AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE
WYDZIAŁ INFORMATYKI, ELEKTRONIKI I TELEKOMUNIKACJI

KATEDRA INFORMATYKI

Praca dyplomowa magisterska

Narzędzia programistyczne wspierające proces testowania aplikacji opartych o kontroler Kinect

Autor: *Piotr Styrna*
Kierunek studiów: *Informatyka*
Opiekun pracy: *dr. inż. Jacek Dajda*

Kraków 2014

Oświadczam, świadomy odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonałem osobiście i samodzielnie i że nie korzystałem ze źródeł innych niż wymienione w pracy

//TODO Serdecznie dziękuję …. Tu ciąg dalszy podziękowań np. dla promotora, żony sąsiada itp.

Spis treści

1. Wstęp

> *„Jeśli debugowanie to proces usuwania błędów z oprogramowania to programowanie musi być procesem ich tworzenia.”* - Edsger Dijkstra

26 września 1983 roku system wczesnego ostrzegania w centrum dowodzenia radzieckich wojsk rakietowych uruchomił omyłkowo alarm jądrowy. Błąd spowodowany był niepoprawną analizą promieniowania słonecznego odbitego od chmur i prawie doprowadził do wystrzelenia 5 tys. rakiet w stronę Stanów Zjednoczonych oraz ich sojuszników. Błędy oprogramowania mogą manifestować w różny sposób. Od zwykłego okienka z błędem do przekłamania danych i dezinformacji. Aby wyeliminować bądź ograniczyć ich ilość powszechną praktyką jest testowanie.

Testowanie oprogramowania nie jest pojęciem nowym. Na przestrzeni czasu w różny sposób podchodzono do owego zagadnienia i przywiązywano do niego coraz większą wagę. W 1979 roku Glenford J. Myers po raz pierwszy wydzielił w procesie tworzenie software-u jego weryfikacje jako osobny etap. W późniejszych latach skupiano się głównie na destruktywnym testowaniu czyli próbie znalezienia błędów w końcowym produkcie. Współczesny cel testowania określa się jako prewencyjny gdyż jego celem jest zapewnienie iż oprogramowanie spełnia wymagania, jest odporne na błędy oraz łatwe w utrzymaniu.

Przyjmuje się iż w przybliżeniu połowa czasu produkcji software-u przeznaczona jest na jego walidację i weryfikacje, a sam proces testowy występuje w każdej fazie tworzenia aplikacji. Rzadko kiedy zdarza się iż oprogramowanie porzucane jest po ukończeniu pierwszej wersji. Coraz częstszym przypadkiem jest przyrostowy proces wytwarzania oprogramowania gdzie w kolejnych wersjach produktu (przykład Windows 98/XP/Vista/7/10) rozszerzane bądź modyfikowane zostają jego funkcjonalności. Główną zaletą testów w takim procesie jest ułatwienie utrzymania kodu i zwiększanie jego odporności na zmiany. Ważnym elementem w testowaniu jest automatyzacja, gdyż w miarę przyrostu funkcjonalności aplikacji zwiększa się jej obszar dziedzinowy a zatem ilość testów. Brak automatyzacji procesu testowego w dużych projektach wiąże się z potrzebą rozbudowy zespołów odpowiedzialnych za jakość oraz poświęcanie ich czasu na ciągłą weryfikację produktu przy każdej zmianie. Automatyzacji może ulec każdy rodzaj testu przy użyciu odpowiednich narzędzi wspierających ten proces. Większość środowisk do pisania oprogramowania posiada wbudowaną możliwość uruchomienia oraz debugowania napisanych testów jednostkowych, a serwery ciągłej integracji w łatwy sposób można skonfigurować do automatycznego przeprowadzania zbioru testów akceptacyjnych.

Niestety dla wielu bardziej specyficznych przypadków automatyzacja procesu testowego nie jest łatwa. Takim właśnie przypadkiem jest automatyzacja testów odpowiadających za interakcję z użytkownikiem. Jednym z głównych problemów testowania tego obszaru jest wielkość jego dziedziny. Tak prosty program jak Microsoft WordPad posiada 325 możliwych operacji GUI a posiada on minimum operacji pozwalających na edycje pliku tekstowego. Dla

bardziej złożonych programów ilość testów drastycznie rośnie. Wielkość aplikacji nie jest jedynym czynnikiem wpływającym na obszar testów. Duże znaczenie ma model sterowania. Najprostszą interakcją jest sterowanie przy użyciu przycisków czego najbardziej znanym przykładem jest klawiatura. Zakres zachowań które może wygenerować jest dość mały gdyż sprowadza się do naciśnięcia poszczególnych klawiszy bądź ich kombinacji. Kolejnym powszechnie znanym sposobem komunikacji jest użycie kursora poruszającego się na ekranie czego przykładem może być mysz komputerowa bądź joystic. Podobnym modelem sterowania są ekrany dotykowe powszechnie stosowane w urządzeniach mobilnych. W obu metodach zakres testów jest duży gdyż musi pokryć przestrzeń dwuwymiarową w jakiej może poruszać się kursor. Istnieją też bardziej zaawansowane modele sterowania choć nie są jeszcze powszechnie używane. Jednym z nich jest model sterowania w przestrzeni trójwymiarowej przy użyciu detekcji ruchów naszego ciała bądź jego części.

Rys 1.1: Kontrolery ruchu

Rysunek 1.1 przedstawia popularne kontrolery na rynku takie jak: Sony PlayStation Move, Microsoft Xbox 360 Kinect oraz Nintendo Wii. Początkowo taki model interakcji zastosowane znalazł głównie w grach video jednak coraz więcej projektów próbuje zintegrować model sterowania do swoich zastosowań.

1.1 Cel pracy

Jak nie trudno się domyślić proces testowania oprogramowania w którym interakcja oparta jest na detekcji ruchu ma ogromną dziedzinę testów gdyż sterowanie operuje w trzech wymiarach. W połączeniu z zakresem testów GUI (interfejsu użytkownika) zapewnienie funkcjonalności takiej aplikacji jest problemem nietrywialnym. Jako iż nie jest to jeszcze popularna metoda interakcji nie zawsze programista ma do dyspozycji narzędzia wspomagające ten proces.

Celem tej pracy jest zbadanie w jaki sposób konwencjonalne metody procesu testowego mogą zostać zastosowane przy pracy z kontrolerem ruchu. Jako przykład kontrolera wykorzystany zostanie popularny sensor Kinect od firmy Microsoft. Należy zanalizować warunki oraz problemy występujące podczas pracy z kontrolerem. Zbadać istniejące metody i narzędzia wspierające proces testowania oprogramowania używającego kontrolera oraz zaproponować i stworzyć własny model narzędzi rozwiązujące wykryte problemy. Pokazać że pomimo faktu iż sterowanie przy pomocy sensora jest dość nietypowym modelem interakcji to części aplikacji odpowiadające za tę mechanikę mogą ulec normalnemu procesowi weryfikacji oraz walidacji.

Aby zrealizować przedstawiony cel pracy przyjęty został poniższy plan badań:

1. **Zbadanie kontrolera Kinect** – jako tytułowe narzędzie pracy pierwszym krokiem realizacji celu jest zbadanie jego możliwości. Należy przedstawić jego budowę oraz sposób działania. Następnie poprzez napisanie paru przykładowych programów bądź analizę istniejących aplikacji zaznajomić się z API dostarczonym przez kontroler. Szczególny nacisk należy zwrócić na wszelkie problemy i niedogodności podczas programowania i używania oraz testowania.
2. **Analiza istniejących narzędzi wspomagających proces testowania** – wyszukanie już istniejących narzędzi realizujących problemy wykryte w pierwszym kroku. Zbadanie czy a jeśli tak to w jakim stopniu znalezione narzędzia są w stanie realizować problemy bądź ułatwić proces testowy. Porównać znalezione narzędzia ze sobą oraz z własnym rozwiązaniem.
3. **Stworzenie narzędzi wspomagających proces testowania** – zaproponowanie koncepcji narzędzi realizujących wykryte problemy. Implementacja zaproponowanego narzędzia wspierającego proces testowania aplikacji wykorzystujących kontroler.
4. **Ocena stworzonych narzędzi wspomagających proces testowania** – prezentacja oraz obiektywna ocena przyjętego sposobu rozwiązania. Implementacja przykładowego programu oraz pokrycie testami partii kodu odpowiadających za analizę danych sensora jako demonstracja działania stworzonych narzędzi.

1.2 Struktura pracy

Rozdział drugi skupi się na zaprezentowaniu popularnych podziałów testów. Przedstawione zostaną ich cele oraz zastosowania zaczynając od testów modułowych a kończąc na testach akceptacyjnych. Poruszony zostanie temat automatyzacji procesu testowego w formie procesu ciągłej integracji.

W rozdziale trzecim przedstawiona zostanie budowa kontrolera Kinect oraz sposób jego działania. Następnie opisane zostaną dane oraz funkcjonalności które kontroler udostępnia (API).

Rozdział czwarty opisuje problemy występujące przy pracy z kontrolerem wraz z sugerowanymi ich rozwiązaniami. Opisane zostają funkcjonalności które stworzony zestaw narzędzi powinien zawierać.

W ramach rozdziału piątego zaprezentowane zostają i przetestowane pod kątem funkcjonalności z poprzedniego rozdziału istniejące narzędzia ułatwiające testowanie programów korzystających z kontrolera.

Rozdział szósty i siódmy skupiają się na stworzonych narzędziach w ramach pracy. W rozdziale szóstym przedstawione zostają koncepcje rozwiązania natomiast w rozdziale siódmym umieszczone i opisane zostają szczegóły implementacyjne.

Celem rozdziału ósmego jest demonstracja oraz weryfikacja działania stworzonych narzędzi. Na podstawie przykładowej aplikacji pokazany zostaje proces integracji z stworzoną biblioteką a następnie opisane zostają testy jednostkowe oraz akceptacyjne przykładowego programu z użyciem zintegrowanej biblioteki.

Na koniec rozdział dziewiąty przedstawia wnioski, sugestie dalszego rozwoju oraz podsumowanie pracy.

2. Techniki testowania oprogramowania

Testowanie jest szerokim zagadnieniem. Istnieje wiele podziałów testów ze względu na różne zagadnienia. Testy można dzielić ze względu na ich wielkość (testy klas, komponentów, podsystemów, systemów), ze względu na ich przeznaczenie, dziedzinę czy charakterystykę. Zaczynając od testów jednostkowych gdzie komponenty testowane są w izolacji od reszty aplikacji. Kończąc na testach akceptacyjnych gdzie końcowy produkt testowany jest przez grupę testów wewnętrznych (testy alfa) oraz zewnętrznych (testy beta). Istnieje też wiele technik oraz modeli związanych z testowaniem. W tym rozdziale postaram się opisać i przybliżyć najbardziej znane metody i rodzaje testów.

2.1 Podział dziedzinowy

Najbardziej naturalnym podziałem testów jest rozróżnianie ich ze względu na dziedzinę. Podział taki agreguje testy w oparciu o obszar przez nie testowany. A wiec zaczynając od testów obejmujących najmniejszy zakres możemy wydzielić:

Testy Modułowe - Poziom testów modułowych inaczej zwany testami jednostkowymi skupia się na obiektach lub funkcjach, Poszczególne metody są testowane w oderwaniu od reszty aplikacji. Ich główna zaleta jest ich precyzja. Dany test odpowiada wyłącznie za minimum funkcjonalności którą testuje. W przypadku jego niepowodzenia programista szybko jest w stanie zorientować się jaka funkcjonalność przestała działać. Testy te nie powodują efektu lawiny w przypadku dużych zmian gdyż tylko testy odpowiadające za naruszone funkcjonalności przestają przechodzić. Poziom ten również odznacza się dużą szybkością testów gdyż obszary są proste (bo jednostkowe) i nie wymagają lub wymagają niewielkiej inicjalizacji aby mogły zostać przetestowane. W skład testów jednostkowych wydzielić możemy analizę ścieżek, klasy równoważności oraz testy wartości brzegowych. Analiza ścieżek polega na przejściu wszystkimi możliwymi drogami jakimi testowana funkcja może podążyć od wejścia do wyjścia. Klasy równoważności są zbiorem danych używanych do przeprowadzenia testu gdzie wykonanie go z użyciem kilku jego elementów uznaje całą jego zawartość za poprawną. Klasy równoważności możemy dzielić na klasy poprawności które zakładają prawidłowy rezultat dla każdego elementu zbioru oraz niepoprawności których rezultaty są negatywne. Analiza wartości brzegowych jest rozwinięciem testowania z użyciem klas równoważności gdzie testom podlegają wartości brzegowe z zbiorów.

Testy Integracyjne - Testy te wykonywane są w celu znalezienia błędów w interfejsach wystawianych praz komponenty oraz interakcjach między nimi. Gdy moduły przetestowane zostają jednostkowo następnym krokiem jest zapewnienie poprawności ich współdziałania z resztą systemu. Testy te można przeprowadzić podejściem bottom-up lub top-bottom. Podejście top-bottom polega na testowaniu największych modułów w pierwszeństwie gdzie moduły znajdujące się niżej w hierarchii zastępowane są zaślepkami (stub). Testowane moduły używane zostają do testów niżej położonych komponentów. Takie testowanie kontynuowane jest do chwili gdy każdy moduł zostaje przetestowany. Podejście bottom-up jak łatwo się

domyślić jest przeciwieństwem pierwszego. Najniżej położone moduły testowane są jako pierwsze a proces kończy się na testach najwyżej położonych komponentów.

Testy Systemowe – Poziom testów przeprowadzany zostaje aby zweryfikować czy system spełnia wymagania zawarte w specyfikacji. Podczas tych testów system testowany jest jako całość (w odróżnieniu od testów modułowych) po kątem zgodności z wymaganiami funkcjonalnymi oraz niefunkcjonalnymi. Testy przeprowadzane są w środowisku zbliżonym do produkcyjnego aby odwzorować realne warunki w których działał będzie system. Jako iż testy wymagają całości systemu nie należą do najszybszych a ich precyzja nie jest duża gdyż testowane funkcjonalności z reguły zależą lub wykorzystują dużo komponentów. Wśród tych testów rozróżnić możemy testy instalacyjne sprawdzające proces wdrożenia systemu, testy funkcjonalne obejmujące zakresem wymagania funkcjonalne, testy interfejsów wystawianych przez system na zewnątrz (API) oraz testy regresywne sprawdzające integralność systemu w czasie zmian (waga tych testów omówiona zostanie w późniejszym podrozdziale).

Testy Akceptacyjne – Jest to walidacja systemu po kątem zgodności z wymaganiami klienta. Testy przeprowadzane zostają w środowisku klienckim lub jego symulacji gdzie wykonywane zostają przekładowe zastosowania systemu. Poza funkcjonalnościami dotyczą również innych aspektów takich jak wydajność, bezpieczeństwo, konfigurowalność, użyteczność, wygląd czy łatwość utrzymania. W dużym uproszczeniu testy te mają udowodnić klientowi iż otrzymuje produkt który zamówił/kupił.

2.2 Weryfikacja a walidacja

Jednym z najbardziej prostych i znanych podziałów jest walidacja i weryfikacja. Podział ten skupia się na idei jaka stoi za testami a nie na jego wielkości.

Weryfikacja („Czy produkt tworzony jest prawidłowo”) polega na dostarczeniu dowodów na poprawne działanie aplikacji. Z reguły sprowadza się to do zapewnienia iż oprogramowanie jest zgodne z swoją specyfikacją. Kod jest łatwy w utrzymaniu a najważniejsze funkcjonalności są pokryte testami. Testy te maja zapewnić wysoką jakość kodu nie zaś jego funkcjonalność. W skład tych testów będą wchodziły testy modułowe sprawdzające poprawność używanych komponentów, testy integracyjne sprawdzające komunikację między komponentami oraz testy systemowe sprawdzające system jako całość.

Walidacja („Czy tworzony produkt jest prawidłowy”) jest procesem sprawdzenia czy aplikacja spełnia potrzeby i wymagania użytkownika. Cześć testów przeznaczona dla klienta których celem jest zapewnienie iż produkt spełni jego oczekiwania. Rodzaj tych testów zwany jest testami akceptacyjnymi. W jego skład wyróżnić możemy testy funkcjonalne, wydajnościowe oraz bezpieczeństwa.

Dla przykładu przyjmijmy iż chcemy przetestować aplikacje której zadaniem jest codzienna aktualizacja danych z portalu sportowego i zapis ich do bazy danych. Program jest dość prosty i składa się z trzech komponentów:

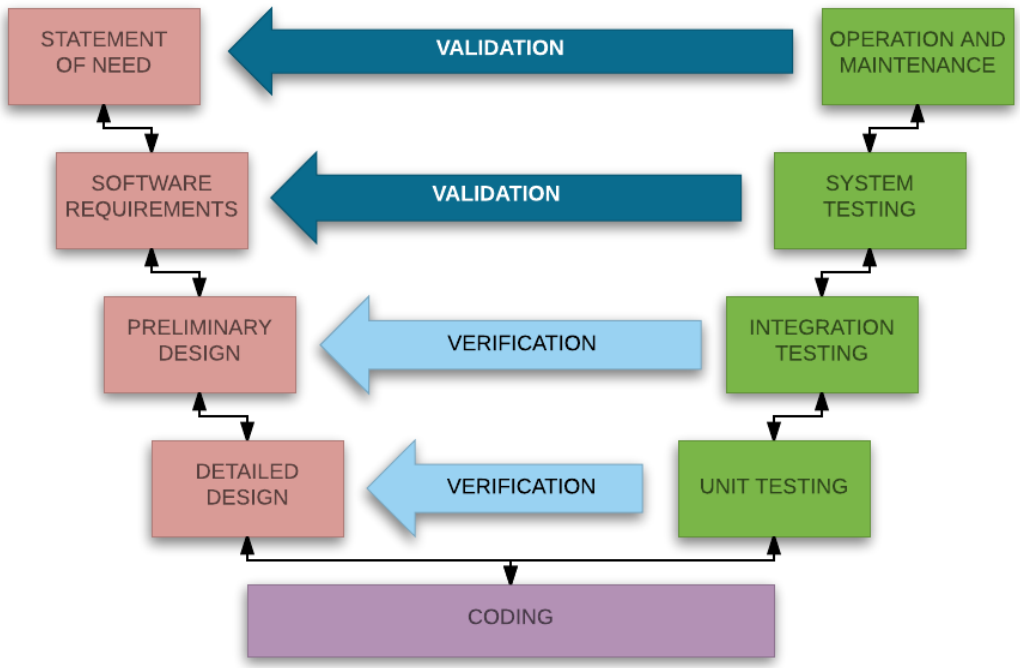
- **DataRepository** (komponentu do zapisu danych)

- **SportDataProvider** (komponent do pobrania danych z serwisu sportowego)

- **SportUpdateManager** (komponentu który zainicjuje codziennie aktualizacje)

W skład testów weryfikacyjnych wydzielić moglibyśmy testy modułowe których celem byłoby sprawdzenie poprawności każdego komponentu z osobna. Testy jednostkowe **DataRepository** sprawdzałyby czy dane poprawnie zapisywane są w bazie natomiast testy **SportDataProvider** skupiałyby się na tym czy komponent podaje poprawne dane z serwisu. Testy integracyjne w tym wypadku polegałyby na objęciu testami komponentu **SportUpdateManager** i zapewnieniu iż dane wymieniane są miedzy dwoma poprzednimi komponentami w poprawny sposób. Testy systemowe polegać by mogły na przetestowaniu całości aplikacji sprawdzając czy aktualizacja została wykonana poprawnie.

W skład testów walidacyjnych moglibyśmy przygotować testy funkcjonalne sprawdzające czy codziennie aktualizacje zostają zainicjalizowane. Sprawdzić jak aplikacja zachowa się w przypadku braku dostępu do serwisu lub bazy oraz czy stworzona aplikacja spełnia wymagania klienta. Testy wydajnościowe sprawdzałyby czy zbyt duża ilość danych nie spowoduje zakłócenia działania, czy nie dochodzi do wycieków pamięci oraz jak aplikacja zachowuje się na różnych środowiskach przy długotrwałym działaniu. Testy bezpieczeństwa skupiały by się na możliwościach zakłócenia działania aplikacji np. przy użyciu sql-injection oraz czy dane dostępu do naszej bazy są dobrze zabezpieczone.



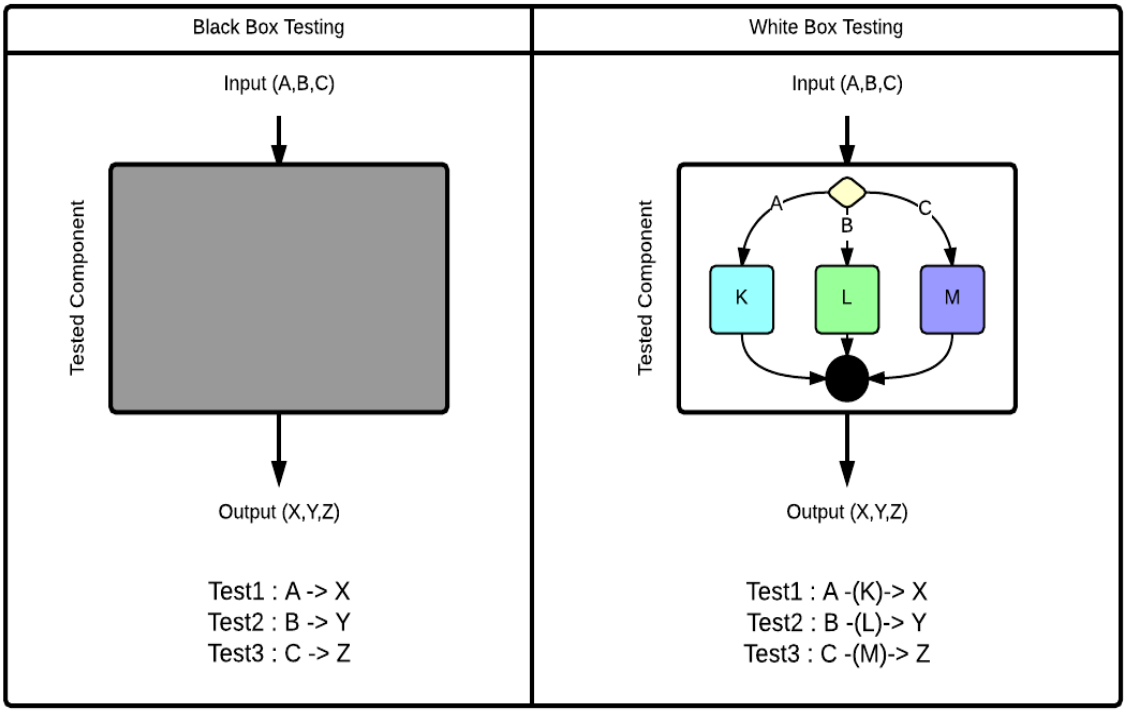
Rys 2.1: V-model wytwarzania oprogramowania

Podział na weryfikacje i walidację bardzo dobrze widać na V-modelu wytwarzania oprogramowania zaprezentowanym na rysunku 2.1. W modelu tym wyróżnia się trzy fazy.

