Zachodniopomorski Uniwersytet Technologiczny w Szczecinie

Wydział Informatyki

Wojciech Pardel

nr albumu: 26815

kierunek studiów: Informatyka

specjalność: Systemy komputerowe i oprogramowanie

forma studiów: stacjonarne pierwszego stopnia

PORÓWNANIE FRAMEWORKÓW DO TWORZENIA GIER SIECIOWYCH W JĘZYKU JAVASCRIPT.

COMPARISON OF FRAMEWORKS FOR CREATING ONLINE GAMES IN JAVASCRIPT.

praca dyplomowa inżynierska

napisana pod kierunkiem:

dr inż. Tomasza Wiercińskiego

Katedra Inżynierii Oprogramowania

Data wydania tematu pracy: 26.01.2015

Data złożenia pracy: 30.04.2016

Szczecin, 2016

OŚWIADCZENIE
AUTORA PRACY DYPLOMOWEJ

Oświadczam, że praca dyplomowa inżynierska pn.

Porównanie frameworków do tworzenia gier sieciowych w języku javascript.....
.....

napisana pod kierunkiem:
dr inż. Tomasza Wiercińskiego .....

jest w całości moim samodzielnym autorskim opracowaniem sporządzonym przy wykorzystaniu wykazanej w pracy literatury przedmiotu i materiałów źródłowych.
Złożona w dziekanacie Wydziału Informatyki

treść mojej pracy dyplomowej w formie elektronicznej jest zgodna z treścią w formie pisemnej i graficznej.

Oświadczam ponadto, że złożona w dziekanacie praca dyplomowa ani jej fragmenty nie były wcześniej przedmiotem procedur procesu dyplomowania związanych z uzyskaniem tytułu zawodowego w uczelniach wyższych.

Wojciech Pardel
……..
podpis dyplomanta

Szczecin, dn. 30.04.2016

Spis treści

Rozdział 1

Wprowadzenie

1.1 Wstęp i cel pracy

Gry komputerowe w dzisiejszych czasach są nierozłączną częścią rozrywki elektronicznej i edukacji. Ich rosnąca popularność wymaga od twórców tworzenia coraz bardziej obszernych programów. Problemem większości osób zajmujących się tworzeniem gier jest ogrom prac jaki należy włożyć aby stworzyć taki program kompletnie od podstaw.Jest to często niewykonalne dla zespołów które nie posiadają dużych zasobów finansowych. Silniki gier powstały aby zaadresować ten problem. Implementują one bowiem podstawową funkcjonalność taką jak efekty graficzne, dźwiękowe, fizyczne,obsługę różnych systemów operacyjnych czy kwestie związanie z sztuczną inteligencją. To właśnie duża liczba systemów operacyjnych jak i również API graficznych doprowadziła do popularyzacji i rozwoju silników gdyż umożliwiają one łatwiejsze portowanie programów na inne platformy. W niniejszej pracy inżynierskiej celem jest porównanie dwóch wiodących framewoków na platformę HTML5 wykorzystującą API graficzne WebGL jakimi są Threee.js i Babylon.js. W tym celu zdefiniujemy czym dokładnie jest silnik gry i przyjrzymy się jego poszczególnym elementom. Następnie przeanalizujemy dostępność poszczególnych elementów w wymienionych silnikach i porównamy je pomiędzy sobą. Weźmiemy również pod uwagę kwestię dokumentacji, licencji na jakiej silnik jest udostępniany. Finalnie po przeanalizowaniu wszystkich wad i zalet danych frameworków dokonam w nich implementacji prostej gry z gatunku perspektywy pierwszej osoby.

1.1.1 Introduction

Computer games nowadays are an integral part of electronic entertainment and education. Their increasing popularity requires developers to create more and more extensive programs. The problem experienced by most people involved in developing games is a huge amount of work that must be done to create such program completely from scratch, which is often not possible for teams that do not have large financial resources. Game engines were created to solve that problem because they implement basic functionality such as: visual effects, sound, physical, support for diffrent operating systems or issues related to artificial intelligence. It is a large number of operating systems as well as graphics APIs that led to the popularization and development of engines because they allow easier porting programs to other development platforms.In this paper I will compare the two leading game frameworks for HTML5 development platform and WebGL graphics api which are Threee.js and Babylon.js. For this purpose, we define exactly what a game engine is and look at the individual elements. Then we analyze the availability of individual components in these engines and compare them with each other. We will also take into consideration the issue of documentation and licenses for which the game engine is available. Finally, after examining all the pros and cons of this game frameworks I will use them to implement a simple first person perspective game.

1.2 Silnik Gry

Silnikiem gry czy frameworkiem nazywamy zbiór wbudowanych modułów takich jak silnik graficzny, silnik fizyczny, system wykrywania kolizji, algorytmy sztucznej inteligencji czy obsługę sieci. Taki silnik projektuje się często w zgodzie z paradygmatem programowania obiektowego OOP najczęściej, choć niekoniecznie w języku C++ ze względu na jego wydajność w porównaniu do języków wyższego poziomu takich jak Java[21, 38].

1.3 Silnik Graficzny

Silnik graficzny jest częścią silnika gry odpowiadającą za wyświetlanie obrazu dwuwymiarowego na podstawie trójwymiarowych danych wejściowych nazywanymi sceną 3D. Silnik musi być w stanie wyświetlać obraz określoną liczbę razy na sekundę aby przedstawić płyny ruch obiektów. Zabieg ten jest nazywany grafiką ruchomą bądź grafiką czasu rzeczywistego. Przyjmuje się że do wiarygodnego i komfortowego odbierania grafiki ruchomej potrzeba jest zdolność

wyświetlania trzydziestu klatek obrazu na jedną sekundę[39]. Proces zamiany sceny 3D na obraz 2D nazywamy potokiem graficznym czyli sekwencyjną listą czynności jaką należy wykonać aby wygenerować reprezentacje obiektów 3D w postaci dwuwymiarowej. Potok graficzny dzielimy głównie na 3 etapy: aplikacyjny, przetwarzania geometrii i rasteryzacji. Etap aplikacyjny jest przetwarzany za pomocą procesora i kontrolowany jak sama nazwa wskazuje przez aplikację. Zazwyczaj zadaniem tego etapu jest wykrywanie i obsługa kolizji pomiędzy obiektami, obliczanie fizyki obiektów czy obsługa urządzeń wejścia/wyjścia. Etap przetwarzania geometrii następuje po przekazaniu przez etap aplikacyjny prymitywów geometrycznych. Odpowiedzialny jest za poprawne rozmieszczenie prymitywów oraz usunięcie niewidocznych dla kamery wierzchołków.Składa się on z następujących operacji[46, 28] :

- Transformacji modelu i widoku polegającą na poprawnym umieszczeniu modeli i kamery w globalnym układzie współrzędnych. Początkowo model jakiegoś obiektu znajduje się w lokalnym układzie współrzędnych co oznacza że nie został poddany żadnym transformacją. Transformacja modelu polega na jego przeniesieniu do globalnego układu współrzędnych w którym znajdują się inne obiekty, co pozwala im na interakcje między sobą[46].

- Cieniowanie wierzchołków polega na nałożeniu na każdy wierzchołek danych niezbędnych do poprawnego wyliczenia oświetlenia i materiałów np. odbite światło lub tekstury. Oświetlenie jest obliczane za pomocą tak zwanego równania oświetlenia które jest zależne od wybranego typu modelu np. model Phonga lub cieniowanie Gourauda[46].

- Rzutowanie tworzy zakres widoku obserwatora (wirtualnej kamery) na scenie trójwymiarowej. Istnieją dwa rodzaje rzutowania ortogonalne i perspektywiczne. Zasięg widzenia rzutowania ortgonalnego jest idealnym sześcianem co sprawia że obiekt niezależnie od odległości zachowuje swoje proporcje. Rzutowanie perspektywiczne jest używane przede wszystkim w grach, gdyż najwierniej oddaje pole widzenia normalnego człowieka - oznacza to że w tym rzutowaniu obiekty położone dalej będą mniej widoczne niż te położone bliżej obserwatora a niektóre kompletnie niewidoczne gdy znajdują się poza określonym kątem[46].

- Obcinanie jest techniką mającą na celu optymalizacje tak aby karta graficzna nie renderowała niewidocznych dla wirtualnej kamery obiektów[46].

- Mapowanie do współrzędnych rastrowych odpowiada za przeniesienie współ-

rzędnych obiektów które znajdują się w zasięgu widzenia wirtualnej kamery na współrzędne, odpowiadające im w oknie w którym nastąpi wyświetlenie renderowanego obrazu[28].

Ostatnim etapem w przetwarzaniu grafiki jest rasteryzacja. Zadaniem tego etapu posiadającego wszystkie dane na temat każdego wierzchołka (pozycja, kolor, połączenie z innymi wierzchołkami) jest obliczenie, które piksele zostaną wypełnione odpowiednim kolorem i wypełnienie tych pikseli. Rasteryzację można podzielić na cztery etapy [46]:

- Inicjalizacje Trójkątów jak sama nazwa wskazuje w tym momencie następują podstawowe obliczenia które potem będą potrzebne do wypełnienia pikseli odpowiednimi kolorami w procesie wypełniania trójkątów[28].

- Wypełnianie trójkątów ma za zadanie określić czy dany piksel należy do trójkąta A czy B. Jeżeli piksel należy do jakiegokolwiek trójkąta, generowany jest dla niego tzw. fragment. Fragmentem nazywamy zbiór danych które w następnym etapie zostaną użyte do obliczania koloru, przezroczystości danego piksela. Generowanie fragmentów odbywa się na podstawie danych interpolowanych z każdego wierzchołka danego trójkąta [46, 40].

- Cieniowanie Pikseli w tym przedostatnim etapie wykorzystujemy fragmenty do obliczania koloru danego piksela który zostanie przekazany do finalnego etapu. Ponadto specyfika dzisiejszych rozwiązań pozwala na dowolne programowanie efektów za pomocą specjalnych programów nazywanych shaderami pisanymi w języku GLSL[46, 40].

- Merging jest finalnym etapem w którym kolor z poprzedniego etapu został zapisany w buforze koloru. Bufor ten reprezentowany jest przez tablicę w której znajdują się trzy wartości kolorów : czerwony, zielony i niebieski które w połączeniu określają kolor piksela. Drugim poważnym zadaniem stojącym przed tym etapem jest tzw. testowanie widoczności które polega na tym aby w buforze kolorów znalazły się tylko dane dla pikseli, które są bliżej wirtualnej kamery. Jednakże nie jest to końcem możliwości gdyż merging pozwala na nakładanie dodatkowych efektów takich jak, przezroczystość (alpha chanell), selektywne wyświetlanie wybranych pikseli (stencil buffer), nakładanie na siebie kilku różnych obrazów aby osiągnąć efekt rozmycia (accumulation buffer)[46].

Kończąc omawianie podstaw silnika graficznego należy wspomnieć że przedstawiony wyżej potok graficzny jest bazą dla innych api graficznych, które stosują

własną implementację zawierającą często dodatkowe opcjonalne funkcjonalności jak shadery geometryczne czy technikę teselacji[46, 28].

1.4 Silnik Fizyczny

Silnik Fizyczny jest częścią silnika gry odpowiedzialną za symulowanie zachowań obiektów zgodnie z prawdziwymi prawami fizycznymi. Silniki fizyczne dzielą się na dwa rodzaje: czasu rzeczywistego i wysokiej precyzji. Silniki czasu rzeczywistego dostarczają wyniki symulacji obiektów w czasie rzeczywistym kosztem uproszczonych symulacji realnych praw fizyki, jak i również przybliżeniem wyników takich kalkulacji[19].

1.5 Sztuczna Inteligencja

Sztuczna Inteligencja odpowiada w grach komputerowych za symulowanie realistycznych zachowań ludzkich w określonych sytuacjach. Wykorzystywane są w tym celu specjalne algorytmy takie jak, sieci neuronowe których używa się do tworzenia inteligentnych przeciwników będących w stanie do podejmowania decyzji na podstawie określonych danych wejściowych. Ważnym zagadnieniem jest również pathfinding czyli zdolność znalezienia przez komputer ścieżki pomiędzy punktem A i B biorąc pod uwagę liczne przeszkody czy inne warunki zdefiniowane przez twórcę. W ninejszej pracy omawiając zagadnienie AI skupimy się głównie na problemie pathfindingu ze względu na charakter projektu zaimplementowanej gry[47].

1.6 Podstawowe definicje

W tym podrozdziale zostaną wyjaśnione podstawowe pojęcia związane z projektowaniem gier, których znajomość będzie kluczowa do zrozumienia przyszłych zagadnień.

1.6.1 Pętla Gry

Pierwszym ważnym zagadnieniem jest pojecie nazywane pętlą gry. Każda gra komputerowa jest w zasadzie interaktywną dynamiczną symulacją dziejącą się w czasie rzeczywistym. Ponieważ gra jest symulacją jej parametry muszą być na bieżąco aktualizowane. Służy do tego właśnie pętla gry, w której zmieniamy zmienne odpowiedzialne za np. położenie gracza i przeciwnika, obecne punkty,

obliczanie algorytmów fizycznych / sztucznej inteligencji. Opisane powyżej rzeczy są również nazywane logiką gry, gdyż odpowiadają za całą warstwę logiczną aplikacji, czyli za to, co się wyświetli lub wydarzy w świecie gry[23].

1.6.2 Ilość klatek na sekundę (FPS)

Pojęcie FPS najlepiej jest wytłumaczyć w taki sposób że gra jest symulacją uwarunkowaną pewnymi zmiennymi, a jej wynik jest prezentowany na ekranie. Zmienne te zmieniają swoje wartości w pewnym odstępach czasowych często nieregularnie. Ilość klatek na sekundę mówi nam ile maksymalnie takich zmian może zostać zaprezentowanych na monitorze w ciągu jednej sekundy. Zdolność ta jest powiązana bezpośrednio z częstotliwością odświeżania obrazu monitora czyli ile razy na sekundę dany monitor jest w stanie wyświetlić obraz. Oznacza to że posiadając monitor o częstotliwości odświeżania 60 Hz gra będzie mogła osiągnąć maksymalną wielkość FPS na poziomie 60 klatek gdyż stosunek częstotliwości do FPS wynosi 1:1[24].

1.6.3 Javascript

Javascript jest skryptowym językiem programowania wykonywanym po stronie klienta opracowanym pierwotnie przez firmę Netscape do wykorzystania w projektowaniu dynamicznych stron internetowych. Na jego podstawie pod koniec lat 90 - tych organizacja ECMA (European Association for Standardizing Information and Communication Systems) opracowała specyfikację obiektowego języka skryptowego o nazwie ECMAScript. Wykorzystuje się go głównie do sprawdzania poprawności formularzy lub tworzenia interfejsów użytkownika dla aplikacji webowych[18, 26].

Rozdział 2

Przedstawienie i porównanie wybranych framewokrów

W tym rozdziale przyjrzymy się bliżej silnikom three js i babylon js oraz porównamy je wykorzystując poniższe kryteria. Ocena będzie polegać na porównaniu realizacji danego kryterium i w zależności który silnik realizuje je lepiej, zostanie przyznany mu jeden punkt a drugiemu zero. Na koniec wystawię swoją własną ocenę każdemu frameworkowi w zakresie 0 – 5 punktów. Finalnie podsumujemy punkty i przekonamy się który silnik jest lepszy.

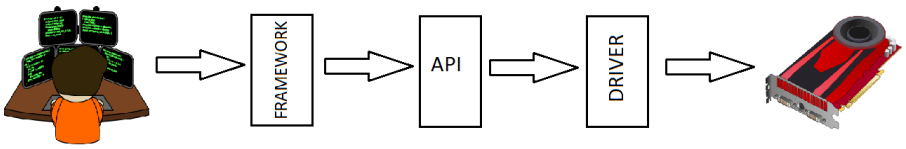
2.1 Przedstawienie porównywanych silników

Pierwsza wersja Babylon js ukazała się 5 lipca 2013r. Silnik został stworzony przez Davida Catuhe, obecnie zaś jest aktywnie wspierany przez społeczność programistów. Główną ideą powstania silnika było uproszczenie tworzenia gier trójwymiarowych na platformę HTML5 wykorzystującej API WebGL[8].

