Animacja

Zadanie kolejne polega na zaimplementowaniu animacji odczytanej z pliku mapy. Zupełnie wystarczające będzie obracanie mapy względem środka obszaru prezentacji.

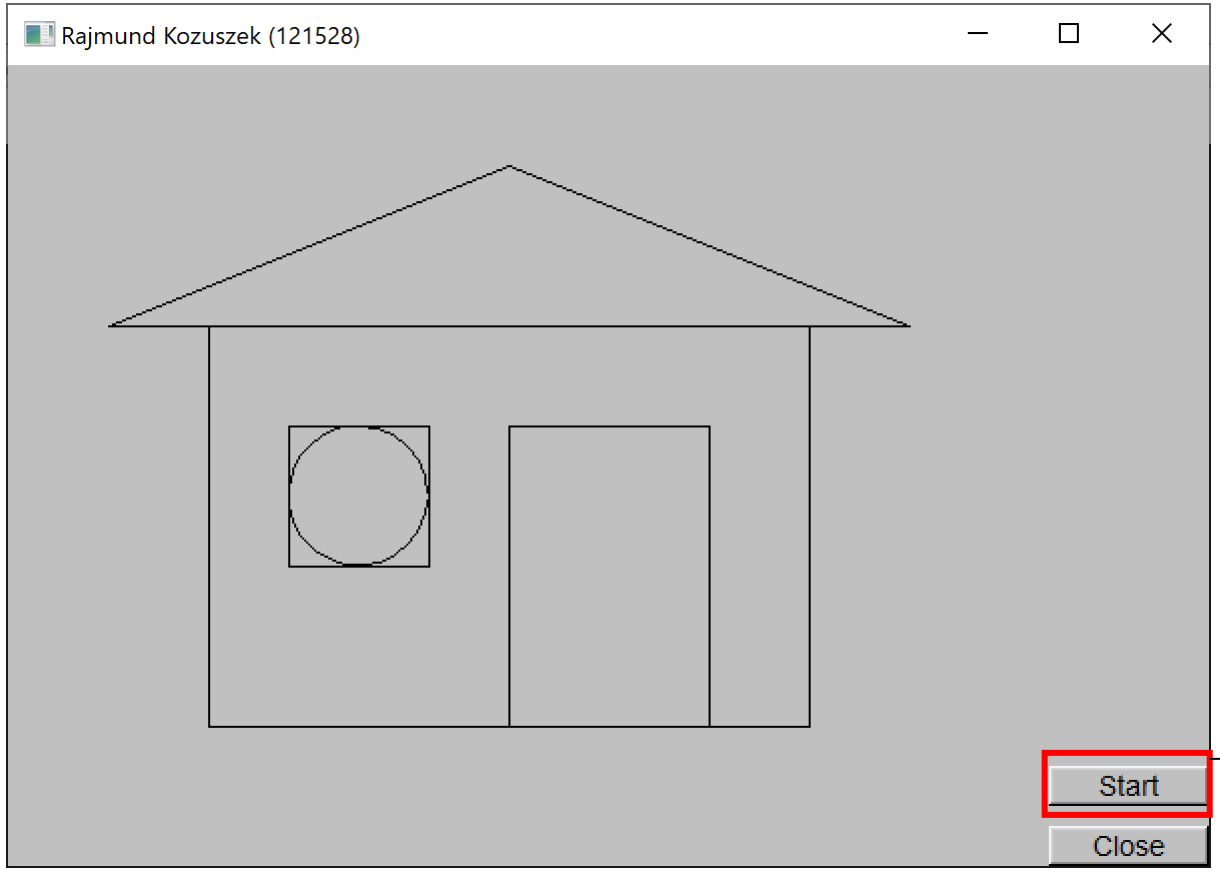
W tym zadaniu będzie istotne zarządzanie pamięcią: wycieki pamięci przy animacji mogą mieć fatalne skutki.

Minimum wystarczającym do zaliczenia tego ćwiczenia (a nawet uzyskania najwyższej liczby punktów) jest właściwa obsługa transformacji dla łamanej (`Line`).

Za każdą dodatkową figurę (maksymalnie 2), która będzie zgrabnie animowana w oknie będzie można dostać dodatkowy punkt (ostatnie małe zadanie).

Ważne jest dopracowanie szczegółów: odpowiednia obsługa zdarzeń timera i zadbanie o aktualizację elementów interfejsu (przycisk Start/Stop).

