Politechnika Krakowska
im. Tadeusza Kościuszki
Wydział Fizyki, Matematyki i Informatyki

Tomasz Bijata

Numer albumu: 108638

Gra Snake z wykorzystaniem rozszerzonej rzeczywistości opartej o markery

Marker-Base Augumented reality on the example of Snake game

Praca inżynierska
na kierunku INFORMATYKA

Praca wykonana pod kierunkiem:
dr Radosław Kycia

Uzgodniona ocena:.....

.....
podpisy promotora i recenzenta

Kraków 2017

Wstęp

Przedmiotem niniejszej pracy inżynierskiej jest implementacja gry opartej na elementach rozszerzonej rzeczywistości (ang. augumented reality). Praca omawia proces tworzenia aplikacji, przedstawia architekturę komponentów, a także zawiera krótkie informacje na temat rozszerzonej rzeczywistości i gier komputerowych.

Cele pracy

Celem pracy jest zaimplementowanie trójwymiarowej gry komputerowej z wykorzystaniem rozszerzonej rzeczywistość. Zawartość pracy przybliża niektóre problemy i prezentuje rozwiązania które są wykorzystywane przy tworzeniu mechaniki gier video. Problem połączenia świata rzeczywistego z elementami generowanymi komputerowo, nie jest banalny i wymaga odpowiedniej wiedzy matematycznej. Rozwiązania przyjęte w części odpowiadającej za logikę aplikacji pozwalają na jej rozbudowę oraz zaimplementowanie innych projektów.

W części teoretycznej poruszone zostały tematy związane z tematyka rozszerzonej rzeczywistości oraz programowanie gier komputerowych. W dalszej części, dużej części pracy skupia się na realizacji projektu i opisie zastosowanych w niej rozwiązań.

Uzasadnienie wybory tematu pracy

Wybór tematu pracy został wybrany zgodnie z zainteresowaniami autora pracy. Implementacja gry połączona z rozszerzoną rzeczywistością jest bardzo dobrą okazją do sprawdzenia posiadanych umiejętności oraz zdobycia nowego doświadczenia z zakresu przetwarzania obrazu i tworzenia gier komputerowych.

Opis projektu

Projekt jest grą, która implementuje popularną grę komputerową typu wąż. Środowisko wygenerowane przez aplikację, w której porusza się gracz jest rysowane na obrazie wejściowym kamery. Obiekty generowane przez grę są odpowiednio transformowane, aby po narysowaniu sprawiały wrażenie, że znajdują się w rzeczywistym środowisku gracza.

Projekt składa się z dwóch głównych modułów i jednego dodatkowego. Podstawowe części projektu stanowią modułu odpowiedzialny za generowanie informacji dotyczących

rozszerzonej rzeczywistości; moduł odpowiada za renderowanie i logikę gry, dodatkowy moduł służy do kalibracji kamery.

Mechanika

Gracz kontroluje długie cienkie stworzenie podobne do węża, które porusza się po planszy i zbiera znajdujące się na niej jedzenie próbując uniknąć zderzenia z przeszkodami oraz z częścią swojego ciała. Po zjedzeniu przez węża jednego kawałka jedzenia, jego długość powiększa się o jeden ‘klocek’, co powoduje utrudnienie rozgrywki. Wąż może poruszać się w czerech kierunkach. Gracz ma możliwość sterowania postacią w lewo lub prawo względem kierunku poruszania się kierowaną postacią.

Narzędzia

Do implementacji projektu nie wykorzystano wysokopoziomowych bibliotek. Autor chcąc zgłębić wiedze na temat niskopoziomowych elementów tworzenia gier i przetwarzania obrazu postanowił wykorzystać następujące technologie:

Język programowania

C++ - język programowania ogólnego przeznaczenia. Charakteryzuje się wysoką wydajnością kodu wynikowego, bezpośrednim dostępem do zasobów sprzętowych i funkcji systemowych, łatwością tworzenia i korzystania z bibliotek (napisanych w C++, C lub innych językach), niezależnością od konkretnej platformy sprzętowej lub systemowej (co gwarantuje wysoką przenośność kodów źródłowych) oraz niewielkim środowiskiem uruchomieniowym. Podstawowym obszarem jego zastosowań są aplikacje i systemy operacyjne.

C++ został zaprojektowany przez Bjarne Stroustrupa jako rozszerzenie języka C o obiektowe mechanizmy abstrakcji danych i silną statyczną kontrolę typów. Zachowanie zgodności z językiem C na poziomie kodu źródłowego pozostaje jednym z podstawowych celów projektowych kolejnych standardów języka

Biblioteki

OpenCV – biblioteka funkcji wykorzystywanych podczas obróbki obrazu, oparta na otwartym kodzie i zapoczątkowana przez Intela. Biblioteka ta jest wieloplatformowa, można z niej korzystać w Mac OS X, Windows jak i Linux. Autorzy jej skupiają się na przetwarzaniu obrazu w czasie rzeczywistym.

DirectX11 jest niskopoziomowym API (interfejsem programowania aplikacji, od ang. Application programming interface), które wspomaga renderowanie grafiki 3D dzięki akceleracji sprzętowej 3D. Direct3D dostarcza interfejsy programowe, za pomocą których

uzyskujemy kontrolę nad kartą graficzną. Direct3D stanowi warstwę pomiędzy aplikacją a kartą graficzną, która sprawia, że nie musimy znać szczegółów mechanizmu działania [1].

Rozszerzona rzeczywistość

Rozszerzona rzeczywistość (and. Augumented reality) polega na przetwarzanie informacji z rzeczywistego otoczenia i wzbogacaniu informacji docierających do użytkownika, pozwalających mu, np. na lepsze zrozumienie rzeczywistości, lub jej doświadczenie. Wirtualna rzeczywistość skupia się na symulacji świata, jej elementy są generowane przy wykorzystaniu nowoczesnych technologii. Rozszerzona rzeczywistość, nie skupia się na symulowaniu światy, tylko na jego bazie dodawana są do niego nowe informacje. Istotną cechą rzeczywistości rozszerzonej jest to, że przetwarzanie informacji odbywa się w czasie rzeczywistym.

Powszechnie uważa się, że termin Augumented reality został zainicjalizowany przez Thomasa Caudella na początku lat 90-tych XX wieku, był on naukowcem pracującym w firmie Boeing. Sformułowanie zostało użyte przez Caldwella do opisu cyfrowego wyświetlacza, który nakładał generowana komputerowo grafikę na obraz widziany przez okulary. Nad rozszerzona rzeczywistością prowadzono praca naukowe już w latach 90-tych, termin ten trafił upowszechnił się w ogólnej świadomości ludzi w XXI wieku, wraz z upowszechnieniem się urządzeń zwanych smartfonami.

W zależności od zastosowania i przeznaczenia rozszerzonej rzeczywistości, aby móc ją wykorzystać niezbędne są urządzenie, sensory pobierające wartości z otaczającego świata. Wspomniane urządzanie początkowo nie były dostępne w urządzeniach elektronicznych o szerokim użyciu. Zastosowanie rozszerzonej rzeczywistość wymagało czasami używania dedykowanych, nie sprzyjało to polarności temu rozwiązaniu ze względu na koszy i małą użytecznością. Dopiero miniaturyzacja urządzeń i prowadzenie między innymi smartfonów pozwoliło na rozpowszechnienie urządzeń które są dostępne dla każdego i mogą wykorzystać możliwości oferowane przez rozszerzoną rzeczywistość. Obecnie do popularnych zastosowań rozszerzonej rzeczywistości należą:

- Medycyna – obrazowanie medyczne, pozwala na dostępność dodatkowych danych, np. na temat czynności narządów wewnętrznych pacjenta [2].
- Muzea – eksponat może zostać wzbogacony informacjami taki jak, np. miejsce odkrycia czy kontekst historyczny [3].

- Marketing, z celu promowania produktów [3].
- Edukacja – zwiększa zainteresowanie tematem, zapewnia niezbędne dane o interesujących obiektach [3].
- Rozrywka – głównie w grach komputerowych [3].

W dalszej części tego dokumentu stosowane są ogólnie przyjęte skróty: VR jako „wirtualna rzeczywistość” (z angielskiego Virtual Reality), AR zamiast „rozszerzona rzeczywistość”.

Gry komputerowe

Gra komputerowa jest rodzajem oprogramowania komputerowego stworzonego do celów rozrywkowych bądź edukacyjnych. Aplikacje stawiają przed użytkownikiem pewien zestaw problemów które mogą być problemami logicznymi, bądź zręcznościowymi. Gry komputerowe są pisane na wiele urządzeń, można je uruchomić na komputerach osobistych, konsolach do gry, telewizorach czy telefonach komórkowych. Gry stawiają przed graczem rożne zadana w zależności od ich gatunku, mogą polegać przykład na eliminacji wirtualnych przeciwników, rywalizacji z innymi graczami lub sztuczna inteligencja, czy rozwiązywaniu problemów logicznych.

Historia gier komputerowych

Okres historii gier komputerowych obejmuje czas od 1947 roku, do czasu, kiedy wynaleziono pierwszy prototyp gry elektronicznej. Gry komputerowe do masowego obiegu weszły za sprawą gry Pong, do 1972 roku były produkcjami akademickimi. Dzięki popularności jakie zyskiwały gry, na rynku pojawiały się odpowiednie platformy do grania – automaty i konsole, a od lat 80. XX na rynku postawiły się komputery osobiste. W latach 90 gry stały się produktem masowym, zaczęły one odługować grafikę trójwymiarową oraz rozgrywkę sieciową.

Za pierwszy prototyp gry komputerowej uznaję się stworzony w 1947 roku przez Amerykanów Thomasa A. Goldsmitha Jr. i Estle Raya program Cathode-Ray Tube Amusement Device, był to analogowy symulator pocisku rakietowego używający lamp elektronowych [4]. Gry komputerowe przyjęły formę graficzną po wynalezieniu w Wielkiej Brytanii komputera EDSAC, była to pierwsza maszyna licząca posiadająca wyświetlacze, które stanowiły prototyp monitora [5]. Pionierską gra, stała się obecnie kultowa gra kółko i krzyżyk stworzona przez pracownia uniwersytetu w Cambridge, Alexander Sandy Douglas

[5]. Grą, która poprowadziła rynek gier o krok dalej, była gra Tennis for Two na podstawie gry w tenisa (1958) stworzona prze Williama Higinbothama. Była to pierwsza graficzna gra komputerowa, wyświetlana na oscyloskopie i posiadająca kontrolery do sterowania.

Generacje konsol - pierwsza generacja

Można przyjąć, że pierwsza generacja trwała w latach siedemdziesiątych a dokładnie w latach 1972-1979. Jednak już koniec lat sześćdziesiątych był bardzo emocjonujący za sprawą Ralpha Beara który był na początku zwykłym elektro - mechanikiem telewizji który w 1966 roku stworzył prostą dwuosobową grę wideo a nazwał ją Chase.

Gra polegała na tym, że dwie kropki na ekranie wzajemnie się ścigały. Ralph Bear wpadł na pomysł, żeby sprzedawać swoją grę firmom zajmującym się telewizją kablową. Trzeba zaznaczyć, że telewizja kablowa w latach 60 i 70-tych przeżywała wielki kryzys. Prace trwały, Ralph udoskonalał swój produkt czego efektem był Brown Box mający kontrolery, pistolet oraz szesnaście przycisków na konsoli.

Druga generacja

Druga generacja to okres pomiędzy 1976 a 1984. Konsole zawierające procesor Fairchild Channel F zaliczane są do drugiej generacji konsol. W 1977 roku Atari wydało własną konsolę, która miała możliwość wymiany kartridżów, nazwaną Video Computer System (VCS) mało kto wtedy wiedział, że dziś będzie to konsola kultowa, bo w 1982 zmieniono jego nazwę na Atari 2600.

Magnavox w 1978 roku wydał własną konsolę z kartridżami, Magnavox Odyssey 2. Konsola nie była taka popularna jak Atari, ale pozwoliło jej to w ciągu pięciu lat sprzedać ponad milion sztuk tej konsoli. Warto także wspomnieć fakt, że ta sama konsola w Europie była sprzedawana jako Videopac G7000.

Trzecia generacja konsol

Trzecia generacja konsol przypada na lata 1983-1995 i jest to era 8-bitowców. Mimo iż poprzednia generacja też już miała 8 bitów to dopiero od tej pory konsole zostały oznaczone jako bitowce. Była to pierwsza generacja po wielkim kryzysie.

Najbardziej popularna i oferującą niesamowitą grafikę oraz gry z duża grywalnością była konsola firmy Nintendo Entertaiment System znana w Japonii jako Famicom. W tym czasie powstało wiele gier znanych do dziś, między innymi: Super Mario Bros, Final Fantasy. Były to gry które zmieniły rynek, pokazały nową jakość, posiadały fabułę.

Na rynku pojawiła się SEGA która okazała się największym konkurentem Nintendo. Sega ze swoim Sonic the Hedgehog zyskała popularność graczy, a konsola Sega Master System okazała się przebojem rynkowym. Na rynku dalej pozostawała firma Atari ze swoim Atari 7800.

Czwarta generacja

Czwarta generacja to głównie wyścig pomiędzy firmami Nintendo oraz Segi. Ta generacja nazywała się erą 16-bitowców. Nintendo stworzyło Super Nintendo Entertainment System natomiast Sega stworzyła konsolę Sega Mega Drive. Firmy rywalizowały na każdym kroku, polegało to przede wszystkim na porównywaniu grafiki, jakie oferowały konsole, oraz na porównywaniu która maskotka firmy jest lepsza, czy to Mario od Nintendo czy Sonic od Segi.

Ta generacja rozwijała ideę poprzedniej generacji, lepsza grafika, tytuły zaczerpniete z poprzednich konsol, chociaż parę nowych też się pojawiło produkcji, jednak dało się zauważyć, że było dużo kontynuacji. Nie była ta generacja tak innowacyja jak poprzednie gdzie zawsze coś było nowością czy były jakieś innowacje w grach. W tej generacji nośnikiem danych był kartridż, podobnie jak w większości dostępnych wtedy konsol.

Piąta generacja

Piąta generacja konsol to era 32- i 64 bitowców. W tej erze powstały konsole zarówno 32-bitowe jak i 64-bitowe, a do rywalizacji na rynku konsol włączyło się SONY.

Określanie liczby bitów nie wzięło się znikąd, gdyż liczba bitów miała pokazywać moc konsoli, zaawansowanie techniczne. Era 32 oraz 64-bitowców zasłynęła jednak nie z ilości bitów, a gier prezentowanych w pełnym trójmiarze. Konsole w tej generacji rewolucjonizowały branżę gier pod względem graficznym. Na ten okres przypada wiele rozpoznawalnych do dziś gier między innymi Super Mario 64 na Nintendo 64 i Tomb Raider na konsolę Saturn, a później także gra wyszła na PlayStation. W tym czasie firmy mocno reklamowały swoje konsoli i produkcje na nie przeznaczone.

Era konsol 32- i 64-bitowców oprócz wejścia do świata 3D wykorzystywała nośniki CD. W czasie tej generacji również toczyły się spory, tym razem dotyczył on nośników; konsola Nintendo wykorzystywała jeszcze kartridże, a Sony wykorzystało innowacyjny wtedy napęd CD. Nintendo broniło się, że kartridż utrudnia piractwo gier oraz szybciej się ładuje, co było prawda jednak kartridż miał małą pojemość, nie mógł pomieścić tyle danych co płyta CD.

Szusta generacja

Szósta generacja ma miejsce na początku XXI wieku, obejmuje gry komputerowe oraz wszelkiego rodzaju konsole dostępne. Urządzenia do obsługi gier były sprzedawane w milionach sztuk. Należą do nich: Sega Dreamcast, Nintendo GameCube, Sony PlayStation 2 i Microsoft Xbox. Wtedy właśnie uzmysłowiono sobie, że konsola wcale nie musi oznaczać tylko gry. Do niektórych z nich dodawano funkcjonalności typowe dla telefonów komórkowych, odtwarzaczy mp3, a nawet palmtopów. Pierwszym takim połączeniem była Nokia N-Gage, która w konsekwencji przekształciła się z telefonu komórkowego w konsolę. W latach 1998 – 2006 znów zaczęto bardzo głośno mówić o nieodpowiednich treściach umieszczanych w grach (seks, przemoc, propagowanie niewłaściwych zachowań), ale mimo niemalejącego oburzenia części społeczeństwa, producenci gier nie zatrzymali się i nadal prężnie się rozwijali.