Ricardo Cabello autor silnika Three js stworzył go głównie do generowania animowanej grafiki trójwymiarowej. Pierwsza wersja frameworka zadebiutowała 24 kwietnia 2010r. Od czasu premiery silnika ponad 390 współautorów połączyło siły w jego dalszym rozwoju[45].

2.2 Funkcjonalności graficzne

Jak wiemy z powyższych przedstawionych prze zemnie informacji każdy framework posiada swój silnik graficzny, który wyświetla obiekty na ekranie. Zajmuję się tym karta graficzna z którą głównie komunikuje się sterownik, zazwyczaj stworzony przez producentów danego układu graficznego. Aby ułatwić programowanie stworzono API które udostępnia dla programistów funkcje upraszczające proces generowania grafiki 3D. Oznacza to że programista poprzez API definiuje wierzchołki sześcianu, następnie sterownik analizuje wywołania API i wykonuje większość żmudnej pracy, jak komunikacja pomiędzy podzespołami sprzętowymi czy zarządzanie pamięcią. Framework musi implementować poszczególne algorytmy graficzne takie jak SSAO czy antyaliasing metodą FXAA i używać wybranego API graficznego, aby karta graficzna mogła generować żądany obraz. Inną możliwością jest obsługa shaderów która pozwala na rozszerzenie niektórych funkcjonalności graficznych ze względu na zastosowanie w dzisiejszych czasach programowalnego potoku graficznego o którym wspomnimy w dalszej części pracy[35].



Rysunek 2.1: Wizualne przedstawienie implementacji funkcjonalności graficznych frameworka poprzez API używające sterownika do kontrolowania karty graficznej. [Źródło : Opracowanie Własne]

2.2.1 Rodzaje Oświetlenia

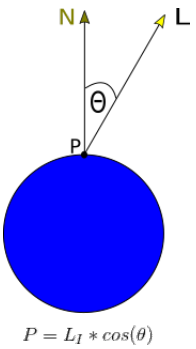
Światło w czysto naukowej definicji jest po prostu falą elektromagnetyczną lub strumieniem cząstek nazywanych fotonami. Na potrzeby grafiki trójwymiarowej przyjmuje się dwa podejścia do symulowania zachowania światła są to [27]:

- Model oświetlenia lokalnego w którym światło dociera do danego obiektu prosto ze swojego źródła[27].
- Model oświetlenia globalnego w którym światło docierające do danego obiektu pochodzi bezpośrednio ze źródła i od innych powiechrzni od których światło to zostało odbite[27].

Wyróżniamy również trzy rodzaje światła oświetlającego obiekty:

- Ambient inaczej światło otaczające jest uproszczeniem modelu oświetlenia globalnego. Światło oświetla każdy obiekt na scenie równomiernie ze wszystkich kierunków. W porównaniu z oświetleniem globalnym nie śledzimy światła odbitego od innych powierzchni tylko przyjmujemy konkretny kolor który będzie równomiernie oświetlał obiekty niezależnie od kierunku lub pozycji obserwatora[36].

- Diffuse zwane również jako światło rozproszone. Światło to pada na obiekt z określonego kierunku ale ze względu na specyfikę materiału obiektu zostaje pochłonięte i odbite w różnych kierunkach. Światło takie może mieć inny kolor ponieważ pod wpływem materiału część parametrów fali reprezentującej światło uległa zmianie. Ponadto część oświetlonej powierzchni będzie jaśniejsza w zależności od kąta pod którym światło uderza w obiekt[36]. Zależność tą definiuje prawo Lamberta które mówi że „*ilość światła docierającego do powierzchni jest proporcjonalna do kosinusa kąta między normalną do powierzchni i kierunkiem do źródła światła*"[29].



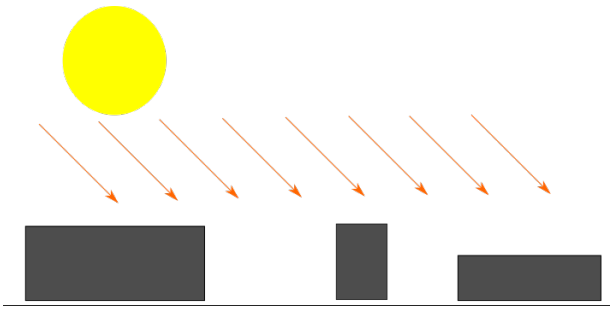
Rysunek 2.2: Ilustracja przedstawia wizualizacje prawa lamberta gdzie jasność w punkcie P zależy od intensywności (Li) światła L pomnożonego przez kosinus kąta pomiędzy wektorem światła L do wektora normalnego powierzchni N. [Źródło : Opracowanie Własne]

- Specular znane jako światło odbite. Specyfiką tego rodzaju oświetlenia jest to że światło oświetlające pewien punkt na obiekcie jest odbijane w jednym konkretnym kierunku. Ponadto światło odbite powoduje wystąpienie jasnej plamy na oświetlanym obiekcie zwanej odbłyskiem[36].

W realnych warunkach żadne światło nie składa się tylko z jednego opisanego wyżej rodzaju, najczęściej jest to kombinacja wszystkich trzech. Dla przykładu załóżmy na chwilę że w ciemnym pokoju włączamy latarkę i kierujemy strumień

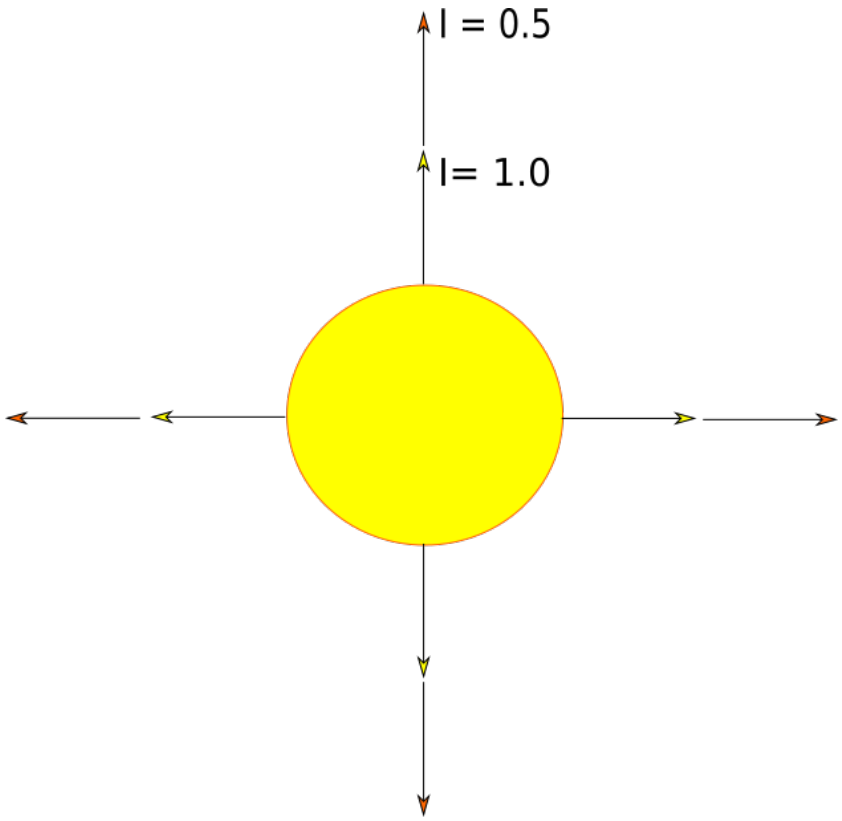
światła na metalowy talerz. Miejsce gdzie następuje kontakt z powierzchnią zauważymy jasny punkt jest nim właśnie odbłysk. Ponadto światło odbiję się od talerza i na suficie zauważmy okrąg świetlny. Ponieważ światło wciąż będzie odbijać się od kolejnych przedmiotów czy nawet cząsteczek kurzu znajdujących się w powietrzu cały pokój będzie nieco jaśniejszy. Należy również pamiętać o tym że efekt jaki zaobserwujemy będzie zależał od typu powierzchni na którą pada światło. Materiał powierzchni danego obiektu może absorbować część padającego światła i emitować je w zmienionej postaci poprzez modyfikację parametrów fali elektromagnetycznej reprezentującej światło. Oczywiście aby wszystkie opisane powyżej efekty miały szanse zaistnieć potrzebny jest obiekt który emituje światło. W grafice komputerowej takie obiekty zwane są źródłami światła, które dzielą się na kilka rodzajów ze względu na sposób jego emisji są to:

- Światła kierunkowe gdzie promienie świetlne emitowane przez taki typ mają zawsze taki sam kierunek i intensywność bez względu na odległość i pozycję[27].



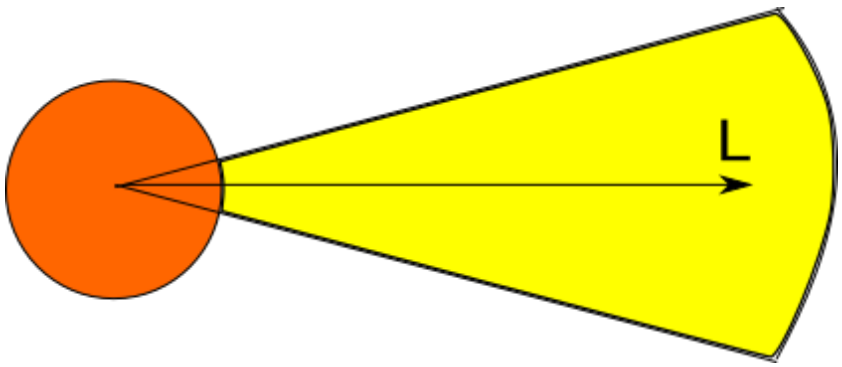
Rysunek 2.3: Graficzne przedstawienie idei światła kierunkowego. [Źródło : Opracowanie Własne]

- Światła punktowe (point lights) promienie światła emitowane przez ten typ oświetlenia rozchodzą się równomiernie w każdym kierunku dodatkowo intensywność maleje wraz z dystansem[27].



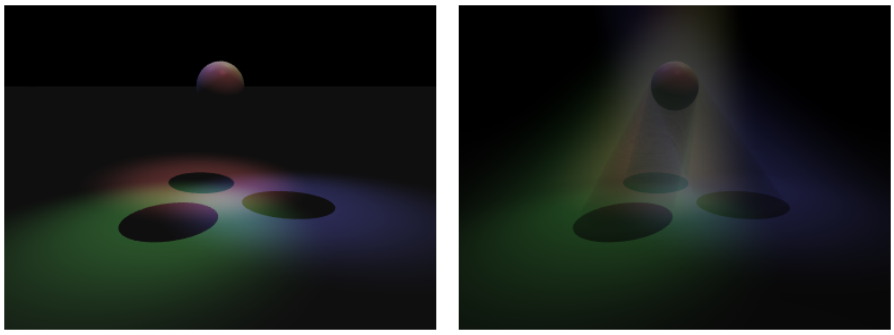
Rysunek 2.4: Wizualizacja zasady działania światła punktowego. [Źródło : Opracowanie Własne]

- Światła reflektorowe (spot lights) zachowują się jak reflektory co oznacza że promienie świetlne rozchodzą się w stożku który jest skierowany w określonym kierunku względem powierzchni[27].



Rysunek 2.5: Ilustracja przedstawia ideę światła reflektorowego. [Źródło : Opracowanie Własne]

Wymienione powyżej źródła świateł dają nam potężne możliwości jeśli chodzi o tworzenie realistycznych scen np. Księżyc jako światło kierunkowe i lampa uliczna na tle kilku budynków może dać nam poczucie oglądania prawdziwego miasta nocą. Dążenie do jak najlepszego odwzorowania świata realnego w grafice komputerowej określamy mianem fotorealizmu[20]. Oprócz wyżej wymienionych sposobów przedstawiania oświetlenia na uwagę zasługuje również technika świateł wolumetrycznych. Czasami w pochmurny dzień zauważmy promienie słoneczne przebijające się przez chmury i rzucające snop światła. Właśnie taki efekt w grafice komputerowej nazywamy światłami wolumetrycznymi[48].

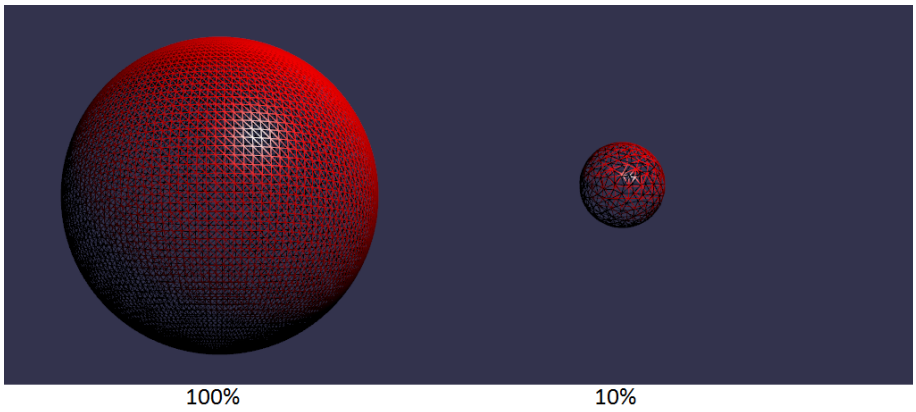


Rysunek 2.6: Przykład oświetlenia wolumetrycznego. Lew ilustracja przedstawia normalne oświetlenie prawa oświetlenie wolumetryczne. [Źródło: [32]]

Analizując silnik Three js stwierdzam obecność podstawowych źródeł oświetlenia takich jak: światło kierunkowe, światło punktowe, światło reflektorowe. Dodatkowo twórca silnika dodał możliwość używania światła otaczającego opisanego powyżej oraz światła hemisferycznego które jest rozwinięciem światła otaczającego. Rzecz polega na tym że część powierzchni obiektu której wektor normalny jest skierowany ku górze otrzymuje ustalony kolor, który możemy nazwać kolorem nieba. Jednocześnie druga część powierzchni, której wektor normalny skierowany jest w dół zostaje napromieniowana tak zwanym kolorem ziemi również zdefiniowanym przez programistę. Dodatkowo three js posiada metodę świateł wolumetrycznych zaimplementowanych jako shader która w moim mniemaniu jest lepsza niż wbudowana funkcjonalność dostępna w Babylon js. Ponadto Babylon js tak jak three js posiada technikę hemisferyczną z tą różnicą iż pozwala ona na zdefiniowanie koloru specular. Mimo wszystko uważam iż pojedynek w zakresie oświetlenia zwycięża three js za lepszą realizację oświetlenia wolumetrycznego pomimo że nie jest natywnym post-procesem silnika[9, 41].

2.2.2 Level of Detail

LOD jest jedną z techniki optymalizacji sceny. Jej celem jest redukcja złożoności modelu 3d poprzez zmniejszenie liczby jego wierzchołków, zarazem zachowując jego spójność. Zmniejszona jakość modelu jest często niedostrzegalna gdyż LOD jest w większości wykonywany na obiektach oddalonych na odpowiednią odległość od obserwatora, gdzie zmniejszenie liczby wierzchołków będzie słabo lub w ogóle niedostrzegalne. Jako że technika LOD zmniejsza liczbę wierzchołków pozwala to karcie graficznej szybciej przetwarzać etapy potoku graficznego głównie geometrii i rasteryzacji[25].



