Projekt PROI - dokumentacja

Jakub Kwaśniak, Hubert Potera, Kajetan Witkowski

June 6, 2024

1 Problematyka projektu

Celem naszego projektu było stworzenie aplikacji znajdującej najkrótszą trasę między dwoma salami na Wydziale Elektroniki i Technik Informacyjnych Politechniki Warszawskiej. W tym celu utworzyliśmy model grafu ważonego sal na naszym wydziale. Ostateczna wersja projektu posiada dwa rodzaje wierzchołków - posiadające sale i nieposiadające. Zdecydowaliśmy się na taki ruch aby wyświetlanie ścieżki było przejrzyste (aby nie była "rysowana na ścianach").

2 Funkcjonalności projektu

2.1 Wyznaczanie ścieżki między salami

W celu znalezienia najkrótszej ścieżki między salami planujemy użyliśmy algorytmu Dijkstry. By umożliwić wykonanie tego zadania stworzyliśmy mapę wydziału, a następnie nanieśliśmy na nią informację o położeniu sal. Dane na temat wierzchołków zostały przez nas zapisane w plikach JSON.

2.2 Prezentowanie wyznaczonej ścieżki

W celu prezentacji trasy powstały rysunki 2D reprezentujące kolejne piętra. W trakcie działania programu, w czasie rzeczywistym rysowana jest wyznaczona przez nas ścieżka. Ze względu na problemy związane z brakiem odpowiedniej biblioteki to reprezentacji obiektów 3D, zrezygnowaliśmy z tego sposobu wyświetlania danych. Była to sytuacja, na którą byliśmy gotowi.

2.3 Informacje o salach - funkcjonalność opcjonalna

Dodatkowo dokonaliśmy implementacji funkcjonalności mającej na celu wyświetlanie informacji o danej sali. Przygotowane przez nas dane posiadają opisy nielicznych sal ze względu na fakt, iż nie był to nasz priorytet w pracy projektowej, jednakże dane są przygotowane tak, że te informacje mogą być w każdej chwili dodane.

3 Narzędzia i biblioteki

3.1 Git i GitLab

Do zarządzania kodem źródłowym oraz współpracy nad projektem wykorzystaliśmy system kontroli wersji Git. Dodatkowo do wspólnej pracy nad kodem wykorzystaliśmy również platformę GitLab, która oprócz hostowania naszego repozytorium udostępniła nam narzędzia do zarządzania projektem i planowania dalszych działań. W celu większej czytelności projektu i poszczególnych gałęzi część z nas lokalnie korzystała również z narzędzia GitLens.

3.2 SDL

SDL to prosta biblioteka programistyczna zapewniająca interfejs dostępu do sprzętu komputerowego, takiego jak klawiatury i myszki, oraz obsługę dźwięku i grafiki. Operuje ona na zdarzeniach i nie jest rozbudowana o wiele funkcjonalności, dzięki czemu jest 'lekka' oraz możliwe jest skonfigurowanie jej z

projektem z poziomu CMakeLists, czym w przypadku prostych rozwiązań zwycięża nad bibliotekami/frameworkami takimi jak Qt. Wykorzystaliśmy ją do do wyświetlania modelu wydziału i ścieżek.

3.3 Program graficzny - Krita

Początkowo do stworzenia modeli 3D na podstawie mapy wydziału planowaliśmy wykorzystać oprogramowanie Blender - oprogramowanie do tworzenia grafiki trójwymiarowej. W trakcie tworzenia projektu postanowiliśmy skupić się jednak na grafice dwuwymiarowej. W tym celu wykorzystaliśmy opensourceowy program Krita umożliwiający tworzenie grafiki rastrowej.