Siódma generacja

Siódma, czyli obecna generacja gier komputerowych, rozpoczęła się w 2005 roku wydaniem konsoli Xbox 360 i zakorzeniła się, kiedy na rynek zostały wypuszczone konsole do gier wideo firm Sony oraz Nintendo (2006 rok). Nazywamy ją erą „high definition". Obecnie możemy sterować grami nie tylko za pomocą joysticka czy przycisków myszki, ale również poruszając własnym ciałem.

Przykładem jest konsola Xbox 360 Kinect. Posiada ona trójwymiarową kamerę oraz czujnik ruchu. Dzięki temu każda wykonywana przez nas czynność wywołuje odpowiednią reakcję w grze.

Proces tworzenie gier komputerowych

Przedsięwzięcie jakim jest wtórzenie gier komputerowych jest dużym wyzwaniem, a wraz z rozwojem rynku gier komputerowych diametralnie się zwiększyło. Produkcje z początku lat dziewięćdziesiątych takie jak nieśmiertelny "Prince of Persia" (1989), czy "Another World" (1991) stworzone były przez pojedynczych programistów. Powstawały wtedy pierwsze zespoły deweloperskie z kilku osób. Rynek gier komputerowych rozwijał się wraz rozwojem komputerów osobistych, co powodowało zwiększone zapotrzebowanie na komputerową rozrywkę. Wraz z zapotrzebowaniem - zespoły powiększyły się i wyspecjalizowały. Liczebność zespołu przygotowującego grę jest zależna od klasy powstającej produkcji. Gry tworzone na gotowym silniku powstają o okresie około kilku miesięcy. Przy ich powstawaniu zazwyczaj bezpośrednio zaangażowanych jest kilku grafików, projektantów poziomów i programista, realizujący tylko bieżące zapotrzebowania

zespołu. Zaangażowanie zespołu jest dużo większe przy dużych produkcjach, na przykład przy ”Wiedźminie” prace koncepcyjne rozpoczęły się w kilkuosobowym zespole. Nad projektem w decydującej fazie było zatrudnionych było ponad sześćdziesiąt osób. W okresie prac, tuż przed wydaniem gry, zespół powiększył się powyżej siedemdziesięciu osób. Poza zespołu, przy produkcji dodatkowo prowadzono współpracę z zewnętrznymi działami kontroli jakości oraz firmami tworzącymi trójwymiarowe modele.

Tworzenie gier jest procesem iteracyjnym. Nad projektem wynikowym pracuje jednocześnie kilka zależnych od siebie zespołów. Bardzo dobrą praktyką jest zaczynanie od prototypu, czuli od prostego szkieletu gry stworzonego małym nakładem sił, aby mus ocenić potencjał projektu. Prototyp jest podstawą do rozpoczęcia rzeczywistej implementacji i mechaniki gry. Dokumentacja projektowa w branży rozrywki elektronicznej jest słabiej rozwinięta niż w oprogramowaniu biznesowym. Rolę klienta może stanowić dystrybutor. Głównym dokumentem projektu gry jest Game Doc, bardzo szczegółowo opisujący całą mechanikę gry. Często jest zbiorem pomniejszych dokumentów opisujących poszczególne elementy projektu. Relacje pomiędzy producentem gry a wydawcą różnią się od tych jakie panują między producentem a klientem biznesowym. W grach dokumentacja pełni mniejszą rolę i jest tworzona raczej na wewnętrzne potrzeby zespołu produkcyjnego. Istotną rzeczą w produkcji gier zajmują narzędzia, są potrzebne zespołom do tworzenia, więc muszą powstać szybko, w początkowych fazach implementacji, wraz z tworzeniem mechanizmów gry. Po wstępnych projektach, możliwie szybko powstaję działająca wersja gry, podstawowe mechanizmy, na których oparta będzie rozgrywka, by pracę rozpocząć mogli projektanci. Początkowe etapy produkcji mogą być pominięte, jeżeli projekt powstanie przy gotowego silnika gry.

Realizacja projektu

Rozszerzona rzeczywistość

Rozszerzona rzeczywistość wymaga od programisty uważnego podejścia przy implementacji. Ważnym elementem jest to, że nie pracuję się całkowicie z wirtualnym środowiskiem, ale z „rzeczywistością mieszaną”. Głównym problemem stanowiącym wyzwanie jest synchronizacja kamery oraz renderowanie obiektów sceny w sposób, aby ruchy urządzenia rejestrującego klatki powodowały odpowiadających ruch sceny o taki sam kąt jak i przesuniecie. Przy implementacji modułu odpowiedzialnego za rozszerzoną

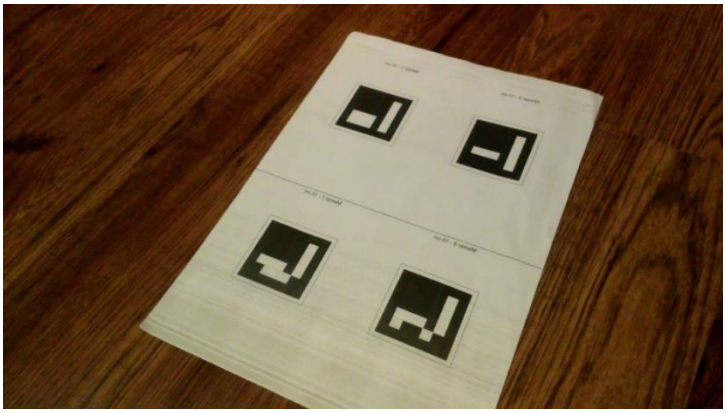
rzeczywistość wykorzystano bibliotekę OpenCV, która została stworzona z myślą o przetwarzaniu obrazu w czasie rzeczywistym.

Wykrywanie znaczników

Głównym elementem rozszerzonej rzeczywistość wykorzystywanym w aplikacją są markery. Ich poprawna detekcja jest kluczowa dla końcowego efektu jaki widoczny będzie dla użytkownika. Poniżej przedstawiono wykorzystaną proceduję detekcji znaczników.

Procedura wykrywania znacznika:

- Konwersja obrazu do skali szarości.
- Wykonaj operację binarną progową.
- Wykrywanie konturu.
- Szukaj możliwych markerów.
- Wykrywanie i zdekodowanie znaczników.



Rys. 1. Przykładowy obraz wejściowy z kamery.

Konwersja obrazu na skalę szarości

Konwersja obrazy do szarości pozwala ułatwić dalszą pracę z obrazem. Markery zazwyczaj składają się tylko z czarnych i białych bloków, co pozwala z nimi łatwiej pracować w skali szarości.

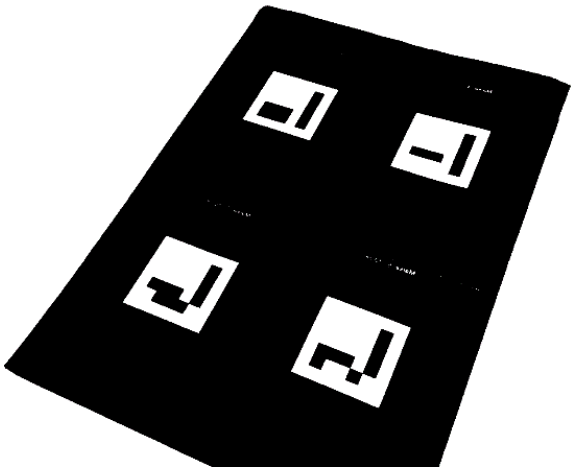
W OpenCV konwersja kolorów jest dość prosta, wystarczy wykonać funkcję `cv::cvtColor` z parametrem `BGRA2GRAY` który określa rodzaj konwersji kolorów.

Binaryzacja obrazu

Operacja binaryzacji tworzy obraz binarny, czyli taki którego piksele mają jedna z dwóch wartości, kolor czerni (zerowa intensywność) lub biały (pełna intensywność). Krok ten jest konieczny, aby przygotować obraz do szukania konturów. Istnieje kilka metod

progowania obrazu, mają one mocne i słabe strony. Najprostszą i najszybszą metodą jest progowanie twarde. W metodzie tej wartość wynikowa zależy od wartości piksela i jakiejś wybranej wartość progowej. Jeżeli natężenie koloru piksela jest większe niż wartość progowa, wynikiem jest biały (255), bądź czarny (0) piksel.

Metoda ta ma ogromną wadę, jest zależna od warunków oświetlenia. Ulepszoną metodą progowania jest progowanie adaptacyjne. Główna różnicą tej metody w stosunku do poprzedniej jest uwzględnienie wszystkich pikseli w danym promieniu wokół badanego piksela. Przy użyciu średniej intensywności progowanie adaptacyjne daje dobre wyniki i zapewnia bardziej niezawodne wykrywanie kątów.



Rys. 2. Obraz wynikowy po zastosowaniu progowania adaptacyjnego na obrazie wejściowym.

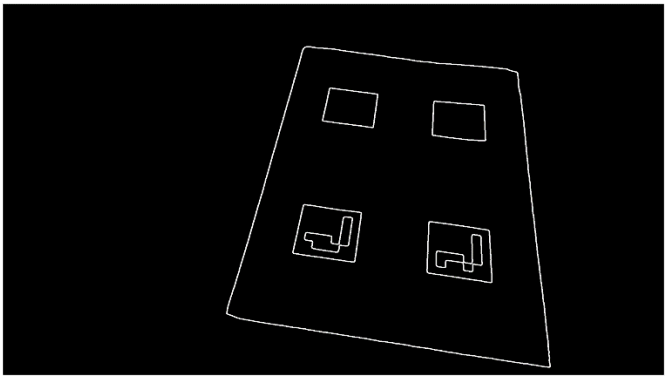
Detekcja krawędzi

Marker wyglądem przypominają kwadratową figurę z czarnymi i białymi obszarami wewnątrz niego. Jednym ze sposobów na zlokalizowanie znacznika jest znalezienie zamkniętego konturu i aproksymacja go do wielokąta zawierającego 4 wierzchołki.

```
void MarkerDetector::findContours(cv::Mat& thresholdImg,
ContoursVector& contours, int minContourPointsAllowed) const
{
     ContoursVector allContours;
     cv::findContours(thresholdImg, allContours, CV_RETR_LIST,
V_CHAIN_APPROX_NONE);

     contours.clear();
     for (size_t i = 0; i<allContours.size(); i++)
     {
```

```
            int contourSize = allContours[i].size();
            if (contourSize >= minContourPointsAllowed)
            {
                contours.push_back(allContours[i]);
            }
        }
}
```

Wynikiem funkcji jest lista wielokątów. Każdy z tych wielokątów reprezentuje jeden konturu. Funkcja filtruje ze znalezionych konturów, te których obwód w pikselach jest mniejszy od wartości zmiennej `minContourPointsAllowed`. Parametr został dodany, ponieważ nie należy dalej przetwarzać konturów które składają się ze zbyt małej ilości punktów. Taki kontur prawdopodobnie nie zawierają w sobie markera lub nie będzie można rozpoznać jego wartości ze względu na niewielki rozmiar.



Rysunek 1: Wizualizacja wykrytych konturów.

Znalezienie kandydatów

Posiadając kontury które potencjalnie zawierają w sobie interesujące na fragmenty obrazu, kolejnym krokiem jest przybliżenie konturu składającego się z pikseli na wielokąta zbudowany z wierzchołków. Przybliżenie wielokąta jest również potrzebne, aby zmniejszyć liczbę punktów, które opisują kształt konturu. Ponieważ znaczniki zawsze są opisywane są przez wielokąt, który zawiera cztery wierzchołki, więc, jeśli przybliżenie wielokąta ma więcej niż lub mniej niż 4 wierzchołki, to na pewno dany kontur nie jest markerem. Poza testowaniem liczy wierzchołów do znalezienia konturów o odpowiednich własnościach wykonywane są również następujące kroki: odległość między wierzchołkami nie może być zbyt mała – pozwala to na wykrycie części [6] przypadków. Z listy potencjalnych konturów będących markerami usuwane są również te które są zbyt blisko innych konturów; usuwanym konturem jest ten o mniejszej powierzchni.

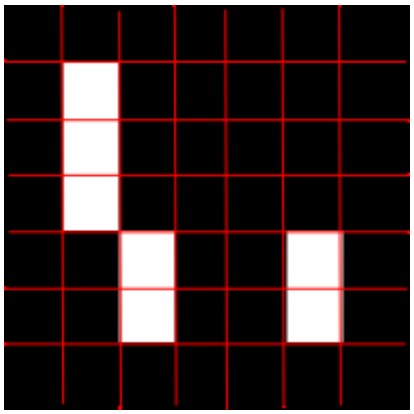
Rozpoznanie markera

Posiadając listę prostopadłościanów, które potencjalnie są markerami, kolejnym krokiem jest rozpoznanie danego fragmentu obrazu, potwierdzenie, że jest to marker, oraz odczytanie zakodowanej w nim informacji. Do sprawdzenia, czy dany kontur jest markerem czy nie, należy wykonać trzy kroki:

- Usunięcie rzutowania perspektywicznego, tak aby uzyskać frontalny widok obszaru prostokąta.
- Następnie wykonujemy progowanie obrazu przy użyciu algorytmu Otsu. Ten algorytm zakłada dwumodalny rozkład i znajduje wartość progową maksymalizuje wariancję ekstra klasy, przy jednoczesnym zachowaniu niskiej wariancji wewnątrz klasy.
- Identyfikację kodu znacznika. Jeśli rozpoznano marker, to należy odczytać zakodowaną w nim informacje. Znacznik jest podzielony na 7 x 7 siatkę, którego wewnętrzne 5 x 5 komórki zawierają informacje o jego identyfikatorze, reszta siatki tworzy zewnętrza krawędź markera. Podczas rozpoznawanie kodu markera najpierw sprawdzamy, czy zewnętrzne czarne obramowanie jest obecne. Następnie odczytujemy wewnętrzne 5 x 5 komórki i sprawdzić, czy stanowią one poprawnie zakodowana informację.

Detekcja znacznika

Każdy marker niesie ze sobą informacje. Wewnętrzny kod markiera zapisany jest w 5 słowach, gdzie każde słowo składa się z 5 bitów. Do kodowania informacji użyto nieznacznie zmodyfikowanego kodu Hamminga. Każde słowo niesie tylko 2 bity informacji z pośród 5 bitowego słowa, pozostałem 3 bity mają zastosowanie do wykrywania błędów. Takie przeznaczenia bitów pozwala na zdefiniowanie do 1024 różnych identyfikatorów. Bity parzystości znajdują na bitach 1, 3 i 5. Bity informacji znajdują się na pozycji 2 i 4. Główną różnicą używanego kodu z kodem Hamminga, jest to, że pierwszy bit jest odwrócony, przykładowo: ID 0 w kodzie Hamminga ma postać 00000, po zastosowania odwrócenia 1'szego bitu wartość wyniesie 10000 dla używanego algorytmu w kodzie. Taka negacja bitu jest używana, aby wyeliminować sytuację, gdzie całkowicie czarny prostokąt jest wykrywana jako poprawny znacznik z ID, zmniejsza to prawdopodobieństwo wystąpienia fałszywych detekcji z obiektami które znajdują się w środowisku widocznym na obrazie wejściowym.



Rys. 3. Marker z widoczną siatką podziału [7].

Poprzez zliczanie liczby czarnych i białych pikseli dla każdej komórki tworzona jest maska 5 x 5-bitowa z kodem znacznika. Aby policzyć liczbę niezerowych pikseli na pewnym wycinku obrazu wykorzystana została funkcja cv :: countNonZero. Funkcja ta zlicza elementy niezerowe danej tablicy 1D i 2D.

Rys. 4. Marker w rożnych orientacjach [7].

Po odczytaniu maski bitów reprezentowanych przez marker, należy sprawdzić w jakim położeniu się znajduje. Istnieją cztery możliwe orientacje obrazu znacznika, należy odnaleźć właściwe położenie znacznika. Marker zawiera trzy bity parzystości na każde dwa bity informacji. Za pomocą bitów parzystości należy znaleźć odległość Hamminga dla każdej możliwej orientacji markera. Właściwe obrócona macierz bitów markera będzie miał zerową odległość Hamminga, podczas gdy dla innych obrotów błąd ten będzie większy niż zero. Jeśli nie zostanie zidentyfikowana pozycja markera dla której błąd Hamminga wyniesie 0, to znaczy, że w poprzednich krokach odczytano fragment obrazu zawierający zły wzór markera (zniszczony obraz lub fałszywie wykrycie markera).