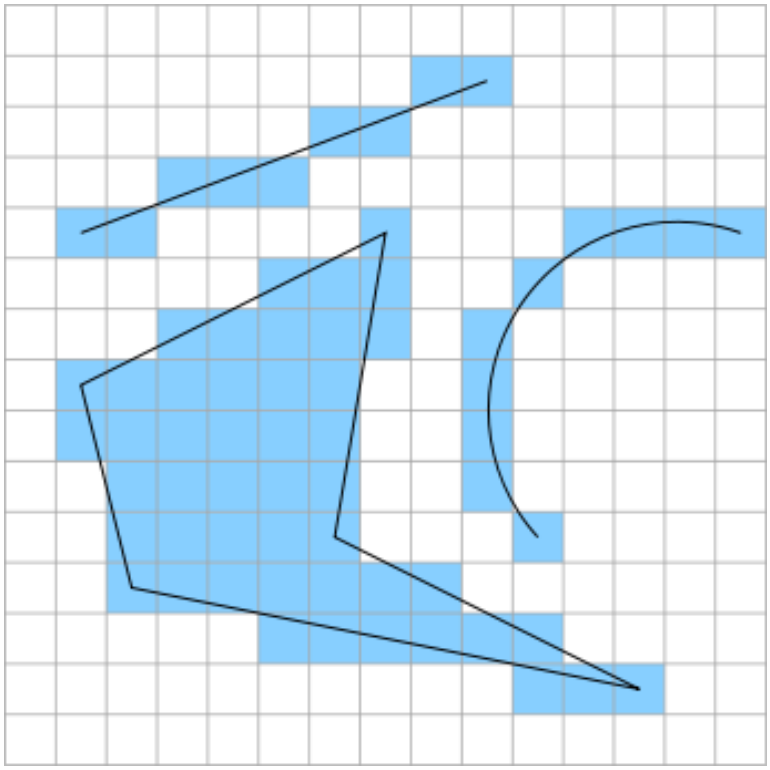
Rysunek 2.7: Przykład zastosowania techniki LOD w BabzlonieJs. Po lewej stronie przedstawiony jest oryginalny obiekt ,po prawej uproszczona geometria. [Źródło: [11]]

Analizując oba silniki stwierdzamy obecność LOD w Babylonie i Threejs. Po porównaniu pomiędzy silnikami mogę z całą pewnością stwierdzić że Babylon js deklasuje rozwiązania LOD w porównaniu z Threejs. Dzieję się to dlatego, iż LOD w Babylonie posiada funkcje która automatycznie uproszcza geometrie co oznacza, że po zdefiniowaniu procentu szczegółowości na określonym dystansie obiektu od obserwatora framewok automatycznie zmniejszy liczbę wierzchołków, krawędzi i w konsekwencji trójkątów, pozwalając etapom potoku graficznego takim jak przetwarzanie geometrii czy rasteryzacja wykonać operacje szybciej co doprowadzi do zwiększenia liczby generowanych klatek na sekundę. Jeżeli chodzi zaś o rozwiązania tej techniki w Three js są one niestety słabsze z tego względu, że nie posiada ona auto upraszczania geometrii, co wymaga od developera użycia zewnętrznych programów takich jak 3dsMax czy Blender aby przygotować co najmniej 2 wersje tego samego modelu z mniejszą ilością wierzchołków,

aby mogły być one wyświetlane na zdefiniowanym przez użytkownika dystansie. Podsumowując kryterium LOD daję bardzo duży plus Babylon js ze względu na obecność auto upraszczania oraz możliwość implementacji własnych algorytmów poprzez implementację odpowiednich metod dlatego właśnie silnik ten otrzymuje jeden punkt[10, 41].

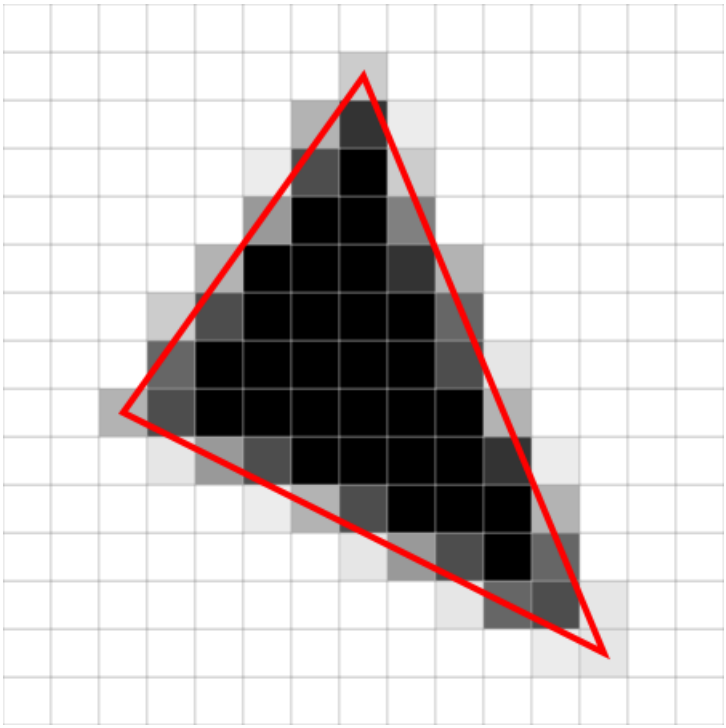
2.2.3 AntiAliasing

Aliasing jest jednym z poważniejszych problemów grafiki komputerowej w zastosowaniach profesjonalnych jak i rozrywkowych. Zjawisko to charakteryzuje się tym że krawędzie obiektów np. trójkątów czy linii krzywych będą zniekształcone przykładowo zamiast łagodnego przejścia linii krzywej zauważymy schodki będące pikselami. Dzieję się tak ze względu na to, iż w trakcie rasteryzacji na etapie wypełniania trójkątów pewne piksele nie są wystarczająco wypełnione powierzchnią danego trójkąta co sprawia, że nie są dla nich generowane fragmenty i taki piksel będzie miał inny kolor np. biały gdyż jego większa powichrzenia będzie należała do innego trójkąta[4].



Rysunek 2.8: Przedstawienie problemu aliasingu polegającego na braku wypełnienia niektórych pikseli przez które przechodzi linia.[Źródło: [33]]

Odpowiedzią na ten problem jest antyaliasing. Polega on na tym że sąsiedztwo piksela krawędzi zostanie uzupełnione kolorem sąsadującego piksela o jasności proporcjonalnej do odległości środka piksela do prostej, która go przecina. Drugim sposobem jest supersampling który renderuje dany obraz w wyższej rozdzielczości następnie degradując go do niższej docelowej rozdzielczości, którą zobaczymy na ekranie. Główną ideą supersamplingu jest to, że jeśli zwiększamy rozdzielczość, to w konsekwencji liczbę pikseli które sąsiadują z pikselem w niższej rozdzielczości. Następnie wykorzystując dodatkowe dane w postaci kilku kolorów uśrednimy go i nałożymy na piksel, który znajduje się w niższej rozdzielczości aby zmniejszyć widoczność zniekształceń. Główną wadą supersamplingu jest wysoki koszt obliczeń gdyż potok graficzny musi wykonać operacje dla minimalnie dwa razy wyższych rozdzielczości. Istnieje również dziedzina która zalicza się do antyaliasingu. Jest nią filtrowanie tekstur - działa ona bardzo podobnie do opisanych powyżej rozwiązań, a różni się ona tym że skupia się na kolorach tekstur nakładanych na piksele co pozwala na ich nałożenie na nietypowe geometrie redukując rozmycie czy ich nadmierną jasność[2, 42].



Rysunek 2.9: Przykład zastosowania AntyAliasingu na trójkącie podczas procesu rasteryzacji. [Źródło: [31]]

Zanim omówimy sposoby nakładania anti aliasingu wspomnimy jeszcze czym

jest post-proces. Dawno temu kiedy karty graficzne i api wykorzystywały tzw. Fixed Function Pipeline developer był ograniczony do funkcji które wykonywały określone zadania takie jak, teksturowanie czy anti aliasing określoną metodą. W tamtych czasach gdy generowano jeszcze grafikę bezpośrednio za pomocą procesora, wprowadzenie Fixed Function Pipeline było nieporównywalnym skokiem wydajnościowym i zostało ciepło przyjęte przez społeczność. Jednakże wkrótce zaczęto zauważać dodatkowe problemy, jakie sprawiał FFP głównie było to ograniczenie swobody w tworzeniu skomplikowanych geometrii czy innych wymagających efektów graficznych. Było więc pewne że pojawi się wkrótce metoda, która zachowując zalety FFP wyeliminuje największe jej wady czyli małą kontrolę nad potokiem graficznym. Taką nadzieję spełnił debiut programowalnego potoku graficznego w którym to developerowi oddano kontrolę nad częścią etapów potoku graficznego. Najważniejszą częścią tego potoku są mini programy nazywane shaderami, które na przykład po przekazaniu do etapu przetwarzania geometrii wierzchołków są w stanie dokonać dodatkowych przekształceń zdefiniowanych przez programistę (vertex shader), stworzyć nową geometrie (geometry shader), modyfikować skomplikowanie istniejącej geometrii (teselation shader). Drugim etapem potoku gdzie możemy stosować shadery jest rasteryzacja - w tym przypadku używamy frgament shader dzięki któremu możemy określić jaki kolor nałożymy na piksel czy zamiast niego zdecydujemy się na teksturę. Wracając do problemu post-procesingu na końcu potoku graficznego piksele o określonych kolorach znajdują się buforze kolorów nazywanego też buforem ramki.Post-processing mówi że możemy pobrać wszystkie piksele z bufora ramki i za pomocą shaderów dokonać ich modyfikacji. Zmodyfikowane w ten sposób piksele możemy ponownie załadować do bufora ramki aby zostały w końcu wyświetlone na ekranie. Taki proces nazywamy właśnie post-procesingiem[40, 17].

Jeżeli chodzi o anti aliasign to posiadamy dwa sposoby jego zaaplikowania podczas rasteryzacji lub jako metodę post-procesową. Przykładem metod podczas rasteryzacji jest supersampling lub multisampling, zaś do metod post procesowych które analizują to co znajduje się buforze ramki wykrywając krawędzie i modyfikując jasność pikseli możemy zaliczyć algorytmy takie jak FXAA czy TXAA. Ponieważ gra komputerowa nie jest kompletnie równa drugiej grze komputerowej i różne metody anti aliasignu w różnych grach będą się sprawdzały raz lepiej a raz gorzej nasze porównanie ograniczymy do ilości dostępnych metod antialiasingu.

Obydwa silniki posiadają metodę post-procesową FXAA babylon js jako wbudowaną a three js jako shader, dodany przez społeczność. Jednakże na prowadzenie wysuwa się three js z metodami SMAA i TAA (temporal anti aliasing

). Wspomnieć należy że obydwa silniki obsługują shadery omówione powyżej, więc każdy programista ma możliwość zaimplementowania własnej metody anti aliasingu, jednakże na ta chwilę nie ma innych metod anti aliasingu w przypadku Babylon js, więc finalnie punkt w tym kryterium otrzymuje Three js[7, 44].

2.3 Dodatkowe Narzędzia

To kryterium ma na celu porównanie dodatkowych narzędzi które wspomagają tworzenie gier komputerowych w porównywalnych silnikach. Aby dokonać obiektywnej oceny zostaną przedstawione narzędzia udostępnione przez twórców danego silnika na swoich stronach internetowych. Następnie każde narzędzie zostanie poddane analizie polegającej na opisaniu jego funkcjonalności i przydatności w procesie tworzenia gry komputerowej.

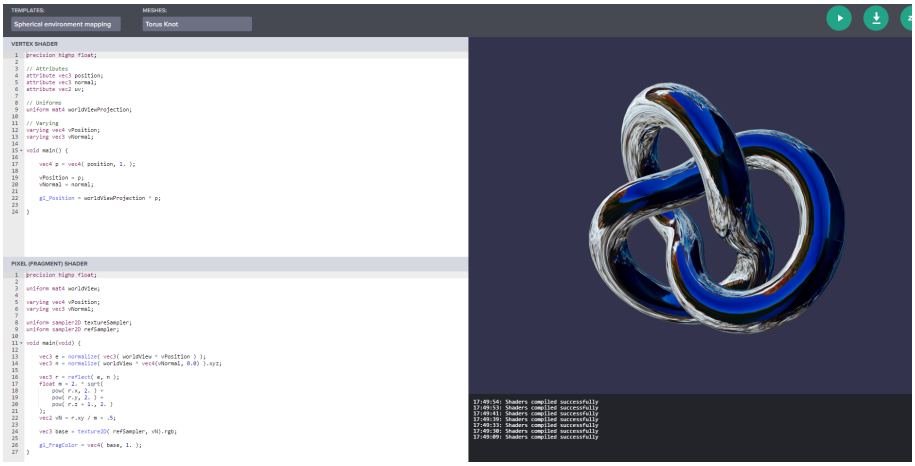
Analizę rozpoczniemy od Babylon js. Na pierwszy rzut oka można stwierdzić że ten framework proponuje więcej w zakresie dodatkowych narzędzi niż three js. Pierwszym bardzo ciekawym i znaczącym dodatkiem jest playground.

Rysunek 2.10: Screenshot narzędzia playground działającego poprzez przeglądarkę internetową. [Źródło : [13]]

Narzędzie to pozwala na skompilowanie i wykonanie kodu bezpośrednio w przeglądarce internetowej. Pozwala to na zapoznanie się z możliwościami silnika pomijając wcześniejszą potrzebę ściągnięcia odpowiednich bibliotek. Otwiera to również nowe możliwości w nauce obsługi frameworka, gdyż playground zawiera większość przykładów wykorzystanych w dokumentacji i samouczkach dotyczących poszczególnych zagadnień. Ponadto mamy mozliwość pobrania całego kodu wykonanego w playground jako plik html lub zapisania utworzonej przez nas sceny do internetowej bazy danych, która będzie dostępna pod wygenerowanym dla nas linkiem URL[13].

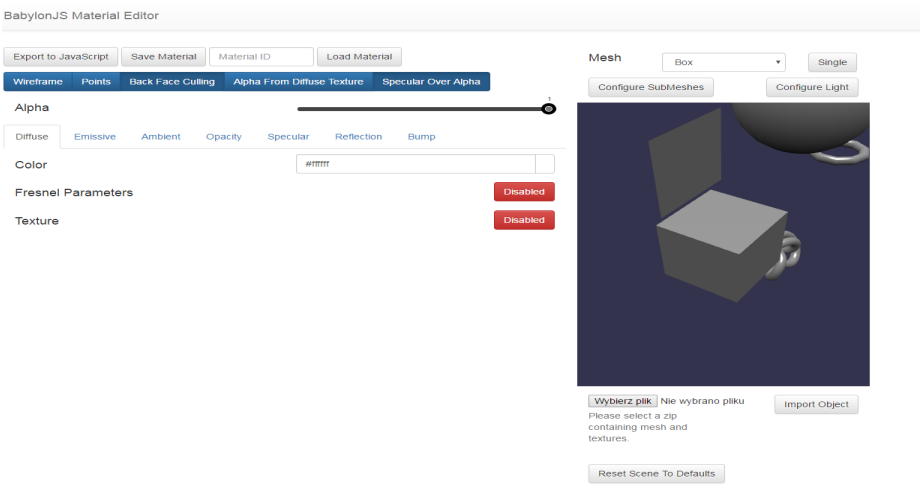
Kolejnym interesującym narzędziem jest cyos. Mimo dziwnej nazwy która na

piewszy rzut oka nie wiele mówi cyos jest edytorem który pozwala na stowrzenie i przetestowanie własnych shaderów. Na chwilę obecną mamy możliwość napisania vertex shaders i fragment shaders które będą modyfikowały udostępnione geometrie którymi są: kula, teren, torus[16].



Rysunek 2.11: lustracja przedstawia narzędzie przeznaczone do tworzenia shaderów. [Źródło : [16]]

Następnym narzędziem na liście jest edytor materiałów. Jego główną zaletą jest możliwość stworzenia materiału i jego eksportu jako kod javascriptowy. Ponadto możemy załadować własny model trójwymiarowy i wykonać materiał dla niego[12].



Rysunek 2.12: Przedstawienie przeglądarkowego narzędzia do tworzenia materiałów. [Źródło : [12]]

Ostatnim omawianym narzędziem Babylon js jest sandbox. Sandbox pozwala na załadowanie scen które zostały wyeksportowane do formatu babylon[37].

Rysunek 2.13: Narzędzie sandbox pozwala na import sceny i jej wizualizację. [Źródło : [37]]

Przechodząc do narzędzi three js z przykrością muszę stwierdzić, że na uwagę zasługuje tylko wizualny edytor sceny który pozwala na: umieszczanie obiektów czy źródeł oświetlenia, nakładanie różnych rodzajów materiałów i modyfikację ich parametrów, tworzenie skryptów za pomocą wbudowanego edytora i na koniec wykonanie zaprojektowanej sceny online[43] . Podsumowując mimo że three js posiada tylko wizualny edytor sceny może on spokojnie rywalizować z playground i edytorem materiałów babylon js, jednakże brak narzędzia do tworzenia shaderów sprawia że w tym kryterium wygrywa babylon js i otrzymuje punkt.

