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#, VB.NET и F#, либо совмещать и то, и другое.

Первая версия - WPF 3.0 вышла вместе с .NET Framework 3.0 и операционной системой Windows Vista в 2006 году. И с тех пор платформа WPF является частью экосистемы .NET и развивается вместе с фреймворком .NET. Например, на сегодняшний день последней версией фреймворка .NET является .NET 8, и WPF полностью поддерживается этой версией фреймворка.

**Возможности WPF**

Что вам, как разработчику, предлагает WPF?

* Использование традиционных языков .NET-платформы - C#, F# 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

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 повышенной производительностью при отрисовке графики.

Кроме того, несмотря на то, что WPF работает поверх кроссплатформенной среды .NET, но в силу природы WPF и зависимости от компонентов Windows, на данный момент создавать приложения на WPF можно только под ОС Windows.

**Эволюция графики в Windows**

Стандартное Windows-приложение при создании пользовательского интерфейса полагается на две основополагающие части операционной системы Windows:

User32

Обеспечивает знакомый внешний вид и поведение таких элементов, как окна, кнопки, текстовые поля и т.п.;

GDI/GDI+

Предоставляет поддержку рисования фигур, текста и изображений за счет дополнительного усложнения (и часто неважной производительности).

С годами обе технологии совершенствовались, и API-интерфейсы, используемые разработчиками для взаимодействия с ними, значительно менялись. Но как бы ни разрабатывалось приложение — с помощью .NET и Windows Forms, (в прошлом) Visual Basic 6 или кода С++ на основе MFC — "за кулисами" работают одни и те же части операционной системы Windows. Новые платформы просто предоставляют улучшенные оболочки для взаимодействия с User32 и GDI/GDI+.

**Использование DirectX**

В Microsoft разработали один обходной путь для преодоления ограничений, присущих библиотекам User32 и GDI/GDI+. Этим путем является DirectX. Он начинался как "топорный", полный ошибок инструментальный набор для создания игр на платформе Windows. Главной его целью была скорость, и потому Microsoft тесно сотрудничала с производителями видеокарт, чтобы обеспечить для DirectX аппаратную поддержку, необходимую для отображения сложных текстур, сехнология WPF в корне меняет ситуацию.

Лежащая в основе WPF графическая технология — это не GDI/GDI+. Теперь это DirectX. Примечательно, что приложения WPF используют DirectX независимо от создаваемого типа пользовательского интерфейса. Это значит, что создается ли сложная трехмерная графика (DirectX's forte), или просто рисуются кнопки и простой текст — вся работа по рисованию проходит через конвейер DirectX. В результате даже самые заурядные бизнес-приложения могут использовать богатые эффекты вроде прозрачности и сглаживания. Также получается выигрыш от аппаратного ускорения, и это означает, что DirectX передает как можно больше работы ***узлу обработки графики (graphics processing unit — GPU)***, который представляет собой отдельный процессор на видеокарте.пециальных эффектов вроде частичной прозрачности и трехмерной графики.

Один компонент, который остается на сцене (в ограниченной степени) — это User32. Это объясняется тем, что WPF по-прежнему полагается на User32 в отношении таких служб, как обработка и маршрутизация ввода, а также определение того, какое приложение какой частью экрана владеет. Однако все рисование производится через DirectX.

Это наиболее существенное изменение в WPF. Технология WPF — это не оболочка для GDI/GDI+. На самом деле это его замена — отдельный уровень, работающий через DirectX.

**Аппаратное ускорение и WPF**

Возможно, вам известно, что видеокарты различаются между собой в плане поддержки специализированных средств визуализации и оптимизации. К счастью, проблемой это не является, поскольку WPF обладает способностью выполнять всю работу с использованием программных вычислений вместо того, чтобы полагаться на встроенную поддержку видеокарты.

В отношении программной поддержки WPF существует одно исключение. Из-за слабой поддержки драйверов WPF выполняет сглаживание трехмерной графики только в случае, если приложение запущено под управлением Windows Vista или Windows 7 (и есть встроенный драйвер WDDM для установленной видеокарты).