Początkowa faza skupia się na sprecyzowaniu głównych potrzeb a następnie określeniu wymagań sprzętowych stawianych aplikacji. Później dochodzi do stworzeniu wstępnego oraz dokładnego planu projektu. Dopiero w drugiej fazie dochodzi do implementacji. Trzecia faza poświęcona jest na walidację oraz weryfikacje, gdzie w odwrotnej kolejności dochodzi do sprawdzenie każdego kroku podjętego w fazie pierwszej. Rozpoczyna się od weryfikacji dokładnego planu projektu w postaci testów modułowych. Następnie weryfikacji ulega wstępny plan w postaci testów integracyjnych. Kolejne fazy klasyfikowane są jako walidacje. Testowany jest cały stworzony system czy spełnia wszystkie wymagania sprzętowe oraz czy realizuje potrzeby stawiane na początku.

2.3 Testy strukturalne i funkcjonalne

Innym podziałem są testy strukturalne i funkcjonalne. Testy strukturalne inaczej zwane testami lustrzanej lub białej skrzynki polegają na takim doborze danych wejściowych aby program przeszedł przez każdą możliwą ścieżkę kodu. Natomiast testy funkcjonalne lub inaczej testy czarnej skrzynki polegają na sprawdzeniu rezultatów na odpowiednie dane wejściowe. Podział ten zaprezentowany został na rysunku 2.2.



Rys 2.2: Testy czarnej/białej skrzynki

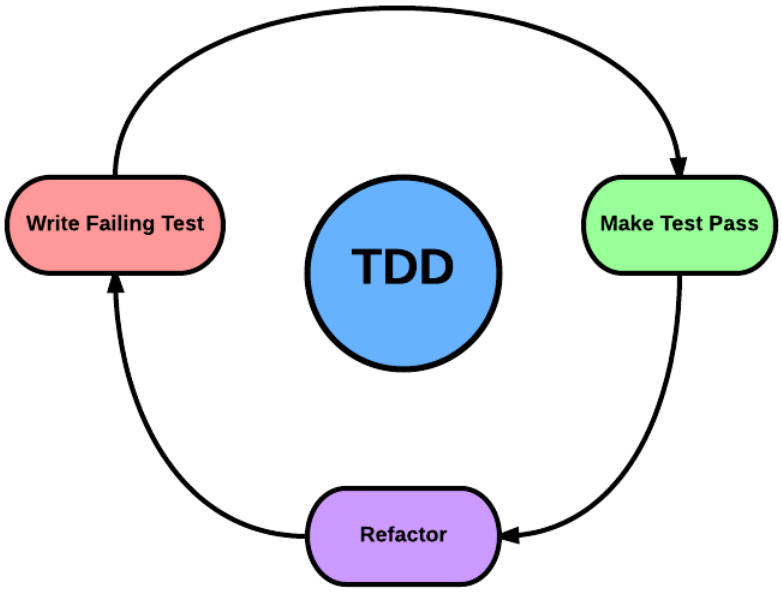
Testy białej skrzynki pisane są z reguły przez twórców testowanego komponentu, ponieważ wymagana jest dogłębna znajomość budowy oraz zachowania kodu wewnętrznego. Testy te nie są w stanie jednak sprawdzić braku funkcjonalności, gdyż skupiają się na

wewnętrznej strukturze zakładając iż funkcjonalność została spełniona. Główna ich wadą jest częsta konieczność modyfikacji kodu na rzecz testu (przysłowiowy „Słoń w Kairze"). Konieczność znajomości kodu także zawęża ilość potencjalnych testerów do osób które tworzyły lub brały udział w tworzeniu funkcjonalności co przekłada się na koszt testów.

Natomiast testy czarnej skrzynki nieraz wykonywane są przez osoby nie posiadające wiedzy programistycznej. Dane testowe nie są oparte o wewnętrzną budowę, lecz o założenia funkcjonalne które spełniać powinien testowany komponent. Ułatwia to sprawdzenie poprawności pod kątem funkcjonalności lecz często testy te nie nadają się do testowania wewnętrznych komponentów które nie udostępniają żadnej funkcjonalności na zewnątrz systemu. Testy te bardziej nadają się do testowania całości systemu lub jego podsystemów. Z racji iż nie jest znana wewnętrzna budowa w przypadku wystąpienia błędu ciężkie lub wręcz niemożliwe jest określenie przyczyny błędu.

2.4 Programowanie sterowane testami

Obecnie dużym zainteresowaniem cieszą się metodyki zwinne gdzie jedna z praktyk jest TDD (Test Driven Development) czyli programowanie sterowane testami. Praktyka ta polega na iteracyjnym tworzeniu oprogramowania gdzie w skład iteracji wchodzi dodanie testu który jeszcze nie jest pozytywny, napisanie minimalnej ilości kodu aby test przeszedł oraz refaktoryzacji kodu napisanego w poprzednim kroku. Praktyka ta zilustrowana jest na rysunku 2.3. Po pewnej ilości iteracji gdy wszystkie wymagania zostają spełnione otrzymujemy wypełni przetestowany komponent.



Rys 2.3: Test Driven Development

Dzięki takiemu podejściu w łatwy sposób zapewnić można duże pokrycie kodu testami a konieczność pisania kodu pod testy (gdyż test powstaje przed kodem) wymusza jego testowalność oraz poprawną strukturę. Praktykę tą łatwo zastosować gdy bardzo dobrze znane

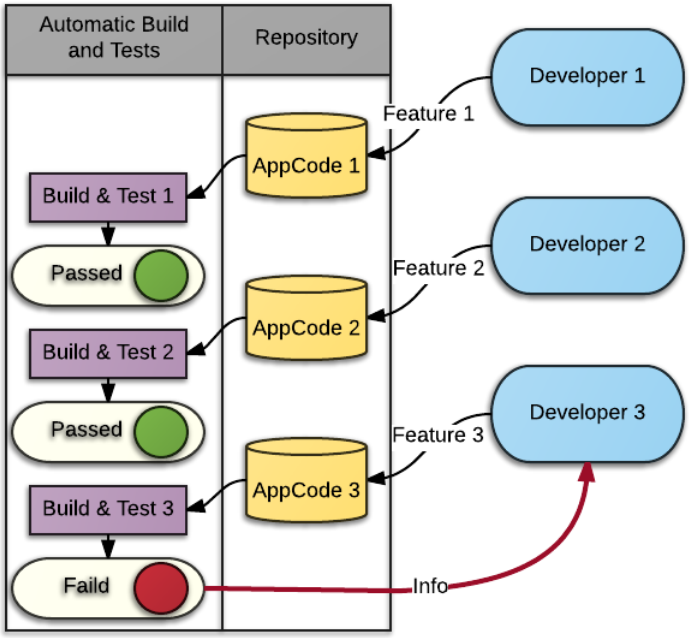
są wymagania tworzonego komponentu. Metodyka ta prowadzi do powstania dużej ilości testów czyniąc kod odpornym na zmiany, gdyż zakłócenie jakiejkolwiek funkcjonalności powoduje błędne zakończenie testów które ją sprawdzają alarmując programistę o zaistniałej sytuacji.

2.5 Automatyzacja i regresja

Testy można dzielić też ze względu na sposób ich wykonywania a wiec na testy automatyczne oraz manualne (których egzekucja wymaga ręcznych czynności). Drugi rodzaj testów potrafi być prawdziwą udręką testerów gdyż rzadko kiedy test wykonywany jest tylko raz w ciągu całego procesu produkcji aplikacji. A skoro testy wykonywane są wielokrotnie ich zbyt duża ilość prowadzi do spadku produktywności a sama czynność powtarzana wielokrotnie zaczyna być monotonna i podatna na błędy ludzkie. Lecz niestety testowanie wielu aspektów nie da się w łatwy sposób zautomatyzować. I choć nakład czasowy nieraz poświęcany automatyzacji procesu testowego jest duży (tworzenie narzędzi/frameworków testowych) nieraz zwraca się on z nawiązką przy długotrwałym utrzymywaniu projektu. Szczególnie dobrym kandydatem na jak największy stopień automatyzacji są testy regresyjne gdyż cechuje je powtarzalność. Testy te mają na celu upewnienie się iż aplikacja działa poprawnie po dokonaniu w niej modyfikacji, dodaniu funkcjonalności czy naprawie błędu. Testy te idealnie nadają się do przyrostowego modelu programowanie gdzie w krótkich cyklach dostarczane są nowe funkcjonalności. Po każdej iteracji regresyjne testy te sprawdzają czy nie naruszono już istniejących funkcjonalności.

2.6 Ciągła integracja

Powyżej opisane rodzaje i techniki testowania mają na celu sprawdzenie różnych aspektów tworzonej aplikacji. Gdy projekt jest duży i pracują nad nim zespoły programistów powszechną technika jest współdzielenie kodu przy pomocy systemu kontroli wersji takiego jak git, tfs czy svn. W rozwiązaniu tym kod programu przechowywany jest w głównym repozytorium gdzie każdy programista wprowadza swoje zmiany. Repozytorium to często zostaje podpięte pod narzędzia ciągłej integracji takie jak Jenkins, Hudson czy Teamcity.



Rys 2.4: Continues Integration

Zadaniem narzędzi ciągłej integracji jest automatyzacja budowy aplikacji w głównym repozytorium oraz sprawdzenie poprawności wszystkich napisanych automatycznych testów (od testów jednostkowych do testów akceptacyjnych) przy każdej wprowadzonej zmianie co demonstruje rysunek 2.4. Podejście to w łatwy sposób alarmuje programistę dokonującego zmian o zakłóceniu funkcjonalności której test przestał dawać pozytywny rezultat. Ciągła integracja redukuje ryzyko wprowadzania zmian oraz czas testowania aplikacji gdyż wersja w głównym repozytorium pozostaje zawsze przetestowana automatycznymi testami. Należy zaznaczyć iż podejście to nie niweluje błędów ale pozwala na ich wczesne wykrycie i szybka interakcję. W stosunku do modelu w którym testowanie odbywa się w fazie końcowej (jak w V-modelu) nie dochodzi do kumulowania się błędów. Gdy dopuszczamy w aplikacji dużą ilość błędów późniejsza próba ich naprawy jest trudniejsza gdyż programista musi się powtórnie wdrożyć w temat który spowodował błąd. Błędy nieraz nakładają się dając dziwne i trudne w wyśledzeniu rezultaty. Ujemy efekt jest też na podłożu psychologicznym. Im więcej jest do zrobienia tym mniej człowiek ma chęci (efekt ten zwany jest syndromem wybitych okien).

3. Praca z kontrolerem ruchu Kinect

Istnieje wiele urządzeń do analizy ruchów ciała. Jednym z najpopularniejszych który zawarty jest w tytule pracy jest kontroler Kinect. Wyróżnia się on na tle konkrecji tym iż nie wymaga noszenia żadnych urządzeń pomagających w detekcji gestu w formie znaczników czy padów. Poniższy rozdział przeznaczony został na analizę sensor. Przedstawiona zostanie jego historia, budowa oraz zasada działania. Na koniec zaprezentowane zostanie interfejs z którego skorzystać może użytkownik przy pracy z sensorem.

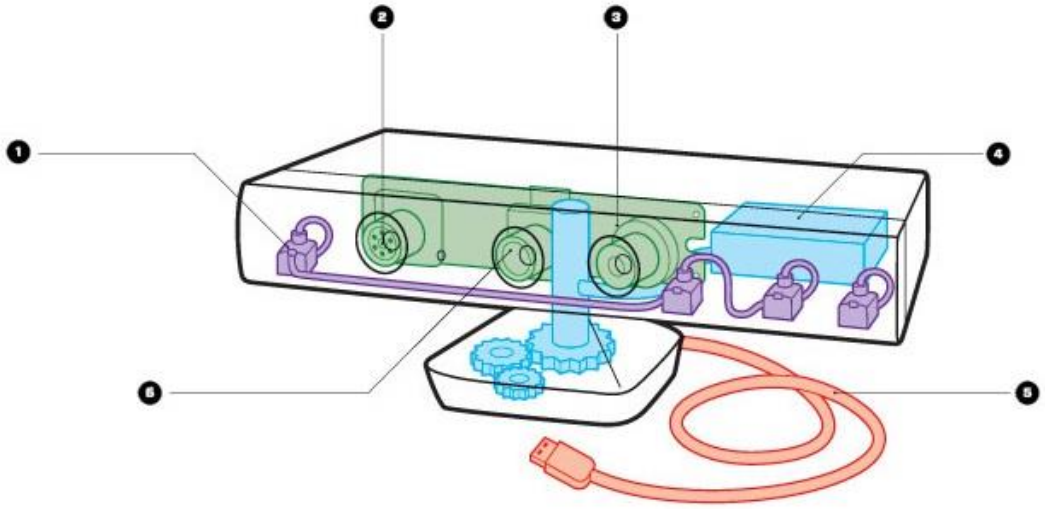
3.1 Historia kontrolera

19 grudnia 2009 roku firma Microsoft wypuściła kontroler Kinect 360 jako dodatkowy sprzęt do konsoli Xbox 360. Urządzenie jako pierwsze oferowało kontrolę przy pomocy gestów bez dodatkowego sprzętu w postaci wszelkiego rodzaju wskaźników, które oferowały konkrecyjne platformy (Nintendo Wii oraz Play Station Move). Kontroler został ciepło przywitany przez konsumentów ustanawiając rekord Guinnessa w kategorii najszybciej sprzedającego się urządzenie notując średnią sprzedaż 133 333 kontrolerów dziennie przez pierwszych 60 dni. Duża pula konsumentów otworzyła wiec nowy rynek, który początkowo był skierowany dla wydawców gier. W 2011 roku Microsoft wypuścił oficjalne wsparcie dla systemów Windows oraz pierwszy SDK (software development kit) do kontrolera umożliwiający pisanie aplikacji w oparciu o języki takie jak C++/CLI, C# oraz Visual Basic .NET. Kontroler szybko znalazł wiele innych zastosowań poza obsługą gier na konsoli takich jak na przykład:

- Sterowanie dronami za pomocą gestów w Projekcie STARMAC na uniwersytecie w Berkeley.
- Sterowanie interfejsem użytkownika w podobny sposób jak w filmie „Raport Mniejszości".
- Uzyskanie rzeczywistości rozszerzonej czyli połączenia świata realnego z generowanym komputerowo (Augmented Reality).
- Jako platforma do skanowania i tworzenia modeli 3D Kinect Fusion.
- Próby rozpoznawania języka migowego.
- W dla szpitalali w calu ułatwienia chirurgom bez dotykowego nawigowania urządzeniami podczas wykonywania operacji.
- Modelowanie świata Minecraft przy pomocy kontrolera Kinect w projekcie KinectCraft.

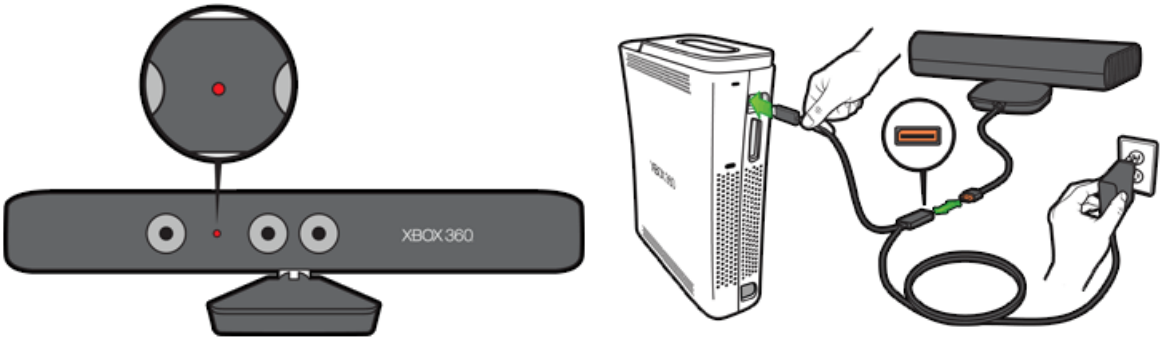
3.2 Budowa

Kontroler Kinect 360 jest niewielkim rozmiarów urządzeniem którego budowa została zaprezentowana na rysunku 3.1. Sensor wyposażony jest w:



Rys 3.1: Budowa kontrolera Kinect

1) **Zestaw czterech mikrofonów** - dzięki którym urządzenie jest w stanie dość dokładnie umiejscowić źródło dźwięku.
2) **Emiter podczerwieni** - Emituje wiązkę promieni podczerwonych które odbijają się od powierzchni wracając maję różne długości.
3) **Kamera głębokości** - Odczytuje wracające promienie podczerwone dzięki którym jest w stanie stworzyć model 3D przestrzeni przed kontrolerem.
4) **Silniku sterujące nachyleniem** - Mechanizm silników umożliwiający automatyczna kalibrację urządzenia.
5) **Kabel połączeniowy** - Zależnie od modelu w wersji 360 E oraz S jest to proste połączenie do portu AUX. W normalnej wersji 360 jest ono rozdzielone między kabel USB oraz zasilacz.
6) **Kamera RGB** - Odpowiada za nałożenie warstwy koloru na model 3D wygenerowany z kamery głębokości.

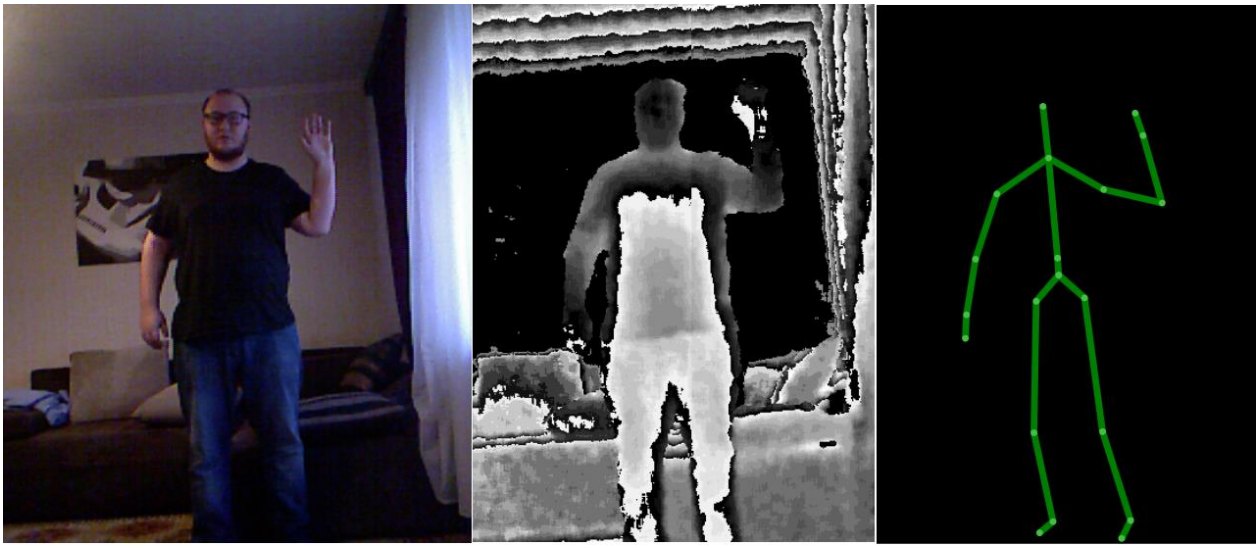


Rys 3.2: Dioda sygnalizacyjna i kabel połączeniowy

Z przodu urządzenia znajduje się dioda sygnalizująca stan urządzenia zaprezentowana na rysunku 3.2. Gdy urządzenie działa bez zakłóceń świeci się kolorem zielonym. Natomiast gdy urządzenie nie jest w stanie poprawnie funkcjonować zapala się na kolor czerwony. Kontroler wydany został w dwóch wersjach początkowej 360 przeznaczonej dla konsoli oraz zmodyfikowanej wersji PC. Wersja dla komputerów została wypuszczona później a sam kontroler został rozszerzony o opcję pozwalającą pracować w bliższej odległości (near mode) oraz o możliwość wykrycia sylwetki siedzącej (10 joint mode). Dodatkowo użytkownik dostał możliwość sterowania niektórymi opcjami kamery oraz rozszerzona została mechanika rozpoznawania ruchów ręki o uścisk (handgrip detection).

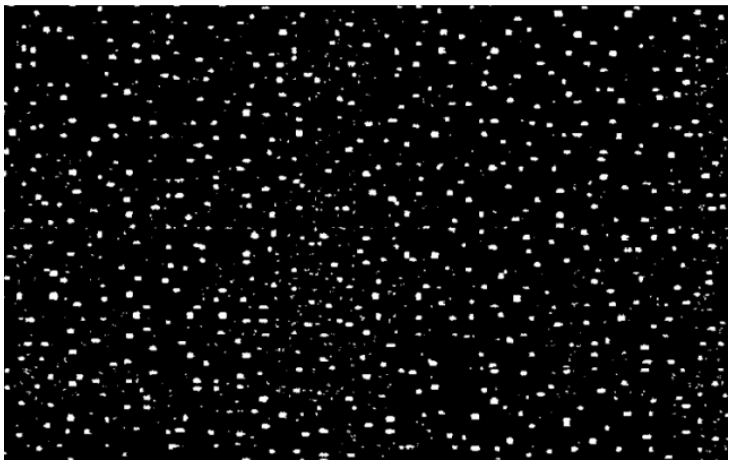
3.3 Przetwarzanie danych

Z kontrolera uzyskać możemy trzy strumienie danych z prędkością 30fps (klatek na sekundę). Są nimi obraz koloru, obraz głębokości oraz pozycje wykrytych szkieletów zaprezentowane na rysunku 3.3.



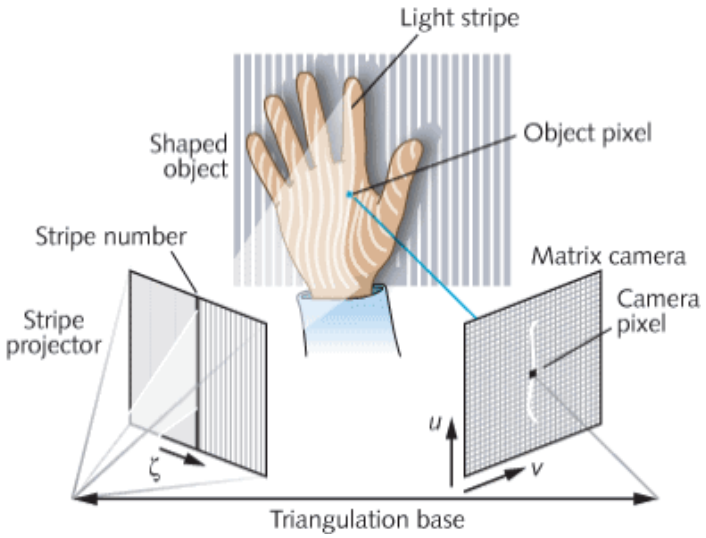
Rys 3.3: Color, Depth, Skeleton Frame

Wszystkie dane kontroler uzyskuje przy użyciu dwóch kamer. Kamera RGB pracuje w rozdzielczości maksymalnej 1280x960 i dostarcza standardowy kolorowy obraz. Nie jest ona używana do uzyskania obrazu głębokości oraz znalezienia szkieletów. Obraz z kamery głębokości uzyskany jest poprzez pomiar i analizę odbitych promieni rzucanych przez emiter podczerwieni.