Odległość hamminga

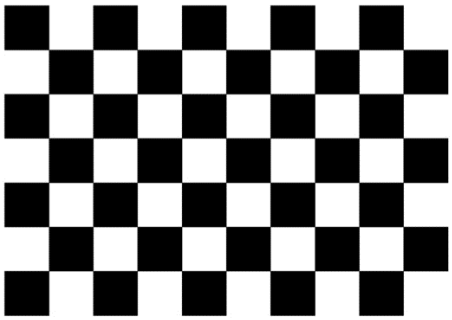
Odległość Hamminga - wprowadzona przez Richarda Hamminga jest miara odmienności dwóch ciągów o takiej samej długości, wyrażająca liczbę miejsc (pozycji), na których te dwa ciągi się różnią. Innymi słowy jest to najmniejsza liczba zmian (operacji zastępowania elementu innym), jakie pozwalają przeprowadzić jeden ciąg na drugi [8].

Zaimplementowany algorytm sprawdzenia odległości Hamminga dla słów zapisanych na markerze.

```
int Marker::hammDistMarker(cv::Mat bits)
{
	// Wszystkie możliwe słowa
	int ids[4][5] =
	{
		{ 1,0,0,0,0 },
		{ 1,0,1,1,1 },
		{ 0,1,0,0,1 },
		{ 0,1,1,1,0 }
	};

	int dist = 0;
	for (int y = 0; y<5; y++) {
		int minSum = 1e5; //odległość Hamminga dla każdego możliwego słowa

		for (int p = 0; p<4; p++) {
			int sum = 0;
			//zliczanie
			for (int x = 0; x<5; x++) {
				sum += bits.at<uchar>(y, x) == ids[p][x] ? 0 : 1;
			}
			if (minSum>sum)
				minSum = sum;
		}
		dist += minSum;
	}
	return dist;
}
```

Kalibracja kamery

Każdy obiektyw ma unikalne parametry, takie jak ogniskowa, punkt skupienia oraz indywidualne zniekształcenia soczewki. Proces obliczania wewnętrznych parametrów aparatu nazywany jest kalibracją kamery. Proces kalibracji jest ważny do aplikacji wykorzystujących rozszerzoną rzeczywistość, ponieważ opisuje transformację perspektywy i zniekształcenia soczewki na obrazie rejestrowanym. Aby użytkownik mógł doznać jak najlepszych doświadczeń z rozszerzona rzeczywistością, wirtualne obiekty powinny zostań narysowane przy wykorzystaniu takie samej perspektywy. Aby dokonać kalibracji kamery, przy korzystaniu funkcji zaimplementowanych w OpenCV potrzebny jest specjalny wzór obrazu (szachownica lub czarny okręgi na białym tle). Do kalibracji niezbędne jest zrobienie około 10-15 zdjęć takiego wzorowego obrazu z różnych punktów widzenia. Algorytm kalibracji następnie znajdzie optymalne wewnętrzne parametry kamery i wektor zniekształceń soczewki [9].



Rys. 5. Obraz wzorcowy służący do kalibracji kamery.

Położenie znacznika w przestrzeni

Celem rozszerzonej rzeczywistości jest próba stworzenia iluzji, ‘zlania’ się wirtualnych obiektów z obrazem świata rzeczywistego. Aby umiejscowić model 3D w scenie, konieczne jest posiadanie informacji o jego położeniu, w tym przypadku będzie to położenie względem urządzenia rejestrującego obraz wejściowy. Wykorzystano euklidesową transformację w kartezjańskim systemie współrzędnych do reprezentowania współrzędnych.

Położenie znacznika w przestrzeni trójwymiarowej oraz odpowiadającej mu projekcji 2D może zostać opisany następującym równaniem:

P = A * [R | T] * M;

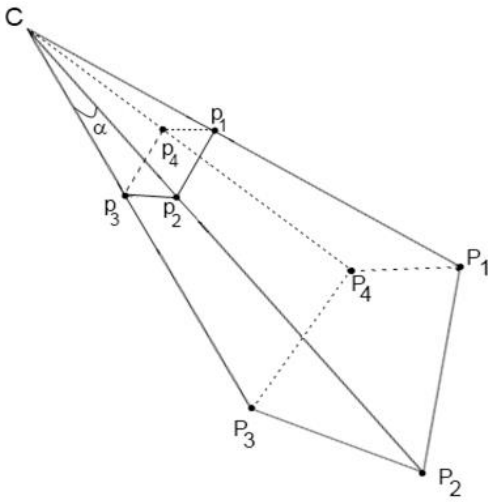
Gdzie:

- M oznacza punkt w przestrzeni 3D
- [R | T] oznacza [3 | 4] macierzy reprezentującej euklidesową transformacji
- A oznacza matrycę kamery lub matrycę wewnętrzne parametry
- P oznacza projekcję M w przestrzeni ekranu

Po wykonaniu etap detekcji markera obecnie zna położenie czterech rogi marker 2D (rzuty w przestrzeni ekranu). W następnym rozdziale będziesz dowiedzieć się, jak uzyskać parametry macierzy A i M wektorów i obliczyć [R | T] transformacji.

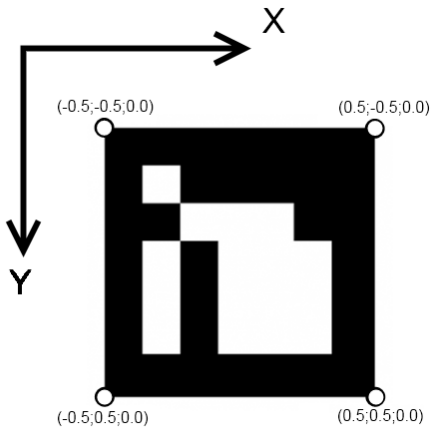
Szacowanie pozycji znacznika

Znając dokładną lokalizację rogów znacznika na obrazie, możliwe jest oszacowanie transformacji między kamerą i markerem w przestrzeni 3D. Ta operacja jest znana jako szacowanie pozycji z korespondencji 2D-3D. Wynikiem procesu szacowania pozycji jest transformacja euklidesowa (który składa się z elementów rotacji i translacji) między aparatem a obiektem.



Rys. 6. Korespondencja punktów 2D-3D [7].

Na powyższym rysunku C określa środek kamery. Punkty P1-P4 są punktami 3D na świecie układu współrzędnych, a punkty P1-P4 są ich odpowiednikami na płaszczyźnie obrazu kamery. Do ustalenia pozycji niezbędne jest znalezienie transformacji między znaną pozycją markera w świecie 3D (P1-P4), a kamera C za pomocą macierzy opisującej własności kamery oraz znanych punktów markera będących projekcją na płaszczyznę obrazu(P1-P4). Współrzędne położenia znacznika w przestrzeni 3D są ustalane w następujący sposób; marker ma zawsze formę kwadratu i wszystkie wierzchołki leżą w jednej płaszczyźnie, co pozwala na zdefiniowanie ich narożników ja na obrazku Rys. 7.



Rys. 7. Współrzędne wierzchołków markera [7].

Do znalezienia pozycji kamery z wyżej opisanej korespondencji 2D-3D użyto funkcji `cv::solvePnP`:

```
void solvePnP(const Mat& objectPoints, const Mat& imagePoints, const
Mat& cameraMatrix, const Mat& distCoeffs, Mat& rvec, Mat& tvec, bool
useExtrinsicGuess=false);
```

Tablica objectPoints zbiór punktów obiektów(zancznikó) w przestrzeń obiektu. Do tego parametruprzekazana zostaje lista współrzędnych markerów w przestrzeni 3D (wektor czterech punktów).Parametr imagePoints jest tablicą odpowiadających punktów obrazu (lub projekcji).Kolejne parametry funkcji.

- cameraMatrix – macierz 3 x 3 kamery opijącują jej wwewnetrzene własności.
- distCoeffs - wektor 4 x 1, 1 x 4, 5 x 1 lub 1 x 5 opisujący zniekształcenia soczewki (9).
- rvec: wynikowy wektor przesunięcia, razem z tvec opsuje transformacje punktu ze współrzędnych modelu do współrzędnych układu kamery.
- tvec – wynikowy wektor przesunięcia.
- useExtrinsicGuess: Jeśli prawda, funkcja użyje dołączonych wektorów rvec i tvec jako początkowych wektorów przybliżeń rotacji i translacji.

Funkcja oblicza transformacji kamery w taki sposób, że minimalizuje błąd odwzorowania, czyli sumę kwadratów odległości między punktami imagePoints, a ich projekcją objectPoints.

W celu uzyskania macierzy 3 x 3 obrotu z wektora obrotu, użyto funkcji cv::Rodrigues. Funkcja ta zamienia ruch obrotowy reprezentowany prze wektor rotacji i zwraca jego odpowiednik jako macierz obrotu.

Ponieważ cv::solvePnP znajduje pozycję kamery w odniesieniu do pozycja markera w przestrzeni trójwymiarowej, należy odwrócić znalezioną transformację. Wynikowa transformacja będzie opisywać położenie markera w systemie współrzędnych kamery, które można wykorzystać w procesie rysowania sceny.

```
for (size_t i = 0; i<detectedMarkers.size(); i++)
{
	Marker& m = detectedMarkers[i];

	cv::Mat_<float> Rvec;
	cv::Mat_<float> Tvec;
	cv::Mat raux, taux;
	cv::solvePnP(m_markerCorners3d, m.points, camMatrix, distCoeff, raux,
taux);
	raux.convertTo(Rvec, CV_32F);
	taux.convertTo(Tvec, CV_32F);

	cv::Mat_<float> rotMat(3, 3);
	cv::Rodrigues(Rvec, rotMat);
```

```
        // Copy to transformation matrix
        for (int col = 0; col<3; col++)
        {
              for (int row = 0; row<3; row++)
              {
                    m.transformation.r().mat[row][col] = rotMat(row, col);
              }
              m.transformation.t().data[col] = Tvec(col);
        }
        m.transformation = m.transformation.getInverted();
}
```

Zarządzanie zasobami

RAII

RAII opisuje zasadę, przejęcia zasobu w konstruktorze klasy i zwolnienia w jego destruktorze. Ponieważ zarówno konstruktor jak i destruktor są wywoływane automatycznie, gdy obiekt jest tworzony lub wychodzi poza zakres, nie ma konieczności śledzenia zasobów ręcznie. RAII jest głównie wykorzystane do automatycznego zarządzania pamięcią (jak w inteligentnych wskaźnikach), ale mogą być stosowane na wszelkiego rodzaju zasobów. Dużą zaletą RAII ponad Instrukcjami alokacji i dealokacji (takich jak `new` / `delete`) jest to, że gwarantuje wystąpienie dealokacji nawet wtedy, gdy istnieje wielu instrukcji powrotu lub wyjątki w funkcji. Aby osiągnąć ten sam bezpieczeństwa z ręcznego zarządzania pamięcią, każda możliwa ścieżka powrotu z funkcji musi być chroniona przez operator `delete`. W rezultacie, kod szybko staje nieczytelny i podatny na błędy.

Models

Klasa z biblioteki DirectXTK służąca do rysowania prostych siatek ze wsparciem dla modeli ładowania z plików .CMO, plików DirectX SDK .SDKMESH i plików .VBO. Jest to implementacja modułu renderującego siatki podobny do projektu XNA Game Studio Model, ModelMesh, ModelMeshPart [10].

Model składa się z jednego lub kilku instancji ModelMesh. Obiekty ModelMesh mogą być współużytkowane przez wiele wystąpień modelu. Instancja ModelMesh składa się z jednego lub większej liczby instancji ModelMeshPart.

Każdy obiekt ModelMeshPart odnosi się do bufora indeksu, bufora wierzchołków, układu wejściowego wierzchołków (ang. input layout), instancji efektów i zawiera różne metadane potrzebne do rysowania geometrii. Każdy ModelMeshPart reprezentuje pojedynczy materiał, który renderowany jest jednym polecenie rysowania.

Obsługa błędów

Większość operacji nie powoduje błędu podczas jej wykonywania, jednakże istnieją instrukcje które mogą je powodować. Błędy należy rozpoznać i sensownie obsłużyć. Przykładowo błąd może wystąpić podczas ładowania model, np. określony plik może nie istnieć lub plik może mieć nieprawidłowy format, posiadać nieprawidłowe odniesienia do inny plików, lub być zbyt duży dla pamięci wideo karty graficznej.

Istnieje kilka strategii reagowania na błędy. W dalszej części opisaną są sposoby obsługi błędów dla przypadku obsługi zasobów dla modeli 3D. Podczas ładowania zasobów, należy wziąć pod uwagę, że taki zasób będzie później potrzebny podczas tworzenia odpowiednich instancji obiektów w grze, więc jeśli taki obiekt zażąda modelu, musi zostać zwrócony jakiś model 3D. Jedną z możliwości byłoby zapewnienie domyślnego obiektu (na przykład, prosty obiekt), dzięki czemu obiekty są zostaną utworzone poprawnie, jednak nie jest to poprawne zachowanie, ponieważ tworzony obiekt powinien mieć odpowiednio załadowane zasoby lub nie utworzyć się wcale i metoda `ModelHolder::load` musi zgłosić błąd.

Zwracanie typu logicznego

Funkcja może zwracać wartość logiczną oznaczającą sukces lub niepowodzenie. Takie podejście ma pewne wady. Nie możemy użyć zwracanej wartości do innego celu, dodatkowo, wywołujący funkcję musi sprawdzić zwracaną wartość za każdym razem wywołuje `ModelHolder::load`, daje to możliwość, że błąd nie zostanie sprawdzony i obsłużony, oraz prowadzi to tworzenia nieczytelnego kodu, który jest pełen kontroli błędów.

Rzucanie wyjątków

Innym podejściem do reagowania na awarię ładowania zasobu jest rzucenie wyjątku. Używany jest standardowy typu wyjątku `std::runtime_error`. Do jego konstruktora, podajemy komunikat o błędzie opisujący problem jak najdokładniej, w tym pliku:

```
std::unique_ptr<DirectX::Model> model =
DirectX::Model::CreateFromCMO(md3dDevice, filename.c_str(), *mfxFactory);
if (model.get() == nullptr) {
      std::wstring wstr;
      wstr = L"ModelHolder::load - Failed to load " + filename;
      std::string str(wstr.begin(), wstr.end());
      throw std::runtime_error(str);
}
```

Wyjątki mają wielką zaletę, że kod użytkownika może nie zawierać sprawdzania pajawienia się po kazdje instrukcji. Urzytkownik klasy `ModelHolder` może mieć teraz następujący kod:

```
ModelHolder mModels;
mModels.load(Models::SnakeHead, L"Models\\SnakeHead.cmo");
mModels.load(Models::Apple, L"Models\\apple.cmo");
mModels.load(Models::GenericPalm, L"Models\\GenericPalm.cmo");
```

Nie trzeba sprawdzić każdego wywołania. Jeśli wystąpi błąd, wyjątek będzie rzucony, dopóki blok `try-catch` go złapie.

Gdy zasób jest załadowany, możemy wstawić go do mapy. Tutaj również możliwe są źródła błędów. Kontener std::map nie pozwoli na wstawienie pary ID-zasób, jeżeli już zawiera takie ID, ponieważ nie może zawierać duplikatów kluczy.

```
auto inserted = mResourceMap.insert(std::make_pair(id,
std::move(resource)));
```

Teraz zmienna `inserted` zawiera parę zawierającą iterator i wartość logiczną, inserted.second to wartość logiczna określająca sukces operacji wstawiania. Wartość fałszywa oznacza, że identyfikator jest już zapisany na mapie. Można w takiej sytuacji rzucić wyjątek `std::runtime_error` ponownie. Jednakże, w przeciwieństwie do błędu dwukrotnego ładowania, wstawianie jest błędem wykonania. Próba wstawienia tego samego ID dwukrotnie do mapy jest błędem logicznym. Oznacza to, że jest to błąd logiczny w aplikacji innymi słowy, bug. Poprawnie napisany program nie będzie próbował załadować tego samego zasobu dwukrotnie. Błędy czasu działania (ang. runtime error) mogą wystąpić w poprawnie napisanym programie, na przykład, gdy użytkownik zmienia nazwy lub przesuwa pliki zasobów. Dla błędów logicznych (ang. Logic errors), biblioteka standardowa zapewnia klasę wyjątek `std::logic_error`.