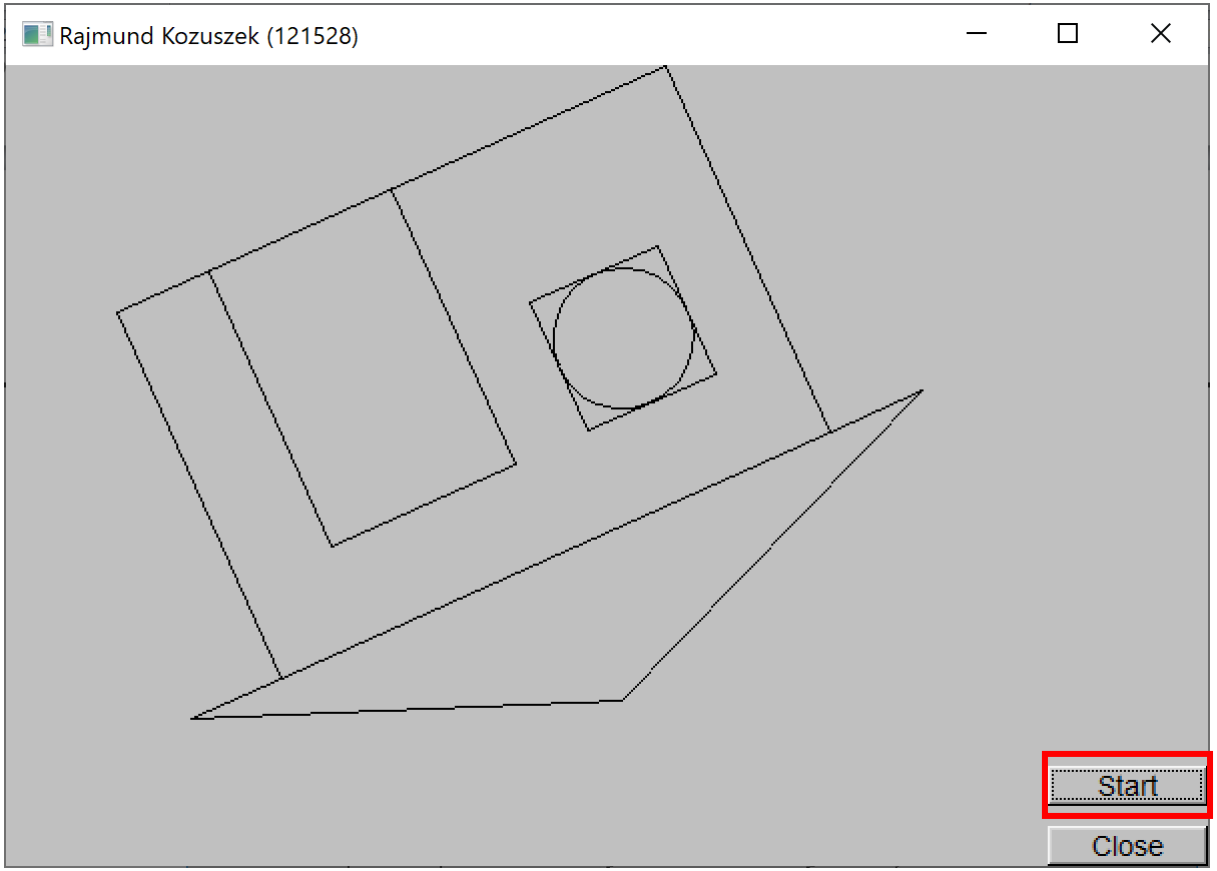
Okno animacji



Zmieniana etykieta przycisku

Uwaga: przyjmijmy, że środek obrotu jest w środku pola prezentacji; u mnie jest to punkt (250,200).

Okno animacji



Elementy animacji

Do zrealizowania animacji trzeba zaimplementować macierze transformacji geometrycznych.
Szczegóły są dokładnie omówione w wykładzie #8.
Infrastrukturę, która pozwoli nam na zrealizowanie animacji, omówimy w trakcie pierwszej części ćwiczeń.
Żeby sprawy nie komplikować zanadto przyjmiemy, że wszystkie obiekty sceny będą animowane tak samo, tzn. będą obracać się wokół ustalonego punktu (finalnie) – wyznaczenie macierzy transformacji będzie stosunkowo proste i pozwoli skoncentrować się na innych elementach związanych z animacją. Przyjmujemy, że scenę czytamy z pliku (wracają figury!) i że użytkownik może uruchamiać i zatrzymywać animację w dowolnym momencie.