Rys 3.4: Sparks pattern

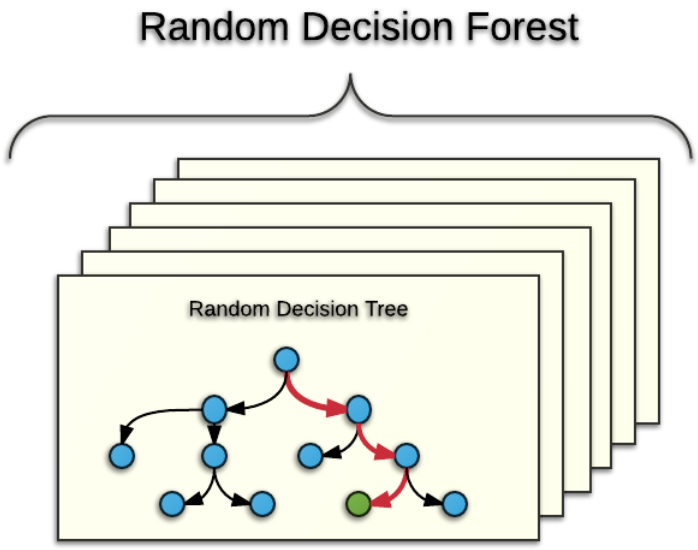
Promienie rzucane są w znany sposób lecz za każdym razem różniący się wzorem od poprzedniego co powoduje widoczne szumy w obrazie głębokości lecz zapewnia większą niezawodność. Sposób rzucanych promieni prezentuje rysunek 3.4. Technika analizy tych promieni zwana jest ustrukturyzowanym światłem (structured light) a schemat jej działania zaprezentowany został na rysunku 3.5. Polega ona na projektowaniu znanej wiązki promieni na scenę a następnie poprzez analizą deformacji odbitych wiązek które natrafiają na różne powierzchnie odwzorowywany jest ich kształt. W ten sposób budowana jest mapa głębokości która każdemu pikselowi przypisuje wartość odpowiadającą jego odległości od kontrolera. Kamera głębokości generuje mapę w maksymalnej rozdzielczości 640x480. Do obliczenia wartości każdego piksela analizowany jest dany sektor promieni. Użyte zostają dwie techniki optyczne: głębokość przez skupienie (depth from focuse) oraz stereo (depth from stereo).



Rys 3.5: Structured light[1]

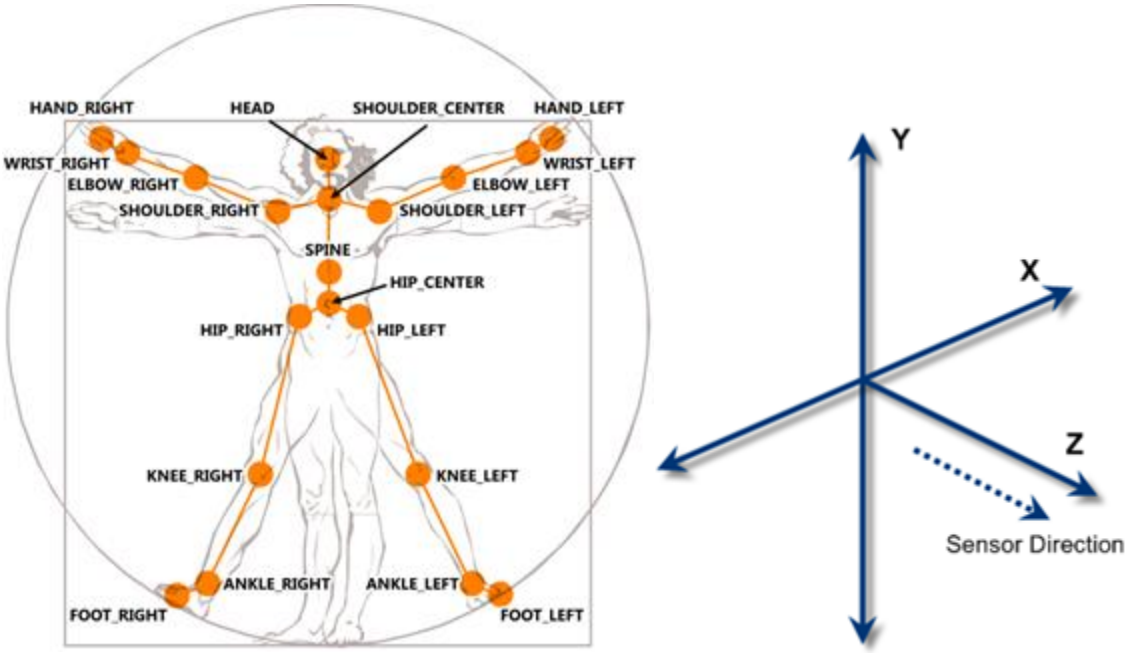
[1]Zewnętrzna grafika (http://www.laserfocusworld.com/content/dam/etc/medialib/new-lib/laser-focus-world/online-articles/2011/01/93355.res/_jcr_content/renditions/pennwell.web.390.296.gif)

Głębokość przez skupienie polega na tym iż obiekty znajdujące się dalej są bardziej niewyraźne i zamazane od obiektów bliżej odbiornika. A wiec im większy szum w liczeniu danego sektora tym dalej powierzchnia się znajduje. Głębokość przez stereo polega na tym obiekty oglądane z dwóch miejsc obok siebie stają się przesunięte w bok względem bardziej im bliżej się znajdują. Kamera analizuje to przesunięcie w wzorze odbitych promieni gdyż wzór projektowany jest z innego miejsca niż jest on obserwowany. Cały proces wyliczenia mapy głębokości został opracowany przez firmę PrimeSense i jego dogłębne szczegóły nie są publicznie dostępne.



Rys 3.6: Decision Forest

Mając mapę głębokości kontroler przystępuje do próby znalezienia sylwetek ludzkich. Do opracowania tego algorytmu Microsoft użył uczenia maszynowego. Zebrano ponad 100 000 dobrze znanych i przygotowanych układów szkieletów. Dla każdego wygenerowano dodatkowe warianty używając metod graficznych tworząc bazę ponad miliona treningowych danych. W celu zmapowania mapy głębokości do części ciała kontroler nauczony został losowego lasu decyzyjnego (randomized decison forest) zaprezentowanego na rysunku 3.6. Jest to bardziej złożona odmiana zwykłego drzewa decyzyjnego gdzie odpowiadając na pytania od bardziej ogólnych do szczegółowych jesteśmy wstanie szybko znaleźć właściwe rozwiązanie. Nauka drzewa decyzyjnego polega na wybraniu następnego pytania takiego które jest najbardziej decyzyjne/pomocne. Kontroler nauczony został wielu drzew decyzyjnych a wynik każdego jest ich składowy wyniku głównego. Nauka lasu decyzyjnego o głębokości 20 dla miliona danych zajęła dzień w środowisku rozporoszonym na ponad tysiącu jednostek obliczeniowych. Dzięki temu kontroler jest w stanie znaleźć obraz ciała w mapie głębokości.

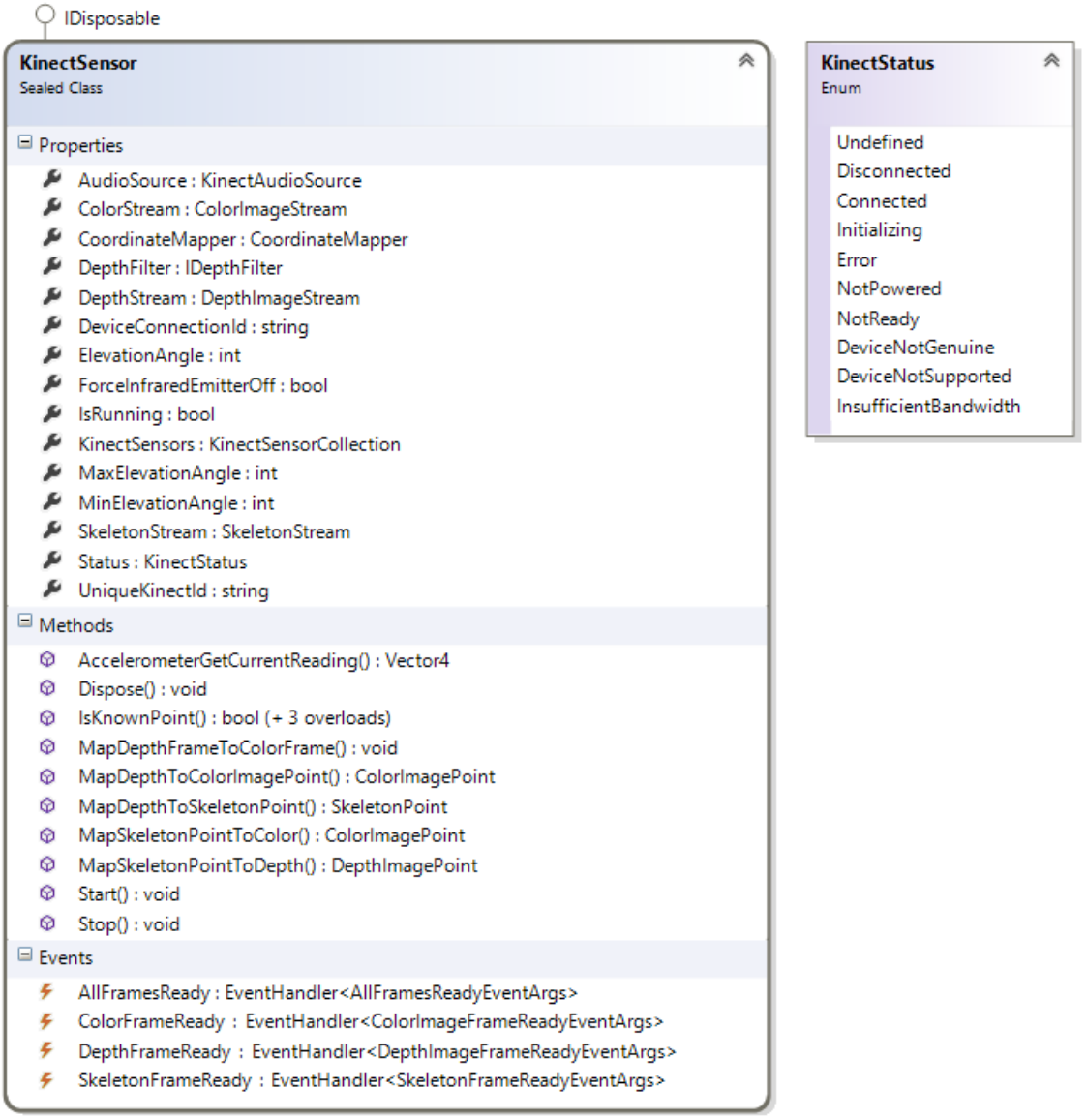


Rys 3.7: Skeleton joins model

Następnie do utworzenia szkieletu zastosowany jest algorytm mean-shift który w dużym skrócie znajduje ośrodki o największym skupieniu punktów. Wyszukuje on w odpowiednich sektorach ciała miejsca skupienia dając współrzędne środków kończyn (współrzędne x i y) wraz z jego wartością z mapy głębokości (współrzędna z). Rysunek 3.7 prezentuje wszystkie rozpoznawane kończyny oraz pokazuje układ współrzędnych których używa kontroler.

3.4 Sposób użycia

Aby móc użyć sensora w aplikacji należy najpierw zainstalować sterowniki. Można użyć oryginalnych sterowników od firmy Microsoft załączonych w SDK. Drugą opcja jest użycie sterowników i nieco innego API od firmy OpenNI (Open Natural Interaction). Choć OpenNI wspiera również język programowania Java to jest mniej znanym i wspieranym rozwiązaniem dlatego w dalszej części skupimy się na API dostarczonym wraz z sterownikiem od firmy Microsoft. Rysunek 3.8 przedstawia diagram klasy kontrolera **KinectSensor**.



Rys 3.8: KinectSensor Class Diagram

Aby uzyskać dostęp do klasy reprezentującej kontroler należy odwołać się do statycznej klasy **KinectSensor**. W kolekcji **KinectSensors** zawiera ona wszystkie podłączone do systemu i wykryte sensory.

```
foreach (var potentialSensor in KinectSensor.KinectSensors)
{
	if (potentialSensor.Status == KinectStatus.Connected)
	{
		//our sensor
		InitializeSensor(potentialSensor);
		break;
	}
}
```

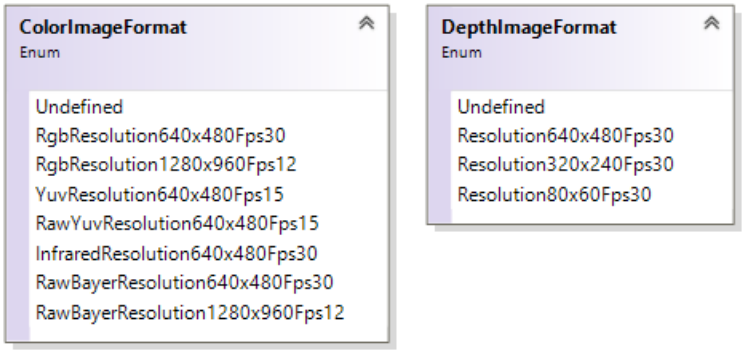
Przed użyciem warto sprawdzić status sensora mówiący o gotowości bądź problemach z urządzeniem. Status kontrolera może przyjąć wartości:

- **Connected** – Sensor jest wypełni gotowy do użycia.
- **DeviceNotGenuine** – Urządzenie nie zostało wykryte jako sensor Kinect.
- **DeviceNotSupported** – Urządzenie nie jest wspierane.
- **Disconnected** – Urządzenie zostało rozłączone.
- **Error** – Zaistniał problem z urządzenie.
- **Initializing** – Sensor jest w trakcie inicjalizacji.
- **InsufficientBandwidth** – Łącznik USB nie posiada wystarczającej szerokości łącza.
- **NotPowered** – Kabel zasilania do urządzenia nie jest podpięty.
- **NotReady** – Urządzenei nie jest jeszcze gotowe.
- **Undefined** – Status przeznaczony na nieobsłużone zdarzenia.

Po znalezieniu klasy kontrolera należy włączyć odpowiednie strumienie. Istnieją trzy możliwe strumienie: koloru, głębokości oraz szkieletu.

```
sensor.ColorStream.Enable(ColorImageFormat.RgbResolution640x480Fps30);
sensor.DepthStream.Enable(DepthImageFormat.Resolution640x480Fps30);
sensor.SkeletonStream.Enable();
```

Na rysunku 3.9 przedstawione zostały dostępne formaty dla strumieni koloru oraz głębokości.



Rys 3.9: Color/Depth Formats

Gdy włączone zostaną interesujące nas strumienie można przystąpić do uruchomienia kontrolera.

```
sensor.Start();
```

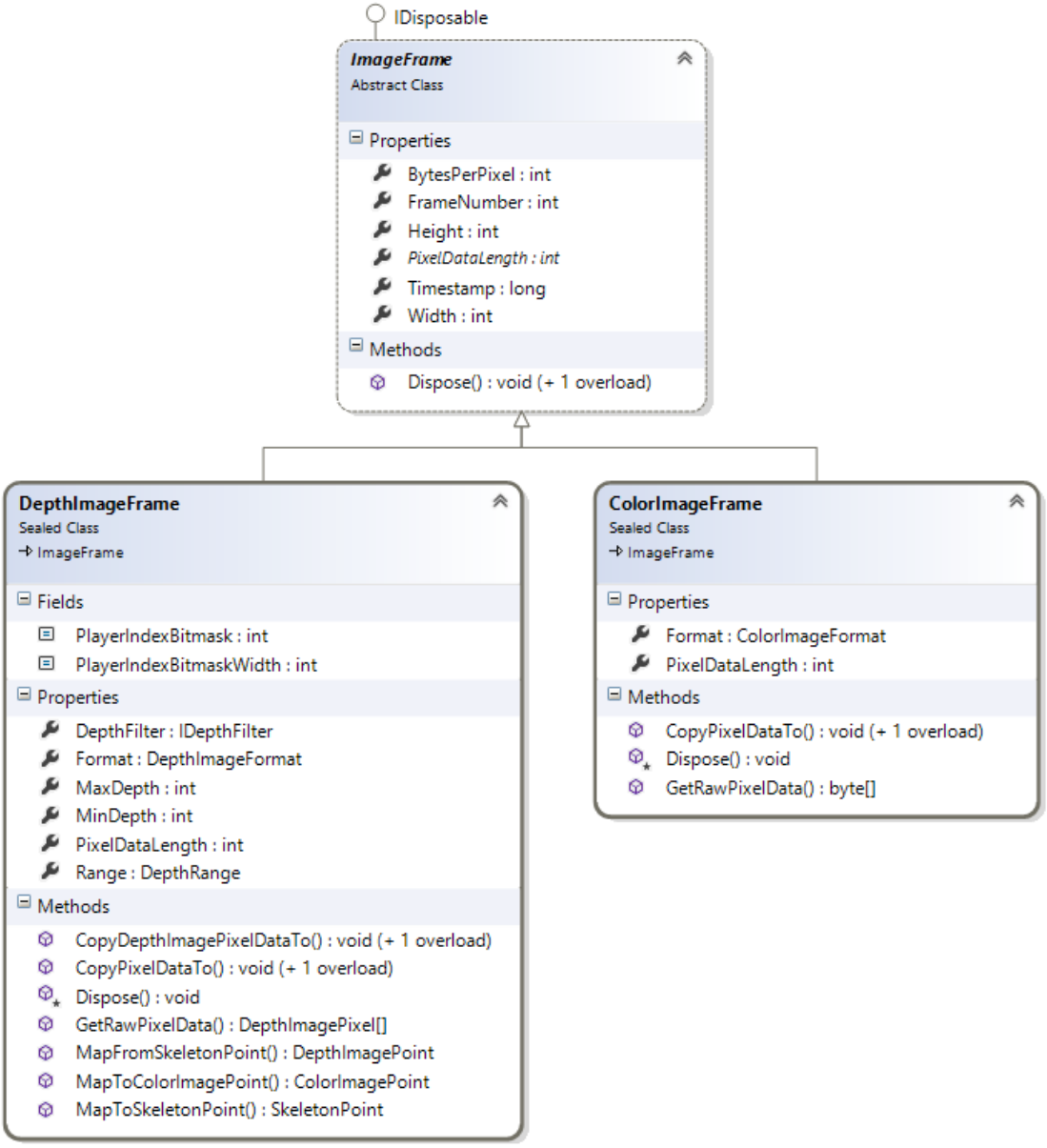
Metoda ta sprawi iż kontroler będzie postrzegany przez inne aplikacje jako zajęty. Aby odczytać napływające ramki należy podpiąć odpowiednie funkcje do zdarzeń (eventów).

```
sensor.ColorFrameReady += MyColorFrameReadyFunction;
sensor.DepthFrameReady += MyDepthFrameReadyFunction;
sensor.SkeletonFrameReady += MySkeletonFrameReadyFunction;
```

lub do funkcji agregującej gdy do analizy potrzebujemy więcej niż jeden strumień.

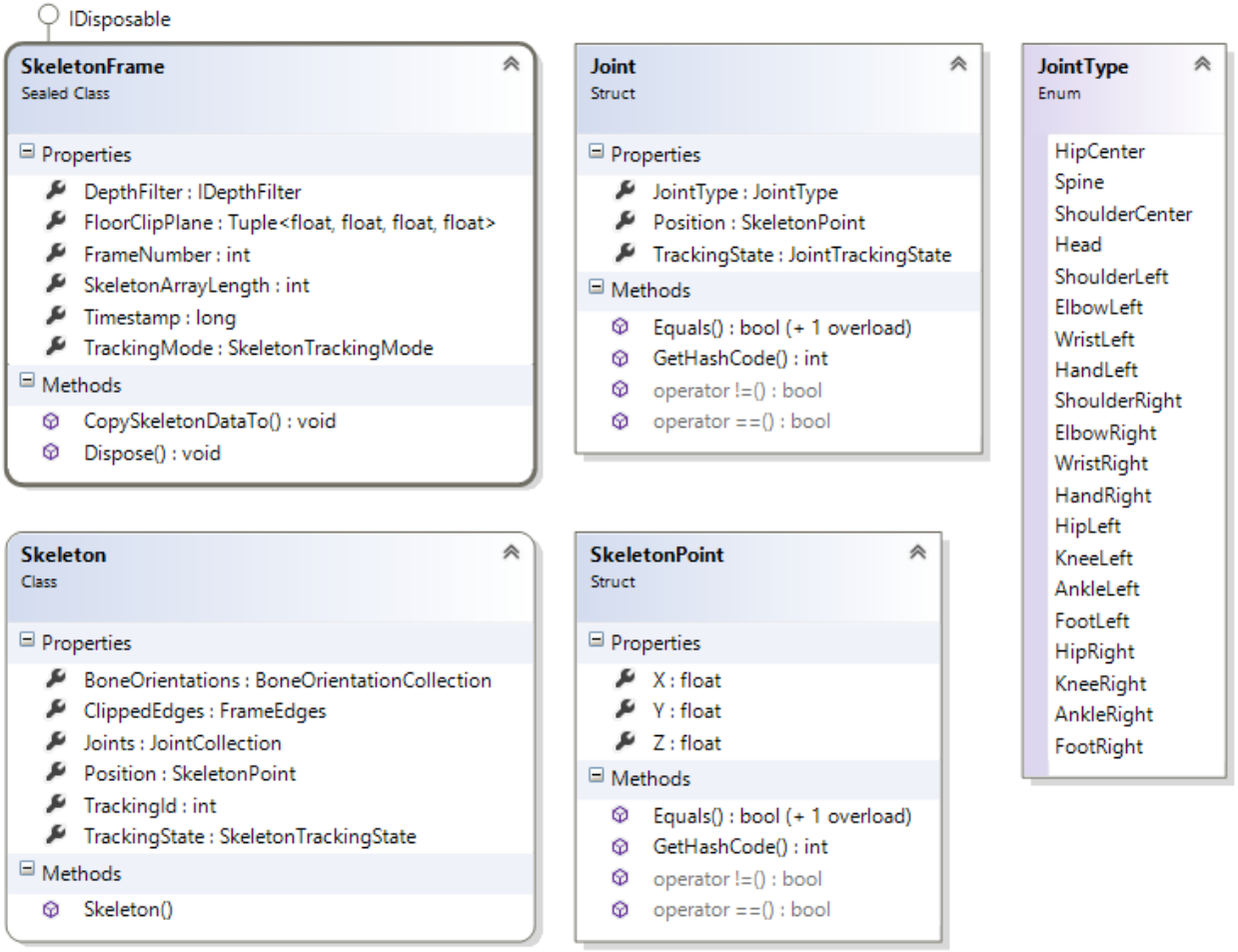
```
sensor.AllFramesReady += MyAllFramesReadyFunction;
```

Funkcje te wywoływane będą gdy kontroler uzyska kolejne ramki. W ich wywołaniu dostaniemy dane reprezentujące poszczególne rodzaje ramek.



