

A G H

AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

**WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI,
INFORMATYKI I INŻYNIERII BIOMEDYCZNEJ
KATEDRA INFORMATYKI STOSOWANEJ**

Praca dyplomowa magisterska

*Prototyp systemu wizualizacji rozszerzonej rzeczywistości,
wzbogaconej o wirtualne przedmioty w oparciu o rozpoznanie
znaczników*

*Augmented reality visualisation system prototype, enhancing a view
with virtual items using marker recognition*

Autor: inż. Przemysław Błasiak

Kierunek studiów: Informatyka

Opiekun pracy: dr inż. Grzegorz Rogus

Kraków, czerwiec 2016

Uprzedzony o odpowiedzialności karnej na podstawie art. 115 ust. 1 i 2 ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t.j. Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.): „ Kto przywłaszcza sobie autorstwo albo wprowadza w błąd co do autorstwa całości lub części cudzego utworu albo artystycznego wykonania, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 3. Tej samej karze podlega, kto rozpowszechnia bez podania nazwiska lub pseudonimu twórcy cudzy utwór w wersji oryginalnej albo w postaci opracowania, artystyczne wykonanie albo publicznie zniekształca taki utwór, artystyczne wykonanie, fonogram, videogram lub nadanie.”, a także uprzedzony o odpowiedzialności dyscyplinarnej na podstawie art. 211 ust. 1 ustawy z dnia 27 lipca 2005 r. Prawo o szkolnictwie wyższym (t.j. Dz. U. z 2012 r. poz. 572, z późn. zm.) „Za naruszenie przepisów obowiązujących w uczelni oraz za czyny uchybiające godność studenta student ponosi odpowiedzialność dyscyplinarną przed komisją dyscyplinarną albo przed sądem koleżeńskim samorządu studenckiego, zwanym dalej „sądem koleżeńskim”, oświadczam, że niniejszą pracę dyplomową wykonałem osobiście i samodzielnie i że nie korzystałem ze źródeł innych niż wymienione w pracy.

.....
(Podpis dyplomanta)

Spis treści

1. Wstęp	5
1.1. Cel pracy	5
1.2. Struktura dokumentu.....	5
2. Aktualny stan wiedzy	7
2.1. System AR	7
2.2. Wybrane zagadnienia platformy iOS.....	12
2.3. Biblioteka OpenCV.....	22
2.4. Wykrywanie znaczników.....	24
3. Przedstawienie problemu.....	29
3.1. Główne założenia.....	29
3.2. Wybrany sposób realizacji.....	29
4. Rozwiązanie.....	32
4.1. Przygotowanie szkieletu aplikacji.....	32
4.2. Przechwytywanie obrazu z kamery	35
4.3. Wykrywanie znaczników.....	38
4.4. Wstawianie wirtualnych przedmiotów	44
4.5. Wyświetlanie obrazu.....	47
4.6. Testowanie rozwiązania.....	50
5. Aplikacja do wizualizacji wnętrz	55
6. Podsumowanie.....	59
7. Bibliografia	60

1. Wstęp

Żyjemy w świecie trzech wymiarów przestrzennych wypełnionym obiektemi, które widzimy, słyszymy, dotykamy i czujemy. Przez tysiące lat ewolucji wykształciliśmy różne sposoby postrzegania otaczającej nas rzeczywistości (1), takie jak widzenie stereoskopowe umożliwiające szacowanie położenia obiektów w przestrzeni trójwymiarowej, do których zdążyliśmy się przyzwyczaić do tego stopnia, że są dla nas czymś naturalnym i z góry zakładamy ich działanie.

Wraz z narodzinami bita i piksela zaczęliśmy tworzyć własny, cyfrowy, nierealny świat. Elementy tej wirtualnej rzeczywistości (VR), oparte na abstrakcyjnym pojęciu informacji, zaczęły dotykać coraz to większej liczby obszarów naszego życia, przez co przepaść między tym co realne a tym co nierealne uwidoczyliła się, rodząc tym samym potrzebę połączenia jednego świata z drugim - nowego podejścia, którym jest rozszerzona rzeczywistość (AR).

1.1. Cel pracy

Celem pracy jest zaprojektowanie i stworzenie prototypu systemu wizualizacji rozszerzonej rzeczywistości - świata realnego wzbogaconego o wirtualne przedmioty, w oparciu o platformę mobilną iOS oraz okulary wirtualnej rzeczywistości. System ma nakładać generowaną w czasie rzeczywistym grafikę 3D na obraz pozyskiwany z kamery telefonu. Przy czym, obiekty świata wirtualnego mają być wstawiane w miejsce znaczników wykrytych na obrazie, po uprzednim oszacowaniu ich pozycji w przestrzeni trójwymiarowej. W celu zapewnienia wysokiego stopnia immersyjności, prototyp ma również umożliwiać zastosowanie okularów wirtualnej rzeczywistości do podglądu generowanego obrazu oraz możliwość interakcji z wirtualnymi przedmiotami.

W niniejszej pracy zostanie opisany proces tworzenia systemu wizualizacji rozszerzonej rzeczywistości w formie aplikacji mobilnej na platformę iOS, począwszy od zagadnienia pozyskiwania klatki obrazu kamery urządzenia, przez problem wykrywania znaczników, po odpowiednie wyświetlanie generowanego obrazu, do którego podglądu można wykorzystać okulary VR.

Ponadto, w ramach pracy powstanie projekt pokazujący praktyczne zastosowanie tworzonyego systemu. Będzie to aplikacja służąca do wizualizacji w czasie rzeczywistym wnętrz pomieszczeń, gdzie w miejsce znaczników będą wstawiane modele 3D mebli, umożliwiającego swobodne poruszanie się i eksplorację rozszerzonej rzeczywistości.

1.2. Struktura dokumentu

Praca składa się z siedmiu rozdziałów. Pierwszy rozdział przybliża pojęcie rozszerzonej rzeczywistości oraz cel i zakres pracy. W rozdziale drugim przedstawiony jest aktualny stan wiedzy na temat rozwiązań takich jak systemy rozszerzonej rzeczywistości, wybrane zagadnienia platformy iOS oraz technik rozpoznawania znaczników i szacowania ich położenia w przestrzeni na podstawie dwuwymiarowego obrazu. Kolejny rozdział zawiera definicję

problemu poruszanego w pracy, którego rozwiążanie jest z kolei opisane w rozdziale czwartym. W tej części dokumentu poruszane są między innymi zagadnienia przechwytywania klatek obrazu, odnajdywania i odczytywania znaczników, wstawiania wirtualnych przedmiotów w ich miejsce, interakcji z rozszerzoną rzeczywistością oraz wyświetlanie obrazu w trybach mono i stereo. Kolejny, piąty rozdział przedstawia przygotowanie przykładowej aplikacji mobilnej, wykorzystującej tworzony system wizualizacji. Natomiast dwa ostatnie rozdziały mieszczą podsumowanie całej pracy oraz bibliografię.

2. Aktualny stan wiedzy

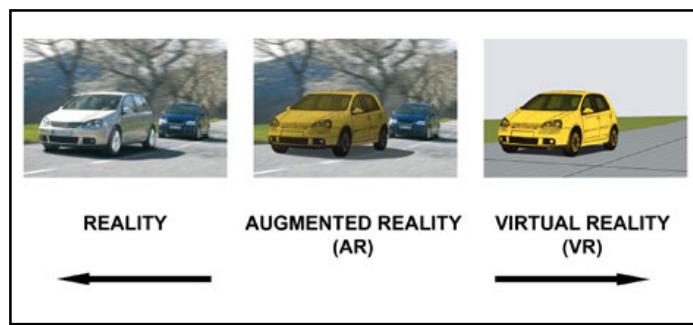
2.1. System AR

System wizualizacji rozszerzonej rzeczywistości (System AR) ma za zadanie przedstawiać w czasie rzeczywistym obraz świata powstałego w wyniku nałożenia wirtualnych obiektów na obraz świata rzeczywistego, zaprzegając do tego celu m.in. metody i algorytmy wizji komputerowej oraz rozpoznawania obiektów. Miarą jakości systemu AR jest stopień immersyjności, której głównym wyznacznikiem jest realizm z jakim obiekty wirtualne łączone są ze światem rzeczywistym.

Do realizacji systemu AR, obok oprogramowania, potrzebny jest sprzęt: procesor, wyświetlacz, czujniki i urządzenia wejściowe. Z uwagi na to, że smartfony i tablety zazwyczaj posiadają wszystkie wyżej wymienione komponenty, wraz z całym zestawem czujników jak akcelerometr, GPS, czy magnetometr, doskonale nadają się one do budowy prostego systemu AR. Jednakże, na rynku istnieją i są aktywnie rozwijane kompleksowe rozwiązania, oparte na dedykowanych urządzeniach, które zapewniają wysoką immersyjność, ale charakteryzuje je jednocześnie stosunkowo wysoka cena.

2.1.1. Systemy AR a VR

System VR ma za zadanie pozwolić użytkownikowi na zanurzenie się w wirtualnym świecie, podczas gdy zadaniem rozszerzonej rzeczywistości jest połączenie tego świata ze światem rzeczywistym. VR jest więc poniekąd częścią składową AR. Jednym z przykładów, który dobrze ilustruje tę relację, są okulary AR, które mogą być zrealizowane przy użyciu okularów VR odpowiedzialnych za wyświetlanie wirtualnych przedmiotów, w połączeniu z kamerami służącymi do przechwytywania obrazu świata rzeczywistego.



Rys. 2–1 Różnica między obrazami wirtualnej i rozszerzonej rzeczywistości.



Rys. 2–2 Oculus Rift jako przykład okularów wirtualnej rzeczywistości. Źródło: (2)

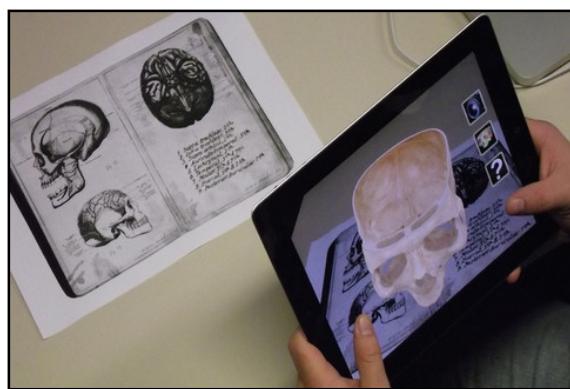
Ze względu różnicę między powyższymi systemami, różnią się pod względem zastosowania. O ile Systemy VR są głównie wykorzystywane do gier wideo i rozrywki, o tyle systemy AR lepiej nadają się do profesjonalnych zastosowań.

2.1.2. Istniejące systemy AR

Na rynku istnieje wiele sposobów realizacji systemów AR. Różnią się one między sobą głównie technikami rejestracji, przetwarzania i wyświetlania obrazu, stosowanymi metodami wizji komputerowej, oraz rodzajem wykorzystywanych czujników i innych podzespołów. Jednym z głównych kryterium klasyfikacji systemów AR jest stosowana technologia wizualizacji, dlatego poniżej zostaną przedstawione najbardziej popularne rozwiązania, ze względu na to kryterium.

Klasyczny wyświetlacz LCD

Systemy AR, w przypadku których wykorzystywany jest wyświetlacz LCD urządzenia przenośnego, laptopa, lub w przypadku komputera stacjonarnego - monitora. To rozwiązanie zapewnia stosunkowo niski stopień realizmu AR, dlatego wykorzystywane jest głównie do testowania algorytmów wizji komputerowej oraz nieprofesjonalnych zastosowań.

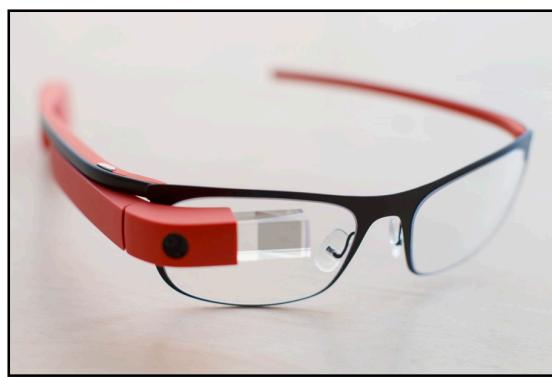


Rys. 2–3 Przykład systemu AR realizowanego za pomocą urządzenia przenośnego. Źródło: (3)

Urządzenie OHMD

OHMD (ang. optical head-mounted display) to urządzenie wyświetlające nakładane na głowę - w formie hełmu, które dodatkowo posiada mały ekran przed jednym lub obojgiem oczu. Ekranom tym towarzyszy odpowiedni układ soczewek, a same osadzone są na półprzezroczystych lustrach, które odbijają rzucany obraz generowany komputerowo - CGI (ang. computer-generated imagery) umożliwiając jednocześnie obserwowanie świata rzeczywistego.

Urządzenia OHMD posiadają wszelkie potrzebne podzespoły do realizacji systemu AR, a nowoczesne rozwiązania tego typu są wyposażone w sensory ruchu dla sześciu stopni swobody: trzy translacyjne, współrzędne środka masy i trzy rotacyjne (obrotowe) – współrzędne kątowe, które określają obrót obiektu w przestrzeni trójwymiarowej. OHMD tworzone były z myślą o stosowaniu w życiu codziennym, ale gwałtownie rośnie popularność zastosowań profesjonalnych tego typu urządzeń, na przykład do pomocy w odnajdywaniu i identyfikacji odpowiedniego towaru w magazynie.



Rys. 2–4 Google Glass jako przykład urządzenia OHMD. Źródło: (4)

Urządzenie HUD

HUD (ang. Head-Up Display) to wyświetlacz przezierny, który prezentuje informacje na powierzchni szklanej nie zasłaniając widoku. Urządzenia tego typu można podzielić na dwa rodzaje:

- Stałe - obraz wyświetlany jest na powierzchni szklanej zamocowanej na stela przed obserwatorem,
- Nahełmowe - powierzchnia szklana porusza się wraz z głową obserwatora.

Wyświetlacze przezirne są używane do prezentacji danych na szybach samolotów, samochodów, motocykli, kasków, itp.



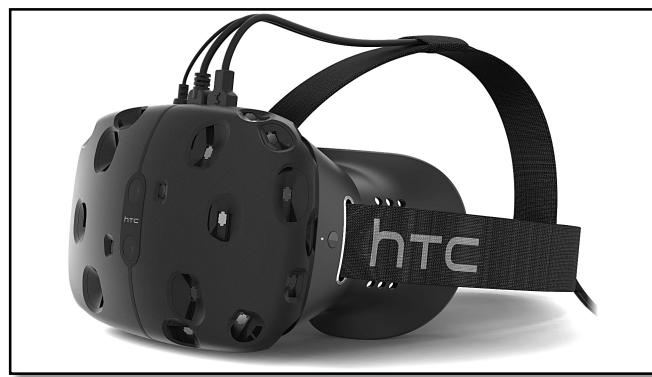
Rys. 2–5 Wyświetlacz HUD w myśliwcu F/A-18 Hornet. Źródło: (3)

Okulary

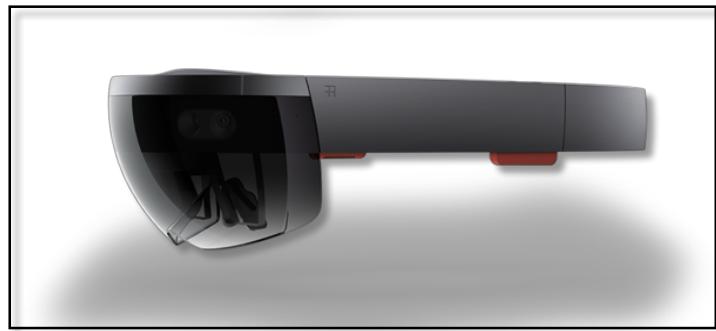
Systemy AR mogą być realizowane za pomocą urządzeń przypominających okulary. Wyróżnia się trzy rodzaje tego typu rozwiązań, ze względu na metodę przechwytywania i wyświetlania obrazu:

- rozwiązania wykorzystujące kamery do przechwytywania obrazu świata rzeczywistego i wyświetlanego go na okularach wraz z elementami AR,
- półprzezroczyste okulary wyświetlające obrazu,
- półprzezroczyste okulary odbijające rzucany obraz.

Rozwiązania tego typu znajdują obecnie zastosowanie najczęściej jako platforma do gier AR i w sektorze edukacji.



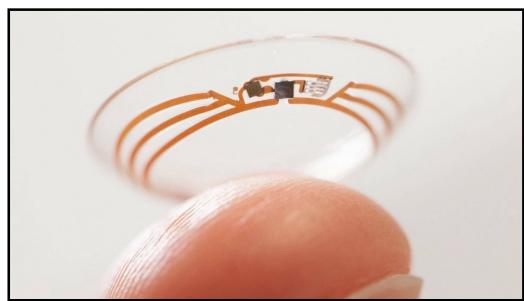
Rys. 2–6 HTC Vive jako przykład okularów AR wykorzystujących kamery. Źródło: (5)



Rys. 2–7 Microsoft HoloLens jako przykład okularów AR z półprzezroczystymi szkłami, na których wyświetlany jest obraz. Źródło: (3)

Szka kontaktowe

Jedną z aktywnie rozwijanych technologii jest rozwiązywanie, które ma pozwolić na stworzenie systemu AR w oparciu o szkła kontaktowe z wbudowaną elektroniką. Rozwiązaniem tym są zainteresowane zarówno małe jak i duże firmy technologiczne, takie jak: Samsung, Google, czy Sony.



Rys. 2–8 Szkło kontaktowe z wbudowanym układem elektronicznym. Źródło: (6)

Projektor VRD

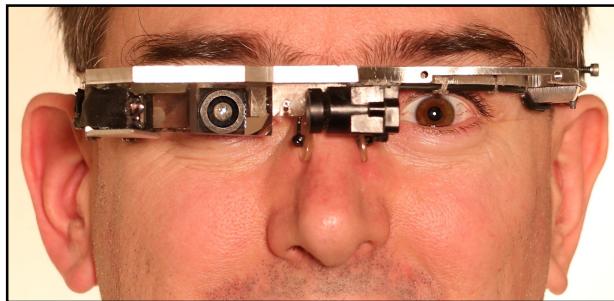
Wartą uwagi technologią jest VRD (ang. Virtual Retinal Display). Działanie urządzeń tego typu polega na rzucaniu obrazu bezpośrednio na siatkówkę oka obserwatora, który ma wrażenie, że patrzy na wiszący w powietrzu ekran. Za wynalazcę VRD uważany jest Kazuo Yoshinaka, ale znaczący wkład w rozwój tej technologii ma również amerykańska uczelnia University of Washington oraz należące do niej laboratorium Human Interface Technology Lab.



Rys. 2–9 Urządzenie Avegant Glyph wykorzystujące technologię VRD. Źródło: (7)

EyeTap

Kolejną aktywnie rozwijaną technologią jest EyeTap, z którą związany jest wynalazca Steve Mann. W tym przypadku urządzenie przechwytuje promienie światła, które normalnie dotarły by do oka i zastępuje je promieniami światła wygenerowanymi komputerowo. Technologia ta wykorzystuje technologię VRD do projekcji obrazu. Najnowsze z rozwiązań tego typu, nazywane są również Generation-4 Glass lub Laser EyeTap.



