Трехмерная графика

Основы работы с трехмерной графикой

Работать с графикой в принципе очень интересно, но работать с трехмерной - особенно интересно. И WPF предлагает нам удобный инструментарий для этого. Конечно, создавать суперсложные трехмерные сцены или игры на WPF - очень не простой процесс, и лучше для этого выбрать DirectX, OpenGl, либо специально заточенные под это фреймворки и движки на подобие Unity, Monogame и т.д.. Однако для относительно несложных в том числе бизнес-приложений трехмерные возможности WPF вполне сойдут.

Для создания трехмерной сцены в приложении WPF требуется несколько компонентов:

- **Окно просмотра (Viewport3D)**, которое и содержит трехмерную сцену.
- **Сам объект или геометрия**.
- **Камера**, которая устанавливает, как сцена или объект будет отображаться.
- **Освещение**, которое и содержит трехмерную сцену.
- **Материал**, который вместе с освещением определяет внешний вид трехмерного объекта.

Итак, контейнером верхнего уровня для трехмерной сцены является объект **Viewport3D**. Формально это такой же объект со всеми свойствами, как кнопка или текстовое поле, которое мы можем позиционировать на форме как угодно.

Для отображения трехмерной сцены в Viewport3D надо установить камеру с помощью свойства **Camera**. Без камеры мы не сможем лицезреть трехмерные объекты.

Для добавления самих трехмерных объектов в Viewport3D предусмотрено свойство **Children**, в котором определяются один или несколько объектов **ModelVisual3D**. А каждый из этих объектов ModelVisual3D, который добавляется в Viewport3D, имеет свойство **Content**. Это свойство принимает в качестве значения один объект **GeometryModel3D**, либо объект **Model3DGroup**, который содержит группу объектов **GeometryModel3D**, которые уже непосредственно устанавливают форму трехмерного объекта.

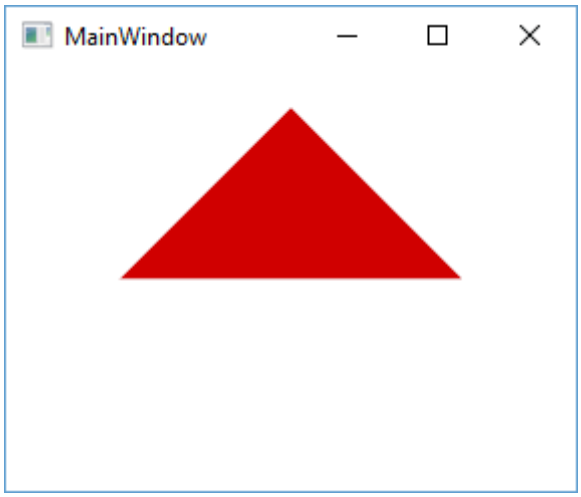
Рассмотрим использование Viewport3D и создание простейшей трехмерной фигуры:

```
<Window x:Class="_3DApp.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:_3DApp"
        mc:Ignorable="d"
        Title="MainWindow" Height="300" Width="300">
    <Grid>
        <Viewport3D>
            <Viewport3D.Camera>
                <!--Установка камеры - перспективная проекция-->
                <PerspectiveCamera Position="0,0,2" LookDirection="0,0,-2" />
            </Viewport3D.Camera>
            <Viewport3D.Children>
                <ModelVisual3D>
                    <ModelVisual3D.Content>
                        <!--Установка освещения - прямой свет-->
                        <DirectionalLight Color="White" Direction="-1,-1,-2" />
                    </ModelVisual3D.Content>
                </ModelVisual3D>
                <ModelVisual3D>
                    <ModelVisual3D.Content>
                        <GeometryModel3D>
                            <!--Определяем геометрию объекта-->
                            <GeometryModel3D.Geometry>
                                <MeshGeometry3D Positions="-0.5,0,0 0,0.5,0 0.5,0,0" TriangleInc
                            </GeometryModel3D.Geometry>
                            <!--Установка материала - красный цвет-->
                            <GeometryModel3D.Material>
                                <DiffuseMaterial Brush="Red" />
                            </GeometryModel3D.Material>
                        </GeometryModel3D>
                    </ModelVisual3D.Content>
                </ModelVisual3D>
            </Viewport3D.Children>
        </Viewport3D>
    </Grid>
</Window>
```

Фактически в этом коде мы можем выделить четыре узловые части:

1. Определение камеры. Без камеры мы не сможем увидеть объект
2. Определение источника света.
3. Определение самого трехмерного объекта
4. Определение материала (цвета) этого объекта

В итоге мы получим треугольник:



Нарисованный нами треугольник вряд ли можно назвать трехмерным, но на самом деле так и есть. Треугольник является примитивом, из которых состоят все остальные более сложные трехмерные элелементы. Например, квадрат состоит из двух треугольников. Из нескольких квадратов состоит куб. Также можно сферу и прочие объекты сложить из треугольников. Поэтому важно отработать всю механику трехмерных объектов WPF сначала на треугольниках, а потом переходить к более сложным объектам.

Среди свойств Viewport3D следует отметить свойство **ClipToBounds**, которое по умолчанию равно True. Из-за этого содержимое Viewport3D растягивается по горизонтали и вертикали, и если какая-то часть выходит за границы Viewport3D, то ее не видно. Если же свойство ClipToBounds, то трехмерный объект накладывается поверх смежных элементов. Данный эффект можно увидеть, если, например, задать отступ для Viewport3D:

```
<Grid>
    <Viewport3D ClipToBounds="False" Margin="0 40 0 0">
```

Теперь рассмотрим по отдельности все части определени трехмерной сцены.

Определение трехмерного объекта

Для создания трехмерного объекта мы можем использовать различные виды элементов:

3D-элемент	2D-аналог	Описание

3D-элемент	2D-аналог	Описание
Visual3D	Visual	Базовый класс для всех трехмерных объектов, которые визуализируются в Viewport3D. В частности, классы **ModelVisual3D** и **ModelUIElement3D**, объекты которых добавляются в Viewport3D через свойство Children, как раз наследуются от Visual3D
Model3D	Drawing	Представляет трехмерную модель, которая визуализируется элементом Visual3D
Geometry3D	Geometry	Абстрактный класс, который представляет трехмерную поверхность элемента. Для использования нам доступен его производный класс MeshGeometry3D
GeometryModel3D	GeometryDrawing	Класс, производный от Model3D. Объединяет объект Geometry3D, преставляющий поверхность 3D-объекта, и объект Material, определяющий материал для покрытия трехмерного объекта
Transform3D	Transform	Определяет трансформации 3D-объектов
UIElement3D	UIElement	Базовый класс для трехмерных объектов, которые могут вести себя как обычные элементы управления, например, реагировать на события

В прошлой теме для задания трехмерного объекта мы использовали следующую разметку xaml:

```
<ModelVisual3D>
    <ModelVisual3D.Content>
        <GeometryModel3D>

            <!--Определяем геометрию объекта-->
            <GeometryModel3D.Geometry>
                <MeshGeometry3D Positions="-0.5,0,0 0,0.5,0 0.5,0,0" TriangleIndices="0, 2,1" />
            </GeometryModel3D.Geometry>

            <!--Установка материала - красный цвет-->
            <GeometryModel3D.Material>
                <DiffuseMaterial Brush="Red" />
            </GeometryModel3D.Material>
        </GeometryModel3D>
    </ModelVisual3D.Content>
</ModelVisual3D>
```

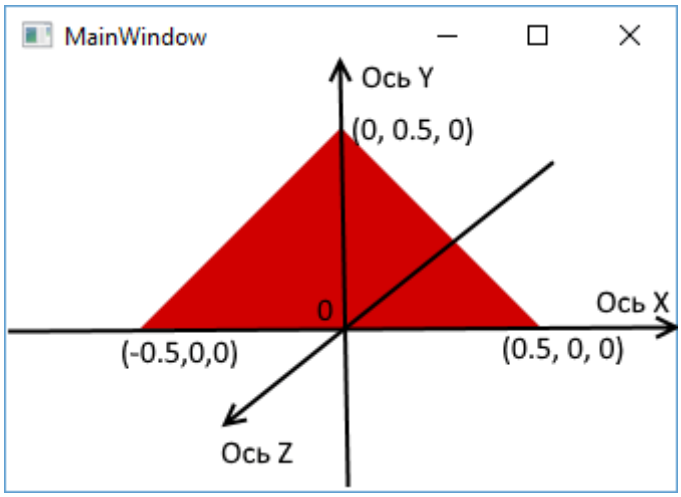
Объект **ModelVisual3D** предсталяет визуализируемый трехмерный объект. Для его установки используется свойство **Content**.

В качестве содержимого этого свойства устанавливается элемент **GeometryModel3D**. Этот элемент объединяет два аспекта: геометрию трехмерного объекта и его материал. Для определения материала применяется свойство **Material**, а для установки геометрии элемента - свойство **Geometry**.

В качестве геометрии устанавливается элемент **MeshGeometry3D**, который преставляет собой **меш** - один из фундаментальных концептов разработки трехмерных объектов. Для определения самого трехмерного объекта у MeshGeometry3D доступны следующие свойства:

- Positions: определяет координаты вершин треугольников. В нашем случае фигура представляет один треугольник, поэтому для его определения потребуется всего три точки. Для каждой точки определяется три координаты x, y, z.
- TriangleIndices: определяет треугольник, принимая набор верши. Например, TriangleIndices="0,2,1" - треугольник строиться от первой вершины, далее к третьей и далее ко второй, которая соединяется с первой.
- Normals: векторы нормалей, которые применяются для вычисления освещенности
- TextureCoordinates: определяет, как двухмерная текстура будет накладываться на трехмерный объект

Итак, чтобы нарисовать объект, мы должны задать его вершины. Для этого в объекте MeshGeometry3D мы устанавливаем свойство Positions. В случае одно треугольника нам нужны только три вершины, которые в координатной системе размещены следующим образом:



Затем с помощью свойства TriangleIndices мы устанавливаем, в какой последовательности эти вершины будут соединены линиями. В данном случае мы указываем, что сначала мы соединяем первую и третью вершины (точку -0.5,0,0 с точкой 0.5,0,0), потом от третьей вершины проводим линию до второй (до точки 0,0.5,0). Почему мы не написали TriangleIndices="0,1,2", что вроде было бы более логично? В WPF действует правило, согласно которому мы должны перечислять точки против часовой стрелки относительно оси Z, поэтому, если бы мы написали 0,1,2, то у нас бы ничего не получилось.

При этом надо отметить, что ось Z направлена на нас, то есть положительные значения после условного нуля будут ближе к нам, а отрицательные - дальше от нас.

Материалы

Кроме геометрии в объекте GeometryModel3D мы еще должны определить материал. Материалы бывают нескольких видов:

- DiffuseMaterial. Создает матовую поверхность объекта, на которой свет распространен равномерно во всех направлениях. Яркость же объекта зависит от угла, под которым свет падает на поверхность.
- SpecularMaterial. Имитирует блестящую поверхность, отражая свет зеркально в противоположном направлении. Зачастую он используется для наложения бликов на темной поверхности, к которой ранее применялся DiffuseMaterial

Для использования материала SpecularMaterial изменим код материала следующим образом:

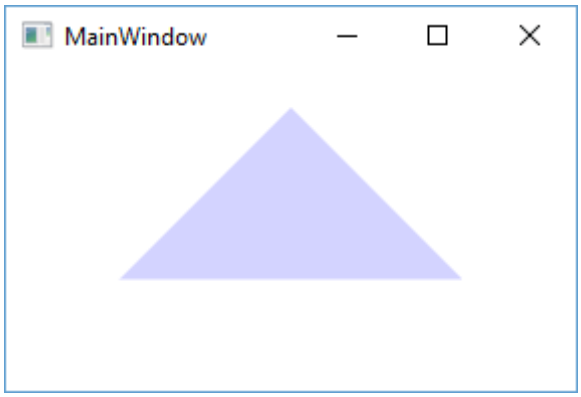
```
<GeometryModel3D.Material>
    <MaterialGroup>
        <DiffuseMaterial Brush="Black" />
        <SpecularMaterial Brush="LightBlue" SpecularPower="19" />
    </MaterialGroup>
</GeometryModel3D.Material>
```

Материал SpecularMaterial имеет свойство SpecularPower, которое позволяет задать размытость цвета. При большем значении цвет становится ярче, а при меньшем значении, наоборот, мы получаем более мягкий, более размытый эффект.

- EmissiveMaterial. Создает светящуюся поверхность.

```
<GeometryModel3D.Material>
    <MaterialGroup>
        <DiffuseMaterial Brush="DarkBlue" />
        <EmissiveMaterial Brush="LightGray" />
    </MaterialGroup>
</GeometryModel3D.Material>
```

При наложении различных цветов они смешиваются, создавая средний оттенок.



- MaterialGroup. Создает комбинацию нескольких материалов.

Кроме свойства Material в объекте GeometryModel3D определено свойство BackMaterial, которое настраивает внешний вид также с помощью материалов для задней поверхности предмета. Но поскольку в данном случае нас задняя поверхность треугольника не интересует, мы не используем это свойство. Но оно в принципе аналогично свойству Material.

Освещение

Установив геометрию и настроив внешний вид объекта, мы можем задать освещение. Без освещения мы бы не могли увидеть объект, кроме того, освещение также влияет на внешний вид, а именно на цвет объекта, наравне с материалами. В WPF система освещения призвана имитировать свет в окружающем нас мире. В то же время эта имитация все-таки далека от реальной. Так, свет отраженный от одного объекта, не будет виден на другом объекте. Также объекты не обрасывают на другие объекты тени.

Свет в WPF определяется следующим образом: сначала вычисляется освещенность для отдельной вершины треугольника, а затем эта освещенность интерполируется на весь треугольник, сглаживаясь с освещенностью двух других вершин.

Для определения света в WPF используется базовый класс Light, но в реальности мы будем работать с производными от него классами:

- DirectionalLight: создает прямые параллельные лучи, которые идут в направлении, указанном в свойстве Direction. В качестве примера подобного источника света можно привести солнце.
- AmbientLight: создает источник рассеянного света. Свет исходит со всех сторон и равномерно покрывает поверхность объекта
- PointLight: создает точечный источник света.
- SpotLight: создает источник, который испускает свет на трехмерную сцену по конусу.

DirectionalLight

Для установки освещения в прошлой теме мы использовали объект DirectionalLight:

```
<ModelVisual3D>
    <ModelVisual3D.Content>
        <DirectionalLight Color="White" Direction="-1,-1,-2" />
    </ModelVisual3D.Content>
</ModelVisual3D>
```

Использованный нами в примере источник освещения задает вектор направления света. Он начинается в точке (0,0,0) и направлен в точку (-1,-1,-2). При этом нам важен угол падения света, а не длина вектора, которая может быть и (-2,-2,-4), но будет означать то же самое, что и направление вектора (-1,-1,-2), поскольку оба вектора эквивалентны.

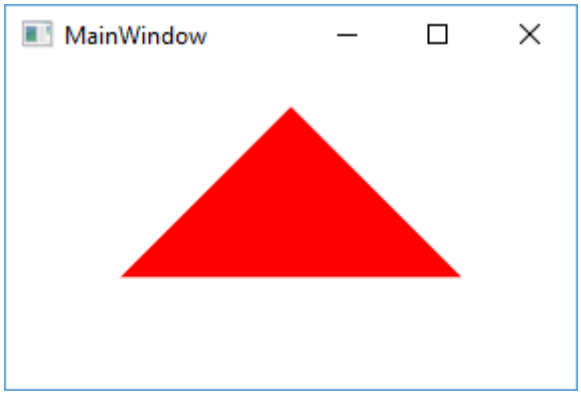
AmbientLight

Представляет рассеяный свет. Также широко распространенный источник света.

```
<AmbientLight Color="Yellow" />
```

Кроме того, для создания рассеянного света мы можем использовать свойство **AmbientColor** у объекта DiffuseMaterial:

```
<DiffuseMaterial Brush="LightCoral" AmbientColor="Yellow" />
```



PointLight

Для создания точечного света нам надо указать точку, в которой находится источник света:

```
<PointLight Color="White" Position="0,0.8,0.25" />
```

SpotLight

Данный источник света похож на PointLight, также лучи света направлены в одну точку, только теперь освещенность ограничивается углом, который определяется конусом:

```
<SpotLight Color="White" Position="1,0.2,0.5"
        Direction="-0.5,-0.5,-0.5" InnerConeAngle="45" OuterConeAngle="90" />
```

Свойства InnerConeAngle и OuterConeAngle определяют форму конуса, используемого для определения освещенности. Если эти свойства равны, то фактически мы получаем стандартный точечный свет PointLight. А свойства Position и Direction также указывают соответственно на положение источника света и направление света.

Камера

Теперь перейдем к последнему моменту в нашем приложении - к камере. От настроек камеры зависит, как трехмерная сцена будет отображаться на экране.

В WPF имеются три типа типа камер:

- PerspectiveCamera. Создает перспективную проекцию трехмерной сцены. В итоге те предметы, которые находятся дальше, выглядят меньше. А те, которые ближе, выглядят

больше - в принципе как и в реальной жизни.

- OrthographicCamera. Создает ортогональную проекцию сцены, в которой не применяется перспектива. И предметы сохраняют свои размеры.
- MatrixCamera. Позволяет задать матрицу преобразования трехмерной сцены в двухмерное содержимое для вывода на экран.

PerspectiveCamera и OrthographicCamera имеют следующие свойства, которые позволяют настроить камеру:

- Position. Задает точку трехмерного пространства, в котором находится камера. Например:

```
<Viewport3D>
    <Viewport3D.Camera>
        <PerspectiveCamera Position="0,0,2" LookDirection="0,0,-2" />
    </Viewport3D.Camera>

    <Viewport3D.Children>
        <ModelVisual3D>
            <ModelVisual3D.Content>
                <DirectionalLight Color="White" Direction="-1,-1,-2" />
            </ModelVisual3D.Content>
        </ModelVisual3D>
        <ModelVisual3D>
            <ModelVisual3D.Content>
                <GeometryModel3D>
                    <GeometryModel3D.Geometry>
                        <MeshGeometry3D Positions="-0.5,0,0 0,0.5,0 0.5,0,0" TriangleIndices="0,
                    </GeometryModel3D.Geometry>
                    <GeometryModel3D.Material>
                        <DiffuseMaterial Brush="Red" />
                    </GeometryModel3D.Material>
                </GeometryModel3D>
            </ModelVisual3D.Content>
        </ModelVisual3D>
    </Viewport3D.Children>
</Viewport3D>
```

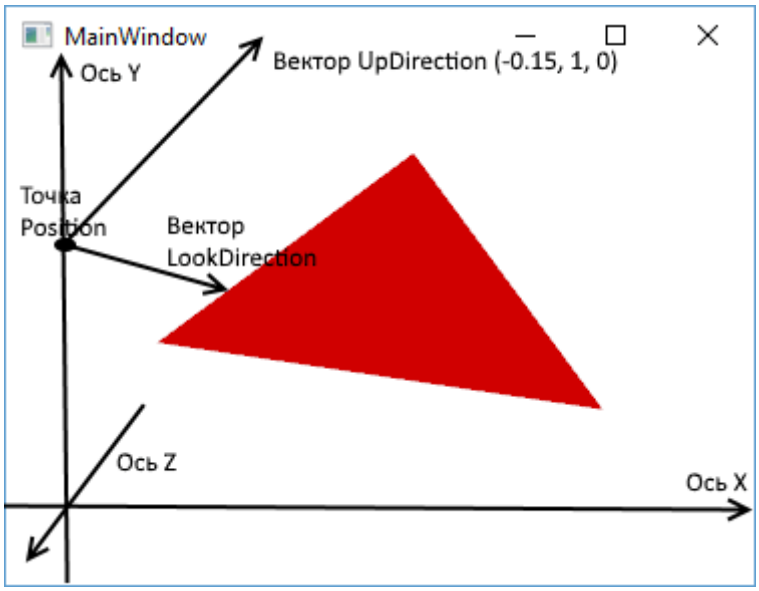
Здесь камера устанавливается в точку (0,0,2), то есть фактически находится впереди треугольника на 2 единицы - как-бы между нами и экраном монитора, на котором изображен

треугольник.

- LookDirection. Определяет вектор направления камеры. Чтобы правильно определить направление камеры, нам надо выбрать точку, на которую будет направлена камера. И затем из координат этой точки вычесть координаты в свойстве Position.

- UpDirection. Задает вектор вертикальной ориентации камеры, точнее ее наклон. В нашем случае данное свойство не было задано, поэтому оно приняло значения по умолчанию - (0,1,0). Данное значение указывает, что камера направлена прямо вверх. Но мы можем наклонить камеру немного по оси X, указав, например, следующее значение:

```
<PerspectiveCamera Position="0,0,2" LookDirection="0,0,-2" UpDirection="-0.15,1,0" />
```



- NearPlaneDistance. Это свойство задает переднюю плоскость отсечения. А все та часть сцены, которая находится к камере ближе значения этого свойства, отображаться не будет. По умолчанию имеет значение 0.125
- FarPlaneDistance. Это свойство задает заднюю плоскость отсечения. А все та часть сцены, которая находится к камере дальше значения этого свойства, отображаться не будет. По умолчанию имеет значение Double.PositiveInfinity

Кроме этих свойств камеры имеют еще свойство, которое задает ширину обзора или угол обзора. Только для различных камер оно называется по разному. Для PerspectiveCamera это свойство **FieldOfView**. Для OrthographicCamera это свойство **Width**. Чем больше значение этого свойства, тем угол обзора камеры.

Создание куба. Текстурирование

В предыдущем примере мы создали треугольник, который, конечно, нельзя считать подлинно трехмерной фигурой. Поэтому сейчас создадим куб.

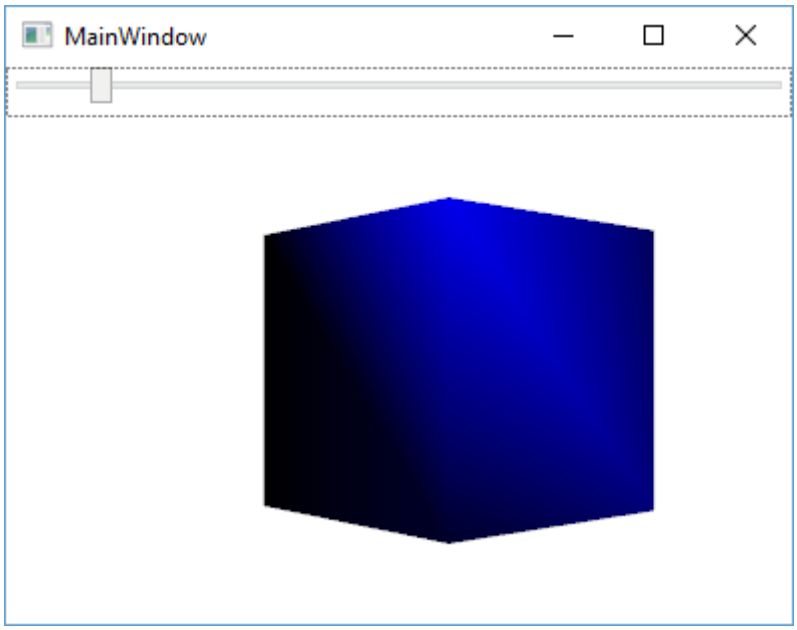
XAML код у нас будет выглядеть следующим образом:

```
<Window x:Class="_3DApp.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:_3DApp"
        mc:Ignorable="d"
        Title="MainWindow" Height="200" Width="300">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition />
        </Grid.RowDefinitions>
        <Slider Height="25" Minimum="0" Maximum="360"
                Value="{Binding ElementName=rotate, Path= Angle}" />
        <Viewport3D Grid.Row="1">
            <Viewport3D.Camera>
                <PerspectiveCamera Position="0.5,0.5,3.5" LookDirection="0,0,-3.5"  />
            </Viewport3D.Camera>
            <Viewport3D.Children>
                <ModelVisual3D>
                    <ModelVisual3D.Content>
                        <DirectionalLight Color="White" Direction="-1,-1,-2" />
                    </ModelVisual3D.Content>
                </ModelVisual3D>
                <ModelVisual3D>
                    <ModelVisual3D.Content>
                        <GeometryModel3D>
                            <GeometryModel3D.Geometry>
                                <MeshGeometry3D Positions="0,0,0 1,0,0 0,1,0 1,1,0
                      0,0,1 1,0,1 0,1,1 1,1,1"
 TriangleIndices="0,2,1 1,2,3 0,4,2 2,4,6
                  0,1,4 1,5,4 1,7,5 1,3,7
                  4,5,6 7,6,5 2,6,3 3,6,7"/>
                            </GeometryModel3D.Geometry>
                            <GeometryModel3D.Material>
                                <DiffuseMaterial Brush="Blue" />
                            </GeometryModel3D.Material>
                        </GeometryModel3D>
                    </ModelVisual3D.Content>
                    <ModelVisual3D.Transform>
                        <RotateTransform3D>
                            <RotateTransform3D.Rotation>
                                <AxisAngleRotation3D x:Name="rotate" Axis="0 1 0" />
                            </RotateTransform3D.Rotation>
                        </RotateTransform3D>
                    </ModelVisual3D.Transform>
                </ModelVisual3D>
            </Viewport3D.Children>
        </Viewport3D>
```

```
    </Grid>
</Window>
```

В отличие от ранее рассмотренного нами примера с треугольником мы сделали несколько изменений - во-первых, увеличилось количество вершин. Чтобы создать куб, нам потребовалось определить восемь вершин, которые составили 6 сторон куба. Каждая такая сторона состоит из двух треугольников, которые мы определили в свойстве TriangleIndices, то есть нам потребовалось определить 12 треугольников.

Второе отличие состоит в том, что мы определили трехмерную трансформацию - поворот вокруг оси Y. О трехмерных трансформациях мы погорим дальше, а пока мы создали элемент Slider поверх окна ViewPort3D, привязав его свойство Value к свойству Angle элемента AxisAngleRotation3D. И благодаря этому мы можем повращать куб вперед и назад.



Нормали и текстурирование

Говоря об освещении я сказал, что освещение вычисляется для каждой вершины, а затем цвет интерполируется на весь треугольник. Для треугольника это нормально. Но при создании куба вы можете увидеть, что на стороне куба в том месте, где находится совмещение треугольников (диагональ квадрата, образуемого двумя треугольниками), подобное освещение работает некорректно. И в этом месте мы можем увидеть полосу. Для сглаживания цветов в объекте **MeshGeometry3D** предназначено свойство **Normals**.

Это свойство содержит нормали для каждой вершины. Нормаль можно представить как вектор, перпендикулярный поверхности треугольника и определяющий, как вершина ориентирована относительно источника света.

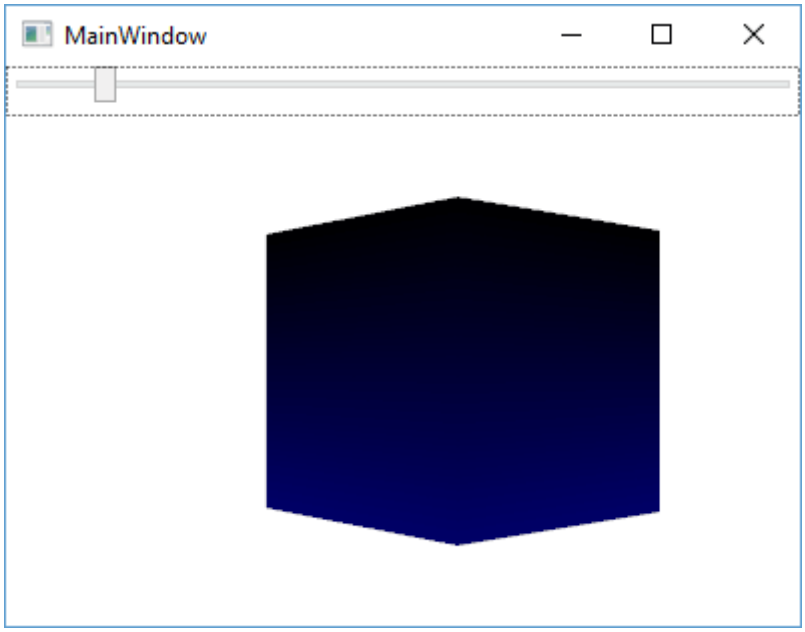
В математическом плане вектор нормали представляет векторное произведение векторов, составляющих две стороны треугольника. Для вычисления вектора нормалей можно воспользоваться средствами самого C#. Так, мы можем использовать следующий код:

```
private Vector3D CreateNormal(Point3D p0, Point3D p1, Point3D p2)
{
    Vector3D v0 = new Vector3D(p1.X - p0.X, p1.Y - p0.Y, p1.Z - p0.Z);
    Vector3D v1 = new Vector3D(p2.X - p1.X, p2.Y - p1.Y, p2.Z - p1.Z);
    return Vector3D.CrossProduct(v0, v1);
}
```

Затем полученные значения надо подставить в свойство Normals. Так, в нашем случае объект MeshGeometry3D будет выглядеть следующим образом:

```
<MeshGeometry3D Positions="0,0,0 1,0,0 0,1,0 1,1,0
                           0,0,1 1,0,1 0,1,1 1,1,1"
        TriangleIndices="0,2,1 1,2,3 0,4,2 2,4,6
                         0,1,4 1,5,4 1,7,5 1,3,7
                         4,5,6 7,6,5 2,6,3 3,6,7"
        Normals="0,1,0 0,1,0 1,0,0 1,0,0
                 0,1,0 0,1,0 1,0,0 1,0,0" />
```

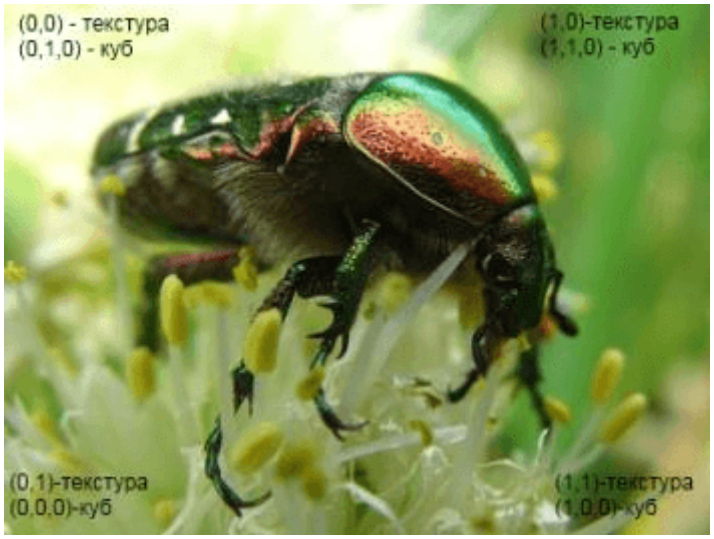
В итоге мы получим более сглаженное изображение куба:



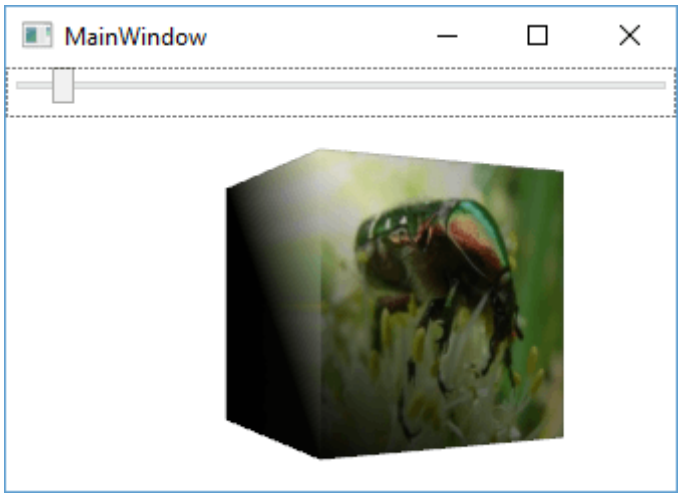
Процесс **текстурирования** трехмерных объектов предусматривает наложение на их поверхность изображений. Сначала мы в качестве кисти материала указываем ImageBrush, который принимает изображение. Затем сопоставляем координаты текстуры с вершинами объекта. Допустим, в файле bronz.jpg у нас находится изображение, тогда весь измененный код объекта GeometryModel3D будет выглядеть так:

```
<GeometryModel3D>
    <GeometryModel3D.Geometry>
        <MeshGeometry3D Positions="0,0,0 1,0,0 0,1,0 1,1,0
                                   0,0,1 1,0,1 0,1,1 1,1,1"
                TriangleIndices="0,2,1 1,2,3 0,4,2 2,4,6
                                 0,1,4 1,5,4 1,7,5 1,3,7
                                 4,5,6 7,6,5 2,6,3 3,6,7"
                TextureCoordinates="0,1 1,1 0,0 1,0
                                    0,1 1,1 0,0 1,0"/>
    </GeometryModel3D.Geometry>
    <GeometryModel3D.Material>
        <DiffuseMaterial>
            <DiffuseMaterial.Brush>
                <ImageBrush ImageSource="bronz.jpg" />
            </DiffuseMaterial.Brush>
        </DiffuseMaterial>
    </GeometryModel3D.Material>
</GeometryModel3D>
```

В данном случае у нас двухмерные координаты текстуры сопоставляются в координатами вершины. Для лицевой стороны куба наложение происходит так:



Правда, если вы запустите приложение, то увидите, что текстурированию подверглись только передняя и задняя стороны куба. Остальные стороны демонстрируют смазанный эффект.



Чтобы избежать подобного эффекта, нам надо определить по четыре разных точки для каждой стороны куба. То есть всего у нас получится 24 вершины, а не 8, как сейчас. После этого мы можем совместить с каждой точкой текстуры вершину, и изображение будет отображаться на каждой стороне куба, хотя объем кода при этом немного возрастет.

Средства разработки 3D объектов

С помощью одной Visual Studio очень не просто создавать сложные трехмерные объекты. Поэтому в данном случае лучше воспользоваться специальными программами. Одной из таких программ является Blender. Для экспорта трехмерных моделей из Blender в XAML можно использовать проект XAML Exporter for Blender.

Трехмерные трансформации и анимации

В прошлой теме в примере в кубом мы использовали вращение - связывали значение движения ползунка с углом поворота куба. Для трехмерных объектов доступны в принципе те же самые трансформации, что и для объектов двухмерных. Мы можем применить трансформацию как на уровне элемента **ModelVisual3D**, так и на уровне элемента **GeometryModel3D**.

Для вращения применяется трансформация **RotateTransform3D**:

```
<ModelVisual3D.Transform>
    <RotateTransform3D>
        <RotateTransform3D.Rotation>
            <AxisAngleRotation3D x:Name="rotate" Axis="0 1 0"  Angle="30" />
        </RotateTransform3D.Rotation>
    </RotateTransform3D>
</ModelVisual3D.Transform>
```

Свойство **Axis** задает ось вращения. Поскольку координаты X и Z имеют 0, куб будет вращаться только вокруг оси Y. Но мы можем вращать сразу вокруг нескольких осей как бы по-диагонали, установив свойство, например, так: Axis="1 1 0". А свойство **Angle** определяет угол, на который поворачивается объект. По умолчанию он равен 0.

Для перемещения применяется трансформация **TranslateTransform3D**:

```
<ModelVisual3D.Transform>
    <TranslateTransform3D x:Name="translate"  OffsetZ="0" />
</ModelVisual3D.Transform>
```

Код ползунка, устанавливающего перемещение по оси Z, мог бы выглядеть так:

Эта трансформация имеет свойства **OffsetX**, **OffsetY** и **OffsetZ**, устанавливающих перемещение объекта вдоль осей X,Y и Z соответственно.

Для масштабирования применяется трансформация **ScaleTransform3D**:

```
<ModelVisual3D.Transform>
    <ScaleTransform3D x:Name="scale" ScaleX="0.8" />
</ModelVisual3D.Transform>
```

Эта трансформация имеет свойства **ScaleX**, **ScaleY** и **ScaleZ**, позволяя масштабировать объект вдоль осей X,Y и Z соответственно.

Также имеется трансформация **MatrixTransform3D**, которая с помощью свойства **Matrix** позволяет задать трансформацию объекта.

Кроме того, мы можем комбинировать несколько трансформаций вместе, поместив их в объект **Transform3DGroup**.

```
<ModelVisual3D.Transform>
    <Transform3DGroup>
        <RotateTransform3D>
            <RotateTransform3D.Rotation>
                <AxisAngleRotation3D x:Name="rotate" Axis="0 1 0" Angle="30" />
            </RotateTransform3D.Rotation>
        </RotateTransform3D>
    </Transform3DGroup>
</ModelVisual3D.Transform>
```

Анимация трехмерных объектов

WPF предоставляет нам специальные классы для анимации в трехмерной сцене. Такие анимации могут быть полезны, например, если мы хотим вращать камеру вокруг 3D-объекта. В частности, WPF предоствляет специальные классы анимаций по ключевым кадрам, которые могут применяться к трехмерным структурам: **Point3DAnimationUsingKeyFrames** и **Vector3DAnimationUsingKeyFrames**:

```
<Window x:Class="_3DApp.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:_3DApp"
        mc:Ignorable="d"
        Title="MainWindow" Height="250" Width="350">
    <Window.Triggers>
        <EventTrigger RoutedEvent="Window.Loaded">
            <BeginStoryboard>
                <Storyboard>
                    <Point3DAnimationUsingKeyFrames
                        Storyboard.TargetName="camera"
                        Storyboard.TargetProperty="Position">
                        <LinearPoint3DKeyFrame Value="0,0.5,3.5" KeyTime="0:0:4"/>
                        <LinearPoint3DKeyFrame Value="-0.3,0.5,3.4" KeyTime="0:0:8"/>
                        <LinearPoint3DKeyFrame Value="-0.5,0.5,3.4" KeyTime="0:0:12"/>
                        <LinearPoint3DKeyFrame Value="-0.2,0.5,3.3" KeyTime="0:0:15"/>
                    </Point3DAnimationUsingKeyFrames>
                    <Vector3DAnimationUsingKeyFrames
                         Storyboard.TargetName="camera"
                        Storyboard.TargetProperty="LookDirection">
                        <LinearVector3DKeyFrame Value="0.1,0,-3.5" KeyTime="0:0:4"/>
                        <LinearVector3DKeyFrame Value="0.2,0,-3.5" KeyTime="0:0:10"/>
                        <LinearVector3DKeyFrame Value="0.3,0,-3.2" KeyTime="0:0:15"/>
                    </Vector3DAnimationUsingKeyFrames>
                </Storyboard>
            </BeginStoryboard>
        </EventTrigger>
    </Window.Triggers>
    <Grid>

        <Viewport3D>
            <Viewport3D.Camera>
                <PerspectiveCamera x:Name="camera" Position="0.5,0.5,3.5" LookDirection="0,0,-3.
            </Viewport3D.Camera>
            <Viewport3D.Children>
                <ModelVisual3D>
                    <ModelVisual3D.Content>
                        <DirectionalLight Color="White" Direction="-1,-1,-2" />
                    </ModelVisual3D.Content>
                </ModelVisual3D>
                <ModelVisual3D>
                    <ModelVisual3D.Content>
                        <GeometryModel3D>
                            <GeometryModel3D.Geometry>
                                <MeshGeometry3D Positions="0,0,0 1,0,0 0,1,0 1,1,0
                      0,0,1 1,0,1 0,1,1 1,1,1"
     TriangleIndices="0,2,1 1,2,3 0,4,2 2,4,6
```

```
                    0,1,4 1,5,4 1,7,5 1,3,7
                    4,5,6 7,6,5 2,6,3 3,6,7"/>
                        </GeometryModel3D.Geometry>
                        <GeometryModel3D.Material>
                            <DiffuseMaterial Brush="Blue" />
                        </GeometryModel3D.Material>
                    </GeometryModel3D>
                </ModelVisual3D.Content>
            </ModelVisual3D>
        </Viewport3D.Children>
    </Viewport3D>
</Grid>
</Window>
```

В данном случае при загрузке окна будут срабатываеть определенные в нем триггеры, которые будут запускать анимацию. Здесь определено две анимации по ключевым кадрам: Point3DAnimationUsingKeyFrames - для анимации свойства Position объекта камеры и Vector3DAnimationUsingKeyFrames - для анимации вектора LookDirection.

В итоге при запуске окна мы получим перемещение камеры относительно трехмерного объекта.

Взаимодействие с трехмерными объектами

ModelUIElement3D

Подобно классам двухмерных элементов в WPF имеются их трехмерные аналоги - Visual3D и UIElement3D. И мы можем использовать два их класса-наследника: **ModelUIElement3D** и **ContainerUIElement3D**. В чем их преимущество перед ModelVisual3D? Поскольку ModelUIElement3D является фактически своего рода элементом управления, мы можем прикрутить к нему события и, например, проверять нажатия мыши по нему. Иначе нам пришлось бы писать гораздо больше кода. Например, изменим код с кубом, применив ModelUIElement3D:

```
<Window x:Class="_3DApp.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:_3DApp"
        mc:Ignorable="d"
        Title="MainWindow" Height="250" Width="350">
    <Grid>

        <Viewport3D>
            <Viewport3D.Camera>
                <PerspectiveCamera x:Name="camera" Position="0.5,0.5,3.5" LookDirection="0,0,-3.
            </Viewport3D.Camera>
            <Viewport3D.Children>
                <ModelVisual3D>
                    <ModelVisual3D.Content>
                        <DirectionalLight Color="White" Direction="-1,-1,-2" />
                    </ModelVisual3D.Content>
                </ModelVisual3D>
                <ModelUIElement3D MouseDown="ModelUIElement3D_MouseDown">
                    <ModelUIElement3D.Model>
                        <GeometryModel3D>
                            <GeometryModel3D.Geometry>
                                <MeshGeometry3D Positions="0,0,0 1,0,0 0,1,0 1,1,0
                      0,0,1 1,0,1 0,1,1 1,1,1"
     TriangleIndices="0,2,1 1,2,3 0,4,2 2,4,6
                      0,1,4 1,5,4 1,7,5 1,3,7
                      4,5,6 7,6,5 2,6,3 3,6,7"/>
                            </GeometryModel3D.Geometry>
                            <GeometryModel3D.Material>
                                <DiffuseMaterial Brush="Blue" />
                            </GeometryModel3D.Material>
                        </GeometryModel3D>
                    </ModelUIElement3D.Model>
                    <ModelUIElement3D.Transform>
                        <Transform3DGroup>
                            <RotateTransform3D>
                                <RotateTransform3D.Rotation>
                                    <AxisAngleRotation3D x:Name="rotate" Axis="0 1 0" Angle="30"
                                </RotateTransform3D.Rotation>
                            </RotateTransform3D>
                        </Transform3DGroup>
                    </ModelUIElement3D.Transform>
                </ModelUIElement3D>
            </Viewport3D.Children>
        </Viewport3D>
    </Grid>
</Window>
```

Таким образом, мы связываем нажатие по кубу с обработчиком события MouseDown, в котором затем мы можем определить все необходимые действия:

```
private void ModelUIElement3D_MouseDown(object sender, MouseButtonEventArgs e)
{
    MessageBox.Show("Произошло нажатие");
}
```

Элемент ContainerUIElement3D является контейнером для других элементов Visual3D, то есть он может вмещать несколько различных элементов ModelUIElement3D, ModelVisual3D и Viewport2DVisual3D

Viewport2DVisual3D

Элемент Viewport2DVisual3D позволяет помещать на поверхности трехмерного объекта двухмерный интерактивный контент, например, кнопки или текстовые поля. При этом данный интерактивный контент может реагировать на действия пользователя. Например, кнопка может реагировать на нажатия.

У элемента Viewport2DVisual3D можно выделить следующие свойства:

- Geometry: определяет меш, который устанавливает поверхность трехмерного объекта
- Visual: определяет двухмерные элементы, которые будут помещаться на трехмерный объект
- Material: материал, используемый для рендеринга двухмерных элементов
- Transform: трасформация, применяемая к 3D-объекту

Изменим приложение, применив Viewport2DVisual3D:

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition />
    </Grid.RowDefinitions>
    <Slider Minimum="0" Maximum="360" Value="{Binding ElementName=rotate, Path= Angle}" />
    <Viewport3D Grid.Row="1">
        <Viewport3D.Camera>
            <PerspectiveCamera x:Name="camera" Position="0.5,0.5,3.5" LookDirection="0,0,-3.5"
        </Viewport3D.Camera>
        <Viewport3D.Children>
            <ModelVisual3D>
                <ModelVisual3D.Content>
                    <DirectionalLight Color="White" Direction="-1,-1,-2" />
                </ModelVisual3D.Content>
            </ModelVisual3D>
            <Viewport2DVisual3D>

                <Viewport2DVisual3D.Geometry>
                    <MeshGeometry3D Positions="0,0,0 1,0,0 0,1,0 1,1,0
          0,0,1 1,0,1 0,1,1 1,1,1"
                                    TriangleIndices="0,2,1 1,2,3 0,4,2 2,4,6
                0,1,4 1,5,4 1,7,5 1,3,7
                4,5,6 7,6,5 2,6,3 3,6,7"
                                    TextureCoordinates="0,1 1,1 0,0 1,0
                   0,1 1,1 0,0 1,0"/>
                </Viewport2DVisual3D.Geometry>

                <Viewport2DVisual3D.Material>
                    <DiffuseMaterial Viewport2DVisual3D.IsVisualHostMaterial="True" />
                </Viewport2DVisual3D.Material>

                <Viewport2DVisual3D.Visual>
                    <Border BorderBrush="Black" BorderThickness="1">
                        <StackPanel>
                            <Button Content="Hello World" />
                            <TextBox>Введи текст</TextBox>
                        </StackPanel>
                    </Border>
                </Viewport2DVisual3D.Visual>

                <Viewport2DVisual3D.Transform>
                    <RotateTransform3D>
                        <RotateTransform3D.Rotation>
                            <AxisAngleRotation3D x:Name="rotate" Axis="0 1 0" Angle="30" />
                        </RotateTransform3D.Rotation>
                    </RotateTransform3D>
                </Viewport2DVisual3D.Transform>
            </Viewport2DVisual3D>
        </Viewport3D.Children>
```

```
    </Viewport3D>
</Grid>
```