Rys 3.10: Color/Depth Frame

Ramki koloru i głębokości dziedziczą z jednej klasy **ImageFrame** zaprezentowane na rysunku 3.10. Posiadają one metody **CopyPixelDataTo**() kopiujące bufory ramek do analizy. Poza mapami pikseli i danymi opisującymi te mapy nie posiadają większej ilości interesujących informacji.



Rys 3.11: Skeleton Frame

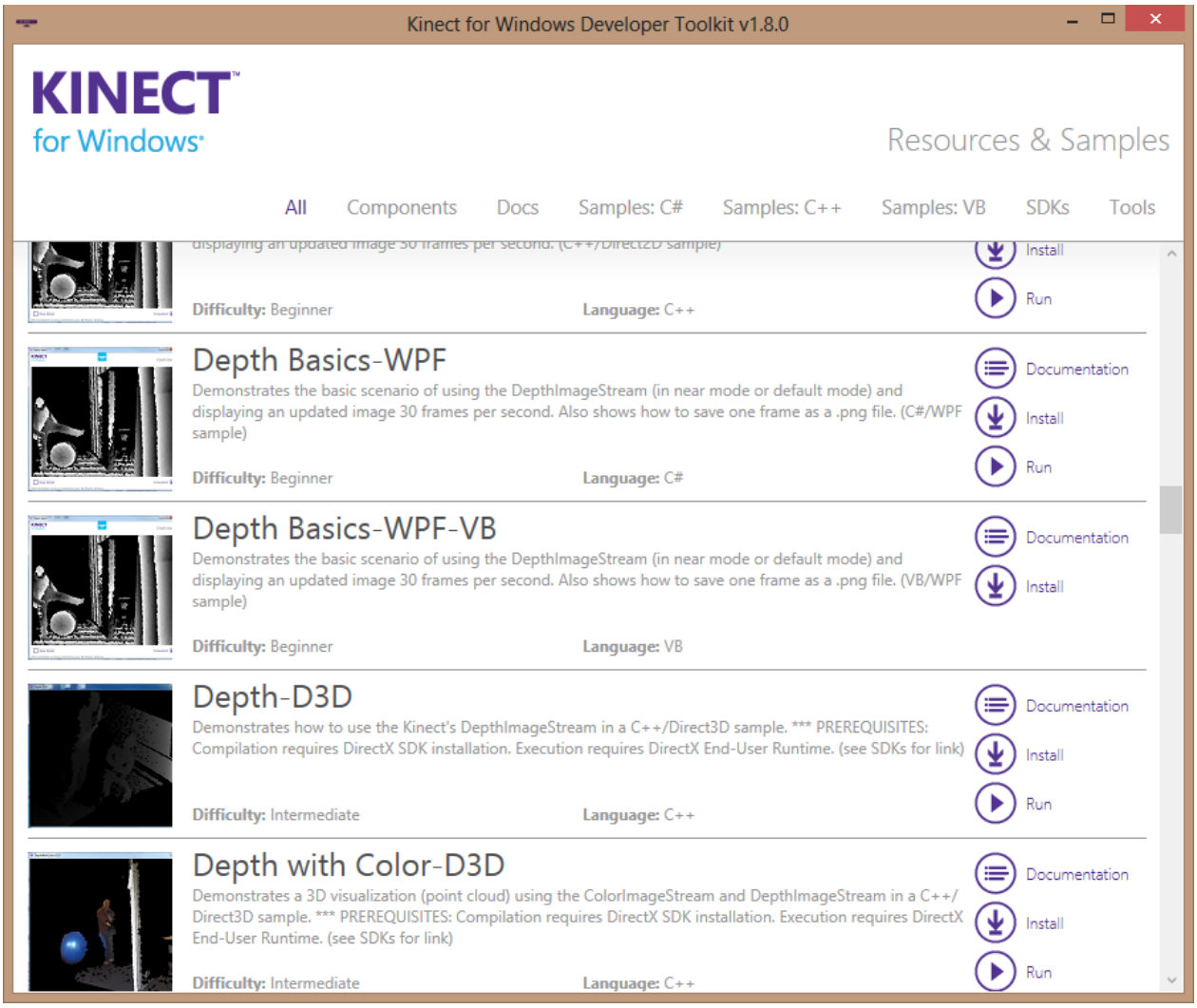
Ramki szkieletu które obrazuje rysunek 3.11 używają bardziej złożonej struktury danych. **SkeletonFrame** udostępnia kopię listy wykrytych szkieletów w kolejności od najbliższego metodą **CopySkeletonDataTo**() podobnie jak poprzednie klasy. Sam szkielet (**Skeleton**) zawiera kolekcje kończyn (**Joints**) które przedstawione są jako punkty w układzie współrzędnych (x,y,z) gdzie współrzędna „z" jest odległością od kontrolera. Pole pozycji (**Position**) zwraca położenie kończyny kręgosłupa jako środka całego szkieletu. Kolekcja kości (**BoneOrientations**) zwraca pary kończyn które się ze sobą łączą przy tworzeniu szkieletu. Wiec w kolekcji tej znajdziemy połączony prawy łokieć z prawą ręką i prawym ramieniem ale nie będzie on połączony z głową.

Oprócz wyżej wymienionych strumieni sensor udostępnia też strumień audio zawierający nagrane polecenia oraz swoją pozycję względem grawitacji zaczytaną z wewnętrznego akcelerometru.

3.5 Pozostałe oprogramowanie

Wraz z SDK firma Microsoft udostępniła szereg przykładów dostępnych przez galerię projektów Kinect Developer Toolkit oraz narzędzie wspierające testowanie Kinect Studio.

Galeria przykładów (zaprezentowana na rysunku 3.12) zawiera projekty z przykładami użycia poszczególnych funkcji sensora w trzech wspieranych językach (C++,C#,VB).



Rys 3.12: Kinect Toolkit

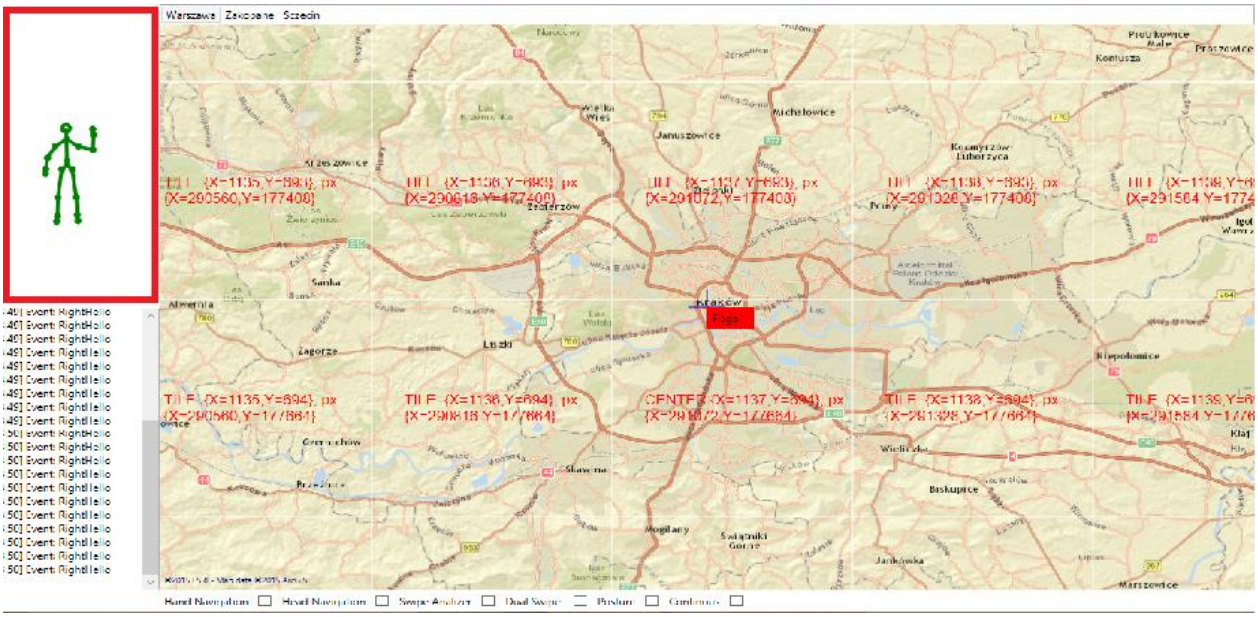
W prosty sposób umożliwia wdrożenie się i rozpoczęcie projektu przez rozbudowę przykładowej aplikacji. Kinect Studio jest prostym narzędziem umożliwiającym nagrywanie oraz odtwarzanie sekwencji ramek. Narzędzie to bardziej szczegółowo zostanie omówione w rozdziale piątym.

4. Problemy testowalności kontrolera

Kontroler ruchu jest narzędziem HID (Human Input/Interface Device). Jak inne narzędzia tego typu pośredniczy w komunikacji między człowiekiem a komputerem. Lecz jest on bardziej zaawansowany niż narzędzia takie jak mysz komputerowa, klawiatura czy joystick, gdyż jego wejściem jest przestrzeń trójwymiarowa. Dlatego też proces testowania aplikacji wykorzystujących dane z sensorów jest skomplikowany i problematyczny. W poniższym rozdziale opisane zostały problemy testowania części aplikacji których zadaniem jest interpretacja danych napływających z kontrolera. Pozostałe części aplikacji zostały pominięte gdyż powinny zostać przetestowane w konwencjonalny sposób.

4.1 Diagnostyka

Kontroler Kinect nie zawsze perfekcyjnie wykrywa sylwetkę gracza dlatego wysoce pomocnym przy tworzeniu aplikacji jest podgląd nadchodzących ramek. Dzięki temu w prosty sposób możemy stwierdzić i zweryfikować poprawność danych wejściowych. Dlatego niemal każdy programista zaczynający pisać aplikacje oparte o kontroler ruchu w pierwszym kroku zacznie od wyświetlenia danych napływających z kontrolera w celach diagnostycznych. Podgląd „na żywo" bo taki jest zazwyczaj cel takiego zabiegu zostaje wiec umieszczony w interfejsie naszego programu.



Rys 4.1: Zrzut ekranu aplikacji demo GMap

Na rysunku 4.1 przedstawiony został zrzut ekranu interfejsu aplikacji GMap. Jest to prosta aplikacja która używa kontrolera do nawigowania mapą. Celem aplikacji była demonstracja biblioteki analizującej gesty z którą miałem styczność podczas Technik obiektowych i komponentowych. Widać iż w celach diagnostycznych do interfejsu dołączony został podgląd szkieletu zaznaczony na rysunku. W wersji końcowej podgląd ten najprawdopodobniej został

usunięty gdyż nie jest częścią funkcjonalności programu i dodany został wyłącznie w celach diagnostycznych jako pomoc. Część ta jest problematyczna gdy tworzone oprogramowanie docelowo nie zawsze musi zawierać warstwę GUI (Graphic User Interface). A nawet jeśli zawiera to dodana funkcjonalność jest zbyteczna i w dalszych częściach programu zostaje usunięta. Choć rozwiązanie wbudowanego podglądu pozwala nam w prosty sposób weryfikować działanie kontrolera to podgląd ten jako iż jest częścią naszego programu nie będzie działał przy wstrzymanym działaniu naszej aplikacji w procesie debugowania. Aby spełnić to wymaganie funkcjonalność podglądu musi zostać zawarta poza kodem aplikacji. Rozwiązanie to zaoszczędzi potrzeby łączenia podglądu w tworzonym interfejsie (gdy taki istnieje).

4.2 Dane wejściowe

Jednym z głównych problemów w testowaniu aplikacji opartych o kontrolery ruch jest złożoność oraz mnogość danych wejściowych. Kinect generuje ramki z domyślną prędkością 30 ramek na sekundę gdzie każda ramka to siatka o rozdzielczości 640 na 480 w przypadku ramek głębi oraz koloru a w przypadku szkieletu jest to zestaw współrzędnych wszystkich kończyn. W normalnym przypadku do testów generowane zostają zbiory danych testowych w celu obserwacji oraz weryfikacji działania poszczególnych komponentów. Dane te są odpowiednio dobrane by testować możliwie jak największą dziedzinę funkcjonalności. Preparowanie sztucznych ramek jest niemal niemożliwe ze względu na ich złożoność. Powtarzanie gestu ręcznie z kolei jest męczące a reprodukcja tego samego ruchu przed kontrolerem którego dokładność przewyższa ludzkie oko jest wręcz niewykonalna. Przeprowadzanie testów na ciągle zmieniających się danych mija się z celem gdyż testy nie są przewidywalne oraz błędne rezultaty niekoniecznie będą się reprodukować co uniemożliwi ich debugowanie i utrudni znalezienie błędów.

Rys 4.2: Różne sylwetki ciała[2]

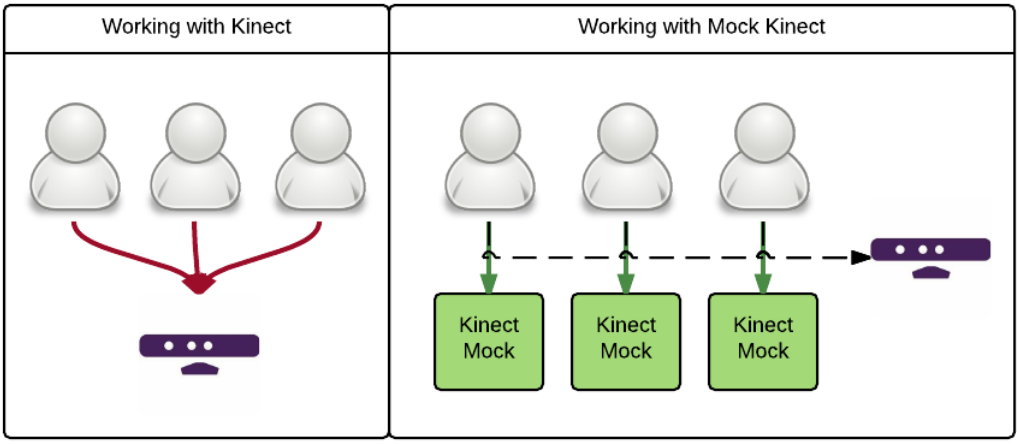
Ludzie znacznie od siebie się różnią co prezentuje rysunek 4.2. Sylwetka osoby która przeprowadza test też w duży sposób ogranicza jego testowe możliwości. Nie jest on w stanie poprawnie przetestować funkcjonalności gdyż do dyspozycji ma jedynie swoja sylwetkę. Wyobraźmy sobie dorosłego programistę który testuje aplikacje przeznaczoną dla dzieci. Dlatego do testów potrzebne jest narzędzie które byłoby w stanie dokładnie nagrać wykonane gesty w celu ich powtórnej analizy. Wtedy dane testowe mogłyby zostać zgromadzona na odpowiedniej (docelowej) grupie testowej a sam programista nie musiałby do niej należeć. Sekwencję gestów powinny móc zostać zapisane do ponownych użyć i redystrybucji.

4.3 Sprzęt

Nad aplikacją z reguły pracuje zespół programistów gdyż czasy samodzielnego wytwarzania oprogramowania „Code hero”[3] już dawno minęły. Nie zawsze do dyspozycji jest odpowiadająca ilość kontrolerów co oznacza iż członkowie zespołu muszą współdzielić sprzęt między sobą. Aby nie wstrzymywać się nawzajem najlepiej jak czas który muszą wykonać na pracy z prawdziwym kontrolerem zostanie sprowadzony do minimum.

[2] Zewnętrzna grafika (http://gvvperfcapeva.mpi-inf.mpg.de/images/ScanDB.png)

[3] Code hero – określenie odnoszące się do czasów gdy oprogramowanie wytwarzane było przez jednostki, programistów pracujących samodzielnie.

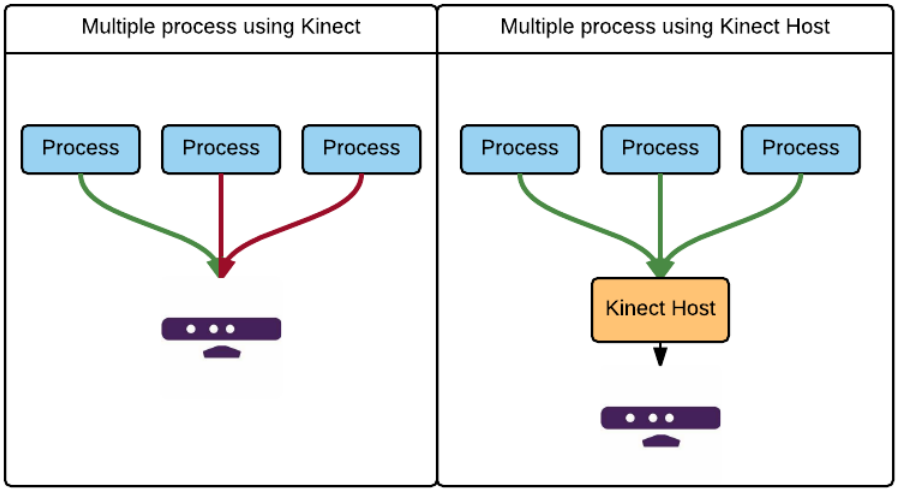


Rys 4.3: Problem współdzielenia kontrolera przez użytkowników

Na rysunku 4.3 zaprezentowane został problem pracy z jednym kontrolerem przy wieloosobowym zespole oraz jak praca wyglądać by mogła przy użyciu odpowiednich narzędzi. Jeśli programiści mają z góry wiadome dane wejściowe np. jako zbiór nagrań to obecność realnego kontrolera staje się zbyteczna. A wiec w idealnym przypadku programista nie powinien potrzebować prawdziwego kontrolera aby móc kontynuować rozwój aplikacji. A jedynie przy testach akceptacyjnych użyty powinien zostać realny kontroler. Takie podejście pomocne jest w sytuacjach gdy nie posiadamy wystarczającej ilości sprzętu na każdego pracownika. A wiec w przypadku uczelni limitowanie tematów prac opartych o kontroler nie musiałoby być uzależnione od ilości posiadanych egzemplarzy.

4.4 Współdzielenie

Kolejnym problemem w obsłudze kontrolera jest brak możliwości współdzielenie danych napływających z kontrolera przez wiele aplikacji. W chwili gdy aplikacja używa kontrolera jest on niedostępny dla innych procesów.



Rys 4.4: Problem współdzielenia kontrolera przez procesy

Wyklucza to współbieżne działanie wielu programów korzystających z kontrolera. Diagram 4.4 reprezentuje opisany problem współdzielenia kontrolera przez procesy. Aby obejść ten problem potrzebny byłby punkt centralny z którego hostowane byłyby ramki i przekazywane do zainteresowanych programów.

4.5 Testy jednostkowe

Aby móc w pełni wykorzystać zalety testów regresyjnych potrzebna jest automatyzacja jak największej ilości testów. Niestety nie jest to możliwe gdy części aplikacji odpowiadające za analizę danych z kontrolera wymagają podpiętego i zainicjalizowanego sprzętu. Głównym celem testów jednostkowych jest w miarę możliwości odseparowanie zależności modułów od siebie i skupieniu się na testowanym komponencie. Daje to możliwość zawężenia pola poszukiwań w przypadku negatywnego rezultatu testu. Dobrym rozwiązaniem byłby komponent który w udawałby „mockował" kontroler dostarczając identyczny interfejs oraz pozwalał na kontrolowanie jego działanie z poziomu kodu. Takie otwarte API spowodowałoby iż nie byłoby problemem objęcie testami i zautomatyzowanie wyżej wymienionych partii kodu.

4.6 Podsumowanie problemu do rozwiązania

Oto podsumowanie problemów wymienionych w tym rozdziale które napotykane są przy programowaniu aplikacji używających kontrolera Kinect:

- Brak prostego sposobu na podgląd i wizualizację danych z kontrolera
- Problem z preparowaniem oraz reprodukcją danych testowych
- Konieczność posiadania aktywnego kontrolera do pracy
- Brak możliwości współdzielenia sensora przez wiele aplikacji

Problemy te choć utrudniają pracę z kontrolerem i sprawiają iż proces testowania oprogramowania używającego sensora Kinect jest nietrywialny mogą zostać rozwiązane w następujący sposób:

- Zewnętrzna aplikacja udostępniająca podgląd wszystkich rodzajów ramek
- Możliwość nagrania i odtworzenia gestów
- Możliwość pracy bez kontrolera korzystając z nagrań bądź innego źródła
- Komunikacja ramek między wieloma programami na zasadzie client-host

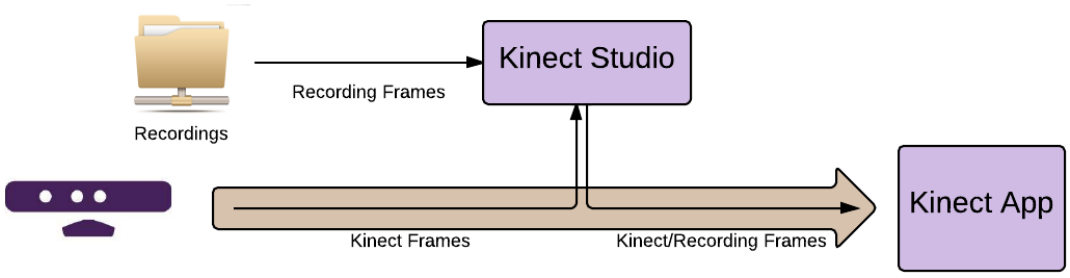
Rozwiązania te zawarte powinny zostać w zestawie narzędzi którego celem byłoby ułatwienia procesu testowania oprogramowania opartego o kontroler.

5. Metody rozwiązania problemu

Rozdział ten przedstawia już istniejące oraz stworzone w ramach pracy narzędzia wspomagające proces testowania aplikacji opartych o kontroler Kinect. Opisane zostaną funkcjonalności każdego znalezionego sposobu na testowanie kodu kontrolera. Na koniec podejścia porównane zostają przez pryzmat realizacji problemów opisanych w poprzednim rozdziale.

5.1 Microsoft Kinect Studio

Powszechnie stosowane podejście TDD w ramach pracy z kontrolerem Kinect może zostać zapewnione poprzez narzędzie Kinect Studio. Narzędzie to dostarczane jest wraz z oficjalnym SDK do pracy z kontrolerem. Umożliwia ono podgląd podstawowych danych wyjściowych kontrolera jak również pozwala na nagranie oraz odtwarzanie wcześniej przygotowanych gestów symulując pracę kontrolera.



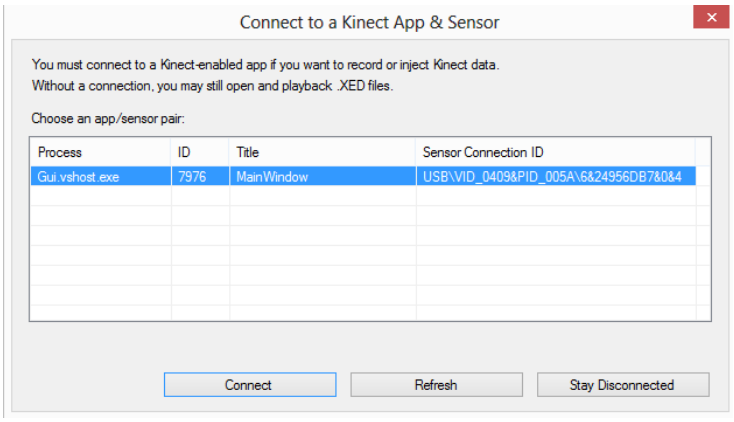
Rys 5.1: Kinect Studio data flow

Na diagramie 5.1 przedstawiony został model przepływu danych przy pracy z narzędziem Kinect Studio. Należy zauważyć iż studio wpina się bezpośrednio w połączenie miedzy aplikacją a kontrolerem. Z tego powodu użytkownik nie jest w stanie kontrolować sposobu połączenia oraz przepływu danych miedzy Kinectem, Kinect Studiem oraz swoją aplikacją. Samo studio nie jest w stanie nawiązać samodzielnie połączenia z aplikacją dlatego konieczny jest aktywny kontroler. Zakres funkcjonalności narzędzia jest następujący:

- Podglądu ramek (z nagrania bądź na żywo)
  - Podgląd ramek koloru
  - Podgląd ramek głębi
  - Podgląd rzutu głębi oraz koloru na scenę 3d
- Nagranie gestów
- Zapis nagrania do pliku

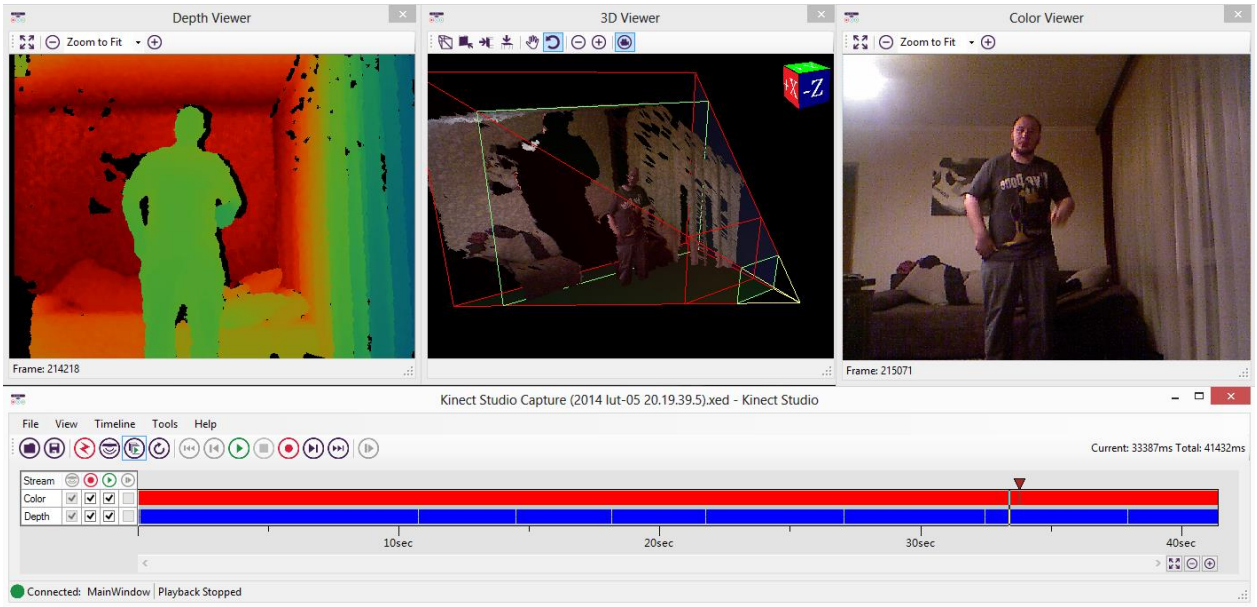
- Wczytanie nagrania z pliku
- Podstawowe operacje na nagraniu
  - Odtworzenie
    - W trybie automatycznym
    - W trybie krokowym
  - Wstrzymywanie
  - Wznawianie
  - Przewijanie

Narzędzie przy uruchomieniu pokazuje okno gdzie użytkownik musi wybrać działającą aplikację opartą na kontrolerze Kinect. Okno to zaprezentowane zostało na rysunku 5.2.



Rys 5.2: Kinect Studio okno połączenia

Następnie użytkownik ma dostęp do prostego GUI które w użytkowaniu przypomina powszechne odtwarzacze video gdzie role filmu pełnią nagrania. Użytkownik może podglądać na żywo stan ramek, bądź wczytać/zapisać nagranie i odtworzyć je automatycznie zastępując ramki napływające z kontrolera. Przydatną funkcjonalność pełni tryb krokowy w którym ramki wysyłane są pojedynczo. Narzędzie działa wyłącznie w trybie administratora i wymagane jest podpięcie pod już działającą aplikacje.



Rys 5.3: Zrzut ekranu aplikacji Kinect Studio

Na rysunku 5.3 przedstawiony jest screenshot z działania aplikacji KinectStudio. Największym atutem narzędzia jest brak jakiejkolwiek ingerencji w kod programu. Aplikacje używające kontrolera nie muszą wiedzieć o Kinect Studio. W tryb odtwarzania wejść możemy w trakcie działania programu w dowolnym momencie nie musząc restartować programu. Narzędzie to idealnie sprawdza się do małych projektów gdzie testów nie jest dużo i ręczne ich uruchamianie nie stanowi problemu.

Niestety narzędzie to jest aplikacją zamkniętą i na chwile obecną nie daje możliwości dostępu do jego funkcjonalności poprzez bibliotekę. Tym sposobem użytkownik nie jest w stanie przygotować testów jednostkowych swojej aplikacji z jego użyciem. I choć narzędzie nadaje się do przygotowania końcowych scenariuszy akceptacyjnych to z powodu braku możliwości ich automatycznego uruchomienia (chyba że poprzez narzędzia które „przeklikiwały" by się poprzez proste GUI) nie nadaje się do większych projektów. Poza tym nie daje ono podglądu ramek szkieletu (widoczne są tylko ramki głębi oraz koloru) oraz nie ma możliwości edycji zapisanych danych. Największym jednak minusem jest fakt iż Kinect Studio potrzebuje aktywnego kontrolera do swojej pracy. Dlatego pomimo gotowych sekwencji gestów użytkownik nie jest w stanie testować swojego kodu bez kontrolera.

Podsumowując Kinect Studio zapewnia podstawowe funkcjonalności do debugowania i ręcznego testowania. Niestety użytkownik nie jest w stanie w pełni zautomatyzować testów swojej aplikacji przy użyciu tego narzędzia.

5.2 MockKinect

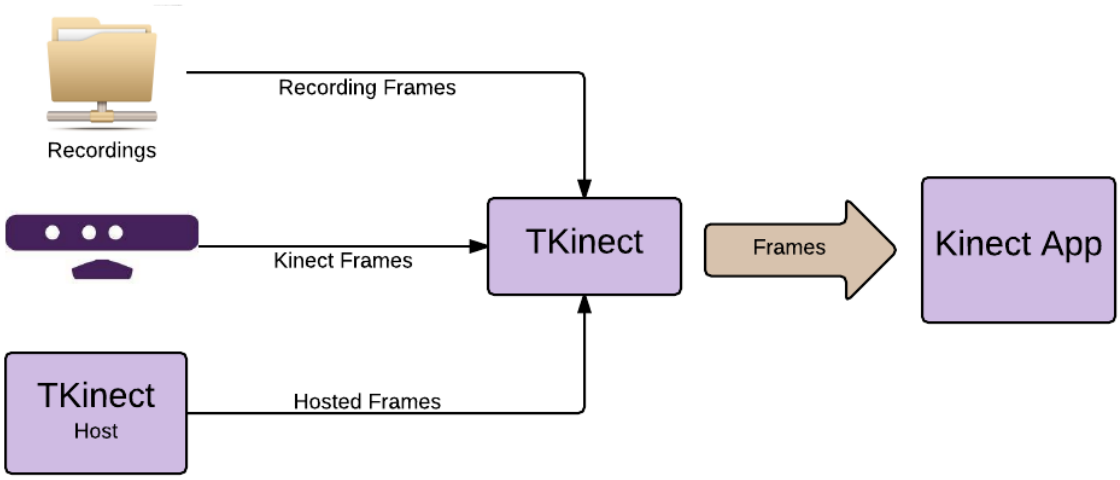
W sieci znaleziony także może zostać projekt MockKinect (http://mockinect.codeplex.com) którego głównym celem jest nagrywanie i odtwarzanie sekwencji gestów. Jest on w stanie nagrać dane z kontrolera oraz odtworzyć je później bez konieczności posiadania Kinecta. Obsługuje jedynie ramki koloru oraz głębi nie dając

możliwości zapisu i wczytania ramek szkieletu. Co sprowadza go do zwykłej kamery z obrazem głębokości. W dokumentacji projektu zamieszczono informację o wsparciu ramek szkieletu. Niestety projekt od dłuższego czasu (2011 roku) nie jest rozwijany i został raczej opuszczony dlatego nie został on bardziej dogłębnie zanalizowany w tej pracy.

5.3 Stworzone narzędzie TKinect i TKinect Studio

W ramach pracy stworzony został zestaw narzędzi o nazwie TKinect którego celem jest umożliwienie wypełni zautomatyzowanego procesu testowania aplikacji opartych o kontroler ruchu Kinect. Głównym celem było stworzenie biblioteki której użytkownik mógłby użyć do pokrycia kodu każdym rodzajem testu zaczynając od testów jednostkowych a kończąc na testach akceptacyjnych. Następnie jako przykład użycia i demonstracja możliwości biblioteki TKinect została opracowana aplikacja graficzna TKinect Studio pełniąca rolę podobną do Kinect Studio opisanego w poprzednim rozdziale. A wiec w stworzony zestaw narzędzi wchodzą:

- **TKinect** - biblioteka umożliwiającą testowanie kodu
- **TKinect Studio** - aplikacja używająca biblioteki TKinect.

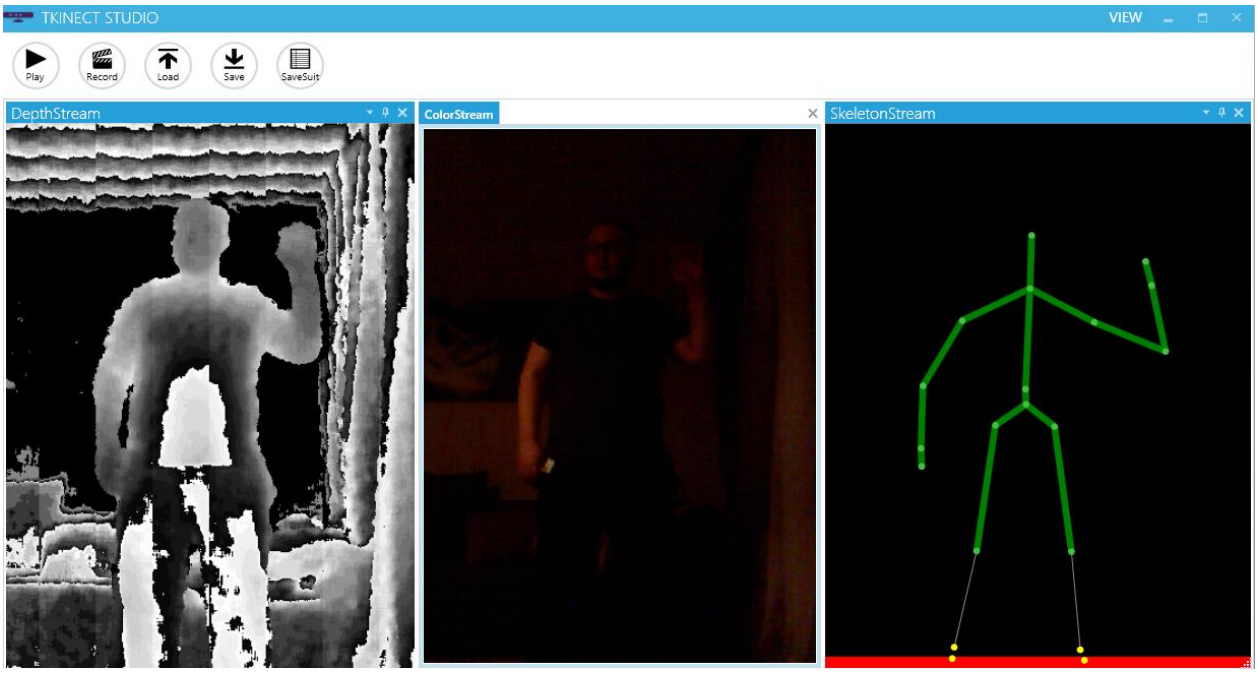


Rys 5.4: TKinect data flow

Na diagramie 5.4 przedstawiony został przepływ danych z użyciem biblioteki TKinect. Należy zwrócić uwagę iż w porównaniu z sztywnym połączeniem miedzy Kinectem i aplikacja Kinect Studio tutaj kontroler jest tylko opcjonalnym źródłem danych na równi traktowanym z inną instancją TKinect w trybie aplikacji hostującej rami czy pliku z zapisem sekwencji gestów. W ten sposób to programista decyduje co będzie źródłem danych a sama aplikacja jest niezależna od obecności kontrolera.

Zakres funkcjonalności narzędzia jest następujący:

- Podglądu ramek (z nagrania bądź na żywo)
  - Podgląd ramek koloru
  - Podgląd ramek głębi
  - Podgląd ramek szkieletu
- Nagranie gestów
- Zapis nagrania do pliku
- Wczytanie nagrania z pliku
- Kompozycja zestawów nagrań jako sekwencji (Test Case)
- Możliwość pracy bez aktywnego kontrolera
- Możliwość nadawania ramek (host mode)
- Możliwość odbierania ramek (client mode)
- Otwarte API do automatyzacji testów



Rys 5.5: TKinect Studio overview

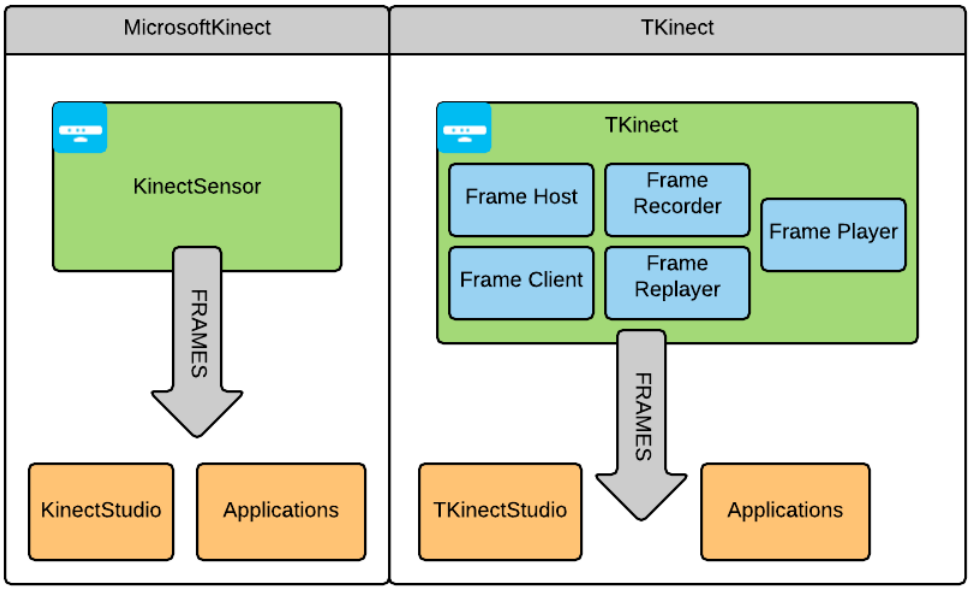
Aby w prosty sposób pokazać możliwości stworzonej biblioteki przygotowana została aplikacja okienkowa na wzór aplikacji Kinect Studio. Rysunek 5.5 prezentuje zrzut ekranu z jej działania.

6. TKinect i TKinect Studio

Stworzonym zestawem narzędzi jest biblioteka TKinect oraz aplikacja TKinect Studio. W skład bibliotek wchodzi wiele komponentów które zapewniają funkcjonalności nagrywania, komunikacji czy wyświetlania. W tym rozdziale przedstawione zostaną główne aspekty oraz problemy jakie stawiane były bibliotece oraz sposób w jaki je rozwiązano.

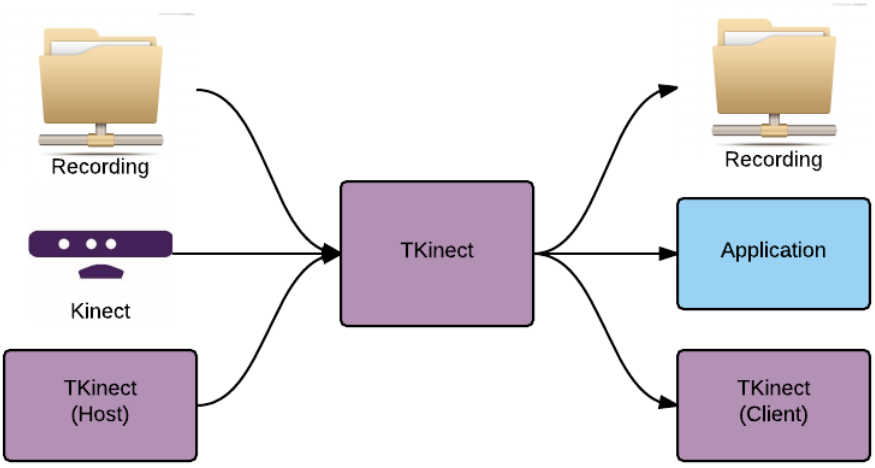
6.1 Zastąpienie sensora

Przy zwykłym użyciu Kinecta korzystając wyłącznie z natywnej biblioteki użytkownik korzysta z klasy **KinectSensor** opisanej w wcześniejszym rozdziale w celu inicjalizacji dostępu do danych sensora. Aby zapewnić wymagane funkcjonalności przechwycona została kontrola nad źródłem oraz przeznaczeniem ramek. A wiec zastąpiony został **KinectSensor** obiektem pochodzącym z stworzonej biblioteki odpowiedzialnym za dostęp do sensora. Klasa ta nosi nazwę projektu czyli TKinect.



Rys 6.1: Funkcja obiektu TKinect

Diagram 6.1 ilustruje rolę tego obiektu jaką jest agregacja wszystkich funkcjonalności biblioteki (wrapping). Opakowuje on komponenty odpowiedzialne za komunikację oraz nagrywanie opisane w poniższych podrozdziałach. Kolejnym jego celem jest zastąpienie (mocking) klasy kontrolera z natywnej biblioteki Microsoftu (**KinectSensor**) opisanego w początkowym rozdziale. Obiekt ten w dużym uproszczeniu zarządza przepływem ramek w aplikacji która go używa. Źródło i cel ramek może zostać ustawione na wiele sposobów.



Rys 6.2: TKinect In/Out

Diagram 6.2 prezentuje możliwe do zdefiniowania źródła i cele ramek. A wiec źródłem ramek w naszej aplikacji może być:

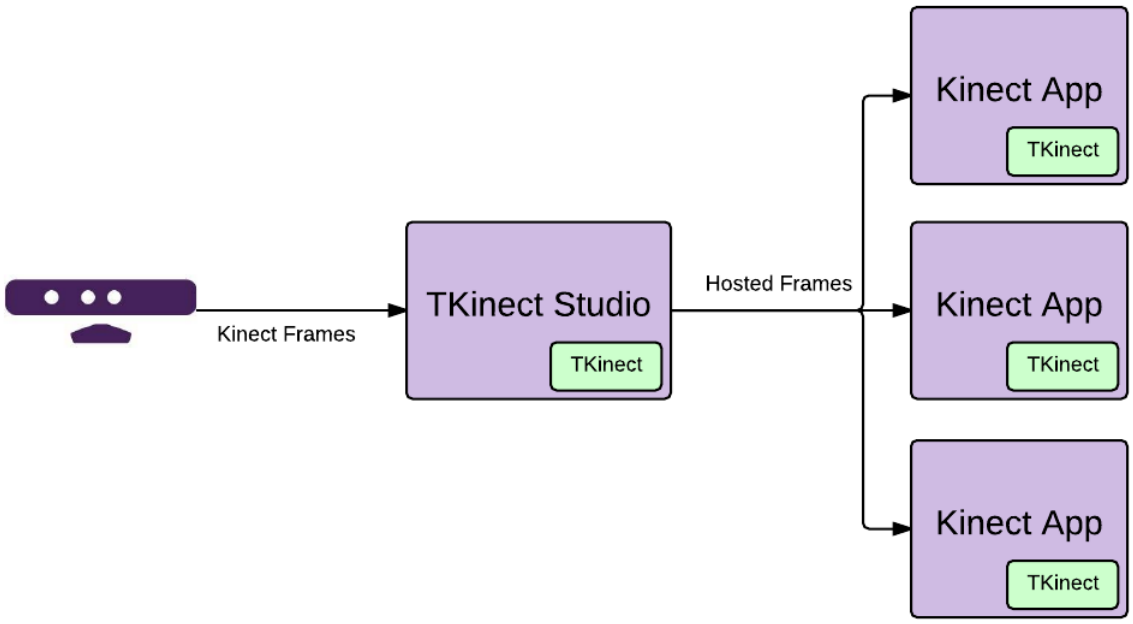
- Prawdziwy sensor - poprzez podpięcie eventów sensora w do odpowiednich funkcji.
- Inna biblioteka TKinect - gdy wybrany zostanie tryb klienta a źródłem jest biblioteka w trybie hosta.
- Nagrania z pliku – gdy wczytane zostaną plik z sekwencją w trybie odtwarzania.

Cel naszych ramek może być następujący:

- Funkcje analizujące aplikacji – poprzez podpięcie pod eventy biblioteki TKinect.
- Inna biblioteka TKinect – gdy wybrany zostanie tryb hosta a celem jest inna biblioteka w trybie klienta.
- Plik z nagraniem – gdy wywołanie zostanie przekierowanie ramek do pliku w trybie nagrywania.

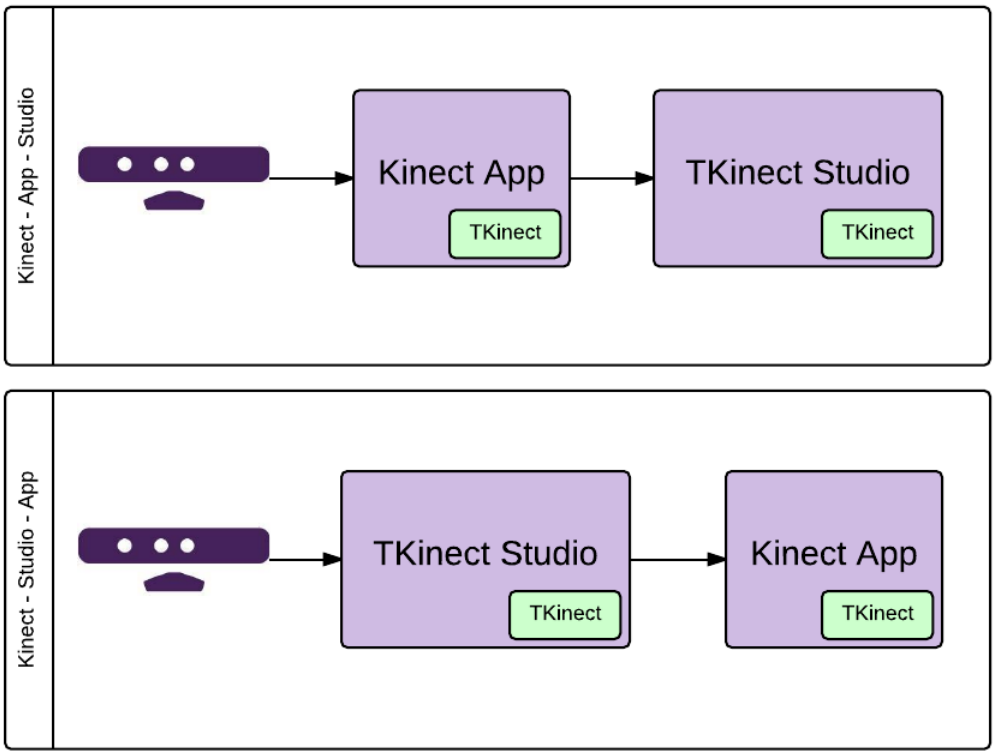
6.2 Komunikacja

Przy podstawowym użyciu kontrolera Kinect po jego inicjalizacji aplikacja przejmuje nad nim całkowitą kontrolę. W ten sposób żaden inny program poza narzędziem Kinect Studio nie jest w stanie współbieżnie korzystać z kontrolera. Jest to spora wada w sytuacji gdy użytkownik chciałby łączyć działanie wielu aplikacji lub choćby użyć swojej własnej aplikacji do testowania oraz prostego podglądu. Aby rozwiązać ten problem wprowadzona została możliwość przekierowania danych z jednej aplikacji do drugiej gdy obie używają biblioteki TKinect. Diagram 6.3 przestawia ten model komunikacji.



Rys 6.3: TKinect hosting frames

Jedna aplikacja (TKinect Studio) pełni rolę hosta dla reszty aplikacji serwując im dane sensora. Podejście to umożliwia współdzielenie kontrolera przez wiele aplikacji. Jedynym narzutem jest możliwość istnienia tylko jednego hosta na kontroler gdyż biblioteka wciąż operuje na natywnej bibliotece która wyznacza takie ograniczenie. Model ten oddziela warstwę sprzętu (kontrolera) od samej aplikacji. Swoboda ta w procesie wymiany ramek nie narzuca konieczności posiadania aktywnego kontrolera gdyż źródłem ramek może być inna aplikacja. Każda aplikacja używająca biblioteki TKinect może pełnić rolę hosta bądź klienta poprzez odpowiednią inicjalizację.

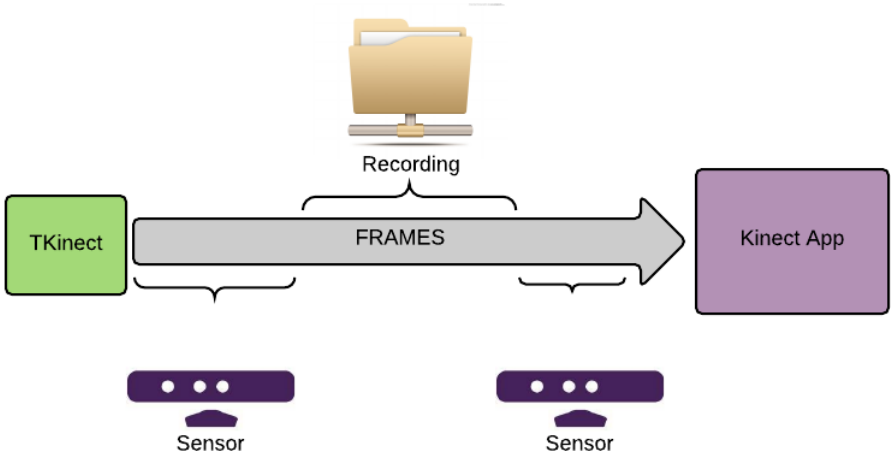


Rys 6.4: TKinect – TKinect Studio

Użytkownik decyduje między wariantami komunikacji przedstawionymi na diagramie 6.4. W pierwszym ustawieniu aplikacja użytkownika pełni rolę hosta dla aplikacji TKinect Studio oraz odbierając ramki z kontrolera. W drugim ustawieniu bardziej przypominającym użycie Kinect Studio aplikacja TKinect Studio odbiera ramki z kontrolera i pełni rolę hosta wysyłając je do aplikacji użytkownika. A wiec wybór między nadawaniem oraz odbieraniem ramek jest wypełni modyfikowalny.

6.3 Nagrywanie

Przy testowaniu aplikacji używających kontrolera ruchu często nie jesteśmy w stanie zreprodukować sekwencji gestu która wywołała niespodziewane działanie aplikacji. Ruchy ludzkie są niedokładne i żaden człowiek nie jest w stanie wykonać dwukrotnie takiego samego gestu. Ponadto testując aplikacje do dyspozycji mamy swoją własną sylwetkę co może spowodować iż u osób mniejszych/większych aplikacja nie rozpozna docelowego gestu lub zachowa się w nieoczekiwany sposób.



Rys 6.5: TKinect record/replay

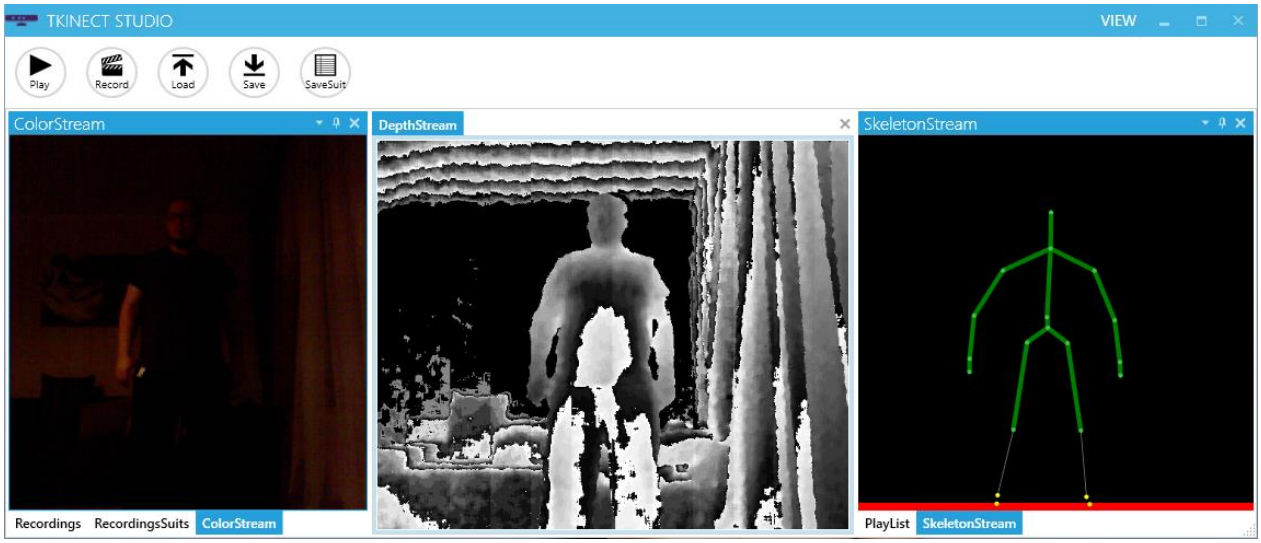
Diagram 6.5 przedstawia możliwość określenia źródła danych wejściowych w bibliotece TKinect. Może nim być plik z nagraniem lub sam kontroler. Aplikacja używająca biblioteki nie zna źródła wiec nie wymagana jest obsługa funkcji odtwarzania w samej aplikacji. Samo wyjście może zostać w każdej chwili przekierowane do pliku w celu późniejszego odtworzenia.

6.4 Wyświetlanie

Aby umożliwić ciągły podgląd danych odtwarzanych bądź napływających z kontrolera Kinecta biblioteka TKinect rozszerzona została o komponenty które w prosty sposób umożliwiają wizualizacje każdego rodzaju ramek. Wizualizacje te nie są ze sobą w żaden sposób zależne co umożliwia selektywny wybór rodzaju danych wejściowych pobieranych z kontrolera. A wiec przy wyłączonym strumieniu głębokości nadal dostępne będą pozostałe podglądy danych czyli podgląd szkieletu oraz koloru. Kod wizualizacyjny nie jest w żaden sposób skomplikowany a załączenie tej funkcjonalności ma na celu ułatwienie wdrożenia się użytkownika w obsługę Kinecta i zaoszczędzenie mu pisania tak prostej funkcjonalności.

6.5 TKinect Studio

W celu demonstracji możliwości biblioteki TKinect stworzona została aplikacja TKinect Studio. Posiada on podgląd trzech rodzajów ramek którego źródłem może być nagranie bądź sensor lub też inna aplikacja gdy studio zostanie uruchomione w trybie klienta. Podgląd ten demonstruje rysunek 6.6. Warstwa Gui została zaimplementowana z użyciem dokowalnych kontrolek firmy Telerik dzięki czemu w łatwy sposób możemy dostosować wygląd studia.



Rys 6.6: TKinect Studio podgląd ramek

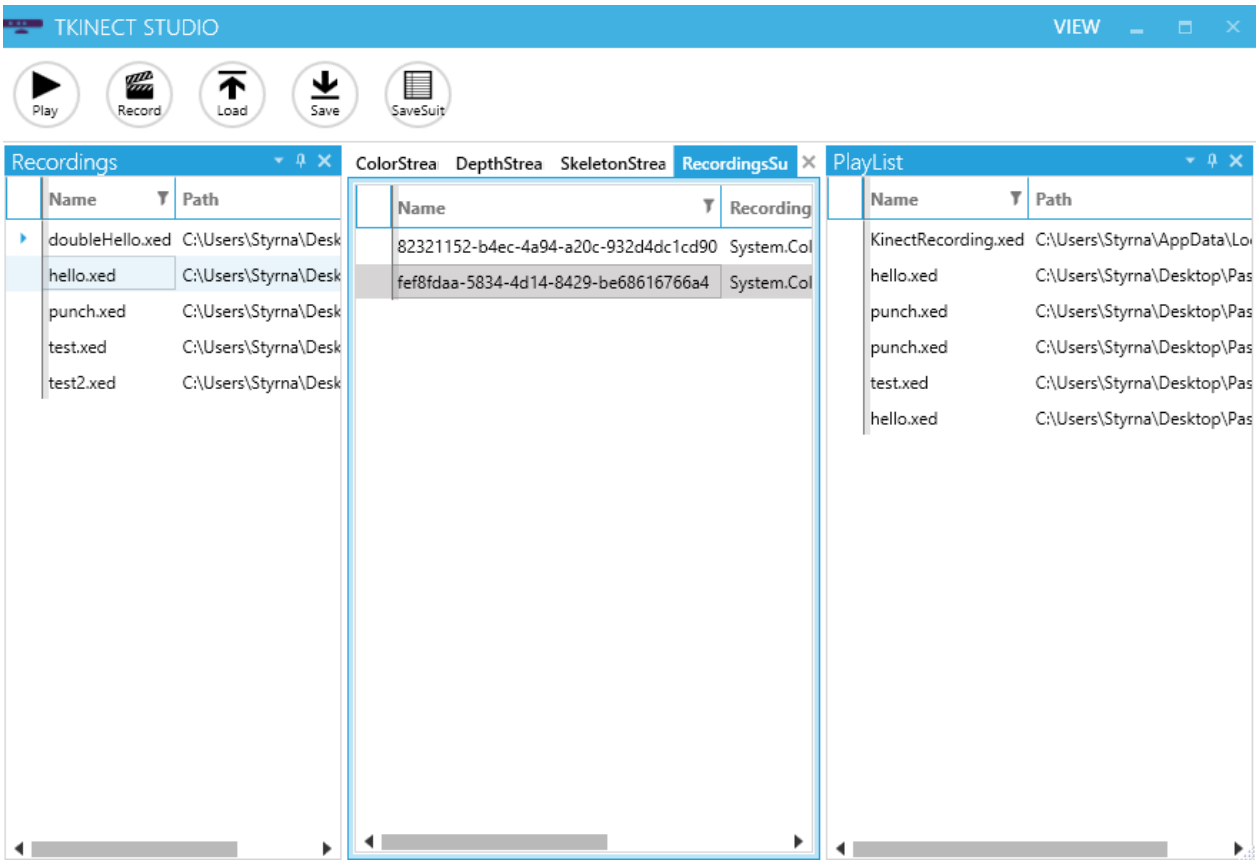
Aplikacja posiada funkcjonalność nagrania gestu. W tym celu należy nacisnąć przycisk **Record** który zapali się na niebiesko gdy studio nagrywa gest co ilustruje rysunek 6.7. Po wykonaniu gestu należy nacisnąć przycisk ponownie a plik z nagraniem zapisany zostanie w tymczasowym folderze dla aplikacji (C:\Users\[User]\AppData\Local\Temp) o nazwie KinectRecording.xed.

Rys 6.7: TKinectStudio przyciski

Aplikacja ponadto posiada możliwość budowania sekwencji nagrań, zapisywania ich w formie modyfikowania do odtworzenia. Funkcjonalności te dostępne są poprzez trzy tabele **Recordings**, **PlayList**, **RecordingSuits** zobrazowane są na rysunku 6.8.

- Tabela **Recordings** przedstawia dostępne nagrania. Aby dodać nagranie do tabeli **Recordings** należy załadować plik z nagraniem przez przycisk **Load** lub upewnić się że nasze nagranie umiejscowione jest w folderze **PassedData** znajdującym się w miejscu odpalenia studia.
- Tabela **PlayList** zawiera aktualny stan nagrań czekających na odtworzenie. Aby dodać nagranie do listy wystarczy dwukrotnie kliknąć na interesujące nas nagranie z tabeli **Recordings**. Dwukrotne kliknięcie na nagranie w tabeli **PlayList** spowoduje jego usunięcie z listy. Po naciśnięciu przycisku **Play** nagrania zostają odtwarzane w kolejności od góry, gdzie po każdym skończonym nagraniu jego wpis znika. Odtwarzanie kończy się gdy tabela **PlayList** jest pusta.
- Tabela **RecordingSuits** służy do zapisania oraz odtwarzania wcześniej skomponowanych playlist. W celu zapisania playlisty należy użyć przycisku SaveSuit który spowoduje dodanie wiersza który reprezentuje zapisaną playliste. Aby wczytać

interesującą nas playliste wystarczy dwukrotnie kliknąć na jej wiersz w tabeli **RecordingSuits** co spowoduje zapełnienie tabeli **PlayList** jego zawartością.



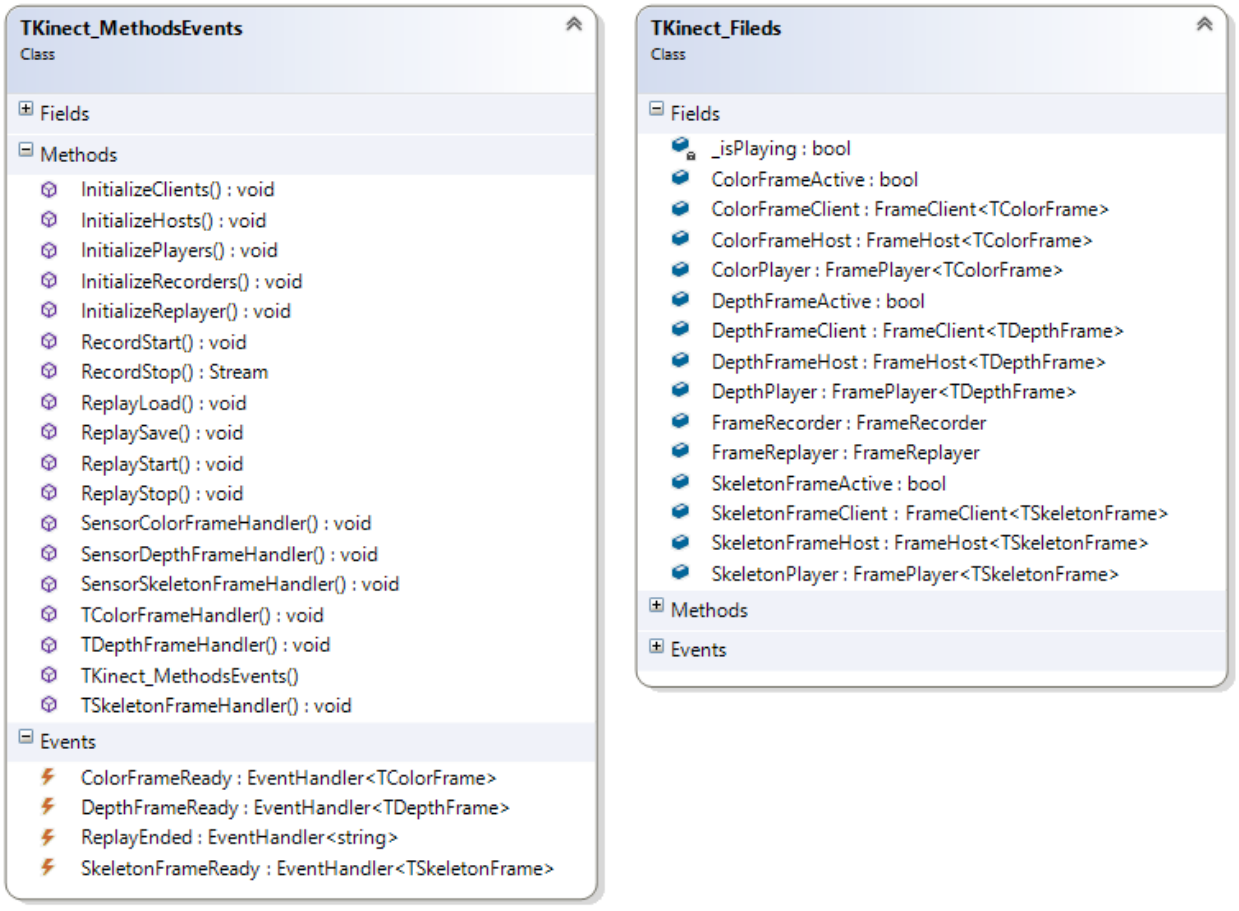
Rys 6.8: Tabele Recordings PlayList oraz RecordingSuits

7. TKinect – szczegóły implementacyjne

Poszczególne aspekty/funkcjonalności opisane w poprzednim rozdziale zostały zrealizowane przez odpowiadające im komponenty/klasy wchodzące w skład biblioteki TKinect. W poniższym rozdziale przedstawiona zostanie ich budowa, inicjalizacja oraz użycie. Przedstawione zostaną także problemy występujące podczas implementacji oraz sposób w jaki zostały rozwiązane.

7.1 TKinect

Obiekt **TKinect** jako iż pełni rolę wrappera opisanego w poprzednim rozdziale posiada referencję do większości komponentów które oferuje biblioteka. Diagram 7.1 ilustruje główna klasę biblioteki.



Rys 7.1: Diagram klasy TKinect

Posiada ona komponenty odpowiadające za komunikację nagrywanie oraz wyświetlanie opisane w poniższych podrozdziałach. W celu jej użycia należy w prosty sposób stworzyć klasę TKinect.

```
//Initialize TKinect
```

```
TKinect = new TKinect.TKinect();
```

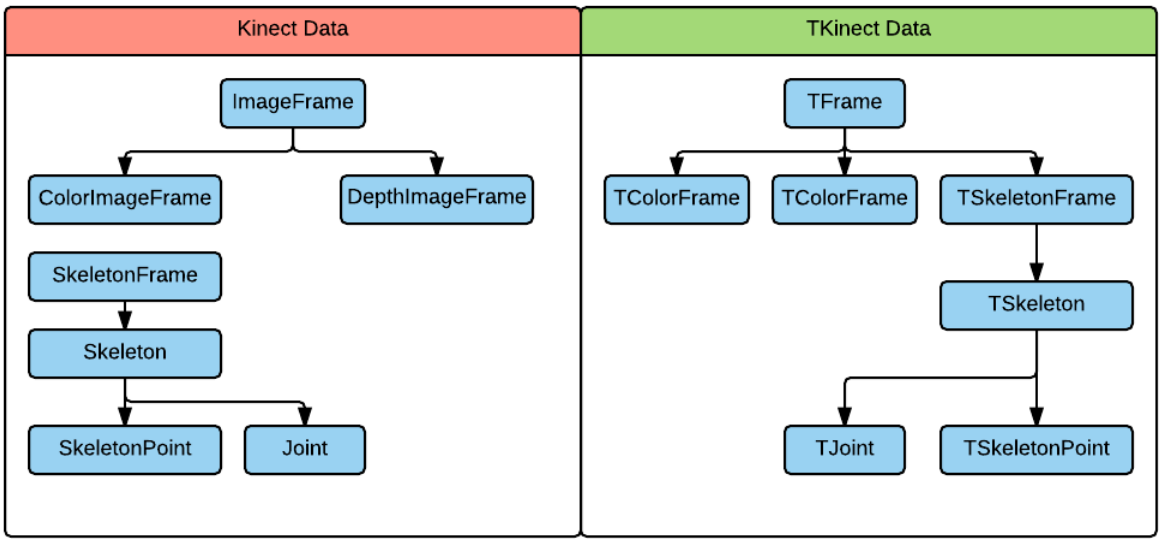
Jako iż jej głównym zadaniem jest naśladowanie klasy sensora tak wiec analogicznie wystawia ona zdarzenia pod które podpiąć należy funkcję przechwytujące nadawane ramki. Ramki odbierane pod tymi zdarzeniami mogą pochodzić od innego obiektu TKinect bądź z nagrania lub też z realnego kontrolera.

```
//Capture frames
TKinect.ColorFrameReady += MyOnColorFrameReadyFunction;
TKinect.DepthFrameReady += MyOnDepthFrameReadyFunction;
TKinect.SkeletonFrameReady += MyOnSkeletonFrameReadyFunction;
```

Następnie należy zdecydować czy realny kontroler powinien zostać podpięty pod nasz testowy sensor. Opcja ta pozwoli nam przekazywać pod wyżej ustawione funkcję dane z kontrolera. Podpięcie realnego kontrolera jest wypełni opcjonalne gdyż biblioteka nie wymusza posiadania kontrolera.

```
//Redirect real sensor frames to TKinect
sensor.ColorFrameReady += TKinect.SensorColorFrameHandler;
sensor.DepthFrameReady += TKinect.SensorDepthFrameHandler;
sensor.SkeletonFrameReady += TKinect.SensorSkeletonFrameHandler;
```

Jak nie trudno zauważyć biblioteka TKinect operuje na innych danych niż sam sensor co ilustruje diagram 7.2.



Rys 7.2: Model danych Kinect/TKinect

Niestety natywny model danych którymi operuje klasa KinectSensor nie posiada publicznego konstruktora oraz jest klasą zamkniętą (sealed). Biblioteka aby móc wczytywać, rozsyłać bądź zapisywać ramki potrzebuje możliwości tworzenia oraz powielania danych z kontrolera. Jako iż natywne klasy danych nie posiadały takiej możliwości zastąpione zostały odpowiednikami tych

klas poprzez dodanie przedrostka T w celu łatwej orientacji która klasa odpowiada której. Tak wiec klasa **SkeletonFrame** posiada odpowiednik **TSkeletonFrame** w stworzonej bibliotece. Zmiana danych jest dużym problemem gdyż wymaga podmiany typu w każdym miejscu gdzie użyte zostały dane pochodzące z sensora. Co w dużych projektach może wiązać się z koniecznością sporego refactoringu. Kolejnym problemem który wiąże się z podmianą typów jest fakt iż jeśli w późniejszej wersji klasy odpowiadające za reprezentacje danych z kontrolera zostaną zmodyfikowane stworzona biblioteka przestanie działać bądź zacznie ukrywać nowo dodane informację, gdyż nie będą one uwzględnione podczas mapowania danych z sensora na dane biblioteki.

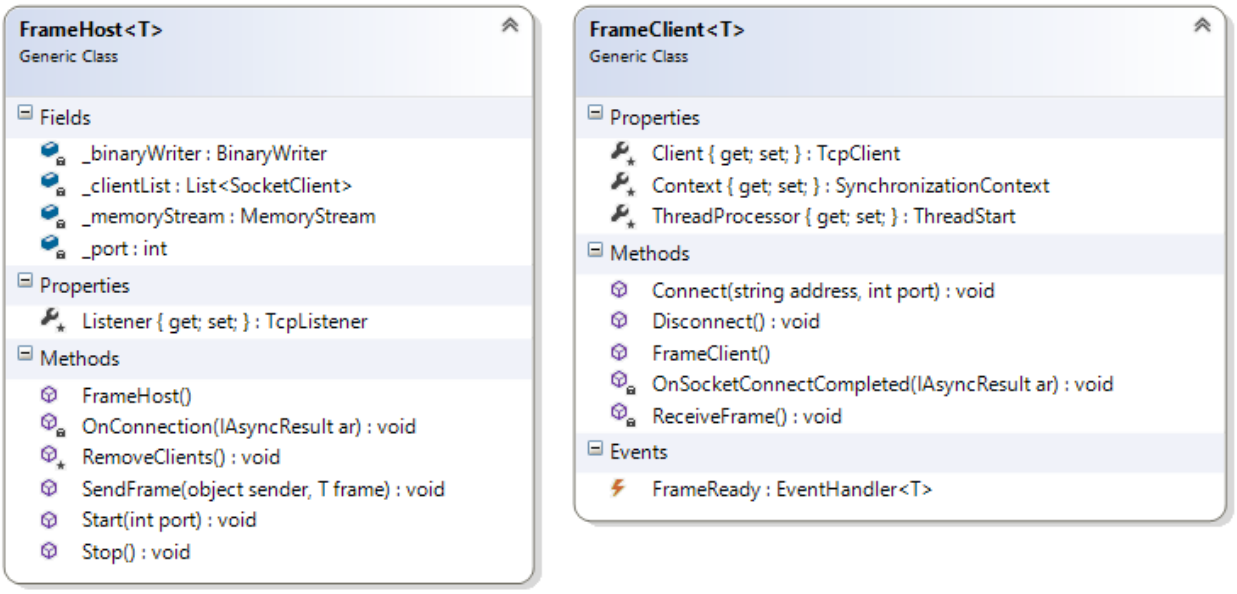
Klasa TKinect posiada referencje do obiektów odpowiedzialnych za komunikację (**FrameHosts/FrameClients**) oraz za nagrywanie (**FrameRecorder/FrameReplayer**) i mimo iż część wystawianych funkcji/zdarzeń bezpośrednio wiąże się z tymi obiektami zostały one zawarte w osobnych obiektach gdyż nie narzuca to na użytkownika używania całości biblioteki w celu skorzystania z jednej jej funkcjonalności. Obiekt TKinect posiada wszystkie komponenty gdyż pełni rolę centralnego punktu z którego użytkownik pobiera dane. Dzięki temu w przypadku gdy użytkownik chce zmienić źródło danych z kontrolera na plik nie musi ingerować w swój kod ani zmieniać źródła z obiektu **FrameClient** na obiekt **FrameReplayer**. Wystarczy jak wywoła odpowiednie funkcję klasy TKinect a logika w niej zawarta wybierze odpowiednie źródło i jego zawartość przekazywana będzie w zdarzeniach wyjściowych.

7.2 Frame Host/Client

Komunikacja z użyciem biblioteki TKinect zapewniona jest przez dwa komponenty, **FrameHost** oraz **FrameClient** gdzie:

- **FrameHost** jest klasą która zajmuje się przesyłaniem ramek do zarejestrowanych klientów oraz rejestrowaniem nowych klientów.
- **FrameClient** odpowiada za odbieranie przesyłanych danych oraz zapisywanie się u hosta.

Diagram 7.3 klas prezentuje budowę obiektów **FrameHost** oraz **FrameClient**.



Rys 7.3: FrameHost/FrameClient

FrameHost

Aby wysyłać ramki należy utworzyć instancję obiektów **FrameHost** obsługującą dany typ ramek. Następnie należy wpiąć pod zdarzenia udostępniające napływające ramki z biblioteki TKinect funkcję które wysyłać je będą z klasy **FrameHost**. Jeśli ramki nie będą napływały do metody **SendFrame** host nic nie będzie wysyłał do klientów. Dla ułatwienia inicjalizacja ta została zawarta w prostej funkcji **InitializeHosts** w klasie TKinect.

```
//InitializeHosts
var ColorFrameHost = new FrameHost<TColorFrame>();
var DepthFrameHost = new FrameHost<TDepthFrame>();
var SkeletonFrameHost = new FrameHost<TSkeletonFrame>();

TKinect.ColorFrameReady += ColorFrameHost.SendFrame;
TKinect.DepthFrameReady += DepthFrameHost.SendFrame;
TKinect.SkeletonFrameReady += SkeletonFrameHost.SendFrame;
```

Za każdy typ odpowiada osobna instancja klasy parametryzowana generycznie przez typ hostowanych ramek (**TColorFrame**/**TDepthFrame**/**TSkeletonFrame**). Ułatwia to precyzowanie oraz zarządzanie danymi które udostępniamy gdyż nie musimy inicjalizować instancji hostującej typ który nie chcemy udostępniać. Następnie należy zainicjalizować hostów na odpowiednich portach.

```
//HOST
TKinect.ColorFrameHost.Start(4501);
TKinect.DepthFrameHost.Start(4502);
TKinect.SkeletonFrameHost.Start(4503);
```

W tym momencie każdy host nasłuchiwał będzie na podanym porcie żądania rejestracji klienta. Gdy klient połączy się z hostem dodany zostanie to listy (**_clientList**). Za każdym razem gdy instancja klasy **FrameHost** dostanie odpowiednią ramkę z podpiętego zdarzenia ramka ta wysłana zostanie do każdego zarejestrowanego klienta metodą **SendFrame**.

FrameClient

Aby odebrać ramki należy utworzyć instancję obiektu **FrameClient** obsługującą dany typ ramek. Klasa udostępniać będzie nadchodzące ramki pod zdarzeniami **FrameReady**. Gdy pod zdarzenia te wpięta zostanie klasa TKinect będzie ona działała w trybie klienta odbierając napływające ramki i udostępniając je w swoich zdarzeniach.

```
//InitializeClients
var ColorFrameClient = new FrameClient<TColorFrame>();
var DepthFrameClient = new FrameClient<TDepthFrame>();
var SkeletonFrameClient = new FrameClient<TSkeletonFrame>();

ColorFrameClient.FrameReady += TKinect.TColorFrameHandler;
DepthFrameClient.FrameReady += TKinect.TDepthFrameHandler;
SkeletonFrameClient.FrameReady += TKinect.TSkeletonFrameHandler;
```

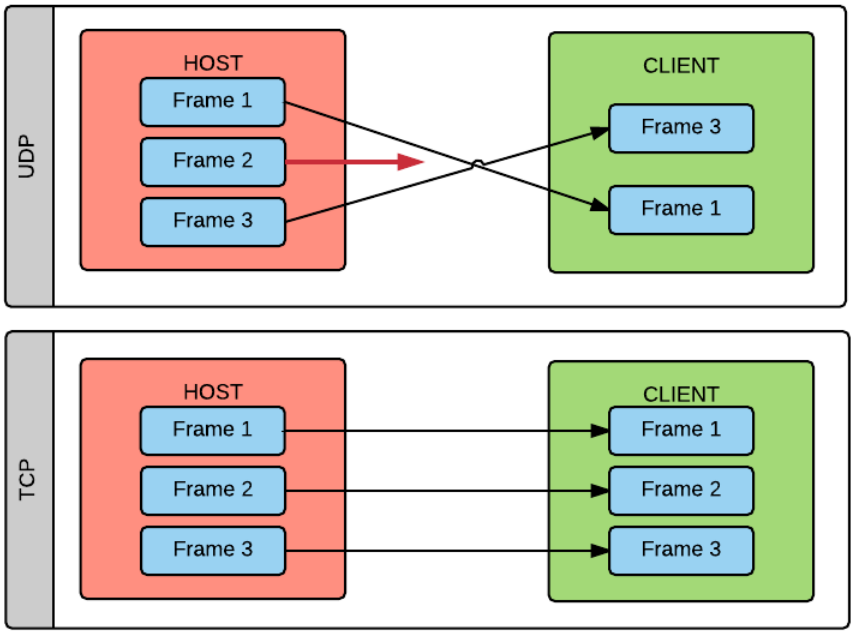
Analogicznie jak w przypadku hostów za każdy typ odpowiada generycznie typowana instancja klasy **FrameClient**. Następnie każdy klient powinien zapisać się u hosta wywołując metodę Connect w której podawany jest adres oraz port.

```
//CLIENT
TKinect.ColorFrameClient.Connect("localhost",4501);
TKinect.DepthFrameClient.Connect("localhost", 4502);
TKinect.SkeletonFrameClient.Connect("localhost", 4503);
```

Po tej operacji jeśli host wysyła dane zaczną one napływać do metod wpiętych pod zdarzenia **FrameReady**.

Komunikacja

W ramach testów rozpatrywane były dwa modele komunikacji TCP oraz UDP. Niestety choć transmisja danych z użyciem protokołu UDP jest szybsza to nie gwarantuje ona dostarczenia ramek. A nawet gdy niewielka ilość danych zostaje zgubiona to nie zapewnia ona poprawnej kolejności odebrania wysłanych komunikatów. Prowadzić to może do zakłóceń i dezinformacji co przedstawia diagram 7.4.



Rys 7.4: Komunikacja z użyciem protokołu TCP/UPD

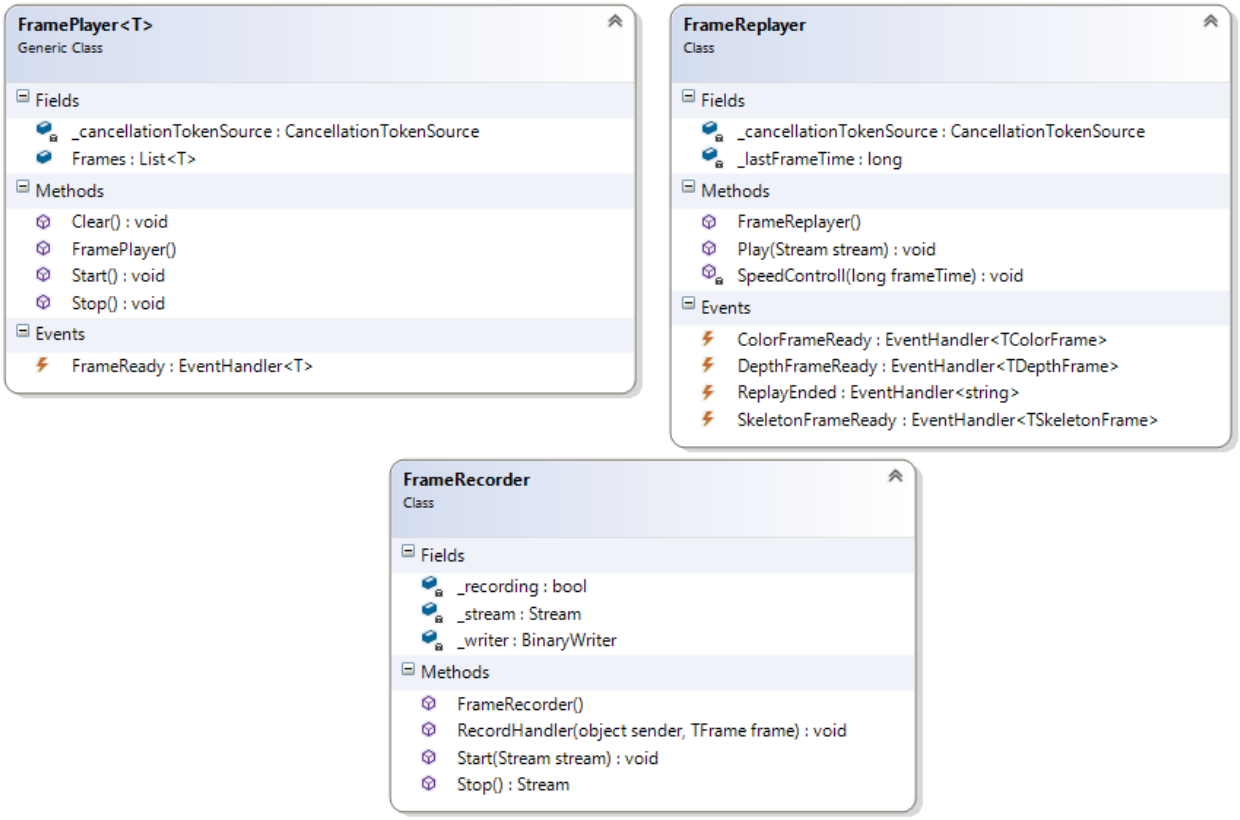
Choć odnotowane zakłócenia były znikome w końcowym rozwiązaniu użyte zostało połączenie TCP gdyż jego wydajność okazała się wystarczająca.

7.3 Frame Player/Replayer/Recorder

Funkcjonalność nagrywania oraz odtwarzania sekwencji gestów zapewniają komponenty **FrameReplayer/FramePlayer** oraz **FrameRecorder** gdzie:

- **FrameRecorder -** obiekt odpowiada za nagrywanie ramek.
- **FrameReplayer/FramePlayer -** obie klasy odpowiadają za odtwarzanie nagranych gestów. **FrameReplayer** odtwarza gesty z źródła na bieżąco natomiast **FramePlayer** ładuje wszystkie ramki do pamięci. **FramePlayer** jest szybszy ale dla długich sekwencji spowoduje zajęcie znacznej części pamięci.

Klasy te zaprezentowane zostały na diagramie 7.5.



Rys 7.5: FramePlayer/FrameRecorder/FrameReplayer

Aby nagrać sekwencje gestów należy utworzyć instancję obiektu **FrameRecorder** oraz „wpiąć" eventy napływających ramek do odpowiednich handlerów.

```
//InitializeRecorder
var FrameRecorder = new FrameRecorder();

ColorFrameReady += FrameRecorder.RecordHandler;
DepthFrameReady += FrameRecorder.RecordHandler;
SkeletonFrameReady += FrameRecorder.RecordHandler;
```

Następnie aby zacząć nagrywanie należy wywołać metodę Start podając stream jako źródło nagrania. W tym monecie do źródła nagrywane zostają każde ramki nachodzące do wcześniej podpiętych handlerów. Aby zakończyć nagrywanie należy wywołać metodę Stop a jej wynikiem jest źródło podane przy starcie nagrywania.

```
//RECORD
FrameRecorder.Start(stream);
//. . .
return FrameRecorder.Stop();
```

Aby odtworzyć nagrane ramki należy utworzyć instancję obiektu **FrameReplayer** obsługującą dany typ ramek. Następnie należy wpiąć funkcje których celem jest odbiór ramek pod odpowiednie eventy.

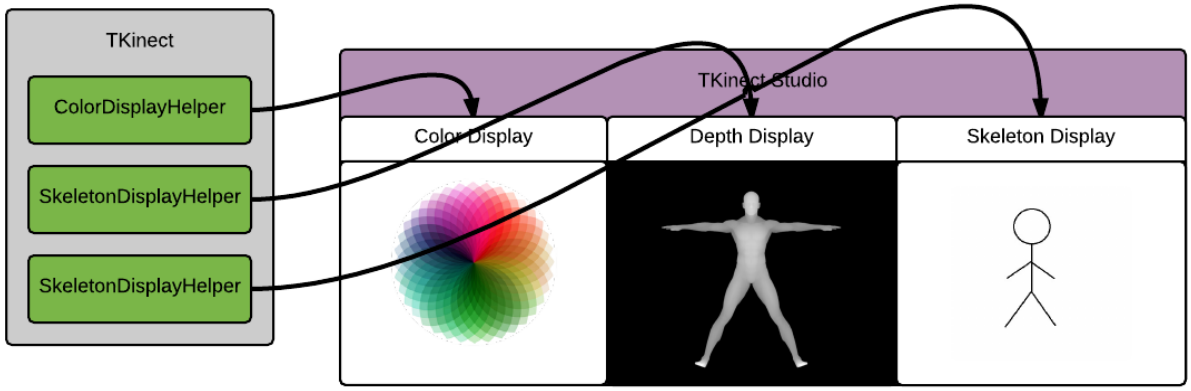
```
//InitializeReplayer
var FrameReplayer = new FrameReplayer();

FrameReplayer.ColorFrameReady += TColorFrameHandler;
FrameReplayer.DepthFrameReady += TDepthFrameHandler;
FrameReplayer.SkeletonFrameReady += TSkeletonFrameHandler;
```

Aby rozpocząć odtwarzanie należy wywołać metodę **Play** gdzie jako argument przekazane zostanie źródło z którego odtworzone zostaną gesty.

```
//REPLAY
FrameReplayer.Play(stream);
```

7.4 DisplayHelpers

Biblioteka także została rozszerzona o klasy których celem jest ułatwienie prezentacji ramek. Klasy te zostały użyte do wizualizacji danych w programi TKinectStudio co prezetuje rysunek 7.6.

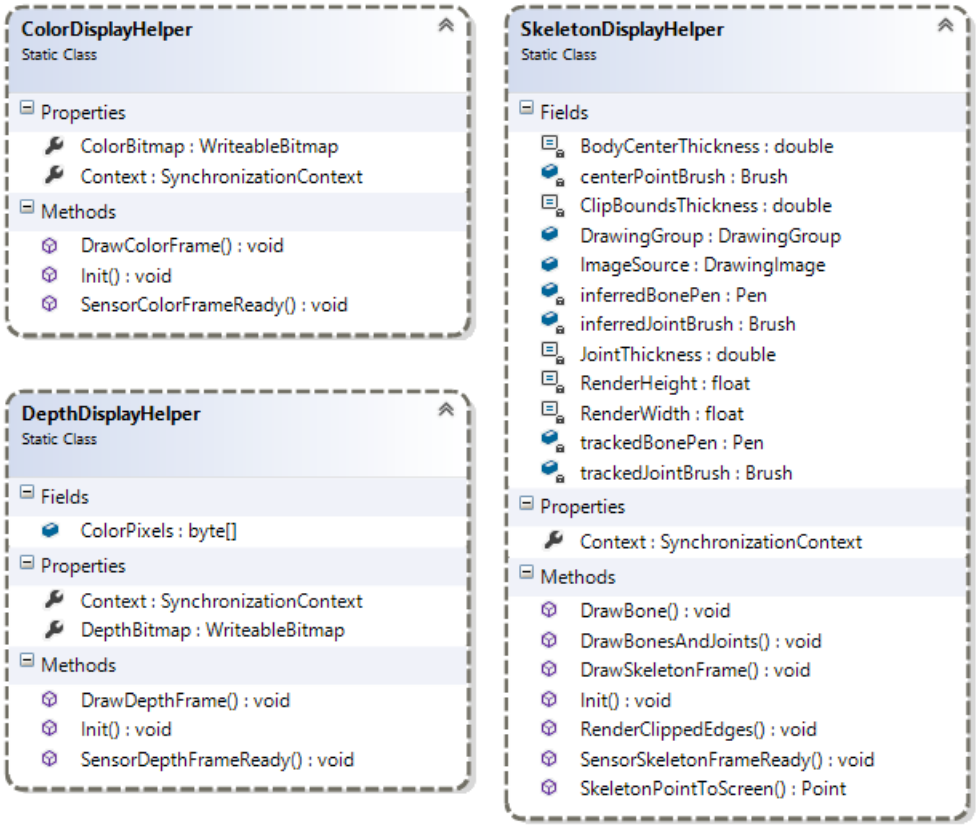


Rys 7.6: Zastosowanie klas DisplayHelper w aplikacji TKinect Studio

Komponenty odpowiadające za wizualizację ramek to klasy **DisplayHelper** gdzie:

- **ColorDisplayHelper** – wizualizacja ramek koloru.
- **DepthDisplayHelper** – wizualizacja ramek głębi.
- **SkeletonDisplayHelper** – wizualizacja ramek szkieletu.

Diagram klas na rysunku 7.7 przedstawia ich budowę.



Rys 7.7: Color/Depth/Skeleton DisplayHelper

Aby wyświetlić ramki należy zainicjalizować odpowiednie klasy statyczne **DisplayHelper** oraz wpiąć w eventy gdzie napływać będą wizualizowane ramki.

```
//COLOR
ColorDisplayHelper.Init();
TKinect.ColorFrameReady += ColorDisplayHelper.SensorColorFrameReady;

//DEPTH
DepthDisplayHelper.Init();
TKinect.DepthFrameReady += DepthDisplayHelper.SensorDepthFrameReady;

//SKELETON
SkeletonDisplayHelper.Init();
TKinect.SkeletonFrameReady += SkeletonDisplayHelper.SensorSkeletonFrameReady;
```

Następnie wystarczy wyświetlić źródła obraz.

```
ColorImage.Source = ColorDisplayHelper.ColorBitmap;

DepthImage.Source = DepthDisplayHelper.DepthBitmap;

SkeletonImage.Source = SkeletonDisplayHelper.ImageSource;
```

8. Ewaluacja stworzonego rozwiązania TKinect

W rozdziale tym na przykładzie prostej aplikacji opartej o kontroler pokazane zostaną niezbędne kroki które należy wykonać w celu użycia biblioteki TKinect. Następnie zaprezentuje jak przy pomocy biblioteki TKinect w pełni pokryć testami partie kodu odpowiedzialne za analizę danych przychodzących z Kinecta w formie testów jednostkowych. Na koniec pokazany zostanie zewnętrzny sposób testowania w formie testów akceptacyjnych.

8.1 Program Hello Kinect

W celu demonstracji integracji z stworzoną biblioteką oraz prezentacji przykładowego testu jednostkowego oraz akceptacyjnego wykorzystana zostanie prosta aplikacja HelloKinectWPF. Jest to okienkowy program stworzony w frameworku WPF (Windows Prezentation Foundation) który wyświetla napis „Hello Kinect" podczas wykrycia gestu „right hello" (prawa ręka u góry).

Rys 8.1: Hello Kinect WPF

Zrzuty ekranu na rysunku 8.1 przedstawiają z lewej aplikacje która jeszcze nie wykryła gestu natomiast z prawej aplikację po wykryciu gestu. Logika aplikacji została zawarta w osobnej bibliotece a dokładniej w jednej klasie **HelloKinect**. Klasa rzuca zdarzenie **HelloDetected** które należy odpowiednio przechwycić gdy wykryty zostaje gest „right hello". A wiec w programie HelloKinectWPF podczas tego zdarzenie zmieniany zostaje tekst wyświetlanego komponentu.

```
private void Window_Initialized(object sender, EventArgs e)
{
    var hello = new HelloKinect.HelloKinect();
    hello.HelloDetected += hello_HelloDetected;

    //... Run Hello Kinect
}

private void hello_HelloDetected(object sender, int helloCount)
{
    this.Dispatcher.Invoke(() =>
```

```
    {
        Label.Content = "Hello Kinect";
    });
}
```

W celu wykrycia gestu użyta została biblioteka Kinect.Toolbox a dokładnie jedna z jej klas **PostureAnalyzer** który jest prostym analizatorem postawy gracza rzucając zdarzenie podczas wykrycia jednego z obsługiwanych gestów. Podczas inicjalizacji (w metodzie konstruktora) obiektu HelloKinect tworzony oraz inicjalizowany jest **PostureAnalyer**.

```
public HelloKinect()
{
    //Initialize KinectToolBox frame collector
    SynchronizationContext.SetSynchronizationContext(new SynchronizationContext());
    var framesCollector = new FramesCollector(KinectSensor, 30);

    //Create and use one of analyzers
    PostureAnalyzer = new PostureAnalyzer(framesCollector, 25);
    Configuration.postureNavigation = true;
    PostureAnalyzer.PostureDetected += PostureAnalyzerOnPostureDetected;
}
```

Gdzie metoda **PostureAnalyzerOnPostureDetected** jest metoda w której reagujemy na wykryty gest i podnosimy własne zdarzenie.

```
private void PostureAnalyzerOnPostureDetected(object sender, PostureEventArgs args)
{
    if(args.Posture == PosturesEnum.RightHello)
    {
        HelloCounter++;
        HelloDetected(this, HelloCounter);
    }
}
```

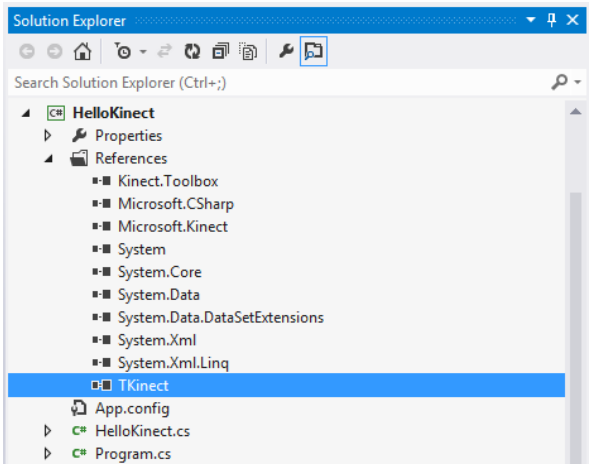
Następne istnieją 3 metody uruchomienia klasy.

1. **RunRealKinect** – metoda wyszukuje realny kontroler i pobiera z niego ramki.
2. **RunKinectReplay** – metoda wczytuje ramki z pliku.
3. **RunKinectClient** – metoda rejestruje się jako klient i nasłuchuje ramek od innego hosta (którym będzie TKinect Studio).

Pełny kod testowej aplikacji umieszczony został w załączniku.

8.2 Integracja z projektem

Pierwszym krokiem jaki należy wykonać aby użyć biblioteki w swoim projekcie jest dodanie referencji do biblioteki TKinect co ilustruje rysunek 8.2.



Rys 8.2: Adding TKinect

Zaczynamy od inicjalizacji obiektu TKinect oraz podłączeniu go do sensora przy pomocy następującego kodu.

```
//Initialize TKinect
TKinect = new TKinect.TKinect();

KinectSensor.SkeletonFrameReady += TKinect.SensorSkeletonFrameHandler;
```

Następnie należy zmienić typ danych na dane używane przez bibliotekę TKinect. Potrzeba tej operacji została opisana w poprzednim rozdziale. Aby tego dokonać należy zmodyfikować lekko bibliotekę analizującą (KinectToolbox) aby używała danych z przedrostkiem „T" (SkeletonFrame -> TSkeletonFrame). A wiec po podmianie typów na klasy z biblioteki TKinect inicjalizajca klasy FrameClollector zmieni się następująco.

```
//FrameCollector using KinectSensor
var framesCollector = new FramesCollector(KinectSensor, 30);

//FrameCollector using TKinect
var framesCollector = new FramesCollector(TKinect, 30);
```

Po wykonaniu powyższych czynności aplikacja zachowując swoją funkcjonalność działa z użyciem biblioteki TKinect i jest gotowa do testów które zaprezentowane w poniższych podrozdziałach.

8.3 Test jednostkowy

Przykładowa aplikacja opisana w na początku tego rozdziału została zintegrowana z biblioteką TKinect. Dzieki temu możemy przystąpić do pokrycia testami warstwy logiki odpowiadającej za analizę ramek. W tym przypadku za analizę odpowiada komponent biblioteki KinectToolBox a konkretnie klasa **PostureAnalyzer**. Jej działanie opiera się na wyrzuceniu eventu **PostureDetected** gdy rozpoznany zostanie jeden z obsługiwanych gestów. W celu napisania testów jednostkowych sprawdzających tą logikę użyta została popularna

biblioteka Nunit. Zaczynamy od przygotowania klasy analizatora do testu w metodzie **[SetUp]**. Pod zdarzenie wykrycia gestu podpinamy testową metodę która zaznacza flagę **_leftHello**. Następnie poprzez wczytanie do biblioteki TKinect nagrania odczekujemy na jego koniec i sprawdzamy stan flagi.

```
[TestFixture]
public class HelloWorldAnalyzerTests
{
    private TKinect.TKinect Kinect { get; set;}
    private PostureAnalyzer PostureAnalyzer { get; set; }

    [SetUp]
    public void Setup()
    {
        Configuration.postureNavigation = true;
        SynchronizationContext.SetSynchronizationContext(new SynchronizationContext());

        Kinect = new TKinect.TKinect();
        var framesCollector = new FramesCollector(Kinect, 30);

        PostureAnalyzer = new PostureAnalyzer(framesCollector, 25);
        PostureAnalyzer.PostureDetected += PostureAnalyzerOnPostureDetected;
    }

    private bool _leftHello = false;
    private void PostureAnalyzerOnPostureDetected(object sender, PostureEventArgs args)
    {
        if (args.Posture == PosturesEnum.LeftHello)
           _leftHello = true;
    }

    [Test]
    public void LeftHelloDetected()
    {
       using (var fileStream = new FileStream(@"RECORDING.xed", FileMode.Open))
       {
           Kinect.RecordStart(fileStream);
       }

       Thread.Sleep(3000);

       Assert.True(_leftHello);
   }
}
```

W ten oto prosty sposób otrzymaliśmy test jednostkowy sprawdzający naszą klasę analizatora pod kątem wczytanego nagrania. Mając dużą bazę nagrań test ten mógłby zostać wywołany na większej ilości przykładów sprawdzając poprawność działania dla różnych sylwetek testowych.

8.4 Test akceptacyjny

Jedną z głównych zalet testów akceptacyjnych jest fakt iż działają one na całości aplikacji a nie tylko jej części. A wiec w przeciwieństwie do testu jednostkowego przedstawionego w poprzednim rozdziale który skupił się na klasie **PostureAnalyzer** skupimy się na programie HelloKinectWPF**.** W celu napisania przykładowego testu akceptacyjnego posłużymy się biblioteką TestStack White która umożliwia uruchomienie oraz nadzór wraz z możliwością

sprawdzenia graficznych komponentów testowanego programu. Nasz test rozpoczyna się od stworzenia hosta TKinect dzięki któremu wczytamy nagrania z zewnątrz do działającej aplikacji.

```
var tkinect = new TKinect.TKinect();
tkinect.SkeletonFrameHost.Start(4503);
```

Korzystając z funkcji TestStack uruchomiona zostanie testowana aplikacja w trybie klienta. Oznacza to iż aplikacja zostanie podłączona do stworzonego hosta i nasłuchiwać będzie ramek które mu wczytamy.

```
Application application = Application.Launch(@"HelloKinectWPF.exe");
Window window = application.GetWindow("Hello Kinect", InitializeOption.NoCache);
window.WaitWhileBusy();
```

Aby dopełnić testu pozostaje wczytać testowe nagranie. Na koniec wyszukując stan pola tekstowego w uruchomionym okienku sprawdzamy czy jego wartość zmieniła się na „Hello Kinect”

```
var fileStream = new FileStream(RecordingPath, FileMode.Open);
tkinect.ReplayStart(fileStream);
Thread.Sleep(3000);

var label = window.Get< Label>(SearchCriteria.ByAutomationId("Label"));

Assert.That(label.Text == "Hello Kinect");
application.Close();
fileStream.Close();
```

9. Zakończenie

„Testowanie oprogramowania może zostać użyte do wskazania błędów ale nigdy nie do pokazania braku ich istnienia” - Edsger Dijkstra

Celem pracy było poddanie analizie narzędzi wspomagających proces testowania oprogramowania opartego o kontroler ruchu Kinect. Wyjaśnione zostały popularne metody testowania programów a następnie podsumowane zostały problemy występujące podczas tworzenia specyficznych aplikacji które używają sensora. Zbadane zostały istniejące rozwiązania pod kątem występujących problemów a na koniec stworzony został zestaw narzędzi rozwiązujący wymienione problemy w którego skład wchodzi biblioteka TKinect oraz aplikacja TKinectStudio. Na podstawie stworzonych narzędzi przygotowany został także program testowy na którym zademonstrowana została integracja z biblioteką TKinect oraz przeprowadzanie testów jednostkowych jak i akceptacyjnych. Cele pracy zostały zatem w zrealizowane.

9.1 Przeprowadzone prace

Pierwszym zadaniem które zostało zrealizowane w ramach tej pracy była analiza pracy z sensorem Kinect. Zbadane została budowa oraz sposób działania kontrolera.

W drugim kroku stworzona została biblioteka TKinect która umożliwiła testowanie aplikacji używających kontrolera.

Podsumowanie wyniku pracy. Co dokładnie zostało zrobione.

9.2 Uwagi oraz ocena przyjętego sposobu rozwiązania

//TODO

+ otwarte api

+

- podmiana typów

9.3 Wizja dalszego rozwoju narzędzi

Stworzony zestaw narzędzi TKinect oraz TKinect Studio możne zostać dalej rozwinięty o dodatkowe funkcjonalności takie jak:

- **Edytor ramek szkieletu** – Posiadając nagranie szkieletu studio rozwinięte mogłoby zostać o edytor w którym użytkownik mógłby edytować bądź tworzyć

własne ramki szkieletu bez konieczności najpierw odtworzenia gestu przed kontrolerem. Funkcjonalność ta użyteczna byłaby w przypadku tworzenia aplikacji których gesty mogą być zbyt trudne to samodzielnego wykonania. Kolejnym zastosowaniem mogłoby być generowanie „szumu" w swoich nagraniach modyfikując przykładowo długości wszystkich kończyn i dzięki temu zwiększając bazę testową.

- **Funkcjonalność krokowego odtwarzania gestów** – Odtwarzanie nagrań nie posiada możliwości trybu krokowego który użyteczny byłby przy debugowaniu oprogramowania.
- **Webowa baza/market nagrań** – Wymienianie się nagraniami w formie plików jest uciążliwe i niewygodne. Nagrania z których korzysta studio mógłby zostać umieszczone na dedykowanym serwerze. Korzystając z niego użytkownicy w łatwy sposób mogliby udostępniać i wymieniać się nagraniami.
- **Wersja studia na urządzenia mobilne** – Inicjalizacja startu i stopu nagrywania poprzez przycisk wymaga jego naciśnięcia co w przypadku nagrywania gestów w pojedynkę wymusza posiadanie bezprzewodowej klawiatury bądź myszy czy innego wskaźnika lub zanieczyszczenie gestu podchodzeniem do komputera. Wersja mobilna studia mogła by być wygodniejszym sposobem kontroli nad nagrywaniem.
- **Integracja z najnowszym kontrolerem** – W sprzedaży jest już nowsza wersja kontrolera Kinect która ukazała się wraz z premierą konsoli Xbox One. Wraz z nią wypuszczone zostało nowe API które w dużej mierze przypomina poprzednie. Należałoby sprawdzić zmiany oraz zintegrować bibliotekę i studio pod nowy kontroler.

Bibliografia

1. **David Catuhe.** Programming with the Kinect for Windows - Software Development Kit. 2012.

2. **Kanglin Li, Mengqi Wu**. Effective Gui Test Automation. 2004.

3. **A.M. Memon, M.E. Pollack, M.L. Soffa**. Using a Goaldriven Approach to Generate Test Cases for GUIs. 1999.

4. **O'Reilly**. The Art of Application Performance Testing. 2009.

5. **K. Beck, C. Andres**. Extreme Programming Explained: Embrace Change. 2004.

6. **M. Fowler**. Cannot Measure Productivity. 2007.

7. **K. Beck**. Test Driven Development: By Example. 2002.

8. **Telerik**. Telerik UI for WPF Documentation.

[http://www.telerik.com/help/wpf/introduction.html]

9. **MSDN**. Working with Kinect Studio.

[http://msdn.microsoft.com/en-us/magazine/jj650892.aspx]

10. **Dgoins Insperience**. Exploring the Kinect Studio v2

[http://dgoins.wordpress.com/2014/03/30/exploring-the-kinect-studio-v2/]

11. **Mahapps.metro**. [http://mahapps.com]

12. **John D`Addamio**. Testing wpf application with the white ui test framework.

[http://blogs.msdn.com/b/john_daddamio/archive/2008/04/04/testing-wpf-applications-with-the-white-ui-test-framework.aspx]

13. **MSDN**. UI Automation Test Library.

[http://msdn.microsoft.com/en-us/library/windows/desktop/jj160543(v=vs.85).aspx]

14. **Dream-in-code**. C# TestStack.White for Beginners.

[http://www.dreamincode.net/forums/topic/322108-c%23-teststackwhite-for-beginners/]

TODO OTHER:

http://www.testingstandards.co.uk/bs_7925-1_online.htm

https://prezi.com/t3bsdiwkgqu5/budowa-zasada-dziaania-oraz-zastosowanie-sensora-microsoft/

http://users.dickinson.edu/~jmac/selected-talks/kinect.pdf

http://hoga.pl/technologie/ciekawe-zastosowanie-kinecta-od-ms/

https://msdn.microsoft.com/en-us/library/hh438998.aspx

http://blog.wilgucki.pl/2010/11/kolejne-zastosowania-zhakowanego.html

http://www.kinectingforwindows.com/2012/09/07/what-is-the-difference-between-kinect-for-windows-kinect-for-xbox360/

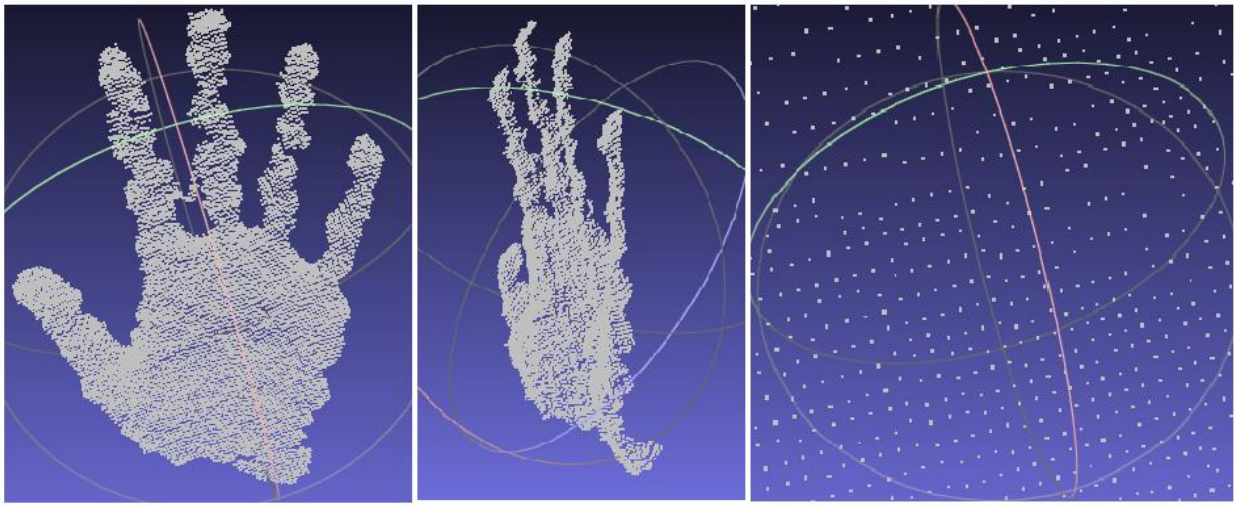
http://www.martinfowler.com/articles/continuousIntegration.html

http://www.laserfocusworld.com/articles/2011/01/lasers-bring-gesture-recognition-to-the-home.html

https://msdn.microsoft.com/en-us/library/hh973072.aspx

Załączniki

1) Mapa głębokości sensora Kinect przedstawiająca dłoń. Zrzut ekranu z programu do wizualizacji grafiki MeshLab.



2) Kod źródłowy testowego programu HelloKinect który został użyty jako przykład integracji bibloteki.

Kod HelloKinect

```
public class HelloKinect
{
    public delegate void HelloHandler(object sender, int helloCount);
    public event HelloHandler HelloDetected;
    public int HelloCounter { get; private set; }

    public KinectSensor Kinect { get; set; }
    public TKinect.TKinect TKinect { get; set; }
    public PostureAnalyzer PostureAnalyzer { get; set; }

    public HelloKinect()
    {
        //Initialize TKinect
        TKinect = new TKinect.TKinect();

        //Initialize KinectToolBox frame collector
        SynchronizationContext.SetSynchronizationContext(new SynchronizationContext());
        var framesCollector = new FramesCollector(TKinect, 30);

        //Create and use one of analyzers
        PostureAnalyzer = new PostureAnalyzer(framesCollector, 25);
        Configuration.postureNavigation = true;
        PostureAnalyzer.PostureDetected += PostureAnalyzerOnPostureDetected;
    }

    public void RunRealKinect()
    {
        int index = 0;
        while (Kinect == null && index < KinectSensor.KinectSensors.Count)
```

```
        {
            try
            {
                Kinect = KinectSensor.KinectSensors[index];
                Kinect.Start();
            }
            catch (Exception e)
            {
                Kinect = null;
            }
            index++;
        }
        if (Kinect != null)
        {
            Kinect.SkeletonStream.Enable(new TransformSmoothParameters
            {
                Smoothing = 0.5f,
                Correction = 0.5f,
                Prediction = 0.5f,
                JitterRadius = 0.05f,
                MaxDeviationRadius = 0.04f
            });
            Kinect.SkeletonFrameReady += TKinect.SensorSkeletonFrameHandler;
            return;
        }
        Console.WriteLine("NO REAL KINECT DETECTED");
    }

    private const string RecordingPath = @"C:\Users\Styrna\Desktop\PassedData\hello.xed";
    public void RunKinectReplay()
    {
        var fileStream = new FileStream(RecordingPath, FileMode.Open);
        TKinect.ReplayStart(fileStream);
    }

    public void RunKinectClient()
    {
        TKinect.SkeletonFrameClient.Connect("localhost", 4503);
    }

    private void PostureAnalyzerOnPostureDetected(object sender, PostureEventArgs args)
    {
        if(args.Posture == PosturesEnum.RightHello)
        {
            HelloCounter++;
            HelloDetected(this, HelloCounter);
        }
    }
}
```

Kod HelloKinectWPF

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
```

```
        InitializeComponent();
    }

    private void Window_Initialized(object sender, EventArgs e)
    {
        var hello = new HelloKinect.HelloKinect();
        hello.HelloDetected += hello_HelloDetected;


        hello.RunRealKinect();
        //hello.RunKinectReplay();
        //hello.RunKinectClient();
    }

    private void hello_HelloDetected(object sender, int helloCount)
    {
        this.Dispatcher.Invoke(() =>
        {
            Label.Content = "Hello Kinect";
        });
    }
}
```