2.4 Obsługa algorytmów sztucznej inteligencji

Te kryterium skupia się na tym czy dany framework posiada zaimplementowane algorytmy sztucznej inteligencji w zakresie pathfindingu czyli znajdowania najkrótszej ścieżki z punktu A do B. Niestety żaden porównywany framework nie posiada zaimplementowanych algorytmów do podejmowania decyzji czy wyszukiwania najkrótszej ścieżki. Kryterium sztucznej inteligencji znalazło się tylko dlatego że algorytm A* został użyty w implementacji gry.

2.5 Zasady Licencjonowania

Każdy program komputerowy napisany przez człowieka jest przejawem jego twórczości. Twórczości która wymaga czasu oraz zaangażowania i dlatego jest chroniona prawnie przed nieuprawnionym, bądź niewłaściwym wykorzystaniem. Ochronę tę definiują prawa autorskie które również stosuje się do frameworków gdyż są tworzone przez ludzi. Kryterium licencjonowania przybliży nam warunki na jakich twórcy Babylon js i Three js udostępnili swoje dzieła. Zanim jednak przejdziemy do omawania licencji wspomnimy jeszcze o tym, czym jest wolne oprogramowanie. Termin ten określa oprogramowanie udostępnione przez jego twórcę które może być uruchamiane, modyfikowane, kopiowane i rozpowszechniane. Aby spełnić założenia wolnego oprogramowania dany program musi gwarantować użytkownikowi cztery podstawowe wolności[34] :

1. „*Wolność uruchamiania programu w dowolnym celu*" [34].
2. „*Wolność analizowania programu oraz dostosowywania go do swoich potrzeb*" [34].
3. „*Wolność rozpowszechniania kopii programu*" [34].
4. „*Wolność udoskonalania programu i publicznego rozpowszechniania własnych ulepszeń, dzięki czemu może z nich skorzystać cała społeczność*" [34].

Realizacja punktu 2 i 4 wymaga od autorów udostępnienia kodu źródłowego oprogramowania czyli implementacji w danym języku programowania. Przechodząc do kwesti licencjonowania Babylon js jest udostępniany na licencji Apache w wersji 2. Licencja ta może być stosowana do oprogramowania komercyjnego jak i wolnego. Zezwala ona na używanie, modyfikowanie i rozpowszechnianie oprogramowania w postaci kodu źródłowego lub postaci binarnej, jednakże nie narzucając obowiązku udostępniania kodu źródłowego. Ponadto chroni ona również twórców programów którzy udostępnili go na tej licencji gdyż nie pozwala na używanie nazw lub znaków handlowych wyjątkiem jest tylko sytuacja, w której chcemy poinformować o pochodzeniu oprogramowania. Dodatkowo licencja automatycznie udziela użytkownikowi prawa do używania technologi które zostały opatentowane. Three js jest udostępniany na mocy licencji MIT. Zezwala ona na nieograniczone używanie, modyfikowanie i rozpowszechnianie jedyny warunek jaki należy spełnić to umieścić informacje o autorze danego oprogramowania. Podsumowując z punktu widzenia developera gry najkorzystniejszą dla niego formą spośród wymienionych powyżej jest licencja Apache ze względu na to, że pozwala twórcy na wykorzystanie jej do celów komercyjnych oraz na

zabezpieczeniu przed wykorzystaniem nazw lub znaków handlowych. Dlatego w zakresie tego kryterium punkt otrzymuje Babylon js udostępniany na licencji Apache[5, 49].

2.6 Obsługa fizyki i wykrywania kolizji

Analizując obsługę fizyki skupimy się na tym czy dany framework wykonuje symulacje zasad fizycznych taki jak grawitacja czy oddziaływania pomiędzy obiektami za pomocą wbudowanych modułów lub bibliotek zewnętrznych. Jeżeli chodzi o wykrywanie kolizji to przedstawię dwie najpopularniejsze metody realizacji tego problemu, a następnie przyjrzymy się rozwiązaniom które zostały udostępnione w porównywalnych silnikach.Pierwszą najbardziej popularną metodą wykrywania kolizji w grach komputerowych są bryły brzegowe. Na dany obiekt trójwymiarowy zostaje nałożona bryła głównie sześcian lub sfera, tak aby pokryła cały model. Następnie jeżeli chcemy przetestować czy dany obiekt nie koliduje z innymi sprawdzamy czy dana bryła brzegowa nie koliduje z inną bryłą. W tej metodzie używa się najczęściej sześcianów (Bounding Box) lub sfer (Bounding Sphere)[15].

Rysunek 2.14: Przykład techniki bryły brzegowej o kształcie sześcianu wypełniającej model trójwymiarowy. [Źródło: [6]]

Drugim najbardziej rozpowszechnionym sposobem wykrycia kolizji jest metoda śledzenia promieni. Jej zasada jest prosta mając dany obiekt 3D i znając jego wymiary wyprowadzamy promień o określonej długości. Promień ten powinien pochodzić z punktu który jest najbardziej wysuniętym miejscem na siatce modelu 3D. Dodatkowo promień taki powinien posiadać wektor kierunku zgodny z kierunkiem przemieszczania się obiektu. Kolizja w takim przypadku zajdzie, gdy promień przetnie bryłę brzegową lub gdy odległość między obiektami po odjęciu promienia porównywalnego obiektu będzie wynosiła zero lub mniej. Istnieje jeszcze jedna możliwość w zakresie metody śledzenia promieni. Jej zasada jest prosta na początku określamy promienie obydwu obiektów 3D i odległość pomiędzy nimi. Następnie po dodaniu dwóch promieni odejmujemy je od odległości pomiędzy obiektami w przypadku gdy odległość wynosi zero zachodzi kolizja między obiektami[15].

Po analizie implementacji metod wykrywania kolizji w threejs stwierdzam obsługę metody śledzenia promieni. Należy jednak wspomnieć iż jest to tylko mechanizm do wyprowadzenia promienia i wykrycia momentu przecięcia z przeszkodą. Wymagane jest tutaj od developera aby sam zdefiniował co należy zrobić gdy wystąpi kolizja. Taki problem nie występuje w przypadku Babylona gdzie dostępne są obydwie metody wykrycia kolizji omówione powyżej. Metoda brył brzegowych po włączeniu nie pozwala elipsoidzie reprezentującej gracza na kontakt z innymi obiektami dla których sprawdzanie kolizji zostało włączone. Dodatkowo możemy zdefiniować sposób reakcji na wykrycie kolizji pomiędzy dwoma obiektami. Jeżeli chodzi zaś o metodę śledzenia promieni to w tym przypadku zasada działania jest taka sama jak w Threejs. Podsumowując Babylon js deklasuje Threejs w zakresie obsługi kolizji ze względu na natywną obsługę metod brył brzegowych i z tego względu w tej rywalizacji otrzymuje punkt[7, 41].

2.7 Dostępność dokumentacji i społeczności

Kolejnym kryterium będzie ilość informacji którą udostępniają twórcy danego silnika, powszechnie zwaną dokumentacją czyli listą funkcji w danym języku programowania realizującą pewne zadania czy zbiorem samouczków będących instrukcją jak osiągnąć pewien cel używając udokumentowanych funkcji danego frameworka. Wspomnimy i porównamy również społeczność programistyczną danego frameworka czyli zbiór developerów tworzących treści w danym silniku skłonnych do udzielania rad i udostępnia własnych materiałów, które pomagają w uczeniu się obsługi danego silnika.

Dokumentacja jest dla dewelopera jednym z najważniejszych czynników wpływających na wybór frameworka za pomocą którego będzie tworzył grę komputerową. Analizując oficjalne dokumentacje udostępniane przez twórców silników mogę z całą pewnością stwierdzić, iż babylon js deklasuje threejs. Przyczyn takiego stanu jest kilka ale najważniejsze z nich to :

- Podział na działy tematyczne np. samouczki, opis klas czy opis dodatkowych narzędzi co wpływa na czytelność i przejrzystość zasobów.
- Obszerny zbiór oficjalnych samouczków wraz z przykładami dla playground.
- Opis eksportowania modeli lub scen z programów do modelowania trójwymiarowego takich jak 3dsMax czy Blender.
- Możliwość wyszukiwania przykładów w bazie playground po słowie kluczowym.
- Dokładny opis klas zawierający nazwę zmiennych i metody.

Właśnie z tych względów babylon js otrzymuje punkt w zakresie oficjalnej dokumentacji gdyż threejs zawiera tylko opis klas. Mimo przewagi babylona w kwestii dokumentacji wypada on blado, jeżeli chodzi o zasoby społecznościowe. Wyraźnym zwycięzcą jest tutaj threejs który oprócz dużej liczby książek zorientowanych na tworzeniu gier posiada również potężne forum dyskusyjne jakim jest reddit, nie wspominając już o wielu nieoficjalnych blogach czy samouczkach video które wyjaśniają meandry silnika. Dlatego właśnie punkt z zakresu społeczności trafia do threejs ze względu na większą liczbę materiałów dostępnych dla użytkownika[7, 41].

2.8 Model płatności silnika

W tym kryterium przedstawimy różne rodzaje płatności za pomocą których twórcy rekompensują zasoby zużyte na stworzenie danego silnika. Produkcja każdego oprogramowania jest kosztowna jeśli nie w wymiarze finansowym to chociażby czasowym. Twórca bowiem musi poświęcić swój czas który mógłby wykorzystać inaczej np. wykonując pracę otrzymywałby wynagrodzenie. Jeśli chodzi zaś o finansowe straty ponoszone przez dewelopera zaliczyć tu możemy wynagrodzenie dla pracowników którzy pomagają nam przy tworzeniu kodu, bądź koszt narzędzi użytych podczas tworzenia oprogramowania. Normalnie twórca programu dajmy na to gry komputerowej po jej ukończeniu wycenia

koszty jej stworzenia i na ich podstawie ustanawia jej cenę. W zależności od popularności takiej gry może ona wygenerować wystarczające przychody które pokryją lub nawet zwrócą z nawiązką koszta jej wyprodukowania. Podobnie jest z silnikami do gier - ich twórcy muszą zrekompensować sobie koszty jego produkcji. Niestety istnieje kilka różnic pomiędzy grami a silnikami przeznaczonymi do ich tworzenia, które zwiększają koszty i utrudniają zarobek. Pierwszą zauważalną różnicą jest okres wsparcia danego produktu. Gra komputerowa jak i każdy program wymaga utrzymania w postaci aktualizacji naprawiających błędy, które uniknęły wykrycia podczas testów lub wsparcia technicznego dla użytkowników. Wsparcie te ma na celu w większości przypadków utrzymanie poprawnego funkcjonowania produktu a czas przez jaki jest zapewniane zależy głównie od jego popularności. Sprawa komplikuje się w przypadku silników. Grafika komputerowa cały czas się rozwija, powstają nowe algorytmy wymagające większej mocy obliczeniowej co wymusza rozwój kart graficznych. Konsekwencją takiej sytuacji jest zmiana podejścia do wsparcia programu. Niewystarczające staje się tylko utrzymywanie poprawnego działania programu, ponieważ ze względu na konkurencję oprogramowanie musi aktywnie rozwijać się, aby mogło wykorzystywać najnowsze techniki generowania grafiki trójwymiarowej. Sprawia to że twórcy silnika zostają postawieni przed poważnym problemem jaki sposób płatności wybrać aby zaspokoił on wszystkie wydatki związane z zwiększonymi kosztami utrzymania produktu. Wydawcy oprogramowania wymyślili więc wiele modelów płatności najciekawszymi i najważniejszymi zasługującymi na uwagę w naszym przypadku są:

- Brak opłat będący wiodącą ideą wolnego oprogramowania które ma służyć ludzkości dzięki swojej otwartości na modyfikacje i bezproblemowe rozpowszechnianie ich. Model ten jest najkorzystniejszy dla użytkowników wraz z połączeniu z licencjami typ MIT czy Apache, gdyż pozwala im na modyfikowanie i rozpowszechnianie oprogramowania bez żadnych konsekwencji prawnych. Z kolei dla twórców w znaczeniu komercyjnym jest to najgorsza opcja, gdyż nie jest rekompensowany koszt stworzenia silnika[14].

- Jednorazowa opłata jest jedną z najpopularniejszych form płatności w przypadku gier komputerowych. Jednakże w przypadku frameworków do gier staje się coraz mniej popularna ze względu na specyfikę takiego oprogramowania. Przyczyna takiego stanu jest stały, bądź nawet czasami rosnący koszt utrzymania takiego oprogramowania, który niestety w większości przypadków nie zostanie pokryty przy zastosowaniu tego modelu płatność. Mówiąc krótko główną wadą jednorazowej opłaty jest to że produkt zostanie użyty wiele razy przy tworzeniu gier jednak licencjodawca

otrzyma zapłatę tylko raz co powoduję duże dysproporcje przychodów pomiędzy twórcami silnika a gier komputerowych[14].

- Subskrypcja czasowa jest jedną z najpopularniejszych form finansowania, utrzymania i rozwoju silnika. Polega ona na uiszczaniu okresowej opłaty w zamian za korzystanie z oprogramowania. Rozwiązanie te wpływa korzystnie na sytuacje twórców jak i również użytkowników programu. Z punktu widzenia użytkowników cena może być bardzo niska, ze względu na to że twórcy otrzymują stały dopływ gotówki co pozwala im pokryć koszta utrzymania i rozwoju silnika. Ponadto deweloperzy gry płacą za czas przez jaki będą używać silnika do tworzenia i utrzymania gry, co sprawia iż w niektórych przypadkach suma wydatków w modelu subskrypcji może być mniejsza niż jednorazowa opłata która może wynosić czasem nawet kilka tysięcy dolarów. Twórcy zaś mają zapewniony stabilny przychód przez co mogą aktywnie rozwijać silnik prowadząc do jego popularyzacji, co dodatkowo wygeneruje większe dochody z tytułu subskrypcji[14].

- Procent od dochodów jest dość popularną formą finansowania niektórych silników. Główną ideą tego modelu jest oddanie pełnej funkcjonalności oprogramowania deweloperom za darmo w zamian za udział w zyskach, jakie taki program wygeneruje. Dla twórców silnika może być to bardziej opłacalne finansowo, ponieważ zyski z sprzedaży gry, to czasami nawet kilka milionów dolarów a średni procent jaki twórcy pobierają wynosi od 5 do 10 procent, co daje zyski dla twórców kilkanaście razy większe niż we wcześniejszych opisanych modelach[14].

Znając już wszystkie najważniejsze z naszego punktu widzenia metody płatności możemy przeanalizować jaki model został wybrany przez twórców babylona i threejs. Obydwa silnika są dostępne w pierwszym omawianym modelu czyli za darmo na licencjach wolnego oprogramowania. Sprawia to że przy porównywaniu w zakresie tego kryterium, oba silniki niczym się nie różnią i dlatego ze względu na ten przysłowiowy remis, każdy otrzymuje zero punktów

2.9 Tabela porównawcza

Kryterium / Funkcjonalność	Dostępności w Three js	Dostępność w Babylon js
2.2.1 Rodzaje Oświetlenia	Podstawowe + Hemisferyczne, Volumetryczne	Podstawowe + Hemisferyczne, Volumetryczne
2.2.2 LOD	TAK brak autoupraszczania geometrii	TAK z autoupraszczaniem geometrii
2.2.3 AntiAliasing	TAK FXAA,SMAA,TAA jako Shadery	TAK wbudowana metoda FXAA
2.3 Dodatkowe Narzędzia	Edytor Wizualny	Sandbox, Playground, CYOS, Edytor Materiałów
2.4 Obsługa AI	BRAK	BRAK
2.5 Licencja	MIT	Apache V2
2.6A Wykrywanie Kolizji	TAK za pomocą RayCaster	TAK natywna Bryły brzegowe lub za pomocą RayCaster
2.6B Fizyka	Brak natywnej implementacji	Brak natywnej implementacji
2.7A Dokumentacja	TAK tylko opis klas	TAK opis klas, samouczki, opis narzędzi
2.7B Społeczność	Duża liczba ksiązek, samouczków, forów dyskusyjnych	mała ilość ksiązek i forów dyskusyjnych
2.8 Model Płatności (Cena)	Darmowy	Darmowy

Rysunek 2.15: Tabela przedstawia najważniejsze różnice pomiędzy silnikami w poszczególnych kryteriach. [Źródło : Opracowanie własne]

2.10 Personalna opinia i podsumowanie

Podczas prac nad własną implementacją gry miałem okazję przetestować oba silniki pod względem ich przydatność w tworzeniu prostej gry 3D. Pierwszą najważniejszą rzeczą podczas pisania kodu było trzymanie się koncepcji programowania obiektowego. Główną ideą był podział poszczególnych funkcjonalności na osobne klasy np. pathfidning jako osobny obiekt realizujący algorytm i zwracający wynik czy stworzenie przeciwnika i zarządzanie jego logiką. W tym zakresie najlepiej sprawował się threejs w którym geometria mogła być tworzona w osobnych klasach, w przeciwieństwie do babylona gdzie niestety nie zawsze było to możliwe. Drugim ważnym czynnikiem jest wbudowanie podstawowych funkcjonalności jak : kamery czy wykrywanie kolizji. Babylon js spełnia te założenie posiadając wbudowane klasy realizujące te zadania w przeciwieństwie do threejs który ich nie posiada, przez co wymagane było poświęcenie większej ilości czasu na ich realizacje. Jednakże największe rozczarowanie przyniósł brak obsługi jakichkolwiek algorytmów sztucznej inteligencji w obydwu silnikach. Jest to bowiem podstawą dzisiejszych gier komputerowych i niestety nie da się zrobić dobrej i interesującej gry, bez obsługi sztucznej inteligencji .

