AKADEMIA GÓRNICZO-HUTNICZA

IM. STANISŁAWA STASZICA W KRAKOWIE



Wydział Inżynierii Metali i Informatyki Przemysłowej



PRACA DYPLOMOWA INŻYNIERSKA

pt.

„Implementacja aplikacji na urządzenia mobilne do projektowania oraz wizualizacji lotu Quadcoptera z wykorzystaniem usługi Google Maps”

Imię i nazwisko dyplomanta: **Kamil Dudek**

Kierunek studiów: **Informatyka Stosowana**

Nr albumu: **268935**

Promotor: dr hab. inż. Łukasz Rauch

Recenzent: dr inż. Krzysztof Regulski

Podpis dyplomanta: Podpis promotora:

Kraków 2017

„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, wideogram 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 godności studenta student ponosi odpowiedzialność dyscyplinarną przed komisją dyscyplinarną albo przed sądem koleżeńskim samorządu studenckiego, zwanym dalej "sądem koleżeńskim"”, oświadczam, że niniejszą pracę dyplomową wykonałem(-am) osobiście i samodzielnie i że nie korzystałem(-am) ze źródeł innych niż wymienione w pracy.”

Kraków, dnia …… Podpis dyplomanta…………….

Spis treści

[1. Wprowadzenie 4](#_Toc471061492)

[2. Problem znalezienia optymalnej drogi 5](#_Toc471061493)

[3. Cel pracy 7](#_Toc471061494)

[4. Rozwiązania obecnie istniejące na rynku 9](#_Toc471061495)

[4.1 Google Maps 9](#_Toc471061496)

[4.2 OMPL – The Open Motion Planning Library 10](#_Toc471061497)

[4.3 OSMBuildings 11](#_Toc471061498)

[4.4 Ogólne zestawienie 11](#_Toc471061499)

[5. Rozwiązanie problemu 13](#_Toc471061500)

[5.1 Definicja mapy 13](#_Toc471061501)

[5.2 Przekształcenie mapy w strukturę grafu 15](#_Toc471061502)

[5.3 Algorytm A\* 17](#_Toc471061503)

[5.4 Wizualizacja 20](#_Toc471061504)

[5.5 Integracja z usługą Google Maps 25](#_Toc471061505)

[6. Wyniki 31](#_Toc471061506)

[6.1 Wpływ wielkości dyskretyzacji na szczegółowość mapy 32](#_Toc471061507)

[6.2 Analiza czasowa działania algorytmu A\* 33](#_Toc471061508)

[7. Podsumowanie 35](#_Toc471061509)

[8. Bibliografia 37](#_Toc471061510)

# Wprowadzenie

Wraz z rozwojem technologicznym jedynie kwestią czasu było stworzenie małych, latających urządzeń, sterowanych zdalnie. Dziś są one znane wszystkim jako drony, tudzież Quadcoptery. Ich cechą jest stosunkowo niewielki koszt produkcji oraz szeroki wachlarz możliwości, dzięki którym w łatwy sposób mogą zastąpić tradycyjne rozwiązania. Już teraz słyszy się o wdrażaniu bezzałogowych statków powietrznych jako wsparcie w akcjach ratunkowych GOPR. Mówi się także o wykorzystaniu ich do rozsyłania paczek w obrębie danej aglomeracji, co już zaczyna testować firma Amazon w swoim programie Amazon Prime Air. Dostarczanie pizzy z wykorzystaniem Quadcoptera rozważa również amerykańska sieć gastronomiczna Domino’s Pizza. Wszystkie te działania jednoznacznie wskazują, iż próby upowszechnienia rozwiązań opartych o drony są sukcesywnie dokonywane, co świadczy o atrakcyjności tego rozwiązania.

W związku z dynamicznym rozwojem strefy bezzałogowych statków powietrznych, w 2014 roku w Kole Naukowym Metalsoft powstał projekt pt. „Quadcopter – Inteligentna poczta”, którego celem było zbudowanie drona oraz stworzenie dedykowanego systemu, który pozwoliłby na autonomiczne sterowanie Quadcopterem. System ten ma samodzielnie planować trasę lotu w oparciu o zbudowaną mapę przestrzeni. Dodatkowo musi on reagować na pojawiające się wokół niego przeszkody, by uniknąć nieprzewidzianych kolizji. Docelowo projekt ten miał za zadanie zbadać użyteczność dronów, które pełniłby funkcję inteligentnej poczty – byłyby w stanie samodzielnie dostarczyć przesyłkę w obrębie zadanego, znanego obszaru. Projekt ten, poza oczywistym zbudowaniem samego drona, musiał się skupić na trzech podstawowych aspektach. Pierwszym jest odpowiedni sposób komunikacji w celu osiągnięcia zdalnego sterowania dronem z poziomu aplikacji mobilnej. Drugim jest odpowiednie rozpoznawanie przeszkód znajdujących się wokół Quadcoptera na podstawie danych z kamer oraz dalmierzy, a także wczytanie ich do mapy przestrzeni systemu. Trzecim, ostatnim aspektem, jest stworzenie odpowiedniej reprezentacji wcześniej wspomnianej mapy przestrzeni oraz zaprojektowanie optymalnej trasy lotu drona, wraz z jej odpowiednią wizualizacją i integracją z usługą Google Maps. Zadaniem tej pracy inżynierskiej jest opracowanie aplikacji mobilnej rozwiązującej ostatni, trzeci aspekt wspomnianego projektu.

# Problem znalezienia optymalnej drogi

Pojawienie się na rynku pierwszych robotów wszelkiej maści, poczynając od tych poruszających się drogą lądową i kończąc na tych manewrujących w przestrzeni powietrznej, od razu zainspirowało programistów do nadania im funkcjonalności automatycznego planowania trasy do wyznaczonego celu. Inspiracja ta zrodziła problem, który uogólniając możemy nazwać problemem znalezienia optymalnej drogi. Aby go rozwiązać, należało przede wszystkim stworzyć odpowiednią reprezentację samej przestrzeni, w której mógłby działać odpowiednio przygotowany algorytm znajdujący pożądaną trasę lotu. Istnieje wiele podejść do reprezentacji przestrzeni znajdującej się wokół danego urządzenia.

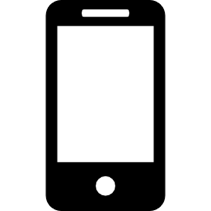
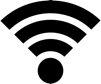
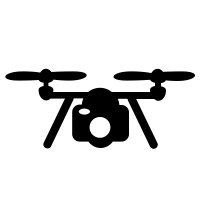
Pierwszym z nich jest zbudowanie modelu wektorowego [1]. Polega on na interpretacji każdego potencjalnego elementu za pomocą uporządkowanych zbiorów współrzędnych, które jednocześnie definiują jego położenie oraz kształt. Model ten traktuje każdy możliwy element mapy jako byt samodzielny, opisując go za pomocą punktów, linii i powierzchni. Model ten nie jest zbytnio czytelny dla człowieka, gdyż jego zapis jest bardzo skomplikowany, szczególnie w przypadku bardziej złożonych obiektów. Wymaga on również bardzo dużej analizy przestrzennej, by móc odpowiednio definiować zbiory opisujące poszczególne elementy znajdujące się na mapie. Wady te zdecydowanie rekompensuje zdecydowanie mały rozmiar danych reprezentujących mapę, co umożliwia utworzenie bardzo szczegółowej mapy przy jednocześnie stosunkowo jej małym rozmiarze. Narzut obliczeniowy wymagany jest jedynie podczas generacji mapy, co związane jest z wspomnianą już koniecznością zaawansowanej analizy przestrzeni[[1]](#footnote-1).

Drugim, konkurencyjnym podejściem do reprezentacji przestrzeni, jest zbudowanie modelu komórkowego. Model ten mając cechy obrazu cyfrowego często nazywany jest także *Pixel-Map*’ą. Polega on na odpowiednim podziale obszaru na komórki zazwyczaj o kwadratowym kształcie, tworząc tym samym siatkę najczęściej przypominającą rozległą szachownicę. Każda z takich komórek przechowuje informację o konkretnym obszarze, który obejmuje swoją wielkością. Komórki te tworzą swego rodzaju macierz, do której elementów możemy odwoływać się za pomocą zwykłych indeksów. Podejście takie jest zdecydowanie bardziej czytelne dla człowieka, gdyż wynik końcowy można w łatwy sposób zwizualizować. Nie wymaga on również specjalnie zaawansowanej analizy przestrzeni, co generuje korzyści w postaci mniejszego narzutu obliczeniowego. Niestety zalety te generują swojego rodzaju koszty, do których zalicza się zdecydowanie większy rozmiar tak zaprezentowanej mapy. Również jej szczegółowość jest dużym stopniu uzależniona od tak zwanej rozdzielczości mapy, czyli wielkości tworzącej jej komórek.

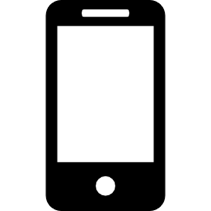
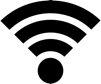
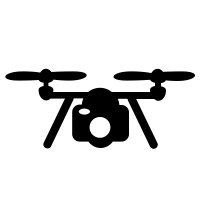
Jak można łatwo zauważyć, sam wybór odpowiedniej reprezentacji terenu jest stosunkowo trudnym wyborem. Należy sobie uświadomić, iż wybór ten będzie rzutował na poziom trudności w rozwiązaniu problemu znalezienia optymalnej drogi. Zauważmy, że mimo zalet znacznie lepszego pod kątem pamięciowym modelu wektorowego, który pozwala z dużą dokładnością przechowywać informację o terenie, narzut obliczeniowy związany z zaawansowanymi technikami analizy przestrzeni znacząco utrudnia znalezienie pożądanej ścieżki. Postać wektorowa reprezentowanych danych podnosi poziom skomplikowania problemu. Nie oznacza to jednak, iż rozwiązanie takie jest niemożliwe do uzyskania, jednak wymagać ono będzie zdecydowanie znacznie większych zasobów obliczeniowych niż te, którymi dysponuje sprzęt kierujący danym robotem. Sytuacja jest całkowicie odwrotna w przypadku modelu komórkowego. Model ten dzięki bardzo prostym analizom przestrzeni oraz bardzo przystępnej reprezentacji danych sprawia, iż liczba podejść do rozwiązania takiego problemu jest nieograniczona. Dzięki temu możemy tak wybrać algorytmy, aby ich złożoność obliczeniowa była adekwatna do sprzętu, na którym mają pracować.

# Cel pracy

Celem omawianej pracy jest stworzenie odpowiedniej reprezentacji mapy, która byłaby przystępna dla systemu działającego na Quadcopterze. Na tak zdefiniowanej mapie potrzeba znaleźć optymalną ścieżkę lotu, którą podążałby dron. Należy zwrócić uwagę, iż system ten jest już obecnie zajęty takimi działaniami jak stabilizacja oraz kontrola lotu. Fakt ten wymusza na nas znalezienie rozwiązania które nie wymaga zbyt wielu zasobów obliczeniowych, dlatego też zdecydowano o tym, aby większość obliczeń związanych ze znajdywaniem drogi była przeniesiona na aplikację mobilną, która jedynie wysyłałby do systemu znajdującego się na Quadcopterze wyliczoną już ścieżkę.



a)



b)

Rys. 1. Możliwe scenariusze komunikacji pomiędzy dronem a aplikacją mobilną: a) scenariusz, w którym aplikacja mobilna wyznacza drogę i przesyła ją do Quadcoptera, b) scenariusz, w którym dron napotyka przeszkodę i prosi o aktualizację trasy

Istnieje wiele możliwych scenariuszy komunikacji pomiędzy dronem a aplikacją mobilną. Na Rys. 1 a) widzimy podstawowy scenariusz, w którym aplikacja mobilna znajduje optymalną drogę i wysyła ją do Quadcoptera. Należy jednak pamiętać, że równie często może pojawić się drugi scenariusz, przedstawiony na Rys. 1 b). Przedstawia on bowiem scenariusz, w którym to dron po napotkaniu przeszkody potrzebuje natychmiastowej aktualizacji trasy. Pytanie, które samoistnie nasuwa się w tym momencie, dotyczy sposobu jej aktualizacji. Czy w takim przypadku dron powinien wysłać prośbę do aplikacji mobilnej? A co w przypadku, gdy aplikacja jest nieosiągalna z powodu kłopotów z siecią lub po prostu jest wyłączona? Możliwe, że znacznie bezpieczniejszym scenariuszem byłoby wykonanie aktualizacji trasy samodzielnie przez drona. Rozwiązanie to jednak wymagałoby utrzymywania dwóch wersji implementacji, co może być trudne w późniejszym etapie rozwoju. Jakkolwiek są to realne scenariusze które należy rozpatrzyć, praca ta w głównej mierze skupia się na pierwszym z nich, przedstawionym na Rys. 1 a).

