Dokumentacja projektu

Bryły Obrotowe

(Solid of Revolution)

Autorzy:

Paweł Noga

Tadeusz Raczek

Łukasz Rados

1. Tytuł oraz autorzy projektu

Celem projektu „*Bryły obrotowe*” jest stworzenie oprogramowania wyświetlającego szkieletowe bryły obrotowe w rzucie prostokątnym lub perspektywicznym. Bryły są generowane w wyniku obrotu wprowadzonej w innym oknie dwuwymiarowej figury: krzywej lub łamanej.

Projekt realizowany jest przez 3 osobowy zespół w składzie:

- Paweł Noga – testowanie kodu, tworzenie GUI oraz nadzór nad dokumentacją
- Tadeusz Raczek – algorytmy obracania i rysowania bryły obrotowej, tworzenie animacji
- Łukasz Rados – algorytmy rysowania figur - krzywych, przygotowanie danych do obrobki

2. Opis projektu

Naszym celem jest stworzenie programu pozwalającego użytkownikowi narysowanie dowolnej krzywej lub figury. Następnie obracamy narysowany kształt wokół osi OZ tworząc tym samym bryłę obrotową. Bryłę można obracać wokół dowolnej osi, z uwzględnieniem niewyświetlania zasłoniętych krawędzi. W każdej chwili bryłę można zapisać do pliku. Dodatkową funkcjonalnością jest możliwość tworzenia animacji obracanej bryły oraz drukowanie aktualnego widoku.

3. Założenia przyjęte przy realizacji projektu

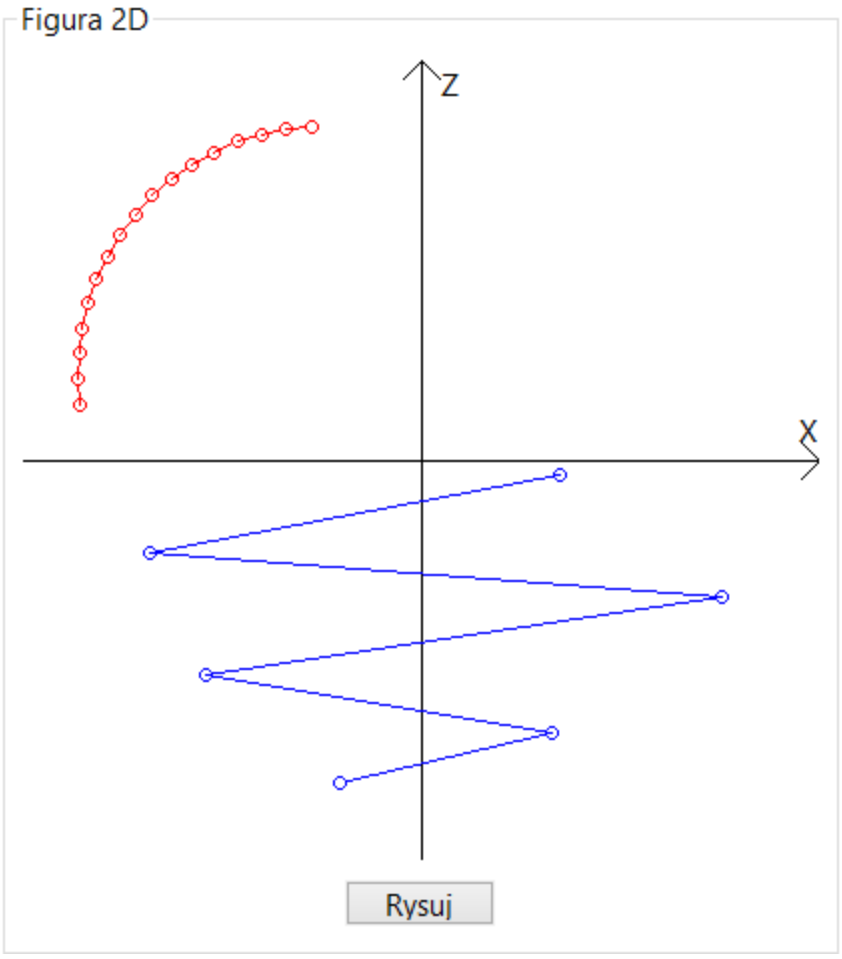
Program ma spełniać następujące założenia:

- możliwość wprowadzania przez użytkownika punktów na płaszczyźnie wyznaczonej przez osie układu współrzędnych OX, OZ.
- możliwość wyczyszczenia ekranu z punktami
- wyświetlenie powstałej bryły obrotowej
- ukrywanie zasłoniętych krawędzi bryły
- możliwość obracania bryły wokół dowolnie wybranej osi X, Y, Z
- możliwość stworzenia animacji obrotu bryły wokół dowolnie wybranych osi
- umożliwienie widoku bryły w perspektywie
- możliwość zapisania wygenerowanej grafiki w postaci 3D
- możliwość zapisania narysowanej figury 2D w postaci pliku TXT
- mozliwość wydruku wygenerowanej bryły

4. Analiza projektu

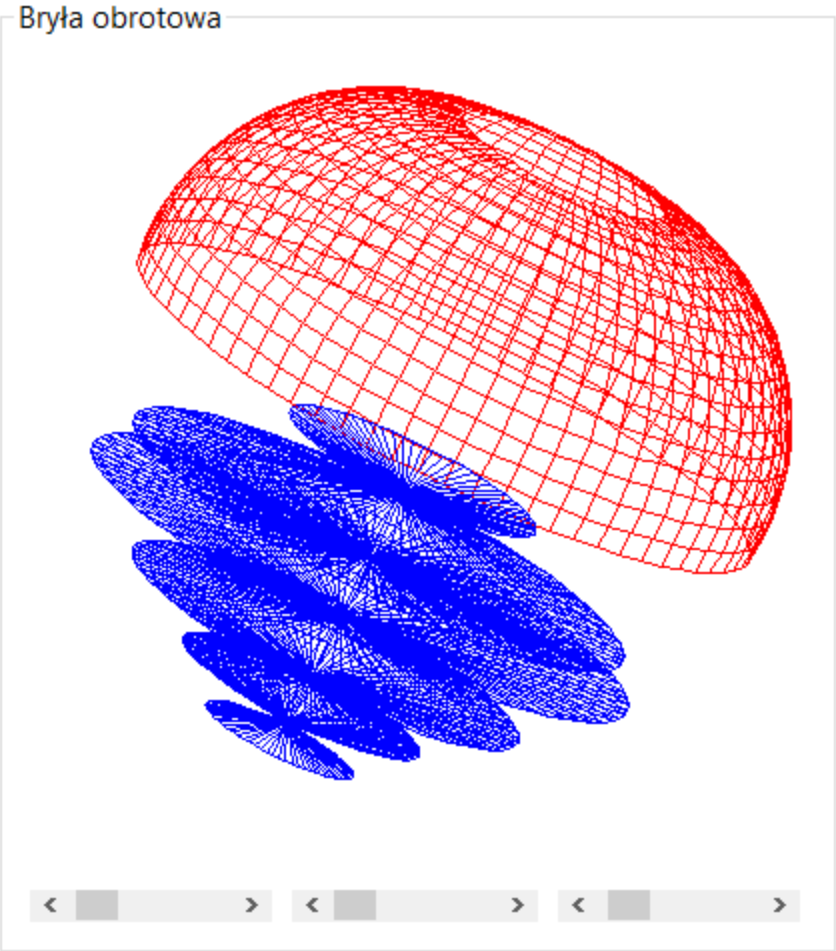
4.1. Specyfikacja danych wejściowych.

Ponieważ celem programu jest stworzenie bryły poprzez obrót linii łamanej wokół osi, dane wejściowe składają się z punktów odpowiadających końcom poszczególnych odcinków. Użytkownik za pomocą myszki wprowadza kolejne punkty, które dla jego wygody od razu łączone są liniami. Może w ten sposób tworzyć dowolne kształty i figury. Przykładowa figura znajduje sie na poniższym rysunku.



Rys. 4.1.1. Wprowadzona figura

Po narysowaniu zadawalającej figury, użytkownika wciska przycisk Rysuj, aby wygenerwować bryłę obrotową.



Rys. 4.1.2. Wygenerowana bryła obrotowa

4.2. Opis oczekiwanych danych wyjściowych.

Ze względu na charakter projektu, głównym sposobem przedstawienia obrobionych danych jest prezentacja bryły obrotowej na ekranie. W naszym przypadku gotowy szkielet zostanie wyświetlony w prawym oknie programu.

Dodatkowa funkcjonalnością naszego programu jest możliwość wyeksportowania wygenerowanej bryły do pliku w formacie BMP. Bryła jest zapisywana w takiej postaci w jakiej jest aktualnie wyświetlana na ekranie. Możemy również zapisać naszą bryłę do formatu txt, aby później móc ją wczytać do programu.