Jak nie animować

To ostatnie założenie wyklucza podejście, które początkującym animatorom narzuca się z ogromną nachalnością:

```
while (nie_koniec_animacji)
{
   Zmodyfikuj_parametry_animacji
   Wyznacz_macierz_transformacji
   Przelicz_współrzędne_figur
   Odepnij_poprzedni_zestaw_kształtów
   Przypnij_nowy_zestaw_kształtów
}
```

Podstawowy problem tego podejścia, to brak przekazania sterowania do biblioteki graficznej (czyli nie dość, że musimy określić *„koniec animacji”* wcześniej, to w trakcie animacji aplikacja nie będzie odpowiadać na akcje użytkownika).

Zdarzenia związane z czasem

Warto wspomnieć o jeszcze jednym mankamencie poprzedniego podejścia: szybkość animacji zależy od wydajności komputera, na którym działa program.
Potrzebna nam będzie informacja o upływie czasu, najlepiej dostarczona z biblioteki obsługującej okna – chodzi o to, żeby pomiędzy momentami (krótkiej) aktywności naszego kodu związanej z animacją interfejs użytkownika odpowiadał na akcje użytkownika.
Podstawową funkcją w `fltk`, która pozwala na uruchomienie wskazanej funkcji po upływie zadanego czasu, jest:

```
void Fl::add_timeout(double t, Fl_Timeout_Handler cb,
                     void * argp = 0)
```

Zdarzenia związane z czasem

Tajemnicze `Fl_Timeout_Handler` jest zdefiniowane tak:

```
typedef void(* Fl_Timeout_Handler) (void *data);
```

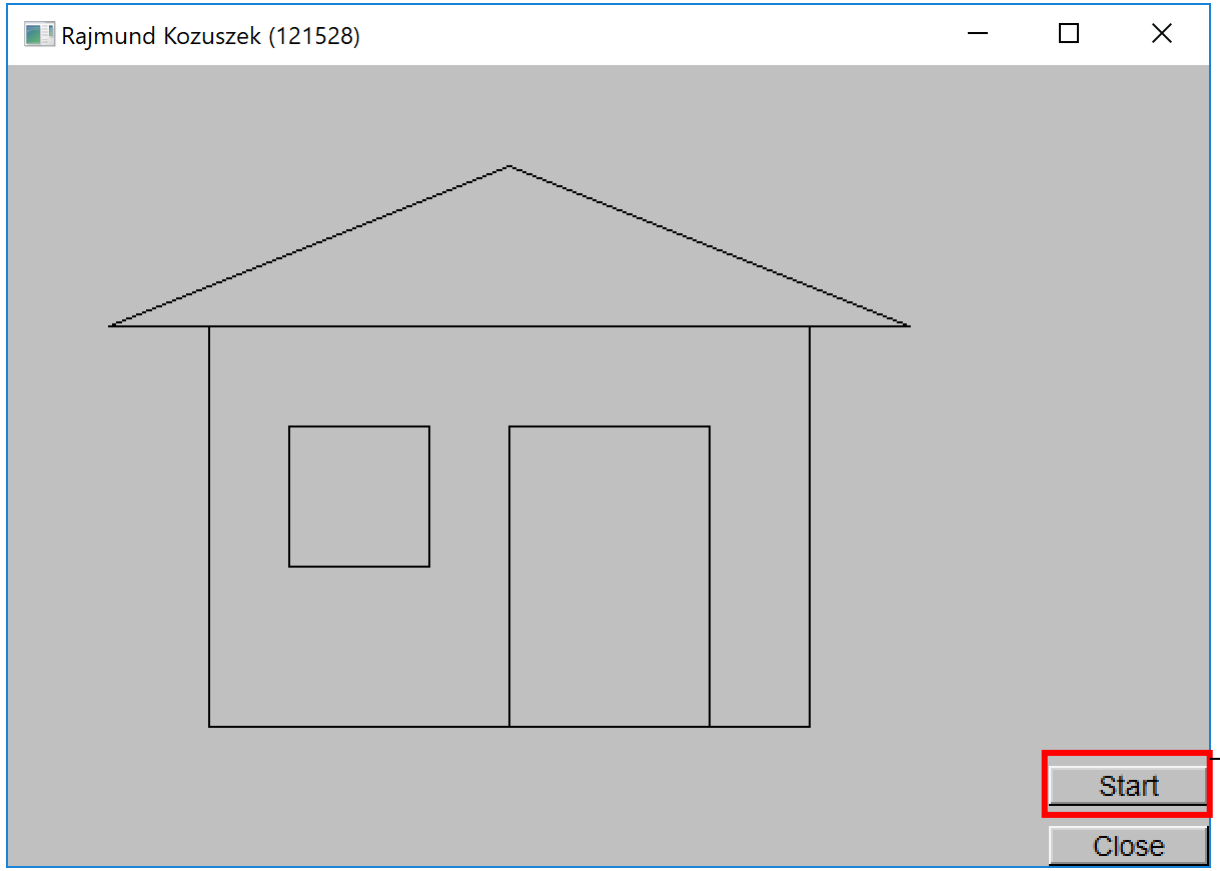
Ewidentnie funkcja zwrotna z jednym parametrem (i ani chybi argument `data` dla funkcji zwrotnej zadajemy w `argp`).