Podejście ścisłej współpracy pomiędzy dronem a aplikacją mobilną umożliwia dodatkowo odpowiednią wizualizację mapy, która pomogłaby końcowemu użytkownikowi wybrać początek oraz koniec trasy, a także zobrazowała mu wyznaczoną ścieżkę lotu. Dodatkowo zyskujemy możliwość łatwej integracji z dodatkowymi usługami takimi jak Google Maps, którą można wykorzystać w celu zainicjalizowania początkowego stanu mapy budynkami otrzymanymi za pośrednictwem tej usługi, co odciąża również samego drona od konieczności początkowego rozpoznawania całego terenu. Jako urządzenia, dla których przeznaczona jest wspomniana aplikacja, wybrano te z systemem Android.

# Rozwiązania obecnie istniejące na rynku

Przed przystąpieniem do realizacji założonego celu, należało dokonać odpowiedniego

przeglądu literatury. Obecnie dostęp do map nie jest już utrudniony, przez co na rynku istnieje spora liczba gotowych i darmowych rozwiązań, z których każdy może skorzystać. Należy jednak przejrzeć je pod kątem integracji z urządzeniami autonomicznie się poruszających, a także urządzeń latających. W tym celu wybrano 3 najpopularniejsze serwisy udostępniających tego rodzaju usługę.

## 4.1 Google Maps

Serwis Google Maps obecnie jest najpopularniejszą usługą tego typu na rynku[[2]](#footnote-2). Aplikacja z tą usługą jest dostarczana na każdym smartfonie z zainstalowanym system Android. Jej szeroki wachlarz możliwości zrzesza coraz większą liczbę zadowolonych użytkowników. Możemy z jej pomocą, poza standardowym przeglądaniem mapy, znaleźć interesującą nas trasę do konkretnego celu. Pod tym względem Google Maps na tle konkurencji wyróżnia się tym, iż podczas wyznaczania trasy bierze pod uwagę informacje o obecnie panującym ruchu na drodze, co umożliwia wielu kierowcom skutecznie omijać korki. Poza możliwością planowania trasy samochodowej, usługa ta umożliwia również zaplanowanie trasy rowerowej czy też pieszej. Ponadto Google Maps posiada wiele autorskich podsystemów. Jednym z nich jest Street View – usługa umożliwiająca tak zwane „wirtualne spacery” po konkretnym fragmencie mapy. Bardzo korzystnym aspektem jest też ostatnio dodana możliwość pracy w trybie „offline”. Poza oczywistymi udogodnieniami warto też zwrócić uwagę na same mapy. Są one bardzo dobrej jakości, co jest spowodowane poprzez bieżące aktualizację. Również i same zdjęcia satelitarne są wysokiej rozdzielczości. Ponadto w obrębie większych aglomeracji coraz częściej można spotkać budynki, które możemy oglądnąć w 3D, co oznacza, iż mają już wczytaną odpowiednią wysokość budynków. Funkcjonalność ta występuje jeszcze na niewielkich obszarach – przykładowo w Krakowie budynki 3D możemy zaobserwować jedynie w okolicach Rynku Głównego, pozostała część miasta nie posiada już takiej możliwości.

Od strony technicznej, Google Maps udostępnia szeroki wachlarz wspieranych platform. Usługę tą z łatwością możemy zintegrować nie tylko na systemach z systemem Android, ale również na konkurencyjnych systemach iOS. Integracja nie stanowi także problemu dla programistów webowych – tutaj mamy przystępne biblioteki dla języka JavaScript. Warto   
w tym miejscu zaznaczyć, iż Google Maps tak naprawdę można postrzegać jako tak zwany Web-Service. Wszystkie operacje na mapach w każdej aplikacji wymagają połączenia   
z serwerem Google Maps, który przetwarza otrzymane zapytania. Dlatego też teoretyczna integracja z tym systemem jest możliwa z praktycznie każdego obecnie dostępnego języka programowania – każdy współczesny język umożliwia wysyłanie zapytań typu REST do zewnętrznych serwisów. Należy jednak wspomnieć, iż nie jest to usługa w pełni darmowa. Do momentu, gdy nie przekroczymy pewnego progu limitu zapytań, jest to usługa bezpłatna, jednak gdy przekroczymy ustalone limity, musimy uiścić odpowiednią kwotę. Limity te definiowane są w postaci „liczba zapytań na dzień”, gdzie liczba zapytań dotyczy konkretnej aplikacji, a nie konkretnego użytkownika. Aby skorzystać z usługi Google Maps, każda aplikacja musi skorzystać z wcześniej uzyskanego klucza dostępu, co oznacza, że wszystkie egzemplarze tej aplikacji korzystają z tego samego klucza. Już tylko przy 25 000 użytkownikach, każdy z nich będzie mógł wykonać średnio jedno zapytanie dziennie, co jest bardzo małym limitem, który wymusza na właścicielach aplikacji zakupu dodatkowych pakietów.

## 4.2 OMPL – The Open Motion Planning Library

Następną tego rodzaju usługą, tym razem już znacznie mniej znaną, jest OMPL[[3]](#footnote-3). Skrót ten pochodzi od „Open Motion Planning Library”. Jak sama nazwa wskazuje, jest to biblioteka przeznaczona do planowania drogi. Biblioteka ta jest jedynie ograniczona do algorytmów planowania drogi, co oznacza, że nie uwzględnia ona żadnych specyfikacji dotyczących środowiska, wykrywania kolizji czy też wizualizacji. Jest to jednak zabieg zamierzony, gdyż biblioteka ta jest zaprojektowana do łatwej integracji z systemami które już zapewniają dodatkowe komponenty [2]. Biblioteka ta jest w całości zaimplementowana w języku C++   
i oferuje bogate zaplecze algorytmiczne. Algorytmy wykorzystywane w tej bibliotece to algorytmy oparte o próbkowanie przestrzeni. Fundamentalną ideą takich algorytmów jest aproksymacja połączeń w przeszukiwanej przestrzeni za pomocą struktury grafu. Przestrzeń może być próbkowana w różny, zależny od algorytmu sposób, a wybrane próbki tworzą wierzchołki grafu aproksymującego [3]. Algorytmy te zaliczają się do klasy algorytmów probabilistycznych, co oznacza, że działają z bardzo wysokim prawdopodobieństwem znalezienia rozwiązania, jednak rozwiązanie to nie jest optymalne. Cechuje je wysoka wydajność obliczeniowa oraz szybkość działania.

Biblioteka ta zyskała powodzenie w dziedzinie robotyki. Jest ona dostarczana wraz   
z systemem ROS – Robot Operating System, który jest często instalowany na różnego rodzaju robotach. Ważną zaletą tej biblioteki jest stosunkowo łatwa możliwość rozszerzania i dodawania własnych algorytmów. Jednakże biblioteka ta nie wizualizuje w żaden sposób znalezionej drogi. Nie integruje się ona również ze standardowymi mapami, co zrzuca obowiązek integracji z istniejącymi mapami na programistę.

## 4.3 OSMBuildings

*OSMBuildings* jest usługą oferującą dostęp do map[[4]](#footnote-4). Skrót *OSM* oznacza „*Open Street Map*”, co już na wstępie sugeruje, że system ten bazuje na mapach pochodzących z projektu „*OpenStreetMap*”. Projekt ten jest w pełni otwarty, tworzony przez dużą społeczność, która codziennie aktualizuje mapy w oparciu nie tylko o zdjęcia satelitarne i dane *GPS*, ale także   
o własne, indywidualne doświadczenia. *OSMBuildings* oferują dokładną reprezentację 3D budynków pobranych z projektu *OSM*. Budynki znajdujące się w tym systemie cechują się wysoką dokładnością, zarówno w aspekcie wysokości budynków, ich kształtu jak i nawet rozmiarem i kształtem okien. Podstawową wadą tej usługi jest utrudniona integracja z innymi systemami. Cały system jest stworzony jako projekt webowy, co byłoby bardzo ciężko zintegrować z urządzeniem mobilnym. Mimo utrudnionej integracji usługa ta udostępnia klientom *Web-Service*. Podobnie jak w przypadku Google Maps, jest on w stanie zwrócić interesujące nas obiekty za pomocą zapytania *REST*, jednak takie rozwiązanie zrzuca na programistę obowiązek ręcznej wizualizacji mapy. Największą wadą tego systemu jest brak jakiegokolwiek mechanizmu planowania drogi – udostępnia on jedynie ogromną bazę danych budynków, które są opisywane bardzo szczegółowo.

## 4.4 Ogólne zestawienie

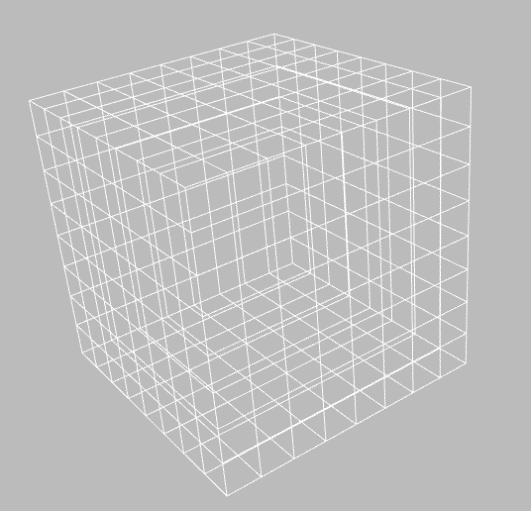
Jak widzimy, obecnie na rynku znajduje się wiele rozwiązań dostarczających skuteczną reprezentację mapy wraz z mechanizmami planowania trasy. Jednakże patrząc w perspektywie omawianej pracy, nie można znaleźć systemu, który spełniałby postawione wymagania. Rozwiązanie, które jest poszukiwane, to system integrujący się z istniejącymi już mapami, który byłby w stanie wyznaczyć optymalną trasę przelotu Quadcoptera. Zauważmy, iż usługa Google Maps pomimo zaawansowanych technik planowania trasy, nie posiada opcji planowania drogi dla dronów. Opcja taka mogłaby uwzględniać fakt, iż urządzenie nie musi poruszać się wyłącznie wzdłuż ulic, ale również nad budynkami i innymi obiektami. Wymagałoby to natomiast informacji o wysokościach budynków, które Google Maps na chwilę obecną nie ma skompletowanej – jedynie niektóre budynki mają informację o swojej wysokości. Z kolei rozwiązanie zaproponowane przez bibliotekę OMPL jest bardzo atrakcyjne pod względem algorytmów znajdowania drogi. Zauważmy jednak, iż biblioteka ta to sam zestaw algorytmów. Nie oferuje ona żadnej integracji z istniejącymi mapami czy też wizualizacji znalezionej drogi. Ponadto jest to biblioteka całościowo stworzona w języku C++, co utrudniałoby jej wykorzystanie na platformie Android, w której naturalnym językiem programowania jest Java. Natomiast OSMBuildings wraz ze swoją bardzo bogatą bazą danych na temat budynków nie posiada żadnego mechanizmu planowania trasy. Wszystkie te aspekty powodują, że przy obecnie istniejących rozwiązaniach nie jesteśmy w stanie wybrać żadnego z nich. Pójście na kompromis w którąkolwiek ze stron jest nieakceptowalne z punktu widzenia projektu „Quadcopter – Inteligentna poczta”. Projekt ten bowiem narzuca na takie oprogramowanie wymagania w postaci wizualizacji mapy oraz wyznaczonej ścieżki dla użytkownika, dobrej integracji z zewnętrznymi usługami oraz urządzeniami mobilnymi a także wyznaczanie trasy lotu Quadcoptera również nad budynkami, co wymaga znajomości ich wysokości. W związku z tym zdecydowano się na implementację własnego rozwiązania, które byłoby pewnego rodzaju hybrydą stojącą pośrodku wyżej wymienionych usług.

# Rozwiązanie problemu

Rozwiązaniem problemu zdefiniowanego przez wymagania projektu „Quadcopter -Inteligentna poczta” jest aplikacja mobilna zaprojektowana na urządzenia z systemem Android. Jej implementacja wymagała rozwiązania szeregu wyżej przedstawionych problemów w taki sposób, aby jej użyteczność była wystarczająca z punktu widzenia wyżej wymienionego projektu. Językiem programowania, który wykorzystano do implementacji była Java, która jest językiem naturalnym dla wybranej platformy mobilnej. Oprogramowanie zostało tworzone przy użyciu framework-u Android SDK w wersji 24, co oznacza, iż poziom kompilacji jest zgodny z systemami Android Nougat (inaczej Android 7.0). Jako iż w momencie implementacji był to system dopiero wdrażany do produkcji, jedynie niewielka liczba urządzeń na rynku posiadała najnowszy system. Aby zachować więc zgodność z większością urządzeń działających z tym systemem, zdecydowano się na kompatybilność wsteczną aż do wersji 16 framework-u Android SDK. Oznacza to, że aplikacja będzie w pełni funkcjonować na systemach Android począwszy od wersji Android Jelly Bean (inaczej Android 4.1), co łącznie na dzień 7 listopada 2016 roku stanowiło aż 97.3% wspieranych urządzeń z systemem Android znajdujących się na rynku[[5]](#footnote-5). Aplikacja była tworzona w całości w IDE (Integrated Development Environment – Zintegrowane środowisko programistyczne) o nazwie „Android Studio”. Po zdefiniowaniu wymagań systemowych przystąpiono do implementacji.

