الرسم والتلوين باستخدام GDI+ و DirectX

لمبرمجي سي شارب

مقـــدمـــة

بسم الله والصلاة والسلام على رسول الله، وبعد:

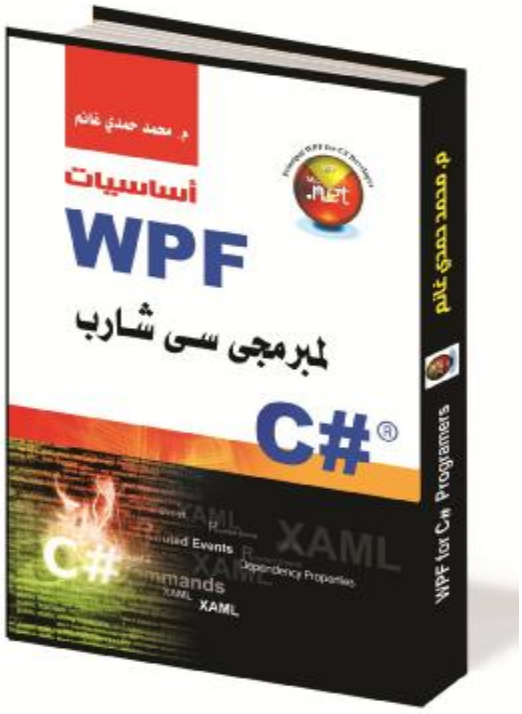
عندما كتبت مرجع "**من الصفر إلى الاحتراف سي شارب**" فـــي عـــام ٢٠٠٨، ومرجع "**من الصفر إلى الاحتراف برمجة نماذج الوزينـــدوز**" في عام ٢٠٠٩، نوهت فيهما إلى أنني أنوي كتابة مرجع عن برمجة الوسائط المتعددة في دوت نت، يتضمن الرسم والتلوين والصوت والفيديو والألعاب.. لكن هذا المرجـــع تأخر بسبب ترتيب أولوياتي في الكتابة عن مواضيع أخرى، ثم وجدت أنـــه لا ضرورة له، لأن **تقنية WPF** ــ التي تعتبر الجيل التالي لتقنية نماذج الويندوز ــ قدمت نموذجا أسهل وأقوى للتعامل مع الرسوم والوسائط المتعددة والتحريك والمجسمات، كما أنها لا تعتمد على تقنيـــة GDI+ المســـتخدمة فـــي نمـــاذج الويندوز، بل تعتمد مباشرة على تقنية DirectX مما يجعلها أقـــوى وأســـهل وتقدم إمكانيات رسومية غير مسبوقة.. لكل هذا كان من الطبيعـــي أن أتخـــذ قرارا بالكتابة عن الوسائط المتعددة من منظور WPF لا من منظور نمـــاذج الويندوز، لأواكب الأحدث.

ولكن لحسن الحظ، أني سبق أن ترجمت مرجع Mastering VB.NET إلى العربية ونشرته مجانا على الشبكة الدولية عام ٢٠٠٣، وهو يحتوي على جزء مفيد عن الرسم والتلوين والصور، كما أني أضفت إلى الترجمة مواضيع لـــم تكن في المرجع الأصلي، مثل التعامل مع الصور في الذاكرة مباشرة، وترجمة لمشروع تعليمي عن DirectX.. لهذا وجدت أنه من المناسب أن أجمع هـــذه الأجزاء في هذا الكتاب وأحولها إلى كود سي شارب وأنشرها مجانا، لأعوض قراء كتبي عن المرجع الذي وعدتهم به ولم أكتبه، إلى أن أقدم بإذن الله الكتاب الجديد عن الوسائط المتعددة في WPF.

محمد حمدي غانم

٢٠١٤/٨/١٢

كتب م. محمد حمدي غانم



- **<u>كتب لمبرمجي سي شارب:</u>**

١. أساسيات WPF لمبرمجي سي شارب
٢. المدخل العملي السريع إلى سي شارب ٢٠١٠
٣. من الصفر إلى الاحتراف: سي شارب ٢٠١٠
٤. من الصفر إلى الاحتراف: برمجة إطار العمل (سي شارب)
٥. من الصفر إلى الاحتراف: برمجة نماذج الويندوز (سي شارب)
٦. من الصفر إلى الاحتراف: برمجة قواعد البيانات (سي شارب)
٧. فيجوال بيزيك وسي شارب: طريقك المختصر للانتقال من إحدى اللغتين إلى الأخرى

(وتوجد نسخة من كل كتاب لمبرمجي VB.NET).

لتفاصيل أكثر عن هذه الكتب:
http://mhmdhmdy.blogspot.com/2010/09/blog-post_9555.html

- **كتب أدبية وفكرية:**

- خرافة داروين، حينما تتحول الصدفة إلى علم.. للتحميل:
http://mhmdhmdy.blogspot.com/2013/11/blog-post_29.html
- دلال الورد (ديوان شعر).. للتحميل:
http://www.mediafire.com/?n1qte7j9hdv1l98
- حائرة في الحب (رواية).. للتحميل:
http://www.mediafire.com/?hd1jy6ca4ay3m9w

- **للتواصل مع الكاتب:**

- بريد الكتروني:
msvbnet@hotmail.com
- المدونة:
http://mhmdhmdy.blogspot.com
- قناة يوتيوب:
http://www.youtube.com/user/mhmdhmdy
- صفحة الشاعر محمد حمدي غانم:
https://www.facebook.com/Poet.Mhmd.Hmdy
- صفحة فيجوال بيزيك وسي شارب:
https://www.facebook.com/vbandcsharp

الفهرس

الفصل الأول

الرسم والتلوين باستخدام GDI+

الفصل الثاني

الأدوات الخاصة Custom Controls

الفصل الثالث

مشروع تعليمي بـــ Direct3D

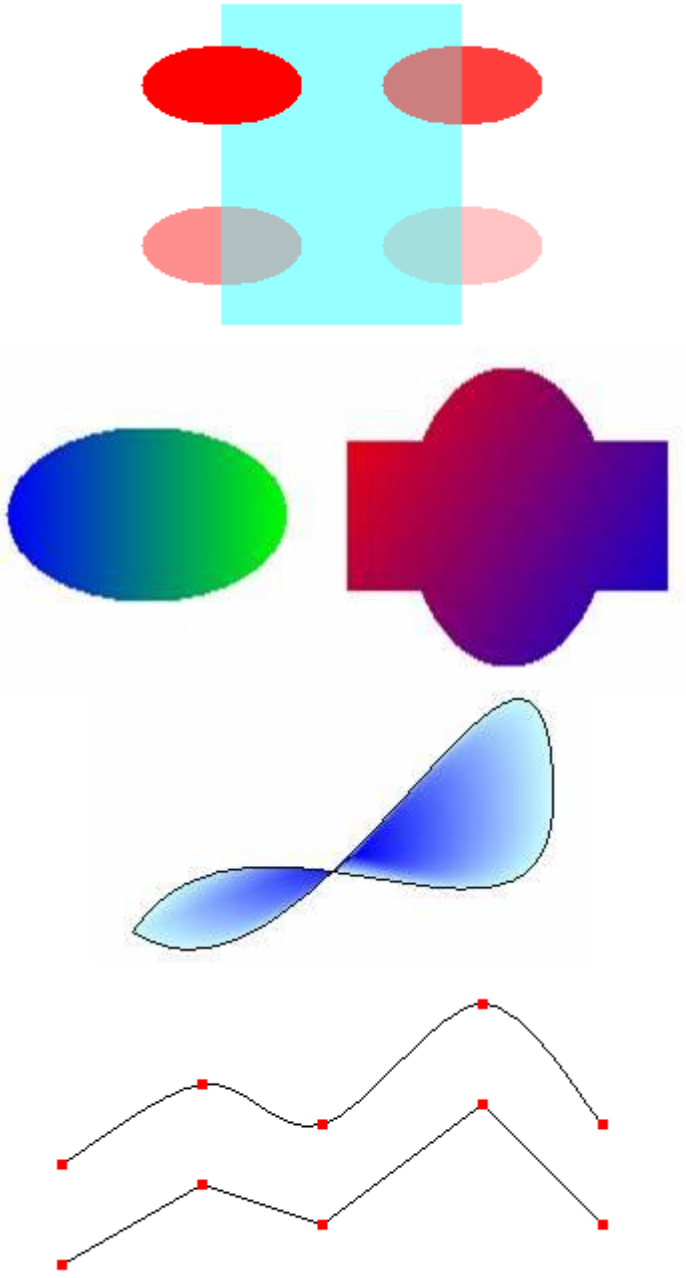
الفصل الأول

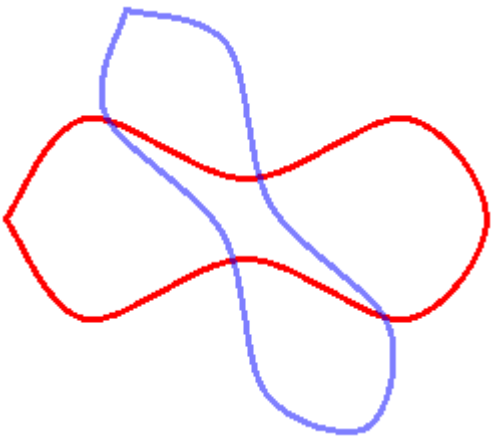
الرسم والتلوين باستخدام GDI+

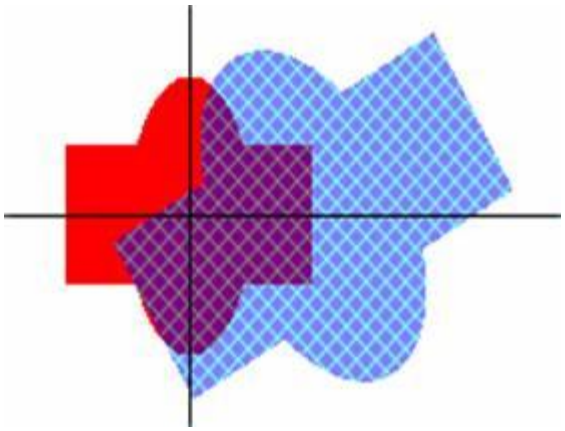
قدرات مدهشة:

هل ترى هذه الرسوم الرائعة؟

هل تصدّق أنّك تستطيع أن ترسم مثلها في كود سي شارب؟







إنّ إطار العمال Net FrameWork. يمنحك العديد مـــن الفئـــات التـــي تستطيع أن تستخدمها في سي شارب لإنشاء تطبيقات رسوم رائعة.

وتنقسم الرسومات Graphics إلى نوعيّن:

١- الرسوم الرياضيّة Vector: وهي التي ترسمها باستخدام دوال سي شارب، مثل دوال رسم الخطوط والمنحنيات التي تعتبـــر إحـــدى أنواع الرسوم الرياضيّة، مثل تلك التي رأيناها بأعلى.

٢- الصور Bitmaps: حيث يمكنك عرضها في العديد مـــن أدوات سي شارب، كما يمكنك إجراء العمليّات عليها عن طريق معالجتها نقطة نقطة.

ولا غنى لك عن استخدام كلتا الوسيلتين، وإن كان استخدامك للصور سيكون أكثر تواترا.

ويمكنك الرسم في سي شارب، تقريبا على أيّ أداة تخطر ببالك، بما في ذلك مربّع النصّ والقائمة ListBox، وإن كان الأكثر شيوعا أن ترسم على النموذج ومربّع الصورة PictureBox.

ولمعظم الأدوات خاصيّة "الصورة" Image أو "صورة الخلفيّة" BackgroundImage، التي يمكنك تغيير قيمتها من نافذة الخصائص في وقت التصميم، أو بكتابة الكود الذي يغيّرهما في البرنامج.

ملاحظة:

عند وضع صورة في خلفيّة أيّ أداة أو نموذج في وقت التصميم، يتمّ نسخها من موقعها الأصليّ إلى ملفّات مشروعك، بحيث لا تقلق إذا تغيّر موقع برنامجك أو موقع الصورة الأصليّة.. أمّا لو استخدمت الكود في تحميل صورة، فإنّها تظلّ في مكانها الأصليّ.. لهذا يجب عليك تجنّب الأخطاء التي يمكن أن تحدث عند حذف هذه الصورة أو تغيّر موقعها.

التعامل مع الألوان

كائن اللون Color Object:

يمكنك تعريف متغيّر من هذا السجلّ كما يلي:

Color myColor = Color.Azure;

ويمنحك سجلّ الألوان Color Structure ١٢٨ خاصية تمثل ١٢٨ لونا بأسمائها الإنجليزيّة (ولا أدّعي أنّني أعرف باللغة العربيّة هذا العدد من أسماء الألوان!!)، مثل اللون اللازورديّ (الأزرق السماويّ) Azure..

ويمكنك أن تتعرّف على هذه الألوان بمجرّد كتابة:

Color.

حيث ستظهر لك قائمة بكلّ قيم المرقّم Color.

ومن أهمّ الألوان اللون الشفّاف Transparent، حيث يمكنك أن تستخدمه لجعل خلفيّة بعض الأدوات ــ كاللافتة ــ شفّافة:

Lable1.BackColor = Color.Transparent;

بالإضافة إلى هذا، يمنحُك هذا السجلّ بعض الوسائل الهامّة، ومنها:

تكوين اللون من مركّباته FromArgb:

يمكنك أن تكوّن ملايين الألوان باستخدام هذه الوسيلة، حيث ترسل لها نسب الأحمر Red والأخضر Green والأزرق Blue لتعيد لك اللون المكوّن منها.. (طبعا اتضح لك أنّ الحروف RGB هي الحروف الأولى من أسماء الألوان الثلاثة).

