Uniwersytet Łódzki
Wydział Matematyki i Informatyki
Kierunek Informatyka

Bartłomiej Rogowski

Nr albumu 313884

Zastosowanie chmury obliczeniowej Microsoft Azure w grach komputerowych.

Praca magisterska napisana
w Katedrze Analizy Nieliniowej
pod kierunkiem dr Cezarego Obczyńskiego

Łódź 2015

Spis treści

Wstęp

W ciągu ostatnich kilkunastu lat dużo zmieniło się w sferze informatyki. Na rynku pojawiły się platformy mobilne (smartfony, tablety, netbooki). Usługi chmurowe znalazły swoje zastosowanie w codziennym życiu. Umacnia się trend do wykorzystywania gotowych silników (zarówno fizycznych jak i graficznych) niżeli tworzenie własnych. Dodatkowo sam Internet, który jeszcze całkiem niedawno był powolny, drogi i słabo dostępny – dziś jest już szybki, tani i ogólnodostępny.

Jednakże, nawet najlepsze rozwiązania posiadają wady. Urządzenia mobilne wciąż posiadają znacznie mniejszą moc obliczeniową niż standardowe komputery czy konsole. Usługi chmurowe ze względu na swoją naturę (brak fizycznego dostępu do nich) wciąż borykają się z brakiem zaufania zarówno ze strony klientów końcowych tak jak i większych przedsiębiorstw. Większe firmy produkujące gry bardzo powoli adaptują się do rynku gotowych silników gier.

Najnowszym trendem informatycznym jest tak zwany „Big Data". Jest to szeroki termin określający zestawy danych tak duże i skomplikowane, iż tradycyjne sposoby przetwarzania jej stają się nieadekwatne.

Zarówno gry komputerowe, aplikacje mobilne czy webowe produkują właśnie tego typu dane. Dane te są następnie przetwarzane w usługach chmurowych. Jest to jeden z najlepszych przykładów uzupełniania problemów istniejących już usług (np. słaba moc obliczeniowa smartfonów) za pomocą technologii serwisów chmurowych.

W najbliższym czasie będziemy świadkami jak coraz więcej aplikacji, w tym gier komputerowych, korzystać będzie z potężnych możliwości usług w chmurze.

Praca magisterska została napisana przy współpracy z firmą Marcin Franc – IT Consulting, jako część aktywności R&D wspieranej bezpośrednio przez Microsoft Polska.

Rozdział 1
Cel i założenia pracy

Celem pracy jest demonstracja połączenia rozwiązań chmurowych (w oparciu o platformę Microsoft Azure), urządzeń mobilnych oraz gotowego silnika gier (Unity3D). Przykład owego rozwiązania będzie składał się z serwera fizycznego znajdującego się w chmurze Azure oraz klienta graficznego połączonego do ów serwera. Aplikacje zarówno klienta jak i serwera będą oparte na silniku Unity3D.

Wykorzystując dane technologie w ten sposób, możemy umożliwić uruchamianie złożonych aplikacji (graficznych i fizycznych) na urządzeniach nieposiadających dużych możliwości obliczeniowych jak na przykład platformy mobilne.

Będzie to możliwe dzięki odseparowaniu warstwy fizycznej od warstwy graficznej, które praktycznie zawsze są ze sobą ściśle związane. Warstwa fizyczna będzie w całości obliczana na serwerze, a warstwa graficzna jedynie na urządzeniu klienckim.

1.1 Przegląd

Praca składa się z 5-ciu rozdziałów podzielonych ze względu na poziom szczegółowości. Rozdział pierwszy to przedstawienie celów oraz założeń projektowych. Rozdział drugi wprowadza podłoże teoretyczne użyte w projekcie. Następny rozdział przedstawia technologie oraz narzędzia wykorzystane do stworzenia omawianej tutaj aplikacji. Dodatkowo znajduje się w nim opis dwóch istotnych technologii – Microsoft Azure oraz Unity3D. Rozdział 4 rozlegle opisuje zarówno architekturę aplikacji jak i szczegóły implementacyjne. Pracę zamyka ostatni rozdział – podsumowanie.

1.2 Załączniki do pracy

Do pracy został dołączony nośnik CD zawierający kod źródłowy aplikacji, gotową do uruchomienia kopię aplikacji w wersji klienckiej i w wersji serwerowej oraz instrukcję uruchomieniową.

Rozdział 2
Podstawowe pojęcia

2.1 Chmura obliczeniowa

Chmura obliczeniowa (ang. cloud computing) odnosi się do praktyki przechodzących usług komputerowych, takich jak obliczenia lub przechowywanie danych w wielu lokalizacjach zdalnych, dostępnych w Internecie. Umożliwia to aplikacjom być obsługiwanym przy użyciu urządzenia z dostępem do Internetu. Chmury mogą być klasyfikowane, jako publiczne, prywatne i hybrydowe.

Przykładem chmury publicznej może być chmura Azure gdzie jej usługodawcą jest firma Microsoft.

2.2 Programowanie rozproszone

Programowanie rozproszone jest techniką wykorzystującą systemy rozproszone. Rozproszony system to system oprogramowania, w którym składniki znajdujące się na komputerach w sieci komunikują się i koordynują swoje działania, przekazując wiadomości. Elementy współdziałają ze sobą w celu uzyskania wspólnego celu.

Trzy istotne cechy systemów rozproszonych to: współbieżność elementów, brak globalnego zegara, i niezależna awaria elementów.

2.3 Architektura Klient-Serwer

Architektura klient-serwer jest sposobem na dostarczenie usługi z centralnego źródła. Jest tylko jeden serwer, który świadczy usługę, a wielu klientów, którzy komunikują się z serwerem, aby konsumować jego produkty/usługi. w tej architekturze, klienci i serwery mają różne zadania. Zadaniem serwera jest odpowiedź na prośby usług od klientów, podczas gdy zadaniem klienta jest korzystanie z danych przedstawionych w odpowiedzi, aby wykonać powierzone mu zadania.

2.4 Testy

2.4.1 Testy jednostkowe

W programowaniu, testy jednostkowe są metodą testowania oprogramowania, w którym poszczególne jednostki kodu źródłowego, zestawy jednego lub więcej modułów programu komputerowego wraz z powiązanymi danymi, procedurami kontroli użytkowania oraz procedurami operacyjnymi, są badane w celu określenia, czy są one zdatne do użytku. Intuicyjnie można określić jednostkę, jako najmniejsza testowalną części aplikacji. w programowaniu proceduralnym, jednostką może być cały moduł, ale powszechnie jest to indywidualna funkcja lub procedura. w programowaniu obiektowym, jednostką może być cały interfejs, jak na przykład klasa, ale może być również indywidualna metoda.

Testy jednostkowe są to krótkie fragmenty kodu utworzone przez programistów podczas procesu rozwoju aplikacji. Idealnie, każdy test jest niezależny od innych. Zamienniki takie jak metody-atrapy, obiekty-atrapy mogą być wykorzystane do testowania modułu w izolacji. Testy jednostkowe zazwyczaj są pisane i prowadzone przez twórców oprogramowania, aby upewnić się, że kod spełnia jego założenia i zachowuje się zgodnie z przeznaczeniem.

2.4.2 TDD

Test-Driven Development (TDD) jest to proces wytwarzania oprogramowania, które opiera się na powtarzaniu bardzo krótkiego cyklu: najpierw programista pisze (początkowo niedziałający) zautomatyzowany przypadek testowy, który definiuje pożądaną poprawę lub nową funkcję, następnie wytwarza minimalną ilość kodu zdać ten test, a na końcu refaktoryzuje nowy kod do dopuszczalnych norm.