Ostatnim według mnie czynnikiem jest ilość materiałów o danym silniku. Pracując z Babylonem miałem dostęp do większej ilości oficjalnych samouczków i dokumentacji, jednakże nie pomagały one rozwiązywaniu drobnych problemów związanych ze specyfiką działania silnika. Three js w tej materii przedstawia się lepiej, gdyż oprócz profesjonalnych książek ma dużą społeczność, która na część takich problemów już się natknęła i mogła udzielić rady. Podsumowując pomimo iż stworzenie gry w babylonie było cięższe, niż w three js to według mnie, jest on przyjaźniejszy jeśli chodzi o tworzenie gier ze względu na lepszą obsługę mechanizmów ważnych podczas tworzenia gier. Mowa tu oczywiście o wcześniej już wspomnianych typach kamery czy wykrywaniu kolizji. Niestety w przypadku three js nie uświadczymy tych mechanizmów, a przynajmniej nie w takim stopniu w jakim udostępnia nam je Babylon js. Kończąc moją ocenę babylonowi przyznaje od siebie 3 punkty a Three js 2.5 punkta. Powodem dla którego żaden silnik nie otrzymał maksymalnej ilości jest brak obsługi sztucznej inteligencji, co jest niestety bardzo rozczarowujące. Finalnie naszą rywalizację wygrywa babylon js który zebrał 8 punktów i dzięki temu w tyle zostawił three js z wynikiem wynoszącym tylko 5.5 punktów.

Rozdział 3

Projekt i implementacja gry

W niniejszym rozdziale przedstawione zostaną szczegóły dotyczące przykładowych implementacji gry w porównywalnych silnikach.

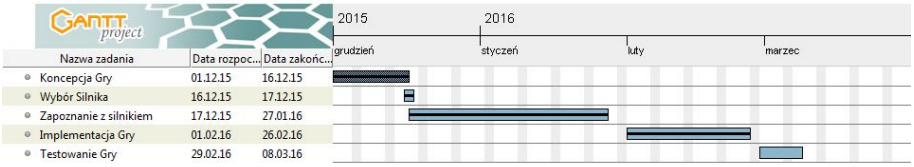
3.1 Opis implementacji

Implementowana gra jest przedstawicielem gatunku First Person Perspective którego głównym założeniem jest postrzeganie świata poprzez wirtualny awatar w taki sam sposób, jak na co dzień sami oglądamy świat. Gracz zostaje umieszczony w losowo wybranym miejscu znajdującym się labiryncie. Jego głównym celem gry jest pokonanie losowo wygenerowanego labiryntu unikając wykrycia przez patrolujących strażników. Strażnikami w grze są obiekty zdolne do wykrycia gracza znajdującego się w zasięgu czystego pola widzenia, określonego przez wektor kierunku. W przypadku zauważenia gra zakończy się wynikiem negatywnym, a użytkownik zostanie o tym poinformowany napisem game over. Alternatywnie jeżeli użytkownik pozostanie niezauważony i zdoła osiągnąć cel zakończy grę wynikiem pozytywnym co sprawi wyświetlenie napisu victory.

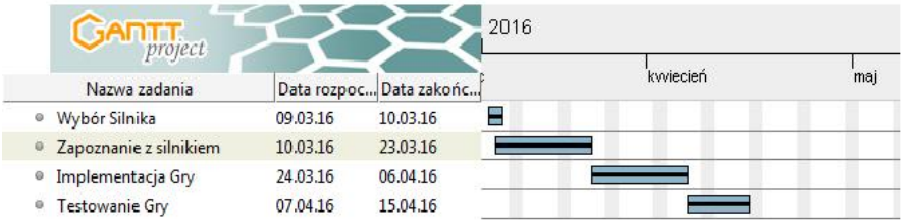
3.2 Wymagania

Powyższa gra do uruchomienia wymaga systemu operacyjnego zdolnego do uruchomienia przeglądarki internetowej obsługującej wykonywanie kodu javascript i api graficznego WebGL. Dodatkowo zalecane jest stworzenie własnego serwera HTTP np. WAMP, XAMPP bądź LAMP aby zapewnić poprawność wyświetlania tekstur.

3.3 Diagramy Gantta

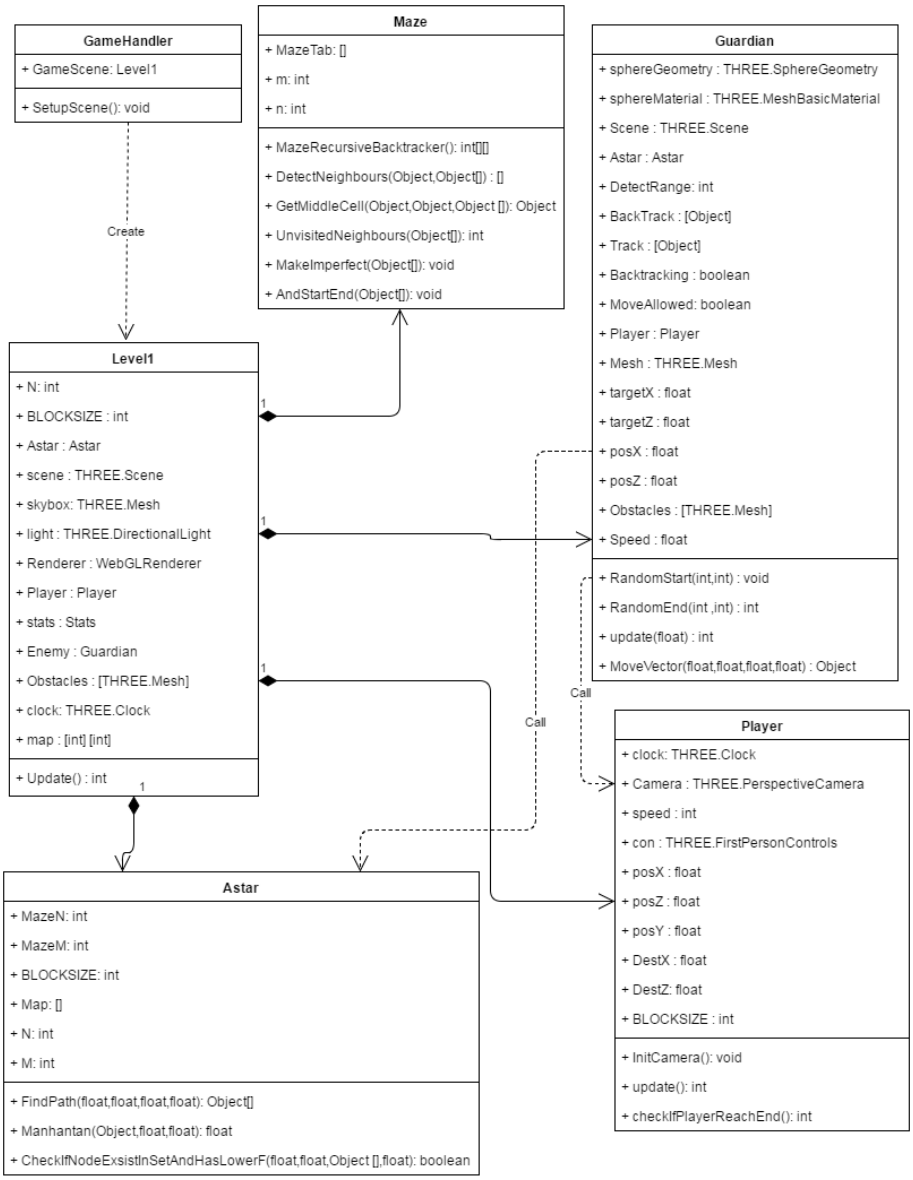


Rysunek 3.1: Diagram Gantta przedstawiający postęp prac nad implementacją gry za pomocą frameworka Babylon js wykonany za pomocą programu GanttProject 2.7. [Źródło : Opracowanie własne]

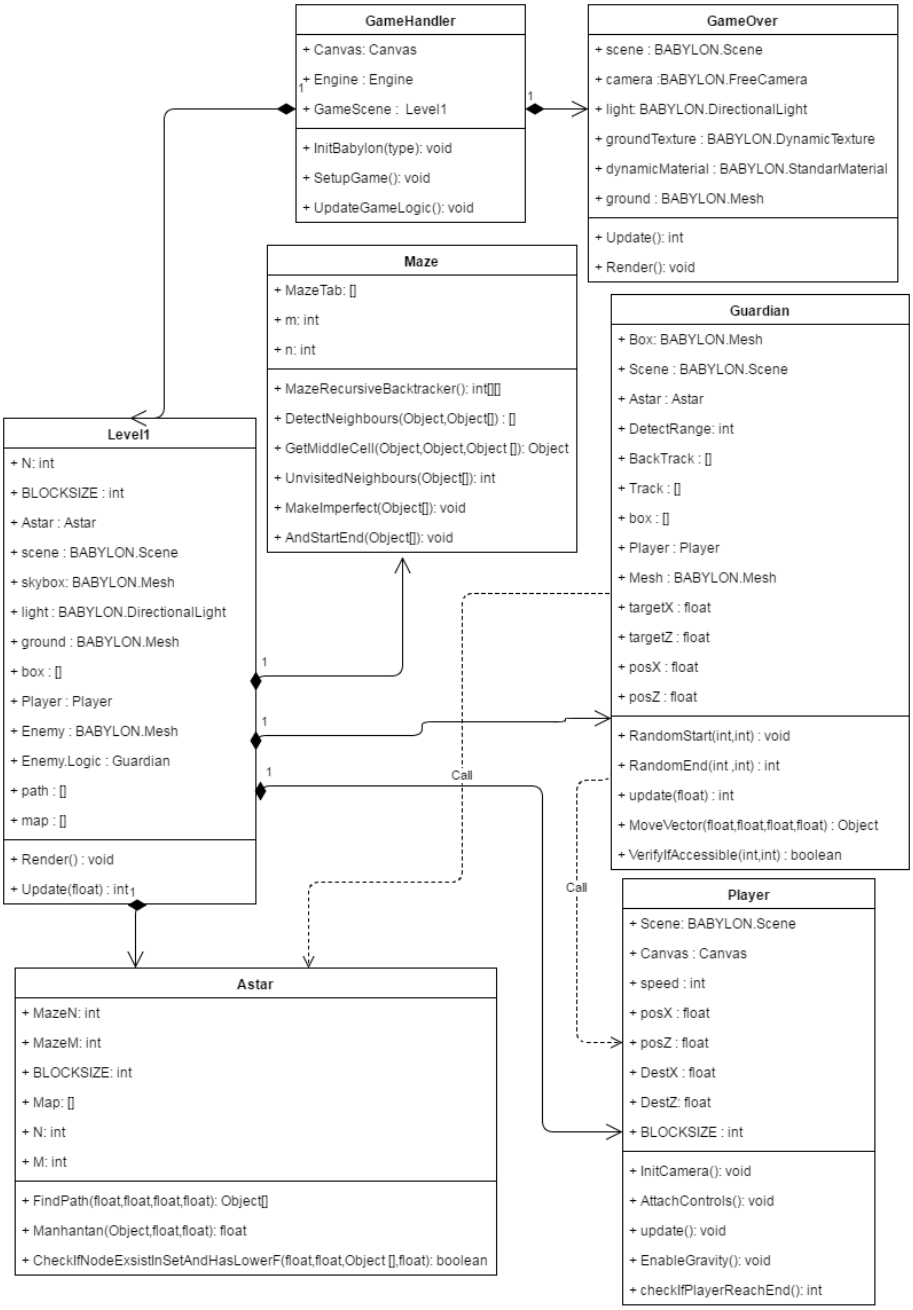


Rysunek 3.2: Diagram Gantta przedstawiający postęp prac nad implementacją gry za pomocą frameworka Threejs wykonany za pomocą programu GanttProject 2.7. [Źródło : Opracowanie własne]

3.4 Diagramy klas

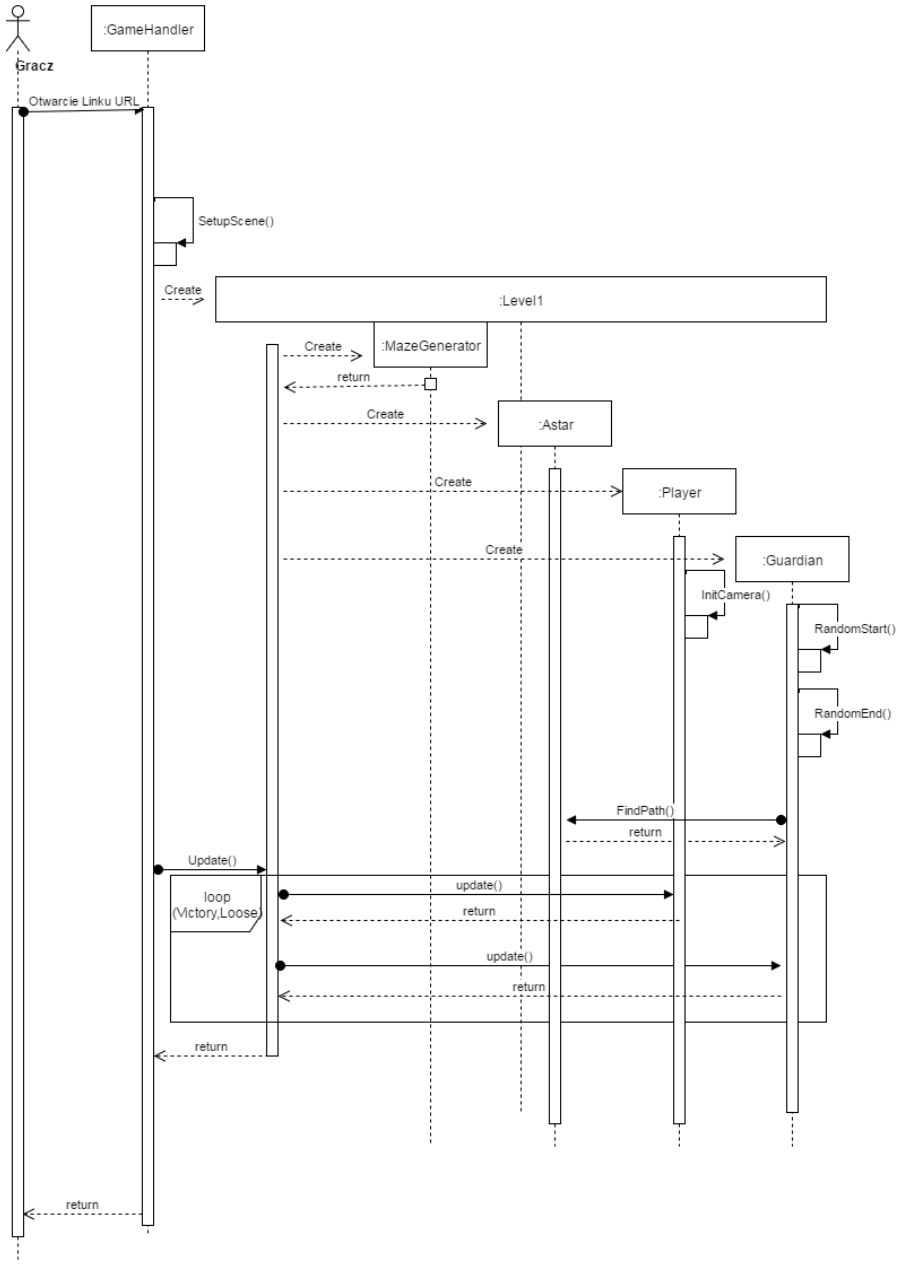


Rysunek 3.3: Diagram klas przedstawiający związki pomiędzy klasami (wersja ThreeJS). [Źródło : Opracowanie własne]