وفي المثال التالي نغيّر لون خلفيّة مربّع النصّ باستخدام دالة تكوين اللون:

```
TextBox1.BackColor = Color.FromArgb(25, 150, 255);
```

وهنا يجب أن تلاحظ شيئين:

١- أنّ كلّ مكوّن من المكوّنات الثلاثة تنحصر قيمته بين ٠ و ٢٥٥، فإذا أرسلت للمعاملات أيّ عدد أكبر من هذا فسيكون تأثيره كتأثير العدد ٢٥٥.

٢- أنّ معكوس أيّ لون (نيجاتيف اللون) ينتج بطرح كلّ مكوّن من مكوّناته الثلاثة من ٢٥٥، وسيتّضح لك ذلك عندما تعرف أنّ مكونات اللون الأبيض هي (٢٥٥، ٢٥٥، ٢٥٥) ومكونات اللون الأسود هي (٠، ٠، ٠).

والجدول التالي يوضّح لك مكوّنات بعض الألوان الهامّة:

اللون	نسبة الأحمر R	نسبة الأخضر G	نسبة الأزرق B
الأسود	٠	٠	٠
الأزرق	٠	٠	٢٥٥
الأخضر	٠	٢٥٥	٠
السماوي	٠	٢٥٥	٢٥٥

الأحمر	٢٥٥	٠	٠
الأحمر البنفسجيّ	٢٥٥	٠	٢٥٥
الأصفر	٢٥٥	٢٥٥	٠
الأبيض	٢٥٥	٢٥٥	٢٥٥
الرمادي	١٢٨	١٢٨	١٢٨

وإذا أردت الحصول على لون فاتح من الألوان الماضية، فاستبدل العدد ١٢٨ بالعدد ٢٥٥ في كلّ مكوّنات اللون.

وهناك صيغة أخرى من هذه الوسيلة، تسمح لك بتحديد درجة شفافية اللون:

FromArgb(شفافية اللون ,نسبة الأحمر ,نسبة الأخضر ,نسبة الأزرق)

ويسمّى معامل الشفافيّة "خليط ألفا" Alpha Blinding، وتتراوح الشفافية ما بين ٠ (شفافية تامّة) و ٢٥٥ (إعتام تامّ).

ومن أروع استخدامات الشفافية، استخدامها لكتابة نصوص تبدو للرائي مجسّمة، وذلك برسم النصّ بلون ما (سنعرف كيفيّة ذلك لاحقا)، ثمّ رسم نفس النصّ بلون آخر وجعله شبه شفّاف، على أن يكون النصّ في المرّة الثانية مزاحا قليلا رأسيّا وأفقيّا.. ويمكنك أن ترى هذه الطريقة في المشروع TextEffects.

كما يمكنك استخدام الشفافية في رسم علامة مائيّة على الصورة عند نشرها على الإنترنت.. هذه العلامة المائيّة لن ترهق العين، ولكنّها ستعوق الآخرين عن استخدام هذه الصورة بدون تصريح منك.

والكود التالي يكتب الجملة "MySite.com" على النموذج نصف شفّافة.

```
var F = new Font("Comic Sans MS", 20,
        FontStyle.Bold);
var B = new SolidBrush(Color.FromArgb(50, 230,
        80, 120));
this.CreateGraphics( ).DrawString("MySite.com", F,
        B, 100, 40);
```

تكوين لون معروف FromKnownColor:

تسمح لك هذه الوسيلة بتكوين اللون من قيم المرقّم KnownColor.. وهذه القيم تندرج تحت طائفتين:

- ١٢٨ لونا بأسمائها الإنجليزيّة المعروفة، مثل KnownColor.Red .
- الألوان الخاصّة بنظام الويندوز (والتي يغيّرها المستخدم من خصائص سطح المكتب لتكون عامّة للويندوز)، مثل لون العناوين KnownColor.ActiveCaption، ولون نصوص العناوين KnownColor.ActiveCaptionText.. إلخ.

GetBrightness:

للحصول على درجة إضاءة اللون.

GetHue:

للحصول على درجة تدرّج اللون.

GetSaturation:

للحصول على درجة تشبّع اللون.

ToArgb:

استخدم هذه الوسيلة للحصول على العدد الدالّ على اللون (وهو يتكوّن من ٤ وحدات 4 Bytes)، حتّى تستطيع استخدامه مع التطبيقات الأخرى التي تتعامل مع الألوان كأرقام وليس ككائنات.

الخصائص:

درجة شفافية اللون A ــ نسبة الأحمر R ــ نسبة الأخضــر G ــ نسبة الأزرق B.

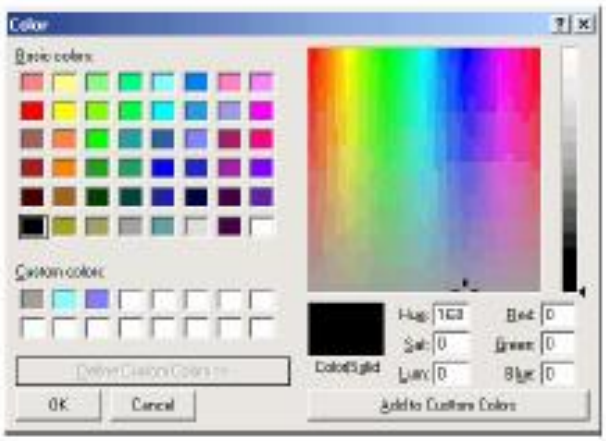
وأعتقد أنّها غنيّة عن التعريف.

والدالة التالية تعيد لك معكوس اللون الذي ترسله لها، مع المحافظة علــى درجة شفافيته كما هي:

```
public Color RevColor(Color Clr)
{
    return Color.FromArgb(Clr.A,
            255 - Clr.R, 255 - Clr.G, 255 - Clr.B);
}
```

مربّع حوار اختيار اللون ColorDialog:

كثيرا ما تريد أن تمنح المستخدم الحرّيّة في اختيار ألوان خلفيّات النوافذ والأدوات.. في هذه الحالة لا بدّ أن تعرض مربّع حوار اختيار الألوان، ليختار منه المستخدم اللون الذي يناسبه.. تمنحك سي شارب هذا المربّع جاهزا، وستجده في صندوق الأدوات باسم ColorDialog، حيث يمكنك أن تضيفه للنموذج كأيّ أداة أخرى.



تعال نرى خصائص مربّع الحوار هذا:

السماح بفتح كامل AllowFullOpen:

يعرض مربّع حوار الألوان زرّا بعنوان "تعريف ألوان مخصّصة" Define Custom Colors.. لو جعلت قيمة هذه الخاصيّة False فسيكون هذا الزر غير فعّال ولن يتمكّن المستخدم من ضغطه.. أما إذا جعلتها True، فسيتمكّن المستخدم من ضغط هذا الزرّ، حيث سيظهر نموذج آخر، يستطيع المستخدم من خلاله تركيب أيّ لون يريده.

أيّ لون AnyColor:

لو جعلت هذه الخاصيّة True، فسيتمّ عرض جميع الألوان الأساسيّة المتاحة في مربّع حوار الألوان.

اللون Color:

ضع في هذه الخاصيّة اللون الذي سيكون محددا عندما يفتح المستخدم مربّع الحوار، وبذلك يمكنك أن تعرض للمستخدم اللون الحالي للأداة التي سيغيّر لونها (لون خلفيّة مربّع النصّ مثلا)، أو تعرض له آخر لون اختاره في المرّة السابقة.

كما أنّ هذه الخاصيّة تسمح لك بمعرفة اللون الذي اختاره المستخدم عند إغلاق مربّع الحوار.. انظر للمثال التالي:

```
// اللون الحاليّ لخلفيّة النموذج هو الذي سيكون محددا عند فتح مربع الحوار
ColorDialog1.Color = this. BackColor;
// عرض مربّع الحوار
// ويجب التأكّد من أنّ المستخدم قد ضغط موافق وليس إلغاء
if (ColorDialog1.ShowDialog( ) == DialogResult.OK)
    // سنجعل خلفيّة النموذج باللون الذي اختاره المستخدم
    this.BackColor = ColorDialog1.Color;
```

ألوان مخصّصة CustomColors:

في الجزء السفليّ من مربّع حوار الألوان منطقة يمكنك فيها وضع ١٦ لونا خاصّا بك.. تضعها كلّها أو بعضها.. ولكي تفعل هذا، عرّف مصفوفة من الأعداد الصحيحة، وضع فيها الأرقام الدالة على الألوان التي تريد ظهورها في المنطقة المخصّصة، وضعها في هذه الخاصيّة:

```
int[ ] colors = {222663, 35453, 7888};
ColorDialog1.CustomColors = colors;
```

ويمكنك أن تستخدم كائن الألوان لتعريف الألوان بأسماء واضحة، ثمّ حوّلها لأعداد صحيحة باستخدام الوسيلة ToArgb.. ونظرا لأنّ أوّل وحدة Byte على يسار اللون مخصّصة للشفافية، فإنّ ذلك قد يؤدّى لأرقام سالبة.. لهذا استخدم دالة العدد المطلق من فئة الدوالّ الرياضيّة Math.Abs للتأكّد من أنّ كلّ الأرقام موجبة:

```
int[ ] colors = {Math.Abs(Color.Gray.ToArgb( )),
                 Math.Abs(Color.Navy.ToArgb( )),
                 Math.Abs(Color.Teal.ToArgb( ))};
```

الألوان المتجانسة فقط SolidColorOnly:

اجعل هذه الخاصيّة True إذا كان برنامجك سيعمل على نظام يعرض ٢٥٦ لونا فقط.

وللتدريب على استخدام هذه الأداة، افحص الأمر Page Color والأمر Text Color في القائمة Customize في المشروع TxtPad في مجلّد برامج هذا الفصل.

رسم الأشكال الرياضيّة

واجهة تصميم الرسومات:

Graphics Design Interface (GDI+):

واجهة تصميم الرسومات (GDI) هي مجموعة من الفئات التي تمكّنك من إنشاء الرسوم والنصوص والصور، وهي بمثابة المحرّك Engine الـــذي يستخدمه الويندوز نفسه في هذه العمليّات.. ولقد تطوّرت تقنية GDI عبر الزمن، حتّى جاءت أحدث نسخة منها مع VS.Net تحمل اسم GDI+.

وتقع فئات هذا المحرّك في الفضاءات التالية، التي يجب عليـــك اســـتيراد بعضها على الأقلّ قبل استخدامها في مشروعك:

System.Drawing
System.Drawing.Drawing2D
System.Drawing.Imaging
System.Drawing.Text

كيف ترسم:

لكي ترسم أيّ شيء يجب أن يتوافر لديك ما يلي:

سطح الرسم Surface:

سطح الرسم في دوت نت هو كائن الرسوم Graphics object، الذي يمدّك بالوسائل التي ترسم الأشكال الأساسيّة وغير الأساسيّة.

وللرسم على نموذج أو أداة، يجب أن نتعامل مع كائن الرسوم الخاصّ بها، والذي تحصل عليه من الوسيلة "إنشاء الرسوم" CreateGraphics.. كما يمكن الرسم على كائن صورة نقطيّة Bitmap Object، ووضع الصورة الناتجة بعد ذلك على النموذج أو الأداة، كما سنرى فيما بعد.

أداة الرسم:

لدينا أداتان رئيسيّتان نرسم بهما:

١. القلم Pen:

ويمكنك استخدامه في رسم الأشكال المكوّنة من حدود، مثل الخطوط lines، والمستطيلات rectangles والمنحنيات curves.

٢. الفرشاة Brush:

ويمكنك استخدامها في رسم المساحات الملوّنة.

والمثال البسيط التالي يوضّح لك كيف ترسم خطّا على النموذج:

```
تجهيز قلم أحمر سمك خطّه نقطتان //
var redPen = new Pen(Color.Red, 2);
نقطة بداية الخطّ //
var point1 = new Point(10,10);
```

```
// نقطة نهاية الخطّ
var point2 = new Point(120,180);
// رسم الخطّ على النموذج
Graphics G = this.CreateGraphics( );
G.DrawLine(redPen, point1, point2);
```

ويمكن دمج كلّ الخطوات السابقة في الخطوة الوحيدة التالية:

```
this.CreateGraphics( ).DrawLine(
        new Pen(Color.Red, 2),
        new Point(10,10), new Point(120,180));
```

وهناك صيغة أخرى للوسيلة DrawLine كالتالي:

```
this.CreateGraphics( ).DrawLine(
      new Pen(Color.Red, 2), 10, 10, 120, 180);
```

حيث أرسلنا إحداثيات الخطّ منفردة، بدلا من إرسالها كنقطتي بداية ونهاية، وإنّ كانت الصيغة الأولى أوضح عند قراءتها من الصيغة الثانية.

أهمّ الكائنات المستخدمة في الرسم

تعتبر الكائنات التالية، كائنات أوّليّة في عمليّة الرسم:

كائن النقطة Point Object:

يمثّل هذا الكائن إحداثي نقطة:

- الإحداثيّ السينيّ X (موضع النقطة أفقيّا،بدءا من أقصى يسار الشاشة).
- والإحداثيّ الصادي Y (موضع النقطة رأسيّا، بدءا من أعلى الشاشة).