Rozdział 3
Narzędzia i technologie

3.1 Język programowania

Językiem programowania użytym w projekcie jest język C#.

C# jest to wieloparadygmatowy język programowania obejmujący silne typowanie, deklaratywność, generyczność, funkcyjność, obiektowość oraz programowania zorientowane komponentowo. Został on opracowany przez firmę Microsoft w ramach inicjatywy .NET, a następnie zatwierdzony jako standard przez ECMA (ECMA-334) oraz ISO (ISO/IEC 23270 : 2006).

C# to jeden z języków programowania przeznaczonych do architektury wspólnego języka (CLI). Jest on natywnie wspierany przez platformę Azure oraz silnik Unity3D.

3.2 Silnik gry

Unity3D to multi-platformowy silnik gier stworzony przez Unity Technologies i jest używanie do tworzenia gier na PC, konsole, urządzenia mobilne oraz strony internetowe.

Moduł skryptowy w tym silniku jest zbudowany w oparciu o technologię Mono – open-sourcową implementację platformy programistycznej .NET. Dzięki temu programiści Unity3D mogą korzystać z pełni funkcjonalnego języka C#.

Unity3D jest znany ze swoich możliwości tworzenia gier na różne platformy. W chwili pisania pracy dostępnych było 21 platform w tym min.: Windows, OS X, Android, iOS, Playstation 3, Playstation 4, Xbox 360, Xbox One czy Wii U.

3.3 Narzędzia i biblioteki programistyczne

3.3.1 Microsoft Visual Studio 2013

Visual Studio jest to zintegrowane środowisko programistyczne stworzone przez firmę Microsoft. Umożliwia ono tworzenie programów zarówno pod systemy Microsoft Windows jak i aplikacje webowe, strony internetowe czy serwisy chmurowe. Visual Studio wykorzystuje programistyczne platformy Microsoft takie jak: Windows API, Windows Presentation Foundation, Windows Store czy Microsoft Azure.

Visual Studio zawiera szereg narzędzi przydatnych programiście jak edytor kodu, autouzupełnianie kodu czy debugger. Wsparte dodatkami takimi jak ReSharper (narzędzia refaktoryzacyjne), dotCover (narzędzie obliczające pokrycie kodu testami jednostkowymi) czy Visual Studio 2013 Tools for Unity (debugger dla edytora Unity) staje się jedynym wymaganym narzędziem dla programisty platformy .NET.

3.3.2 Biblioteki pomocnicze

Bibliotekami użytymi w projekcie były jedynie biblioteki wspierające testy jednostkowe na platformie .NET.

- NUnit – open-sourcowa platforma programistyczna testów jednostkowych
- NFluent – biblioteka asercji dla testów jednostkowych
- NSubsitute/RhinoMocks – biblioteka atrap (umożliwia separację komponentów, co ułatwia testowanie)

3.4 Microsoft Azure

Microsoft Azure jest to platforma chmury obliczeniowej i infrastruktury, stworzona przez Microsoft, do budowania, wdrażania i zarządzania aplikacjami oraz usługami za pośrednictwem globalnej sieci zarządzanych przez Microsoft centrach danych. Obsługuje wiele różnych języków programowania, narzędzi i struktur, w tym zarówno specyficzne dla Microsoft systemy jak i oprogramowanie firm trzecich.

Platforma Azure zapewnia pełną gamę rozwiązań chmurowych zaczynając od przechowywania danych poprzez wirtualizację maszyn kończąc na uczeniu maszynowym.

W projekcie wykorzystane zostały usługi Azure Blob Storage (do przechowywania plików aplikacji) oraz Virtual Machines do hostowania aplikacji w chmurze.

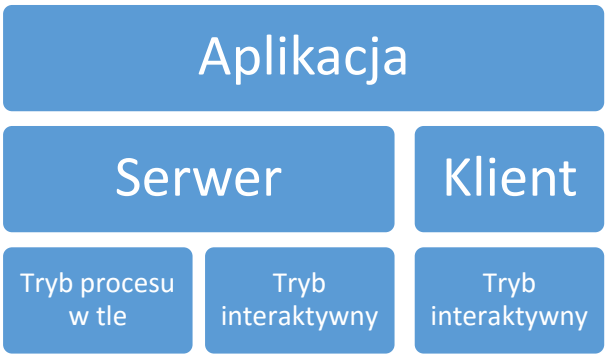
3.5 Repozytorium kodu

System kontroli wersji użyty w projekcie to Git. Jest to rozproszony system kontroli wersji gdzie każdy katalog roboczy jest pełnoprawnym repozytorium. Każde repozytorium Git posiada pełne możliwości śledzenia zmian, niezależny dostęp do sieci czy serwera centralnego.

Zdalnym serwerem Git dla niniejszego projektu było Visual Studio Online. Samo Visual Studio Online jest usługą typu „software as a service" dostępna na platformie Azure. Udostępnia ona gamę narzędzi developerskich takich jak: edytor kodu online, automatyzacja budowania projektu, tablica zadań czy użyte w projekcie repozytorium kodu Git.

Rozdział 4
Aplikacja

Aplikacja została podzielona na dwie części.



Rysunek 1: Podział aplikacji

Pierwsza realizuje zadanie serwera – udostępnia zasoby obliczeniowe komputera, na którym została uruchomiona, poprzez symulację fizyczną opartą na silniku Unity3D. Dodatkowo część serwerowa może być uruchamiana w 2 trybach: jako proces w tle, bez renderowania grafiki bądź na pierwszym planie jako interaktywna aplikacji z renderowną sceną.

Druga część aplikacji zajmuje się konsumowaniem zasobów serwera oraz prezentacji ich użytkownikowi. Program kliencki łączy się do wersji serwerowej, następnie jednocześnie przesyła dane wejściowe użytkownika do serwera jak i odbiera wyniki symulacji. Otrzymane dane są następnie prezentowane użytkownikowi.

4.1 Architektura

Kod aplikacji został podzielony na dwa główne moduły.

Moduł silnika – część kodu odpowiedzialna za zarządzanie silnikiem Unity3D. w tej sekcji znajdują się klasy odpowiedzialne za renderowanie, fizykę, interakcję użytkownika z silnikiem oraz wykorzystanie modułu niezależnego.

Moduł niezależny – część główna aplikacji. Obsługuje ona wszystkie wewnętrzne procesy, które nie są zależne od silnika gry[1]. Znajduje się tutaj logika zarządzająca połączeniem internetowym, serializacją danych, tworzeniem wiadomości do komunikacji klient – serwer oraz wiele innych.

4.2 Moduł silnika

Moduł silnika składa się z 3 części:

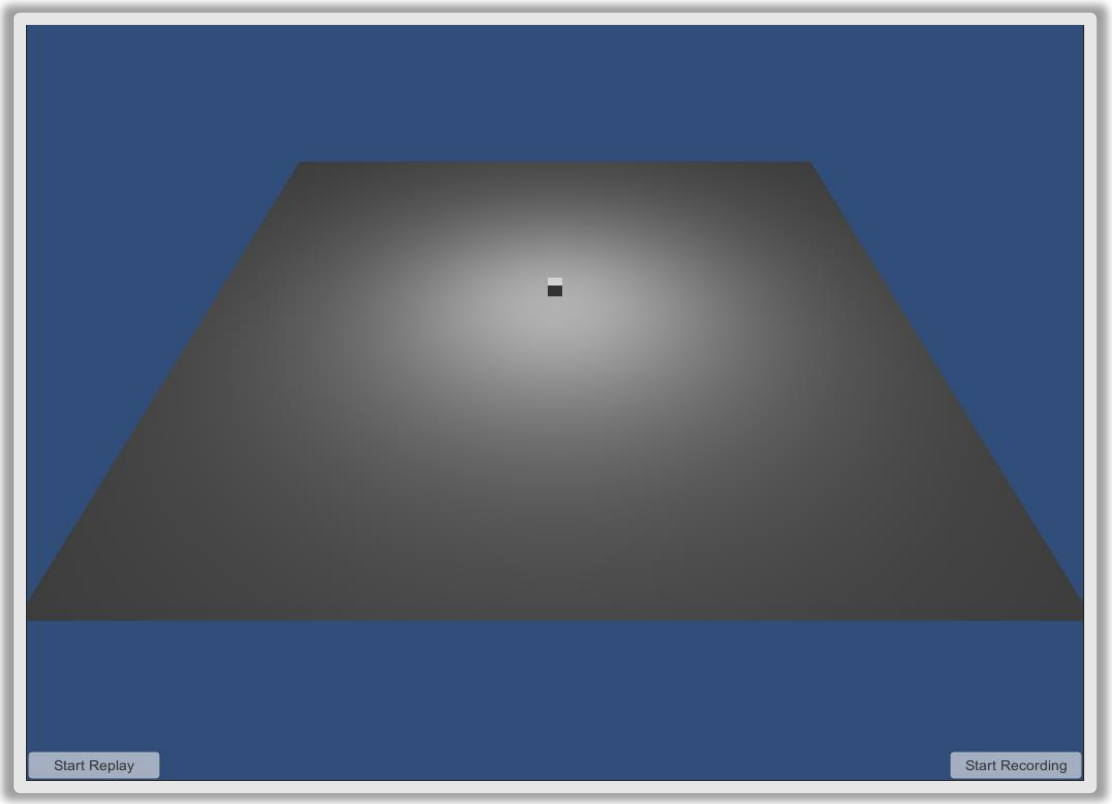
- Sceny – gotowe poziomy silnika służące w większości przypadków celom testowym
- Skrypty – kod źródłowy odpowiedzialny za funkcjonalności ściśle związane z silnikiem Unity3D
- Assety – elementy graficzne użyte w scenach

W tym rozdziale szczególna uwaga poświęcona została scenom, mniejsza skryptom oraz assety z racji natury graficznej zostały pominięte.

[1] Wyjątkiem są typy proste Vector3 czy Quaternion, które dla uproszczenia zostały wykorzystane bezpośrednio z bibliotek silnika Unity3D. w przypadku zmiany silnika na inny, podmiana tych komponentów jest trywialna.

4.2.1 Sceny

4.2.1.1 Test01



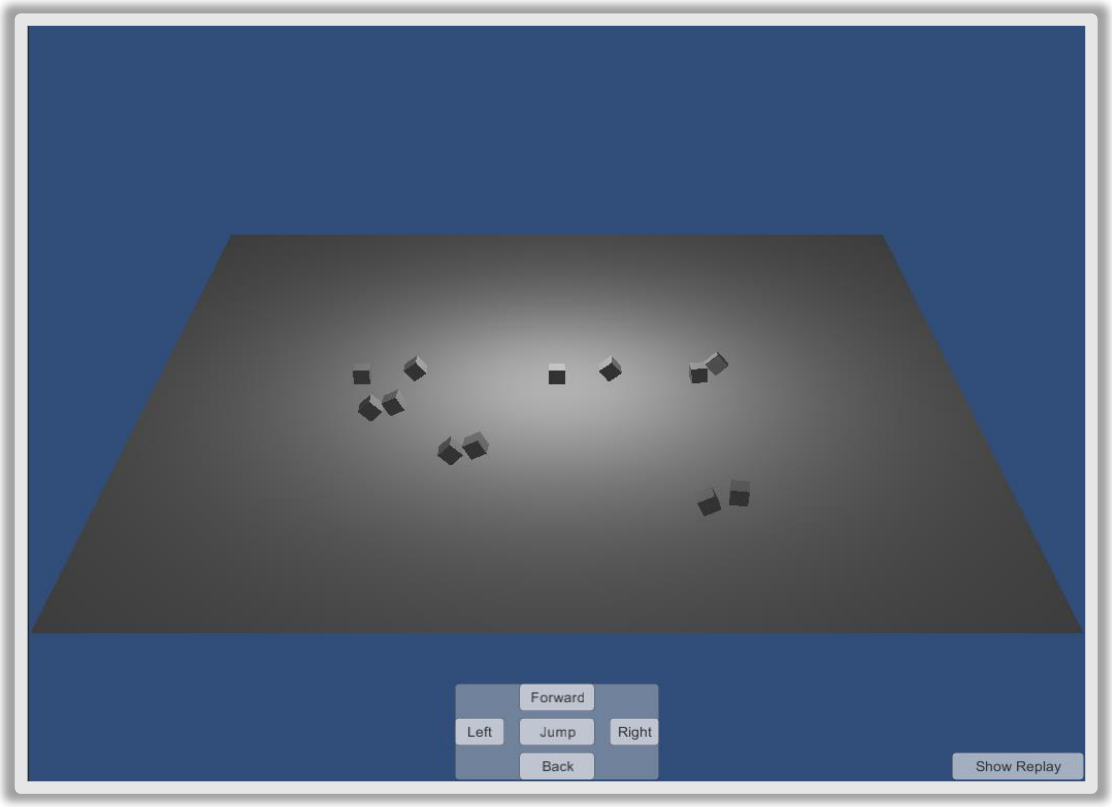
Rysunek 2: Scena Test01

Pierwsza powstała scena w projekcie. Zawiera ona test modułu odpowiedzialnego za zapisywanie danych otrzymywanych od użytkownika a następnie ich odtwarzania. w tym wypadku za pomocą klawiatury (W,S,A,D) użytkownik może poruszać sześcianem widocznym na rysunku. Przycisk „Start Recording" uruchamia system zapisywania danych z klawiatury. Przycisk „Start Replay" uruchamia zapisaną wcześniej sekwencję ruchu obiektu.

Wraz ze sceną powstały tutaj dwa podstawowe moduły projektu: Input, Recording oraz TimedList.

Wynik testu dowiódł, iż możliwe jest nagrywanie ruchów oraz ich odgrywanie, lecz nie jest to najlepsze rozwiązanie. Wbudowany w Unity3D silnik fizyczny nie jest deterministyczny oraz zależność otrzymywanych danych od użytkownika z ilością renderowanych klatek na sekundę powoduje, iż wyniki symulacji mogą się od siebie zasadniczo różnić.

4.2.1.2 Test02

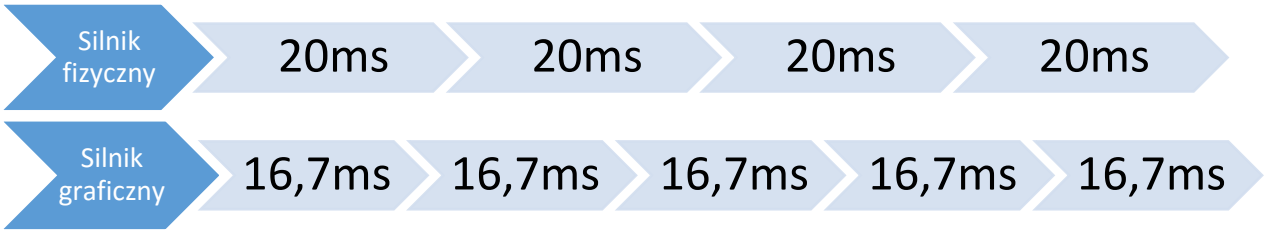


Rysunek 3: Scena Test02

Kolejny test sprzężonych modułów Input, Recording i TimedList. Użytkownik za pomocą przycisków widocznych w dolnej części ekrany może poruszać wszystkim sześcianami znajdującymi się na scenie. Przycisk „Show Replay" powoduje ustawienie sześcianów w pozycji początkowej oraz odegranie nagranej poprzednio sekwencji.