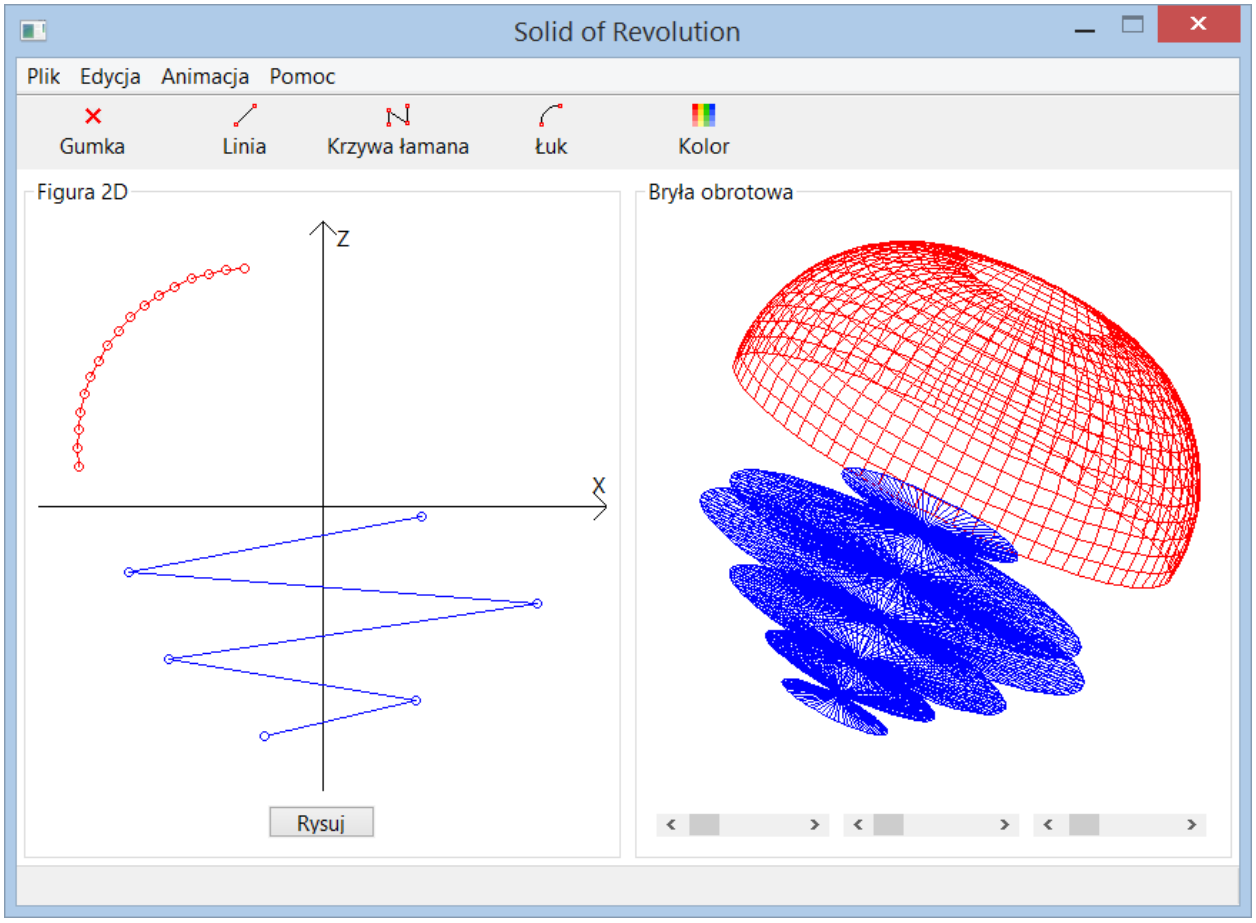
4.3. Zdefiniowanie struktur danych

Ze względu na wykorzystanie języka C++ (a zwłaszcza STL) oraz biblioteki wxWidgets oraz ich obiektową naturę, wszelkie dane wymagane przez aplikacje są reprezentowane przez instancje klas. Szczegółowy opis użytych klas wraz z odpowiadającym diagramem zostanie przedstawiony w dalszej części dokumentacji.

Ponieważ nie znamy liczby puntów, jakie poda nam użytkownik najbardziej optymalnym wyborem jest użycie standardowej klasy **std::vector**. Jako że wiekszość algorytmów używanych przez nas wymaga operacji na macierzach, używamy rownież klas **Matrix** i **Vector** (plik **vecmat.h**).

