Odbiorca dokumentacji: dr inż. Rafał Biedrzycki

ZPR
Zaawansowane programowanie w C++

Dokumentacja końcowa

1. Skład zespołu

Konrad Gotfryd – ds. jakości 237642 e-mail: k.gotfryd@stud.elka.pw.edu.pl
Filip Wróbel – właściciel produktu 259294 e-mail: fwrobel@mion.elka.pw.edu.pl

Adres na platformie *github*: https://github.com/evolvazpr/evolva

2. Część symulacyjna (backend)

Krótki opis tego jak działa symulacja, czego nie udało nam się zaimplementować, na jakie napotkaliśmy problemy. Nasz projekt otrzymał roboczą nazwę *evolva* – taką samą jak nie najnowsza już gra, gdzie postacie mogły ewoluować na różny sposób.

2.1 Działanie symulacji:

- pole gry ma stałe wymiary i składa się z komórek
- każda komórka może mieć dwa rodzaje podłoża: ziemię lub trawę
- podłoże jest zmienne → trawa generuje się na zasadzie automatu komórkowego (zmodyfikowana Gra w Życie Conwaya)
- każda komórka może być pusta lub zajęta przez jakiś obiekt
- obiektami są: stworzenia (*unit*), które dzielą się na roślinożerców (*herbivore*) i mięsożerców (*carnivore*), zwłoki / padlinę (*flesh*) i drzewa (*tree*)
- automatycznie generują się drzewa na takiej samej zasadzie jak trawa, dodatkowo jeżeli pole trawiaste otoczone jest samą trawą, tam też – niezależnie od zasad automatu – powstanie drzewo; oprócz tego drzewa mogą się pojawiać w losowych miejscach (częściowo zapobiega to wymieraniom drzew, a istnieje także szansa, że powstanie nowy las)
- drzewa i zwłoki mają zaimplementowane różne funkcje kwadratowe, które wpływają na zmianę ich poziomu energii (kaloryczności); w ten sposób drzewa do pewnego momentu rosną, a zwłoki w pewnym momencie znikają (powiedzielibyśmy, że się rozkładają)
- każde stworzenie ma swój kod genetyczny, który możemy nazwać kodem DNA (w rzeczywistości nie modeluje to struktury DNA, ale możemy przyjąć tę nazwę przez analogię do naszego świata); zdecydowana większość własności kodu jest znormalizowana do wartości 0 – 100
- jedną z najważniejszych właściwości, które są zapisane w kodzie jest współczynnik mięsożerności

i roślinożerności – przyjmuje się, że stworzenie jest x-żerne, jeśli właściwość x-żerności jest większa lub równa 50
- gra podzielona jest na tury, które można utożsamiać z dniami

- stworzenia mają cechy takie jak: wiek (w turach), ilość energii (utożsamiana z poziomem glukozy → ilość pożywienia), poziom zmęczenia (redukowany snem), punkty zdrowia (znormalizowane od 0 do 100)
- w każdej turze, stworzenia poruszają się według ściśle określonej kolejności – wyliczany jest specjalny współczynnik szybkości (na podstawie kodu genetycznego i np. chwilowego zdrowia czy ilości energii – im stworzenie więcej zjadło tym jest wolniejsze) i to on decyduje o kolejności
- stworzenia mogą ze sobą współżyć, a w konsekwencji mieć potomstwo (mięsożercy z mięsożercami, a roślinożercy z roślinożercami, jednak istnieje możliwość wymieszania się gatunków *)
- nie ma rozróżnienia na płcie, każde stworzenia może być matką, jednak stworzenia w ciąży nie mogą zapładniać innych stworzeń
- stworzenia mogą także jeść, walczyć (tylko mięsożercy; w celu zabicia, a następnie jedzenia), spać (w celu zredukowania poziomu zmęczenia)
- stworzenia mogą umrzeć ze starości, na skutek odniesionych w walce obrażeń, z powodu złego stanu zdrowia, z braku snu, z głodu albo podczas porodu
- jeżeli stworzeniu brakuje energii dla dziecka, które właśnie ma się urodzić, to stworzenie poroni, a z poronionego płodu nie powstaje padlina
- kiedy stworzenie umrze, pozostawia po sobie zwłoki
- roślinożercy jedzą drzewa, a mięsożercy padlinę – zwłoki
- mięsożerca, który jest głodny zaatakuję roślinożercę jeśli nie ma w pobliżu padliny
- w wyniku walki może zginąć zarówno roślinożerca jak i mięsożerca
- roślinożerca może uciec, tym samym uniknie walki
- wyliczana jest funkcja przystosowania (atrakcyjności), dzięki temu stworzenia chcą współżyć z jak najlepszymi osobnikami
- stworzenie w okresie połogu nie może zajść w ciąże, a sam połóg trwa zawsze 1 dzień
- początkowe ustawienia można wczytywać i zapisywać w formacie XML