Różnicą w stosunku do poprzedniego testu są zapisywane dane. Zamiast danych od użytkownika, program zapisuje dla każdego z sześcianów, różnice w pozycji i obrocie pomiędzy iteracjami silnika fizycznego. Następnie, przy odgrywaniu zapisanej sekwencji, sześciany usuwane są z silnika fizycznego i poruszane są jedynie przy pomocy zapisanych wcześniej danych.

Ponieważ w czasie odgrywania sekwencji ruchów silnik fizyczny nie jest wykorzystywany zapisane dane muszą być odgrywane podczas iteracji silnika graficznego. Jest to duży problem, ponieważ silnik renderujący działa w zupełnie innej częstotliwości niż silnik fizyczny.



Rysunek 4: Różnica w czasie działania silników fizycznego i graficznego

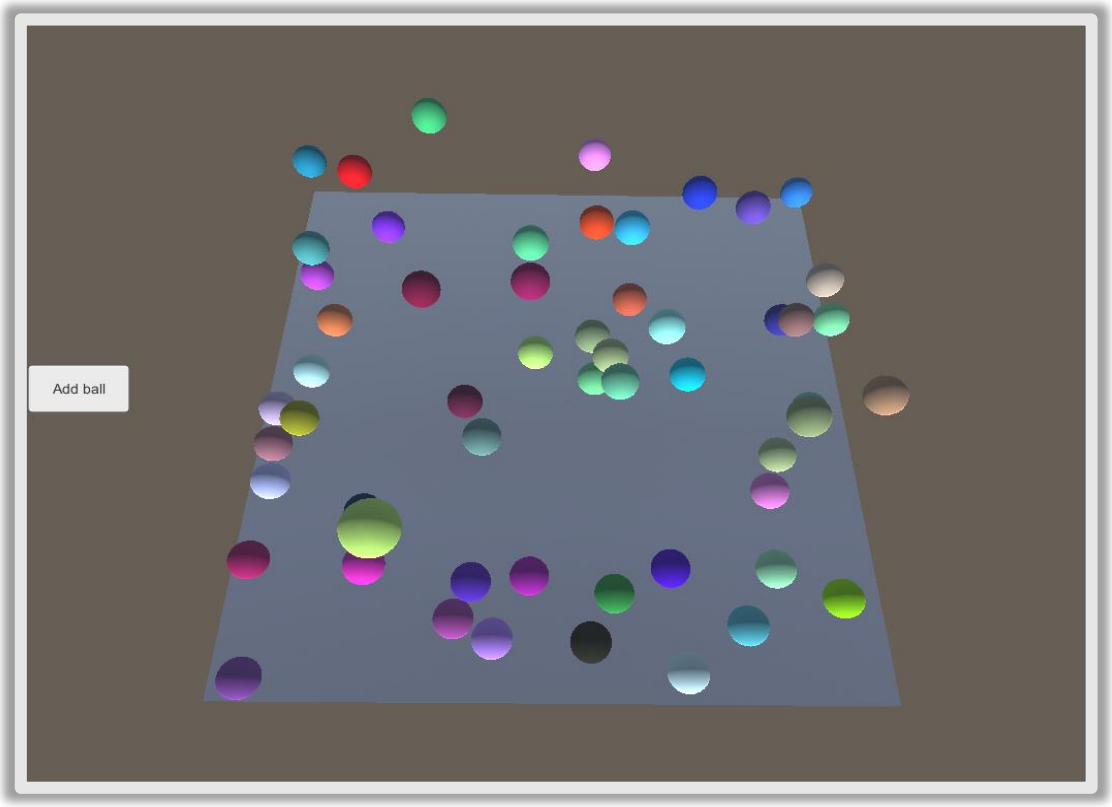
Dodatkowo warto zaznaczyć, iż silnik fizyczny ma stały ułamek sekundy (0,02) w którym działa a silnik graficzny jest zależny od mocy sprzętu oraz od docelowej wartości renderowanych klatek na sekundę (tutaj 60k/s).

Ze względu na wymienione powyżej różnice w czasie wykonywania nagranych sekwencji ruchu, zostały zastosowane odpowiednie dla każdej zmiennej (pozycja, obrót) algorytmy interpolacji czasowej.

Wynik testów był jak najbardziej pozytywny. Udało się pokazać, że nagrywanie samych różnic powoduje nie tylko wyniki praktycznie identyczne[2] ale i umożliwia wykluczenie silnika fizycznego przy odgrywaniu zapisanych sekwencji.

[2] Różnice w wynikach były spowodowane brakiem dostatecznej dokładności liczb zmiennoprzecinkowych używanych w silniku Unity3D. Różnice te były nie dostrzegalne na wyrenderowanej scenie.

4.2.1.3 Test03



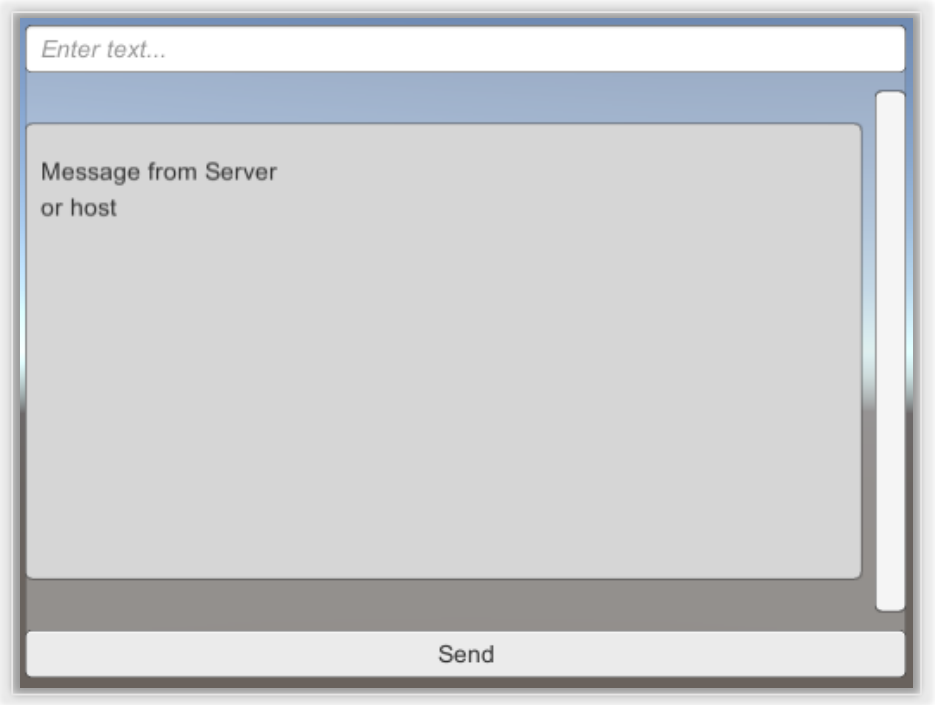
Rysunek 5: Scena Test03

Test silnika fizycznego Unity3D z użyciem prostych obiektów typu sfera.