4.4. Specyfikacja interfejsu użytkownika

Interfejs naszego programu składa się z okna posiadającego menu, podzielonego na dwie głowne części. Po lewej stronie znajduje sie moduł odpowiadający za rysowanie i wprowadzanie danych, nad którym znajdują sie rownież kontrolki ułatwiające rysowanie. Po prawej stronie umiejscowiony jest moduł odpowiedzialny za wyświetlanie i manipulowanie bryła obrotową.



Rys. 4.4.1: Interfejs programu

Szczegółowy opis najważniejszych elementów interfejsu:

1. **Menu główne:**
    a. **Plik** - zawiera opcje odpowiedzialne za zapis i odczyt bryły, jej eksport, rozpoczęcia rysowania od początku, a także zamknięcie programu.
    b. **Edycja** - zawiera opcje odpowiedzialne za pokazywanie punktow podczas rysowania, zamykanie krzywych, przyciąganie do punktu a także włączenie perspektywy.
    c. **Animacja** - zawiera opcje odpowiedzialne za animację.
    d. **O programie** - wyświetla okienko z podstawowymi informacjami o programie.
2. **Moduł rysowania** - składa sie z paska narzędziowego, pola do rysowania oraz przycisku rysującego bryłę.
3. **Moduł bryły obrotowej** - składa się z okna wyświetlającego obróconą bryłę, jak i trzech kontrolek pozwalających obracać ją wokoł dowolnej osi.

4.5. Wyodrębnienie i zdefiniowanie zadań

Caly projekt można podzielić na tworzone niezależnie moduły logiczne. Podział ten pozwala na rownoległą pracę, jak i wykonywanie testów na poszczególnych modułach, co pozwoli wykryć wiele błędów jeszcze przed połączeniem ich w całość. Można dzieki temu uniknąć późniejszego żmudnego przeglądania kodu i usuwania usterek po etapie scalania.

Moduły wyodrębnione w naszej aplikacji to:

- **interfejs graficzny** odpowiedzialny za komunikację z użytkownikiem
- algorytmy i klasy odpowiedzialne za **rysowanie i przechowywanie punktów**
- algorytmy odpowiedzialne za **przekształcenia macierzowe** oraz **rzutowanie**
- algorytmy i klasy odpowiedzialne za **wyświetlanie i manipulacje** bryłą obrotową
- implementacja funkcjonalności związanych z **zapisywaniem, wczytywaniem oraz eksportem danych**

4.6. Wybór narzędzi programistycznych.

Wybranym przez nas językiem programowania był C++, ze względu na wybór metodologii programowania obiektowego. Zdecydowaliśmy się na wybór środowiska wxDevC++. Oferuje ono satysfakcjonujący nas zestaw obiektów do budowy interfejsu graficznego. Dodatkowo programowanie w zintegrowanym środowisku pozwala na łatwe zarządzanie klasami, kompilację oraz debugowanie. Dodatkowym argumentem była znajomość środowiska przez wszystkich uczestników projektu.

5. Podział pracy i analiza czasowa

Pracę nad projektem rozdzielaliśmy na bieżąco między poszczególne osoby. Dokładna tabela przedstawiająca opis wykonywanych czynności, czas trwania oraz ich wykonawcę została przedstawiona poniżej.

Lp	Planowany czas trwania	Wykonawca	Opis
1	2-4 godz.	TR, PN, ŁR	Omowienie problemu, wybór narzedzi, wstepny podział
2	3-5 godz	TR, PN, ŁR	Sporzadzenie diagramów UML, projekt interfejsu
3	2-3 dni	PN	Tworzenie interfejsu uzytkownika
4	3-5 dni	ŁR	Implementowanie modułu rysowania
5	3-5 dni	TR	Implementacja modułu obracania bryły
6	2-3 dni	TR, PN, ŁR	Testowanie stworzonych modułów
7	3-5 dni	TR, PN, ŁR	Scalanie modułów
8	1-2 dni	ŁR	Implementacja zapisu, odczytu, eksportu
9	1 dzień	TR	Implementacja drukowania
10	3-5 dni	TR	Implementacja perspektywy, animacji
11	3-5 dni	TR, PN, ŁR	Testowanie aplikacji
12	1 dzień	PN, ŁR	Tworzenie dokumentacji