Это значит, что при рисовании трехмерных фигур на компьютере с Windows ХР вместо гладких линий будут получены ступенчатые ломаные. Однако для двумерной графики сглаживание обеспечивается всегда, независимо от операционной системы и поддержки драйверов.

Наличие мощной видеокарты не дает абсолютной гарантии, что вы получите максимальную, с аппаратной поддержкой производительность на WPF. Программное обеспечение также играет важную роль. Например, WPF не может обеспечить аппаратного ускорения на видеокартах, если используются устаревшие драйверы. (Для устаревших видеокарт, такие драйверы, скорее всего, будут единственно доступными.) Технология WPF также обеспечивает более высокую производительность в средах операционных систем Windows Vista и Windows 7, где она может пользоваться преимуществами новой *модели дисплейных драйверов Windows (Windows Display Driver Model — WDDM)*. Модель WDDM предлагает несколько важных усовершенствований по сравнению с Windows ХР Display Driver Model (XPDM). Что более важно, WDDM позволяет запланировать несколько операций GPU одновременно и отображать страницы памяти видеокарты на обычную системную память, если вся память видеокарты израсходована.

Запомните в качестве главного эмпирического правила: WPF предоставляет некоторого рода аппаратное ускорение всем драйверам WDDM и драйверам XPDM, созданным после ноября 2004 г., когда Microsoft издала новые руководства по разработке драйверов. Разумеется, уровень поддержки отличается. Когда инфраструктура WPF запускается в первый раз, она оценивает видеокарту и назначает ей рейтинг от 0 до 2.

Среди обещаний, связанных с WPF, было и то, что вам не нужно беспокоиться о деталях и сложностях, связанных со специфическим аппаратным обеспечением. Технология WPF достаточно интеллектуальна, чтобы по возможности использовать аппаратную оптимизацию, но в случае неудачи все будет обработано программно. Поэтому если вы запустите WPF-приложение на компьютере с унаследованной видеокартой, интерфейс будет выглядеть так, как он был разработан. Конечно, программные альтернативы могут оказаться значительно медленнее, так что вы столкнетесь с тем, что компьютеры со старыми видеокартами не очень хорошо отрабатывают расширенные приложения WPF — особенно те, что включают сложную анимацию или другие сложные графические эффекты. На практике может быть принято решение упростить некоторые сложные эффекты в пользовательском интерфейсе, в зависимости от уровня аппаратной поддержки, доступной клиенту (определяется *свойством RenderCapability.Tier*).

Целью WPF является взвалить на видеокарту как можно больше работы, чтобы сложные графические процедуры ограничивались возможностями визуализации (узлом обработки графики), а не вычислительной мощностью процессора (центральным процессором компьютера). При таком подходе центральный процессор высвобождается для другой работы, видеокарта используется максимально эффективно и появляется возможность пользоваться преимуществами новых видеокарт по мере их появления.

**Уровни WPF**

Видеокарты значительно различаются между собой. Когда WPF оценивает видеокарту, то учитывает множество факторов, включая объем памяти видеокарты, поддержку построителей текстур (встроенные процедуры вычисления пиксельных эффектов наподобие прозрачности), вершинных построителей текстур (встроенные процедуры вычисления значений вершин треугольника, которые применяются при текстурировании трехмерных объектов). На основе всех этих деталей определяется значение уровня визуализации WPF.

WPF распознает три уровня визуализации:

Уровень визуализации 0

Видеокарта не предоставляет никакого аппаратного ускорения. Это соответствует версии DirectX ниже 7.0.

Уровень визуализации 1

Видеокарта обеспечивает частичное аппаратное ускорение. Это соответствует версии DirectX выше 7.0, но ниже 9.0.

Уровень визуализации 2

Все средства, которые могут быть ускорены аппаратно, будут ускорены. Это отвечает версии DirectX 9.0 и выше.

**Преимущества WPF**

Даже если бы единственным достоинством WPF было аппаратное ускорение через DirectX, это уже стало бы значительным усовершенствованием, хоть и не революционным. Однако WPF на самом деле включает целый набор высокоуровневых служб, ориентированных на прикладных программистов.

Ниже приведен список некоторых наиболее существенных изменений, которые привнесла с собой технология WPF в мир программирования Windows-приложений:

Веб-подобная модель компоновки

Вместо того чтобы фиксировать элементы управления на месте с определенными координатами, WPF поддерживает гибкий поток, размещающий элементы управления на основе их содержимого. В результате получается пользовательский интерфейс, который может быть адаптирован для отображения высокодинамичного содержимого или к разным языкам.

Богатая модель рисования

Вместо рисования пикселей в WPF вы имеете дело с примитивами — базовыми фигурами, блоками текста и прочими графическими ингредиентами. Кроме того, доступны такие новые средства, как действительно прозрачные элементы управления, возможность укладывания друг на друга множества уровней с разной степенью прозрачности, а также встроенная поддержка трехмерной графики.

Развитая текстовая модель

После многих лет нестандартной обработки текстов WPF наконец-то предоставляет Windows-приложениям возможность отображения расширенного стилизованного текста в любом месте пользовательского интерфейса. И если нужно отображать значительные объемы текста, для повышения читабельности можно воспользоваться развитыми средствами отображения документов, такими как переносы, разбиение на колонки и выравнивание.

Анимация

В WPF нет необходимости использовать таймер для того, чтобы заставить форму перерисовать себя. Вместо этого доступна анимация — неотъемлемая часть платформы. Анимация определяется декларативными дескрипторами, и WPF запускает ее в действие автоматически.

Поддержка аудио и видео

Прежние инструментальные наборы для построения пользовательских интерфейсов, такие как Windows Forms, были весьма ограничены в работе с мультимедиа. Однако WPF включает поддержку воспроизведения любого аудио или видеофайла, поддерживаемого проигрывателем Windows Media, позволяя воспроизводить более одного медиафайла одновременно. Что еще больше впечатляет — WPF предоставляет в ваше распоряжение инструменты для интеграции видеосодержимого в остальную часть пользовательского интерфейса, позволяя выполнять такие экзотические трюки, как размещение видеоокна на поверхности вращающегося трехмерного куба.

Стили и шаблоны

Стили позволяют стандартизировать форматирование и многократно использовать его по всему приложению. Шаблоны дают возможность изменить способ отображения элементов, даже таких основополагающих, как кнопки. Построение интерфейса с обложками еще никогда не было таким простым.

Команды

Большинству пользователей известно, что не имеет значения, откуда они инициируют команду открытия (Open) — через меню или панель инструментов; конечный результат один и тот же. Теперь эта абстракция доступна коду — можно определять команды приложения в одном месте и привязывать их к множеству элементов управления.

Декларативный пользовательский интерфейс

Хотя можно конструировать окно WPF в коде, в Visual Studio используется другой подход. Содержимое каждого окна сериализуется в виде XML-дескрипторов в документе XAML. Преимущество состоит в том, что пользовательский интерфейс полностью отделяется от кода, и дизайнеры графики могут использовать профессиональные инструменты для редактирования файлов XAML, улучшая внешний вид всего приложения. (XAML — это сокращение от Extensible Application Markup Language (расширяемый язык разметки приложений).)

Приложения на основе страниц

Используя WPF, можно строить браузер-подобные приложения, которые позволяют перемещаться по коллекции страниц, оснащенной кнопками навигации "вперед" и "назад". WPF автоматически обрабатывает все сложные детали, такие как хронология посещения страниц. Проект можно даже развернуть в виде браузерного приложения, которое выполняется внутри Internet Explorer.

Silverlight

Как и .NET Framework в целом, WPF представляет собой технологию, ориентированную на Windows. Это значит, что приложения WPF могут использоваться только на компьютерах, работающих под управлением операционной системы Windows. Приложения WPF, основанные на браузерах, ограничены аналогичным образом — они работают только на компьютерах Windows, хотя поддерживают браузеры и Internet Explorer, и Firefox.

Эти ограничения не изменятся: в конце концов, отчасти целью Microsoft в отношении WPF является использование широких возможностей компьютеров Windows и сохранение инвестиций в такие технологии, как DirectX. Однако технология Silverlight спроектирована как подмножество платформы WPF, работает в любом современном браузере (Firefox, Google Chrome и Safari) за счет использования подключаемого модуля, и открыта для других операционных систем, таких как GNU/Linux и Mac OS. Этот амбициозный проект вызвал значительный интерес среди разработчиков.

Во многих отношениях технология Silverlight основана на WPF и включает в себя многие соглашения WPF (наподобие разметки XAML). Тем не менее, Silverlight не охватывает ряд областей, среди которых трехмерная графика и отображение форматированных документов. В будущих выпусках Silverlight могут появиться некоторые новые средства, но наиболее сложные из них — вряд ли.

Конечной целью Silverlight является предоставление мощного ориентированного на разработчика конкурента Adobe Flash. Однако Flash обладает ключевым преимуществом — он используется в веб-приложениях повсеместно, и подключаемые модули Flash установлены почти везде. Чтобы заставить разработчиков перейти на новую, менее устоявшуюся технологию, Microsoft придется снабдить Silverlight средствами следующего поколения, обеспечить основательную совместимость и непревзойденную проектную поддержку.

**Независимость от разрешения**

Традиционные Windows-приложения связаны определенными предположениями относительно разрешения экрана. Обычно разработчики рассчитывают на стандартное разрешение монитора (вроде 1024x768 пикселей) и проектируют свои окна с учетом этого, стараясь обеспечить разумное поведение при изменении размеров в большую и меньшую сторону.

Проблема в том, что пользовательский интерфейс в традиционных Windows-приложениях не является масштабируемым. В результате, если вы используете монитор с высоким разрешением, который располагает пиксели более плотно, окно приложения становится меньше и читать текст в нем труднее. Эта проблема особенно актуальна для новых мониторов, которые имеют высокую плотность пикселей и соответственно работают с более высоким разрешением.

Например, легче встретить мониторы (особенно на портативных компьютерах), которые имеют плотность пикселей в 120 dpi или 144 dpi (точек на дюйм), чем более традиционные 96 dpi. При их встроенном разрешении эти мониторы располагают пиксели более плотно, создавая напрягающие глаз мелкие элементы управления и текст. В идеале приложения должны использовать более высокую плотность пикселей, чтобы отобразить больше деталей. Например, монитор с высоким разрешением может отображать одинакового размера значки панели инструментов, но использовать дополнительные пиксели для отображения мелкой графики. Подобным образом можно сохранить некоторую базовую компоновку, но обеспечить более высокую четкость деталей.

(Комментарий.

**DPI**это сокращение от английского "**d**ots **p**er **i**nch" и переводится как "точек на дюйм". Количество пикселей на единицу длины называется разрешением. Чем больше точек на дюйм, тем выше разрешение и тем самым - качественнее изображение.

**72 DPI**(или 96) – файлы для интернет, где одна точка на экране вашего монитора соответствует одному пикселю изображения. Еще называют – экранное разрешение изображения. Больше чем 72 DPI делать картинку для Интернет – нет смысла, так как монитор все равно больше не отобразит.

*72 точки на дюйм = 28 точкам на сантиметр.*

**300 DPI** необходимо для качественной печати в типографии. Количество точек на дюйм значительно увеличено, чтобы повысить качество изображения. Но для каждого типа печати это значение может меняться. В целом, если печатный продукт рассматривается человеком на близком расстоянии, то количество точек на дюйм должно быть высоким. Это журналы, буклеты, листовки. Для макета, который будут рассматривать на расстоянии требования ниже. К примеру, для билборда это значение может быть 56 dpi и ниже.  
*300 точек на дюйм = 118 точкам на сантиметр.*

)

По разным причинам такое решение было невозможно в прошлом. Хотя можно изменять размер графического содержимого, нарисованного в GDI/GDI+, компонент User32 (который генерирует визуальное представление распространенных элементов управления) не поддерживает реального масштабирования.

WPF не страдает от этой проблемы, потому что самостоятельно визуализирует все элементы пользовательского интерфейса — от простых фигур до таких распространенных элементов управления, как кнопки. В результате если вы создаете кнопку шириной в 1 дюйм на обычном мониторе, она останется шириной в 1 дюйм и на мониторе с высоким разрешением. WPF просто визуализирует ее более детализировано, с большим количеством пикселей.

Так выглядит картина в целом, но нужно уточнить еще несколько деталей. Самое важное, что следует осознать — WPF базирует свое масштабирование на системной установке DPI, а не на DPI физического дисплейного устройства. Это совершенно логично — в конце концов, если вы отображаете приложение на 100-дюймовом проекторе, то, скорее всего, отойдете подальше на несколько шагов и будете ожидать увидеть огромную версию окон. Конечно, не желательно, чтобы WPF масштабировал приложение, уменьшая его до "нормального" размера. Аналогично, если вы используете портативный компьютер с дисплеем высокого разрешения, то хотите увидеть несколько уменьшенные окна; это цена, которую приходится платить за то, чтобы уместить всю информацию на маленьком экране. Более того, у разных пользователей разные предпочтения на этот счет. Некоторым нужны расширенные подробности, в то время как другие хотят увидеть больше содержимого.

Так каким же образом WPF определяет, насколько большим должно быть окно приложения? Краткий ответ состоит в том, что при вычислении размеров WPF использует системную установку DPI. Но чтобы понять, как это в действительности работает, необходимо более детально ознакомиться с системой измерений WPF.

**Единицы WPF**

Окно WPF и все элементы внутри него измеряются в *независимых от устройства единицах*. Такая единица определена как 1 /96 дюйма. Чтобы понять, что это означает на практике, нужно рассмотреть пример.

Предположим, что вы создаете в WPF маленькую кнопку размером 96x96 единиц. Если вы используете стандартную установку Windows DPI (96 dpi), то каждая независимая от устройства единица измерения соответствует одному реальному физическому пикселю. Это потому, что WPF использует следующее вычисление:

*[Размер в физических единицах] = [Размер в независимых от устройства единицах] х [DPI системы] =*

*= 1/96 дюйма х 96 dpi = 1 пиксель*

По сути, WPF предполагает, что ему нужно 96 пикселей, чтобы отобразить один дюйм, потому что Windows сообщает ему об этом через системную настройку DPI. Однако в действительности это зависит от применяемого дисплейного устройства. Например, рассмотрим 20-дюймовый жидкокристаллический монитор с максимальным разрешением в 1600x1200 пикселей. Используя теорему Пифагора, вы можете вычислить плотность пикселей для этого монитора, как показано ниже:

*[DPI экрана] = (1600^2 + 1200^2)^0.5 пикселей / 20 дюймов = 100 dpi*

В этом случае плотность пикселей составляет 100 dpi — немного больше того, что предполагает Windows. В результате на этом мониторе кнопка размером 96x96 пикселей будет несколько меньше одного дюйма.

С другой стороны, рассмотрим 15-дюймовый жидкокристаллический монитор с разрешением 1024x768 пикселей. Здесь плотность пикселей составит около 85 dpi, поэтому кнопка размером 96x96 пикселей окажется размером немного больше 1 дюйма.

В обоих случаях, если вы уменьшите размер экрана (скажем, переключившись на разрешение 800x600), то кнопка (и любой другой экранный элемент) станет пропорционально больше. Причина в том, что системная установка DPI останется 96 dpi. Другими словами, Windows продолжает предполагать, что 96 пикселей составляют дюйм, несмотря на то, что при меньшем разрешении потребуется существенно меньше пикселей.

Возможно, вам известно, что жидкокристаллические мониторы создаются с единственным разрешением, которое называется *естественным разрешением*. При более низком разрешении монитору приходится использовать интерполяцию, чтобы заполнить лишние пиксели, это может вызвать нерезкость. Чтобы получить наилучшее качество изображения, всегда лучше использовать естественное разрешение. Если хотите иметь более крупные окна, кнопки и текст, рассмотрите вместо этого возможность модификации системной установки DPI.

Системная установка DPI

До сих пор пример кнопки WPF работал точно так же, как любой другой интерфейсный элемент в Windows-приложении любого иного типа. Отличие проявляется при изменении вашей системной установки DPI. В предыдущем поколении Windows это средство иногда называли крупными шрифтами. Это потому, что системная установка DPI влияет на размер системных шрифтов, часто оставляя прочие детали неизменными.

Многие Windows-приложения не полностью поддерживают увеличенные установки DPI. В худшем случае увеличение системной установки DPI может привести к появлению окон, в которых некоторое содержимое увеличено, а другое — нет, что может привести к утере части содержимого или даже к нефункциональным окнам.

Здесь поведение WPF отличается. WPF воспринимает системную установку DPI естественным образом и без особых затрат. Например, если вы измените системную установку DPI на 120 dpi (распространенный выбор пользователей экранов с большим разрешением), WPF предполагает, что для заполнения дюйма пространства нужно 120 пикселей. WPF использует следующее вычисление для определения того, как он должен транслировать логические единицы в физические пиксели устройства:

*[Размер в физических единицах] = [Размер в независимых от устройства единицах] х [DPI системы] =*

*= 1/96 дюйма х 120 dpi =1,25 пикселя*

Другими словами, когда вы устанавливаете системную настройку DPI в 120 dpi, то механизм визуализации WPF предполагает, что одна независимая от устройства единица измерения соответствует 1,25 пикселя. Если вы отображаете кнопку 96x96, то физический ее размер составит 120x120 пикселей (потому что 96 х 1,25 = 120). Именно такого результата вы и ожидаете — кнопка размером в 1 дюйм имеет такой же размер на мониторе с повышенной плотностью пикселей.

Такое автоматическое масштабирование было бы не слишком полезным, если бы касалось только кнопок. Но WPF использует независимые от устройства единицы для всего, что отображает, включая фигуры, элементы управления, текст и любые другие ингредиенты, которые помещаются в окно. В результате можно изменять системную установку DPI, как вам заблагорассудится, и WPF незаметно подгонит размеры окон приложения.

В зависимости от системной установки DPI вычисляемый размер пикселя может быть выражен дробным значением. Можно предположить, что WPF просто округляет все размеры до ближайшего пикселя. Однако по умолчанию WPF поступает несколько иначе. Если грань элементов приходится на точку между пикселями, WPF использует сглаживание, чтобы размыть эту грань. Это может показаться странным решением, но на самом деле оно вполне оправдано. Элементы управления не обязательно должны иметь прямые четкие грани, если для их отображения применяется специально отрисованная графика, поэтому некоторая степень сглаживания все равно необходима.

**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 использовать 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:

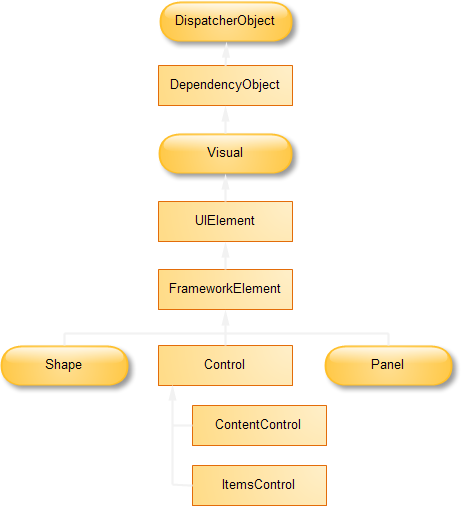
* **milcore.dll**: собственно обеспечивает интеграцию компонентов WPF с DirectX. Данный компонент написан на неуправляемом коде (С/С++) для взаимодействия с DirectX.
* **WindowsCodecs.dll**: библиотека, которая предоставляет низкоуровневую поддержку для изображений в WPF

Еще ниже собственно находятся компоненты операционной системы и DirectX, которые произвоят 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.

**Иерархия классов**

Ниже показан базовый обзор некоторых ключевых ветвей иерархии классов. Важно отметить, что основные пространства имен WPF начинаются в System.Windows (например, System.Windows, System.Windows.Controls и System.Windows.Media). Единственным исключением являются пространства имен, начинающиеся с System.Windows.Forms, которые относятся к инструментам Windows Forms.



**System.Threading.DispatcherObject**

Приложения WPF используют знакомую однопоточную модель (*single-thread affinity — STA*), а это означает, что весь пользовательский интерфейс принадлежит единственному потоку. Взаимодействовать с элементами пользовательского интерфейса из других потоков небезопасно. Чтобы содействовать работе этой модели, каждое WPF-приложение управляется диспетчером, координирующим сообщения (появляющиеся в результате клавиатурного ввода, перемещений курсора мыши и таких процессов платформы, как компоновка). Будучи унаследованным от DispatcherObject, каждый элемент пользовательского интерфейса может удостовериться, выполняется ли код в правильном потоке, и обратиться к диспетчеру, чтобы направить код в поток пользовательского интерфейса.

Этот тип включает одно свойство, представляющее интерес — Dispatcher, которое возвращает ассоциированный объект System.Windows.Threading.Dispatcher. Класс Dispatcher — это точка входа в очередь событий приложения WPF, предоставляющая базовые конструкции для работы с параллелизмом и многопоточностью. По большому счету, это низкоуровневый класс, который в большинстве приложений WPF может быть проигнорирован.

**System.Windows.DependencyObject**

В WPF центральный путь взаимодействия с экранными элементами пролегает через свойства. На ранней стадии цикла проектирования архитекторы WPF решили создать более мощную модель свойств, которая положена в основу таких средств, как уведомления об изменениях, наследуемые значения по умолчанию и более экономичное хранилище свойств. Конечным результатом стало средство свойств зависимости (dependency property). За счет наследования от DependencyObject, классы WPF получают поддержку свойств зависимости.

Базовый класс DependencyObject предоставляет два ключевых метода для всех производных типов: GetValue() и SetValue(). С помощью этих членов можно устанавливать само свойство. Другие части инфраструктуры позволяют "регистрировать" тех, кто может использовать свойства зависимости или присоединяемые свойства.

Хотя свойства зависимости — это ключевой аспект разработки WPF, большую часть времени их детали скрыты от глаз.

**System.Windows.Media.Visual**

Каждый элемент, появляющийся в WPF, в основе своей является Visual. Класс Visual можно воспринимать как единственный объект рисования, инкапсулирующий в себе инструкции рисования, дополнительные подробности рисования (наподобие отсечения, прозрачности и настроек трансформации) и базовую функциональность (вроде проверки попадания). Класс Visual также обеспечивает связь между управляемыми библиотеками WPF и сборкой milcore.dll, которая визуализирует отображение. Любой класс, унаследованный от Visual, обладает способностью отображаться в окне. Если вы предпочитаете создавать свой пользовательский интерфейс с применением легковесного API-интерфейса, не обладающего высокоуровневыми средствами WPF, то можете программировать непосредственно с использованием объектов Visual.

Применение типа Visual (и его потомков вроде DrawingVisual) обеспечивает наиболее легковесный способ визуализации графических данных, но также подразумевает участие большого объема управляемого кода для обеспечения работы всех необходимых служб.

**System.Windows.UIElement**

Класс UIElement добавляет поддержку таких сущностей WPF, как *компоновка (layout)*, *ввод (input)*, *фокус (focus)* и *события (events)* — все, что команда разработчиков WPF называет аббревиатурой LIFE. Например, именно здесь определен двухшаговый процесс измерения и организации компоновки. Здесь же щелчки кнопками мыши и нажатия клавиш трансформируются в более удобные события, такие как MouseEnter. Как и со свойствами, WPF реализует расширенную систему передачи событий, именуемую *маршрутизируемыми событиями (routed events)*.

**System.Windows.FrameworkElement**

Класс FrameworkElement — конечный пункт в центральном дереве наследования WPF. Он реализует некоторые члены, которые просто определены в UIElement. Например, UIElement устанавливает фундамент для системы компоновки WPF, но FrameworkElement включает ключевые свойства (вроде HorizontalAlignment и Margin), которые поддерживают его. UIElement также добавляет поддержку привязки данных, анимации и стилей — все они являются центральными средствами.

**System.Windows.Shapes.Shape**

От этого класса наследуются базовые фигуры, такие как Rectangle, Polygon, Ellipse, Line и Path. Эти фигуры могут использоваться наряду с более традиционными графическими элементами Windows вроде кнопок и текстовых полей.

**System.Windows.Controls.Control**

Элемент управления (control) — это элемент, который может взаимодействовать с пользователем. К нему очевидным образом относятся такие классы, как TextBox, Button и ListBox. Класс Control добавляет дополнительные свойства для установки шрифта, а также цветов переднего плана и фона. Но наиболее интересная деталь, которую он предоставляет — это поддержка шаблонов, которая позволяет заменять стандартный внешний вид элемента управления собственным рисованием.

Control, так же, определяет свойства для установки размеров элемента управления, прозрачности, порядка обхода по нажатию клавиши <Tab>, дисплейного курсора, цвета фона и т.д.

**System.Windows.Controls.ContentControl**

Это базовый класс для всех элементов управления, которые имеют отдельный фрагмент содержимого. Сюда относится все — от скромной метки Label до окна Window. Наиболее впечатляющая часть этой модели заключается в том, что единственный фрагмент содержимого может быть чем угодно — от обычной строки до панели компоновки, содержащей комбинацию других фигур и элементов управления.

Например, когда речь идет о типичном элементе управления "кнопка", то обычно предполагается, что его содержимым будет базовый строковый литерал (ОК, Cancel, Abort и т.п.). В случае использования XAML для описания элемента управления WPF, и значение, которое необходимо присвоить свойству Content, может быть выражено в виде простой строки, можете установить свойство Content внутри открывающего определения элемента.

Содержимое может быть любым. Например, предположим, что нужна "кнопка", которая содержит в себе нечто более интересное, чем простая строка, возможно, специальную графику или текст. На других платформах построения пользовательских интерфейсов, таких как Windows Forms, пришлось бы строить специальный элемент управления, что потребовало бы написания значительного объема кода и сопровождения нового класса. С моделью содержимого WPF это не требуется.

Когда в свойстве Content должно быть установлено значение, которое не может быть выражено простым массивом символов, его нельзя присвоить с использованием атрибута в открывающем определении элемента управления. Вместо этого понадобится определить данные содержимого неявно, внутри контекста элемента.

**System.Windows.Controls.ItemsControl**

Это базовый класс для всех элементов управления, которые отображают коллекцию каких-то единиц информации, вроде ListBox и TreeView. Списочный элемент управления замечательно гибок; например, используя встроенные средства класса ItemsControl, можно трансформировать обычный ListBox в список переключателей, список флажков, упорядоченный набор картинок или комбинацию совершенно разных элементов по своему выбору. Фактически в WPF все меню, панели инструментов и линейки состояния на самом деле являются специализированными списками и классами.

**System.Windows.Controls.Panel**

Это базовый класс для всех контейнеров компоновки — элементов, которые содержат в себе один или более дочерних элементов и упорядочивают их в соответствии с определенными правилами компоновки. Эти контейнеры образуют фундамент системы компоновки WPF, и их использование — ключ к упорядочиванию содержимого наиболее привлекательным и гибким способом.