## 5.1 Definicja mapy

Kluczowym elementem implementacji był dobór odpowiedniej reprezentacji mapy. Mapa ta musi przede wszystkim mieć postać łatwo czytelną zarówno dla drona, jak i dla człowieka. Dodawanie nowych elementów do tej mapy nie może być skomplikowane obliczeniowo, ponieważ dron może wykonywać takie operacje na bieżąco, podczas lotu. Jest to związane z tym, że ewentualne wykrycie nowej przeszkody musi zostać szybko uwzględnione w logicznej reprezentacji mapy. Z tego powodu zdecydowano się wzorować na tak zwanej PixelMap-ie. Opiera się ona w głównej mierze na rastrowej reprezentacji obrazu znanej z grafiki komputerowej [4]. Podejście to przede wszystkim opiera się na założeniu, że teren pozbawiony jest naturalnych wypukłości i jest płaski. Teren ten następnie jest dzielony na małe kwadraty   
o ustalonym rozmiarze, którym przyporządkowane są liczby naturalne. Kwadraty te można porównać do pikseli w rastrowym obrazie. Jeżeli do kwadratu przypisana jest liczba „zero”, oznacza to, że obszar ten jest wolny, czyli nie ma w tym miejscu żadnej przeszkody, tak więc dron może przelecieć przez ten kawałek przestrzeni. Jeżeli przypisana wartość jest większa od zera, oznacza to, że przez ten obszar dron nie jest w stanie się przemieścić, gdyż znajduje się tam jakaś przeszkoda. Jako iż głównymi przeszkodami drona będą budynki, przyjęto, iż liczba ta będzie prezentować wysokość budynku, do którego przynależy dany fragment obszaru. Jednakże w odróżnieniu od PixelMap-y (a także od obrazu rastrowego), których reprezentację można logicznie zapisać w formie liczbowej macierzy dwuwymiarowej, z naturalnych potrzeb tego projektu mapa jest prezentowana w formie trójwymiarowej macierzy. Forma taka jest wymagana, aby uwzględnić możliwość przelotu nad danym budynkiem. W efekcie powstała mapa przypomina nałożenie na siebie kilkunastu dwuwymiarowych PixelMap, które razem tworzą przestrzenny prostopadłościan. Wizualnie można go sobie wyobrazić jako figurę geometryczną (jak przedstawiono na Rys. 2.) ułożoną z wielu, mniejszych prostopadłościanów, mogących reprezentować konkretny obszar rzeczywistej przestrzeni. Do tak małego elementu można przypisać pewien stan, który będzie informował drona o możliwości przelotu przez dany obszar.



Rys. 2. Przykładowa wizualizacja trójwymiarowej PixelMap-y. Podejście trójwymiarowe w dużym stopniu przypomina podział przestrzeni na szereg małych sześcianów, z których każdy posiada określony stan.

Każdy pojedynczy element jest identyfikowany za pomocą trzech współrzędnych, przy czym początek układu współrzędnych znajduje się w prawym, dolnym rogu mapy.   
W implementacji mapa ta jest przedstawiona jako trójwymiarowa tablica typu Integer, zawierających informację o stanie danej komórki. Ponadto stworzono dodatkową warstwę abstrakcji reprezentującej budynek. Budynek ten jest opisywany poprzez podstawowe parametry, takie jak jego wysokość, szerokość oraz długość. Ponadto posiada informację   
o współrzędnych punktach, które go tworzą, dzięki czemu można go w łatwy sposób odnieść do mapy, na której został zdefiniowany. Dzięki takiej strukturze Quadcopter jest w stanie w łatwy i przede wszystkim szybki sposób dodać nowy budynek w momencie rozpoznania oraz zidentyfikowania go. Wystarczy bowiem, iż zdefiniuje on jedynie ciąg współrzędnych, na których znajduje się przeszkoda, co już w zupełności wystarczy do aktualizacji mapy. Przykłady użycia stworzonej struktury przedstawiono na Rys. 3

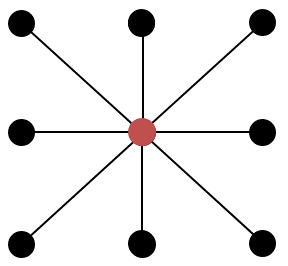
Map map = new Map(20, 37, 40);  
  
map.addBuilding(new Building(14, 8, 4, 5, 8));  
map.addBuilding(new Building(4, 2, 3, 6, 2));  
  
map.addBuildingPoint(2, 3, 4);  
map.addBuildingPoint(2, 4, 4);  
map.addBuildingPoint(3, 3, 4);  
map.addBuildingPoint(3, 4, 4);

Rys. 3. Przykładowe użycie stworzonych struktur w kodzie. Struktura umożliwia dodawanie przeszkód zarówno w formie budynków, jak i współrzędnych.

## 5.2 Przekształcenie mapy w strukturę grafu

Kolejnym krokiem w implementacji aplikacji było stworzenie mechanizmu, który byłby w stanie odnaleźć w wyżej przedstawionej reprezentacji terenu optymalną ścieżkę. Aby tego dokonać należało odpowiednio zinterpretować wcześniej stworzoną mapę, tak aby mechanizm znajdywania drogi odpowiednio mógł na niej operować. Zdecydowano się na konwersję trójwymiarowej przestrzeni na graf ważony. W ten sposób można z łatwością wykorzystać całą gamę algorytmów operujących na grafach, które wyznaczają najkrótszą ścieżkę w grafie. Przekształcenie to odbywa się w bardzo prosty sposób – każdy element mapy konwertowany jest na wierzchołek grafu, natomiast wagi połączeń między wierzchołkami stanowi koszt przelotu do sąsiedniego wierzchołka. Kluczowym elementem jest struktura takiego wierzchołka. Każdy z tych wierzchołków w swojej strukturze przechowuje następujące informacje:

* Lista sąsiedztwa: każdy wierzchołek przechowuje w liście referencje do najbliższych, otaczających go wierzchołkach
* Wagi połączeń: każdy wierzchołek jest w stanie określić wagę połącznia (czyli koszt przejścia) do wierzchołka sąsiedniego, co przedstawiono na Rys. 4.



Rys. 4. Sąsiedztwo dla wierzchołka na poziomie 2D. Dla danego przykładu, wierzchołek posiada jeszcze 9-ciu sąsiadów nad sobą oraz 9-ciu sąsiadów pod sobą, co łącznie daje 26-ciu sąsiadów.

* Referencję do rodzica: w przypadku, gdy wierzchołek ten będzie należał do ścieżki, będzie on przechowywał w sobie referencję do swojego rodzica. Rodzic określa wierzchołek poprzedzający w wyznaczonej ścieżce. Przykładowo, rodzicem końcowego wierzchołka będzie przedostatni wierzchołek ścieżki, którego rodzicem będzie poprzedni wierzchołek, aż do wierzchołka startowego. Wierzchołek startowy, jako jedyny wierzchołek w ścieżce, nie posiada ustawionej referencji do rodzica. Przykład utworzonej w ten sposób ścieżki znajduje się na Rys. 5.

Rys. 5. Przykładowa struktura ścieżki w grafie. Wierzchołek zielony jest punktem początkowym, wierzchołek czerwony jest punktem końcowym, czarne wierzchołki reprezentują punkty pośrednie tworzące ścieżkę. Kierunek strzałki wskazuje rodzica wierzchołka, od którego strzałka wychodzi.

* Przechodniość wierzchołka – informuje o tym, czy przez dany wierzchołek można przejść, czy nie (bo przykładowo wierzchołek ten należy do budynku lub innej przeszkody, przez którą nie można przejść)

Taka struktura reprezentująca wierzchołek pozwala w łatwy sposób przechowywanie informacji o sąsiadach konkretnego wierzchołka oraz koszcie połączeń pomiędzy nimi. Dzięki takiej strukturze jesteśmy także w łatwy sposób ocenić, czy dany wierzchołek należy do ścieżki, poprzez sprawdzenie referencji do jego rodzica. W zaproponowanym podejściu koszty połączeń pomiędzy wierzchołkami dobierany jest na podstawie energii potrzebnej na pokonanie danej odległości. Przykładowo, na podstawie testów doświadczalnych wywnioskowano, że znacznie większa energia potrzebna jest na podniesienie Quadcoptera w górę niż na przelot w linii prostej, nie zmieniając jego wysokości, stąd też waga połączenia pomiędzy wierzchołkiem znajdującym się ponad obecnie rozpatrywanym punktem będzie większa od wagi połączenia pomiędzy wierzchołkiem znajdującym się na tej samej wysokości.

Mechanizm konwersji mapy wymagał stworzenia nowych struktur, których zadaniem jest reprezentacja mapy w postaci grafu. W klasie *Map* dodano metodę *createGraph* mającą na celu stworzenie oraz inicjalizację grafu na podstawie obecnego stanu mapy. Jak widzimy na Rys. 6, przed inicjalizacją grafu następuje proces wykrywania budynków na podstawie wprowadzonych ich współrzędnych.

public void createGraph(){

if (buildingsPoints.size() > 0){  
 List<Building> detectedBuildings =

Building.*createBuildingsFromPoints*(buildingsPoints);  
 for (Building b: detectedBuildings){  
 for (Building.Point p: b.getBuildingPoints()) {  
 for (int i =0; i < p.height; i++) {  
 map[p.x][p.y][i] = p.height;  
 }  
 }  
 }  
 buildings.addAll(detectedBuildings);

}  
 connectionList = new ConnectionList(width, depth, height);  
 connectionList.initialize(map);  
}

Rys. 6. Metoda createGraph znajdująca się w klasie Map, dokonująca inicjalizacji grafu.

## 5.3 Algorytm A\*

Kluczowym etapem całego projektu jest odpowiednie znajdywanie optymalnej drogi przelotu Quadcoptera, co w gruncie rzeczy, dzięki wyżej opisanej konwersji, sprowadziło się do znalezienia najkrótszej ścieżki w grafie. Początkowo planowano wykorzystanie do tego celu popularnego algorytmu Dijkstry [5]. Zapewnia on, że znaleziona ścieżka jest zawsze ścieżką optymalną, co gwarantuje poprawność rozwiązania. Jednakże dalsza analiza kosztu obliczeniowego wykazała, iż metoda ta ma zbyt dużą złożoną obliczeniową dla grafów gęstych, do których zalicza się również graf wygenerowany na podstawie mapy terenu. Działanie tego algorytmu opiera się bowiem na przetwarzaniu każdego możliwego wierzchołka grafu w celu znalezienia optymalnej ścieżki. Powoduje to, iż w takim przypadku zyskujemy kwadratową złożoność obliczeniową, co oznacza, iż liczba operacji potrzebnych do znalezienia ścieżki jest wprost proporcjonalna do kwadratu liczby danych wejściowych. Tak duża złożoność z punktu widzenia projektu jest niedopuszczalna. Algorytm znajdywania drogi powinien zostać tak dobrany, aby jego złożoność obliczeniowa była jak najmniejsza. Nie oznacza to, że złożoność takiego algorytmu jest nieakceptowalna dla urządzeń mobilnych – czterordzeniowe procesory w smartfonach dzisiaj nie są już dla nikogo zaskoczeniem, więc jego odpowiednie zrównoleglenie powinno zapewnić satysfakcjonujące wyniki. Jednakże w perspektywie,   
w której sam Quadcopter zmuszony by był do samodzielnego przeliczenia drogi (na przykład w przypadku utraty komunikacji z kontrolerem), taka złożoność mogłaby się okazać krytyczną dla mikrokomputera kontrolującego cały system lotu drona. W związku z tym zaczęto poszukiwanie innej metody znajdywania najkrótszej ścieżki w grafie, czego wynikiem okazał się algorytm A\* (A-Star) [6]. Jest to powszechnie wykorzystywana metoda we wszelkiego rodzaju grach komputerowych, w których należy w szybki sposób znaleźć drogę w tak zwanych mapach kafelkowych. Algorytm ten zalicza się do algorytmów heurystycznych, którego wynik jest zupełny i optymalny, co daje nam gwarancję znalezienia ścieżki jeśli tylko taka istnieje, przy czym ścieżka ta jest optymalna. Przy odpowiednio zdefiniowanej heurystyce, algorytm ten jest dopuszczalny (tzn. zawsze dostarcza prawidłowe rozwiązanie), ponadto przeszukuje najmniej węzłów spośród pozostałych algorytmów dopuszczalnych. Przy odpowiednio zdefiniowanej funkcji heurystycznej (tzn. w taki sposób, by odchylenie jej wartości od faktycznej odległości od celu było nie większe niż logarytm z liczby określającej wartość tej odległości), algorytm ten charakteryzuje się również optymalną złożonością obliczeniową klasy wielomianowej [7]. Dzięki odpowiednio przyjętej heurystyce, złożoność tego algorytmu znacząco się zmniejsza.

Algorytm A\* podczas obliczania kosztu ścieżki bazuje na prostej funkcji kosztu przebycia drogi, która przyjmuje następującą wartość:

Przy czym:

* – funkcja opisująca koszt przebycia z wierzchołka startowego do wierzchołka aktualnie rozpatrywanego
* – funkcja heurystyczna (tzw. heurystyka), przedstawiającą szacunkową odległość od wierzchołka aktualnie rozpatrywanego do wierzchołka docelowego. Dla potrzeb aplikacji do wyznaczenia odległości przyjęto metrykę euklidesową.

Funkcja ta musi być wyznaczana dla każdego wierzchołka osobno, co w implementacji rozwiązano poprzez przeniesienie logiki obliczania wartości tej funkcji do klasy reprezentującej wierzchołek. Dzięki temu jesteśmy w stanie określić koszt ścieżki dla każdego wierzchołka podając jedynie referencję do punktu początkowego oraz końcowego. Warto w tym miejscu podkreślić, iż algorytm Dijkstra może być rozważany jako jeden z przypadków użycia algorytmu A\*, w którym funkcja heurystyczna zawsze przyjmuje wartość 0 [8].

Podstawowa zasada algorytmu A\* opiera się na minimalizacji wyżej przedstawionej funkcji reprezentującej koszt przebycia drogi. Ponadto, algorytm ten podczas działania przechowuje informację o dwóch listach – otwartej oraz zamkniętej liście wierzchołków. Otwarta lista wierzchołków zawiera wszystkie wierzchołki, które mogą (ale nie muszą) być dołączone do szukanej ścieżki w następnym kroku. Zamknięta lista wierzchołków reprezentuje wierzchołki, które już zostały przeszukane, co pozwala uniknąć wielokrotnego rozpatrywania pojedynczego wierzchołka. Na samym początku punkt startowy jest dodawany do listy otwartych wierzchołków. Pojedynczy przebieg algorytmu składa się z następujących kroków:

1. Z listy otwartych wierzchołków wybierz ten, którego wartość funkcji jest najmniejsza. Usuń go z listy otwartych wierzchołków i dodaj go do listy zamkniętych wierzchołków. Oznacz go jako „obecnie rozpatrywany”.
2. Sprawdź sąsiedztwo wierzchołka „obecnie rozpatrywanego”.
   1. Jeżeli sąsiad nie należy do listy zamkniętych wierzchołków, jest przechodni oraz nie należy jeszcze do listy otwartych wierzchołków, dodaj go do tej listy. Ustaw rodzica dodanego sąsiada na obecnie rozpatrywany wierzchołek.
   2. Jeżeli sąsiad obecnie znajduje się na liście otwartych wierzchołków, sprawdź czy ścieżka od punktu początkowego do tego wierzchołka nie jest korzystniejsza. Aby tego dokonać, sprawdź wartość funkcji dla tego wierzchołka, reprezentującej koszt dotarcia do niego od wierzchołka startowego. Jeżeli wartość dla tego wierzchołka będzie mniejsza od wartości dla wierzchołka obecnie rozpatrywanego, ustaw jego rodzica na wierzchołek obecnie rozpatrywany oraz przelicz ponownie wartości funkcji .

Wyżej opisany algorytm został zaimplementowany w sposób przedstawiony na Rys. 7.

public List<Position> findPath (Position startPosition, Position endPosition, ConnectionList map) {  
 Node start = map.getNode(startPosition);  
 Node end = map.getNode(endPosition);  
 List<Node> closed = new ArrayList<>();  
 List<Node> opened = new ArrayList<>();  
  
 //Add the start node to opened list  
 opened.add(start);  
 Node current = opened.get(0);  
 while (current != end) {  
 //Looking for the smallest F value in opened list  
 for (Node node : opened) {  
 if (node == opened.get(0) || node.getF() <= current.getF()) {  
 current = node;  
 }  
 }  
 //remove current point from opened list  
 opened.remove(current);  
 current.setOpened(false);  
  
 //add current point to closed list  
 closed.add(current);  
 current.setClosed(true);  
  
 //Getting list of adjacent nodes  
 List<Node> adjacentNodes = current.getAdjacentNodes();  
 for (Node child : adjacentNodes) {  
 if (child.isClosed() || !child.isWalkable()) {  
 continue;  
 }  
 if (child.isOpened()) {  
 if (child.getG() > child.getG(current)) {  
 // Change its parent and g score  
 child.setParent(current);  
 child.computeScores(end);  
 }  
 } else {  
 //Add it to the opened list with current point as parent  
 opened.add(child);  
 child.setOpened(true);  
  
 // Compute it's g, h and f score  
 child.setParent(current);  
 child.computeScores(end);  
 }  
 }  
 }  
 // Finally find path and return it  
 List<Position> path = new ArrayList<>();  
 while (current.hasParent() && current != start){  
 path.add(current.getPosition());  
 current = current.getParent();  
 }  
 path.add(startPosition);  
 return path;  
}

Rys. 7. Implementacja algorytmu A-Star, znajdująca się w klasie AStarPathfinder.

Algorytm kontynuuje swoje poszukiwanie do momentu, aż wierzchołek reprezentujący punkt końcowy nie zostanie oznaczony jako „obecnie rozpatrywany”. W momencie osiągnięcia wierzchołka końcowego, ścieżka do niego od punktu początkowego zostaje znaleziona. Aby ją wyznaczyć wystarczy jedynie „wrócić się” do punktu początkowego, zaczynając od punktu końcowego, przechodząc poprzez referencję do rodzica wierzchołka, aż wierzchołek startowy nie zostanie osiągnięty.

## 5.4 Wizualizacja

Implementacja algorytmu szukającego drogi była kluczowym elementem pracy, jednak nie był to ostatni jej etap. Aby umożliwić użytkownikom przystępny sposób reprezentacji znalezionego rozwiązania, należało zwizualizować zarówno stworzoną mapę, jak i znalezioną ścieżkę. Wizualizacja powinna w łatwy dla użytkownika sposób przedstawić trójwymiarową przestrzeń, w której porusza się Quadcopter. Użytkownik powinien być w stanie dowolnie nią obracać, przybliżać lub oddalać. Jako iż aplikacja służy do znajdywania drogi, użytkownik musi mieć także możliwość do wprowadzania współrzędnych punktu startowego oraz końcowego.

Ponieważ wizualizacja wymaga używania elementów trójwymiarowej grafiki komputerowej nie można w tym celu użyć standardowych, wysokopoziomowych kontrolek oferowanych przez framework Android-a. Aby móc wyświetlać interaktywne elementy 3D, należy użyć w tym celu wbudowanej we framework biblioteki OpenGL. W celu przyśpieszenia implementacji postanowiono skorzystać z dodatkowej biblioteki LibGDX[[6]](#footnote-6). Biblioteka ta dostarcza programiście dodatkowych narzędzi pozwalających uniknąć niskopoziomowego programowania w OpenGL. Narzędzia te umożliwiają posługiwanie się nieco wyższym poziomem abstrakcji, skutecznie i optymalnie zamieniając ją w szereg wektorów i macierzy reprezentujących typy proste wymagane przez OpenGL. Ponadto biblioteka LibGDX umożliwia stworzenie aplikacji hybrydowych, działających na wielu platformach. Poza platformą Android, biblioteka umożliwia uruchomienie tworzonego oprogramowania jako zwykłej aplikacji desktopowej lub webowej. Aby zapewnić hybrydowość aplikacji, należy podzielić kod całego projektu na moduły. Modułowość aplikacji łatwo można osiągnąć w systemie Gradle[[7]](#footnote-7), który jest naturalnym system budowania projektu dla aplikacji na platformy Android. Gradle jest konkurencyjnym systemem do Maven[[8]](#footnote-8), który jest popularnie stosowany w projektach pisanych w języku Java. Za pomocą systemu Gradle, stworzono 3 moduły w projekcie, jak przedstawiono na poniższym rysunku:

Rys. 8. Graf przedstawiający strukturę projektu wykorzystanego w implementacji

Jak przedstawiono na Rys. 8, cały projekt jest teraz podzielony na 3 moduły. Pierwszy z nich słusznie zawdzięcza swoją nazwę od angielskiego słowa rdzeń, gdyż można go przyrównać do silnika całej aplikacji. W module tym powinien znaleźć się cały kod odpowiadający za logikę oraz renderowanie obrazu. Pozostałe moduły, czyli „*android*” oraz „*desktop*” służą za punkty startowe dla poszczególnych platform. Przykładowo, gdy aplikacja będzie uruchamiana na telefonie z systemem Android, moduł „android” będzie odpowiadał za jej inicjalizację, by następnie móc uruchochomić zawartość modułu „*core*”. Dlatego też moduł ten zawiera kod, który realizuje funkcjonalności specyficzne dla platformy. W przypadku platformy Android będzie to stworzenie odpowiedniej struktury aplikacji, w której znajdą się zarówno definicje uprawnień aplikacji (*AndroidManifest.xml*), katalogi z zasobami aplikacji (przykładowo ikony aplikacji) oraz same źródła aplikacji, korzystające z podstawowych komponentów framework-u. Przykładem takiego podstawowego komponentu na platformie jest Activity. Komponent ten jest ściśle związany z systemem – stanowi punkt startowy aplikacji oraz posiada swój własny cykl życia, uwarunkowany w dużej mierze od samego systemu. Tak samo wygląda sprawa w przypadku modułu „desktop”, który będzie odpowiadał za inicjalizację okna, w której ma się wyrenderować aplikacja. Pozostała część logiki aplikacji – nie tylko ta mówiąca o tym co ma zostać wyrenderowane, ale także jak aplikacja ma reagować na polecenia użytkownika oraz jaką ma spełniać funkcjonalność, znajduje się w module „core”, który jest wspólny dla wszystkich innych modułów reprezentujących poszczególne platformy. Taka hybrydowość jest bardzo wydajnym podejściem, bowiem umożliwia ona programiście stworzenie jednolitego, wspólnego modelu dla wszystkich możliwych platform, odciążając go tym samym od konieczności implementacji różnych sposobów interakcji z użytkownikiem. Biblioteka LibGDX dostarcza spójną warstwę abstrakcji, która przykładowo umożliwia obsługę aplikacji zarówno za pomocą myszki na komputerze jak i za pomocą gestów wykonywanych na dotykowym ekranie telefonu komórkowego, bez potrzeby implementacji osobnych metod specyficznych dla konkretnego rodzaju sprzętu. Nie oznacza to jednak, że rezygnujemy całkowicie z możliwości konkretnej platformy – LibGDX umożliwia tworzenie konkretnych wersji poszczególnych klas, które będą wykorzystane tylko w momencie uruchomienia aplikacji na platformie, na którą te klasy są przeznaczone. Podejście takie jednak nie jest zalecane, gdyż tworzenie implementacji wyspecjalizowanych pod kątem konkretnej platformy pozbawia aplikację hybrydową wszystkich jej zalet.

W związku z powyżej przedstawioną koncepcją aplikacji hybrydowej, cały kod odpowiedzialny za reprezentację mapy, przekształcanie jej w graf a także wyszukiwanie drogi został przeniesiony do wspólnego modułu „core”. Dodatkowo, w module tym pojawiły się klasy odpowiedzialne za samo renderowanie tejże mapy. Biblioteka LibGDX pozwala uniknąć skomplikowanego tworzenia trójwymiarowych elementów używając specyficznych dla OpenGL typów prymitywnych. Zostały one zastąpione pojęciem Modelu. Model w ujęciu LibGDX reprezentuje pewnego rodzaju wzór, według którego ma powstać konkretny element na scenie. Model nie jest w żaden sposób związany ze sceną – nie ma on informacji o tym,   
w którym miejscu na scenie ma on zostać umieszczony. Posiada on tylko i wyłącznie informacje o swoim wyglądzie, m. in. o materiale, z którego jest wykonany oraz o swoim kształcie. Kształt ten może być prosty, które tworzą podstawowe elementy geometryczne, takie jak prostopadłościany, sześciany, ostrosłupy, walce czy też kule. Model może też składać się   
z tak zwanych części („*Parts*”), co umożliwia tworzenie bardziej skomplikowanych kształtów. Nie zagłębiając się w szczegóły techniczne, Model posiada informacje typu „jak obiekt powinien wyglądać”. Informację na temat „jak dany obiekt wyrenderować” zawarte są w Instancji Modelu („*ModelInstance*”). Przechowuje ona informację o tym, z jakich typów prymitywnych obiekt powinien być zbudowany, a także w którym miejscu na scenie powinien on zostać wyrenderowany. Mówiąc prościej, Instancja Modelu zajmuje się stworzeniem odpowiedniej siatki Modelu, którą silnik OpenGL będzie w stanie poprawnie zinterpretować oraz wyświetlić. W celach optymalizacyjnych, wiele Instancji Modelu można łączyć w jedną siatkę, która umożliwi szybsze renderowanie obiektów. W tym celu należy stworzyć specjalny bufor („*ModelCache*”), który jest w stanie scalać wszystkie obiekty typu ModelInstance w jedną, spójną siatkę, co umożliwi jej efektywne wyrenderowanie silnikowi OpenGL.

Podczas implementacji powstała osobna klasa „*MapRenderer*”, odpowiedzialna ze samo renderowanie obiektu mapy. Renderuje ona regularną, prostokątną siatkę reprezentującą wymiary mapy, a także wszystkie zawarte w niej budynki. W przypadku, gdy budynek stworzony został za pomocą listy współrzędnych, dla każdego z punktów tworzona jest taka sama liczba Instancji Modelu. Instancje Modelu tworzone są z Modelu, który jest zwykłym sześcianem o wymiarach pojedynczej kratki na wcześniej wyrenderowanej siatce. Instancje te następnie łączone są w jedną siatkę za pomocą bufora scalającego „*ModelCache”*, który jest już optymalnie renderowany na scenie. Na Rys. 9 przedstawiony został fragment kodu odpowiedzialny za powyższą operację. Ponadto klasa „*MapRenderer”* zajmuje się także wyświetlaniem znalezionej ścieżki, która złożona jest z szeregu Instancji Modelu kuli o średnicy pojedynczej kratki tworzącej siatkę, jak przedstawiono na Rys. 10.

private void generateMap(){  
 mesh = new ModelInstance(map.getModel()); //place mesh in the center of view  
 disposables.add(map.getModel());  
  
 for(Building b: map.getBuildings()){  
 if (b.shouldMergePoints()) {  
 List<ModelInstance> pointInstances = new ArrayList<>();  
 for (Building.Point point: b.getBuildingPoints()){  
 pointInstances.add(new ModelInstance(b.getModel(),

computePosition(point)));  
 }  
 ModelCache cache = new ModelCache();  
 cache.begin();  
 cache.add(pointInstances);  
 cache.end();  
 buildings.add(cache);  
 disposables.add(cache);  
 } else {  
 ModelInstance instance = new ModelInstance(b.getModel(),

computePosition(b));  
 buildings.add(instance);  
 }  
 disposables.add(b.getModel());  
 }  
}

Rys. 9. Funkcja generująca wizualną strukturę mapy. W przypadku, gdy budynek jest złożony ze współrzędnych które powinny zostać scalone, korzysta się ze bufora scalającego („ModelCache”).

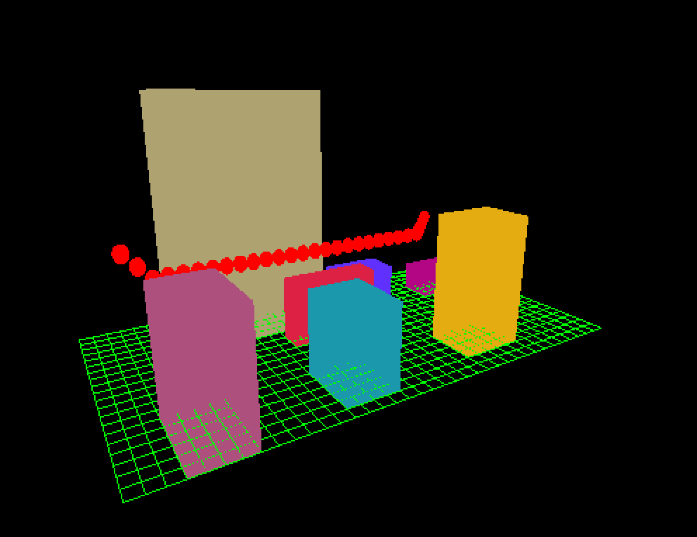
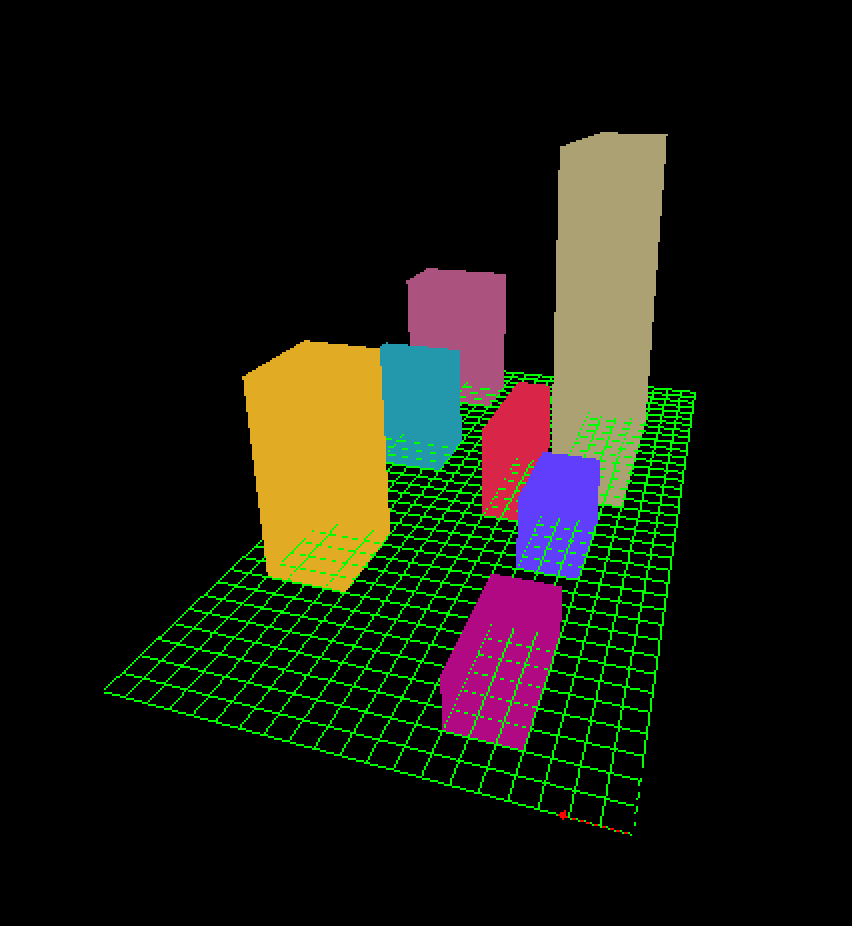
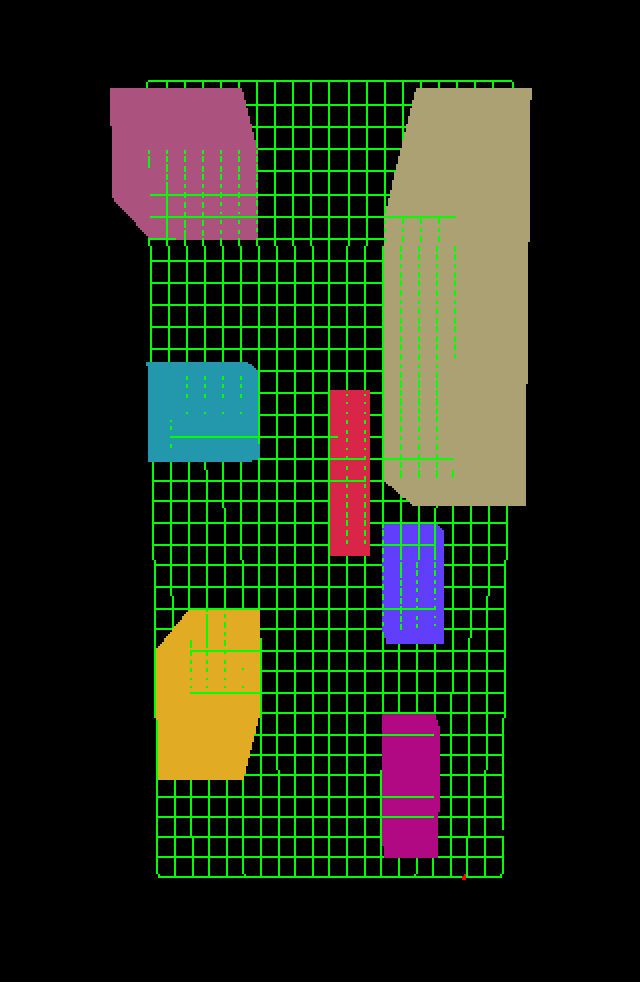
public void drawPath(List<Position> path){  
 Model pathPart = modelBuilder.createSphere(Map.*NODE\_SIZE*, Map.*NODE\_SIZE*,

Map.*NODE\_SIZE*, 10, 10,  
 new Material(ColorAttribute.*createDiffuse*(new Color(Color.*RED*))),  
 VertexAttributes.Usage.*Position* | VertexAttributes.Usage.*ColorPacked*);  
  
 ModelCache modelCache = new ModelCache();  
 modelCache.begin();  
 for (Position pos: path){  
 modelCache.add(new ModelInstance(pathPart, computePosition(pos)));  
 }  
 modelCache.end();  
 this.path = modelCache;  
  
 disposables.add(pathPart);  
 disposables.add(modelCache);  
}

Rys. 10. Funkcja odpowiadająca za rysowanie znalezionej ścieżki. Kolejne punkty wyznaczonej ścieżki reprezentowane są za pomocą figury w kształcie kuli.

public void render(){  
 modelBatch.begin(camera);  
 modelBatch.render(buildings, environment);  
 modelBatch.render(mesh, environment);  
 if (path != null) {  
 modelBatch.render(path, environment);  
 }  
 modelBatch.end();  
}

Rys. 11. Najważniejsza metoda klasy „MapRenderer” odpowiadająca za finalne wyświetlenie całego modelu mapy.



a)

b)

c)

Rys. 12. Wizualizacja mapy za pomocą biblioteki LibGDX. a) widok prostopoadły, b) widok boczny, c) widok boczny z wyrenderowanym przykładem znalezionej ścieżki

Najważniejszą metodą w klasie „*MapRenderer*” jest funkcja „*render”*, przedstawiona na Rys. 11. Jest ona wywoływana przez główną część aplikacji (klasa „PathfinderApp”) średnio 30 razy na sekundę i jest odpowiedzialna za ostateczne wyrenderowanie modelu mapy. Warto w tym miejscu zauważyć, iż mapa ma taki sam wygląd niezależnie od urządzenia, na którym została wyświetlona. Na Rys. 12 a) widzimy mapę wyrenderowaną na smartfonie z systemem Android, natomiast na Rys. 12 b) i c) znajduje się ta sama mapa, jednak otworzona za pomocą aplikacji desktopowej działającej na systemie OSX.

## 5.5 Integracja z usługą Google Maps

Ostatnim elementem omawianego projektu jest integracja z usługą Google Maps. Obecna modułowa struktura projektu uwzględnia działanie aplikacji również w trybie desktopowym. Ponieważ praca w jako cel przewiduje działanie aplikacji jedynie na urządzeniach z systemem Android, integracja ta została wykonana tylko w zakresie modułu „android”. Nie oznacza to jednak, że jest ona niemożliwa do przeprowadzenia w zakresie aplikacji desktopowej.

Na obecnym etapie aplikacja mobilna posiada tylko jedną aktywność (*Activity*). *Activity* jest natywnym elementem platformy Android. Aplikacje mobilne muszą posiadać co najmniej jedną aktywność, ponieważ reprezentuje ona wizualny komponent widoczny dla użytkownika aplikacji. Komponent taki posiada odpowiedni cykl życia oraz mechanizm nawigacji, umożliwiający przełączanie się pomiędzy innymi aktywnościami[[9]](#footnote-9). Zadaniem obecnie zaimplementowanej, jedynej aktywności w aplikacji jest uruchomienie kodu znajdującego się w module „*core*”, czego efektem jest natychmiastowe pojawienie się wizualizacji mapy   
w bibliotece LibGDX. Ponieważ celem integracji jest przekształcenie mapy zaczerpniętej   
z usługi Google Maps, postanowiono dodać dodatkowe *Activity*, które by było nowym punktem startowym aplikacji mobilnej. *Activity* to ma za zadanie otworzyć mapy z usługi Google Maps, które rozszerzono dodatkowo o funkcjonalność przeglądania map poprzez dodanie nowego natywnego komponentu *Fragment*. Google Maps wraz ze swoją biblioteką dostarcza przygotowany już *MapFragment*, który służy do załadowania, wyświetlania   
i przeglądania mapy przez końcowego użytkownika. *Activity* ze wspomnianym fragmentem ustawiono jako nowy punkt startowy aplikacji w pliku *AndroidManfiest.xml*, dzięki czemu po uruchomieniu aplikacji jako pierwszy ekran pojawi się wspomniana mapa.

Integracja usługi Google Maps z konwencją mapy, przyjętą w omawianej pracy, na pierwszy rzut oka wydaje się być skomplikowana. Ciężko jest pozyskać z usługi Google Maps informację na temat budynków w postaci umożliwiającej łatwą integrację z konwencją PixelMap-y. Przede wszystkim utrudnienie to jest spowodowane przez fakt, iż każdy obiekt pojawiający się na mapie traktowany jest jako zbiór punktów tworzących wielokąt („*Polygon”*), niezależnie od tego, czy jest to budynek czy też droga. Ponadto, punkty te są de facto współrzędnymi geograficznymi, określającymi ich położenie na mapie, co powoduje kolejne problemy przy próbie ich interpretacji w tak naprawdę względnym układzie współrzędnych, nie mających większego związku ze współrzędnymi geograficznymi. Kolejny problem, który stoi na przeszkodzie, jest sposób pozyskiwania tych danych. Konieczne jest bowiem ciągłe wysyłanie zapytań do serwera o współrzędne danych budynków, co w rezultacie mogłoby doprowadzić do szybkiego przekroczenia dziennego limitu zapytań dla aplikacji. W związku z tym postanowiono zintegrować usługę Google Maps w całkowicie odrębny sposób. Ponieważ obecnie przyjęta konwencja mapy jest technicznie zbliżona do PixelMap-y, mającą swoje odzwierciedlenie w grafice rastrowej, postanowiono odwzorować wyświetlany fragment mapy właśnie na podstawie analizy obrazu, która pozwoliłaby na wydobycie informacji   
o znajdujących się na rozpatrywanym obszarze budynkach i przeniesieniem ich do przyjętej konwencji reprezentacji mapy.

Pierwszym etapem konwersji mapy było stworzenie zrzutu mapy aktualnie wyświetlanej   
w formacie bitmap-y. Mapa ta jednak wyświetla poza interesującymi nas budynkami także ulice oraz inne obiekty, które nie powinny stanowić przeszkody dla trasy drona. Aby je wyeliminować, po naciśnięciu przycisku z mapy usuwane są wszystkie elementy niebędącymi budynkami. Można to osiągnąć poprzez metody dostarczane przez usługę Google Maps, mianowicie poprzez narzucenie na mapę nowego stylu. Styl mapy jest okrojony do tego stopnia, aby pozostały na niej tylko budynki. Zrzut do bitmap-y jest robiony dopiero w momencie, gdy na mapie znajdują się tylko elementy będące potencjalną przeszkodą dla drona. Przykład działania takiego procesu został przedstawiony na Rys. 13, w którym wykorzystano fragment mapy kampusu AGH. Po lewej stronie znajduje się fragment widoku aplikacji, na którym użytkownik może operować na mapie poprzez jej przesuwanie, oddalanie czy też obracanie. Po wybraniu fragmentu mapy i kliknięciu przycisku „Capture map”, z mapy usuwane są wszelkie nieznaczące z punktu widzenia drona elementy, takie jak drogi czy też fragmenty zieleni. Na Rys. 13 po lewej stronie umieszczony został ten sam fragment mapy, na którym widoczne są już tylko budynki. Dopiero dla tak przygotowanej mapy tworzony jest zrzut, który będzie wykorzystywany w dalszej części procesu.



Rys. 13 Proces oczyszczania wyświetlanej mapy ze wszystkich elementów niestanowiących przeszkody dla drona na przykładzie mapy fragmentu kampusu AGH.

Utworzony zrzut mapy jest następnie poddawany procesowi obróbki graficznej, który również został zaimplementowany w ramach omawianej pracy. Algorytm przetwarzania jest bardzo prosty. Pierwszym jego etapem jest taki sposób przetworzenia bitmap-y, aby pozostały na niej tylko i wyłącznie przeszkody, które należy przenieść do systemu drona. W tym celu usuwane są wszystkie piksele, których kolor odpowiada kolorowi podłoża mapy. Następnie tak przygotowane zdjęcie poddawane jest procesowi dyskretyzacji, polegające na podziale obszaru na mniejsze fragmenty. Ilość podziałów w pionie i poziomie odpowiada późniejszym wymiarom mapy interpretowanym przez system Quadcoptera. Dla każdego z fragmentów określany jest stan, który pozwoli nam zidentyfikować, czy należy go rozpatrywać jako przeszkodę czy też nie. Stan ten jest ustalany na podstawie zagęszczenia pikseli reprezentujących budynek w obrębie rozpatrywanego fragmentu. Gdy liczba pikseli, których kolor odpowiada reprezentacji budynków w obszarze danego fragmentu, będzie większa niż wcześniej z góry przyjęty poziom (przykładowo 85%), uznaje się, że segment ten jest częścią przeszkody. Dodaje się go tym samym do listy punktów, z których następnie zostaną utworzone budynki w procesie tworzenia wewnętrznej reprezentacji mapy.



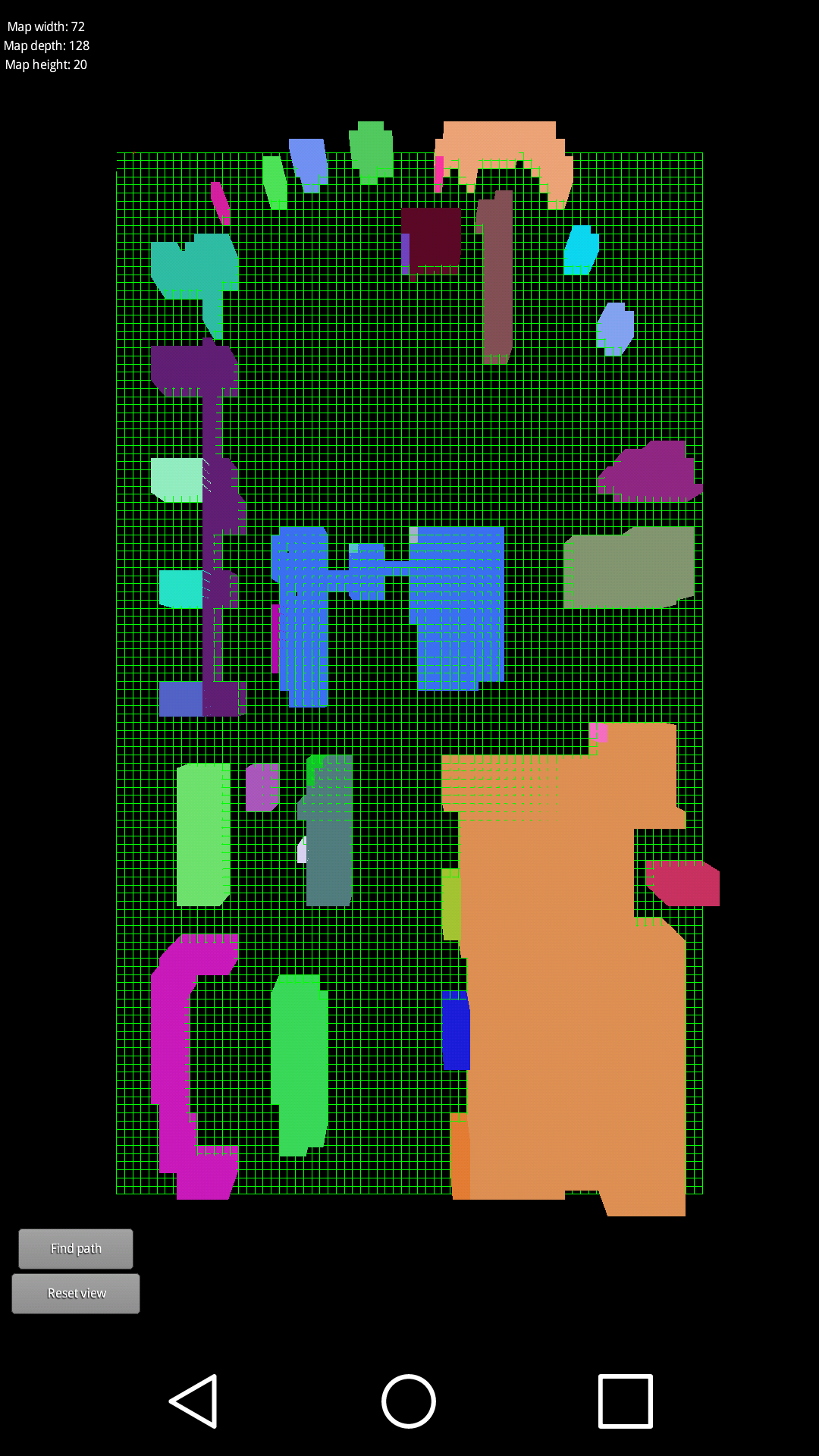
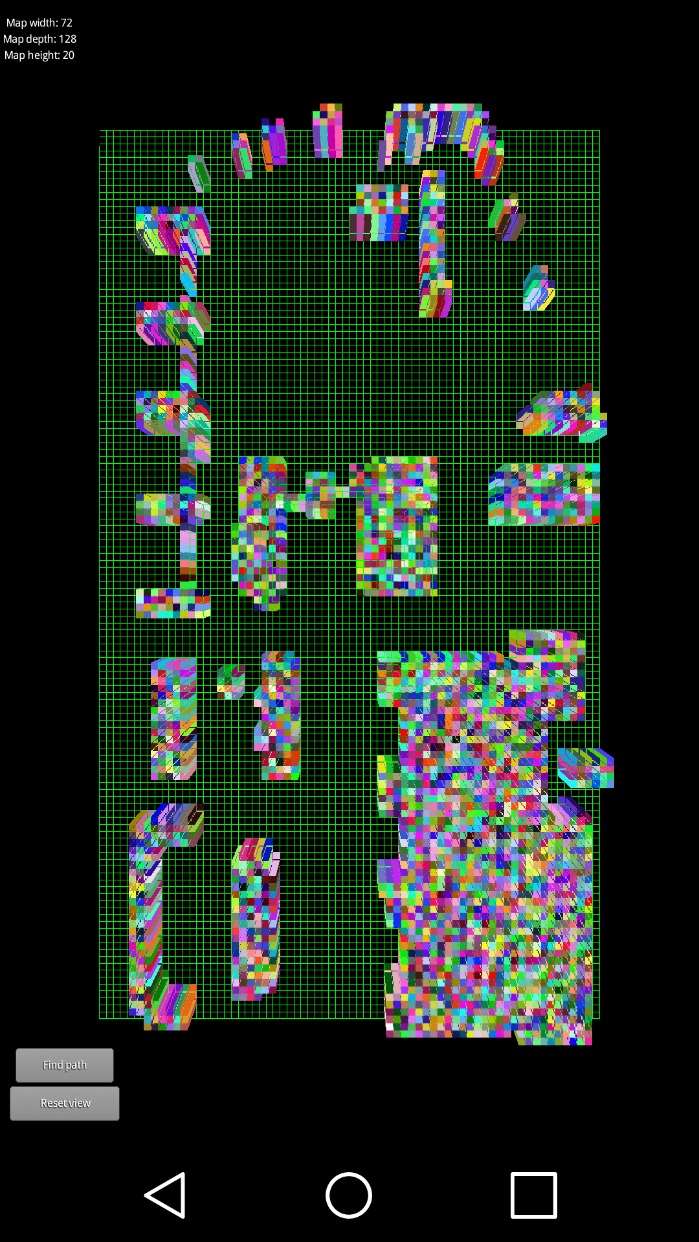
a)

b)

Rys. 14 Zrzuty mapy a) po wyodrębnieniu budynków, b) po dyskretyzacji obszaru. Żaden z powyższych widoków nie jest widoczny dla użytkownika podczas korzystania z aplikacji.

Na Rys. 14 a) przedstawiono wynik procesu przetwarzania zrzutu mapy, dzięki któremu udało się wydzielić znajdujące się na niej budynki. Tak przygotowana bitmapa uległa późniejszej dyskretyzacji na równe obszary, w tym przykładzie o wielkości 15x15 pikseli, dzięki czemu   
w łatwy sposób można określić, które punkty konwertowanej mapy powinny być interpretowane jako części budynku. Znając położenie punktów możemy stworzyć na ich podstawie mapę, wykorzystując wcześniej zaimplementowane mechanizmy.

Przyjęta implementacja mapy zakłada, iż informacja o budynku dostarczana jest   
w postaci listy punktów, które go tworzą. Konieczne jest więc rozdzielenie pojedynczej, zbiorczej listy zawierającej współrzędne wszystkich punktów na mniejsze listy, które reprezentują konkretne pozycje budynku. Ujmując rzecz prościej, należy pogrupować znajdujące się w jednej liście współrzędne w mniejsze listy, które przechowywać będą współrzędne punktów tworzących konkretny budynek. Takie podejście jest konieczne z punktu widzenia wydajności aplikacji. Aby model mapy Quadcoptera był w stanie w szybki   
i optymalny sposób się wyrenderować, punkty muszą być pogrupowane w struktury budynków. Dzięki temu będziemy mogli użyć wspomnianego w rozdziale 6.4 mechanizmu „ModelCache”, który jest w stanie scalić wiele pojedynczych siatek graficznych w jedną, większą siatkę, co umożliwi znaczne przyśpieszenie procesu renderowania. Jak widzimy na Rys. 15 a), ominięcie procesu grupowania spowodowało pojawienie się 2774 prostopadłościanów. Na każdym z nich należy wywołać kosztowną operację renderowania. Tak duża ilość instancji ModelInstance powoduje, że aplikacja staje się nieresponsywna. W związku z powyższym, przed finalnym utworzeniem nowej mapy, użyto naiwnego algorytmu grupującego punkty, sprawdzającego przynależność do danego budynku dzięki poszukiwaniu swoich sąsiadów. Wynik działania tego algorytmu przedstawiony został na Rys. 15 b), a jego implementacja znajduje się na Rys. 16. Jest to algorytm bardzo prosty, przez co staje się nieodporny na różnorodność warunków brzegowych, co uwidacznia się w wyrenderowanych budynkach. Często bowiem zdarzają się sytuację, w których dany budynek nie jest rozpoznany jako całość, sprawiając wrażenie, iż składa się on z kilku mniejszych budynków. Mimo swych niedoskonałości, algorytm przyniósł oczekiwany skutek. W przypadku widocznym na Rys. 15 zmniejszył on liczbę instancji do wyrenderowania z 2774 do 37, powodując tym samym zmniejszenie liczby wywołań operacji renderowania o około 74 razy, czego skutkiem jest ogromny skok wydajnościowy. Dzięki temu zabiegowi aplikacja stała się również w pełni responsywna, co umożliwia użytkownikowi jej łatwą obsługę, a także jest zgodne z zalecanymi zasadami tworzenia aplikacji mobilnych rekomendowanych przez platformę Android.



a)

b)

Rys. 15. Przykład stworzonej mapy na podstawie map pozyskanych z usługi Google Maps, a) wizualizacja bez procesu grupującego, b) wizualizacja po zastosowaniu procesu grupującego.

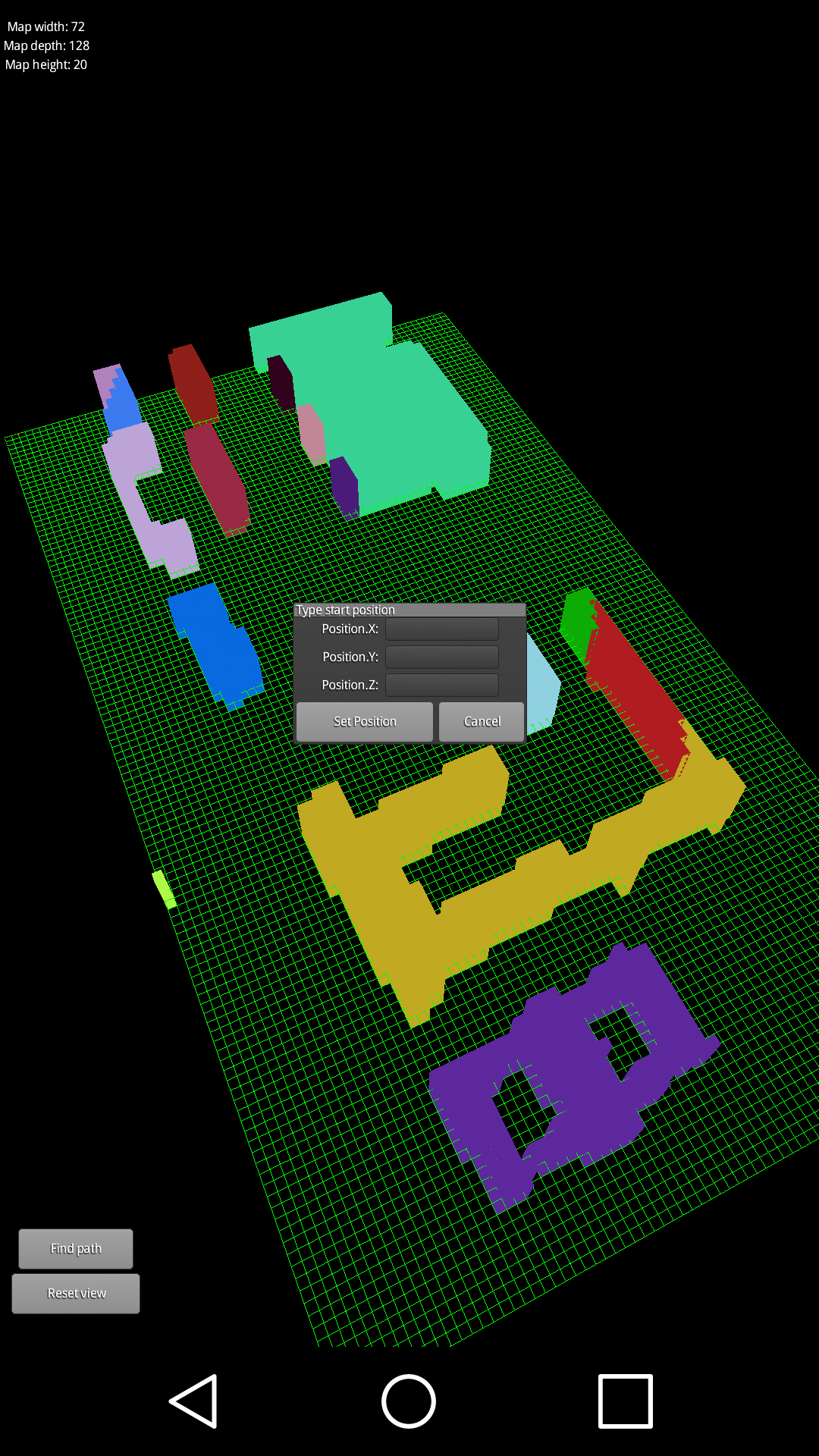
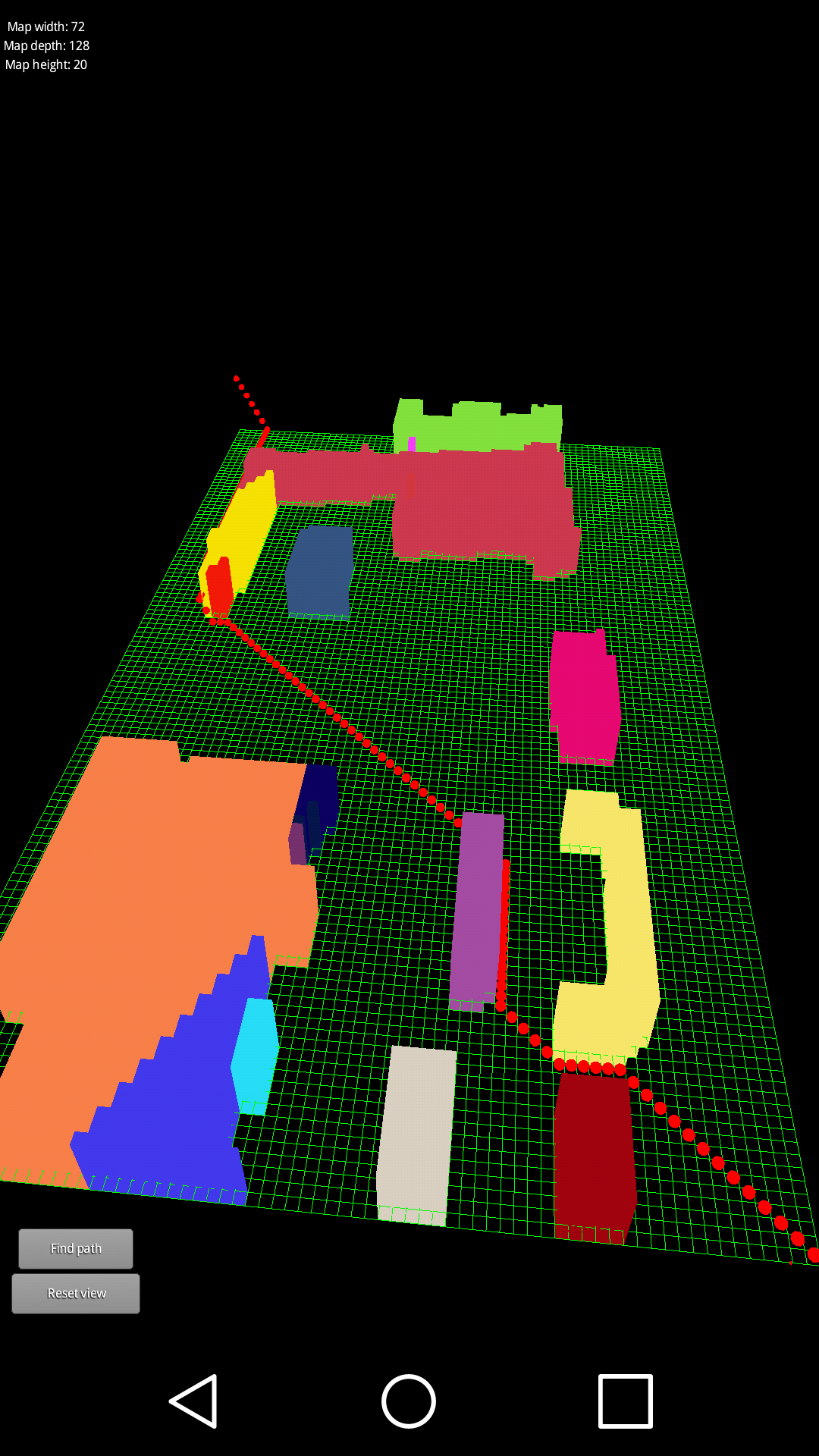
protected static List<Building> createBuildingsFromPoints(List<Point> pointList) {  
 List<Building> detectedBuildings = new ArrayList<>();  
 Collections.*sort*(pointList);  
  
 //Add first building with fist point...  
 Point firstPoint = pointList.get(0);  
 Building firstBuilding = new Building(firstPoint.x, firstPoint.y, 1, 1,

firstPoint.height);  
 firstBuilding.buildingPoints.add(firstPoint);  
  
 pointList.remove(0);  
 detectedBuildings.add(firstBuilding);  
 for (Point point : pointList) {  
 Building belongingBuilding = point.getBelongingBuilding(detectedBuildings);  
 if (belongingBuilding == null) {  
 belongingBuilding = new Building(point.x, point.y, 1, 1, point.height);  
 detectedBuildings.add(belongingBuilding);  
 }  
 belongingBuilding.buildingPoints.add(point);  
 }  
 return detectedBuildings;  
}

Rys. 16. Zastosowana implementacja naiwnego algorytmu grupującego.

# Wyniki

W wyniku wyżej opisanego procesu implementacji powstała w pełni funkcjonalna aplikacja, umożliwiająca użytkownikowi wybór punktów początkowych oraz końcowych. Użytkownik wskazuje te punkty poprzez wpisanie ich współrzędnych w wyświetlanym oknie dialogowym, które następnie są walidowane oraz wprowadzane do systemu znajdywania najkrótszej ścieżki, czego przykład jest widoczny na Rys. 17 a). Aplikacja ta jednocześnie wizualizuje reprezentację terenu rozumianą przez system sterujący Quadcopterem, a także umożliwia łatwe zainicjalizowanie mapy używając do tego usługi Google Maps. Ułatwia również podgląd znalezionej drogi, dzięki czemu użytkownik jest w stanie zweryfikować proponowaną ścieżkę. Znaleziona droga wyświetlana jest w formie okrągłych zbioru kul, jak przedstawiono na Rys. 17 b).



a)

b)



Rys. 17. Przykładowe użycie stworzonej aplikacji, a) wpisywanie współrzędnych punktu startowego, b) wizualizacja znalezionej ścieżki

W ramach omawianej pracy, postanowiono sprawdzić działanie aplikacji skupiając się na analizie dwóch czynników: wpływu wielkości dyskretyzacji na szczegółowość mapy oraz analizę czasową działania algorytmu A\*.

## 6.1 Wpływ wielkości dyskretyzacji na szczegółowość mapy

Pierwszym czynnikiem, jaki postanowiono zbadać używając stworzonej aplikacji, był wpływ stopnia dyskretyzacji na dokładność konwertowanej mapy. Jako iż algorytm ten w głównym stopniu opiera się o analizę graficzną badanej przestrzeni, siłą rzeczy narażony jest na niedokładności związane z jej interpretacją. Dokładność mapy jest kluczowym czynnikiem w autonomicznym systemie sterowania Quadcopterem, ponieważ jej brak może doprowadzić do kolizji drona z fragmentem budynku. Do takiej sytuacji może dojść, gdy pewien obszar zostanie pominięty podczas analizy terenu, przez co zostanie pominięty podczas interpretacji mapy.

Testy postanowiono przeprowadzić na jednym budynku o nieregularnych kształtach. Przyjęto trzy podstawowe konfiguracje, różniące się od siebie poziomem dyskretyzacji poprzez wielkość segmentu mapy. Zebrane wyniki zebrano w tabeli przedstawionej na Rys. 18.

|  |  |  |
| --- | --- | --- |
| Rozmiar pojedynczego segmentu (w pikselach) | Zrzut budynku | |
| Przed konwersją | Po konwersji (obrót o 180O) |
| 10x10 | ../../../Desktop/AndroidTool/dudek-MapActivity-12162016205941.png | ../../../Desktop/AndroidTool/dudek-MapActivity-12162016205108.png |
| 15x15 | ../../../Desktop/AndroidTool/dudek-MapActivity-12162016205435.png |
| 20x20 | ../../../Desktop/AndroidTool/dudek-MapActivity-12162016205549.png |

Rys. 18. Wyniki przedstawiające wpływ rozmiaru dyskretyzacji na dokładność mapy.

Budynek, który został użyty do celów testowych, bardzo dobrze odzwierciedla specyfikę problemu. Posiada on bowiem smukłą budowę, którego bryła posiada wiele cienkich elementów. Zachowując poziom dyskretyzacji przy wielkości segmentu 10x10 budynek został idealnie odzwierciedlony w utworzonej mapie – konwersja przy tak dużym poziomie dyskretyzacji nie spowodowała zmiany jego kształtu, wszystkie jego krawędzie zostały bardzo dobrze odwzorowane. Sytuacja zaczyna się zaburzać w przypadku segmentacji o wielkości 15x15. Tutaj możemy już zauważyć, iż zwiększenie rozmiaru segmentu spowodowało już poważne zniekształcenie. W skutek zmniejszenia poziomu dyskretyzacji nastąpiło pominięcie jednej z wąskich ścian budynku, przez co rozdzielił się ona na dwie, mniejsze budowle. Możemy także zauważyć, iż obrys figury przestaje odzwierciedlać faktyczny kształt budynku. Sytuacja pogarsza się już całkowicie po zwiększeniu rozmiaru segmentu do 20x20 pikseli. Budowla całkowicie zmieniła swój kształt po procesie konwersji, który pominął poprzeczne, najcieńsze ściany budynku. Zamiast pojedynczego budynku, na mapie znalazło się pięć osobnych budynków. Tak fatalna dyskretyzacja mogłaby niewątpliwie doprowadzić do kolizji Quadcoptera ze ścianą budynku.

## 6.2 Analiza czasowa działania algorytmu A\*

Kolejnym interesującym czynnikiem, któremu warto się bliżej przyjrzeć, jest szybkość działania zaimplementowanego algorytmu A\* służącego do znajdywania optymalnej ścieżki. W tym celu postanowiono zmierzyć czasy znajdywania ścieżki z punktu początkowego znajdującego się na początku układu do punktu końcowego na przeciwległym końcu mapy. Współrzędne tych punktów miały wspólną trzecią składową co oznacza, iż znajdywały się na tej samej wysokości. Mapę wypełniono w niewielkim stopniu budynkami o regularnym, kwadratowym kształcie, leżącymi na przekątnej mapy tak, aby stanowiły one potencjalną przeszkodę. Testom poddano konfiguracje, różniące się między sobą szerokością oraz długością mapy, przy stałej wysokości. Rys. 19 przedstawia zebrane wyniki w formie wykresu.

Rys. 19. Wpływ rozmiaru mapy na czas znajdywania ścieżki.

Łatwo zauważyć, iż czas znajdywana drogi rośnie liniowo wraz ze wzrostem rozmiaru mapy. Należy w tym miejscu zaznaczyć, iż wraz z rozmiarem mapy, wzrastał także dystans pomiędzy punktami początkowymi a końcowymi. Heurystyczny charakter wykorzystanego algorytmu wpłynął zdecydowanie korzystnie na czas obliczeń. Przy czterokrotnym zwiększaniu rozmiaru mapy, czas obliczeń rośnie jedynie dwukrotnie, co jest jak najbardziej wynikiem zadowalającym.

# Podsumowanie

W ramach przestawionej pracy stworzono w pełni funkcjonalną aplikację mobilną, pozwalającą użytkownikowi znaleźć ścieżkę lotu Quadcoptera pomiędzy dwoma punktami na urządzeniach z systemem Android. Zaprojektowano i zbudowano odpowiedni mechanizm reprezentacji terenu, łączący w sobie pożądane cechy wielu rozwiązań obecnie istniejących na rynku. Cechuje się on czytelnością zarówno dla użytkownika jak i samego drona, umożliwiając mu jednocześnie szybkie nanoszenie nowo wykrytych przeszkód w czasie rzeczywistym. Dodawanie nowych obiektów w tak przyjętej reprezentacji mapy nie jest kosztowne obliczeniowo, dzięki czemu może być wykonywane na urządzeniach dysponujących małą mocą obliczeniową. Takie rozwiązanie czyni je atrakcyjnym z punktu widzenia architektury sprzętowej Quadcoptera, której wydajność może być niewystarczająca dla konkurencyjnych rozwiązań. Ze względu na grafowy charakter algorytmów znajdywania drogi, konieczne było stworzenie mechanizmu przekształcającego mapę w strukturę grafu ważonego. Mechanizm ten umożliwił wykorzystanie szerokiej gamy algorytmów znajdywania najkrótszej drogi w grafach. W przedstawionej pracy stworzono własną implementację algorytmu A\*, który posiada właściwości heurystyczne, czyniące go zarówno wysoce wydajnym jak i efektownym. Stworzona implementacja wyróżnia się na tle już istniejących dzięki operowaniu na przestrzeni trójwymiarowej, co w głównej mierze umożliwił wspomniany mechanizm konwersji mapy na strukturę grafu ważonego. Mimo iż algorytm A\* jest głównie wykorzystywany w grach, gdzie rozpatrywany teren jest ujmowany w formie dwuwymiarowej, sprawdził się on równie dobrze w nietypowej, trójwymiarowej formie mechanizmu znajdywania drogi przelotu Quadcoptera. Zaimplementowana aplikacja umożliwia także integrację z usługą Google Maps, która pozwala na zainicjalizowanie utworzonej mapy istniejącymi już budynkami. Stworzony system integracji bazuje na procesie obróbki graficznej mapy wyświetlanej przez Google Maps, który opiera się na odpowiedniej dyskretyzacji rozpatrywanego obszaru. Ważnym aspektem aplikacji jest także sama wizualizacja terenu, do której wykorzystano nietypową bibliotekę cechującą się właściwościami hybrydowymi. Aby umożliwić użytkownikowi podgląd wygenerowanej mapy w formie trójwymiarowej, stworzono odpowiednią architekturę ściśle związaną ze stworzoną formą reprezentacji terenu. W celu zachowania responsywności aplikacji zastosowano również autorski algorytm grupujący obiekty graficzne, a także użyto technik łączenia siatek tych obiektów.

Jakkolwiek cel projektu został spełniony, nie oznacza to końca prac nad implementacją. Wciąż istnieje wiele elementów, które zostaną dodane w przyszłych iteracjach implementacyjnych. Pierwszą rzeczą, którą wciąż wymaga ulepszenia, jest integracja z usługą Google Maps. Nadal bowiem brakuje odpowiedniego mechanizmu, który byłby w stanie odczytywać wysokości budynków i przenosić je do tworzonej mapy. Nierozwiązany także został problem przesyłu danych z urządzenia mobilnego na drona, co wymaga zaimplementowania osobnego mechanizmu komunikacji. Mechanizm ten musiałby uwzględniać wiele sytuacji ekstremalnych, jak chociażby brak połączenia z Quadcopterem, czy też prośba ponownego przeliczenia trasy od strony drona. Może okazać się konieczne wyodrębnienie mechanizmu obliczania trasy w taki sposób, aby w razie braku połączenia dron był w stanie samodzielnie przeliczyć na nowo trasę lotu.

Poza koniecznością pokrycia brakujących przypadków użycia, istnieje także wiele możliwości dalszego rozwoju istniejącej już implementacji. Jedną z nich jest sposób dokonywania dyskretyzacji, który w obecnej formie zakłada stały podział przestrzeni. W przyszłości mechanizm ten może zostać zmieniony na bardziej elastyczny, który byłby w stanie w każdym miejscu w przestrzeni dowolnie zmieniać zagęszczenie siatki. Dzięki takiemu podejściu można by było zwiększać szczegółowość mapy w miejscach, które tego potrzebują, co mogłoby rozwiązać problem z dokładnością reprezentacji budynków.

# Bibliografia

1. **Gaździcki Jerzy**, *Systemy informacji przestrzennej*, PPWK Warszawa, 1990.
2. **Şucan Ioan A. , Moll Mark and Kavraki Lydia.** *The Open Motion Planning Library.*  IEEE Robotics & Automation Magazine, 2011.
3. **Mark Moll, et al.** *Teaching Motion Planning Concepts to Undergraduate Students.* IEEE Workshop on Advanced Robotics and Its Social Impacts (ARSO), 2011.
4. **Foley James D.**, *Computer Graphics: Principles and Practice*, Addison-Wesley Professional, 1995.
5. **Cormen Thomas H., Leiserson Charles E., Rivest Ronald L, Clifford Stein**, *Wprowadzenie do algorytmów*, Wydawnictwo Naukowe PWN, 2013.
6. **Peter E. Hart, Nils J. Nilsson, Bertram Raphael,** *A Formal Basic for the Heuristic Determination of Minimum Cost Paths*, IEEE Transactions on Systems Science and Cybernetics SSC4, 1968.
7. **Ciach Maciej, Dudek Kamil, Foremny Adrian, Furman Szymon, Bachniak Daniel**, *Projektowanie drogi lotu Quadcoptera z wykorzystaniem metod optymalizacji wielokryterialnej*, Zeszyty Studenckiego Towarzystwa Naukowego Nr 31 Publikacje Laureatów, 2015.
8. **De Smith Michael J., Goodchild Michael F., Longley Paul**, *Geospatial Analysis: A Comprehensive Guide to Principles, Techniques and Software Tools*, Troubadour Publishing Ltd, 2007.

1. Pyka Krystian, *Modele reprezentacji danych przestrzennych*. Wykład z przedmiotu Systemy Informacji o Terenie, *http://home.agh.edu.pl/~krisfoto/lib/exe/fetch.php?media=sit:4\_wektor\_raster.pdf* (dostęp 30.10.2016) [↑](#footnote-ref-1)
2. Janet Wagner, *Top 10 Mapping APIs: Google Maps, Microsoft Bing Maps and MapQuest*, Programmable Web, *https://www.programmableweb.com/news/top-10-mapping-apis-google-maps-microsoft-bing-maps-and-mapquest/analysis/2015/02/23* (dostęp 26.12.2016) [↑](#footnote-ref-2)
3. Strona domowa serwisu *OMPL*, *http://ompl.kavrakilab.org/* (dostęp 15.11.2016) [↑](#footnote-ref-3)
4. Strona domowa usługi *OSMBuildings*, *https://osmbuildings.org/* (dostęp 15.11.2016) [↑](#footnote-ref-4)
5. Oficjalna strona internetowa przeznaczona dla programistów platformy Android, *https://developer.android.com/about/dashboards/index.html#Platform* (dostęp 7.11.2016) [↑](#footnote-ref-5)
6. Oficjalna dokumentacja biblioteki LibGDX**,** *https://github.com/libgdx/libgdx/wiki/*(dostęp 15.11.2016) [↑](#footnote-ref-6)
7. Petri Kainulainen**,** *Getting Stared with Gradle*,*https://www.petrikainulainen.net/getting-started-with-gradle/*(dostęp 7.12.2016) [↑](#footnote-ref-7)
8. Oficjalna strona domowa narzędzia Maven, *https://maven.apache.org/* (dostęp 7.12.2016) [↑](#footnote-ref-8)
9. Cooper Bruce, *Activities, Tasks and Intents, Oh My!, https://www.sitepoint.com/activities-tasks-and-intents-oh-my/* (dostęp 27.12.2016) [↑](#footnote-ref-9)