Rys. 2–10 Wynalazca Steve Mann z urządzeniem Laser EyeTap. Źródło: (3)

2.2. Wybrane zagadnienia platformy iOS

System operacyjny iOS został stworzony przez firmę Apple Inc. i jest przeznaczony dla urządzeń mobilnych, takich jak iPhone, iPad, oraz iPod Touch. Jego premiera miała miejsce w 2007 roku, kiedy to ówczesny prezes Apple - Steve Jobs zaprezentował pierwszego iPhone'a. System początkowo nazywał się OS X, po czym został przemianowany na iPhone OS i ostatecznie iOS. Aktualną wersją systemu jest iOS 10.

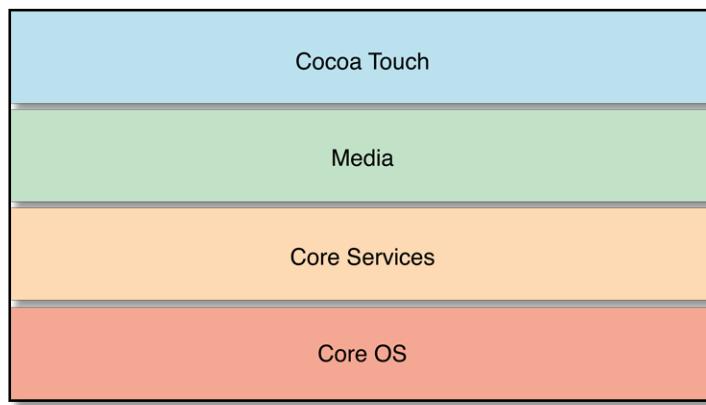
Środowiskiem programistycznym rekomendowanym przez Apple do tworzenia aplikacji natywnych jest Xcode IDE oraz zestaw narzędzi i systemowych frameworków iOS Software Development Kit (SDK). W chwili obecnej możliwe jest rozwijanie aplikacji w dwóch językach: Objective-C (8) oraz Swift (9), przy czym ten drugi jest językiem zalecanym i aktualnie silnie rozwijanym. Dodatkowo, języki te zapewniają mechanizmy współpracy między sobą oraz językami C i C++.

W poniższych rozdziałach zostaną opisane wybrane zagadnienia platformy, które są kluczowym elementem niniejszej pracy, takie jak korzystanie z poziomu języka Swift z bibliotek języka C++, dołączanie biblioteki OpenCV do projektu aplikacji mobilnej, wykrywanie znaczników przy jej pomocy, pozyskiwanie klatki obrazu z kamery urządzenia w czasie rzeczywistym, oraz opis wysokopoziomowego frameworka iOS do grafiki 3D o nazwie SceneKit.

2.2.1. Architektura systemu

System iOS charakteryzuje budowa warstwowa - można go podzielić na 4 warstwy abstrakcji (Rys. 2–11). W obrębie każdej z nich znajduje się grupa frameworków dostępnych dla programistów. Warstwy, które znajdują się niżej zawierają frameworki odpowiedzialne za niskopoziomowe, podstawowe usługi i technologie, podczas gdy warstwy polożone wyżej budują na nich dostarczając wysokopoziomowych rozwiązań. Opisane warstwy abstrakcji to:

- Cocoa Touch - najwyższa z warstw, zapewnia wiele wysokopoziomowych usług oraz zawiera główne frameworki służące do budowy aplikacji iOS.
- Media - warstwa multimedialna, gromadząca frameworki odpowiedzialne za obsługę obrazu i dźwięku.
- Core Services - grupa podstawowych frameworków służących do zarządzania pracą aplikacji.
- Core OS - najniższa spośród 4 warstw, wiąże oprogramowanie ze sprzętem i w jej skład wchodzi m.in. jądro Mach.

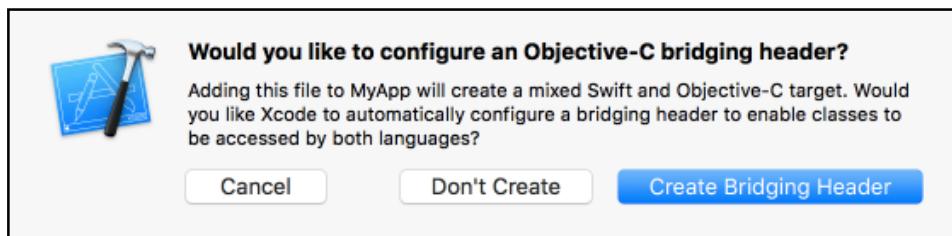


Rys. 2–11 Warstwy abstrakcji systemu iOS. Źródło: (10)

Generalnie zalecane jest korzystanie w pierwszej kolejności z wysokopoziomowych rozwiązań, zagłębiając się w niższe warstwy tylko jeśli jest to konieczne do osiągnięcia celu. W niniejszej pracy wykorzystywane będą głównie frameworki dwóch najwyższych warstw: Cocoa Touch oraz Media.

2.2.2. Korzystanie z bibliotek C++ w Swift

Z jednej strony, język Swift potrafi bezpośrednio współpracować z językiem C, oraz pośrednio z językiem Objective-C - przez tzw. Objective-C Bridging Header, czyli plik nagłówkowy, w którym określa listę plików źródłowych Objective-C, do których pliki Swift projektu mają mieć dostęp. Środowisko Xcode oferuje automatyczne wygenerowanie wspomnianego pliku nagłówkowego, gdy do projektu w Swift dodawany jest plik źródłowy Objective-C i vice versa.



Rys. 2–12 Okno dialogowe środowiska Xcode oferujące wygenerowanie pliku Objective-C Bridging Header.

Z drugiej strony, język Objective-C jest nadzbiorem języka C, więc naturalnie może bezpośrednio z nim współpracować. Natomiast w przypadku języka C++, może być z nim mieszany w pojedynczym pliku źródłowym jako Objective-C++, co zostanie dokładniej wyjaśnione w kolejnym dziale.

Podsumowując, język Swift może współpracować z Objective-C, który z kolei może korzystać z kodu języka C++ przez Objective-C++. Dlatego kod C++ opakowuje się za pomocą plików źródłowych Objective-C++, którym towarzyszą pliki nagłówkowe Objective-C, importowane przez Objective-C Bridging Header. W ten sposób powstaje most między kodem biblioteki C++ a kodem aplikacji iOS w języku Swift.

Objective-C++

Nazwa Objective-C++ nie jest nazwą języka programowania, a zaledwie umownym określeniem na mieszany kod języków Objective-C oraz C++ (11). Połączenie kodu obu języków w jednym pliku jest możliwe, ponieważ zarówno kompilator GCC jak i Clang potrafią kompilować pliki źródłowego obu z nich. Jednakże, z uwagi na różnice w modelu obiektowym obu języków istnieją pewne ograniczenia, np. klasa C++ nie może dziedziczyć po klasie Objective-C. Rozszerzeniem pliku, w którym można stosować Objective-C++ jest ".mm".

Import plików nagłówkowych

Plik nagłówkowy biblioteki OpenCV dodanej w wyżej opisany sposób można dołączać w plikach Objective-C++ za pomocą dyrektywy:

```
#include <opencv2/opencv.hpp>
```

Przy czym, możliwe jest również dołączanie pojedynczych plików nagłówkowych biblioteki.

2.2.3. Przechwytywanie obrazu z kamery w czasie rzeczywistym

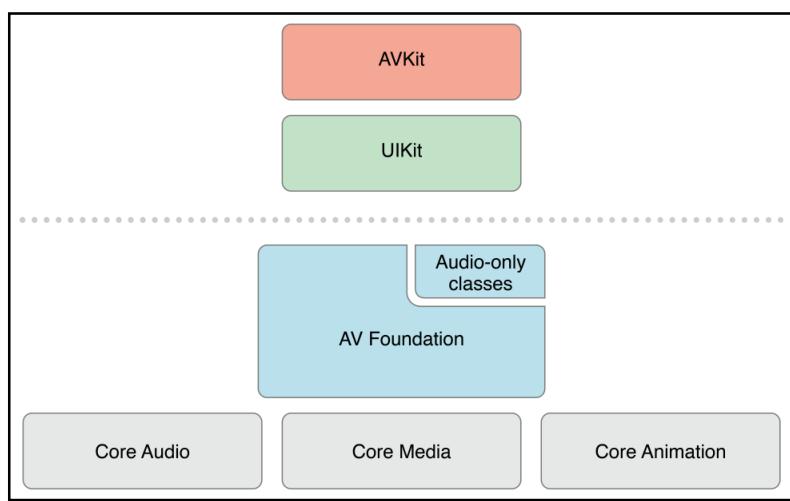
W celu przechwytywania obrazu w czasie rzeczywistym potrzebna jest wysoka wydajność, co można osiągnąć uzyskując bezpośredni dostęp do klatek dostarczanych na bieżąco przez kamerę urządzenia. Na platformie iOS jest to możliwe dzięki interfejsom dostarczonym przez framework AVFoundation, który zapewnia funkcjonalność niezbędną do bezpośredniego czytania buforów obrazu w pamięci.

Możliwe są również inne podejścia do problemu przechwytywania obrazu na iOS (12), jak wykorzystanie gotowego kontrolera widoku UIImagePickerController frameworka UIKit opakowującego cały proces dostępu do kamery. Jednakże, to podejście wymaga osobnej implementacji ekstrahowania pojedynczej ramki obrazu przez co jest mniej wydajne.

Framework AVFoundation

AVFoundation (13) to standardowy framework iOS, który zapewnia funkcjonalność oraz interfejsy potrzebne do pracy z danymi audio i wideo. Przykładowo, framework ten może posłużyć do analizy, stworzenia, edycji, bądź kodowania plików multimedialnych. Za pomocą AVFoundation można również uzyskać dostęp do strumieni wejściowych audio i wideo urządzenia w celu ich przetwarzania w czasie rzeczywistym i odtwarzania.

Framework ten mieści się w warstwie Media architektury systemu iOS (Rys. 2–11), opierając się na frameworkach tej samej warstwy: Core Audio, Core Media, oraz Core Animation. Pozostałe frameworki multimedialów: AVKit oraz UIKit, charakteryzuje dużo wyższy poziom abstrakcji.

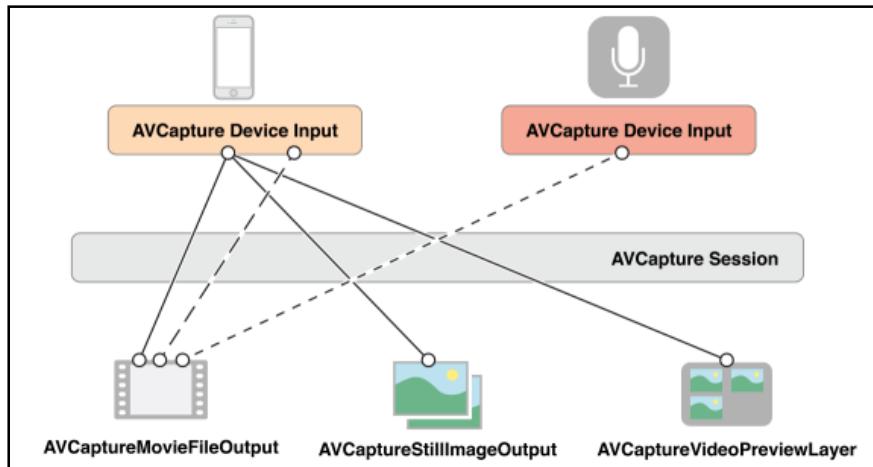


Rys. 2–13 Architektura frameworków multimedialnych systemu iOS. Źródło: (13)

Przechwytywanie obrazu z AVFoundation

Główną klasą wspomagającą proces przechwytywania obrazu wideo w frameworku AVFoundation jest AVCaptureSession, która reprezentuje sesję kamery i jest odpowiedzialna za koordynację przepływu danych audio oraz wideo między wejściami a wyjściami (Rys. 2–14). Obok potrzeby stworzenia obiektu koordynującego sesję, potrzebne są również:

- instancja klasy AVCaptureDevice reprezentującej urządzenie wejściowe, takie jak kamera czy mikrofon,
- instancja klasy dziedziczącej po AVCaptureInput w celu konfiguracji portów urządzenia wejściowego,
- instancja klasy dziedziczącej po AVCaptureOutput by zarządzać i określić format danych wyjściowych.



Rys. 2–14 Pojedyncza sesja koordynująca wiele wejść i wyjść. Źródło: (14)

Najpierw należy skonfigurować wejście sesji. W tym celu trzeba uzyskać interesującą nas instancję AVCaptureDevice reprezentującą urządzenie przechwytyjące - kamerę telefonu. Właściwy obiekt można znaleźć na liście dostępnych urządzeń, którą zwraca statyczna metoda devices klasy AVCaptureDevice.

Po tym, powinno się stworzyć instancję AVCaptureDeviceInput w oparciu o wcześniej przygotowany obiekt AVCaptureDevice. Kolejnym krokiem jest dostosowanie jakości i własności przechwytywanego obrazu, określając parametry takie jak: format zapisu piksela, ilość klatek na sekundę, rozdzielcość, pole widzenia, obecność stabilizacji obrazu, autofocus itp. Można to zrealizować na dwa sposoby:

- wybierając jeden z gotowych zestawów ustawień zwanych preset'ami (Rys. 2–15) sesji kamery,
- konfiguруjąc dokładny format wideo dla wejścia sesji, w oparciu o listę dostępnych formatów (Rys. 2–16), dla używanego urządzenia przechwytyującego - kamery.

Symbol	Resolution	Comments
AVCaptureSessionPresetHigh	High	Highest recording quality. This varies per device.
AVCaptureSessionPresetMedium	Medium	Suitable for Wi-Fi sharing. The actual values may change.
AVCaptureSessionPresetLow	Low	Suitable for 3G sharing. The actual values may change.
AVCaptureSessionPreset640x480	640x480	VGA.
AVCaptureSessionPreset1280x720	1280x720	720p HD.
AVCaptureSessionPresetPhoto	Photo	Full photo resolution. This is not supported for video output.

Rys. 2–15 Lista gotowych zestawów ustawień dla kamery. Źródło: (14)

Format	Resolution	FPS	HRSI	FOV	VIS	Max Zoom	Upscales	AF	ISO	SS	HDR
420v	1280x720	5 - 240	1280x720	54.626	YES	49.12	1.09	1	29.0 - 928	0.000003-0.200000	NO
420f	1280x720	5 - 240	1280x720	54.626	YES	49.12	1.09	1	29.0 - 928	0.000003-0.200000	NO
420v	1920x1080	2 - 30	3264x1836	58.040	YES	95.62	1.55	2	29.0 - 464	0.000013-0.500000	YES
420f	1920x1080	2 - 30	3264x1836	58.040	YES	95.62	1.55	2	29.0 - 464	0.000013-0.500000	YES
420v	1920x1080	2 - 60	3264x1836	58.040	YES	95.62	1.55	2	29.0 - 464	0.000008-0.500000	YES
420f	1920x1080	2 - 60	3264x1836	58.040	YES	95.62	1.55	2	29.0 - 464	0.000008-0.500000	YES

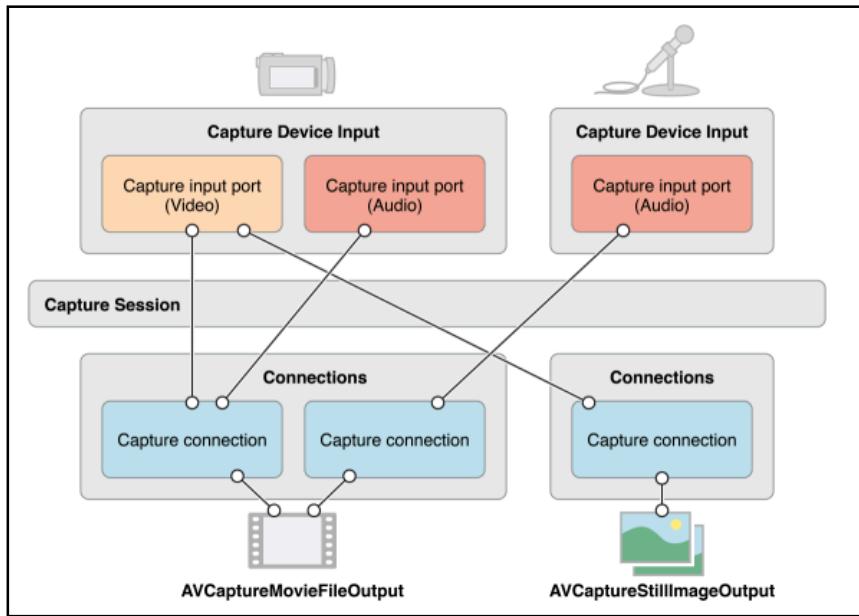
Rys. 2–16 Przykładowa lista dostępnych formatów dla kamery. Źródło: (12)

Kolejnym krokiem jest zdefiniowanie wyjścia sesji, czyli obiektu klasy dziedziczącej po AVCaptureOutput. Są cztery możliwości:

- [AVCaptureMovieFileOutput](#) - wyjście w formie pliku wideo.
- [AVCaptureVideoDataOutput](#) - wyjście w formie nieskompresowanych ramek wideo.
- [AVCaptureMovieFileOutput](#) - wyjście w formie bufora próbek audio.
- [AVCaptureStillImageOutput](#) - wyjście w formie wysokiej jakości nieruchomego obrazu lub zestawu obrazów, wraz z towarzącymi im metadanymi.

W tym przypadku odpowiednią klasą jest AVCaptureVideoDataOutput z uwagi na potrzebę pozyskiwania klatek wideo w czasie rzeczywistym.

Po stworzeniu obiektów reprezentujących wejście oraz wyjście należy dodać je do sesji kamery za pomocą metod addOutput(_) oraz addInput(_) obiektu klasy AVCaptureSession. Spowoduje to automatycznie stworzenie obiektów typu AVCaptureConnection odpowiadających każdemu z połączeń wejście-wyjście (Rys. 2–17). Obiekty te mogą być później używane do monitorowania oraz kontroli przepływu danych między dwoma końcami połączenia.



Rys. 2–17 Obiekty typu `AVCaptureConnection` reprezentujące połączenie między wejściami i wyjściami. Źródło: (14)

W końcu, by uzyskać dostęp do przechwytywanych klatek wideo należy w formie bufora próbek obrazu należy posłużyć się mechanizmem delegacji implementując metody protokołu `AVCaptureAudioDataOutputSampleBufferDelegate`. Teraz wystarczy rozpoczęć działanie sesji wywołując metodę `startRunning()` na obiekcie, który ją reprezentuje. Przechwytywane klatki obrazu będą dostępne w formie bufora próbek obrazu.

Dodatkowo, pracując z kamerą urządzenia iOS warto mieć na uwadze to, że w celu jej wykorzystania w aplikacji potrzebna jest zgoda użytkownika. Przy pierwszym uruchomieniu aplikacji w miejscu, gdzie używana jest kamera, po stworzeniu obiektu typu `AVCaptureDeviceInput` odnoszącego się do kamery, system iOS automatycznie pokaże okno dialogowe z pytaniem o to pozwolenie. Przy czym, możliwe jest zrobienie tego własnoręcznie wcześniej.

2.2.4. Framework SceneKit

SceneKit (15) to standardowy framework iOS, który służy do tworzenia aplikacji oraz gier wykorzystujących grafikę 3D. Biblioteka ta jest połączeniem silnika renderingu o wysokiej wydajności z wysokopoziomowym interfejsem programistycznym (API). SceneKit wspiera import, manipulację oraz rendering obiektów 3D. W przeciwieństwie do niskopoziomowych API jak OpenGL, który wymaga szczegółowej implementacji algorytmu renderowania by wyświetlić prostą scenę, SceneKit wymaga tylko opisu sceny, akcji i animacji.

