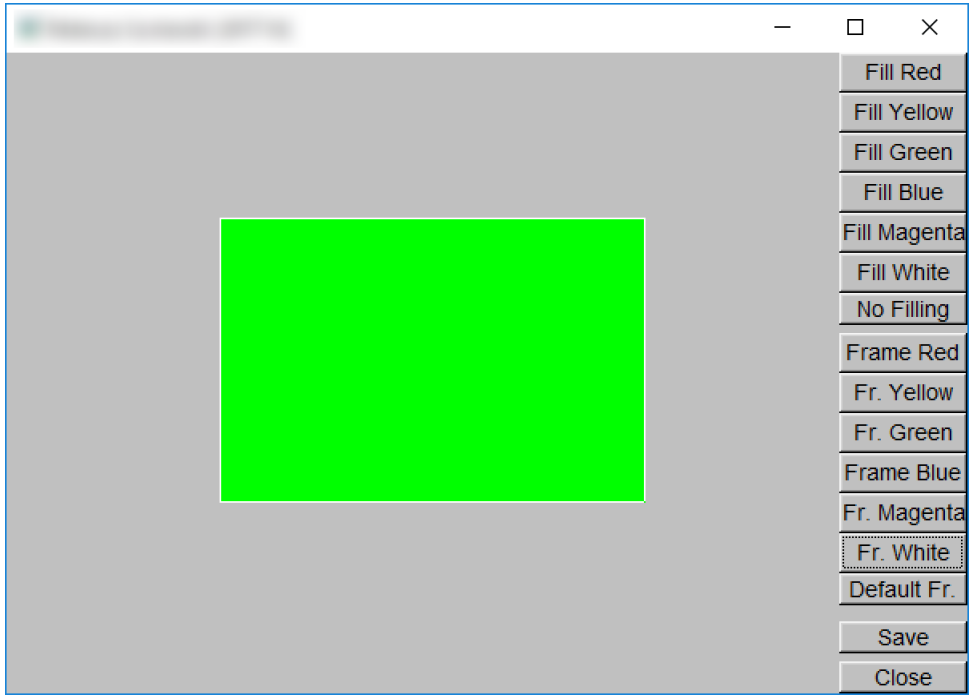
Obsługa menu w 2^5 kroków

Inspiracją do kolejnego zadania było rozwiązanie zadania, które dostałem w jednej z poprzednich edycji:



Niewątpliwie wodotrysków dodanych do rysowania prostokątów jest tutaj sporo (dawały dodatkowe dwa punkty), ale zostały wykonane brutalnymi metodami. Można zrobić to zgrabniej, oszczędzając i miejsce w oknie, i ilość pisanego kodu.

Problem mnóstwa przycisków

Mnogość przycisków zajęła nie tylko cenne miejsce w obszarze okna, ale też dała mnóstwo, bardzo do siebie podobnych, fragmentów kodu. O ile na tym etapie jest zrozumiałe, że każdy przycisk miał swoją osobną funkcję *callback* (`cb_fillred`, `cb_fillyellow`, itd.), to powielenie podobnego zestawu bezparametrowych metod w `myWindow` było niewątpliwie wynikiem inercji (i sprawnego działania schowka).
W tym zadaniu trzeba utworzyć specyficzne „menu” wykorzystujące przyciski z biblioteki `Graph_lib`. Żeby skoncentrować się na podstawowej części zadania, w obszarze rysowania umieścimy tylko jeden kształt (prostokąt). Temu prostokątowi będziemy zmieniać kolor wypełnienia i konturu korzystając z dwóch menu do zmiany koloru.

Funkcjonalność menu

Funkcjonalność menu, którą chcemy uzyskać można podsumować w trzech punktach:

1. Kliknięcie przycisku menu powinno spowodować wyświetlenie pod nim szeregu przycisków ustawiania koloru (rozwinięcie menu).
2. Kliknięcie przycisku menu przy rozwiniętej liście przycisków powinno zwinąć menu.
3. Kliknięcie przycisku koloru powinno zwinąć menu i ustawić stosowny kolor kształtu.