6. Opracowanie i opis niezbędnych algorytmów

6.1. Rysowanie figur na podstawie wprowadzonych punktów

Rysowanie figury 2D odbywa się na płótnie reprezentującym układ współrzędnych XZ przeskalowany do zakresu [-1; 1] × [-1; 1]. Kolejne elementy są dodawane poprzez wybranie odpowiedniej opcji z paska narzędzi i zaznaczenie pewnej liczby puntków. Odpowiednie współrzędne punktów są wyliczane według wzoru:

$$tmp.x = 2 \cdot (2 \cdot click.x - graph.width)$$

$$tmp.y = 2 \cdot (2 \cdot click.y - graph.height)$$

W klasie głównej **SOR** znajduje się składnik **_shapes** będący wektorem wskaźników do obiektów typu **Shape**, każdy zaś z takich obiektów ma wektor odpowiadających mu punktów. Rysowanie kształtu odbywa się poprzez przeiterowanie po wszystkich punktach i połączenie ich linią (oraz naniesieniu samych punktów, jeśli wybrano stosowną opcję w menu) w wybranym kolorze. O ile samo rysowanie realizowane jest w identyczny sposób dla wszystkich kształtów, to sposób wypełnienia wektora punktami jest różny. W następnych rozdziałach opisano poszczególne figury.

6.1.1. Odcinek

Po nastąpieniu pierwszego kliknięcia, pozycja punktu zapisywana jest w zmiennej **_lastAddedPoint**. Następnie, po ponownym kliknięciu tworzony jest obiekt typu **Line**, do którego konstruktora przekazywane współrzędne sczytanych punktów. Gotowy obiekt dodawany jest do wektora kształtów **_shapes** i program jest gotowy do rysowania kolejnych kształtów.

6.1.2. Krzywa łamana

Procedura rysowania krzywej łamanej jest niemal identyczna do rysowania odcinka, jednak po dodaniu drugiego punktu nie dodajemy figury do wektora **_shapes**. Zapisywana jest ona w **_currentShape** i kolejne kliknięcia powodują wywołanie metody **addPoint()**. Rysowanie krzywej łamanej jest kontynuowane aż do wystąpienia jednej z sytuacji

- Podwójnego kliknięcia (po dodaniu uprzednio co najmniej dwóch punktów)
- Kliknięcia na pierwszy z dodanych do krzywej punktów (jeżeli zaznaczono ***Edycja > Przyciągaj do punktu***).

W przypadku zaznaczenia opcji ***Widok > Zamykaj krzywe*** oraz narysowania co najmniej trzech punktów, jeżeli zakończono rysowanie krzywej w sposób określony w punkcie 1, dodany zostanie jeszcze jeden odcinek łączący ostatni i pierwszy z narysowanych punktów.

6.1.3. Łuk

Rysowanie łuku odbywa się poprzez wybranie dwóch punktów - początku P1 i końca P2 łuku - odległych o **d**. Następnie włączany jest tryb edycji, w którym możemy określić odchylenie środka łuku (punkt P3) od odcinka łączącego punkty. Trzecie kliknięcie oznacza akceptację widocznego kształtu.
Rysowany łuk jest przybliżany za pomocą okręgu przechodzącego przez 3 punkty (początkowy, końcowy i aktualnie podświetlony), którego promień ograniczony został do **0.5d**. Oznacza to, że w maksymalnym wychyleniu, łuk jest półokręgiem o promieniu **0.5d**. Środek okręgu obliczany jest zgodnie ze wzorem:

$$m = 2 \cdot (P1.y \cdot P3.x - P1.y \cdot P2.x + P2.y \cdot P1.x - P2.y \cdot P3.x + P3.y \cdot P2.x - P3.y \cdot P1.x)$$

$$x_s = P2.x^2 \cdot P3.y + P2.y^2 \cdot P3.y - P1.x^2 \cdot P3.y - P1.y^2 \cdot P3.y + P1.x^2 \cdot P2.y + P1.y^2 \cdot P2.y - P3.x^2 \cdot P2.y - P3.y^2 \cdot P2.y + P3.x^2 \cdot P1.y + P3.y^2 \cdot P1.y - P2.x^2 \cdot P1.y - P2.y^2 \cdot P1.y$$

$$y_s = P1.x^2 \cdot P3.x + P1.y^2 \cdot P3.x - P2.x^2 \cdot P3.x - P2.y^2 \cdot P3.x + P3.x^2 \cdot P2.x + P3.y^2 \cdot P2.x - P1.x^2 \cdot P2.x - P1.y^2 \cdot P2.x + P2.x^2 \cdot P1.x + P2.y^2 \cdot P1.x - P3.x^2 \cdot P1.x - P3.y^2 \cdot P1.x$$

Ro - odległość pomiędzy (x_s, y_s) a dowolnym z punktów Pi

Aby umożliwić stworzenie bryły obrotowej, łuk dzielony jest następnie na ilość odcinków równą: **50 * D**, gdzie D to odległość punktu P3 od odcinka P1P2
Uzależnienie ilości odcinków od promieniu okręgu ma na celu zachowanie gładkiego kształtu krzywej dla dużych Ro, przy zachowaniu rozsądnie małej ilości punktów dla krótkich krzywych.