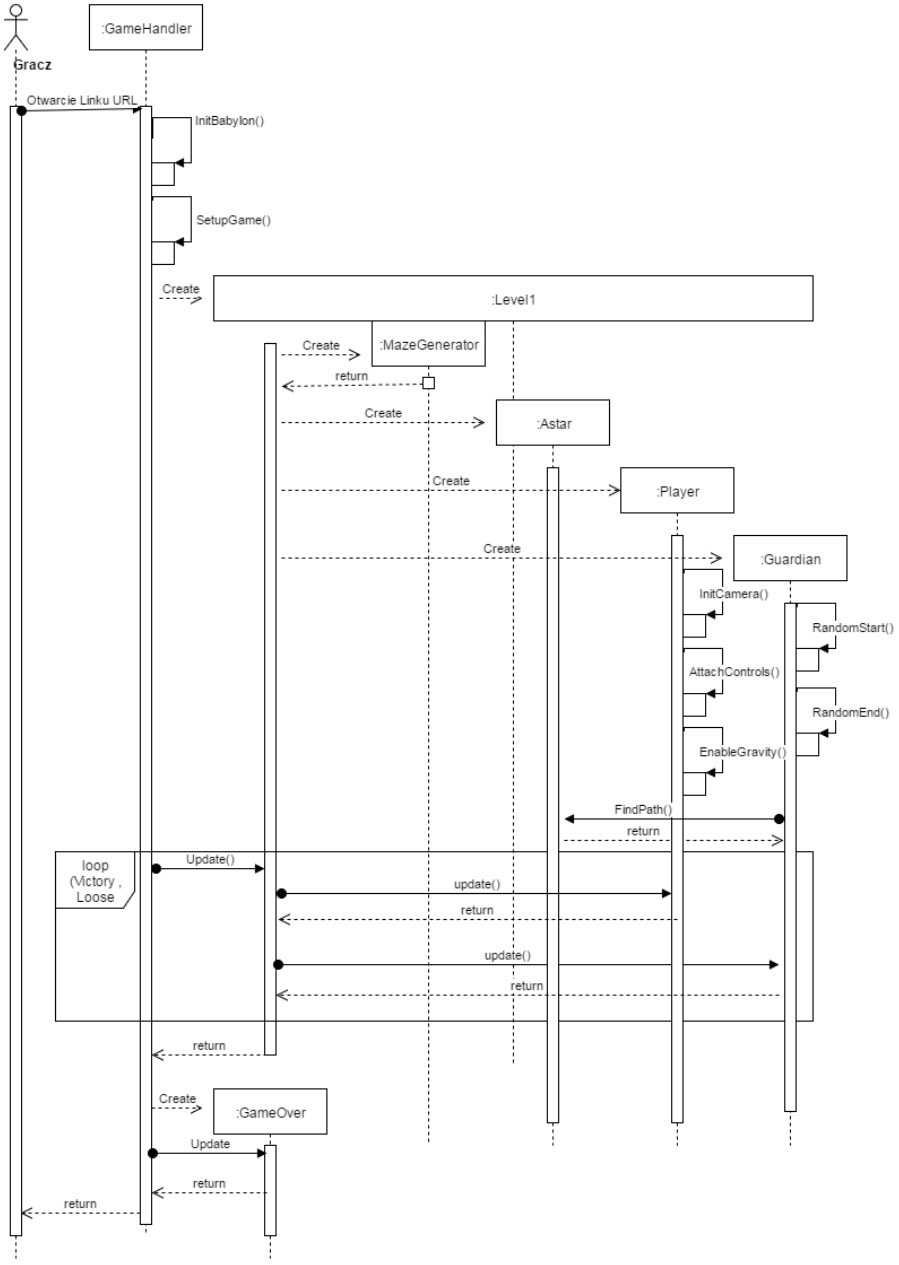


Rysunek 3.4: Diagram klas przedstawiający związki pomiędzy klasami (wersja BabylonJS). [Źródło : Opracowanie własne]

3.5 Diagramy Sekwencji



Rysunek 3.5: Diagram Sekwencji przedstawia interakcję pomiędzy obiektami (wersja ThreeJs) [Źródło: Opracowanie własne]



Rysunek 3.6: Diagram Sekwencji przedstawia interakcję pomiędzy obiektami (wersja BabylonJs) [Źródło: Opracowanie własne]

3.6 Omówienie elementów sztucznej inteligencji

W poprzednim rozdziale w którym porównywaliśmy silniki ze względu na ich funkcjonalności wspomnieliśmy o kryterium obsługi sztucznej inteligencji. Nie-

stety temat ten nie został wtedy rozwinięty z powodu braku obsługi SI w porównywalnych silnikach. Jednakże ze względu na obecność mechanizmów sztucznej inteligencji i ich dużej roli w grach komputerowych zdecydowałem się na przedstawienie zastosowanych przeze mnie rozwiązań w tej dziedzinie. Pierwszym ważnym elementem SI zastosowanym w mojej grze jest algorytm wyszukiwania najkrótszej ścieżki pomiędzy dwoma punktami A*. Algorytm odwiedza wszystkie możliwe punkty na drodze do celu starając się znaleźć taką ścieżkę której koszt opisany równaniem *f(x) = g(x) + h(x)* będzie najmniejszy. W kalkulacji kosztu ścieżki biorą udział dwa najważniejsze czynniki którymi są [1]:

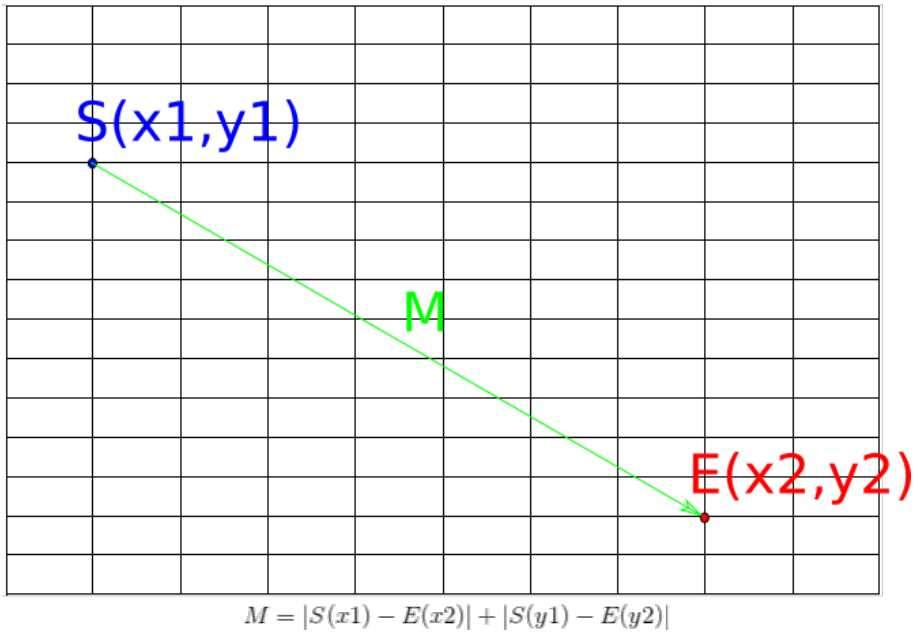
1. Koszt *g(x)* określający długość drogi począwszy od wierzchołka startowego do obecnego x[1].

2. Koszt *h(x)* będący wynikiem heurystyki przewidującej drogę z wierzchołka obecnego x do wierzchołka docelowego[1].

Czynnikiem sprawiającym iż a* jest najlepszą metodą poszukiwania najkrótszej drogi jest koszt *h(x)* reprezentowany przez heurystykę. Heurystyka w czysto teoretycznym znaczeniu jest po prostu metodą, która dąży do znalezienia rozwiązania problemu dla którego optymalny wynik nie jest gwarantowany. W kontekście omawianego algorytmu heurystyka oblicza odległość od wierzchołka x do docelowego, co nakierowuje program na optymalne rozwiązanie, pomijając obliczenia które nie zwróciły by wyniku lub prowadziły do wyników nie optymalnych . Jedną z najważniejszych rzeczy jeżeli chodzi o heurystykę w A* jest to aby nie przeszacowywała kosztu od wierzchołka x do punktu końcowego. Skutkować to może nieoptymalnym lub błędnym wynikiem algorytmu. W niniejszej implementacji użyta została heurystyka o nazwie *manhattan distance* będącą jedną z podstawowych metod wyliczania odległości na kwadratowej siatce. Poniższe obrazki zawierają pseudokod algorytmu i wyjaśnienie dotyczące heurystki *manhattan distance*. Pełny kod implementacji A* jest dostępny na płycie CD dołączonej do pracy.[22, 1].

```
function A*(start,goal)
    closedset := the empty set                 % Zbiór wierzchołków przejrzanych.
    openset := set containing the initial node % Zbiór wierzchołków nie odwiedzonych.
    g_score[start] := 0                        % Długość optymalnej trasy.
    while openset is not empty
        x := the node in openset having the lowest f_score[] value
        if x = goal
            return reconstruct_path(came_from,goal)
        remove x from openset
        add x to closedset
        foreach y in neighbor_nodes(x)
            if y in closedset
                continue
            tentative_g_score := g_score[x] + dist_between(x,y)
            tentative_is_better := false
            if y not in openset
                add y to openset
                h_score[y] := heuristic_estimate_of_distance_to_goal_from(y)
                tentative_is_better := true
            elseif tentative_g_score < g_score[y]
                tentative_is_better := true
            if tentative_is_better = true
                came_from[y] := x
                g_score[y] := tentative_g_score
                f_score[y] := g_score[y] + h_score[y] % Przewidywany dystans od startu do celu przez y.
    return failure

function reconstruct_path(came_from,current_node)
    if came_from[current_node] is set
        p = reconstruct_path(came_from,came_from[current_node])
        return (p + current_node)
    else
        return the empty path
```

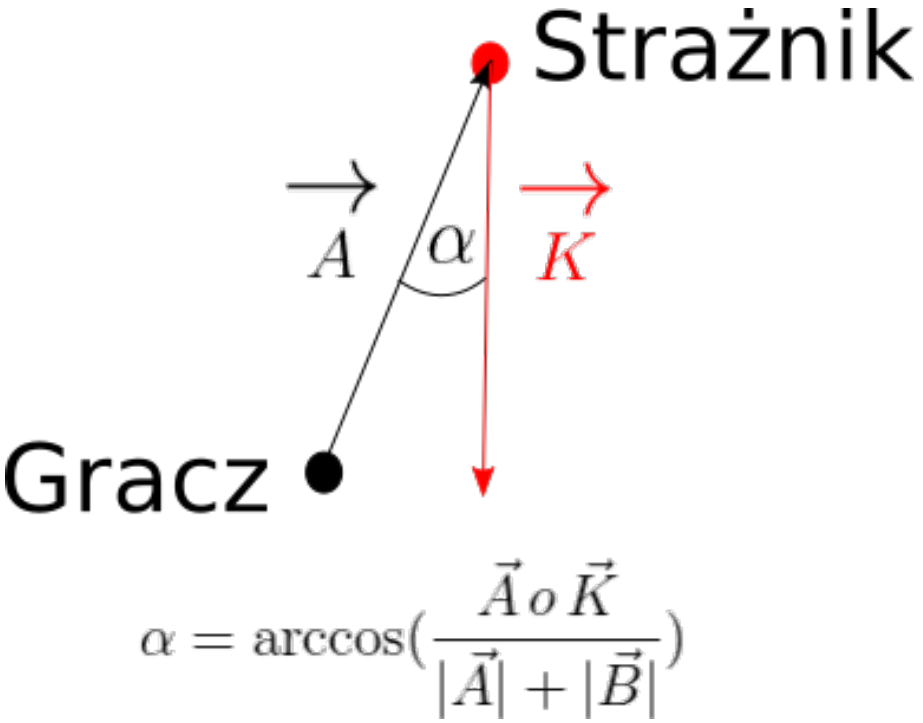
Rysunek 3.7: Ilustracja przedstawiająca pseudokod algorytmu znajdowania najkrótszej ścieżki pomiędzy dwoma punktami A. [Źródło : [1]]*



$$M = |S(x1) - E(x2)| + |S(y1) - E(y2)|$$

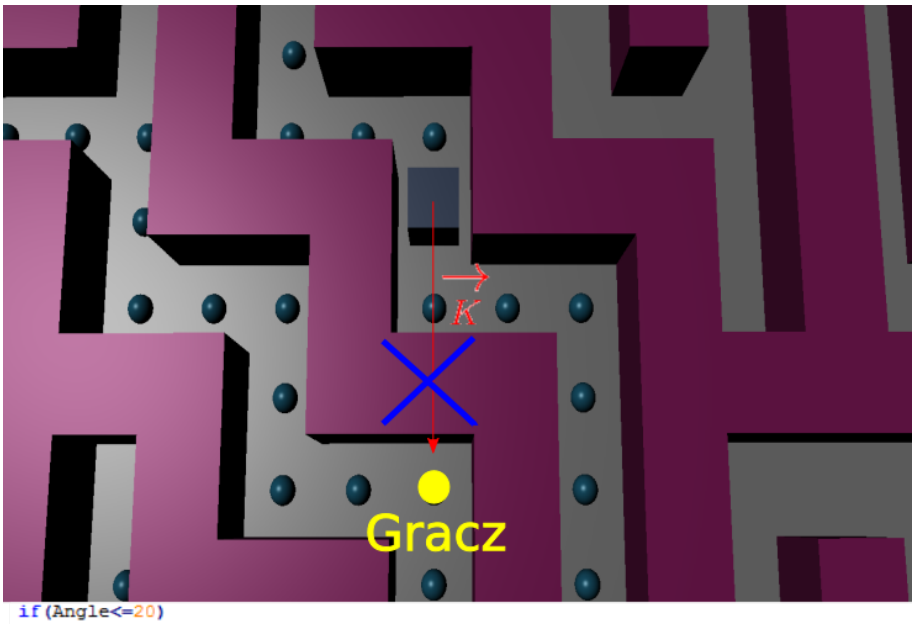
Rysunek 3.8: Ilustracja przedstawiająca heurystykę manhanttan distance polegającą na wyliczeniu. [Źródło : Opracowanie własne]

Drugim jak i zarazem ostatnim elementem sztucznej inteligencji użytej w grze jest wykrywanie gracza na podstawie jego pozycji względem strażnika. Najpierw obliczana jest długość wektora pomiędzy graczem a przeciwnikiem następnie ten sam proces powtarzany jest dla wektora kierunku strażnika. Ostatnim etapem jest wyliczenie iloczynu skalarnego tych dwóch wektorów który później jest dzielony przez sumę długości wektora gracza i strażnika. Finalnym krokiem jest podstawienie całego wyniku do funckji arccos dzięki której otrzymamy kąt pomiędzy graczem a strażnikiem[3].



Rysunek 3.9: Ilustracja przedstawia kalkulacje kąta alfa pomiędzy graczem a przeciwnikiem. [Źródło : Opracowanie własne]

Opisane powyżej obliczenia są niewystarczające ponieważ pomimo że mamy pewność iż gracz jest w zasięgu widzenia przeciwnika to nie wiemy czy linie widoku nie przesłania mu jakaś przeszkoda. Rozwiązaniem tego problemu jest wykonanie rzutu promienia (RayCast) w kierunku zgodnym z wektorem przemieszczenia strażnika o długości widzenia przeciwnika i sprawdzenie czy promień napotka jakąś przeszkodę.