Jak radzić sobie w przypadku wystąpienia tych wyjątków? Nie można zapomnieć o obsłudze rzuconych wyjątków, ponieważ nieobsłużone wyjątki w aplikacji zastaną wychwycone przez system operacyjny i spowoduje to krytyczny błąd aplikacji.

Błędy logiczne nie mogą wystąpić w końcowej wersji aplikacji. Kontynuowanie działania aplikacji od ich wystąpienia jest niebezpieczne, ponieważ jego logika jest niepoprawna, i istnieje ryzyko błędnego jego działania, jeśli zignorujemy błąd. Co należy zrobić, jeśli wywołana zostanie funkcja load () z tym samym identyfikatorem, lecz różnymi nazwami plików? Nie wiadomo, z jakim identyfikatorem zasób jest powiązany. Jeśli później będzie trzeba uzyskać dostęp do zasobu przez jego ID, możemy dostać nieodpowiedni zasobów, a tym samym wyświetlać zły obiekt na ekranie. W przypadku błędu logicznego, program powinien natychmiast przerwać działanie.

Asercje

Makro `assert` ocenia swój argument, jeżelim ma wartość fałszywa w trybie debugowania zostaje wyzwalany punkt przerwania wstrzymując wykonanie programu i bezpośrednio wskazuje na źródło
błędu. W trybie kompilacji release, asercje są zoptymalizowane, nie tworzą narzutów związanych ze sprawdzanie błędów, które nie mogą wystąpić. Wyrażenie assert jest całkowicie usunięte w trybie release, więc używane jest tylko do kontroli błędów
a nie w celu realizacji rzeczywistej funkcjonalności.

Przykładowo asercje w programie wykorzystano w metodzie `ModelHolder::get()`, gdzie żądany model może nie istnieć w mapie. Jest to błąd logiczny: należy załadować modele, zanim zażądany zostanie do nich dostępu. W związku z tym należy sprawdzić czy znaleziono załadowany model dla danego ID używając assert:

```
DirectX::Model & ModelHolder::get(Models::ID id)
{
	auto found = mResourceMap.find(id);
	assert(found != mResourceMap.end());
	return *found->second;
}
```

Renderowanie sceny

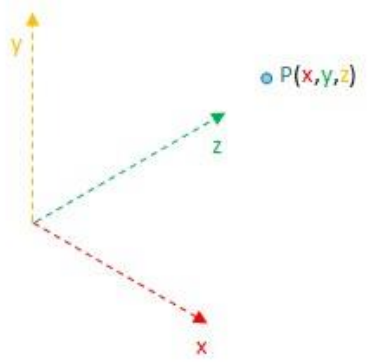
Rodział ten, przedstawia w jaki sposób gra reprezetowana zostaje na ekranie. W jaki sposób narysowane są wszystkie obiekty, elementy interfejsu. Zastosowanym rozwiązaniem jest, posiadanie rożnych sekwencyjnych kontenerów, przez którą można iterować. Każdy element posiada funkcję `Entity::draw()`, który rysuje dany dany obiekt na ekaranie. W trakcie rysowanie sceny z wykozystaniem przezroczystych obiektów, należy upewnić się, że obiekty te rysowane jako ostatnie.

Rysowanie w przestrzeni 3D

Rysowanie w przestrzeni, a właściwie zrozumienie istoty umieszczania obiektów w przestrzeni, jest kluczowe przy tworzeniu aplikacji 3D.

Punkt

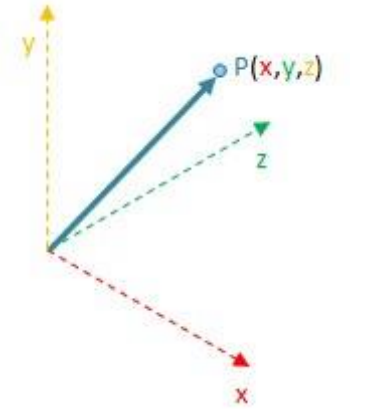
Pozycja punktu w przestrzeni opisywana jest przez trzy podstawowe składowe: x, y, z. Pierwszy i ostatni parametr określają położenie punktu na płaszczyźnie, natomiast drugi parametr określa wysokość, na jakiej ma znaleźć się punkt.



Rys. 8. Punkt w przestrzeni trójwymiarowej [11].

Wektor

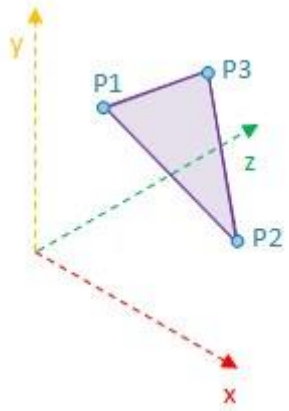
DirectX oferuje gotową klasę `DirectX::XMVECTOR` pozwalającą w bardzo prosty sposób tworzyć wektory, które mogą być wykorzystywane do różnego rodzaju transformacji obiektów w przestrzeni. Jako punkt zaczepienia przyjmuje się początek układu współrzędnych, zatem pomija się podawanie punktu początkowego (zaczepienia) wektora. Podaje się jedynie współrzędne x, y, z punktu, na który zwrócony jest wektor.



Rys. 9. Wektor w przestrzenie trójwymiarowej [11].

Vertex

Wszystkie obiekty w przestrzeni 3d tak naprawdę składają się z, przekształconych w różny sposób, obiektów dwuwymiarowych. Elementarną figurą, jest trójkąt. Trójkąt je używana ze względu na możliwość „zbudowania” z niego dowolnej bryły przestrzennej, a dokładność, z jaką zostanie ona odwzorowana, zależy od ilości trójkątów wykorzystanych przy jej tworzeniu. Trójkąt składa się z trzech punktów. W przypadku Verteksów tworzy się specjalne struktury, które oprócz współrzędnych punktów przyjmują również jako parametr informację o kolorze punktu bądź o teksturach, a także o tak zwanych wektorach normalnych.

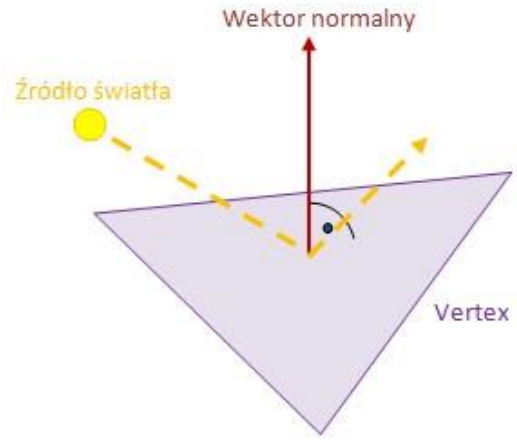


Rys. 10. Vertex w przestrzeni trójwymiarowej [11].

Wektor normalny

Wektory normalne to wektory jednostkowe (przyjmują wartości 0 lub 1), są niezbędne, aby DirectX poprawnie mógł obliczyć natężenie światła padającego na vertex. Wektor normalny musi być prostopadły do verteksa, do którego jest przypisany, jest to wymagane, aby tzw. `dotProduct` w jednostkach cieniowania karty graficznej został poprawnie obliczony (na bazie tego wektora jest obliczony kąt padania światła na vertex, od którego zależy natężenie światła padającego na dany vertex).

Całą sytuację obrazuje następujący rysunek:



Rys. 11. Wektor normalny verteksa [11].

Macierz

Macierze w DirectX należy rozumieć jako zbiór punktów w przestrzeni trójwymiarowej. Kolejne wiersze oznaczają kolejne współrzędne (x, y, z), natomiast kolumny oznaczają kolejne punkty.

Typowa macierz wygląda zatem następująco:

$$\begin{bmatrix} P_{11} & P_{12} & P_{13} \\ P_{21} & P_{22} & P_{23} \\ P_{31} & P_{32} & P_{33} \end{bmatrix}$$

Ponieważ obiekt w przestrzeni (np. Model 3d lub inny dowolny zbiór verteksów) jest opisywany przez wiele punktów, więc aby dokonać specyficznych typów transformacji, np. obrotu, przy zachowaniu stałych odległości między punktami należącymi do jednego obiektu, niezbędne są macierze. Oczywiście nic nie stoi na przeszkodzie, aby stworzyć macierz o dowolnej strukturze, dowolnej ilości kolumn czy też wierszy, co tak naprawdę sprowadzałoby się do stworzenia tablicy dwuwymiarowej, jednak w naszym przypadku omawiane są macierze w kontekście rysowania w przestrzeni w DirectX i na nich skupi się ten opis.

Do narysowania obiektów niezbędne jest określenie podstawowych macierzy. Są to: worldMatrix (Macierz świata), viewMatrix (Macierz widoku) oraz projectionMatrix (Macierz projekcji).

Macierz świata (worldMatrix)

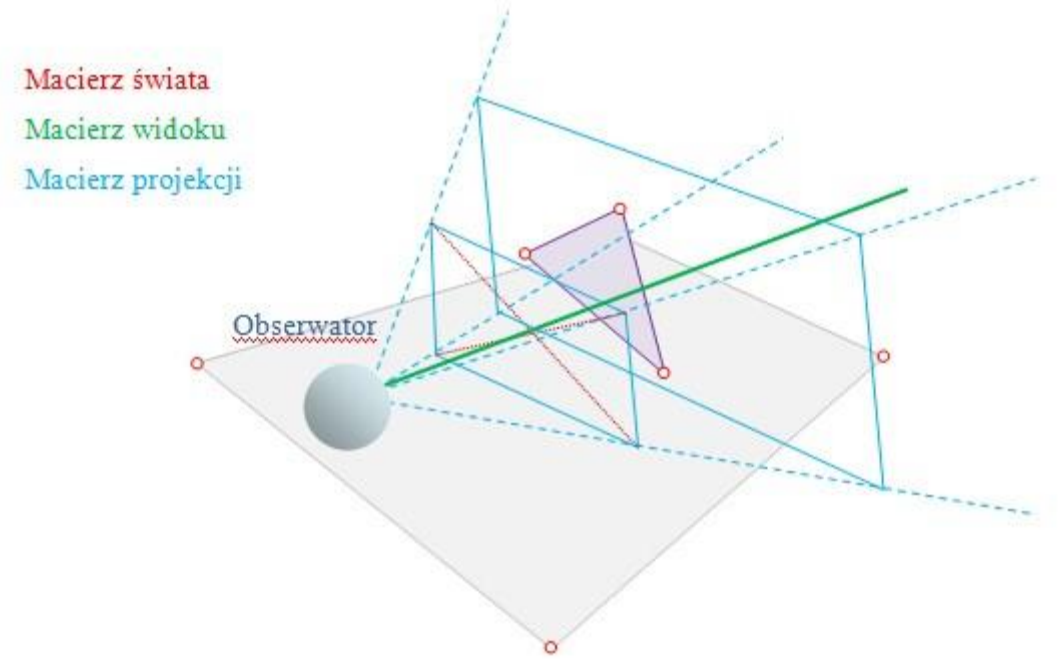
Wewnątrz tej macierzy zawarte są informacje dotyczące położenie wszystkich punktów dla danego obiektu bądź też zbioru obiektów.

Macierz widoku (viewMatrix)

Macierz ta określa kierunek, w jakim zwrócony jest obserwator (punkt w przestrzeni, na który spogląda). Macierz ta jest niezbędna, aby karta graficzna „wiedziała", który fragment sceny i w jaki sposób ma renderować (rysować).

Macierz projekcji (projectionMatrix)

Macierz ta określa obszar, który ma zostać zaprezentowany. Można to porównać do sytuacji, gdzie oczy są bardziej lub mniej zmrużone. Im bardziej otwarte powieki, tym większe pole widzenia, im bardziej powieki są przymknięte, tym mniejsze pole jest widoczne. Może ta analogia nie jest do końca zrozumiała, ale rysunek poniżej powinien wyjaśnić całą koncepcję.



Rys. 12. Graficzna reprezentacja macierzy świata, widoku i projekcji [11].

W powyższym przykładzie na scenie znajdują się dwa obiekty, o czym świadczą czerwone punkty na ich krańcach, fioletowy vertex oraz szara płaszczyzna, która została narysowana dla łatwiejszego zrozumienia problemu. Macierz widoku wskazuje na vertex. W przyszłości, jeżeli w metodzie Update() każdorazowo macierz widoku będzie aktualizowana i będzie wskazywać na obiekt, który się porusza, zaobserwować będzie można ruch kamery (zawsze wskazywany punkt – znajduje się w środku kadru).

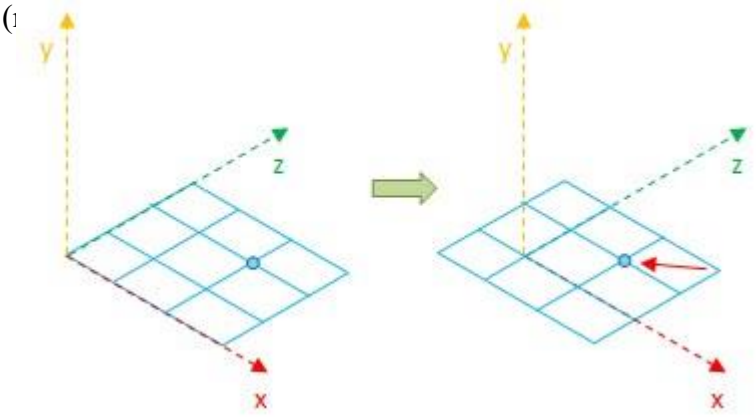
Transformacje w przestrzeni

Transformacji obiektów w przestrzenie bezpośrednio odnoszą się do teorii przedstawionej w punkcje „Rysowanie w przestrzeni”.

W przypadku transformacji wszystkie obliczenia odbywają się na macierzach.

Przesunięcie

Translacja jest przekształceniem polegającym na przesunięciu obiektu o podany wektor (1).

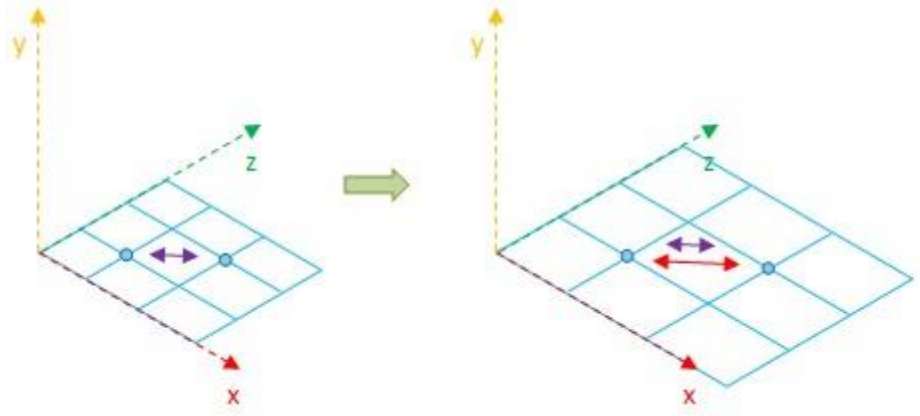


Rys. 13. Przesunięcie punktu w przestrzeni [11].

Skalowanie

Użyjemy teraz macierzy w celu wykonania transformacji skali obiektu. Co należałoby zrobić, aby przeskalować obiekt? Wystarczyłoby przemnożyć macierz, która reprezentuje położenie punktów tego obiektu, przez określoną liczbę.

Sama koncepcja może być zobrazowana następująco:



Rys. 14. Skalowanie macierzy [11].

Skalowanie jest operacją, która zmienia wielkość obiektu, wykonywana jest względem każdej z osi przyjmując odpowiedni współczynnik dla każdej z nich.

```
Scale = DirectX::XMMatrixScaling(1, 0.5f, 1);
```

Powyższa operacja stworzy w zmiennej `Scale` macierz, przez którą pomnożenie spowoduje, iż obiekt zachowa swoje rozmiary dotyczące szerokości i długości, jednak zostanie „spłaszczony” o połowę, na co wskazuje drugi parametr. Innymi słowy, wszystkie punkty w macierzy obiektu zostaną przemnożone przez:

- Dla parametru x – przez 1,
- Dla parametru y – przez 0,5,
- Dla parametru z – przez 1.

