1 Построение графического изображения, заданного внешним файлом

В этом примере рассмотрим вариант построения изображения, состоящего из набора отрезков, заданных во входном файле.

1.1 Формат входного файла

Для простоты исполнения, будем использовать следующий формат входного файла. Набор отрезков задается тестовым файлом. Каждый отрезок представляется одной строкой файла, в которой перечислены 4 числа — координаты начала и конца отрезка, за которыми может идти последовательность символов — идентификатор отрезка. В файле могут встречаться пустые строки, а также строки комментариев. Строка комментариев должна начинаться с символа `#`.

Например файл

```
# первый отрезок
 3 89.3 34 45 AB

# второй отрезок
21 19.4 23 56.5 CD
```

содержит данные о двух отрезках AB и CD, где отрезок AB проходит от точки $(3, 89.3)$ до точки $(34, 45)$, а отрезок CD — от точки $(21, 19.4)$ до точки $(23, 56.5)$.

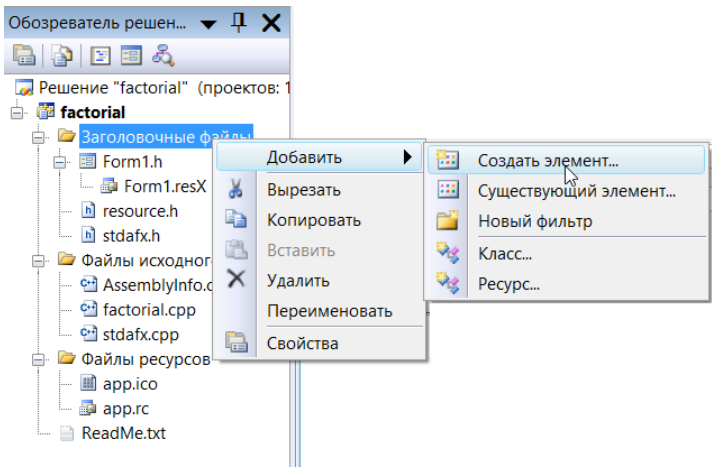
Будем считать, что файл всегда содержит корректные данные (т. е. обработку ошибок при загрузке файла проводить не будем).

1.2 Вспомогательные структуры данных

Создайте новый проект Windows Forms.

Прежде чем загружать данные из файла опишем структуры данных с которыми будем работать. Дополнительные описания выполним в заголовочном файле `Transform.h`, который подключим к проекту.

Чтобы добавить заголовочный файл в проект, щелкните правой кнопкой мыши в Обозревателе решений на папке *Заголовочные файлы*, далее меню *Добавить* → *Создать Элемент*.



В появившемся окне выберите категорию (слева) *Код* и шаблон (справа) *Заголовочный файл (.h)*. В нижней части окна введите имя заголовочного файла `Transform.h`. Нажмите кнопку *Добавить*.

В появившемся окне редактора добавим описание структуры

```
value struct point {
        float x, y;
};
```

для двумерной точки.

Двумерный отрезок будем представлять структурой

```
value struct line {
        point start, end;
        System::String^ name;
};
```

в которой `start` и `end` — точки начала и конца отрезка и `name` — название отрезка.

Теперь нужно подключить файл `Transform.h` к проекту с помощью директивы `#include`: откройте файл ⟨*имя проекта*⟩`.cpp` (для этого достаточно дважды кликнуть мышью на имени этого файла в панели обозревателя решений) и добавьте после строки

```
#include "stdafx.h"
```

ПЕРЕД строкой

```
#include "Form1.h"
```

следующий код

```
#include "Transform.h"
```

Теперь заголовочный файл подключен к проекту.

Забегая вперед: нам понадобятся при обработке входного файла процедуры для работы с файловыми и строковыми потоками. Поэтому здесь же, перед только что добавленной строкой подключим еще две библиотеки:

```
#include <fstream>
#include <sstream>
```

1.3 Список отрезков для отображения

Будем загружать данные из входного файла в коллекцию отрезков `List`, которую добавим к форме. Для этого перейдем к редактированию файла `Form1.h`: в дизайнере формы нажмите правую кнопку мыши на форме и выберите пункт *Перейти к коду*.

Сразу после описания деструктора формы (обычно строки 37–43 кода)

```
~Form1()
{
    ...
}
```

добавьте строки

```
1 private:
2         System::Collections::Generic::List<line> lines;
```

Здесь мы описали список `lines`, каждый элемент которого — элемент типа `line`. В дальнейшем будем предполагать, что в этом списке находится набор отрезков для отображения.

1.4 Первоначальная инициализация списка

Для первоначальной инициализации списка добавим к форме обработчик события *Load*. В тело процедуры обработчика добавим код

```
lines.Clear();
```

т.е. в начале работы приложения очищаем список отрезков (берем пустой список).

1.5 Отрисовка списка отрезков

Добавим для формы обработчик события *Paint* (так же как в предыдущих заданиях). В этой процедуре сначала заведем переменную `g`, ссылающуюся на область рисования в форме, очистим область для рисования `g` и опишем перо `blackPen` для вычерчивания отрезков.

После этого в области рисования `g` нарисуем все линии, хранящиеся в списке `lines`. Для этого организуем цикл по элементам этого списка:

```
for (int i = 0; i < lines.Count; i++) {
    ...
}
```

В этом цикле каждый отрезок начертим пером `blackPen`:

```
g->DrawLine(blackPen, lines[i].start.x, lines[i].start.y,
                      lines[i].end.x, lines[i].end.y);
```

Теперь проект можно запустить. Но так как список `lines` пуст — ничего нарисовано не будет.

1.6 Чтение входного файла

Добавим в нашу форму загрузку данных из файла. Для этого добавьте на форму элемент *OpenFileDialog*. Поменяйте его атрибуты:

- *(Name) = openFileDialog*
- *Title = Открыть файл*
- *DefaultExt = txt*
- *Filter = Текстовые файлы (*.txt)|*.txt|Все файлы (*.*)|*.**

Кроме этого добавим на форму элемент *Button* и установим его атрибуты

- *(Name) = btnOpen*
- *Text = Открыть*

На нажатие кнопки *btnOpen* (Событие *Click* кнопки) добавим следующие действия: показываем диалог открытия файла
`this->openFileDialog->ShowDialog()`, и проверяем, завершит ли он свою работу в результате нажатия кнопки *OK*:

```
if ( this->openFileDialog->ShowDialog() ==
         System::Windows::Forms::DialogResult::OK){
    ...
}
```

Если да, то организуем чтение из выбранного файла. При этом будем использовать стандартный потоковый ввод (соответствующие библиотеки для этого мы уже подключили). Для этого от файлового диалога нам понадобится только имя выбранного файла, но преобразованное в формат аргумента процедуры открытия файла.

```
wchar_t fileName[1024];
for (int i = 0; i < openFileDialog->FileName->Length; i++)
    fileName[i] = openFileDialog->FileName[i];
fileName[openFileDialog->FileName->Length] = '\0';
```

Теперь откроем файл и проверяем его успешное открытие

```
std::ifstream in;
in.open(fileName);
if ( in.is_open() ) {
    ...
}
```

Если файл открыт успешно, то очищаем список `lines`, после чего читаем и обрабатываем каждую строку файла:

```
lines.Clear();
std::string str;
getline (in, str);
while (in) {
    ...
    getline (in, str);
}
```

Для каждой прочитанной строки сначала проводим проверку — не является ли она пустой строкой или строкой с комментариями:

```
if ((str.find_first_not_of(" \t\r\n") != std::string::npos)
    && (str[0] != '#')) {
    ...
}
```

Если строка таковой не является, то она инициализирует некоторый отрезок. В этом случае представляем строку в виде стандартного строкового потока, из которого читаем четыре координаты концов отрезка и строку, которую превращаем в имя отрезка (читаем из потока значение типа `std::string`, а сохраняем в структуру отрезка в виде значения типа `System::String^`). Полученный отрезок добавляем в конец списка `lines`.

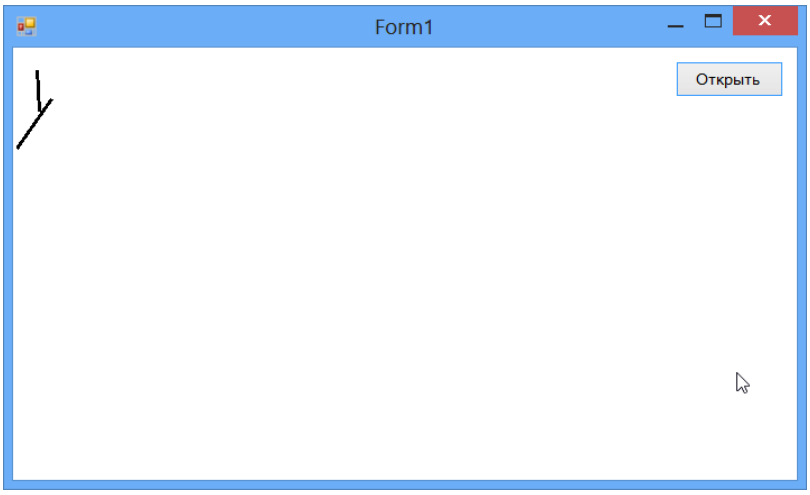
```
std::stringstream s(str);
line l;
s >> l.start.x >>  l.start.y >> l.end.x >> l.end.y;
std::string linename;
s >> linename;
l.name = gcnew String(linename.c_str());
lines.Add(l);
```

По окончании считывания файла закроем поток и обновим изображение на форме.

В результате процедура — обработчик события *Click* должна принять следующий вид:

```
private: System::Void btnOpen_Click(System::Object^  sender, System::EventArgs^  e) {
             if (this->openFileDialog->ShowDialog() ==
                 System::Windows::Forms::DialogResult::OK) {
                     wchar_t fileName[1024];
                     for (int i = 0; i < openFileDialog->FileName->Length; i++)
                         fileName[i] = openFileDialog->FileName[i];
                     fileName[openFileDialog->FileName->Length] = '\0';
                     std::ifstream in;
                     in.open(fileName);
                     if ( in.is_open() ) {
                         lines.Clear();
                         std::string str;
                         getline (in, str);
                         while (in) {
                             if ((str.find_first_not_of(" \t\r\n") != std::string::npos)
                                 && (str[0] != '#')) {
                                     std::stringstream s(str);
                                     line l;
                                     s >> l.start.x >>  l.start.y >> l.end.x >> l.end.y;
                                     std::string linename;
                                     s >> linename;
                                     l.name = gcnew String(linename.c_str());
                                     lines.Add(l);
                             }
                             getline (in, str);
                         }
                     }
                     this->Refresh();
             }
         }
```

Теперь при запуске приложения и открытии файла с показанным в начале примера содержанием, получим на форме изображение двух отрезков.



2 Преобразования построенного изображения

Добавим в наше приложение реакции на нажатие клавиш для изменения отображения загруженного набора отрезков (сдвига изображения, поворота и т. п.). При таких манипуляциях отрезки в списке `lines` меняться не будут, а будет изменяться только матрица преобразования T.

Перед отрисовкой каждого отрезка из набора `lines` будем применять к концам отрезка преобразование T. Пара полученных точек будет задавать отрезок для отрисовки.

Преобразование будем реализовывать как совмещенное преобразование, полученное последовательным произведением матриц элементарных преобразований, соответствующих нажатиям клавиш.

2.1 Добавление возможности использовать в проекте матричные операции

Прежде всего нам понадобятся матричные операции для выражения преобразований в матричной форме. Добавим в проект некоторые из этих операций. Сначала дополним заголовочный файл `Transform.h`. Перейдите к его редактированию.

Добавим в конец имеющегося кода дополнительные строки

```
1  #define M 3
2  typedef float vec[M];
3  typedef float mat[M][M];
4
5  extern mat T;
6
7  void times(mat a, mat b, mat c);
8  void timesMatVec(mat a, vec b, vec c);
9  void set(mat a, mat b);
10 void point2vec(point a, vec b);
11 void vec2point(vec a, point &b);
12 void makeHomogenVec(float x, float y, vec c);
13 void unit(mat a);
14 void move(float Tx, float Ty, mat c);
15 void rotate(float phi, mat c);
16 void scale(float S, mat c);
```

Первая из добавленных строк определяет константу `M` — порядок матриц преобразований (у нас двумерные преобразования, поэтому матрицы нужны порядка 3).

Во второй и третьей строке — введение псевдонимов для типов данных матриц и векторов размерности `M`.

В пятой строке описывается матрица `T` — матрица совмещенного преобразования для нашего приложения.

Процедуры `times` и `timesMatVec` умножают `a` на `b`, где `a` — это матрица, а `b` — матрица или вектор соответственно, и результат записывает в `c`.

Процедура `set` переписывает матрицу `a` в матрицу `b`.

Процедура `point2vec` преобразует декартовы координаты точки, заданные структурой типа `point` в однородные координаты, заданные вектором `vec`, а процедура `vec2point` — делает обратное преобразование.

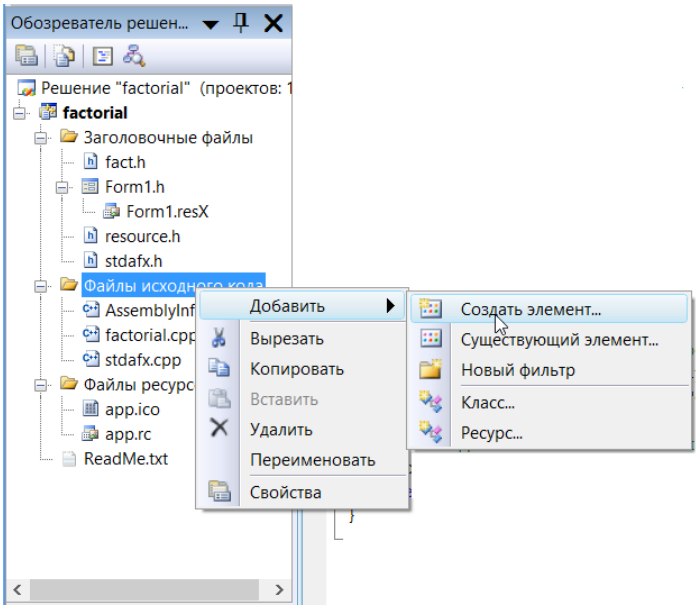
Процедура `makeHomogenVec` преобразует координаты `x` и `y` в вектор однородных координат.

Процедура `unit` преобразует свой аргумент в единичную матрицу.

Процедуры `move`, `rotate` и `scale` преобразуют аргумент `c` в матрицу преобразования переноса, поворота и масштабирования соответственно.

Добавим файл исходного кода `Transform.cpp` с реализацией этих процедур в проект. Для этого в панели обозревателя решений нажмите правую

кнопку мыши на папке *Файлы исходного кода* и в появившемся меню выберите *Добавить* → *Создать элемент*.



В появившемся окне выберите категорию (слева) *Код* и шаблон (справа) Файл C++ (.cpp). В нижней части окна введите имя файла `Transform.cpp`. Нажмите кнопку *Добавить*.

В созданном окне редактора опишите вышеперечисленные процедуры:

```
#pragma once
#include "stdafx.h"
#include "Transform.h"
#include <math.h>

mat T;

void times(mat a, mat b, mat c) {
    for(int i = 0; i < M; i++) {
        for(int j = 0; j < M; j++) {
            float skalaar = 0;
            for(int k = 0; k < M; k++)
                skalaar += a[i][k] * b[k][j];
            c[i][j] = skalaar;
        }
    }
}

void timesMatVec(mat a, vec b, vec c) {
    for(int i = 0; i < M; i++) {
        float skalaar = 0;
        for(int j = 0; j < M; j++)
            skalaar += a[i][j] * b[j];
        c[i] = skalaar;
    }
}
```

```
27
28 void set(mat a, mat b) {
29     for(int i = 0; i < M; i++)
30         for (int j = 0; j < M; j++)
31             b[i][j] = a[i][j];
32 }
33
34 void point2vec(point a, vec b) {
35     b[0] = a.x; b[1] = a.y; b[2] = 1;
36 }
37
38 void vec2point(vec a, point &b) {
39     b.x = ((float)a[0])/a[2];
40     b.y = ((float)a[1])/a[2];
41 }
42
43 void makeHomogenVec(float x, float y, vec c){
44     c[0] = x; c[1] = y; c[2] = 1;
45 }
46
47 void unit(mat a) {
48     for (int i = 0; i < M; i++) {
49         for (int j = 0; j < M; j++) {
50             if (i == j) a[i][j] = 1;
51             else a[i][j] = 0;
52         }
53     }
54 }
55
56 void move(float Tx, float Ty, mat c) {
57     unit (c);
58     c[0][M-1] = Tx;
59     c[1][M-1] = Ty;
60 }
61
62 void rotate(float phi, mat c) {
63     unit (c);
64     c[0][0] = cos(phi); c[0][1] = -sin(phi);
65     c[1][0] = sin(phi); c[1][1] = cos(phi);
66 }
67
68 void scale(float S, mat c) {
69     unit (c);
70     c[0][0] = S; c[1][1] = S;
71 }
```

2.2 Включение преобразований отрезков перед отрисовкой

Теперь у нас есть глобальная переменная `T` — матрица совмещенного преобразования.

В результате преобразований отрезки в списке `lines` изменяться не будут, а будет меняться только матрица `T`. Применять это преобразование к отрезкам будем непосредственно перед отрисовкой.

Сначала инициализируем матрицу `T` при запуске приложения. Для этого в тело обработчика события *Load* формы добавим код

```
unit (T);
```

т. е. в начале работы приложения присвоим матрице `T` единичную матрицу.

Изменим обработчик события `Paint` следующим образом. Каждую точку конца отрезка превратим в тройку однородных координат (`A` и `B` соответственно для точки начала и конца отрезка):

```
vec A, B;
point2vec(lines[i].start, A);
point2vec(lines[i].end, B);
```

применим к ним преобразование `T`

```
vec A1, B1;
timesMatVec(T,A,A1);
timesMatVec(T,B,B1);
```

получим точки в однородных координатах `A1` и `B1` соответственно. Переведем полученные координаты обратно в декартову систему.

```
point a, b;
vec2point(A1, a);
vec2point(B1, b);
```

После этого изобразим отрезок в полученных координатах.

```
g ->DrawLine(blackPen, a.x, a.y, b.x, b.y);
```

Процедура-обработчик события *Paint* примет следующий вид

```
private: System::Void Form1_Paint(System::Object^  sender, System::Windows::Forms::PaintEventArgs^  e) {
             System::Drawing::Graphics^ g = e->Graphics;
             System::Drawing::Pen^ blackPen = gcnew Pen(Color::Black);
             blackPen->Width = 4;
             for (int i = 0; i < lines.Count; i++) {
                 vec A, B;
                 point2vec(lines[i].start, A);
                 point2vec(lines[i].end, B);
                 vec A1, B1;
                 timesMatVec(T,A,A1);
                 timesMatVec(T,B,B1);
                 point a, b;
                 vec2point(A1, a);
                 vec2point(B1, b);
                 g ->DrawLine(blackPen, a.x, a.y, b.x, b.y);
             }
         }
```

Проект можно запустить. Так как `T` — тождественное преобразование и `T` пока не изменяется в нашей программе — приложение будет работать так же как работало при предыдущем запуске.

2.3 Добавление обработки нажатия клавиш

Установим реакцию приложения на нажатие клавиш. Добавим следующие реакции: смещение изображения по горизонтали и вертикали (задействуем 4 клавиши), поворот изображения относительно начала координат по часовой стрелке (1 клавиша), увеличение изображения относительно начала координат (1 клавиша). Будем использовать клавиши W, S, A, D для смещения вверх, вниз, влево и вправо соответственно. Клавишу E для поворота по часовой стрелке и X для увеличения.

При нажатии на любую из этих клавиш будем формировать матрицу соответствующего преобразования и домножать на нее матрицу `T`, после чего обновим изображение на форме. При нажатии на остальные клавиши будем формировать в качестве матрицы преобразования единичную матрицу (тем самым `T` при нажатии таких клавиш домножаться не будет).

Первым делом установим для формы значение атрибута

- *KeyPreview = True*

для того, чтобы форма принимала нажатие клавиш на себя, независимо от того, какой объект на форме является активным.

Добавим к форме процедуру-обработчик события *KeyDown*. В тело созданной процедуры добавим следующий код. Опишем матрицу преобразования и вспомогательную матрицу для записи результата перемножения матриц.

```
mat R, T1;
```

Добавим оператор `switch` в котором действия будут зависеть от значения поля `KeyCode` параметра `e` процедуры:

```
switch(e->KeyCode){
    ...
}
```

В случае когда нажата одна из клавиш W, S, A, D формируем в `R` матрицу переноса изображения на одну точку вверх, вниз, влево или вправо соответственно.

```
1 case Keys::W :
2     move(0, -1, R);
3     break;
4 case Keys::S :
5     move(0, 1, R);
```

```
    break;
case Keys::A :
    move(-1, 0, R);
    break;
case Keys::D :
    move(1, 0, R);
    break;
```

В случае нажатия E формируем в `R` матрицу поворота по часовой стрелке на угол 0.05 радиан.

```
case Keys::E :
    rotate(0.05, R);
    break;
```

Если нажата клавиша X, то в `R` записываем матрицу увеличения изображения в 1.1 раза соответственно.

```
case Keys::X :
    scale(1/1.1, R);
    break;
```

В остальных случаях в матрицу `R` записываем единичную матрицу.

```
default :
    unit(R);
```

После определения матрицы `R` (после оператора `switch`) домножим матрицу `T` слева на `R`, после чего обновим форму.

```
times(R,T,T1);
set(T1, T);
this->Refresh();
```

В результате процедура-обработчик события `KeyDown` примет следующий вид

```
private: System::Void Form1_KeyDown(System::Object^  sender, System::Windows::Forms::KeyEventArgs^  e) {
            mat R, T1;
            switch(e->KeyCode) {
                case Keys::W :
                    move(0, -1, R);
                    break;
                case Keys::S :
                    move(0, 1, R);
                    break;
                case Keys::A :
                    move(-1, 0, R);
                    break;
                case Keys::D :
                    move(1, 0, R);
                    break;
                case Keys::E :
                    rotate(0.05, R);
                    break;
                case Keys::X :
                    scale(1.1, R);
                    break;
                default :
                    unit(R);
            }
            times(R,T,T1);
            set(T1, T);
            this->Refresh();
        }
```

И, наконец, внесем изменение в процедуру-обработчик нажатия кнопки *btnOpen*. Будем возвращать матрице `T` значение единичной матрицы при загрузке нового файла. Вставим после строки очищающей список

```
lines.Clear();
```

строку, записывающую в матрицу `T` единичную матрицу.

```
unit(T);
```

Запустите приложение. Загрузите изображение. Попробуйте изменить положение и масштаб изображения.