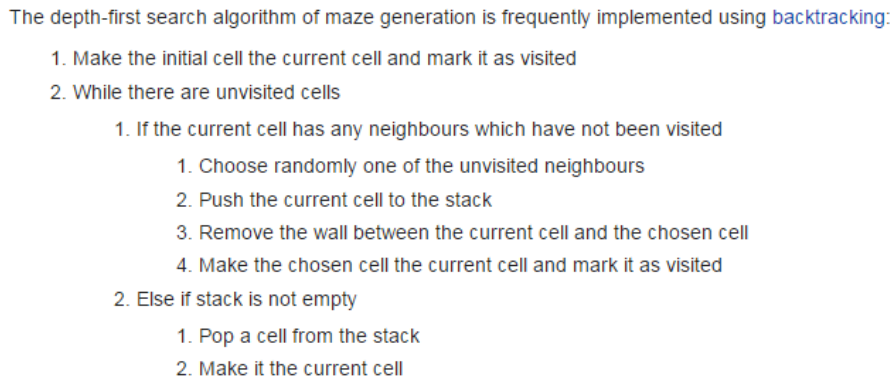

```
if(Angle<=20)
{
    //Sprawdź czy na lini widoku nie ma przeszkody
    var RayCast = new BABYLON.Ray(new BABYLON.Vector3(this.posX,0,this.posZ),
    new BABYLON.Vector3(this.targetX,0,this.targetZ),this.DetectRange);
    if(!RayCast.intersectsBox(this.Box))
    {

        return 2;// Zostałeś zauważony
    }
}
```

Rysunek 3.10: Ilustracja przedstawia problem braku czystej linii widzenia i jego rozwiązanie w postaci części kodu odpowiedzialnego za rzut promienia i detekcje w przypadku napotkania przeszkody. [Źródło : Opracowanie własne]

3.7 Proceduralne generowanie labiryntu

Większość kodu gry jest różna od siebie w zależności od wykorzystanego frameworka. Jednakże pewna jego część jest wspólna gdyż realizuje zadania które są niezależne od używanego silnika. Przykładem takiej części są opisane w powyższym podrozdziale elementy sztucznej inteligencji. W tym podrozdziale skupimy się na kolejnej części kodu niezależnej od frameworka realizującej proceduralne generowanie mapy jaką jest labirynt. Labiryntem zaś nazywamy zbiór obiektów na płaszczyźnie, które odpowiednio ułożone utworzą sieć ścieżek pośród których możliwe będzie znalezienie drogi pomiędzy dwoma wybranymi punktami. Istnieje wiele rodzajów labiryntów. Skupimy się jednak na typie nieperfekcyjnym. W normalnym typie labiryntu zwanym również perfekcyjnym nie ma pętli, to

znaczy że istnieje dokładnie jedna ścieżka z punktu A do B. Z kolei w labiryncie nieperfekcyjnym możliwe jest wytyczenie więcej niż jednej ścieżki pomiędzy punktem startowym a docelowym. Niniejsza implementacja gry korzysta z labiryntu nieperfekcyjnego generowanego za pomocą algorytmu recursive backtracker. Na początku algorytm generuje labirynt perfekcyjny, który jest następnie przekształcany losowo w typ nieperfekcyjny[30].



Rysunek 3.11: Ilustracja przedstawia algorytm recursive backtracker służący do generowania labiryntu perfekcyjnego. [Źródło : [30]]

Za przekształcenie labiryntu perfekcyjnego wygenerowanego przez algorytm przedstawiony na powyższej ilustracji odpowiada kod umieszczony poniżej.

```
MazeGenerator.prototype.MakeImperfect = function(Tab)
{
    for(var i = 1 ; i< this.n -1 ; i++)
    {
        for(var j = 1 ; j< this.m - 1; j++)
        {
            // | * |
            if(Tab[i][j-1].isWall == 0 && Tab[i][j+1].isWall == 0 && Tab[i][j].isWall == 1 )
            {
                if(Math.random() <= 0.05)
                {
                    Tab[i][j].isWall = 0 ;
                }
            }
            //  -
            //  *
            //  -
            if(Tab[i-1][j].isWall == 0 && Tab[i+1][j].isWall == 0 && Tab[i][j].isWall == 1 )
            {
                if(Math.random() <= 0.05)
                {
                    Tab[i][j].isWall = 0 ;
                }
            }
        }
    }
}
```

Rysunek 3.12: Ilustracja przedstawia kod konwertujący wygenerowany labirynt perfekcyjny na nieprefekcyjny. [Źródło : Opracowanie własne]

Jego zadaniem jest iteracja tablicy dwuwymiarowej reprezentującej labirynt w przypadku wykrycia sytuacji w której istnieją trzy fragmenty ściany zgrupowane w jednej konkretnej orientacji. W takim przypadku środkowy fragment zostanie usunięty dzięki czemu zostanie utworzona nowa ścieżka. Rodzaj takiego labiryntu został wybrany z powodu zwiększenia poziomu trudności gdyż nie jest już takie łatwe rozwiązanie problemu metodą odwiedzenia wszystkich odnóg i powrotu w przypadku napotkania ślepego zaułku. Pełny kod implementacji proceduralnego generowania labiryntu jest dostępny na płycie CD dołączonej do pracy.

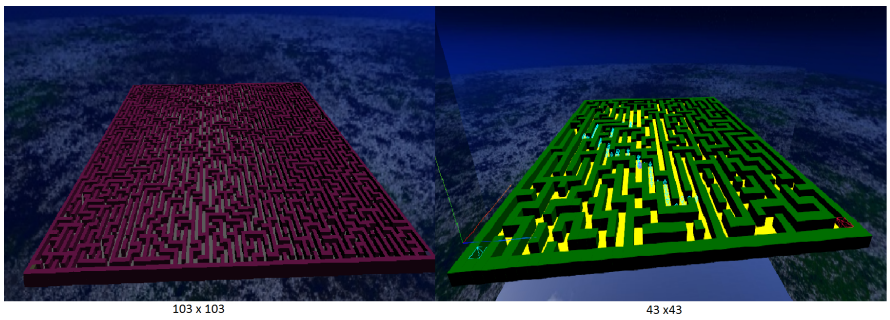
3.8 Test wydajności silników na podstawie implementacji

Poprzedni rozdział przybliżył nam różnice silników pod względem zdefiniowanych przez nas funkcjonalności. Jednakże czasami takie porównanie może być w przypadku programistów gier niewystarczające. Dzieje się tak ponieważ czasami przy projektowaniu niektórych gier najważniejsza jest ich wydajność co możemy wyrazić poprzez jak najwyższą ilość klatek na sekundę. W tym rozdziale skupimy się na analizie wydajności frameworków wykorzystując do tego celu implementację gry opisaną powyżej. Test będzie polegał na uruchomieniu

gry w dwóch frameworkach z różnym parametrem wielkości mapy oraz jednakową określoną rozdzielczością. Analize zostaną podane dwie zmienne : czas potrzebny do renderowania jednej ramki podany w milisekundach (frametime) i ilość wygenerowanych klatek na sekundę (Fps). Platformą na której zostaną przeprowadzone testy jest komputer stacjonarny o następujących parametrach:

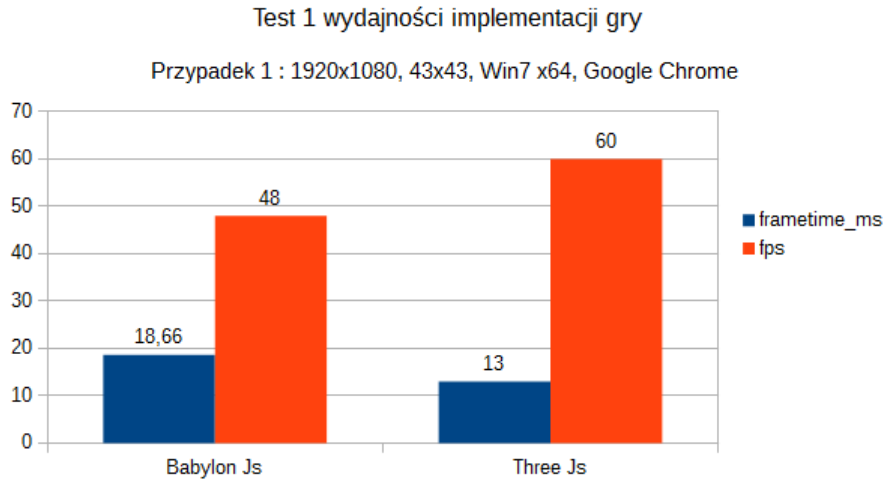
- Procesor : Intel Core i-5 4460 3.2GHz
- Pamięć RAM : GOOD RAM 2x4GB DDR3 1600Mhz CL9
- Karta Graficzna : Asus GeForce GTX 960 4GB STRIX OC
- Płyta Główna: Asus B85M-G
- System Operacyjny : Windows 7 Professional SP1 x64 / Ubuntu 16.04 LTS x64

Zanim przejdziemy do testów poniżej zaprezentowane zostaną testowane rozmiary map aby zobrazować nam wizualnie wyzwania wydajnościowe z jakim silniki będą zmuszone zmierzyć się.

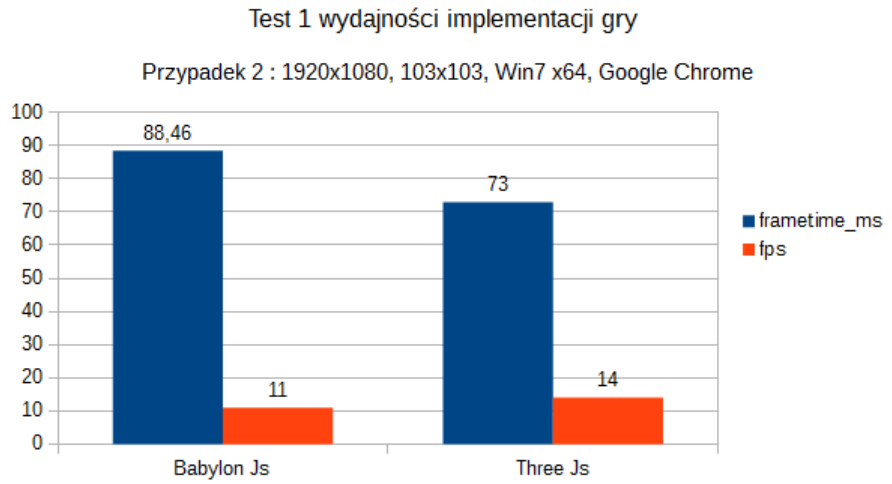


Rysunek 3.13: Ilustracja przedstawiająca rozmiar map w przypadku wielkości 103x103(po lewej stronie) i 43x43(prawa część obrazka) [Źródło : Opracowanie własne]

Wykonane zostaną łącznie cztery testy każdy z nich dzielący się na dwa przypadki charakteryzujące się zmianą wielkości mapy. Każdy test będzie różnił się od siebie rodzajem wykorzystanego systemu operacyjnego lub przeglądarki internetowej. Pozwoli nam to odpowiedzieć nie tylko na pytanie który framework ma lepszą wydajność ale również w jakich warunkach dany silnik wykazuję się lepszą wydajnością. Nie przedłużając rozpoczniemy od testu pierwszego który porówna wydajność silników na systemie operacyjnym windows 7 z wykorzystaniem przeglądarki Google Chrome.

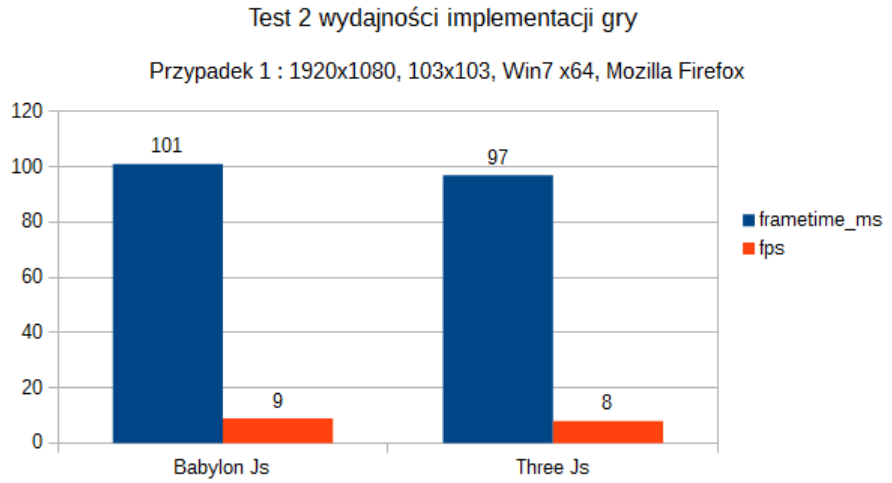


Rysunek 3.14: Wykres przedstawiający osiągi silników w przypadku parametrów opisanych w podtytule [Źródło : Opracowanie własne]

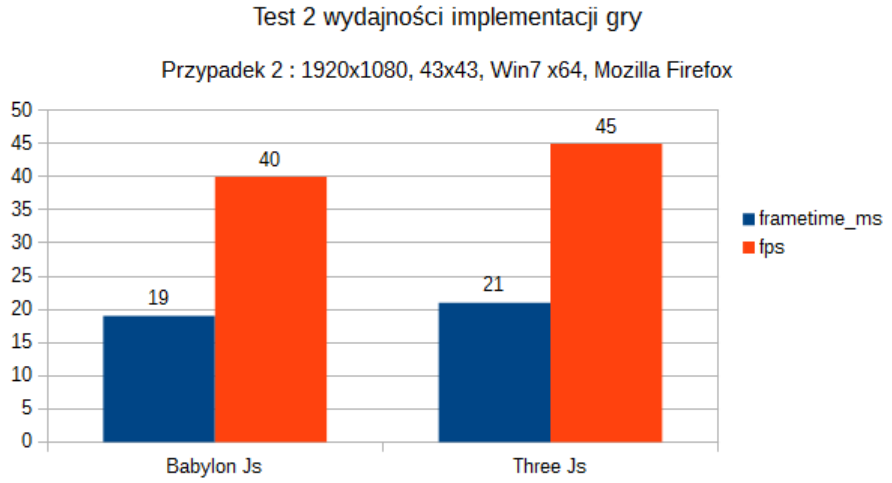


Rysunek 3.15: Wykres przedstawiający osiągi silników w przypadku parametrów opisanych w podtytule [Źródło : Opracowanie własne]

Po pierwszym teście możemy stwierdzić dużą przewagę wydajnościową (ok 20%) three js nad babylon js wykorzystującej przeglądarkę google chrome. Kolejny test przeprowadzimy na tym samym systemie operacyjnym zmianie ulegnie tylko przeglądarka internetwowa która będzie mozilla firefox.

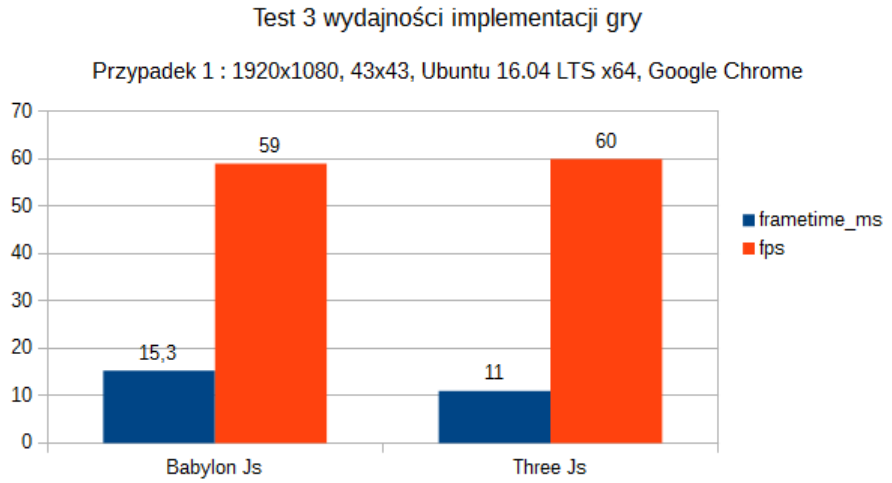


Rysunek 3.16: Wykres przedstawiający osiągi silników w przypadku parametrów opisanych w podtytule [Źródło : Opracowanie własne]

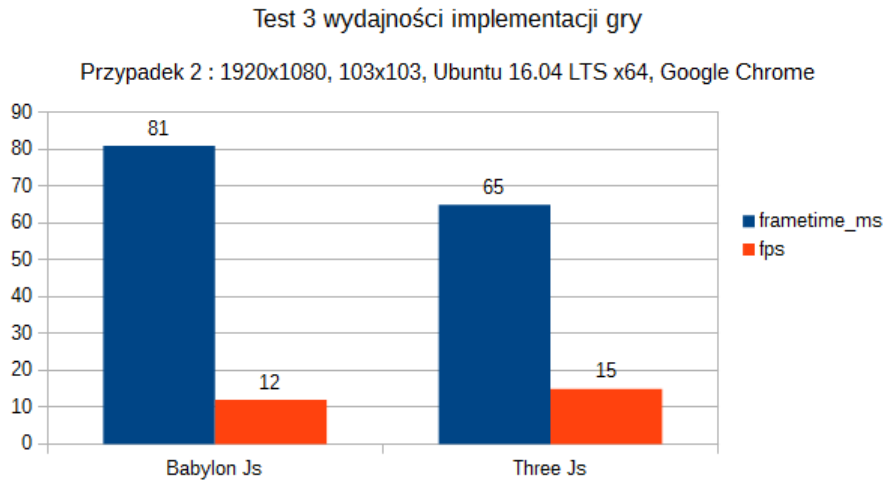


Rysunek 3.17: Wykres przedstawiający osiągi silników w przypadku parametrów opisanych w podtytule [Źródło : Opracowanie własne]

W porównaniu do poprzedniego testu możemy zauważyć ogólny spadek wydajności do 25% głównym powodem jest tutaj zmiana przeglądarki na mozille firefox która radzi sobie gorzej z renderowaniem i przetwarzaniem danych niż google chrome. Ponad to, w przypadku dużej mapy i threejs wystąpiły poważne problemy ze stabilnością przeglądarki.