Matematycznie całe skalowanie przedstawia się następująco (dla vertexa), macierz skalowania:

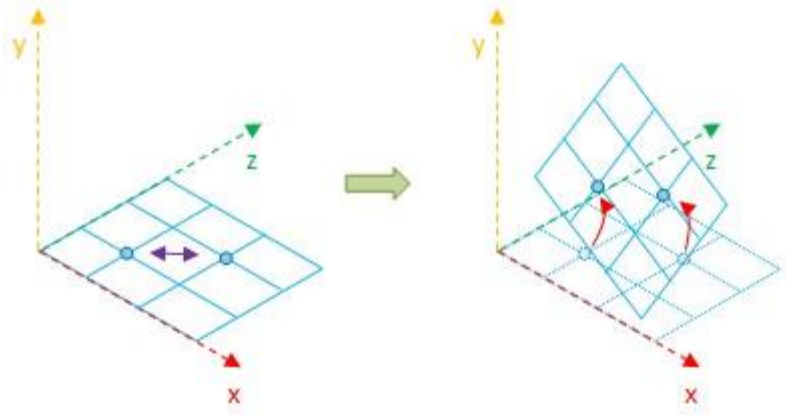
$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0{,}5 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Sama transformacja wykona się według następującego schematu:

$$\begin{aligned} X_{przeskalowany} &= 1 * X_{oryginalny} \\ Y_{przeskalowany} &= 0{,}5 * Y_{oryginalny} \\ Z_{przeskalowany} &= 1 * Z_{oryginalny} \end{aligned}$$

Obracanie

W przypadku obracania wzajemne odległości między punktami muszą zostać zachowane. Zobrazować tę transformację można następująco:



Rys. 15. Obracanie grupy punktów w przestrzeni [11].

Cała koncepcja matematyczna opisująca transformację w przestrzeni sprowadza się do przemnożenia macierzy danej (w tym przypadku będzie to punkt) przez macierz rotacji, w której umieszczone zostaną odpowiednie wartości funkcji trygonometrycznych dla kątów, o jaki użytkownik chce obrócić punkt.

Dla przykładu, obrót o 45˚ wzdłuż osi z, punktu (3, 2, 1) można matematycznie przedstawić w następujący sposób:

$$\begin{bmatrix} X_{obrócony} \\ Y_{obrócony} \\ Z_{obrócony} \end{bmatrix} = \begin{bmatrix} \cos\frac{\pi}{4} & \sin\frac{\pi}{4} & 0 \\ -\sin\frac{\pi}{4} & \cos\frac{\pi}{4} & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 3 \\ 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 0{,}707 * 3 + 0{,}707 * 2 + 0 * 1 \\ -0{,}707 * 3 + 0{,}707 * 2 + 0 * 1 \\ 0 * 1 + 0 * 1 + 1 * 1 \end{bmatrix} = \begin{bmatrix} 3{,}535 \\ -0{,}707 \\ 1 \end{bmatrix}$$

Niektóre z funkcji definiujących przekształcenia obrotu na macierzach, które oferuje DirectX:

```
DirectX::XMMatrixRotationQuaternion()
DirectX::XMMatrixRotationX()
DirectX::XMMatrixRotationy()
DirectX::XMMatrixRotationZ()
```

Praktyczne użycie wymienionych funkcji sprowadza się do podania odpowiedniego parametru (kąta w radianach) i przemnożenie aktualnej pozycji przez macierz zwróconą przez jedną z powyższych metod.

Zarządzanie obiektami sceny

Korzeń sceny

W celu zarządzania współrzędnymi relatywnymi w sposób przyjazny dla użytkownika, utworzono korzeń-sceny (ang. Scene-graph), jest to drzewiasta struktura składająca się z wielu węzłów, zwanych dalej węzłami sceny. Każdy węzeł sceny może być obiektem, który będzie rysowany na ekranie.

Każdy węzeł może mieć dowolną ilość węzłów potomnych, które adaptują przekształcenia ich węzła nadrzędnego podczas renderowania. Węzły potomne zawierają informacje o pozycji, obrocie i skali w stosunku do rodzica. Scene-graph zawiera korzeń sceny, którego instancja występuje w świecie tylko raz. Korzeń sceny znajduje się ponad wszystkimi innymi węzłami w hierarchii, nie ma węzła nadrzędnego.

Węzeł sceny

Reprezentacja węzła sceny znajdującego się w korzeniu sceny znajduje się klasie o nazwie `SceneNode`. Korzeń sceny posiada pośredni wszystkie węzły sceny, w związku z tym jest odpowiedzialny za ich cykl życia i destrukcji. Każdy węzeł sceny służy do przechowywania wszystkich jego węzłów potomnych. Jeśli węzeł zostaje zniszczony, jego potomkowie zostają zniszczeni razem z nim. Jeśli elementem niszczonym jest korzeń sceny, wtedy cała scena zostanie usunięta.

```
class SceneNode
{
public:
	typedef std::unique_ptr<SceneNode> Ptr;
public:
	SceneNode(Category::Type category = Category::None);
```

```
        void attachChild(Ptr child);
        Ptr detachChild(const SceneNode& node);
        ***
private:
        std::vector<Ptr> mChildren;
        SceneNode* mParent;
};
```

Do przechowywania dzieci, wykorzystano kontener STL `std::vector`. Nie można użyć std::`vector<SceneNode>`, ponieważ typy element musi być typem kompletnym (klasa staje się typem kompletnym na końcu jej definicji), a ponieważ klasa jest polimorficzna, oraz w celu uniknięcia ręcznego zarzadzania pamięcią użyto typu `std::vector<std::unique_pt<SceneNod>>`.

Dodawanie i usuwanie węzła

Interfejs do wstawienia lub usuwania węzłów potomnych z węzłów sceny został zaimplementowany przy pomocy dwóch przedstawionych funkcji:

```
void attachChild(Ptr child);
Ptr detachChild(const SceneNode& node);
```

Pierwsza metoda zajmuje przejmuje własność węzła sceny i dodaje węzeł do kontenera przechowującego potomków danego węzła. Przed dodaniem, ustawiona zostaje wartość wskaźnika na obiekt rodzica, którym jest węzeł, do którego aktualnie dodawany jest węzeł:

```
void SceneNode::attachChild(Ptr child)
{
        child->mParent = this;
        mChildren.push_back(std::move(child));
}
```

Druga metoda jest nieco bardziej skomplikowana. Najpierw należy znaleźć określony węzeł w kontenerze, użyto tutaj wyrażenia lambda. Algorytm `STL std::find_if()` zwraca iterator do znalezionego elementu, który należy sprawdzić, czy jest poprawny za pomocą asercji.

```
SceneNode::Ptr SceneNode::detachChild(const SceneNode& node)
{
        auto found = std::find_if(mChildren.begin(), mChildren.end(), [&]
(Ptr& p) { return p.get() == &node; });
        assert(found != mChildren.end());

        Ptr result = std::move(*found);
        result->mParent = nullptr;
        mChildren.erase(found);
        return result;
}
```

W drugiej części węzeł zostaje usunięty z kontenera oraz odpowiednie zmienne przyjmują nowe wartości. Wskaźnik do węzła nadrzędnego ustawiany jest na wartość `nullptr`.

Rysowanie wezła sceny

Klasa SceneNode reprezentuje obiekt, który zostanie narysowany na ekranie, dlatego musi udostępniać odpowiedni interfejs do jego obsługi. Klasa w tym celu posiada trzy metody:

- `virtual void Draw(ID3D11DeviceContext* dc, const Camera& camera, const CommonStates& states) const;`
- `virtual void drawCurrent(ID3D11DeviceContext* deviceContext, const Camera& camera, const DirectX::CommonStates& states) const;`
- `void drawChildren(ID3D11DeviceContext* deviceContext, const Camera& camera, const DirectX::CommonStates& states) const;`

Metoda wirtualna `Draw()` służy o obsługi rysowania obiektu, wywołane są w niej dwie kolejne funkcje. Funkcję wirtualną `drawCurrent()` implementuje algorytm rysowania aktualnego obiektu i może zostać przesłonięta prze obiekty dziedziczące po SceneNode, ponieważ obiekt `SceneNode` nie ma żadnych instrukcji rysujących.

Metoda `drawChildren()` jest częścią implementacji systemu zarzadzania obiektami, i wywołuje operacje rysowania dla obiektów potomnych węzła:

```
for (const Ptr& child : mChildren)
	child->Draw(deviceContext, camera, states);
```

Powiązanie jednostek z zasobami

Tworzenie obiektu wymaga do jego poprawnego zainicjalizowanie odpowiednich zasobów, zazwyczaj jest to trójwymiarowy model, obiektu, ale może to być również dźwięk, tekstura lub skrypty związane z daną jednostką. W tej implementacji do stworzenia obiektu potrzebny będzie tylko model 3d oraz odpowiedni typ identyfikatora zasobu. Dla modeli `Pickup` przygotowano wyliczenie `Pickup::Type`. Wyliczenie zawiera identyfikatory modeli, po jednym dla każdego z typów obiektu `Pickup`.

```
enum Type
{
	Apple,
	Banana,
	ExtraLive,
	TypeCount
};
```

Podczas konstrukcji obiektu należy pobrać model ze zmiennej `models`, będącej argumentem konstruktora odwołując się do odpowiedniego identyfikator zasobu W jej konstruktora, chcemy zainicjować ikonki z właściwą teksturę. Konstruktom mógł by przyjmować jako argument bezpośrednio wymagany model, ale podejście z przekazaniem instancji klasy zarządzającej zasobami, że wiedza na temat wykorzystywanego zasobu jest lokalna dla klasy i podczas tworzenie obiektu nie trzeba wiedzieć z jakiego modelu korzysta. Dodatkowo, jeśli

obiekt będzie wymagać więcej niż jeden zasób, to nie będzie potrzeby zmieniać prototypu konstruktora. Deklaracja konstruktora klasy Pickup:

```
Pickup(Type type, ModelHolder& models);
```

Klasa `ModelHolder`, zapewnia metodę `const DirectX::Model& get(Models::ID id) const` którą można wywołać z odpowiednim identyfikatorem, aby uzyskać odpowiadający model.

Rysowanie sceny

Warstwy sceny

Do scene-gaph dodano możliwość podziału sceny na warstwy. Podział taki pozwala na ustalenie kolejności rysowania grup obiektów oraz daje możliwość transformacji całej warstwy poprzez opisywane już spółrzędne relacyjne.

Proces grupowania można łatwo zautomatyzować przy użyciu istniejącej struktury sceny. Warstwą nazwano grupę obiektów, które są wspólnie rysowane. Wewnątrz warstwy, kolejność rysowania nie ma znaczenia. Reprezentacja warstwy jest pusty węzeł sceny, będący bezpośrednim dzieckiem korzenia sceny. Węzeł reprezentujący warstwę nie zawiera grafiki, jest tylko odpowiedziały za woje dzieci. Można przypisać odpowiednie transformacje do pewnej co w efekcie automatyczne przekształci odpowiednie węzły z warstwie. W aplikacja wykorzystano trzy warstwy: jedną dla tła, jedną dla wszystkich obiektów wygenerowanych przez moduł rozszerzonej rzeczywistości, poza obiektem reprezentującym gracza. Ostatnia warstwa zawiera transformacje obiektów do układu współrzędnych w którym znajduję się gracz i wszystkie obiekty które znajdują się w tym układzie są dodawane do tej warstwy. Ostatni element w wyliczeniu LayerCount nie jest używany w odniesieniu do warstwy; przechowuje on łączną ilość warstw.

```
enum Layer
{
	Background,
	AR,
	Board,
	LayerCount
};
```

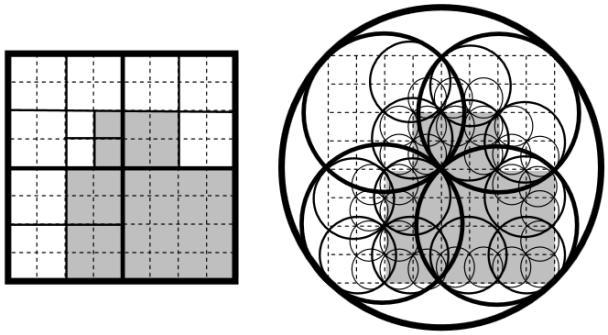
Optymalizacja rysowania

Żądanie obiektu do narysowania się na ekranie, jest jednym z najkosztowniejszych operacji jakie można zrobić w grze, w związku z tym, w każdej grze, w której ilości obiektów jest większa niż kilka, należy rozważyć kwestię wydajności związaną z rysowaniem obiektów.

Jednym ze sposobów, aby to zrobić jest użycie obcinania (ang. culling), Termin ten obejmuje szeroki zakres techniki. Zastosowanie technik obcinania daje zawsze bardzo dobre efekty optymalizacyjne i prawie zawsze warto wykonać trochę dodatkowej pracy w celu ich zaimplementowania. Obcinanie polega na sprawdzeniu czy obiekt jest w prostokącie widzenia i tylko on zostaje narysowany, jeżeli się w nim znajduje, jest to podstawowa technika obcinania.

Sceny z dużą ilością obiektów muszą stosować bardziej zaawansowane techniki obcinania obiektów. Fakt, że potrzebna jest iteracja po potencjalnie tysiącach lub milionach obiektów w każdej klatce będzie wpływać na wydajność. Z tego powodu twórcy gier zazwyczaj wdrażają jakiś rodzaj dzielenia przestrzennego. Przestrzenny podział można zrobić na wiele sposobów, ale polega to zawsze na podziale sceny na wiele komórek. Każdy obiekt będzie należeć do jakiejś komórki. Tym sposobem można obcinać całe grupy obiektów unikając iteracji po każdym z nich.

Jednym z najbardziej popularnych sposobów na podział przestrzeni jest użycie drzewa czwórkowego(quadtree). Drzewo dzieli przestrzeni na mniejsze części, dzieląc ją na cztery równe ćwiartki, a następnie każdą z tych ćwiartek na cztery itd. Hierarchiczne podejście pozwala pozbyć się iteracji dla każdego obiektu, a przeprowadzić jedynie testy w dół drzewa, jeśli wszystkie komórki przejdą test. Implementacja quadtrees zmniejsza się złożoność algorytmów z O(n) do O(log n), co jest bardzo dobrą optymalizacją. Drzewo quad jest również bardziej wydajne, jeśli chodzi o wyszukiwanie obiektów w scenie.



Rysunek 2: Przykładowa wizualizacja quad i circle trees [12]

Alternatywą do drzewa czwórkowych, mogą być drzewa kołowe (ang. circle tree), w tej metodzie komórki są opisane okręgiem. Pozwala to na inny podział obiektów i niższy koszt obliczania testów. Wybór pomiędzy wykorzystaniem jednego lub drugiego algorytmu zależy od wielu czynników. Dla każdej sceny można wybrać najbardziej odpowiedni algorytm, który daje najbardziej wydajny wynik.

Rozdzielczość, proporcje obrazu, widok

Rozdzielczość jest liczbą pikseli monitora lub okna aplikacji na wyświetlaczu. Zwykle określona jest jako szerokość x wysokość, na przykład 1024 x 768. Najczęściej rysowana scena 3D powinna zostać narysowane rysowana tak, aby zajęła cały obszar bufora tylnego. DirectX daje również możliwość, aby scena została narysowana w prostokątnym podobszarze bufora. Prostokątny obszar bufora tylnego, do którego rysowana zostaje scena, to okno widoku (ang. viewport).

Najprostszym przypadkiem użycia okna widoku, jest u użycie okna o ustalonym rozmiarze i widoku o dokładnie takich samych wymiarach, co oznacza, że proporcje widzenia i okna będą identyczne. Stosunek wyświetlania jest bardzo ważny, ponieważ, jeżeli widok nie jest proporcjonalny do wielkości okna, klient zobaczy widok, w którym świat został rozciągnięty, lub skurczony.