`add_timeout` wystarczyłby na dobrą sprawę do animacji, ale mamy jeszcze drugą funkcję:

```
void Fl::repeat_timeout(double t, Fl_Timeout_Handler cb,
                        void * argp = 0)
```

Różnica między `add` i `repeat` jest subtelna: `add` odlicza czas od momentu wywołania, natomiast `repeat` od zakończenia poprzedniego odliczania czasu.

Interfejs programu



Zmieniana etykieta przycisku

Uwaga: przyjmijmy, że środek obrotu jest w środku pola prezentacji; pamiętamy go oczywiście w obiekcie klasy myWindow.

Główny animator

Głównym animatorem jest funkcja `callback`, która zadaje, za jaki czas ma być „wzbudzona” ponownie:

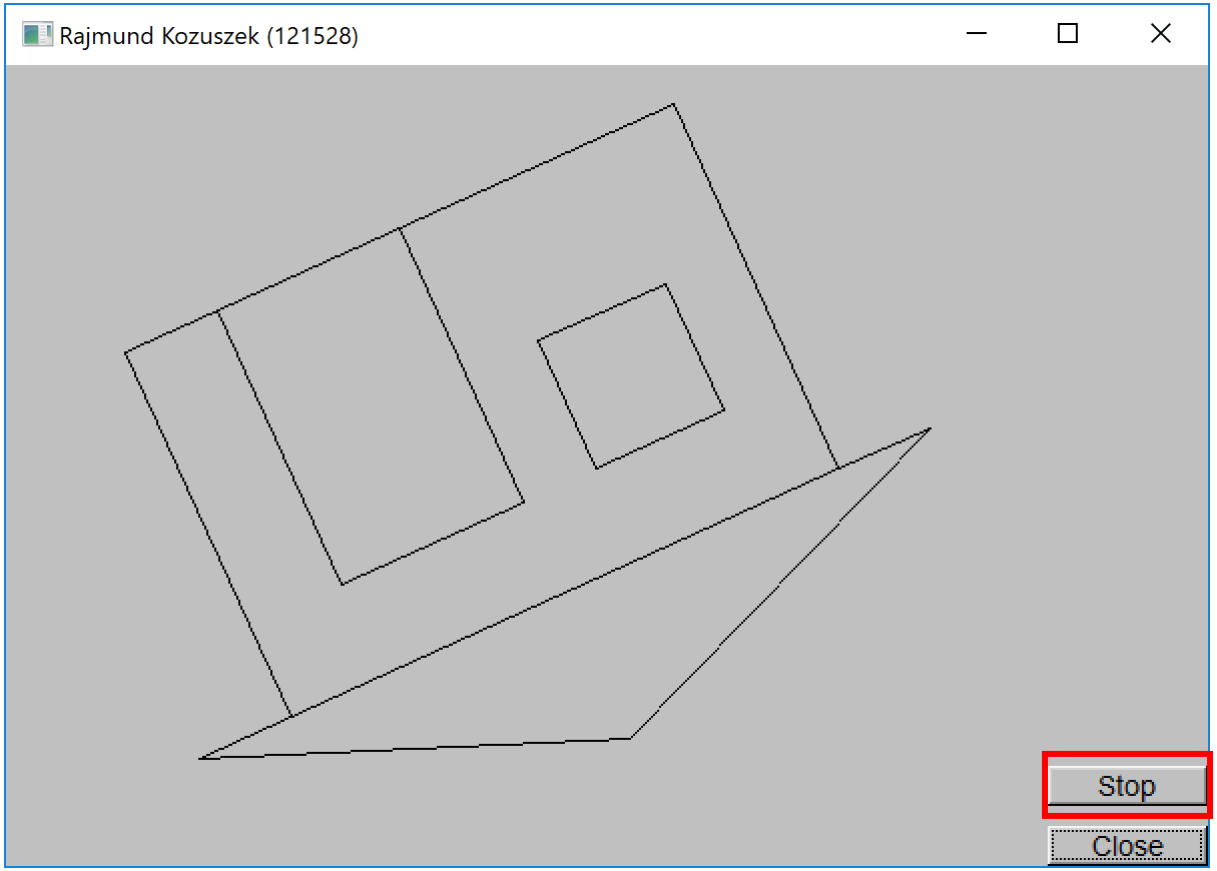
```
void myWindow::timer_callback(Address addr)
{
   myWindow *pWnd = static_cast<myWindow*>(addr);
   pWnd->rotationAngle += 0.05f;
   pWnd->refreshMap();
   if (pWnd->animationRunning)
      Fl::repeat_timeout(0.25, timer_callback, pWnd);
}
```

„Parametry” animacji

Żądanie ponownego uruchomienia funkcji po określonym czasie.