ولإنشاء نقطة جديدة، يجب أن تحدد إحداثييها (الأفقيّ ثمّ الرأسيّ) كالتالي:

```
Point P1 = new Point( );
P1.X = 34;
P1.Y = 50;
```

أو باختصار:

```
Point P1 = new Point(34 50);
```

ويقبل هذا الكائن الإحداثيات في صورة أعداد صحيحة Integers، فإذا أردت استخدام وحدة قياس غير الوحدة Pixel، فقد تكون قيم الإحداثيات أعدادا مفردة Single.. في هذه الحالة استخدم الكائن PointF، وهو مماثل لكائن النقطة Point، إلا إنّه يقبل الإحداثيات من النوع float.

كائن الحجم Size Object:

مشابه لكائن النقطة، ولكنّه يمتلك خاصيتين: العرض Width، والارتفاع Height، ويتمّ تعريفه كالتالي:

```
Size S = new Size(100, 20);
```

أو بطريقة أخرى:

```
Size S = new Size(new Point(100, 20));
```

وهناك نوع آخر من كائن الحجم، هو كائن الحجـــم ذو الدقّـــة العشـــريّة SizeF، وهو مماثل لكائن الحجم Size، إلا إنّه يقبل معاملات من النـــوع float.

كائن المستطيل Rectangle Object:

يتطلّب تعريف المستطيل، موقع رأسه العلويّ الأيمن وعرضه وارتفاعه:

```
Rectangle box = new Rectangle(X, Y, العرض, الارتفاع);
```

وهناك صيغة أخرى باستخدام كائن النقطة وكائن الحجم:

```
Rectangle box = new Rectangle(new Point(1,1),
                              new Size(100,20)S);
```

كما يمكنك أن تنشئ مستطيلا، ثمّ تضع قيم خصائصه كالتالي:

```
var box = new Rectangle( );
box.X = 1;
box.Y = 1;
box.Width = 100;
box.Height = 20;
```

وهناك نوع آخر من كائن المستطيل، هو كائن المستطيل ذو الدقّة العشريّة RectangleF، وهو مماثل للمستطيل Rectangle، إلا إنّه يقبل معاملات من النوع float.

كائن القلم Pen Object:

يتطلّب تعريف هذا الكائن اللون والسُّمك (ولو حذفت السمك من التعريف فسيتمّ اعتباره ١):

Pen P = new Pen(Color.Black, 3);

وهناك صيغة أخرى تسمح لك بتعريف اللون من كائن فرشاة كالتالي:

var patternPen = new Pen(brush, width);

وفي هذه الحالة يمكنك استخدام هذا القلم لتلوين الحدّ الخارجيّ لشكل بطريقة معيّنة (تحدّدها الفرشاة).

ولديك مجموعة من الأقلام الجاهزة محدّدة الألوان وسمكها ١، يمنحها لك مرقّم الأقلام Pens Structure.. مثل القلم الأزرق Pens.Blue.. ويمكنك أن تستخدم هذه الأقلام مباشرة بدون تعريف، كمعامل في أيّ وسيلة تتطلّب قلما.

خصائص القلم:

اتصّال الخط LineJoin:

تحدّد الطريقة التي ستلتحم بها الأشكال المتجاورة.

StartCap, EndCap:

تحددان شكل بداية ونهاية الخطّوط المرسومة.

DashCap

تحدّد شكل بداية ونهاية التقطيعات في الخطّ المتقطّع.

طراز التقطيع DashStyle:

استخدم هذه الخاصيّة لرسم الخطّ متقطّعا بالطراز الذي تريده.. وتأخذ هذه الخاصيّة قيمة من قيم المرقّم DashStyle وهي:

متّصل Solid ــ شرطة Dash ــ منقّط Dot

شرطة ونقطة DashDot ــ شرطة ونقطتان DashDotDot

مخصّص Custom.

نوع القلم PenType:

تحدّد نوع القلم، الذي هو أحد القيم التالية:

ملء مظلّل HatchFilled ــ لون متدرّج LinearGradient

لون متشعّب التدرّج PathGradient ــ لون ثابت SolidColor

ملء بخامة TextureFill.

ستتساءل هنا، لماذا تتكلّم هذه القيم عن ملء الشكل باللون، مع أنّها خاصّة بالقلم وليس بالفرشاة؟

لا تنسَ أنّ القلم يرسم خطوطا، يمكن أن تكون سميكة.. ونحن نتكلّم عن كيفيّة ملء هذه الخطوط السميكة بالألوان.

كائن الفرشاة Brush Object:

تمكّنك الفرشاة Brush من ملء الأشكال بلون واحد أو بألوان متدرّجة أو بصورة.

ولا يمكنك تعريف فرشاة منفردة، فهذه الفئة ترثها أنواع الفرش المختلفة، وهي:

الفرشاة الصلبة SolidBrush:

تملأ الشكل بلون واحد.

```
SolidBrush sBrush = new SolidBrush(لون الفرشاة);
```

وسنتعرّف عليها بالتفصيل لاحقا.

فرشاة التظليل HatchBrush:

تملأ الشكل بتشكيلة معيّنة، من أكثر من ٥٠ تشكيلة جاهزة، يمكنك اختيارها من خلال المرقّم HatchStyle.. ويتمّ تعريف هذه الفرشاة كما يلي:

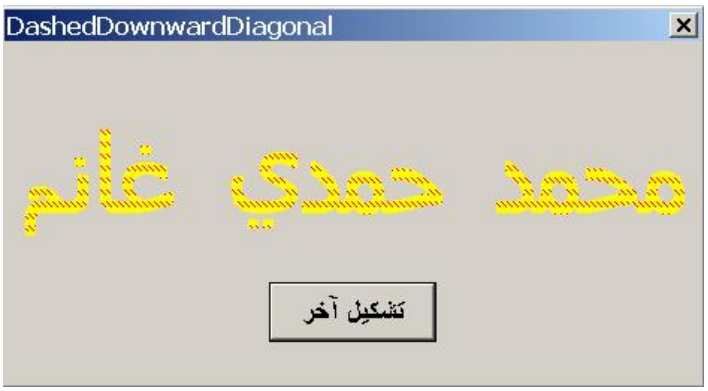
```
HatchBrush hBrush = new HatchBrush (طراز الفرشاة,
                    لون الخلفيّة, لون خطوط التظليل);
```

وهذه بعض تشكيلات التظليل، التي يمكنك استخدامها كقيم للمعامل الأوّل:

ForwardDiagonal	خطوط قطريّة من أعلى اليسار إلى أسفل اليمين.
BackwardDiagonal	خطوط قطريّة من أعلى اليمين إلى أسفل اليسار.

Cross	خطوط أفقيّة ورأسيّة متقاطعة.
DiagonalCross	خطوط قطريّة متقاطعة.
Horizontal	خطوط أفقيّة.
Vertical	خطوط رأسيّة.

ويمكنك استعراض هذه التشكيلات المختلفة، في المشروعين HatchBrushEnum و HatchBruchText في مجلّد برامج هذا الفصل.



الفرشاة المتدرّجة LinearGradientBrush:

تملأ الشكل بألوان متدرّجة.. ويبدأ التدرّج من لون ما وينتهي إلى لون آخر.

وسنتعرّف عليها بالتفصيل لاحقا.

فرشاة متشعّبة التدرّج PathGradientBrush:

تملأ مسار الشكل بألوان متشعّبة التدرّج، حيث يبدأ التدرّج بلون واحد، ولكنّه ينتهي إلى ألوان مختلفة.

وسنتعرّف عليها بالتفصيل لاحقا.

فرشاة الخامة TextureBrush:

تملأ الشكل بصورة.. ويتمّ تبليط Tiling الشكل بالصورة ليملأها كلّيا، بمعنى أنّ الصورة تتكرّر على حسب ما يتطلّب الأمر.. تخيّل مدى روعة الشكل الناتج من ملء دائرة أو منحنى بصورة معيّنة!

ملحوظة١:

في أيّ نوع من أنواع الفرش السابقة (والأقلام ومسار الرسوم كذلك) يمكن استخدام معامل الشفافية في تكوين الألوان المرسلة كمعاملات لهذه الفئات، وبهذا يمكنك الحصول على أشكال شفافة بأيّ نسبة تريدها.. فمثلا: لو لوّنت أيّ شكل بالفرشاة التالية، فسيكتسب لونا أحمر نصف شفّاف، حيث سيمتزج لونه بلون خلفيّته:

```
SolidBrush sBrush = new SolidBrush(
        Color.FromArgb(128, 255, 0, 0));
```

ملحوظة٢:

يمنحك سجلّ الفرش Bruches Structure مجموعة من الفرش الجاهزة بألوان مختلفة.. مثل Brushes.Red التي تمنحك فرشاة تلوين حمراء.

كائن الرسوم Graphics Object:

أدوات كثيرة لا تتوقّعها تمنحك كائن رسوم، ممّا يمكّنك من الرسم عليها.. خذ مربّع النصّ مثلا!

ولكي تحصل على كائن الرسوم من الأداة، استخدم الوسيلة CreateGraphics الخاصة بها.

ولكي تستخدم كائن الرسوم، ضمّن المكتبة Drawing2D في مشروعك أولا:

```
using System.Drawing.Drawing2D;
```

ولتعريف متغيّر من هذا الكائن، لاستخدامه للرسم على مربّع الصورة، استخدم الجملة التالية:

```
Graphics G = PictureBox1.CreateGraphics( );
```

ملحوظة:

يرتبط كائن الرسوم عند إنشائه بمساحة سطح النموذج أو الأداة، فإذا تغيّرت هذه المساحة لا يشعر كائن الرسوم بهذا التغيير.. لهذا فإنّ أنسب مكان تعرّف فيه هذا الكائن بالنسبة للنموذج هو حدث رسم النموذج Paint event، حتّى يعاد إنشاء كائن الرسوم كلّما حدث تغيير للنموذج يستدعي إعادة رسمه.

خصائص كائن الرسوم:

DpiX و DpiY:

تمثّل هاتان الخاصيتان عدد النقاط في كلّ بوصة Dots per inch (Dpi) أفقيّا (على المحور X) ورأسيّا (على المحور Y).

وحدات الصفحة PageUnit:

في الوضع التلقائيّ، تقاس كلّ الأطوال بالنقطة Pixel.. فإذا أردت تغيير ذلك، فاختر لهذه الخاصيّة أيّ وحدة قياس أخرى من المرقّم GraphicsUnit، الذي قيمه كالتالي:

Document	١,٣ بوصة.
Inch	بوصة.
Millimeter	ملليمتر.
Pixel	نقطة على شاشة الكمبيوتر (وهي القيمة الافتراضيّة).
Point	نقطة في ورقة الطباعة، وهي تساوي ١,٧٢ بوصة.
World	حدّد أنت القيمة التي تريد القياس بها.

كيفية رسم النصوص TextRenderingHint:

تأخذ هذه الوسيلة واحدة من قيم المرقّم TextRenderingHint، لإضافة تأثيرات معيّنة تحسّن من شكل النصّ المرسوم.

طراز التنعيم SmoothingMode:

مماثلة للخاصيّة السابقة، ولكنّها تنطبق على كلّ الأشكال، وليس النصوص فقط.. وهي تأخذ قيمة من قيم المرقّم SmoothingMode.

رسم الأشكال بوسائل كائن الرسوم:

يمتلك كائن الرسوم العديد من الوسائل التي تمكّنك من رسم الأشكال.. ولكي نشرح هذه الوسائل، تعالَ نعرّف كائن رسوم يرسم على النموذج، وكائن قلم أزرق:

```
Graphics G = this.CreateGraphics( );
var P = new Pen(Color.Blue);
```

يرجع احتياجنا للقلم إلى أنّ أوّل معامل في وسائل رسم الأشكال هو القلم.. وطبعا باقي المعاملات هي إحداثيات الشكل.

دعنا نبدأ برسم مستطيل:

```
G.DrawRectangle(P, 10, 10, 200, 150);
```

والآن سنرسم قُطرَي المستطيل:

```
G.DrawLine(P, 10, 10, 210, 160);
G.DrawLine(P, 210, 10, 10, 160);
```

جرّب ذلك في المشروع SimpleShapes.

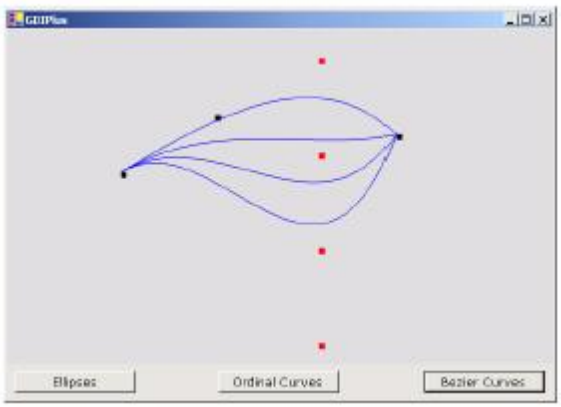
والجدول التالي يوضّح لك الوسائل التي ترسم الأشكال المختلفة (تلك التي تبدأ بالمقطع Draw)، والوسائل التي تملأ كل شكل منها بلون معيّن ــ إذا كان الشكل مغلقا بالطبع ــ (وهي تبدأ بالمقطع Fill)، مع ملاحظة أنّ استخدام وسيلة الملء يملأ الشكل إذا كان موجودا على النموذج، أو ترسمه ممتلئا باللون إذا لم يكن موجودا من قبل.