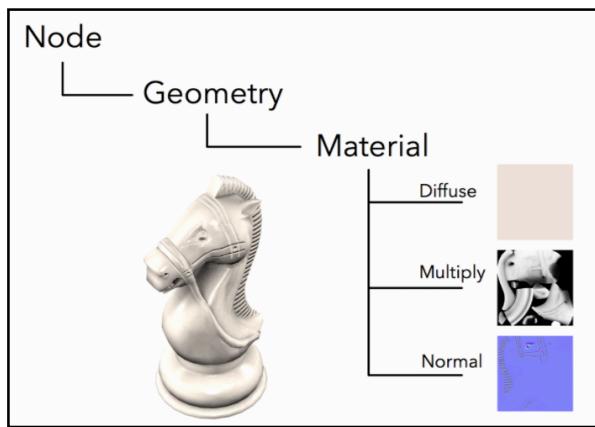
Węzły

Framework SceneKit organizuje zawartość sceny za pomocą hierarchii węzłów - obiektów typu `SCNNode` (16). Każdy węzeł ma pozycję, rotację oraz skalę w stosunku do swojego węzła-rodzica, który z kolei jest relatywny do swojego rodzica i tak aż do węzła-

korzenia. By umieścić obiekt w scenie należy dodać go jako dziecko innego węzła sceny. W celu zarządzania hierarchią służą metody:

- `addChildNode(_:)` - dodanie węzła jako dziecka,
- `insertChildNode(_: atIndex:)` - dodanie węzła jako dziecka na określoną pozycję,
- `removeFromParentNode()` - usunięcie relacji węzła z rodzicem

Kształt oraz wygląd węzłów jest opisywany za pomocą tzw. geometrii - obiektów typu `SCNGeometry` (Rys. 2–18). SceneKit oferuje zestaw gotowych geometrii, które można dostosować do własnych potrzeb, takich jak kula, prostopadłościan, czy płaszczyzna. Możliwe jest jednak wczytanie geometrii modelu 3D stworzonego w osobnym programie do tworzenia grafiki 3D. Framework SceneKit wspiera wiele popularnych rozszerzeń, jak `obj` czy `dae`. Wygląd węzła można dostosować korzystając z tzw. materiałów - obiektów typu `SCNMaterial`, w obrębie danej geometrii.



Rys. 2–18 Geometria jako składowa węzła. Źródło: (16)

Oświetlenie

Oświetlenie w framework SceneKit jest w pełni dynamiczne, przez co jest proste w użyciu ale mniej zaawansowane niż w przypadku kompleksowych silników graficznych (16). Każdy węzeł może stać się źródłem światła przez przypisanie obiektu reprezentującego światło - `SCNLight`, do właściwości `light` węzła. Rodzaje oświetlenia dzielą się na 4 typy:

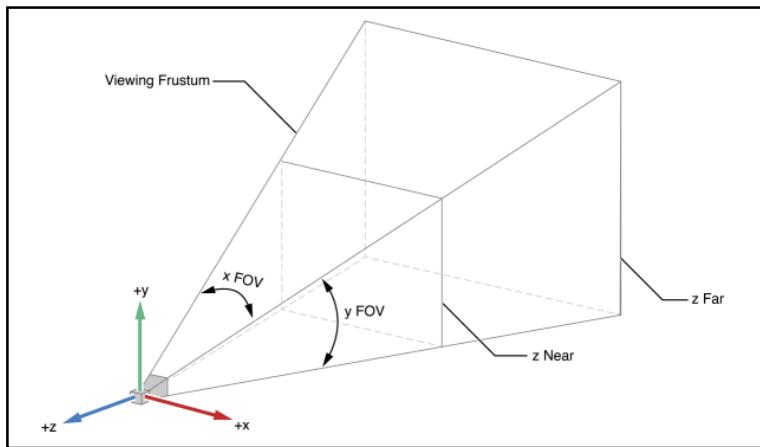
- Ambient - oświetlenie równomierne ze wszystkich stron,
- Directional - oświetlenie ukierunkowane o stałej intensywności,
- Omni - oświetlenie punktowe,
- Spotlight - oświetlenie obszaru na kształt rożka.



Rys. 2–19 Oświetlenie dynamiczne w SceneKit. Źródło: (16)

Kamera

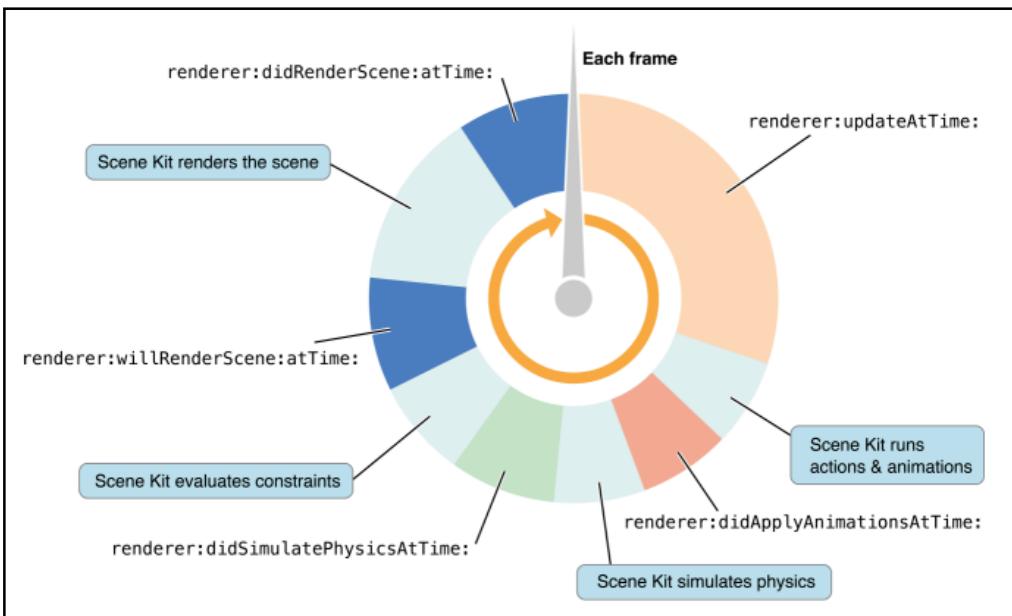
Kamerę reprezentuje obiekt typu `SCNCamera`, który podobnie jak w przypadku oświetlenia, jest sładową węzła i definiuje atrybuty kamery (17). Węzeł zawierający kamerę definiuje tzw. punkt widzenia, czyli pozycję oraz orientację kamery. Ponadto, widok renderowanej sceny oraz perspektywa zależy od ustawień kamery takich jak pole widzenia, limity widoczności oraz głębię ostrości.



Rys. 2–20 Parametry rzutu perspektywicznego kamery w SceneKit. Źródło: (17)

Pętla gry

By stworzyć aplikację lub grę przy użyciu SceneKit należy przygotować obiekt typu `SCNView`, lub innego typu zgodnego z protokołem `SCNSceneRenderer`, by wyświetlać przy jego użyciu scenę. Framework pozwala na udział w pętli renderowania i animacji za pomocą mechanizmu delegacji, w oparciu o protokół `SCNSceneRendererDelegate`, który delegat powinien implementować. Renderując scenę przy użyciu obiektu typu `SCNSceneRenderer` uruchamia on pętlę gry i rysuje scenę. Każda klatka przechodzi przez odpowiednie etapy, na których wywołuje określone metody delegata (Rys. 2–21).



Rys. 2–21 Pętla gry w SceneKit. Źródło: (15)

Obsługa interakcji

SceneKit korzysta z tych samych mechanizmów dla zdarzeń wywołanych przez dotyk i gesty co aplikacje iOS, z tą różnicą, że w tym przypadku do dyspozycji ma tylko jedno okno, które wyświetla scenę (16). Dla niektórych z gestów jest to wystarczające, jak gest przesunięcia palcem czy szczypnięcia, lecz dla pełnej obsługi takich wydarzeń jak dotknięcie ekranu potrzebne są dodatkowe informacje, np. który obiekt mieści się w obrębie dotkniętego obszaru. Dlatego właśnie SceneKit daje możliwość przeprowadzenia tzw. hit testu za pomocą metody `hitTest(_ options:)`, która zwraca tablicę elementów, dla każdego przecięcia geometrii węzła sceny z promieniem (odcinkiem) wyprowadzonym z kamery przez dany punkt - najczęściej punkt dotyku ekranu. Każdy z rezultatów zawiera informacje o węźle, którego geometrię przeciął promień, oraz szczegółowe informacje na temat przecięcia, takie jak współrzędne w 3D. Przykładowo, gdy potrzebna jest wiedza na temat pierwszego węzła, który został uderzony przez promień, potrzebny kod w języku Swift wygląda w ten sposób:

```
if let firstHit = sceneView.hitTest(tapLocation, options: nil)?.first as? SCNHitTestResult {
    let hitNode = firstHit.node
}
```

Wspólny z OpenGL kontekst renderowania

SceneKit pozwala wykorzystać obiekt typu `SCNRenderer`, który reprezentuje renderer sceny, do renderowania bezpośrednio do kontekstu OpenGL. Dzięki temu możliwe jest dodawanie zawartości rysowanej przez SceneKit do aplikacji, która już renderuje pewną treść przy pomocy OpenGL. By skorzystać z tej funkcjonalności frameworka wystarczy zainicjalizować renderer sceny przy użyciu konstruktora `init(context:options:)`, który tworzy go na podstawie podanego kontekstu OpenGL, oraz rysować scenę przy użyciu metody `renderAtTime(_:)`.

2.3. Biblioteka OpenCV

OpenCV (18) (Open Source Computer Vision) to biblioteka funkcji oraz algorytmów służących do analizy i przetwarzania obrazu w czasie rzeczywistym, wykorzystujących metody wizji komputerowej oraz uczenia maszynowego. Biblioteka ta opiera się na otwartym kodzie i jest dostępna na licencji BSD. Ponadto charakteryzuje ją modułowość oraz wsparcie dla wielu platform komputerowych i mobilnych. Jednym z głównych założeń biblioteki jest wysoka wydajność, ponieważ została ona stworzona na potrzeby aplikacji, który działają w czasie rzeczywistym. Dlatego biblioteka ta jest napisana natywnie w języku C/C++, przy czym udostępnia interfejsy oraz wrappery dla innych języków programowania takich jak C#, Python, MATLAB czy Java.

Bibliotekę OpenCV charakteryzuje struktura modułowa (19), w rezultacie czego składa się z wielu współdzielonych i statycznych bibliotek programistycznych. Dostępne są następujące moduły:

- core - zawiera definicje podstawowych struktur danych i funkcji, używanych przez inne moduły.
- imgproc - moduł przetwarzania obrazu, który jest odpowiedzialny m.in. za liniowe i nieliniowe filtrowanie obrazu, przekształcenia geometryczne, konwersje między przestrzeniami barw, i tym podobne.
- video - moduł analizy video, w którego skład wchodzą algorytmy śledzenia obiektów, odjemowania tła oraz wykrywanie ruchu.
- calib3d - odpowiedzialny za kalibrację kamer pojedynczych oraz stereo, definiujący m.in. algorytmy rekonstrukcji 3D i szacowania pozy obiektów.
- features2d - dostarcza narzędzi do wykrywania istotnych cech obrazów.
- objdetect - moduł odpowiedzialny za wykrywanie obiektów zdefiniowanych wcześniej klas, takich jak oczy, twarz, czy samochód.
- highgui - wysokopoziomowy moduł, który dostarcza łatwych w użyciu interfejsów dla przechwytywania obrazu z kamery oraz podstawowych elementów interfejsu użytkownika.
- gpu - zawiera definicje algorytmów z innych modułów, które wykorzystują procesory graficzne (GPU) do obliczeń równoległych.
- moduły pomocnicze, np. wiązania języka Python.

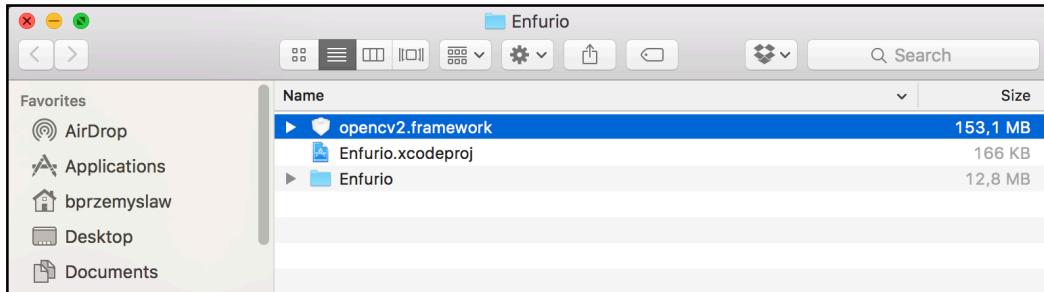
2.3.1. Dodanie biblioteki OpenCV do projektu iOS

Pobranie frameworka

OpenCV oficjalnie wspiera platformę iOS począwszy od wersji 2.4.2, a odpowiedni pakiet dystrybucyjny jest dostępny do pobrania na stronie biblioteki (20). Pakiet ten zawiera bibliotekę w formie frameworka iOS (21) - paczki plików, w której mieszą się pliki nagłówkowe oraz statycznie łączone biblioteki. Jest to wygodny sposób dystrybucji skompilowanych uprzednio bibliotek do programistów.

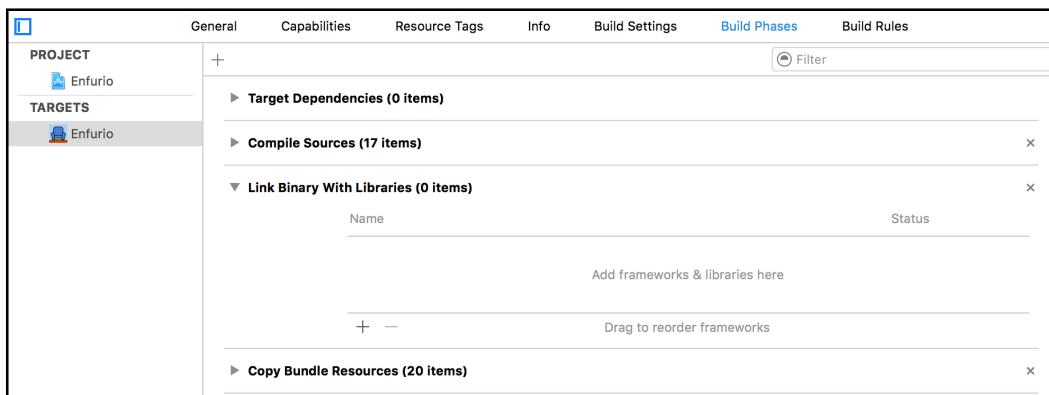
Dodatek do projektu

W celu pełnego dołączenia biblioteki OpenCV do projektu należy dodać zarówno pobrany framework jak i standardowy framework iOS o nazwie AssetsLibrary.framework, który jest wymagany do działania biblioteki. By dodać pobrany framework należy umieścić go w katalogu projektu (Rys. 2–22).

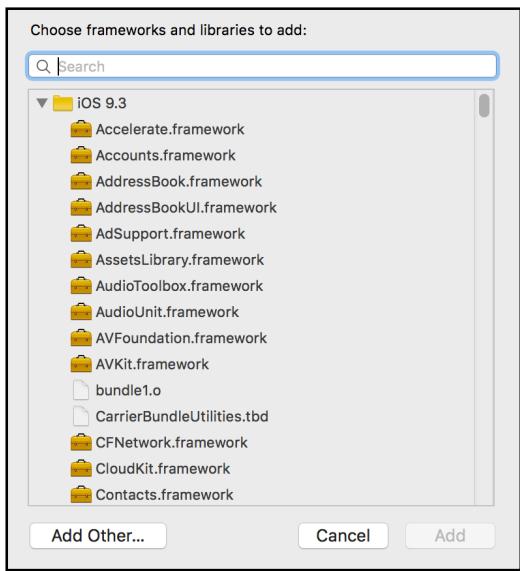


Rys. 2–22 Zawartość katalogu projektu po dodaniu frameworka OpenCV.

Następnie, powinno się poinformować środowisko Xcode o potrzebie uwzględnienia powyższego frameworka w etapie budowania projektu. Żeby to uczynić należy otworzyć ustawienia projektu i w zakładce Build Phases rozwinąć listę Link Binary with Libraries (Rys. 2–23), po czym kliknąć ikonkę ze znakiem plus. W tym momencie zostanie przedstawione okno z listą możliwych do dodania standardowych frameworków (Rys. 2–24). Tutaj należy kliknąć przycisk Add Other i za pomocą okna dialogowego, które się pojawi, wskazać lokalizację frameworka, który został wcześniej umieszczony w katalogu projektu.



Rys. 2–23 Lista Link Binary with Libraries w zakładce Build Phases projektu.



Rys. 2–24 Okno z listą standardowych frameworków.

Ostatnim krokiem jest dodanie frameworka AssetsLibrary.framework, który jest wymagany do poprawnego działania biblioteki OpenCV, z listy standardowych frameworków iOS (Rys. 2–24).

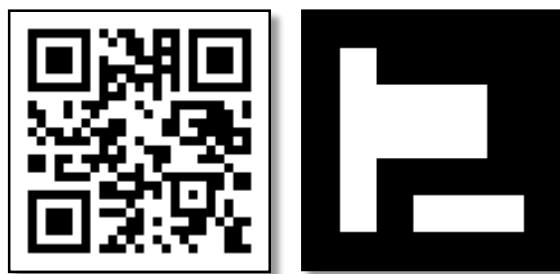
2.4. Wykrywanie znaczników

Szacowanie położenia obiektu w przestrzeni 3D jest bardzo ważnym elementem rozszerzonej rzeczywistości jako aplikacji metod wizji komputerowej. Proces ten polega na znalezieniu zależności między punktami w świecie rzeczywistym a odpowiadającym im punktom na rzucie 2D. Jest to złożony problem, dlatego powszechnie stosuje się znaczniki (markery) w celu jego uproszczenia. Istnieją również rozwiązania niekorzystające z markerów, lecz nie jest to przedmiotem niniejszej pracy.

2.4.1. Rodzaje znaczników

Wśród rodzajów markerów, które różnią się takimi cechami jak kształt, geometria, czy sposób kodowania, można wyróżnić m.in.:

- Znacznik binarny (Rys. 2–25) - dwuwymiarowy, binarny kod graficzny.
- Kod QR (Rys. 2–25) - dwuwymiarowy, alfanumeryczny, kwadratowy kod graficzny opracowany przez japońską firmę Denso-Wave w 1994 roku. Jest to szczególny przypadek znacznika binarnego.
- Obraz 2D - dwuwymiarowy obraz, w przypadku którego wykrywanie polega na metodach rozpoznawania obrazu.
- Obiekt 3D (Rys. 2–26) - trójwymiarowy obiekt, w przypadku którego rozpoznawanie polega dopasowaniu do określonej bryły geometrycznej,
- Tekst - dwuwymiarowy, alfanumeryczny, w przypadku którego wykorzystywane jest rozpoznawanie tekstu.



Rys. 2–25 Przykład kodu QR (lewy obrazek) oraz markera binarnego (prawy obrazek).



Rys. 2–26 Cylinder jako znacznik w formie obiektu 3D - puszki.