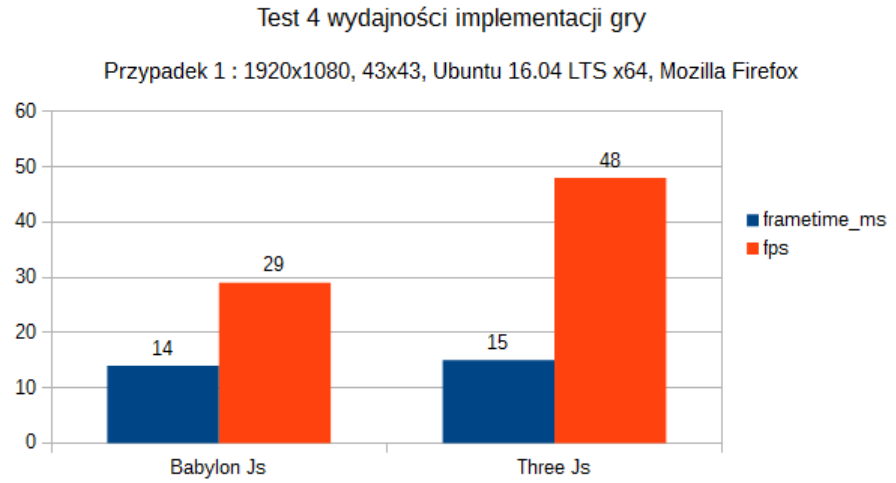


Rysunek 3.18: Wykres przedstawiający osiągi silników w przypadku parametrów opisanych w podtytule [Źródło : Opracowanie własne]

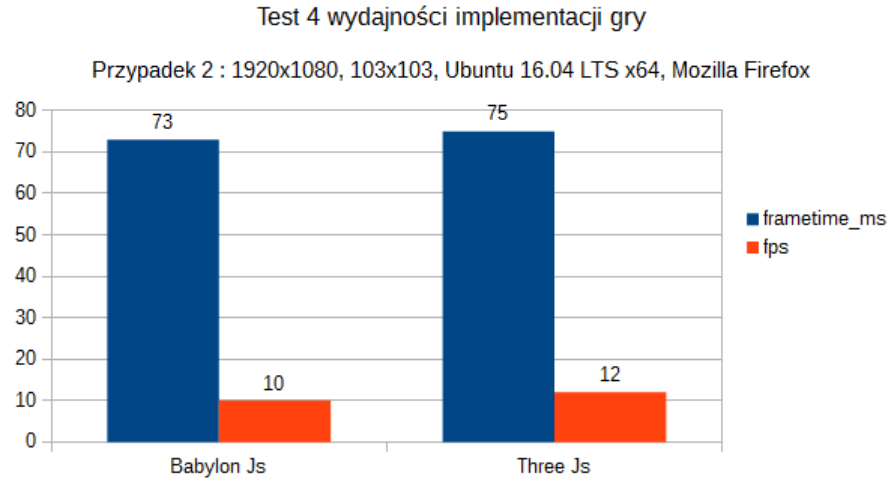


Rysunek 3.19: Wykres przedstawiający osiągi silników w przypadku parametrów opisanych w podtytule [Źródło : Opracowanie własne]

Po przeanalizowaniu danych z tego testu, można stwierdzić iż zmiana systemu operacyjnego poprawiła osiągi babylona na małej mapie, jednakże ze względu na to, iż wyniki dla większej mapy pokrywają się, można uznać to za anomalię.



Rysunek 3.20: Wykres przedstawiający osiągi silników w przypadku parametrów opisanych w podtytule [Źródło : Opracowanie własne]



Rysunek 3.21: Wykres przedstawiający osiągi silników w przypadku parametrów opisanych w podtytule [Źródło : Opracowanie własne]

Ostatni test nie wyróżnia się za bardzo spośród innych oprócz dość dużego spadku wydajności w przypadku babylona dla małej mapy, niestety nie byłem w stanie stwierdzić przyczyny takiego stanu.

Rozdział 4

Podsumowanie pracy

Głównym celem powyższej pracy inżynierskiej było porównanie dwóch frameworków do tworzenia gier sieciowych w języku javascript i wykonanie przykładowych implementacji gier. Wybór padł na dwa popularne silniki którymi są Babylonjs i Threejs, głównie ze względu na brak opłat związanych z ich użytkowaniem. Rozdział pierwszy niniejszej pracy przybliża podstawowe zagadnienia towarzyszące grafice komputerowej i silnikom. W następnym rozdziale dokonane zostało porównanie silników na podstawie kryteriów zdefiniowanych również w tym samym rozdziale. Ostatni rozdział przedstawia szczegóły dotyczące implementacji gier oraz testy wydajnościowe oparte na wspomnianej grze. Przedstawione w niniejszej pracy zagadnienia i kryteria dotyczące silników nie wyczerpują całkowicie tematu gdyż obecnych jest wiele innych funkcjonalności dostępnych w opisywanych silnikach. Przyczyną braku ich opisania było skupienie się na funkcjonalnościach i zagadnieniach które według mnie są najważniejsze z punktu widzenia dewelopera gier między innymi: obsługa fizyki, sztucznej inteligencji, wykrywania kolizji, modelu płatności, sposobu licencjonowania i pozostałych zagadnień omówionych w tejże pracy. Porównanie przedstawionych funkcjonalności polegało na ustaleniu obecności danego kryterium i stwierdzeniu w którym przypadku zostało ono zrealizowane w dokładniejszym stopniu. Gdy poprzedni sposób okazywał się niewystarczający ze względu na specyfikę danego kryterium było ono wówczas oceniane pod kątem przydatności dla dewelopera. Największymi problemami z którymi przyszło się zmierzyć podczas opracowywanie powyższego zagadnienia mojej pracy były:

1. Ustalenie najważniejszych kryteriów z punktu widzenia dewelopera gier
2. Brak dodania przez autora three js przykładowych funkcjonalności rozszerzających do główniej biblioteki co sprawia że zainteresowani użytkownicy

muszą dokładnie przeszukać całe repozytorium aby dowiedzieć się o istnieniu tychże funkcjonalności

Kolejnym sposobem porównania frameworków było przeprowadzenie testów wydajnościowych na podstawie których stwierdzam iż w tej materii zwycięża three js. Deweloperzy którzy cenią przede wszystkim płynność gier, powinni wybrać właśnie ten silnik a jeżeli cenią wygodę tworzenia to polecam babylona js. Mimo wystąpienia powyżej opisanych komplikacji, mogę z całą stanowczością stwierdzić osiągnięcie głównego celu, jakim było porównanie framewokrów do tworzenia gier sieciowych w języku javascript. Wykonano również implementacje gier w obydwu silnikach, która została szczegółowo opisana w rozdziale trzecim. Podsumowując na podstawie kryteriów jak i również własnych doświadczeń po przeprowadzeniu porównania silników stwierdzam, iż najlepszym frameworkiem do tworzenia gier spośród opisanych powyżej jest babylon js.

Spis rysunków

Rozdział 5

Bibliografia

[1] Algorytm a*. https://pl.wikipedia.org/wiki/Algorytm_A*. Dostęp: 24-04-2016.

[2] Antyaliasing. https://pl.wikipedia.org/wiki/Antyaliasing. Dostęp: 24-04-2016.

[3] Maths - angle between vectors. http://www.euclideanspace.com/maths/algebra/vectors/angleBetween/. Dostęp: 24-04-2016.

[4] Aliasing. https://pl.wikipedia.org/wiki/Aliasing_(grafika_komputerowa). Dostęp: 24-04-2016.

[5] Apache license. https://pl.wikipedia.org/wiki/Apache_License. Dostęp: 24-04-2016.

[6] Bryła brzegowa otaczająca model głowy. https://pl.wikipedia.org/wiki/Bry%C5%82a_brzegowa#/media/File:BoundingBox.jpg. Dostęp: 24-04-2016. File: BoundingBox.jpg.

[7] Babylon js documentation. http://doc.babylonjs.com/. Dostęp: 24-04-2016.

[8] Babylon.js. https://github.com/BabylonJS/Babylon.js. Dostęp: 24-04-2016.

[9] Lights. https://doc.babylonjs.com/tutorials/Lights. Dostęp: 24-04-2016.

[10] In-browser mesh simplification (auto-lod). https://doc.babylonjs.com/tutorials/In-Browser_Mesh_Simplification_(Auto-LOD). Dostęp: 24-04-2016.

[11] Babylon js lod playground example. http://www.babylonjs-playground.com/#1ED15P#1. Dostęp: 24-04-2016.

[12] Babylon js material editor. http://materialeditor.raananweber.com/. Dostęp: 24-04-2016.

[13] Babylon js playground. http://www.babylonjs-playground.com/. Dostęp: 24-04-2016.

[14] Craig Chapple. How low can the big three game engines go ? http://www.develop-online.net/analysis/how-low-can-the-big-three-game-engines-go/0205400, 4 2015. Dostęp: 24-04-2016.

[15] Wykrywanie kolizji. https://pl.wikipedia.org/wiki/Wykrywanie_kolizji. Dostęp: 24-04-2016.

[16] Babylon js cyos. http://www.babylonjs.com/cyos/. Dostęp: 24-04-2016.

[17] David Davidović. The end of fixed-function rendering pipeline. http://gamedevelopment.tutsplus.com/articles/the-end-of-fixed-function-rendering-pipelines-and-how-to-move-on--cms-21469, 7 2014. Dostęp: 24-04-2016.

[18] Ecmascript. https://pl.wikipedia.org/wiki/ECMAScript. Dostęp: 24-04-2016.

[19] Silnik fizyki. https://pl.wikipedia.org/wiki/Silnik_fizyki. Dostęp: 24-04-2016.

[20] Fotorealizm (grafika komputerowa). https://pl.wikipedia.org/wiki/Fotorealizm_(grafika_komputerowa). Dostęp: 24-04-2016.

[21] Game Engine. https://en.wikipedia.org/wiki/Game_engine. Dostęp: 24-04-2016.

[22] Heurystyka. https://pl.wikipedia.org/wiki/Heurystyka_(informatyka). Dostęp: 24-04-2016.

[23] Jason Gregory. *Game Engine Architecture*, chapter 7, pages 340–341. Taylor and Francis Group, Boca Raton, 2 edition, 2015.

[24] Jason Gregory. *Game Engine Architecture*, chapter 7, pages 348–349. Taylor and Francis Group, Boca Raton, 2 edition, 2015.

[25] Jason Gregory. *Game Engine Architecture*, chapter 10, page 448. Taylor and Francis Group, Boca Raton, 2 edition, 2015.

[26] Javascript. https://pl.wikipedia.org/wiki/JavaScript. Dostęp: 24-04-2016.

[27] Oświetlenie (grafika komputerowa). https://pl.wikipedia.org/wiki/O%C5%9Bwietlenie_(grafika_komputerowa). Dostęp: 24-04-2016.

[28] Radosław Mantiuk. wykład gk potok graficzny. http://rmantiuk.strony.wi.ps.pl/data/wyklad_gk_potok_graficzny.pdf. Dostęp: 24-04-2016.

[29] Radosław Mantiuk. wykład gk potok graficzny. http://rmantiuk.strony.wi.ps.pl/data/wyklad_gk_potok_graficzny.pdf. Dostęp: 24-04-2016. Slajd: 9.

[30] Maze generation algorithm. https://en.wikipedia.org/wiki/Maze_generation_algorithm. Dostęp: 24-04-2016.

[31] Wojciech Muła. Antyaliasing: przykład rasteryzacji trójkąta. https://pl.wikipedia.org/wiki/Rasteryzacja#/media/File:Rasterisation-triangle_example.svg. Dostęp: 24-04-2016. Plik:Rasterisation-triangle example.svg.

[32] Wojciech Muła. Kula oświetlona przez trzy światła. po lewej zwykłe oświetlenie, po prawej – światła wolumetryczne. https://pl.wikipedia.org/wiki/%C5%9Awiat%C5%82o_wolumetryczne#/media/File:Volumetric_lights_example.png. Dostęp: 24-04-2016. Plik:Volumetric lights example.png.

[33] Wojciech Muła. Przykład rasteryzacji odcinka, łuku oraz wielokąta. https://pl.wikipedia.org/wiki/Rasteryzacja#/media/File:Rasterization_bw.svg. Dostęp: 24-04-2016. Plik:Rasterization bw.svg.

[34] Wolne oprogramowanie. https://pl.wikipedia.org/wiki/Wolne_oprogramowanie. Dostęp: 24-04-2016.

[35] Richard S.Wright Jr., Benjamin Lipchak. *OpenGL Księga eksperta*, volume 1, chapter 2, pages 61–63. Helion, Gliwice, 3 edition, 2005. tłum. Wojciech Moch, Rafał Jońca, Marek Pętlicki.

[36] Richard S.Wright Jr., Benjamin Lipchak. *OpenGL Księga eksperta*, volume 1, chapter 5, pages 234–245. Helion, Gliwice, 3 edition, 2005. tłum. Wojciech Moch, Rafał Jońca, Marek Pętlicki.

[37] Babylon js sandbox. http://www.babylonjs.com/sandbox/. Dostęp: 24-04-2016.

[38] Silnik Gry. https://en.wikipedia.org/wiki/Game_engine. Dostęp: 24-04-2016.

[39] Silnik Graficzny. https://pl.wikipedia.org/wiki/Silnik_graficzny. Dostęp: 24-04-2016.

[40] Fragment (computer graphics). https://en.wikipedia.org/wiki/Fragment_(computer_graphics). Dostęp: 24-04-2016.

[41] Documentation. http://threejs.org/docs/. Dostęp: 24-04-2016.

[42] Texture filtering. https://en.wikipedia.org/wiki/Texture_filtering. Dostęp: 24-04-2016.

[43] Three js editor. http://threejs.org/editor/. Dostęp: 24-04-2016.

[44] Three js github. https://github.com/mrdoob/three.js/. Dostęp: 24-04-2016.

[45] Three.js. https://en.wikipedia.org/wiki/Three.js. Dostęp: 24-04-2016.

[46] Tomas Akenine-Möller, Eric Haines, Naty Hoffman. *Real-Time Rendering*, chapter 2, pages 11–26. Taylor and Francis Group, Boca Raton, 3 edition, 2008.

[47] Video game bot. https://en.wikipedia.org/wiki/Video_game_bot. Dostęp: 24-04-2016.

[48] Volumetric lighting. https://en.wikipedia.org/wiki/Volumetric_lighting. Dostęp: 24-04-2016.

[49] Licencja x11. https://pl.wikipedia.org/wiki/Licencja_X11. Dostęp: 24-04-2016.

Dodatek A

Zawartość płyty CD

Do niniejszej pracy inżynierskiej dołączona została płyta CD zawierająca następujące elementy:

- Pracę inżynierską w formacie elektronicznym (plik pdf) zgodną z wersją papierową
- Implementacje gry wykonaną w obydwu frameworkach
- Wszytkie kody źródłowe klas wykorzystanych w implementacji gier i opisanych w niniejszej pracy