DrawArc:

ترسم قوسا.

DrawBezier:

ترسم منحنى بيزيير، وهو منحنى له نقطتا بداية ونهاية، ويمرّ بنقطتين أخريين تتحكّمان في انحنائه، كما في الشكل:



DrawBeziers:

ترسم مجموعة من منحنيات بيزيير معا.

DrawClosedCurve و **FillClosedCurve**:

لرسم منحنى مغلق.

DrawCurve:

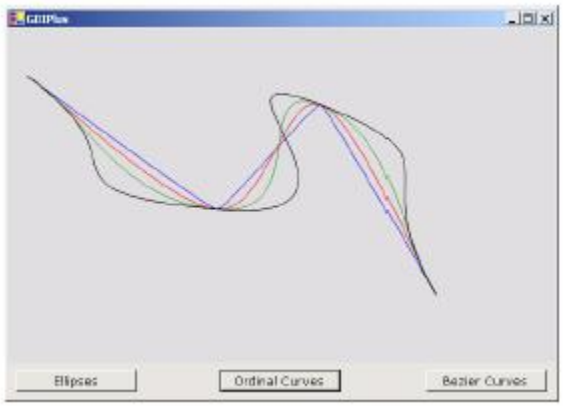
ترسم منحنى يمرّ بنقاط معيّنة (على الأقلّ ٤ نقاط، بداية ونهاية ونقطتان على المنحنى):

Graphics.DrawCurve(القلم, _

(درجة الانحناء, مصفوفة من النقاط التي يمرّ بها المنحنى);

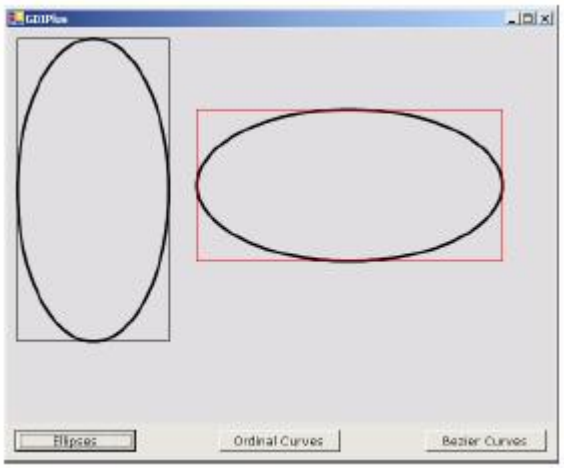
ولو لم ترسل المعامل الأخير، فسيتمّ رسم المنحنى بدرجة انحناء ١.

والشكل التالي يبيّن مجموعة من المنحنيات التي تمرّ بنفس النقاط، ولكنّ لكلّ منها درجة انحناء مختلفة:



DrawEllipse و **FillEllipse:**

لرسم قطع ناقص، بمعرفة المستطيل الذي يحتويه.. والقطع الناقص هو شكل بيضاويّ له قطران غير متساويين (مساويان لضلعي المستطيل الذي يحتويه).



ويمكنك أن تستخدم هذه الوسيلة لرسم الدوائر، وذلك برسم القطع الناقص داخل مربّع، وفي هذه الحالة سيكون طول ضلع المربّع مساويا لقطر الدائرة..والجملة التالية ترسم دائرة زرقاء نصف قطرها ٥٠:

```
G.DrawEllipse(Pens.Blue, _
    new Rectangle(50, 20, 100, 100));
```

ويقع مركز هذه الدائرة في مركز المربّع الذي هو (٥٠ + [١٠٠] ÷ ٢ ، ٢٠ + [١٠٠] ÷ ٢)، أي النقطة (١٠٠ ، ٧٠).

وكقاعدة: يقع مركز الدائرة عند النقطة:

(س + نصف طول ضلع المربّع ، ص + نصف طول ضلع المربّع).

DrawIcon:

ترسم أيقونة داخل كائن الرسوم.. هذه الأيقونة سيتمّ مطّها لتملأ مساحة كائن الرسوم.

DrawIconUnstretched:

ترسم أيقونة داخل كائن الرسوم، ولكن دون مطّها.

DrawImage:

ترسم صورة داخل كائن الرسم، مع مطّها لتملأ مساحة كائن الرسوم.

DrawImageUnscaled:

ترسم صورة داخل كائن الرسم، ولكن دون تغيير حجمها.

DrawLine:

ترسم خطّا يصل بين نقطتين.

DrawLines:

ترسم مجموعة من الخطوط.

DrawPath و FillPath:

لرسم جميع الأشكال الموجودة في كائن مسار الرسوم GraphicsPath object.. وسنتعرّف عليه لاحقا.

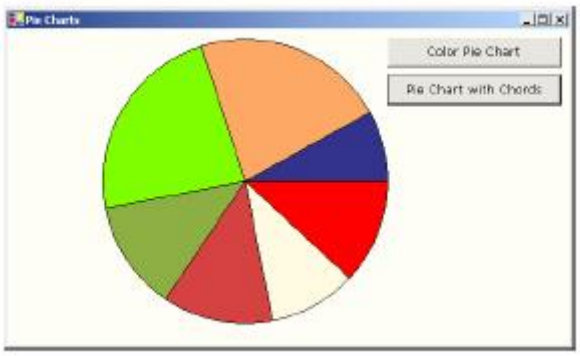
DrawPie و FillPie:

لرسم شريحة من قطع ناقص (أو دائرة)، عبارة عن قوس وضلعين يصلان نهايتي القوس بمركز الدائرة.. ولرسم هذا القوس أرسل لهذه

الوسيلة المستطيل الذي يحتوي على القطع الناقص، مـــع زاويـــة بدايـــة القوس، وزاوية انفراج ضلعي القوس.. مثال:

```
G.DrawEllipse(Pens.Blue, _
    new Rectangle(50, 20, 100, 300), 50, 30);
```

وأهمّ استخدام لهذا الشكل هو الرسوم البيانيّة التي توضّح النسب كشرائح ملوّنة من دائرة (مثل الرسم الذي يوضّح لك المساحة المشغولة والمساحة الفارغة من القرص الصلب في الويندوز).



DrawPolygon و FillPolygon:

لرسم مضلّع مغلق له أيّ عدد من الرءوس.. هذه الرءوس ترسلها إلـــى هذه الوسيلة في صورة مصفوفة من النقاط.

DrawRectangle و FillRectangle:

لرسم مستطيلا.

DrawRectangles و FillRectangles:

لرسم مجموعة من المستطيلات.

DrawString:

ترسم نصّا بخطّ معيّن.

FillRegion:

لملء كائن من النوع Region بلون معيّن.

ولن تجد صعوبة في استخدام هذه الوسائل.. فقط لاحظ ما يلي:

- أوّل معامل في دوال رسم الأشكال هو كائن القلم Pen Object، وذلك لتحديد لون الخطّ وسمكه.
- أوّل معامل في دوال ملء وتلوين الأشكال المغلقة هو كائن الفرشاة Brush Object، وذلك لتحديد لون التلوين وطرازه.
- باقي معاملات وسائل الرسم والتلوين توضّح موضع الشكل ومقاييسه.
- الوسائل التي ترسم أكثر من شكل في نفس الوقت، تتطلّب ــ كمعامل ــ مصفوفة تحمل مواضع ومقاييس هذه الأشكال.. فمثلا: الوسيلة DrawRectangles تقبل مصفوفة تحتوي على كائنات من النوع Rectangle objects.

كائن مسار الرسوم GraphicsPath Object:

يمثّل هذا الكائن رسما مغلقا يتكوّن من مجموعة من الخطوط والمستطيلات والمنحنيات.

وأبسط طريقة لتكوين هذا الكائن، هي تعريف نسخة جديدة منه، واستخدام وسائله التالية لإضافة الرسومات إليه:

إضافة قوس **AddArc** ــ إضافة قطع ناقص **AddEllipse** ــ إضافة مضلّع **AddPolygon** ــ إضافة منحنى بيزير التكعيبيّ **AddBezier** ــ إضافة خطّ **AddLine** ــ إضافة مستطيل **AddRectangle** ــ إضافة منحنى **AddCurve** ــ إضافة منحنى مغلق **AddClosedCurve** ــ إضافة شريحة من دائرة **AddPie** ــ إضافة نصّ **AddString**.

وهناك المزيد من الوسائل تسمح لك بإضافة عدد من الأشكال دفعة واحدة، مثل:

إضافة منحنيات بيزير **AddBeziers** ــ إضافة خطوط **AddLines** ــ إضافة مستطيلات **AddRectangles**.

ليس هذا فحسب، بل يمكنك إضافة مسار رسومات آخر إلى المسار الحاليّ باستخدام الوسيلة **AddPath**.

طبعا لاحظت أنّ أسماء هذه الوسائل تتشابه مع أسماء وسائل الرسم بكائن الرسوم Graphics.. بل إنّهما يتشابهان في المعاملات.. والاختلاف الوحيد بينهما، هي أنّ وسائل الرسم في كائن الرسوم لها معامل زائد، هو المعامل الأوّل، الذي تحدّد فيه القلم الذي سترسم به.. ونظرا لأنّ كائن مسار الرسوم لا يرسم هذه الرسوم على النموذج أو الأدوات بنفسه، فإنّك لا تحتاج لتحديد القلم له.. ولرسم الأشكال الموجودة في هذا المسار

استخدم الوسيلة DrawPath الخاصّة بكائن الرسوم Graphics Object.. حينئذٍ يمكنك تحديد القلم الذي سترسم به.

وفي المثال التالي ننشئ مسار رسوم ونضيف له قطعا ناقصا ومنحنى بيزير:

```
var myPath = new GraphicsPath( );
myPath.AddEllipse(new Rectangle(10, 30, 40, 50));
mypath.AddPie(100, 50, 100, 200, 90, 10);
this.CreateGraphics( ).DrawPath(Pens.Aqua, myPath);
```

هناك فوائدَ جمّةً لكائن مسار الرسوم، فقدرتك على رسم شكل معقّد يتكوّن من مجموعة مختلفة من الأشكال، والتعامل معه باعتباره شكلا واحدا، يمنحك القدرة على تكرار رسم هذا الشكل لأيّ عدد من المرّات باستدعاء الوسيلة DrawPath، مع قدرتك على تغيير القلم (اللون والسمك) في كلّ مرّة، وقدرتك على ملء الشكل بألوان مختلفة أو صور مختلفة في كلّ مرّة.

وكذلك يُستخدم مسار الرسوم في إنشاء كائن مسار متدرّج اللون PathGradient، وهو ما يسمح لك برسم أشكال ساحرة، كما سنرى فيما بعد.

الألوان المتدرّجة Gradients

لدينا نوعان من التدرّج:

- المتدرّجات الخطّيّة Linear Gradients:
- المتدرّجات المتشعّبة Path Gradients:

المتدرّجات الخطّيّة Linear Gradients:

يتمّ إنشاء هذا النوع من الألوان المتدرّجة بالفرشاة LinearGradientBrush.. ابدأ بتعريف متغيّر من هذه الفرشاة كالتالي:

```
var lgBrush = new LinearGradientBrush(مستطيل,
        طراز التدرّج,لون النهاية,لون البداية);
```

هذه الفرشاة مؤهّلة لملء مساحة مماثلة لمساحة المستطيل المحدّد في المعامل الأوّل، بألوان تتدرّج من لون البداية حتّى تصل للون النهاية.. فإذا كان الشكل الذي ستملؤه الفرشاة أصغر من المستطيل، فإنّ جزءا فقط من التدرّج هو الذي سيملأ الشكل.. أمّا لو كان الشكل أكبر من المستطيل، فسيتمّ تكرار التدرّج حتّى يملأ كلّ الشكل.

ويحدّد طراز التدرّج اتّجاه التدرّج، ويمكن أن يأخذ قيمةً من القيم التالية:

BackwardDiagonal	تدرّج قطريّ من أعلى اليمين إلى أسفل اليسار.
ForwardDiagonal	تدرّج قطريّ من أعلى اليسار إلى أسفل اليمين.
Horizontal	تدرّج أفقيّ.
Vertical	تدرّج رأسيّ.

والجزء التالي من الكود موجود في المشرع GDIPlusGradients فـــي مجلّد برامج هذا الفصل، وهو يملأ مستطيلين بألوان متدرّجة:

```
Graphics G = this.CreateGraphics( );
G.Clear(this.BackColor);
RectangleF R = new RectangleF(20, 20, 300, 100);
Color startColor = Color.BlueViolet;
Color EndColor = Color.LightYellow;
var LGBrush = new LinearGradientBrush(R,
                    startColor, EndColor,
                    LinearGradientMode.Horizontal);
var R1 = new Rectangle(20, 20, 200, 100);
G.FillRectangle(LGBrush, R1);
var R2 = new Rectangle(20, 150, 600, 100);
G.FillRectangle(LGBrush, R2);
```

وتوجد عدّة صيغ أخرى لتعريف هذه الفرشاة.. فمثلا، يمكنـــك اســـتخدام نقطتين بدلا من المستطيل، هاتان النقطتان ســـتمثلان رأســـي المســـتطيل المتقابلين (العلويّ الأيسر، والسفليّ الأيمن).. وهما في نفس الوقت تحدّدان اتجاه التدرّج، حيث سيكون في اتجاه الخطّ الواصل بين النقطتـــين (ممّـــا سيتيح لك اتجاهات غير ممكنة في صيغة المستطيل وطراز التدرّج).

```
var lgBrush = new LinearGradientBrush(نقطة البداية,
                    لون النهاية, نقطة النهاية, لون البداية);
```

كما أنّ هناك صيغة تتيح لك تحديد زاوية التدرّج، بدلا من طراز التدرّج:

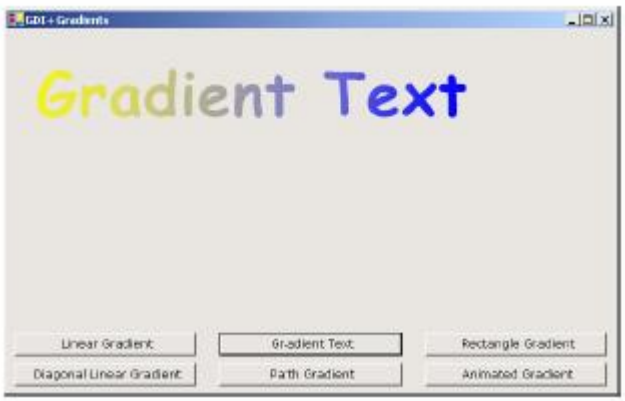
```
var lgBrush = new LinearGradientBrush(مستطيل,
                    زاوية التدرّج, لون النهاية, لون البداية);
```

حيث تقاس هذه الزاوية بالنسبة للمحور الأفقيّ، في اتجاه عقارب الساعة.