W przypadku tej aplikacji wykorzystujących rozszerzoną rzeczywistość, bardzo ważne jest zachowanie oryginalnego obrazu. W celu zapewnienia jak najlepszych doznań użytkownikowi aplikacji, okno aplikacji musi mieć taką rozdzielczość jak kamera, którą użyto do kalibracji. Obraz wejściowy służący do przetwarzania w czasie rzeczywistym musi mieć takie same parametry kamery, by móc poprawnie obliczyć transformacje wykrytych markerów.

Obsługa wejścia - komendy i sterowanie

Informacje o wydarzeniach

Ogólnie rzecz biorąc, wydarzenia (ang. Events) są obiektami, które są wyzwalane, gdy coś się dzieje; głównie związane jest to z prowadzonymi przez użytkownika działaniami, np. kliknięcie myszka, naciśniecie klawisza itd. Zdarzenia są konstruowane przez system operacyjny. System Windows powiadamia okna o zaistniałych zdarzeniach, posługując się komunikatami (ang. messages). Komunikaty te są odbieranie przez procedurę zdarzeniową okna, która zajmuje się ich przetwarzaniem. Praca ta jest zwykle widoczna jako odpowiednie działania programu: reakcje na kliknięcia w przyciski, przyciśnięcia klawiszy itd.

Komunikaty sterują funkcjonowaniem aplikacji i pozwalają jej działać zgodnie z oczekiwaniami programisty, użytkownika. Realizacja tych oczekiwań odbywa się drogą poprawnej współpracy z mechanizmem komunikatów Windows. Na ten mechanizm ten składa się pętla (pompa) komunikatów (ang. message loop) oraz procedury zdarzeniowe okien. Ta pierwsza zajmuje się pobieraniem od systemu informacji o zdarzeniach i kierowaniem ich do procedury zdarzeniowej; procedury zdarzeniowe: w nich następuje

odczytanie danych niesionych przez komunikat oraz ustalona przez twórcę aplikacji interpretacja zdarzenia.

Biblioteka DirectXTK posiada klasy dla wiadomości dotyczących klawiatury, myszki i joystick'a, zapewniają one warstwę abstrakcji, która jest łatwiejsza w użyciu od tradycyjnej obsługi komunikatów systemy Windows.

Aplikacja SnakeAR jest programem działającym z trybie rzeczywistym, nie czeka tylko na komunikaty i reaguje na nie, ale renderuje grafikę dla użytkownika w czasie rzeczywistym, ponieważ zmiany w grze występują cały czas zależnie od zachowania obiektów a nie tylko na reakcje użytkownika. Z tego powodu do obsługi komunikatów z systemu została użyta funkcja PeekMessage która nie czeka na pojawienie się nowego komunikatu w kolejce, jeżeli ta jest pusta zwraca wartość false [13]:

```
if(PeekMessage( &msg, 0, 0, 0, PM_REMOVE ))
{
      TranslateMessage( &msg );
      DispatchMessage( &msg );
}
else
{
      // działania aplikacji
}
```

Poniżej omówiono kilka obsługiwanych typów wiadomości.

Komunikaty okien

W tej sekcji opisano komunikaty związane z oknami, są zdarzenia, które dotyczą okien bezpośrednio.

- `WM_SIZE`: To zdarzenie jest wywoływane, gdy rozmiar okna jest zmieniany. Najczęściej jest to, gdy użytkownik przeciągnie za krawędzie okna ręcznie zmieniając jego rozmiar. Oczywiście, zmiany rozmiaru okna są możliwe tylko wtedy, gdy stworzone okno na nie pozwala (13).

- `WM_ACTIVATE`: Zdarzenia te pochodzą z okna, kiedy zyskuje lub traci fokusu (ang. fokus). Otrzymanie tego komunikatu oznacza, że użytkownik wybrał nowe okno o będzie ono otrzymać dane wprowadzone przez użytkownika. To czy okno straciło fokus czy zyskało przekazane jest w dolnym słowie parametru `WPARAM` funkcji obsługującej pętlę komunikatów. Zdarzenia te są interesujące, aby wstrzymać grę, jeśli użytkownik kliknie poza oknem lub przełączy się od innej aplikacji [13].

Komunikaty klawiatury

Klawiatura stanowi podstawowy element wprowadzania danych komputera i komunikacji z nim.

- Kiedy użytkownik naciska klawisz, aplikacja otrzymuje komunikat `WM_KEYDOWN`, w raz z nim w parametrach `WPARAM` i `LPARAM` zawarte są dodatkowe informacje, np. numerze klawisza, aktywnych klawiszach specjalnych.
- Komunikat `WM_KEYUP` jest odpowiednikiem komunikatu `WM_KEYDOWN`. Jest wyzwalany, kiedy klawisz zostaje zwolniony.

Pobieranie stanu w czasie rzeczywistym

Problem z wyżej omówionymi zdarzeniami jest taki, że są one 'ogłaszane' jeden raz, kiedy wystąpi zmiana stanu, nie można przy ich pomocy odczytać jak stan urządzeń wejściowych wygląda w rzeczywistości w danej chwili.
Można rozwiązać ten problem poprzez wykożystanie tabeli, która zapamiętuje aktualny stan klawisza. Biblioteka DirectX, aby ułatwić zarządzanie wejściem z klawiatury posiada w DirectXTK klasę `Keyboard` która ułatwia zarządzanie stanem klawiatury, posiada metody które aktualizują jej stan, pozwalają na dostęp do tych stanów w czasie rzeczywistym, kiedy i gdziekolwiek są potrzebne. Klasa `DirectX::KeyboardStateTracker` pozwala na sprawdzenie stanu danego klawisza, w podobny sposób co pętla komunikatów.

Więcej informacji na ich temat można znaleźć w dokumentacji DirectXTK [10].

Komunikaty i zdarzenia czasu rzeczywistego, kiedy są używane

Wydarzenia i sprawdzanie stanu urządzenia jest zależnie od kontekstu. Zasadą jest: jeśli trzeba odczytać, czy nastąpiła jakaś zmiana wtedy używane jest odczytanie stanu z komunikatów. Jeśli należy znać aktualny stan, trzeba użyć odczytu w czasie rzeczywistym.

Przykład:

```
Keyboard::KeyboardStateTracker tracker;
auto state = DirectX::Keyboard::Get().GetState();
tracker.Update(state);

if (state.Left)
// rób coś, jeżeli klawisz Left jest naciśniety

if (tracker.pressed.Left)
// rób coś, jeżeli klawisz Left został naciśniety
```

W pierwszym if należy coś zrobić w czasie, gdy klawisz jest wciśnięty, podczas gdy w drugiej wersji instrukcje wykonane zostaną tylko raz po naciśnięciu przycisku.

Komunikacja oparta o system komend

Wyżej zostało pokazane, jak wejścia mogą być obsługiwane. W opisywanej grze, można je obsługiwać w następujący sposób (metody snake są fikcyjne, aby pokazać zasadę działania):

```
// One-time events
Keyboard::KeyboardStateTracker tracker;
auto state = DirectX::Keyboard::Get().GetState();
tracker.Update(state);
if (tracker.pressed.Space)
      snake->ActivateShield();
}
// Real-time input
if (state.Left)
      snake->moveLeft();
if (state.Right)
      snake->moveRight();
```

Istnieje kilka problemów związanych z tym podejściem. Z jednej strony, w kodzie są na sztywno wybrane klawisze, więc użytkownik nie może wybrać WASD zamiast klawiszy strzałek do poruszania się. Z drugiej strony, osbługa wejścia i kod logiczny zgromadzony zostaje w jednym miejscu. Kod logiczny rośnie wraz z implementacją dodatkowych funkcji i sprawia, że początkowo prosty kod staje się skomplikowany.

Taki kod powinien mieć oddzieloną część logiczną i obsługę wejścia co można uzyskać implementując system komend przedstawiony w dalszej części.

Komendy

Tutaj polecenia oznaczają komunikaty, które są wysyłane do różnych obiektów w grze. Polecenie jest w stanie zmodyfikować obiekt i wydawać im rozkazy takie jak, np. poruszane obiektu, wystrzał z broni lub wywołać eksplozję. Struktura zawiera pole, które może być wysyłane przez dowolny obiekt gry reprezentowany przez węzeł sceny

```
struct Command
{
      std::function<void(SceneNode&, float)> action;
};
```

Polem struktury `Command`, jest zmienna `action` zawierająca funkcję, która implementuje rozkaz, instrukcje wydany dla danego obiektu. Pierwszym parametrem jest referencja do węzła sceny, którego dotyczy komenda. Drugi parametr oznacza różnicę czasu dla bieżącej ramki. Przykładowe użycie komendy:

```
void moveLeft(SceneNode& node, float dt)
{
      node.move(-30.f * dt 0.f);
}
```

```
Command c;
c.action = &moveLeft;
```

Za pomocą wyrażenia lambda równoważna funkcja może zostać zapisywana w następujący sposób (14):

```
c.action = [] (SceneNode& node, float dt)
{
	node.move(-30.f * dt.asSeconds(), 0.f);
};
```

Można zdefiniować dowolną operację na węźle sceny wewnątrz obiektu Command. Zaletą polecenia nad bezpośrednim wywołaniem funkcji jest abstrakcja: Nie trzeba wiedzieć, w którym węźle sceny zostanie wywołana funkcja, trzeba tylko określić działanie który jest na nim wykonane. System przekazywania wiadomości jest odpowiedzialny za dostarczenie komendy do odpowiednich odbiorców.

Kategorie

W celu zapewnienia właściwego dostarczania komunikatów, obiekty w grze są podzielone na różne kategorie. Każda kategoria to grupa jednego lub wielu obiektów, które mogą otrzymywać podobne polecenia. Na przykład, wąż gracza ma własną kategorie, wrogowie są inna, przeszkody są inną i tak dalej. Enum definiuje kategorie występujące w aplikacji. Każda kategoria za wyjątkiem `None` inicjowana liczbą całkowitą, której jeden bit ustawiony na 1, a reszta są ustawione na 0:

```
namespace Category
{
	enum Type
	{
		None = 0,
		SceneAirLayer = 1 << 0,
		PlayerSnake = 1 << 1,
		Pickup = 1 << 3,
		Obstacle = 1 << 4,
	};
}
```

Możliwe jest połączenie różnych kategorii bitowym operatorem |. Klasa SceneNode posiada wirtualną metodę, która zwraca kategorię obiektu. W klasie bazowej zwracamy Category::Scene jak kategorię domyślną:

```
unsigned int SceneNode::getCategory() const
{
	return Category::Scene;
}
```

W klasach pochodnych można przesłonić metodę `getCategory()` by zwracała odpowiednia kategorie dla obiektu. Obiekt może należeć do wielu kategorii, gdzie operator |

jest stosowany do łączenia kategorie. Z tego powodu `getCategory()` zawraca zmienną typu `unsigned int`, a nie `Category::Type`.

W grze kategorie przechowywane są jako zmienne typu `unsigned int`. Oznacza to mały problem, którego trzeba być świadomym: flagi typu integer nie są bezpieczne typowo. Liczby bez znaczenia mogą być przypisane do zmiennej przechowującej kategorię, nie tylko z przestrzeni nazw `Category`. Większym problem może stanowić pomieszanie różnych typów enum:

```
Command command;
command.category = Pickup::Apple; // tylko jabłka będę odbiorcami komend
```

Przypadkowo, `Category::Type` i `Pickup::Type` zostały pomylone, ale kod nadal kompiluje, co prowadzi do nieprzyjemnych błędów. Bezpieczeństwo typu można osiągnąć poprzez przeładowanie operatorów bitowych dla wyliczenia lub dedykowany szablon klasy `Flags<Enum>`.

Ostatecznie struktura `Command` posiada składową `category`, która przechowuje adresata polecenia:

```
struct Command
{
	Command();

	std::function<void(SceneNode&, float)>	action;
	unsigned int					category;
};
```

Domyślny konstruktor inicjuje category do Category::None. Przypisując inna wartość do niego, można dokładnie określić, który obiekt otrzyma polecenie. Przykładowo, jeżeli polecenie ma być wykonane przez wszystkie obiekty, z wyjątkiem gracza, kategoria może być ustawić odpowiednio:

```
Command command;
command.action = ...;
command.category = Category::Pickup
| Category::Obstacle;
```

Wykonywanie komend

W celu wykonania komend, funkcja musi być wywołana na obiektach odbierających komunikaty. W świecie gry, rozkazy są przekazywane do korzenia sceny, wewnątrz którego są dystrybuowane do wszystkich węzłów sceny które są odpowiednimi obiektami w grze. Każdy węzeł sceny jest zobowiązany do przesłania komend do swoich dzieci.

Metoda `SceneNode::oncommand()`, jest wywoływana za każdym razem, kiedy komenda jest przekazywana do korzenia sceny. Po pierwsze, jeśli bieżący obiekt jest odpowiedniej kategorii to można wykonać polecenie, wywołując akcje zawartą w zmiennej `command`. Druga część `onCommand()` przekazuje polecenia do wszystkich węzłów potomnych:

```
void SceneNode::onCommand(const Command& command, float dt)
{
	// Command current node, if category matches
	if (command.category & getCategory())
		command.action(*this, dt);

	// Command children
	for (Ptr& child : mChildren)
		child->onCommand(command, dt);
}
```

Kolejka komend

Mając interfejs do dystrybucji komend wewnątrz korzenia sceny, należy jeszcze zapewnić sposób transport poleceń do świata gry. Do tego celu służy klasa `CommandQueue`. Ta klasa jest bardzo prostą klasą opakowująca kolejkę poleceń. Kolejka FIFO (first in, first out) jest strukturą danych która zapewnia, że elementy, które są wkładane najpierw usuwane są w pierwszej kolejności. Tylko pierwszy element może być dostępny. Standardowa biblioteka c++ zapewnia adapter kontenera `std::deque`.

```
class CommandQueue
{
	public:
		void				push(const Command&
command);
		Command				pop();
		bool				isEmpty() const;

	private:
		std::queue<Command>		mQueue;
};
```

Zapewnia on tylko trzy metody, które bezpośrednio odpowiadają za wywołanie metod obiektu mQueue;

Klasa `World` posiada instancję `CommandQueue`. W funkcji `World::update()` wszystkie polecenia, które zostały dodane do kolejki komunikatów od ostatniej klatki są przekazywane do korzenia sceny:

```
void World::Update(float dt)
{
...
// Forward commands to scene graph
	while (!mCommandQueue.isEmpty())
```

```
            mSceneGraph.onCommand(mCommandQueue.pop(), dt);
      mSceneGraph.Update(dt, mCommandQueue);
}
```

`SceneNode::onCommand()` rozsyła komendy do wszystkich swoich potomków. Klasa `World` zawiera również funkcję pozywającą na dostęp do kolejki komunikatów z poza klasy:

```
CommandQueue & World::getCommandQueue()
{
      return mCommandQueue;
}
```

Obsługa wejścia gracza

Obsługa wejścia jest zaimplementowana w osobnej klasie nazwanej `Player`. Klasa Player zawiera dwie metody reagowania na zdarzenia, oraz na zdarzenia czasu rzeczywistego:

```
class Player
{
public:
      void handleEvent(const sf::Event& event, CommandQueue& commands);
      void handleRealtimeInput(CommandQueue& commands);
};
```

Metody te są wywoływane z klasy `SnakeApp`, wewnątrz metody `MsgProc()`. Wszystkie zdarzenia klawiatury są obsługiwane w klasie `Player`:

```
LRESULT SnakeApp::MsgProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM
lParam)
{
      CommandQueue& commands = mWorld->getCommandQueue();

      // update keyboards state
      mPlayer.WndProc(hwnd, msg, wParam, lParam);

      mPlayer.handleEvent(commands);
      mPlayer.handleRealtimeInput(commands);

      return D3DApp::MsgProc(hwnd, msg, wParam, lParam);
}
```

`Player::handleRealtimeInput()`, tworzy nową komendę w każdej klatce, gdy strzałka jest naciśnieta:

```
void Player::handleRealtimeInput(CommandQueue & commands)
{
      auto state = keyboard->GetState();

      if (state.Up) {
            // rób cos gdy wciśnięty klawisz Up
            // np.
            Command resetPosition;
            resetPosition.category = Category::PlayerSnake;
            resetPosition.action = derivedAction<Snake>([](Snake& a, float
dt) { a.accelerate(DirectX::XMFLOAT3(0.0f, 0.0f, 0.0f)); });
            commands.push(resetPosition);
```

```
	}
}
```