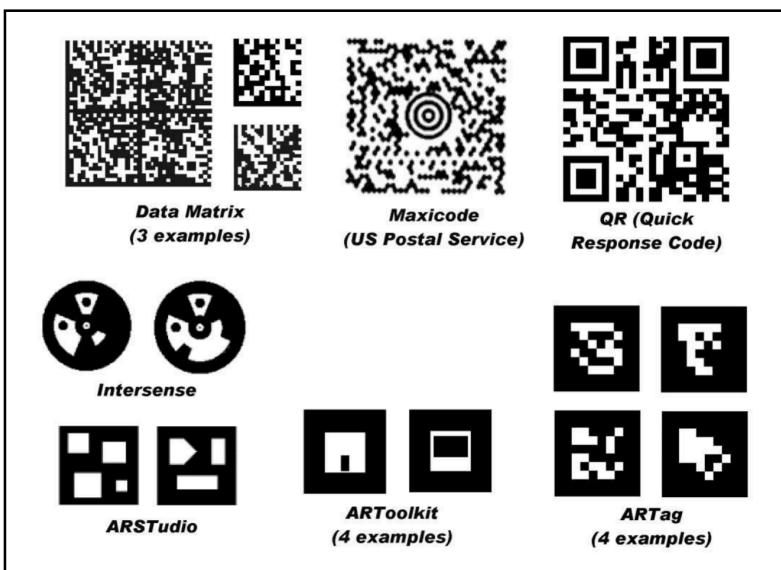
2.4.2. Znacznik binarny

Jednym z najpopularniejszych podejść jest korzystanie ze znaczników binarnych. Markery tego typu mogą przyjmować wiele kształtów () i być kodowane za pomocą różnych algorytmów, np. kodem Hamminga. Głównymi zaletami tego rodzaju markerów są:

- możliwość jednoznacznego określenia położenia znacznika w przestrzeni trójwymiarowej,
- prostota kodowania znacznika,
- brak potrzeby nauczania, np. w przeciwieństwie do sieci neuronowych używanych do rozpoznawania obrazu,
- stosunkowo wysoka wydajność, ze względu na proste kształty, takie jak czworokąty czy okręgi.

Z drugiej strony, wadami tego rozwiązania są:

- nieelegancki wygląd markera, w przeciwieństwie do markerów wykorzystujących obrazy 2D lub obiekty 3D,
- markery nie są intuicyjne, ponieważ wygląd znacznika nie odzwierciedla elementu wirtualnego, który będzie wstawiony w jego miejsce.



Rys. 2–27 Różne kształty i sposoby kodowania markerów binarnych. Źródło: (22)

2.4.3. Wykrywanie kwadratowych znaczników binarnych

Dla danego obrazu, na którym widoczne są markery binarne, wynikiem procesu wykrywania znaczników powinna być lista odnalezionych znaczników, wraz z informacją dla każdego z nich na temat:

- unikalnego numeru identyfikującego (ID) dany marker,
- położenia markera, czyli pozycji każdego z jego czterech kątów.

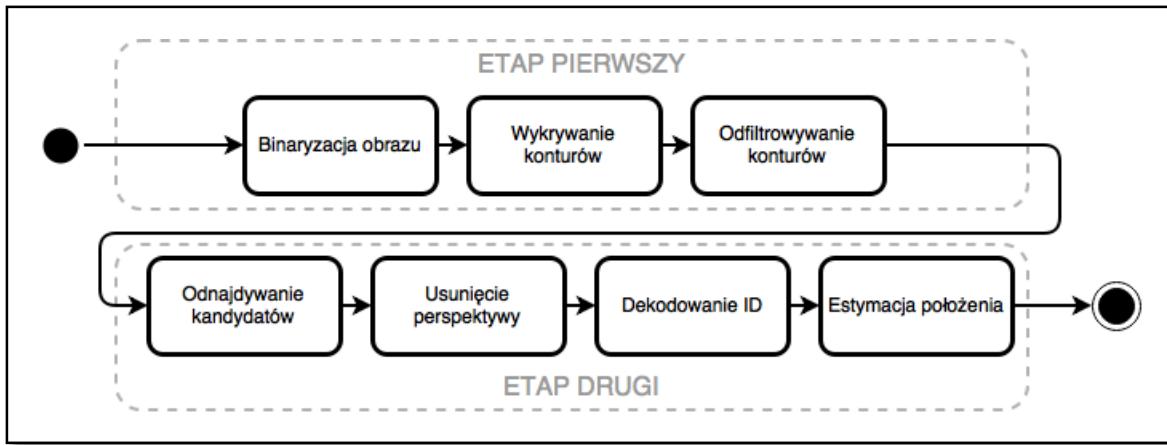
Działanie algorytmu wykrywania znaczników można podzielić na dwa główne etapy, ze względu na powiązane ze sobą kroki (Rys. 2–28), które obejmują:

1. Etap wykrywania kandydatów na markery,
2. Etap dekodowania markerów.

W pierwszym etapie obraz jest analizowany pod kątem kwadratowych kształtów, które mogą być znacznikami. Etap ten rozpoczyna się od progowania (ang. thresholding) adaptacyjnego z pojedynczym progiem - binaryzacja. Następnie, identyfikowane są kontury na zbinaryzowanym obrazie, przy czym te, które nie są wypukłe lub nie mogą zostać w przybliżeniu określone jako kwadratowe, są odrzucane. Pod koniec nakładane są dodatkowe filtry, by odfiltrować zbyt małe, zbyt duże kontury, kandydatów, kontury wewnętrzne kandydata, itp. Rezultatem tego etapu są wycinki obrazu zawierające kandydatów na markery.

W drugim etapie, po odnalezieniu kandydatów na znaczniki, potrzebna jest ich dalsza analiza i przetwarzanie, w celu zdekodowania zapisanych w nich informacji. Etap ten rozpoczyna się od określenia i nałożenia odpowiednich transformacji na obraz - wycinek zawierający kandydata na znacznik, w celu usunięcia perspektywy. Następnie, przeprowadzane jest progowanie, by wyróżnić białe fragmenty od czarnych. Po tym, obraz dzielony jest na komórki, zgodnie z zadanym rozmiarem markera, grubością jego konturów, oraz ilością białych lub czarnych pikseli dla każdej z komórek, która jest potrzebna do określenia, czy dana

komórka jest biała czy czarna. Pod koniec, następuje próba odkodowania markera, wykorzystując przy tym techniki korekcji błędów i sprawdzania parzystości bitów.



Rys. 2–28 Diagram przedstawiający kroki działania algorytmu wykrywania znaczników.

Kodowanie Hamminga

Jednym z najbardziej popularnych sposobów kodowania kwadratowych znaczników binarnych jest wykorzystanie jednego z wariantów kodu Hamminga - binarnego liniowego kodu z kontrolą parzystości bitów, wynalezionej w 1950 roku przez Richarda Hamminga.

Algorytm kodowania przy pomocy tego kodu jest następujący:

1. Pozycje, które są potęgami liczby 2 są bitami parzystości.
2. Pozycje niebędące potęgami liczby 2 są bitami informacyjnymi.
3. Bit parzystości wskazuje na parzystość określonej grupy bitów w słowie - jego pozycja determinuje, które bity powinny być sprawdzone, a które nie:
 - a. Bit parzystości na pozycji 1: opuszcza 0 bitów, sprawdza 1 bit, opuszcza 1 bit, sprawdza 1 bit, opuszcza 1 bit itd.
 - b. Bit parzystości na pozycji 2: opuszcza 1 bit, sprawdza 2 bity, opuszcza 2 bity sprawdza 2 bity, opuszcza 2 bity itd.
 - c. Bit parzystości na pozycji 4: opuszcza 3 bity, sprawdza 4 bity, opuszcza 4 bity sprawdza 4 bity, opuszcza 4 bity itd.

Pozycja bitu	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	
Bit parzystości (p), danych (d)	p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8	d9	d10	d11	p16	d12	d13	d14	d15	
Sekwencja sprawdzanych bitów	p1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
	p2		X	X				X	X						X	X			X	X	...
				X	X	X	X					X	X	X	X					X	
								X	X	X	X	X	X	X							
																	X	X	X	X	X

Rys. 2–29 Zasada kodowania dla słowa o długości 20 (15 bitów danych, 5 bitów parzystości). Źródło: (23).

2.4.4. Biblioteki AR na urządzenia mobilne

Wykrywanie znaczników można zaimplementować w oparciu o bibliotekę wizji komputerowej, takiej jak OpenCV, lub za pomocą gotowej biblioteki rozszerzonej rzeczywistości. Warto przyjrzeć się popularnym rozwiązaniom dostępnym na rynku, które oferują wsparcie dla platform mobilnych:

- ARToolKit (24) - wieloplatformowa biblioteka, wokół której skupiona jest duża społeczność programistów. Wśród funkcjonalności które wspiera są: śledzenie cech naturalnych, śledzenie znaczników, wsparcie dla kamer stereoskopowych, wsparcie dla silnika Unity3D oraz OpenSceneGraph. ARToolKit to projekt open-source.
- Vuforia (25) - wieloplatformowa biblioteka firmy PTC Inc. do zastosowań profesjonalnych, cechująca się dobrym wsparciem ze strony producenta. Obok wykrywania i śledzenia znaczników, pozwala również na wykrywanie obiektów (np. butów), figur geometrycznych (np. cylindra), obrazu oraz tekstu. Vuforia jest biblioteką płatną, lecz producent daje możliwość korzystania z jej funkcjonalności pod warunkiem, że zostanie zastosowany znak wodny Vuforia.
- Wikititude (26) - wieloplatformowa biblioteka firmy Wikitude. Wśród funkcjonalności, które oferuje są m.in.: rozpoznawanie i śledzenie obrazu, renderowanie modeli 3D w ich miejsce, oraz wsparcie dla aplikacji rozszerzonej rzeczywistości opartych o lokalizację. Wikitude jest biblioteką płatną, ale udostępnia darmową wersję próbную na podobnych zasadach co Vuforia - ze znakiem wodnym.
- CraftAR (27) - wieloplatformowa biblioteka firmy Catchoom. Zapewnia ona implementację rozpoznawania obrazu, oraz wstawianie i wyświetlanie wirtualnych elementów w jego miejsce. Producent oferuje również dedykowaną aplikację do przygotowywania nakładanej na obraz grafiki 3D. CraftAR jest płatna, lecz pozwala na darmowe korzystanie z jej funkcjonalności z pewnymi miesięcznymi limitami.
- ARLab (28) - biblioteka firmy ARLab dla platform mobilnych, która oferuje m.in. śledzenie obrazów i obiektów, oraz narzędzia potrzebne do przygotowania własnej grafiki 3D, która ma być nakładana na śledzone obiekty. ARLab jest biblioteką płatną.

Wymienione biblioteki AR nierzadko wspierają cały proces generowania obrazu rozszerzonej rzeczywistości w oparciu o znaczniki - od wykrywania markerów, przez nakładanie modeli 3D wirtualnych obiektów, po możliwość obsługę interakcji i wyświetlanie ich w różnych trybach.

3. Przedstawienie problemu

Problemem, którego dotyczy niniejsza praca jest stworzenie prototypu systemu wizualizacji rozszerzonej rzeczywistości, w oparciu o wykrywanie znaczników. Na rynku dostępne są podobne rozwiązania, ale proste systemy tego typu nie zapewniają wysokiej immersyjności, natomiast zaawansowane rozwiązania są dopiero rozwijane i stosunkowo drogie. W pracy postanowiono połączyć prostotę systemu AR realizowanego na urządzeniu mobilnym w oparciu o znaczniki, z wysoką immersyjnością zapewnianą przez stosowanie relatywnie tanich okularów wirtualnej rzeczywistości do podglądu obrazu. W poniższych rozdziałach zostaną przedstawione główne założenia co do tworzonego w ramach pracy systemu, oraz wybrane podejścia do zagadnień z nimi związanych.

3.1. Główne założenia

Głównymi założeniami systemu tworzonego w ramach pracy są:

- podgląd wizualizacji w czasie rzeczywistym,
- wstawianie modeli 3D w miejsce znaczników, odzwierciedlając przy tym ich położenie w przestrzeni trójwymiarowej,
- możliwość interakcji z wstawianymi modelami,
- wysoki stopień immersyjności, poprzez wykorzystanie okularów wirtualnej rzeczywistości do podglądu wizualizacji,
- przedstawienie zastosowania systemu w praktyce, na przykładzie aplikacji mobilnej.
- działanie na urządzeniach przenośnych z systemem iOS

3.2. Wybrany sposób realizacji

3.2.1. Przechwytywanie obrazu z kamery

W celu przechwytywania obrazu rzeczywistości zdecydowano się na wykorzystanie tylnej kamery urządzenia iPhone, dzięki czemu możliwa będzie rejestracja widoku przed obserwatorem i jednoczesny podgląd na ekranie urządzenia.

Ponadto, postanowiono skorzystać z natywnej biblioteki AVFoundation systemu iOS do przechwytywania obrazu z kamery, ponieważ zastosowanie tego rozwiązania pozwala na dostęp do pojedynczej klatki obrazu oraz zapewnia wysoką wydajność.

3.2.2. Wykrywanie znaczników

W pracy zdecydowano się na samodzielna implementację algorytmu wykrywania znaczników, w oparciu o bibliotekę wizji komputerowej OpenCV. Przyjęto to podejście, ponieważ zapewnia ono większą kontrolę nad rozwiązaniem i eliminuje zależność od rozbudowanej biblioteki AR, której duża część pozostała by niewykorzystana.

3.2.3. Wstawianie wirtualnych przedmiotów

Do wstawiania elementów świata wirtualnego w postaci modeli 3D postanowiono wykorzystać natywny framework SceneKit systemu iOS. Wybór ten uzasadniony jest m.in. tym, że biblioteka ta jest:

- prosta w użyciu dzięki wysokopoziomowemu API, które jest spójne z innymi frameworkami iOS,
- wydajna, ponieważ jest wspierana natywnie,
- daje możliwość renderowania treści przy użyciu istniejącego kontekstu OpenGL (obraz z kamery),
- zapewnia obsługę interakcji z modelami 3D.

Do tego celu mogłyby posłużyć każda inna biblioteka grafiki 3D wspierająca platformę iOS, ale samo dołączenie jej do projektu stanowiłoby kolejne zagadnienie, a korzystanie nie byłoby tak proste jak w przypadku frameworka SceneKit, który jest częścią platformy.

Rozważano również implementację przy użyciu biblioteki OpenGL, eliminując potrzebę korzystania z dodatkowych narzędzi oraz zapewniając większą kontrolę nad tworzonym system, lecz z uwagi na złożoność problemu tworzenia grafiki 3D przy użyciu tak niskopoziomowego rozwiązania jak OpenGL, zrezygnowano z tego podejścia.

3.2.4. Wyświetlanie obrazu

Generowany w czasie rzeczywistym obraz rozszerzonej rzeczywistości postanowiono wyświetlać na ekranie urządzenia iOS. Przy czym, zdecydowano się na wsparcie dwóch trybów podglądu obrazu:

- tryb mono - obraz wyświetlany jest na ekranie telefonu w formie pojedynczego widoku, służącego do podglądu wizualizacji bezpośrednio na urządzeniu przenośnym,
- tryb stereo - obraz wyświetlany jest na ekranie telefonu w formie dwóch, odpowiednio rozmieszczonych i zniekształconych widoków, które pozwalają na podgląd wizualizacji przez okulary VR.

Podgląd w trybie stereo

Do podglądu wizualizacji w trybie stereo potrzebne są okulary VR. Zdecydowano się na wykorzystanie okularów Google Cardboard (29), ponieważ są powszechnie dostępne, tanie i wystarczające do tworzonego rozwiązania - umożliwiają zarówno podgląd obrazu stereo, jak i prostą interakcję z ekranem telefonu, ponieważ posiadają przycisk fizyczny. Okulary Cardboard charakteryzuje duże wsparcie społeczności skupionej wokół problemu wirtualnej rzeczywistości oraz firmę Google.

Dodatkową zaletą zastosowania okularów Cardboard jest ich kartonowa obudowa, dzięki której możliwe jest łatwe dostosowanie okularów do rozwiązania, poprzez wycięcie otworu na tylną kamerę urządzenia przenośnego.



Rys. 3–1 Okulary VR firmy Google - Google Cardboard.

4. Rozwiązanie

Na rozwiązanie problemu, który jest poruszany w niniejszej pracy składają się następujące etapy realizacji:

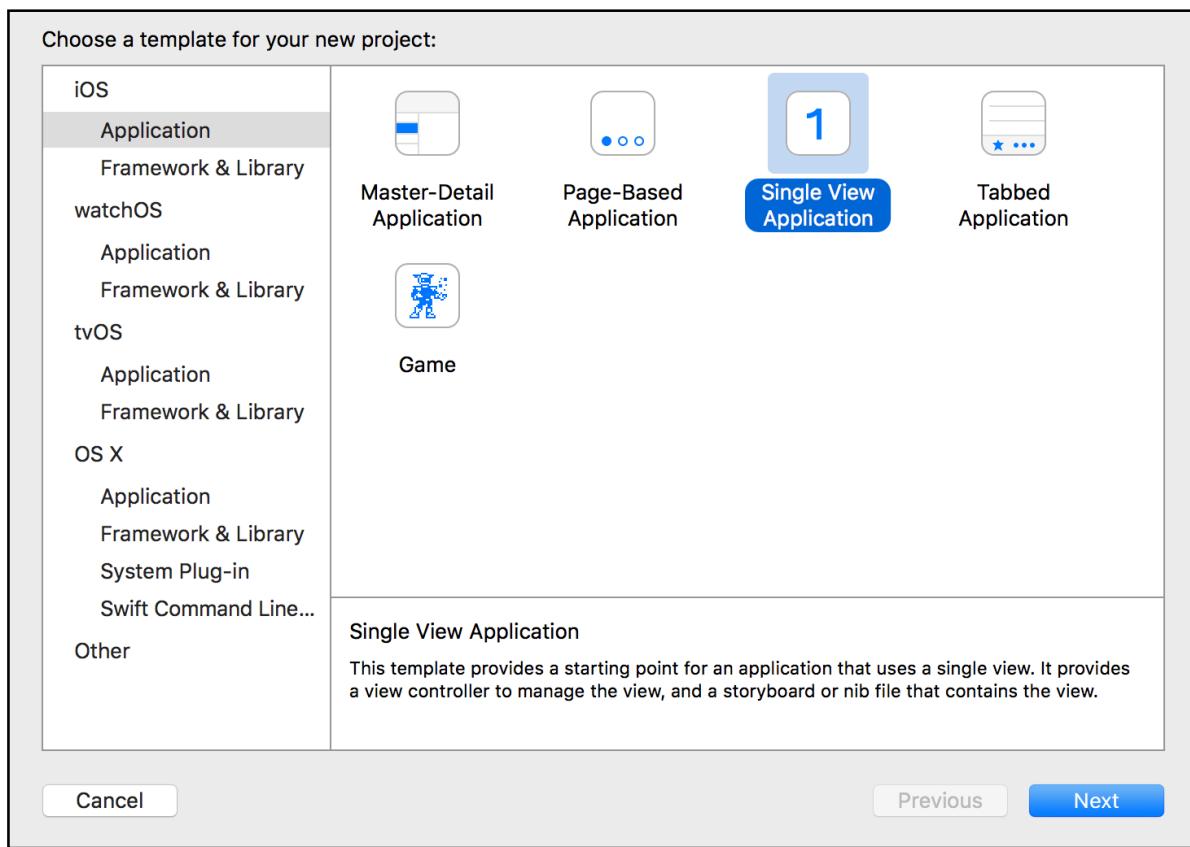
- Przygotowanie szkieletu aplikacji - obejmuje utworzenie projektu i przygotowanie środowiska do dalszej pracy, wliczając w to dołączenie odpowiednich bibliotek.
- Przechwytywanie obrazu z kamery - obejmuje konfigurację kamery potrzebną do uzyskania dostępu do klatki obrazu, oraz jej przygotowanie do dalszego przetwarzania.
- Wykrywanie znaczników - obejmuje przygotowanie znaczników, wraz z opisem sposobu kodowania, oraz implementację algorytmu ich wykrywania.
- Wstawianie wirtualnych przedmiotów - obejmuje proces przygotowania modeli 3D wirtualnych obiektów, wstawiania ich oraz obsługę interakcji.
- Wyświetlanie obrazu - obejmuje realizację wyświetlania obrazu w trybach mono i stereo.