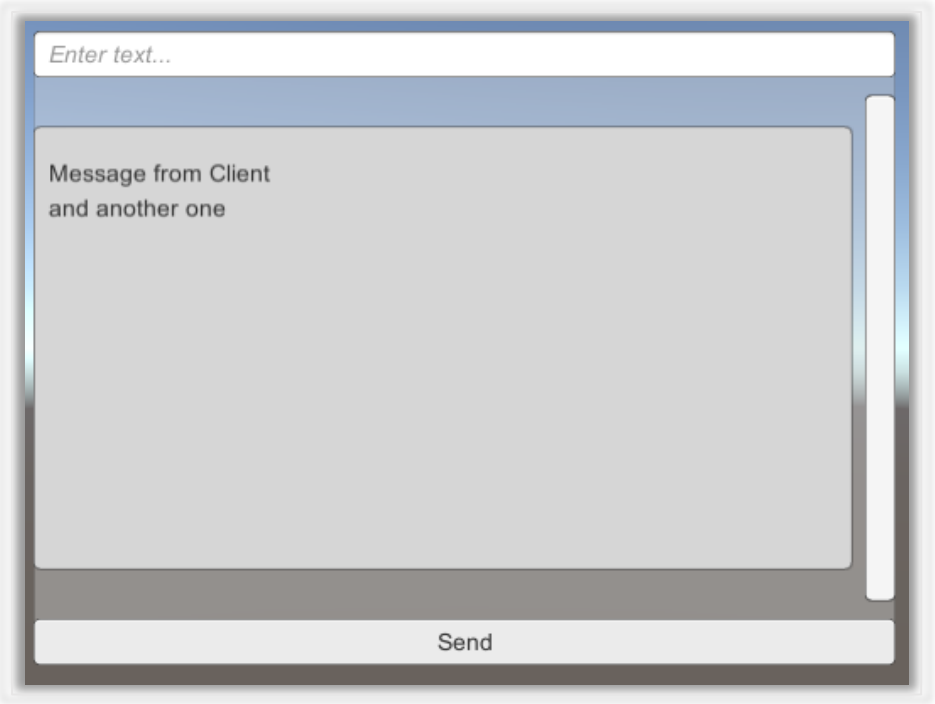
Celem testu było sprawdzenie czy można użyć dużej ilości obiektów sferycznych, aby zmaksymalizować czas procesora wykorzystywanego przez silnik fizyczny i tym w tym samym czasie zminimalizować zasoby zużywane na silnik graficzny.

Niestety, ale w przypadku tego testu, duża ilość obiektów kulistych spowodowała podobne koszty zasobów obliczeniowych zarówno dla silnika fizycznego jak i graficznego.

4.2.1.4 Test04



Rysunek 6: Scena Test04 - Ekran klienta



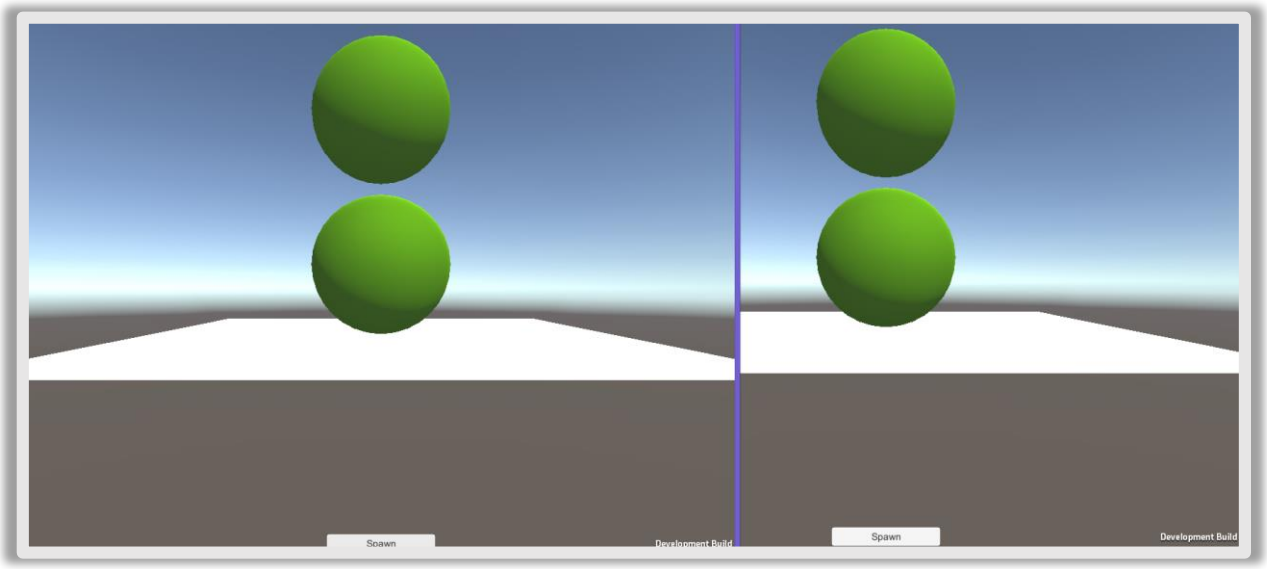
Rysunek 7: Scena Test04 - Ekran serwera

Pierwszy test modułów Transmission i Connectivity.

Scena testowa składa się z bardzo prostego interfejsu użytkownika, który umożliwia przesyłanie wiadomości tekstowych pomiędzy instancjami aplikacji.

Implementacja z użyciem stosu TCP/IP okazała się bardzo optymalnym rozwiązaniem. Połączenie oparte o adresy IP spowodowało, iż uruchomienie aplikacji w chmurze Azure (z użyciem usługi Virtual Machines) nie przysporzyło żadnego problemu.

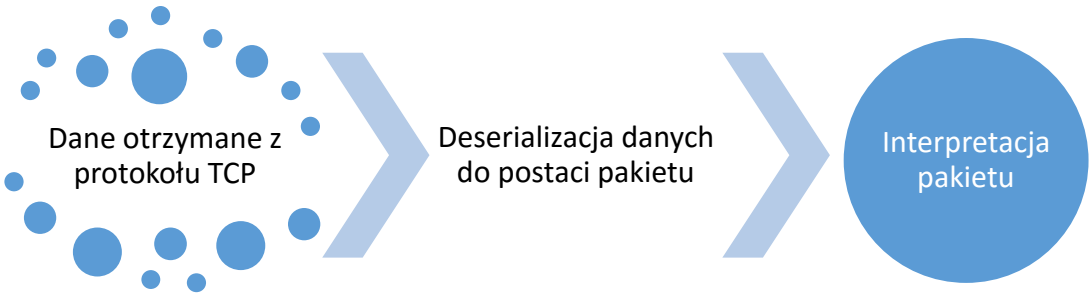
4.2.1.5 Test05



Rysunek 8: Scena Test05 - Ekrany serwera i klienta

Kolejny test modułów Transmission i Connectivity, tym razem wzbogaconych o moduły Serialization oraz Packets.

Test mający na celu przetestowanie działania większości niskopoziomowych modułów w projekcie. Przykład powyżej przedstawia dwie instancje aplikacji: serwera oraz klienta uruchomionych na jednym komputerze. Scena ilustruje synchronizację dodawania obiektów do sceny.



Rysunek 9: Diagram procesu otrzymywania danych

Rysunek 10: Diagram procesu wysyłania danych

Na powyższych diagramach widzimy procesy obsługujące synchronizację zdarzeń w aplikacji. Wersja zarówno kliencka jak i serwerowa programu mają stale aktywne oba procesy przetwarzania informacji. Jednostką informacji w tym przypadku jest pakiet (moduł Packets). Pakiet tworzony jest w trakcie pojawiania się nowych zdarzeń w aplikacji np. otrzymywania danych od użytkownika, zmiany pozycji obiektu czy pojawiania się nowego.

W tym teście zakres pakietów został ograniczony do jednego – „Spawn Packet”. Pakiet ten określa, w którym miejscu na scenie powinien pojawić się nowy obiekt oraz jakiego typu powinien on być.