ويمكنك إضافة معامل أخير على الصيغة السابقة.. لو جعلته True، فستتأثّر الزاوية بالتحويلات Transform المجراة على الإحداثيات (حيث يتمّ قياسها بالنسبة للمحاور بعد تدويرها)..أو False إذا كنت تريد أن يتمّ قياس الزاوية بطريقة مستقلّة.

ويمكن استخدام الألوان المتدرّجة لملء المنحنيات والمضلّعات الأخرى.. في هذه الحالة يتمّ ملء المستطيل الذي يحتوي المنحنى أو المضلّع، ولكن لا يتمّ عرض إلا الجزء الواقع داخل الشكل.

والطريف أنّك تستطيع رسم نصّ على النموذج بألوان متدرّجة.. هذا الكود أيضا موجود في مشروع GDIPlusGradients وهو يفعل ذلك:




```
Graphics G = this.CreateGraphics( );
// مسح أيّ رسوم على النموذج، مع المحافظة على لون الخلفيّة
G.Clear(this.BackColor);
G.TextRenderingHint = TextRenderingHint.AntiAlias;
Font largeFont = new Font("Comic Sans MS", 48,
        FontStyle.Bold, GraphicsUnit.Point);
PointF gradientStart = new PointF(0, 0);
string txt = "Gradient Text";
// معرفة مساحة النصّ
SizeF txtSize = G.MeasureString(txt, largeFont);
```

```
PointF gradientEnd = new PointF(txtSize.Width,
        txtSize.Height);
var grBrush = new LinearGradientBrush(gradientStart,
        gradientEnd, Color.Yellow, Color.Blue);
G.DrawString(txt, largeFont, grBrush, 20, 20);
```

الجدير بالذكر، أنّ المتدرّج الخطّي يمنحك هذه الإمكانيّات الإضافيّة:

١- استخدام أكثر من لونين للتدرّج:

وذلك باستخدام خاصيّة InterpolationColors.

٢- تحديد كيفيّة مزج الألوان:

وذلك باستخدام فئة المزج Blend Class أو الوسيلتين SetSigmaBellShape و SetBlendTriangularShape .

٣- تحويل إحداثيات المتدرّج:

وذلك باستخدام خاصيّة Transform.

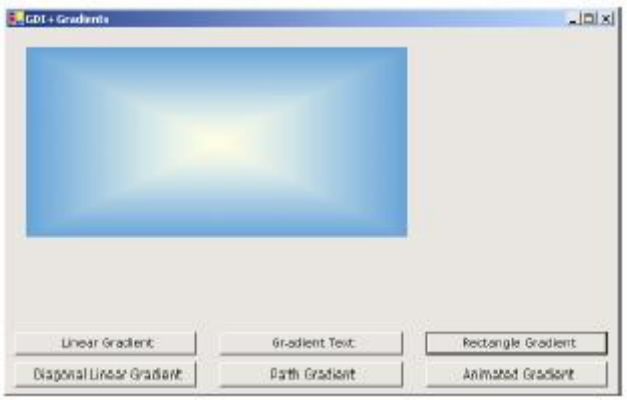
٤- إنشاء متدرّج يبدأ من لون في المركز، ويتدرّج إلى لون آخر على الطرفين.

٥- إنشاء متدرّج خطّي غير منتظم التدرّج Non-uniform Linear Gradient.

ويمكنك التعرّف على كلّ هذه الإمكانيّات، بالاستعانة بملفات المساعدة المرفقة باللغة.. ألا تعتقد أنّ الوقت قد حان لتعتمد على نفسك قليلا؟

المتدرّجات المتشعّبة Path Gradients:

يمكنك استخدام الفرشاة PathGradientBrush لرسم ألوان متدرّجة، تبدأ من لون معيّن، وتنتشر في اتجاهات مختلفة، لتنتهي في كل اتجـــاه منهـــا بلون مختلف.



ولاختبار هذه الإمكانية الرائعة، افحص المشروع GDIPlusGradients.

وهذه الفرشاة تناسب تلوين كائن مسار الرســـوم GraphicsPath، لهـــذا فإنّك تحدد لها الكائن الذي ستستخدم في تلوينه عند تعريفها:

```
var pgBrush = new LinearGradientBrush(GraphicsPath);
```

والآن استخدم الخاصيتين التاليتين من خصائص هذه الفرشاة لتحديد ألوان التدرّج:

لون المركز CenterColor:

لتحديد لون المركز، الذي يبدأ منه التدرّج.. ولو لم تحدّد هذه الخاصـــيّة، فستكون قيمتها الافتراضيّة هي مركز الشكل الذي سيتمّ تلوينه.

الألوان المحيطة SurroundColors:

مصفوفة من الألوان بعدد رءوس الأشكال الموجودة في كائن مسار الرسوم.. مثال:

```
Color[ ] Colors = {Color.Yellow, Color.Green,
                   Color.Blue};
pgBrush.SurroundColors = Colors;
```

وبعد أن تنشئ كائن الفرشاة وتضبط خصائصه، استدعِ الوسيلة FillPath الخاصّة بكائن الرسوم لتلوين الأشكال الموجودة في مسار الرسوم.

والمثال التالي يريك كيف تلوّن مستطيلا بألوان متدرّجة من الأحمر في مركزه، إلى الأصفر والأخضر والأزرق والـ Cyan في أطرافه.. ويمكنك تجربة هذا المثال في مشروع GDIPlusGradients.. لاحظ أنّنا رسمنا المستطيل كأربعة خطوط مغلقة وأضفناها لمسار الرسوم:

```
Graphics G = this.CreateGraphics( );
G.Clear(this.BackColor);
GraphicsPath path = new GraphicsPath( );
path.AddLine(new Point(10, 10), new Point(400, 10));
path.AddLine(new Point(400, 10), new Point(400, 250));
path.AddLine(new Point(400, 250), new Point(10, 250));
var pathBrush = new PathGradientBrush(path);
Color centerColor = Color.Red;
Color[ ] surroundColors = { Color.Yellow,
        Color.Green, Color.Blue, Color.Cyan };
pathBrush.CenterColor = centerColor;
pathBrush.SurroundColors = surroundColors;
G.FillPath(pathBrush, path);
```

ولا يوجد ما يمنع أن تضيف مستطيلا (أو أيّ شكل آخر) لكائن مسار الرسوم ثم تلوّنه بالفرشاة متشعّبة التدرّج.

تحويل الإحداثيّات Coordinate Transformations:

لدينا ثلاثة أنواع من تحويلات الإحداثيّات:

١ - تغيير مقياس الرسم Scaling:

بحيث يتغيّر حجم الشكل تكبيرا أو تصغيرا.. وتقوم بذلك الوسيلة ScaleTransform من وسائل كائن الرسوم Graphics Object:

Graphics.ScaleTransformation(نسبة التكبير أفقيّا,
نسبة التكبير رأسيّا);

فإذا كانت النسبة أصغر من واحد كانت العمليّة تصغيرا، وإذا كانت أكبر من ١ كانت تكبيرا.

٢ - تغيير الموضع Translation:

بحيث نحرّك الشكل من موضعه.. وتقوم بذلك الوسيلة TranslateTransform من وسائل كائن الرسوم:

Graphics.TranslateTransform(مقدار الانتقال أفقيّا,
مقدار الانتقال رأسيّا);

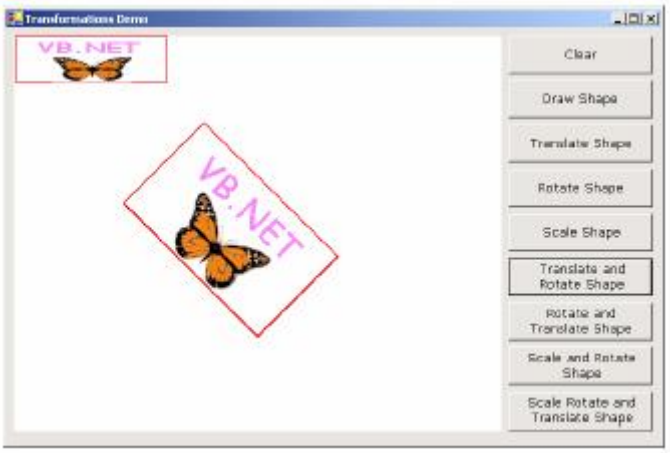
فإذا كان مقدار الانتقال موجبا انتقل الشكل لليمين أو لأسفل، وإذا كان سالبا، انتقل الشكل لليسار أو لأعلى.

١ - الدوران Rotation:

بحيث ندير الشكل بأيّ زاوية.. وتقوم بذلك الوسيلة RotateTransform من وسائل كائن الرسوم.

Graphics.RotateTransform(زاوية الدوران);

وتنحصر زاوية الدوران بين ٠ و ٣٦٠ درجة.



لاحظ أنّ هذه التحويلات تراكميّة، فمثلا: تدوير الشكل ٣٠ درجة، ثمّ تدويره ٤٠ درجة، يكافئ تدويره ٧٠ درجة مباشرة.

هذا التراكم ناتج من أنّ هذه التحويلات يتمّ حفظها في مصفوفة تسمّى "مصفوفة التحويلات" Transformation matrix.

فإذا أردت أن تعيد الإحداثيّات لوضعها الأصليّ، فاستخدم الوسيلة ResetTransform.

ويمكنك اختبار هذه التحويلات في المشروع GDIPlusTransformations.

قص منطقة الرسم Clipping:

إذا أردت أن تقصّ منطقة من النموذج أو الأداة لتكون هي المنطقة التي يرسم عليها كائن الرسوم، فاستخدم الوسيلة SetClip الخاصّة بكائن الرسوم Graphics object.

وأوّل معامل تأخذه هذه الوسيلة، يحدّد منطقة الرسم.. في هذا الحالة يمكنك أن تجعل المعامل مستطيلا.. فإذا شئت أن تجعل للمنطقة أيّ شكل أكثر تعقيدا، فاستخدم كائن مسار الرسوم GraphicsPath، أو منطقة Region (المنطقة هي سجلّ يتكوّن من أشكال بسيطة، حيث تتكوّن المنطقة من اتحاد هذه الأشكال، أو من تقاطعها، أو من فرق أحدها من الآخر، أو من المنطقة التي تقع خارج التقاطع).

وهناك معامل آخر اختياريّ، يحدّد كيف ستتم إضافة هذه المنطقة للنموذج أو المنطقة المعرّفة سابقا في النموذج.. ويأخذ هذا المعامل قيمة من قيم المرقّم CombineMode، وهي:

Complement	منطقة الرسم الجديدة هي التي تقع خارج اتحاد المنطقتين.
Exclude	منطقة الرسم الجديدة هي التي تقع في المنطقة الجديدة وحدها، خارج منطقة التقاطع.

Intersect	منطقة الرسم الجديدة هي منطقة التقاطع.
Replace	المنطقة الجديدة ستكون بديلة لسابقتها.
Union	منطقة الرسم الجديدة هي ناتج اتّحاد المنطقتين.
XOR	منطقة الرسم الجديدة هي ناتج اتحاد المنطقتين ما عدا منطقة تقاطعهما.

وبعد تحديد منطقة الرسم، فإنّ أيّ شكل ترسمه لن يُرسم إلا في هذه المنطقة، وأيّ تحويل للإحداثيّات لن ينطبق إلا على هذه المنطقة.

ولإزالة المنطقة المقصوصة، والعودة للرسم على النموذج كلّه، استخدم الوسيلة ResetClip.

ويريك مشروع Clipping، كيف تعرض النصّ والصورة داخل قطع ناقص.