Każdy spośród wyżej wymienionych etapów realizacji zostanie szerzej opisany w poniższym dziale. Dodatkowo, rozwiązanie zostanie przetestowane i przeprowadzona będzie analiza pod kątem jakości tworzonego systemu, którego główną miarą jest immersyjność.

4.1. Przygotowanie szkieletu aplikacji

4.1.1. Utworzenie projektu

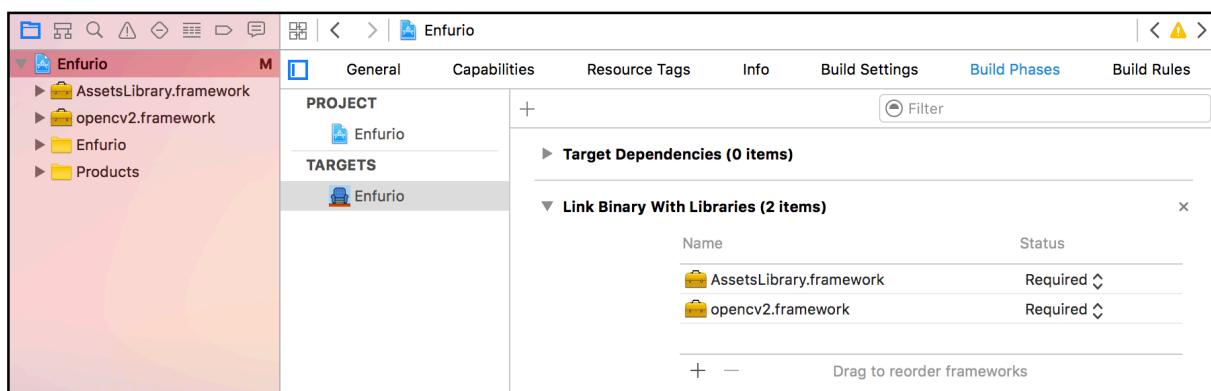
Projekt aplikacji mobilnej iOS stworzono przy użyciu szablonu Single View Application (Rys. 4–1), który jest punktem startowym dla aplikacji korzystających z jednego głównego widoku. Zapewnia on klasę kontrolera połączoną z widokiem, którym zarządza, wraz z plikiem typu storyboard, w którym znajduje się prototyp tego widoku. Jako język główny projektu wybrano Swift, a jako minimalną wspieraną wersję systemu iOS 9.0.



Rys. 4–1 Szablon Single View Application aplikacji mobilnej iOS.

4.1.2. Dodanie biblioteki OpenCV

W celu umożliwienia korzystania z biblioteki OpenCV w projekcie, pobrano pakiet dystrybucyjny w wersji 2.4.13, udostępniony dla platformy iOS na oficjalnej stronie OpenCV (20). Następnie, dołączono do projektu pobrany framework, wraz z potrzebnymi zależnościami, zgodnie z opisem zawartym w rozdziale 2.3.1. Powstałą w wyniku tego procesu strukturę i listę zależności przedstawia poniższy rysunek:



Rys. 4–2 Struktura oraz zależności projektu po dodaniu biblioteki OpenCV.

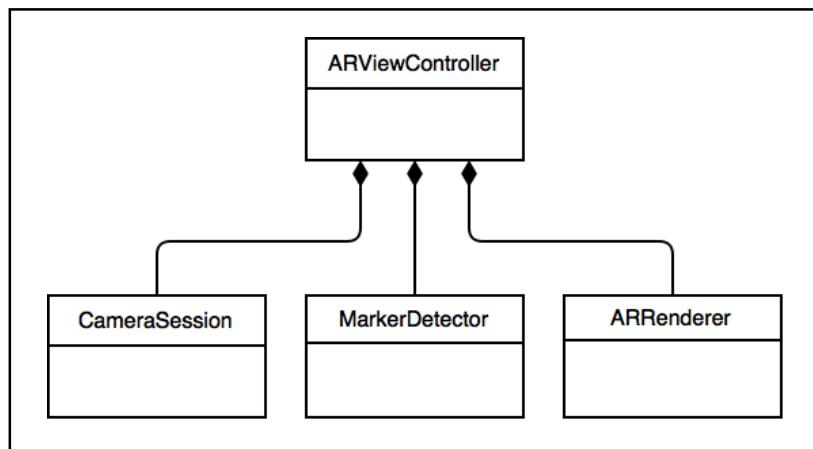
Dodatkowo, by zapewnić dostęp do funkcjonalności biblioteki OpenCV z poziomu języka Swift, przygotowano typy opakowujące - tzw. wrappery, korzystając z Objective-C++. Sporządzanie mostu między językiem Swift oraz C++ zostało szczegółowo opisane w rozdziale

2.2.2. Wrappery sporządzono dla prototypów klas C++, które będą potrzebne do bezpośredniej komunikacji z frameworkm: klasy Marker - reprezentującej znacznik, oraz klasy MarkerDetector - zapewniającej interfejs dla algorytmu wykrywania znaczników.

4.1.3. Architektura aplikacji

Aplikacja reprezentująca tworzony system opiera się na jednym głównym widoku - stąd wybór szablonu Single View Application. Przy czym, kontroler ARViewController zarządzający tym widokiem jest odpowiedzialny za koordynację procesu powstawania obrazu rozszerzonej rzeczywistości oraz jego prezentację. W tym celu wykorzystuje następujące komponenty aplikacji:

- Komponent sesji kamery - dostarczanie klatek obrazu z kamery.
- Komponent wykrywania znaczników - odnajdywanie znaczników na obrazie.
- Komponent wizualizacji - konstrukcja i wyświetlanie obrazu rozszerzonej rzeczywistości.



Rys. 4–3 Diagram przedstawiający uproszoną architekturę aplikacji.

Komponent sesji kamery, który mieści klasa CameraSession, jest odpowiedzialny za dostarczanie nowych klatek obrazu z kamery do dalszego przetwarzania. Oznacza to, że sesja kamery realizuje takie zadania jak wybór urządzenia przechwytyującego, konfigurację jego parametrów, oraz przechwytywanie klatek w tle i ich konwersję do formatu gotowego do dalszego przetwarzania.

Z kolei implementacja wykrywania znaczników zawarta jest w klasie MarkerDetector, która zapewnia interfejs do wymiany informacji dotyczących odnalezionych znaczników na danym obrazie. MarkerDetector jest typem bezpośrednio korzystającym z biblioteki OpenCV, dlatego zdecydowano się na jego implementację przy pomocy języka C++. Dużą zaletą tego rozwiązania jest też możliwość wykorzystania tej klasy jako modułu na innych niż iOS platformach.

Natomiast komponent odpowiedzialny za konstrukcję i wyświetlanie obrazu AR - komponent wizualizacji, mieści w obrębie klasy bezpośrednio go reprezentującej -

ARRenderer, całą logikę potrzebną do wygenerowania obrazu rozszerzonej rzeczywistości w oparciu o obraz z kamery oraz informacje dotyczące odnalezionych na nim znaczników.

W końcu, klasa ARViewController, koordynuje wymianę informacji między poszczególnymi komponentami i wykorzystuje uzyskany w rezultacie obraz rozszerzonej rzeczywistości do przedstawienia go w widoku, którym zarządza.

4.2. Przechwytywanie obrazu z kamery

Za przechwytywanie klatek obrazu z kamery w czasie rzeczywistym odpowiedzialny jest komponent sesji kamery, który opiera swoje działanie o framework AVFoundation systemu iOS. W poniższych rozdziałach zostanie szerzej opisana realizacja zadań tej klasy, takich jak konfiguracja sesji kamery, wraz z konfiguracją urządzenia przechwytyującego, oraz proces przygotowywania klatki obrazu do dalszego przetwarzania przez inne komponenty.

4.2.1. Konfiguracja sesji kamery

By odpowiednio przygotować sesję kamery urządzenia iOS, zgodnie z opisem w rozdziale 2.2.3 niniejszej pracy, potrzebna jest konfiguracja wejść i wyjść sesji, oraz ustawienie parametrów urządzenia przechwytyującego.

To zadanie realizowane jest przez funkcję configureCaptureSession() klasy CameraSession, która z kolei korzysta z osobnych funkcji do realizacji każdego z wymienionych podzadań. Poniżej znajduje się definicja omawianej funkcji:

```
func configureCaptureSession() {
    self.captureSession.beginConfiguration()

    // Camera
    self.setupCaptureSessionInput()
    self.setupCaptureSessionOutput()
    self.configureCameraDevice()

    self.captureSession.commitConfiguration()
}
```

Za konfigurację wejścia sesji kamery odpowiedzialna jest funkcja setupCaptureSessionInput() klasy CameraSession, w której tylna kamera urządzenia dodawana jest jako urządzenie wejściowe. Poniżej znajduje się definicja omawianej funkcji:

```
func setupCaptureSessionInput() {
    do {
        let possibleCameraInput: AnyObject? = try AVCaptureDeviceInput(
            device: self.cameraDevice
        )
        let input = possibleCameraInput as! AVCaptureDeviceInput
        if self.captureSession.canAddInput(input) {
            self.captureSession.addInput(input)
        }
    }
    catch {
        print("Failed to setup camera input: \(error)")
    }
}
```

Z kolei odpowiednie przygotowanie wyjścia sesji ma miejsce w funkcji setupCaptureSessionOutput() klasy CameraSession. Najpierw obiekt cameraVideoOutput klasy AVCaptureVideoDataOutput określany jest jako delegat, który będzie otrzymywać dane wyjściowe sesji - bufor pamięci zawierający klatki obrazu z kamery. Następnie, ustawiany jest format wyjściowy klatek, a po tym następuje próba przypisania skonfigurowanego obiektu cameraVideoOutput jako wyjścia sesji. Poniżej znajduje się definicja omawianej funkcji:

```
func setupCaptureSessionOutput() {
    self.cameraVideoOutput.setSampleBufferDelegate(self,
        queue: dispatch_queue_create(
            "camera sample buffer delegate", DISPATCH_QUEUE_SERIAL)
    )
    self.cameraVideoOutput.videoSettings = NSDictionary(
        object: NSNumber(unsignedInt: kCVPixelFormatType_32BGRA),
        forKey: kCVPixelBufferPixelFormatKey as NSString
    ) as [NSObject: AnyObject]
    if self.captureSession.canAddOutput(self.cameraVideoOutput) {
        self.captureSession.addOutput(self.cameraVideoOutput)
        let videoOutputConnection = self.cameraVideoOutput
            .connectionWithMediaType(AMMediaTypeVideo)
        videoOutputConnection.videoOrientation = .Portrait
    }
    self.cameraVideoOutput.alwaysDiscardsLateVideoFrames = true
}
```

Natomiast sama konfiguracja kamery odbywa się w funkcji configureCameraDevice() klasy CameraSession. Proces rozpoczyna ustawienie odpowiedniego formatu przechwytywanego obrazu. Tutaj następuje próba narzucenia formatu o zadanej rozdzielcości i ilości klatek na sekundę, ale w przypadku gdyby nie było to możliwe (urządzenie nie obsługuje danego formatu), zostanie przypisany jeden z presetów, o których była mowa w rozdziale 3.2.1. Poniżej znajduje się definicja omawianej funkcji:

```
func configureCameraDevice() {
    let desiredFramerate = 30.0
    let desiredResolution = CMVideoDimensions(width: 1280, height: 720)

    let desiredFormat: AVCaptureDeviceFormat? = self.cameraDevice.formats
        .filter({
            let resolution = CMVideoFormatDescriptionGetDimensions(
                $0.formatDescription)
            let frameRateRanges = $0.videoSupportedFrameRateRanges
                as! [AVFrameRateRange]
            let supportsRequiredResolution = resolution.width ==
                desiredResolution.width &&
                resolution.height == desiredResolution.height
            let supportsRequiredFrameRate =
                frameRateRanges[0].maxFrameRate == desiredFramerate
            return supportsRequiredResolution && supportsRequiredFrameRate
        }).first as? AVCaptureDeviceFormat

    if let desiredFormat = desiredFormat {
        do {
            try self.cameraDevice.lockForConfiguration()
            self.cameraDevice.activeFormat = desiredFormat
            self.cameraDevice.activeVideoMinFrameDuration =
                CMTIMEMake(1, Int32(desiredFramerate))
            self.cameraDevice.activeVideoMaxFrameDuration =

```

```

        CMTIMEMake(1, Int32(desiredFramerate))
    self.cameraDevice.focusMode =
        AVCaptureFocusMode.ContinuousAutoFocus
    self.cameraDevice.unlockForConfiguration()
}
catch {
    print("Failed to lock camera for configuration: \(error)")
}
}
else {
    print("Desired camera format is not available on your device,
          falling back to default")
    self.captureSession.sessionPreset = AVCaptureSessionPresetMedium
}
}

```

4.2.2. Przygotowanie klatki obrazu

Klatki obrazu są dostarczane poprzez delegację - wywoływaną jest funkcja `captureOutput(_:didOutputSampleBuffer:fromConnection:)` w postaci bufora. Bufor ten w pierwszym kroku konwertowany jest do bufora obrazu. Następnie na jego podstawie przygotowywane są dwie wersje klatki: do wyświetlenia oraz do dalszego przetwarzania. Klatka do wyświetlenia otrzymywana jest w wyniku konstrukcji obrazu na podstawie bufora, natomiast w przypadku wersji do dalszej obróbki miejsce ma konwersja do klatki w formacie BGRA, na którym operują wykorzystywane na kolejnych etapach przetwarzania obrazu funkcje biblioteki OpenCV. Poniżej znajduje się definicja omawianej funkcji:

```

func captureOutput(
    captureOutput: AVCaptureOutput!,
    didOutputSampleBuffer sampleBuffer: CMSampleBuffer!,
    fromConnection connection: AVCaptureConnection!) {

    // Get pixel buffer
    let pixelBuffer = CMSampleBufferGetImageBuffer(sampleBuffer)!

    // Lock the buffer
    CVPixelBufferLockBaseAddress(pixelBuffer, 0)

    // Get frame image
    let frameImage = CIImage(CVPixelBuffer: pixelBuffer)

    // Get frame in BGRA format
    let baseAddress = UnsafeMutablePointer<UInt8>(
        CVPixelBufferGetBaseAddress(pixelBuffer)
    )
    let width: size_t = CVPixelBufferGetWidth(pixelBuffer)
    let height: size_t = CVPixelBufferGetHeight(pixelBuffer)
    let stride: size_t = CVPixelBufferGetBytesPerRow(pixelBuffer)
    let frameBGRA: BGRAVideoFrame = BGRAVideoFrame(
        width: width,
        height: height,
        stride: stride,
        data: baseAddress
    )

    // Call delegate
    self.delegate?.cameraSessionDidCaptureOutput(
        frameImage: frameImage,

```

```

        frameBGRA: frameBGRA
    )

// Unlock the buffer
CVPixelBufferUnlockBaseAddress(pixelBuffer, 0)
}

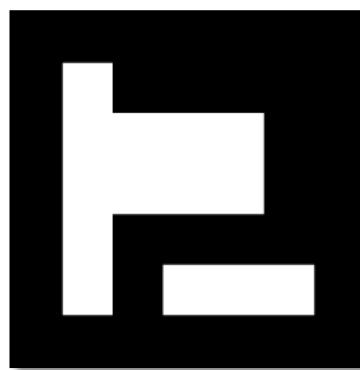
```

4.3. Wykrywanie znaczników

Kluczowym elementem tworzonego systemu AR jest wykrywanie znaczników. W poniższych rozdziałach zostanie opisana implementacja potrzebnego algorytmu, oparta na bibliotece OpenCV, oraz wybrany typ i kodowanie znaczników.

4.3.1. Znaczniki

Znaczniki binarne, które postanowiono zastosować w celu realizacji systemu AR, zostały zaprojektowane jako siatka 5x5 białych i czarnych kwadratów, z dodatkową czarną ramką o grubości odpowiadającej jednemu kwadratowi.



Rys. 4–4 Przykładowy znacznik wspierany przez tworzony system.

Kodowanie

W pracy zastosowano zmodyfikowaną wersję kodowania Hamminga, które zostało opisane w rozdziale 2.4.3. Wspomniana modyfikacja polega na odwróceniu wartości pierwszego bita, by uniemożliwić odczytanie czarnego kwadratu jako poprawnego znacznika, stąd przykładowo markerowi o ID równym 0 odpowiada reprezentacja w postaci binarnej 10000, zamiast 00000. Dzięki wykorzystaniu tego typu kodowania, system umożliwia zakodowanie 1024 unikalnych numerów identyfikacyjnych znaczniki.

4.3.2. Implementacja algorytmu

Zgodnie z opisem algorytmu wykrywania kwadratowych znaczników binarnych zawartym w rozdziale 2.4, zaimplementowano go w oparciu o następujące kroki:

1. Binaryzacja obrazu.
2. Wykrywanie konturów.
3. Odfiltrowywanie konturów.
4. Odnajdywanie kandydatów.
5. Usunięcie perspektywy.

6. Dekodowanie ID.
7. Estymacja położenia.

W poniższych podrozdziałach zostanie opisana implementacja każdego z wyżej wymienionych kroków.

