Раздел 1. Приложения Windows Presentation Foundation(лекции 1-4)

Лекция 1. Введение. Технология WPF (Windows Presentation Foundation)

Технология WPF (Windows Presentation Foundation) является часть экосистемы платформы .NET и представляет собой подсистему для построения графических интерфейсов.

Если при создании традиционных приложений на основе WinForms за отрисовку элементов управления и графики отвечали такие части ОС Windows, как User32 и GDI+, то приложения WPF основаны на **DirectX**. В этом состоит ключевая особенность рендеринга графики в WPF: используя WPF, значительная часть работы по отрисовке графики, как простейших кнопочек, так и сложных 3D-моделей, ложиться на графический процессор на видеокарте, что также позволяет воспользоваться аппаратным ускорением графики.

Одной из важных особенностей является использование языка декларативной разметки интерфейса XAML, основанного на XML: вы можете создавать насыщенный графический интерфейс, используя или декларативное объявление интерфейса, или код на управляемых языках C# и VB.NET, либо совмещать и то, и другое.

Преимущества WPF

Что вам, как разработчику, предлагает WPF?

- Использование традиционных языков .NET-платформы - C# и VB.NET для создания логики приложения
- Возможность декларативного определения графического интерфейса с помощью специального языка разметки XAML, основанном на xml и представляющем альтернативу программному созданию графики и элементов управления, а также возможность комбинировать XAML и C#/VB.NET
- Независимость от разрешения экрана: поскольку в WPF все элементы измеряются в независимых от устройства единицах, приложения на WPF легко масштабируются под разные экраны с разным разрешением.
- Новые возможности, которых сложно было достичь в WinForms, например, создание трехмерных моделей, привязка данных, использование таких элементов, как стили, шаблоны, темы и др.
- Хорошее взаимодействие с WinForms, благодаря чему, например, в приложениях WPF можно использовать традиционные элементы управления из WinForms.
- Богатые возможности по созданию различных приложений: это и мультимедиа, и двухмерная и трехмерная графика, и богатый набор встроенных элементов управления, а также возможность самим создавать новые элементы, создание анимаций, привязка данных, стили, шаблоны, темы и многое другое

- Аппаратное ускорение графики - вне зависимости от того, работаете ли вы с 2D или 3D, графикой или текстом, все компоненты приложения транслируются в объекты, понятные Direct3D, и затем визуализируются с помощью процессора на видеокарте, что повышает производительность, делает графику более плавной.
- Создание приложений под множество ОС семейства Windows - от Windows XP до Windows 10

В тоже время WPF имеет определенные ограничения. Несмотря на поддержку трехмерной визуализации, для создания приложений с большим количеством трехмерных изображений, прежде всего игр, лучше использовать другие средства - DirectX или специальные фреймворки, такие как Monogame или Unity.

Также стоит учитывать, что по сравнению с приложениями на Windows Forms объем программ на WPF и потребление ими памяти в процессе работы в среднем несколько выше. Но это с лихвой компенсируется более широкими графическими возможностями и провышенной производительностью при отрисовке графики.

Архитектура WPF

Схематически архитектуру WPF можно представить следующим образом:

Как видно на схеме, WPF разбивается на два уровня: managed API и unmanaged API (уровень интеграции с DirectX). Managed API (управляемый API-интерфейс) содержит код, исполняемый под управлением общеязыковой среды выполнения .NET - Common Language Runtime. Этот API описывает основной функционал платформы WPF и состоит из следующих компонентов:

- PresentationFramework.dll: содержит все основные реализации компонентов и элементов управления, которые можно используются при построении графического интерфейса
- PresentationCore.dll: содержит все базовые типы для большинства классов из PresentationFramework.dll
- WindowsBase.dll: содержит ряд вспомогательных классов, которые применяются в WPF, но могут также использоваться и вне данной платформы

Unmanaged API используется для интеграции вышележащего уровня с DirectX:

- milcore.dll: собственно обеспечивает интеграцию компонентов WPF с DirectX. Данный компонент написан на неуправляемом коде (C/C++) для взаимодействия с DirectX.
- WindowsCodecs.dll: библиотека, которая предоставляет низкоуровневую поддержку для изображений в WPF

Еще ниже собственно находятся компоненты операционной системы и DirectX, которые производят визуализацию компонентов приложения, либо выполняют прочую низкоуровневую обработку. В частности, с помощью низкоуровневого интерфейса Direct3D, который входит в состав DirectX, происходит трансляция

Здесь также на одном уровне находится библиотека user32.dll. И хотя выше говорилось, что WPF не использует эту библиотеку для рендеринга и визуализации, однако для ряда вычислительных задач (не включающих визуализацию) данная библиотека продолжает использоваться.