رسم النصوص:

لرسم نصّ على نموذج أو أداة، استخدم الصيغة التالية:

Graphics.DrawString(النصّ, الخطّ, الفرشاة, X, Y);

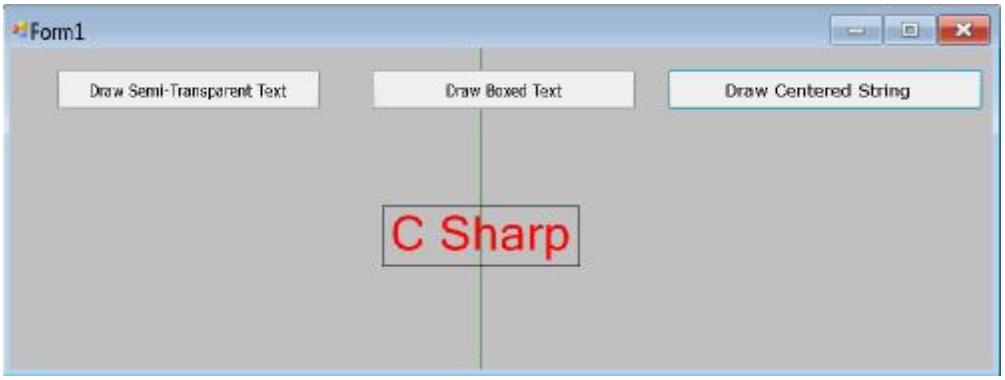
حيث X و Y يمثّلان نقطة بداية الكتابة (أعلى نقطة يسارا في المنطقة التي تحتوي النصّ).

ولمعرفة المساحة التي يشغلها النصّ على النموذج أو الأداة، استخدم الوسيلة "قياس النصّ" MeasureString، حيث تعيد لك كائن حجم بدقّة عشريّة SizeF، بحيث يمكنك استخدام خاصيتيه العرض والارتفاع لمعرفة مساحة النصّ.

والمثال التالي يوضّح لك كيف النصّ في مركز النموذج تماما:

```
var textSize = G.MeasureString("C Sharp", this.Font);
int X = (int)(this.Width – textSize.Width) / 2;
int Y = (int)(this.Height – textSize. Height) / 2;
G.DrawString("C Sharp", this.Font, brush, X, Y);
```

ويمكنك تجربة ذلك في المشروع TextEffects في برامج هذا الفصل.



وهناك صيغة أخرى ترسم النصّ داخل المستطيل الذي تحدّده لها، حتّى لو اقتضى الأمر كتابة النصّ على أكثر من سطر داخل المستطيل (إذا كان النصّ أطول من عرض المستطيل):

```
Graphics.DrawString(النصّ, الخطّ, الفرشاة, مستطيل);
Graphics.DrawString(النصّ, الخطّ, الفرشاة, مستطيل, تنسيق النصّ);
```

ولمعرفة عدد السطور التي كتب فيها النصّ، استخدم الوسيلة MeasureString.. طبعا ستتعجب: ألم نستخدم هذه الوسيلة لمعرفة أبعاد النصّ؟

نعم، وهناك صيغة أخرى منها تنتهي بمعاملين مرجعيّين، يرجعان لك عدد سطور النصّ، وعدد الأعمدة التي كتب عليها:

```
e.Graphics.MeasureString(النصّ, الخطّ,
    كائن حجم به أبعاد المستطيل الذي يحتوي النصّ _
    تنسيق النصّ, عدد السطور, عدد الأعمدة);
```

ويمكنك تجربة ذلك في مشروع TextEffects في مجلّد برامج هذا الفصل.

ولكن ما هو المعامل تنسيق النصّ، ذلك الذي ظهر في أكثر من وسيلة؟ هذا المعامل من النوع StringFormat، وهو يمنحك الخصائص التالية:

محاذاة النصّ Alignment:

تأخذ هذه الخاصيّة واحدة من قيم المرقّم StringAlignment، وهي:

Center	يتم توسيط النصّ في مركز المستطيل.
Far	تتمّ محاذاة النصّ بعيدا عن ركن المستطيل العلويّ الأيسر.
Near	تتمّ محاذاة النصّ قريبا من ركن المستطيل العلويّ الأيسر.

قصّ النصّ Trimming:

تستخدم هذه الخاصيّة لتوضيح كيف سيتمّ قصّ النصّ إذا تجاوز طوله حدود المستطيل.. وتأخذ هذه الخاصيّة القيم التالية:

Character	يتمّ قصّ النصّ عند أقرب حرف لحدود المستطيل.
EllipsisCharacter	يتمّ قصّ النصّ عند أقرب حرف لحدود المستطيل، وتوضع بعض النقاط في نهاية النصّ للدلالة على أنّ باقي النصّ غير ظاهر.
EllipsisPath	يتمّ حذف جزء من منتصف النصّ، وتوضع مكانه بعض النقاط.
Word	يتمّ قصّ النصّ عند أقرب كلمة لحدود المستطيل.
EllipsisWord	يتمّ قصّ النصّ عند أقرب كلمة لحدود المستطيل، وتوضع بعض النقاط في نهاية النصّ للدلالة على أنّ باقي النصّ غير ظاهر.
None	لا يتمّ قصّ النصّ.

مؤشّرات التنسيق FormatFlags:

يمكن لهذه الخاصيّة أن تأخذ قيمةً أو أكثر من قيم المرقّم StringFormatFlags، ومن أهمّها DirectionRightToLeft لتحديد أنّ اتجاه النصّ من اليمين لليسار، DirectionVertical لكتابة النصّ رأسيّا.

والمثال التالي يريك كيف تكتب رأسيّا على النموذج:

```
Graphics G = this.CreateGraphics( );
StringFormat SF = new StringFormat( );
SF.FormatFlags = StringFormatFlags.DirectionVertical;
G.DrawString("C Sharp", this.Font, Brushes.Red,
    80, 80, SF)
```

التعامل مع الصور

مربّع الصورة PictureBox:

تمنحك سي شارب أداةً متخصّصةً في عرض الصور والتعامل معها، تلك هي مربّع الصورة PictureBox.. ولكي تعرض صورةً فيه، استخدم الخاصيّة Image في نافذة الخصائص، حيث ستجد زر انتقال في خانة القيمة.. اضغط هذا الزر ليظهر لك مربّع حوار فتح ملفّ.. اختر الصورة التي تريد عرضها واضغط موافق.. هذه الصورة سيتمّ حفظها في ملف مصادر النموذج (وهو ملفّ يحمل نفس اسم النموذج ولكن امتداده ".resx").. لهذا لن تحتاج لتوزيع الصورة الأصليّة مع برنامجك، فهي محتواةٌ فيه بالفعل.

ومن خصائص مربّع الصورة الهامّة ما يلي:

طراز الحجم SizeMode:

تمكّنك هذه الخاصيّة من اختيار كيفيّة ظهور الصورة ومحاذاتها في مربّع الصورة.. ولهذه الخاصيّة القيم التالية:

Normal	تظهر الصورة طبيعية كما هي بدون تعديل.. فإذا كانت أكبر من مربّع الصورة، فسيختفي جزء منها، وإذا كانت أصغر، فلن تملأ كل مساحة مربّع الصورة.
CenterImage	سيتمّ عرض الصورة بحيث تتوسّط مربّع الصورة.

StretchImage	سيتمّ مطّ الصورة أو تقليصها بحيث تشغل مساحة مربّع الصورة بالضبط.. ورغم أنّ هذه الطريقة هي أفضل طريقة لعرض الصورة، إلا إنّ عليك أن تراعي تناسب أبعاد مربّع الصورة مع أبعاد الصورة، حتّى لا تتشوّه الصورة عند مطّها.. افحص مشروع ImageLoad في مجلّد برامج هذا الفصل، لترى كيف نحافظ على تناسب مقاييس الصورة.
AutoSize	سيتمّ تغيير أبعاد مربّع الصورة ليطابق أبعاد الصورة.. إنّ هذا الاختيار غير مفضّل، لأنّ وضع صورة كبيرة في مربّع الصورة سيجعله يغطّى باقي الأدوات الموجودة على النموذج.

طراز الحافّة BorderStyle:

وهي تأخذ قيم المرقّم BorderStyle وهي: ثابتة مجسّمة Fixed3D ــ ثابتة مفردة FixedSingle ــ بدون حافّة None.

الصورة Image:

تسمح لك بتحديد الصورة التي تعرضها في مربّع الصورة.

صورة الخلفيّة BackgroundImage:

تسمح لك بتحديد الصورة التي تعرضها في خلفيّة مربّع الصورة.

كائن الصورة Image Object

كلّ شيء في سي شارب ما هو إلا كائن، لهذا فمن الطبيعيّ أن يكون هناك كائن يمكنك من خلاله التعامل مع الصورة.

فمثلا، خاصيّة Image الخاصّة بمربّع الصورة تمثّل كائن صورة Image object.. هذا الكائن يمثّل الصورة الموجودة في مربّع الصورة، ويمنحك الخصائص والوسائل اللازمة للتعامل معه.

ويمكن أن تعرّف كائن صورة، وتضع فيه الصورة التي تحملها خاصيّة Image كالتالي:

```
Image img = PictureBox1.Image;
```

كما يمكنك أن تحمّل صورة من ملفّ وتضعها في كائن صورة عن طريق الوسيلة FromFile ثمّ تضعها في مربّع الصورة كالتالي:

```
Image img = Image.FromFile("Butterfly.jpg");
PictureBox1.Image = img;
```

خصائص كائن الصورة:

الخصائص التالية للقراءة فقط:

الدقّة الأفقيّة HorizontalResolution:

والدقّة الرأسيّة VerticalResolution:

تعيدان دقّة تمثيل الصور DPI (عدد النقط في كل بوصة Pixels -per-inch).

العرض Width والارتفاع Height:

واضحتان.. لاحظ أن قسمة قيمة الخاصية Width على قيمة الخاصية HorizontalResolution سيعطيك عرض الصورة بالبوصة.. وبالمثل: قسمة قيمة الخاصية Height على قيمة الخاصية VerticalResolution سيعطيك ارتفاع الصورة بالبوصة.

لاحظ أيضا أنّ تغيير عرض وارتفاع مربّع الصورة PictureBox، لا يؤثّر على عرض وارتفاع كائن الصورة Image المعروض فيه، فهاتان الخاصيتان تشيران دائما إلى مقاييس الصورة الأصليّة، وليس النسخة المعروضة في مربّع الصورة.

تنسيق النقط PixelFormat:

هناك العديد من تنسيقات النقاط، يجمعها المرقّم PixelFormat.. فهناك مثلا القيمة Format24bppRgb، وهي تدلّ على أنّ كل نقطة في الصورة تمثّل اللون بـ ٢٤ خانة ثنائيّة 24 bits per pixel.

الوسائل:

تدوير وعكس RotateFlip:

تمكّنك هذه الوسيلة من تدوير الصورة أو عكسها أو كلا الأمرين معا.

Image.RotateFlip(X);

حيث X هي إحدى قيم المرقّم RotateFlipType التالية:

RotateNoneFlipNone	لا تدوير ولا عكس.. واضح طبعا أنّك بهذا لن تفعل شيئا في الصورة (يفيد هذا في إلغاء عمليات التدوير والعكس السابقة، لإعادة الصورة إلى وضعها الأصليّ).
Rotate90FlipNone	تدوير الصورة بزاوية ٩٠ بدون عكسها.
Rotate90FlipX	تدوير الصورة بزاوية ٩٠ وعكسها أفقيّا.
Rotate90FlipXY	تدوير الصورة بزاوية ٩٠ وعكسها أفقيّا ورأسيّا.
Rotate90FlipY	تدوير الصورة بزاوية ٩٠ وعكسها رأسيّا.

وأعتقد أنّك تستطيع فهم باقي هذه التعبيرات:

Rotate180FlipNone ـ Rotate180FlipX ـ Rotate180FlipXY

Rotate180FlipY ـ Rotate270FlipNone ـ Rotate270FlipX

Rotate270FlipXY ـ Rotate270FlipY ـ RotateNoneFlipX

RotateNoneFlipXY ـ RotateNoneFlipY.

مثال:

```
PictureBox1.Image.RotateFlip(
        RotateFlipType.RotateNoneFlipY);
PictureBox1.Refresh( );
```

لاحظ استخدام الوسيلة Refresh لإعادة رسم الصورة، وإلا فلن يرى المستخدم أيّ تغيير.

ويمكنك التدريب على هذه الوسيلة في المشروع LoadImage.

الحصول على صورة مصغّرة GetThumbnailImage:

تمكّنك هذه الوسيلة من الحصول على نسخة مصغّرة من الصورة، حيث يمكنك استخدامها في عرض "كتالوج" الصور للمستخدم، لتسمح له باختيار صورة منها لعرضها كاملة.. ولهذه الوسيلة الصيغة التالية:

```
Image.GetThumbnailImage(العرض, الارتفاع, null,
        default(IntPtr));
```

حيث العرض والارتفاع يحدّدان بُعدَيِ الصورة المصغّرة الناتجة.. مثال:

```
PictureBox1.Image = img.GetThumbnailImage(32, 32,
        null, default(IntPtr));
```

ولديك في مجلّد برامج هذا الفصل، المشروع Thumbnails، وهو يسمح للمستخدم باختيار مجلّد، فيفحص البرنامج ملفّات هذا المجلّد ويعرض صورا مصغّرة على النموذج لكلّ الصور الموجودة في هذا المجلّد.. ويتمّ عرض كلّ صورة مصغّرة في مربّع صورة خاصّ بها، حيث يتمّ إنشاؤه في وقت التنفيذ وضبط إحداثياته.. وعندما يضغط المستخدم أيّ صورة مصغّرة، يتمّ عرض صورتها الأصليّة كاملة في نموذج آخر.