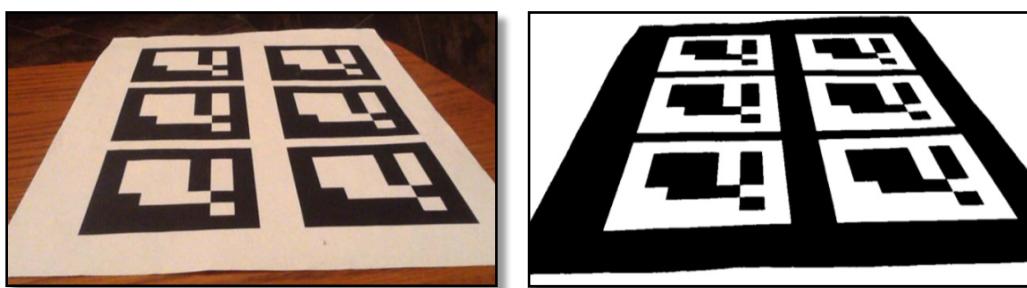
Binaryzacja obrazu

Na początku tego kroku, by ułatwić dalsze przetwarzanie i analizę obrazu, zostanie on przekonwertowany do skali szarości. Do tego celu posłużyono się funkcją `cv::cvtColor()` biblioteki OpenCV, która odpowiednio przetwarza przyjmowany przez nią jako argument obraz formacie BGRA. Wspomniany proces odbywa się w funkcji `prepareImage()` klasy `MarkerDetector`:

```
void MarkerDetector::prepareImage(const cv::Mat& bgraMatrix, cv::Mat&
grayscale) const
{
    cv::cvtColor(bgraMatrix, grayscale, CV_BGRA2GRAY);
}
```

Następnie, odbywa się progowanie (ang. thresholding) obrazu, czyli każdego pikselowi przypisywany jest kolor czarny lub biały, w zależności od wyniku funkcji progowania. Przy okazji tworzonego systemu, do realizacji progowania z pojedynczym progiem zastosowano popularną metodę Otsu (30) autorstwa Nobuyuki Otsu, która dobiera odpowiednią wartość progową na podstawie różnicy między jasnością pikseli w tle oraz pikseli na pierwszym planie. Wynikiem tej funkcji dla piksela, którego jasność przekracza ustaloną wartość progową, będzie kolor biały, a w przeciwnym wypadku czarny. Funkcją OpenCV, która została wykorzystana do realizacji tego zadania jest `cv::threshold()` i jej wywołanie znajduje się w funkcji `performThreshold()` klasy `MarkerDetector`:

```
void MarkerDetector::performThreshold(const cv::Mat& grayscale, cv::Mat&
thresholdImg) const
{
    cv::threshold(grayscale, thresholdImg, 127, 255, cv::THRESH_BINARY_INV
                  | cv::THRESH_OTSU);
}
```

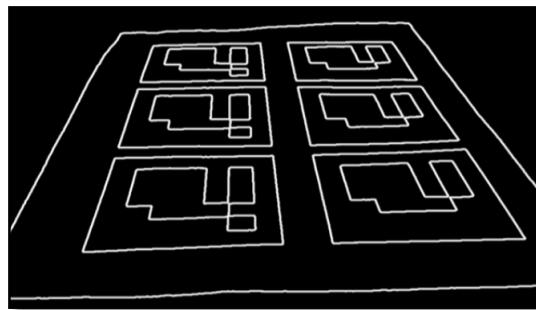


Rys. 4–5 Obraz przed (po lewej) i po binaryzacji (po prawej). Źródło: (31)

Wykrywanie i odfiltrowywanie konturów

Po binaryzacji obrazu, przekazywany jest on na wejście funkcji cv::findContours() biblioteki OpenCV, której rezultatem jest lista wielokątów reprezentujących pojedyncze kontury znalezione na obrazie. Następnie za małe kontury są odrzucane, ponieważ poniżej pewnego rozmiaru markera na obrazie, jego odczyt i tak jest niemożliwy. Poniżej znajduje się definicja funkcji findContours() klasy MarkerDetector, która realizuje wspomniane zadania:

```
void MarkerDetector::findContours(cv::Mat& thresholdImg, ContoursVector&
contours, int minContourPointsAllowed) const
{
    ContoursVector foundContours;
    cv::findContours(thresholdImg, foundContours, CV_RETR_LIST,
        CV_CHAIN_APPROX_NONE);
    contours.clear();
    for (size_t i=0; i<foundContours.size(); i++)
    {
        int contourSize = foundContours[i].size();
        if (contourSize > minContourPointsAllowed)
        {
            contours.push_back(foundContours[i]);
        }
    }
}
```



Rys. 4–6 Odnalezione kontury na obrazie. Źródło: (31).

Odnajdywanie kandydatów

Następnym krokiem po wykryciu konturów jest odnajdywanie kandydatów na znaczniki, za które odpowiedzialna jest funkcja findCandidates() klasy MarkerDetector. Głównymi operacjami przeprowadzanymi na tym etapie są: aproksymacja konturów do wielokątów, odrzucanie mało prawdopodobnych kandydatów, takich jak wielokąty o liczbie wierzchołków większej lub mniejszej niż 4, czy wielokątów niewypukłych. Poniżej znajduje się główny fragment omawianej funkcji:

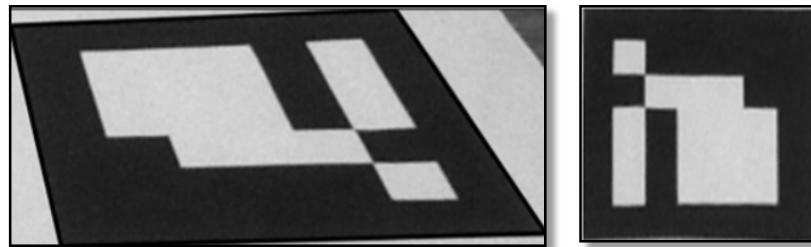
```
...
// Approximate to a polygon
double eps = contours[i].size() * 0.05;
cv::approxPolyDP(contours[i], approxCurve, eps, true);

// Check if it has exactly 4 vertices
if (approxCurve.size() != 4) {
    continue;
}
```

```
// Check if the polygon is convex
if (!cv::isContourConvex(approxCurve)) {
    continue;
}
...
```

Usunięcie perspektywy

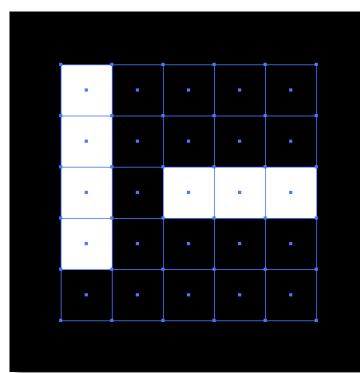
By przykodować kandydatów na znaczniki do próby zdekodowania ich ID, należy najpierw przeprowadzić operację usunięcia perspektywy. Do tego celu służy funkcja biblioteki OpenCV cv::getPerspectiveTransform(), która zwraca macierz transformacji na podstawie dwóch par czterech punktów. Pierwszym argumentem tej funkcji są współrzędne czterech wierzchołków kandydata na znacznik znajdującego się na obrazie, natomiast drugą parę stanowią współrzędne punktów markera na obrazie wzorcowym markera.



Rys. 4–7 Znaczniki przed (po lewej) i po (po prawej) zdjęciu perspektywy. Źródło: (31).

Dekodowanie ID

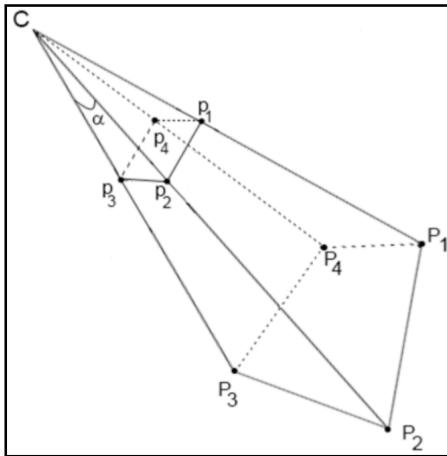
Do zliczenia ilości białych pikseli na danym obrazie można wykorzystać funkcję cv::countNonZero biblioteki OpenCV. Funkcja ta zlicza niezerowe elementy dla danej tablicy dwuwymiarowej - w tym przypadku tablicy kolorów pikseli, w której kolor biały reprezentuje wartość 1. Z kolei funkcja cv::Mat potrafi stworzyć wycinek obrazu z oryginalnego obrazu. Przy użyciu tych metod wycinane są kolejne kwadraciki znacznika, a dla każdego z nich sprawdzany jest stosunek czarnych do białych pikseli. Na tej podstawie konstruowana jest mapa bitowa opisująca kodowanie znacznika. W końcu ID markera jest odkodowywane.



Rys. 4–8 Znacznik z widoczną siatką kwadratów kodujących.

Estymacja położenia

Ostatnim krokiem algorytmu wykrywania znaczników jest przybliżenie ich położenia w przestrzeni 3D względem urządzenia przechwytyującego, na podstawie dwuwymiarowego obrazu. Rezultat tego kroku zostanie później wykorzystany do transformacji wirtualnych elementów wstawianych w miejsce znaczników.



Rys. 4–9 Rzutowanie punktów 3D na 2D. Źródło: (31).

Punkt C oznacza położenie kamery. Punkty P1-P4 to punkty w przestrzeni 3D. Punkty p1-p4 to punkty powstałe w wyniku rzutowania P1-P4 na płaszczyznę obrazu kamery.

Jedną z czynności, które należy wykonać przed estymacją położenia jest poprawa dokładności wykrytych konturów dla każdego z markerów. Można to zrealizować przy pomocy funkcji cv::cornerSubPix() biblioteki OpenCV. Operacja ta jest dużo bardziej kosztowna od samego wykrywania konturów, dlatego właśnie jest wykonywana dopiero na tym etapie.

Drugą potrzebną czynnością jest pozyskanie wewnętrznych parametrów urządzenia przechwytyującego, czyli w naszym wypadku tylnej kamery urządzenia iOS, takich jak pole widzenia czy ogniskowa. Biblioteka OpenCV daje możliwość przeprowadzenia kalibracji urządzenia, ale możliwe jest również samodzielne wyznaczenie owych parametrów korzystając z opisanej w rozdziale 2.2.3 biblioteki systemowej AVFoundation. W pracy wykorzystano drugie podejście, ponieważ pozwala na dynamiczne wyznaczenie parametrów podczas działania aplikacji. Poniższy fragment przedstawia funkcję cameraParametersForCurrentFormat() realizującą to zadanie:

```
func cameraParametersForCurrentFormat() -> (fx: Float, fy: Float, cx: Float, cy: Float) {
    let format = self.cameraDevice.activeFormat
    let dim = CGSize(width: UIScreen.mainScreen().bounds.width * UIScreen.mainScreen().scale, height: UIScreen.mainScreen().bounds.height * UIScreen.mainScreen().scale)
    let cx = Float(dim.width) / 2.0
    let cy = Float(dim.height) / 2.0
    let HFOV = format.videoFieldOfView
    let VFOV = ((HFOV)/cx)*cy
    let fx = abs(Float(dim.width) / (2.0 * tan(HFOV / 180.0 * Float(M_PI) / 2.0)))
    let fy = abs(Float(dim.height) / (2.0 * tan(VFOV / 180.0 * Float(M_PI) / 2.0)))
}
```

```

        return (fx, fy, cx, cy)
}

```

Znając dokładne współrzędne wierzchołków znaczników na obrazie dwuwymiarowym, oraz wewnętrzne parametry kamery, skorzystano z funkcji cv::solvePnP() biblioteki OpenCV w celu odnalezienia transformacji przestrzeni euklidesowej, składającą się z rotacji oraz translacji, między znacznikiem a urządzeniem rejestrującym obraz. Funkcja cv::solvePnP() znajduje położenie kamery, dlatego wynikowa macierz transformacji jest na końcu odwracana, by w ten sposób uzyskać informację o położeniu markera. Poniżej znajduje się definicja funkcji estimatePosition() klasy MarkerDetector, która realizuje wspomniane zadania:

```

void MarkerDetector::estimatePosition(std::vector<Marker>& detectedMarkers)
{
    for (size_t i=0; i<detectedMarkers.size(); i++)
    {
        Marker& m = detectedMarkers[i];

        // Find transformation
        cv::Mat Rvec;
        cv::Mat_<float> Tvec;
        cv::Mat raux,taux;
        cv::solvePnP(m_markerCorners3d, m.points, camMatrix, distCoeff,
                     raux, taux);

        // Get the rotation vector
        raux.convertTo(Rvec,CV_32F);
        Rvec.at<float>(0,2) = -Rvec.at<float>(0,2);
        Rvec.at<float>(0,1) = -Rvec.at<float>(0,1);

        // Get the translation vector
        taux.convertTo(Tvec ,CV_32F);
        Tvec.at<float>(0,0) = -Tvec.at<float>(0,0);

        // Construct a rotation matrix
        cv::Mat_<float> rotMat(3,3);
        cv::Rodrigues(Rvec, rotMat);

        // Copy to transformation matrix
        for (int col=0; col<3; col++)
        {
            for (int row=0; row<3; row++)
            {
                m.transformation.r().mat[row][col] = rotMat(row,col);
            }
            m.transformation.t().data[col] = Tvec(col);
        }

        // Invert the transformation to get marker pose, instead of the
        // camera pose
        m.transformation = m.transformation.getInverted();
    }
}

```

4.4. Wstawianie wirtualnych przedmiotów

Znając przybliżone położenie markerów w przestrzeni 3D, w stosunku do kamery, można wstawić wirtualne przedmioty w miejsce znaczników. W poniższym rozdziale zostaną opisane główne kroki potrzebne do realizacji tego etapu tworzenia systemu AR, takie jak przygotowanie modeli 3D obiektów wirtualnych, renderowanie ich w miejsce znaczników, oraz dodanie możliwości interakcji z nimi. Głównym narzędziem, które jest wykorzystywane na tym etapie jest opisany w rozdziale 2.2.4 framework SceneKit systemu iOS.

4.4.1. Przygotowanie modeli 3D

Żeby móc w kolejnym kroku wstawić wirtualne obiekty do obrazu, przygotowano najpierw ich modele 3D. Silnik SceneKit oferuje pełne wsparcie dla popularnego formatu COLLADA (.dae), dlatego skorzystano z niego w niniejszej pracy. Po przygotowaniu modelów 3D obiektów w tym formacie, dodano je do projektu. Następnie zaimplementowano tworzenie węzłów sceny na ich podstawie, korzystając z możliwości utworzenia obiektu typu SCNScene w oparciu o plik modelu i wyciągnięciu z niej szukanego obiektu typu SCNNode. Realizacja tego zadania opiera się o następujące dwie linijki kodu:

```
// Load Collada models (DAE)
let modelScene = SCNScene(named: modelName + ".dae")!
let modelNode = modelScene.rootNode.childNodes.first!
```

Wczytywane w ten sposób modele zawierają informacje o wykorzystywanych przez nie teksturach, mapach normalnych itp., które również dodano do projektu aplikacji.

4.4.2. Wstawianie obiektów

W celu wstawienia w ten sposób przygotowanych modeli 3D wirtualnych przedmiotów w miejsce znaczników, najpierw przygotowano obiekt typu SCNRenderer odpowiedzialny za renderowanie sceny, w oparciu o podany przy jego tworzeniu kontekst OpenGL:

```
self.renderer = SCNRenderer(context: self.glContext!, options: nil)
```

Następnie, przygotowano scenę do renderowania elementów rozszerzonej rzeczywistości w funkcji prepareARScene(_) klasy ARRender. Na tym etapie do sceny dodano: kamerę, światło, oraz celownik na środku widoku, który jest potrzebny do późniejszej realizacji obsługi interakcji. Poniżej znajduje się definicja wspomnianej funkcji:

```
func prepareARScene() -> SCNScene {
    let scene = SCNScene()

    // create and add a camera to the scene
    let cameraNode = SCNNode()
    cameraNode.camera = SCNCamera()
    cameraNode.camera!.zNear = 1
    cameraNode.camera!.zFar = 100
    scene.rootNode.addChildNode(cameraNode)
    self.cameraNode = cameraNode
    self.renderer?.pointOfView = cameraNode

    // Add light
```

```

let lightNode = SCNNode()
let light = SCNLight()
light.type = SCNLightTypeOmni
light.color = UIColor.whiteColor()
lightNode.light = light
lightNode.position = SCNVector3Make(0, 0, 0)
scene.rootNode.addChildNode(lightNode)

// Add an aim
let overlaySKScene = SKScene(size: CGSize(width: self.screenRect.width,
    height: self.screenRect.height))
self.renderer?.overlaySKScene = overlaySKScene
let aimNode = SKShapeNode(circleOfRadius: 8)
aimNode.position = CGPointMake(overlaySKScene.frame.width/2.0,
    overlaySKScene.frame.height/2.0)
aimNode.lineWidth = 5
aimNode.strokeColor = UIColor.blackColor().colorWithAlphaComponent(0.7)
self.renderer?.overlaySKScene?.addChild(aimNode)

return scene
}

```

W końcu, w funkcji renderARElementsToContext(markers:frameTime:) klasy ARRenderer w miejsce znaczników wstawiane są modele. W zależności od tego, czy znacznik pojawił się na obrazie po raz pierwszy, zmienił tylko swoje położenie, lub zniknął z pola widzenia, odpowiadające im modele są dodawane, przemieszczane, lub usuwane ze sceny. Dodatkowo, każdy z modeli poddawany jest transformacji, która została wyznaczona dla każdego ze znaczników na wcześniejszym etapie estymacji położenia. Pod koniec scena renderowana jest do kontekstu OpenGL przy pomocy funkcji renderAtTime(_) biblioteki SceneKit. Poniżej znajduje się definicja funkcji renderARElementsToContext(markers:frameTime:) klasy ARRenderer, która realizuje wspomniane zadania:

```

func renderARElementsToContext(markers markers: [MarkerWrapper], frameTime: CFTimeInterval) {

    let knownMarkers: [MarkerWrapper] = markers.filter({ $0.modelType != nil })

    // Add/modify present markers
    for marker in knownMarkers {

        // Create a marker node for the detected marker, if not there
        let markerNode: SCNNode
        if self.markerNodeByID[Int(marker.markerID)] == nil {

            // Create the node
            markerNode = self.templateMarkerNode.flattenedClone()
            self.markerNodeByID[Int(marker.markerID)] = markerNode

            // Add virtual object model appropriate for the marker type
            if let modelNode = self.modelDesigner
                .currentModelNodeForType(marker.modelType!) {

                markerNode.addChildNode(self.modelDesigner.
                    woodenFloorNode.flattenedClone())
                markerNode.addChildNode(ModelNode(node: modelNode, type:
                    marker.modelType!))
            }
        }
    }
}

```

```

        }
    }
    else {
        markerNode = self.markerNodeByID[Int(marker.markerID)]!
    }

    // Add node to the scene if not there
    if markerNode.parentNode != self.renderer?.scene?.rootNode {
        self.renderer?.scene?.rootNode.addChildNode(markerNode)
    }

    markerNode.transform =
        SCNMatrix4FromGLKMatrix4(marker.transformationMatrix)
    self.currentMarkerTransform = markerNode.transform
}

// Remove marker nodes for absent markers
for (_, value) in self.markerNodeByID.enumerate() {
    let (markerID, markerNode) = value
    if !markers.contains({ Int($0.markerID) == markerID }) {
        markerNode.removeFromParentNode()
    }
}

self.renderer!.renderAtTime(frameTime)
}

```

4.4.3. Obsługa interakcji

Dla dodanych w miejsce znaczników wirtualnych przedmiotów dodano podstawową obsługę interakcji, która polega na możliwości zmiany wariantu modelu obiektu po stuknięciu w ekran, pod warunkiem, że celownik nakierowany jest na ten obiekt. Skorzystano w tym celu z mechanizmu rozpoznawania gestów systemu iOS.

