Złożenie pracy online:
2012-02-06 13:22:47
Kod pracy:
6818

Łukasz Raszyk
(nr albumu: 17921*INF/INŻ)

Praca inżynierska

Vego 3D – projekt gry społecznościowej 3D w środowisku przeglądarki internetowej

Vego 3D – 3D social game project in a web browser environment

Wydział: Informatyki

Kierunek: Informatyka

Specjalność: technologie multimedialne

Promotor: dr Franciszek Białas

Nowy Sącz, 2012

SPIS TREŚCI

1 WPROWADZENIE

Jak łatwo zauważyć, ostatnie lata w branży internetowej stoją głównie pod znakiem wszelkiego rodzaju serwisów internetowych gromadzących wokół siebie społeczności, blogów informacyjnych i opiniotwórczych, mikroblogów oraz serwisów służących dzieleniu się swoją pasją i twórczością artystyczną. Popularność tego typu środków wymiany informacji świadczy jednoznacznie o czysto ludzkiej potrzebie nawiązywania kontaktów, szukania interesujących znajomości oraz miłego spędzania czasu w internecie. Za początek internetowej komunikacji można uznać serwisy czatowe, IRC oraz fora dyskusyjne, które wraz z biegiem czasu ewoluowały do większych, bardziej otwartych i przystępnych platform. Obecnie mamy do czynienia z rynkiem wypełnionym dziesiątkami większych lub mniejszych portali społecznościowych, o których sukcesie decyduje głównie poziom interakcji użytkownika z serwisem oraz interakcji pomiędzy samymi użytkownikami za pomocą tego serwisu.

Statystyki pokazują, że niezmiennie od lat równie dużym zainteresowaniem cieszą się gry internetowe/przeglądarkowe, które często stanowią miłą odskocznię od codziennych zajęć i pozwalają na interaktywną zabawę w internecie. Wraz ze wzrostem popularności serwisów społecznościowych, gry internetowe przejęły ich cechy i część funkcjonalności, pozwalając użytkownikom na wspólną zabawę w kręgu znajomych. Wiele z tych gier nawet „żyje" w ścisłej symbiozie z tymi serwisami, współdzieląc platformę i bazę użytkowników. Większość z tych gier (typu tzw. „casual") jest jednak mocno ograniczona funkcjonalnie i oferuje dość prostą rozgrywkę oraz niezbyt zaawansowaną technologicznie stronę wizualną – najczęściej korzystają z platformy Adobe Flash, przez co są zamknięte w dwóch wymiarach. Tego typu gry są skierowane do jak największej grupy odbiorców o niewygórowanych oczekiwaniach.

Mówiąc o ograniczeniach gier przeglądarkowych trudno uniknąć porównań z „pełnoprawnymi" grami wideo (komputerowymi i konsolowymi), które tworzą odrębny rynek, skierowany do bardziej wymagających graczy. Są to produkty kładące nacisk nie tylko na interesującą i wciągającą rozgrywkę, ale także na wrażenia audiowizualne – zaawansowaną technologicznie grafikę 3D, wysoki poziom artyzmu grafiki i animacji oraz bogactwo dźwięku. Wśród tych gier również można znaleźć tytuły nastawione na rozgrywkę sieciową, lecz próżno w nich jeszcze szukać cech społecznościowych, którymi mogą odznaczyć się gry przeglądarkowe. Wyjątek mogą stanowić jedynie gry typu *Massive Multiplayer Online*, które pozwalają na eksplorację ogromnych, wirtualnych światów wykonywanie zadań wspólnie z przyjaciółmi, tworzenie i rozwijanie swoich awatarów (wirtualnych reprezentacji użytkowników).

Kluczem do odniesienia sukcesu w każdej branży, nie tylko internetowej, jest innowacyjność, stosowanie najnowszych technologii oraz podążanie za trendami i potrzebami konsumentów. Celem poniższej pracy była próba wypracowania koncepcji produktu kierującego się powyższą myślą oraz zrealizowanie projektu, łączącego wszystkie najlepsze cechy wspomnianych wcześniej usług internetowych i produktów. W efekcie powstało połączenie portalu społecznościowego oraz gry sieciowej o cechach i jakości topowej gry wideo, czyli w skrócie gra społecznościowa - Vego 3D.

2 OPIS PROJEKTU

2.1 KONCEPCJA PROJEKTU

Vego 3D (nazwa pochodzi od wyrażenia *virtual ego*, czyli wirtualne alter-ego) z założenia miało stanowić platformę komunikacyjną i rozrywkową przeznaczoną do wirtualnych spotkań w wirtualnym świecie z możliwością nawiązywania kontaktu z innymi napotkanymi użytkownikami oraz eksplorację świata, stworzonego na potrzeby projektu. Platforma posiadałaby swoją mechanikę świata oraz zasady uczestnictwa, przez co można byłoby nazwać ją grą. Priorytetem Vego 3D była prostota, łatwość i szybkość dostępu do gry, pozwalająca na dotarcie do odbiorców z każdej grupy wiekowej, czerpanie maksymalnej satysfakcji z użytkowania platformy oraz zyskanie jak największej liczby użytkowników, których obecność również przyczynia się do atrakcyjności i sensu samej rozgrywki.

Każdy użytkownik, trafiający na stronę internetową projektu, miałby szansę darmowej rejestracji oraz utworzenia swojego awatara, czyli wirtualnej reprezentacji siebie w świecie gry. Jednym z celów gry byłby rozwój tego awatara, zaopatrywanie go w nowsze, bardziej modne i oryginalne ubrania. Dodatkowo, każdy uczestnik otrzymywałby podczas rejestracji swoje pierwsze mieszkanie, które później mógłby umeblować i urządzić wedle własnego uznania. Po zakończonym procesie rejestracji (oraz później w każdym innym, dowolnym momencie), użytkownik miałby do wyboru Grę lub Sklep. Na początku przygody z Vego 3D, każdy uczestnik dysponowałby startową liczbą kredytów, które stanowiłyby wirtualną walutę w świecie Vego 3D. Użytkownik posiadający dodatnią liczbę kredytów miałby możliwość wydania ich w Sklepie, kupując dla swojego awatara nowe ubrania lub nowe meble do swojego wirtualnego mieszkania. Sklep pozwalałby na przeglądanie przedmiotów z podziałem na kategorię oraz oferowałby ich podgląd przed zakupem, zarówno ubrań, jak i mebli. Po zakupie ubrań istniałaby możliwość ich założenia na swojego awatara, zarówno w sklepie, jak i w grze. Natomiast zakupione meble byłyby dostępne w odpowiednim panelu po wejściu do swojego apartamentu w grze. Użytkownik za pomocą posiadanych mebli miałby szansę urządzenia swojego mieszkania we własnym stylu, tworząc je oryginalnym i niepowtarzalnym. Takie małe kreowanie własnego „świata" byłoby nie tylko dobrą zabawą, ale także stwarzałoby możliwość pochwalenia się swoim gustem przed innymi użytkownikami, po zaproszeniu ich do swojego apartamentu – cecha społecznościowa.

Po wejściu do samej gry, użytkownik trafiałby do swojego apartamentu, który stanowiłby bazę do dalszych przygód. Stamtąd, mógłby się przemieszczać do innych lokacji, innych części świata, w których poruszają się pozostali użytkownicy. Przemieszczanie się po świecie, można byłoby rozwiązać poprzez menu z listą, mapę lokacji oraz portale. Napotykając na innych awatarów podczas eksploracji, użytkownik mógłby nawiązać z nimi kontakt. Wspólna komunikacja odbywałaby się na zasadzie czatu globalnego oraz czatów prywatnych prowadzonych pomiędzy dwiema osobami. Dodatkową atrakcją byłaby również możliwość interakcji ze światem poprzez zastosowanie interaktywnych, „klikalnych" obszarów, pozwalających na wykonywanie dodatkowych czynności, np. siadanie na ławce w parku lub w fotelu we własnym mieszkaniu.

2.2 SZCZEGÓŁOWY OPIS FUNKCJONALNOŚCI

2.2.1 SERWIS WWW

Użytkownik po pierwszym wejściu na stronę Vego 3D widzi poniższy ekran, który pozwala na rejestrację lub zalogowanie się do gry. Oba formularze posiadają walidację wprowadzanych danych. Formularz szybkiej rejestracji, po wprowadzeniu podstawowych danych, przenosi do dalszej części rejestracji, gdzie użytkownik zobowiązany jest do podania kolejnych danych, m.in. hasła do swojego konta.

Rys. 2.1 Widok strony głównej serwisu przed zalogowaniem *źródło: opracowanie własne*

Po zarejestrowaniu się lub zalogowaniu, w miejscu formularza logowania pojawia się nazwa zalogowanego użytkownika, jego obrazek oraz liczba kredytów, jakie ma aktualnie na koncie. Początkowa liczba kredytów wynosi 10000. W menu dostępne są odnośniki do dwóch głównych modułów strony – Gry (Game) i Sklepu (Shop) – które są niewidoczne, gdy użytkownik nie jest zalogowany.

Rys. 2.2 Widok strony głównej serwisu po zalogowaniu *źródło: opracowanie własne*

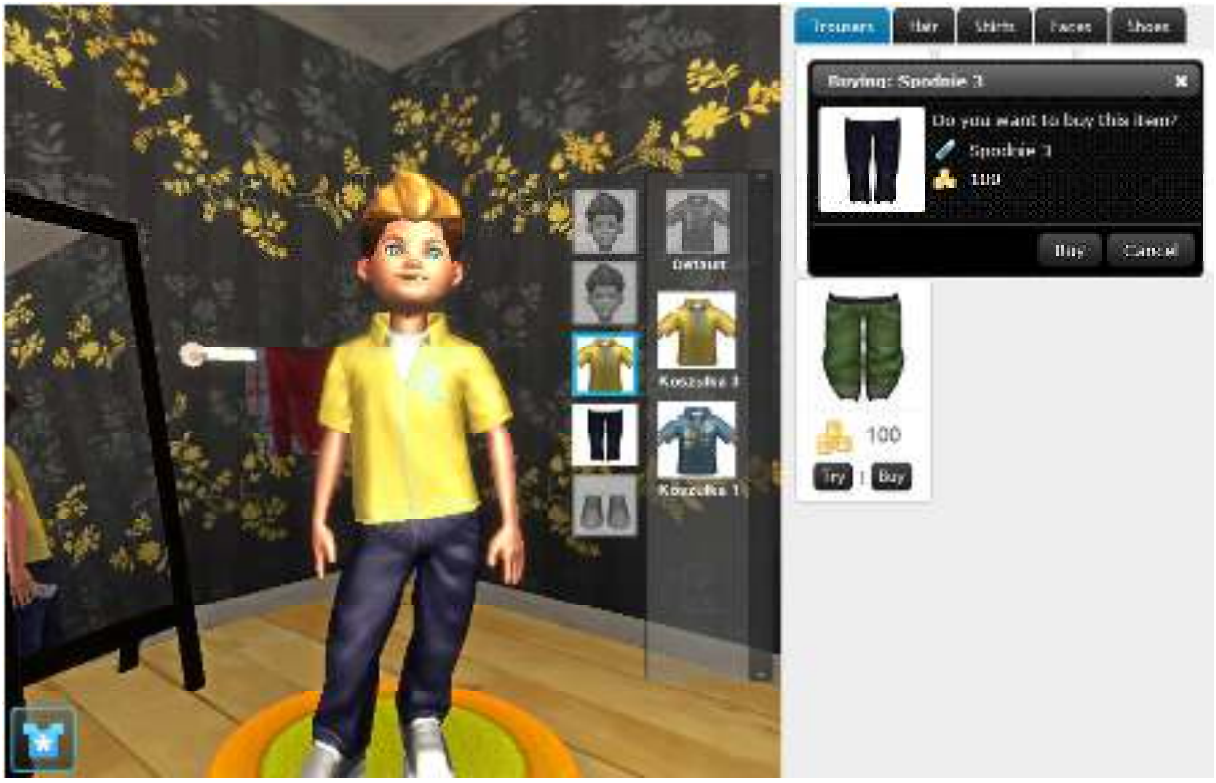
Wylogowanie się z serwisu zamyka wszystkie otwarte i aktywne akcje (w tym kończy obecność w wirtualnym świecie gry) i przenosi użytkownika do ekranu początkowego, pozwalając mu na późniejsze zalogowanie lub zalogowanie innemu użytkownikowi.

2.2.2 *SKLEP GRY*

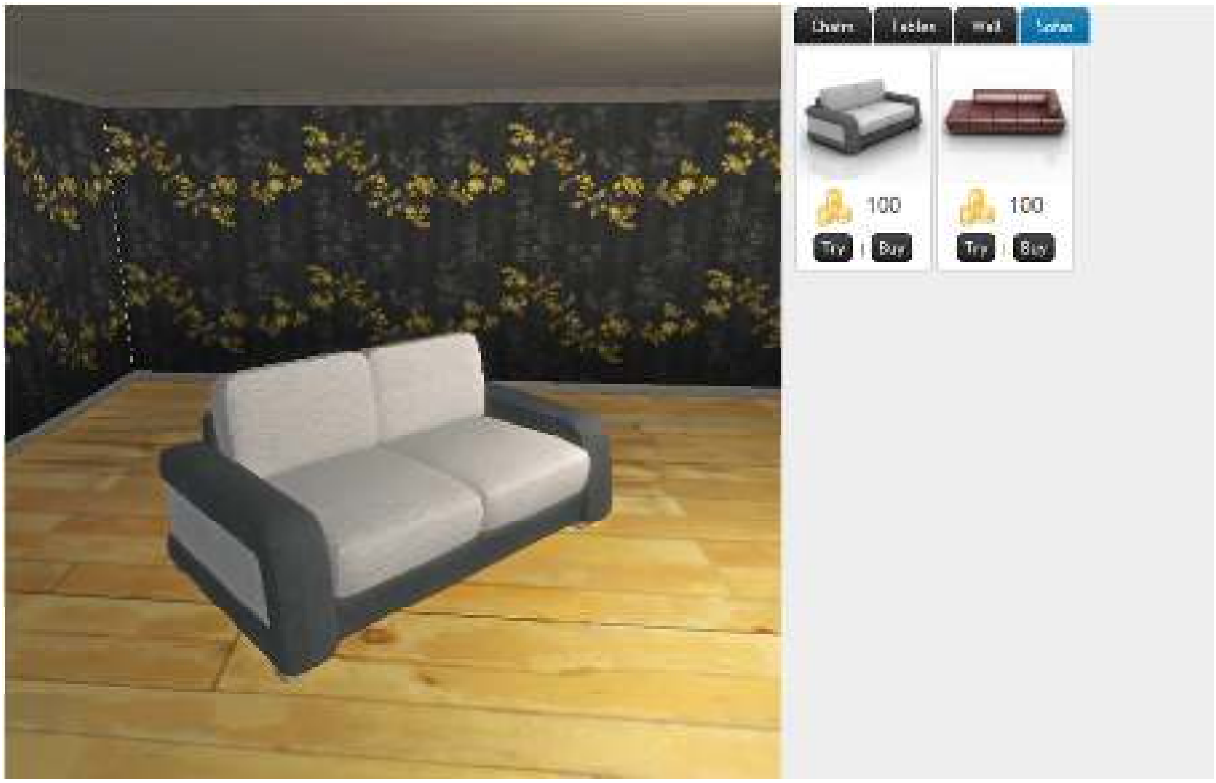
Gdy użytkownik wybierze Sklep, otwiera się widok (na poniższym zrzucie ekranu), który składa się przede wszystkim z dwóch części – podglądu 3D oraz listy przedmiotów. Domyślną kategorią sklepu są ubrania, lecz istnieje możliwość przełączenia kategorii, np. na meble. Podczas zmiany kategorii sklepu, przełączany jest także widok kamery w podglądzie 3D. W prawej kolumnie, nad listą przedmiotów, znajduje się seria zakładek, które odzwierciedlają kategorie przedmiotów, które są aktualnie przeglądane – domyślnie ubrań. Każdy przedmiot widoczny na liście składa się z miniaturki, ceny oraz posiada dwa przyciski – Try (Przymierz) oraz Buy (Kup).

Rys. 2.3 Widok sklepu *źródło: opracowanie własne*

Po kliknięciu na Przymierz, w podglądzie 3D ładowany i wyświetlany jest model danego przedmiotu. W przypadku ubrania, znajduje się ono w odpowiednim miejscu na awatarze użytkownika – na danym slocie, zamiennie z ubraniem, które wcześniej miał założone awatar. Istotnym elementem interfejsu nie tylko sklepu, ale także samej gry jest Szafa, która odzwierciedla listę ubrań, które posiada użytkownik oraz listę ubrań, które aktualnie ma założone awatar. Podział posiadanych ubrań w szafie został zrealizowany na podstawie slotów, do których przynależą ubrania, np. spodnie wybierane są na slocie o symbolu nóg/spodni. Z kolei przycisk Kup pokazuje okienko modalne służące do potwierdzenia zakupu, które wyświetla także dodatkowe szczegóły przedmiotu.



Rys. 2.4 Podgląd szafy użytkownika oraz okienko kupowania przedmiotu *źródło: opracowanie własne*



Rys. 2.5 Widok sklepu z aktywną kategorią mebli oraz podglądem modelu *źródło: opracowanie własne*

2.2.3 KLIENT GRY

Gdy użytkownik dokona już pierwszych zakupów w Sklepie, może wejść do wirtualnego świata Vego 3D poprzez naciśnięcie zakładki Gra. Pierwszą lokacją, w której pojawia się awatar użytkownika jest jego apartament, który otrzymuje za darmo po utworzeniu konta. Apartament nie tylko stanowi bazę do dalszych przygód w świecie gry, ale również zapewnia dodatkową rozrywkę poprzez urządzanie go według własnych upodobań za pomocą mebli dostępnych w Sklepie. Aktywacja trybu edycji mieszkania następuje po naciśnięciu przycisku z ikoną domku. Powyżej tego przycisku znajduje się również przycisk aktywujący podgląd szafy.

W dolnym lewym rogu okna gry, obok wspomnianych przycisków, które stanowią jedne z głównych celów i akcji w grze, widoczne jest okno czatu. Czat jest podzielony na dwa rodzaje: globalny i prywatny. Globalny czat przypisany jest do lokacji, w której aktualnie znajduje się użytkownik i tylko w jej obrębie może prowadzić komunikację z napotkanymi awatarami. Czat globalny jest oznaczony za pomocą ikony globu, który stanowi również zakładkę aktywującą czat. Obok niej pojawiają się zakładki czatów prywatnych, które są inicjowane poprzez wzajemne zapraszanie się użytkowników do wspólnych konwersacji.

W górnej części okna gry znajdują przyciski nawigacyjne, pozwalające na wybranie innych lokacji, do których chce przenieść się użytkownik (w tym do innych swoich apartamentów). Obok widoczny jest suwak pozwalający na dynamiczną zmianę pory dnia.

Rys. 2.6 Główny widok klienta gry *źródło: opracowanie własne*

Będąc w swoim apartamencie, użytkownik może uruchomić tryb edycji mieszkania, który pozwoli na jego urządzenie, wykorzystując meble, które użytkownik wcześniej zakupił w Sklepie. Po kliknięciu na przycisk konfiguratora, pojawia się lista mebli z podziałem na kategorię. Po kliknięciu i przeciągnięciu miniaturki danego mebla, jego model pojawia się w apartamencie, w miejscu upuszczenia miniaturki. Następnie mebel można ustawić w dowolnym miejscu, obrócić lub zduplikować oraz usunąć. W trybie edycji można zaznaczać również inne, dotychczas znajdujące się w apartamencie meble. Po kliknięciu na nie także są kontrolki edycji – widoczne elementy interfejsu, umożliwiające modyfikacje pozycji, obrotu itp. Mebel przesuwa się klikając na niego i przeciągając kursorem myszy w inne miejsce. Obracać natomiast można za pomocą okręgu otaczającego model, klikając i przeciągając kursorem.

Rys. 2.7 Widok edycji apartamentu użytkownika *źródło: opracowanie własne*

Po ponownym kliknięciu na przycisk konfiguratora, następuje jego dezaktywacja. Edycja wszystkich mebli zostaje zablokowana, uniemożliwiając ich przypadkowe przesunięcie lub obrót. Tym samym możliwa staje się interakcja awatara użytkownika z meblem. Podchodząc do danego mebla, np. fotelu i klikając na miejsce, które potencjalnie może być miejscem siedzącym, awatar użytkownika siada w tym miejscu (pod warunkiem, że jest wolne). Następuje zablokowanie sterowania awatarem, a jego odblokowanie i wstanie z miejsca siedzącego staje się możliwe dopiero po ponownym kliknięciu na mebel. Miejsc siedzących w ramach jednego mebla może być więcej (np. kanapa), dlatego równocześnie miejsce może zająć większa liczba użytkowników niż tylko jeden gracz.

Rys. 2.8 Widok interakcji użytkownika z obiektem mebla *źródło: opracowanie własne*

Użytkownik znajdując się w grze, w swoim apartamencie, posiada możliwość przeniesienia się do innych lokacji, posługując się menu dostępnym pod przyciskiem w górnej części okna gry. Po wybraniu z listy innej lokacji, rozpoczyna się jej ładowanie, a następnie awatar użytkownika pojawia się w miejscu startowym w wybranej lokacji. Tu rozpoczyna się jego przygoda w świecie Vego 3D – może zwiedzać lokacje i wchodzić z nimi w interakcję oraz prowadzić komunikację z innymi użytkownikami, których napotka na swej drodze.

Użytkownik, widząc innego awatara, może na niego kliknąć w celu uruchomienia dowolnej akcji. Pojawia się menu postaci, na którym widoczna jest nazwa i obrazek użytkownika oraz interakcje, jakie można wykonać na danym użytkowniku. Dostępne są dwie akcje: zaproszenie do czatu oraz zaproszenie do apartamentu. Wybranie zaproszenia do czatu, powoduje wysłanie do danego użytkownika komunikatu z pytaniem o rozpoczęcie konwersacji. W przypadku akceptacji zaproszenia u obu użytkowników otwiera się nowa zakładka prywatnej rozmowy w oknie czatu. Na zakładce widnieje nazwa użytkownika, z którym prowadzona jest rozmowa a jej kolor oznacza aktywność danego okna czatu. Przełączając pomiędzy zakładkami (w tym globem, który oznacza czat globalny) można aktywować okno innego czatu. Zaproszenie do apartamentu z kolei wysyła komunikat do danego użytkownika z propozycją przeniesienia się do wybranego mieszkania, które następuje po akceptacji zaproszenia.

Rys. 2.9 Widok menu awatara oraz okna prywatnego czatu *źródło: opracowanie własne*

Wszystkie akcje, które użytkownik wykonuje swoim awatarem i które potrafi zaobserwować, czy to zmiana pozycji, obrót, animacja, czy interakcja ze środowiskiem, są widoczne także po stronie innych użytkowników. Dzięki temu inni gracze mogą zobaczyć w tym samym momencie, jak awatar innego użytkownika wchodzi do lokacji, wskakuje na podest lub siada na ławce, zajmując tym samym miejsce, które staje się niedostępne dla innych awatarów. Zmiana ubioru awatara, widoczna dla użytkownika-właściciela, jest także widoczna dla innych użytkowników. Daje to możliwość zaprezentowania innym graczom posiadanych ubrań, pochwalenie się nimi, zmianę ich w zależności od sytuacji. Także konfiguracja apartamentu nie tylko daje efekt po stronie jego właściciela, ale również po stronie wszystkich innych gości znajdujących się w tym mieszkaniu, którzy zostali do niego wcześniej zaproszeni. Jeśli użytkowników-gości jest w apartamencie więcej, niż przewiduje to liczba miejsc siedzących, właściciel może w czasie rzeczywistym dołożyć kolejne krzesła, sofy itp. Każda aktywność użytkownika oraz jej brak jest odpowiednio oznaczana m.in. poprzez symbole statusów, które znajdują się nad głowami awatarów. Dzięki temu, od razu staje się jasne, czy użytkownik w danej chwili, rozmawia, przebiera się, czy odszedł na chwilę od komputera. To wszystko powoduje, że wirtualny świat staje się żywy, a gracze, którzy go tworzą, łączą się w wirtualną społeczność. Płynność komunikacji i możliwość interakcji użytkowników ze sobą i środowiskiem sprawia, że Vego 3D staje się drugim, alternatywnym, wirtualnym życiem.

2.3 ZAŁOŻENIA TECHNICZNE

Vego 3D jako gra internetowa była projektowana z myślą o działaniu wyłącznie pod przeglądarką internetową w celu minimalizacji czynności, jakie należy wykonać do momentu wejścia do wirtualnego świata. Priorytetem był łatwy i szybki dostęp użytkownika do wszystkich funkcji gry. W efekcie aplikacja kliencka została zamknięta w kontener obsługiwany przez odpowiednią wtyczkę i osadzona na stronie internetowej. Tym sposobem użytkownik nie zostaje zmuszony do instalacji dodatkowego oprogramowania, ponieważ całość gry dostępna jest z pozycji przeglądarki. Dodatkowo, treść gry dostarczana jest użytkownikowi na bieżąco i pobrane z serwera zostają te dane i te elementy, które użytkownik potrzebuje w danej chwili. Dzięki temu mechanizmowi Vego 3D uzyskało przewagę nad grami sieciowymi, wymagającymi pobrania całości gry w postaci instalatora i zainstalowania aplikacji na dysku twardym. Nie dzieje się tak jednak za każdym razem, kiedy użytkownik otworzy grę, ponieważ dane załadowane wcześniej nie są pobierane ponownie, lecz są przetrzymywane w pamięci tymczasowej przeglądarki. Ponadto, dzięki temu, że gra dostępna jest ze strony internetowej i odtwarzana za pomocą wtyczki, możliwe jest jej uruchomienie na dowolnym komputerze, ponieważ gra (w przeciwieństwie do instalowanych gier) nie jest uzależniona od treści zainstalowanych na dysku twardym. Dzięki temu, użytkownik rozpoczynający przygodę w wirtualnym świecie Vego 3D na swoim komputerze osobistym, może ją kontynuować na innych komputerach, w domu, w pracy, w szkole.

Jedną z bardziej istotnych cech i zalet przyjętej koncepcji technicznej, jest powiązanie klienta gry z serwisem WWW, w którym ten klient został osadzony. Taki zabieg daje możliwość przerzucenia części funkcjonalności, głównie związanej z interfejsem, do środowiska, które dysponuje rozbudowanymi formularzami oraz wyświetla ładnie sformatowane wyniki zapytań z bazy danych. Dodatkową cechą tego rozwiązania jest możliwość zastosowania (choć zależna od wtyczki obsługującej klienta gry) komunikacji w czasie rzeczywistym klienta gry ze stroną za pomocą JavaScript, co pozwala na przesyłanie danych oraz zdalne wywoływanie procedur. Za pomocą JavaScript rozwiązana jest także cała nawigacja w serwisie. Czyni to całość maksymalnie interaktywną i szybką w działaniu – bez potrzeby odświeżenia strony. Najlepszy przykład może stanowić wirtualny sklep, w którym użytkownicy kupują przedmioty dla swoich awatarów. Przeglądanie i kupowanie przedmiotów powinno być szybkie, łatwe i dawać możliwość podglądu wszystkiego przed zakupem. W tym celu strona sklepu, która wyświetla z bazy danych listę przedmiotów, powinna komunikować się z podglądem 3D (obsługiwanym przez tę samą wtyczkę co klient gry) w celu pokazania modelu danego przedmiotu, np. przymierzanego ubrania na awatarze.

Kolejnym kluczowym aspektem budowy struktury całej aplikacji jest sposób komunikacji klienta gry z oprogramowaniem serwerowym. Tyczy się to zarówno komunikacji klient – klient, jak i klient – baza danych. Pierwszy typ komunikacji służy głównie do maksymalnie szybkiego transferu pakietów od klienta do klienta, zawierających informacje o położeniu, obrocie, animacji oraz stanie danego awatara w danej lokacji, czyli informacji odświeżanych z dużą częstotliwością. Drugi typ komunikacji służy do pobierania z bazy danych informacji m.in. o konfiguracji ubrań awatara lub konfiguracji mebli w apartamencie. Do rozwiązania obu przypadków najlepiej zastosować serwer gniazd oraz usługi sieciowe.

2.4 TECHNOLOGIE I NARZĘDZIA

Wybór technologii użytych w projekcie nie był przypadkowy i stanowił nie lada wyzwanie, gdyż od właściwego ich doboru i możliwości ścisłej współpracy między sobą zależała skuteczność i niezawodność działania całego projektu jako jednej aplikacji. Po wielu poszukiwaniach i eksperymentach udało się utworzyć właściwą kombinację technologii, której podstawę stanowi głównie platforma .NET. Na jej bazie powstał serwis WWW, baza danych i usługi sieciowe, a język C# oraz biblioteki współdzieli również klient gry oraz oprogramowanie serwerowe do komunikacji w czasie rzeczywistym.

2.4.1 PLATFORMA WWW

Do zbudowania serwisu WWW posłużyła technologia Microsoft ASP .NET, która kompleksowo odpowiedziała na potrzeby projektu, dostarczając bazę danych oraz platformę do budowy aplikacji internetowych opartą o wzorzec projektowy *Model-View-Controller* – MVC Framework w wersji 3. Przewagą tego rozwiązania nad innymi, opartymi w głównej mierze na PHP, jest szybkość i przyjazność procesu tworzenia. Dzięki architekturze stworzonej do budowy złożonych aplikacji i zawierającej wiele gotowych mechanizmów i komponentów, programista jest zwolniony z obowiązku pisania własnych implementacji silników sterujących aplikacją, co pozwala skupić się na właściwej logice aplikacji. Dotyczy to głównie elementów infrastrukturalnych, m.in. mechanizmów zarządzania stanem aplikacji oraz mechanizmów autoryzacji. Dodatkowo, przyjęty model programistyczny pozwala na zachowanie pełnej obiektowości w projekcie, do którego pisania skorzystano z języka C#. Dzięki silnikowi widoków można łatwo umieszczać właściwe mechanizmy aplikacji, często związane z obsługą bazy danych, bezpośrednio w kodzie HTML.

Jako oprogramowanie bazy danych użyto Microsoft SQL Server, który ściśle współpracuje z technologią ASP .NET, co pozwala na łatwą konfigurację i łatwy dostęp do bazy z poziomu samej aplikacji, dzięki wbudowanym mechanizmom. W celu większej kontroli nad bazą danych oraz jej strukturą, skorzystano z metody *Code First*, służącej do obiektowej budowy struktury bazy z poziomu kodu aplikacji. Budowa oraz inicjalizacja bazy odbywa się podczas pierwszego uruchomienia aplikacji oraz podczas kolejnych zmian w strukturze, co eliminuje konieczność bezpośredniego zarządzania nią z zewnątrz. Pozwala to także na korzystanie ze struktury bazy (w postaci modelu) podczas pisania mechanizmów pobierania i zapisu danych w bazie z poziomu aplikacji oraz podczas pisania usług sieciowych. Usługi sieciowe z kolei, które również są częścią całej platformy, zbudowane są na bazie *Windows Communication Foundation* i służą do obsługi bazy danych za pomocą zdalnych metod, które mogą być wywoływane nie z samej aplikacji WWW, ale także innych, zewnętrznych usług. W przypadku Vego 3D, z usług sieciowych skorzystano podczas komunikacji klienta gry z bazą danych. Usługi sieciowe na bieżąco dostarczają klientowi danych o profilach użytkowników, ich konfiguracji, posiadanych przedmiotach, w celu właściwego wyświetlenia ich reprezentacji w grze.

2.4.2 KLIENT GRY

Zgodnie z założeniami dotyczącymi funkcji klienta gry, będącego częścią serwisu WWW, potrzebna była technologia, która potrafiłaby sprostać wymienionym wcześniej wymaganiom i umożliwiałaby odtwarzanie gry w kontenerze wtyczki na stronie, pod przeglądarką internetową. Najczęściej stosowanym tu rozwiązaniem jest technologia Adobe Flash, która jeszcze do niedawna nie oferowała możliwości wyświetlania grafiki 3D i nie posiadała wsparcia sprzętowego ze strony GPU (całość grafiki przeliczana jest programowo za pomocą procesora). Flash pomimo swej popularności i wysokiej penetracji wtyczki Flash Player, nie był jednak stworzony do budowy skomplikowanych gier, lecz co najwyżej niewielkich mini-gier z grafiką 2D. Aby sprostać wymaganiom projektu, należało znaleźć technologię oferującą silnik do budowy gier 3D oraz odpowiednie narzędzia. Na rynku jest dostępnych wiele silników, które gwarantują bardzo dobre efekty wizualne końcowych produktów, oferują rozbudowane narzędzia i atrakcyjne warunki licencji. Wśród tych silników znajdują się popularne: Unreal Engine 3, CryEngine, Torque Game Engine. Każdy z tych silników umożliwia dosyć wygodne stworzenie pełnego, profesjonalnego, komercyjnego produktu, działającego na niejednej platformie, lecz niestety żaden z nich nie spełniał podstawowego warunku – nie pozwalał na umieszczenie gry pod przeglądarką internetową, zgodnie z ustalonymi wcześniej wytycznymi. Z pomocą jednak przyszła inna technologia – Unity – narzędzie/silnik oferujący wszystko, co zostało wcześniej wspomniane, dodatkowo ze wsparciem publikacji gier/aplikacji pod przeglądarki internetowe.

Unity to zintegrowane środowisko do tworzenia gier lub innych interaktywnych, multimedialnych treści, takich jak wizualizacje czy animacje 3D w czasie rzeczywistym. Środowisko może być uruchamiane tylko na systemach Windows i Mac OS X, lecz produkty końcowe mogą być eksportowane również pod konsole, urządzenia mobilne, lub przeglądarki internetowe w postaci wtyczki Unity Web Player. Unity składa się na serię narzędzi projektowych i twórczych, pozwalających stworzyć produkt komercyjny od początku do końca. Unity powstało na bazie platformy Mono, jako narzędzie kompatybilne z platformą .NET, zawierające w sobie kompilator języka C# i umożliwiające podłączenie bibliotek .NET (niestety jednak o ograniczonej funkcjonalności). Dzięki temu, stała się możliwa, przynajmniej częściowo, integracja z resztą projektu, czyli serwisem WWW i bazą danych w postaci jednej solucji środowiska Microsoft Visual Studio.

Zalety wtyczki Unity Web Player (względem użytkownika końcowego):

- wtyczka w całości waży tylko 3MB
- instalacja nie wymaga zrestartowania przeglądarki
- instalacja nie przenosi użytkownika do innej strony
- instalacja nie zawiera żadnego Google Toolbar lub innego adware'u
- nie powoduje problemów przy starszych kartach graficznych (wysoka kompatybilność)

2.4.3 SERWER GRY

Vego 3D, oferujące tylko rozgrywkę sieciową i dającą głównie możliwość komunikacji z innymi użytkownikami, z którymi można przemierzać rozległy, wirtualny świat, wymaga przede wszystkim niezawodnego oprogramowania serwerowego, służącego do szybkiej wymiany danych pomiędzy klientami w czasie rzeczywistym. Po raz kolejny, aby nie wyważać otwartych drzwi, zdecydowano się na użycie gotowego rozwiązania w postaci serwera gniazd. Photon Network Engine, podobnie jak Raknet (na którym zbudowany jest standardowy moduł sieciowy Unity), jest oprogramowaniem typu "middleware". W skład Photona wchodzą zestawy SDK dla wielu platform włączając w to Flash, Windows, .NET, Unity3D, iOS i Android. Photon Network Engine przystosowany jest także do obsługi dużych gier typu MMO.

Zalety Photon Network Engine:

- Serwer zbudowany jest na natywnym C/C++ i używa protokołu UDP w celu najwyższej wydajności, podczas gdy aplikacje pod serwer pisane są w .NET/C#
- Dodatkowa warstwa silnika sieciowego na protokole UDP zapewnia niezawodność i jest dostępna oddzielnie dla każdej operacji
- Serwer potrafi obsłużyć nawet dziesiątki tysięcy równoczesnych połączeń
- Bardzo dobre wsparcie dla Unity
- Obsługa dużej ilości użytkowników (równoczesnych połączeń)
- Wbudowany moduł MMO
- Wsparcie dla aplikacji klienckiej w Unity
- Najlepsza wydajność i niezawodność z pośród dostępnych na rynku rozwiązań serwerowych dla gier MMO

W Vego 3D za serwerową logikę gry odpowiada dedykowana aplikacja, która jest uruchamiana podczas startu Photona. Wszystkie funkcje gry (wywoływanie zdalnych procedur, przechowywanie danych itp.) są zaimplementowane w aplikacji. Źródła napisane w C# kompilowane są do kilku plików i umieszczane w folderze, który jest ostatecznie ładowany i uruchamiany poprzez natywny rdzeń Photona. Zazwyczaj cała logika gry dostarczana jest w ramach pojedynczej aplikacji, lecz Photon umożliwia także uruchomienie większej ilości aplikacji w tym samym czasie.

Komunikacja klienta z serwerem odbywa się w Photonie na zasadzie operacji i zdarzeń. Operacje to wywołania przez klientów zdalnych procedur zdefiniowanych w aplikacji serwerowej i mogą zwracać rezultaty. Zdarzenia natomiast są najczęściej wynikiem wywołania operacji przez innego klienta i mogą dostarczać informacje o innych klientach. Ponadto, podczas komunikacji klienta z serwerem, Photon samodzielnie zajmuje się serializacją i transferem danych.

3 REALIZACJA PROJEKTU

3.1 BAZA DANYCH

W projekcie Vego 3D istnieje jedna baza danych, w oparciu o którą został zbudowany zarówno serwis WWW, jak i klient gry. Ujednolicenie bazy pozwoliło na skorelowanie ze sobą wszystkich elementów gry, które powinny posiadać swój odpowiednik w bazie danych – m.in. konta użytkowników, profile ich awatarów, posiadane przedmioty. Za system baz danych posłużył Microsoft SQL Server 2008. Zrezygnowano natomiast z tradycyjnych schematów budowania tabelowej struktury bazy za pomocą zewnętrznych narzędzi typu Microsoft SQL Server Management Studio, a zamiast tego posłużono się metodą *Code-First*, która jest jednym z wzorców projektowych w ADO .NET Entity Framework. Cechą tej metody jest obiektowa budowa modelu bazy za pomocą zestawu klas bezpośrednio w samym projekcie. Następnie dzięki tzw. kontekstowi bazy, jest możliwy dostęp do bazy, jej struktury i rekordów, przy użyciu wcześniej zdefiniowanego modelu w postaci klas.

Zastosowano następujące klasy, których odzwierciedleniem są tabele w bazie SQL:

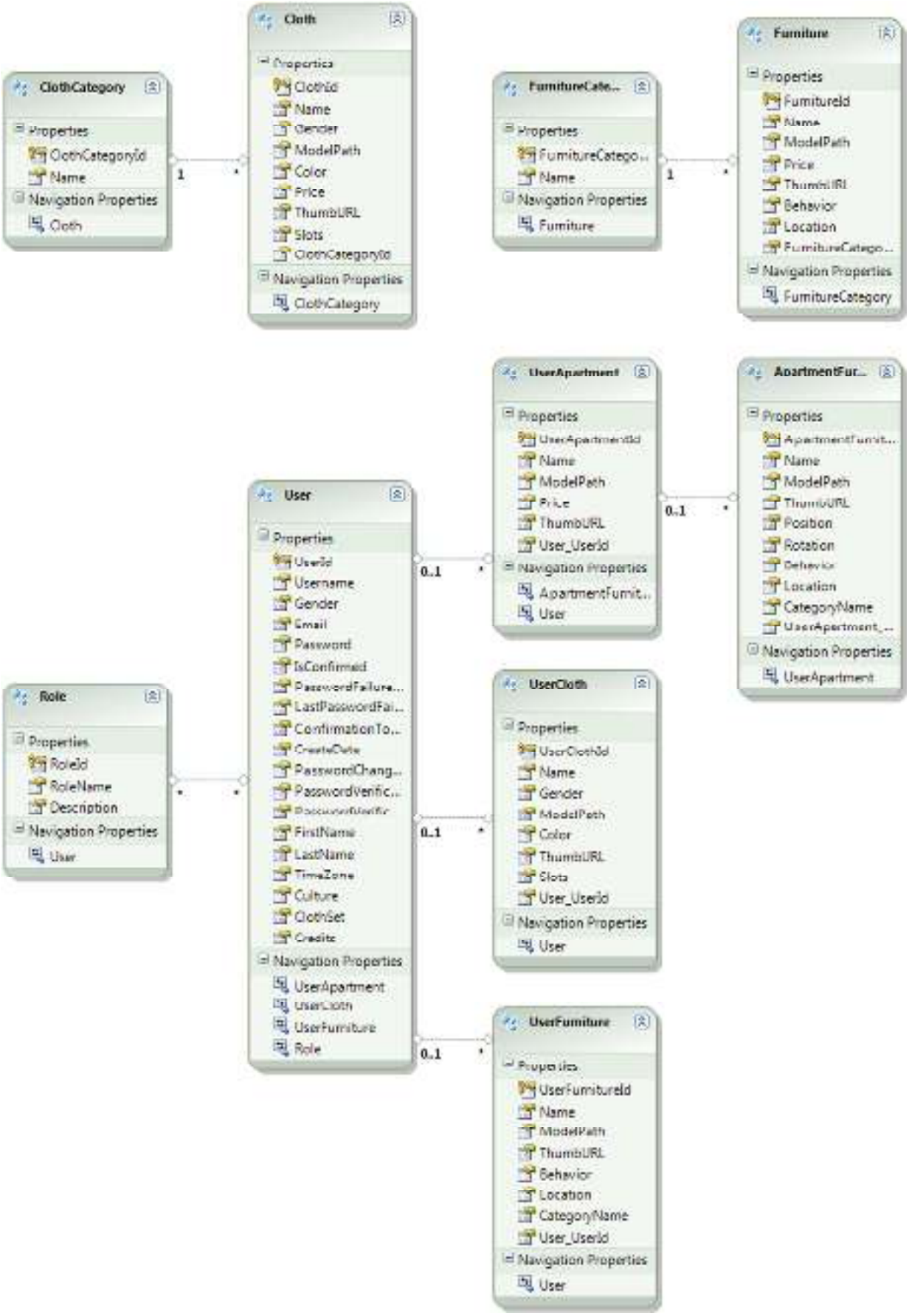
- *Cloth* – klasa reprezentująca obiekt ubrania dostępnego w sklepie, np. „Spodnie”
- *ClothCategory* – klasa reprezentująca kategorię ubrań, np. „Część górna”
- *Furniture* – klasa reprezentująca obiekt mebla dostępnego w sklepie, np. „Krzesło”
- *FurnitureCategory* – klasa reprezentująca kategorię mebli np. „Stoliki”
- *User* – klasa reprezentująca konto użytkownika, zawierająca również dane awatara
- *UserApartment* – klasa reprezentująca obiekt apartamentu, należącego do użytkownika
- *UserCloth* – klasa reprezentująca obiekt ubrania, kupionego w sklepie i należącego już do użytkownika
- *UserFurniture* – klasa reprezentująca obiekt mebla, kupionego w sklepie i należącego już do użytkownika
- *ApartmentFurniture* – klasa reprezentująca obiekt mebla, kupionego w sklepie i należącego już do użytkownika oraz przypisanego do konkretnego apartamentu, gdzie posiada np. pewną pozycję

Podczas rejestracji użytkownika i tworzenia dla niego konta w serwisie do bazy dodawany jest obiekt (rekord) klasy *User* (przy użyciu CodeFirst Membership Provider, pozwalającego na zamianę standardowego mechanizmu kont użytkowników na mechanizm oparty o metodę Code-First). Obiekt użytkownika posiada listy obiektów *UserApartment*, *UserCloth* oraz *UserFurniture*. Na początku użytkownik dostaje swoje pierwsze mieszkanie – tworzony i dodawany do listy jest obiekt typu *UserApartment*. Podczas zakupów w Sklepie są natomiast dodawane obiekty typu *UserCloth* oraz *UserFurniture*, które są tworzone/klonowane według wzorców, którymi są „sklepowe” klasy *Cloth* oraz *Furniture*. Obiekty te zostają na stałe przypisane do danego użytkownika. Z kolei kiedy gracz dodaje mebel do swojego apartamentu, tworzony jest obiekt klasy *ApartmentFurniture*, który jest dodawany do listy mebli w obiekcie apartamentu danego gracza.

Metoda Code-First w ADO .NET Entity Framework, jak było to wspomniane wcześniej, pozwala na tworzenie tabel bazy SQL na podstawie klas zdefiniowanych w źródłach projektu w języku C#. W Vego 3D klasy modelu bazy wraz z kontekstem oraz usługami sieciowymi są częścią projektu serwisu WWW i funkcjonują w jednej przestrzeni nazw *Vego3D.Web*. W celu lepszego zobrazowania oraz pokazania relacji, poniżej zaprezentowany został przykład kodu takich klas: *Cloth* oraz *ClothCategory*, które są wykorzystywane w Sklepie. Każde pole klasy ma swój odpowiednik w polu tabeli dzięki mapowaniu typów pomiędzy schematem źródłowym (modelem) a schematem bazy. EF na podstawie schematu sam tworzy relacje, znajduje klucze itp.

```
public class Cloth
{
    [Key()]
    public Guid ClothId { get; set; }
    [Required()]
    [MaxLength(250)]
    public string Name { get; set; }
    [Required()]
    [MaxLength(10)]
    public string Gender { get; set; }
    [Required()]
    [MaxLength(250)]
    public string ModelPath { get; set; }
    [Required()]
    public string Color { get; set; }

    public string Price { get; set; }
    public string ThumbURL { get; set; }

    public string Slots { get; set; }

    public virtual Guid ClothCategoryId { get; set; }
    public virtual ClothCategory ClothCategory { get; set; }
}

public class ClothCategory
{
    [Key()]
    public Guid ClothCategoryId { get; set; }
    [Required()]
    [MaxLength(250)]
    public string Name { get; set; }

    public virtual ICollection<Cloth> Clothes { get; set; }
}
```

Warto również wspomnieć, że podczas inicjalizacji bazy danych (w momencie zakładania serwisu gry na serwerze hostingowym) tabela z użytkownikami jest pusta, lecz Sklep nie może pozostać pusty. Należało zatem też zadbać o inicjalizację bazy o początkowe obiekty (rekordy) w postaci ubrań i mebli w Sklepie, tak aby każdy (w tym pierwszy) użytkownik mógł dokonać zakupów. W tym celu została napisana klasa, która jest inicjatorem bazy i przy pomocy klas pomocniczych (budowniczych) tworzy i dodaje do bazy obiekty przedmiotów i ich kategorie podczas startu całej aplikacji.



Rys. 3.1 Diagram całej bazy danych w Vego 3D *źródło: opracowanie własne*

3.1.1 USŁUGI SIECIOWE

Platforma WWW, która działa w oparciu o ASP .NET MVC 3, korzysta z danych (wybierając, dodając, modyfikując oraz usuwając je) za pomocą kontrolerów, które z kolei pobierają dane z kontekstu bazy danych. Całość (komunikacja strona WWW – baza danych) doskonale współgra ze sobą w ramach jednego projektu w środowisku IDE. Jednak w przypadku klienta gry, który funkcjonuje jako odrębna aplikacja w formie treści wtyczki Unity Web Player, komunikacja z bazą danych musi odbywać się na inny sposób.

W wyniku tego, że Unity bazuje na platformie Mono, a nie na platformie Microsoft .NET, istnieje wiele funkcjonalnych ograniczeń bibliotek .NET, z których część nie jest obsługiwana i dołączana podczas kompilacji projektu w Unity. To powoduje m.in. to, że nie istnieje możliwość bezpośredniego połączenia z bazą danych z poziomu klienta gry z wykorzystaniem bibliotek .NET. Dlatego też, należało znaleźć rozwiązanie, które byłoby interoperacyjne, niezależne od platformy, na której działa projekt. Z racji tego, że Unity doskonale wspiera protokół HTTP dzięki odpowiednim klasom, zaistniała możliwość wykorzystania tego protokołu do transferu danych pomiędzy bazą a klientem gry za pomocą usług. Do tego celu należało zastosować usługi sieciowe (ang. *web services*), które zawierałyby szereg metod wykonujących operacje na bazie danych na podstawie dostarczonych argumentów. W projekcie Vego 3D zdecydowano się na użycie usług *Windows Communication Foundation* (WCF), które doskonale współgrają z ADO .NET Entity Framework, ułatwiając pracę na bazie danych dzięki jej obiektowości.

WCF zostało stworzone z myślą o łatwym i wygodnym korzystaniu z usług sieciowych w formie komponentów jako zintegrowana technologia do komunikacji, działająca na wielu protokołach. WCF jako część platformy Microsoft .NET nie jest niestety obsługiwana na platformie Mono, ale nie jest to dużą przeszkodą, ponieważ z usług sieciowych WCF można korzystać nie tylko z poziomu .NET, ale również na dowolnej platformie obsługującej protokół HTTP. Dzięki temu, wystarczy wywołać usługę za pomocą jej adresu HTTP, aby w rezultacie otrzymać odpowiedź zawierającą oczekiwane dane.

Dane, które są zwracane przez usługi sieciowe, muszą być odpowiednio serializowane, aby ułatwić ich przesyłanie oraz aby druga strona oczekująca ich zwrócenia mogła je właściwie zinterpretować (deserializując je). W efekcie podczas generowania rezultatu operacji, dane muszą zostać obudowane w odpowiednią strukturę i format. W Vego 3D dane zwracane przez usługi sieciowe mają postać tekstową w formacie JSON. Jest to popularny, lekki format, niezależny od konkretnego języka. Dla Unity został napisany również konwerter, który pozwala z łatwością rozkodowywać rezultaty operacji do postaci Hashtable. Następnie dane są parsowane i na ich podstawie tworzone są obiekty klas występujących w samym kliencie gry, np. Cloth lub Furniture.

Wywołania operacji w kliencie gry następują w oddzielnych wątkach (asynchronicznie), aby główny wątek gry nie był spowalniany poprzez oczekiwanie na rezultaty tych operacji. W każdym wątku nawiązywane jest połączenie z usługą za pomocą adresu HTTP, podawana jest nazwa operacji oraz jej argumenty, a następnie oczekiwany jest rezultat w formacie JSON.

Przykład operacji pobierającej aktualny zestaw ubrań awatara danego użytkownika:

```
[OperationContract]
[WebInvoke(Method = "GET", ResponseFormat = WebMessageFormat.Json)]
public ClothesListObject GetUserClothSet(string userID)
{
    using (var db = new DBContext())
    {
        var clothesLists = from user in db.Users
                           where user.UserId.CompareTo(new Guid(userID)) == 0
                           select user.Clothes;

        User user_object = db.Users.FirstOrDefault(u => u.UserId == new
Guid(userID));
        string[] clothIDs = {};
        int length = 0;
        if (user_object.ClothSet != "") {
            clothIDs = user_object.ClothSet.Split('|');
            length = clothIDs.Length;
        }

        ClothesListObject clothesList = new ClothesListObject {
            ClothesList = new ClothObject[length]
        };
        int i = 0;
        foreach (string clothID in clothIDs)
        {
            if (clothID != "")
            {
                foreach (UserCloth cloth in clothesLists.ToArray()[0])
                {
                    if (cloth.UserClothId == new Guid(clothID))
                    {
                        clothesList.ClothesList[i++] =
ClothObject.ParseCloth(cloth);
                    }
                }
            }
        }
        return clothesList;
    }
}
```

Klasy zawierające dane o ubraniach, które są dostarczane m.in. przez powyższą operację:

```
[DataContract]
public class ClothObject
{
    public ClothObject(Guid id, string name, string gender, string modelPath,
string color, string price, string thumbURL, string slots, Guid clothCategoryId)
    {
        Id = id;
        Name = name;
        Gender = gender;
        ModelPath = modelPath;
        Color = color;
        Price = price;
        ThumbURL = thumbURL;
        Slots = slots;
        ClothCategoryId = clothCategoryId;
    }
```

```
    public ClothObject() { }

    public static ClothObject ParseCloth(UserCloth cloth)
    {
        ClothObject clothObject = new ClothObject();
        clothObject.Id = cloth.UserClothId;
        clothObject.Name = cloth.Name;
        clothObject.Gender = cloth.Gender;
        clothObject.ModelPath = cloth.ModelPath;
        clothObject.Color = cloth.Color;
        clothObject.ThumbURL = cloth.ThumbURL;
        clothObject.Slots = cloth.Slots;
        return clothObject;
    }

    [DataMember]
    public Guid Id { get; set; }

    [DataMember]
    public string Name { get; set; }

    [DataMember]
    public string Gender { get; set; }

    [DataMember]
    public string ModelPath { get; set; }

    [DataMember]
    public string Color { get; set; }

    [DataMember]
    public string Price { get; set; }

    [DataMember]
    public string ThumbURL { get; set; }

    [DataMember]
    public string Slots { get; set; }

    [DataMember]
    public Guid ClothCategoryId { get; set; }
}

[DataContract]
public class ClothesListObject
{
    public ClothesListObject() { }

    [DataMember]
    public ClothObject[] ClothesList { get; set; }
}
```

Powyższa przykładowa operacja może być wywołana z klienta przy użyciu poniższego adresu:

```
http://188.165.231.45/Vego3D/Services/DBServices.svc/GetUserClothSet?userID=64cf1c57-c921-48e4-ba41-132f70ba83ad
```

gdzie *userID* to ID użytkownika pobrane wcześniej z bazy za pomocą innej operacji na podstawie jego nazwy.

3.2 SERWIS WWW

Nieodłącznym elementem Vego 3D, jako rozbudowanej gry internetowej, jest serwis WWW, który nie tylko stanowi kontener dla klienta gry, ale jest przede wszystkim integralną częścią gry, przejmując duży fragment jej funkcjonalności. Klient gry jest tylko „oknem" na wirtualny świat 3D, który do pełnego i poprawnego działania potrzebuje mechanizmów „webowych" dostarczających i weryfikujących dane.

Jednym z takich mechanizmów jest autoryzacja użytkowników za pomocą formularza logowania, dzięki któremu użytkownicy uzyskują dostęp do serwisu i gry. System obsługi kont użytkowników wraz z autoryzacją (*Membership Provider*) jest częścią szkieletu ASP .NET MVC 3 udostępnianego bezpłatnie przez Microsoft. Niestety, programista nie ma dostępu do struktury bazodanowego obiektu użytkownika (który jest częścią standardowego systemu kont) z poziomu kodu, a jedynie z poziomu bazy, co uniemożliwiało unifikację z resztą struktury bazy Vego 3D, która według założeń miała być budowana i inicjalizowana tylko za pomocą metody Code-First. Jedynym rozwiązaniem pozostało stworzenie nowego systemu kont, dedykowanego dla projektu, posiadającego obsługę pozostałych elementów struktury bazy danych. Z pomocą jednak przyszli autorzy bezpłatnego rozszerzenia do MVC 3 w postaci własnego systemu kont opartego o metodę Code-First (*CodeFirst Membership Provider*). System ten tworzy w bazie obiekty użytkowników na podstawie klasy User, która jest dołączona do rozszerzenie. Staje się zatem możliwa edycja jej struktury, co pozwala na powiązanie jej z pozostałymi klasami projektu, tak aby obiekt użytkownika zawierał informacje o awatarze – ubraniach, meblach i apartamentach.

Struktura serwisu Vego 3D oraz nawigacja w nim jest bardzo prosta. W projekcie ograniczono się do podstawowej funkcjonalności, jaką jest klient gry 3D oraz sklep. Chcąc rozbudować projekt do komercyjnej jakości, aby miał szansę przetrwać na rynku i zarobić na siebie, należałoby zadbać także o moduły charakterystyczne dla portali społecznościowych. Modułami takimi z pewnością są: wewnętrzna poczta (wiadomości prywatne pomiędzy użytkownikami), system zawierania znajomości, możliwość konfigurowania własnego profilu przez użytkownika oraz dodatkowo rankingi użytkowników, galerie zdjęć, forum dyskusyjne. Niestety, z racji skali takiego przedsięwzięcia oraz ograniczeń czasowych, zdecydowano się na zaprojektowanie i zbudowanie jedynie zalążka gry, który jednak stanowi idealną podstawę do dalszej rozbudowy. Zatem, jak zostało to przedstawione wcześniej w opisie funkcjonalności, serwis WWW składa się z dwóch głównych modułów: Gry i Sklepu. Odnośniki do tych modułów w menu głównym są początkowo niedostępne i wymagają autoryzacji użytkownika. Dopiero po zalogowaniu się użytkownika w serwisie możliwa jest jego identyfikacja i pobranie podstawowych informacji o nim w celu wykonania dalszych akcji, m.in. w celu autoryzacji klienta gry oraz okna podglądu awatara w sklepie.

Całość serwisu została zbudowana w technologii ASP .NET korzystając z szkieletu MVC 3, pozwalając na wygodne tworzenie podstron za pomocą widoków i sterujących nimi kontrolerów. Kontrolery w głównej mierze dostarczają danych potrzebnych do wyświetlenia treści w widoku strony. Dane te zostają pobierane w kontrolerze wykorzystując kontekst bazy danych. Wszystkie te operacje odbywają się po stronie serwera, gdzie aplikacja nawiązuje połączenie z bazą. Działanie serwisu zależy jednak nie tylko od kodu serwerowego, ale także

od skryptów, które będą na bieżąco w przeglądarce internetowej obsługiwać pewne akcje – nawigację strony oraz komunikację pomiędzy jej elementami. W tym celu wykorzystano język JavaScript oraz bibliotekę jQuery. Wykorzystanie jQuery pozwoliło nie tylko na efektowne wyświetlanie treści, ale przede wszystkim korzystając z technologii Ajax (obsługiwanej przez jQuery) wyświetlać treści asynchronicznie, bez potrzeby odświeżania strony. Dodatkową zaletą JavaScript okazała się także możliwość komunikacji strony z wtyczką Unity za pomocą tego języka. Posłużyło to m.in. do mechanizmu autoryzacji klienta gry, przekazując do niego nazwę użytkownika i na jej podstawie możliwe stało się pobieranie dalszych informacji z bazy, który były wymagane do właściwego wyświetlenia np. jego awatara w świecie 3D.

3.2.1 *Sklep gry*

Sklep w Vego 3D jest idealnym przykładem na pokazanie wszystkich opisanych powyżej zależności i mechanizmów, które pozwalają współdziałać kilku różnym technologiom. Językiem i technologią, która spaja pozostałe technologie jest JavaScript oraz biblioteka jQuery, która stała się podstawą w opracowaniu nawigacji Sklepu. Sklep dzieli się na dwa typy przeglądanej zawartości: ubrań oraz mebli. W celu aktywowania innego typu (domyślnie są to ubrania), należy wybrać zakładkę Clothes lub Furnitures znajdujące się pod menu głównym. Po naciśnięciu na daną zakładkę zamiast zwykłego odnośnika HTML uruchamiana jest akcja w kontrolerze sklepu, która pobiera z bazy listę kategorii danego typu (np. listę kategorii ubrań, czyli spodnie, koszule itp.) oraz zwraca ją w postaci widoku, który ładowany jest w konkretnym miejscu szablonu strony (według identyfikatora kontenera). Cała operacja odbywa się asynchronicznie poprzez zastosowanie technologii Ajax, która potrafi odświeżyć tylko wycinek strony bez potrzeby odświeżenia całości, co oszczędza czas ładowania strony i poprawia estetykę nawigacji. Podczas wykonywania operacji Ajax, ASP umożliwia określenie dodatkowych opcji takich jak, *OnBegin* lub *OnComplete*, których zdefiniowanie pozwala na wywołanie kolejnych funkcji JavaScript. W przypadku przełączania zakładek odnoszących się do typu sklepu, uruchamiane są dwie funkcje. Jedna odpowiada za przełączenie grafiki zakładki na aktywną, a druga za przesłanie informacji (w postaci zdalnego wywołania funkcji) do okna Unity, w celu zmiany widoku typu sklepu w podglądzie 3D. Objawia się to m.in. innym obrotem kamery, która zwrócona jest w stronę aktualnie oglądanego przedmiotu, w przypadku ubrań jest to awatar użytkownika, a w przypadku mebli model pojawiający się w innym miejscu pokoju wirtualnej przymierzalni.

Przykład wywołania akcji w ASP podczas naciśnięcia na zakładkę określającej typ sklepu:

```
<ul>
    <li id="shopTab_clothes">@Ajax.ActionLink("Clothes", "Clothes", new { }, new
AjaxOptions { UpdateTargetId = "shopCategory", OnBegin = "ChangeShop('clothes')",
OnComplete = "ActivateTab('shop', 'clothes');" })</li>

    <li id="shopTab_furnitures">@Ajax.ActionLink("Furnitures", "Furnitures", new {
}, new AjaxOptions { UpdateTargetId = "shopCategory", OnBegin =
"ChangeShop('furnitures')", OnComplete = "ActivateTab('shop', 'furnitures');"
})</li>
</ul>
```

Definicja jednej z powyższych wywoływanych funkcji JavaScript - *ChangeShop*:

```
function ChangeShop(type)
{
    var shop = GetShop();

    if (shop)
        shop.SendMessage("BrowserServices", "ChangeCategory", type);
}
```

Powyżej przedstawiona funkcja *ChangeShop* posiada w swojej definicji wywołanie metody obiektu *shop*, który odzwierciedla kontener Unity zawierający podgląd 3D przymierzalni. Obiekt kontenera został zdefiniowany w innej funkcji i jest typu *UnityObject*. Klasa *UnityObject* natomiast została dostarczona przez autorów Unity w postaci gotowego skryptu i służy do osadzania kontenerów wtyczki Unity Web Player na stronie HTML. Metoda *SendMessage* jest zatem funkcją zdefiniowaną w tym skrypcie i służy do zdalnych wywoływani innych funkcji, znajdujących się w kodzie aplikacji, która została osadzona na stronie i która jest dostępna pod danym obiektem, w tym przykładzie pod obiektem *shop*. Nazwa zdalnie wywoływanej funkcji oraz jej argumenty dostarczane są jako argumenty metody *SendMessage* – „ChangeCategory", czyli nazwa funkcji zmieniającej kategorię/typ sklepu. Podobnie został zrealizowany mechanizm autoryzacji zarówno podglądu przymierzalni, jak i klienta gry, który za pomocą metody *SendMessage* dostarcza aplikacji Unity danych o użytkowniku w postaci tzw. tokenu autoryzacji.

Po załadowaniu kategorii danego typu przedmiotów (ubrań lub mebli) pojawiają się kolejne zakładki, tym razem w prawej kolumnie sklepu. Zakładki te dodawane są w pętli, która przebiega przez całą listę obiektów kategorii, która została wcześniej dostarczona przez kontroler. Aktywowanie jednej z zakładek powoduje asynchroniczne załadowanie następnych treści – listy przedmiotów należących do danej kategorii. Domyślnie aktywowana jest pierwsza w kolejności zakładka. Również w tym przypadku zastosowana zostaje technologia Ajax.

Przykład wstawienia zakładek kategorii:

```
<ul>
    @foreach (var item in Model)
    {
        <li id='@("categoryTab_" +
item.ClothCategoryId)'>@Ajax.ActionLink(item.Name, "ClothCategory", new {
ClothCategoryId = item.ClothCategoryId }, new AjaxOptions { UpdateTargetId =
"categoryItems", OnComplete = "ActivateTab('category', '" +
item.ClothCategoryId.ToString() + "');" })</li>
    }
</ul>
```

Ponownie następują odnośniki z akcją, której jedna z opcji to funkcja JavaScript, która przełącza grafikę zakładki na aktywną (*ActivateTab*). Odbywa się to na zasadzie podmian klas CSS w czasie rzeczywistym za pomocą identyfikatora elementu listy jako części struktury strony HTML. Jest to jedna z zalet użycia biblioteki jQuery.

Funkcja ta oraz funkcja deaktywująca zakładki przedstawia się następująco:

```
function ActivateTab(type, id) {
    if (type == 'shop') {
        currentShop = id;
    }
    if (type == 'category')
        currentCategory = id;
    DeactivateTabs(type);
    $("#" + type + "Tab_" + id).addClass("ui-tabs-selected ui-state-active");
}

function DeactivateTabs(type) {
    $("li[id^='" + type + "Tab_']").each(function (i, v) {
        $(v).removeClass("ui-tabs-selected ui-state-active");
    });
}
```

Kiedy dana kategoria przedmiotów zostaje załadowana, ukazuje się lista tych przedmiotów, które wyświetlane są w sposób odpowiednio sformatowany, przedstawiając miniaturkę przedmiotu, jego cenę oraz przyciski Try (Przymierz) i Buy (Kup). Identyfikator, cena oraz adres URL do miniaturki danego przedmiotu zostają pobrane z modelu, który podczas ładowania strony (a właściwie tylko jej wycinka – listy przedmiotów) został przekazany z kontrolera do widoku strony. Przycisk Try powoduje uruchomienie funkcji JavaScript, która wykorzystując wspomnianą już wcześniej metodę *SendMessage* obiektu Unity, powoduje zdalne wywołanie operacji odpowiedzialnej za załadowanie odpowiedniego modelu w podglądzie 3D. Jako argument podaje się nazwę funkcji (np. „ViewCloth” lub „ViewFurniture”) oraz identyfikator przedmiotu (w bazie danych). W efekcie po naciśnięciu przycisku/odnośnika HTML w sposób natychmiastowy ładowana jest odpowiednia treść w kontenerze Unity Web Player. Z kolei przycisk Buy inicjuje zakup przedmiotu poprzez wywołanie okienka modalnego potwierdzającego zakup. Obie akcje Try i Buy bazują na identyfikatorze przedmiotu, który został ukryty w nazwie ich klasy CSS. Przypisanie tej nazwy zostało dokonane dynamicznie, podczas wyświetlenia listy przedmiotów posługując się modelem dostarczonym przez kontroler. Mając wszystkie informacje na temat danego obiektu, łatwo można było wykorzystać identyfikator w celu oznaczenia elementów struktury strony, aby potem wygodnie dało się szukać i rozpoznawać te elementy w kodzie JavaScript. Poniższy przykład przedstawia sposób generowania listy przedmiotów:

```
@foreach (var item in Model) {
    <li>
        <a id='@("thumb_" + item.ClothId)' class="thumb" href="#">
            <img src='@Url.Content("~/Content/clothes/" + item.ThumbURL)'
alt='@item.Name' />
        </a>
        <div class="shopItemLabels">
            <img id='@("price_" + item.ClothId)'
src='@Url.Content("~/Content/images/icon-money.png")' alt='@item.Price' />
            <span>@item.Price</span>
        </div>
        <div class="shopItemButtons">
            <a id='@("try_" + item.ClothId)' href="#">Try</a> | <a id='@("buy_" +
item.ClothId)' href="#">Buy</a>
        </div>
    </li>
}
```

Korzystając z identyfikatora przedmiotu ukrytego w nazwie klas CSS obu przycisków Try i Buy, zdefiniowanie ich akcji podczas naciśnięcia zostało zrealizowane następująco:

```
function AddEventsTry() {
    $("a[id^='try_']").unbind();
    $("a[id^='try_']").each(function (i, v) {
        $(v).click(function (el) {
            idHTML = this.id.split("_");
            id = idHTML[1];

            AddTryClick(id);
            return false;
});});}

function AddEventsBuy() {
    $("a[id^='buy_']").unbind();
    $("a[id^='buy_']").each(function (i, v) {
        $(v).click(function (el) {
            idHTML = this.id.split("_");
            id = idHTML[1];

            AddBuyClick(id);
            return false;
});});}
```

Funkcja *AddTryClick,* jak już było to wspomniane wcześniej, odpowiedzialna jest za załadowanie modelu o danym identyfikatorze, natomiast *AddBuyClick* uruchamia okno zakupu przedmiotu. Następnie, kontynuując, po potwierdzeniu zakupu, dokonywany jest zakup przedmiotu (poprzez wywołanie odpowiedniej akcji w ASP), odejmowana jest liczba kredytów od puli użytkownika oraz wyświetlany jest komunikat o statusie wykonanej czynności (za pomocą BlockUI – dodatku do jQuery). Zakup nie zawsze musi zakończyć się powodzeniem, ponieważ użytkownik może nie posiadać wymaganej liczby kredytów, lub użytkownik już posiada dany przedmiot. W takiej sytuacji wyświetlany komunikat posiada odpowiednią informację.

3.3 KLIENT GRY

Klient gry, który jest kluczową częścią serwisu i uruchamiany jest w kontenerze wtyczki, stanowi jednak oddzielną aplikację, stworzoną przy użyciu technologii Unity. Aplikacja Unity przejmuje większość funkcjonalności gry będąc „oknem” na wirtualny, trójwymiarowy świat Vego 3D. To w niej użytkownik za pomocą swojego awatara może eksplorować lokacje gry (z których składa się świat), komunikować się z innymi użytkownikami i wchodzić z nimi w interakcję. W przeciwieństwie do innych, pokrewnych gier oraz aplikacji, które bardziej przypominają interaktywne czaty 3D, Vego 3D oferuje swobodne poruszanie się po świecie, nawiązując tym samym do gier MMO. Awatar użytkownika posiada kontroler pozwalający na chód, bieg i skok w dowolnym kierunku oraz kamerę patrzącą z trzeciej osoby, która podąża za awatarem umożliwiając jednocześnie zbliżenie i obrót widoku.

Jednym z istotnych aspektów budowy klienta gry była organizacja zasobów gry w taki sposób, aby użytkownikowi w danym momencie dostarczyć to, co jest mu aktualnie potrzebne i aby zminimalizować czas ładowania kolejnych treści. Vego 3D jest grą internetową, dlatego wszystkie zasoby muszą być dostarczane za pomocą sieci. Dokonanie właściwego podziału zasobów i realizacja mechanizmów ich używania pomogły jednak na stworzenie systemu płynnego ładowania świata i elementów gry. W efekcie główny plik wykonawczy aplikacji (obsługiwany przez wtyczkę Unity Web Player) waży ok. 500KB. Znajdują się w nim przede wszystkim skrypty/komponenty, grafika interfejsu oraz potrzebne biblioteki. Całość grafiki 3D została jednak umieszczona w oddzielnych plikach (tzw. *Asset Bundles*), które ładowane są w odpowiednim momencie. W ten sposób został dokonany podział ubrań (modele + materiały z teksturami), mebli oraz całych lokacji. Każdy plik zasobu zawiera nie tylko sam model z materiałem, ale także inne informacje o transformacji (początkowe położenie, obrót, skala) oraz posiadanych komponentach (skryptach) wraz z wartościami pól. Jest to tzw. *prefab* (prefabrykat), który może zostać wykorzystany (instancjonowany) w grze bez dodatkowych ustawień.

Plik wykonawczy klienta gry jest zbiorem komponentów, które odpowiadają za zarządzanie głównymi modułami gry, takimi jak klient sieciowy, GUI (graficzny interfejs), usługi przeglądarkowe, menadżer scen i apartamentów, menadżer przedmiotów (ubrania, meble), konfigurator mieszkania. Wszystkie skrypty, które obsługują mechanikę gry, a także zachowanie poszczególnych obiektów na scenach, znajdują się w jednym miejscu, w głównym pliku. Dzięki temu późniejsze aktualizacje mechaniki fragmentów gry polegają na aktualizacji i przekompilowaniu głównego pliku, bez potrzeby aktualizacji zasobów (które często posiadają referencje do poszczególnych klas).

W momencie, kiedy użytkownik nawigując po serwisie WWW wybierze grę, rozpoczyna się inicjalizacja klienta gry. Kiedy zostaje załadowana aplikacja (główny plik gry, odtwarzany za pomocą wtyczki), Unity wysyła do strony, w którym został osadzony klient, zapytanie o autoryzację - komunikacja, jak wcześniej zostało to opisane, jest zrealizowana za pomocą JavaScript. Skrypt osadzony na stronie zawiera definicję funkcji *unity_LoginAuthorization*, która wywołana zdalnie w rezultacie wywołuję funkcję zwrotną *LoginAuthorization*. Tym razem jednak podawany jest argument, którym jest token zawierający nazwę zalogowanego użytkownika. Taki mechanizm autoryzacji zapobiega niechcianemu uruchomienia aplikacji, spoza środowiska Vego 3D, np. lokalnie, po pobraniu głównego pliku na dysk. Następnie, gdy zostanie dokonana weryfikacja użytkownika, rozpoczyna się pobieranie z bazy danych podstawowych informacji o nim, które pozwolą na dostosowanie elementów gry. Pobrane w tym momencie zostają m.in. ID użytkownika oraz lista posiadanych apartamentów. Proces pobierania informacji z bazy danych, jak już wcześniej było to opisane, zrealizowane zostało za pomocą usług sieciowych.

Przykład wywołania i wykorzystania usługi sieciowej w kliencie gry w celu pobrania z bazy danych i przetworzenia informacji o użytkowniku:

```
public void GetUserByName(GameClient client, string userName)
{
    StartCoroutine(GetUserByName_Client_Thread(client, userName));
}

private IEnumerator GetUserByName_Client_Thread(GameClient client, string userName)
{
    string url = GameClient.ROOT_WEB_URL +
"Services/DBServices.svc/GetUserByName?userName=" + userName;
    WWW www = new WWW(url);
    yield return www;

    string json_result = www.text;
    Hashtable result = (Hashtable)(MiniJSON.JsonDecode(json_result));
    client.userID = result["Id"].ToString();
    client.userGender = result["Gender"].ToString();

    client.loadUserID = true;
    client.loadUserGender = true;
}
```

Wszystkie metody korzystające z usług wymagają obudowania ich w oddzielne wątki (w Unity tzw. *coroutine*), aby zapytania mogły odbywać się asynchronicznie i czekanie na rezultat nie wstrzymywało funkcjonowania gry. Następnie w obiekcie typu *GameClient* (główna klasa klienta gry) ustawiane są odpowiednie flagi, aby była możliwa kontynuacja przepływu logiki gry.

3.3.1 KONSTRUKCJA WIRTUALNEGO ŚWIATA

Kiedy zostanie pobrana lista apartamentów zalogowanego użytkownika, rozpoczyna się ładowanie pierwszej lokacji – domyślnego apartamentu, pierwszego na liście użytkownika. Wszystkie lokacje (sceny) dostępne w grze pobierane są jako oddzielne zasoby z repozytorium znajdującego się na serwerze WWW. Pobieranie następuje przez protokół HTTP za pomocą wbudowanej klasy WWW i odbywa się asynchronicznie w oddzielnym wątku. Podczas ładowania wyświetlany jest ekran ładowania, który znika po załadowaniu lokacji. W projekcie stworzono prosty mechanizm przechowywania załadowanych już lokacji oraz wszystkich innych zasobów pobieranych przez HTTP – tzw. *cache*. Dzięki temu, podczas kolejnych prób pobrania tego samego elementu, dany element jest wyszukiwany w odpowiednim obiekcie zbiorczym zawierającym listę elementów tego typu i jeśli został tam dodany wcześniej (podczas pierwszego załadowania), to jest on używany z tej listy. Mechanizm ten działa jednak tylko w obrębie jednej sesji – od chwili uruchomienia aplikacji do jej zamknięcia (np. poprzez zamknięcie strony lub przeglądarki). Pomimo tego, ładowanie tych samych elementów w obrębie kilku sesji również może trwać krócej, a elementy mogą zostać załadowane z dysku, ale pod warunkiem, że przeglądarka, z której korzysta użytkownik, ma włączone przechowywanie na dysku pobranych danych – *cache* przeglądarki.

W momencie kiedy załadowana zostaje pierwsza lokacja (w obrębie jednej sesji), załadowany zostaje także awatar użytkownika. Podczas wejścia do lokacji, tworzony jest lokalny obiekt aktora – postaci, która posiada takie cechy, jak m.in.: przynależność do danego użytkownika (według jego ID), nazwa (według nazwy użytkownika), płeć, aktualnie założone ubrania, posiadane ubrania (tzw. szafa użytkownika), posiadane meble oraz obiekt wizualnej reprezentacji awatara (model z animacjami), a także flagi określające jego status (czy jest ładowany, czy pisze wiadomość na czacie, czy jest nieaktywny - z dala od komputera, czy siedzi na krześle). Za obiekt aktora odpowiedzialna jest klasa *Actor*, która posiada odpowiednie pola i metody obsługujące wymienione wyżej cechy. Klasa ta jest wspólna dla wszystkich awatarów znajdujących się w lokacji, bez względu na to, czy obiekt aktora należy do lokalnego użytkownika (aktualnie zalogowanego) czy zdalnego (innego użytkownika, napotkanego w lokacji). W celu rozróżnienia natomiast stosuje się klasy pochodne *ActorLocal* oraz *ActorRemote*. Podział ten jest konieczny, ponieważ obiekty te zachowują się inaczej: obiekt lokalny aktora, należący do lokalnego użytkownika posiada kontroler odpowiedzialny za sterowanie postacią, a obiekt zdalny z kolei posiada mechanizm stałego aktualizowania swoich parametrów (położenia, obrotu, animacji itp.) na podstawie danych dostarczonych przez serwer gniazd. Użytkownik lokalny nie ma wpływu na obiekty zdalne, nie może nimi sterować, może tylko obserwować ich zachowanie.

Aktor lokalny wraz z danymi z bazy oraz modelem postaci zostaje załadowany podczas pierwszego wejścia do świata (w obrębie jednej sesji). Potem, kiedy zmienia lokację, przenosi się do innej za pomocą nawigacji w grze, awatar nie jest usuwany i tworzony od nowa, lecz jego czynności są jedynie „zamrażane” na czas zmiany zawartości lokacji. W przeciwieństwie do lokalnego awatara, awatary innych użytkowników są tworzone i usuwane w momencie kiedy wchodzą lub wychodzą z gry. Jednak dzięki cache’owi przeglądarki i pobranym już wcześniej modelom, ich ponowne wejście do świata nie wiąże się z ponownym pobraniem ich modeli, lecz jedynie z załadowaniem ich z dysku. Ładowanie modeli awatarów następuje zatem za każdym razem, kiedy użytkownik pojawia się w lokacji (za pierwszym razem w przypadku lokalnego awatara). Jest to czynność wspólna dla wszystkich aktorów i odbywa się zaraz po utworzeniu obiektu aktora i pobraniu informacji z bazy odnośnie aktualnie założonych ubrań. Po pobraniu z bazy listy ubrań, następuje załadowanie ich modeli, które również stanowią odrębne zasoby pobierane z repozytorium na serwerze WWW.

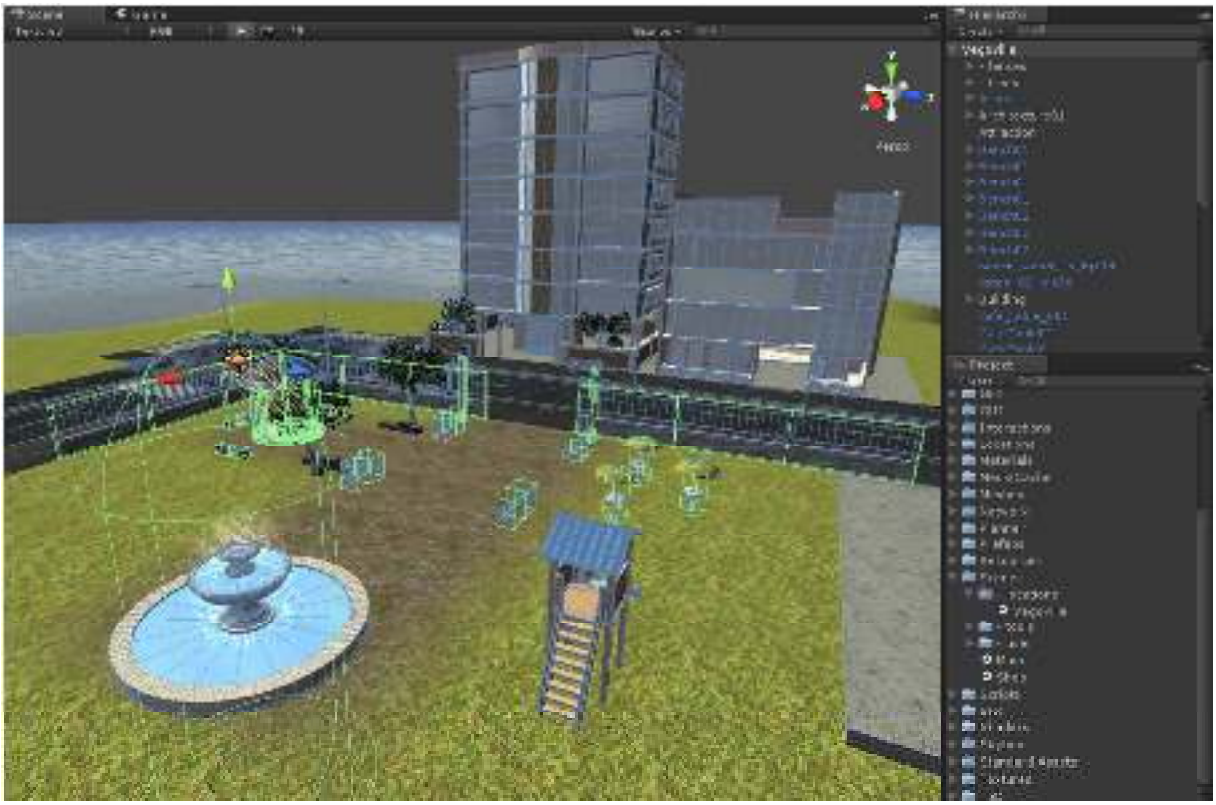
Mechanizm zmiany lokacji, jak już wspomniano wcześniej, odbywa się poprzez pobranie i załadowanie odpowiedniego zasobu (sceny). Scena lokacji jest strukturą poszczególnych obiektów, odpowiedzialnych grafikę 3D, efekty specjalne, dźwięk lub skrypty. Wszystkie obiekty w scenach lokacji w Vego 3D są podpięte do jednego głównego obiektu, który posiada nazwę danej sceny oraz skrypt *SceneSetup* odpowiedzialny za ustawienia danej lokacji. Jednym z takich ustawień jest czas pobrany z serwera, który reguluje porę dnia. Każda lokacja może być wyposażona w obiekt *TimeOfDay*, który po ustawieniu czasu, przedstawia aktualną pozycję słońca oraz odpowiednie kolory nieba. Czas natomiast jest pobierany z serwera poprzez asynchroniczne użycie odpowiedniej funkcji w usłudze sieciowej (podobnie, jak zostało to przedstawione wcześniej, w przypadku pobrania danych o użytkowniku). Główny obiekt sceny, która zostaje załadowana, umieszczany jest na pustej scenie głównej aplikacji Unity. Wraz z całą grafiką lokacji umieszczany jest także marker

odzwierciedlający startową pozycję postaci. Kiedy użytkownik pojawia się w danej lokacji, jego awatar przyjmuje pozycję tego markera, a sam marker jest usuwany. Z kolei kiedy użytkownik opuszcza lokację i przenosi się do innej, główny obiekt sceny jest usuwany i zastępowany obiektem nowej sceny.

Sceny apartamentu użytkownika oraz lokacji wspólnej Vegoville, składają się w głównej mierze z modeli 3D, które w Unity zostały w większości obudowane w prefabrykaty. Wszystkie modele są darmowe (na licencji do niekomercyjnego użytku) i zostały pobrane z internetu, z następujących portali:

- turbosquid.com
- 3dmodelfree.com
- archive3d.net
- artist-3d.com
- virtualworlds.wikia.com

Wybór modeli nie był przypadkowy, ponieważ liczyła się nie tylko ich estetyka, ale również ich złożoność, która przekłada się na obciążenie względem wydajności i płynności gry. Większość dostępnych modeli jest tworzona z myślą o wykorzystaniu ich w statycznym renderowaniu w programach graficznych, nie o użyciu ich w silnikach gier. Zmienia to bowiem podejście w tworzeniu samego modelu, który musi zostać odpowiednio zoptymalizowany pod kątem liczby wierzchołków oraz użytych materiałów i tekstur. W Vego 3D zostały zatem użyte modele o możliwie jak najniższej złożoności przy uwzględnieniu, oczywiście, ich estetyki. Celem było uzyskanie liczby ok. 200 tys. (lub mniej) trójkątów na jedną scenę, co i tak wydaje się czymś dość kosztownym.



Rys. 3.2 Widok z edytora Unity na scenę Vegoville *źródło: opracowanie własne*

Oprócz samych modeli 3D poszczególnych obiektów, sceny składają się również z elementów takich jak: obiekty kolizyjne (ograniczające obszar, w którym może poruszać się awatar użytkownika, efekty specjalne (woda oraz systemy cząsteczkowe, np. fontanna), oraz światła. Woda (w tle), która charakteryzuje się odbiciami oraz refrakcją, korzysta z dostarczonego przez Unity skryptu oraz materiału. Teren natomiast został wyrzeźbiony dzięki wbudowanemu narzędziu do tego służącemu, który nie tylko pozwala na zmiany kształtu, ale także malowanie wybranymi teksturami w wybranych miejscach terenu.

W projekcie Vego 3D nie przywiązano dużej wagi do pięknej i rozbudowanej grafiki, ale skupiono się przede wszystkim na mechanice gry, która stanowi podstawę funkcjonowania projektu. Ograniczono się zatem do dwóch scen/lokacji, kilkunastu modeli otoczenia oraz kilku dostępnych w sklepie modeli mebli. Dopracowane mechanizmy gry pozwalają jednak na szybkie, wygodne i bezproblemowe dodawanie kolejnych lokacji, modeli lub przedmiotów w sklepie. Napisano także dodatkowe skrypty dla edytora Unity, które są odpowiedzialne za eksport modeli przedmiotów oraz lokacji z Unity do formatu *Asset Bundles*, czyli plików – zasobów, które umieszczone na serwerze WWW, mogą być z niego strumieniowane.

3.3.2 KONFIGURACJA WYGLĄDU AWATARA

Vego 3D pozwala użytkownikowi na skonfigurowanie swojego awatara według własnego uznania za pomocą różnych ubrań, które są dostępne w sklepie. Użytkownik może przymierzyć dowolne ubranie, a po zakupie go założyć, lecz może go później zdjąć i w dowolnym momencie zamienić na inne, które posiada w swojej szafie. Wizualne reprezentacje awatarów w całości składają się z kilku połączonych modeli ubrań, które podzielone są na kategorie - sloty. Do każdego slotu pasuje konkretny rodzaj ubrania, których w Vego 3D wyróżniono pięć: *hair*, *face*, *top*, *pants*, *shoes.* Wszystkie modele ubrań podczas procesu tworzenia musiały zostać nałożone na szkielet postaci poprzez przypisanie wierzchołków modelu do kości szkieletu. Każdy model ubrania musiał posiadać także materiał, do którego przypisana została odpowiednia tekstura. Istnieje również możliwość zdefiniowania kilku materiałów, co skutkuje powieleniem sztuki ubrania (modelu), lecz z innym materiałem – inną teksturą, innymi kolorami. Następnie tak przygotowane ubrania musiały zostać wyeksportowane do formatu zasobów – *Asset Bundles*, aby istniała możliwość ich dynamicznego, pojedynczego ładowania w grze. Ubraniom należało również nadać unikalne nazwy, po których dało się również szybko stwierdzić do jakiej kategorii – slotu należą. Zdecydowano się na następującą formułę: [płeć]_[nazwa_slotu]-[ID]_[wersja_ubrania], przykład: *male_pants-2_blue*.

Podstawą mechanizmu konfiguracji postaci (oraz poniekąd też pomysłu na projekt) stał się projekt – demo udostępniony przez twórców Unity jako przykład realizacji kreatora postaci – dostępny pod adresem http://unity3d.com/support/resources/example-projects/charactercustomization. Całość (źródła rozszerzenia oraz grafika 3D) została udostępniona zupełnie za darmo, do dowolnego wykorzystania i modyfikowania. Modele postaci (chłopaka i dziewczyny) zostały w całości zaadaptowane w Vego 3D, lecz zostały

dodane także dodatkowe tekstury kilku ubrań (z logiem Vego 3D), które zostają startowymi, domyślnymi ubraniami dla każdego nowo utworzonego awatara. Również wspomniane rozszerzenie w postaci kilku klas C#, które zajmuje się tworzeniem i ładowaniem ubrań dla postaci, zostało dołączone do projektu Vego 3D z drobnymi modyfikacjami. Generator postaci (bo tak można nazwać to rozszerzenie) posiada funkcje, które odpowiadają za kolekcjonowanie modeli i materiałów ubrań oraz ich eksport do formatu *Asset Bundles*. Pliki wyjściowe składają się z plików bazowych (zawierających szkielet oraz animacje postaci) oraz z plików z modelami ubrań (każdy plik zawiera również kolekcję pasujących do niego materiałów).

Generator postaci, podczas procesu tworzenia awatara, pobiera z repozytorium odpowiednie zasoby ubrań, elementy, które następnie łączy w całość. Wcześniej jednak tworzy ich listę na podstawie parametrów jakie zostają mu dostarczone podczas wywołania. Parametry zapisywane są w formie łańcucha znaków, który generowany jest przez funkcję odpowiedzialną za pobranie zestawu ubrań, który aktualnie posiada użytkownik. Funkcja ta operuje, podobnie jak opisane wcześniej metody, na usłudze sieciowej, za pomocą której pobiera listę ubrań. Zarówno obiekt generatora postaci, jak i wywołania metod menadżera ubrań (bazującego na usługach sieciowych), są częścią klasy *Actor*, z której korzystają wszystkie obiekty awatarów na scenie danej lokacji. Klasa *Actor* podczas inicjalizacji swojego obiektu odpowiedzialna jest w pierwszej kolejności (po pobraniu podstawowych danych o użytkowniku – ID, nazwy, oraz płci) za załadowanie aktualnie założonych ubrań oraz, jeśli awatar należy do lokalnego użytkownika, także listy posiadanych ubrań (w szafie) oraz listy posiadanych mebli. Ładowanie aktualnego zestawu ubrań odbywa się poprzez wywołanie w obiekcie aktora funkcji *LoadCurrentClothSet*, która korzystając z globalnego obiektu klasy *ClothManager* wywołuje funkcję *GetUserCurrentClothSet*, która przyjmując jako parametr ID użytkownika, pobiera za pomocą usługi sieciowej listę ubrań. Następnie, po otrzymaniu rezultatu w formacie JSON, lista jest konwertowana (parsowana) do klasy *ClothSet*, która posiada metody pozwalające m.in. na podmianę, usuwanie ubrań, a także generowanie łańcucha znaków przedstawiającego konfigurację ubrań w zestawie. Tak wygenerowany łańcuch można z kolei przekazać do generatora postaci, aby otrzymać końcowy model.

Przykład pobrania aktualnego zestawu ubrań użytkownika za pomocą usługi sieciowej:

1. Wywołanie funkcji, która szuka obiektu aktora według podanego ID użytkownika, a następnie uruchamia wątek (*coroutine*).

```
public void GetUserCurrentClothSet(string userID)
{
    Actor[] actors = GameObject.FindObjectsOfType(typeof(Actor)) as Actor[];
    foreach (Actor actor in actors)
    {
        if (actor.userID == userID)
            StartCoroutine(GetUserCurrentClothSet_Thread(actor));
    }
}
```

2. Asynchroniczne skorzystanie z usługi sieciowej, pobranie oraz zdekodowanie rezultatu zapytania z formatu JSON do wewnętrznych klas projektu.

```
private IEnumerator GetUserCurrentClothSet_Thread(Actor actor)
{
    string url = GameClient.ROOT_WEB_URL +
"Services/DBServices.svc/GetUserClothSet?userID=" + actor.userID;
    WWW www = new WWW(url);
    yield return www;

    string json_result = www.text;
    Hashtable result = (Hashtable)(MiniJSON.JsonDecode(json_result));

    ClothSet clothSet = ClothManager.GetDefaultClothSet(actor.gender);
    foreach (Hashtable cloth in result["ClothesList"] as ArrayList)
    {
        if (cloth != null)
            clothSet.ChangeCloth(ClothManager.ParseStringToCloth(cloth));
    }

    actor.currentClothSet = clothSet;
    actor.RefreshCurrentClothes();
}
```

Po zakończeniu operacji następuje odświeżenie aktualnego zestawu ubrań, aktualizacja interfejsu szafy (miniaturek ubrań w odpowiednich slotach) oraz wyświetlenie (wygenerowanie) modelu awatara, składającego się z nowych ubrań za pomocą funkcji *DisplayCurrentClothSet*. W definicji tej funkcji z kolei znajduje się wywołanie generatora postaci, ładującego nową konfigurację ubrań za pomocą łańcucha znaków, dostarczonego przez obiekt typu *ClothSet*, którym jest *currentClothSet* należący do obiektu aktora.

Załadowane ubrania użytkownik może naturalnie zmieniać w dowolnym momencie poprzez modyfikację obiektu *currentClothSet* i użycie jej metod: *ChangeCloth* lub *RemoveCloth*. Obie metody dokonują zmian na liście ubrań na podstawie dostarczonych argumentów, którymi są obiekty typu *Cloth*. Tak zmieniony zestaw ubrań musi ponownie zostać odświeżony aby wygenerować nowy model awatara. Zmiany muszą także zostać zapisane w bazie, aby przy następnym wejściu gracza do świata, założone ubrania zostały zachowane. Odbywa się to poprzez wykorzystanie funkcji *SetUserCurrentClothSet* z globalnego obiektu klasy *ClothManager*. Należy również pamiętać, aby inni użytkownicy także zobaczyli zmianę ubrań, dlatego trzeba ich o tym powiadomić, za pomocą serwera gniazd. Użytkownicy dostając informację o zmianach w zestawie ubrań innego użytkownika, mogą samodzielnie odświeżyć jego model, pobierając z bazy danych aktualną konfigurację (która oczywiście musi wcześniej zostać tam zapisana).

Użytkownik oprócz listy aktualnie założonych ubrań, posiada również listę wszystkich posiadanych ubrań, która tworzy szafę użytkownika. Lista ubrań jest podzielona na kategorie (względem slotów ubrań) i jest pobierana przez menadżera w podobny sposób, przy użyciu usługi sieciowej. Lista jest następnie używana przez interaktywny interfejs graficzny szafy, aby w przystępnej formie, w postaci listy, miniaturek ubrań oraz slotów umożliwić podmianę ubrań.

Rys. 3.3 Widok szkieletu postaci z nałożonymi modelami ubrań w programie 3ds Max

źródło: opracowanie własne

3.3.3 KONFIGURACJA APARTAMENTU AWATARA

Użytkownik znajdujący się w swoim apartamencie może wejść w tryb jego edycji aktywując przycisk akcji za to odpowiedzialny. Odblokowany zostaje wtedy kreator (obiekt typu *Planner*), który stanowi stały, globalny element aplikacji. Aby jego odblokowanie było możliwe, właścicielem apartamentu, w którym przebywa użytkownik musi być on sam oraz podczas aktywacji kreatora należy przypisać do niego dany obiekt apartamentu. Odbywa się to w momencie załadowania lokacji – apartamentu.

Każdy apartament posiada w bazie danych listę mebli, które się w nim znajdują. Analogicznie do ubrań, tworzą one aktualny zestaw, który jednak nie jest przypisany do awatara, lecz do jego apartamentu. Podczas załadowania lokacji, która jest apartamentem, następuje skorzystanie z funkcji *GetUserCurrentFurnitureSet* z globalnego obiektu menadżera mebli typu *FurnitureManager*, która za pomocą usługi sieciowej pobiera listę mebli w danym apartamencie. Po pobraniu rezultatu następuje jego parsowanie do obiektu typu *FurnitureSet*, który posiada listę obiektów typu *Furniture* oraz metody na nich operujące, np. *AddFurniture* oraz *RemoveFurniture*. Po załadowaniu listy mebli w apartamencie, menadżer scen inicjalizuje ładowanie ich modeli, które rozpoczyna się od zakolejkowania ich zasobów w cache'u modeli, aby uniknąć kilkukrotnego pobrania tego samego elementu oraz by pobrane modele mogły być natychmiastowo użyte. Kiedy model mebla zostaje załadowany, następuje jego umieszczenie na scenie apartamentu. Czynność tę wykonuje obiekt kreatora, który odpowiedzialny jest za wszystkie możliwe działania dotyczące konfigurowania mieszkania, bez względu na to czy są to działania automatyczne dokonywane przez menadżer scen, czy bezpośrednie działania użytkownika w trybie edycji. Wykorzystana zostaje funkcja *SpawnFurniture*, która instancjonuje obiekt mebla na scenie o pozycji i obrocie, których dane zostały pobrane z bazy. Do modelu (a właściwie prefabrykatu) zostaje dodany także skrypt *EditableFurniture*, który umożliwia (podczas aktywnego trybu edycji apartamentu) modyfikowanie parametrów mebla względem apartamentu.

W momencie włączenia kreatora, wszystkie obiekty typu *ActionSpot* zostają ukryte, aby podczas edycji apartamentu użytkownik nie mógł usiąść na żadnym obiekcie, ani nie mógł wykonać na nim żadnej interakcji, która odbywa się w normalnym trybie. W trybie edycji istnieje za to możliwość zaznaczenia obiektu poprzez kliknięcie na nim. Po zaznaczeniu wyświetlają się dodatkowe elementy, które służą za część interfejsu kreatora – okrąg ze strzałkami otaczający model mebla oraz dwie ikony 2D unoszące się nad modelem. Okrąg posiada skrypt *RotateCircle* i jest częścią edytowanego mebla, który posiada komponent *EditableFurniture*. Okrąg ten służy do obrotu modelu poprzez naciśnięcie i przytrzymanie przycisku myszy. Trzymając przycisk i poruszając myszą w lewo lub w prawo nadawany zostaje odpowiedni kąt obrotu. Po puszczeniu przycisku następuje natychmiastowe odwołanie się do komponentu *EditableFurniture* i wywołanie funkcji *Move*, która nadaje meblowi nową pozycję i obrót (względem danego apartamentu), których wartości zostają zapisane do bazy danych za pomocą usługi sieciowej. Takie samo wywołanie następuje również oczywiście podczas przesuwania mebla, do którego prowadzi kliknięcie i przytrzymanie przycisku myszy na modelu mebla. Wówczas, podczas przeciągania myszy, model mebla przesuwa się w wybranym kierunku, równolegle do podłoża. Natomiast ikony, które są widoczne nad modelem mebla, kiedy jest on zaznaczony/aktywny służą do klonowania oraz usuwania mebla. Podczas klonowania obiektu mebla, tworzony jest zupełnie nowy obiekt, który posiada własny, unikalny identyfikator w bazie danych. Reszta parametrów zostaje jednak przekopiowana, gdzie klonowany obiekt mebla staje się wzorcem, na podstawie którego przepisuje się przede wszystkim nazwę zasobu (modelu), kategorię oraz typ mebla. Kolejnym elementem interfejsu kreatora jest lista mebli, która analogicznie do listy ubrań przedstawia cały zbiór mebli posiadanych przez użytkownika. Użytkownik w dowolnym momencie może przeciągnąć miniaturkę mebla z listy na scenę apartamentu, w celu umieszczenia (instancjonowania) mebla. Podczas kliknięcia na miniaturkę pojawia się załadowany model danego mebla i przeciągając kursorem myszy można go przesuwać nadając mu początkowe położenie. W tym momencie wykorzystywana jest podobna akcja tworzenia (instancjonowania) mebla w apartamencie, co w przypadku klonowania, lecz tym razem wzorcem staje się obiekt mebla znajdującego się na liście posiadanych przez użytkownika. W obu jednak przypadkach tworzony jest nowy bazodanowy obiekt mebla, któremu zostaje przydzielone nowe unikalne ID. Taki obiekt może następnie zostać dodany do listy mebli danego apartamentu. Odpowiadają za to funkcje *AddFurniture* oraz *DuplicateFurniture* w klasie *ApartmentManager*, których wywołanie powoduje uruchomienie usługi sieciowej. Odpowiednie metody usługi z kolei tworzą nowy obiekt na podstawie obiektu, którego ID zostaje dostarczone poprzez argument metody. Obiekt o podanym ID zostaje wyszukany (w zależności od wywoływanej metody) na liście posiadanych mebli użytkownika lub na liście mebli znajdujących się w danym apartamencie.

Meble oprócz podziału na kategorie, do których przypisuje się je głównie pod względem domniemanej funkcjonalności, są także rozróżniane względem dwóch parametrów: umiejscowienia (*Location*) oraz zachowania (*Behaviour*). Określenie tych parametrów pozwala na umieszczenie mebla w apartamencie w sposób przewidywany przez użytkownika. Parametr *Location* może przyjąć trzy wartości: *Ground, Walls, Ceiling*. Wartości te są na stałe przypisane do przedmiotu w bazie danych i oznaczają (w tej kolejności) umiejscowienie mebla na podłodze, na ścianie lub na suficie. Tym sposobem model kanapy można umieścić tylko na podłodze, półkę z książkami na ścianie, a lampę (wiszącą) na suficie. Do

zidentyfikowania poszczególnych części apartamentu w Unity zastosowano warstwy, odpowiadające poszczególnym miejscom docelowym. Mechanizm odpowiedzialny za instancjonowanie i przesuwanie obiektów na scenie korzysta z tych warstw, aby rozpoznać siatki modeli, na które składa się model apartamentu, tak aby dokonywać przesunięć mebli na ich powierzchni. Pobieranie punktu/pozycji 3D z powierzchni modelu (oraz informacji o nim) na podstawie pozycji 2D kursora myszy następuje poprzez tzw. rzucanie promieni – *Raycast*, jako funkcja statyczna wbudowanej w Unity klasy *Physics*.

Fragment kodu z ustalaniem pozycji (*newPos* + *offset*) i obrotu obiektu na warstwie *Walls*:

```
if (this.layer == Layer.Walls)
{
    if (Physics.Raycast(ray, out targetHit, Mathf.Infinity, wallsCollisionLayer))
    {
        newPos = targetHit.point;
        Quaternion targetRotation = Quaternion.identity;
        if (targetHit.collider.gameObject.transform.rotation.eulerAngles.x == 270)
            targetRotation = Quaternion.FromToRotation(targetHit.transform.up,
targetHit.normal);
        else
            targetRotation = Quaternion.FromToRotation(-
targetHit.transform.forward, targetHit.normal);

        this.transform.rotation = Quaternion.Euler(0, targetRotation.eulerAngles.y,
this.transform.rotation.eulerAngles.z);

        if (Vector3.forward == transform.forward)
            offSet = new Vector3(currentOffSet.x, currentOffSet.y, 0);
        if (Vector3.forward == -transform.forward)
            offSet = new Vector3(currentOffSet.x, currentOffSet.y, 0);
        if (Vector3.right == transform.forward)
            offSet = new Vector3(0, currentOffSet.y, currentOffSet.z);
        if (Vector3.right == -transform.forward)
            offSet = new Vector3(0, currentOffSet.y, currentOffSet.z);

        canMove = true;
    }
}
```

Drugi parametr obiektu mebla – *Behaviour*, przyjmujący wartości *Default* (domyślny) lub *Item*, określa zachowanie się obiektu na scenie i jest przeważnie stosowany tylko dla obiektów z warstwy *Ground*. Wartość *Item* wyróżnia fakt, że przypisuje się ją do obiektów mniejszych, np. akcesoriów, które mogą zostać umieszczone na innych meblach w apartamencie. Za przykład może posłużyć telewizor umieszczony na stoliku. Stolik jest większym meblem, który zawsze stoi na podłodze, więc może przyjąć wartość *Default*. Natomiast telewizor jest obiektem, który może, a nawet powinien być postawiony na czymś wyższym, np. stoliku – wtedy pole *Behaviour* przyjmuje wartość *Item*. Technicznie zostało to rozwiązane również za pomocą rzucania promieni, gdzie sprawdzana i ustalana jest odległość od podłoża z uwzględnieniem obiektów znajdujących się pomiędzy podłożem a obiektem typu *Item*.

Konfigurator apartamentu pozwala na jego edycję w czasie rzeczywistym, nawet w momencie, kiedy awatar użytkownika znajduje się w nim i porusza pomiędzy dodawanymi i przesuwanymi meblami. Należało jednak pamiętać, że w apartamencie oprócz użytkownika –

właściciela mogą przebywać także inni użytkownicy, którzy zostali do niego wcześniej zaproszeni. Jeśli właściciel apartamentu na bieżąco może obserwować zmiany jakie wprowadza w swoim mieszkaniu, inni użytkownicy także powinni mieć taką możliwość aby nie następowały żadne różnice w interakcji oraz problemy z kolizjami i pozycją awatarów. W tym celu zdecydowano się na pełną synchronizację czynności kreatora z widokiem danego apartamentu po stronie innych użytkowników. Oznacza to, że użytkownik będące w apartamencie gościem widzi kiedy właściciel dodaje mebel, przesuwa go po podłożu, obraca, klonuje, bądź usuwa. Dzięki temu wszyscy użytkownicy przebywający w apartamencie są na bieżąco ze zmianami. Synchronizacja odbywa się dzięki dwóm aspektom: natychmiastowym zapisie do bazy danych wszelkich zmian dokonywanych w apartamencie oraz natychmiastowym wysyłaniu poprzez serwer gniazd informacji o zmianach.

3.3.4 *INTERAKCJA Z UŻYTKOWNIKAMI I OTOCZENIEM*

Vego 3D w obecnej postaci nie oferuje zbyt wiele pod kątem interakcji ze światem i nie posiada rozbudowanej funkcjonalności pozwalającej na większą grywalność i na dłuższe zainteresowanie użytkowników. Niemniej gra stanowi ku temu solidną podstawę, a jej obecne mechanizmy odpowiadają wykonywaniu podstawowych czynności. Najważniejszym elementem gry sieciowej jest komunikacja z innymi graczami oraz możliwość interakcji z nimi, wpływanie na ich zachowanie, wspólne tworzenie pewnej społeczności skupionej wokół świata gry.

Interakcja z użytkownikiem rozpoczyna się przede wszystkim od momentu kliknięcia na jego awatara. Po najechaniu na niego kursorem myszy model awatara podświetla się (zmienia się materiał modelu), co stanowi pewną zachętę i w domyśle wskazuje na interaktywność danego modelu. Po kliknięciu na niego pojawia się okno menu, które obecnie zawiera odnośniki do dwóch akcji, jakie można wykonać na danym użytkowniku – zaproszenie do czatu oraz zaproszenie do apartamentu. Obie akcje są inicjowane poprzez wysłanie zaproszenia do danego użytkownika i są odkładane do momentu, kiedy nadejdzie odpowiedź – pozytywna lub negatywna. Po obu stronach wyświetlane są odpowiednie komunikaty w postaci okien dialogowych z przyciskami decyzyjnymi.

Interakcja z otoczeniem działa na podobnej zasadzie, co interakcja z awatarami innych użytkowników i sprowadza się do klikania na interaktywne obiekty, które podświetlają się po najechaniu na nie kursorem myszy. Podświetlenie oraz etykieta wyświetlająca nazwę danego obiektu są zatem uniwersalną i jednoznaczną wskazówką, że po kliknięciu zachodzi pewna akcja. Na obecnym etapie jedyną interaktywną akcją, którą może zainicjować użytkownik na obiekcie będącym częścią lokacji jest siadanie, np. na krześle lub kanapie. Obiekt dający taką możliwość posiada komponent/klasę *SittingSpot*. Klasa ta posiada pole *type* określające typ miejsca spośród wartości enumerycznych: *Chair, Sofa, Bench*. Klasa *SittingSpot* jest pochodną klasy *ActionSpot*, która w założeniach miała stanowić bazę również dla innych klas, odpowiedzialnych za inne sposoby interakcji, lecz posiadających wspólne cechy dotyczące zachowania awatara. Jedną z takich cech jest pozycja miejsca (*spot*), z którą związany jest awatar wchodzący w interakcję – może się znajdować bezpośrednio w tym miejscu lub z

pewnym przesunięciem (*offset*). Awatar podczas wykonywania akcji (np. siedzenie) na danym miejscu jest zamrożony, czyli nie może się poruszać ani wykonywać żadnych animacji, poza tymi, które są definiowane poprzez parametr klasy *ActionSpot*. Kolejną właściwością jest „zajętość" danego miejsca przez użytkownika, co oznacza, że miejsce będące zajętym, uniemożliwia interakcję innego użytkownika. Za przykład może posłużyć *spot* krzesła, na którym nie mogą jednocześnie siedzieć dwie osoby. Podczas interakcji z obiektem, użytkownik zostaje do niego przypisany (klasa *Actor* zawiera referencję do aktualnego obiektu typu *ActionSpot*) – po „wejściu" na obiekt, informacja o tym zdarzeniu jest rozsyłana do wszystkich pozostałych użytkowników przebywających w danej lokacji, tak aby oznaczyć to miejsce jako już niedostępne dla nich.

Obiekty typu *ActionSpot* mogą zostać rozbudowane do obiektów klas pochodnych, które będą korzystać z idei oraz podstawowej mechaniki interaktywnych obiektów – miejsc (*spotów*). W ten sposób można zrealizować przede wszystkim czynności uruchamiające animacje awatara, który wykonuje pewną akcję na obiektach lokacji, np. gra na automacie, pije napój przy stoliku, naciska przycisk windy itp. Wszystkie te akcje mogą współdzielić zasadę działania – unieruchamiać i ustawiać awatara w danej pozycji, a potem inicjować czynność, która może zostać dowolnie rozbudowana. Dodatkowo, co zostało już wykorzystane przy obiektach posiadających komponent *SittingSpot*, obiekty interaktywne mogą posiadać więcej niż jeden obiekt miejsca (*spotu*), np. sofa – co umożliwia zajęcie jednego z kilku dostępnych miejsc, a obszar kolizji obiektu jest podzielony na kilka mniejszych. Podział pozwala na wykrycie kliknięcia na pojedyncze miejsce w ramach całego obiektu.

3.4 SERWER GRY I KOMUNIKACJA

Komunikacja pomiędzy użytkownikami w czasie rzeczywistym w Vego 3D odbywa się za pomocą serwera gniazd, którym jest Photon Network Engine. Photon jest rozbudowanym narzędziem, szkieletem do budowy aplikacji sieciowych posiadającym interfejsy programowania (API) dla klienta oraz serwera. Korzystając z API został stworzony moduł sieciowy w kliencie gry oraz aplikacja serwerowa, która została uruchomiona na serwerze w celu obsługi wszystkich klientów podłączających się do gry. Photon wraz z zestawem narzędzi oraz bibliotek (SDK) oferuje dokumentację oraz przykłady, które pomagają zrozumieć zagadnienie w przystępny sposób i umożliwiają dość szybkie stworzenie modułu sieciowego w ramach własnego projektu.

W Vego 3D za bazę aplikacji serwerowej posłużył jeden z przykładów przedstawiający obsługę gry typu MMO, który zawierał pewne mechanizmy sprzyjające konstrukcji świata i mechaniki Vego 3D. Źródła zostały przebudowane i zaadaptowane do solucji całego projektu (zgodnie z licencją) i umieszczone w oddzielnym projekcie pod przestrzenią nazw *Vego3D.Server*. Idea komunikacji pozostała ta sama, lecz aplikacja została znacznie rozbudowana, aby sprostać wymogom funkcjonalnym projektu. Cała aplikacja jest biblioteką DLL, która po skompilowaniu jest wysyłana na serwer i dynamicznie dołączana do Photona, a następnie uruchamiana podczas startu serwera.

Idea konstrukcji wirtualnego świata pod względem obsługi sieciowej została ujęta w sposób dosłowny, lecz przy użyciu zmodyfikowanego tradycyjnego mechanizmu pokojów, który często stanowi podstawę konstrukcji rozgrywki sieciowej w grach. Różnica polega jednak na tym, że użytkownicy (ani nikt inny) nie mają kontroli nad procesem tworzenia pokojów, ani nie istnieje żadna tzw. poczekalnie, w której dostępne byłyby pokoje. Zastosowany w Vego 3D mechanizm gry MMO podnosi rangę pokoju do idei świata – dużego pokoju, który na potrzeby projektu ogranicza się do pojedynczej lokacji, która może być dowolnie duża i obsługiwać dowolną liczbę użytkowników. Przy takim podejściu należało jednak pamiętać, że żaden klient nie byłby w stanie poradzić sobie z wyświetlaniem treści graficznej oraz komunikacji ze wszystkimi użytkownikami w obrębie ogromnej lokacji – świata. Dlatego też w celach optymalizacyjnych wykorzystano wbudowane mechanizmy tzw. obszaru zainteresowania (*Interest Area)*, które pozwalają na dostarczanie użytkownikowi danych z serwera dotyczących wyłącznie obszaru, który znajduje się najbliżej użytkownika. W efekcie oznacza to, że gracz nie będzie widział awatarów innych użytkowników w dalszej odległości, gdyż zwyczajnie go nie „interesują". Obszar świata/lokacji jest definiowany przez obiekt klasy *MmoWorld*, który tworzony jest przy pierwszej próbie wejścia użytkownika do lokacji o danej nazwie. Jedną z podstawowych cech świata jest wirtualny obszar, składający się z tzw. kafli o podanym rozmiarze, w których mogą znajdować się użytkownicy posiadający własną pozycję względem tego obszaru.

Mechanizm komunikacji, czyli wymiany danych, pomiędzy użytkownikami, a także serwerem a użytkownikami, zgodnie z API został zrealizowany na zasadzie operacji oraz zdarzeń. Operacje są to metody, których definicje znajdują się w aplikacji serwerowej, a wywołania następują po stronie klienta na zasadzie wywołania zdalnych procedur. Operacje mogą zwracać rezultaty, których może oczekiwać klient. Ich identyfikacja odbywa się poprzez wykorzystanie kodów (tzw. kodów operacji), których lista znajduje się w klasie *OperationCode* w przestrzeni nazw *Vego3D.Common*. Za przykład może posłużyć operacja *Move*, która jest wywoływana po stronie klienta, za każdym razem kiedy użytkownik poruszy (zmieni pozycję) swoim awatarem. Argumentami tej operacji jest pozycja oraz obrót awatara, które są zapisywane na serwerze, a następnie rozsyłane do pozostałych użytkowników, którzy przebywają w tej lokacji. Przesyłanie danych z serwera do klienta odbywa się z kolei na zasadzie zdarzeń. Zdarzenia są pewnego rodzaju wyzwalaczami, które niosą ze sobą informacje i inicjują po stronie klienta odpowiednią reakcję. Zdarzenia najczęściej są wywoływane na serwerze i są efektem przeprowadzenia operacji. Podobnie jak operacje, zdarzenia identyfikuje się również za pomocą kodów, które zapisane są w klasie *EventCode*. Przestrzeń nazw *Vego3D.Common,* ogólnie rzecz biorąc, zawiera klasy, z których korzysta nie tylko aplikacja serwerowa, ale także kliencka, np. współdzieląc kody operacji, zdarzeń, parametrów oraz inne typy enumeryczne. Przykład zdarzenia może stanowić *ItemMoved*, które po wykonaniu operacji *Move* na serwerze dostarcza wszystkim użytkownikom w świecie informację o nowej pozycji i obrocie awatara użytkownika (klienta) o podanym ID. Po otrzymaniu tego zdarzenia, każdy klient odświeża obiekt awatara tego klienta na podstawie danych, które zostały dostarczone przez to zdarzenie.

Schemat budowy aplikacji serwerowej odnosi się do pojedynczego obiektu klienta, łączącego się z serwerem i względem niego realizowana jest obsługa wszystkich operacji i zdarzeń. Naturalnie jednak w rezultacie, podczas działania aplikacji, obsługiwani są wszyscy

użytkownicy za pomocą takiego schematu. Klient, który nawiązał połączenie z serwerem reprezentowany jest za pomocą klasy *MmoPeer,* dziedziczącej po wbudowanej klasie *Peer*. Klasa ta zawiera definicje kilku pierwszych podstawowych operacji, które są przeprowadzane od momentu połączenia z serwerem do momentu wejścia do świata, m.in. *EnterWorld*. Operacja *EnterWorld* odpowiedzialna jest za stworzenie obiektu typu *MmoActor*, który zarządza komunikacją użytkownika znajdującego się już w świecie – zawiera definicje operacji. Następie tworzony jest obiekt typu *MmoItem*, który bezpośrednio odpowiada wizualnemu obiektowi awatara i posiada takie cechy, jak nazwa, pozycja, obrót, aktualna animacja, zajmowane miejsce (*spot*), aktualnie edytowany apartament itp. Aktor (*MmoActor*) operuje na tym obiekcie aktualizując jego dane i dostarczając informacje o nim. Na koniec, po wykonaniu wszystkich czynności w *EnterWorld*, operacja zwraca rezultat pozytywny, jeśli nie wystąpiły żadne błędy. Po odebraniu rezultatu przez klienta, który zainicjował wcześniej tę operację, rozpoczyna się przygotowanie do wejścia do wybranego świata/lokacji.

Źródła aplikacji klienckiej do komunikacji z serwerem Photon znajdują się z kolei w oddzielnym projekcie - *Vego3D.Client*, który (podobnie jak aplikacja serwerowa) jest kompilowany do biblioteki DLL. Biblioteka zostaje następnie dołączona do aplikacji Unity, która wykorzystuje ją do komunikacji z aplikacją serwerową – wywołuje operacje i nasłuchuje zdarzenia, których obsługa jest realizowana za pomocą implementacji interfejsu. Główną klasą biblioteki klienckiej jest klasa *Game*, która odpowiada za nawiązanie sparametryzowanego połączenia z serwerem, nasłuchiwanie przychodzących rezultatów i zdarzeń oraz wysyłanie operacji. Obsługa rezultatów i zdarzeń zrealizowana jest za pomocą aktualnego stanu, czyli jednej z klas implementujących interfejs *IGameLogicStrategy*. Każda tzw. strategia zawiera definicje funkcji, które realizują inną obsługę rezultatów i zdarzeń w zależności od stanu w jakim znajduje się klient – czy oczekuje na połączenie z serwerem, czy jest połączony, lub czy znajduje się już w świecie. Wspomniana wcześniej operacja *EnterWorld* zwraca rezultat, który wychwytywany jest w stanie *Connected*, gdzie następuje ustawienie aktualnego stanu na *WorldEntered* (za pomocą klasy *Game*). Następnie za pomocą odpowiedniego interfejsu informowana o tym jest aplikacja Unity, która zajmuje się wizualnym umieszczeniem awatara użytkownika w nowym świecie/lokacji. Służąca do tego klasa interfejsu *IGameListener* jest natomiast implementowana przez klasę *GameClient*, która znajduje się w projekcie aplikacji Unity i która jest jednocześnie główną klasą, odpowiedzialną za przetwarzanie (już na logikę gry) informacji otrzymanych z modułu klienckiego.

Po wejściu klienta w stan *WorldEntered* (kiedy awatar użytkownika pojawił się w lokacji) możliwy staje się do wywołania szereg operacji odpowiadających za rozsyłanie informacji związanych z awatarem, np. *Move*, która wywoływana jest za każdym razem, kiedy awatar (lokalny) zmieni pozycję lub obrót.

Przykład wywołania operacji po stronie klienta:

```
Operations.Move(this.Game, this.Id, this.Type, newPosition, newRotation,
this.Game.Settings.SendReliable);
```

Definicja funkcji inicjującej operację *Move* po stronie klienta:

```
public static void Move(Game game, string itemId, byte? itemType, float[] position,
float[] rotation, bool sendReliable)
{
    var data = new Hashtable { { (byte)ParameterCode.Position, position } };
    if (itemId != null)
    {
        data.Add((byte)ParameterCode.ItemId, itemId);
    }

    if (itemType.HasValue)
    {
        data.Add((byte)ParameterCode.ItemType, itemType.Value);
    }

    if (rotation != null)
    {
        data.Add((byte)ParameterCode.Rotation, rotation);
    }

    game.SendOperation(OperationCode.Move, data, sendReliable,
Settings.ItemChannel);
}
```

Wszystkie operacje po stronie klienta dostępne są jako statyczne funkcje klasy *Operations* w przestrzeni nazw *Vego3D.Client* i mogą być wywołane w dowolnym miejscu, w zależności od potrzeb. Powyższa funkcja inicjująca operację *Move* dla obiektu awatara posiada wywołanie w klasie *MyItem* (w *Vego3D.Client*), która odpowiada za sieciową logikę lokalnego obiektu awatara (lub ew. innych obiektów należących do użytkownika). Wywołania operacji mogą jednak występować również bezpośrednio w logice gry (przestrzeń *Vego3D.Client.Main* – projekt Unity), tak jak ma to miejsce w przypadku interfejsu graficznego, gdzie wyświetlane są komunikaty dotyczące zaproszeń. Realizacja zaproszeń do prywatnego czatu (podobnie jak do apartamentu użytkownika) odbywa się za pomocą trzech operacji: *ChatInvite*, *ChatAccept*, *ChatReject*, które są wywoływane przy naciśnięciu odpowiednich przycisków w GUI.

Operacje, których definicje znajdują się w aplikacji serwerowej, odpowiedzialne są w pierwszej kolejności za wyszukanie w świecie klienta o podanym ID (dostarczonym jako argument operacji), a następnie wykonaniu na nim właściwej operacji. W przypadku operacji *Move* o jej wykonaniu (czyli zaktualizowaniu pozycji i obrotu awatara danego użytkownika) zostają poinformowani wszyscy użytkownicy znajdujący się w danej lokacji. Natomiast w przypadku operacji dotyczących m.in. zaproszeń, gdzie komunikacja odbywa się tylko pomiędzy dwoma użytkownikami, o wyniku operacji jest informowany tylko drugi użytkownik, który jest celem tej operacji. Wtedy tylko jemu zostaje przekazana odpowiednia informacja w postaci zdarzenia, np. *ChatInvited* lub *ApartmentInvited*.

W przypadku operacji, które dotyczą bezpośrednio awatara użytkownika i jego właściwości, ich obsługą zajmuje się klasa *MmoActor*, która w pierwszej części weryfikuje istnienie klienta o podanym ID, a następnie wywołuje funkcję, która przeprowadza właściwą operację – na obiekcie typu *MmoItem*. W przypadku operacji *Move* jest to zapisanie wartości do obiektu serwerowego, aby umożliwić późniejsze pobranie wszystkich danych o obiekcie.

Definicja operacji *Move* po stronie serwera wywołanej zdalnie przez klienta:

```
[Operation(OperationCode = (byte)OperationCode.Move)]
public OperationResponse OperationMove(Peer peer, OperationRequest request)
{
    var operation = new Move(request);
    if (!operation.IsValid)
    {
        return new OperationResponse(request,
(int)ErrorCode.InvalidOperationParameter, operation.GetErrorMessage());
    }

    Item item;
    if (false == operation.ItemType.HasValue ||
string.IsNullOrEmpty(operation.ItemId))
    {
        item = this.Avatar;
        operation.ItemId = item.Id;
        operation.ItemType = item.Type;
    }
    else if (this.TryGetItem(operation.ItemType.Value, operation.ItemId, out item)
== false)
    {
        return operation.GetOperationResponse((int)ErrorCode.ItemNotFound,
"ItemNotFound");
    }
    return this.ItemOperationMove((MmoItem)item, operation);
}
```

Właściwa realizacja operacji oraz rozesłanie zdarzenia (*ItemMoved*):

```
private OperationResponse ItemOperationMove(MmoItem item, Move operation)
{
    MethodReturnValue result = this.CheckAccess(item);
    if (result)
    {
        float[] oldPosition = item.Coordinate;
        float[] oldRotation = item.Rotation;
        item.Move(operation.Position);
        item.Rotation = operation.Rotation;

        var eventInstance = new ItemMoved
        {
            ItemId = item.Id,
            ItemType = item.Type,
            OldPosition = oldPosition,
            Position = operation.Position,
            OldRotation = oldRotation,
            Rotation = operation.Rotation
        };

        var message = new ItemEventMessage(item,
eventInstance.GetEventData((byte)EventCode.ItemMoved,
operation.OperationRequest.Reliability, Settings.ItemEventChannel));
        item.EventChannel.Publish(message);

        operation.OperationRequest.OnCompleted();
        return null;
    }
    return operation.GetOperationResponse(result);
}
```

3.5 INSTALACJA I KONFIGURACJA

Vego 3D, jako gra sieciowa działająca pod przeglądarką internetową, wymaga opublikowania na serwerze WWW klienta gry wraz z całym serwisem, który go obudowuje i który jest nieodłączną częścią gry. Z racji wykorzystania technologii ASP .NET do stworzenia serwisu WWW, wymagany jest odpowiedni hosting obsługujący technologie Microsoft, który dodatkowo oferuje dostęp do systemu baz danych Microsoft SQL Server. Z kolei aplikacja serwerowa, która korzysta z Photon Network Engine, musi zostać zainstalowana na serwerze dedykowanym, gdyż żaden przeciętny i darmowy hosting WWW nie oferuje możliwości pełnego dostępu do systemu, na którym należy zainstalować Photon Network Engine. Nic nie stoi jednak na przeszkodzie, aby serwer dedykowany, na którym postawiony jest Photon, użyć także jako serwer WWW, na którym zostanie umieszczony klient gry. Zatem cała gra oraz oprogramowanie potrzebne do jej działania może znaleźć się w jednym miejscu, na jednym serwerze i tak też zostało to zrealizowane w praktyce podczas instalacji projektu Vego 3D ramach testów.

W efekcie cały projekt został umieszczony na serwerze pod adresem IP *188.165.231.45*, pod którym dostępny jest serwis WWW na protokole HTTP – pod adresem *http://188.165.231.45/Vego3D*. Serwer jest dedykowany, na którym został zainstalowany system Windows Server 2008, oferujący zdalny dostęp przez protokół RDP. Jako oprogramowanie serwera WWW użyto IIS 7.0, którego skonfigurowano do działania na ASP .NET w wersji 4.0, aby możliwa była współpraca z najnowszą wersją ADO .NET Entity Framework 4 i metodą *Code-First*. Skonfigurowano w projekcie również połączenie z bazą danych, aby korzystało z właściwej instancji serwera bazy uruchomionej na serwerze dedykowanym. Aktualizacja plików serwisu WWW odbywa się poprzez system kontroli wersji lub skonfigurowane konto FTP, lecz wygodniej byłoby jednak skorzystać z automatycznego wdrażania bezpośrednio z poziomu środowiska Visual Studio.

Na tym samym serwerze zainstalowano również Photon Network Engine, który działa jako usługa systemowa i ma włączony autostart (na wypadek nieplanowanych restartów serwera). Photon został aktywowany darmową licencją, która nie wprowadza ograniczeń funkcjonalnych, a jedynie posiada limit do 100 jednoczesnych połączeń klientów. Do poprawnego działania serwera gniazd należało skonfigurować zaporę sieciową, aby odblokować potrzebne porty: TCP – 4530, 843, 943 oraz UDP – 5055. Tym samym spełniono wymagania opisane pod adresem *http://doc.exitgames.com/v2/overview/requirements/*. Następnie do folderu z aplikacjami Photona skopiowano folder z wygenerowanymi bibliotekami aplikacji serwerowej Vego 3D oraz skonfigurowano parametry jej uruchamiania. W celu ułatwienia aktualizacji plików aplikacji, podobnie jak w przypadku serwisu WWW, zastosowano system kontroli wersji bazujący na oprogramowaniu VisualSVN Server.

Do uruchomienia klienta gry z kolei nie jest potrzebna żadna instalacja, oprócz jednorazowej (być może dokonanej przez użytkownika już wcześniej) instalacji wtyczki Unity Web Player. Vego 3D jest grą internetową dostępną z poziomu przeglądarki internetowej, co wyklucza jakąkolwiek inną instalację lub konfigurację.

4 PODSUMOWANIE

Nie byłoby do końca prawdą stwierdzenie, iż projekt gry społecznościowej Vego 3D został ukończony, ponieważ natura tego typu produkcji pokazuje, jak wiele pracy trzeba włożyć w tak duże przedsięwzięcie, które zasadniczo i tak nigdy nie zostaje ukończone. Rozbudowane gry sieciowe oferujące bogate wirtualne światy i skomplikowaną mechanikę są często długoterminową inwestycją, z którą wiąże się nieustanny rozwój, uzupełnianie i dodawanie nowych treści, poprawianie istniejących elementów gry, balansowanie rozgrywki, dbanie o użytkowników, stawianie przed nimi wyzwań i nagradzanie ich. Siłą gier sieciowych jest wirtualna społeczność, która tworzy świat gry i sprawia, że żyje on własnym życiem.

Vego 3D było próbą stworzenia zaplecza technicznego oraz podstawowej mechaniki dla tego typu produkcji i z czystym sumieniem można stwierdzić, że prawie wszystkie założenia zostały zrealizowane, z całkiem dobrym skutkiem. Projekt stanowi nowoczesny model gry sieciowej, zarówno pod względem koncepcyjnym jak i pod względem rozwiązań technicznych. Największą zaletą i niemałą rewolucją jest przeniesienie gry pod przeglądarkę, co stwarza duży potencjał pod względem zachęcenia i przyciągnięcia uwagi wielu użytkowników, także (a może przede wszystkim) tych, którzy są zbyt leniwi lub słabo obeznani, aby pobrać i zainstalować grę na dysku. Vego 3D pozwala na rozpoczęcie gry już w niecałą minutę od wejścia na stronę WWW.

Celem niniejszej pracy było nie tylko stworzenie podstaw do zbudowania dużej gry sieciowej, ale także udowodnienie, że jest to możliwe do osiągnięcia przy sporych ograniczeniach czasowych i narzędziowych, niezbyt dużej wiedzy z tej dziedziny oraz nawet możliwe do zrealizowania przez jedną osobę. Jeszcze niedawno wydawało się to całkiem nieprawdopodobne, aby produkcje MMO były tworzone przez małe zespoły, gdyż projekty na tę skalę wymagają ogromnych środków czasowych, finansowych oraz odpowiedniego zaplecza technicznego i osobowego. Jednak zastosowanie właściwych technologii, które wspomagają pracę, zawierają wiele gotowych rozwiązań i pozwalają tworzyć (również prototypować) szybko i wygodnie optymalizuje pracę i pomaga skupić się na budowie mechaniki i rozwijaniu koncepcji projektu. To właśnie odpowiedni dobór narzędzi przyczynił się do realizacji i ukończenia założonych celów.

Najlepszą rekomendacją i oceną tej pracy jest fakt, iż większość zastosowanych w niej rozwiązań stało się już bazą dla komercyjnego produktu (tzw. startupu), który został uruchomiony pod koniec 2011 roku. W związku z tym, wiele rozwiązań technicznych w Vego 3D zostało wypracowanych wcześniej, zanim doszło do realizacji niniejszej pracy dyplomowej. To również pokazuje jak szeroki temat obejmuje ta praca i ile czasu zostało poświęconego zgłębieniu wszystkich opisanych w pracy rozwiązań, których i tak znaczna część została zastąpiona (względem wspomnianego komercyjnego projektu) przez inne, nowsze technologie.

BIBLIOGRAFIA

Uwaga: aktualność wszystkich zamieszczonych poniżej odnośników została zweryfikowana w dniu 21.01.2012 r.

[1] S. Walther, *ASP.NET 2.0. Księga eksperta*, Helion, Gliwice 2008

[2] Microsoft Developer Network, *Data Access and Modeling*, http://msdn.microsoft.com/en-us/library/951h6we4.aspx

[3] MSDN Blogs, *ADO.NET Team Blog*, http://blogs.msdn.com/b/adonet/

[4] S. Guthrie, *Code-First Development with Entity Framework 4*, http://weblogs.asp.net/scottgu/archive/2010/07/16/code-first-development-with-entity-framework-4.aspx

[5] Unity Technologies, dokumentacja i materiały referencyjne, http://unity3d.com/support/documentation/

[6] Unity Technologies, *Character Customization & AssetBundles*, http://blogs.unity3d.com/2009/11/25/character-customization-assetbundles/

[7] Exit Games, *Photon Network Engine Online Documentation*, http://doc.exitgames.com/v2/

[8] Modele 3D:

- http://turbosquid.com
- http://3dmodelfree.com
- http://archive3d.net
- http://artist-3d.com
- http://virtualworlds.wikia.com

[9] Grafika 2D:

- http://iconfinder.com

SPIS ILUSTRACJI