Test przeszedł pomyślnie pokazując, że podstawowe moduły aplikacji poprawnie ze sobą współpracują oraz generują satysfakcjonujące wyniki czasowe – brak widocznych opóźnień po stronie klienta.

4.2.1.6 JengaTest

Rysunek 11: Scena JengaTest

Drugi test silnika fizycznego.

Tym razem pomysł na zastosowanie silnika został wzięty z prawdziwego życia. Została stworzona symulacja wieży Jenga[3]. Użytkownik może swobodnie poruszać kamerą w obrębie sceny oraz dodawać nowe wieże. Nowe obiekty dodawane są w tym samym miejscu, co poprzednie – powoduje to dużą ilość kolizji, co z kolei skutkuje zwiększoną pracą silnika fizycznego.

Test wypadł jednoznacznie. Przy wykorzystaniu dużej ilości kolizji oraz bardzo prostej bryły (prostopadłościan) można uzyskać wystarczająco dużą różnicę wykorzystywanych zasobów między silnikiem fizycznym a graficznym. Na powyższej ilustracji zobaczyć można iż około 7000 obiektów potrafi wywołać różnicę w stosunku 80 : 1 między silnikiem fizycznym a graficznym.

[3]W grze Jenga wykorzystywane są prostokątne klocki o wymiarach 1.5 × 2.5 × 7.5 cm do budowania pionowej wieży. Bloki umieszczane są obok siebie wzdłuż ich dłuższego boku, a prostopadle do poprzedniego poziomu. Każdy poziom wieży składa się z 3 klocków.

4.2.1.7 NebulaTest

Scena końcowa oraz gotowy produkt demonstracyjny[4].

Ostatnia faza projektu wykorzystuje poprzedni test z wieżą Jenga modyfikując go tak, aby całość obliczeń fizycznych była wykonywana po stronie serwera a klient jedynie renderował efekt obliczeń.

Stworzone rozwiązanie Nebula [5] umożliwia uzyskanie takiego samego efektu końcowego (renderowana gra) dla o wiele słabszych urządzeń niż wcześniej było możliwe. Słabsze urządzenia mogą skupić większość swojej mocy obliczeniowej na silniku graficznym, co powoduje wzrost jakości renderowanego obrazu nie tracąc przy tym poziomu skomplikowania symulacji fizycznych.

4.2.2 Skryptowanie

Wszystkie skrypty silnika Unity3D, stworzone na potrzeby tej pracy, zostały napisane z użyciem języka C#.

Pisanie skryptów dla silnika Unity3D nie różni się bardzo od standardowego programowania w języku C#. Praktycznie nie ma tutaj żadnych różnic w porównaniu ze standardowym kodem pisanym pod platformę .NET. Jedynym wymaganiem jest korzystanie z owej platformy w wersji nie wyższej niż 3.5.

Różnice zaczynają być widocznie, gdy chcemy gotowy kod wykorzystać w silniku. Jedyną opcją uruchomienia naszego kodu w środowisku Unity3D jest wykorzystanie klasy „MonoBehavior". Klasa ta definiuje podstawowe właściwości obiektu w silniku gry. Aby wykorzystać napisaną przez nas klasę wymagane jest dodanie relacji dziedziczenia pomiędzy klasą, która chcemy dodać do sceny a klasą „MonoBehaviour". Powoduje to możliwość dołączenia ów klasy w tzw: „Komponencie" do sceny.

Klasa „MonoBehaviour" definiuje listę metod i właściwości, z których klasa dziedzicząca może korzystać. Min. „Update" – metoda wywoływana przez silnik graficzny przy każdorazowym rysowaniu sceny, „FixedUpdate" – metoda wywoływana przez silnik

[4] Dostępny na nośniku załączonym z pracą
[5] Nazwa robocza

fizycznym przy każdorazowym przeliczaniu ciał fizycznych oraz wiele więcej. Wywoływanie własnego kodu jest możliwe tylko poprzez te metody.

API silnika Unity3D zachęca prostotą i łatwością wykorzystania, co umożliwia bardzo szybki postęp przy pracach w tym silniku. Jednakże wymóg bezpośredniego dziedziczenia z klasy „MonoBehaviour" stawia przed programistą duży problem – brak możliwości testowania jednostkowego. Klasy dziedziczące z „MonoBehaviour" nie mogą być utworzone poza silnikiem Unity3D. Wskazane jest, więc przenoszenie kodu do klas niezależnych (nieposiadających zależności od silnika Unity3D), aby umożliwić jak największą testowalność.

Najważniejszym skryptem w module silnika jest klasa PhysicsObject. Wykorzystuje on moduł niezależny TimedList, aby dostarczyć niezbędną funkcjonalność interpolacji czasowej dla wartości pozycji oraz obrotu dowolnego obiektu w silniku Unity3D. Skrypt ten wykorzystuje podstawowe algorytmy interpolacji liniowej. Wartości interpolowane są w skali od 0 do 1 oraz zwracane są zawsze dwa elementy.

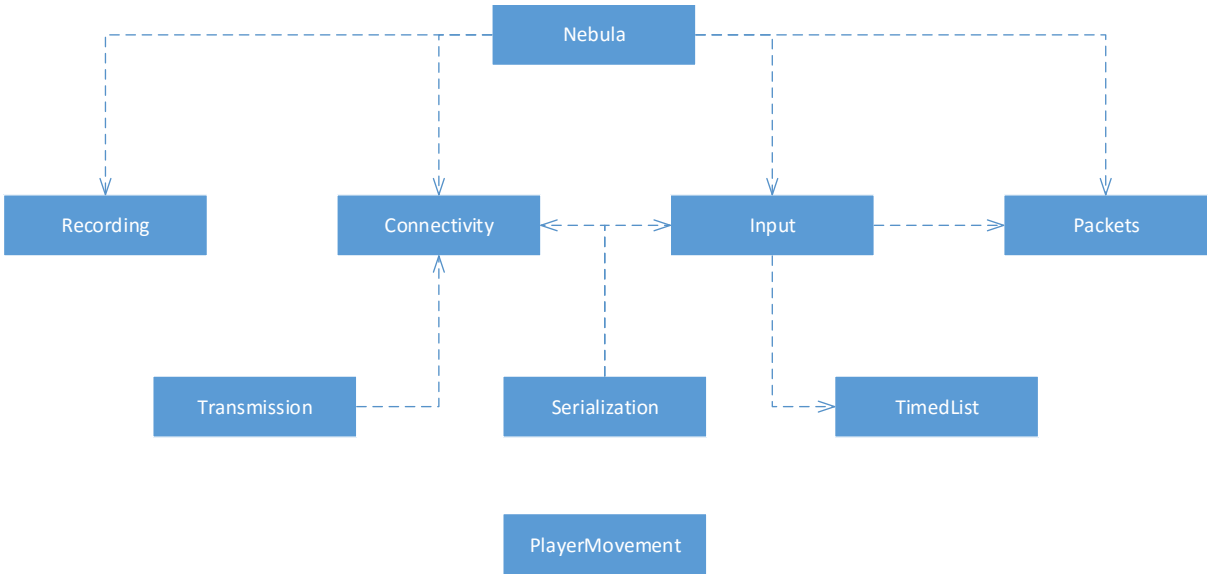
```
private TimedList<Vector3> GetNewVectorList()
{
        return new TimedList<Vector3>((vector3, value) =>
           new[] { vector3 * value, vector3 * (1 - value) });
}

private TimedList<Quaternion> GetNewQuaternionList()
{
        return new TimedList<Quaternion>((originalQuaternion, value) =>
           {
              var firstQuaternion = Quaternion.Lerp(new Quaternion(),
                      originalQuaternion, value);
              var secondQuaternion = originalQuaternion *
                      Quaternion.Inverse(firstQuaternion);
              return new[] { firstQuaternion, secondQuaternion };
            });
}
```

Fragment kodu 1: Funkcje podziału elementu czasowego

4.3 Moduł niezależny

Moduł niezależny składa się z kilkunastu bibliotek kodu C# podzielonych wedle konkretnych odpowiedzialności.