3.4 Biblioteka do parsowania plików *.json

Do parsowania plików *.json wykorzystywanych do przechowywania informacji o piętrach oraz salach wykładowych/ćwiczeniowych/laboratoryjnych na wydziale EITI PW została wykorzystana biblioteka nlohmann/json: JSON for Modern C++. Została przez nas uznana za najlepszą spośród 3 bibliotek (nlohmann/json, boost.json, rapidJson), w których zostały napisane wersje testowe klasy jsonReader. Nlohmann/json posiada funkcjonalność, która pozwoliła w przejrzysty sposób parsować kolejne elementy ... pliku do oddzielnych obiektów, a następnie odczytywać z nich oczekiwane pola. Co więcej gwarantuje ona wsparcie nowych wersji cmake'a oraz C++, co było dla nas kluczowe przy tworzeniu tego projektu - np. rapidJson nie posiada wsparcia dla nowych wersji cmake.

3.5 Algorytm Dijkstry

3.5.1 Opis działania

Algorytm Dijsktry jest algorytmem zachłannym, którego celem jest znalezienie najkrótszej ścieżki między dwoma wierzchołkami grafu w grafie nieskierowanym. Do realizacji tego zadania algorytm wykorzystuje kolejkę priorytetową (gdzie priorytetem jest odległość od wierzchołka startowego), a następnie na kolejnych wierzchołkach znajdujących się w kolejce algorytm wykonuje relaksację.

Algorithm 1 Algorytm Dijkstry

```
0: function DIJKSTRA(start)
0:   odwiedzone ← {fałsz} × liczba wierzchołków
0:   kolejka_priorytetowa Q ← {start}
0:   while Q niepuste do
0:     aktualny_wierzchołek ← min(Q)
0:     for następnik in aktualny_wierzchołek.następnik do
0:       if następnik istnieje then
0:         dokonaj relaksacji
0:         if odwiedzone[następnik] == fałsz then
0:           Q dodaj następnik
0:         end if
0:       end if
0:     end for
0:     odwiedzone[aktualny_wierzchołek] ← prawda
0:   end while
0: end function=0
```

3.5.2 Złożoność pamięciowa i obliczeniowa

Algorytm Dijsktry do przechowywania danych o wierzchołkach do odwiedzenia używa kolejki priorytetowej, której maksymalny rozmiar wynosi V (liczba wierzchołków). Dodatkowo posiada informacje o odwiedzonych już wierzchołkach (V) tablice odległości od wierzchołka startowego (V). Aby odtworzyć ścieżkę, którą algorytm znalazł, potrzebna jest jeszcze informacja o poprzednikach dla każdego wierzchołka (w naszej implementacji następników) (V^2). Podsumowując, złożoność pamięciowa prezentuje się następująco: $V + V + V + V^2 = 3V + V^2$ co zamyka się w dosyć korzystnym O(V^2). Warto zaznaczyć, że w naszej

implementacji część z tych informacji przetrzymuje klasa Graph, a część z tej odpowiedzialności została przeniesiona na klasę Node.

Złożoność obliczeniowa naszej implementacji algorytmu Dijkstry wynosi $O(n^2)$ ze względu na użycie przez nas tablicy do reprezentacji kolejki priorytetowej. Dodatkowo należy pamiętać, że wykonywane są jeszcze operacje relaksacji, jednak nie zmieniają one złożoności w notacji O.

4 Podjęte decyzje projektowe

4.1 Grupowanie sal

Ze względu na ogromną liczbę sal na naszym wydziale część z nich została przez nas przypisana do jednego wierzchołka. Rozwiązało to problem pojawiający się gdy sale są w niewielkiej odległości od siebie i implementacja większej ilości wierzchołków była zbędna oraz spowolniłaby działanie algorytmu. Jest to rozwiązanie mające pomijalny wpływa na dokładność obliczania ścieżek.

4.2 Podział kodu

W celu zachowania dobrych praktyk projektowych oraz przestrzegania zasad KISS, SOLID projekt został podzielony na wiele plików, zawierających pojedyncze klasy, których metody mają w miarę możliwości ograniczoną odpowiedzialność. Projekt zawiera 4 podkatalogi - src: pliki *.cpp z logiką programu, definicjami metod, include: pliki nagłówkowe *.hpp z deklaracjami klas i ich pól, assets: z plikami potrzebnymi do działania programu takimi jak pliki *.json oraz schematy pięter *.bmp, tests: lokalizacja zawierająca pliki z testami klas.

4.2.1 Obsługa plików

Wiele danych potrzebnych do funkcjonowania programu jest zapisane w plikach w formacie *.json. Istnieją więc klasy, których zadaniem jest odczytanie tych plików i zapisanie tych danych w klasach, które umożliwiają wygodną ich obsługę.

4.2.2 Część logiczna

Ta część naszego projektu odpowiedzialna była za tworzenie obiektów: Node(wierzchołek grafu), Classroom(sala lekcyjna) oraz Graph(graf). Dzięki zastosowaniu odpowiednich metod interfejsy klas umożliwiały łatwy dostęp do poszczególnych informacji, jak również, były zdolne do zmiany stanów poszczególnych obiektów w sposób wytyczający ścieżkę między dwoma wierzchołkami.

4.2.3 Interfejsy

Zbiór klas odpowiedzialnych za kontakt z użytkownikiem. Odpowiadają zarówno za pobieranie od niego danych przez argumenty wywołania (ArgParser) i konsolę (ConsoleInterface) oraz wciśnięte przyciski przy włączonym oknie (InputManager), jak i za wyświetlanie tego okna z odpowiednią ścieżką (Renderer).

4.2.4 Zarządzanie programu

Ta część zawiera główną pętlę programu i wywołuje odpowiednie metody z innych części, składając program w jedną całość (ProgramManager). Odpowiada również za przechowywanie odpowiednich danych i przetwarzanie ich w zależności od danych wejściowych użytkownika (ResourceManager, LogicManager).

4.3 Wzorce projektowe

W naszym projekcie zastosowano elementy wzorca projektowego "Łańcuch zobowiązań". Do obsługi eventów z biblioteki SDL, które są wykorzystywane między innymi do zmiany pokazywanego piętra za pomocą strzałek, użyliśmy wzorca projektowego "Polecenie". Wzorzec ten zamienia żądanie w samodzielny obiekt, który zawiera wszystkie niezbędne informacje do wykonania danej operacji.

5 Napotkane problemy

5.1 Biblioteki oraz frameworki

Niestety wiele rozbudowanych i wyspecjalizowanych bibliotek takich jak biblioteka graficzna Qt wymagają pobrania całej biblioteki oraz środowiska graficznego umożliwiającego tworzenie i edycję interfejsu graficznego dla projektu. Z pewnością jest to bardzo wygodne rozwiązanie w przypadku tworzenia całej aplikacji i wypuszczenia jej w formie wykonywalnej (*.exe), jednakże w przypadku tego projektu wiązałoby się to z oddzielną instalacją biblioteki oraz jej IDE, jak również ustawianiu odpowiednich zmiennych środowiskowych na każdym urządzeniu chcącym uruchomić projekt. Dlatego zdecydowaliśmy się skorzystać z nieco starszej i prymitywniejszej biblioteki SDL2 (Qt obsługuje obiekty, SDL2 jedynie zdarzenia), która może być pobrana i konfigurowana z projektem za pośrednictwem narzędzia cmake.

6 Zbudowanie, uruchomienie oraz testowanie projektu

6.1 Zbudowanie projektu

Należy utworzyć katalog *build* w katologu głównym projektu i przejść do niego *cd build*. Znajdując się w katalogu build należy skonfigurować cmake'a (polecenia pliku CMakeLists.txt) poprzez komendę *cmake* .. (konfiguracja bez testów) lub *cmake* $-D\ BUILD\_TESTS = ON$ .. (konfiguracja z testami), po wykonaniu cmake'a można zbudować projekt poprzez komendę *cmake* $--$*build* . - projekt zbuduje się razem z testami jedynie jeśli podczas konfiguracji została dodana flaga $-D\ BUILD\_TESTS = ON$.

6.2 Uruchomienie projektu

Logikę programu (po zbudowaniu projektu, znajdując się w folderze *build*) uruchamiamy poprzez komendę $./main$.

6.3 Testowanie projektu

Testy programu (po zbudowaniu projektu, znajdując się w folderze *build*) uruchamiamy poprzez komendę $./test/TESTS$.