Bez pierwszego uruchomienia tej funkcji się nie obejdzie, ale sprawa jest dość oczywista: w reakcji na naciśnięcie `Start` wołamy `add_timeout` (to oczywiście niejedyna rzecz, jaką trzeba zrobić po naciśnięciu `Start`).

Problemy z kształtami



Zaznaczony na obrazku przycisk ma zmienioną etykietę, ale to najmniejszy problem. Czy widzicie o wiele poważniejszy?

Wszystko łamaną

Oczywiście, prostokąt zamienił się w łamaną!

Na pierwszy rzut oka może się wydawać, że podejście do prostokątów i kół będzie wymagało sporo dodatkowego zachodu: wyciągamy za pomocą `get_shape()` stosowny kształt i kombinujemy co dalej.
Ja sugerowałbym rozwiązanie śmiałe i oszczędzające sporo pracy. Klasę `figure` rozbudowałbym o wirtualną metodę `get_points`, dającą wektor punktów łamanej reprezentujący daną figurę. Domyślna implementacja zwraca po prostu kopię `fdef` (czyli jest dobra dla `Line`), natomiast dla `Rect` trzeba wyliczyć stosowną *piątkę* punktów (używamy `Open_polyline`). Z `Circ` będzie jeszcze więcej zabawy ☺
Ale za to wyświetlanie będzie proste!
(Jedna funkcja transformująca punkty, bo tylko punkty nam zostały).

Zatrzymanie animacji

Pozostaje ustalić, w jaki sposób zatrzymać animację.

Jeśli scena kręci się w oknie, to informację o naciśnięciu `Stop` (to ten sam przycisk, co `Start`, tylko ma inną etykietę) dostaniemy, kiedy `timer_callback` czeka, aż upłynie czas do wyświetlenia kolejnej ramki animacji.

Podpowiedź jest zawarta w kodzie `timer_callback`: do zatrzymania animacji wystarczy nie wołać ponownie powtórki; sprawę powinno zatem załatwić ustawienie `animationRunning` na `false`.

Zostanie jeszcze delikatna sprawa przycisku `Close` – czy można bez szkody zamknąć okno w trakcie animacji?

Reorganizacja kodu

Projekt jest stosunkowo niewielki, ale już teraz pojawia się w nim bardzo niedobry objaw: wszystko zależy od wszystkiego.

Żeby skorzystać z `FPoint` lub `figure`, musimy włączyć do projektu `graph_lib`, bo obie klasy zależą od `graph_lib`.

Damy temu odpór reorganizując kod: definicja `FPoint` pójdzie do osobnego pliku, a wiązanie z `graph_lib` usuniemy, usuwając operator konwersji typu (zdefiniujemy go „przy oknie”).

Definicje figur (i funkcji pomocniczych) zostaną w plikach `figure`, a powiązanie z `graph_lib` wyrugujemy pozbywając się metody `get_shape` (zastąpi ją `get_points`, która zwraca wektor `FPoint`).

Macierze powinny zależeć co najwyżej od `FPoint`.

W ten sposób `graph_lib` będzie potrzebne tylko i wyłącznie w plikach `drawing` (implementujących `myWindow`).

Oddawanie

Rozwiązanie zadania składa się z 8 plików:

`fpoint.h`	– zawiera definicję FPoint i funkcji operujących
`fpoint.cpp`	– zawiera implementacje FPoint
`figure.h`	– zawiera definicje klas figur i deklaracje funkcji pomocniczych
`figure.cpp`	– implementacje klas figur i f. pomocniczych
`drawing.h`	– definicje klas obsługi interfejsu
`drawing.cpp`	– implementacje klas obsługi interfejsu
`matrix.h`	– implementacja szablonu klasy `matrix`
`anim_test.cpp`	– funkcja `main`.

Rozwiązanie (spakowane w archiwum zip) proszę złożyć w Moodle do:

14 maja 2023 23:59