Rysunek 12: Diagram zależnościu modułu niezależnego

Poza zilustrowanymi powyżej bibliotekami każda z nich ma dedykowaną jej bibliotekę testów jednostkowych. Ponieważ żadna z bibliotek nie posiada zależności z silnikiem Unity3D możliwe było uzyskanie pokrycia kodu testami na poziomie 95%.

4.3.1 Nebula

Biblioteka wykorzystująca większość podmodułów niższego poziomu, aby dostarczyć jej użytkownikowi gotowe rozwiązanie programistyczne. Jej zadaniem jest połączenie wszystkich zależności w funkcjonalną całość.

4.3.2 Recording

Moduł zawierający klasy pomocnicze, których zadaniem jest uproszczenie dostępu do najważniejszych danych obiektu fizycznego.

Interfejs IGameObject deklaruje 4 podstawowe właściwości obiektu synchronizowanego pomiędzy instancjami aplikacji. Kolejno są to: pozycja, obrót, informacja czy obiekt jest wykluczony z obliczeń fizycznych oraz typ obiektu.

```
public interface IGameObject
{
	Vector3 Position { get; set; }
	Quaternion Rotation { get; set; }
	bool IsDestroyed { get; }
	string Type { get; }
}
```

Fragment kodu 2: Interfejs IGameObject

Biblioteka Recording zajmuje się również podstawowymi obliczeniami dla pozycji oraz obrotu. Udostępnione są informacje o różnicy w pozycji/obrocie względem poprzedniej wartości ów zmiennej. (Wykorzystywane po stronie serwera)

```
public virtual Vector3 GetPositionDiff()
{
	return _gameObject.Position - _lastPosition;
}

public virtual Quaternion GetRotationDiff()
{
	return Quaternion.Inverse(_lastRotation) * _gameObject.Rotation;
}
```

Fragment kodu 3: Funkcje zwracające różnice w pozycji/obrocie obiektu

Jak również działania modyfikujące dane zmienne. (Wykorzystywane po stronie klienta)

```
public void ApplyPositionDiff(Vector3 vector3)
{
	_gameObject.Position += vector3;
}

public void ApplyRotationDiff(Quaternion quaternion)
{
	_gameObject.Rotation *= quaternion;
}
```

Fragment kodu 4: Procedury zmiany wartości pozycji/obrotu obiektu

4.3.3 Connectivity

Biblioteka definiująca interfejsy/klasy abstrakcyjne przeznaczone do wysyłania/odbierania pakietów informacji między klientem a serwerem oraz serializacji/deserializacji ów pakietów.

Interfejs komunikacyjny został podzielony na 3 części.

```
public interface IProtocolConnectivity
{
	void Start();
	void Stop();
}

public interface IReciveTransmissionProtocol
{
	IEnumerable<string> GetPackets();
}

public interface ISendTransmissionProtocol
{
	void SendPacket(string packet);
}
```

Fragment kodu 5: Interfejsy sieciowe

Kolejno: protokół połączeniowy (IProtocolConnectivity), protokół odbierania (IReciveTransmissionProtocol) i protokół wysyłania (ISendTransmissionProtocol).

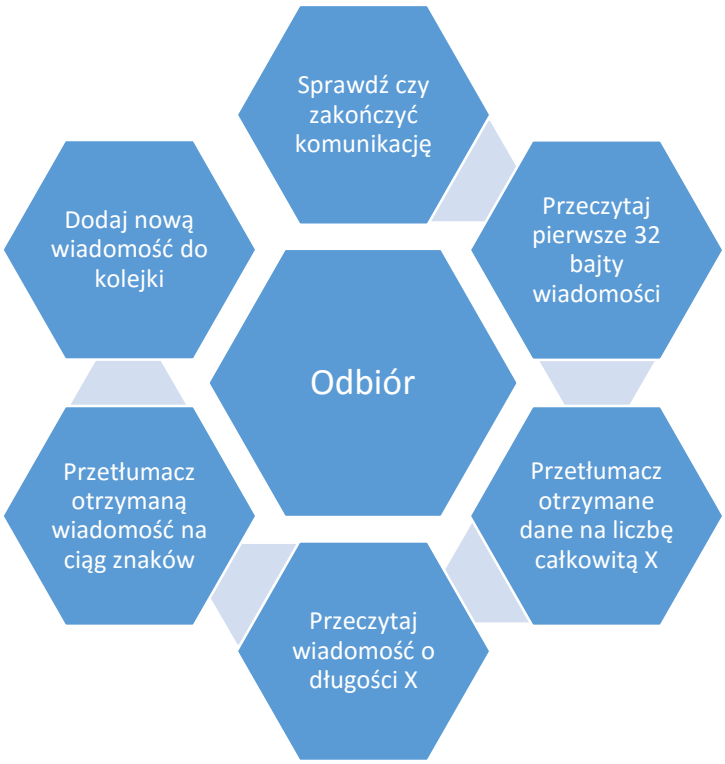
4.3.4 Transsmission

Moduł implementujący interfejsy połączeniowe biblioteki Connectivity z wykorzystaniem protokołu TCP.

Moduł Transsmission udostępnia dwie klasy implementujące połączenie sieciowe:

- TcpClientTransmissionProtocol – klasa wykorzystująca klienta TCP
- TcpListenerTransmissionProtocol – klasa wykorzystująca serwer nasłuchujący TCP

Obie klasy implementują wysyłanie oraz otrzymywanie wiadomości sieciowych za pomocą wątków oraz kolejek. Wątki obsługują asynchroniczne czytanie/wpisywanie danych do gniazdek sieciowych a kolejki udostępniają nowo otrzymane wiadomości oraz miejsce składowania wiadomości gotowych do wysłania.



Rysunek 13: Proces działania wątku odbierającego informacje z sieci

Rysunek 14: Proces działania wątku wysyłającego informacje do sieci

4.3.5 Input

Biblioteka wspierająca nagrywanie danych otrzymywanych od użytkownika.

Udostępniane są tutaj dwie struktury danych definiujące schemat danych pobieranych od użytkownika aplikacji.

```
public struct RecordedInput
{
	public readonly float Duration;
	public InputData[] Data;
}

public struct InputData
{
	public readonly float Value;
	public readonly string Name;
}
```

Fragment kodu 6: Struktury danych użytkownika

Owe struktury są następnie wykorzystywane w klasie InputRecorder udostępniające gotowe mechanizmy zapisywania oraz odczytywania otrzymywanych danych.

4.3.6 TimedList

Jeden z najważniejszych modułów niskiego poziomu. Jego zadaniami są:

- Przetrzymywanie dowolnych wartości razem z informacją o czasie ich trwania
- Udostępnianie dostępu do zapisanych wartości zależnie od podanego czasu

Klasa TimedList wykorzystywana jest to uniezależniana otrzymywanych danych z silnika graficznego od ilości renderowanych klatek na sekundę. Dane zapisywane przez tą klasę są przetrzymywane w strukturze TimedElement (T jest dowolnym typem przetrzymywanej wartości).

```
public struct TimedElement<T>
{
	public T Value;
	public float Duration;
}
```

Fragment kodu 7: Element czasowy

Ponieważ otrzymywane dane od użytkownika pobierane są w czasie iteracji silnika graficznego a następnie odgrywane już w czasie iteracji silnika fizycznego, wymagane jest uniezależnienie otrzymywanych informacji do czasu obydwu silników.

