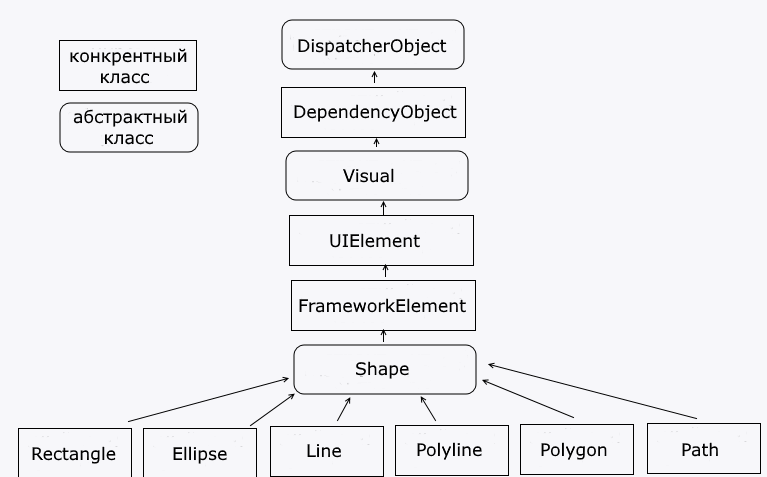
# Введение в графику

## Работа с графикой

Одним из способов построения двухмерной графики в окне — это использование фигур. Фигуры фактически являются обычными элементами, как например кнопка или текстовое поле. К фигурам относят такие элементы как **Polygon** (Многоугольник), **Ellipse** (овал), **Rectangle** (прямоугольник), Line (обычная линия), **Polyline** (несколько связанных линий). Все они наследуются от абстрактного базового класса *System.Windows.Shapes.Shape*:



От базового класса они наследуют ряд общих свойств:

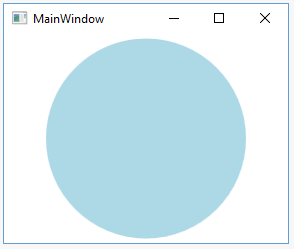
* **Fill** заполняет фон фигуры с помощью кисти - аналогичен свойству Background у прочих элементов
* **Stroke** задает кисть, которая отрисовывает границу фигуры - аналогичен свойству BorderBrush у прочих элементов
* **StrokeThikness** задает толщину границы фигуры - аналогичен свойству BorderThikness у прочих элементов
* **StrokeStartLineCap** и **StrokeEndLineCap** задают для незамкнутых фигур (Line) контур в начале и в конце линии соответственно
* **StrokeDashArray** задает границу фигуры в виде штриховки, создавая эффект пунктира
* **StrokeDashOffset** задает расстояние до начала штриха
* **StrokeDashCap** задает форму штрихов

### Ellipse

**Ellipse** представляет овал:

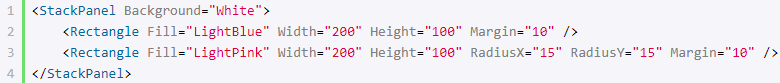


При одинаковой ширине и высоту получается круг:



### Rectangle

**Rectangle** представляет прямоугольник:



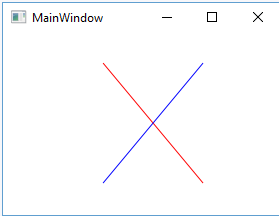
С помощью свойств **RadiusX** и **RadiusY** можно округлить углы прямоугольника:



### Line

Line представляет простую линию. Для создания линии надо указать координаты в ее свойствах **X1**, **Y1**, **X2** и **Y2**. При этом надо учитывать, что началом координатной системы является верхний левый угол:

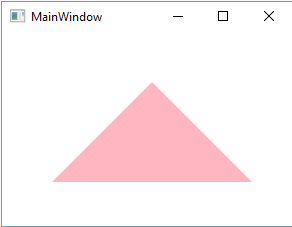




### Polygon

**Polygon** представляет многоугольник. С помощью коллекции **Points** элемент устанавливает набор точек - объектов типа Point, которые последовательно соединяются линиями, причем последня точка соединяется с первой:



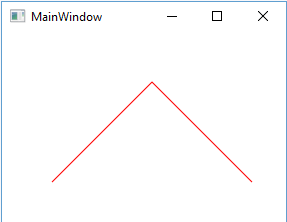


В данном случае у нас три точки (50, 150), (150, 50) и (250, 150), которые образуют треугольник.

### Polyline

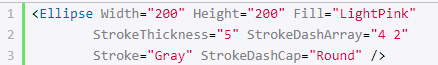
**Polyline** представляет набор точек, соединенных линиями. В этом плане данный элемент похож на Polygon за тем исключением, что первая и последняя точка не соединяются:

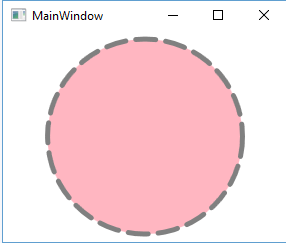




### Настройка контура

С помощью ряда свойств мы можем настроить отображение контура. Например:





Цвет самого контура определяется с помощью свойства **Stroke**, а его толщина - с помощью **StrokeThickness**.

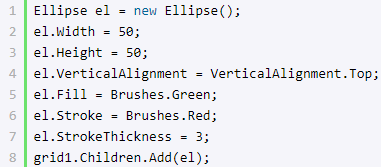
**StrokeDashArray** устанавливает длину штрихов вместе с отступами. Например, StrokeDashArray="4 2" устанавливает длину штриха в 4 единицы, а последующего отступа в 2 единицы. И эти значения будут повторяться по всему контуру. При другой установке, например, StrokeDashArray="1 2 3" уже задается два штриха. Первый штрих имеет длину в 1 единицу, а второй - в 3 единицы и между ними расстояние в 2 единицы. И так вы можем настроить количество штрихов и расстояний между ними.

**StrokeDashCap** задает форму на концах штрихов и может принимать следующие значения:

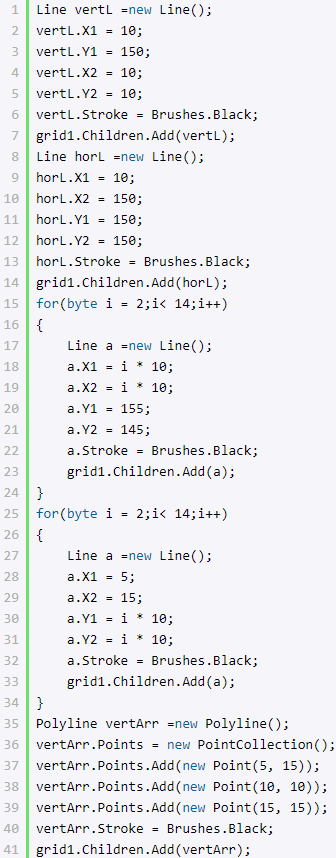
* **Flat**: стандартные штрихи с плоскими окончаниями
* **Square**: штрихи с прямоугольными окончаниями
* **Round**: штрихи с округлыми окончаниями
* **Triangle**: штрихи с окончаниями в виде треугольников

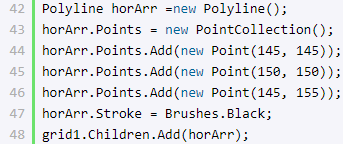
### Программное рисование

Создание фигур программным образом осуществляется так же, как и создаются и добавляются все остальные элементы:



Нарисуем, к примеру, координатную плоскость:





## Пути и геометрии

Фигуры удобны для создания самых простейших рисунков, дизайна, однако что-то более сложное и комплексное с их помощью сделать труднее. Поэтому для этих целей применяется класс **Path**, который представляет геометрический путь. Он также, как и фигуры, наследуется от класса Shape, но может заключать в себе совокупность объединенных фигур. Класс Path имеет свойство **Data**, которое определяет объект Geometry - геометрический объект для отрисовки. Этот объект задает фигуру или совукупность фигур для отрисовки.

Класс *Geometry* - абстрактный, поэтому в качестве объекта используется один из производных классов:

* **LineGeometry** представляет линию, эквивалент фигуры Line
* **RectangleGeometry** представляет прямоугольник, эквивалент фигуры Rectangle
* **EllipseGeometry** представляет эллипс, эквивалент фигуры Ellipse
* **PathGeometry** представляет путь, образующий сложную геометрическую фигуру из простейших фигур
* **GeometryGroup** создает фигуру, состоящую из нескольких объектов Geometry
* **CombinedGeometry** создает фигуру, состоящую из двух объектов Geometry
* ***StreamGeometry*** - специальный объект Geometry, предназначенный для сохранения всего геометрического пути в памяти

Например, использование **LineGeometry**:



будет аналогично следующему объекту **Line**:

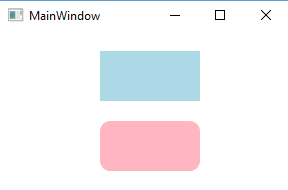


Свойства **StartPoint** и **EndPoint** задают начальную и конечную точки линии.

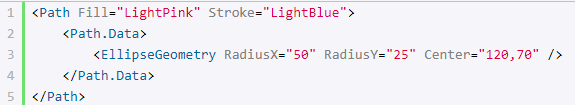
**RectangleGeometry**:



Свойство **Rect** задает параметры прямоугольника в формате "координата X, координата Y ширина, высота". Также с помощью свойств **RadiusX** и **RadiusY** можно задать радиус скругления углов прямоугольника.

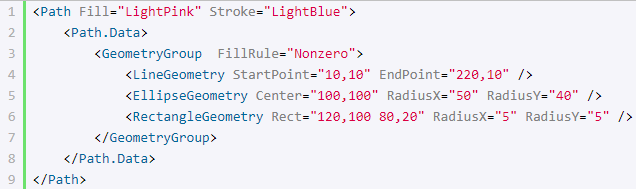


**EllipseGeometry**:



Свойство **Center** устанавливает центр овала, а свойста **RadiusX** и **RadiusY** - радиусы.

**GeometryGroup** объединяет несколько геометрий:



Объект **GeometryGroup** устанавливает свойство **FillRule**. Если оно равно **EvenOdd** (значение по умолчанию), то перекрывающиеся поверхности двух геометрий являются прозрачными. А при значении FillRule="**Nonzero**" (как в данном случае), перекрывающиеся поверхности геометрий будут окрашены также, как и остальные части пути.

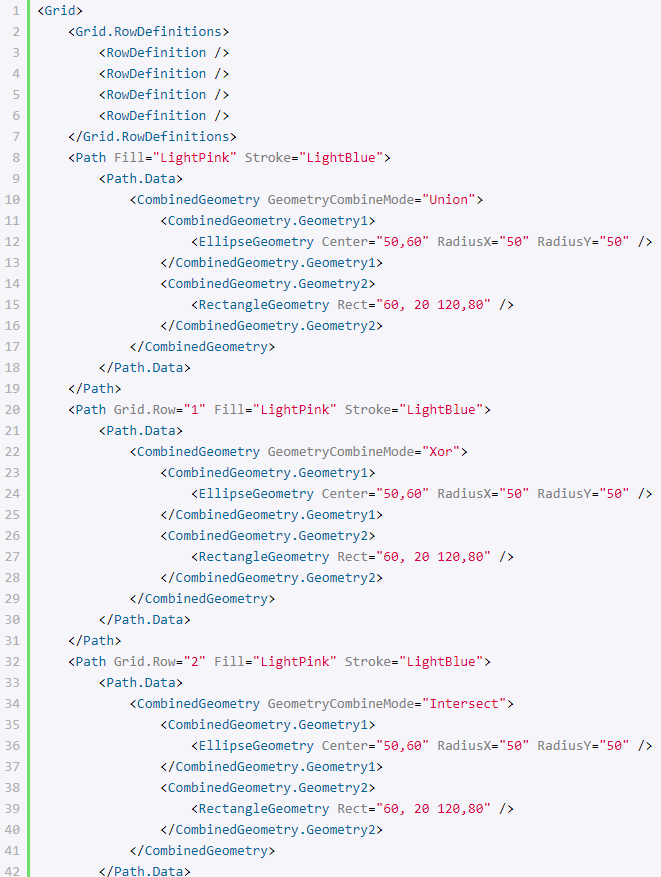


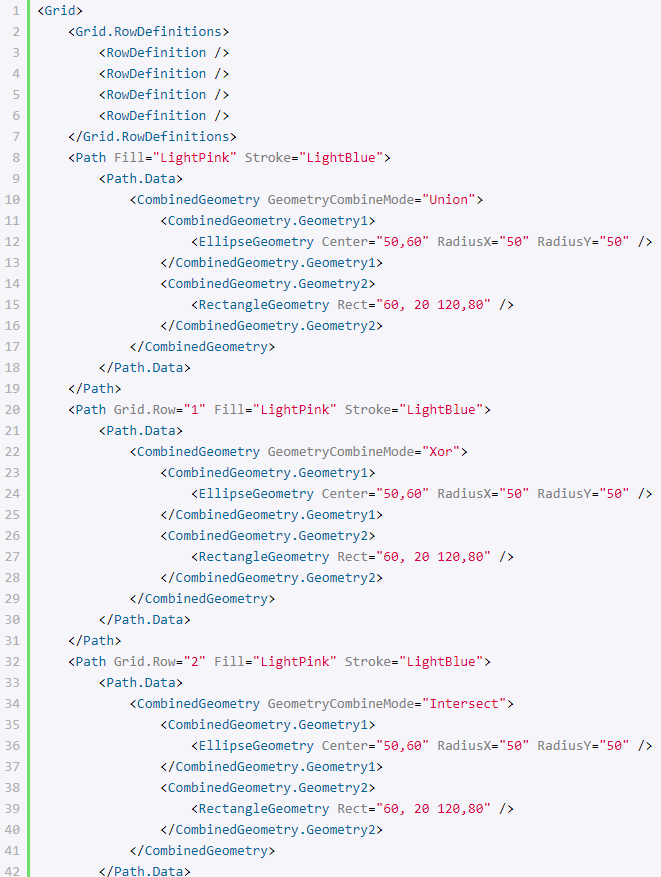
### CombinedGeometry

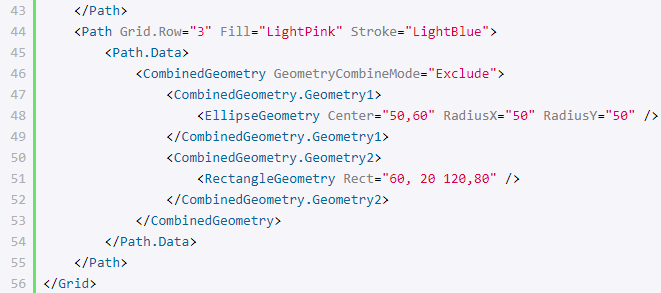
**CombinedGeometry** состоит из двух геометрий. В этом он похож на GeometryGroup, который также может объединять две геометрии. Однако между ними есть различия. Отличие состоит в том, что объект **CombinedGeometry** имеет свойство **GeometryCombinedMode**, которое указывает модель перекрытия двух геометрий:

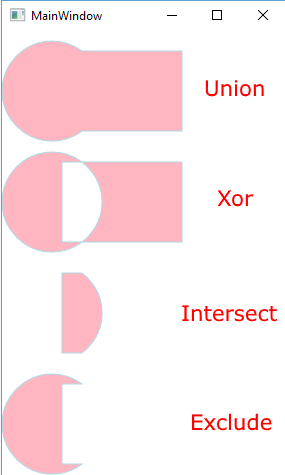
* **Union**: фигура включает обе геометрии
* **Intersect**: фигура включает область, которая одновременно принадлежит обеим геометриям
* **Xor**: фигура включает только непересекающие области геометрий
* **Exclude**: фигура включает первую геометрию с исключением тех областей, которые принадлежат также и второй геометрии

Применим все способы:









## PathGeometry

**PathGeometry** позволяет создавать более сложные по характеру геометрии. PathGeometry содержит один или несколько компонентов **PathFigure**. Объект **PathFigure** в свою очередь формируется из сегментов. Все сегменты наследуются от класса **PathSegment** и бывают нескольких видов:

* **LineSegment** задает отрезок прямой линии между двумя точками
* **ArcSegment** задает дугу
* **BezierSegment** задает кривую Безье
* **QuadraticBezierSegment** задает квадратичную кривую Безье
* **PolyLineSegment** задает сегмент из нескольких линий
* **PolyBezierSegment** задает сегмент из нескольких кривых Безье
* **PolyQuadraticBezierSegment** задает сегмент из нескольких квадратичных кривых Безье

Эти сегменты составляют свойство **Segment** объекта PathFigure. Кроме того, PathFigure имеет еще несколько важных свойств:

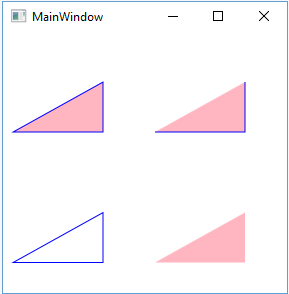
* ***StartPoint*** - точка начала первой фигуры
* **IsClosed** - если значение равно true, то первая и последняя точки (если они не совпадают) соединяются
* **IsFilled** - если значение равно true, то площадь внутри пути окрашивается кистью, заданной свойством Fill у объекта Path

### Линии

Создание линий с помощью **PathGeometry**:





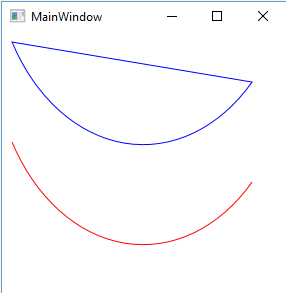


Управляя свойством IsClosed мы можем получить как замкнутый, так и незамкнутый контур. А при применении свойства Fill в элементе Path содержимое контура заполняется цветом. Таким образом, мы можем получить не просто линию, а целостные фигуры, заполненные цветом.

### Дуга



Для создания дуги у ArcSegment задается свойство **Point**, которое указывает на конечную точку дуги (начальная точка задается через свойство StartPoint элемента PathFigure), а свойство **Size** устанавливает размер окружностей, по которым строится дуга

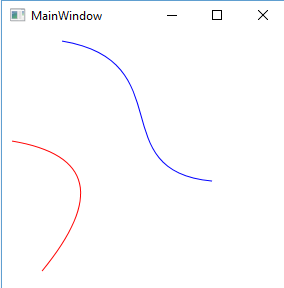
.

### Кривые Безье

Кривые Безье представляют линии, для построения которых применяются сложные математические преобразования. В WPF кривые Безье представлены различными типами: простые и квадратичные кривые. Для построения кривых Безье используются начальная и конечная точки, а также ряд промежуточных точек:

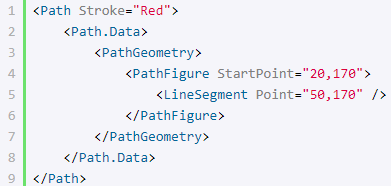


Начальная точка кривых устанавливается с помощью свойства StartPoint элемента PathFigure. Для простой кривой Безье свойства Point1 и Point2 задают промежуточные точки, а Point3 является конечной точкой. Для квадратичной кривой Point2 - конечная точка, а Point1 - промежуточная.



### Сокращенная запись пути

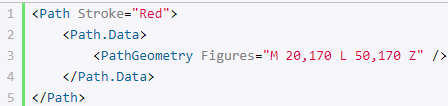
Также принят упрощенный вариант записи фигур. Например, следующее описание фигуры



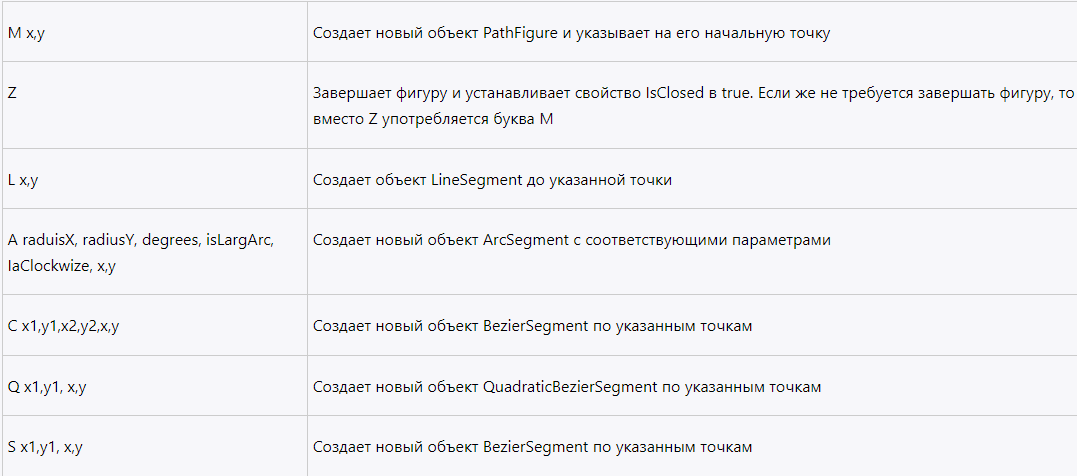
можно написать следующим образом



или

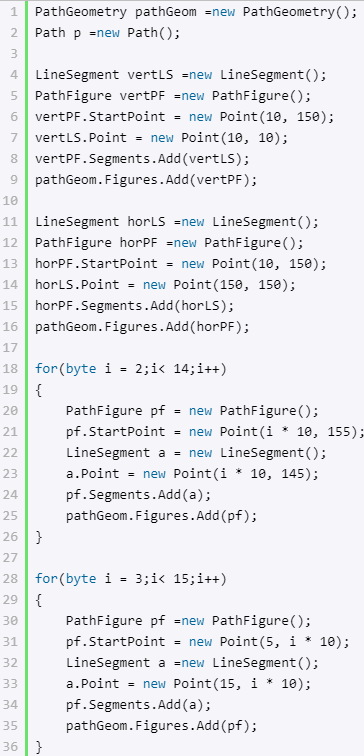


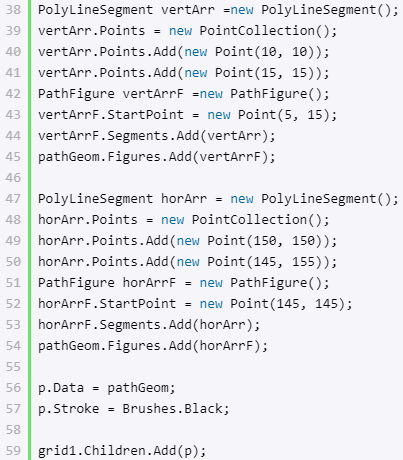
Что в данном случае означает сокращенная запись?



### Программное создание сегментов

Создадим программно координатную плоскость с использованием объекта **PathGeometry**:





## Трансформации

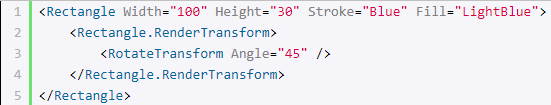
Трансформации представляют инструмент изменения положения или размера элементов WPF. Трансформации могут быть полезны в тех ситуациях, когда надо изменить положение элемента, либо анимировать. Все трансформации наследуются от абстрактного базового класса **System.Windows.Media.Transform** и представляют следующие классы:

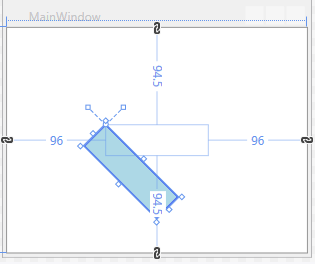
* TranslateTransform: сдвигает элементы по горизонтали и вертикали
* RotateTransform: вращает элемент
* ScaleTransform: выполняет операции масштабирования
* SkewTransform: изменяет позицию элемента путем наклона на определенное количество градусов
* MatrixTransform: изменяет координатную систему в соответствии с определенной матрицей
* TransformGroup: представляет группу трансформаций

### RotateTransform

RotateTransform поворачивает элемент вокруг оси на определенное количество градусов. Данный объект принимает три основых параметра:

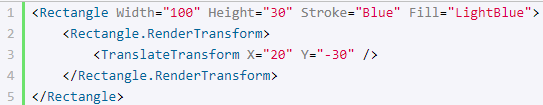
* Angle: угол поворота
* CenterX: устанавливает центр вращения по оси X
* CenterY: устанавливает центр вращения по оси Y

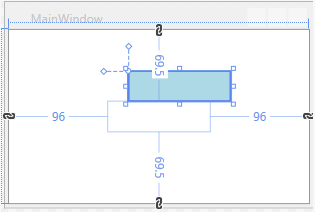




### TranslateTransform

**TranslateTransform** позволяет сместить положение элемента по оси X, с помощью свойства X, и по оси Y - с помощью свойства Y.

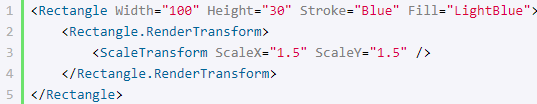


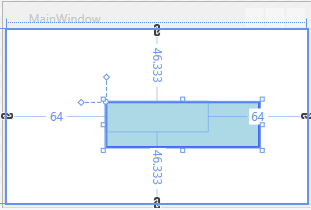


### ScaleTransform

Обеспечивает масштабирование элемента на определенную величину. Для изменения ширины надо задать свойство ScaleX, а для изменения длины - свойство ScaleY. Кроме того, также имеются свойства **CenterX** и **CenterY**, позволяющие позиционировать элемент.

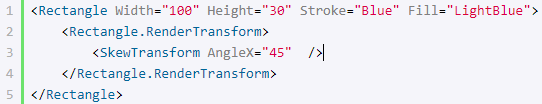
Например, увеличение прямоугольника в полтора раза:

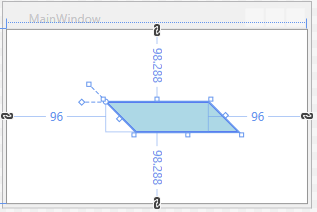




### SkewTransform

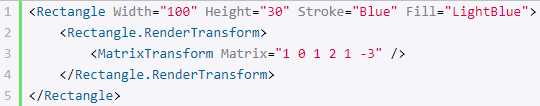
SkewTransform позволяет задать наклон элемента вдоль оси X с помощью свойства **AngleX**, и по оси Y - с помощью свойства **AngleY**. А с помощью свойств **CenterX** и **CenterY** можно изменить положение элемента относительно осй X и Y:

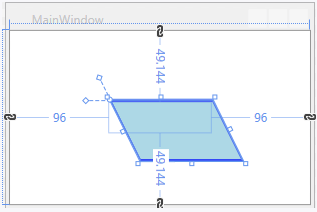




### MatrixTransform

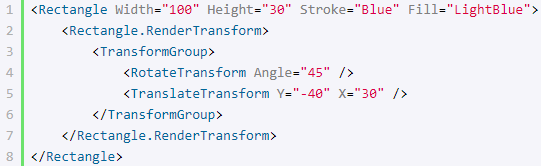
Осуществляет матричное преобразование элемента. В свойстве **Matrix** мы задаем первые два столбца, которые применяются при преобразовании. Последний столбец по умолчанию имеет значения {0 0 1}.





### TransformGroup

TransformGroup позволяет комбинировать различные трансформации вместе:



### RenderTransform и LayoutTransform

Для применения трансформаций у фигур и стандартных элементов управления WPF используются свойства **RenderTransform** и **LayoutTransform**. Несмотря на то, что для обоих свойств трансформации задаются одинаково, их действие различается. Так, свойство LayoutTransform применяется до компоновки элемента, а RenderTransform - после, поэтому одинаковые трансформации для этих свойств могут давать немного разные результаты:



