń gry na mniejsze podsektory. Dzięki temu możliwe jest:

* **Szybkie określanie pozycji gracza:** Na podstawie pozycji gracza w świecie gry, klasa BSP jest w stanie określić, w którym subsektorze znajduje się gracz, co jest kluczowe dla renderowania i detekcji kolizji.
* **Zarządzanie stronami płaszczyzn:** Klasa umożliwia określenie, po której stronie danej płaszczyzny (frontowej czy tylnej) znajduje się gracz. To pozwala na efektywne wykluczanie niewidocznych segmentów podczas renderowania.
* **Określanie wysokości terenu:** Funkcja getSubSectorHeight umożliwia pobieranie wysokości podłogi w aktualnym subsektorze, co jest kluczowe dla poruszania się gracza oraz interakcji z otoczeniem. # CollectableSystem

## Wprowadzenie

CollectableSystem to system odpowiedzialny za zarządzanie i obsługę obiektów, które gracz może zbierać w trakcie gry. Mogą to być różnego rodzaju przedmioty, takie jak amunicja, zdrowie, klucze, czy inne power-upy, które zwiększają zdolności gracza lub pozwalają mu na dalszy postęp w grze.

## Struktura

* **Klasa CollectableSystem:**
  + Jest to klasa pochodna z klasy bazowej System, która integruje się z resztą silnika gry za pomocą mechanizmu ECS.
  + Klasa przechowuje listę encji, które posiadają komponenty pozwalające na interakcję z systemem zbierania.
* **Metoda update():**
  + Metoda ta jest kluczowa w tym systemie, ponieważ przetwarza wszystkie encje w systemie, sprawdzając, czy gracz zbliżył się wystarczająco blisko, aby zebrać przedmiot.
  + W momencie zebrania przedmiotu, metoda ta usuwa encję z systemu i aktualizuje stan gracza, np. zwiększa jego zdrowie lub dodaje amunicję.

# PlayerControllSystem

## Wprowadzenie

PlayerControllSystem jest odpowiedzialny za przetwarzanie wejść od gracza i przekładanie ich na ruch i inne interakcje postaci w grze. System ten zajmuje się interpretacją sygnałów wejściowych, takich jak ruchy myszy czy naciśnięcia klawiszy, i odpowiednim reagowaniem na nie.

## Struktura

* **Klasa PlayerControllSystem:**
  + Klasa ta rozszerza System i implementuje logikę kontroli gracza.
  + Zarządza komponentami encji gracza, takimi jak pozycja (TransformComponent) i stan (PlayerStateComponent).
* **Metoda update():**
  + Metoda ta przetwarza dane wejściowe od gracza, takie jak ruchy myszy, naciskanie klawiszy, i odpowiednio aktualizuje pozycję gracza oraz inne elementy związane z kontrolą postaci.
  + Obsługuje również inne akcje gracza, takie jak skakanie, strzelanie czy interakcja z obiektami.

# GameRenderingSystem

## Wprowadzenie

GameRenderingSystem odpowiada za renderowanie sceny gry, czyli przekształcanie danych 3D w obraz 2D wyświetlany na ekranie. System ten uwzględnia tekstury, oświetlenie oraz inne elementy graficzne, aby zapewnić odpowiednie wyświetlanie świata gry.

## Struktura

* **Klasa GameRenderingSystem:**
  + Klasa ta jest odpowiedzialna za przetwarzanie i renderowanie wszystkich widocznych segmentów mapy oraz obiektów w grze.
  + Korzysta z danych zawartych w komponentach graficznych (GameDrawableComponent), a także z informacji dotyczących pozycji gracza i obiektów.
* **Metoda render():**
  + Główna metoda odpowiedzialna za przekształcanie sceny 3D na obraz 2D.
  + Wykorzystuje strukturę BSP do wykluczania niewidocznych segmentów i zapewnienia, że tylko te elementy, które są w polu widzenia gracza, zostaną wyrenderowane.
  + Obsługuje również efekty takie jak cieniowanie, animacje i inne elementy graficzne.

# MinimapRenderingSystem

## Wprowadzenie

MinimapRenderingSystem zajmuje się renderowaniem minimapy, która pokazuje graczowi orientację na poziomie i umożliwia szybkie zlokalizowanie ważnych punktów, takich jak wrogowie, cele czy przedmioty.

## Struktura

* **Klasa MinimapRenderingSystem:**
  + Odpowiada za tworzenie uproszczonego obrazu poziomu gry w postaci minimapy, którą można wyświetlać w rogu ekranu.
  + Korzysta z danych dotyczących układu mapy, pozycji gracza oraz innych istotnych elementów, aby zapewnić aktualne i czytelne odwzorowanie terenu.
* **Metoda render():**
  + Metoda ta renderuje minimapę na ekranie, uwzględniając pozycję gracza oraz widoczność innych istotnych obiektów.
  + Dostosowuje również skalę i orientację minimapy, tak aby była ona intuicyjna i łatwa do interpretacji przez gracza.