Dodawanie elementów do czasowych (TimedElement) nie jest skomplikowane.

```
public void Add(T value, float duration)
{
      if(duration <= 0f)
         return;

      AddElement(value, duration);
}

private void AddElement(T value, float duration)
{
       _elements.Add(new TimedElement<T>(value,duration));
}
```

Fragment kodu 8: Dodawanie elementu czasowego

Prawdziwe problemy pojawiają się przy zwracaniu trzymanych elementów biorąc pod uwagę podawany czas.

```
public IEnumerable<TimedElement<T>> Take(float duration)
{
      if (duration <= 0f)
          return Enumerable.Empty<TimedElement<T>>();

      var toReturn = GetElements(duration);

      return toReturn;
}

private IEnumerable<TimedElement<T>> GetElements(float duration)
{
       int amount = 0;
       while (duration > 0f && amount < _elements.Count)
       {
          duration -= _elements[amount].Duration;
          amount++;
       }

       var toReturn = _elements.Take(amount).ToArray();

       if (LastElementMustBeModified(duration))
       {
          var indexToModify = toReturn.Length - 1;
          var originalDuration = toReturn[indexToModify].Duration;
          toReturn[indexToModify].Duration += duration;

          var splitPercent = duration / originalDuration;

          var splitResult = _splitFunc(toReturn[indexToModify].Value,
                                Math.Abs(splitPercent));
          toReturn[indexToModify].Value = splitResult[0];

          _elements[indexToModify] =
                 new TimedElement<T>(splitResult[1], -duration);

          amount--;
       }

       for (int i = 0; i < amount; i++)
          _elements.RemoveAt(0);

       return toReturn;
}

private bool LastElementMustBeModified(float duration)
{
       return duration < 0f;
}
```

Fragment kodu 9: Pobieranie elementów czasowych

W załączonym fragmencie kodu możemy zauważyć, iż najpierw ignorowane są elementy o czasie mniejszym niż 0. Następnie zwracana jest lista elementów czasowych zależna od podanego czasu.

Pobieranie listy elementów dla określonego czasu polega na obliczeniu ile przechowywanych do tej pory wartości ma w sumie wartość równą podanemu czasowi (zaokrąglając czas ostatniego elementu w dół). Następnie obliczona ilość elementów jest pobierana z listy przechowywującej wszystkie elementy. Kolejnym krokiem jest analiza ostatniego elementu z poprzednio wybranych. Sprawdzane jest czy musi zostać on podzielony. Jeżeli suma czasów wybranych elementów była zaokrąglana w dół to znaczy, że element musi zostać poddany interpolacji. Obliczany jest, zatem wymagany ułamek oryginalnego czasu potrzeby do zrównania całkowitego czasu wybranych elementów z podanym do funkcji czasem. Wyliczony ułamek jest stosowany do podzielenia ostatniego elementu czasowego na dwa. Wynikiem podziału jest zinterpolowany element czasowy domykający listę wybranych wcześniej elementów. Dodatkowo powstały element zinterpolowany o pozostały czas jest z powrotem dodawany do listy przechowywanych elementów.

4.3.7 Packets

Biblioteka zawierająca definicję podstawowych pakietów informacyjnych użytych w aplikacji.

```
public struct DestroyPacket : IPacket
{
	public Guid Id;
}

public struct MovePacket : IPacket
{
	public Guid Id;
	public Vector3 Move;
	public Quaternion Rotation;
	public float Duration;
}

public struct SpawnPacket : IPacket
{
	public Guid Id;
	public Vector3 Position;
	public Quaternion Rotation;
	public string Type;
}
```

Fragment kodu 10: Pakiety informacyjne

- DestroyPacket – informacja, iż obiekt nie jest już dłużej przeliczany przez silnik fizyczny
- MovePacket – informacja o zmianie pozycji oraz obrotu obiektu fizycznego
- SpawnPacket – informacja o nowopowstałym obiekcie fizycznym

4.3.8 Serialization

Moduł udostępniający klasy konwertujące typy wykorzystywane w projekcie na łańcuchy znakowe. Dostępnie serializatory/deserializatory umożliwają konwersję typów:

- DestroyPacket
- MovePacket
- SpawnPacket
- Vector3
- Quaternion
- Guid
- RecordedInput

4.3.9 PlayerMovement

Biblioteka pomocnicza udostępniająca klasy do poruszania obiektami w silniku Unity3D.

Rozdział 5
Podsumowanie

Utworzona aplikacja z powodzeniem demonstruje możliwości wspierania istniejących rozwiązań (Unity3D) w dziedzinie gier komputerowych o nowoczesne technologie chmurowe (Azure). Gotowe rozwiązanie umożliwia zrównanie możliwości obliczeniowych (w aspekcie symulacji fizycznych) pomiędzy platformami mobilną a stacjonarną. Dodatkowo wspierane przez platformę Microsoft Azure gwarantuje skalowalność oraz niezawodność.

Unity3D, jako silnik gier udowodnił, że z powodzeniem może stanowić solidną bazę projektów zarówno graficznych jak i fizycznych. Możliwość zastosowania go, jako serwer bez funkcjonalności graficznych jest dodatkowym atutem – szczególnie biorąc pod uwagę jego multiplatformowość.

Microsoft Azure umożliwił bezbolesne przetestowanie aplikacji w środowisku chmurowym. Przy pomocy wirtualnych maszyn hostowanych na serwerach Azure testy funkcjonalne aplikacji przebiegały bardzo szybko i bezproblemowo. Sama aplikacji, mimo iż niezależna od konkretnego usługodawcy chmurowego, mogłaby z powodzeniem skorzystać z szeregu dodatkowych funkcjonalności udostępnianych przez Microsoft Azure – szczególnie w dalszych aspektach jej rozwoju. Analiza danych graczy czy automatyczne skalowanie serwerów to tylko wierzchołek góry lodowej.

Bardzo ważnym aspektem projektu okazało się wydzielenie kodu niezależnego od silnika Unity3D. Pozwoliło to dynamiczny rozwój najważniejszych modułów aplikacji. Testy jednostkowe zaoszczędziły mnóstwo czasu, który inaczej byłby stracony na wielogodzinnych sesjach debugowania się przez silnik Unity3D.

Projekt pozostaje otwarty na rozszerzanie o nowe funkcjonalności czy usprawnienia. Architektura aplikacji pozwala na swobodną wymianę komponentów oraz dodawanie nowych. Przykładem może być wymiana protokołu TCP na UDP czy dodanie nowych pakietów informacyjnych. Wykorzystując możliwości platformy Azure w aspekcie przechowywania ogromnych ilości danych możliwe było by zapisywanie całych sesji symulacyjnych a następne odtwarzanie ich na żądanie.

Symulacje fizyczne to oczywiście nie jedyna ścieżka rozwoju usług chmurowych w grach komputerowych. Istnieją już gotowe rozwiązania w dziedzinach analizy danych z gier wieloosobowych, rozproszonej sztucznej inteligencji czy przewidywania ruchów i trendów.

Bibliografia

[1] Joseph Albahari, Ben Albahari, C# 5.0 in a Nutshell: The Definitive Reference

[2] Robert. C. Martin, Czysty kod

[3] Robert. C. Martin, Mistrz czystego kodu

[4] Unity Technologies, Unity Scripting Reference, http://docs.unity3d.com/ScriptReference/

[5] Networked Physics, http://gafferongames.com/networked-physics

[6] Game Networking, http://gafferongames.com/networking-for-game-programmers

[7] Microsoft, Microsoft API and reference catalog, https://msdn.microsoft.com/en-us/library/ms123401.aspx

[8] Git Reference, https://git-scm.com/docs

Indeks ilustracji

Indeks fragemntów kodu