Dla komunikatów występujących jednowyrazowo, obsługa komunikatów jest bardzo podobna. Poniższy prosty przykład pokazuje, jak wypisywać pozycje gracza po naciśnięciu klawisza P:

```
void Player::handleEvent(CommandQueue & commands)
{
	if (tracker.pressed.P)
	{
		Command output;
		output.category = Category::PlayerSnake;
		output.action = [](SceneNode& s, float)
		{
			std::cout << s.getTransform().mTranslate.x << ","
				<< s.getTransform().mTranslate.y << ","
				<< s.getTransform().mTranslate.z << "\n";
		};
		commands.push(output);
	}
}
```

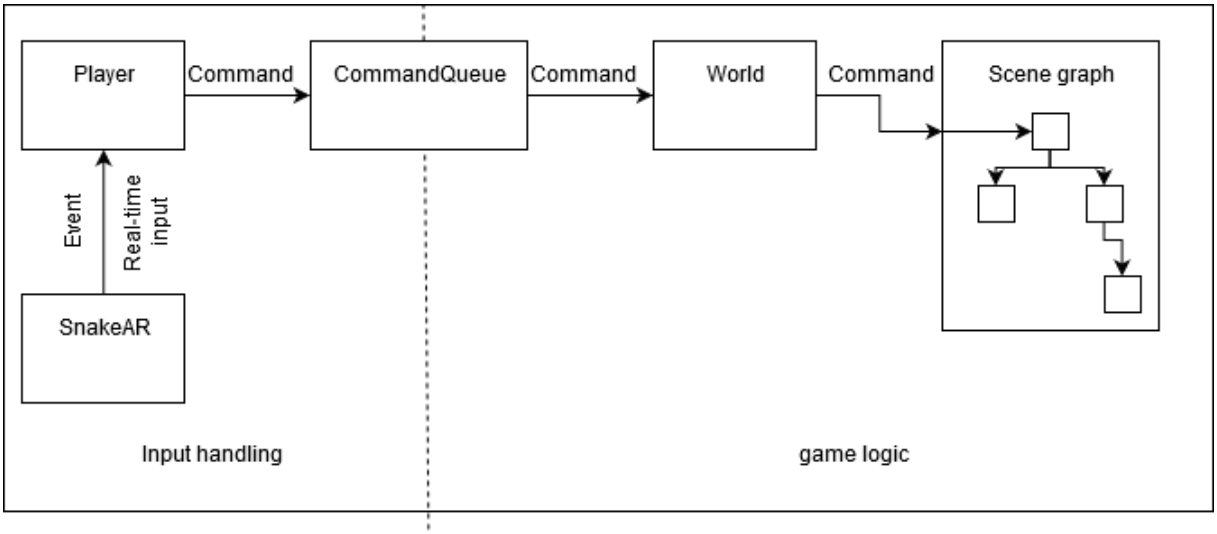
Komendy te są dodawane do kolejki komunikatów, skąd zostaną przekazane do korzenia sceny by odpowiedni obiekt mógł wykonać zapisaną w nim akcje.

Konstrukcje `std::function` oraz wyrażenia lambda, pozwalają szybko tworzyć złożone funkcje. Nie jest konieczne wykorzystywanie wyrażeń lambda, można zamiast nich używać funktorów które posiadają przeciążony `operator()`, ale w niektórych sytuacjach, lambdy pozwalają wyrazić semantykę znacznie bardziej zwarty sposób.

System komend w skrócie

Powyżej przedstawiono wiele elementów składających się na system oparty komendy, mnogość elementów sprawia, że nie rozumiejąc w pełni tego systemu łatwo można się pogubić w którymś momencie jego implementacji. Poniższy schemat daje podgląda na cała funkcjonalność. Zasadniczo system jest podzielony na część generującą komendy i część dotyczącą ich obsługi. Interesujące nas zdarzenia systemu Windows są obsługiwane przez główna klasę aplikacji (`SnakeApp`) i przekazywane do `Player`, który przekształca komunikaty wygenerowane prze system na komendy i dostarcza je do `CommandQueue`. Ten sam proces ma miejsce dla wydarzeń czasu rzeczywistego, `Player` sprawdza aktualny stan wejścia i dodaje odpowiednie polecenia do kolejki. Klasa `CommandQueue` przechowuje kolejkę poleceń i działa jako pomost między obsługą wejścia i logiki gry. Po stronie logiki gry, w klasie `World` pobierane są polecenia z CommandQueue i wysyłane zostają do głównego węzła sceny, wewnątrz którego polecenia są rozdzielane w zależności od ich kategorii. Ostatecznie, funkcje zapisanych w każdej komendzie są stosowane do

odpowiedniego obiektu gry. Poniższy schemat przedstawia przegląd systemu komunikatów i sposób jej zintegrowania z architekturą gry:



Rys. 16. System komunikatów.

Mechanizm komunikacji ogólnego przeznaczania

W trakcie procesu projektowania systemu przesyłania komunikatów, uwzględniono hermetyzacje systemu komunikatów, zależności do innych składników oraz jego wielokrotne wykorzystanie. Jedynie klasa `Player` jest bezpośrednio połączona z wejściem. Klasy `Command` i `CommandQueue` Pozwalają zaimplementować niemal dowolne działanie na węzłach sceny. Takie podejście do systemu komunikatów sprawia, że możliwe jest wykorzystanie go do innych celów, takich jak kontrola sieci lub sztuczna inteligencja. Przykładowo na potrzeby gry można używać systemu komunikatów do powiadamiania obiektów w świecie gry o wydarzeniach które mają w niej miejsce. Wszystko, co należy zrobić, to utworzyć odpowiednie polecenie, dodać go do kolejki, a zostanie ono automatycznie dostarczane do węzłów sceny.

Player jako osobna klasa

Klasa `Player` jest kontrolerem jednostek, obiektów w grze. Klasa reprezentuje wejście gracza w świecie gry. W opisywanej aplikacji, sprowadza się to jedynie do manipulowania węzłami w głównym węźle sceny.

Taka implementacja pozwala w bardziej przejrzysty sposób zaimplementować obsługę wejść dla gracza i daje ładne rozdzielenie zewnętrznych sygnałów wejściowych od gracza i logiki gry. Można w ten sposób bardziej komfortowo pracować, nie mając wielkiego blok kodu wewnątrz hierarchii obiektów.

Implementacja rozgrywki

W tym rozdziale przedstawiono istotne właściwości jakie obiektu muszą posiadać w celu implementacji rozgrywki, przedstawiono implementacje użytych klas obiektów, zaprezentowano sposób podbierania danych do ich tworzenia. W dalszej części omówiony został zaimplementowany system kolizji.

Wyposażenie jednostek

Kształtowanie świata gry wymaga obiektów które będę posiadały odpowiednie własności do implementacji ich zachowań. Instancje konkretnych obiektów będą dziedziczyć po klasie bazowej Entity, która to z kolei dziedziczy po SceneNode. Klasy SceneNode oraz Enity zawierają wspólne cechy dla wszystkich obiektów w grze, co znacznie ułatwia implementowanie nowych obiektów na ich podstawie.

Punkty życia

W celu umożliwienia zaimplementowania podstawowych elementów rozgrywki, obiekty muszą posiadać odpowiednie atrybuty. Definicji klasy Entity, posiada zmienna, która przechowuje aktualne Punkty życia (HP). Jednostka zostaje zniszczona, jak tylko HP będzie równe lub spadnie poniżej zera. Poza zmienną klasa dostarcza funkcji składowych, które pozwalają modyfikować punkty HP.

Punkty życia pozwalają na zaimplementowanie dodatkowych elementów rozgrywki, jak chociażby życie gracza, które może oznaczać, ile jeszcze szans mu pozostało.

```
class Entity : public SceneNode
{
public:
	explicit Entity(int hitpoints);

	int getHitpoints() const;
	void repair(int points);
	void damage(int points);
	void destroy();
	bool isDestroyed() const override;

private:
	int mHitpoints;
};
```

Rodzaję obiektów

Obiekty które można zbierać

W celu stworzenia graczowi możliwości poczucia postępu, oraz by gra nie polegała tylko na poruszaniu się i omijaniu przeszkód zaimplementowano klase `Pickup`. Zwykle tego typu obiekty w grach pojawiają się po zabiciu jakiegoś wroga i przynoszą różnorakie korzyści

graczowi po ich podniesieniu. Na potrzeby gry SnakeAR są one generowane w losowych miejscach przez klasę `World` reprezentującą świat, w który gracz się porusza.

Zachowanie obiektów Pickup jest proste i polega tylko na zastosowaniu odpowiedniego efektu na graczu, kiedy tylko zostanie przez niego dotknięty i zniknięciu ze sceny.

```
class Pickup : public Entity
{
public:
	enum Type
	{
		Apple,
		Banana,
		ExtraLive,
		TypeCount
	};


public:
	Pickup(Type type, ModelHolder& models);

	virtual unsigned int getCategory() const;
	virtual BoundingBox	getBoundingRect() const;

	void apply(Snake& player) const;


protected:
	void updateCurrent(float dt, CommandQueue& commands) override;
	void drawCurrent(ID3D11DeviceContext* dc, const Camera& camera, const
CommonStates& states) const override;

private:
	bool mIsMarkedForRemoval;

	Type mType;
	DirectX::Model* mModel;
};
```

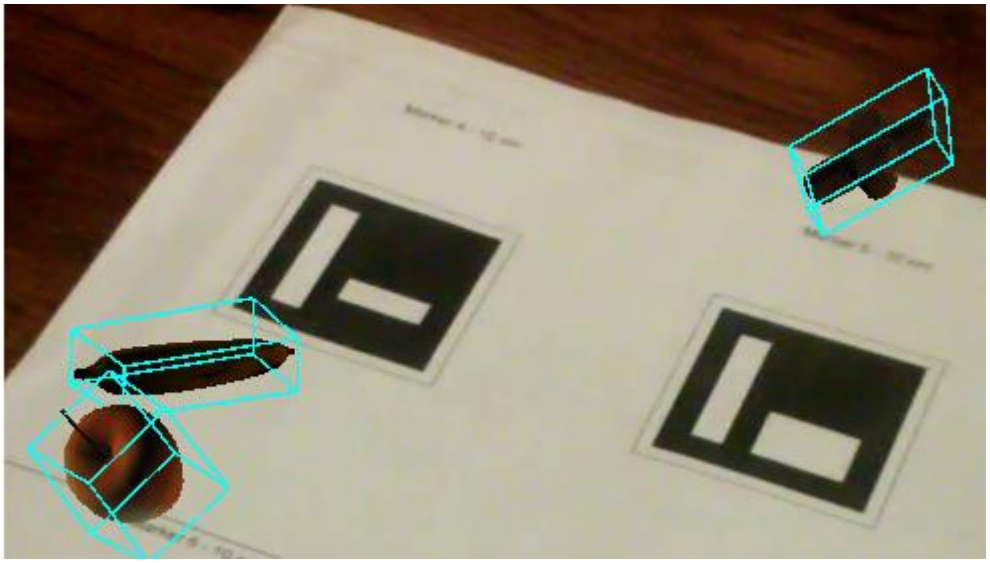
Klasa posiada przygotowaną tabele z danymi, konstruktor przypisuje do obiektu odpowiedni model 3D. Funkcja `apply()` używa tabeli do wykonania odpowiedniej akcji. Obiekt funkcyjny wykowywany jest z obiektem `Snake` jako argument. Funkcja `initializePickupData()` inicjuje tabele z danymi dla obiektów Pickup.

```
void Pickup::apply(Snake & player) const
{
	Table[mType].action(player);
}
std::vector<PickupData> initializePickupData()
{
	std::vector<PickupData> data(Pickup::TypeCount);

	data[Pickup::Apple].model = Models::Apple;
	data[Pickup::Apple].action = [](Snake& a)
```

```
        {
                a.addPoints(40);
                a.grow();
        };

        data[Pickup::Banana].model = Models::Banana;
        data[Pickup::Banana].action = [](Snake& a)
        {
                a.addPoints(80);
                a.grow();
        };

        data[Pickup::ExtraLive].model = Models::ExtraLive;
        data[Pickup::ExtraLive].action = std::bind(&Snake::addLive, _1);

        return data;
}
```

Klasa `Pickup` wywołuje zdefiniowane funkcje z klasy `Snake` która reprezentuje gracza, pozwala to na modyfikowanie jego stanu. Funkcje zdefiniowane w akcjach mogą dodawać punktów, uzupełniać życie lub wykonywać inne rzeczy, na które pozwala klasa `Snake`.

Przykładowy obraz pokazujący dwa obiekty różne obiekty Pickup (Jabłko, banan i dodatkowe życie), są one losowo generowane na płaszczyźnie, w której porusza się gracz.



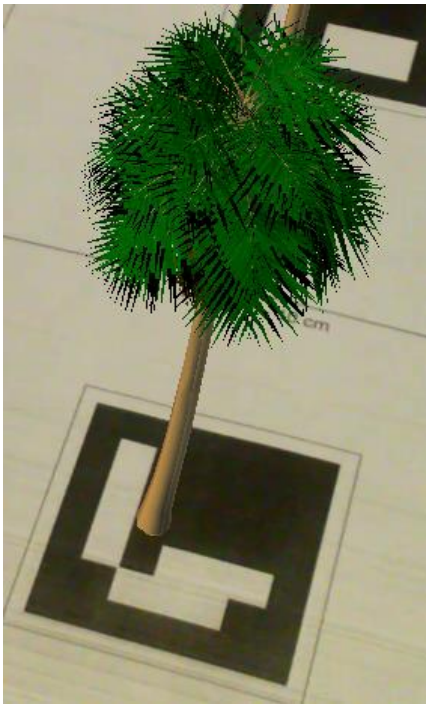
Rys. 17. Obiekty typu Pickup w świecie gry, z sześcianami wyznaczającymi kolizje.

Tworzenie przeszkód

Przeszkody są instancjami klasy Obstacle, pojawiają się na ekranie w miejscach wyznaczonych przez markery które gracz umieścił w widoku kamery. Przeszkody dodawane są do warstwy AR, w niej przechowywane są obiekty identyfikowane przez znaczniki.

Przeszkody nie są przechowywane w tej samej warstwie co znacznik oznaczający gracza, ponieważ tutaj każdy obiekt ma inna macierz transformacji, a w warstwie, gdzie znajduje się

gracz i obiekty leżące w tej warstwie, transformacja zaaplikowana jest do całej warstwy. Większość właściwości jest identyczna jak dla obiektów klasy Pickup, poza tym, że mają inną akcje przypisaną na wystąpienie kolizji.



Rys. 18. Okiekt typu przeszkoda.

Kolizje

Wykrywanie kolizji

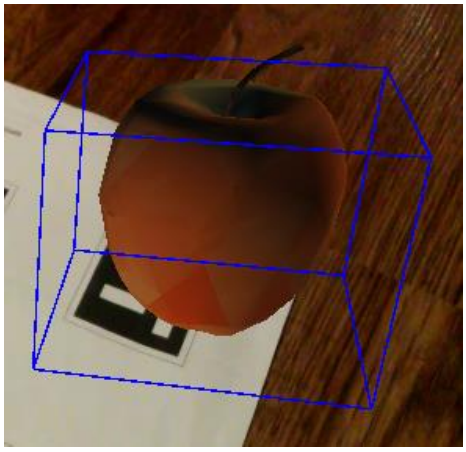
Realizacja interakcji pomiędzy obiektami w świecie gry realizowana jest poprzez system kolizji między nimi. Większość interakcji będzie miała postać kolizji: gracz dotyka przeszkodę i traci życie, gracz dotyka `Pickup` i zyskuje pewne efekty, itd.