W algorytmie rysowania łuku zastosowana została także sztuczka mająca na celu wyeliminowanie dzielenia przez zero, które mogło pojawić się w sytuacji:

- Zaznaczenia punktów P1, P2 i P3 leżących na jednej prostej (promień okręgu przechodzącego przez te punkty sięga nieskończoności)
- Wybranie punktów P1 i P2 o tych samych współrzędnych x lub y (w przypadku tych samych współrzędnych x niemożliwe jest wyznaczenie prostej o wzorze y = ax + b).

Problem ten rozwiązany został poprzez dodawanie do jednej z niebezpiecznych wartości bardzo małego ułamka (rzędu 0.001). We wstępie do tego rozdziału wspomniany został fakt skalowania układu współrzędnych do zakresu [-1; 1] × [-1; 1]. Oczywiście, w momencie wyrysowywania wykresu na ekran, punkty są skalowane do oryginalnych wartości w pikselach. Dodanie 0.001 do wartości rzeczywistej nie powoduje zmiany wartości w pikselach (a jeśli już powoduje, to zmiana rzędu 1px jest w pełni akceptowalna).

6.2 Usuwanie figury z obszaru rysowania

Dowolną z rysowanych krzywych można skasować używając narzędzia ***Gumka***. Po przesunięciu kursora myszy nad dowolny z punktów wchodzących w skład figury (przez punkt rozumiemy miejsce załamania krzywej) zostaje ona podkreślona kolorem czerwonym, a wszystkie jej wierzchołki są powiększane. Kliknięcie na tak poświetloną krzywą powoduje usunięcie jej z pamięci.

6.3. Generowanie bryły obrotowej

Figura płaska dzielona jest na odcinki z których każdy zapisywany jest w przestrzeni trójwymiarowej z wartością Z = 0. Następnie wykonywane są obroty tych lini wokół osi OX o niewielki kąt generując kolejne odcinki będące częścią bryły. Na ekranie wyświetlana jest siatka bryły.

6.4. Zapis figury do pliku

W programie udostępniono możliwość zapisania narysowanej figury 2D do pliku TXT. Tak zapisany plik, można następnie wczytać przy kolejnym uruchomieniu ekranu i kontynuować pracę.
W związku z brakiem gotowych metod do serializacji w języku C++ (istnieją oczywiście biblioteki jak chociażby Boost Serialization, jednak używanie tak rozbudowanych narzędzi do tak prostego zadania byłoby - kolokwialnie mówiąc - "*strzelaniem z armaty do muchy*"), przygotowany został dokładny schemat zapisywania figur. W pierwszej linijce pliku podawana jest ilość zapisanych kształtów, następnie pojawia się ich lista. Opis każdej figury składa się z:

1. Liczby 999999 jako ograniczenia wczytywania.
2. Ilości punktów wchodzących w skład figury.
3. Rodzaju figury (1 - linia, 2 - łuk, 3 - krzywa łamana)
4. Koloru (zapisanego jako pojedynczy int)
5. Listy punktów w postaci: 1000 * x 1000 * y
6. Liczby 999999 jako ograniczenia końcowego.

6.5. Wyświetlanie animacji

Jedną z rozszerzonych opcji programu jest generowanie animacji na podstawie odpowiednio wprowadzonych danych (czyli kątów początkowych i końcowych dla każdej z osi). Niestety, ze względu na ograniczenia biblioteki wxWidgets, nie udało się nam zrealizować zapisu animacji do pliku GIF (twórcy wxWidgets nie zakupili licencji na tworzenie animacji GIF, odpowiednia opcja pojawiła się dopiero w wersji 2.9).
Po wybraniu w menu ***Animacja > Ustawienia*** użytkownik ma możliwość wybrania opcji rysunku oraz zapisania ich. Po zaznaczeniu ***Animacja > Utwórz animację*** i kliknięciu na ***Rysuj*** animacja jest uruchamiana. W celu zatrzymania animacji należy ponownie kliknąć przycisk rysowania.