История развития

WPF является частью экосистемы .NET и развивается вместе с фреймворком .NET и имеет те же версии. Первая версия WPF 3.0 вышла вместе с .NET 3.0 и операционной системой Windows Vista в 2006 году. С тех пор платформа последовательно развивается. Последняя версия WPF 4.6 вышла параллельно с .NET 4.6 в июле 2015 года, ознаменовав девятилетие данной платформы.

Для создания приложений с помощью технологии WPF нам потребуется среда разработки Visual Studio. Я в дальнейшем буду использовать бесплатный полнофункциональный выпуск **Visual Studio 2015 Community**. Данный выпуск можно найти на странице Visual Studio 2015.

Итак, откроем Visual Studio Express 2015 и в меню File (Файл) выберем пункт New (Создать) -> Project... (Проект...). Перед нами откроется диалоговое окно создания проекта, в котором мы выберем шаблон WPF Application:

Укажем проекту какое-нибудь имя и нажмем кнопку OK. И Visual Studio создаст нам новый проект

По умолчанию студия открывает создает и открывает нам два файла: файл декларативной разметки интерфейса *MainWindow.xaml* и файл связанного с разметкой кода *MainWindow.xaml.cs*. Файл MainWindow.xaml имеет два представления: визуальное - в режиме WYSIWIG отображает весь графический интерфейс данного окна приложения, и под ним декларативное объявление интерфейса в XAML. Если мы изменим декларативную разметку, например, определим там кнопку, то эти изменения отображаться в визуальном представлении.

Структура проекта

В структуре проекта WPF следует выделить следующие моменты. Во-первых, в проекте имеется файл *App.xaml* и связанный с ним файл кода *App.xaml.cs* - это глобальные файлы для всего приложения, позже мы о них поговорим подробнее. А пока только следует знать, что App.xaml задает файл окна программы, которое будет открываться при запуске приложения. Если вы откроете этот файл, то можете найти в нем строку `StartupUri="MainWindow.xaml"` - то есть в данном случае, когда мы запустим приложение, будет создаваться интерфейс из файла MainWindow.xaml.

Далее в структуре определены файл разметки *MainWindow.xaml* и файл связанного кода *MainWindow.xaml.cs*. Файл MainWindow.xaml и представляет определение окна приложение, которое мы увидим при запуске.

Лекция 2. XAML

Введение в язык XAML

XAML (eXtensible Application Markup Language) - язык разметки, используемый для инициализации объектов в технологиях на платформе .NET. Применительно к WPF (а также к Silverlight) данный язык используется прежде всего для создания пользовательского интерфейса декларативным путем. Хотя функциональность XAML только графическими интерфейсами не ограничивается: данный язык также используется в технологиях WCF и WF, где он никак не связан с графическим интерфейсом. То есть его область шире. Применительно к WPF мы будем говорить о нем чаще всего именно как о языке разметки, который позволяет создавать декларативным путем интерфейс, наподобие HTML в веб-программировании. Однако опять же повторюсь, сводить XAML к одному интерфейсу было бы неправильно, и далее на примерах мы это увидим.

XAML - не является обязательной частью приложения, мы вобще можем обходиться без него, создавая все элементы в файле связанного с ним кода на языке C#. Однако использование XAML все-таки несет некоторые преимущества:

- Возможность отделить графический интерфейс от логики приложения, благодаря чему над разными частями приложения могут относительно автономно работать разные специалисты: над интерфейсом - дизайнеры, над кодом логики - программисты.
- Компактность, понятность, код на XAML относительно легко поддерживать.

При компиляции приложения в Visual Studio код в xaml-файлах также компилируется в бинарное представление кода xaml, которое называется BAML (Binary Application Markup Language). И затем код baml встраивается в финальную сборку приложения - exe или dll-файл.

Структура и пространства имен XAML

При создании нового проекта WPF он уже содержит файлы с кодом xaml. Так, создаваемый по умолчанию в проекте файл *MainWindow.xaml* будет иметь следующую разметку:

```
<Window x:Class="XamlApp.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:XamlApp"

        mc:Ignorable="d"

        Title="MainWindow" Height="350" Width="525">

    <Grid>



    </Grid>

</Window>
```

Если вы совершенно не знакомы с xaml и с xml, то даже этот небольшой минимальный код окна может вызывать затруднения.

Подобно структуре веб-страничке на html, здесь есть некоторая иерархия элементов. Элементов верхнего уровня является `Window`, который представляет собой окно приложения. При создании других окон в приложении нам придется всегда начинать объявление интерфейса с элемента Window, поскольку это элемент самого верхнего уровня.

Кроме Window существует еще два элемента верхнего уровня:

- Page
- Application

Элемент Window имеет вложенный пустой элемент `Grid`, а также подобно html-элементам ряд атрибутов (Title, Width, Height) - они задают заголовок, ширину и высоту окна соответственно.

Пространства имен XAML

При создании кода на языке C#, чтобы нам были доступны определенные классы, мы подключаем пространства имен с помощью директивы `using`, например, `using System.Windows;`.

Чтобы задействовать элементы в XAML, мы также подключаем пространства имен. Вторая и третья строчки как раз и представляют собой пространства имен, подключаемые в проект по умолчанию. А атрибут xmlns представляет специальный атрибут для определения пространства имен в XML.

Так, пространство имен http://schemas.microsoft.com/winfx/2006/xaml/presentation содержит описание и определение большинства элементов управления. Так как является пространством имен по умолчанию, то объявляется без всяких префиксов.

http://schemas.microsoft.com/winfx/2006/xaml - это пространство имен, которое определяет некоторые свойства XAML, например свойство Name или Key. Используемый префикс x в определении `xmlns:x` означает, что те свойства элементов, которые заключены в этом пространстве имен, будут использоваться с префиксом x - `x:Name` или `x:Key`. Это же пространство имен используется уже в первой строчке x:Class="XamlApp.MainWindow" - здесь создается новый класс MainWindow и соответствующий ему файл кода, куда будет прописываться логика для данного окна приложения.

Это два основных пространства имен. Рассмотрим остальные:

- xmlns:d="http://schemas.microsoft.com/expression/blend/2008": предоставляет поддержку атрибутов в режиме дизайнера. Это пространство имен преимущественно предназначено для другого инструмента по созданию дизайна на XAML - Microsoft Expression Blend
- xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006": обеспечивает режим совместимости разметок XAML. В определении объекта Window двумя строчками ниже можно найти его применение:

  ```
  mc:Ignorable="d"
  ```

  Это выражение позволяет игнорировать парсерам XAML во время выполнения приложения дизайнерские атрибуты из пространства имен с префиксом d, то есть из "http://schemas.microsoft.com/expression/blend/2008"

- xmlns:local="clr-namespace:XamlApp": пространство имен текущего проекта. Так как в моем случае проект называется XamlApp, то пространство имен называется аналогично. И через префикс *local* я смогу получить в XAML различные объекты, которые я определил в проекте.

Важно понимать, что эти пространства имен не эквивалентны тем пространствам имен, которые подключаются при помощи директивы `using` в c#. Так, например, http://schemas.microsoft.com/winfx/2006/xaml/presentation подключает в проект следующие пространства имен:

- System.Windows
- System.Windows.Automation
- System.Windows.Controls
- System.Windows.Controls.Primitives
- System.Windows.Data
- System.Windows.Documents
- System.Windows.Forms.Integration
- System.Windows.Ink
- System.Windows.Input
- System.Windows.Media
- System.Windows.Media.Animation
- System.Windows.Media.Effects
- System.Windows.Media.Imaging
- System.Windows.Media.Media3D
- System.Windows.Media.TextFormatting
- System.Windows.Navigation
- System.Windows.Shapes
- System.Windows.Shell

Элементы и их атрибуты

XAML предлагает очень простую и ясную схему определения различных элементов и их свойств. Каждый элемент, как и любой элемент XML, должен иметь открытый и закрытый тег, как в случае с элементом Window:

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

Либо элемент может иметь сокращенную форму с закрывающим слешем в конце, наподобие:

```
<Window />
```

Но в отличие от элементов xml каждый элемент в XAML соответствует определенному классу C#. Например, элемент Button соответствует классу `System.Windows.Controls.Button`. А свойства этого класса соответствуют атрибутам элемента Button.

Например, добавим кнопку в создаваемую по умолчанию разметку окна:

```
<Window x:Class="XamlApp.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:XamlApp"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <Grid x:Name="grid1">
        <Button x:Name="button1" Width="100" Height="30" Content="Кнопка" />
    </Grid>
</Window>
```

Сначала идет элемент самого высшего уровня - Window, затем идет вложенный элемент Grid - контейнер для других элементов, и в нем уже определен элемент Button, представляющий кнопку.

Для кнопки мы можем определить свойства в виде атрибутов. Здесь определены атрибуты `x:Name` (имя кнопки), `Width`, `Height` и `Content`. Причем, атрибут `x:Name` берется в данном случае из пространства имен "http://schemas.microsoft.com/winfx/2006/xaml", которое сопоставляется с префиксом x. А остальные атрибуты не используют префиксы, поэтому берутся из основного пространства имен "http://schemas.microsoft.com/winfx/2006/xaml/presentation".

Подобным образом мы можем определить и другие атрибуты, которые нам нужны. Либо мы в общем можем не определять атрибуты, и тогда они будут использовать значения по умолчанию.

Определив разметку xaml, мы можем запустить проект, и нам отобразится графически весь код xaml - то есть наша кнопка:

Специальные символы

При определении интерфейса в XAML мы можем столкнуться с некоторыми ограничениями. В частности, мы не можем использовать специальные символы, такие как знак амперсанда &, кавычки " и угловые скобки < и >. Например, мы хотим, чтобы текст кнопки был следующим: `<"Hello">`. У кнопки есть свойство `Content`, которое задает содержимое кнопки. И можно предположить, что нам надо написать так:

```
<Button Content="<"Hello">" />
```

Но такой вариант ошибочен и даже не скомпилируется. В этом случае нам надо использовать специальные коды символов:

Символ	Код
<	<
>	>
&	&
"	"

Например:

```
1<Button Content="&lt;&quot;Hello&quot;&gt;" />
```

Еще одна проблема, с которой мы можем столкнуться в XAML - добавление пробелов. Возьмем, к примеру, следующее определение кнопки:

```
<Button> Hello  World</Button>
```

Здесь свойство Content задается неявно в виде содержимого между тегами `<Button>....</Button>`. Но несмотря на то, что у нас несколько пробелов между словами "Hello" и "World", XAML по умолчанию будет убирать все эти пробелы. И чтобы сохранить пробелы, нам надо использовать атрибут `xml:space="preserve"`:

```
<Button xml:space="preserve">
    Hello         World
</Button>
```

Лекция 3. Файлы отделенного кода

При создании нового проекта WPF в дополнение к создаваемому файлу *MainWindow.xaml* создается также файл отделенного кода *MainWindow.xaml.cs*, где, как предполагается, должна находится логика приложения связанная с разметкой из MainWindow.xaml. Файлы XAML позволяют нам определить интерфейс окна, но для создания логики приложения,

например, для определения обработчиков событий элементов управления, нам все равно придется воспользоваться кодом C#.

По умолчанию в разметке окна используется атрибут x:Class:

```
<Window x:Class="XamlApp.MainWindow"

.......
```

Атрибут x:Class указывает на класс, который будет представлять данное окно и в который будет компилироваться код в XAML при компиляции. То есть во время компиляции будет генерироваться класс XamlApp.MainWindow, унаследованный от класса `System.Windows.Window`.

Кроме того в файле отделенного кода *MainWindow.xaml.cs*, который Visual Studio создает автоматически, мы также можем найти класс с тем же именем - в данном случае класс XamlApp.MainWindow. По умолчанию он имеет некоторый код:

```
using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows;

using System.Windows.Controls;

using System.Windows.Data;

using System.Windows.Documents;

using System.Windows.Input;

using System.Windows.Media;

using System.Windows.Media.Imaging;

using System.Windows.Navigation;

using System.Windows.Shapes;
```

```
namespace XamlApp
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }
    }
}
```

По сути пустой класс, но этот класс уже выполняет некоторую работу. Во время компиляции этот класс объединяется с классом, сгенерированным из кода XAML. Чтобы такое слияние классов во время компиляции произошло, класс XamlApp.MainWindow определяется как частичный с модификатором partial. А через метод `InitializeComponent()` класс MainWindow вызывает скомпилированный ранее код XAML, разбирает его и по нему строит графический интерфейс окна.

Взаимодействие кода C# и XAML

В приложении часто требуется обратиться к какому-нибудь элементу управления. Для этого надо установить у элемента в XAML свойство `Name`.

Еще одной точкой взаимодействия между xaml и C# являются события. С помощью атрибутов в XAML мы можем задать события, которые будут связанны с обработчиками в коде C#.

Итак, создадим новый проект WPF, который назовем XamlApp. В разметке главного окна определим два элемента: кнопку и текстовое поле.

?

```
1 <Window x:Class="XamlApp.MainWindow"
2         xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3         xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4         xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
          xmlns:mc="http://schemas.openxmlformats.org/markup-
```

```
5 compatibility/2006"
6         xmlns:local="clr-namespace:XamlApp"
7         mc:Ignorable="d"
8         Title="MainWindow" Height="350" Width="525">
9     <Grid x:Name="grid1">
10        <TextBox x:Name="textBox1" Width="150" Height="30"
VerticalAlignment="Top" Margin="20" />
11        <Button x:Name="button1" Width="100" Height="30" Content="Кнопка"
  Click="Button_Click" />
12    </Grid>
13 </Window>
```

И изменим файл отделенного кода, добавив в него обработчик нажатия кнопки:

```
?
1 using System.Windows;
2
3 namespace XamlApp
4 {
5     public partial class MainWindow : Window
6     {
7         public MainWindow()
8         {
9             InitializeComponent();
10        }
11
11        private void Button_Click(object sender, RoutedEventArgs e)
12        {
13            string text = textBox1.Text;
```

```
14              if (text != "")
15              {
16                  MessageBox.Show(text);
17              }
18          }
19      }
20  }
21
```

Определив имена элементов в XAML, затем мы можем к ним обращаться в коде c#: `string text = textBox1.Text.`

При определении имен в XAML надо учитывать, что оба пространства имен "http://schemas.microsoft.com/winfx/2006/xaml/presentation" и "http://schemas.microsoft.com/winfx/2006/xaml" определяют атрибут Name, который устанавливает имя элемента. Во втором случае атрибут используется с префиксом x: `x:Name`. Какое именно пространство имен использовать в данном случае, не столь важно, а следующие определения имени `x:Name="button1"` и `Name="button1"` фактически будут равноценны.

В обработчике нажатия кнопки просто выводится сообщение , введенное в текстовое поле. После определения обработчика мы его можем связать с событием нажатия кнопки в xaml через атрибут Click: `Click="Button_Click"`. В результате после нажатия на кнопку мы увидим в окне введенное в текстовое поле сообщение.

Лекция 4. Создание элементов в коде C#

Еще одну форму взаимодействия C# и XAML представляет создание визуальных элементов в коде C#. Например, изменим код xaml следующим образом:

?

```
1 <Window x:Class="XamlApp.MainWindow"
2         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:XamlApp"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <Grid x:Name="layoutGrid">

    </Grid>
</Window>
```

Здесь для элемента Grid установлено свойство `x:Name`, через которое мы можем к нему обращаться в коде. И также изменим код C#:

```
using System.Windows;
using System.Windows.Controls;

namespace XamlApp
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();

            Button myButton = new Button();
            myButton.Width = 100;
            myButton.Height = 30;
```

```
14              myButton.Content = "Кнопка";
15              layoutGrid.Children.Add(myButton);
16          }
17      }
18  }
19
```

В конструкторе страницы создается элемент Button и добавляется в Grid. И если мы запустим приложение, то увидим добавленную кнопку:

or.FromRgb.