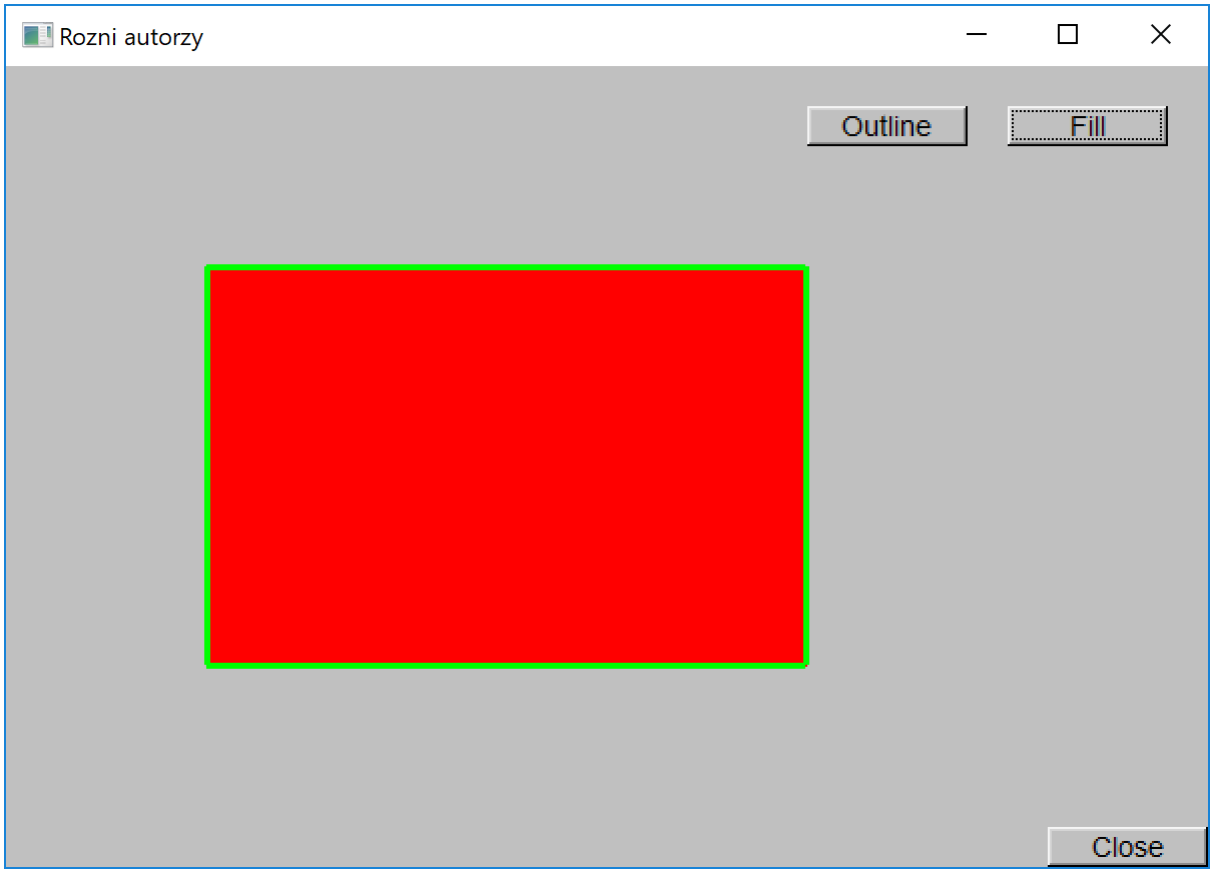
Tak wygląda to z punktu widzenia użytkownika; faktycznie, kliknięcia będą po prostu generować komunikaty przesyłane do właściwego adresata, po to, żeby na końcu obiekt okna zainicjował odpowiednią akcję.

Wymagania niefunkcjonalne

W trakcie tworzenia programu chciałbym:

1. Użyć możliwie niewielu funkcji `callback`.
2. Posługiwać się dobrze zdefiniowanymi strukturami (nazwy składowych) i wyliczeniami (coś mówiące identyfikatory).
3. Implementowanie metod i funkcji tylko w pliku `mywindow.cpp`.
4. Kompilować tworzony kod możliwie jak najczęściej; uruchamiać go o ile tylko wynik kompilacji na to pozwala.
5. Po uzyskaniu pełnej funkcjonalności sprawdzić ją uważnie a następnie przejrzeć kod i uporządkować go.