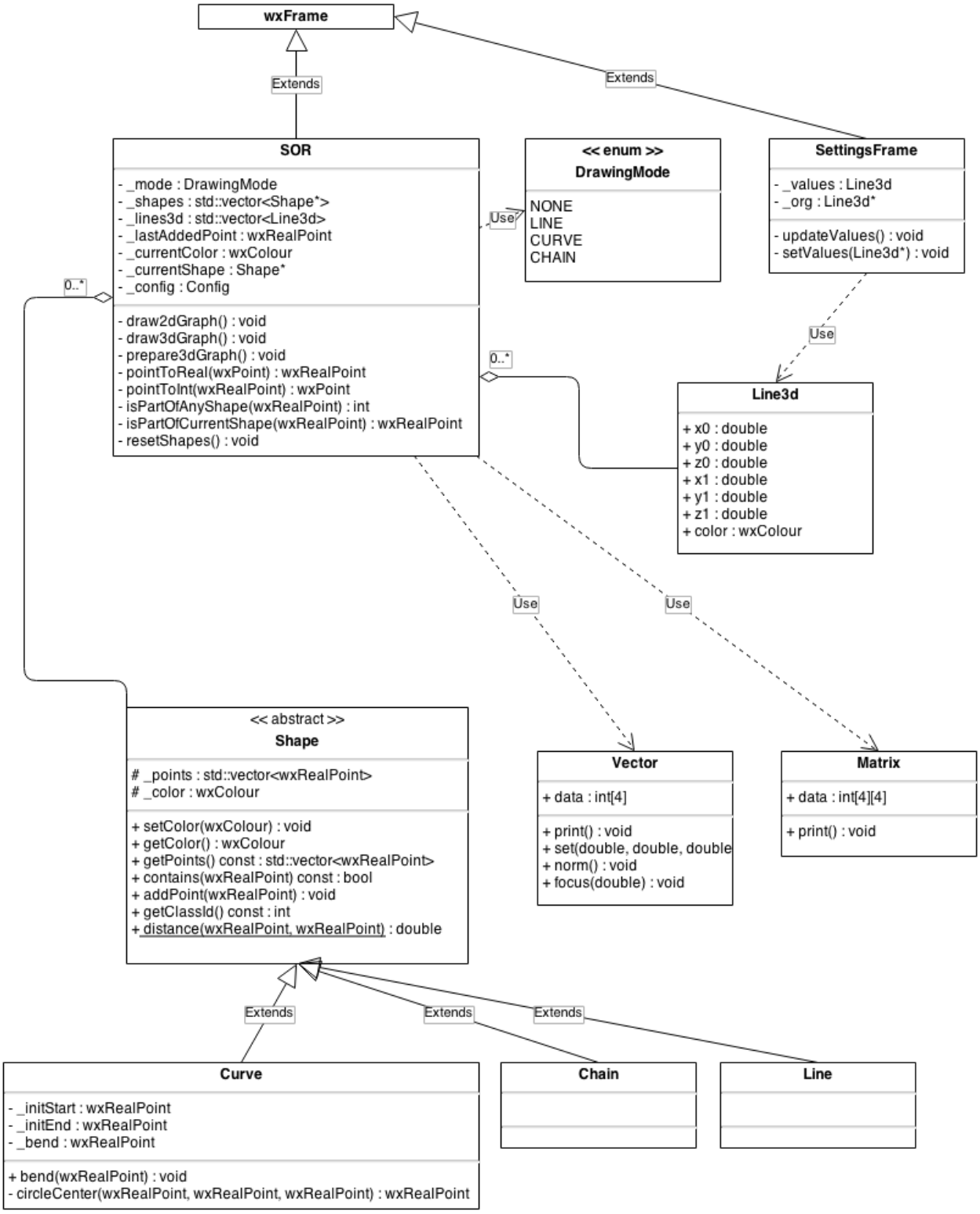
Animację zrealizowano dzięki obiektowi klasy **wxTimer** ustawionego z interwałem 100ms. Niestety, wraz ze wzrostem ilości odcinków, animacja staje się wolniejsza (rysowanie trwa dłużej). Ze względu na brak czasu niedogodność ta nie została poprawiona.

7. Kodowanie.

Program napisany został w sposób obiektowy. Główna klasa - SOR - dziedziczy po klasie wxFrame z biblioteki wxWidgets. W ramach projektu stworzono również szereg metod i klas pomocniczych. Przedstawione zostały one na diagramie klas poniżej.
Przede wszystkim, stworzono klasy odpowiedzialne za reprezentację poszczególnych figur (linii - **Line**, krzywych łamanych - **Chain** oraz łuków - **Curve**), z uwzględnieniem abstrakcyjnej klasy Shape. Ponadto, dodane zostały klasy **Vector** i **Matrix**[1], wraz z dodatkowym funkcjami globalnymi (funkcji tych nie uwzględniono na diagramie UML).
Stworzono także pomocniczą klasę **SettingsFrame** wyświetlającą ustawienia animacji.
Na diagramie UML pominięto funkcje zwane potocznie “seterami” i “geterami” oraz metody odpowiedzialne za obsługę standardowych zdarzeń w programie. Ponadto, nie znalazła się na nim klasa wewnętrzna **SOR::Config**, która jest po prostu kontenerem na zmienne globalne.

[1] Implementację tych klas oparto o dostarczony w ramach laboratorium nr 5 plik vecmat.h

7.1. Diagram UML



Rysunek 7.1.1. Diagram klas (UML)

7.2. Metody pomocnicze

W celu wygody dalszego pisania kodu, programowanie rozpoczęto od napisania kilku metod pomocniczych odpowiedzialnych za:

7.2.1. Przeliczanie współrzędnych

Przeliczanie puntków z zakresu [-1; 1] × [-1; 1] na [-w/2; w/2] × [-h/2; h/2] (w, h - rozmiary wykresu w px) oraz odwrotnie:

```
wxRealPoint SOR::pointToReal(wxPoint point) {
  wxSize size = Graph2d->GetSize();
  return wxRealPoint(
    point.x / (double)(2 * size.GetWidth()),
    point.y / (double)(2 * size.GetHeight())
  );
}

wxPoint SOR::pointToInt(wxRealPoint point) {
  wxSize size = Graph2d->GetSize();
  return wxPoint(
    point.x * size.GetWidth() / 2,
    point.y * size.GetHeight() / 2
  );
}
```

7.2.2. Sprawdzanie, czy punkt należy do figury

Metoda sprawdza, czy dany punkt (niekoniecznie będący wierzchołkiem) należy do danego kształtu. Ze względu na długość kodu, odsyłamy do kodu źródłowego (Shape.cpp, linie 3-27).

Ze względu na ilość kodu źródłowego, nie prezentujemy w dokumentacji jego fragmentów. Zapraszamy do zapoznania się z odpowiednimi plikami.

8. Testowanie.

Aplikacja była na bieżąco testowana w czasie implementacji poszczególnych modułów. Wszystkie wykryte błędy zostały usunięte. Środowiskiem testowym były komputery z systemami operacyjnymi Windows 7 i Windows 8.

Lp	**Problem**	**Rozwiązanie**
1	Migotanie obrazu podczas rysowania	Użycie wxBufferedDC w celu zapewnienia podwójnego buforowania
2	Dzielenie przez 0 podczas obliczania współczynnika kierunkowego prostej przechodzącej przez zaznaczone punkty.	Dodanie do mianownika niewielkich wartości (rzędu 0.001), które podczas rysowania na ekran były gubione (konwersja double -> int), więc nie wpływały na kształ krzywej
3	Nieskończony promień rysowanej krzywej przy zaznaczeniu trzech punktów leżących na jednej prostej	Rozwiązanie jak wyżej.
4	Dodatkowa linia pomiędzy pierwszym a ostatnim punktem podczas rysowania krzywej	Zmieniono zakres pętli dodającej punkty do krzywej
5	Błąd podczas zapisu i wczytywania figury	Podczas wczytywania użyto typu int do odczytania wartości x i y punktów. Dzielenie ich przez 10000 skutkowało otrzymaniem zer.
6	Niepoprawna kolejność rysowania ścianek bryły obrotowej	Wprowadzenie sortowania kolejności wyświetlanych krawędzi od tych znajdujących się najgłębiej, do tych najbliżej obserwatora.

9. Wdrożenie,raport i wnioski.

Udało nam się zrealizować wszystkie podstawowe założenia projektu. Dodatkowo rozszerzyliśmy aplikację o niektóre z wymagań rozszerzonych. Efekt pracy zgodny jest z naszymi początkowymi oczekiwaniami (bazującymi na wymaganiach projektu).