* Każdy rodzaj stworzeń jest zainteresowany tylko stworzeniami podobnymi do siebie pod względem żerności. Aby jednostki się zaczęły krzyżować, w początkowych populacjach powinien być niezerowy współczynnik drugiej żerności, bo wtedy istnieje szansa, że podczas mutacji i krzyżowania, następne pokolenie będzie miało poziom drugiej żerności na tyle wysoki, że zostanie uznane w praktyce za wszystkożerne.

2.2. Nie udało nam się zaimplementować niektórych planowanych przez nas funkcjonalności:

- zatrutych drzew – niektóre drzewa mogłyby być zatrute, a inteligentne osobniki by lepiej je rozróżniały; z trucizną jedne stworzenia by sobie radziły lepiej, inne gorzej
- możliwości jedzenia trawy przez roślinożerców w sytuacji braku drzew
- innych zmysłów oprócz wzroku, zwłaszcza słuchu, który miał polegać na możliwości np. usłyszenia szumu drzewa (percepcji, że to drzewo jest gdzieś daleko), albo usłyszenia zagrożenia
- „inteligencji roju”, która miała polegać na tym, że jeżeli w zasięgu słuchu znajdował się osobnik o większym poziomie inteligencji, to można było go „zapytać” o to czy np. drzewo jest trujące, albo pomoc w walce z agresorem.

2.3. Co mogłoby zostać inaczej wykonane:

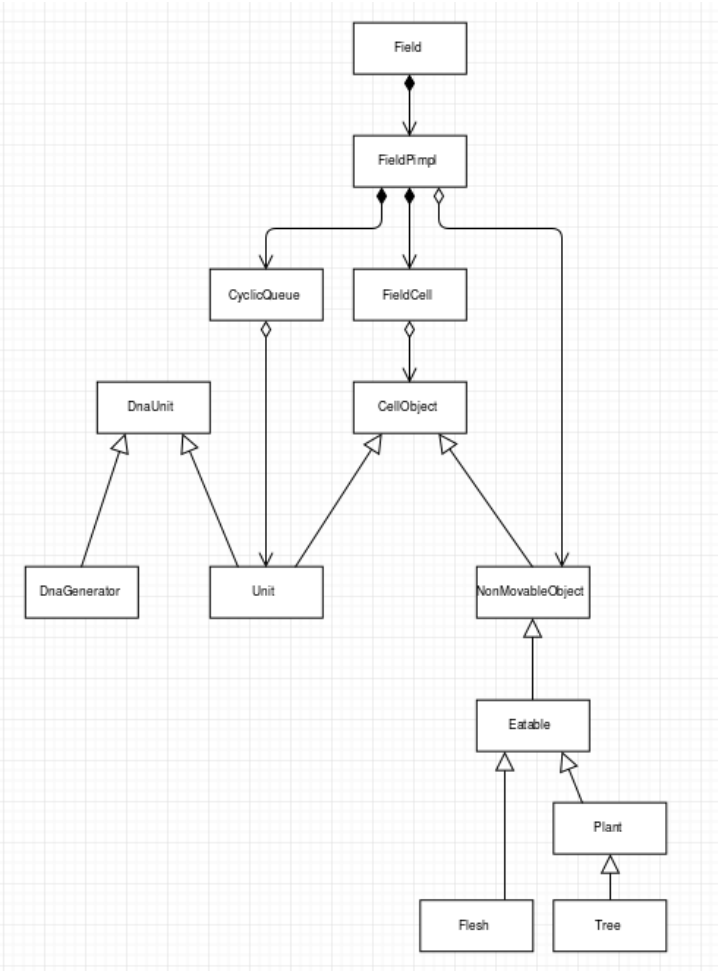
- Sposób określania typu obiektu jest niezbyt elegancki i niekonsekwentny. Używanie typu

wyliczeniowego, tablicy boolowskiej, analizy kodu genetycznego i różnych klas to raczej błąd i brak pomysłu na jednoznaczną hierarchię obiektów.

- Zmysł widzenia został zaimplementowany przy użyciu wyrażeń lambda i funktorów. Wprowadziło to niepotrzebny chaos, kod stał się niezbyt czytelny (trzeba przeskakiwać pomiędzy różnymi fragmentami kodu w czasie analizowania jednej funkcji) i trudno jest go zmodyfikować. Należałoby ten fragment przepisać i lepiej zastanowić się nad sposobem implementacji.
- Używaliśmy boost::multi_array, ale wydaje nam się, że w tym przypadku doskonale by się sprawdził podwójny std::vector.
- Styl kodowania jest miejscami niespójny. Nie stosowaliśmy z góry określonej długości linii. Kod może zostać uznany za niełady.

2.4. Na jakie napotkaliśmy problemy:

- Pewnym problemem na początku było użycie sprytnych wskaźników. Napotkaliśmy na problem z zapętlaniem się std::shared_ptr, ale użycie std::weak_ptr go rozwiązało.
- Problemem projektowym było wybranie odpowiedniego silnika graficznego, który spełniłby nasze wymagania.
- Mimo użycia sprytnych wskaźników nie uniknęliśmy problemów w wyciekami pamięci i *segmentation fault*, ale udało nam się je rozwiązać.
- W początkowej wersji *CyclicQueue* implementacja została wykonana na podstawie *std::forward_list*, jednak się nie sprawdziła, ponieważ i tak należało mieć wskaźnik na poprzedni element, co sprowadzało się do użycia *std::list*. Ta implementacja była poprawna, ale niepotrzebna, ponieważ została wykonana w ten sposób, że kolejka cały czas się zapętlała, a nowy element od razu był sortowany wewnątrz kolejki. Przez to silne stworzenia cały czas mogły coś robić podczas jednej tury (nawet kilkakrotnie), a slabe wcale nie mogły. Zmieniliśmy imlementację na *std::priority_queue* i dodatkową listę, która przechowuję stworzenia, które w następnej turze będą się poruszać. Rozwiązało to problem.
- Kompilacja trwała stosunkowo długo (głównie przez boost::multi_array). Rozwiązaniem okazało się zastosowanie idiomu *pimpl*, który jest z powodzeniem wykorzystywany w klasie *Field* i dodatkowo ukrywa część implementacji.



2.5. Opis klas, które występują w programie i diagram.

Wstęp

Zanim przejdziemy do opisu klas, wyjaśniamy, że zdecydowana większość obiektów jest przechowywana jako sprytny wskaźnik *std::shared_ptr*. Ponadto wiemy, że niektóre klasy można by było ze soba połączyć, ale nie zrobiliśmy tego, ponieważ przy takim podziale klas łatwiej by było nam zaimplementować następne klasy.

Field

Główną klasą jest klasa *Field*, która została zaprojektowana jak singleton. Ona skupia w sobie całą symulacje, jest kontenerem dla innych składników symulacji i organizuje jej pracę. Zdecydowaliśmy się na singleton, ponieważ przez analogię do naszego świata – świat jest jeden, czyli pole gry. Moglibyśmy jednak w prosty sposób zmodyfikować program, aby było możliwe istnienie kilku pól

gry jednocześnie. *Field* dodaje obiekty do mapy, pilnuje aby nie powstały kolizje, uśmierca stworzenia, wprawia w ruch drzewa i trawę (automaty komórkowe).

FieldPimpl

Wewnątrz *Field* znajduje się *FieldPimpl*, który jest obiektem *private implementation*. Dzięki niemu możliwe było ukrycie części implementacji (nie było potrzeby ukrywania całej). Ponadto skrócił się czas kompilacji, ponieważ nie musieliśmy dołączać bardzo czasochłonnych – w sensie kompilacji – bibliotek, których używa klasa *Field* np. boost. W dalszej części będziemy jednak traktować *Field* i *FieldPimpl* jak jeden obiekt aby nie wprowadzać zbytniego skomplikowania.

FieldCell

Pole fizycznie jest zbudowane z mapy obiektów klasy *FieldCell* (komórek pola gry), a sama mapa to *boost::multi_array*. Każda komórka ma jeden z dwóch rodzajów podłoża (ziemia, trawa) i może być zajęta przez jakiś obiekt, albo pusta. Oprócz tego, komórka przechowuje swoje globalne współrzędne na mapie.

CellObject

Główną bazową klasą dla wszystkich obiektów jest *CellObject*. Obiekt ma tym w postaci zmiennej typu wyliczeniowego, unikalny identyfikator obiektu, posiada także wskaźnik (tym razem *std::weak_ptr* aby uniknąć zapętlania) na komórkę, w której się znajduje.

DnaUnit

Podstawową klasą bazową dla obiektów, które posiadają kod genetyczny jest *DnaUnit*. Jest tam zawarty kod genetyczny w postaci mapy klucz → wartość, gdzie kluczami są ciągi znaków, oznaczające jaką właściwość, a wartościami liczby zmiennoprzecinkowe, które w większości są znormalizowane do wartości 0 – 100.

DnaGenerator

Klasa pochodną od *DnaUnit* jest *DnaGenerator*, która jest służy jako fabryka obietków (a wł. ich kodów genetycznych). Generator ma własny kod genetyczny i może stworzyć nowy, którego wartości będą losowe, ale będą miały zadane odchylenie. Dzięki temu cała populacja stworzeń może być do siebie bardzo podobna, ale jednocześnie każdy osobnik będzie inny.

Unit

Najważniejszą klasą, która reprezentuje stworzenie jest klasa *Unit*. Dziedziczy zarówno po *DnaUnit* jak i *CellObject*. Każde stworzenie ma swoje parametry, które nie zależą od kody DNA, np. ilość punktów zdrowia czy wiek. Klasa *Unit* daje metody, dzięki którym każdy osobnik może aktualizować swój stan (wiek, zmęczenie etc.) i pozwala stworzeniu myśleć. Metoda *Think()* jest tutaj kluczowa, ponieważ to ona jest procesem decyzyjnym, który determinuje co dana jednostka *chce* zrobić. Oprócz tego posiada metody odpowiedzialne za poruszanie się jednostki, atakowanie, kopulacje itd. Każde stworzenie może być albo roślinożercą, albo mięsożercą. Nie ma podziału na 2 klasy pochodne. Traktuje się to jako jakiś parametr.

CyclicQueue

Wszystkie stworzenia są umieszczone w kolejce – obiekcie klasy *CyclicQueue*. To klasa, która zawiera w sobie kolejkę priorytetową i drugi kontener na obiekty, które powinny się znaleźć w tej kolejce od następnej rundy. Z kolejki wyjmowane są obiekty w kolejności malejącej względem pewnej wyliczonej szybkości poruszania się. Po wyjęciu ich i obsłużeniu (metody *Update()* i *Think()*), stworzenie trafia do drugiego kontenera. Jeżeli jakieś nowe stworzenie się pojawiło podczas którejś z tur to trafia ono do kontenera, a nie do kolejki. Kiedy nastaje nowa runda (dzieje się tak kiedy kolejka jest już pusta), to elementy trafiają z kontenera z powrotem do kolejki z wyłączeniem tych, które w poprzedniej rundzie umarły. W ten sposób kolejka priorytetowa staje się cykliczna i działa aż nie umrą wszystkie stworzenia.

NonMovableObject

Obiekty, które nie są stworzeniami są typu *NonMovableObject* i w klasie *Field* są przechowywane po prostu w obiekcie *std::vector*. To tylko klasa bazowa dla innych klas, które reprezentują nie poruszajace

się obiekty.

Eatable
Klasa *Eatable* oznacza obiekty, które mogą być zjedzone. Wszystkie te obiekty mają podobny interfejs, tzn. metodę *Eat()* jak i parametry energii. Ponadto każdy taki obiekt wraz z funkcją kwadratową może zmieniać ilość energii jaką w sobie ma, ale tylko do pewnego progu.

Flesh
Każda jednostka kiedy umrze (poza płodami) pozostawia po sobie zwłoki, które utrzymują się jakiś czas dopóki się nie rozłożą. Obiektami, które są zwłokami są obiekty klasy *Fless* i to one są zjadane przez mięsożerców.

Plant
Klasa *Plant* oznacza każdy obiekt, który jest rośliną. Roślinożercy mogą jeść tylko obiekty, które są klast *Plant*.

Tree
Klasa *Tree* to wszystkie drzewa na mapie.

3. Część graficzna (frontend)

3.1. Działanie interfejsu graficznego:

- Użytkownik posiada możliwość wyboru czy nastąpi jedynie ruch pojedynczej postaci, czy też wykona się cała lub kilka rund.
- Użytkownik może zmienić szybkość poruszania się obiektów przez interfejs, lub też zmienić rozmiar wyświetlanych obiektów (przez plik logic.xml).
- Obiekty reprezentujące postacie poruszające się są animowane.

3.2. Popełnione błędy:

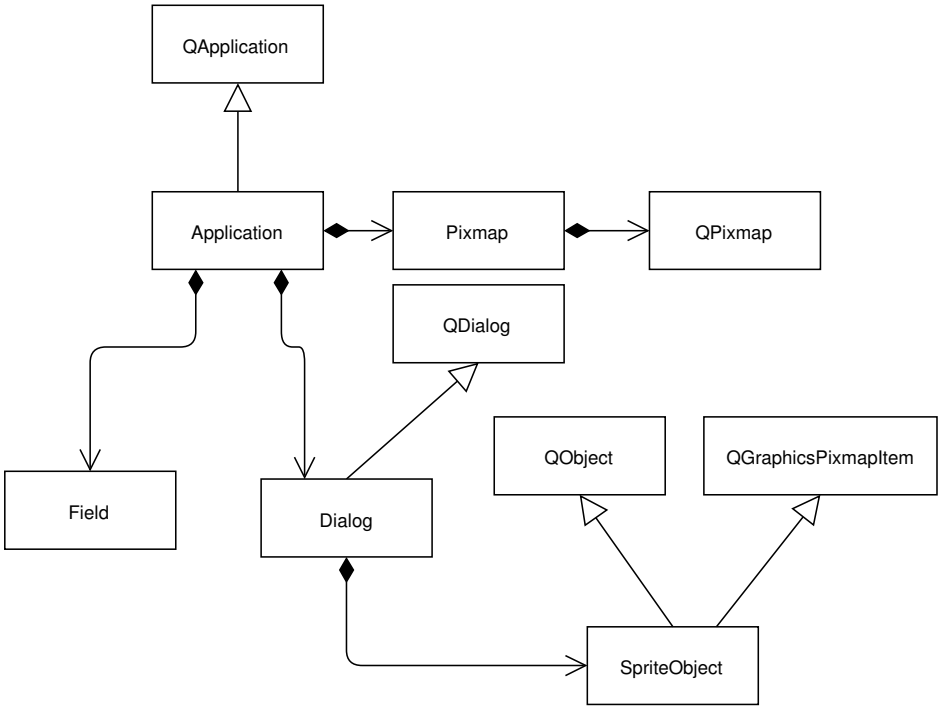
- Programowanie pod zbyt dużą presją czasu.
- Brak regulacji szybkości przełączania pomiędzy poszczególnymi klatkami animacji obiektu poruszającego się.
- Brak dostatecznej optymalizacji (metoda wyszukiwania Dialog::SearchObject jest niewłaściwa).
- Zbyt ubogi interfejs (brak możliwości dokonywania zmian grafiki z menu itp.).
- Brak zapisu wyniku symulacji.

3.3. Opis napotkanych problemów:

- Użycie sprytnych wskaźników z obiektami klas biblioteki Qt powodowało błędy w działaniu. Problem ich użycia nie został rozwiązany.
- Problem z optymalizacją. Przy jednoczesnym symulowaniu powyżej 3 rund, czas po którym uzyskuje się odpowiedź programu jest za długi. Pomimo wprowadzenia poprawek (eliminacja ciągłego wczytywania plików graficznych z dysku, eliminacja częstego alokowania i dealokowania pamięci dynamicznie) nie uzyskałem zadowalającego wyniku. Problem nie został rozwiązany, lecz stanowi ciekawe wyzwanie, które warto rozwiązać.

3.4. Opis klas, które występują w programie i diagram:

- Application – klasa zawierająca metody do inicjalizacji logiki programu, interfejsu graficznego. Stanowi proxy pomiędzy logiką a GUI.
- Pixmap – klasa zawierająca prototypy obrazków obiektów SpriteObject.
- Dialog – klasa reprezentująca okno interfejsu.
- SpriteObject – klasa reprezentująca obiekty graficzne wyświetlane w oknie Dialog.



4. Podsumowanie

Chcielibyśmy opisać w jaką stronę według nas projekt mógłby się rozwijać, a także dlaczego nie oddaliśmy projektu w terminie. Ponadto zawrzemy kilka słów podsumowania.

4.1. Nie oddaliśmy sprawozdania w terminie z kilku powodów:

- Na początku wydania projektu wykonaliśmy bardzo dużo pracy, potem nasze tempo zmalało. Byliśmy bardzo pewni, że projekt jest już prawie gotowy i na ostatnie dni przed oddaniem okazało się, że wiele rzeczy nie działa jak powinno i musieliśmy to naprawić. Zdobyliśmy pewne doświadczenie i teraz byśmy takie błędy i niepoprawne zachowania aplikacji znaleźli wcześniej, aby móc oddać projekt w terminie.
- Opuścił nas jeden z członków zespołu. Na początku rozplanowaliśmy pracę na 3 osoby, a potem musieliśmy zmienić założenia dotyczące poświęconego czasu.
- Stwierdziliśmy, że wolimy oddać projekt po czasie, ale taki, co do którego jesteśmy pewni niż terminowo oddać niesprawny produkt.

4.2. Co chcielibyśmy dodać do projektu w przyszłości i jak widzielibyśmy jego rozwój:

- Zaimplementować wszystkie rzeczy, których nie udało nam się zaimplementować w terminie, a wspominaliśmy o nich wcześniej.
- Wprowadzić więcej gatunków stworzeń.
- Rozwinąć różne własności DNA i zaimplementować to na co wpływają w symulacji.
- Zwiększyć możliwości krzyżowania się różnych stworzeń.
- Zmienić podejście wyglądu stworzeń. Przede wszystkim chodzi o możliwość generowania wyglądu stworzeń na podstawie DNA.
- Dodać do funkcji decyzyjnej jakikolwiek algorytm uczący się. Najchętniej dodalibyśmy sztuczną sieć neuronową, która modelowałaby niskopoziomowe zachowania różnych gatunków z dodatkową siecią (czy modyfikacją tej sieci), która odpowiadała by za zachowania konkretnego osobnika. Musielibyśmy wtedy poważnie przemyśleć koncepcje przekazywana „myślenia” w kodzie genetycznym.
- Dodać możliwość istnienia kilku światów jednocześnie. Chociażby dlatego, żeby można było porównać jak mogą się potoczyć losy tych samych stworzeń. jeśli będą w różnych sytuacjach.
- Rozwinąć statystyki dotyczące całego świata. Dodać rysowanie wykresów.
- Wprowadzić kod genetyczny do roślin.
- Napisać osobny algorytm, który wyszukiwałby scenariuszy gry, dla których populacje mięsożerców i roślinożerców spełniałyby równanie Lotki-Volterry.

4.3. Tabela z czasem, jaki poświęciliśmy na projekt ZPR.

Autor	Filip Wróbel	Czas	Konrad Gotfryd	Czas
Zadania	Implementacja podstawowego podziału gry na klasy	10	Implementacja podstawowego podziału gry na klasy	8
	Zaprojektowanie genetyki	5	Implementacja GUI w Qt	15
	Implementacja mechanizmów świata gry	5	Wczytywanie zapisywanie plików konfiguracyjnych w XML	10
	Implementacja reguł decyzyjnych dla stworzeń	15	Testy automatyczne, jednostkowe, scenariusze testowe	1
	Testy, analiza i diagnoza problemów, szukanie rozwiązań	15	Testy, analiza i diagnoza problemów, szukanie rozwiązań	10
Łączny czas	50		41	

Jest to więcej czasu niż planowaliśmy przeznaczyć. Stało się tak, bo musieliśmy inaczej podzielić pracę, a poza tym nie mamy jeszcze doświadczenia w estymowaniu czasu, który nam zajmie praca nad pewną częścią projektu.

4.4. Podsumowanie

Uważamy, że projekt, który zrobiliśmy jest udany. Udało nam się zrobić zdecydowaną większość rzeczy, jakie sobie wcześniej założyliśmy. Kod nie jest tak dobry, jak mógłby być i jaki wiemy, że moglibyśmy go zrobić. W szczególności brakuje dokumentacji kodu**.** Nie było dla nas obojętne, że jeden z nas zrezygnował, ale daliśmy sobie mimo wszystko radę, a to szczególna lekcja, bo nie przewidziana przez przedmiot ZPR, a mogąca się zdarzyć w prawdziwym życiu przy pracy nad jakimś projektem. Należy jednak podkreślić, że członek naszego zespołu, który nas opuścił, uczciwie nam powiedział, że nie jest w stanie zrobić swojej części, przeprosił nas i zachował się jak najbardziej w porządku w stosunku do nas. Nowością dla nas była praca zespołowa, wykorzystywaliśmy różne metody komunikacji, używaliśmy też githuba. Przy pracy nad projektem nauczyliśmy się wiele nowych rzeczy np. podstaw algorytmów

genetycznych, pracy z Qt czy kilkoma bibliotekami *boost*, których wcześniej nie znaliśmy, a także utrwaliliśmy i umocniliśmy swoją wiedzę na temat sprytnych wskaźników. Poza tym temat projektu bardzo nas zainteresował, wymagał od nas kreatywności i pozwolił nam się wiele nauczyć.