Efekt końcowy



Pierwsze kroki

32 kroki prowadzące do końcowego programu wyglądają następująco:

1. Utworzenie nowego projektu (`zad_8`) na podstawie „standardowego” szablonu z biblioteką `Graph_lib`.
2. Zmiana nazwy pliku testu na `zad_8.cpp`.
3. Utworzenie klasy `MenuWindow`, dziedziczącej publicznie po `Graph_lib::Window`, rozdzielonej na plik `menuwindow.h` (definicje klas, struktur i wyliczeń) oraz `menuwindow.cpp` (implementacje metod).
4. Dodanie przycisku `Close` do zamykania okna i sprawdzenie jego działania.
5. Dodanie kształtu (`Graph_lib::Rectangle`) jako standardowej składowej `MenuWindow`; inicjowanie w liście inicjacyjnej konstruktora `MenuWindow`.

6. Zdefiniowanie klasy `MenuHeader` dziedziczącej publicznie po `Graph_lib::Button`.
7. Zdefiniowanie konstruktora klasy `MenuHeader`:

```
MenuHeader(Graph_lib::Point loc, int w, int h,
    const std::string& label);
```

   Jak widać, w konstruktorze nie ma wskazania na funkcję `Callback` – w tworzonym w konstruktorze przycisku trzeba ustawić ją na `nullptr`.
8. Dodanie do `MenuWindow` składowej `btn_fill` (typu `MenuHeader`) i zainicjowanie jej w konstruktorze. Z pewnych względów umieśćmy górny lewy róg we współrzędnych (50, 50).
9. Sprawdzenie, że przycisk `Fill` jest widoczny w oknie (trzeba pamiętać o wywołaniu `attach()` – jak dla zwykłego przycisku – w konstruktorze `MenuWindow`).

10. Ponieważ informację o etykietach przycisków i kolorach chcę przekazywać do funkcji `MenuHeader::attach()`, to najpierw należy zdefiniować strukturę `colorSpec`:

```
struct colorSpec
{
   std::string      label;
   Graph_lib::Color color;
};
```

11. Teraz mogę zająć się funkcją `attach()` `wmMenuHeader`:

```
void attach(MenuWindow* pWnd,
   const std::vector<colorSpec>& btns);
```

12. Do wywołania będą mi potrzebne kolory; definiuję stosowną statyczną i prywatną składową w klasie `MenuWindow`:

```
static std::vector<colorSpec> fill_colors;
```

13. W pliku `myWidow.cpp` dokładam definicję i inicjalizację `fill_colors`:

```
std::vector<colorSpec> MenuWindow::fill_colors{
    { "No fill", Color::invisible },
    { "Red", Color::red },
    { "Green", Color::green },
    { "Blue", Color::blue },
};
```

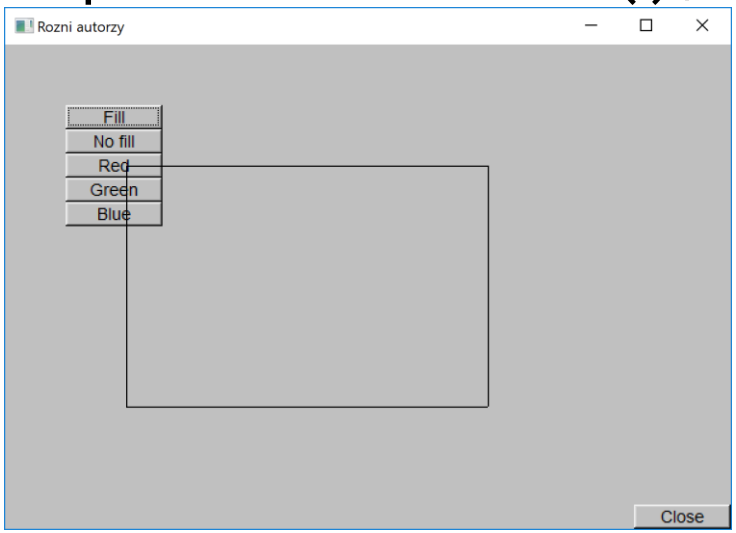
14. Dokładam definicję `MenuHeader::attach`, w której:
    - Zapamiętuję wskazanie na obiekt `MenuWindow`, w którym jest menu,
    - Tworzę wszystkie przyciski kolorów (korzystam z wektora wskazań na `Graph_lib::Button`),
    - Skoro pojawiło się new, to od razu tworzę destruktor `MenuHeader` i wołam `delete` dla wszystkich przycisków,
    - Dołączam przycisk menu do tego okna,
    - Zmieniam wywołanie `attach` w konstruktorze okna.

(w tym momencie przycisk menu powinien już być widoczny w oknie)

Dalsze kroki

15. Dołożenie funkcji `Callback` do `MenuHeader` (nazwałem tę funkcję `cb_openClose`, bo będzie odpowiedzialna za otwarcie i zamknięcie menu).
    Tu trzeba pamiętać o dołożeniu w `MenuHeader::attach` informacji o funkcji zwrotnej w widgecie oraz o ustawieniu wartości drugiego parametru, który będzie przekazany do funkcji `Callback`:
    ```
    pw->callback(reinterpret_cast<Fl_Callback*>(cb_openClose),
    this);
    ```
    Na konsoli widzę, że funkcja jest wywoływana po naciśnięciu przycisku `Fill`, zatem pora na wyświetlenie przycisków. Definiuję publiczną metodę `showMenu`, która ma „przypiąć” przyciski do okna.
17. Wszystko zrobione należycie, ale przyciski się nie pojawiają ☹
    Po sprawdzeniu, że przyciski mam, że są ustawione, że mają właściwe współrzędne i etykiety dochodzę do wniosku, że okno

najpewniej nie wie, że powinno się odświeżyć. Dodaję w `showMenu` wywołanie `pParent->redraw()`, i:



18. Jest nieźle, ale `attach` (przypinanie widgetu do okna) jest dość kosztowną operacją. Podpięcie przycisków do okna przerzucę do `menuHeader::attach`, natomiast w `showMenu` będę wywoływać funkcję `show` (a w `hideMenu` za moment – `hide`).
19. Wyszło, ale nie bardzo, bo po utworzeniu widzę menu już rozwinięte. Implementuję od razu `menuHide`, żeby wywołać w `menuHeader::attach` po utworzeniu i podpięciu przycisków do okna.

Jako bonus przychodzi możliwość usunięcia z `showMenu` `pParent->redraw()` (w `hideMenu` nawet go nie umieszczałem).

20. Ponieważ menu powinno „wiedzieć” czy jest rozwinięte, czy nie, dodaję składową `expanded` (typu `bool`) i ustawiam ją stosownie w `showMenu` i w `hideMenu`.
21. Pora zająć się problemem rysowania menu pod prostokątem. To kolejna oznaka, że o rozwijaniu i zwijaniu menu powinno decydować okno. Trzeba wziąć pod uwagę, że okno będzie mieć co nieco do obsługi:
    a. Otwieranie menu
    b. Zamykanie menu
    c. Ustawianie koloru

    I to w dodatku docelowo z więcej niż jednego menu.

Zacznę od przygotowania struktury, która będzie zbierać wszystkie informacje potrzebne dla okna (i menu):

```
struct actionDescriptor
{
    enum Action { NoAction, Menu_open, Menu_close, Menu_select };
    MenuWindow  *pParent = nullptr;
    MenuHeader  *pMenu = nullptr;
    Action      menu_action = NoAction;
    Color       selected_color = Color::invisible;
};
```

Uwaga: Żeby pomieścić się na stronie trochę uprościłem tę definicję – uzupełnienie brakujących elementów jest elementem zadania ☺.

Zwróćcie uwagę, że wszystkie składowe struktury są inicjowane przy definicji – to oznacza, że domyślny konstruktor „zastanie” już te wartości ustawione. Czyli nie musimy go w ogóle definiować!

22. Mając taką strukturę, będę mógł przekazać do okna (do nowej metody publicznej `menuAction`) komplet informacji. Definiuję tę metodę – na razie tylko z obsługą akcji `Menu_open` i `Menu_close`.

23. Teraz porządki w funkcji `cb_openClose` i jej parametrach. Po pierwsze, chciałbym żeby jako drugi parametr do tej funkcji trafiło wskazanie na deskryptor akcji (a tam znajdę wskazanie na obiekt `MenuHeader` i na obiekt `MenuWindow`).
    Do `MenuHeader` dodaję składową `mAction` typu `actionDescriptor`. Wartości wskazań w tym obiekcie inicjuję w `MenuHeader::attach` i podaję wskazanie na tę składową do przekazania funkcji zwrotnej:
    ```
    pw->callback(reinterpret_cast<Fl_Callback*>(cb_openClose), &mAction);
    ```
24. Teraz w `cb_openClose` wystarczy mi ustawić w deskryptorze akcję i wywołać `menuAction` z `MenuWindow`:
    ```
    void menuHeader::cb_openClose(Address, Address pDsc)
    {
        actionDescriptor *pAD = reinterpret_cast<actionDescriptor*>(pDsc);
        if (pAD->pMenu->expanded)
            pAD->menu_action = actionDescriptor::Menu_close;
        else
            pAD->menu_action = actionDescriptor::Menu_open;
        pAD->pParent->menuAction(pAD);
    }
    ```

25. Działanie nie zmieniło się, ale teraz mogę powalczyć z prostokątem. Jeśli mam zamiar wyświetlić menu, to trzeba „odczepić” kształt od okna – nie będzie wtedy odrysowywany po widgetach. Przy zamykaniu menu kształt trzeba „przyczepić” ponownie.
26. Wreszcie można zająć się kolorami. Rozwiązanie z deskryptorem jest niezłe i chciałbym je zastosować również tutaj. Żeby nie mnożyć nad miarę deskryptorów użyję zdefiniowanego już deskryptora akcji. Jako że taki deskryptor ma mieć każdy przycisk, to potrzebuję nową klasę `MenuItem`, która dziedziczy publicznie po `Graph_lib::Button`.

Jej definicję widzę tak:

```
class MenuItem : public Graph_lib::Button
{
public:
    MenuItem(Graph_lib::Point loc, int w, int h,
                  const std::string& label);
    void attach(MenuWindow* pWnd, MenuHeader *pMenu,
                Graph_lib::Color color, Graph_lib::Callback cb_setColor);
private:
    actionDescriptor buttonAction;
};
```

27. Konstruktor jest oczywisty. W `attach()` należy:
    a. Ustawić sensownie wszystkie składowe `buttonAction`
    b. Przypiąć przycisk do okna
    c. Wskazać jako funkcję zwrotną `cb_setColor`, a jako jej argument przekazać wskazanie na `buttonAction`

    Kolejność punktów b i c jest bardzo istotna.
28. Po usunięciu ewentualnych błędów kompilacji można ruszyć do zmian w `MenuHeader`. Najpierw trzeba zdefiniować statyczną

metodę `cb_setColor`. Jej jedynym zadaniem jest przekazanie opisu akcji do `menuAction` w `MenuWindow`.

29. Więcej zabawy będzie z wymianą przycisków typu `Button` na przyciski typu `MenuItem`. Najszybszą, choć trudną dla osób nerwowo reagujących na błędy kompilacji, drogą jest wymiana typu `Graph_lib::Button` na `MenuItem` w deklaracji składowej `buttons` w definicji `MenuHeader` i usunięcie błędów kompilacji, które to spowoduje.
30. Niestety, wywołanie niewłaściwego `attach` dla `MenuItem` w `MenuHeader::attach` nie powoduje błędu kompilacji. Aktualne wywołanie:
    ```
    pWnd->attach(*buttons.back());
    ```
    zamieniam na:
    ```
    buttons.back()->attach(pWnd, this, btn.color, cb_setColor);
    ```
31. Wszystko działa nienagannie, ale kolory się nie zmieniają. To dość oczywiste, bo `MenuWindow::menuAction` nie obsługuje jeszcze akcji `Menu_select`. Przy ustawianiu koloru trzeba

zamknąć menu, ustawić kolor wypełnienia (albo konturu) i przypiąć prostokąt ponownie do okna.

32. Działa cudnie. Zostaje przesunąć menu na docelową pozycję, dodać drugie dla koloru konturu i przystosować `menuAction` do obsługi dwóch menu wyboru koloru. To ostatnie, choć wygląda niewinnie, wymaga więcej pracy, niż się wydaje: trzeba zadbać o to, żeby w danym momencie było rozwinięte tylko jedno menu.

Oddawanie

Rozwiązanie zadania składa się z 2 plików:

`menuwindow.h`	– zawiera definicje klas przycisku, menu oraz okna i struktur pomocniczych
`menuwindow.cpp`	– implementacje klas przycisku, menu oraz okna
`zad_8.cpp`	– funkcja `main` (tej funkcji nie oddajecie Państwo).

Rozwiązanie proszę załadować w `Moodle` do:

28 maja 2023 23:59