Klasa `DirectX::Model` podczas ładowania modelu oblicza sześcian otaczający. Jest to najmniejszy możliwy sześcian, który całkowicie zawiera model. Taki sześcian (Bounding box) stanowi przybliżenie kształtu jednostki, co sprawia, że obliczenia kolizji są dużo prostsze. Przykład implementacji: getWorld() zwraca macierz przekształceń obiektu w świecie gry, na te przekształcenia składają się pomnożone macierze transformacji od korzenia sceny do danego modelu. BoundingBox:: Transform przekształca sześcian o wyliczone transformacje, może to spowodować powiększenie, jeśli obiekt został obrócony.

```
BoundingBox Pickup::getBoundingRect() const
{
	BoundingBox transformedAABB;
	XMMATRIX transforms = getWorld() * this->get;
	mModel->meshes[0]->boundingBox.Transform(transformedAABB,
transforms);
```

```
        return transformedAABB;
}
```

Aby umożliwić debugowanie kolizji i podgląd jak wyglądają takie sześciany w świecie gry, stworzona została funkcja `drawBoundingRect()` w SceneNode.cpp. W ostateczniej wersji aplikacje linia odpowiednia linia z `SceneNode::draw()` musi zostać za komentowana by nie rysowały się te dodatkowe obiekty.



Rys. 19. Sześcian kolizji obiektu.

System kolizji posiada prostą funkcję, która sprawdza czy występuje kolizja między dwoma obiektami. Funkcja sprawdza czy sześciany otaczające obiekty się przecinają. Takie uproszczenie sprawdzania modeli do jednego otaczającego sześcianu nie jest bardzo dokładne, ale łatwe w implementacji oraz wystarczająco dobre do celów aplikacji.

```
bool collision(const SceneNode & lhs, const SceneNode & rhs)
{
        return lhs.getBoundingRect().Intersects(rhs.getBoundingRect());
}
```

Szukanie kolizji obiektów

Do określenia czy kolizja występuje używamy funkcji `collision()`. Obiekty między którymi występuje kolizja przechowywane są w `std::pair<SceneNode*, SceneNode*>` dla którego zdefiniowany jest `SceneNode::typedef Pair`. Znalezione pary przechowywane są w kontenerze std::set.

Aby odnaleźć szukane pary należy porównać każdy węzeł sceny z każdym innym węzłem w scenie oraz obliczyć, czy występuje między nimi kolizja. Do wyszukiwania kolizji w sposób rekurencyjny, zaimplementowane zostały dwie metody. Pierwsza z nich to `checkNodeCollision()`, sprawdza kolizję pomiędzy `*this` i jego węzłami potomnymi, oraz węzłem sceny przekazanym jako argument funkcji.

Pierwsze trzy linie testują czy występuje kolizja oraz czy węzły sceny są identyczne (jednostka nie powinna powodować kolizji sama z sobą). Sprawdzana zostaje również wartość z wywołania funkcji `isDestroyed()` ponieważ obiekty, które zostały zniszczone, a nie są już częścią gry. Jeśli para obiektów spełnia te warunki, zostaje wstawiona do kontenera. Algorytm STL `std::minmax()` i zwraca parę która posiada mniejszy pierwszy element, a drugi większy, w tym przypadku wartościami są adresy węzłów sceny. Jeżeli obiekt A zderzy się z obiektem B, to `std::set` wraz z `std::minmax()` zapewnia, że taka para zostanie wstawiona tylko raz (nie dwie pary A-B, B-A). Druga cześć funkcji powoduje rekurencyjne sprawdzenie kolizji dla wszystkich węzłów potomnych obiektu `*this`.

```
void SceneNode::checkNodeCollision(SceneNode & node, std::set<Pair>&
collisionPairs)
{
	if (this != &node && collision(*this, node) && !isDestroyed() &&
!node.isDestroyed())
		collisionPairs.insert(std::minmax(this, &node));

	for (Ptr& child : mChildren)
		child->checkNodeCollision(node, collisionPairs);
}
```

Powyższa funkcja testuje kolizje w danym węźle sceny. Do przetestowania kolizji w dla całej sceny napisano dodatkowo funkcje `checkSceneCollision ()`:

```
void SceneNode::checkSceneCollision(SceneNode & sceneGraph, std::set<Pair>&
collisionPairs)
{
	checkNodeCollision(sceneGraph, collisionPairs);

	for (Ptr& child : sceneGraph.mChildren)
		checkSceneCollision(*child, collisionPairs);
}
```

Reagowanie na kolizje

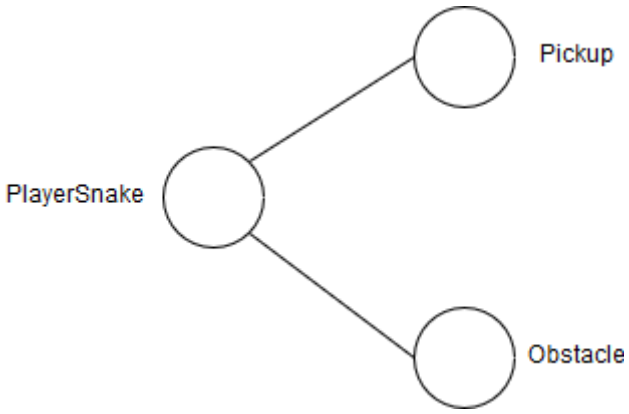
W tej części opisywane zostały jakie skutki wywiera kolizja obiektów na przebieg rozgrywki. Posiadając kontener z wszystkimi wykrytymi zderzeniami między węzłami sceny, należy na nie odpowiednie zareagować. Iterując po każdej parze węzłów, należy ustalić jakiej są kategorii. Zaimplementowano funkcję pomocniczą, która zwraca true, jeśli dane para węzłów jest odpowiednich kategorii. Funkcja dodatkowo sprawdza dopasowanie kategorii pierwszego obiektu do drugiego jak i odwrotnie. W drugim przypadku elementy pary zostają zamienione, aby odpowiadały porządkowi w jakim kategorie przekazane zostały przez argumenty fukcji.

```
bool matchesCategories(SceneNode::Pair& colliders, Category::Type type1,
Category::Type type2)
```

```
{
	unsigned int category1 = colliders.first->getCategory();
	unsigned int category2 = colliders.second->getCategory();

	if (type1 & category1 && type2 & category2)
		return true;
	else if (type1 & category2 && type2 & category1)
	{
		std::swap(colliders.first, colliders.second);
		return true;
	}
	else
		return false;
}
```

Nasz sklep rzeczywista funkcja jest obecnie raczej proste. Sprawdzimy cały wykres sceny kolizje, i wypełnić zestaw z parami kolizji. Następnie, iterację planie i rozróżniać kategorii kolizji.

```
void World::handleCollisions()
{
	std::set<SceneNode::Pair> collisionPairs;
	mSceneGraph.checkSceneCollision(mSceneGraph, collisionPairs);

	for (SceneNode::Pair pair : collisionPairs)
	{
		if (matchesCategories(pair, Category::PlayerSnake,
Category::Pickup))
		{
			// Reakcja na kolizje gracza z Pickup
		}
	}
}
```



Rys. 20. Diagram kolozji.

Aplikacja reaguje na dwa typy kolizji, do ich obsłużenia potrzebne jest wywołanie dwóch funkcji `matchesCategories()` aby obsługiwać wszystkie możliwe kombinacje. Reakcja na zderzenie gracza z obiektem typu `pickup` przebiega n następujący sposób: obiekt pickup aplikuje swój efekt na obiekt gracza. Dla przypadku kolizji gracza z przeszkodą, gracz traci życie i jego pozycja zostaje zresetowana do początkowej

```
if (matchesCategories(pair, Category::PlayerSnake, Category::Pickup))
{
	auto& player = static_cast<Snake&>(*pair.first);
	auto& pickup = static_cast<Pickup&>(*pair.second);

	pickup.apply(player);
	pickup.destroy();
}
else if (matchesCategories(pair, Category::PlayerSnake,
Category::Obstacle))
{
	auto& player = static_cast<Snake&>(*pair.first);
	auto& obstacle = static_cast<Pickup&>(*pair.second);

	player.looseLive();
	player.resetPosition();
}
```

Optymalizacje kolizji

Ponieważ zastosowano testowanie wszystkich możliwych kombinacji węzłów sceny, liczba testów kolizji rośnie kwadratowo wraz z liczbą węzłów sceny. Może się to stać wąskim gardłem aplikacji pod względem wydajności, jeśli w scenie jest bardzo wiele podmiotów. Istnieje kilka sposobów by zapobiec temu problemowi. Po pierwsze, liczba porównań może zostać zmniejszona. Rekurencja może zostać zastąpiona integracją; Jednym z możliwych rozwiązań jest napisanie klasy iterator, która przemierza drzewo sceny, pozwoliłoby to uniknąć sprawdzania czy dana kombinacja obiektów już została przetestowana oraz nie trzeba by testować, czy węzeł testowany jest sam ze sobą.

Przechowując wskaźniki do obiektów, które biorą udział z kolizjach w oddzielnych kontenerach, można również zmniejszyć niepotrzebne dodatkowe testy. Podejścia przedstawiony wyżej są dobre na początek, ale złożoność czasowa wciąż pozostaje na poziomie n^2. W dużym świecie, gdzie między obiektami występują często duże odległości, takie testowanie w większości przypadków oznacza brak kolizji, ponieważ większość z nich są zbyt daleko, aby doszło do kolizji. Optymalizacja będzie opierać się na lokalnym sprawdzaniu kolizji. Sprawdzane są tylko obiekty, które są blisko siebie. Aby to osiągnąć, świat może być podzielony na siatkę komórek równej wielkości. Każda jednostka jest przypisany do komórki. W wykrywaniu kolizji biorą udział tylko obiekty wewnątrz tej samej komórki i komórki bezpośrednio sąsiadujące, co drastycznie zmniejsza ilość wymaganych porównań. Implementacja tego sposobu porównań wymagała by użycia dodatkowych struktur danych, takich jak quadtrees. Optymalizacje mimo iż przynoszą korzyści, mają również swoje wady, do głównych z nich należy ich implementacja. Im optymalizacja jest większa, tym proces wdrażania staje się coraz bardziej skomplikowany. Sporą uwagę trzeba poświęcić, aby, np. dla siatki utrzymać jej stan zsynchronizowany ze światem. Za każdym razem, kiedy,

jednostka się porusza może przenieść się do innej komórki, więc wymagane jest jego śledzenie. Nowo utworzone podmioty muszą zostać dodane, a zniszczone podmioty muszą być usunięte z odpowiedniej komórki.

Optymalizacje kolizji stają się koniecznością, gdy kreowany świat i liczba obiektów jest duża. Da światów z niewielką ilość przedmiotów dodatkowe mechanizmy wymagane do optymalizacji są niewspółmierne do zysków, co jest powodem, dla którego zaimplementowano najprostszy mechanizm kolizji w opisywanej aplikacji.

Usuwanie obiektów

W trakcie rozgrywki obiekty zmieniają swój stan, między innymi powinny zostać usunięte ze sceny po ich zniszczeniu. Zniszczony węzłem nie zostaje usunięty natychmiast. Raz na klatkę korzeń sceny iteruje po wszystkich obiektach, sprawdza, czy zostały zniszczone i odłącza je od ich rodziców. Do testowania czy dany węzeł został zniszczony zaimplementowano wirtualną metodę `SceneNode::isDestroyed()`. Domyślnie zwraca ona wartość fałszu, ale obiekty potomne mogą określić własne warunki, w których uznane będę za zniszczone. Dla obiektów `Entity` będzie to mieć miejsce w przypadku gdy `hitpoints` będzie miała wartość zero lub mniejszą:

```
bool Entity::isDestroyed() const
{
	return mHitpoints <= 0;
}
```

Dodatkowo utworzono funkcje wirtualną, która definiuje czy dane węzeł został oznaczony do usunięcia ze sceny. Funkcja wprowadza dodatkową logikę, która pozwala obsłużyć sytuację, kiedy obiekt został zniszczony, ale nie powinien jeszcze usunięty ze sceny. Domyślnie węzeł zostaje oznaczony do usunięcia, jeśli jest zniszczony.

```
bool SceneNode::isMarkedForRemoval() const
{
	return isDestroyed();
}
```

Usuwanie obiektów z sceny ma miejsce w metodzie World::update() i rozpoczyna się od korzenie sceny. Proces usuwania obiektów reprezentowany jest przez poniższy kod:

```
void SceneNode::removeWrecks()
{
	auto wreckfieldBegin = std::remove_if(mChildren.begin(),
mChildren.end(), std::mem_fn(&SceneNode::isMarkedForRemoval));
	mChildren.erase(wreckfieldBegin, mChildren.end());

	std::for_each(mChildren.begin(), mChildren.end(),
std::mem_fn(&SceneNode::removeWrecks));
}
```

W pierwszej części, `std::remove_ if()` modyfikuje kontener, tak że wszystkie aktywne węzły znajdują się na początku, a te oznaczone go usunięcia zlokalizowane są na końcu. Wywołanie funkcji `erase()` niszczy obiekty ze sceny. W drugiej części, funkcji czynność jest wywołana rekurencyjne dla pozostałych węzłów.

Zakończenie

Podsumowanie

Niniejsza praca była dobrą okazją do zweryfikowania swoich umiejętności, zdobycia dodatkowego doświadczania oraz przemyśleń dotyczących programowania, przetwarzania grafiki i gier komputerowych.

W części teoretycznej pracy chciałem pokrótce przedstawić tematykę rozszerzonej rzeczywistości oraz gier wideo. Główną część pracy stanowi praktyczna realizacja projektu, przedstawia główne algorytmy i struktury danych wykorzystane przy implementacji projektu.

Programowanie gier komputerowych jest dość specyficzną dziedziną programowania. Z uwagi na bardzo szeroki zakres problemów jakie stawianą są przed programistą. Choć tworzenie gier często sprowadza się do podobnego procesu, i wykorzystywania podobnych wzorów projektowych to każdy projekt jest inny i wymaga często innego podejścia do różnych problemów.

Bibliografia

[1]. Luna Frank. *Introduction to 3D GAME PROGRAMMING WITH DIRECTX 11.* Dulles : David Pallai, 2012.

[2]. *In Vivo versus Augmented Reality Exposure in the Treatment of Small Animal Phobia: A Randomized Controlled Trial.* Botella Cristina i inni. 2010.

[3]. *Augmented reality technologies, systems and applications.* Carmigniani Julie i inni. 2011.

[4]. Bartosik Daniel. Kamienie milowe w grach. *CD-Action.* 2010, 11.

[5]. Kluska Bartłomiej. *Dawno temu w grach.* Łódź : Inicjatywa wydawnicza ORKA, 2008.

[6]. False positives and false negatives - Wikipedia. [Online] [Dostęp: 06.3.2017.] https://en.wikipedia.org/wiki/False_positives_and_false_negatives.

[7]. Baggio Daniel Lélis i inni. *Mastering OpenCV with Practical Computer Vision Projects.* Birmingham : Packt Publishing Ltd., 2012.

[8]. Odległość Hamminga – Wikipedia, wolna encyklopedia. *Wikipedia, wolna encyklopedia.* [Online] 5 06 2016. https://pl.wikipedia.org/wiki/Odleg%C5%82o%C5%9B%C4%87_Hamminga.

[9]. Bradski Gary i Kaehler Adrian. *Learning OpenCV: Computer Vision with the OpenCV Library.* Sebastopol : O'Reilly Media, 2008.

[10]. Walbourn Chuck. Home · Microsoft/DirectXTK Wiki · GitHub. *How people build software · GitHub.* [Online] 14 Styczeń 2017. [Dostęp: 18.02.2017.] https://github.com/Microsoft/DirectXTK/wiki.

[11]. Pośliński Dawid. Wstęp do grafiki 2D/3D. *Nauka programowania z Microsoft Developer Network | MSDN.* [Online] 16 02 2011. [Zacytowano: 3 3 2017.] https://msdn.microsoft.com/pl-pl/library/wstep-do-grafiki-2d-3d.aspx.

[12]. *The circle tree – a hierarchical struct ure for efficient storage, access and multi-scale representation of spatial data.* Moore Antoni. Dunedin : brak nazwiska, 2002.

[13]. Kuczmarski Karol. (3.2) Anatomia okna. [Online] 2004. [Dostęp: 15.02.2017.] http://xion.org.pl/files/texts/mgt/html/3_2.html.

[14]. Lambda expressions (since C++11). [Online] [Dostęp: 7.3.2017.] http://en.cppreference.com/w/cpp/language/lambda.

[15]. Moreira Artur, Haller Jan i Hansson Henrik Vogelius. *SFML Game Development.* Birmingham : Packt Publishing Ltd., 2013.

[16]. Laganière Robert. *OpenCV 2 Computer Vision Cookbook.* Olton : Packt Publishing Ltd., 2011.

Spis rysunków: