Akademia Górniczo-Hutnicza
im. Stanisława Staszica w Krakowie

Wydział Elektrotechniki, Automatyki, Informatyki i Inżynierii Biomedycznej

KATEDRA INFORMATYKI STOSOWANEJ

PRACA MAGISTERSKA

ADAM RZEPKA

ANALIZA MOŻLIWOŚCI WYKORZYSTANIA NOWYCH TECHNOLOGII ZAWARTYCH W HTML5 DO REALIZACJI GRY TYPU FPS

PROMOTOR:
dr inż. Grzegorz Rogus

Kraków 2013

OŚWIADCZENIE AUTORA PRACY

OŚWIADCZAM, ŚWIADOMY ODPOWIEDZIALNOŚCI KARNEJ ZA POŚWIADCZENIE NIEPRAWDY, ŻE NINIEJSZĄ PRACĘ DYPLOMOWĄ WYKONAŁEM OSOBIŚCIE I SAMODZIELNIE, I NIE KORZYSTAŁEM ZE ŹRÓDEŁ INNYCH NIŻ WYMIENIONE W PRACY.

. . . . . .

PODPIS

AGH

University of Science and Technology in Krakow

Faculty of Electrical Engineering, Automatics, Computer Science and Biomedical Engineering

DEPARTMENT OF APPLIED COMPUTER SCIENCE

MASTER OF SCIENCE THESIS

ADAM RZEPKA

ANALYSIS

SUPERVISOR:
Grzegorz Rogus Ph.D

Krakow 2013

podziękowania

Spis treści

1. Wprowadzenie

Standard HTML 5 otwiera przed twórcami aplikacji internetowych nowe możliwości. Technologie z nim związane, takie jak WebGL, spowodowały, że również autorzy zaawansowanych gier komputerowych zaczęli przyglądać się platformie internetowej, w nadziei znalezienia nowego rynku dla swoich produktów.

Niniejsza praca jest próbą zbadania tego tematu przez osobę należącą do branży twórców gier. Podczas tworzenia gry w technologii HTML 5 zostanie położony nacisk na porównanie z tradycyjnymi grami, a następnie podsumowanie zalet i wad nowego podejścia.

1.1. Cele pracy

Praca ma następujące cele:

1. Stworzenie gry typu FPS z wykorzystaniem technologii zawartych w HTML 5.
2. Analiza możliwości tych technologii w porównaniu do tradycyjnego podejścia do tworzenia gier.

1.2. Teza pracy

Teza pracy:
Technologia HTML 5 umożliwia tworzenie zaawansowanych gier 3D.

1.3. Zawartość pracy

W następnym rozdziale zostaną omówione kwestie związane z tematem pracy. W pierwszej części rozdziału następuje przedstawienie HTML 5 i związanych z nim technologii istotnych z punktu widzenia tworzenia gry. Następnie będą wyjaśnione pojęcia związane z grami komputerowymi – gry typu FPS, silnik gry, czy tryb multiplayer. Rozdział kończy się zaprezentowaniem kilku poprzednich prób związanych z grami HTML 5.

Rozdział 3 to założenia co do tworzonego projektu oraz omówienie budowy jako gry komputerowej oraz aplikacji sieciowej.

W rozdziale 4 znajduje opis implementacji gry, z omówieniem poszczególnych komponentów. W tym rozdziale zostaje też krótko przedstawiona gra z perspektywy końcowego użytkownika.

Kolejny rozdział prezentuje analizę stworzonego projektu. Skupia się przede wszystkim na różnicach w procesie tworzenia gry HTML 5 i gry działającej natywnie. Wymienia zalety i wady nowego podejścia.

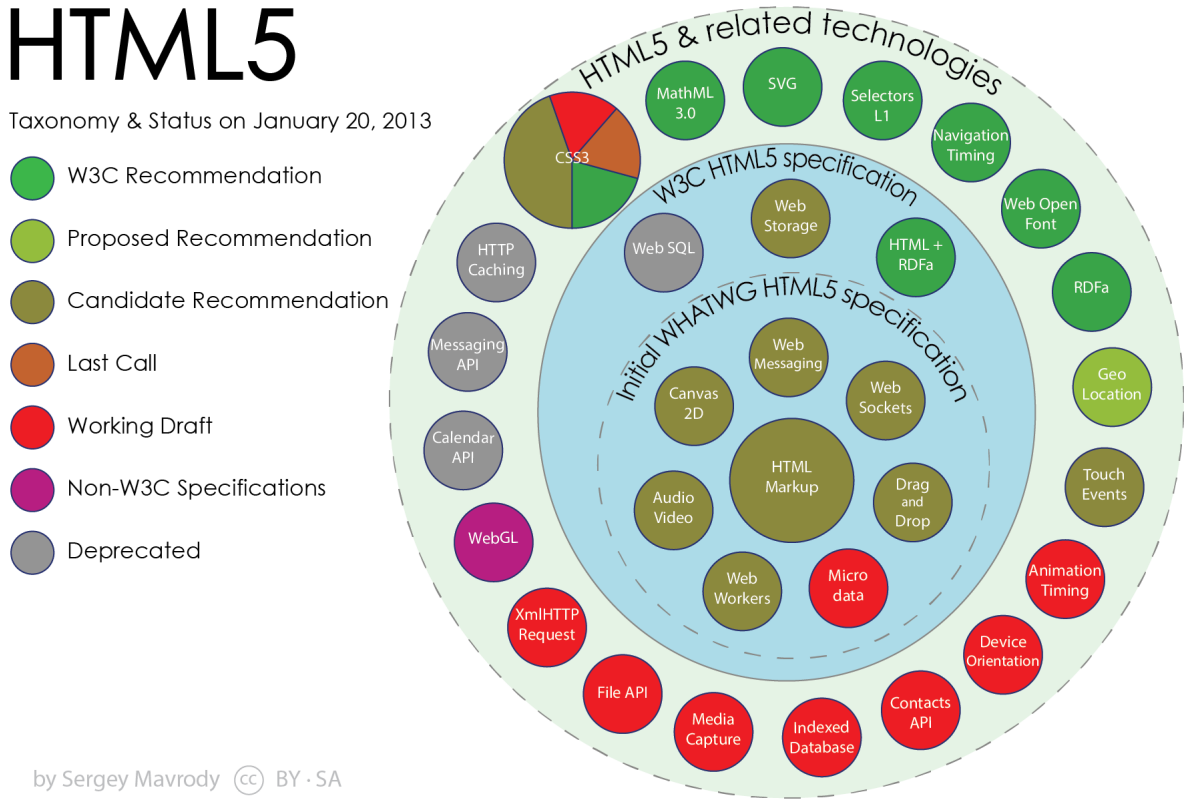
Ostatni rozdział to podsumowanie pracy z oceną realizacji celów i prawdziwości tezy pracy.

2. Omówienie dziedziny

2.1. HTML5

HTML5 jest kolejną wersją języka HTML służącego to tworzenia stron WWW, opracowywany przez World Wide Web Consortium (W3C) i Web Hypertext Application Technology Working Group (WHATWG). Standard jest w większości ukończony i jego ostateczna specyfikacja ma być wydana do końca roku 2014, natomiast w ciągu kolejnych dwóch lat ma nastąpić wydanie wersji 5.1. Jest on pomyślany jak następca HTML 4.1 i XHTML 1.0.

HTML5 wprowadza wiele nowych tagów (np. canvas, video, audio) oraz interfejsów programistycznych wymienionych w dalszych podrozdziałach. Wraz z nowymi możliwościami HTML5 wraz z językiem JavaScript, stał się platformą umożliwiającą tworzenie rozbudowanych aplikacji oraz gier.



Rysunek 2.1: HTML5 i związane z nim technologie oraz ich status.

Pełna specyfikacja HTML5 znajduje się w [13].

2.1.1. JavaScript

JavaScript jest skryptowym językiem programowania osadzony w przeglądarkach internetowych i wykorzystywany do tworzenia interaktywnych stron oraz aplikacji internetowych. Formalnie nie jest związany z HTML5, jednak jest jedynym językiem obsługiwanym przez wszystkie przeglądarki, dlatego wszystkie wymienione w dalszych podrozdziałach interfejsy programistyczne są stworzone dla tego języka.

JavaScript powstał 1995 w firmie Netscape, a następnie został ustandaryzowany przez ECMA (stąd stosowana formalnie nazwa ECMAScript). Nazwa i składnia przypomina Javę, jednak jest to podobieństwo mylące. JavaScript jest językiem dynamicznym, posiadającym wiele cech języków funkcyjnych, a obiektowość jest oparta na prototypach, a nie klasach. Specyfikacja języka ECMAScript jest publicznie dostępna na stronie [2], natomiast wyczerpujące wprowadzenie można znaleźć w [3].

Początkowo JavaScript był używany do wyświetlania prostych animacji, komunikatów i wstępnej walidacji danych w formularzach. Z w JavaScripcie czasem zaczęły powstawać coraz bardziej rozbudowane aplikacje i okazało się, że wydajność interpretowanego języka skryptowego z Garbage Collectorem (odśmiecaczem) jest coraz większym problemem. Rozpoczął się wyścig pomiędzy twórcami przeglądarek o jak najszybszy silnik skryptowy. W rezultacie, obecnie w większości przeglądarek skrypty są kompilowane do kodu natywnego przez kompilator JIT (Just In Time), z zastosowaniem wielu technik optymalizacji kodu, a Garbage Collector jest coraz szybszy. Dostępne są również rozbudowane narzędzia do profilowania kodu JavaScript.

Ta sytuacja sprawiła, że JavaScript zaczął się nadawać do robienia wymagających obliczeniowo gier 3d. Oczywiście wciąż trzeba unikać szczególnie nieefektywnych konstrukcji tego języka i starać się alokować jak najmniej obiektów, aby ograniczyć przestoje powodowane przez działanie odśmiecacza. Jednak przy zachowaniu tych zasad, wydajność współczesnych silników JavaScript powinna być wystarczająca dla wielu gier.

asm.js

asm.js jest to podzbiór JavaScript opracowany przez Mozillę, który zapewnia najszybsze wykonanie. Programowanie zgodnie z asm.js jest jednak bardzo żmudne, dlatego podzbiór ten jest głównie pomyślany jak cel kompilatorów innych języków do JavaScript (np. Emscripten [link/footnote]). Nie istnieje obecnie translator dowolnego kodu JavaScript do asm.js. Ponieważ gra będąca przedmiotem niniejszej pracy powstaje w JavaScript, tematyka asm.js nie będzie szerzej omawiana. Więcej informacji można znaleźć na stronie internetowej [7].

2.1.2. WebGL

WebGL jest to API dla języka JavaScript służące do wyświetlania grafiki 3d w przeglądarce. Z tego powodu jest to najbardziej istotna technologia z punktu widzenia twórców gier. Pomimo, że formalnie WebGL nie jest jeszcze częścią standardu HTML5, to jest on zaimplementowany we wszystkich znaczących przeglądarkach.

WebGL bazuje na standardzie OpenGL ES 2.0 (Open Graphics Library for Embedded Systems) przeznaczonym dla urządzeń mobilnych, który z kolei jest uproszczoną wersją OpenGL (Open Graphics Library) – otwartego API do renderowania grafiki 3d, będącym jednym z dwóch (obok Direct3D) szeroko stosowanych API do wyświetlania grafiki w grach i programach wykorzystujących 3d. Wszystkie te standardy zostały opracowane przez konsorcjum Khronos Group (wcześniej ARB), zrzeszające wszystkie (poza Microsoftem) większe firmy zainteresowane tematem grafiki 3d.

Prace nad WebGL rozpoczęła Mozilla w 2006 roku. A w następnym roku Firefox oraz Opera miały już pierwsze działające implementacje. Były one kontynuowane w grupie roboczej WebGL Working Group działającej w ramach Khronos Group z udziałem m. in. Mozilli, Opery, Google oraz Apple. W 2011 roku ukończona została pierwsza wersja standardu, a w 2013 ruszyły prace nad wersją 2.0 bazującą na OpenGL 3.0. W tym też roku Microsoft wydał Internet Explorer 11 z obsługą WebGL, pomimo początkowej niechęci do tego standardu (Microsoft promuje swoją technologię Direct3D, konkurencyjną w stosunku do OpenGL). Tym samym ostatnia z liczących się przeglądarek dodała obsługę WebGL.

WebGL zaprojektowano dla języka JavaScript, w przeciwieństwie do OpenGL i OpenGL ES, które przeznaczone są głównie dla języka C. Jednakże jest interfejsem bardzo niskopoziomowym, w zasadzie dokładnie przełożonym na JavaScript OpenGL ES. Dodano jedynie kilka usprawnień (jak wczytywanie tekstury z elementu HTML <img>) oraz dodatkową walidację danych, istotną w środowisku tekstowym. Z tego powodu programiści JavaScript mogą uznać API za trudne i nieprzystępne. Z drugiej jednak strony, taka implementacja gwarantuje maksymalną szybkość działania, a programiści znający OpenGL mogą w zasadzie natychmiast zacząć używać WebGL. Specyfikacja WebGL znajduje się na stronie [5].

2.1.3. Web Workers

Współcześnie każdy domowy komputer posiada wielordzeniowy procesor. W tym samym kierunku podążają procesory w urządzeniach mobilnych. Aby maksymalnie wykorzystać moc obliczeniową, konieczne jest tworzenie aplikacji współbieżnych. Niestety JavaScript przez długi czas tego nie umożliwiał. Zmienia to API Web Workers należące do standardu HTML5.

Worker jest to odpowiednik wątku dla języka JavaScript. Jednakże jego sposób działania upodabnia go bardziej do procesu – poszczególne workery nie mogą współdzielić pamięci i komunikują się wyłącznie za pomocą wysyłanych wiadomości. Dzięki takiej konstrukcji można uniknąć wielu problemów związanych z klasyczną wielowątkowością, jednak odbywa się to kosztem wydajności z powodu konieczności częstego kopiowania danych. Dodatkowo API dostępne dla workera jest bardzo ograniczone (np. nie ma dostępu do elementów HTML).

Te cechy powodują, że aby osiągnąć zysk wydajnościowy przy pomocy Web Workers, trzeba od początku projektować aplikację pod kątem tego API – tak aby ograniczyć ilość przesyłanych danych.

Należy zaznaczyć, iż taki sposób tworzenia aplikacji wielowątkowych jest nieobcy programistom gier, gdyż podobny model wymuszał procesor Cell w konsoli PlayStation3. Procesor ten składa się z jednego głównego rdzenia (tzw. Power Processing Element) oraz kilku rdzeni wspomagających (tzw. Synergistic Processing Element), które również nie mogą współdzielić pamięci. Podobnie jak Web Workerze, tak w programie działającym na SPE, dostępne API jest bardzo ograniczone.

2.1.4. Web Sockets

Wiele współczesnych gier umożliwia wspólną rywalizację wielu graczy za pośrednictwem internetu (tzw. multiplayer). W celu implementacji gry sieciowej konieczne jest API do komunikacji internetowej.

Interfejs Web Sockets jest odpowiednikiem systemowych gniazd sieciowych i umożliwia komunikację w czasie rzeczywistym pomiędzy aplikacją JavaScript, a serwerem za pośrednictwem protokołu TCP. Jest on częścią HTML5 i dostępny we wszystkich znaczących przeglądarkach.

2.1.5. WebRTC

Wprowadzenie interfejsu Web Sockets było istotnym krokiem z punktu widzenia twórców gier sieciowych, ma on jednak pewne ograniczenia. Jest oparty o protokół TCP, którego niezawodność powoduje opóźnienia w transmisji danych (przesyłanie potwierdzeń, retransmisje itp.). Dodatkowo cały ruch musi być kierowany przez serwer, gdyż bezpośrednia komunikacja pomiędzy przeglądarkami jest niemożliwa. To sprawia, że Web Sockets nie nadają się do bardzo dynamicznych gier.

Rozwiązanie tego problemu nadeszło ze strony interfejsu WebRTC (od Real Time Communication), mającego służyć przede wszystkim do przesyłanie wideo i dźwięku w czasie rzeczywistym pomiędzy przeglądarkami, dzięki czemu możliwe byłoby zbudowanie komunikatora wideo w przeglądarce.

WebRTC umożliwia również przesyłanie dowolnych danych tekstowych i binarnych, a ponieważ działa w oparciu o protokół UDP i łączy bezpośrednio przeglądarki (peer-to-peer), idealnie nadaje się do szybkich gier sieciowych. Oczywiście użycie bezpołączeniowego protokołu UDP powoduje, że aplikacja sama musi zapewnić poprawność działania w przypadku błędów przesyłu.

WebRTC był początkowo tworzonym przez Google, lecz wkrótce prace przejęła grupa robocza w W3C, złożona m.in. z Google, Mozilli i Opery. Niestety Microsoft odmówił wsparcia dla WebRTC i – jak to miało miejsce już wielokrotnie – zaczął tworzyć swój alternatywny standard: CU-RTC-WEB (Customizable, Ubiquitous Real-Time Communication over the Web). Dlatego WebRTC nie jest obsługiwany w Internet Explorer i nie ma żadnych planów jego implementacji. Co więcej, żadna z przeglądarek nie posiada jeszcze pełnej i stabilnej implementacji (stan na początek 2014 roku). Jednakże w niektórych (Chrome, Firefox) API już na tyle dojrzało, że można go z powodzeniem używać.

2.1.6. Web Audio API

Poza grafiką, w grach bardzo ważna jest również oprawa dźwiękowa. O ile odtwarzanie muzyki nie było problemem od czasu wprowadzenia taga <audio> to udźwiękowienie efektów w grze wymagało bardziej zaawansowanego API.

W tym celu do HTML5 zostało wprowadzone Web Audio API. Jest to wysokopoziomowy, zaawansowany interfejs do przetwarzania i odtwarzania dźwięków w JavaScript. Umożliwia on wszystko co potrzebne do udźwiękowienia gry – dźwięk przestrzenny, nakładanie wielu efektów itp.

Aktualnie (2014) Web Audio API jest wspierane przez wszystkie większe przeglądarki oprócz Internet Explorer.

2.1.7. Pozostałe przydatne API

HTML5 zapewnia wiele interfejsów programistycznych przydatnych w wielu aplikacjach i grach w zależności od potrzeb. Wśród bardziej przydatnych należałoby wymienić:

- Gamepad API – obsługa gamepadów, joysticków, kierownic komputerowych itp.
- Mouse Lock API – możliwość ukrycia i zablokowania kursora myszy,
- Fullscreen API
- Web Storage – przechowywanie danych po stronie klienta (np. w celu cache'owania danych gry),
- File API – operowanie na plikach,
- XmlHTTPRequest – asynchroniczne pobieranie plików z serwera (standard de facto od dłuższego czasu),
- Device Orientation i Touch Events – dla urządzeń mobilnych.

2.2. Gry Komputerowe

Gry komputerowe to w dzisiejszych czasach ogromny rynek zapewniający rozrywkę milionom graczy na całym świecie. Historia gier zaczęła się niedługo po wynalezieniu komputerów. Początkowo były to bardzo proste programy, jednak z czasem stawały się coraz bardziej skomplikowane, zarówno pod względem technicznym (grafika, dźwięk), jak i pod względem mechaniki (zasad gry).

Rysunek 2.2: Jedna z najsłynniejszych gier w historii – Super Mario Bros.

Gry opanowały wiele platform sprzętowych. Wcześniej utożsamiane głównie z automatami i konsolami, wkrótce odniosły wielki sukces na komputerach osobistych, a współcześnie na telefonach i tabletach. Co prawda gry działające w przeglądarce pojawiały się od początku istnienia języka JavaScript,

czy platformy Adobe Flash (wcześniej Macromedia Flash), jednak były one utożsamiane z bardzo prostą rozgrywką i ubogą warstwą audiowizualną. Wraz w nadejściem ery HTML5 i technologii z nim związanych (przede wszystkim WebGL), zaistniała możliwość ekspansji bardziej zaawansowanych gier na ten – stosunkowo jeszcze młody – rynek.

Zainteresowani tematyką tworzenia gier znajdą obszerne wprowadzenie w [4].

W następnych podrozdziałach zostanie przedstawiony gatunek gier FPS oraz dwie gry tego typu, szczególnie związane z niniejszym opracowaniem, a także silnik gry, na którym są oparte. Następnie wprowadzone zostaną koncepcje związane z architekturą gier sieciowych (mutliplayer).

2.2.1. Gry FPS

Przedmiotem pracy jest stworzenie gry typu FPS działającej w przeglądarce.

Skrót FPS oznacza First Person Shooter. Jest to strzelanina w której obserwujemy świat oczami bohatera. Gry typu FPS należą do najstarszych i najczęściej występujących gier 3d. Z jednej strony mają one dość prostą mechanikę i zasady, z drugiej natomiast, często są bardzo zaawansowane technologicznie i bywają pionierskie w dziedzinach takich jak realistyczna grafika, czy rozgrywka sieciowa. Dlatego właśnie taki gatunek został wybrany do implementacji w HTML5.

Rysunek 2.3: Współczesna gra FPS – Battlefield 4

2.2.2. Quake III Arena

W roku 1996 ukazała się gra Quake, która odniosła wielki sukces, głównie dzięki rewolucyjnej grafice i dopracowanej rozgrywce sieciowej. Twórcą gry była firma id Software. Rok później powstała kolejna część - Quake II.

Z racji ogromnej popularności trybu dla wielu graczy, kolejna część – Quake III Arena – zupełnie porzuciła tradycyjny tryb fabularny i skupiła się tylko na rozgrywce sieciowej. Została wydana w roku 1999. Gra również odniosła sukces i do dziś pozostaje popularna pośród graczy. Profesjonalne turnieje gier sieciowych (tzw. e-sport) bardzo często w swoim programie mają rozgrywki w Quake III Arena. W roku 2000 ukazał się dodatek do gry pod nazwą Quake III Team Arena.

Rysunek 2.4: Scena z gry Quake III Arena

Siedem lat później, w roku 2007 id Software rozpoczął prace nad wydaniem Quake III Arena i Team Arena na przeglądarki internetowe pod nazwą Quake Live. Kod gry uruchamiany był poprzez specjalnie stworzoną wtyczkę do przeglądarki. W związku z tym rozgrywka była możliwa tylko na wybranych systemach operacyjnych i przeglądarkach, konieczna też była instalacja wtyczki przez gracza.

w międzyczasie, w roku 2005 ukazał się Quake IV, który kontynuował wątki fabularne z Quake II. Gra pomimo dobrych ocen w prasie nie przyjęła się tak dobrze wśród graczy, głównie ze względu na brak znaczących ulepszeń w trybie sieciowym.

Quake III oraz Quake Live jest oparty na silniku gry o nazwie id Tech 3, o którym będzie mowa w podrozdziale 2.2.4.

2.2.3. OpenArena

W roku 2005 główny programista id Software, John Carmack, ogłosił uwolnienie kodu źródłowego silnika id Tech 3 na licencji GNU GPL. Uwalnianie źródeł swoich starszych technologii jest już tradycją w tej firmie.

Wkrótce rozpoczęły się prace nad wieloma grami open source korzystającymi z tego silnika. Jedną z najbardziej dojrzałych jest OpenArena.

OpenArena jest właściwie klonem Quake III Arena. Ponieważ z pierwowzoru został udostępniony tylko kod, fani utworzyli potrzebne zasoby gry (modele, tekstury itp.), również na licencji GPL. Wpro-

wadzono też wiele usprawnień do kodu gry, dodając np. obsługę nowszej wersji OpenGL i shaderów [1]. Tym samym OpenArena stała się pełnoprawną i całkowicie otwartą grą.

Grę OpenArena można pobrać ze strony `http://openarena.ws/`.

2.2.4. id Tech 3

Każda bardziej zaawansowana gra posiada w kodzie źródłowym mniej lub bardziej rozgraniczony rdzeń nazywany silnikiem gry. Odpowiada on za wczytywanie i wyświetlanie świata 3d, odtwarzanie dźwięków, obsługę sieci oraz wiele innych zadań. Często jeden silnik jest wykorzystywany przy tworzeniu wielu gier.

id Software zawsze wykorzystywało swoje autorskie silniki, których źródła były publikowane po utworzeniu nowej wersji. Quake III Arena, a potem OpenArena wykorzystywały id Tech w wersji 3.

Silnik ten był bardzo zaawansowany technologicznie jak na swoje czasy. Do wyświetlania grafiki konieczny był sprzętowy akcelerator graficzny. Pośród ciekawszych możliwości tego silnika należy wymienić:

- wyświetlanie powierzchni opartych na krzywych sklejanych,
- system tzw. skryptów shaderów, dzięki którym graficy mieli dużą kontrolę nad wyglądem każdej powierzchni [2],
- wyświetlanie generowanych wcześniej map oświetlenia, zapewniających bardziej realistyczny wygląd otoczenia,
- przestrzenna mgła,
- odbicia lustrzane,
- zaawansowany system synchronizacji stanu gry przez sieć,
- QVM – wbudowana wirtualna maszyna wykonująca kod gry.

Interesujące omówienie architektury silnika id Tech 3 znajduje się w [12].

2.2.5. Multiplayer

Multiplayer jest to tryb gry, w którym uczestniczy wielu graczy, zazwyczaj za pośrednictwem Internetu. Tryb ten jest często spotykany w grach FPS, ale także w grach wyścigowych, czy strategiach. Wiele gier i graczy skupia się wyłącznie na grze sieciowej. Zdarza się, że w jednej rozgrywce uczestniczy na raz tysiące graczy [3]

[1] Shader - niewielki program działający na karcie graficznej, opisujący właściwości wierzchołków i pikseli.

[2] Nie należy mylić tego systemu nieistniejącymi jeszcze shaderami działającymi na karcie graficznej. Skrypty z id Tech3 były jednak pewną namiastką shaderów w dzisiejszym rozumieniu.

[3] Są to tak zwane gry MMO – Massive Multiplayer Online

Tryb multiplayer wymaga, aby wszyscy gracze widzieli ten sam stan gry. Synchronizacja gry pomiędzy komputerami w sieci jest zagadnieniem nietrywialnym, szczególnie w przypadku gier bardzo dynamicznych i posiadających złożoną mechanikę (a przez to złożony opis stanu gry).

Z punktu widzenia organizacji w sieci, możliwe są dwie architektury:

– Peer-to-peer – nie ma głównego komputera, każdy komputer rozsyła do wszystkich pozostałych informacje o każdej akcji wykonanej przez gracza lokalnego. Architektura ta jest rzadko używana, ze względu na trudność z synchronizacją i problemy z nieuczciwymi graczami.

– Klient - serwer – najczęściej używana. Jeden z komputerów jest serwerem gry. Na nim odbywa się cała symulacja, a wszyscy klienci służą jedynie jako terminale, wysyłając informacje o naciśniętych przez gracza przyciskach i otrzymując migawkę (snapshot) aktualnego stanu gry do wyrenderowania.

Należy zaznaczyć, że serwer niekoniecznie musi być dedykowaną maszyną utrzymywaną przez producenta gry. W przypadku gier w niewielką liczbą uczestników, zazwyczaj jeden z komputerów graczy pełni rolę serwera. Dodatkowo, bardzo często producent zapewnia tzw. lobby server, czyli główny serwer, który służy m. in. do odnajdywania toczących się właśnie rozgrywek i inicjalizacji połączenia. W związku z tym pojęcie serwer może mieć dwa różne znaczenia i wymaga doprecyzowania, dlatego na potrzeby dalszej części pracy wprowadzimy następujące pojęcia:

– serwer główny lub lobby – serwer utrzymywany przez dewelopera, zbierający informacje o toczących się grach i ułatwiający inicjalizację połączenia,

– serwer gry – instancja gry która zarządza konkretną rozgrywką

W przypadku gry przeglądarkowej, serwer gry działa w przeglądarce, co może wprowadzić dodatkowe nieporozumienie, gdyż w aplikacjach webowych przeglądarka jest tożsama z klientem HTTP.

Zagadnienie programowania gry multiplayer jest bardzo obszerne. Jeden z dokładniejszych opisów można znaleźć w [6].

2.3. Poprzednie prace

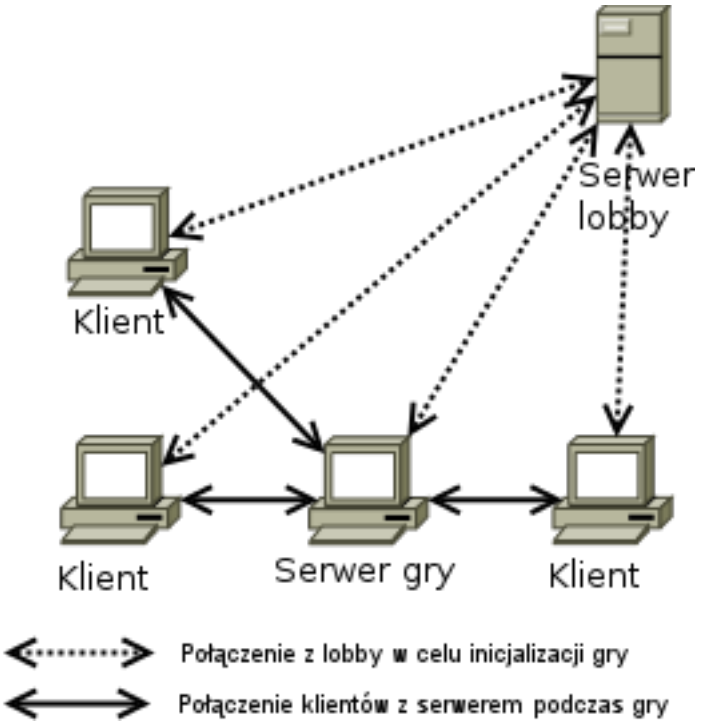
W niniejszym rozdziale zaprezentowane zostaną projekty, mające związek z tematyką pracy.

2.3.1. Quake II w przeglądarce

W roku 2010 dwaj pracownicy Google, Joel Webber i Ray Cromwell udostępnili swój projekt – port Quake II na przeglądarki z wykorzystaniem WebGL. Była to jedna z pierwszych prac, które udowodniły możliwość zaistnienia gier 3d w przeglądarkach.

quake2-gwt-port, bo tak został nazwany projekt, powstał poprzez skompilowanie w GWT [4] kodu Jake2 – portu Quake II do Javy. Projekt ten nie powstał bezpośrednio w JavaScript, przez co nie integruje

[4]GWT – Google Web Toolkit – SDK do tworzenie aplikacji webowych w języku Java, posiadające również kompilator części klienckiej do JavaScript. `http://www.gwtproject.org/`



Rysunek 2.5: Schemat połączeń sieciowych w grze multiplayer typu klient - serwer, jeden z komputerów graczy pełni rolę serwera gry.

się zbyt dobrze ze środowiskiem webowym, a do uruchomienia konieczne było wiele łat i rozwiązań prowizorycznych. Dodatkowo w roku 2010 nie istniało jeszcze Mouse Lock API, przez co sterowanie grą jest bardzo trudne.

Z tych powodów projekt ten należy uznać raczej za demo technologiczne niż grę. Mimo to był to ważny krok milowy dla technologii WebGL.

Strona projektu: `https://code.google.com/p/quake2-gwt-port/`.

2.3.2. Quake III level viewer

W tym samym roku Brandon Jones zaprezentował swoje demo o nazwie q3bsp. Jest to level viewer[5] poziomów z Quake III Arena. W przeciwieństwie do quake2-gwt-port, q3bsp został w całości napisany w JavaScripcie, dzięki czemu mógł użyć wiele udogodnień zapewnianych przez środowisko przeglądarki (np. zapewnione wczytywanie plików graficznych, interfejs użytkownika w HTML), a jego wydajność jest dużo lepsza, pomimo bardziej zaawansowanej grafiki.

Projekt ten również zyskał duży rozgłos w środowisku deweloperów WebGL i poza nim. Można go zobaczyć na stronie `http://media.tojicode.com/q3bsp/`. Część kodu źródłowego q3bsp (udostępnionego na licencji zlib) została wykorzystana w niniejszej pracy.

[5]Level viewer - narzędzie umożliwiające podgląd danego poziomu gry, bez uruchamiania logiki rozgrywki

2.3.3. Banana Bread

Banana Bread jest portem gry Cube2:Sauerbraten (`http://sauerbraten.org/`) na JavaScript zrobionym przez Mozillę. Powstał przez kompilację z C++ kompilatorem Emscripten.

Jest to pełnoprawna i działająca gra, w którą można zagrać na stronie `https://developer.mozilla.org/pl/demos/detail/bananabread`.

3. Projekt

W niniejszym rozdziale zostanie przedstawiony zakres projektu oraz budowa typowej gry, a następnie w odniesieniu do niej architektura projektu.

3.1. Zakres projektu

Przedmiotem niniejszej pracy jest stworzenie gry będącej klonem OpenArena (rozdział 2.2.3). Gra będzie nosić nazwę WebArena w nawiązaniu do tytułu oryginału i technologii w jakiej powstaje.

3.1.1. Założenia

Głównym celem projektu jest zbadanie procesu tworzenia gry w technologii HTML 5. Z tego powodu został położony nacisk na to, aby odtworzyć jak najwięcej elementów typowej gry (które jednak same w sobie mogą występować w bardzo podstawowej formie), rozwiązując problemy wynikające z nowego środowiska.

Założenia:

- Silnik renderujący (renderer) 3d bazujący na WebGL.
- Wykorzystanie wielowątkowości przy użyciu Web Workers.
- Tryb multiplayer z wykorzystaniem specjalnie napisanej, wysokopoziomowej biblioteki umożliwiającej łatwą synchronizację stanu dowolnej gry.
- Warstwa sieciowa ma wykorzystywać Web Sockets do komunikacji z lobby (inicjalizacja połączenia) oraz WebRTC do synchronizacji stanu gry.
- Jak najszybsze wczytywanie potrzebnych zasobów gry, aby gracz mógł natychmiastowo dołączyć do gry.
- Możliwość tworzenia meczy i dołączania do nich na podstawie udostępnianego adresu URL.

Dwa ostatnie punkty mają służyć możliwie najpełniejszemu wykorzystaniu możliwości jakie daje platforma sieciowa. Gracz może natychmiast dołączyć do meczu, w którym uczestniczy jego znajomy na podstawie dostarczonego adresu URL. Po dodaniu integracji z mediami społecznościowymi (nie będzie realizowana w tym projekcie), gra mogłaby zyskać dużą popularność dzięki takiemu modelowi szybkiego dostępu.

3.1.2. Różnice w stosunku do gry OpenArena

Tworzenie gry komputerowej jest bardzo czasochłonnym zadaniem. Wieloosobowe zespoły pracują często przez wiele lat nad ukończeniem produktu. Dlatego, aby zrealizowanie projektu było możliwe należy poczynić pewne uproszczenia. Z drugiej strony, założeniem pracy jest jak najbardziej dokładne przetestowanie technologii HTML5 pod kątem tworzenia gry.

Z tych powodów, ważne jest, aby projekt zawierał wszystkie istotne komponenty znajdujące się standardowo w innych grach, ale by były nieco uproszczone. W porównaniu do OpenArena pojawią się więc następujące różnice:

– tylko jeden poziom,

– jeden typ broni,

– brak apteczek, pancerzy itp.,

– brak menu głównego,

– jeden tryb gry sieciowej (tzw. deathmatch – "każdy na każdego"),

– brak przeciwników sterowanych przez sztuczną inteligencję (rozgrywka tylko z innymi graczami),

– kilka mniej istotnych różnic graficznych (np. brak mgły),

– brak dźwięku.

Jak widać większość braków dotyczy samej rozgrywki, a nie silnika gry i można byłoby je łatwo dodać dysponując większymi zasobami. Największym brakiem jest brak dźwięku i w przypadku rozszerzenia projektu, należałoby się nim zająć w pierwszej kolejności. Dzięki Web Audio API (rozdział 2.1.6) technologia nie jest tu ograniczeniem.

3.1.3. Ograniczenia gry jako aplikacji internetowej

Z punktu widzenia aplikacji internetowej, projekt również musiał zostać ograniczony. Aby nadawała się ona dla szerszego grona użytkowników należałoby dodać:

– możliwość rejestracji i logowania,

– rankingi graczy,

– integracja z mediami społecznościowymi np. w celu udostępniania utworzonych meczy i znajdowania graczy chętnych do rywalizacji,

– konfiguracja swojego awatara w grze,

– konfiguracja tworzonego meczu.

Te elementy również zostały pominięte z powodu ograniczonych zasobów. Wyznaczają one możliwy kierunek przyszłego rozwoju projektu.

3.2. Budowa gry komputerowej

Większość elementów projektu pokrywa się z typową grą, dlatego omówienie projektu należy rozpocząć od pobieżnego omówienia ogólnej budowy gier. Dokładniejszy opis jest poza zakresem pracy ze względu na swoją złożoność. Zainteresowani tematem powinni rozpocząć jego zgłębianie od [4].

3.2.1. Pętla główna

W sercem każdej gry, w szczególności gry sieciowej jest pętla główna która wykonuje się wiele razy na sekundę, wykonując aktualizację stanu gry i renderowanie sceny gry. W uproszczeniu pętla taka wygląda następująco:

1. Sprawdzenie stanu urządzeń wejściowych (mysz, klawiatura, gamepad itp.).
2. Zaktualizowanie stanu gry na podstawie stanu poprzedniego i nowego wejścia od gracza.
3. Renderowanie uaktualnionej sceny gry.

W zależności od konkretnej gry poszczególne etapy mogą się różnić. Na przykład, w zależności od tego, czy gra ma zaimplementowaną symulację fizyczną lub sztuczną inteligencję, elementy te muszą zostać obsłużone w odpowiedniej kolejności podczas aktualizacji stanu gry.

Dodatkowo, multiplayer bardzo mocno modyfikuje standardowy model. W najprostszej wersji gry sieciowej w architekturze klient serwer należy rozróżnić dwie osobne pętle główne dla klienta i serwera. Strona klienta wygląda następująco:

1. Sprawdzenie stanu urządzeń wejściowych.
2. Wysłanie stanu wejścia do serwera.
3. Pobranie nowego stanu gry (snapshotu) z serwera.
4. Renderowanie uaktualnionej sceny.

Natomiast strona serwera:

1. Pobranie aktualizacji wejścia od klientów
2. Zaktualizowanie stanu gry na podstawie stanu poprzedniego i nowych wejść od klientów.
3. Wysłanie aktualnego stanu gry do wszystkich klientów.

Należy zauważyć, że tylko serwer przelicza stan gry, dzięki czemu jest on spójny na wszystkich komputerach. Komputery klienckie występują tylko jako terminale – pobierają stan urządzeń wejściowych i renderują rezultat otrzymany od serwera.

Trzeba jednak zaznaczyć, że jest to wersja naiwna, która nie sprawdzi się w przypadku bardziej dynamicznych gier. Od momentu wciśnięcia przycisku przez gracza do pojawienia się na ekranie odpowiednich zmian, upłynęło by zbyt dużo czasu.

Dlatego część aktualizacji stanu gry związaną bezpośrednio z ruchem gracza na ekranie, wykonuje się również po stronie klienta, a następnie – podczas pobierania nowego stanu z serwera – uzgadnia wyniki pomiędzy wersją obliczoną po stronie serwera i klienta. Należy to zrobić w taki sposób, aby gracz nie dostrzegł przeskoków w ruchu obiektów na ekranie (albo dostrzegł niewielkie), a jednocześnie zachować pierwszeństwo serwera w decydowaniu o stanie gry.

Dokładniejsze omówienie tematu projektowania i implementacji gry multiplayer znajduje się w [6].

3.2.2. Komponenty silnika gry

Kod pętli głównej potrzebuje do działania wiele komponentów należących do silnika gry. Są to przede wszystkim:

- Renderer – umożliwia renderowanie obiektów stworzonych w programach do grafiki 3d, za pośrednictwem API takiego jak WebGL.
- Obsługa wejścia – zapewnia wysokopoziomowe API do obsługi urządzeń wejściowych.
- Wczytywanie zasobów – przed rozpoczęciem gry, a często również w jej trakcie, konieczne jest wczytanie, rozpakowanie i odpowiednie przetworzenie plików potrzebnych w grze. W przypadku gry sieciowej konieczne jest uprzednie ściągnięcie plików z serwera HTTP.
- Biblioteka sieciowa – umożliwia nawiązanie początkowego połączenia, a także zapewnia możliwość synchronizacji stanu gry.

Powyższe komponenty występują w grze WebArena. W innych grach może ich występować więcej w zależności od potrzeb (np. podsystem dźwiękowy, biblioteka do symulacji fizycznej, czy narzędzia do tworzenia sztucznej inteligencji).

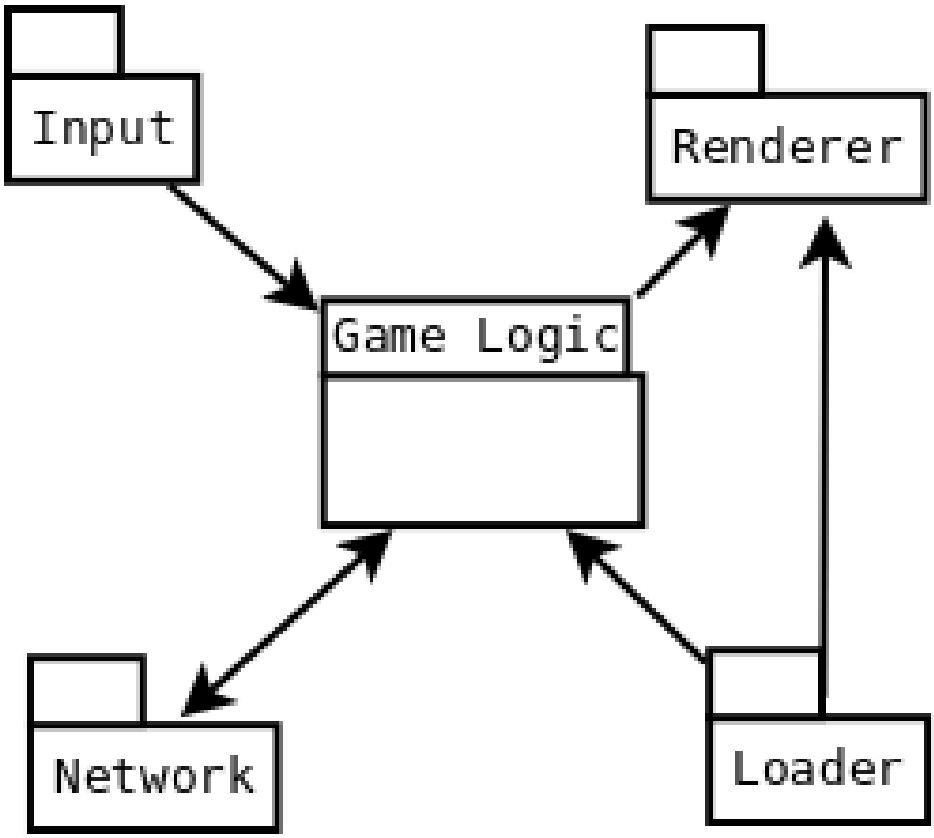
Dokładniejsze omówienie implementacji poszczególnych komponentów w grze WebArena znajduje się w rozdziale 4.

3.2.3. Współbieżność

Jednym z prostszych sposobów na wykorzystanie współbieżności w grach jest rozdzielenie pracy różnych podsystemów pomiędzy wątkami. Ten sposób został wybrany do WebArena, między innymi ze względu na swoją prostotę, ale również dlatego, że pozwala on zminimalizować punkty synchronizacji i ilość przesyłanych danych pomiędzy wątkami, co jest bardzo istotne, biorąc pod uwagę sposób działania Web Workers – brak współdzielonej pamięci (rozdział 2.1.3). Wadą takiego sposobu jest niska skalowalność – jeżeli podsystemy gry zostały podzielone na dwa wątki, użytkownik posiadający czterordzeniowy procesor nie zyska dodatkowego przyśpieszenia. Mimo wszystko, nawet rozdzielenie pracy pomiędzy dwa rdzenie powinno dać znaczące przyśpieszenie.

W przypadku WebArena podział wygląda następująco:

- wątek główny – renderer, obsługa wejścia, pomniejsze podsystemy,
- Web Worker – logika gry, synchronizacja sieciowa.



Rysunek 3.1: Komponenty gry wraz z przepływem danych pomiędzy nimi.

Z powodu ograniczeń Web Workers, większość podsystemów bezpośrednio odwołujących się do zewnętrznych API (np. WebGL, HTML DOM itd.), musi pracować w wątku głównym. Dodatkowo podczas wczytywania zasobów uruchamiane są dodatkowe wątki do czasochłonnego parsowania plików, które mogłoby pogorszyć responsywność aplikacji.

Schemat 3.2 przedstawia rozmieszczenie komponentów gry pomiędzy Web Workery. Jak widać, warstwa sieciowa została rozdzielona na dwie części. Ponieważ API WebRTC jest dostępne tylko dla głównego wątku, część odpowiedzialna za nawiązanie i utrzymywanie połączenia sieciowego działa na nim. Część odpowiedzialna za synchronizację, współdzieli pamięć z logiką gry, dlatego pracuje na drugim wątku. Komponenty pomiędzy workerami komunikują się tylko za pomocą przesyłanych wiadomości.

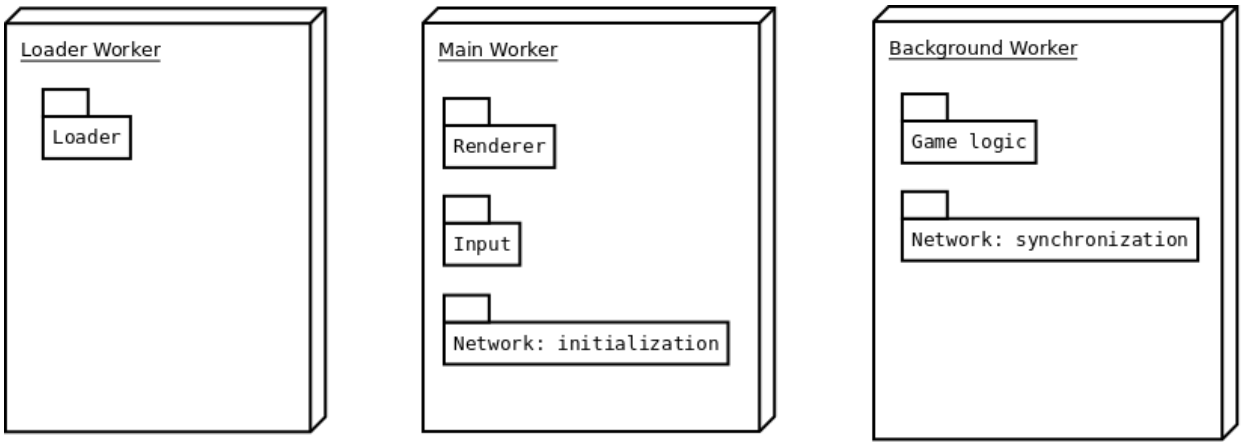
Dyskusję na temat współbieżności w grach i różnych podejść do niej można znaleźć między innymi w [4], [8] i [1].

Więcej szczegółów na temat implementacji wielowątkowości w grze WebArena znajduje się w rozdziale 4.3.

3.3. Architektura sieciowa

Jako, że WebArena jest systemem rozproszonym, należy również omówić architekturę sieciową gry jako całego systemu – role urządzeń sieciowych biorących udział w jego budowaniu.

Serwer lobby pełni jednocześnie dwie funkcje:



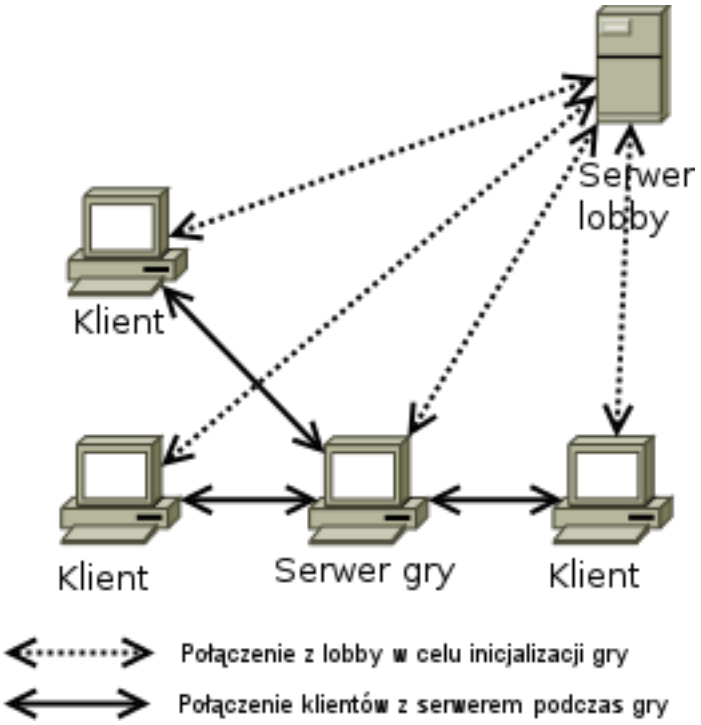
Rysunek 3.2: Podział komponentów gry pomiędzy wątki.

– jest serwerem HTTP hostującym kod źródłowy gry w JavaScript i pliki z zasobami gry,

– jest serwerem Web Sockets, przekazującym pakiety pomiędzy klientami i serwerem w początkowej fazie nawiązywania połączenia.

Kod serwera jest bardzo prosty i właściwie ogranicza się do statycznego serwowania plików oraz pasywnego przekazywania pakietów pomiędzy klientami i serwerem, w celu nawiązania połączenia WebRTC. Z tego powodu serwer sieciowy nie będzie tutaj szerzej omawiany.

Zgodnie z treścią rozdziału 2.2.5, klient oraz serwer gry są aplikacją działającą w przeglądarce internetowej.



Rysunek 3.3: Architektura sieciowa gry.

4. Implementacja

W niniejszym rozdziale zostanie bardziej szczegółowo opisana realizacja komponentów gry opisanych w części 3.

4.1. Wyjaśnienie dotyczące zamieszczonych diagramów UML

W niniejszej części pracy będzie wykorzystywany język UML do przedstawienia implementacji poszczególnych komponentów. Język ten pomimo swej użyteczności nie jest jednak idealny do przedstawiania kodu programu stworzonego w JavaScript, dlatego trzeba na początku wyjaśnić kilka nieścisłości.

UML operuje na pojęciach znanych z języków obiektowych, wykorzystujących klasy. Są to pojęcia takie jak klasa, interfejs, metoda, dziedziczenie itp. Jednakże w JavaScript obiektowość jest realizowana przez prototypy[1], a więc terminy te nie występują w samym języku. Język ten jednak jest na tyle elastyczny, że możliwa jest realizacja konstrukcji odpowiadających im. Dodatkowo, często stosuje się znaczniki JSDoc[2], aby jawnie wskazać takie konstrukcje.

W dalszej części opisu, pojęcia te będą stosowane zgodnie z terminologią UML.

4.2. Użyte narzędzia

4.2.1. Google Closure

4.3. Wielowątkowość

Realizacja wielowątkowości w Web Workers wymaga, aby poszczególne wątki komunikowały się jedynie za pomocą wiadomości mogących zawierać jedynie niektóre typy danych (zakazane jest na przykład przesyłanie obiektów HTML DOM). Współdzielenie kodu oraz pamięci jest niemożliwe, nie można więc wywołać funkcji "pomiędzy"workerami.

Te ograniczenia sprawiają, że należy z góry zaplanować jak zostanie podzielony kod pomiędzy wątki. Jest to jednak rozwiązanie mało elastyczne. Trudno przełączać się pomiędzy wersją jedno i wielowątkową w celu porównania wydajności i łatwiejszego wyszukiwania błędów. Dodatkowo wysyłanie komunikatów jest mniej wygodne niż proste wywołanie funkcji.

[1] Więcej na ten temat paradygmatu programowania przez prototypowanie w [3].

[2] JSDoc – język używany do dokumentowania kodu w JavaScript podobny do Javadoc, znanego z Javy.

W WebArena został więc zaimplementowana wysokopoziomowa biblioteka do obsługi Web Workers, wykorzystująca elastyczność języka JavaScript.

Oto jej cechy:

4.3.1. Zdalne wywoływanie metod

Po zarejestrowaniu obiektu w jednym workerze, biblioteka umożliwia utworzenie obiektu proxy w drugim workerze na podstawie podanego interfejsu [3]. Dzięki temu komunikacja pomiędzy workerami jest ukryta pod postacią zwykłego wywołania metody. Dodatkowo obsługiwane jest przekazywanie funkcji jako tzw. callback, co na poziomie Web Workers jest niemożliwe. Jest to system podobny do zdalnego wywoływania metod (np. RMI w języku Java), jednak prostszy w użyciu, dzięki dynamicznemu charakterowi języka JavaScript.

Zalety takiego systemu:

– prostota użycia – wystarczy zadeklarować interfejs dla obiektu, z proxy zostanie utworzone automatycznie,

– możliwość przekazywania funkcji jako argumentu,

– łatwe przełączanie pomiędzy trybem jedno i wielowątkowym (wystarczy zamiast proxy podstawić prawdziwy obiekt.

4.3.2. Zdarzenia

Biblioteka może być użyta również do globalnej (międzywątkowej) obsługi zdarzeń.

Często jeden z komponentów jest zainteresowany zdarzeniem zachodzącym w innym komponencie. Jeśli jednak pracują one w innych workerach, przekazanie funkcji jako callback jest niemożliwe. Dlatego trzeba użyć warstwy pośredniczącej. W przypadku WebArena warstwa ta umożliwia zarejestrowanie funkcji mającej reagować na zdarzenia oraz powiadamianie o występujących zdarzeniach pomiędzy wątkami.

4.3.3. Pule wątków

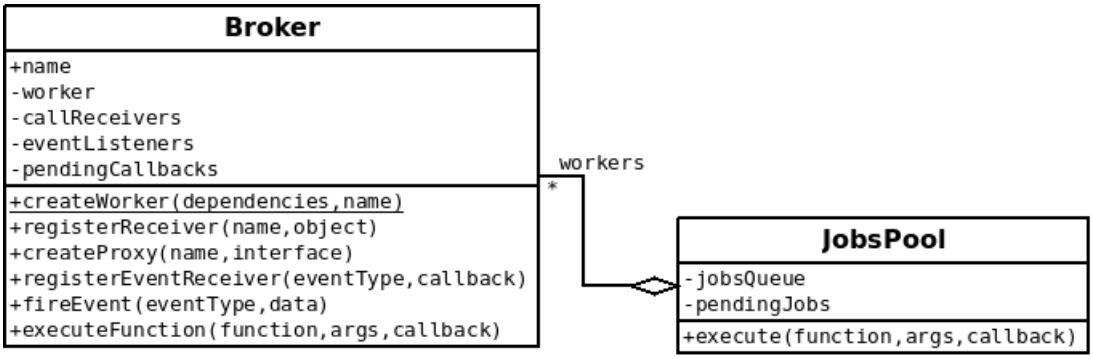
Pula wątków (Thread pool) jest to grupa wątków do których można szybko oddelegować wykonanie małej części logiki bez konieczności tworzenia nowego wątku.

Tworzenie wątku może potrwać na tyle długo, aby zniweczyć wszelkie korzyści wynikające z oddelegowania części pracy na osobny rdzeń procesora. Dodatkowo API Web Workers z reguły wymusza, aby pliki z kodem źródłowym były z góry przydzielone do konkretnego wątku. Proste przekazanie funkcji do wykonania w innym workerze nie zadziała.

[3] JavaScript jest językiem bezklasowym, więc nie ma tam pojęcia tradycyjnego interfejsu (jak np. w języku Java, czy C#). W związku z tym przyjęło się deklarować interfejs po prostu jako obiekt z pustymi metodami.

Dlatego kolejnym zadaniem wykonywanym przez omawianą bibliotekę jest obejście tego ograniczenia, w celu umożliwienia oddelegowania wykonania danej funkcji do puli wątków. Dodatkowo zadania takie są automatycznie kolejkowane, a po ich wykonaniu wyniki zwracane do wywołującego wątku.

4.3.4. Implementacja biblioteki



Rysunek 4.1: Uproszczony diagram UML biblioteki

Diagram 4.1 przedstawia w uproszczeniu implementację omawianej biblioteki. Składa się ona z dwóch głównych klas znajdujących się na diagramie oraz kilku mniej istotnych, które zostały pominięte dla zachowania czytelności. Klasa Broker jest klasą najważniejszą i odpowiada za opakowanie Web Workera w przyjazny interfejs omówiony w poprzednich podrozdziałach. Klasa JobsPool realizuje ideę puli wątków.

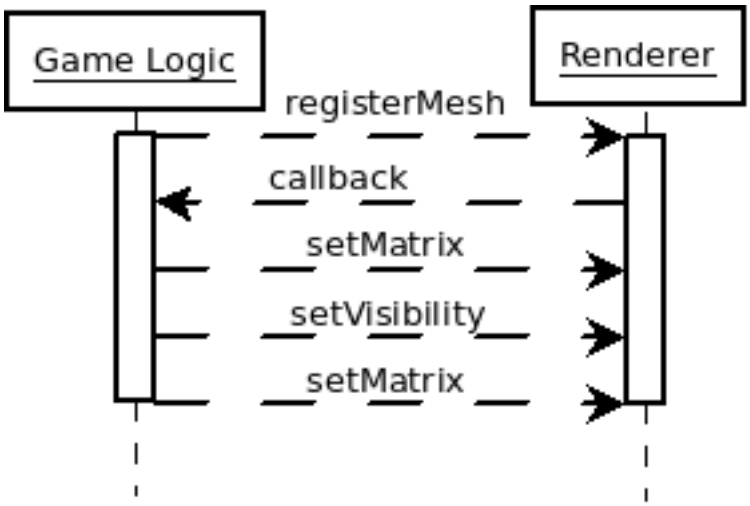
4.4. Renderer

Renderer jest częścią silnika gry odpowiedzialną za renderowanie obiektów ze sceny gry w 3d. Ukrywa on niskopoziomowe wywołania API graficznego (w tym przypadku WebGL) przed programistą gry.

Ponieważ WebGL jest dostępny tylko dla głównego workera, renderer ten nie może pracować na innych wątkach. W celu osiągnięcia maksymalnej wydajności, komunikacja pomiędzy logiką gry, działającą w innym workerze, a rendererem, została ograniczona do minimum. W tym samym celu, została ona zaprojektowana całkowicie asynchronicznie.

Jak widać na schemacie 4.2, gra rejestruje na początku obiekt, który ma być wyświetlony. Gdy rejestracja zostanie zakończona, renderer wysyła wiadomość zwrotną do gry. Po zarejestrowaniu obiektu, renderer wyświetla go zgodnie z zapamiętanym stanem (macierz transformacji, widoczność itd.), dopóki ten nie zostanie zmieniony.

Następnie komunikacja przepływa już tylko w jedną stronę. Gra wysyła komunikaty w celu zmiany stanu obiektu w scenie 3d, bez oczekiwania na odpowiedź. Dzięki temu wątek gry wykonuje się dalej podczas przekazywania i przetwarzania wiadomości. W celu dalszego zmniejszenia narzutu wysyłania komend, są one kolejkowane po stronie gry i wysyłane większymi partiami w jednej wiadomości.

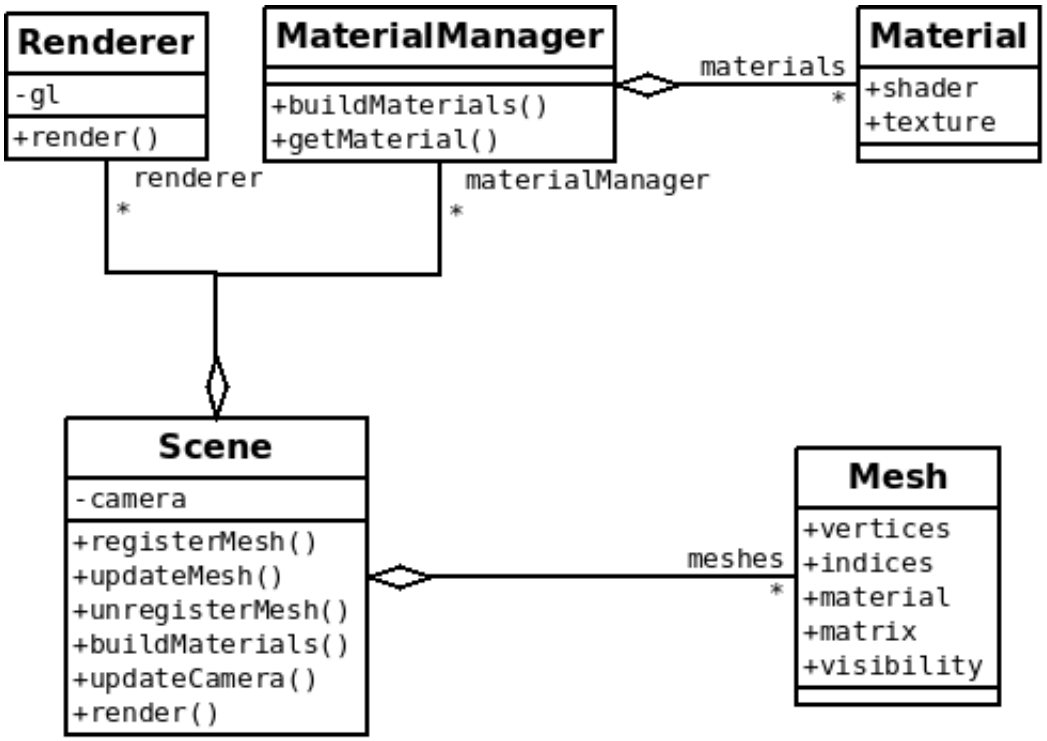


Rysunek 4.2: Asynchroniczna komunikacja gry z rendererem

Cały komunikacja jest opakowana w mechanizm zdalnego wywołania metod (rozdział 4.3.1), dzięki czemu nie trzeba ręcznie tworzyć i wysyłać wiadomości. Dodatkowo, można łatwo przełączać grę w tryb jednowątkowy w celu łatwiejszego wykrywania błędów i porównania wydajności.

4.4.1. Struktura biblioteki renderera

Diagram 4.3 prezentuje uproszczoną strukturę klas związanych z renderowaniem gry.



Rysunek 4.3: Uproszczony diagram klas związanych z rendererem

W następnych podrozdziałach znajduje się omówienie najważniejszych elementów biblioteki.

Renderer

Klasa o nazwie takiej samej jak nazwa całej biblioteki, odpowiada za bezpośrednią, niskopoziomową komunikację z WebGL. Pośredniczy ona pomiędzy wysokopoziomową klasą sceny, operującej na pojęciach obiektów, ich materiałów i pozycji, a niskopoziomowym API do rysowania trójkątów za pomocą shaderów. Klasa renderera potrzebuje stanu obiektu, który otrzymuje od sceny oraz materiał[4], przechowywany w menedżerze materiałów.

Scena

Klasą, która zarządza całym modułem renderera jest scena. Przechowuje ona stan wszystkich zarejestrowanych przez grę obiektów graficznych oraz bezpośrednio odbiera i wykonuje komendy zmiany stanu obiektu wysłane przez grę.

Po wywołaniu funkcji renderującej, scena przekazuje do niskopoziomowej klasy renderera stan wszystkich obiektów razem z potrzebnymi

Menedżer materiałów

Do zadań menedżera materiałów należy tworzenie, przechowywanie i zarządzanie materiałami graficznymi. Podczas inicjalizacji gry, menedżer tworzy potrzebne materiały. Podczas ich tworzenia, konieczne jest między innymi wysłanie tekstur do pamięci karty graficznej za pośrednictwem WebGL, a także skompilowanie używanych shaderów. Klasa ta dba również o to, aby materiały nie dublowały się, zajmując niepotrzebnie pamięć.

4.5. Zasoby gry

WebArena wykorzystuje wiele rodzajów zasobów potrzebnych do wyświetlania grafiki i sterowania logiką gry. Są to następujące pliki:

– pliki bsp z opisami poziomów gry,

– pliki md3 z modelami 3d postaci, broni i innych przedmiotów,

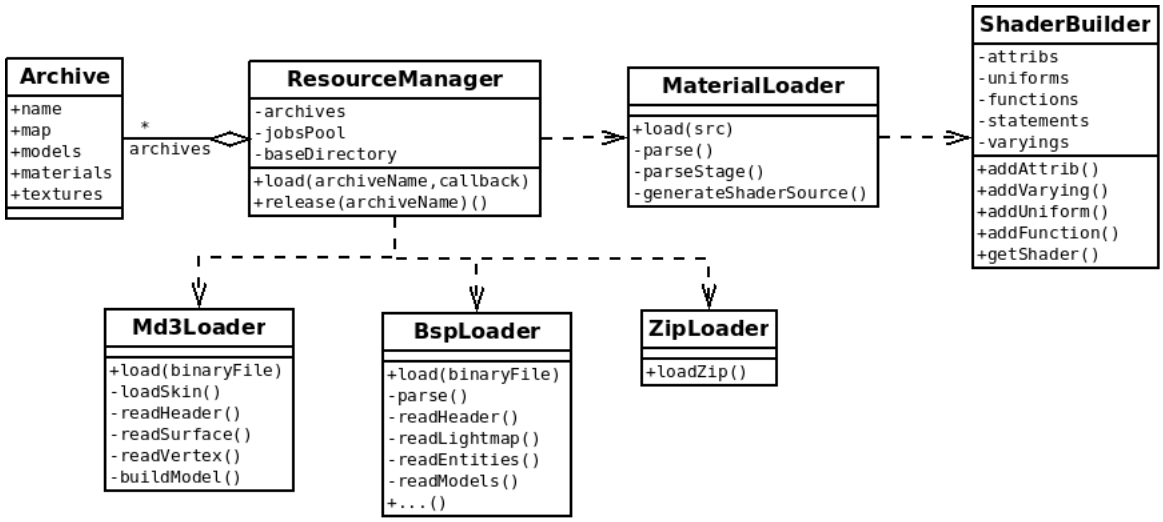
– opisy materiałów graficznych,

– tekstury,

– różnorakie pliki konfiguracyjne.

W związku z tym, że WebArena wykorzystuje zasoby gry OpenArena, wszystkie pliki są zgodne z formatem używanym przez Quake III Arena. W dalszych podrozdziałach nastąpi omówienie kodu odpowiedzialnego za wczytywanie zasobów wraz z pobieżnym omówieniem formatów plików.

[4]Materiał graficzny – zbiór właściwości opisujących wygląd obiektu, takich jak kolor, tekstura, shader wykorzystywany do renderowania itp.



Rysunek 4.4: Uproszczony diagram klas związanych z zarządzaniem zasobami.

4.5.1. Pliki BSP

Pliki BSP służą do opisu geometrii i wyglądu poziomu. Format ten wywodzi się z Quake III. Nazwa BSP oznacza Binary Space Partitioning i jest to również nazwa struktury danych (drzewo BSP), w którym przechowywana jest geometria całego poziomu. Drzewo BSP przyśpiesza wiele operacji wykonywanych w trakcie gry, a przede wszystkim wykrywanie kolizji gracza lub pocisku z przedmiotami w świecie gry.

Kolejną ważną cechą plików BSP jest przechowywanie tzw. lightmapy, która służy do przedstawiania realistycznego oświetlenia poziomu.

Plik BSP dzieli się na kilkanaście sekcji, przechowujących między innymi informacje o:

– budowie drzewa BSP,

– wierzchołkach,

– materiałach graficznych,

– lightmapie,

– efektach (takich jak np. mgła).

Dokładną specyfikację pliku BSP można znaleźć na na stronie [11]. W WebArena do parsowania pliku został częściowo wykorzystany kod z dema Brandona Jonesa opisanego w rozdziale 2.3.2.

Ponieważ parsowanie pliku jest czasochłonne, jego wykonanie odbywa się na osobnym wątku za pośrednictwem biblioteki opisanej w rozdziale 4.3.

4.5.2. Pliki MD3

MD3 jest formatem plików modeli graficznych używanym przez Quake III Arena. W plikach tych przechowywane są wszystkie ruchome modele, takie jak postacie, bronie itp.

Ich ważną cechą jest możliwość przechowywania animacji (w przeciwieństwie do plików BSP). Jest to animacja wierzchołkowa, w której dla każdej klatki animacji, przechowywane są pozycje wszystkich wierzchołków. Sposób ten powoduje duże zużycie pamięci, jednak jego zaletą jest prostota.

Kod do wczytywania plików MD3 został napisany specjalnie dla WebArena w oparciu o specyfikację dostępną na stronie [10] i tak jak w przypadku BSP, wykonywany jest na osobnym wątku.

4.5.3. Pliki materiałów graficznych i tekstury

Quake III Arena powstał w czasach gdy jeszcze niemożliwe było tworzenie shaderów. Na karcie graficznej mógł działać tylko predefiniowany przez producenta program do renderowania, można było sterować jedynie jego parametrami. Był to tzw. fixed pipeline.

Aby obejść ograniczenia tego rozwiązania i umożliwić grafikom elastyczne definiowanie wyglądu powierzchni, id Tech 3 wprowadził pliki tekstowe z opisem materiałów graficznych, częściowo pełniących również rolę dzisiejszych shaderów. Pliki te również były nazywane shaderami, co może wprowadzać pewną niejasność, dlatego w tej pracy konsekwentnie będzie używane pojęcie materiału.

Listing 4.1: Przykładowy opis materiału

```
models/powerups/ammo/machammo
{
        {
                map models/powerups/ammo/ammobox.tga
                rgbGen lightingDiffuse
        }
        {
                map models/powerups/ammo/ammolights.tga
                blendfunc blend
                rgbGen const ( 1 1 0 )
                alphaGen wave sawtooth 0 1 0 1
        }
}
```

Dokładną specyfikację plików materiałów można znaleźć w [9].

Ponieważ przestarzały fixed pipeline jest całkowicie usunięty z OpenGL ES i WebGL, dlatego konieczne było przetłumaczenie materiałów na kod shadera. Jest to realizowane podczas wczytywania materiału. Większość kodu wczytującego i tłumaczącego materiały jest pobrane z pracy Brandona Jonesa (rozdział 2.3.2).

4.5.4. Menedżer zasobów

Menedżer zasobów zarządza wczytywaniem i przechowaniem zasobów z pomocą wyżej wymienionych parserów. Wszystkie pliki wczytywane są asynchronicznie, za pomocą puli wątków opisanej w rozdziale 4.3.3. Dzięki temu interfejs użytkownika nie blokuje się nawet podczas wczytywania bardzo

dużych plików. Menedżer dba również, aby żaden plik nie został wczytany dwukrotnie, co oznaczałoby marnowanie pamięci.

W przypadku gry sieciowej nie można pominąć kwestii przesyłania plików przez internet. Wszystkie zasoby muszą być pobrane z serwera przed rozpoczęciem rozgrywki. Konieczne jest jednak maksymalne skrócenie czasu oczekiwania gracza.

Ściąganie każdego pliku osobno jest bardzo nieefektywne, z drugiej jednak strony pobieranie wszystkich zasobów w jednym dużym archiwum trwało by zbyt długo. Dlatego trzeba podzielić pliki na kilka archiwów, tak aby gracz pobierał jedynie te potrzebne do zagrania w dany poziom.

W tym celu został napisany skrypt działający po stronie serwera, który analizuje plik BSP lub MD3 i pakuje go do jednego archiwum zip razem z wszystkimi plikami, do których się odwołuje (materiały, tekstury, modele). Dzięki temu gra pobiera jedynie kilka plików zip, które są po stronie przeglądarki rozpakowywane.

Komunikacja odbywa się za pomocą technologii AJAX[5].

4.6. Synchronizacja sieciowa

Jedną z najciekawszych części projektu jest biblioteka służąca do synchronizacji stanu gry pomiędzy komputerami w sieci. Biblioteka napisana jest w taki sposób, aby łatwo można jej było użyć w innych projektach.

Ręczne pisanie kodu do synchronizacji danych jest bardzo żmudnym zadaniem. Trzeba pamiętać o wielu niskopoziomowych problemach, takich jak zgubione pakiety (przez zawodny protokół UDP), czy też zbyt duży ich rozmiar. Dlatego istotne jest wprowadzenie mechanizmu, który zautomatyzuje synchronizację. W idealnym rozwiązaniu, programista gry wskazuje jedynie które pola w poszczególnych obiektach powinny być przesyłane, dając również wskazówki np. co do wymaganej dokładności, czy częstotliwości synchronizacji. Taki sposób programowania jest zgodny z paradygmatem programowania deklaratywnego, w którym programista określa jaki cel chce osiągnąć, a nie musi podawać sposobu na jego osiągnięcie.

Biblioteka do synchronizacji powinna więc spełniać następujące założenia:

– prostota użycia,

– zwolnienie programisty gry z konieczności pamiętania o niskopoziomowych problemach komunikacji sieciowej,

– możliwość rekurencyjnego synchronizowania całych drzew obiektów,

– kompresja różnicowa (przesyłanie jedynie zmian stanu) dla zmniejszenia rozmiarów danych,

– bezproblemowe działanie w warunkach często traconych pakietów.

[5] AJAX (Asynchronous JavaScript and XML) – technologia służąca do wykonywania asynchronicznych zapytań HTML z języka JavaScript.

4.6.1. Sposób użycia biblioteki

Na listingu 4.2 przedstawiono kod synchronizujący jeden z obiektów gry – jest to obiekt reprezentujący postać gracza. Pozostałe funkcje tego obiektu nie są teraz istotne.

Listing 4.2: Kod synchronizujący jeden z obiektów gry

```
/**
 * @public
 * @param {network.ISynchronizer} sync
 */
game.Player.prototype.synchronize = function (sync) {
    // poniższe trzy obiekty zostaną zsynchronizowane rekurencyjnie
    this.head = sync.synchronize(this.head, Type.OBJECT);
    this.torso = sync.synchronize(this.torso, Type.OBJECT);
    this.legs = sync.synchronize(this.legs, Type.OBJECT);

    // INT8 określa typ i wymaganą dokładność (8 bitów)
    this.legsState = sync.synchronize(this.legsState, Type.INT8);
    this.torsoState = sync.synchronize(this.torsoState, Type.INT8);

    this.lastYaw = sync.synchronize(this.lastYaw, Type.FLOAT32);
    this.legsAngle = sync.synchronize(this.legsAngle, Type.FLOAT32);
};
```

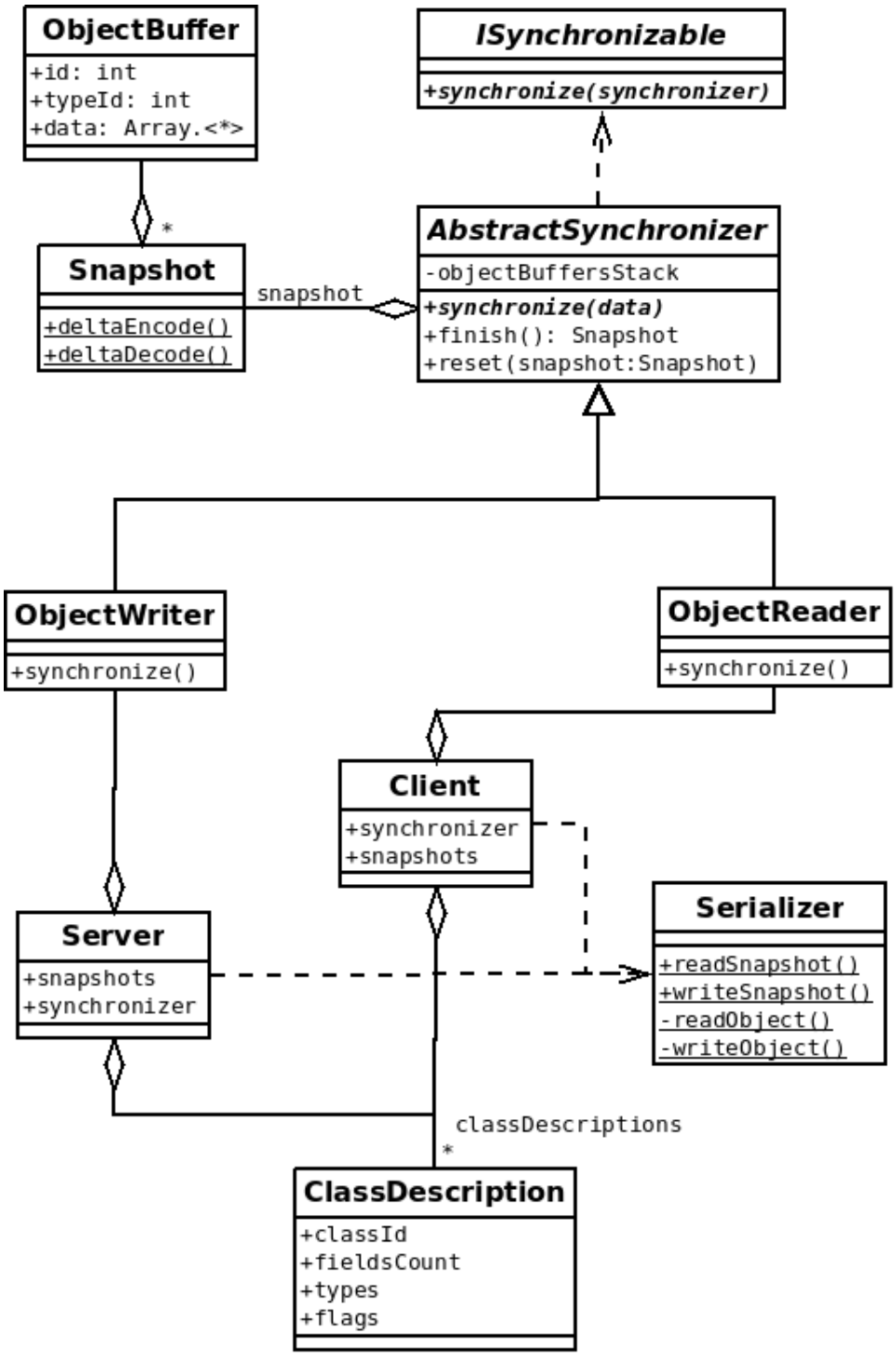
Każdy obiekt, który ma być synchronizowany musi definiować funkcję *synchronize*, która jako argument przyjmuje obiekt synchronizatora (przedstawiony w następnym podrozdziale). Synchronizacja jest przemyślana w taki sposób, aby nie było konieczne pisanie osobnych funkcji odczytujących i zapisujących (choć jest to możliwe w razie konieczności). Dzięki temu unika się błędów związanych z różną kolejnością zapisu i odczytu składowych lub pominięcia jednej z nich.

Synchronizator pamięta czy pracuje w trybie odczytu i zapisu. Jeśli jest to tryb odczytu, jego funkcja *synchronize* zwraca dokładnie taką wartość jaką otrzymała i obiekt pozostaje niezmieniony.

Jak widać, możliwa jest synchronizacja nie tylko typów prostych, ale również obiektów zdefiniowanych przez użytkownika. Dzięki temu można jednym wywołaniem zapisać i odczytać całą scenę gry.

4.7. Omówienie struktury biblioteki

Rysunek 4.5 przedstawia diagram klas biblioteki. Najważniejszą klasą jest *AbstractSynchronizer*, z której dziedziczą odpowiednio *ObjectReader* i *ObjectWriter*. *ObjectReader* jest przekazywany odczytywanym obiektom do funkcji *synchronize*, zaś *ObjecWriter*, jest używany podczas zapisywania obiektu.



Rysunek 4.5: Diagram klas biblioteki do synchronizacji sieciowej.

Każdy obiekt który ma być synchronizowany, musi implementować interfejs *ISynchronizable*[6].

Całym procesem synchronizacji sterują klasy *Client* lub *Server*, w zależności od tego, czy gra działa jako klient czy serwer. Dodatkowe klasy zajmują się przechowywaniem całego stanu gry, kompresją

[6]Oczywiście, z racji tego, że JavaScript jest językiem dynamicznym i nie ma w nim pojęcia interfejsu, mowa tu tylko o implementacji interfejsu w sensie koncepcyjnym. Z punktu widzenia języka, wystarczy jeśli dany obiekt posiada odpowiednie metody.

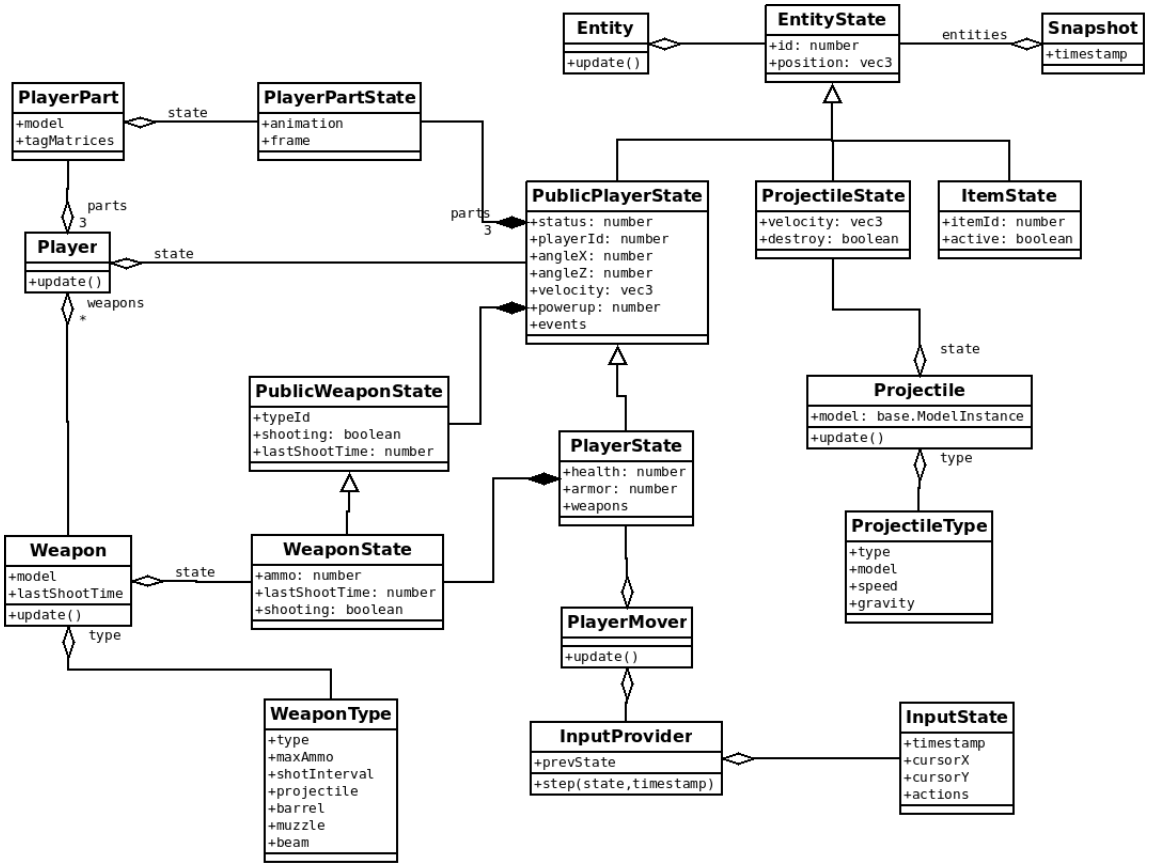
różnicową oraz serializacją stanu do postaci binarnej.

4.8. Logika gry

Kod sterujący logiką korzysta z wszystkich wyżej wymienionych komponentów. Działa równolegle do kodu renderera, na osobnym wątku. Ma swoją pętlę główną, podczas której następuje synchronizacja sieciowa, aktualizacja stanu gry, oraz przesłanie aktualizacji stanu do renderera. Jej implementacja jest zgodna z opisem z rozdziału 3.2.1, z jedną tylko istotną różnicą – logika gry nie uruchamia renderowania, a jedynie przesyła do wątku renderera (który cały czas renderuje świat gry) aktualizację stanu.

Tradycyjnie do implementacji logiki gry często używa się języków skryptowych, które są bardziej przystępne i elastyczne od języków natywnych oraz umożliwiają szybką aktualizację kodu w czasie działania gry, co znacząco wpływa na szybkość jej tworzenia. Takie rozwiązanie ma jednak wadę w postaci konieczności osadzenia w grze wirtualnej maszyny i udostępnienia dla niej odpowiedniego API. W przypadku JavaScript, cała gra jest napisana w języku skryptowym, nie ma więc konieczności dodawania drugiego języka i tworzenia warstwy pośredniczącej.

4.8.1. Omówienie struktury



Rysunek 4.6: Diagram klas przedstawiający logikę gry

Diagram 4.6 przedstawia strukturę klas składających się na stan gry. Wszystkie obiekty znajdujące się w świecie gry wywodzą się z klasy *EntityState*. Do obiektów tych należą:

– gracze (*PlayerState*),

– pociski, takie jak rakiety (*ProjectileState*),

– przedmioty, które gracze mogą zbierać, takie jak bronie, amunicja, apteczki (*ItemState*).

Postać gracza jest podzielona na trzy części (*PlayerPartState*) – głowę, tułów i nogi, z których każda animuje się oddzielnie. Dzięki temu możliwe jest jednoczesne bieganie w dowolnym kierunku i strzelanie. Nie wszystkie możliwości, które daje przedstawiona struktura zostały wykorzystane w aktualnej wersji gry. Na razie nie ma jeszcze przedmiotów do podnoszenia oraz rakiet, jednak przy istniejącym kodzie ich dodanie jest już łatwym zadaniem.

4.9. Opis z punktu widzenia użytkownika

Gotową grę można zobaczyć na stronie `http://game-webarena.rhcloud.com/`.

Po wejściu na stronę, automatycznie uruchamiana jest gra w trybie serwera. Generowany jest też link, który można przesłać znajomemu, chcącemu dołączyć do gry. Po wejściu w podany URL, zostanie on podłączony do gry jako klient.

Sterowanie jest bardzo podobne do innych gier typu FPS:

– mysz – rozglądanie się,

– lewy przycisk myszy – strzał,

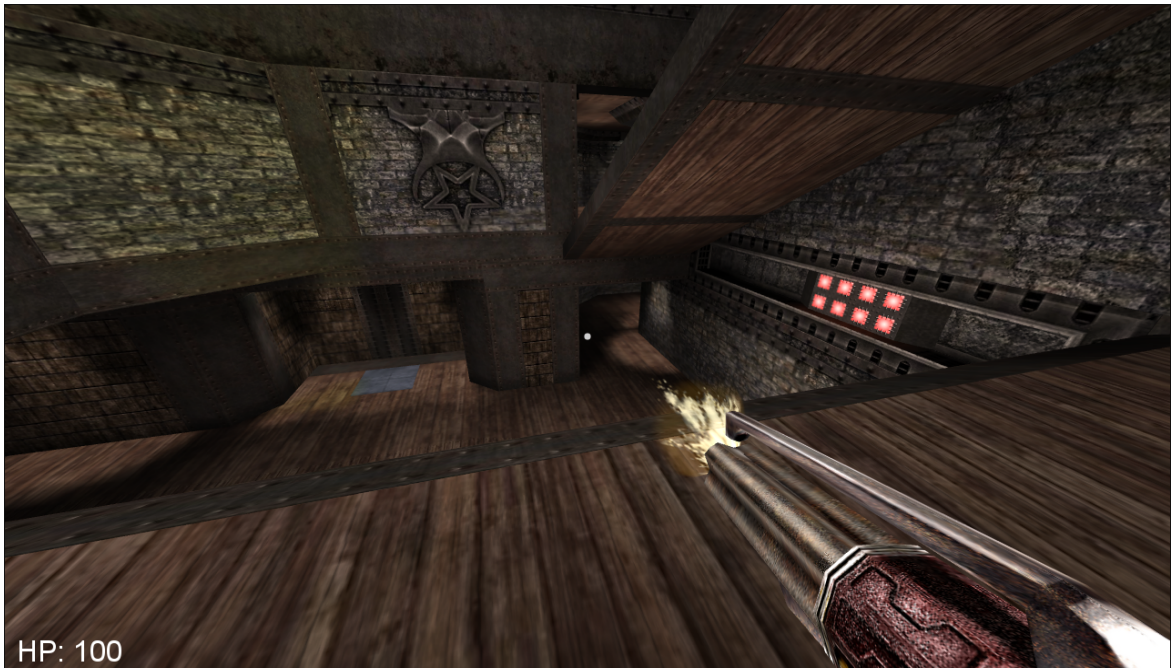
– spacja – skok,

– W, S, A, D – poruszanie się (odpowiednio: do przodu, do tyłu, lewo, prawo),

– ctrl – kucnięcie.

Gra działa na przeglądarkach Google Chrome i Mozilla Firefox w najnowszych wersjach. Obsługa innych przeglądarek może zostać dodana, kiedy będą już miały zaimplementowaną obsługę wszystkich potrzebnych standardów i wystarczająco szybki silnik JavaScript.

Grafika 4.7 przedstawia zrzut ekranu z gry.



Rysunek 4.7: Zrzut ekranu z gry Web Arena

5. Analiza

Niniejszy rozdział zaczyna się od podsumowania zrealizowanego projektu. Następnie na podstawie zebranych doświadczeń nastąpi analiza przydatności technologii HTML 5 do realizacji gier 3d.

5.1. Stopień realizacji projektu

Realizowany projekt – WebArena – miał być okrojonym klonem gry OpenArena, działającym w środowisku webowym.

O ile gra jest w pewnym stopniu uproszczona w porównaniu do pierwowzoru, o tyle udało się odtworzyć większość funkcjonalności silnika gry, a więc dodawanie kolejnych funkcjonalności na poziomie samej rozgrywki jest już zadaniem stosunkowo prostym (choć wymaga trochę czasu) i nie wpłynie w znacznym stopniu na wydajność gry czy jej ogólną konstrukcję.

Największym brakiem w grze jest brak dźwięków. Nie wynika on jednak z niedostatków technologicznych HTML 5, gdyż standard oferuje bardzo rozbudowaną bibliotekę do obsługi dźwięku (rozdział 2.1.6, a jedynie z ograniczonych zasobów czasowych.

Największym wyzwaniem była realizacja renderera i multiplayera. Jednocześnie są to najważniejsze komponenty dla sieciowej gry 3D. W WebArana zostały one odtworzone z prawie całą funkcjonalnością dostępną w pierwowzorze. Renderer ma następujące braki:

– brak wyświetlania mgły i wody,

– brak wyświetlania śladów po kulach na ścianach,

– brak luster,

– brak optymalizacji polegającej na nierenderowaniu odległych przedmiotów.

Dodanie tych funkcjonalności nie będzie trudnym zadaniem przy istniejących już podstawach, a poza ostatnim punktem nie wpływają one znacząco na wydajność gry.

Jeśli chodzi o część odpowiedzialną za multiplayer, to jedynym poważniejszym brakiem jest przyjazna dla użytkownika obsługa utraty połączenia sieciowego.

5.2. Porównanie z grami natywnymi

5.2.1. Wydajność

Największą obawą deweloperów gier co do gier przeglądarkowych jest wydajność języka JavaScript w stosunku do C i C++. W najbardziej zaawansowanych graficznie grach każdy niemalże cykl procesora jest ważny. W ich przypadku języki natywne są jedynym możliwym wyborem. Tego typu gry to jednak niewielki ułamek całego rynku.

Od kilku lat trwa wyścig pomiędzy twórcami przeglądarek o najszybszy silnik JavaScript. W rezultacie, aktualnie JavaScript jest prawdopodobnie najszybszym językiem skryptowym. Jest to osiągnięte poprzez kompilację do kodu natywnego przed wykonaniem, profilowanie w czasie rzeczywistym i optymalizację najczęściej wykonywanego kodu, cache'owanie informacji o typie oraz bardzo szybkie odśmiecanie.

W efekcie, podczas tworzenia WebArana nie pojawił się ani raz problem ze zbyt niską wydajnością języka. Gra na średniej klasy komputerze sprzed kilku lat działa płynnie, a możliwych jest wiele optymalizacji w rendererze, które jeszcze przyspieszą działanie gry.

Istotny jest również sposób realizacji API WebGL. Jest ono bardzo niskopoziomowe i bardzo podobne do natywnego OpenGL. Shadery wykonują się na procesorze karty graficznej dokładnie tak samo jak w przypadku OpenGL. Dzięki temu wydajność WebGL w większości wypadków jest praktycznie taka sama jak OpenGL.

Podczas tworzenia gry trzeba jednak zwracać uwagę na kwestię istnienia garbage collectora. Pomimo, że te zaimplementowane we współczesnych przeglądarkach są bardzo szybkie, to jeśli aplikacja będzie alokować pamięć bardzo intensywnie, może się okazać, że odśmiecanie powoduje zauważalną pauzę. Z tego powodu najlepiej alokować większe ilości pamięci podczas wczytywania gry, a w czasie działania ograniczyć tworzenie nowych obiektów do minimum.

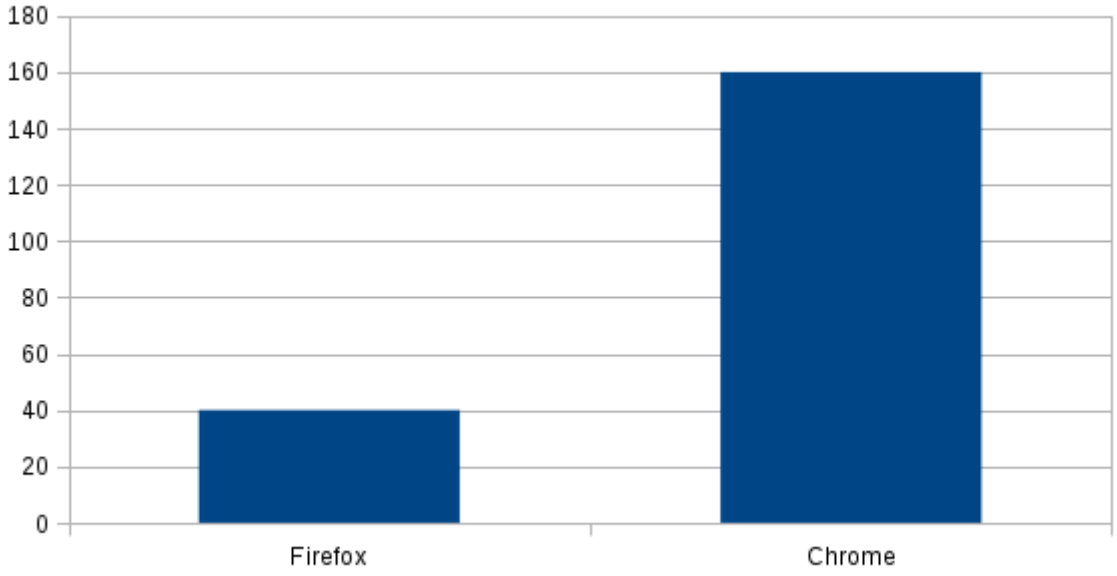
Należy zaznaczyć, że przekazane w tej sekcji dotyczą przede wszystkim przeglądarek Google Chrome i Mozilla Firefox, gdyż aktualnie tylko na nich działa gra (ze względu na implementacją potrzebnych standardów). Można się jednak spodziewać, że pozostałe przeglądarki zaoferują podobną wydajność silników skryptowych.

Na wykresie 5.1 przedstawiono średnią ilość klatek na sekundę[1]. Jak widać, Google Chrome jest szybsze, jednak Firefox również zdołał wygenerować około 40 klatek co jest wynikiem wystarczającym do komfortowej gry. Trzeba zwrócić uwagę, że przy standardowych ustawieniach przeglądarki, sterownik karty graficznej i tak ograniczy ilość klatek do 60, w celu synchronizacji z częstotliwością odświeżania monitora. Końcowy użytkownik nie zauważy więc tak dużej różnicy w wydajności pomiędzy przeglądarkami, jak widać to na porównaniu.

5.2.2. Wielowątkowość

Tak jak napisano w rozdziale 2.1.3, wielowątkowość jest realizowana w JavaScript przez API WebWorkers, które nakłada wiele ograniczeń. Workery są bardziej podobne do procesów niż wątków, po-

[1] FPS (frames per second) – ilość klatek wyrenderowanych podczas jednej sekundy. Popularna metryka wydajności gry.



Rysunek 5.1: Średnia ilość klatek na sekundę w grze

nieważ nie mogą współdzielić danych, ani kodu. Dodatkowo większość funkcji JavaScript jest niedostępnych poza głównym workerem (np. WebGL, WebRTC, Web Audio API), musi on więc pośredniczyć we wszystkich wywołaniach API.

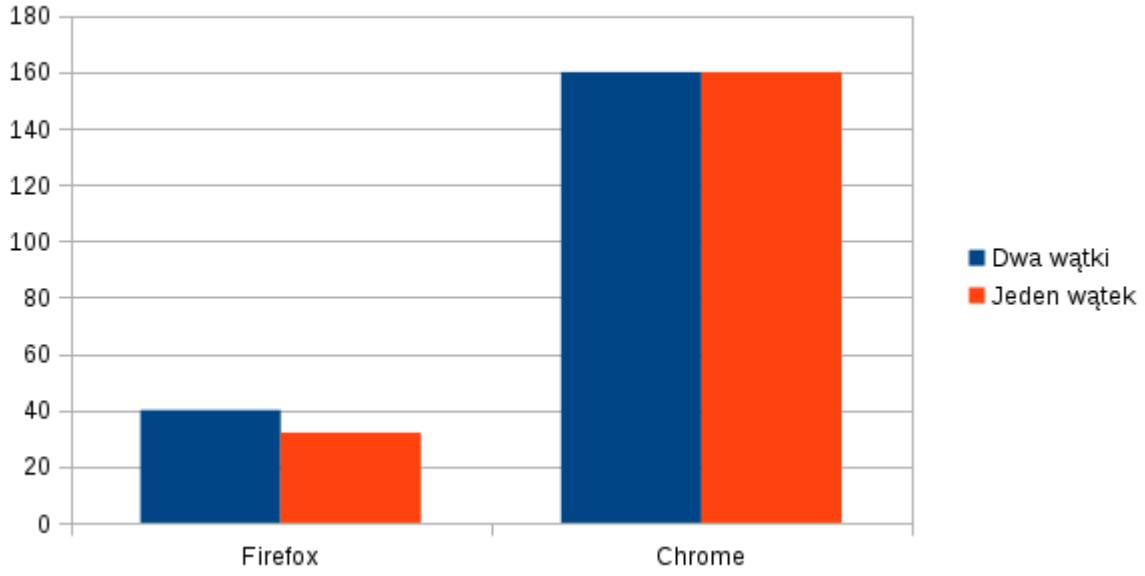
Taka konstrukcja sprawia, że bardzo trudno jest uzyskać znaczący wzrost wydajności dzięki rozdzieleniu pracy na np. dwa wątki. W przypadku WebArena można to było łatwo zauważyć dzięki bibliotece opisanej w rozdziale 4.3, która umożliwia szybkie przełączenie pomiędzy trybem jednowątkowym i dwuwątkowym. W pierwszym trybie, renderer i logika działają na wątku głównym, w drugim zaś są podzielone na dwa wątki.

Obserwacje wykazały, że wydajność gry, mierzona ilością wyświetlonych klatek na sekundę, zmienia się w niewielkim stopniu podczas przełączania trybów.

Wykres 5.2 przedstawia porównanie wydajności tych trybów. W Firefoksie udało się osiągnąć pewien zysk rozdzielając pracę na dwa wątki, jednak jest to zysk niewielki. Jedną z przyczyn takiej sytuacji jest to, że w celu komunikacji pomiędzy workerami konieczne jest kopiowanie dużych ilości danych, które nie mogą być współdzielone. To z kolei powoduje częstsze wywoływanie garbage collectora, co niweluje część zysków wynikających ze zrównoleglenia.

W Chrome nie występuje zauważalna różnica w wydajności. Jest to prawdopodobnie spowodowane tym, że silnik JavaScript w tej przeglądarce jest już na tyle szybki, że wąskim gardłem staje się procesor graficzny. Wobec tego przyśpieszanie kodu działającego na procesorze głównym, nie może zwiększyć ogólnych wyników.

Przy tych pesymistycznych wnioskach trzeba jednak zwrócić na to, że WebArena nie ma zbyt skomplikowanej logiki, przez co główny wątek (renderujący) ma więcej pracy niż wątek logiki gry. W takiej sytuacji niemożliwe jest bardzo duże przyśpieszenie przez podział na dwa wątki.



Rysunek 5.2: Porównanie ilość klatek na sekundę w trybie jednowątkowym i dwuwątkowym

Gry w których występuje np. symulacja fizyki i zaawansowana animacja szkieletowa, wciąż warto jest stosować Web Workers, najlepiej z dużą ilością krótkich zadań do wykonania, które łatwo oddelegować na dowolną ilość wątków. Takie rozwiązanie przyniosłoby dobre efekty zwłaszcza na procesorach posiadających więcej niż dwa rdzenie. Jeżeli jednak logika gry jest prosta, koszt projektowania systemu wielowątkowego specjalnie pod Web Workers, prawdopodobnie nie jest warty potencjalnych zysków.

Należy też zwrócić uwagę na jeszcze jedną zaletę Web Workers. Nawet jeżeli dodatkowe workery nie przyniosą zysków wydajnościowych, to umożliwią wykonywanie czasochłonnych zadań (takich jak parsowanie pliku binarnego) bez blokowania interfejsu użytkownika. Jest to bardzo istotne patrząc od strony ergonomii użytkowania gry.

5.2.3. Proces programowania gry

Tworzenie gry przeglądarkowej z punktu widzenia programisty wygląda inaczej niż w przypadku natywnych gier. JavaScript jest językiem skryptowym, uruchamianym w przeglądarce.

Ma to następujące zalety:

– JavaScript jest bardzo elastyczny i programuje się w nim łatwiej niż np. w C.

– Brak konieczności kompilacji sprawia, że bardzo szybko można wprowadzać i testować poprawki.

– Możliwość edycji kodu w przeglądarce, co jeszcze bardziej przyśpiesza testowanie poprawek.

– Dobre narzędzia do debugowania kodu JavaScript oraz błędów w wyświetlaniu grafiki 3d.

Z kolei do wad należą:

– Mniejsza wydajność w porównaniu z językami natywnymi (aczkolwiek w większości przypadków wystarczająca),

– Brak kompilacji powoduje, że później zostają wykryte błędy,

– Elastyczność JavaScriptu w rękach niedoświadczonego programisty może prowadzić do powstania źle zorganizowanego i trudnego w utrzymaniu kodu.

– Mniejsza dostępność gotowych bibliotek wspomagających tworzenie gier.

5.2.4. Środowisko przeglądarkowe

Fakt, że gra działa wewnątrz strony internetowej powoduje, że wiele problemów, koniecznych do rozwiązania podczas pisania gry natywnej, po prostu nie istnieje.

Jeden z przykładów to parsowanie plików graficznych. Jeżeli tekstury są dostarczane w jednym z formatów obsługiwanych przez przeglądarkę, nie trzeba pisać dodatkowego parsera, gdyż przeglądarka zajmie się ich wczytaniem. Obrazek taki można następnie przekazać do WebGL jako teksturę lub po prostu wyświetlić jako element interfejsu, również bez konieczności tworzenia własnego renderera 2D.

Podobnie jest z plikami dźwiękowymi, czy filmami występującymi w grach jako wprowadzenie w fabułę.

Jeszcze większą zaletą osadzenia gry w środowisku strony internetowej, jest dostęp do drzewa dokumentu HTML. Dzięki temu programista ma do dyspozycji gotowe rozwiązanie do tworzenia interfejsu użytkownika. W dodatku jest to jedno z najbardziej przemyślanych i wypróbowanych rozwiązań, a jednocześnie jedno z najpotężniejszych pod względem oferowanych możliwości.

Należy tutaj zaznaczyć, że stworzenie dobrego graficznego interfejsu użytkownika w grach jest trudnym i żmudnym zadaniem. Już samo prawidłowe renderowanie tekstów jest zadaniem nietrywialnym. Mając do dyspozycji HTML, problem ten znika.

5.2.5. Wieloplatformowość

Twórcy gier dążą do tego, aby ich produkty działały na jak największej ilości urządzeń. Jednak proces portowania gry na nową platformę jest bardzo często trudny i nieopłacalny. W przypadku HTML 5 sprawa jest prosta – jeżeli przeglądarka na danej platformie oferuje wszystkie potrzebne API, to gra po prostu będzie działać.

Już teraz WebGL jest dostępne na wszystkich liczących się platformach PC (Windows, Mac, Linux), oraz wielu mobilnych. Należy założyć, że również przeglądarki działające na konsolach do gier nowej generacji zapewnią wkrótce obsługę tego standardu, co otworzy możliwość dotarcia do ich użytkowników dla niezależnych deweloperów[2].

Oczywiście samo uruchomienie gry to nie wszystko. Często trzeba dostosować sposób kontroli w grze (np. dla ekranów dotykowych), czy wyświetlania na mniejszym ekranie. Mimo to HTML 5 jest dużym ułatwieniem dla tworzenia gier wieloplatformowych.

[2]Gry na konsole takie jak PlayStation czy Xbox mogą tradycyjnie tworzyć tylko certyfikowani deweloperzy.

5.2.6. Marketing i Monetyzacja

Aby sprzedać grę należy dotrzeć do jak największej liczby odbiorców. Tutaj platforma internetowa pokazuje swoje kolejne zalety. Jedną z nich jest łatwa integracja z serwisami społecznościowymi, które są kluczowe dla zaistnienia w świadomości użytkowników.

Kolejną istotną cechą jest łatwa dostępność gry. Nie ma potrzeby czasochłonnego pobierania i instalacji. Wystarczy kliknięcie w link i po chwili można zacząć rozgrywkę.

W tym miejscu trzeba jednak napisać o prawdopodobnie aktualnie największym problemie gier HTML 5. Jest nim trudność monetyzacji gry wynikająca z tego, że rynek dopiero zaczyna się rozwijać.

Dla gier działających natywnie na PC istnieje wiele sklepów i platform dystrybucji cyfrowej. Są one bardzo popularne wśród graczy, można więc za ich pośrednictwem łatwo trafić do klientów zainteresowanych płaceniem za gry. Dodatkowo sklep zajmuje się obsługą transakcji.

Dla gier webowych takie platformy dopiero się pojawiają i na razie mają niewielką liczbę klientów. Dlatego aktualnie najlepszym sposobem na zarobienie na grze jest model free to play. W tego typu gry każdy może zacząć grać za darmo i płacić wedle uznania za dodatkowe usługi. Niestety taki model nie nadaje się do każdej gry.

Jeden z serwisów który może w przyszłości zmienić tę sytuację jest `turbulenz.com`. Oferuje on już kilka darmowych gier, pracuje także nad wprowadzeniem płatności. Co więcej, oferuje też darmowy, kompletny silnik do tworzenia gry w HTML 5, który można wykorzystać również poza serwisem.

5.3. Podsumowanie przydatności HTML5 do tworzenia gier

W tym podrozdziale znajduje się krótkie podsumowanie wymienionych wcześniej zalet i wad tworzenia gier w HTML 5.

5.3.1. Zalety

- Łatwość i szybkość tworzenia kodu w JavaScript.
- Bogate API HTML (np. do tworzenia interfejsu użytkownika).
- Narzędzia ułatwiające tworzenie aplikacji internetowych.
- Wieloplatformowość.
- Łatwa integracja z serwisami społecznościowymi.
- Natychmiastowa dostępność dla graczy.

5.3.2. Wady

- Mniejsza wydajność JavaScript w porównaniu do języków natywnych.
- Brak tradycyjnych wątków.

– Mniejsza dostępność bibliotek wspomagających tworzenie gier.

– Elastyczność i brak kompilacji JavaScript może stać się wadą.

– Trudność monetyzacji.

6. Podsumowanie

Nowe technologie związane z HTML 5 stwarzają dla twórców gier szansę zdobycia nowych rynków. Niniejsza praca miała na celu zbadanie tematu tworzenia gier w HTML 5, na przykładzie gry typu FPS. Podczas pisania pracy udało się stworzyć projekt, odtwarzający jedną ze znanych gier w przeglądarce internetowej.

6.1. Ocena realizacji celów pracy

Praca miała następujące cele:

1. Stworzenie gry typu FPS z wykorzystaniem technologii zawartych w HTML 5.
2. Analiza możliwości tych technologii w porównaniu do tradycyjnego podejścia do tworzenia gier.

Cele te udało się zrealizować.

Realizację celu pierwszego można śledzić w rozdziałach 3 i 4. Cel drugi został zrealizowany w rozdziale 5.

6.2. Ocena prawdziwości tezy pracy

Niniejsza praca dowodzi prawdziwości tezy: *Technologia HTML 5 umożliwia tworzenie zaawansowanych gier 3D.*

6.3. Możliwości dalszego rozwoju projektu

O ile projekt realizuje badawczy cel pracy, o tyle prezentuje raczej poziom technologicznego eksperymentu niż produktu komercyjnego. Możliwe jest jednak rozbudowanie go, aby nadawał się do upublicznienia w internecie. Oto potencjalne kierunki rozwoju gry:

– ulepszenie i optymalizacja renderera,

– dodanie nowych broni i przedmiotów do gry,

– nowe poziomy,

– nowe tryby gry (np. drużynowy),

- nowe modele postaci,
- integracja z portalami społecznościowymi,
- rankingi graczy,
- system zabezpieczający przed oszukiwaniem w rozgrywce sieciowej,
- wprowadzenie opcjonalnych płatności za dodatkową zawartość w grze.

Bibliografia

[1] J. Andrews. Designing the framework of a parallel game engine. https://software.intel.com/en-us/articles/designing-the-framework-of-a-parallel-game-engine/, 2008. [Online; accessed 2014-05-11].

[2] ECMA. Ecmascript specification. http://www.ecma-international.org/publications/standards/Ecma-262.htm, 2014. [Online; accessed 2014-04-16].

[3] D. Flanagan. *JavaScript: The Definitive Guide*. O'Reilly Media, 2011.

[4] J. Gregory. *Game Engine Architecture*. A K Peters, Ltd., 2009.

[5] K. Group. Webgl 1.0 specification. https://www.khronos.org/registry/webgl/specs/1.0/, 2013. [Online; accessed 2014-04-16].

[6] H. H. Jouni Smed. *Algorithms and Networking for Computer Games*. Wiley, 2006.

[7] Mozilla. asm.js. http://asmjs.org/, 2014. [Online; accessed 2014-04-16].

[8] V. Mönkkönen. Multithreaded game engine architectures. http://www.gamasutra.com/view/feature/130247/multithreaded_game_engine_.php, 2006. [Online; accessed 2014-05-11].

[9] B. H. Paul Jaquays. Quake iii arena shader manual. http://toolz.nexuizninjaz.com/shader/, 1999. [Online; accessed 2014-05-28].

[10] PhaethonH. Description of md3 format. http://www.icculus.org/~phaethon/q3/formats/md3format.html, 2011. [Online; accessed 2014-05-28].

[11] K. Proudfoot. Unofficial quake 3 map specs. http://www.mralligator.com/q3/, 2000. [Online; accessed 2014-05-28].

[12] F. Sanglard. Quake 3 source code review. http://fabiensanglard.net/quake3/, 2012. [Online; accessed 2014-04-16].

[13] W3C. Html5 specification. http://www.w3.org/TR/html5/, 2014. [Online; accessed 2014-04-16].