هذا هو الكود الذي يعرض الصور المصغّرة:

```
FileInfo FI = null;
PictureBox PBox = null;
Image img = null;
int Left = 280;
int Top = 40;
// احذف كلّ مربّعات الصور الموجودة على النموذج
for (int ctrl = this.Controls.Count - 1; ctrl >= 2; ctrl--)
   this.Controls.Remove(this.Controls[ctrl]);

// استخدم الوسيلة التالية لمحو أيّ رسومات غير دائمة من على النموذج
this.Invalidate( );
foreach (string file in Directory.GetFiles(
            Directory.GetCurrentDirectory( )))
{
   FI = new FileInfo(file);
   string ext = FI.Extension.ToLower();
   if (ext == ".gif" || ext == ".jpg" || ext == ".bmp" ||
       ext == ".ico")
   {
      PBox = new PictureBox(); // إنشاء مربّع صورة جديد
      // تحميل الصورة
      img = Image.FromFile(FI.FullName);
      // وضع الصورة المصغّرة في مربّع الصورة
      PBox.Image = img.GetThumbnailImage(64, 64,
            null, default(IntPtr));
      // إذا تجاوز موضع اليسار لمربّع الصورة الجديد عرض النموذج،
      // فضعه في بداية صفّ تالي
      if (Left > 580)
      {
         Left = 280;
         Top = Top + 74;
      }
```

```
        // ضبط إحداثيات مربّع الصورة
        PBox.Left = Left;
        PBox.Top = Top;
        PBox.Width = 64;
        PBox.Height = 64;
        PBox.Visible = true;
        // لا بدّ من الاحتفاظ باسم ملفّ الصورة،
        // حتّى يمكن عرضها عند الضغط على الصورة المصغّرة
        PBox.Tag = FI.FullName;
        // عرض مربّع الصورة على النموذج
        this.Controls.Add(PBox);
        // إنشاء مستجيب لحدث ضغط مربّع الصورة
        PBox.Click += new EventHandler(OpenImage);
        // (إضافة عرض مربّع الصورة + ١٠ (مسافة فاصلة
        // إلى المتغيّر الدال على موضع مربّع الصورة الجديد
        Left = Left + 74;
    }
}
```

وهذا هو الكود الذي يعرض الصورة عند الضغط على صورتها المصغّرة:

```
string FileName = ((PictureBox)sender).Tag.ToString( );
MessageBox.Show("will open " + FileName);
PreviewForm imgForm = new PreviewForm( );
imgForm.PictureBox1.Image =
Image.FromFile(FileName);
imgForm.Show( );
```

حفظ Save:

استخدم هذه الوسيلة لحفظ الصورة الموجودة في كائن الصـــورة، كملـــفّ على الجهاز.. مثال:

PictureBox1.Image.Save("c:\\tmpImage.bmp");

ويمكنك إرسال معامل ثان يمثل النوع الذي تريد حفـــظ الصـــورة بـــه.. ويستقبل هذا المعامل إحدى قيم المرقّم ImageFormat وهو معرّف فـــي النطاق System.Drawing.Imaging.. ولهذا المرقّم القيم التالية:

Bmp	Bitmap، وهو أبسط أنواع الصور، حيث يـــتمّ حفـــظ الصورة في هيئة مصفوفة ثنائيّة البعـــد (جـــدول مـــن صفوف وأعمدة).. وبهذا فإنّ كل خانة تحمل الإحداثيّات المناظرة لنقطة في الصورة (رقمها في الصف ينـــاظر الإحداثيّ الأفقيّ، ورقمها في العمود يمثّـــل الإحـــداثيّ الرأسّي)، بينما قيمة الخانة تمثّل لون النقطة.
Emf	Enhanced Windows metafile.
Exif	Exchangeable Image Format.
Gif	Graphics Interchange Format.
Icon	أيقونة.
Jpeg	Joint Photographic Experts Group.
MemoryBmp	لحفظ الصورة كمصفوفة ذاكرة.
Png	W3C Portable Network Graphics.
Tiff	Tagged Image File Format.
Wmf	Windows metafile.

الصور ولوحة القصاصات Clipboard:

يمكنك لنسخ النصوص والصور إلى لوحة القصاصات أو لصقها منها باستخدام الوسيلتين GetDataObject و SetDataObject التابعتين لكائن لوحة القصاصات Clipboard.

والجملة التالية تنسخ الصورة المعروضة في مربّع الصورة:

```
Clipboard.SetDataObject(PictureBox1.Image);
```

ولو أحببت أن تبقى الصورة في لوحة القصاصات بعد إغلاق برنامجك فاستخدم الصيغة:

```
Clipboard.SetDataObject(PictureBox1.Image, true);
```

ولاستعادة الصورة من لوحة القصاصات، يمكنك استخدام الوسيلة GetDataObject.. هذه الوسيلة تُرجع "كائن واجهة البيانات" IDataObject، وهو يمنحك الوسيلتين:

اقرإ البيانات GetData:

تعيد لك محتويات لوحة القصاصات.. ونظرا لأنّ لوحة القصاصات قد تحتوي على بيانات من أنواع مختلفة معا في نفس الوقت، فإنّك تمدّ هذه الوسيلة بمعاملٍ يمثّل نوع البيانات الذي تريده.

هل البيانات موجودة GetDataPresent:

ترجع True إذا كان نوع البيانات الذي أرسلته لها كمعامل هو نوع البيانات الموجودة في لوحة القصاصات.. ويمكنك أن ترسل لهذه الوسيلة معاملا ثانيا، إذا جعلت قيمته True، فإنّ لوحة القصاصات تحاول تحويل البيانات الموجودة بها إلى النوع الذي أرسلته في المعامل الأوّل.

ويمكنك استخدام الجملة التالية لعرض الصورة من لوحة القصاصات في مربّع الصورة:

```
PictureBox1.Image = Clipboard.GetDataObject(
                    ).GetData(DataFormats.Bitmap);
```

ولمزيد من التدريب، لديك مشروع ImageClipboard في مجلّد برامج هذا الفصل.

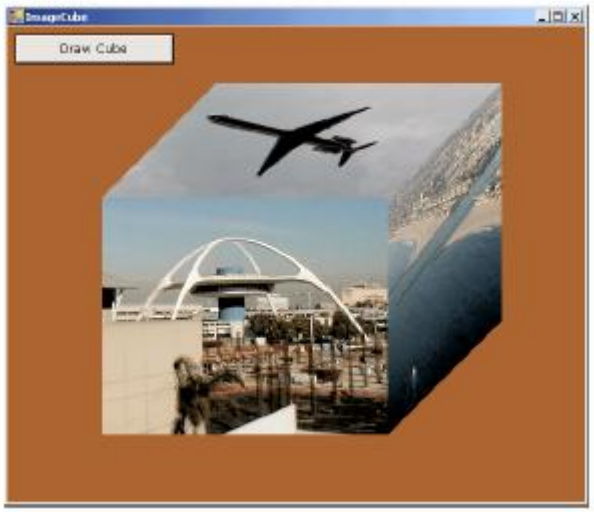
رسم صورة DrawImage:

يمنحك كائن الرسوم Graphics الوسيلة DrawImage لتمكّنك من رسم الصور على أيّ سطح تريده.. ولهذه الوسيلة ٣٠ صيغة مختلفة، قد تحتاج لكتاب بمفردها!!.. لهذا سنكتفي بشرح أربع صيغ:

```
Graphics.DrawImage(الصورة, نقطة رأسها العلويّ الأيسر);
Graphics.DrawImage(الصورة, المستطيل الذي يحتويها);
Graphics.DrawImage(الصورة, points( ));
```

وفي الصيغة الثالثة، المعامل الثاني هو مصفوفة نقاط، حيث تضع فيها ثلاثة نقاط، تمثّل ثلاثة من رءوس متوازي الأضلاع، الذي تريد أن تحوّر الصورة لترسم بداخله.. هذه الرءوس الثلاثة هي العلويّ الأيسر والعلوي الأيمن، والسفليّ الأيسر.. أمّا الرأس الرابع فيمكن استنتاجه، ولا حاجة بك لإرساله.. ولديك في مجلّد برامج هذا الفصل، المشروع المثير ImageCube، وهو يرسم مكعّبا ثلاثيّ الأبعاد، ويضع صورة على كلّ وجه من أوجهه الثلاثة المواجهة للمستخدم.. فعل هذا بسيط جدّا، فأنت تعرف أنّك ترسم المكعّب بثلاثة متوازيات أضلاع، أحدها في المواجهة،

والثاني يعطي الإحساس بالوجه الجانبيّ، والثالث يعطي الإحساس بقمّة المكعّب.. اختبر هذا المشروع الشيّق.

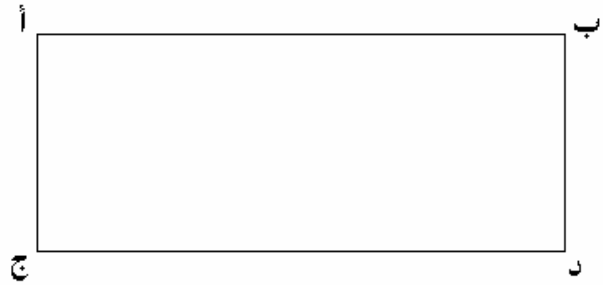


وهناك ملاحظة طريفة أخرى بخصوص هذه الصيغة.. فبإمكانك أن (تلخبط) ترتيب مصفوفة النقاط، وبذلك تقلب الصورة أو تعكسها، نتيجة تغيير مواضع رءوسها!.. طبعا لا يشترط دائما أن تعرض الصورة في متوازي أضلاع.. تذكّر فقط أنّ المستطيل هو متوازي أضلاع قائم الزاوية!

أو يمكنك أن تجعل رأسين متطابقين لترسم الصورة في مثلث!

أبدِع كما يحلو لك، فعلى حسب ترتيب رءوس المضلّع، سيتمّ قلب الصورة أو تدويرها أو عكسها أو مطّها أو تقليصها.. المهمّ أن تعرف ما تفعله.

افترض أنّ مستطيل الصورة هو أ ب ج د، كما هو موضح في الصورة:



الجدول التالي يوضّح لك بعض طرق التلاعب بهذه الصورة، على حسب ترتيب إضافة النقاط إلى المصفوفة:

ترتيب مصفوفة النقاط	التأثير
أ ب ج	الصورة كما هي.
ج د أ	عكس الصورة بالنسبة للمحور الأفقيّ.
ب أ د	عكس الصورة بالنسبة للمحور الرأسيّ.
د ب ج	عكس الصورة بالنسبة للقطر ب ج.
أ ج د	عكس الصورة بالنسبة للقطر أ د.
ج أ د	تدوير الصورة ٩٠ درجة في اتجاه عقارب الساعة.
ب د أ	تدوير الصورة ٩٠ درجة عكس اتجاه عقارب الساعة.
أ ج ج	رسم الصورة في مثلث!

كما أنّك تستطيع استخدام أيّ نقط أخرى تقع داخل الصورة أو خارجها، لرسم شرائح ممطوطة أو مضغوطة من الصورة!
وهكذا.. يمكنك أن تفعل بالصورة ما تشاء.

أمّا الصيغة الرابعة للوسيلة DrawImage فهي تسمح لك بتحديد سمات الصورة:

Graphics.DrawImage(الصورة, points( ), مستطيل,
السمات, وحدة القياس)

ويمثّل المستطيل الجزء الذي تريد رسمه من الصورة.. وتمثّل وحدة القياس الوحدة التي ستقاس بها أبعاد المستطيل.

أمّا المعامل الأخير فهو كائن من النوع ImageAttributes، حيث يمنحك بعض الوسائل لتغيير سمات الصورة، مثل جعلها تظهر على النموذج بالتدريج، ومثل تصحيح ألوانها.

ابدأ بتعريف كائن من هذا النوع، مع ملاحظة أنّه ينتمي لفضاء الاسم System.Drawing.Imaging:

```
var attr = new ImageAttributes( );
```

ومن وسائل هذا الكائن:

تغيير إضاءة الصورة SetGamma:

إذا تركت هذه الخاصيّة بالقيمة ١، فلن يتغيّر شيءٌ في لون الصورة، لكن لو جعلتها أكبر من ١ فستصير ألوان الصورة فاتحةً أكثر، وإذا جعلتها أصغر من ١ فستصير ألوان الصورة أغمق.

والكود التالي يريك كيف ترسم صورة في مربّع الصورة، مع تفتيح لونها بنسبة ٢٥%:

```
var attr = new ImageAttributes( );
attrs.SetGamma(1.25);
var dest = new Rectangle(0, 0, _
      PictureBox1.Width, PictureBox1.Height);
G.DrawImage(img, dest, 0, 0, img.Width, img.Height,
      GraphicsUnit.Pixel, attrs);
```

هذا بافتراض أنّ G هو كائن رسوم يرسم في مربّع الصورة، و img هو كائن صورة تمّ تحميل الصورة به.

قنوات فصل اللون SetOutputChannel:

تقنية فصل الألوان هي تقنية معروفة لزيادة جودة طباعة الصور الملوّنة، حيث تتمّ طباعة الصورة على نفس الورقة أربع مرّات.. في كلّ مرّة يتمّ طباعة لون معيّن تمّ فصله من الصورة بمفرده.. وفي الغالب تُفصل الصورة إلى الألوان: السماوي cyan والبنفسجيّ المحمرّ magenta والأصفر yellow والأسود black، والتي تختصر بالتعبير CMYK.