Najpierw, dodano obsługę gestu stuknięcia w ekran, która ma się odbywać przez wywołanie funkcji `tapGesture(_)` klasy ARRenderer po każdorazowym stuknięciu w ekran:

```
self.glView.addGestureRecognizer(UITapGestureRecognizer(target: self,
action: #selector(ARRender.tapGesture(_))))
```

Następnie, w obrębie funkcji `tapGesture(_)` zaimplementowano odnajdywanie węzła związanego z modelem obiektu wirtualnego, który został przecięty przez promień testowy w wyniku zastosowania funkcji `hitTest(_:options:)`, której działanie zostało opisane w rozdziale 2.2.4. W tym miejscu dodano również mechanizm zmiany wariantu modelu dla danego znacznika. Poniżej znajduje się pełna definicja funkcji `tapGesture(_)` klasy ARRenderer:

```
@IBAction func tapGesture(recognizer:UITapGestureRecognizer) {

    // Get hit test results
    let results =
        self.renderer!.scene!.rootNode.hitTestWithSegmentFromPoint(
            SCNVector3Make(0, 0, 0),
            toPoint: SCNVector3Make(0, 0, -self.zFar),
            options: nil)

    // Find first virtual object node that was hit
}
```

```

if let modelNodeResult =
    results.filter({ $0.node is ModelNode }).first {

    let modelNode = modelNodeResult.node as! ModelNode
    self.modelDesigner.changeModelVariantForType(modelNode.modelType)

    // Change the virtual object node to the other variant
    modelNode.removeFromParentNode()
    self.markerNodeByID[modelNode.modelType.rawValue]?
        .addChildNode(self.modelDesigner.currentModelNodeForType(
            modelNode.modelType)!)
}

}
}

```

4.5. Wyświetlanie obrazu

Ostatnim etapem tworzenia obrazu rozszerzonej rzeczywistości przez system AR jest połączenie obrazu rzeczywistego z nałożonymi elementami świata wirtualnego, oraz wyświetlenie go w odpowiedni sposób. Zgodnie z założeniami tworzony system AR implementuje dwa tryby wyświetlania obrazu wynikowego: mono i stereo.

Klatki obrazu przygotowane zgodnie z opisem w rozdziale 4.2 przekazywane są do funkcji render(cameraFrame:markers:) klasy ARRender, która jest odpowiedzialna za łączenie i wyświetlanie obrazu rzeczywistego oraz elementów wirtualnych. Poniżej znajduje się jej definicja:

```

func render(cameraFrame cameraFrame: CIImage, markers: [MarkerWrapper]) {
    objc_sync_enter(self)

    let frameTime = CFAbsoluteTimeGetCurrent()

    // Prepare AR enhanced frame (off-screen)
    self.clearGlBuffer()
    self.glView.bindDrawable()
    self.renderImageToContext(cameraFrame)
    self.renderARElementsToContext(markers: markers, frameTime: frameTime)
    var arEnhancedFrame = EAGLContext.imageWithCurrentContext(fromRect:
self.screenRect)

    // Process the frame
    if self.stereoOn {
        self.applyDistortionToFrame(&arEnhancedFrame, padding: 140)
    }

    // Render processed frame (on-screen)
    self.clearGlBuffer()
    self.renderImageToContext(arEnhancedFrame, stereo: self.stereoOn)

    objc_sync_exit(self)

    dispatch_async(dispatch_get_main_queue()) {
        self.glView.display()
    }
}

```

Najpierw odbywa się renderowanie poza ekranem (bez wyświetlania, ang. off-screen rendering) klatki obrazu z kamery, co realizuje funkcja `renderImageToContext(_)` klasy `ARRender`, której działanie zostanie dokładniej opisana w dalszych rozdziałach, ze względu na wsparcie dla różnych trybów wyświetlania.

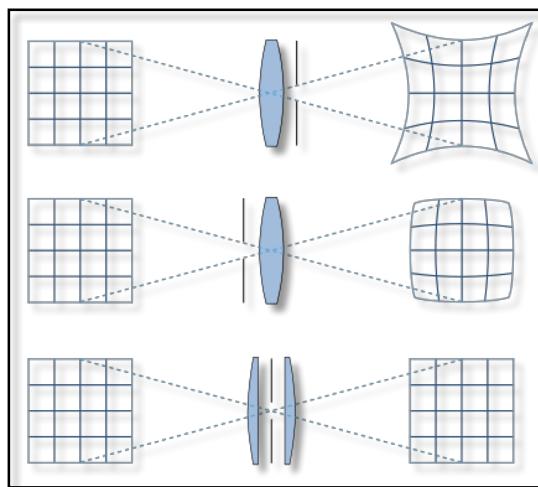
Następnie, przy użyciu tego samego kontekstu, renderowana jest scena AR zgodnie z opisem w rozdziale 4.4.2. Po tym kroku, tworzona jest klatka na podstawie bieżącego kontekstu renderowania, która przedstawia kompletny obraz rozszerzonej rzeczywistości.

W końcu, mając do dyspozycji klatkę AR, w zależności od aktywnego trybu wyświetlania, nakładane są odpowiednie przekształcenia i klatka jest wyświetlana.

4.5.1. Tryby mono i stereo

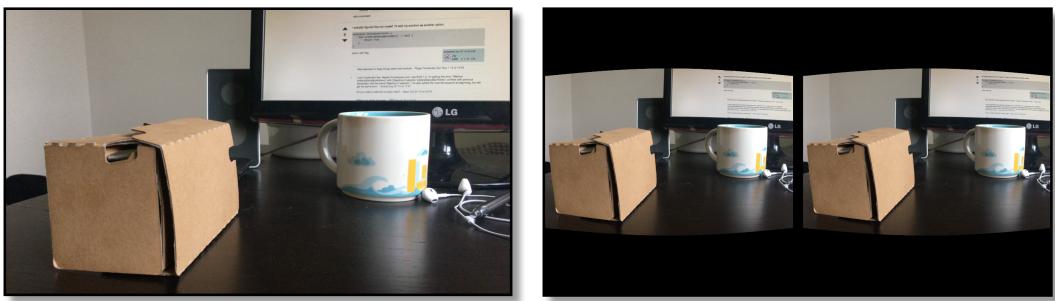
W trybie pojedynczego obrazu - mono, klatka obrazu dopasowywana jest do widoku, w którym jest wyświetlana, w ten sposób, że wypełnia on całą dostępną przestrzeń przy jednoczesnym zachowaniu proporcji obrazu.

Natomiast w przypadku trybu stereo, obraz renderowany jest dwukrotnie - raz dla prawego oka, raz dla lewego oka, w tym samym rozmiarze lecz w innych miejscach. Przy próbie podglądu wyświetlonego w ten sposób obrazu przez okulary VR, uwidacznia się wada optyczna zwana dystorsją, którą powodują soczewki okularów. Wada ta polega na różnym powiększeniu obrazu w różnych odległościach od osi optycznej okularów i wyróżnia się dwa jej rodzaje: zmieniającą kształt prostokątnego obrazu w beczkę (dystorsja beczkowa), oraz w poduszkę (dystorsja poduszkowa). W przypadku okularów VR występuje dystorsja poduszkowa, więc w tworzonym systemie obrazy są celowo zniekształcone za pomocą przeciwnego do niej zniekształcenia - dystorsji beczkowej, by w efekcie uzyskać prawidłowy obraz.



Rys. 4–10 Rodzaje dystorsji.

Dystorsja poduszkowa (u góry), beczkowa (po środku), brak dystorsji (na dole).



Rys. 4–11 Wyświetlany obraz w trybie mono (po lewej) oraz stereo (po prawej).

Poniżej przedstawiona jest pełna implementacja funkcji renderImageToContext(_:stereo:) klasy ARRenderer, odpowiedzialnej za wyświetlanie obrazu w wybranym trybie:

```
func renderImageToContext(image: CIImage, stereo: Bool = false) {
    if stereo {
        let eyeViewPortSize = CGSizeMake(self.screenRect.width * 0.6,
                                         self.screenRect.height * 0.49)
        let leftEyePosition = CGPointMake(
            self.screenRect.width * 0.5 - eyeViewPortSize.width/2.0,
            (self.screenRect.height - (eyeViewPortSize.height * 2.0)) +
            eyeViewPortSize.height
        )
        let rightEyePosition = CGPointMake(
            self.screenRect.width * 0.5 - eyeViewPortSize.width/2.0,
            0
        )
        let fittingImage = image.imageByFittingToSize(eyeViewPortSize)
        self.ciContext!.drawImage(
            fittingImage,
            inRect: CGRectMake(
                leftEyePosition.x,
                leftEyePosition.y,
                fittingImage.extent.width,
                fittingImage.extent.height
            ),
            fromRect: fittingImage.extent
        )
        self.ciContext!.drawImage(
            fittingImage,
            inRect: CGRectMake(
                rightEyePosition.x,
                rightEyePosition.y,
                fittingImage.extent.width,
                fittingImage.extent.height
            ),
            fromRect: fittingImage.extent
        )
    } else {
        let fittingImage = image.imageByFittingToSize(
            CGSizeMake(width: self.screenRect.width, height:
                      self.screenRect.height))
        self.ciContext!.drawImage(fittingImage, inRect: fittingImage.extent,
                               fromRect: fittingImage.extent)
    }
}
```

Natomiast funkcja `applyDistortionToFrame(_:padding:)` klasy `ARRenderer` służy do celowego zniekształcania obrazu za pomocą dystorsji beczkowej:

```
func applyDistortionToFrame(inout frame: CIImage, padding: CGFloat = 0) {  
    let originalFrameExtent = frame.extent  
    frame = frame  
        .imageByApplyingFilter("CIBumpDistortion", withInputParameters: [  
            kCIInputCenterKey: CIVector(  
                x: self.screenRect.width/2.0,  
                y: self.screenRect.height/2.0  
            ),  
            kCIInputRadiusKey: NSNumber(double: 2000),  
            kCIInputScaleKey: NSNumber(double: 0.3)  
        ]  
    )  
        .imageByCroppingToRect(CGRectMake(-padding,-  
padding,originalFrameExtent.width + (2*padding), originalFrameExtent.height  
+ (2*padding)))  
        .imageByApplyingTransform(CGAffineTransformMakeTranslation(padding,  
padding))  
}
```

4.6. Testowanie rozwiązania

4.6.1. Testowanie wykrywania znaczników

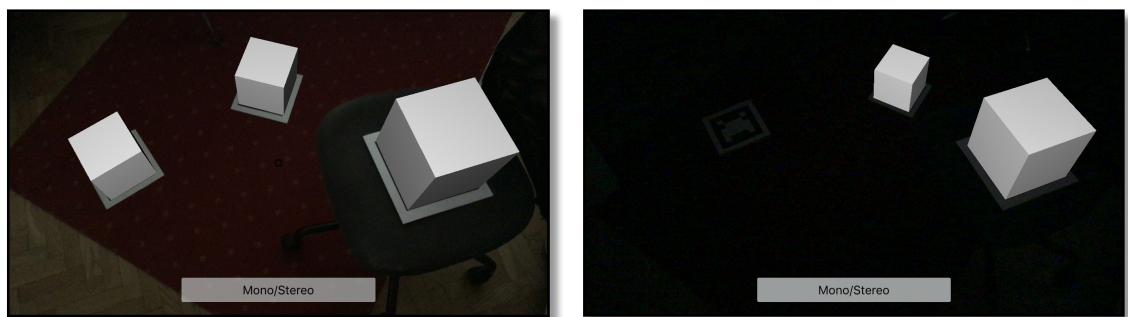
Oświetlenie

Zachowanie implementowanego w systemie algorytmu wykrywania znaczników przetestowano pod względem jego działania w różnych warunkach oświetleniowych:

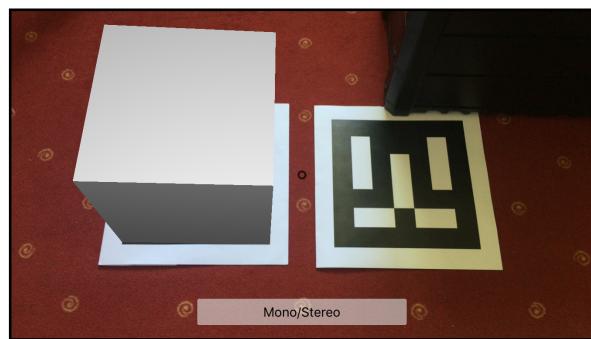
- naturalne oświetlenie
- sztuczne oświetlenie
- słabe oświetlenie
- nieregularne oświetlenie
- różne kąty padania promieni słonecznych



Rys. 4–12 Test wykrywania znaczników przy naturalnym (po lewej) oraz sztucznym (po prawej) oświetleniu.



Rys. 4–13 Test wykrywania znaczników przy słabym (po lewej) i bardzo słabym (po prawej) oświetleniu.



Rys. 4–14 Test wykrywania znaczników przy nieregularnym oświetleniu.



Rys. 4–15 Test wykrywania znaczników przy różnych kątach padania światła.

Na podstawie otrzymanych wyników testów można wyciągnąć wniosek, że poprawnemu działaniu systemu sprzyja zarówno oświetlenie naturalne jak i sztuczne (Rys. 4–12). Zauważono również, że wykrywanie znaczników działa dobrze przy słabym oświetleniu,

a trudności pojawiają się dopiero przy bardzo słabym oświetleniu (Rys. 4–13). Wyraźną przeszkodę stanowią jednak: kąt padania światła taki, że duża ilość promieni światła odbitego wpada do kamery (Rys. 4–15), oraz nieregularne oświetlenie, na przykład gdy znacznik jest częściowo zacieniony (Rys. 4–14).

Warto zwrócić ponownie uwagę na fakt, że mimo bardzo słabego, lecz równomiernego oświetlenia, wykrywane są aż 2 na 3 znaczniki. Dzieje się tak prawdopodobnie za sprawą algorytmu Otsu wykorzystywanego w tworzonym systemie na etapie binaryzacji obrazu, który pozwala na dynamiczne dopasowanie wartości progowej w oparciu o różnicę jasność pikseli obrazu.

Odległość od znacznika

Zweryfikowano również jakość rozwiązań pod kątem zasięgu działania algorytmu wykrywania znaczników - maksymalnej i minimalnej odległości obserwatora od znacznika.



Rys. 4–16 Test zasięgu wykrywania znaczników.

Na podstawie przeprowadzonych testów zauważono, że dla znacznika o wymiarach 7cm x 7cm, przy korzystnych warunkach oświetleniowych, maksymalny zasięg wykrywania znaczników wynosi 3 metry, a płynne działanie (bez migotania) jest zapewnione w odległości mniejszej lub równej 2,5 metra. Z drugiej strony minimalną odległość wymaganą do wykrycia znacznika jest odległość, która pozwala go jeszcze ująć w całości na obrazie.

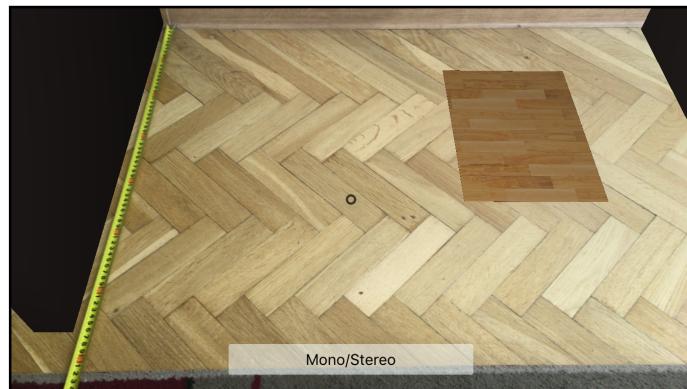
4.6.2. Testowanie modeli 3D

Wstawiane w miejsce znaczników modele 3D obiektów wirtualnych przetestowano pod kątem odzwierciedlenia ich rzeczywistych wymiarów na wizualizacji. W tym celu sprawdzono

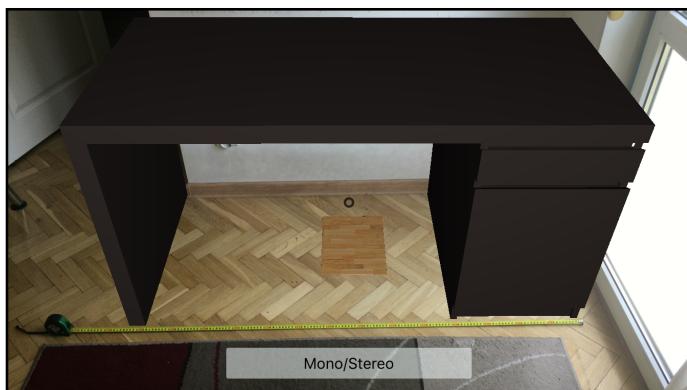
czy wstawiane modele mebli IKEA (32) mają podobne rozmiary do tych określonych w oficjalnej dokumentacji IKEA.



Rys. 4–17 Pomiar wysokości wirtualnego biurka.



Rys. 4–18 Pomiar głębokości wirtualnego biurka.

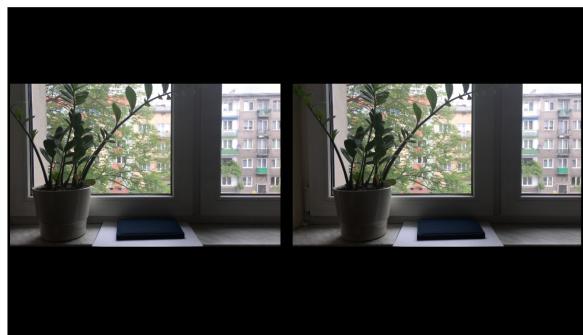


Rys. 4–19 Pomiar szerokości wirtualnego biurka.

Na podstawie wykonanych pomiarów zauważono, że wymiary mebli wirtualnych tylko nieznacznie odbiegają od wymiarów rzeczywistych i odwzorowanie jest wystarczająco dokładne do wizualizacji umebli wirtualnych. Przykładowo, wymiary wirtualnego biurka wstawionego w miejsca znacznika wynoszą: 142 cm szerokości, 66 cm głębokości, oraz 74 cm wysokości, podczas gdy wymiary tego modelu biurka według dokumentacji (33) wynoszą: 140 cm szerokości, 65 cm głębokości, oraz 73 cm wysokości.

4.6.3. Testowanie trybu stereo

W przypadku realizacji trybu stereo wyświetlania obrazu rozszerzonej rzeczywistości, parametry dystorsji wyznaczono eksperymentalnie. W wyniku wykonanych testów zauważono, że zastosowanie filtra dystorsji beczkowej o określonym promieniu i skali, dla każdego z obrazów odpowiadających oczom z osobna, pozwala na wyeliminowanie niepożądanego efektu spowodowanego przez zastosowanie okularów VR do podglądu.



Rys. 4–20 Prawidłowy obraz, bez zniekształceń.



Rys. 4–21 Efekt dystorsji poduszkowej powstały w wyniku podglądu obrazu przez soczewki okularów VR.



Rys. 4–22 Efekt dystorsji poduszkowej obrazu powstały w wyniku celowego zniekształcenia obrazu.

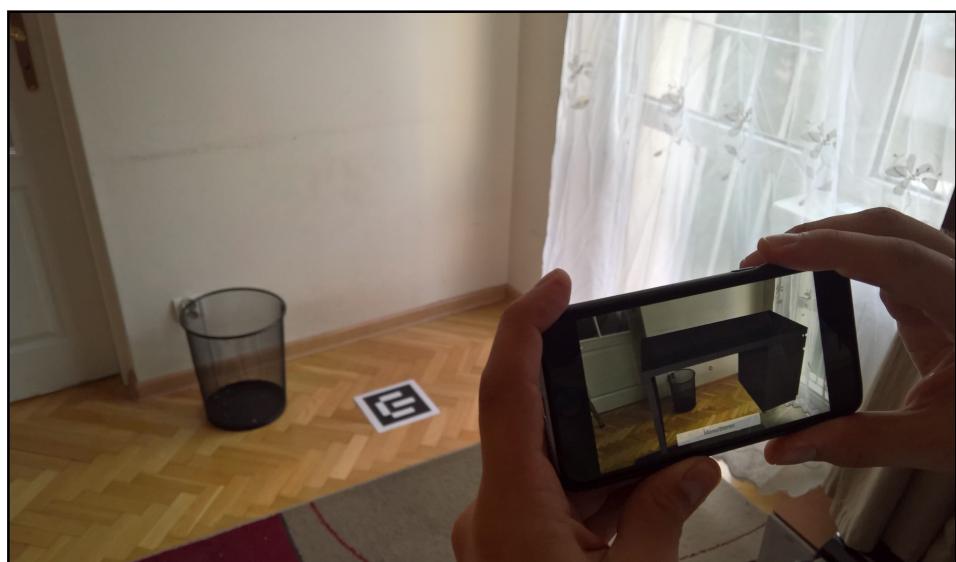
W wyniku przeprowadzonych testów udało się wyznaczyć parametry dystorsji, które w dużym stopniu (lecz nie całkowicie) eliminują wadę optyczną spowodowaną przez stosowanie okularów VR. Co prawda wciąż pozostają częściowe zniekształcenia po bokach obrazu, lecz mimo to zastosowanie tego rozwiązania pozwala na znaczącą poprawę jakości tworzonego systemu AR.

5. Aplikacja do wizualizacji wnętrz

Aplikacja ta służy do wizualizacji wnętrz pomieszczeń w czasie rzeczywistym i stanowi przykład praktycznego zastosowania tworzonego w ramach pracy systemu AR. Jej działanie polega na wstawianiu modelów 3D mebli w miejsca przygotowanych wcześniej znaczników.

Enfurio charakteryzuje również możliwość podglądu wygenerowanego obrazu na ekranie urządzenia iOS w dwóch trybach: pojedynczego obrazu, oraz pary obrazów - w przypadku stosowania okularów VR, dzięki czemu możliwy jest spacer po wypełnionym wirtualnymi meblami pokoju.

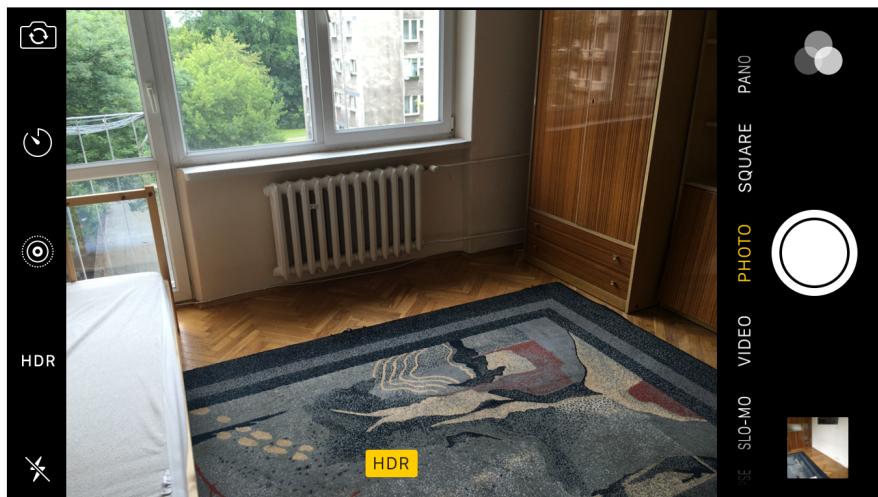
Ponadto, aplikacja pozwala na zmianę wariantu wstawianego typu mebla poprzez nakierowanie na dany mebel celownikiem wyświetlonym na ekranie, a następnie stuknięciu w ekran.



Rys. 5–1 Stworzona aplikacja Enfurio do wizualizacji wnętrz pomieszczeń.

Do przygotowania aplikacji wykorzystano stworzony w ramach niniejszej pracy system wizualizacji rozszerzonej rzeczywistości, który zaprojektowano w taki sposób, żeby potrzebne modyfikacje systemu obejmowały tylko trzy zagadnienia:

- przygotowania modeli 3D mebli
- przygotowania znaczników
- dodania powiązania między znacznikami i odpowiadającymi im modelami



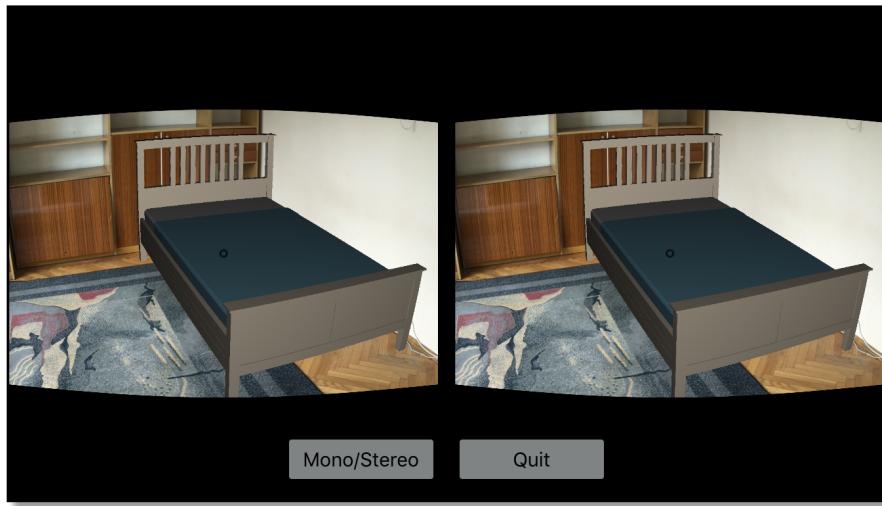
Rys. 5–2 Widok pomieszczenia przy podglądzie obrazu z kamery telefonu.



Rys. 5–3 Widok pomieszczenia generowany przez Enfurio, w trybie pojedynczego obrazu, oraz wariancie A mebli.



Rys. 5–4 Widok pomieszczenia generowany przez Enfurio, w trybie pojedynczego obrazu, oraz wariancie B mebli.

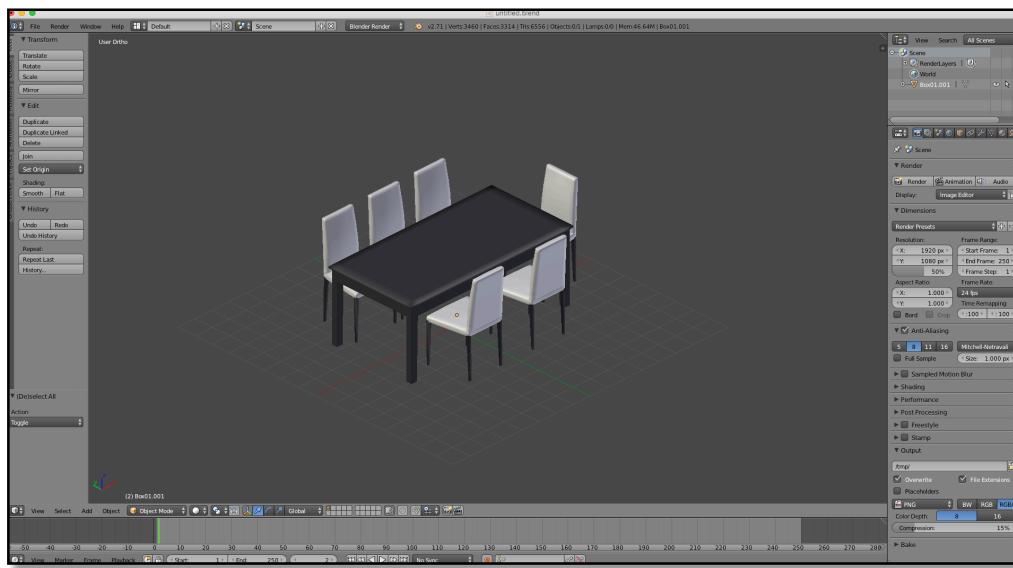


Rys. 5–5 Widok pomieszczenia generowany przez Enfurio, w trybie okularów VR.

5.1.1. Przygotowanie modeli 3D

Wstawiane w miejsce znaczników elementy grafiki 3D zostały pobrane ze strony (32) udostępniającej odpowiednio przygotowane (odzwierciedlające rzeczywiste wymiary) modele mebli 3D, który są charakterystyczne dla firmy IKEA.

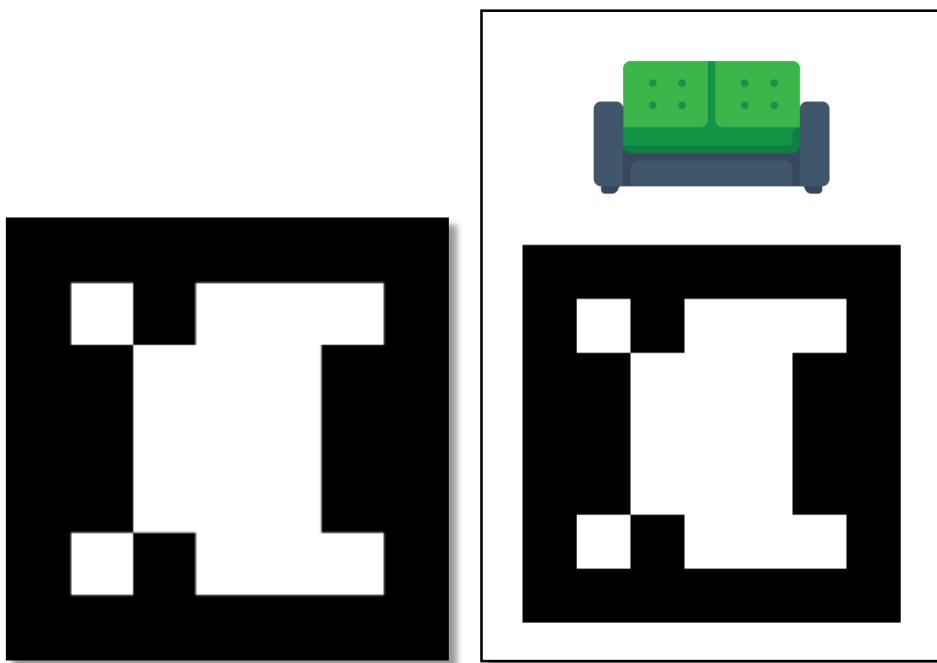
Następnie, pobrane meble poddano przeskalowaniu, wyśrodkowaniu, oraz konwersji do formatu COLLADA (.dae) korzystając z narzędzia open-source do pracy z grafiką 3D o nazwie Blender (34). Po tym, powiązane z modelami pliki .dae oraz tekstury dodano do projektu aplikacji.



Rys. 5–6 Przykładowy model 3D mebli, w programie Blender.

5.1.2. Przygotowanie znaczników

Po przygotowaniu modeli 3D mebli, zaprojektowano trzy znaczniki, w oparciu o kodowanie Hamminga opisane w rozdziale 2.4.3. Dla każdego z markerów dodatkowo sporządzono symboliczną oprawę graficzną.



Rys. 5–7 Przykładowy zaprojektowany znacznik (po lewej), oraz jego wersja z pełną oprawą graficzną (po prawej).

5.1.3. Powiązanie znaczników z modelami

W kolejnym kroku powiązano modele 3D mebli z odpowiadającymi im znacznikami. W pierwszym etapie dodano nowe przypadki do typu wyliczeniowego ModelType:

```
enum ModelType: Int {
    case Wardrobe = 509
    case Bed = 305
    case Desk = 787

    init?(markerID: Int) {
        guard let type = ModelType(rawValue: markerID) else { return nil }
        self = type
    }

    static let availableValues = [Wardrobe, Bed, Desk]
}
```

W drugim etapie załadowano odpowiednie modele do systemu poprzez dodanie wywołania funkcji loadModelNodesFromModels(_:modelName:) klasy ModelDesigner, która jako argumenty przyjmuje typ modelu oraz nazwy modeli, które powinny zostać do danego typu przypisane. Poniżej znajdują się przykładowe wywołania omawianej funkcji:

```
self.loadModelNodesFromModels(markerType: ModelType.Wardrobe, modelName:
    ["IKEA_wardrobe_ODDA", "IKEA_wardrobe_HEMNES"])
self.loadModelNodesFromModels(markerType: ModelType.Desk, modelName:
    ["IKEA_desk_MALM", "IKEA_desk_HEMNES"])
self.loadModelNodesFromModels(markerType: ModelType.Bed, modelName:
    ["IKEA_bed_MALM", "IKEA_bed_HEMNES"])
```

6. Podsumowanie

W pracy przedstawiono proces tworzenia systemu wizualizacji rozszerzonej rzeczywistości w czasie rzeczywistym, opierający swoje działanie na algorytmie wykrywania znaczników, działającym na urządzeniach z systemem iOS. Proces ten został opisany począwszy od pozyskiwania obrazu z kamery urządzenia mobilnego, przez algorytm wykrywania znaczników zaimplementowany w oparciu o bibliotekę OpenCV, po samodzielne wyświetlanie obrazu.

Pokazano również praktyczne zastosowanie stworzonego systemu AR jako aplikacji do wizualizacji wnętrz pomieszczeń, co w połączeniu z możliwością podglądu obrazu przez okulary VR w czasie rzeczywistym, pozwoliło osiągnąć w stopniu zadowalającym zamierzony cel pracy - wysoką immersyjność rozwiązania.

Autor pracy dostrzega perspektywy dalszego rozwoju systemu, zarówno poprzez wydzielenie go jako natywnej biblioteki AR dla systemu iOS, jak i wprowadzenie usprawnień oraz dodanie nowych funkcjonalności. Na przykład interakcja użytkownika z modelami mogłaby zostać wzbogacona o możliwość dostosowania położenia i obrotu wstawianego modelu, algorytm wykrywania znaczników mógłby zostać usprawniony, tak by nierównomierne oświetlenie nie stanowiło tak dużego utrudnienia w pracy systemu jak obecnie, a same modele mogłyby być wybierane przez użytkownika bezpośrednio z katalogu producenta mebli.

7. Bibliografia

1. **Darwin Karol.** *O powstawaniu gatunków drogi doboru naturalnego*. Warszawa : Wydawnictwo Uniwersytetu Warszawskiego, 2013.
2. **Oculus VR, LLC.** Oculus Rift. *Oculus*. [Online] 2016. <https://www.oculus.com/en-us/rift/>.
3. **Wikimedia Foundation Inc.** Augmented reality. *Wikipedia, the free encyclopedia*. [Online] 2016. https://en.wikipedia.org/wiki/Augmented_reality#Software_and_algorithms.
4. **Nazarian Robert.** Google Glass coming to AT&T stores. *TalkAndroid.com*. [Online] 2014. <http://www.talkandroid.com/209434-google-glass-coming-to-att-stores/>.
5. **HTC Corporation.** Product Hardware. *Vive*. [Online] 2016. <https://www.htcvive.com/eu/>.
6. **Sawh Michael.** Life through a smart contact lens. *Wareable*. [Online] 2016. <http://www.wearable.com/wearable-tech/best-smart-contact-lens>.
7. **Avegant Corp.** Product. *Avegant Glyph*. [Online] 2016. <https://www.avegant.com/product>.
8. **Apple Inc.** Objective-C. *iOS Developer Library*. [Online] 2015. <https://developer.apple.com/library/ios/documentation/General/Conceptual/DevPedia-CocoaCore/ObjectiveC.html>.
9. —. About Swift. *Swift.org*. [Online] 2016. <https://swift.org/about/>.
10. —. About the iOS Technologies. *iOS Developer Library*. [Online] 2014. <https://developer.apple.com/library/ios/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/Introduction/Introduction.html>.
11. **Koneva Anna.** Interoperating Between C++ and Objective-C. *Dr. Dobb's*. [Online] 2014. <http://www.drdobbs.com/cpp/interoperating-between-c-and-objective-c/240165502>.
12. **Stellingwerff Adriaan.** Capturing Video on iOS. *objc.io*. [Online] 2015. <https://www.objc.io/issues/23-video/capturing-video/>.
13. **Apple Inc.** About AVFoundation. *iOS Developer Library*. [Online] 2015. https://developer.apple.com/library/ios/documentation/AudioVideo/Conceptual/AVFoundationPG/Articles/00_Introduction.html.
14. —. Still and Video Media Capture. *iOS Developer Library*. [Online] 2015. https://developer.apple.com/library/ios/documentation/AudioVideo/Conceptual/AVFoundationPG/Articles/04_MediaCapture.html.
15. —. SceneKit Framework Reference. *iOS Developer Library*. [Online] 2016. https://developer.apple.com/library/ios/documentation/SceneKit/Reference/SceneKit_Framework/.
16. **Rönnqvist David.** Scene Kit. *objc.io*. [Online] 2014. <https://www.objc.io/issues/18-games/scenekit/>.
17. **Apple Inc.** SCNCamera Class Reference. *iOS Developer Library*. [Online] 2016. https://developer.apple.com/library/ios/documentation/SceneKit/Reference/SCNCamera_Class/.
18. **OpenCV Developers Team.** ABOUT | OpenCV. [Online] <http://opencv.org/about.html>.
19. —. Introduction - OpenCV. [Online] <http://docs.opencv.org/2.4/modules/core/doc/intro.html>.
20. —. DOWNLOADS | OpenCV. [Online] <http://opencv.org/downloads.html>.

21. **Apple Inc.** iOS Frameworks. *iOS Developer Library*. [Online] 2014.
<https://developer.apple.com/library/ios/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/iPhoneOSFrameworks/iPhoneOSFrameworks.html>.
22. **Fiala Mark.** Proc. IEEE Conf. on Computer Vision and Pattern Recognition. brak miejsca : IEEE, 2005.
23. **Wikimedia Foundation Inc.** Kod Hamminga. *Wikipedia, wolna encyklopedia*. [Online] 2016.
https://pl.wikipedia.org/wiki/Kod_Hamminga.
24. **DAQRI.** About ARToolKit. *ARToolKit.org*. [Online] 2016. <http://artoolkit.org>.
25. **PTC Inc.** Features. *Vuforia*. [Online] 2016. <https://www.vuforia.com>.
26. **Wikitude GmbH.** Augmented Reality SDK. *Wikitude*. [Online] 2016.
<http://www.wikitude.com/products/wikitude-sdk/>.
27. **Catchoom Technologies.** Augmented Reality. *Solutions from Catchoom*. [Online] 2016.
<http://catchoom.com/solutions/augmented-reality/>.
28. **ARLab.** Augmented reality SDKs. *ARLab*. [Online] 2016. <http://www.arlab.com/products>.
29. **Google Inc.** Google Cardboard. *Google VR*. [Online] 2016.
<https://vr.google.com/cardboard/index.html>.
30. **Greensted Dr. Andrew.** Otsu Thresholding. *The Lab Book Pages*. [Online] 2010.
<http://www.labbookpages.co.uk/software/imgProc/otsuThreshold.html>.
31. **Baggio Daniel Lélis, i inni.** Mastering OpenCV with Practical Computer Vision Projects. brak miejsca : Packt Publishing, 2012.
32. **J. Lim Joseph, Pirsiavash Hamed i Torralba Antonio.** Dataset for IKEA 3D models and aligned images. *IKEA Dataset*. [Online] <http://ikea.csail.mit.edu>.
33. **Inter IKEA Systems B.V.** MALM Biurko. *IKEA*. [Online] 2016.
<http://www.ikea.com/pl/pl/catalog/products/60214159/#/00214157>.
34. **Blender Foundation.** Home. *Blender*. [Online] 2016. <https://www.blender.org>.
35. **OpenCV.** Detection of ArUco Markers. *OpenCV*. [Online] 2015.
http://docs.opencv.org/3.1.0/d5/dae/tutorial_aruco_detection.html#gsc.tab=0.
36. **Wojas Sebastian.** Metody przetwarzania obrazów z wykorzystaniem biblioteki OpenCV. 2010.