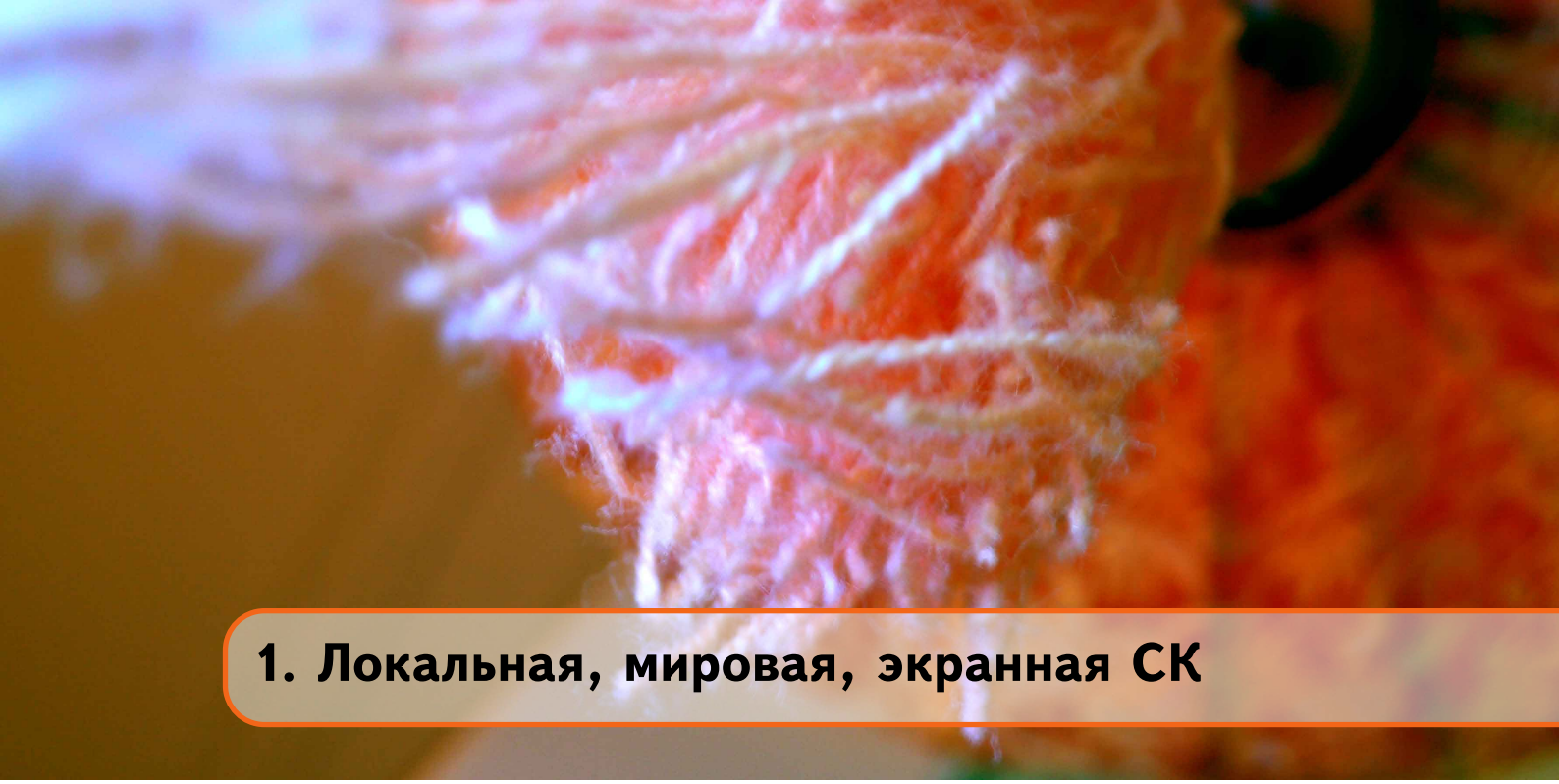
II Системы координат



1. Локальная, мировая, экранная СК

1.1 Предварительная подготовка

За основу проекта для четвертого задания возьмем уже готовый проект, получившийся в результате выполнения третьего задания.

Сделайте копию папки с проектом третьего задания (чтобы не повторять предварительные установки). Имя проекта должно остаться без изменений.

В рамках этого задания Вы продолжите работать с рисунком, соответствующим Вашему варианту в задании 2. В ходе построения решения мы будем изменять те или иные готовые фрагменты, поэтому предварительно производить «очистку» проекта не следует.

1.2 Предмет реализации

Мы внесем в проект следующие изменения.

1. Ограничим область на форме, в которой будет выводиться изображение, реализовав один из алгоритмов отсечения отрезков.
2. Реализуем отрисовку изображения, заданного в мировой системе координат, состоящего из рисунков, заданных в своих локальных координатах. Мы организуем переход от локальной системы координат к мировой, а уже от мировой системы координат к системе координат экрана.

1.3 Ограничение области видимости

1.3.1 Вывод прямоугольника на форме

Организуем в окне очерченную прямоугольную область, в которой будем выводить наше изображение.

Прежде чем изображать этот прямоугольник, добавим в форму его параметры — расстояния от прямоугольника до левой, правой, верхней и нижней границы окна. Для этого в файле `MyForm.h` к параметрам класса формы (в той строке файла, в которой было описание

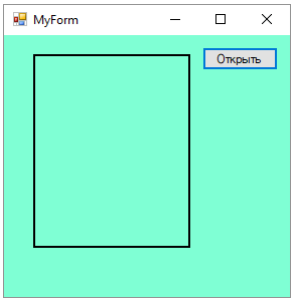
параметра `keepAspectRatio`, см. вторую главу первой части методических рекомендаций) добавим описание параметров

```
private: float left = 30, right = 100, top = 20, bottom = 50; // расстояния до границ окна
```

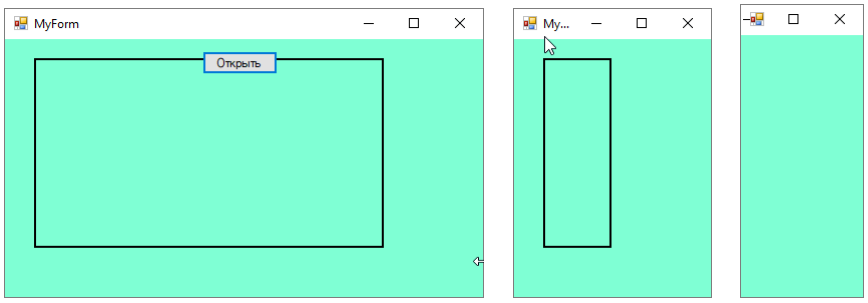
Теперь в обработчике события *Paint* сразу после очистки области рисования добавим описание переменной `rectPen` — пера для черчения прямоугольника. Здесь же организуем вывод самого прямоугольника, пользуясь стандартным методом `DrawRectangle`:

```
Pen^ rectPen = gcnew Pen(Color::Black, 2);
g->DrawRectangle(rectPen, left, // от левого
                          top,  // верхнего угла отмеряем вниз и вправо
                          ClientRectangle.Width - left - right,    // ширина
                          ClientRectangle.Height - top - bottom); // высота
```

Запуск приложения приведет к появлению следующего изображения.



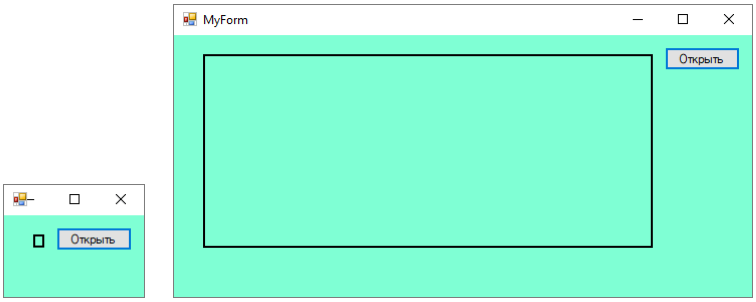
Растяжение окна не повлияет на расстояние прямоугольника от границ окна, но приведет к тому, что кнопка сможет пересечься с прямоугольником или вообще исчезнуть из поля зрения. Более того, уменьшение окна может привести к исчезновению самого прямоугольника.



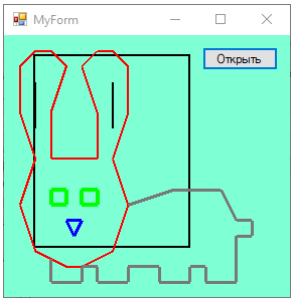
Чтобы исправить это, сначала привяжем кнопку к правому верхнему углу окна, чтобы при растяжении окна она всегда находилась бы на одной позиции относительно этого угла. Для этого перейдите в конструктор формы, откройте панель свойств кнопки и исправьте свойство *Anchor* на “*Top, Right*” (по умолчанию, значение свойства — “*Top, Left*”).

Чтобы прямоугольник на форме не исчезал, ограничим размер формы снизу. Для этого в конструкторе формы откройте панель свойств формы и измените значение свойства *MinimumSize* на “*155, 120*”.

Теперь, когда будем в запущенном приложении изменять размер формы, не сможем уменьшить форму так, чтобы прямоугольник пропал из поля зрения, а кнопка *Открыть* теперь привязана к правой границе окна.



Если в нашем окне с прямоугольником открыть какое-то изображение, то оно будет выводиться просто поверх прямоугольника.



Мы же хотим, чтобы прямоугольник ограничивал наше изображение. То есть части отрезков, которые находятся за пределами внутренней области прямоугольника, не должны отображаться.

1.3.2 Отсечение отрезков относительно сторон прямоугольника

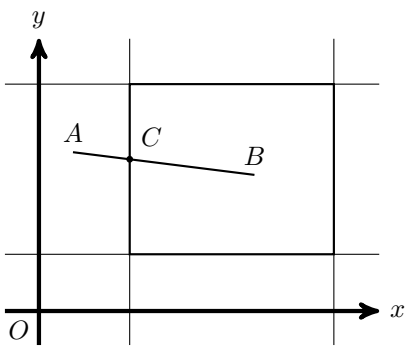
Воспользуемся алгоритмом Коэна—Сазерленда для отсечения лишних частей изображения.

Алгоритм Коэна—Сазерленда

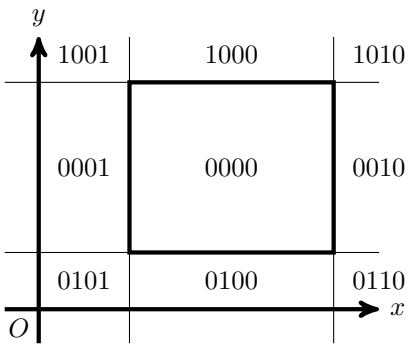
Данный алгоритм предполагает, что область видимости — прямоугольник, стороны которого параллельны осям координат.

Этот алгоритм позволяет быстро выявить отрезки, которые могут быть полностью приняты или полностью отброшены. Вычисление пересечений требуется когда отрезок не попадает ни в один из этих классов.

Идея алгоритма состоит в следующем: Прямые линии, ограничивающие область видимости делят всё двумерное пространство на 9 областей.



Каждой из областей присваивается четырехразрядный двоичный код. Например



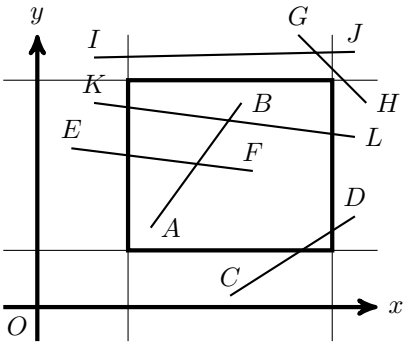
Здесь:

- единица в первом разряде — область слева от левой границы области видимости;
- единица во втором разряде — область справа от правой границы области видимости;
- единица в третьем разряде — область под нижней границей области видимости;
- единица в четвертом разряде — область над верхней границей области видимости;

Две конечные точки отрезка получают четырехразрядные двоичные коды, соответствующие областям, в которые они попали.

Определение того, является ли отрезок видимым, выполняется следующим образом:

- если коды обоих концов отрезка равны 0, то отрезок целиком внутри окна, отсечение не нужно, отрезок принимается как тривиально видимый (отрезок AB на рисунке). В противном случае логическое «ИЛИ» кодов концов отрезка не равно нулю.



- если логическое поразрядное «И» кодов обоих концов отрезка не равно нулю, то отрезок целиком вне окна, отсечение не нужно, отрезок отбрасывается как тривиально невидимый (отрезок IJ на рисунке);
- если логическое поразрядное «И» кодов обоих концов отрезка равно нулю, то отрезок подозрительный, он может быть частично видимым (отрезки CD, EF, KL) или целиком невидимым (отрезок GH); для него нужно определить координаты пересечений со сторонами окна и для каждой полученной части определить тривиальную видимость или невидимость. При этом для отрезков EF и GH потребуется вычисление одного пересечения, для остальных (CD и KL) — двух.

При расчете пересечения используется горизонтальность либо вертикальность сторон окна, что позволяет определить координату x или y точки пересечения без вычислений.

Несколько детальней метод Коэна–Сазерленда представлен в алгоритме 1.

Реализуем этот метод.

Реализация алгоритма Коэна—Сазерленда

Будем проводить реализацию алгоритма в отдельном файле заголовка. В обозревателе решений добавьте к проекту еще один файл заголовка, который назовите `Clip.h` .

Алгоритм 1: Алгоритм Коэна—Сазерленда

начало алгоритма
- Пусть AB очередной отрезок для отсечения;
- Определить C_1 и C_2 — коды областей, в которые попали точки A и B соответственно;
- **цикл пока** $(C_1 \mid C_2) \neq 0$ **выполнять**
  - *// Как минимум один из концов отрезка за пределами области видимости*
  - **если** $(C_1 \& C_2) \neq 0$ **то**
    - Видимой части нет. Выйти из алгоритма.
  - **если** $C_1 = 0$ **то** Поменять местами значения A и B, C_1 и C_2;
  - *// Теперь A точно за пределами области видимости.*
  - Определить одну из границ области видимости, за которой лежит точка A;
  - Найти новую точку A — точку пересечения AB с этой границей;
  - Обновить C_1 для точки A.
- В AB находится видимая часть исходного отрезка.

конец алгоритма

Так как точки представляются объектами типа `vec2`, нужно подключить `Matrix.h`. Кроме того, нам понадобится процедура `std::swap` для обмена значений переменных, поэтому понадобится еще подключить файл `algorithm`.

```
#include "Matrix.h"
#include <algorithm>
```

Сначала опишем функцию `codeKS`, вычисляющую код точки `P`, при заданных значениях `minX`, `minY`, `maxX`, `maxY` — минимальных и максимальных значениях координат x и y. Код точки будем представлять в виде целого числа без знака, в котором нас будут интересовать четыре младших двоичных разряда.

```
unsigned int codeKS(vec2 P, float minX, float minY, float maxX, float maxY) {
  unsigned int code = 0;  // заготовка для кода точки

  return code;
}
```

Для установки единицы в некотором двоичном разряде кода будем прибавлять к числу соответствующую степень двойки. Получим следующий фрагмент для формирования всего кода

```
// дальнейшие комментарии предполагают, что система координат правая
if (P.x < minX) { // точка левее области видимости
  code += 1;      // получаем единицу в первом разряде
}
else if (P.x > maxX) { // точка правее области видимости
  code += 2;           // получаем единицу во втором разряде
}
if (P.y < minY) { // точка ниже области видимости
  code += 4;      // получаем единицу в третьем разряде
}
else if (P.y > maxY) { // точка выше области видимости
  code += 8;           // получаем единицу в четвертом разряде
}
```

Общий вид функции `codeKS` следующий

```
unsigned int codeKS(vec2 P, float minX, float minY, float maxX, float maxY) {
  unsigned int code = 0;  // заготовка для кода точки
  // дальнейшие комментарии предполагают, что система координат правая
  if (P.x < minX) { // точка левее области видимости
    code += 1;      // получаем единицу в первом разряде
  }
  else if (P.x > maxX) { // точка правее области видимости
    code += 2;           // получаем единицу во втором разряде
  }
  if (P.y < minY) { // точка ниже области видимости
    code += 4;      // получаем единицу в третьем разряде
  }
  else if (P.y > maxY) { // точка выше области видимости
    code += 8;           // получаем единицу в четвертом разряде
  }
  return code;
}
```

Теперь опишем процедуру `clip` для отсечения отрезка с концами `A` и `B` относительно области, заданной параметрами `minX`, `minY`, `maxX`, `maxY`. Процедура будет возвращать `false`, если отрезок полностью невидим. Если часть отрезка видима, то результатом процедуры будет `true`, а в параметры процедуры `A` и `B` будут записаны координаты концов видимой части отрезка. Получим следующую заготовку для процедуры.

```
bool clip(vec2 &A, vec2 &B, float minX, float minY, float maxX, float maxY) {

}
```

Далее идем строго в соответствии с алгоритмом 1.

Сначала определим коды областей для `A` и `B`.

```
unsigned int codeA = codeKS(A, minX, minY, maxX, maxY); // код области точки A
unsigned int codeB = codeKS(B, minX, minY, maxX, maxY); // код области точки B
```

Организуем цикл, который будет работать пока коды обоих концов не равны нулю. Если мы успешно выйдем из этого цикла, то видимая часть отрезка вычислена, достаточно вернуть `true` в качестве результата.

```
while (codeA | codeB) {

}
return true;
```

Внутри цикла сначала проверим результат поразрядного «И». Если он будет ненулевой, то заканчиваем процедуру.

```
if (codeA & codeB) { // поразрядное И не равно нулю
  return false;      // отрезок полностью невидим
}
```

Если мы не вышли из цикла и не вышли из процедуры, значит один из концов отрезка за пределами области видимости. Переобозначим отрезок так, чтобы этим концом оказалась точка `A`.

```
if (codeA == 0) { // если A видима, то B невидима
  std::swap(A, B); // меняем местами A и B
  std::swap(codeA, codeB); // меняем местами их коды
}
```

Теперь определяем, за какой чертой лежит точка A. Это можно сделать, проверив на ненулевой результат поразрядное пересечение кода точки A с соответствующей степенью двойки.

```
// вычисляем новые координаты точки A
if (codeA & 1) { // точка A левее области видимости

}
else if (codeA & 2) { // точка A правее области видимости

}
else if (codeA & 4) { // точка A ниже области видимости

}
else { // точка A выше области видимости

}
```

В первом случае переносим точку A на границу прямоугольника $x =$ minX. Для этого находим новые координаты точки A — координаты точки пересечения этой границы и отрезка AB.

```
A.y = A.y + (B.y - A.y) * (minX - A.x) / (B.x - A.x);
A.x = minX;
```

Проводим аналогичные вычисления в трех остальных случаях.

После вычисления новых координат точки A обновим соответствующий код области.

```
// обновляем код области для точки A
codeA = codeKS(A, minX, minY, maxX, maxY);
```

Общий вид процедуры clip следующий.

```
1  bool clip(vec2 &A, vec2 &B, float minX, float minY, float maxX, float maxY) {
2    unsigned int codeA = codeKS(A, minX, minY, maxX, maxY); // код области точки A
3    unsigned int codeB = codeKS(B, minX, minY, maxX, maxY); // код области точки B
4    while (codeA | codeB) {
5      if (codeA & codeB) { // поразрядное И не равно нулю
6        return false;      // отрезок полностью невидим
7      }
8      if (codeA == 0) { // если A видима, то B невидима
9        std::swap(A, B); // меняем местами A и B
10       std::swap(codeA, codeB); // меняем местами их коды
11     }
12     // вычисляем новые координаты точки A
13     if (codeA & 1) { // точка A левее области видимости
14       A.y = A.y + (B.y - A.y) * (minX - A.x) / (B.x - A.x);
15       A.x = minX;
16     }
17     else if (codeA & 2) { // точка A правее области видимости
18       A.y = A.y + (B.y - A.y) * (maxX - A.x) / (B.x - A.x);
19       A.x = maxX;
20     }
21     else if (codeA & 4) { // точка A ниже области видимости
22       A.x = A.x + (B.x - A.x) * (minY - A.y) / (B.y - A.y);
23       A.y = minY;
24     }
25     else { // точка A выше области видимости
26       A.x = A.x + (B.x - A.x) * (maxY - A.y) / (B.y - A.y);
27       A.y = maxY;
28     }
29     // обновляем код области для точки A
```

```
30      codeA = codeKS(A, minX, minY, maxX, maxY);
31    }
32    return true;
33  }
```

Получившийся файл заголовка подключим к проекту. В файле `MyForm.cpp` перед строкой `#include "MyForm.h"` вставьте строку

```
#include "Clip.h"
```

Пока проект должен функционировать без изменений.

Отсечение выводимых отрезков

Снова обратимся к обработчику события *Paint* в файле `MyForm.h`.

Сразу после отрисовки прямоугольника добавим описание параметров `minX`, `minY`, `maxX`, `maxY` (для удобства, чтобы не проводить вычисления многократно), в соответствии с параметрами прямоугольника.

```
float minX = left, maxX = ClientRectangle.Width - right; // пределы изменения x
float minY = top, maxY = ClientRectangle.Height - bottom; // пределы изменения y
```

Теперь обратимся к циклу, в котором происходит отрисовка отрезков. Он сейчас выглядит следующим образом.

```
1 vec2 start = normalize(T * vec3(lines.vertices[0], 1.0)); // начальная точка первого отрезка
2 for (int j = 1; j < lines.vertices.size(); j++) { // цикл по конечным точкам (от единицы)
3   vec2 end = normalize(T * vec3(lines.vertices[j], 1.0)); // конечная точка
4   g->DrawLine(pen, start.x, start.y, end.x, end.y);  // отрисовка отрезка
5   start = end;  // конечная точка текущего отрезка становится начальной точкой следующего
6 }
```

Здесь мы каждый раз рисуем отрезок между точками `start` и `end`. Заменим строку 4 этого листинга на условную конструкцию: прежде чем чертить отрезок, проведем его отсечение и узнаем, останутся ли у него после этого видимые части.

```
if (clip(start, end, minX, minY, maxX, maxY)) { // если отрезок видим
  // после отсечения, start и end - концы видимой части отрезка
  g->DrawLine(pen, start.x, start.y, end.x, end.y);  // отрисовка видимой части
}
```

Но заметим, что в последней строке цикла (строка 5 листинга выше) требуется исходное значение координат точки `end` — до отсечения. Поэтому, перед добавленным условным оператором сохраним значение `end` во временной переменной

```
vec2 tmpEnd = end; // продублировали координаты точки для будущего использования
```

и заменим в последней строке цикла имя `end` на имя этой переменной.

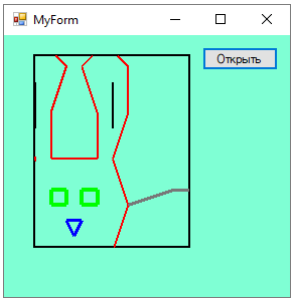
```
start = tmpEnd;  // конечная точка отрезка становится начальной точкой следующего
```

Получим общий вид цикла отрисовки.

```
1 vec2 start = normalize(T * vec3(lines.vertices[0], 1.0)); // начальная точка первого отрезка
2 for (int j = 1; j < lines.vertices.size(); j++) { // цикл по конечным точкам (от единицы)
3   vec2 end = normalize(T * vec3(lines.vertices[j], 1.0)); // конечная точка
```

```
  vec2 tmpEnd = end; // продублировали координаты точки для будущего использования
  if (clip(start, end, minX, minY, maxX, maxY)) { // если отрезок видим
    // после отсечения, start и end - концы видимой части отрезка
    g->DrawLine(pen, start.x, start.y, end.x, end.y);  // отрисовка видимых частей
  }
  start = tmpEnd;  // конечная точка неотсеченного отрезка становится начальной точкой следующего
}
```

Если сейчас запустим проект и откроем файл с изображением, будут выводиться только те части отрезков, которые попали внутрь прямоугольника.



Изменение первоначального вывода

При загрузке и первоначальном выводе изображения мы сохраняем его пропорции и масштабируем так, чтобы оно полностью (для заданных параметров `Vx` и `Vy`) поместилось в открытое окно. Но теперь изображение отсекается прямоугольником и хотелось бы, чтобы изображение так же первоначально выводилось в прямоугольнике, как раньше в окне.

Напомним, как формировалось начальное преобразование для вывода изображения в окне. За это отвечает фрагмент программного кода в процедуре `btnOpen_Click`

```
aspectFig = Vx / Vy; // обновление соотношения сторон
float Wx = ClientRectangle.Width; // размер окна по горизонтали
float Wy = ClientRectangle.Height; // размер окна по вертикали
float aspectForm = Wx / Wy; // соотношение сторон окна рисования
// коэффициент увеличения при сохранении исходного соотношения сторон
float S = aspectFig < aspectForm ? Wy / Vy : Wx / Vx;
float Ty = S * Vy; // смещение в положительную сторону по оси Oy после смены знака
initT = translate(0.f, Ty) * scale(S, -S); // преобразования применяются справа налево
        // сначала масштабирование, а потом перенос
        // в initT совмещаем эти два преобразования
```

Здесь мы обращаемся к размерам и соотношению сторон окна. Но теперь нам нужны размеры и соотношение сторон прямоугольника.

Пусть теперь `Wx` и `Wy` будут шириной и высотой прямоугольника. Заменим строки 2 и 3 приведенного листинга на вычисление этих величин (мы уже это делали при вычерчивании прямоугольника).

```
float Wx = ClientRectangle.Width - left - right; // ширина прямоугольника
float Wy = ClientRectangle.Height - top - bottom; // высота прямоугольника
```

Переменную `aspectForm` заменим на `aspectRect` (просто переименуем по смыслу).

```
float aspectRect = Wx / Wy; // соотношение сторон прямоугольника
// коэффициент увеличения при сохранении исходного соотношения сторон
float S = aspectFig < aspectRect ? Wy / Vy : Wx / Vx;
```

Наконец, как Вы помните, преобразование переноса (`translate`) состояло в сдвиге всего изображения вниз (по оси y), чтобы совместить его верхний левый угол с левым

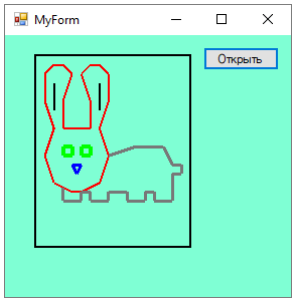
верхним углом окна. Но теперь нам нужно провести совмещение угла изображения с углом прямоугольника. Для этого нужен ещё один дополнительный сдвиг вниз и вправо, до координат левого верхнего угла прямоугольника.

```
float Tx = left; // смещение в положительную сторону по оси Ox
float Ty = S * Vy + top; // смещение в положительную сторону по оси Oy
initT = translate(Tx, Ty) * scale(S, -S); // преобразования применяются справа налево
        // сначала масштабирование, а потом перенос
        // в initT совмещаем эти два преобразования
```

Получим переписанный фрагмент в общем виде.

```
aspectFig = Vx / Vy; // обновление соотношения сторон
float Wx = ClientRectangle.Width - left - right; // ширина прямоугольника
float Wy = ClientRectangle.Height - top - bottom; // высота прямоугольника
float aspectRect = Wx / Wy; // соотношение сторон прямоугольника
// коэффициент увеличения при сохранении исходного соотношения сторон
float S = aspectFig < aspectRect ? Wy / Vy : Wx / Vx;
float Tx = left; // смещение в положительную сторону по оси Ox
float Ty = S * Vy + top; // смещение в положительную сторону по оси Oy
initT = translate(Tx, Ty) * scale(S, -S); // преобразования применяются справа налево
        // сначала масштабирование, а потом перенос
        // в initT совмещаем эти два преобразования
```

Теперь, если запустим проект и загрузим изображение,первоначальное масштабирование будет происходить относительно прямоугольника.



Немного оптимизации

Мы добавили в форму ряд вычислений, связанных с параметрами прямоугольника. Зачастую одни и те же вычисления проводятся множество раз. Например, ширина прямоугольника вычисляется каждый раз, когда происходит событие *Paint*. Кроме того, ширина вычисляется при каждой загрузке файла. Но изменение значения ширины прямоугольника происходит только когда изменяется размер формы.

Аналогичные рассуждения можно провести для большинства введенных в этом разделе параметров. Поэтому, стоит ввести глобальные для обработчиков событий параметры, которые будут корректироваться при изменении размеров формы.

Сейчас у нас уже есть параметры формы

```
private: float left = 30, right = 100, top = 20, bottom = 50; // расстояния до границ окна
```

Добавим к ним дополнительные параметры.

```
private: float left = 30, right = 100, top = 20, bottom = 50; // расстояния до границ окна
         float minX = left, maxX; // диапазон изменения координат x
         float minY = top, maxY; // диапазон изменения координат y
         float Wcx = left, Wcy; // координаты левого нижнего угла прямоугольника
         float Wx, Wy; // ширина и высота прямоугольника
```

Здесь вводится много синонимов (например, `left` , `minX` , `Wcx`) для того, чтобы сохранить единство обозначений в рамках фрагментов программы.

Инициализация и изменение параметров, у которых пока не установлено значение, происходит по одному и тому же алгоритму. Поэтому, опишем вспомогательный приватный метод, который будет это делать. В строках, следующих за описанием параметров, добавим процедуру `rectCalc` .

```
private: System::Void rectCalc() {
  maxX = ClientRectangle.Width - right; // диапазон изменения координат x
  maxY = ClientRectangle.Height - bottom; // диапазон изменения координат y
  Wcy = maxY; // координаты левого нижнего угла прямоугольника
  Wx = maxX - left; // ширина прямоугольника
  Wy = maxY - top; // ширина и высота прямоугольника
}
```

Вызов этой процедуры необходим перед самой первой отрисовкой прямоугольника. Добавим его в тело обработчика события *Load* (он должен был остаться пустым в результате выполнения третьего задания). Процедура `MyForm_Load` примет вид.

```
1 private: System::Void MyForm_Load(System::Object^  sender, System::EventArgs^  e) {
2   rectCalc();
3 }
```

В обработчике события *Resize* сейчас содержится единственная строка.

```
Refresh();
```

Добавим перед этой строкой код, который будет изменять значения параметров, зависящих от размера видимой области рисования.

```
rectCalc();
```

Обработчик события *Resize* в общем виде будет выглядеть следующим образом.

```
1 private: System::Void MyForm_Resize(System::Object^  sender, System::EventArgs^  e) {
2   rectCalc();
3   Refresh();
4 }
```

Теперь используем введенные параметры в обработчике события *Paint*. Во фрагменте

```
1 g->DrawRectangle(rectPen, left, // от левого
2             top,  // верхнего угла отмеряем вниз и вправо
3             ClientRectangle.Width - left - right,    // ширина
4             ClientRectangle.Height - top - bottom); // высота
5 float minX = left, maxX = ClientRectangle.Width - right; // пределы изменения x
6 float minY = top, maxY = ClientRectangle.Height - bottom; // пределы изменения y
```

ширина и высота прямоугольника нам уже известны, поэтому их вычисления заменим на значения соответствующих параметров. Пределы изменения x и y также вынесены в значения параметров, поэтому последние две строки являются лишними.

Таким образом, вместо приведенного фрагмента вставим следующий.

```
g->DrawRectangle(rectPen, left, top, Wx, Wy);
```

Процедура `MyForm_Paint` на данном этапе примет вид

```
private: System::Void MyForm_Paint(System::Object^  sender, System::Windows::Forms::PaintEventArgs^  e) {
  Graphics^ g = e->Graphics;
  g->Clear(Color::Aquamarine);

  Pen^ rectPen = gcnew Pen(Color::Black, 2);
  g->DrawRectangle(rectPen, left, top, Wx, Wy);

  for (int i = 0; i < figure.size(); i++) {
    path lines = figure[i]; // lines - очередная ломаная линия
    Pen^ pen = gcnew Pen(Color::FromArgb(lines.color.x, lines.color.y, lines.color.z));
    pen->Width = lines.thickness;

    vec2 start = normalize(T * vec3(lines.vertices[0], 1.0)); // начальная точка первого отрезка
    for (int j = 1; j < lines.vertices.size(); j++) { // цикл по конечным точкам (от единицы)
      vec2 end = normalize(T * vec3(lines.vertices[j], 1.0)); // конечная точка
      vec2 tmpEnd = end; // продублировали координаты точки для будущего использования
      if (clip(start, end, minX, minY, maxX, maxY)) { // если отрезок видим
        // после отсечения, start и end - концы видимой части отрезка
        g->DrawLine(pen, start.x, start.y, end.x, end.y);  // отрисовка видимых частей
      }
      start = tmpEnd;  // конечная точка неотсеченного отрезка становится начальной точкой следующего
    }
  }
}
```

В процедуре `btnOpen_Click` стали лишними строки, в которых вычисляются значения `Wx` и `Wy`. Их следует убрать.

Запустите проект. Он должен работать корректно.

1.4 Локальная (объектная) система координат

При построении изображения в ходе выполнения второго задания мы выбрали систему координат, в которой задавали координаты концов отрезков. Наша выбранная система координат — правая, а начало координат находится в левом нижнем углу картинки. Получая матрицу `initT` при загрузке изображения, мы учитывали этот факт.

По большому счету, можно было ввести первоначальную систему координат произвольно. В выбранной системе координат точка начала координат может располагаться в любом месте изображения и даже за его пределами. Это лишь добавило бы какие-то лишние преобразования.

Представим себе задачу, в которой наше изображение является лишь одной из многих деталей, а нам требуется разместить эту деталь в нужной позиции, масштабе и ракурсе.

Обычно, нужная позиция задается точкой, с которой необходимо совместить определенную точку изображения. Например, сейчас в нашем проекте кроме нашего изображения присутствует еще и прямоугольник, к которому мы привязываем левый верхний угол нашей картинки, т. е. в нашем случае такой точкой привязки является верхний левый угол.

Понятно, что выбор точки привязки влияет на начальное преобразование изображения.

Давайте изменим проект, чтобы при загрузке изображения совмещались не левые верхние углы прямоугольника и картинки, а центры. Можно это сделать в следующем порядке:

1. в системе координат рисунка перенести начало координат в центр рисунка;
2. масштабировать изображение в нужное число раз;
3. сдвинуть изображение, чтобы центр рисунка сместился из начала координат в нужную точку (в центр прямоугольника).

Первое из этих преобразований, в нашем случае, можно реализовать с помощью матрицы переноса на

$$-\,\mathtt{Vx}\,/2$$

вдоль оси x и

$$-\,\mathtt{Vy}\,/2$$

вдоль оси y (сдвиг с координатами центра рисунка, взятыми с отрицательными (обратными) координатами). Второе преобразование есть то же самое масштабирование, которое уже

имеется. Третье — сдвиг на

$$\texttt{Wx}\,/2 + \texttt{Wcx}$$

вдоль оси x и

$$\texttt{Wcy} - \texttt{Wy}\,/2$$

вдоль оси y (сдвиг с координатами центра прямоугольника). В результате, начальное преобразование `initT` можно выразить произведением трех матриц этих преобразований.

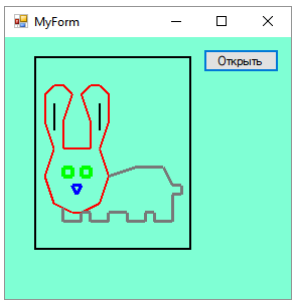
Обратимся к фрагменту кода процедуры `btnOpen_Click`, ответственному за формирование матрицы `initT`.

```
// коэффициент увеличения при сохранении исходного соотношения сторон
float S = aspectFig < aspectRect ? Wy / Vy : Wx / Vx;
float Tx = left; // смещение в положительную сторону по оси Ox
float Ty = S * Vy + top; // смещение в положительную сторону по оси Oy
initT = translate(Tx, Ty) * scale(S, -S); // преобразования применяются справа налево
        // сначала масштабирование, а потом перенос
        // в initT совмещаем эти два преобразования
```

Заменим эти строки на реализацию того, что мы только что описали.

```
// смещение центра рисунка с началом координат
mat3 T1 = translate(-Vx / 2, -Vy / 2);
// масштабирование остается прежним, меняется только привязка
// коэффициент увеличения при сохранении исходного соотношения сторон
float S = aspectFig < aspectRect ? Wy / Vy : Wx / Vx;
mat3 S1 = scale(S, -S);
// сдвиг точки привязки из начала координат в нужную позицию
mat3 T2 = translate(Wx / 2 + Wcx, Wcy - Wy / 2);
// В initT совмещаем эти три преобразования (справа налево)
initT = T2 * (S1 * T1);
```

Если запустить проект и загрузить рисунок из файла, то получим изображение.



Как видим, центр рисунка переместился в центр окна.

Примененная здесь стратегия достаточно действенна. Первым преобразованием мы переместили точку привязки (центр рисунка) в начало координат, поэтому масштабирование не привело к изменению положения этой точки — после масштабирования она осталась в начале координат. Осталось сместить эту точку в нужную позицию. Центр рисунка и центр прямоугольника здесь играли посредственную роль, на их месте могли быть произвольные точки рисунка и прямоугольника: единственное условие — нам должны быть известны координаты этих точек.

Таким образом, при задании рисунка мы используем координаты, заданные в некоторой системе координат, которую называют локальной или объектной. Выполняя привязку рисунка, т е. выполняя преобразование `initT`, мы организуем переход от этой системы координат к той, в которой организуется привязка. В нашем случае второй системой координат является система координат экрана. Матрицу начального преобразования будем называть модельной матрицей.

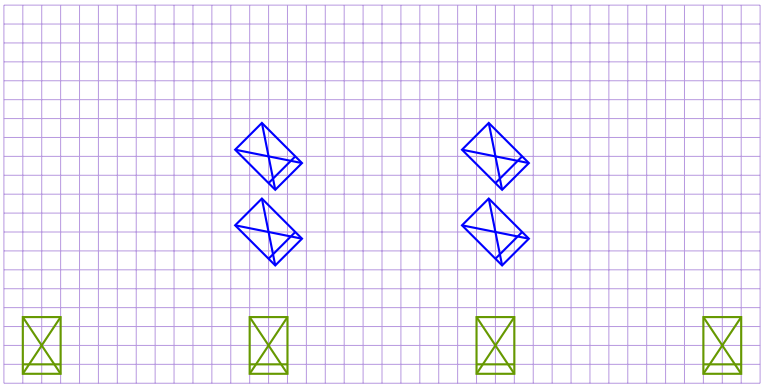
1.5 Мировая система координат

1.5.1 Структура изображения

Несколько поменяем концепцию нашего изображения. Построим сложное изображение, в котором наш рисунок, наряду с ещё одним, присутствует несколько раз.

Представим, что полное изображение представляет собой прямоугольную область, в которой обозначены позиции и ракурс размещения отдельных копий рисунков.

Например,



Этой схемой задано изображение, в котором участвуют два рисунка на различных позициях. Синие прямоугольники соответствуют позициям одного рисунка, зеленые — второго. Вся схема (весь полный рисунок) занимает 80×20 клеток. Размеры (в количестве клеток) каждого прямоугольника можно увидеть (здесь это 2×3). Точкой привязки рисунка является центр прямоугольника (отмечен пересечением диагоналей). Дополнительная линия вдоль одной из сторон прямоугольника соответствует нижней стороне рисунка для случая, когда соотношение его сторон не больше единицы, или правой стороне рисунка для остальных случаев.

Для того, чтобы задать все изображение, так же как и в задании 2, введем систему координат, точка начала которой будет находиться в нижнем левом углу. Тогда получается, что для каждого экземпляра одного рисунка мы должны сохранить для последующего использования собственное начальное преобразование (которое можно представить модельной матрицей).

1.5.2 Структуры данных

Получается, что нам нужна новая структура данных для хранения рисунков с их преобразованием.

Опишем эту структуру данных в файле заголовка `Figure.h`. Перейдем к его редактированию.

Опишем здесь класс `model`, который будет описывать каждую составляющую нашего изображения: набор объектов `path`, заданных в своей локальной системе координат, в совокупности с модельной матрицей.

```
class model {
public:
  std::vector<path> figure; // составляющие рисунка
  mat3 modelM; // модельная матрица

};
```

Эта структура подразумевает, что мы будем хранить отдельный список составляющих рисунка (`figure`) для каждого его экземпляра в изображении.

Добавим простой конструктор от двух аргументов, который будет объединять все заданные аргументы в объект.

```
model(std::vector<path> fig, mat3 mat) {
  figure = fig;
  modelM = mat;
}
```

Получим общее представление описания класса `model` .

```
class model {
public:
  std::vector<path> figure; // составляющие рисунка
  mat3 modelM; // модельная матрица
  model(std::vector<path> fig, mat3 mat) {
    figure = fig;
    modelM = mat;
  }
};
```

Будем считать, что в ходе чтения файла с описанием изображения мы будем создавать объекты класса `model` и помещать их в список, к которому будем обращаться в обработчике события *Paint*.

Но прежде чем перейти к изменению программы, оговорим изменения в формате входного файла.

1.5.3 Формат входного файла

Входной файл будет содержать описание каждого объекта, выводимого на рисунке, в его локальных координатах. Каждый отдельный рисунок будет описываться так же, как и раньше, но описание будет начинаться с команды `model` , сигнализирующей о том, что описание очередного рисунка закончено и начат новый рисунок.

Кроме того, в файле будут содержаться команды, описывающие последовательность преобразований для каждой привязки рисунка.

Преобразования задаются из предположения, что в результате команды `model` рисунок смещается так, чтобы центр рисунка совпал с началом координат, и масштабируется так, чтобы рисунок был вписан (при равномерном масштабировании) в квадрат 2×2.

Командой `figure` будем отмечать, что формирование начального преобразования для очередной копии рисунка завершено, а последующие преобразования приведут нас к расположению следующей копии рисунка (или другого рисунка).

Так как одна последовательность преобразований может быть началом другой последовательности, введем еще две дополнительные команды, которые позволят сохранять накопленную последовательность в стеке (`pushTransform`) и извлекать сохраненную ранее последовательность из стека (`popTransform`).

Команда `frame` будет теперь относиться не к рисунку, а ко всему изображению (как и раньше). Размеры же отдельного рисунка будет задаваться как параметры команды `model` . Другими двумя параметрами команды `model` будут координаты той точки рисунка, которой он привязывается к общему изображению.

Итак, предполагаем, что кроме команд, описанных в инструкции к заданию 3, во входном файле могут встречаться строки команд следующего вида:

`model mVcx mVcy mVx mVy` команда начала описания нового рисунка. Все последующие команды `path` будут относиться к этому рисунку. Параметры `mVx` и `mVy` — размеры рисунка по горизонтали и вертикали в локальных единицах измерения; `mVcx` и `mVcy` — координаты точки привязки (центра рисунка) в объектной системе координат;

`figure` (без параметров) команда окончания формирования начального преобразования для очередной копии рисунка;

`translate Tx Ty` преобразование сдвига с коэффициентами `Tx`, `Ty`;

`rotate Phi` преобразование поворота относительно начала координат против часовой стрелки на угол `Phi`, где `Phi` задан в градусах;

`scale S` преобразование масштабирования относительно начала координат в `S` раз;

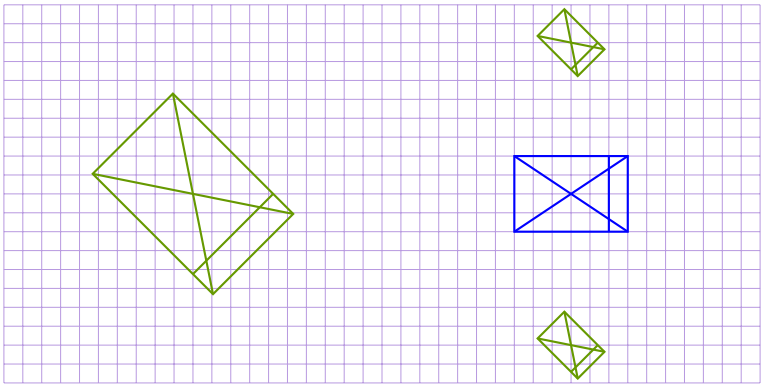
`pushTransform` сохранение накопленного совмещенного преобразования в стеке для возможного последующего возвращения к нему;

`popTransform` извлечение из стека преобразований (с удалением из стека) матрицы преобразования, определенной этим преобразованием.

Предполагаем, что каждый рисунок задан в правой системе координат и выводится в правой системе координат.

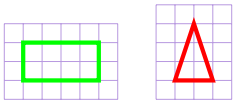
Таким образом, каждой команде `figure` будет соответствовать список объектов `path`, накопленный после последней команды `model` и модельная матрица, полученная в результате перемножения матриц преобразований, соответствующих командам `translate`, `scale` и `rotate`.

Давайте попробуем описать в файле изображение, заданное следующей схемой.



Размер самого большого прямоугольника — 6×9 клеток. Размер самого маленького — 2×3 клеток. Угол поворота зеленых прямоугольников — 45 градусов.

На месте прямоугольников в изображении должны быть привязаны рисунки.



Сначала определимся с системой координат на схеме. Пусть начало координат лежит в левом нижнем углу, а две клетки соответствуют единице.

Рассмотрим сначала размещение первого изображения на месте зеленых прямоугольников. Помним, что после команды `model` центр рисунка находится в начале координат, а его ширина равна двум, т. е. весь рисунок расположен от -1 до 1 по оси Ox (так как ширина рисунка больше высоты).

Тогда, чтобы первое изображение разместить на месте самого большого прямоугольника, нужно:

1. Повернуть рисунок на отрицательный угол -45 градусов относительно начала координат (так как правая сторона рисунка должна соответствовать дополнительной линии в прямоугольнике).
2. Масштабировать в 2.25 раза относительно начала координат (так как большая длина большого прямоугольника равна 4.5 (9 клеток), а ширина рисунка — 2).
3. Сдвинуть рисунок, чтобы его центр (который все ещё находится в начале координат) попал в точку с координатами $(5, 5)$.

Если бы мы с самого начала размещали не самый большой, а один из малых зеленых прямоугольников, то последовательность преобразований была бы следующей.

1. Повернуть рисунок на отрицательный угол -45 градусов относительно начала координат (так как правая сторона рисунка должна соответствовать дополнительной линии в прямоугольнике).
2. Масштабировать в 0.75 раза относительно начала координат (так как большая длина малого прямоугольника равна 1.5 (3 клетки), а ширина рисунка — 2).
3. Сдвинуть рисунок, чтобы его центр (который все ещё находится в начале координат) попал в точку с координатами $(15, 1)$.

Но последовательность преобразований задается во входном файле сквозным образом и чтобы провести вторую последовательность преобразований, следовало бы вернуться к отправной точке, применив обратные преобразования. Но мы вместо этого будем сохранять накопленное преобразование и откатываться к нему при необходимости. Две приведенные последовательности команд вместе с командами размещения остальных элементов можно объединить в одну следующим образом.

1. Сохранить накопленное преобразование — отправную точку. Так как еще преобразований не было, то это преобразование выразится единичной матрицей.
2. Повернуть рисунок на отрицательный угол -45 градусов относительно начала координат (так как правая сторона рисунка должна соответствовать дополнительной линии в прямоугольнике).
3. Сохранить накопленное преобразование — поворот на -45 градусов, чтобы не выполнять преобразование второй раз, когда будем размещать малый прямоугольник.
4. Масштабировать в 2.25 раза относительно начала координат (так как большая длина большого прямоугольника равна 4.5 (9 клеток), а ширина рисунка — 2).
5. Сдвинуть рисунок, чтобы его центр (который все ещё находится в начале координат) попал в точку с координатами $(5, 5)$. Здесь заканчиваем последовательность преобразований для самого большого прямоугольника.
6. Откатиться к последнему сохраненному преобразованию (поворот уже совершен).
7. Масштабировать в 0.75 раза относительно начала координат (так как большая длина малого прямоугольника равна 1.5 (3 клетки), а ширина рисунка — 2).
8. Сдвинуть рисунок, чтобы его центр (который все ещё находится в начале координат) попал в точку с координатами $(15, 1)$. Здесь заканчиваем последовательность преобразований для нижнего малого прямоугольника.
9. Сдвинуть рисунок, чтобы его центр (который в точке $(15, 1)$) сместился в точку $(15, 9)$. Здесь заканчиваем последовательность преобразований для верхнего малого прямоугольника.
10. Откатиться к последнему сохраненному преобразованию. Так как при откате к преобразованию оно удаляется из стека, то преобразование на момент «поворот уже совершен» в стеке уже отсутствует, а наверху стека находится сохраненное перед ним — отправная точка.
11. На этот момент уже должно быть прочитано описание второго рисунка. Повернуть рисунок на отрицательный угол 90 градусов относительно начала координат

(так как нижняя сторона рисунка должна соответствовать дополнительной линии в прямоугольнике).

12. Масштабировать в 1.5 раза относительно начала координат (так как большая длина синего прямоугольника равна 3 (6 клеток), а высота рисунка $-$ 2).
13. Сдвинуть рисунок, чтобы его центр (который все ещё находится в начале координат) попал в точку с координатами $(15, 5)$. Здесь заканчиваем последовательность преобразований для синего прямоугольника.

Тогда можем всё сказанное оформить в виде следующего входного файла.

```
frame 20 10 # размеры всего изображения из расчета одна единица - две клетки

# первый рисунок
model 1.5 1 3 2 # центр в точке (1.5, 1), размеры 3x2
color 0 255 0 # цвет зеленый
thickness 3 # толщина линии 3
path 5 # путь из четырех ребер (пять вершин)
0.5 0.5 # левый нижний угол
0.5 1.5 # левый верхний угол
2.5 1.5 # правый верхний угол
2.5 0.5 # правый нижний угол
0.5 0.5 # левый нижний угол

# преобразования и размещения по описанию
pushTransform # сохранить отправную точку
rotate -45 # поворот на -45 градусов
pushTransform # сохранить преобразование поворота
scale 2.25 # масштабирование до большого прямоугольника
translate 5 5 # перенос центра рисунка в точку (5,5)
figure # запомнить положение и ракурс первого рисунка
popTransform # откатились к преобразованию поворота
scale 0.75 # масштабирование до малого прямоугольника
translate 15 1 # установить в позицию нижнего малого прямоугольника
figure # запомнить положение и ракурс второго экземпляра рисунка
translate 0 8 # передвинуться в позицию (15,9) из (15,1)
figure # запомнить положение и ракурс третьего экземпляра рисунка
popTransform # откатились к стартовой позиции

# второй рисунок
model 1 1.25 2 2.5 # параметры рисунка с треугольником
color 255 0 0 # цвет красный
path 4 # четыре точки в маршруте
0.5 0.5 # нижний левый угол
1 2 # верхний угол
1.5 0.5 # нижний правый угол
0.5 0.5 # нижний левый угол

# преобразования и размещения по описанию
rotate 90 # поворот на 90 градусов
scale 1.5 # масштабирование до синего прямоугольника
translate 15 5 # сдвиг в нужную позицию
figure # запомнить положение и ракурс рисунка
```

Сохраните этот набор команд в файле `Geometric.txt` .

Получили полное описание изображения со своей системой координат, в которой организована привязка двух рисунков (точнее, четырех рисунков), заданных в своих локальных координатах. Система координат всего изображения, в которой привязываются его составные элементы, называется мировой системой координат.

1.5.4 Организация чтения входного файла

Представим теперь в какой форме будут храниться составляющие части нашего изображения.

Раньше у нас из объектов `path` собирался один рисунок, который затем вычерчивался. Глобальная переменная `figure` представляла список этих объектов. Теперь же нам нужен список рисунков. Каждый рисунок будем представлять объектом `model` и, следовательно, нам понадобится глобальный список таких объектов. Список `figure` теперь можно удалить или сделать локальным там, где такой список потребуется.

Переменные `Vx` и `Vy`, `aspectFig` у нас остаются в той же роли, что и раньше, но теперь они будут относиться ко всему изображению.

Параметры `mVcx`, `mVcy`, `mVx`, `mVy` нужны только для формирования модельной матрицы и их не требуется где-то сохранять.

Перейдем к описанию блока глобальных переменных в файле `MyForm.h` и заменим там описание списка ломаных

```
vector<path> figure;
```

на описание списка рисунков

```
vector<model> models;
```

Теперь внесем изменения в непосредственно процесс чтения файла. Обратимся к процедуре `btnOpen_Click`.

В порядке чтения файла у нас будет обновляться список `models` (а не `figure`). Поэтому, после успешного открытия файла заменим очистку списка `figure` на очистку списка `models`. То есть заменим строку

```
figure.clear(); // очищаем имеющийся список ломаных
```

на строку

```
models.clear(); // очищаем имеющийся список рисунков
```

Здесь нам потребуется описание дополнительных переменных:

- понадобится матрица, в которой будем накапливать матричное преобразование;
- понадобится матрица, в которой сохраним начальное преобразование отдельного рисунка — смещение центра рисунка в начало координат и его масштабирование;
- потребуется стек матриц, в котором будем сохранять матрицы по команде `pushTransform`;
- нужен локальный список ломаных для отдельного рисунка.

Эти переменные потребуются нам, когда файл будет успешно открыт.

Найдите строки, с описанием переменных `thickness`, `r`, `g`, `b`:

```
float thickness = 2; // толщина со значением по умолчанию 2
float r, g, b; // составляющие цвета
```

Перед этими строками вставим описание вышеупомянутых переменных.

```
mat3 M = mat3(1.f); // матрица для получения модельной матрицы
mat3 initM; // матрица для начального преобразования каждого рисунка
vector<mat3> transforms; // стек матриц преобразований
vector<path> figure; // список ломаных очередного рисунка
```

Реакция на чтение команд, описанных нами при выполнении задания 3, остается прежней. Нам нужно определить порядок обработки дополнительных команд. У нас уже есть ветви оператора `if` для различных команд из файла:

```
if (cmd == "frame") { // размеры изображения
 ...
}
else if (cmd == "color") { // цвет линии
 ...
}
else if (cmd == "thickness") { // толщина линии
 ...
}
else if (cmd == "path") { // набор точек
 ...
}
```

Здесь многоточие заменяет блоки кода. Добавим к этой условной конструкции дополнительные ветви

```
else if (cmd == "model") { // начало описания нового рисунка

}
else if (cmd == "figure") { // формирование новой модели

}
else if (cmd == "translate") { // перенос

}
else if (cmd == "scale") { // масштабирование

}
else if (cmd == "rotate") { // поворот

}
else if (cmd == "pushTransform") { // сохранение матрицы в стек

}
else if (cmd == "popTransform") { // откат к матрице из стека

}
```

Заполним пробелы в этой конструкции.

В случае, если прочитана команда `model`, необходимо прочитать из потока 4 числа, из которых следует сформировать преобразование для рисунка, которое смещает его центр с началом координат и масштабирует его, чтобы он вписался в квадрат со стороной 2. Похожие действия мы выполняли в случае команды `frame`.

Итак, сначала опишем переменные и считаем их значения из файла

```
float mVcx, mVcy, mVx, mVy; // параметры команды model
s >> mVcx >> mVcy >> mVx >> mVy; // считываем значения переменных
```

После этого в матрице `initM` сохраним нужный перенос и масштабирование. Коэффициент масштабирования вычисляем тем же образом, что и при обработке команды `frame`, только сейчас нам заранее известно, что соотношение сторон квадрата, в который мы вписываем рисунок, равно единице, а сторона квадрата равна двум. Еще одно отличие в том, что в

нашем случае имеет место преобразование из правой системы координат в правую, поэтому отрицательного коэффициента масштабирования не требуется.

```
float S = mVx / mVy < 1 ? 2.f / mVy : 2.f / mVx;
// сдвиг точки привязки из начала координат в нужную позицию
// после которого проводим масштабирование
initM = scale(S) * translate(-mVcx, -mVcy);
```

Кроме того, следует очистить временный список `figure` для наполнения его ломаными, относящимися к новому рисунку.

```
figure.clear();
```

В случае получения команды `figure` нужно из списка `figure` и модельной матрицы сформировать объект `model` для текущего рисунка и поместить его в список `models`. Стоит учитывать, что модельная матрица для рисунка должна представлять собой преобразование в котором сначала производится начальная установка рисунка в начало координат, а потом проводится привязка, описанная командами `translate`, `scale` и `rotate`, т. е. она получается в результате произведения матриц `M` и `initM`.

```
models.push_back(model(figure, M * initM)); // добавляем рисунок в список
```

Обработка команд `translate`, `scale` и `rotate` сводится к считыванию параметров преобразования и домножению матрицы `M` на соответствующую матрицу.

В случае команды `translate`:

```
float Tx, Ty; // параметры преобразования переноса
s >> Tx >> Ty; // считываем параметры
M = translate(Tx, Ty) * M; // добавляем перенос к общему преобразованию
```

В случае команды `scale`:

```
float S; // параметр масштабирования
s >> S; // считываем параметр
M = scale(S) * M; // добавляем масштабирование к общему преобразованию
```

При организации поворота следует принять во внимание, что величина угла, которую мы считываем из файла, задается в градусах. Поэтому нужно сделать перевод значения параметра из градусов в радианы перед его передачей функции получения матрицы поворота. Здесь нам потребуется значение константы π, которое можем получить обратившись к `System::Math::PI`. Кроме того, нужно учесть, что функция `rotate` для параметра `theta` возвращает матрицу поворота на угол `theta` против часовой стрелки в левой системе координат. Здесь же мы имеем правую систему координат, поэтому следует передавать этой функции угол с обратным знаком.

```
float theta; // угол поворота в градусах
s >> theta; // считываем параметр
M = rotate(-theta / 180.f * Math::PI) * M; // добавляем поворот к общему преобразованию
```

Когда получена команда `pushTransform`, просто отправляем матрицу `M` в стек `transforms`.

```
transforms.push_back(M); // сохраняем матрицу в стек
```

На команду `popTransform` извлекаем матрицу из стека `transforms` и присваиваем её переменной `M`.

```
M = transforms.back(); // получаем верхний элемент стека
transforms.pop_back(); // выкидываем матрицу из стека
```

В результате процедура `btnOpen_Click` должна принять следующий вид.

```
private: System::Void btnOpen_Click(System::Object^  sender, System::EventArgs^  e) {
  if (openFileDialog->ShowDialog() == System::Windows::Forms::DialogResult::OK) {
    // в файловом диалоге нажата кновка OK
    // перезапись имени файла из openFileDialog->FileName в fileName
    wchar_t fileName[1024];  // переменная, в которой посимвольно сохраним имя файла
    for (int i = 0; i < openFileDialog->FileName->Length; i++)
      fileName[i] = openFileDialog->FileName[i];
    fileName[openFileDialog->FileName->Length] = '\0';

    // объявление и открытие файла
    ifstream in;
    in.open(fileName);
    if (in.is_open()) {
      // файл успешно открыт
      models.clear(); // очищаем имеющийся список рисунков
      // временные переменные для чтения из файла
      mat3 M = mat3(1.f); // матрица для получения модельной матрицы
      mat3 initM; // матрица для начального преобразования каждого рисунка
      vector<mat3> transforms; // стек матриц преобразований
      vector<path> figure; // список ломаных очередного рисунка
      float thickness = 2; // толщина со значением по умолчанию 2
      float r, g, b; // составляющие цвета
      r = g = b = 0; // значение составляющих цвета по умолчанию (черный)
      string cmd; // строка для считывания имени команды
      // непосредственно работа с файлом
      string str; // строка, в которую считываем строки файла
      getline(in, str); // считываем из входного файла первую строку
      while (in) { // если очередная строка считана успешно
        // обрабатываем строку
        if ((str.find_first_not_of(" \t\r\n") != string::npos) && (str[0] != '#')) {
          // прочитанная строка не пуста и не комментарий
          stringstream s(str); // строковый поток из строки str
          s >> cmd;
          if (cmd == "frame") { // размеры изображения
            s >> Vx >> Vy; // считываем глобальные значение Vx и Vy
            aspectFig = Vx / Vy; // обновление соотношения сторон
            float aspectRect = Wx / Wy; // соотношение сторон прямоугольника
            // смещение центра рисунка с началом координат
            mat3 T1 = translate(-Vx / 2, -Vy / 2);
            // масштабирование остается прежним, меняется только привязка
            // коэффициент увеличения при сохранении исходного соотношения сторон
            float S = aspectFig < aspectRect ? Wy / Vy : Wx / Vx;
            mat3 S1 = scale(S, -S);
            // сдвиг точки привязки из начала координат в нужную позицию
            mat3 T2 = translate(Wx / 2 + Wcx, Wcy - Wy / 2);
            // В initT совмещаем эти три преобразования (справа налево)
            initT = T2 * (S1 * T1);
            T = initT;
          }
          else if (cmd == "color") { // цвет линии
            s >> r >> g >> b; // считываем три составляющие цвета
          }
          else if (cmd == "thickness") { // толщина линии
            s >> thickness; // считываем значение толщины
          }
          else if (cmd == "path") { // набор точек
            vector<vec2> vertices; // список точек ломаной
            int N; // количество точек
            s >> N;
            string str1; // дополнительная строка для чтения из файла
            while (N > 0) { // пока не все точки считали
              getline(in, str1); // считываем в str1 из входного файла очередную строку
              // так как файл корректный, то на конец файла проверять не нужно
              if ((str1.find_first_not_of(" \t\r\n") != string::npos) && (str1[0] != '#')) {
                // прочитанная строка не пуста и не комментарий
                // значит в ней пара координат
                float x, y; // переменные для считывания
                stringstream s1(str1); // еще один строковый поток из строки str1
                s1 >> x >> y;
                vertices.push_back(vec2(x, y)); // добавляем точку в список
                N--; // уменьшаем счетчик после успешного считывания точки
              }
            }
            // все точки считаны, генерируем ломаную (path) и кладем ее в список figure
            figure.push_back(path(vertices, vec3(r, g, b), thickness));
          }
          else if (cmd == "model") { // начало описания нового рисунка
```

```
78            float mVcx, mVcy, mVx, mVy; // параметры команды model
79            s >> mVcx >> mVcy >> mVx >> mVy; // считываем значения переменных
80            float S = mVx / mVy < 1 ? 2.f / mVy : 2.f / mVx;
81            // сдвиг точки привязки из начала координат в нужную позицию
82            // после которого проводим масштабирование
83            initM = scale(S) * translate(-mVcx, -mVcy);
84            figure.clear();
85          }
86          else if (cmd == "figure") { // формирование новой модели
87            models.push_back(model(figure, M * initM));
88          }
89          else if (cmd == "translate") { // перенос
90            float Tx, Ty; // параметры преобразования переноса
91            s >> Tx >> Ty; // считываем параметры
92            M = translate(Tx, Ty) * M; // добавляем перенос к общему преобразованию
93          }
94          else if (cmd == "scale") { // масштабирование
95            float S; // параметр масштабирования
96            s >> S; // считываем параметр
97            M = scale(S) * M; // добавляем масштабирование к общему преобразованию
98          }
99          else if (cmd == "rotate") { // поворот
100           float theta; // угол поворота в градусах
101           s >> theta; // считываем параметр
102           M = rotate(-theta / 180.f * Math::PI) * M; // добавляем поворот к общему преобразованию
103         }
104         else if (cmd == "pushTransform") { // сохранение матрицы в стек
105           transforms.push_back(M); // сохраняем матрицу в стек
106         }
107         else if (cmd == "popTransform") { // откат к матрице из стека
108           M = transforms.back(); // получаем верхний элемент стека
109           transforms.pop_back(); // выкидываем матрицу из стека
110         }
111       }
112       // считываем очередную строку
113       getline(in, str);
114     }
115     Refresh();
116   }
117 }
118 }
```

1.5.5 Отрисовка изображения, заданного в мировой системе координат

Внесем изменения в обработчик события *Paint* чтобы изображение, заданное списком `models` корректно выводилось. В этой процедуре у нас уже есть цикл, который проводил обработку списка `figure` в прежнем варианте программы.

```
1  for (int i = 0; i < figure.size(); i++) {
2    path lines = figure[i]; // lines - очередная ломаная линия
3    Pen^ pen = gcnew Pen(Color::FromArgb(lines.color.x, lines.color.y, lines.color.z));
4    pen->Width = lines.thickness;
5
6    vec2 start = normalize(T * vec3(lines.vertices[0], 1.0)); // начальная точка первого отрезка
7    for (int j = 1; j < lines.vertices.size(); j++) { // цикл по конечным точкам (от единицы)
8      vec2 end = normalize(T * vec3(lines.vertices[j], 1.0)); // конечная точка
9      vec2 tmpEnd = end; // продублировали координаты точки для будущего использования
10     if (clip(start, end, minX, minY, maxX, maxY)) { // если отрезок видим
11       // после отсечения, start и end - концы видимой части отрезка
12       g->DrawLine(pen, start.x, start.y, end.x, end.y);  // отрисовка видимых частей
13     }
14     start = tmpEnd;  // конечная точка неотсеченного отрезка становится начальной точкой следующего
15   }
16 }
```

Теперь же у нас аналог списка `figure` присутствует в каждом элементе списка `models`. То есть, нам нужно организовать внешний цикл по списку `models`, в котором будет извлекаться список `figure` из каждого элемента, для которого, в свою очередь, будет выполняться приведенный выше цикл отрисовки. При этом в самом цикле, перед применением преобразования `T` следует не забыть применить преобразование начальной установки, заданное модельной матрицей.

Итак, сначала организуем внешний цикл вокруг имеющегося цикла. Для того, чтобы в имеющийся код добавлять минимальное количество изменений, введем переменную

`figure` на замену имевшейся ранее глобальной, которой будем присваивать список ломаных очередного рисунка.

```
for (int k = 0; k < models.size(); k++) { // цикл по рисункам
  vector<path> figure = models[k].figure; // список ломаных очередного рисунка
  ... // исходный цикл по элементам figure
}
```

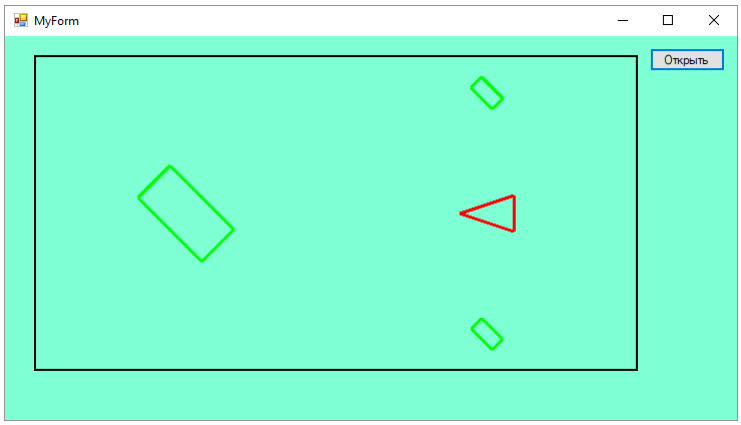
Внутри цикла дважды происходит умножение на матрицу `T`. Нам же теперь нужно умножать сначала на модельную матрицу, а потом на `T`. Чтобы многократно не вычислять произведение матрицы `T` на модельную матрицу, заведем перед циклом по `i` временную переменную `TM`, в которую сохраним его значение, а умножение на `T` заменим умножением на `TM`.

```
mat3 TM = T * models[k].modelM; // матрица общего преобразования рисунка
```

Обработчик события *Paint* примет следующий вид.

```
private: System::Void MyForm_Paint(System::Object^  sender, System::Windows::Forms::PaintEventArgs^  e) {
  Graphics^ g = e->Graphics;
  g->Clear(Color::Aquamarine);

  Pen^ rectPen = gcnew Pen(Color::Black, 2);
  g->DrawRectangle(rectPen, left, top, Wx, Wy);
  for (int k = 0; k < models.size(); k++) { // цикл по рисункам
    vector<path> figure = models[k].figure; // список ломаных очередного рисунка
    mat3 TM = T * models[k].modelM; // матрица общего преобразования рисунка
    for (int i = 0; i < figure.size(); i++) {
      path lines = figure[i]; // lines - очередная ломаная линия
      Pen^ pen = gcnew Pen(Color::FromArgb(lines.color.x, lines.color.y, lines.color.z));
      pen->Width = lines.thickness;

      vec2 start = normalize(TM * vec3(lines.vertices[0], 1.0)); // начальная точка первого отрезка
      for (int j = 1; j < lines.vertices.size(); j++) { // цикл по конечным точкам (от единицы)
        vec2 end = normalize(TM * vec3(lines.vertices[j], 1.0)); // конечная точка
        vec2 tmpEnd = end; // продублировали координаты точки для будущего использования
        if (clip(start, end, minX, minY, maxX, maxY)) { // если отрезок видим
          // после отсечения, start и end - концы видимой части отрезка
          g->DrawLine(pen, start.x, start.y, end.x, end.y);  // отрисовка видимых частей
        }
        start = tmpEnd;  // конечная точка неотсеченного отрезка становится начальной точкой следующего
      }
    }
  }
}
```

Теперь проект можно запустить. Растянем окно и откроем файл `Geometric.txt`. Получим изображение, подобное следующему.



1.6 Задание для самостоятельной работы

Задание 4

1. Создайте приложение, включающее в себя все, что было описано выше в качестве примера. Проект должен назваться Вашей фамилией, записанной латинскими буквами.
2. В итоговом проекте мы ввели прямоугольник (область видимости), в котором выводится изображение. Но повороты, отражения и масштабирование происходят все еще относительно центра окна. Переопределите реакции на нажатие клавиш **Q**, **E**, **R**, **Y**, **Z**, **X**, **U**, **J**, **I**, **K**, **O**, **L** так, чтобы преобразования проводились относительно центра области видимости.
3. Выберите для себя свободный вариант задания 4 на портале course.sgu.ru (Ссылка «Выбор варианта для задания 4»).
4. Создайте текстовый файл, описывающий изображение, по схеме, соответствующей Вашему варианту в задании 4. При этом, каждый зеленый прямоугольник схемы соответствуют экземпляру Вашего изображения в заданиях 2 и 3, а синий — изображению, данному вам в качестве примера в задании 3 в файле `Hare_full.txt`.
5. Архив получившегося проекта загрузите на портал как ответ на задание 4. В архив необходимо дополнительно включить текстовый файл с описанием Вашего изображения и текстовый файл `Geometric.txt`.

■

1.7 Контрольные вопросы

Подготовьте ответы на следующие вопросы.

1. Если мы хотим программным путем нарисовать на форме квадрат, который был бы всегда привязан к правому нижнему углу и имел бы постоянные размеры (например, 100 пискселей), что нам необходимо сделать и в каких процедурах?
2. Как можно получить первоначальное преобразование `initT` для привязки правого нижнего угла рисунка к правому нижнему углу прямоугольника, при котором ширина рисунка вдвое меньше ширины прямоугольника, если к рисунку применяется равномерное масштабирование?
3. Начальное преобразование `initT`, полученное при выполнении третьего задания, состояло из масштабирования и переноса, причем коэффициенты переноса зависели от коэффициентов масштабирования. Каким образом можно получить эквивалентное первоначальное преобразование, чтобы коэффициенты переноса не зависели от `S`?
4. Каково предназначение алгоритма Коэна—Сазерленда?
5. Какое максимальное число проходов цикла может сделать алгоритм Коэна—Сазерленда в наихудшем случае?
6. В какой системе координат мы запускаем в проекте алгоритм Коэна—Сазерленда: в локальной, мировой или системе координат экрана?
7. В комментариях к файлу `Clip.h` есть неточность, относительно наших запусков алгоритма Коэна—Сазерленда. В чем она заключается?
8. Как программным путем узнать, что в двоичном представлении целого числа в заданном разряде находится 0?
9. Как программным путем узнать, что в двоичном представлении целого числа в заданном разряде находится 1?
10. При создании реакции на команду `rotate` мы изменили знак угла в радианах.

Зачем мы это сделали? Почему нужно сделать именно так?

11. Как перевести величину угла из градусов в радианы? Укажите место в программе, где выполняется такое преобразование.
12. Что такое модельное преобразование? Укажите место в программе, где применяется модельное преобразование.
13. Укажите место в программе, где происходит переход от мировой системы координат в систему координат экрана.
14. Что в итоговом проекте определяется переменными `Vx` и `Vy` ?
15. Что из себя представляют элементы списка `models` ?
16. Как провести привязку к точке в мировой системе координат рисунка, заданного своими размерами и координатами точки привязки, с масштабированием в `S` раз, но без изменения ракурса? Выведите формулу преобразования.
17. Как задать входной файл в формате задания 4, чтобы изображение состояло только из Вашего рисунка без дополнений (в виде полей или преобразований)?
18. Можно ли было построить проект, который имел бы точно такую же функциональность, как в итоговом проекте, с таким же форматом входных файлов, но без глобального списка `models` , а только с единым списком `figure` для всех ломаных, представляющих изображение в мировой системе координат? Обоснуйте ответ.
19. Как можно изменить проект, чтобы рисунки в Вашем варианте задания не вписывались в прямоугольники, а полностью их заполняли?

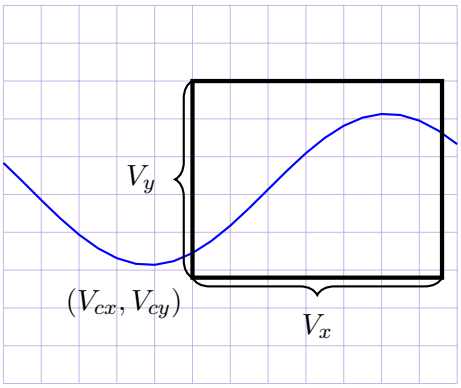
2. Построение графика функции

2.1 Предмет реализации

В ходе выполнения этого урока мы

- реализуем построение графика функции одной переменной;
- реализуем построение графика функции двух переменных в аксонометрической проекции.

Построение графика будем осуществлять следующим образом. В координатном пространстве графика определим область видимости — прямоугольник со сторонами, параллельными осям координат.



Ту часть графика, что попала в область видимости изобразим в прямоугольнике на форме: масштабируем область видимости вместе с графиком по осям Ox и Oy так, чтобы её контуры совпали с контурами прямоугольника на форме.

Координатные оси чертить не будем. Вместо этого добавим к графику координатную сетку, привязанную к окну, но с числовыми метками, соответствующими точкам графика.

В трехмерном случае (при построении графика функции двух аргументов) будем действовать подобным же образом, только в пространстве графика будем выбирать параллелепипед с ребрами, параллельными осям координат, который будем специальным образом совмещать с окном на форме.

После вывода графика в окно формы назначим горячие клавиши для того, чтобы график сдвигать в окне. При этом реализуем следующую стратегию. Для того, чтобы изменить выводимую часть графика в окне, мы переопределим прямоугольник области видимости: для смещения графика в окне достаточно изменить координаты левого нижнего угла прямоугольника, а для масштабирования графика — масштабировать прямоугольник в пространстве графика.

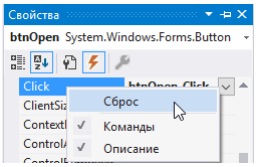
2.2 Предварительная подготовка

За основу проекта для пятого задания возьмем уже готовый проект, получившийся в результате выполнения четвертого задания.

Сделайте копию папки с проектом четвертого задания (чтобы не повторять предварительные установки). Имя проекта должно остаться без изменений.

Новый проект не предполагает некоторые из разработанных ранее функциональностей, поэтому выполним его предварительную очистку.

Прежде всего, мы не будем осуществлять здесь никакой загрузки файлов, поэтому нам не понадобится кнопка `btnOpen` и обработчик её нажатия. Для того, чтобы их удалить, сначала перейдите в конструктор формы, откройте окно свойств кнопки *Открыть*, в окне свойств, на странице *События* найдите событие *Click*. Щелкните правой кнопкой мыши на имени события и в контекстном меню выберите *Сброс*. Так мы отвязали от кнопки *Открыть* соответствующий обработчик события. Теперь в конструкторе формы удалите кнопку.



Перейдите к коду `MyForm.h`. В нем можно удалить процедуру `btmOpen_Click`.

Мы не будем использовать здесь описаний рисунков и не будем организовывать потоковый ввод-вывод. Поэтому в коде `MyForm.cpp` можно убрать строки

```
#include <fstream>
#include <sstream>
#include "Figure.h"
```

после чего из проекта можно удалить файл заголовка `Figure.h`: в обозревателе решений нажмите правую кнопку мыши на заголовке файла `Figure.h` и из контекстного меню выберите *Удалить*.

Вернемся к коду `MyForm.h`. В блоке описания глобальных переменных следует убрать описание списка рисунков и нам больше не нужна переменная, отвечающая за соотношение сторон изображения.

```
float aspectFig; // соотношение сторон рисунка
vector<model> models;
```

В обработчике события *Paint* уберем основной цикл, отвечающий за отрисовку изображения (оставим только вычерчивание прямоугольника области видимости). Процедура `MyForm_Paint` пример вид

```
private: System::Void MyForm_Paint(System::Object^  sender, System::Windows::Forms::PaintEventArgs^  e) {
  Graphics^ g = e->Graphics;
  g->Clear(Color::Aquamarine);

  Pen^ rectPen = gcnew Pen(Color::Black, 2);
  g->DrawRectangle(rectPen, left, top, Wx, Wy);

}
```

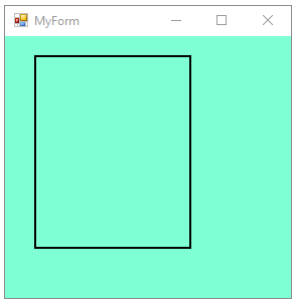
Так как мы удалили процедуру, считывающую файл, то удалили и инициализацию матриц `initT` и `T`. Проведем инициализацию этих матриц в обработчике события *Load*.

```
initT = mat3(1.f);
T = initT;
```

В обработчике события *KeyDown* удалите определение всех вспомогательных переменных и реакции на нажатие всех клавиш, кроме `Keys::Escape`. Обработчик события примет следующий вид.

```
private: System::Void MyForm_KeyDown(System::Object^  sender, System::Windows::Forms::KeyEventArgs^  e) {
  switch (e->KeyCode) {
  case Keys::Escape:
    T = initT;
    break;
  default:
    break;
  }
  Refresh();
}
```

Проект можно запустить. Должно выводиться растяжимое окно с прямоугольником.



2.3 Реализация дополнительного математического аппарата

При построении двумерного графика нам потребуется реализация умножения матриц порядка 2. При дальнейшем построении трехмерного графика потребуются дополнительно операции с векторами и матрицами порядка 4. Внесем соответствующие изменения в файл `Matrix.h`.

Для реализации класса `mat2` нам понадобится полноценный класс `vec2`, включающий операцию `*` и операцию скалярного произведения. Доопределим этот класс по аналогии с классом `vec3`. Получим

```
class vec2 {
public:
  float x, y;
  vec2(float a, float b) : x(a), y(b) {}
  vec2() {}
  vec2& operator*=(const vec2& v) {
```

```
    x *= v.x;
    y *= v.y;
    return *this;
  }
  const vec2 operator*(const vec2& v) {
    return vec2(*this) *= v;
  }
  float& operator[](int i) {
    return ((float*)this)[i];
  }
};

float dot(vec2 v1, vec2 v2) {
  vec2 tmp = v1 * v2;   // вычисляем произведения соответствующих координат
  return tmp.x + tmp.y; // и возвращаем их сумму
}
```

Определим класс `mat2` после описания `mat3`. Описание `mat2` будет отличаться от `mat3` отсутствием третьей строки, типом строк `vec2` вместо `vec3` и заменой литерала 3 на 2 в циклах. Получим следующий фрагмент кода.

```
class mat2 {
public:
  vec2 row1, row2;

  mat2(vec2 r1, vec2 r2) : row1(r1), row2(r2) {}
  mat2(float a) {
    row1 = vec2(a, 0.f);
    row2 = vec2(0.f, a);
  }
  mat2() {}

  vec2& operator[](int i) {
    return ((vec2*)this)[i]; // массив значений типа vec2
  }

  mat2 transpose() {
    mat2 tmp(*this); // делаем временную копию матрицы
    for (int i = 0; i < 2; i++)
      for (int j = 0; j < 2; j++)
        (*this)[i][j] = tmp[j][i]; // заменяем элементы текущего объекта
                                   // из временной копии
    return *this;
  }

  const vec2 operator* (const vec2 &v) {
    vec2* res = new(vec2);              // создаем новый вектор (для результата)
    for (int i = 0; i < 2; i++) {
      (*res)[i] = dot((*this)[i], v); // i-й элемент вектора - скалярное произведение
    }
    return *res;
  }

  mat2& operator*= (const mat2 &m) {
    mat2 A(*this), B(m); // создаем копии исходных матриц
    B.transpose();       // транспонируем вторую матрицу
    for (int i = 0; i < 2; i++)
      (*this)[i] = A * B[i]; // в i-ю строку текущего объекта записываем
                  // результат перемножения первой матрицы с i-й строкой
```

```
                    // транспонированной матрицы,
    return (*this).transpose(); // транспонируем текущий объект, получаем результат
  }
  const mat2 operator* (const mat2 &m) {
    return mat2(*this) *= m;
  }
};
```

Добавим в описание класса описание дополнительного конструктора для получения матрицы порядка 2 из матрицы третьего порядка отсечением в ней последней строки и последнего столбца.

```
mat2(mat3 m) {
  row1 = vec2(m[0][0], m[0][1]);
  row2 = vec2(m[1][0], m[1][1]);
}
```

Теперь по аналогии с классом `vec3` перед классом `mat3` опишем класс `vec4`, добавив в качестве составной части дополнительную координату `a`.

```
class vec4 {
public:
  float x, y, z, a;
  vec4() {}
  vec4(float a, float b, float c, float d) : x(a), y(b), z(c), a(d) {}
  vec4(vec3 v, float c) : vec4(v.x, v.y, v.z, c) {}
  vec4& operator*=(const vec4& v) {
    x *= v.x;
    y *= v.y;
    z *= v.z;
    a *= v.a;
    return *this;
  }
  const vec4 operator*(const vec4& v) {
    return vec4(*this) *= v; // делаем временную копию текущего объекта,
                  // которую домножаем на данный вектор,
                  // и возвращаем ее как результат
  }
  float& operator[](int i) {
    return ((float*)this)[i]; // ссылку на текущий объект рассматриваем как ссылку
                   // на нулевой элемент массива значений типа float,
                   // после чего, обращаемся к его i-му элементу
  }
};
```

После описания класса четырехмерных векторов опишем функции для их скалярного произведения и нормализации (перехода от однородных координат в евклидовы).

```
float dot(vec4 v1, vec4 v2) {
  vec4 tmp = v1 * v2;  // вычисляем произведения соответствующих координат
  return tmp.x + tmp.y + tmp.z + tmp.a; // и возвращаем их сумму
}

vec3 normalize(vec4 v) {
  // делим первые три координаты на значение четвертой
  return vec3(v.x / v.a, v.y / v.a, v.z / v.a);
}
```

Перед описанием `mat3` опишем класс для четырехмерных матриц `mat4` (опять по аналогии с `mat3`).

```
class mat4 {
public:
  vec4 row1, row2, row3, row4;

  mat4(vec4 r1, vec4 r2, vec4 r3, vec4 r4) : row1(r1), row2(r2), row3(r3), row4(r4) {}
  mat4(float a) {
    row1 = vec4(a, 0.f, 0.f, 0.f);
    row2 = vec4(0.f, a, 0.f, 0.f);
    row3 = vec4(0.f, 0.f, a, 0.f);
    row4 = vec4(0.f, 0.f, 0.f, a);
  }
  mat4() {}

  vec4& operator[](int i) {
    return ((vec4*)this)[i]; // массив значений типа vec4
  }

  mat4 transpose() {
    mat4 tmp(*this); // делаем временную копию матрицы
    for (int i = 0; i < 4; i++)
      for (int j = 0; j < 4; j++)
        (*this)[i][j] = tmp[j][i]; // заменяем элементы текущего объекта
                                   // из временной копии
    return *this;
  }

  const vec4 operator* (const vec4 &v) {
    vec4* res = new(vec4);              // создаем новый вектор (для результата)
    for (int i = 0; i < 4; i++) {
      (*res)[i] = dot((*this)[i], v); // i-й элемент вектора - скалярное произведение
    }
    return *res;
  }

  mat4& operator*= (const mat4 &m) {
    mat4 A(*this), B(m); // создаем копии исходных матриц
    B.transpose();       // транспонируем вторую матрицу
    for (int i = 0; i < 4; i++)
      (*this)[i] = A * B[i]; // в i-ю строку текущего объекта записываем
                             // результат перемножения первой матрицы с i-й строкой
                             // транспонированной матрицы,
    return (*this).transpose(); // транспонируем текущий объект, получаем результат
  }
  const mat4 operator* (const mat4 &m) {
    return mat4(*this) *= m;
  }

};
```

Наконец, в класс `mat3` добавим дополнительный конструктор, заполняющий матрицу третьего порядка из первых трех строк и первых трех столбцов объекта `mat4`.

```
mat3(mat4 m) {
  row1 = vec3(m.row1.x, m.row1.y, m.row1.z);
  row2 = vec3(m.row2.x, m.row2.y, m.row2.z);
  row3 = vec3(m.row3.x, m.row3.y, m.row3.z);
}
```

2.4 Построение графика функции одной переменной

2.4.1 Параметры прямоугольника

Итак, представим, что в пространстве графика функции задан прямоугольник со сторонами, параллельными осям координат. Считаем, что система координат правая. Параметры прямоугольника:

- координаты левого нижнего угла (V_{cx}, V_{cy});
- размеры прямоугольника: по горизонтали — V_x, по вертикали — V_y.

В нашей программе представим эти параметры как двумерную точку `Vc` и двумерный вектор `V`. В коде `MyForm.h` в блоке описания глобальных переменных вместо

```
float Vx; // размер рисунка по горизонтали
float Vy; // размер рисунка по вертикали
```

опишем эти два вектора

```
vec2 Vc; // координаты левого нижнего угла
vec2 V;  // размеры прямоугольника в пространстве графика
```

Эти переменные будут хранить у начальные значения этих параметров. Кроме них создадим аналогичные переменные для рабочих значений этих параметров.

```
vec2 Vc_work, V_work; // рабочие параметры прямоугольника
```

Значение величин `Vc` и `V` зададим жестко в обработчике события *Load*. Пусть это будет квадрат со стороной 4, с центром в начале координат.

```
Vc = vec2(-2.f, -2.f);
V = vec2(4.f, 4.f);
```

Матрица `T` как и раньше будет представлять совмещение всех накопленных преобразований. Тогда рабочее значение `Vc_work` вычисляется просто умножением этой матрицы на однородные координаты точки `Vc`.

На значение вектора `V` влияют все преобразования, совмещенные в `T`, кроме переносов. Матрицу преобразования, в которой совмещены все преобразования за исключением переносов, можно получить из матрицы `T` отсечением последней строки и последнего столбца.

Значения рабочих переменных `Vc_work` и `V_work` нужно инициализировать при загрузке формы, а затем пересчитывать при обновлении изображения в окне формы после нажатия горячих клавиш. Так как обновление значений будет происходить в одном и том же порядке, оформим его в виде процедуры. После процедуры `rectCalc` добавим описание процедуры `worldRectCalc` без параметров.

```
private: System::Void worldRectCalc() {

}
```

В ней сначала пересчитаем значение `Vc`.

```
Vc_work = normalize(T * vec3(Vc, 1.f));
```

Для пересчета `V_work` создадим матрицу второго порядка из матрицы `T`, которую умножим на `V`.

```
V_work = mat2(T) * V;
```

Получим следующий общий вид процедуры `worldRectCalc`.

```
private: System::Void worldRectCalc() {
  Vc_work = normalize(T * vec3(Vc, 1.f));
  V_work = mat2(T) * V;
}
```

Теперь вызовем эту процедуру из обработчика события *Load* после инициализации `T`, `Vc` и `V`.

```
worldRectCalc();
```

Обработчик события *Load* примет вид

```
private: System::Void MyForm_Load(System::Object^  sender, System::EventArgs^  e) {
  Vc = vec2(-2.f, -2.f);
  V = vec2(4.f, 4.f);
  initT = mat3(1.f);
  T = initT;
  rectCalc();
  worldRectCalc();
}
```

Теперь реализуем отрисовку самого графика.

2.4.2 Параметры отрезков

Сначала определим функцию, для которой будем рисовать график. После описания процедуры `worldRectCalc` добавим описание функции

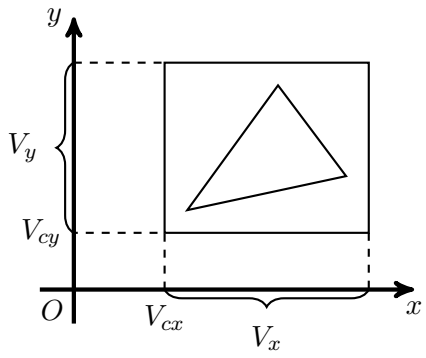
```
private: float f(float x) {
  return x * sin(x);
}
```

График функции, в отличие от изображений в предыдущих проектах, в общем случае нельзя представить в виде конечного набора отрезков. Более того, можно проводить бесконечную детализацию элементов графика (в программе такая детализация ограничивается лишь точностью используемых вычислений). Но в прямоугольнике на форме — конечное число точек (пикселей). Поэтому степень разбиения деталей графика будем выбирать исходя из разрешения прямоугольника на форме.

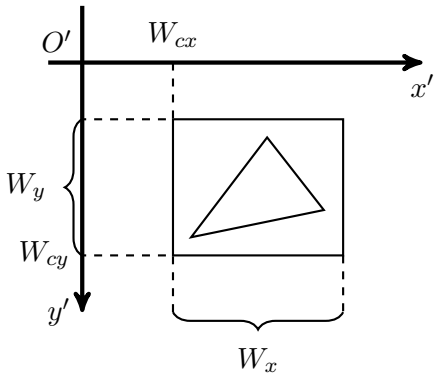
Видимую часть графика будем представлять набором отрезков, длина проекции которых на ось Ox равна 1. Пользуемся тем фактом, что у функции для каждого значения аргумента может быть только один результат. Поэтому организуем цикл по всем x в прямоугольнике на форме и для каждого значения найдем соответствующее значение y. Соседние такие точки графика будем соединять отрезком.

В предыдущих проектах отрезки задавались в мировой системе координат, которые мы пересчитывали в систему координат экрана. Здесь же нам следует отталкиваться от системы координат экрана: для определенных значений координат на форме необходимо выявить какие фрагменты им соответствуют в мировой системе координат.

Пусть в мировой системе координат выделен прямоугольник с параметрами (V_{cx}, V_{cy}), V_x, V_y,



а окно на форме имеет соответствующие параметры (W_{cx}, W_{cy}), W_x, W_y



Совмещение первого прямоугольника со вторым (переход от мировой системы координат к системе координат экрана) можно выразить соотношением

$$\begin{cases} x' = W_{cx} + \dfrac{x - V_{cx}}{V_x} W_x, \\ y' = W_{cy} - \dfrac{y - V_{cy}}{V_y} W_y. \end{cases}$$

Здесь (x, y) — координаты в мировой системе координат, (x', y') — координаты на экране.

Так как в нашей программе будет цикл по координате x на экране, то первое соотношение будет использоваться в обратную сторону: для каждого x' будем вычислять x

$$x = V_{cx} + (x' - W_{cx}) \frac{V_x}{W_x}.$$

Из-за того, что шаг цикла по x' равен одному, то $(x' - W_{cx})$ есть номер точки, а значение V_x/W_x есть шаг по x в мировой системе координат, соответствующий единице в системе координат экрана. Это равенство можно перефразировать в виде рекуррентного соотношения:

$$\begin{cases} x_0 = V_{cx}, \\ x_i = x_{i-1} + \dfrac{V_x}{W_x}, \end{cases}$$

где x_i — координаты x в мировой системе координат, соответствующие точкам в системе координат экрана, перечисленным слева направо.

Стоит отметить, что прямоугольник в мировой системе координат может быть выбран так, что не для каждого x соответствующая точка графика попадает в прямоугольник. Будем решать эту задачу с помощью процедуры отсечения, реализованной в предыдущем уроке.

2.4.3 Реализация отрисовки

Итак, перейдем к отрисовке графика. Будем это делать в обработчике события *Paint*.

Сначала опишем перо, которым будем вычерчивать график.

```
Pen^ pen = gcnew Pen(Color::Blue, 1);
```

Сохраним значение шага по x в мировой системе координат.

```
float deltaX = V_work.x / Wx; // шаг по x в мировых координатах
```

Как и в предыдущих проектах будем вычерчивать множество отрезков от `start` до `end`. Определим точку начала первого отрезка.

```
vec2 start; // точка начала отрезка в координатах экрана
float x, y; // переменные для координат точки в мировой СК
start.x = Wcx; // для начальной точки первого отрезка устанавливаем координату x
x = Vc_work.x; // координата x начальной точки первого отрезка в мировых координатах
y = f(x); // координата y начальной точки в мировых координатах
// вычисляем соответствующее значение в координатах экрана
start.y = Wcy - (y - Vc_work.y) / V_work.y * Wy;
```

Организуем цикл до тех пор, пока координата x точки начала отрезка не достигнет правого крайнего значения в прямоугольнике на форме — `maxX`.

```
while (start.x < maxX) {

}
```

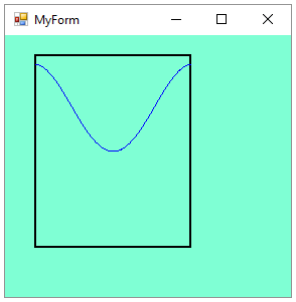
В цикле сначала вычислим точку конца отрезка.

```
vec2 end;// точка конца отрезка в координатах экрана
end.x = start.x + 1.f; // координата x отличается на единицу
x += deltaX; // координата x конечной точки отрезка в мировых координатах
y = f(x); // координата y конечной точки в мировых координатах
// вычисляем соответствующее значение в координатах экрана
end.y =  Wcy - (y - Vc_work.y) / V_work.y * Wy;
```

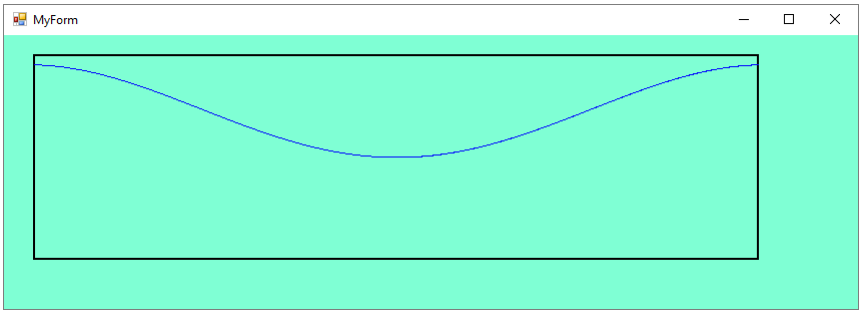
Проведем отсечение отрезка относительно области видимости на форме, предварительно сохранив координаты конца отрезка для следующей итерации в качестве начальной точки.

```
vec2 tmpEnd = end;
bool visible = clip(start, end, minX, minY, maxX, maxY);
if (visible) { // если отрезок видим
  // после отсечения, start и end - концы видимой части отрезка
  g->DrawLine(pen, start.x, start.y, end.x, end.y);  // отрисовка видимых частей
}
// конечная точка неотсеченного отрезка становится начальной точкой следующего
start = tmpEnd;
```

Проект можно запустить. В прямоугольнике будет выводиться график.



Окно можно растянуть. Пропорционально растяжению окна будет изменяться и вид графика в окне.



2.4.4 Назначение горячих клавиш

Добавим горячие клавиши, меняющие вид графика в окне.

Назначим на клавишу **A** такой сдвиг прямоугольника в мировой системе координат влево, что в системе координат экрана приведет к сдвигу графика вправо в окне на один пиксел (т. е. мы смотрим на график как бы через объектив камеры, и такой сдвиг подобен сдвигу камеры влево).

Как мы уже оговаривали ранее, расстоянию в один пиксел по горизонтали в системе координат экрана соответствует расстояние в мировой системе координат равное V_x/W_x. Поэтому, для реализации упомянутой операции нужно применить к точке `Vc_work` преобразование перенос с этим значением сдвига по оси x в отрицательном направлении. Так как все такие преобразования накапливаются в матрице `T`, домножим эту матрицу на матрицу соответствующего преобразования

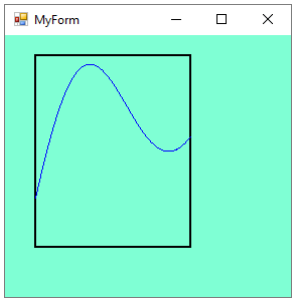
В обработчике события *KeyDown* добавим реакцию на нажатие клавиши **A**.

```
case Keys::A:
  T = translate(-V_work.x / Wx, 0.f) * T; // сдвиг графика вправо на один пиксел
  break;
```

Чтобы изображение в окне пересчиталось в соответствии с примененным преобразованием, нужно после оператора `switch` перед вызовом процедуры `Refresh` добавить вызов процедуры пересчета параметров прямоугольника в мировой системе координат.

```
worldRectCalc();
```

Теперь при запуске график можно сдвинуть вправо в окне.



Добавим преобразование равномерного увеличения прямоугольника в мировой системе координат относительно его центра в 1.1 раза (такое преобразование будет подобно отъезду камеры от объекта, т. е. приведет к увеличению обзора или, другими словами, уменьшению деталей графика в окне).

Преобразование масштабирования относительно центра будет складываться из трех элементарных преобразований: перенос начала координат в центр окна, масштабирование, перенос начала координат в исходную точку.

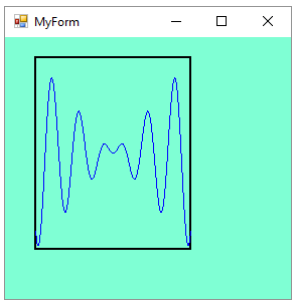
Координаты центра окна можно вычислить из текущих значений величин `Vc_work` и `V_work`. Добавим вычисление этих значений перед оператором `switch`.

```
float centerX = Vc_work.x + V_work.x / 2; // координаты центра прямоугольника
float centerY = Vc_work.y + V_work.y / 2; // в мировой системе координат
```

Теперь назначим реакцию на клавишу **Z**: реализуем упомянутую последовательность преобразований.

```
case Keys::Z:
  T = translate(-centerX, -centerY) * T; // перенос начала координат в центр
  T = scale(1.1) * T; // масштабирование относительно начала координат
  T = translate(centerX, centerY) * T; // возврат позиции начала координат
  break;
```

Нажатие клавиши **Z** в запущенном проекте приведет к уменьшению графика.



Так как на клавишу **Escape** установлена замена матрицы `T` на единичную матрицу, её нажатие приведет сбросу всех преобразований и вычерчиванию графика в первоначальном виде.

Обработчик события *KeyDown* на данном этапе примет вид.

```
1 private: System::Void MyForm_KeyDown(System::Object^  sender, System::Windows::Forms::KeyEventArgs^  e) {
2   float centerX = Vc_work.x + V_work.x / 2; // координаты центра прямоугольника
3   float centerY = Vc_work.y + V_work.y / 2; // в мировой системе координат
4   switch (e->KeyCode) {
5   case Keys::Escape:
6     T = initT;
7     break;
```

```
  case Keys::A:
    T = translate(-V_work.x / Wx, 0.f) * T; // сдвиг графика вправо на один пиксел
    break;
  case Keys::Z:
    T = translate(-centerX, -centerY) * T; // перенос начала координат в центр
    T = scale(1.1) * T; // масштабирование относительно начала координат
    T = translate(centerX, centerY) * T; // возврат позиции начала координат
    break;
  default:
    break;
  }
  worldRectCalc();
  Refresh();
}
```

2.4.5 Построение графика функции, определенной не на всем пространстве аргументов

Предположим, что нам нужно изобразить график функции, определенной не на всей числовой оси. Заменим тело функции `f`, чтобы она возвращала значение такой функции

```
return x * sin(log (x));
```

Эта функция определена только для положительных значений `x`.

Для подобных функций необходимо изменить порядок вывода графика в окне: прежде чем пытаться вычислить координаты концов очередного отрезка, необходимо выяснить, определена ли функция на заданном значении x. Для такой проверки определим функцию `f_exists`. Пусть эта функция получает аргумент x и возвращает истину, если функция `f` для него определена. Опишем `f_exists` для нашей функции `f` (сразу же после описания функции `f`).

```
private: bool f_exists(float x) {
  return x > 0;
}
```

Теперь перед обратимся к обработчику события *Paint*. Перед вычислением координат начальной точки опишем логическую переменную `hasStart`, которой будем присваивать истину, когда начальная точка отрезка определена.

```
bool hasStart;
```

Вычисление значения ординаты начальной точки

```
y = f(x); // координата y начальной точки в мировых координатах
// вычисляем соответствующее значение в координатах экрана
start.y = Wcy - (y - Vc_work.y) / V_work.y * Wy;
```

заменим на условную конструкцию, в которой будем производить это вычисление только тогда, когда функция в этой точке определена.

```
hasStart = f_exists(x);
if (hasStart) {
  y = f(x); // координата y начальной точки в мировых координатах
  // вычисляем соответствующее значение в координатах экрана
  start.y = Wcy - (y - Vc_work.y) / V_work.y * Wy;
}
```

После описания переменной `end` добавим описание `hasEnd`.

```
bool hasEnd;
```

Теперь проведем замену блока

```
y = f(x); // координата y конечной точки в мировых координатах
// вычисляем соответствующее значение в координатах экрана
end.y = Wcy - (y - Vc_work.y) / V_work.y * Wy;
```

на подобный вышеупомянутый условный блок.

```
hasEnd = f_exists(x);
if (hasEnd) {
  y = f(x); // координата y начальной точки в мировых координатах
  // вычисляем соответствующее значение в координатах экрана
  end.y = Wcy - (y - Vc_work.y) / V_work.y * Wy;
}
```

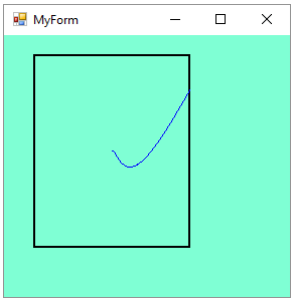
Внесем изменения в вычисление значения переменной `visible` : отрезок будет видимым, если у него есть начальная точка, есть конечная точка, и после отсечения от него остается видимая часть.

```
bool visible = hasStart && hasEnd && clip(start, end, minX, minY, maxX, maxY);
```

Наконец, перед переходом к следующей итерации в конце цикла следует присвоить значение `hasEnd` переменной `hasStart` .

```
hasStart = hasEnd;
```

Запуск проекта приведет к выводу графика функции.



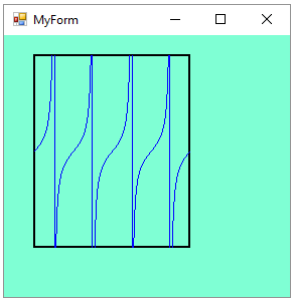
Такая стратегия построения графика кажется вполне удовлетворительной. Но давайте еще раз поменяем функцию, для которой строится график. Пусть функция такова, что разрыв в области определения приводит к диаметральным значениям функции по сторонам разрыва. Такой функцией, например, может являться функция тангенс. Заменим тело функции `f` на следующее.

```
return tan(x);
```

Если придерживаться предыдущей стратегии, то в функции `f_exists` будем возвращать `false` для тех аргументов, для которых тангенс не определен. Тангенс определен для тех значений аргумента, при которых косинус не равен нулю. Заменим тело функции `f_exists` соответствующим оператором.

```
return cos(x) != 0.f;
```

Но после запуска окажется, что в каждой точке разрыва будет присутствовать вертикальная линия.



Дело в том, что точки разрыва — единичные точки на вещественной оси, а когда мы вычисляем вещественные координаты концов отрезков, маловероятно, что точка, соответствующая определенному значению x на экране попадет точно в точку разрыва. Поэтому присутствуют отрезки (вертикальные линии), левый конец которых перед точкой разрыва, а правый — за точкой разрыва.

Можно избежать наличие таких отрезков, если мы скажем, что функция `f` не существует не только в точке разрыва, но и в некоторой её окрестности. Ширину окрестности стоит выбирать в зависимости от масштаба графика. Достаточно, чтобы диаметр окрестности был не больше, чем длина отрезка в мировых координатах, соответствующая одному пикселу на экране, т. е. V_x/W_x.

Для нашей функции `f` точки разрыва встречаются с шагом π в обе стороны от $x = \pi/2$. В качестве критерия определенности функции для конкретного x можно было бы взять проверку условия

$$\left| x - \frac{\pi}{2} - \pi \cdot \left\lfloor \frac{x - \frac{\pi}{2}}{\pi} \right\rfloor \right| > \frac{\frac{V_x}{W_x}}{2},$$

но такие вычисления накапливают много неточностей из-за множества округлений.

Более корректно можно сформулировать условие так

$$|\cos x| > \left| \cos\left(\frac{\pi}{2} + \frac{\frac{V_x}{W_x}}{2} \right) \right|,$$

Но мы здесь воспользуемся фактом, что арккосинус возвращает значения от 0 до π. Тогда можем критерий определенности функции выразить условием

$$\left| \arccos(\cos x) - \frac{\pi}{2} \right| < \frac{\frac{V_x}{W_x}}{2}.$$

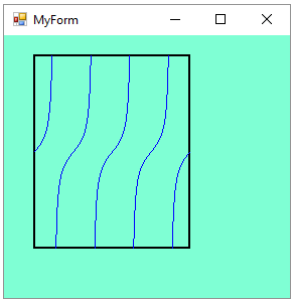
Домножим обе части неравенства на 2, получим

$$|2 \arccos(\cos x) - \pi| < \frac{V_x}{W_x}.$$

Переопределим выражение в операторе `return` функции `f_exists` соответствующим образом.

```
return fabs(2.f * acos(cos(x)) - Math::PI) > V_work.x / Wx;
```

Можно обнаружить, что вертикальные линии пропали.



Чтобы не вычислять значение выражение V_x/W_x при каждом запуске `f_exists`, добавим для этой функции дополнительный аргумент `delta`, которому будем передавать уже вычисленное значение этой дроби. Функция `f_exists` примет вид:

```
1 private: bool f_exists(float x, float delta) {
2   return fabs(2.f * acos(cos(x)) - Math::PI) > delta;
3 }
```

При вызове `f_exists` в обработчике события *Paint* подставим в качестве второго аргумента значение `deltaX`.

2.4.6 Раскраска графика

Давайте изменим программу так, чтобы цвет графика менялся в зависимости от значения функции. Пусть цвет графика изменяется от синего до зеленого снизу вверх в нижней половине области видимости и от зеленого до красного — в верхней половине.

Будем получать цвет из трех компонент модели RGB, в которой значения компонент будут изменяться пропорционально вертикальному положению точки в прямоугольнике.

Высота прямоугольника — V_y. Таким образом, если значение y от V_{cy} до $V_{cy} + V_y/2$, то цвет точки должен изменяться от синего до зеленого. Это означает, что синяя компонента изменяется от 255 до 0, зеленая — от 0 до 255, а красная равна нулю. Когда y располагается в диапазоне от $V_{cy} + V_y/2$ до $V_{cy} + V_y$ — цвет точки изменяется от зеленого до красного, т. е. зеленая компонента изменяется от 255 до 0, красная — от 0 до 255, а синяя равна нулю.

Если координата y попала в диапазон от V_{cy} до $V_{cy} + V_y/2$, то зеленую компоненту можно вычислить как произведение

$$g = 255 \cdot \frac{y - V_{cy}}{\frac{V_y}{2}} = 510 \cdot \frac{y - V_{cy}}{V_y}.$$

Синюю компоненту можно получить как разность

$$b = 255 - g.$$

В верхней половине красная компонента получается по формуле

$$r = 255 \cdot \frac{y - \left(V_{cy} + \frac{V_y}{2}\right)}{\frac{V_y}{2}} = 255 \cdot \left(\frac{y - V_{cy}}{\frac{V_y}{2}} - 1\right) = 255 \cdot \frac{y - V_{cy}}{\frac{V_y}{2}} - 255 = 510 \cdot \frac{y - V_{cy}}{V_y} - 255.$$

Зеленая же компонента вычисляется как разность

$$g = 255 - r = 255 - 510 \cdot \frac{y - V_{cy}}{V_y} + 255 = 510 - 510 \cdot \frac{y - V_{cy}}{V_y}.$$

Стоит отметить, что величина $(y - V_{cy})/V_y$ для точек внутри области видимости изменяется от 0 до 1. Эта величина участвует при вычислении значения `end.y` в цикле отрисовки графика. Воспользуемся этим.

Будем определять цвет отрезка как цвет конечной точки отрезка. В начале цикла опишем переменную `deltaY` для сохранения значения $(y - V_{cy})/V_y$, а так же переменные `red`, `green`, `blue` — для значений компонент RGB.

```
float deltaY; // высота точки в прямоугольнике (доля общей высоты)
float red, green, blue; // компоненты цвета отрезка
```

Строку вычисления `end.y`

```
1 // вычисляем соответствующее значение в координатах экрана
2 end.y = Wcy - (y - Vc_work.y) / V_work.y * Wy;
```

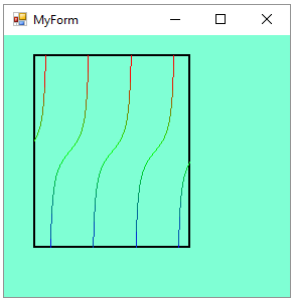
заменим строками с предварительным вычислением и сохранением высоты точки.

```
// вычисляем соответствующее значение в координатах экрана
deltaY = (y - Vc_work.y) / V_work.y;
end.y = Wcy - deltaY * Wy;
```

Теперь, в случае видимости отрезка, непосредственно перед его отрисовкой, вычислим значения `red`, `green`, `blue` и переопределим цвет пера `pen`.

```
if (deltaY > 1.f) deltaY = 1.f; // нормализуем значение высоты точки
if (deltaY < 0.f) deltaY = 0.f; // на случай, если отрезок отсекался
green = 510.f * deltaY; // предварительное вычисление произведения
if (deltaY < 0.5) { // если точка ниже середины области видимости
  // компонента зеленого уже вычислена
  blue = 255.f - green; // синий дополняет зеленый
  red = 0.f; // красный равен нулю
}
else { // если точка не ниже середины
  blue = 0.f; // синий равен нулю
  red = green - 255.f;   // вычисляем красный и зеленый
  green = 510.f - green; // с использованием вычисленного произведения
}
pen->Color = Color::FromArgb(red, green, blue); // меняем цвет пера
```

После запуска проекта получим график, изменяющий свой цвет по мере изменения координаты y.



Обработчик события *Paint* на данном этапе примет следующий вид.

```
private: System::Void MyForm_Paint(System::Object^  sender, System::Windows::Forms::PaintEventArgs^  e) {
  Graphics^ g = e->Graphics;
  g->Clear(Color::Aquamarine);

  Pen^ rectPen = gcnew Pen(Color::Black, 2);
  g->DrawRectangle(rectPen, left, top, Wx, Wy);

  Pen^ pen = gcnew Pen(Color::Blue, 1);
  float deltaX = V_work.x / Wx; // шаг по x в мировых координатах

  bool hasStart;

  vec2 start, end; // концы отрезка в координатах экрана
  float x, y; // переменные для координат точки в мировой СК
  start.x = Wcx; // для начальной точки первого отрезка устанавливаем координату x
  x = Vc_work.x; // координата x начальной точки первого отрезка в мировых координатах
  hasStart = f_exists(x, deltaX);
  if (hasStart) {
    y = f(x); // координата y начальной точки в мировых координатах
    // вычисляем соответствующее значение в координатах экрана
    start.y = Wcy - (y - Vc_work.y) / V_work.y * Wy;
  }
  while (start.x < maxX) {
    vec2 end;// точка конца отрезка в координатах экрана
    bool hasEnd;
    float deltaY; // высота точки в прямоугольнике (доля общей высоты)
    float red, green, blue; // компоненты цвета отрезка
    end.x = start.x + 1.f; // координата x отличается на единицу
    x += deltaX; // координата x конечной точки отрезка в мировых координатах
    hasEnd = f_exists(x, deltaX);
    if (hasEnd) {
      y = f(x); // координата y начальной точки в мировых координатах
      // вычисляем соответствующее значение в координатах экрана
      deltaY = (y - Vc_work.y) / V_work.y;
      end.y = Wcy - deltaY * Wy;
    }
    vec2 tmpEnd = end;
    bool visible = hasStart && hasEnd && clip(start, end, minX, minY, maxX, maxY);
    if (visible) { // если отрезок видим
          // после отсечения, start и end - концы видимой части отрезка
      if (deltaY > 1.f) deltaY = 1.f; // нормализуем значение высоты точки
      if (deltaY < 0.f) deltaY = 0.f; // на случай, если отрезок отсекался
      green = 510.f * deltaY; // предварительное вычисление произведения
      if (deltaY < 0.5) { // если точка ниже середины области видимости
        // компонента зеленого уже вычислена
        blue = 255.f - green; // синий дополняет зеленый
        red = 0.f; // красный равен нулю
      }
      else { // если точка не ниже середины
        blue = 0.f; // синий равен нулю
        red = green - 255.f;   // вычисляем красный и зеленый
        green = 510.f - green; // с использованием вычисленного произведения
      }
      pen->Color = Color::FromArgb(red, green, blue); // меняем цвет пера
      g->DrawLine(pen, start.x, start.y, end.x, end.y);  // отрисовка видимых частей
    }
    // конечная точка неотсеченного отрезка становится начальной точкой следующего
    start = tmpEnd;
    hasStart = hasEnd;
  }
}
```

На текущий момент данный проект завершен. Вы вернетесь к нему при выполнении заданий самостоятельной работы.

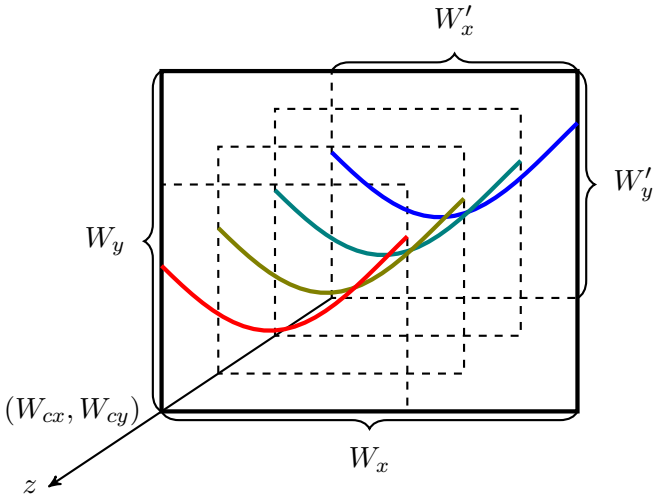
2.5 Построение графика функции двух переменных

В этой половине урока мы модифицируем наш проект, чтобы в нем отрисовывался трехмерный график функции двух переменных $y = f(x, z)$.

Сделайте копию проекта, в которой будем проводить изменения (в результате выполнения пятого задания должны быть загружены два проекта: отдельно для двумерного и трехмерного графика).

Идея отрисовки трехмерного графика достаточно проста. Как и раньше, в мировой системе координат имеем ограниченную область видимости — параллелепипед с ребрами, параллельными осям координат. Кроме зависимости y от x, добавилась зависимость от z. Если мы зафиксируем конкретное значение $z = z^*$, то получим двумерный график $y = f(x, z^*)$. Если будем фиксировать значения z с некоторым постоянным шагом, то для каждого из фиксированных значений сможем построить двумерный график.

Чтобы придать такому набору двумерных графиков вид единого трехмерного, изобразим их со смещением относительно друг друга по вертикали и горизонтали. То есть, для каждого фиксированного z^* будем рисовать двумерный график функции $y = f(x, z^*)$ в своем рабочем прямоугольнике с шириной W'_x, высотой W'_y и координатами левого нижнего угла $(W_{cx}(z^*), W_{cy}(z^*))$.



Будем считать, что координата z является индикатором удаленности точки от наблюдателя: чем меньше координата z, тем дальше точка. Тогда отрисовывая двумерные графики функции $y = f(x, z^*)$ в цикле по z^* от наименьшего значения (V_{cz}) до наибольшего $(V_{cz} + V_z)$ мы получаем подобие трехмерного образа, в котором близлежащие фрагменты графика могут затенять собой более удаленные.

Давайте реализуем эту идею.

2.5.1 Изменение размерности исходных данных

Предполагаем, что нам дана функция двух аргументов. Начнем изменения с функций `f` и `f_exists`. Для примера заменим их следующими функциями.

```
private: float f(float x, float z) {
  return x * sin(sqrtf(x * x + z * z));
}
private: bool f_exists(float x, float z, float delta) {
  return true;
}
```

В мировой системе координат видимая область у нас теперь ограничена параллелепипедом. Его параметрами будут так же являться угловая точка (V_{cx}, V_{cy}, V_{cz}) (точка

параллелепипеда с наименьшими координатами) и вектор, задающий размеры параллелепипеда (V_x, V_y, V_z). В блоке описания глобальных переменных заменим тип соответствующих переменных с `vec2` на `vec3`.

```
vec3 Vc; // координаты дальнего левого нижнего угла
vec3 V; // размеры параллелепипеда в пространстве графика
vec3 Vc_work, V_work; // рабочие параметры параллелепипеда
```

Матричные преобразования теперь будут применяться к трехмерным точкам в однородных координатах. Поэтому матрицы должны быть четырехмерными.

```
mat4 T; // матрица, в которой накапливаются все преобразования
mat4 initT; // матрица начального преобразования
```

Исправим обработчик события *Load*. Изменим размерность начального значения матрицы `initT`.

```
initT = mat4(1.f);
```

Пусть наш параллелепипед будет задан с центром в начале координат и со сторонами длиной 4. Изменим размерность значений `Vc` и `V`.

```
Vc = vec3(-2.f, -2.f, -2.f);
V = vec3(4.f, 4.f, 4.f);
```

Теперь изменим размерность значений в процедуре `worldRectCalc`.

При переводе точки `Vc` в однородные координаты будет получаться четырехмерный вектор.

```
Vc_work = normalize(T * vec4(Vc, 1.f));
```

При удалении последнего столбца и последней строки из матрицы `T` будет получаться матрица размерности 3.

```
V_work = mat3(T) * V;
```

Прежде чем вносить изменения в обработчик события *KeyDown* добавим в проект функции, выдающие матрицы соответствующих трехмерных преобразований. Для этого откроем файл `Transform.h`. Добавим здесь функции `translate` и `scale` от трех параметров, выдающие соответственно матрицы трехмерного преобразования переноса и масштабирования (по аналогии с имеющимися функциями `translate` и `scale`).

```
mat4 translate(float Tx, float Ty, float Tz) {
  mat4 *res = new mat4(1.f); // создали единичную матрицу
  (*res)[0][3] = Tx; // поменяли
  (*res)[1][3] = Ty; // значения в последнем столбце
  (*res)[2][3] = Tz; //
  return *res;
}

mat4 scale(float Sx, float Sy, float Sz) {
  mat4 *res = new mat4(1.f); // создали единичную матрицу
  (*res)[0][0] = Sx; // поменяли
  (*res)[1][1] = Sy; // значения на главной диагонали
  (*res)[2][2] = Sz; //
```

```
  return *res;
}
```

Теперь вернемся к коду файла `MyForm.h` к изменению обработчика события *KeyDown*.

В начале процедуры добавим описание и вычисление переменной для третьей координаты центра параллелепипеда.

```
float centerX = Vc_work.x + V_work.x / 2; // координаты центра параллелепипеда
float centerY = Vc_work.y + V_work.y / 2; // в мировой системе координат
float centerZ = Vc_work.z + V_work.z / 2;
```

В операторе `switch` в реакции на нажатие клавиши **A** необходимо добавить третий аргумент к вызову процедуры `translate` : так как перенос остается сдвигом только по оси Ox, третий параметр должен равняться нулю.

```
case Keys::A:
  T = translate(-V_work.x / Wx, 0.f, 0.f) * T; // сдвиг графика вправо на один
  ↪ пиксел
  break;
```

В реакции на нажатие **Z** вызовы `translate` и `scale` тоже изменим на вызовы соответствующих функций от трех аргументов.

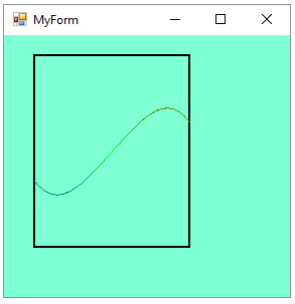
```
case Keys::Z:
  T = translate(-centerX, -centerY, -centerZ) * T; // перенос начала координат в центр
  T = scale(1.1, 1.1, 1.1) * T; // масштабирование относительно начала координат
  T = translate(centerX, centerY, centerZ) * T; // возврат позиции начала координат
  break;
```

Чтобы сделать проект работоспособным внесем изменения в обработчик события *Paint*. Здесь нам понадобится координата z, которую следует передавать функциям `f` и `f_exists` . В качестве такой величины временно возьмем третью координату заданного угла параллелепипеда. Перед описанием переменной `start` инициализируем переменную `z` .

```
float z = Vc_work.z; // координата z наиболее удаленных точек графика
```

Теперь подставим это значение вторым аргументом в каждом из двух вызовов функций `f` и `f_exists` : при вычислении переменных `hasStart` , `hasEnd` нужно заменить `f_exists(x, deltaX)` на `f_exists(x, z, deltaX)` ; при вычислении переменной `y` для начальной и конечной точки в мировых координатах нужно заменить `f(x, z)` на `f_exists(x, z)` .

Теперь проект снова можно запустить: будет выведен двумерный график для фиксированного z.



2.5.2 Добавление третьего измерения в график

Реализуем идею, сформулированную в начале этого раздела.

После описания параметров `Wx`, `Wy` добавим переменные, в которых будем сохранять значения W'_x и W'_y.

```
float Wx_work, Wy_work; // ширина и высота области вывода одной линии графика
```

Значения этих параметров будем вычислять как долю от W_x и W_y, заданную следующими двумя параметрами, которые опишем здесь же, присвоив им значения по умолчанию `0.6`.

```
float Wx_part = 0.6, Wy_part = 0.6; // доля Wx_work, Wy_work от Wx, Wy соответственно
```

Кроме этого, опишем еще две переменные для координат левого нижнего угла самого дальнего (самого верхнего правого) прямоугольника с графиком (см. рисунок в начале раздела 2.5).

```
float Wcx_work, Wcy_work; // координаты левого нижнего угла самого дальнего прямоугольника
```

Будем вычерчивать двумерные графики в каждом рабочем прямоугольнике с координатой y_c нижнего левого угла, пробегающей в цикле от `Wcy_work` до `Wcy` с шагом 1. Переход от одной величины к другой смоделирует для нас движение по размерности z. Опишем дополнительный параметр для этой разности, который назовем `Wz_work`.

```
float Wz_work; // количество рабочих прямоугольников
```

В общем виде блок описания параметров формы пример вид

```
1  private: float left = 30, right = 100, top = 20, bottom = 50; // расстояния до границ окна
2     float minX = left, maxX; // диапазон изменения координат x
3     float minY = top, maxY; // диапазон изменения координат y
4     float Wcx = left, Wcy; // координаты левого нижнего угла прямоугольника
5     float Wx, Wy; // ширина и высота прямоугольника
6     float Wx_work, Wy_work; // ширина и высота области вывода одной линии графика
7     float Wx_part = 0.6, Wy_part = 0.6; // доля Wx_work, Wy_work от Wx, Wy соответственно
8     float Wcx_work, Wcy_work; // координаты левого нижнего угла самого дальнего прямоугольника
9     float Wz_work; // количество рабочих прямоугольников
```

Теперь в конце процедуры `rectCalc` произведем вычисление описанных параметров.

```
Wx_work = Wx_part * Wx; // вычисление ширины и высоты
Wy_work = Wy_part * Wy; // рабочего прямоугольника
Wcx_work = maxX - Wx_work; // вычисление координат нижнего левого
Wcy_work = minY + Wy_work; // угла самого дальнего рабочего прямоугольника
Wz_work =  Wcy - Wcy_work; // количество рабочих прямоугольников
```

Процедура `rectCalc` примет вид

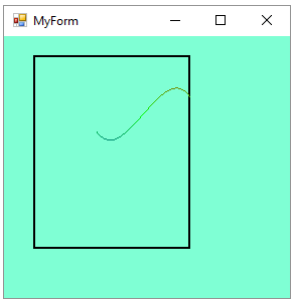
```
1  private: System::Void rectCalc() {
2    maxX = ClientRectangle.Width - right; // диапазон изменения координат x
3    maxY = ClientRectangle.Height - bottom; // диапазон изменения координат y
4    Wcy = maxY; // координаты левого нижнего угла прямоугольника
5    Wx = maxX - left; // ширина прямоугольника
6    Wy = maxY - top; // ширина и высота прямоугольника
7    Wx_work = Wx_part * Wx; // вычисление ширины и высоты
8    Wy_work = Wy_part * Wy; // рабочего прямоугольника
9    Wcx_work = maxX - Wx_work; // вычисление координат нижнего левого
10   Wcy_work = minY + Wy_work; // угла самого дальнего рабочего прямоугольника
11   Wz_work =  Wcy - Wcy_work; // количество рабочих прямоугольников
12 }
```

Обратимся к процедуре `MyForm_Paint`.

В строках кода, относящихся к отрисовке графика, необходимо заменить значения `Wx` и `Wy` на соответствующие значения `Wx_work` и `Wy_work`, относящиеся к рабочему прямоугольнику (конкретнее — в трех местах кода: при вычислении значений `deltaX`, `start.y` и `end.y`).

Также нужно заменить `Wcx` и `Wcy` на `Wcx_work` и `Wcy_work` (при вычислении значений `start.x`, `start.y` и `end.y`).

Обновленный график не будет занимать всю область видимости, а будет ограничен невидимым прямоугольником размеры которого составляют 60% от исходных размеров области видимости.



Добавим вычерчивание графика для других координат z.

После описания переменной `deltaX` вычислим шаг в мировой системе координат по оси z.

```
float deltaZ = V_work.z / Wz_work; // шаг по z в мировых координатах
```

Кроме этого нужно вычислить величину сдвига очередного прямоугольника относительно предыдущего. Величина сдвига по y всегда равна единице (одному пикселу). Величину сдвига по x вычислим и сохраним в дополнительной переменной.

```
float deltaWcx = (Wcx_work - Wcx) / Wz_work; // шаг прямоугольников по x в координатах экрана
```

Теперь организуем внешний цикл относительно уже имеющихся в наличии циклов. Как мы уже говорили, цикл будет перечислять прямоугольники (координаты левого нижнего угла). Можем описать его циклом с тремя параметрами: `z`, `Wcx_w`, `Wcy_w`, где `Wcx_w` и `Wcy_w` — координаты левого нижнего угла очередного прямоугольника.

```
float z = Vc_work.z; // координата z соответствующая дальнему прямоугольнику
// координаты левого нижнего угла рабочего прямоугольника (инициализация)
float Wcx_w = Wcx_work, Wcy_w = Wcy_work;
while (Wcy_w <= Wcy) { // пока не перебрали все прямоугольники


  Wcy_w += 1.f; // переходим к следующему прямоугольнику, он будет ниже на один пиксел
  Wcx_w -= deltaWcx; // и левее на некоторое значение
  z += deltaZ; // вычисляем соответствующее значение z для очередного прямоугольника
}
```

Первая строка приведенного фрагмента кода у нас уже есть. Добавим после нее две другие строки. Окончание цикла будет непосредственно перед закрытием процедуры (перед последней закрывающей фигурной скобкой). Соответственно, внутри цикла нужно заменить

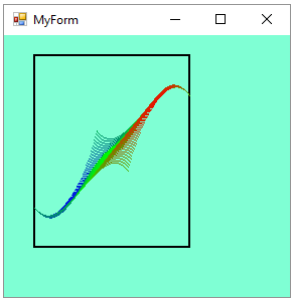
вхождения `Wcx_work` на `Wcx_w` и `Wcy_work` на `Wcy_w` (при вычислении `start.x`, `start.y` и `end.y`). Кроме того, внутри цикла, перед строкой

```
while (start.x < maxX) {
```

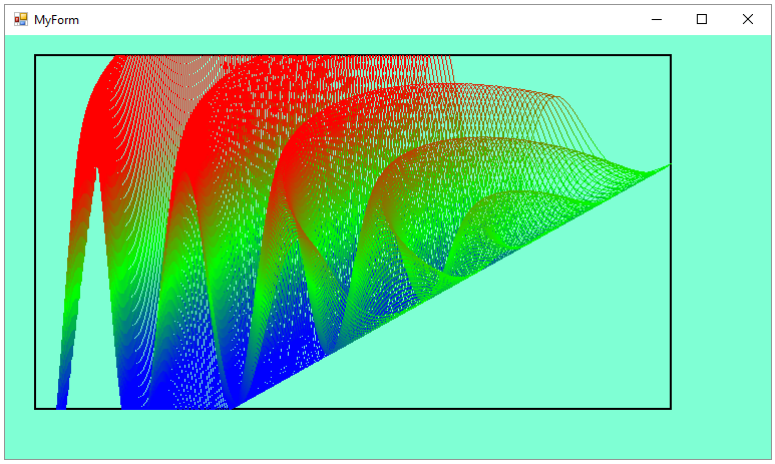
следует переопределить значение `maxX`, т. к. в противном случае каждая линия графика будет рисоваться не до границы рабочего прямоугольника, а до правого края очерченного прямоугольника.

```
float maxX = Wcx_w + Wx_work; // максимальное значение x в рабочем прямоугольнике
```

Если запустить проект, то получим «трехмерное» изображение.



Окно с графиком можно растянуть, масштабировать и сдвинуть в окне (с помощью уже определенных клавиш **A** и **Z**). Пример получившегося изображения приведен на следующем рисунке.



!!! Чем больше Вы растягиваете окно с графиком, тем из большего количества отрезков он состоит, а следовательно, обработка растянутого графика будет проводиться медленнее: горячие клавиши могут срабатывать с некоторой задержкой.

Фрагмент кода обработчика события *Paint*, ответственный за отрисовку графика в общем виде примет следующий вид.

```
1 // ОТРИСОВКА ГРАФИКА
2 Pen^ pen = gcnew Pen(Color::Blue, 1);
3 float deltaX = V_work.x / Wx_work; // шаг по x в мировых координатах
4 float deltaZ = V_work.z / Wz_work; // шаг по z в мировых координатах
5
```

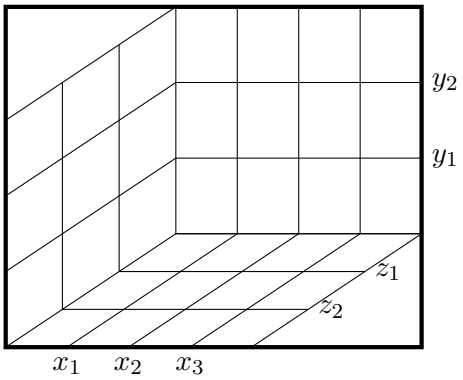
```
float deltaWcx = (Wcx_work - Wcx) / Wz_work; // шаг прямоугольников по x в координатах экрана

bool hasStart;

// цикл по прямоугольникам
float z = Vc_work.z; // координата z соответствующая дальнему прямоугольнику
// координаты левого нижнего угла рабочего прямоугольника (инициализация)
float Wcx_w = Wcx_work, Wcy_w = Wcy_work;
while (Wcy_w <= Wcy) { // пока не перебрали все прямоугольники
  vec2 start, end; // концы отрезка в координатах экрана
  float x, y; // переменные для координат точки в мировой СК
  start.x = Wcx_w; // для начальной точки первого отрезка устанавливаем координату x
  x = Vc_work.x; // координата x начальной точки первого отрезка в мировых координатах
  hasStart = f_exists(x, z, deltaX);
  if (hasStart) {
    y = f(x, z); // координата y начальной точки в мировых координатах
    // вычисляем соответствующее значение в координатах экрана
    start.y = Wcy_w - (y - Vc_work.y) / V_work.y * Wy_work;
  }
  float maxX = Wcx_w + Wx_work; // максимальное значение x в рабочем прямоугольнике
  while (start.x < maxX) {
    vec2 end; // точка конца отрезка в координатах экрана
    bool hasEnd;
    float deltaY; // высота точки в прямоугольнике (доля общей высоты)
    float red, green, blue; // компоненты цвета отрезка
    end.x = start.x + 1.f; // координата x отличается на единицу
    x += deltaX; // координата x конечной точки отрезка в мировых координатах
    hasEnd = f_exists(x, z, deltaX);
    if (hasEnd) {
      y = f(x, z); // координата y начальной точки в мировых координатах
      // вычисляем соответствующее значение в координатах экрана
      deltaY = (y - Vc_work.y) / V_work.y;
      end.y = Wcy_w - deltaY * Wy_work;
    }
    vec2 tmpEnd = end;
    bool visible = hasStart && hasEnd && clip(start, end, minX, minY, maxX, maxY);
    if (visible) { // если отрезок видим
                   // после отсечения, start и end - концы видимой части отрезка
      // вычичление цвета отрезка
      if (deltaY > 1.f) deltaY = 1.f; // нормализуем значение высоты точки
      if (deltaY < 0.f) deltaY = 0.f; // на случай, если отрезок отсекался
      green = 510.f * deltaY; // предварительное вычисление произведения
      if (deltaY < 0.5) { // если точка ниже середины области видимости
        // компонента зеленого уже вычислена
        blue = 255.f - green; // синий дополняет зеленый
        red = 0.f; // красный равен нулю
      }
      else { // если точка не ниже середины
        blue = 0.f; // синий равен нулю
        red = green - 255.f;   // вычисляем красный и зеленый
        green = 510.f - green; // с использованием вычисленного произведения
      }
      pen->Color = Color::FromArgb(red, green, blue); // меняем цвет пера
      // отрисовка отрезка
      g->DrawLine(pen, start.x, start.y, end.x, end.y);  // отрисовка видимых частей
    }
    // конечная точка неотсеченного отрезка становится начальной точкой следующего
    start = tmpEnd;
    hasStart = hasEnd;
  }
  Wcy_w += 1.f; // переходим к следующему прямоугольнику, он будет ниже на один пиксел
  Wcx_w -= deltaWcx; // и левее на некоторое значение
  z += deltaZ; // вычисляем соответствующее значение z для очередного прямоугольника
}
```

2.5.3 Добавление координатной сетки

Добавим к графику трехмерную координатную сетку, отражающую значения, соответствующие точкам графика.

Пусть наш график будет рисоваться на фоне координатной сетки (т. е. сначала будет начерчена сетка, а затем график).

Координатная сетка будет состоять из трех расчерченных плоскостей, параллельных координатным плоскостям.



Будем считать, что у нас задается количество делений по каждой из осей. Так на приведенном рисунке, пространство по оси Ox делится на 4 части, а по осям Oy и Oz — на 3 части.

Опишем параметры формы, отвечающие за количество делений координатной сетки, среди других параметров. Пусть количество отрезков, на которое делится каждая сторона прямоугольника равно 5.

После описания переменной `Wz_work` добавим

```
int numXsect = 5, numYsect = 5, numZsect = 5; // количество секций координатной сетки по осям
```

Обратимся снова к обработчику события *Paint*.

Сначала опишем перо для вычерчивания линий сетки и кисть для вывода подписей к линиям.

```
Pen^ gridPen = gcnew Pen(Color::Black, 1);
SolidBrush^ drawBrush = gcnew SolidBrush(Color::Black);
System::Drawing::Font^ drawFont = gcnew System::Drawing::Font("Arial", 8);
```

Давайте добавим цикл, вычерчивающий координатную сетку по оси Ox.

Определим расстояние между линиями сетки.

```
float gridStep_x = Wx_work / numXsect; // расстояние между линиями сетки по x
```

Вычислим, расстояние между линиями сетки в мировых координатах.

```
float grid_dX = V_work.x / numXsect; // расстояние между линиями сетки по x в мировых координатах
```

Теперь организуем цикл, в котором будем вычерчивать `numXsect` $+1$ линию сетки.

```
float tick_x = Vc_work.x; // значение соответствующее первой линии сетки
for (int i = 0; i <= numXsect; i++) { // цикл по количеству линий

  tick_x += grid_dX; // вычисляем значение, соответствующее следующей линии
}
```

Каждая линия, соответствующая координате x будет состоять из двух частей: нижней диагональной (линия в плоскости, параллельной xOz) и вертикальной (линия в плоскости, параллельной xOy). Диагональные линии начинаются на нижней стороне прямоугольника области видимости и заканчиваются на нижней стороне наиболее удаленного рабочего прямоугольника. Вертикальные линии соединяют нижнюю и верхнюю стороны рабочего прямоугольника.

```
float tmpXCoord_d = Wcx + i * gridStep_x; // нижняя координата x i-й диагональной линии
float tmpXCoord_v = Wcx_work + i * gridStep_x; // координата x i-й вертикальной линии
// i-я диагональная линия
g->DrawLine(gridPen,tmpXCoord_d, Wcy, tmpXCoord_v, Wcy_work);
// i-я вертикальная линия
g->DrawLine(gridPen, tmpXCoord_v, Wcy_work, tmpXCoord_v, minY);
```

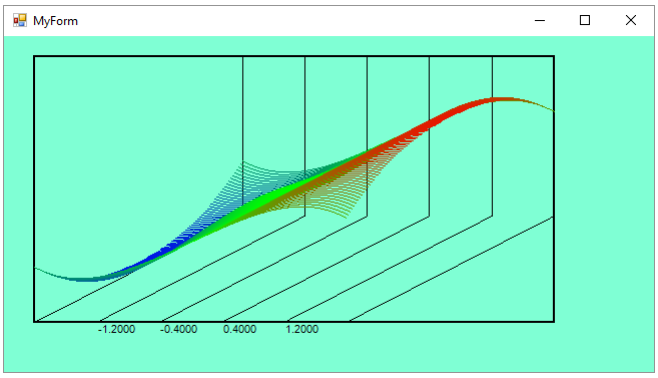
Для линий сетки, отличных от крайних, выведем соответствующее значение x в единицах измерения мировых координат. Выводим вещественное значение, превратив его в строку, оставив 4 знака после запятой.

```
if (i > 0 && i < numXsect) // если линия не крайняя
  // выводим текст в нижней точке диагональной линии
  g->DrawString(tick_x.ToString("F4"), drawFont, drawBrush, tmpXCoord_d, Wcy);
```

Получили в общем виде следующий фрагмент, вычерчивающий координатную сетку по оси Ox.

```
1  // координатная сетка по x
2  float gridStep_x = Wx_work / numXsect; // расстояние между линиями сетки по x
3  float grid_dX = V_work.x / numXsect; // расстояние между линиями сетки по x в мировых координатах
4  float tick_x = Vc_work.x; // значение, соответствующее первой линии сетки
5  for (int i = 0; i <= numXsect; i++) { // цикл по количеству линий
6    float tmpXCoord_d = Wcx + i * gridStep_x; // нижняя координата x i-й диагональной линии
7    float tmpXCoord_v = Wcx_work + i * gridStep_x; // координата x i-й вертикальной линии
8    // i-я диагональная линия
9    g->DrawLine(gridPen, tmpXCoord_d, Wcy, tmpXCoord_v, Wcy_work);
10   // i-я вертикальная линия
11   g->DrawLine(gridPen, tmpXCoord_v, Wcy_work, tmpXCoord_v, minY);
12   if (i > 0 && i < numXsect) // если линия не крайняя
13     // выводим текст в нижней точке диагональной линии
14     g->DrawString(tick_x.ToString("F4"), drawFont, drawBrush, tmpXCoord_d, Wcy);
15   tick_x += grid_dX; // вычисляем значение, соответствующее следующей линии
16 }
```

Если запустить проект и растянуть окно, будет подобная картина.



Давайте, по аналогии, добавим координатную сетку по оси Oz.

Сначала определяем шаг сетки по z в экранных и мировых координатах. Концы всех линий координатной сетки по z лежат на диагональных отрезках. Для вычисления координат очередной точки вычислим два шага: по вертикали и горизонтали.

```
gridStep_x = (Wx - Wx_work) / numZsect; // расстояние между вертикальными линиями сетки по горизонтали
float gridStep_y = Wz_work / numZsect; // расстояние между горизонтальными линиями сетки по вертикали
float grid_dZ = V_work.z / numZsect; // расстояние между линиями сетки по $z$  в мировых координатах
```

Далее, организуем цикл

```
float tick_z = Vc_work.z; // значение, соответствующее первой линии сетки
for (int i = 0; i <= numZsect; i++) { // цикл по количеству линий

  tick_z += grid_dZ; // вычисляем значение, соответствующее следующей линии
}
```

Каждая линия сетки состоит из двух частей: вертикальной и горизонтальной.

```
float tmpXCoord_v = Wcx_work - i * gridStep_x; // координата x вертикальных линий
float tmpYCoord_g = Wcy_work + i * gridStep_y; // координата y горизонтальных линий
float tmpXCoord_g = tmpXCoord_v + Wx_work; // вторая координата x горизонтальных линий
// i-я вертикальная линия
g->DrawLine(gridPen, tmpXCoord_v, tmpYCoord_g, tmpXCoord_v, tmpYCoord_g - Wy_work);
// i-я горизонтальная линия
g->DrawLine(gridPen, tmpXCoord_v, tmpYCoord_g, tmpXCoord_g, tmpYCoord_g);
```

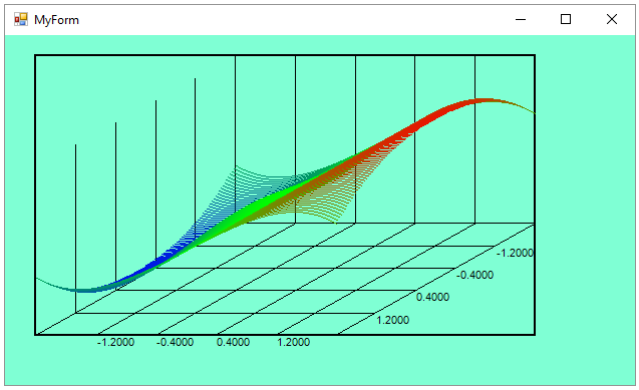
Выведем значения, соответствующие линиям координатной сетки.

```
if (i > 0 && i < numZsect) // если линия не крайняя
  // выводим текст в правой точке горизонтальной линии
  g->DrawString(tick_z.ToString("F4"), drawFont, drawBrush, tmpXCoord_g, tmpYCoord_g);
```

Общий вид блока отрисовки координатной сетки по z следующий.

```
1  // координатная сетка по z
2  gridStep_x = (Wx - Wx_work) / numZsect; // расстояние между вертикальными линиями сетки по горизонтали
3  float gridStep_y = Wz_work / numZsect; // расстояние между горизонтальными линиями сетки по вертикали
4  float grid_dZ = V_work.z / numZsect; // расстояние между линиями сетки по $z$  в мировых координатах
5  float tick_z = Vc_work.z; // значение, соответствующее первой линии сетки
6  for (int i = 0; i <= numZsect; i++) { // цикл по количеству линий
7    float tmpXCoord_v = Wcx_work - i * gridStep_x; // координата x вертикальных линий
8    float tmpYCoord_g = Wcy_work + i * gridStep_y; // координата y горизонтальных линий
9    float tmpXCoord_g = tmpXCoord_v + Wx_work; // вторая координата x горизонтальных линий
10   // i-я вертикальная линия
11   g->DrawLine(gridPen, tmpXCoord_v, tmpYCoord_g, tmpXCoord_v, tmpYCoord_g - Wy_work);
12   // i-я горизонтальная линия
13   g->DrawLine(gridPen, tmpXCoord_v, tmpYCoord_g, tmpXCoord_g, tmpYCoord_g);
14   if (i > 0 && i < numZsect) // если линия не крайняя
15     // выводим текст в правой точке горизонтальной линии
16     g->DrawString(tick_z.ToString("F4"), drawFont, drawBrush, tmpXCoord_g, tmpYCoord_g);
17   tick_z += grid_dZ; // вычисляем значение, соответствующее следующей линии
18 }
```

Запуск проекта осуществится в подобной форме:



Получим следующий общий вид обработчика события *Paint*.

```
private: System::Void MyForm_Paint(System::Object^  sender, System::Windows::Forms::PaintEventArgs^  e) {
  Graphics^ g = e->Graphics;
  g->Clear(Color::Aquamarine);

  Pen^ rectPen = gcnew Pen(Color::Black, 2);
  g->DrawRectangle(rectPen, left, top, Wx, Wy);

  // ВЫЧЕРЧИВАНИЕ КООРДИНАТНОЙ СЕТКИ
  Pen^ gridPen = gcnew Pen(Color::Black, 1);
  SolidBrush^ drawBrush = gcnew SolidBrush(Color::Black);
  System::Drawing::Font^ drawFont = gcnew System::Drawing::Font("Arial", 8);

  // координатная сетка по x
  float gridStep_x = Wx_work / numXsect; // расстояние между линиями сетки по x
  float grid_dX = V_work.x / numXsect; // расстояние между линиями сетки по x в мировых координатах
  float tick_x = Vc_work.x; // значение, соответствующее первой линии сетки
  for (int i = 0; i <= numXsect; i++) { // цикл по количеству линий
    float tmpXCoord_d = Wcx + i * gridStep_x; // нижняя координата x i-й диагональной линии
    float tmpXCoord_v = Wcx_work + i * gridStep_x; // координата x i-й вертикальной линии
    // i-я диагональная линия
    g->DrawLine(gridPen, tmpXCoord_d, Wcy, tmpXCoord_v, Wcy_work);
    // i-я вертикальная линия
    g->DrawLine(gridPen, tmpXCoord_v, Wcy_work, tmpXCoord_v, minY);
    if (i > 0 && i < numXsect) // если линия не крайняя
      // выводим текст в нижней точке диагональной линии
      g->DrawString(tick_x.ToString("F4"), drawFont, drawBrush, tmpXCoord_d, Wcy);
    tick_x += grid_dX; // вычисляем значение, соответствующее следующей линии
  }

  // координатная сетка по z
  gridStep_x = (Wx - Wx_work) / numZsect; // расстояние между вертикальными линиями сетки по горизонтали
  float gridStep_y = Wz_work / numZsect; // расстояние между горизонтальными линиями сетки по вертикали
  float grid_dZ = V_work.z / numZsect; // расстояние между линиями сетки по $z$  в мировых координатах
  float tick_z = Vc_work.z; // значение, соответствующее первой линии сетки
  for (int i = 0; i <= numZsect; i++) { // цикл по количеству линий
    float tmpXCoord_v = Wcx_work - i * gridStep_x; // координата x вертикальных линий
    float tmpYCoord_g = Wcy_work + i * gridStep_y; // координата y горизонтальных линий
    float tmpXCoord_g = tmpXCoord_v + Wx_work; // вторая координата x горизонтальных линий
    // i-я вертикальная линия
    g->DrawLine(gridPen, tmpXCoord_v, tmpYCoord_g, tmpXCoord_v, tmpYCoord_g - Wy_work);
    // i-я горизонтальная линия
    g->DrawLine(gridPen, tmpXCoord_v, tmpYCoord_g, tmpXCoord_g, tmpYCoord_g);
    if (i > 0 && i < numZsect) // если линия не крайняя
      // выводим текст в правой точке горизонтальной линии
      g->DrawString(tick_z.ToString("F4"), drawFont, drawBrush, tmpXCoord_g, tmpYCoord_g);
    tick_z += grid_dZ; // вычисляем значение, соответствующее следующей линии
  }

  // ОТРИСОВКА ГРАФИКА
  Pen^ pen = gcnew Pen(Color::Blue, 1);
  float deltaX = V_work.x / Wx_work; // шаг по x в мировых координатах
  float deltaZ = V_work.z / Wz_work; // шаг по z в мировых координатах

  float deltaWcx = (Wcx_work - Wcx) / Wz_work; // шаг прямоугольников по x в координатах экрана

```

```
  bool hasStart;

  // цикл по рабочим прямоугольникам
  float z = Vc_work.z; // координата z соответствующая дальнему прямоугольнику
  // координаты левого нижнего угла рабочего прямоугольника (инициализация)
  float Wcx_w = Wcx_work, Wcy_w = Wcy_work;
  while (Wcy_w <= Wcy) { // пока не перебрали все прямоугольники
    vec2 start, end; // концы отрезка в координатах экрана
    float x, y; // переменные для координат точки в мировой СК
    start.x = Wcx_w; // для начальной точки первого отрезка устанавливаем координату x
    x = Vc_work.x; // координата x начальной точки первого отрезка в мировых координатах
    hasStart = f_exists(x, z, deltaX);
    if (hasStart) {
      y = f(x, z); // координата y начальной точки в мировых координатах
      // вычисляем соответствующее значение в координатах экрана
      start.y = Wcy_w - (y - Vc_work.y) / V_work.y * Wy_work;
    }
    float maxX = Wcx_w + Wx_work; // максимальное значение x в рабочем прямоугольнике
    while (start.x < maxX) {
      vec2 end;// точка конца отрезка в координатах экрана
      bool hasEnd;
      float deltaY; // высота точки в прямоугольнике (доля общей высоты)
      float red, green, blue; // компоненты цвета отрезка
      end.x = start.x + 1.f; // координата x отличается на единицу
      x += deltaX; // координата x конечной точки отрезка в мировых координатах
      hasEnd = f_exists(x, z, deltaX);
      if (hasEnd) {
        y = f(x, z); // координата y начальной точки в мировых координатах
        // вычисляем соответствующее значение в координатах экрана
        deltaY = (y - Vc_work.y) / V_work.y;
        end.y = Wcy_w - deltaY * Wy_work;
      }
      vec2 tmpEnd = end;
      bool visible = hasStart && hasEnd && clip(start, end, minX, minY, maxX, maxY);
      if (visible) { // если отрезок видим
             // после отсечения, start и end - концы видимой части отрезка
        // ВЫЧИЧЛЕНИЕ ЦВЕТА ОТРЕЗКА
        if (deltaY > 1.f) deltaY = 1.f; // нормализуем значение высоты точки
        if (deltaY < 0.f) deltaY = 0.f; // на случай, если отрезок отсекался
        green = 510.f * deltaY; // предварительное вычисление произведения
        if (deltaY < 0.5) { // если точка ниже середины области видимости
          // компонента зеленого уже вычислена
          blue = 255.f - green; // синий дополняет зеленый
          red = 0.f; // красный равен нулю
        }
        else { // если точка не ниже середины
          blue = 0.f; // синий равен нулю
          red = green - 255.f;    // вычисляем красный и зеленый
          green = 510.f - green; // с использованием вычисленного произведения
        }
        pen->Color = Color::FromArgb(red, green, blue); // меняем цвет пера
        // ОТРИСОВКА ОТРЕЗКА
        g->DrawLine(pen, start.x, start.y, end.x, end.y);  // отрисовка видимых частей
      }
      // конечная точка неотсеченного отрезка становится начальной точкой следующего
      start = tmpEnd;
      hasStart = hasEnd;
    }
    Wcy_w += 1.f; // переходим к следующему прямоугольнику, он будет ниже на один пиксел
    Wcx_w -= deltaWcx; // и левее на некоторое значение
    z += deltaZ; // вычисляем соответствующее значение z для очередного прямоугольника
  }
}
```

2.6 Задание для самостоятельной работы

Задание 5

1. Создайте два проекта, включающие в себя все, что было описано выше в качестве примера: отдельно для построения двумерного и трехмерного графика. Каждый проект должен назваться Вашей фамилией, записанной латинскими буквами.
2. В проекте для построения трехмерного графика:
   (a) Добавьте координатную сетку по оси Oy так, как показано на рисунке в начале раздела 2.5.3. Метки координатной сетки должны выводиться справа от правых концов горизонтальных отрезков.
   (b) Добавьте реакции на нажатия клавиш:
   - **D**, **W**, **S**, **R**, **F** — смещение графика в окне влево, назад (на наблюдателя), вперед (от наблюдателя), вниз и вверх на 1 пиксел;
   - **X** — равномерное уменьшение параллелепипеда видимости (в мировых координатах) в 1.1 раз;
   - **Q**/**E**, **C**/**V** — увеличение/уменьшение размера рабочего прямоугольника по оси Ox и Oy соответственно в 1.1 раза. Максимальный размер рабочего прямоугольника по каждой из осей не должен превышать 0.9 от размера прямоугольника области видимости. Минимальный размер рабочего прямоугольника по каждой из осей не должен быть меньше 0.2 от размера прямоугольника области видимости;
   - **T**/**G**, **Y**/**H**, **U**/**J** — увеличение/уменьшение параллелепипеда видимости (в мировых координатах) относительно его центра в 1.1 раза по оси Ox, Oy, Oz, соответственно;
   - **1**/**2**, **3**/**4**, **5**/**6** — увеличение/уменьшение количества секций координатной сетки по оси Ox, Oy, Oz, соответственно. Минимальное количество — 2.

   (c) каждая линия графика функции сейчас вычисляется в рабочем прямоугольнике, но отсекается не относительно него, а относительно очерченного прямоугольника на форме. Измените проект так, чтобы отсечение производилось относительно рабочих прямоугольников.
3. В проекте для построения двумерного графика:
   (a) Добавьте ПОВЕРХ ГРАФИКА координатную сетку, состоящую из вертикальных и горизонтальных линий. Метки координатной сетки должны выводиться снизу от вертикальных линий и справа от горизонтальных.
   (b) Добавьте реакции на нажатия клавиш:
   - **D**, **W**, **S** — смещение графика в окне влево, вниз и вверх на 1 пиксел;
   - **X** — равномерное уменьшение прямоугольника в мировых координатах в 1.1 раза;
   - **T**/**G**, **Y**/**H** — увеличение/уменьшение прямоугольника в мировых координатах в 1.1 раза по оси Ox, Oy, соответственно;
   - **1**/**2**, **3**/**4** — увеличение/уменьшение количества секций координатной сетки по оси Ox, Oy, соответственно. Минимальное количество — 2.
4. В качестве результата выполнения задания должны быть загружены два файла: архивы двух получившихся проектов. К стандартным именам архивов в конце нужно добавить `2D` и `3D`, соответственно для двумерного и трехмерного случая.

■

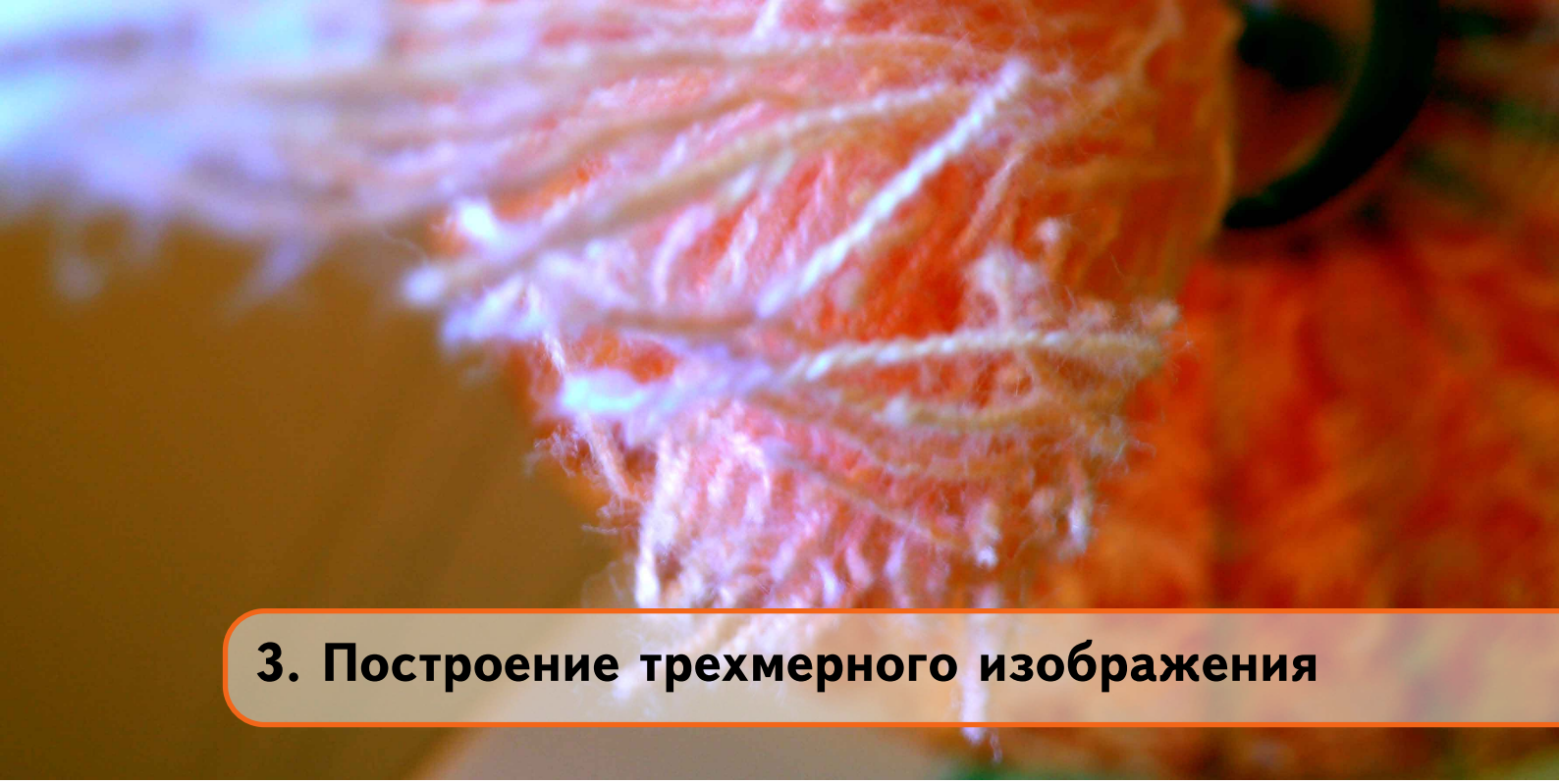
2.7 Контрольные вопросы

Подготовьте ответы на следующие вопросы.

1. Укажите в программном коде `Matrix.h` функцию для вычисления скалярного произведения двумерных/четырехмерных векторов. Объясните ее работу (добавив объяснение задействованных функций и операций).
2. Укажите в программном коде `Matrix.h` метод для вычисления произведения двумерных/четырехмерных матриц. Объясните ее работу (добавив объяснение задействованных функций и операций).
3. Объясните смысл значения переменной `V`. Что означает преобразование заданное выражением `V_work = mat2(T) * V;` в двумерном случае или выражением `V_work = mat3(T) * V`; в трехмерном случае?
4. Мы выводим график как набор отрезков, длина проекции которых на ось Ox в координатах экрана равна 1. Почему бы тогда нам не изобразить график просто набором точек (если считаем, что ширина точки на экране равна 1)? Как изменится вид графика, если его выводить набором точек?
5. Для чего нам нужно функция `f_exists`? В каких случаях эта функция должна выдавать `true` или `false`?
6. Для чего функции `f_exists` параметр `delta`? Что и почему должно передаваться этому параметру в качестве значения?
7. Как в проектах реализуется сдвиг и масштабирование графика на форме: за счет чего это происходит?
8. Как так происходит, что сдвиг графика в окне осуществляется на 1 пиксел, независимо от масштаба графика? Ведь мы осуществляем сдвиг в мировой системе координат, а не в системе координат экрана.
9. Что будет происходить с графиком на форме, если в проект отрисовки двумерного графика добавить преобразование поворота относительно центра прямоугольника (по подобию с преобразованием масштабирования)?
10. Что будет происходить с графиком на форме, если в проект отрисовки двумерного графика добавить преобразование зеркального отражения относительно какой либо оси, проходящей через центр прямоугольника (по подобию с преобразованием масштабирования)?
11. Почему при растяжении окна формы график растягивается вместе с ней? Укажите фрагменты программы, ответственные за это.
12. В каких единицах измерения вычисляется значение переменной `deltaY` в обработчике события *Paint*? Какой смысл этого значения?
13. Почему значение зеленой составляющей цвета отрезка в нижней половине области видимости вычисляется как произведение `510.f * deltaY`?
14. Почему, при вычислении значения синей составляющей цвета отрезка в нижней половине области видимости, мы вычисляем ее как разность `255.f - green`?
15. Укажите в программном коде `Transform.h` функцию возвращающую матрицу двумерного/трехмерного преобразования переноса. Объясните ее работу.
16. Укажите в программном коде `Transform.h` функцию возвращающую матрицу двумерного/трехмерного преобразования масштабирования. Объясните ее работу.
17. Укажите фрагмент кода построения трехмерного графика, в котором вычисляются размеры рабочего прямоугольника.
18. Для чего нужна переменная `tmpEnd`? Почему бы там, где она используется, не

использовать вместо нее просто значение `end` ?

19. Укажите фрагмент кода `MyForm.h` , в котором происходит отсечение отрезка относительно рабочего прямоугольника.
20. Что произойдет с проектом отрисовки трехмерного графика, если в нем снять ограничения значений переменных `Wx_part` и `Wy_part` ?



3. Построение трехмерного изображения

В этом уроке мы реализуем построение проволочного трехмерного образа. Будем проводить построение одной модели, составленной из трехмерных ломаных, с отсечением относительно прямоугольника в окне, как мы это делали в 4-м уроке. Поэтому, в качестве заготовки для проекта шестого задания возьмем проект от четвертого. Но заменим в нем файлы `Transform.h` и `Matrix.h` на обновленные версии этих файлов из проекта для построения трехмерного графика пятого задания.

3.1 Предмет реализации

В ходе урока мы добавим к прообразу третью размерность и переопределим операции для манипуляций изображением в окне.

Как и в четвертом задании, мы будем собирать сцену для изображения из моделей, заданных набором ломаных линий. Описание сцены в мировой системе координат так же будем считывать из файла. Но теперь точки и все преобразования будут трехмерными.

Структура входного файла с описанием сцены у нас сохранится, за исключением команды `frame` . Вместо этой команды мы введем две новые: команду `camera` , указывающую положение и ракурс камеры (системы координат наблюдателя), и команду `screen` , указывающую размеры и положение окна наблюдения.

Проекцию трехмерной сцены на окно наблюдения будем выводить на форме, совместив окно наблюдения с прямоугольником при помощи операции кадрирования.

Добавим в приложение реакции на нажатия клавиш, позволяющие менять положение и ракурс наблюдателя в трехмерном пространстве, положение окна наблюдения, тип проекции на окно наблюдения.

!!! В задачи этого урока не входят задачи трехмерного отсечения, хотя мы и осуществим переход в пространство отсечения. Поэтому итоговое изображение, построенное на форме в некоторых случаях может показаться Вам некорректным.

3.2 Реализация трехмерных операций

В ходе построения проекта нам потребуются дополнительные матричные операции и матричные преобразования, т. е. необходимо провести некоторые дополнения к файлам `Transform.h` и `Matrix.h`.

3.2.1 Дополнительные матричные операции

В `Matrix.h` в ходе выполнения пятого задания уже определены необходимые структуры данных (трехмерные и четырехмерные матрицы, трехмерные и четырехмерные векторы).

Опишем дополнительные конструкторы к классам `vec2` и `vec3`, получающие координаты векторов от объектов классов `vec3` и `vec4`, соответственно (т. е. из трехмерного вектора получится двумерный и из четырехмерного — трехмерный отбрасыванием последней координаты).

Так как класс `vec2` описан перед классом `vec3`, а `vec3` перед `vec4`, проведем описание в следующем порядке. В начале файла, перед описанием класса `vec2` добавим декларацию нижеприведенных описаний классов `vec3` и `vec4`.

```
class vec3;
class vec4;
```

В классе `vec2` добавим декларацию конструктора, что мы собираемся описать.

```
vec2(vec3 v);
```

Таким же образом в класс `vec3` добавим декларацию нового конструктора.

```
vec3(vec4 v);
```

Теперь ниже описания класса `vec4` добавим описания самих конструкторов, не забывая отметить принадлежность процедур к соответствующим классам.

```
vec2::vec2(vec3 v) : x(v.x), y(v.y) {}
vec3::vec3(vec4 v) : x(v.x), y(v.y), z(v.z) {}
```

Нам нужно доопределить операции сложения и вычитания трехмерных векторов, операцию сложения матриц третьего порядка, операции домножения векторов и матриц на число.

Сначала добавим в описание класса `vec3` реализацию сложения и вычитания векторов (по аналогии с операцией умножения).

```
vec3& operator+=(const vec3& v) {
  x += v.x;
  y += v.y;
  z += v.z;
  return *this;
}
const vec3 operator+(const vec3& v) {
  return vec3(*this) += v; // делаем временную копию текущего объекта,
                           // к которой добавляем данный вектор,
                           // и возвращаем ее как результат
}
vec3& operator-=(const vec3& v) {
  x -= v.x;
  y -= v.y;
```

```
  z -= v.z;
  return *this;
}
const vec3 operator-(const vec3& v) {
  return vec3(*this) -= v; // делаем временную копию текущего объекта,
                           // из которой вычитаем данный вектор,
                           // и возвращаем ее как результат
}
```

Добавим в класс `vec3` операцию умножения вектора на число (пользуясь спецификой инфиксной операции умножения вектора на вектор).

```
vec3& operator*=(const float& n) {
  (*this) *= vec3(n, n, n); // создаем вектор из трех копий числа
                            // и домножаем на него исходный вектор
  return *this;
}
const vec3 operator*(const float& n) {
  return vec3(*this) *= n;
}
```

Теперь в описание класса `mat3` добавим реализацию операции сложения матриц.

```
mat3& operator+= (const mat3 &m) {
  mat3 B(m); // создаем копию прибавляемой матрицы
  for (int i = 0; i < 3; i++)
    (*this)[i] += B[i];
  return *this;
}
const mat3 operator+ (const mat3 &m) {
  return mat3(*this) += m;
}
```

Здесь же (в классе `mat3`), по аналогии с операцией сложения, добавим реализацию операции умножения матрицы на число.

```
mat3& operator*= (const float &n) {
  for (int i = 0; i < 3; i++)
    (*this)[i] *= n; // домножаем на заданное число каждую строку
  return *this;
}
const mat3 operator* (const float &n) {
  return mat3(*this) *= n;
}
```

Нам понадобится операция векторного произведения $\bar{p} \times \bar{q}$ трехмерных векторов $\bar{p}$ и $\bar{q}$. Реализуем её в матричной форме, пользуясь соотношением $\bar{p} \times \bar{q} = [\bar{p}]_\times \cdot \bar{q}$. Для этого опишем функцию `crossM`, выдающую для трехмерного вектора $\bar{p}$ матрицу

$$[\bar{p}]_\times = \begin{bmatrix} 0 & -p_3 & p_2 \\ p_3 & 0 & -p_1 \\ -p_2 & p_1 & 0 \end{bmatrix}.$$

```
mat3 crossM(vec3 p) { // матрица первого множителя векторного произведения
  return mat3(vec3(0.f, -p.z, p.y),
```

```
                vec3(p.z, 0.f, -p.x),
                vec3(-p.y, p.x, 0.f));
}
```

Мы описали эту функцию ещё и потому, что нам в дальнейшем потребуется матрица множителя векторного произведения.

Теперь опишем функцию `cross`, подсчитывающую векторное произведение в матричной форме.

```
vec3 cross(vec3 p, vec3 q) {
  return crossM(p) * q;
}
```

Опишем функцию `length` для вычисления длины трехмерного вектора. Так как при этом понадобится вычисление квадратного корня, подключим в начале файла

```
#include <math.h>
```

Непосредственно длину вектора можно вычислить как корень квадратный из его скалярного квадрата.

```
float length(vec3 p) {
  return sqrtf(dot(p, p));
}
```

Теперь можно добавить функцию для нормализации вектора, в смысле получения единичного вектора, сонаправленного с исходным. Как мы знаем, такой вектор можно получить из исходного, разделив каждую его координату на длину вектора. Воспользуемся тем, что если из координат вектора $\bar{p} = (p_1, p_2, p_3)$ и его длины $|\bar{p}|$ составим четверку однородных координат $(p_1, p_2, p_3, |\bar{p}|)$, то переходе из однородных координат в евклидовы получим тройку координат $\left(\frac{p_1}{|\bar{p}|}, \frac{p_2}{|\bar{p}|}, \frac{p_3}{|\bar{p}|}\right)$, что как раз и является нормализованным вектором. Опишем функцию `norm`, вычисляющую нормализованный вектор по приведенному алгоритму.

```
vec3 norm(vec3 p) { // нормализация вектора p
  return normalize(vec4(p, length(p)));
}
```

!!! Следует отметить, что вызов конструктора `vec4(p, length(p))` не переводит вектор `p` в однородные координаты с коэффициентом $\alpha =$ `length(p)`, а просто составляет четырехмерный вектор из координат заданного трехмерного и коэффициента `length(p)`.

3.2.2 Трехмерные преобразования

Теперь перейдем в файл `Transform.h` и доопределим в нем недостающие трехмерные преобразования.

У нас уже реализованы функции, выдающие матрицы переноса и масштабирования. Преобразование вращения реализуем как преобразование Родригеса: вращение относительно оси, проходящей через начало координат, заданной единичным трехмерным вектором $\bar{n}$.

Напомним формулу Родригеса для получения такой матрицы:

$$R(\bar{n}) = E + [\bar{n}]_\times \sin\vartheta + [\bar{n}]_\times^2 (1 - \cos\vartheta), \tag{3.1}$$

где E — единичная матрица, ϑ — угол вращения.

Вам нужно самостоятельно реализовать функцию `rotate` от параметров `theta` и `n`, где `theta` — заданный угол, а `n` — направляющий вектор произвольной длины (отличной от нуля), возвращающую матрицу 4×4, полученную по формуле Родригеса.

```
mat4 rotate(float theta, vec3 n) {

}
```

При реализации функции `rotate` обратите внимание на то, что вектор $\bar{n}$ в формуле Родригеса — единичный, а соответствующий параметр функции — вектор произвольной длины.

Добавим вспомогательную функцию `rotateP`, возвращающую матрицу вращения на заданный угол относительно оси, заданной вектором, проходящей через заданную точку.

```
mat4 rotateP(float theta, vec3 n, vec3 P) {

}
```

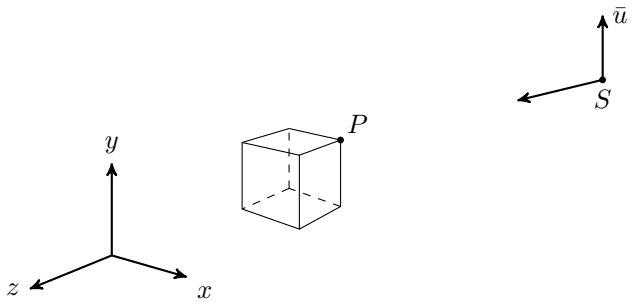
Понятно, что в этой матрице должны совместиться три преобразования: перенос начала координат в точку `P`, вращение относительно оси `n` на заданный угол и перенос начала координат обратно. Перемножим эти матрицы справа налево, получим

```
mat4 rotateP(float theta, vec3 n, vec3 P) {
  return translate(P.x, P.y, P.z) *
         (rotate(theta, n) * translate(-P.x, -P.y, -P.z));
}
```

3.2.3 Переход к системе координат наблюдателя

Реализуем функцию `lookAt`, реализующую переход в систему координат наблюдателя. Рассмотрим подробно процесс формирования этого преобразования.

Пусть задана трехмерная сцена в мировой системе координат.



Будем предполагать, что мировая система координат — правая.

Пусть на сцене заданы (см. рисунок)

- точка наблюдения $S = (x_s, y_s, z_s)$,
- точка $P = (x_p, y_p, z_p)$ на которую направлен вектор наблюдения,
- вектор $\bar{u} = (u_x, u_y, u_z)$ указывающий условное направление вверх.

Система координат наблюдателя — правая декартова система координат, начало которой лежит в точке наблюдения, а ось Oz противоположно направлена вектору наблюдения. Таким образом, для перехода от мировой системы координат к системе координат наблюдателя необходимо:

1. перенести начало координат в точку наблюдения;
2. организовать вращение осей координат таким образом, чтобы ось Oz была направлена в сторону, противоположную вектору наблюдения, а ось Oy была направлена вверх (лежала в одной плоскости с осью Oz и вектором $\bar{u}$).

Первое преобразование простое: достаточно к однородным координатам каждой точки применить преобразование переноса с коэффициентами $T_x = -x_s$, $T_y = -y_s$, $T_z = -z_s$, т. е. необходимо выполнить преобразование заданное матрицей

$$T = \begin{bmatrix} 1 & 0 & 0 & -x_s \\ 0 & 1 & 0 & -y_s \\ 0 & 0 & 1 & -z_s \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Для второго преобразования сначала найдем направляющие векторы $\mathbf{e}'_1$, $\mathbf{e}'_2$, $\mathbf{e}'_3$ для осей системы координат наблюдателя в мировой системе координат.

Направляющий вектор для оси Oz должен быть направлен от точки P к точке S. Следовательно его направление совпадает с направлением вектора $\bar{v}_{PS} = \bar{s} - \bar{p}$, где $\bar{s}$ и $\bar{p}$ — радиус-векторы точек S и P, соответственно. Таким образом

$$\mathbf{e}'_3 = \frac{\bar{v}_{PS}}{|\bar{v}_{PS}|} = \frac{\bar{s} - \bar{p}}{|\bar{s} - \bar{p}|}.$$

Так как вектор $\bar{u}$ указывает примерное направление оси Oy правой системы координат, то если векторно умножить его на полученный вектор $\mathbf{e}'_3$, получим вектор, направление которого совпадает с направлением оси Ox

$$\mathbf{e}'_1 = \frac{\bar{u} \times \mathbf{e}'_3}{|\bar{u} \times \mathbf{e}'_3|}.$$

В правой части равенства присутствует деление на длину вектора $\bar{u}$ для нормализации.

Наконец, чтобы получить направляющий вектор для оси Oy достаточно векторно умножить $\mathbf{e}'_3$ на $\mathbf{e}'_1$.

$$\mathbf{e}'_2 = \frac{\mathbf{e}'_3 \times \mathbf{e}'_1}{|\mathbf{e}'_3 \times \mathbf{e}'_1|}.$$

Теперь можно получить матрицу вращения системы координат, совместив в ней векторы $\mathbf{e}'_1$, $\mathbf{e}'_2$, $\mathbf{e}'_3$ в качестве строк:

$$R = \begin{bmatrix} & \mathbf{e}'_1 & & 0 \\ & \mathbf{e}'_2 & & 0 \\ & \mathbf{e}'_3 & & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Таким образом, для перехода из мировой системы координат в систему координат наблюдателя необходимо для каждой точки $(\chi, \gamma, \zeta, \alpha)$ выполнить преобразование

$$\mathit{LookAt}(S, P, \bar{u}) = R \cdot T$$

Вам нужно самостоятельно реализовать функцию `lookAt`

```
mat4 lookAt(vec3 S, vec3 P, vec3 u) {

}
```

возвращающую матрицу 4×4, полученную в приведенном выше порядке.

3.2.4 Матрицы проекций

Для перехода в пространство отсечения реализуем матрицы прямоугольной и перспективной проекции.

Напомним,что матрица прямоугольной проекции имеет вид

$$Ortho(l, r, b, t, z_n, z_f) = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{z_f-z_n} & -\frac{z_f+z_n}{z_f-z_n} \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

матрицы для перспективной проекции —

$$Frustum(l, r, b, t, n, f) = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix},$$

и

$$Perspective(fovy, aspect, n, f) = \begin{bmatrix} \frac{1}{aspect} \operatorname{ctg} \frac{fovy}{2} & 0 & 0 & 0 \\ 0 & \operatorname{ctg} \frac{fovy}{2} & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}.$$

Здесь (l, b, z_n), (r, t, z_n) — координаты левого нижнего и правого верхнего угла окна наблюдения в системе координат наблюдателя, z_f — минимальная координата z, видимая наблюдателю, n и f расстояния от наблюдателя до плоскости окна наблюдения и до плоскости горизонта, соответственно, $fovy$ — угол зрения по вертикали, $aspect$ — соотношение сторон окна наблюдения.

Реализуем функцию, выдающую первую из этих матриц, составив ее из строк необходимых значений.

```
mat4 ortho(float l, float r, float b, float t, float zn, float zf) {
  return mat4(vec4(2.f / (r - l), 0.f, 0.f, -(r + l) / (r - l)),
              vec4(0.f, 2.f / (t - b), 0.f, -(t + b) / (t - b)),
              vec4(0.f, 0.f, -2 / (zf - zn), -(zf + zn) / (zf - zn)),
              vec4(0.f, 0.f, 0.f, 1.f));
}
```

По такому же принципу самостоятельно опишите функции, выдающие матрицы перспективной проекции.

```
mat4 frustum(float l, float r, float b, float t, float n, float f) {

}

mat4 perspective(float fovy, float aspect, float n, float f) {

}
```

!!! Так как в C++ отсутствует стандартная функция вычисления котангенса, следует пользоваться соотношением $\operatorname{ctg}\alpha = 1/\operatorname{tg}\alpha$.

3.2.5 Преобразование кадрирования

Функцию `cadrRL`, возвращающую матрицу для двумерного преобразования кадрирования, заданного параметрами V_{cx}, V_{cy}, V_x, V_y, W_{cx}, W_{cy}, W_x, W_y, где первые четыре параметра — параметры исходного кадра, а вторая четверка параметров — параметры целевого кадра. При этом предполагаем, что исходный кадр задан в правой системе координат, а целевой — в левой. В этом случае кадрирование представляется формулами

$$x' = W_{cx} + \frac{x - V_{cx}}{V_x} W_x,$$
$$y' = W_{cy} - \frac{y - V_{cy}}{V_y} W_y.$$

Это преобразование можно представить как последовательные преобразования:

1. Перенос начала координат в точку (V_{cx}, V_{cy});
2. Масштабирование с коэффициентами $\frac{W_x}{V_x}$, $\frac{-W_x}{V_x}$;
3. Перенос начала координат в точку $(-W_{cx}, -W_{cy})$.

Оформим это в одной функции.

```
mat3 cadrRL(vec2 Vc, vec2 V, vec2 Wc, vec2 W) {
  return translate(Wc.x, Wc.y) *
         (scale(W.x / V.x, -W.y / V.y) * translate(-Vc.x, -Vc.y));
}
```

Так как до сих пор мы только добавляли в проект дополнительные процедуры, он должен компилироваться без ошибок и работать с прежней функциональностью.

3.3 Добавление третьей размерности

3.3.1 Структуры данных для сцены

Будем считать, что трехмерная сцена в мировой системе координат у нас собирается из моделей, представленных набором трехмерных ломаных, подобно сцене четвертого задания.

Изменим структуру данных для модели. Перейдем к изменению файла `Figure.h` и здесь изменим размерность точек (из двумерных в трехмерные) в классе `path` и размерность модельной матрицы (из 3×3 в 4×4) в классе `model`.

Получим файл со следующим содержимым.

```
1  #pragma once
2  #include "Matrix.h"
3  #include <vector>
4
5  class path {
6  public:
7    std::vector<vec3> vertices; // последовательность точек
8    vec3 color; // цвет, разбитый на составляющие RGB
9    float thickness; // толщина линии
10   path(std::vector<vec3> verts, vec3 col, float thickn) {
11     vertices = verts;
12     color = col;
13     thickness = thickn;
```

```
  }
};

class model {
public:
  std::vector<path> figure; // составляющие рисунка
  mat4 modelM; // модельная матрица
  model(std::vector<path> fig, mat4 mat) {
    figure = fig;
    modelM = mat;
  }
};
```

3.3.2 Формат входного файла

За основу входного файла возьмем файл, полученный в задании 4. Например, в ходе выполнения 4-го урока был получен файл `Geometric.txt` со следующим содержанием (см. подробнее раздел 1.5.3).

```
frame 20 10 # размеры всего изображения из расчета одна единица - две клетки

# первый рисунок
model 1.5 1 3 2 # центр в точке (1.5, 1), размеры 3x2
color 0 255 0 # цвет зеленый
thickness 3 # толщина линии 3
path 5 # путь из четырех ребер (пять вершин)
0.5 0.5 # левый нижний угол
0.5 1.5 # левый верхний угол
2.5 1.5 # правый верхний угол
2.5 0.5 # правый нижний угол
0.5 0.5 # левый нижний угол

# преобразования и размещения по описанию
pushTransform # сохранить отправную точку
rotate -45 # поворот на -45 градусов
pushTransform # сохранить преобразование поворота
scale 2.25 # масштабирование до большого прямоугольника
translate 5 5 # перенос центра рисунка в точку (5,5)
figure # запомнить положение и ракурс первого рисунка
popTransform # откатились к преобразованию поворота
scale 0.75 # масштабирование до малого прямоугольника
translate 15 1 # установить в позицию нижнего малого прямоугольника
figure # запомнить положение и ракурс второго экземпляра рисунка
translate 0 8 # передвинуться в позицию (15,9) из (15,1)
figure # запомнить положение и ракурс третьего экземпляра рисунка
popTransform # откатились к стартовой позиции

# второй рисунок
model 1 1.25 2 2.5 # параметры рисунка с треугольником
color 255 0 0 # цвет красный
path 4 # четыре точки в маршруте
0.5 0.5 # нижний левый угол
1 2 # верхний угол
1.5 0.5 # нижний правый угол
0.5 0.5 # нижний левый угол

# преобразования и размещения по описанию
rotate 90 # поворот на 90 градусов
scale 1.5 # масштабирование до синего прямоугольника
translate 15 5 # сдвиг в нужную позицию
figure # запомнить положение и ракурс рисунка
```

Представим, что изображение, которое мы здесь описываем, — не двумерное, а трехмерное, точки которого лежат в одной плоскости $z = 0$.

Тогда каждая модель должна иметь размер по третьей размерности, а каждая точка, включая точку центра модели — третью координату.

При описании команд преобразований, масштабирование оставим равномерным, заданным одним параметром. Но преобразование переноса пополним третьим параметром — смещением по оси Oz. У преобразования поворота (вращения) также добавим дополнительный параметр, заданный тремя координатами — координатами вектора, задающего ось вращения, проходящую через начало координат.

Таким образом, получаем следующий набор возможных команд.

`model mVcx mVcy mVcz mVx mVy mVz` команда начала описания нового объекта. Все последующие команды `path` будут относиться к этому объекту. Параметры `mVx`, `mVy` и `mVz` — размеры объекта по осям координат в локальных единицах измерения; `mVcx`, `mVcy` и `mVcz` — координаты точки привязки (центра объекта) в объектной системе координат;

`figure` (без параметров) команда окончания формирования начального преобразования для очередной копии объекта;

`translate Tx Ty Tz` преобразование сдвига с коэффициентами `Tx`, `Ty`, `Tz`;

`rotate Phi nx ny nz` преобразование вращения относительно оси, заданной вектором (`nx`, `ny`, `nz`), проходящей через начало координат, против часовой стрелки на угол `Phi`, где `Phi` задан в градусах;

`scale S` преобразование масштабирования относительно начала координат в `S` раз;

`pushTransform` сохранение накопленного совмещенного преобразования в стеке для возможного последующего возвращения к нему;

`popTransform` извлечение из стека преобразований (с удалением из стека) матрицы преобразования, определенной этим преобразованием.

Команду `frame` исключим из файла (она задавала размеры двумерного кадра). Вместо этого введем две дополнительные команды.

`camera Sx Sy Sz Px Py Pz ux uy uz` команда установки камеры в точку с координатами (`Sx`, `Sy`, `Sz`), с направлением наблюдения в точку (`Px`, `Py`, `Pz`), при векторе направления вверх (`ux`, `uy`, `uz`);

`screen fovy aspect near far` команда первоначальной установки окна наблюдения (и пирамиды видимости) на расстоянии `near` от наблюдателя, с соотношением сторон `aspect`, с вертикальным углом обзора `fovy`, заданным в градусах. Параметр `far` задает расстояние до горизонта.

Исправим наш входной файл в соответствии с этим форматом команд: к каждой точке добавим третью координату — 0, к каждому сдвигу добавим третий параметр — 0, к каждому повороту добавим координаты направляющего вектора оси Oz — $(0, 0, 1)$. Кроме того, для моделей установим размер по третьей оси — 1, а третью координату для точки привязки — 0.

Команду `frame` удалим. Вместо нее добавим команды `camera` и `screen`.

Установим камеру так, чтобы окно наблюдения находилось в плоскости нашего изображения. Можно сделать это установив камеру с вертикальным углом обзора 90° в точке, отстоящей от центра рисунка по оси Oz на половину его высоты. Так как наше двумерное изображение было 20×10 (параметры команды `frame`), а все изображение находилось в положительной части координатных осей с левым нижним углом в начале координат, то установим камеру в точке $(10, 5, 5)$, смотрящей в сторону точки $(10, 5, 0)$ и с направлением

вверх, заданным вторым координатным вектором $(0, 1, 0)$.

```
camera 10 5 5 10 5 0 0 1 0
```

Угол обзора камеры — прямой, соотношение сторон окна наблюдения — соотношение сторон рисунка, в нашем случае — два к одному, т. е. равно 2. Расстояние до плоскости окна наблюдения — 5, расстояние до горизонта установим равное 20.

```
screen 90 2 5 20
```

Получим файл со следующим содержанием.

```
# установка камеры в точку (10,5,5) направленной в точку (10,5,0)
# с направлением вверх (0,1,0)
camera 10 5 5 10 5 0 0 1 0
# установка окна с углом обзора 90 градусов
# с соотношением сторон 2:1 на расстоянии 5 от наблюдателя (от камеры)
# расстояние до горизонта - 20
screen 90 2 5 20

# первый рисунок
model 1.5 1 0 3 2 1 # центр в точке (1.5, 1), размеры 3x2
color 0 255 0 # цвет зеленый
thickness 3 # толщина линии 3
path 5 # путь из четырех ребер (пять вершин)
0.5 0.5 0 # левый нижний угол
0.5 1.5 0 # левый верхний угол
2.5 1.5 0 # правый верхний угол
2.5 0.5 0 # правый нижний угол
0.5 0.5 0 # левый нижний угол

# преобразования и размещения по описанию
pushTransform # сохранить отправную точку
rotate -45 0 0 1 # поворот на -45 градусов
pushTransform # сохранить преобразование поворота
scale 2.25 # масштабирование до большого прямоугольника
translate 5 5 0 # перенос центра рисунка в точку (5,5)
figure # запомнить положение и ракурс первого рисунка
popTransform # откатились к преобразованию поворота
scale 0.75 # масштабирование до малого прямоугольника
translate 15 1 0 # установить в позицию нижнего малого прямоугольника
figure # запомнить положение и ракурс второго экземпляра рисунка
translate 0 8 0 # передвинуться в позицию (15,9) из (15,1)
figure # запомнить положение и ракурс третьего экземпляра рисунка
popTransform # откатились к стартовой позиции

# второй рисунок
model 1 1.25 0 2 2.5 1 # параметры рисунка с треугольником
color 255 0 0 # цвет красный
path 4 # четыре точки в маршруте
0.5 0.5 0 # нижний левый угол
1 2 0 0 # верхний угол
1.5 0.5 0 # нижний правый угол
0.5 0.5 0 # нижний левый угол

# преобразования и размещения по описанию
rotate 90 0 0 1 # поворот на 90 градусов
scale 1.5 # масштабирование до синего прямоугольника
```

```
translate 15 5 0 # сдвиг в нужную позицию
figure # запомнить положение и ракурс рисунка
```

3.3.3 Чтение входного файла

Перейдем теперь к изменению файла `MyForm.h` .

Прежде чем изменять фрагмент считывания информации из файла, давайте внесем изменения в блок описания глобальных переменных. Сейчас он состоит из следующих описаний.

```
float Vx; // размер рисунка по горизонтали
float Vy; // размер рисунка по вертикали
float aspectFig; // соотношение сторон рисунка
vector<model> models;
mat3 T; // матрица, в которой накапливаются все преобразования
```

В качестве глобальных переменных нам потребуются параметры, считанные из файла: описание трехмерной сцены с параметрами камеры, а так же аналоги этих переменных, описывающие рабочее состояние изображения (значение параметров после выполнения манипуляций с изображением).

Из всех перечисленных сейчас переменных нам останется нужной только переменные `models` и `T` . Остальные описания переменных удалим. В переменной `models` будем сохранять, как и раньше, список моделей, составляющих трехмерную сцену. У матрицы `T` увеличим размерность. Остальная информация, считываемая из входного файла — параметры камеры и окна наблюдения. Опишем набор переменных для этих параметров.

```
vector<model> models;
mat4 T; // матрица, в которой накапливаются все преобразования
vec3 S, P, u; // координаты точки наблюдения
              // точки, в которую направлен вектор наблюдения
              // вектора направления вверх
float dist; // вспомогательная переменная - расстояние между S и P
float fovy, aspect; // угол обзора и соотношение сторон окна наблюдения
float fovy_work, aspect_work; // рабочие переменные для fovy и aspect
float near, far; // расстояния до окна наблюдения и до горизонта
float n, f; // рабочие переменные для near и far
float l, r, t, b; // рабочие вспомогательные переменные
                  // для значений координат левой, правой,
                  // нижней и верхней координаты в СКН
```

Кроме этих переменных опишем переменную `pType` перечислимого типа, возможные значения которой — типы проекций при переходе к двумерным координатам (в дальнейшем мы будем менять тип текущей проекции).

```
enum projType { Ortho, Frustum, Perspective } pType; // тип трехмерной проекции
```

Теперь перейдем к изменению процедуры `btnOpen_Click` .

Последовательно внесем изменения в тело процедуры, касающиеся размерностей и параметров команд входного файла. В частности, в строках

```
mat3 M = mat3(1.f); // матрица для получения модельной матрицы
mat3 initM; // матрица для начального преобразования каждого рисунка
vector<mat3> transforms; // стек матриц преобразований
```

следует изменить упоминание типа `mat3` на `mat4`. Получим

```
mat4 M = mat4(1.f); // матрица для получения модельной матрицы
mat4 initM; // матрица для начального преобразования каждого рисунка
vector<mat4> transforms; // стек матриц преобразований
```

Пока пропустим блок, связанный с чтением команды `frame`: вернемся к нему позднее.

В блоке, связанном с командой `path` следует изменить размерность считываемых точек: заменить описание

```
vector<vec2> vertices; // список точек ломаной
```

на

```
vector<vec3> vertices; // список точек ломаной
```

и во фрагменте

```
1 float x, y; // переменные для считывания
2 stringstream s1(str1); // еще один строковый поток из строки str1
3 s1 >> x >> y;
4 vertices.push_back(vec2(x, y)); // добавляем точку в список
```

следует добавить переменную `z` и изменить размерность создаваемого вектора. Получим

```
float x, y, z; // переменные для считывания
stringstream s1(str1); // еще один строковый поток из строки str1
s1 >> x >> y >> z;
vertices.push_back(vec3(x, y, z)); // добавляем точку в список
```

В блоке команды `model`

```
1 float mVcx, mVcy, mVx, mVy; // параметры команды model
2 s >> mVcx >> mVcy >> mVx >> mVy; // считываем значения переменных
3 float S = mVx / mVy < 1 ? 2.f / mVy : 2.f / mVx;
4 // сдвиг точки привязки из начала координат в нужную позицию
5 // после которого проводим масштабирование
6 initM = scale(S) * translate(-mVcx, -mVcy);
```

следует описать две дополнительные переменные: для третьей координаты центра и для размера по оси Oz. Кроме этого, функции преобразований масштабирования и смещения вызовем от параметров, приводящих к получению матриц четвертого порядка (функции от трех аргументов). Получим

```
float mVcx, mVcy, mVcz, mVx, mVy, mVz; // параметры команды model
s >> mVcx >> mVcy >> mVcz >> mVx >> mVy >> mVz; // считываем значения переменных
float S = mVx / mVy < 1 ? 2.f / mVy : 2.f / mVx;
// сдвиг точки привязки из начала координат в нужную позицию
// после которого проводим масштабирование
initM = scale(S, S, S) * translate(-mVcx, -mVcy, -mVcz);
```

Стоит обратить внимание, что здесь, для упрощения составления сцены, мы не анализируем размеры по оси Oz при изменении масштабов модели.

В блоке, соответствующем команде `translate`

```
float Tx, Ty; // параметры преобразования переноса
s >> Tx >> Ty; // считываем параметры
M = translate(Tx, Ty) * M; // добавляем перенос к общему преобразованию
```

добавим третий параметр смещения. Получим

```
float Tx, Ty, Tz; // параметры преобразования переноса
s >> Tx >> Ty >> Tz; // считываем параметры
M = translate(Tx, Ty, Tz) * M; // добавляем перенос к общему преобразованию
```

В блоке команды `scale` следует изменить строку с вызовом функции `scale`

```
M = scale(S) * M; // добавляем масштабирование к общему преобразованию
```

указав при вызове два дополнительных параметра (с тем же значением)

```
M = scale(S, S, S) * M; // добавляем масштабирование к общему преобразованию
```

Наконец, в блоке `rotate`

```
float theta; // угол поворота в градусах
s >> theta; // считываем параметр
M = rotate(-theta / 180.f * Math::PI) * M; // добавляем поворот к общему преобразованию
```

опишем и считаем три параметра для вектора, задающего ось вращения, после чего используем этот вектор в вызове функции для получения матрицы вращения. Так как матрица вращения определена для правой системы координат, то знак `theta` изменять не нужно.

```
float theta; // угол поворота в градусах
float nx, ny, nz; // координаты направляющего вектора оси вращения
s >> theta >> nx >> ny >> nz; // считываем параметры
// добавляем вращение к общему преобразованию
M = rotate(theta / 180.f * Math::PI, vec3(nx, ny, nz)) * M;
```

Теперь вернемся к блоку, соответствующему команде `frame` .

```
if (cmd == "frame") { // размеры изображения
  s >> Vx >> Vy; // считываем глобальные значение Vx и Vy
  aspectFig = Vx / Vy; // обновление соотношения сторон
  float aspectRect = Wx / Wy; // соотношение сторон прямоугольника
  // смещение центра рисунка с началом координат
  mat3 T1 = translate(-Vx / 2, -Vy / 2);
  // масштабирование остается прежним, меняется только привязка
  // коэффициент увеличения при сохранении исходного соотношения сторон
  float S = aspectFig < aspectRect ? Wy / Vy : Wx / Vx;
  mat3 S1 = scale(S, -S);
  // сдвиг точки привязки из начала координат в нужную позицию
  mat3 T2 = translate(Wx / 2 + Wcx, Wcy - Wy / 2);
  // В initT совмещаем эти три преобразования (справа налево)
  initT = T2 * (S1 * T1);
  T = initT;
}
```

Вместо этого блока добавим два новых, соответствующих командам `camera` и `screen`

```
if (cmd == "camera") { // положение камеры

}
else if (cmd == "screen") { // положение окна наблюдения

}
```

Блок, соответствующий команде `camera` будет состоять только из считывания значений параметров в соответствующие глобальные переменные

```
if (cmd == "camera") { // положение камеры
  s >> S.x >> S.y >> S.z; // координаты точки наблюдения
  s >> P.x >> P.y >> P.z; // точка, в которую направлен вектор наблюдения
  s >> u.x >> u.y >> u.z; // вектор направления вверх
}
```

В блоке команды `screen` кроме считывания параметров следует перевести угол из градусов в радианы. Поэтому для считывания параметра `fovy` используем вспомогательную рабочую переменную `fovy_work` .

```
else if (cmd == "screen") { // положение окна наблюдения
  s >> fovy_work >> aspect >> near >> far; // параметры команды
  fovy = fovy_work / 180.f * Math::PI; // перевод угла из градусов в радианы
}
```

Для вычисления начальных значений всех вспомогательных рабочих переменных опишем дополнительную процедуру `initWorkPars` . Поместим ее перед обработчикам события *Paint*.

```
private: System::Void initWorkPars() { // инициализация рабочих параметров камеры

}
```

Здесь, во-первых, присвоим всем рабочим дубликатам параметров камеры значения исходных параметров.

```
n = near;
f = far;
fovy_work = fovy;
aspect_work = aspect;
```

Чтобы вычислить значения параметров `l` , `r` , `b` , `t` решим соответствующий треугольник. Получим

```
float Vy = 2 * near * tan(fovy / 2);
float Vx = aspect * Vy;
l = -Vx / 2;
r = Vx / 2;
b = -Vy / 2;
t = Vy / 2;
```

Вычислим расстояние `dist` .

```
dist = length(P - S);
```

Наконец, в качестве начального значения матрицы `T` возьмем матрицу перехода в систему координат наблюдателя с параметрами команды `camera`.

```
T = lookAt(S, P, u);
```

Таким образом, процедура `initWorkPars` примет вид

```
private: System::Void initWorkPars() { // инициализация рабочих параметров камеры
  n = near;
  f = far;
  fovy_work = fovy;
  aspect_work = aspect;
  float Vy = 2 * near * tan(fovy / 2);
  float Vx = aspect * Vy;
  l = -Vx / 2;
  r = Vx / 2;
  b = -Vy / 2;
  t = Vy / 2;
  dist = length(P - S);
  T = lookAt(S, P, u);
}
```

Поместим вызов

```
initWorkPars();
```

в процедуре `btnOpen_Click`, непосредственно перед вызовом

```
Refresh();
```

Процедура `btnOpen_Click` примет вид

```
private: System::Void btnOpen_Click(System::Object^  sender, System::EventArgs^  e) {
  if (openFileDialog->ShowDialog() == System::Windows::Forms::DialogResult::OK) {
    // в файловом диалоге нажата кнопка OK
    // перезапись имени файла из openFileDialog->FileName в fileName
    wchar_t fileName[1024];  // переменная, в которой посимвольно сохраним имя файла
    for (int i = 0; i < openFileDialog->FileName->Length; i++)
      fileName[i] = openFileDialog->FileName[i];
    fileName[openFileDialog->FileName->Length] = '\0';

    // объявление и открытие файла
    ifstream in;
    in.open(fileName);
    if (in.is_open()) {
      // файл успешно открыт
      models.clear(); // очищаем имеющийся список рисунков
      // временные переменные для чтения из файла
      mat4 M = mat4(1.f); // матрица для получения модельной матрицы
      mat4 initM; // матрица для начального преобразования каждого рисунка
      vector<mat4> transforms; // стек матриц преобразований
      vector<path> figure; // список ломаных очередного рисунка
      float thickness = 2; // толщина со значением по умолчанию 2
      float r, g, b; // составляющие цвета
      r = g = b = 0; // значение составляющих цвета по умолчанию (черный)
      string cmd; // строка для считывания имени команды
      // непосредственно работа с файлом
      string str; // строка, в которую считываем строки файла
      getline(in, str); // считываем из входного файла первую строку
      while (in) { // если очередная строка считана успешно
        // обрабатываем строку
        if ((str.find_first_not_of(" \t\r\n") != string::npos) && (str[0] != '#')) {
          // прочитанная строка не пуста и не комментарий
          stringstream s(str); // строковый поток из строки str
```

```
33          s >> cmd;
34          if (cmd == "camera") { // положение камеры
35            s >> S.x >> S.y >> S.z; // координаты точки наблюдения
36            s >> P.x >> P.y >> P.z; // точка, в которую направлен вектор наблюдения
37            s >> u.x >> u.y >> u.z; // вектор направления вверх
38          }
39          else if (cmd == "screen") { // положение окна наблюдения
40            s >> fovy_work >> aspect >> near >> far; // параметры команды
41            fovy = fovy_work / 180.f * Math::PI; // перевод угла из градусов в радианты
42          }
43          else if (cmd == "color") { // цвет линии
44            s >> r >> g >> b; // считываем три составляющие цвета
45          }
46          else if (cmd == "thickness") { // толщина линии
47            s >> thickness; // считываем значение толщины
48          }
49          else if (cmd == "path") { // набор точек
50            vector<vec3> vertices; // список точек ломаной
51            int N; // количество точек
52            s >> N;
53            string str1; // дополнительная строка для чтения из файла
54            while (N > 0) { // пока не все точки считали
55              getline(in, str1); // считываем в str1 из входного файла очередную строку
56              // так как файл корректный, то на конец файла проверять не нужно
57              if ((str1.find_first_not_of(" \t\r\n") != string::npos) && (str1[0] != '#')) {
58                // прочитанная строка не пуста и не комментарий
59                // значит в ней пара координат
60                float x, y, z; // переменные для считывания
61                stringstream s1(str1); // еще один строковый поток из строки str1
62                s1 >> x >> y >> z;
63                vertices.push_back(vec3(x, y, z)); // добавляем точку в список
64                N--; // уменьшаем счетчик после успешного считывания точки
65              }
66            }
67            // все точки считаны, генерируем ломаную (path) и кладем ее в список figure
68            figure.push_back(path(vertices, vec3(r, g, b), thickness));
69          }
70          else if (cmd == "model") { // начало описания нового рисунка
71            float mVcx, mVcy, mVcz, mVx, mVy, mVz; // параметры команды model
72            s >> mVcx >> mVcy >> mVcz >> mVx >> mVy >> mVz; // считываем значения переменных
73            float S = mVx / mVy < 1 ? 2.f / mVy : 2.f / mVx;
74            // сдвиг точки привязки из начала координат в нужную позицию
75            // после которого проводим масштабирование
76            initM = scale(S, S, S) * translate(-mVcx, -mVcy, -mVcz);
77            figure.clear();
78          }
79          else if (cmd == "figure") { // формирование новой модели
80            models.push_back(model(figure, M * initM));
81          }
82          else if (cmd == "translate") { // перенос
83            float Tx, Ty, Tz; // параметры преобразования переноса
84            s >> Tx >> Ty >> Tz; // считываем параметры
85            M = translate(Tx, Ty, Tz) * M; // добавляем перенос к общему преобразованию
86          }
87          else if (cmd == "scale") { // масштабирование
88            float S; // параметр масштабирования
89            s >> S; // считываем параметр
90            M = scale(S, S, S) * M; // добавляем масштабирование к общему преобразованию
91          }
92          else if (cmd == "rotate") { // поворот
93            float theta; // угол поворота в градусах
94            float nx, ny, nz; // координаты направляющего вектора оси вращения
95            s >> theta >> nx >> ny >> nz; // считываем параметры
96            // добавляем вращение к общему преобразованию
97            M = rotate(theta / 180.f * Math::PI, vec3(nx, ny, nz)) * M;
98          }
99          else if (cmd == "pushTransform") { // сохранение матрицы в стек
100           transforms.push_back(M); // сохраняем матрицу в стек
101         }
102         else if (cmd == "popTransform") { // откат к матрице из стека
103           M = transforms.back(); // получаем верхний элемент стека
104           transforms.pop_back(); // выкидываем матрицу из стека
105         }
106       }
```

```
        // считываем очередную строку
        getline(in, str);
      }
      initWorkPars();
      Refresh();
    }
  }
}
```

3.3.4 Отрисовка трехмерного образа

Внесем изменения в обработчик события *Paint*. Эта процедура сейчас имеет вид

```
private: System::Void MyForm_Paint(System::Object^  sender, System::Windows::Forms::PaintEventArgs^  e) {
  Graphics^ g = e->Graphics;
  g->Clear(Color::Aquamarine);

  Pen^ rectPen = gcnew Pen(Color::Black, 2);
  g->DrawRectangle(rectPen, left, top, Wx, Wy);
  for (int k = 0; k < models.size(); k++) { // цикл по рисункам
    vector<path> figure = models[k].figure; // список ломаных очередного рисунка
    mat3 TM = T * models[k].modelM; // матрица общего преобразования рисунка
    for (int i = 0; i < figure.size(); i++) {
      path lines = figure[i]; // lines - очередная ломаная линия
      Pen^ pen = gcnew Pen(Color::FromArgb(lines.color.x, lines.color.y, lines.color.z));
      pen->Width = lines.thickness;

      vec2 start = normalize(TM * vec3(lines.vertices[0], 1.0)); // начальная точка первого отрезка
      for (int j = 1; j < lines.vertices.size(); j++) { // цикл по конечным точкам (от единицы)
        vec2 end = normalize(TM * vec3(lines.vertices[j], 1.0)); // конечная точка
        vec2 tmpEnd = end; // продублировали координаты точки для будущего использования
        if (clip(start, end, minX, minY, maxX, maxY)) { // если отрезок видим
          // после отсечения, start и end - концы видимой части отрезка
          g->DrawLine(pen, start.x, start.y, end.x, end.y);  // отрисовка видимых частей
        }
        start = tmpEnd;  // конечная точка неотсеченного отрезка становится начальной точкой следующего
      }
    }
  }
}
```

Последовательно исправим код. Перед циклом по рисункам следует определить значения матриц преобразований: матрицу перехода из системы координат наблюдателя в пространство отсечения и матрицу для преобразования кадрирования.

В качестве матрицы перехода в пространство отсечения выбираем матрицу проекции со значениями рабочих параметров, в зависимости от значения переменной `pType` .

```
mat4 proj; // матрица перехода в пространство отсечения
switch (pType) {
case Ortho: // прямоугольная проекция
  proj = ortho(l, r, b, t, -n, -f);
  break;
case Frustum: // перспективная проекция с Frustum
  proj = frustum(l, r, b, t, n, f);
  break;
case Perspective: // перспективная проекция с Perspective
  proj = perspective(fovy_work,aspect_work, n, f);
  break;
}
```

В пространстве отсечения (после применения матрицы проекции) все точки окна наблюдения располагаются в диапазоне от -1 до 1 по каждой из осей. Поэтому в операции

кадрирования имеем значения параметров

$$V_{cx} = -1; \qquad V_x = 2;$$
$$V_{cy} = -1; \qquad V_y = 2.$$

Следовательно матрица для операции кадрирования будет вычисляться следующим образом

```
// матрица кадрирования
mat3 cdr = cadrRL(vec2(-1.f, -1.f), vec2(2.f, 2.f), vec2(Wcx, Wcy), vec2(Wx, Wy));
```

Так как преобразования, заданные матрицами `T` и `proj` последовательно выполняются для каждой точки сцены, перемножим их перед входом в цикл, а результат произведения запишем в матрицу `C`.

```
mat4 C = proj * T; // матрица перехода от мировых координат в пространство отсечения
```

Внутри цикла исправим размерность у матрицы `TM` и заменим в выражении для её вычисления матрицу `T` на `C`: заменим строку

```
mat3 TM = T * models[k].modelM; // матрица общего преобразования рисунка
```

на

```
mat4 TM = C * models[k].modelM; // матрица общего преобразования модели
```

Вычисление координат начала и конца отрезков будем выполнять в два этапа. Сейчас координаты начала отрезка вычисляются в строке

```
vec2 start = normalize(TM * vec3(lines.vertices[0], 1.0)); // начальная точка первого отрезка
```

Теперь, в результате подобного преобразования, должна получаться трехмерная точка

```
// начальная точка первого отрезка в трехмерных евклидовых координатах
vec3 start_3D = normalize(TM * vec4(lines.vertices[0], 1.0));
```

Чтобы из трехмерной точки получить двумерную, отбросим третью координату, перейдем к однородным координатам, проведем кадрирование, после чего вернемся снова к евклидовым координатам.

```
// начальная точка первого отрезка в координатах экрана
vec2 start = normalize(cdr * vec3(vec2(start_3D), 1.f));
```

Подобные преобразования проведем для вычисления конечной точки. Заменим строку

```
vec2 end = normalize(TM * vec3(lines.vertices[j], 1.0)); // конечная точка
```

на

```
// конечная точка отрезка в трехмерных евклидовых координатах
vec3 end_3D = normalize(TM * vec4(lines.vertices[j], 1.0));
// конечная точка отрезка в координатах экрана
vec2 end = normalize(cdr * vec3(vec2(end_3D), 1.f));
```

Обработчик события *Paint* примет следующий вид.

```
private: System::Void MyForm_Paint(System::Object^  sender, System::Windows::Forms::PaintEventArgs^  e) {
  Graphics^ g = e->Graphics;
  g->Clear(Color::Aquamarine);

  Pen^ rectPen = gcnew Pen(Color::Black, 2);
  g->DrawRectangle(rectPen, left, top, Wx, Wy);

  mat4 proj; // матрица перехода в пространство отсечения
  switch (pType) {
  case Ortho: // прямоугольная проекция
    proj = ortho(l, r, b, t, -n, -f);
    break;
  case Frustum: // перспективная проекция с Frustum
    proj = frustum(l, r, b, t, n, f);
    break;
  case Perspective: // перспективная проекция с Perspective
    proj = perspective(fovy_work, aspect_work, n, f);
    break;
  }
  // матрица кадрирования
  mat3 cdr = cadrRL(vec2(-1.f, -1.f), vec2(2.f, 2.f), vec2(Wcx, Wcy), vec2(Wx, Wy));
  mat4 C = proj * T; // матрица перехода от мировых координат в пространство отсечения
  for (int k = 0; k < models.size(); k++) { // цикл по моделям
    vector<path> figure = models[k].figure; // список ломаных очередной модели
    mat4 TM = C * models[k].modelM; // матрица общего преобразования модели
    for (int i = 0; i < figure.size(); i++) {
      path lines = figure[i]; // lines - очередная ломаная линия
      Pen^ pen = gcnew Pen(Color::FromArgb(lines.color.x, lines.color.y, lines.color.z));
      pen->Width = lines.thickness;
      // начальная точка первого отрезка в трехмерных евклидовых координатах
      vec3 start_3D = normalize(TM * vec4(lines.vertices[0], 1.0));
      // начальная точка первого отрезка в координатах экрана
      vec2 start = normalize(cdr * vec3(vec2(start_3D), 1.f));
      for (int j = 1; j < lines.vertices.size(); j++) { // цикл по конечным точкам (от единицы)
        // конечная точка отрезка в трехмерных евклидовых координатах
        vec3 end_3D = normalize(TM * vec4(lines.vertices[j], 1.0));
        // конечная точка отрезка в координатах экрана
        vec2 end = normalize(cdr * vec3(vec2(end_3D), 1.f));
        vec2 tmpEnd = end; // продублировали координаты точки для будущего использования
        if (clip(start, end, minX, minY, maxX, maxY)) { // если отрезок видим
          // после отсечения, start и end - концы видимой части отрезка
          g->DrawLine(pen, start.x, start.y, end.x, end.y);  // отрисовка видимых частей
        }
        start = tmpEnd;  // конечная точка неотсеченного отрезка становится начальной точкой следующего
      }
    }
  }
}
```

3.3.5 Запуск проекта

Для того, чтобы мы смогли запустить проект, осталось внести изменения в обработчик события *KeyDown*. Удалим в нем все предварительные вычисления перед оператором `switch` и все реакции на нажатия клавиш, кроме реакции на **Escape**.

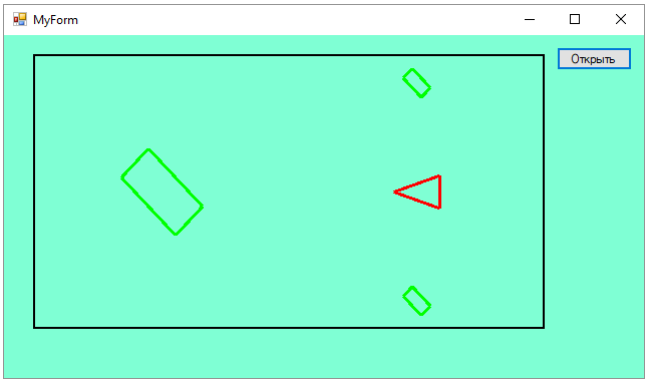
Исправим реакцию на **Escape**: при нажатии этой клавиши установим значения всех рабочих параметров в первоначальные. То есть, поместим здесь вызов процедуры `initWorkPars`.

Обработчик события *KeyDown* на данном этапе примет следующий вид.

```
private: System::Void MyForm_KeyDown(System::Object^  sender, System::Windows::Forms::KeyEventArgs^  e) {
  switch (e->KeyCode) {
  case Keys::Escape:
    initWorkPars();
    break;
  default:
    break;
  }
```

```
9   Refresh();
10 }
```

Если запустить проект, загрузить исправленный файл `Geometric.txt` и растянуть окно, чтобы соотношение сторон прямоугольника в окне равнялось примерно 2:1 (так как теперь рисунок в окне растягивается вместе с окном), то это приведет к следующему изображению



3.4 Изменение используемой матрицы проекции

Реализуем переключение матрицы проекции с помощью нажатия клавиш.

В обработчике события *KeyDown* добавим реакцию на нажатие клавиши **3** — изменим матрицу используемой проекции на матрицу *Perspective*. Для этого просто поменяем значение переменной `pType` на `Perspective`. Само вычисление матрицы у нас уже присутствует в обработчике события *Paint*.

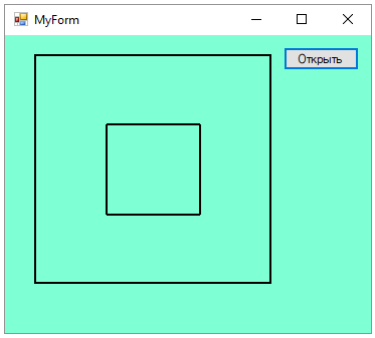
```
case Keys::D3:
  pType = Perspective;
  break;
```

На клавишу **1** вернемся к использованию матрицы прямоугольной проекции.

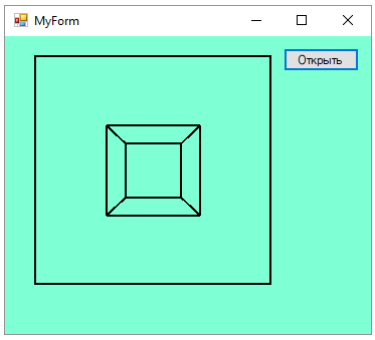
```
case Keys::D1:
  pType = Ortho;
  break;
```

Запуск проекта с последующей загрузкой файла `Geometric.txt` не выявит различий от выбора матрицы проекции. Это обусловлено тем, что изображение, описанное в этом файле, — плоское и находится в плоскости окна наблюдения.

В задании 6 приложен файл `cube.txt` с описанием куба. Если загрузить этот файл в приложении, то ортогональная проекция описанной трехмерной сцены будет иметь следующий вид.



Если же изменить матрицу проекции на `Perspective`, то получим



3.5 Движение камеры

Преобразование, заданное матрицей `lookAt` и записанное первоначально в матрице `T` переводит нашу сцену в систему координат наблюдателя. Теперь смещение камеры можно осуществить переходом к новой системе координат наблюдателя, параметры которой задаются относительно исходной. Тогда каждое такое смещение можно реализовать, домножив матрицу `T` слева на матрицу перехода к новой системе координат наблюдателя.

Реализуем, например, сдвиг камеры вперед на единицу при нажатии клавиши **W**. В системе координат наблюдателя такое преобразование будет означать сдвиг начала координат в точку $(0, 0, -1)$ с сохранением направления всех осей. Это можно выразить переходом в систему координат наблюдателя с направлением наблюдения в точку $(0, 0, -2)$ и с вектором направления вверх $(0, 1, 0)$. Таким образом получим следующий фрагмент кода.

```
case Keys::W:
  T = lookAt(vec3(0, 0, -1), vec3(0, 0, -2), vec3(0, 1, 0)) * T;
  break;
```

По аналогии реализуем сдвиг назад на единицу на нажатие клавиши **S**

```
case Keys::S:
  T = lookAt(vec3(0, 0, 1), vec3(0, 0, 0), vec3(0, 1, 0)) * T;
  break;
```

и сдвиг влево на единицу на нажатие **A**

```
case Keys::A:
  T = lookAt(vec3(-1, 0, 0), vec3(-1, 0, -1), vec3(0, 1, 0)) * T;
  break;
```

На клавишу **R** похожим образом назначим поворот наблюдателя относительно оси Oz на угол 0.1 радиан. Такое преобразование будет осуществляться переходом в новую систему координат наблюдателя, у которой изменится направление вверх. Чтобы изменить направление вверх возьмем вектор $(0, 1, 0)$ и повернем его относительно Oz (вектора $(0, 0, 1)$) на заданный угол. Получим преобразование

```
case Keys::R: {
  vec3 u_new = mat3(rotate(0.1, vec3(0, 0, 1))) * vec3(0, 1, 0);
  T = lookAt(vec3(0, 0, 0), vec3(0, 0, -1), u_new) * T;
  break;
}
```

Более интересным является вращение относительно оси, заданной направляющим вектором оси Ox или Oy. Реализуем на нажатие клавиши **T** разворот камеры на угол 0.1 радиан по вертикали, а на нажатие **Shift-T** — вращение камеры относительно оси вокруг условной точки P — точки, отстоящей от начала координат в отрицательном направлении оси Oz на расстояние `dist`.

```
case Keys::T: {
  if (Control::ModifierKeys == Keys::Shift) {

  }
  else {

  }
  break;
}
```

В случае `else` (клавиша **Shift** не нажата) нужно повернуть точку P относительно оси Ox и это же преобразование применить к вектору направления вверх.

```
mat4 M = rotate(0.1, vec3(1, 0, 0)); // матрица вращения относительно Ox
vec3 u_new = mat3(M) * vec3(0, 1, 0); // вращение направления вверх
vec3 P_new = normalize(M * vec4(0, 0, -1, 1)); // вращение точки, в которую смотрит наблюдатель
T = lookAt(vec3(0, 0, 0), P_new, u_new) * T;
```

В случае, когда дополнительно нажата клавиша **Shift**, нужно повернуть точку начала координат относительно оси параллельной оси Ox, проходящей через условную точку P.

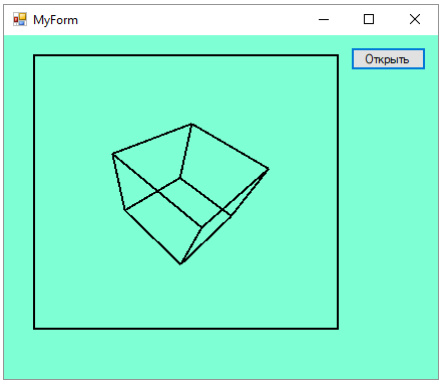
```
mat4 M = rotateP(0.1, vec3(1, 0, 0), vec3(0, 0, -dist)); // матрица вращения относительно точки P
vec3 u_new = mat3(M) * vec3(0, 1, 0); // вращение направления вверх
vec3 S_new = normalize(M * vec4(0, 0, 0, 1)); // вращение начала координат
// переход к СКН в которой начало координат в новой точке, а направление
// наблюдения - в точку P
T = lookAt(S_new, vec3(0, 0, -dist), u_new) *  T;
```

Общий вид реакции на нажатие **T** будет следующий.

```
case Keys::T: {
  if (Control::ModifierKeys == Keys::Shift) {
    mat4 M = rotateP(0.1, vec3(1, 0, 0), vec3(0, 0, -dist)); // матрица вращения
    ↪ относительно точки P
    vec3 u_new = mat3(M) * vec3(0, 1, 0); // вращение направления вверх
    vec3 S_new = normalize(M * vec4(0, 0, 0, 1)); // вращение начала координат
    // переход к СКН в которой начало координат в новой точке, а направление
    // наблюдения - в точку P
    T = lookAt(S_new, vec3(0, 0, -dist), u_new) *  T;
  }
  else {
    mat4 M = rotate(0.1, vec3(1, 0, 0)); // матрица вращения относительно Ox
    vec3 u_new = mat3(M) * vec3(0, 1, 0); // вращение направления вверх
    vec3 P_new = normalize(M * vec4(0, 0, -1, 1)); // вращение точки, в которую смотрит
    ↪ наблюдатель
    T = lookAt(vec3(0, 0, 0), P_new, u_new) * T;
  }
  break;
}
```

!!! Из-за специфики преобразования `lookAt` при вращении относительно оси, параллельной Ox, вектор `u_new` вычислять не обязательно, а можно передать вместо него вектор `vec3(0, 1, 0)`.

Запустите проект, загрузите сцену с кубом, включите перспективное преобразование и опробуйте нажатия назначенных клавиш, управляющих положением камеры.



В частности, первоначальное положение точки P в сцене — в центре куба. Поэтому, нажатие **Shift-T** после загрузки изображения и включения перспективы приведет к вращению куба относительно его центра (движению камеры вокруг куба).

3.6 Изменение параметров окна наблюдения

Добавим реакцию на нажатие клавиши **I**/**Shift-I** — увеличение/уменьшение рабочего значения параметра `t` на 1 в единицах системы координат наблюдателя.

```
case Keys::I:
  if (Control::ModifierKeys == Keys::Shift) {
    t -= 1;
  }
  else {
    t += 1;
  }
  break;
```

По аналогии определим уменьшение/увеличение параметра `l` на нажатие клавиши **J**/**Shift-J**.

```
case Keys::J:
  if (Control::ModifierKeys == Keys::Shift) {
    l += 1;
  }
  else {
    l -= 1;
  }
  break;
```

Запустите проект. Реакцию на нажатия клавиш **I** и **J** можно увидеть, когда включена прямоугольная проекция. Посмотрите, как влияют преобразования, назначенные в этих реакциях, на положение и передвижения камеры.

Обработчик события *KeyDown* примет следующий вид.

```
private: System::Void MyForm_KeyDown(System::Object^  sender, System::Windows::Forms::KeyEventArgs^  e) {
  switch (e->KeyCode) {
  case Keys::Escape:
    initWorkPars();
    break;
  case Keys::W:
    T = lookAt(vec3(0, 0, -1), vec3(0, 0, -2), vec3(0, 1, 0)) * T;
    break;
  case Keys::S:
    T = lookAt(vec3(0, 0, 1), vec3(0, 0, 0), vec3(0, 1, 0)) * T;
    break;
  case Keys::A:
    T = lookAt(vec3(-1, 0, 0), vec3(-1, 0, -1), vec3(0, 1, 0)) * T;
    break;
  case Keys::R: {
    vec3 u_new = mat3(rotate(0.1, vec3(0, 0, 1))) * vec3(0, 1, 0);
    T = lookAt(vec3(0, 0, 0), vec3(0, 0, -1), u_new) * T;
    break;
  }
  case Keys::T: {
    if (Control::ModifierKeys == Keys::Shift) {
      mat4 M = rotateP(0.1, vec3(1, 0, 0), vec3(0, 0, -dist)); // матрица вращения относительно точки P
      vec3 u_new = mat3(M) * vec3(0, 1, 0); // вращение направления вверх
      vec3 S_new = normalize(M * vec4(0, 0, 0, 1)); // вращение начала координат
                            // переход к СКН в которой начало координат в новой точке, а направление
                            // наблюдения - в точку P
      T = lookAt(S_new, vec3(0, 0, -dist), u_new) *  T;
    }
    else {
      mat4 M = rotate(0.1, vec3(1, 0, 0)); // матрица вращения относительно 0x
      vec3 u_new = mat3(M) * vec3(0, 1, 0); // вращение направления вверх
      vec3 P_new = normalize(M * vec4(0, 0, -1, 1)); // вращение точки, в которую смотрит наблюдатель
      T = lookAt(vec3(0, 0, 0), P_new, u_new) * T;
    }
    break;
  }
  case Keys::I:
    if (Control::ModifierKeys == Keys::Shift) {
      t -= 1;
    }
    else {
      t += 1;
    }
    break;
  case Keys::J:
    if (Control::ModifierKeys == Keys::Shift) {
      l += 1;
    }
    else {
      l -= 1;
    }
    break;
  case Keys::D1:
    pType = Ortho;
    break;
  case Keys::D3:
    pType = Perspective;
    break;
  default:
    break;
  }
  Refresh();
}
```

!!! В задании 6 приведены примеры моделей (файлы `cube.txt`, `pyramide.txt`, `car.txt`, `bunny.txt`, `engine.txt`), которые Вы можете загрузить в приложение и с помощью которых можете исследовать зависимость изображения от параметров трехмерной сцены.

3.7 Задание для самостоятельной работы

Задание 6

1. Создайте проект, включающий в себя все, что было описано выше в качестве примера. Проект должен назваться Вашей фамилией, записанной латинскими буквами.
2. В файле `Transform.h` должна присутствовать реализация функций
   - `rotate(theta, n)` для вращения на угол `theta` относительно оси, заданной вектором `n`;
   - `lookAt(S, P, u)` для перехода в систему координат наблюдателя с центром в точке `S`, вектором наблюдения направленным в точку `P` и направлением вверх, заданным вектором `u`;
   - `frustum(l, r, b, t, n, f)` для получения матрицы перспективной проекции на окно наблюдения, ограниченное по оси Ox значениями `l` и `r`, по Oy — значениями `b` и `t`, лежащее в плоскости, отстоящей от наблюдателя на расстояние `n` и с расстоянием до горизонта — `f`;
   - `perspective(fovy, aspect, n, f)` для получения матрицы перспективной проекции на окно наблюдения с соотношением сторон `aspect`, отстоящее от наблюдателя на расстояние `n`, с углом вертикального обзора `fovy` и с расстоянием до горизонта — `f`.
3. Добавьте реакции на нажатия клавиш:
   - **2** — включение перспективной проекции с использованием матрицы `Frustum`;
   - **D** — смещение камеры строго вправо на одну единицу в системе координат наблюдателя с сохранением ракурса (с сохранением направления всех осей системы координат наблюдателя);
   - **Shift-W**, **Shift-S**, **Shift-A**, **Shift-D** — «медленное» смещение камеры: смещение, аналогичное имеющемуся, но не на единицу, а на 0.1;
   - **Y** — поворот камеры относительно оси Oz по часовой стрелке на угол 0.1 радиан;
   - **G**/**Shift-G** — разворот камеры на 0.1 радиан по часовой стрелке относительно оси, параллельной Ox и проходящей через начало координат/условную точку P;
   - **F**/**Shift-F**, **H**/**Shift-H** — разворот камеры на 0.1 радиан против и по часовой стрелке относительно оси, параллельной Oy и проходящей через начало координат/условную точку P;
   - **K**/**Shift-K**, **L**/**Shift-L** — изменение значений параметров `b` и `r` на единицу, приводящее к увеличению/уменьшению окна наблюдения;
   - **U**/**Shift-U** — увеличение уменьшение параметра `n` на 0.2, ограничив его снизу значением 0.1 и сверху значением (`f` -0.1);
   - **O**/**Shift-O** — увеличение уменьшение параметра `f` на 0.2, ограничив его снизу значением (`n` $+0.1$);
   - **B**/**Shift-B** — увеличение уменьшение параметра `dist` на 0.2, ограничив его снизу значением 0.1;
   - **Z**/**Shift-Z** — увеличение уменьшение параметра `fovy_work`, ограничив его значения диапазоном от 0.3 до 3 радиан;
   - **X**/**Shift-X** — увеличение уменьшение параметра `aspect_work` на 0.05, ограничив его снизу значением 0.01;
4. Измените файл, описывающий двумерное изображение, соответствующее Вашему

варианту в задании 4, так, чтобы он описывал то же самое плоское, но в трехмерной системе координат. Файл должен корректно загружаться в приложение. Получившийся файл назовите `Geometric3D.txt`.

5. Исправьте файл `Geometric3D.txt` следующим образом. Разместите экземпляры рисунков, составляющих ваше изображение, в том же ракурсе, но на разной глубине (с разными координатами z точки привязки) так, чтобы первоначальная прямоугольная проекция соответствовала схеме и пропорциям исходного рисунка. Получившийся файл назовите `Geometric3D-1.txt`.
6. Исправьте файл `Geometric3D-1.txt` следующим образом:
   - на месте общего зайца должна быть помещена модель куба из файла `cube.txt`;
   - на месте изображения вашего варианта задания 2 должна быть помещена пирамида из файла `pyramide.txt`;
   - если какой-то экземпляр рисунка в схеме имеет поворот, то соответствующую модель нужно дополнительно развернуть на тот же угол относительно оси, проходящей через её центр, заданной вектором $(1, 1, 0)$.

   Получившийся файл назовите `Geometric3D-2.txt`.
7. В качестве результата выполнения задания должен быть загружен два архив получившегося проекта со стандартным именем. Архив должен включать файлы `Geometric3D.txt`, `Geometric3D-1.txt` и `Geometric3D-2.txt`.

■

3.8 Контрольные вопросы

Подготовьте ответы на следующие вопросы.

1. Объясните, почему операции с матрицами, реализуемые парой функций `crossM` и `cross`, действительно обеспечивают получение значения векторного произведения.
2. Объясните смысл нашей реализации функции вычисления длины вектора (`length`).
3. Объясните смысл нашей реализации функции нормализации вектора (`norm`).
4. Останется ли формула Родригеса, приведенная в разделе 3.2.2 корректной, если предположить, что вектор $\bar{n}$ в ней не единичный? Аргументируйте свой ответ.
5. Если во входном файле описана модель с начальной командой

   ```
   model 4 8 16 8 16 32
   ```

   какие размеры она будет иметь в мировой системе координат, если к ней не применять дополнительных преобразований масштабирования? Объясните свой ответ.
6. Объясните смысл параметров команды входного файла `screen`. Как от параметров этой команды зависит положение и размер окна наблюдения? Аргументируйте свой ответ.
7. Укажите в программном коде `MyForm.h` фрагменты, в которых происходит переход от локальных координат модели к мировой системе координат и системе координат наблюдателя. Объясните смысл всех задействованных операций и переменных.
8. В разделе 3.5 есть замечание, что из-за специфики преобразования `lookAt`, при вращении относительно оси, параллельной Ox, вектор `u_new` вычислять не обязательно, а можно передать вместо него вектор `vec3(0, 1, 0)`. В чем заключается эта специфика и почему возможна такая подмена третьего параметра?

9. Если загрузить в приложение описание из файла `Geometric.txt` (измененного в соответствии с этим уроком), включить прямоугольную проекцию и осуществлять движение вперед, назначенное на клавишу **W**, то изображение в окне меняться не будет. Почему?
10. Если загрузить в приложение описание из файла `Geometric.txt` (измененного в соответствии с этим уроком), включить перспективную проекцию и осуществлять движение вперед, назначенное на клавишу **W**, пройти сквозь изображение, то перед нами вскоре снова окажется это же изображение, но вывернутое наизнанку, причем при нашем продолжении движения вперед оно будет от нас отдаляться. Почему это происходит?
11. Если загрузить в приложение описание из файла `bunny.txt` , то при переключении типа проекции от прямоугольной к перспективной (если не делать никаких других преобразований) размеры всех отрезков в сцене уменьшатся. Почему это происходит?
12. Если запустить приложение, загрузить в него описание из файла `car.txt` и некоторое время осуществлять вращение, назначенное на клавишу **R** (не выполняя других преобразований), то машина будет растягиваться и сжиматься в зависимости от угла поворота. Объясните, почему это происходит.
13. Если загрузить в приложение описание куба, включить перспективную проекцию и некоторое время осуществлять вращение, назначенное на клавишу **T**, то в окне появятся какие-то отрезки, не похожие на части куба. Откуда они берутся?
14. У функций `Ortho` и `Frustum` почти одинаковый набор координат. В чем отличие проекций, получаемых с помощью матриц, возвращаемых этими функциями?
15. Перспективное преобразование в проекте реализуется с помощью матриц, возвращаемых функциями `Frustum` и `Perspective` . Объясните разницу между этими преобразованиями.
16. Объясните смысл фрагментов кода, назначенных на нажатия клавиш для переноса/поворота камеры.
17. Почему при растяжении окна формы изображение растягивается вместе с ним? Укажите фрагменты программы, ответственные за это.
18. Что изменяется в работе приложения при изменении параметра `dist` ?
19. Как будет меняться изображение в окне приложения при изменении параметра `n` в прямоугольной и перспективной проекции? Почему?
20. Как будет меняться изображение в окне приложения при изменении параметра `f` в прямоугольной и перспективной проекции? Почему?
21. Почему в обработчике события *Paint* первые два фактических параметра вызова функции `cadrRL` — векторы `vec2(-1, -1)` и `vec2(2, 2)` ?
22. В разделе 3.3.3 при описании процедуры `initWorkPars` проводится «решение треугольника». Объясните смысл каждого действия этого фрагмента процедуры `initWorkPars` .