وللحصول على هذه النسخ من الصورة، استدع هذه الوسيلة أربع مرّات بالمعاملات التالية:

```
attrs.SetOutputChannel(
    ColorChannelFlag.ColorChannelC); // cyan
attrs.SetOutputChannel (
    ColorChannelFlag.ColorChannelM); // magenta
attrs.SetOutputChannel(
    ColorChannelFlag.ColorChannelY); // yellow
attrs.SetOutputChannel(
    ColorChannelFlag.ColorChannelK); // black
```

وهناك صيغة أخرى كالتالي:

```
attrs.SetOutputChannel(
    ColorChannelFlag.ColorChannelLast);
```

وذلك لاستدعاء الوسيلة مرّة أخرى بآخر لون تمّ استدعاؤها به.

وبخصوص باقي صيغ الوسيلة DrawImage، فأرجو أن تتعرّف عليها، إمّا عن طريق ملفات المساعدة، أو عن طريق تلميح الشاشة الذي يظهر عند كتابة اسم الوسيلة في محرّر الكود.. مع ملاحظة أنّ معظم هذه الوسائل تتلقّي معاملا، عبارة عن مستطيل يمثّل المساحة التي سترسم فيها الصورة.. فإذا كانت أبعاد الصورة مختلفة عن أبعاد المستطيل، يتمّ مطّها أو تقليصها لتتناسب معه.

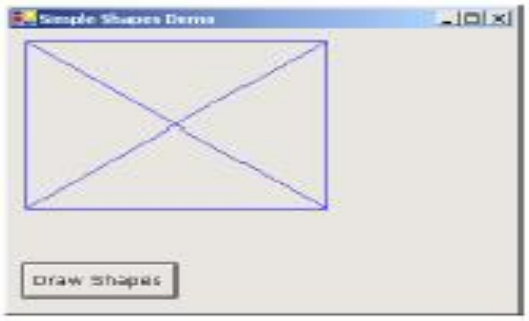
كما أنّ بعض الصيغ تتلقّي مستطيلين: الأوّل يحدّد الجزء الذي تريد رسمه من الصورة، والآخر يوضّح المساحة التي سترسم هذا الجزء فيها.

الصور النقطيّة Bitmaps

الصورة هي مصفوفة من الألوان مرتّبة في صفوف وأعمدة.. ونتيجة للألوان وترتيبها، تبدو للعين كأنّها صورة متصلة مفعمة بالتفاصيل.
وهنا يجب توضيح الفارق بين كائن الصورة Image، وكائن الصورة النقطيّة Bitmap.. فالكائن Image يمثل صورة ثابتة، لا تستطيع تغيير نقاطها كلاّ على حدة.. أمّا كائن الصورة النقطيّة Bitmap، فهو يسمح لك بالتعامل مع الصورة نقطة نقطة.. وهذه هي الطريقة التي يمكنك بها تعديل جزء من الصورة.
تعال نرَ كيف يمكننا استخدام كائن الصورة النقطيّة لتثبيت الرسوم.

إبقاء الرسوم على النموذج والأدوات:

كلّ ما ترسمه على النموذج والأدوات سيختفي لو صغّرت النموذج ثمّ كبّرته، أو جعلت نموذجا آخر يخفي جزءا من نموذجك ثم أبعدته.. الصور فقط هي التي لا تُمحى إلا عندما تزيلها أنت.. بطريقة أخرى: الرسوم ــ على خلاف الصور ــ مؤقّتة، ولا تُضاف لكائن الرسوم.
ولحلّ هذه المشكلة، يمكنك أن تكتب كود الرسم في الحدث Paint، ليتمّ تنفيذه كلّما احتاج النموذج لإعادة رسمه.
كما تحتوي فئة النموذج الأم Form Class على إجراء محمي Protected يدعى OnPaint هو الذي يقوم بإطلاق الحدث Paint، ويمكنك أن تستبدل هذا الإجراء Override في فئة النموذج الخاصة بك، بإجراء آخر بنفس الاسم، لكي يُنفّذ كلّما احتاج النموذج لإعادة رسمه.. لقد كتبنا فيه الكود الذي يرسم الشكل التالي على النموذج:




```
protected override void OnPaint(PaintEventArgs e)
{
    Graphics G = e.Graphics;
    Pen P = new Pen(Color.Blue);
    G.DrawRectangle(P, 10, 10, 200, 150);
    G.DrawLine(P, 10, 10, 210, 160);
    G.DrawLine(P, 210, 10, 10, 160);
}
```

ولكنّ المشكلة لم تحلّ بعد!!.. فلو كانت الرسوم معقّدة وتستهلك وقتا، فسيبدو أداء برنامجك عقيما كلّما انتقلت من النموذج ثمّ عدت إليه.

الحلّ هو أن نضيف الرسومات إلى كائن صورة نقطية Bitmap، بحيث تصبح كالصور لا نحتاج لإعادة رسمها.

ابدأ بتعريف كائن صورة نقطيّة له نفس مساحة مربّع الصورة.. لاحظ أن الكائن Bitmap مماثل للكائن Image، لدرجة أنّك تستطيع تحميل الصورة من أحدهما للآخر:

```
var bmp = new Bitmap(PictureBox1.Width,
                     PictureBox1.Height);
```

هذا الكائن يمثّل صورة فارغة الآن.

الخطوة الثانية هي أن نربط كائن الصورة الخاصّ بمربّع الصورة، بكائن الصورة النقطيّة:

```
PictureBox1.Image = bmp;
```

وأنت تعرف طبعا أنّ هذا التساوي مرجعيّ ref.. الآن صار أيّ تغيير يحدث للكائن bmp، يحدث كذلك لكائن الصورة.

الخطوة الثالثة هي أن ننشئ كائن رسوم يرسم على الصورة النقطيّة، وذلك كالتالي:

```
Graphics G = Graphics.FromImage(bmp);
```

منذ الآن، كلّ ما سنرسمه بكائن الرسوم سيتمّ رسمه على كائن الصورة النقطيّة bmp، وبالتالي سيتأثّر به كائن الصورة Image الخاصّ بمربّع الصورة، وبالتالي يصبح رسما دائما في مربّع الصورة.

هذا هو الكود وقد جمعناه معا في دالّة واحدة، ليسهل استخدامها بعد ذلك:

```
public Graphics GetGraphicsObject(PictureBox PBox)
{
    var bmp = new Bitmap(PBox.Width, PBox.Height);
    PBox.Image = bmp;
    Graphics G = Graphics.FromImage(bmp);
    return G;
}
```

والآن كلّ ما عليك هو إرسال اسم مربّع الصورة إلى هذه الدالة، لتعيد إليك كائن رسوم يرسم رسوما ثابتة على سطحه.

انظر كيف نستخدم هذه الدالة في المشروع SimpleGraphics في مجلّد برامج هذا الفصل.

ولكي يمكنك أن تفعل المثل مع النموذج، فعليك استخدام الخاصيّة BackgroundImage لأنه لا يمتلك الخاصية Image.

ملحوظة:

يمتلك النموذج والأدوات الوسيلة Invalidate، لمسح الأشـــكال غيـــر الثابتة من عليها.. ولو ناديت هذه الوسيلة بدون معاملات فستؤثّر على كلّ مساحة النموذج أو الأداة.. ولو ناديتها وأرســـلت لهـــا مســـتطيلا كمعامل، فستؤثّر على مساحة النموذج أو الأداة، التي تقع داخـــل هـــذا المستطيل.. طبعا هذه الوسيلة لن تؤثّر على الرســـوم التـــي أضـــفناها لصرة نقطيّة.

التعامل مع نقط الصورة في الذاكرة مباشرةً:

تتسم الوسيلتان GetPixel و SetPixel بنوع ملحوظ من البطء، حتّى إنّك ستلاحظه في عمليّة تافهة كعكس الألوان.. المشكلة أنّ التعامل مع الصور لا يكون بنفس بساطة برنامج عكس الألوان، فلو فحصت العمليّات الأخرى الموجودة في مشروع ImageProcessing، فستجد أن تغيير قيمة أيّ نقطة يستلزم إجراء بعض العمليّات الحسابيّة على النقاط الثمانية المحيطة بها من الجهات الأربع (وأحيانا تمتدّ العمليات إلى غير ذلك من النقاط).. في هذه الحالة سيكون من العبث استدعاء الوسيلة GetPixel تسع مرّات لتغيير كل نقطة!

وأبسط تفكير يردُ على الخاطر في هذه الحالة، هو قراءة كل نقاط الصورة مرّة واحدة وحفظها في مصفوفة، والتعامل بعد ذلك مع المصفوفة عند الحاجة لقراءة أيّ نقطة.

ولكنّ هذا لن يحلّ مشكلة البطء نهائيّا، كما أنّه يستهلك الذاكرة!

لهذا فإنّ لدينا حلاّ آخر.. ما رأيك لو تعاملنا مع النقاط في أماكن تخزينها في الذاكرة مباشرة؟

إنّ ذلك سيكون صعبا نوعا ما، وفيه بعض المخاطرة، فمن الممكن أن تؤدّى إلى إغلاق البرنامج أو الويندوز كلّه، لو كتبت في منطقة خاطئة من الذاكرة، فأفسدت بعض بيانات البرامج الأخرى أو النظام.

توكّل على الله وهيّا اتبع معي هذه الخطوات:

في البداية تعال نختار الجزء الذي نريد التعامل معه من الصورة، ونضعه في كائن بيانات الصورة النقطيّة BitmapData.. ولاستخدام هذا الكائن، لا تنسَ أن تكتب جملة الاستيراد التالية في مشروعك:

```
using System.Drawing.Imaging;
```

عرّف كائن صورة نقطيّة وضع فيه صورة مربّع النصّ:

```
var Bitmap = new Bitmap(PictureBox1.Image);
```

بعد ذلك عرّف مستطيلا له نفس مساحة الصورة:

```
var Rect = new Rectangle(0, 0, Bitmap.Width,
          Bitmap.Height);
```

ثمّ عرّف كائن بيانات الصورة النقطيّة كالتالي:

```
var BitmapData = new BitmapData( );
```

ثمّ ضع الصورة في كائن بيانات الصورة النقطيّة باستخدام الوسيلة "إغلاق خانات الصورة" LockBits:

```
BitmapData = Bitmap.LockBits(Rect,
   ImageLockMode.WriteOnly, Bitmap.PixelFormat);
```

هذه الوسيلة تمنع تغيير موضع الصورة في الذاكرة (بفعل جامع القمامة GC أو الويندوز) إلى حين الانتهاء من التعامل معها.. وهي تأخذ ثلاثة معاملات: أوّلها مستطيل يحدّد المساحة التي ستمنع التعامل معها من الصورة، والثاني نوع الإغلاق، وقد اخترنا منع الكتابة على الصورة، والثالث هو نوع تنسيق الصورة.

والآن صار كائن بيانات الصورة يشير إلى بيانات الصورة في الذاكرة، ولكي نحصل على عنوان أوّل نقطة في الصورة في الذاكرة، سنستخدم الوسيلة Scan0.. حيث سنقوم بحفظ العنوان في متغيّر مخصّص لهذا النوع من البيانات، هو المؤشّر Pointer من النوع "مؤشّر صحيح" IntPtr:

```
IntPtr pixels = BitmapData.Scan0( );
```

ولاستخدام هذا المؤشر، لدينا فئة تسمّى Marshal، تمكّننا من القراءة من الذاكرة والكتابة فيها، عن طريق المؤشّرات، وهي موجودة في النطاق:
System.Runtime.InteropServices

وتمنحك الفئة Marshal عشرات الوسائل للتعامل مع الذاكرة، ولكنّ ما يهمّنا هنا هو القراءة والكتابة في الذاكرة.. وفي هذا الصدد لدينا العديد من الوسائل المتماثلة، التي لا تختلف إلا في عدد الوحدات التي تقرؤها أو تكتبها في الذاكرة.. فللقراءة يمكنك استخدام:

اقرأ وحدة ReadByte ــ اقرأ عدد صحيح قصير ReadInt16 ــ اقرأ عدد صحيح ReadInt32 ــ اقرأ عدد صحيح طويل ReadInt64.

وكلّ هذه الوسائل تأخذ معاملين: المؤشّر الذي يشير لموضع الذاكرة، وعدد الوحدات التي تريد إضافتها على موضع هذا المؤشّر Offset، حتّى تتمكّن من قراءة المواضع التالية له.. وطبعا تعيد كلّ وسيلة نفس نوع البيانات الذي تقرأه.

وللكتابة في الذاكرة استخدم الوسائل:

اكتب وحدة WriteByte ــ اكتب عدد صحيح قصير WriteInt16 ــ اكتب عدد صحيح WriteInt32 ــ اكتب عدد صحيح طويل WriteInt64.

وتأخذ هذه الوسائل نفس معاملي وسائل القراءة، بالإضافة لمعامل ثالث، هو القيمة التي تريد أن تكتبها في الذاكرة، مع مراعاة أن تكون من نفس النوع الذي تكتبه الوسيلة.

بقي شيء هامّ، هو معرفة كيف تحفظ النقاط في الذاكرة:

إنّ كل نقطة تأخذ عددا من وحدات الذاكرة Bytes يختلف على حسب نوع تنسيق الصورة.. لقد افترضنا هنا أنّ الصورة ملوّنة، وأنّ كلّ نقطة من نقاطها تخزّن في أربع وحدات ذاكرة.

ويجب أن تعرف، أنّ التعامل مع الذاكرة سيختلف عن التعامل مع الصورة، فقد كنا نحصل على النقطة بمعرفة الصف والعمود.. ليس في

الذاكرة هذا التقسيم، وإنّما يتمّ حفظ نقاط الصورة على التوالي: توضع نقاط الصفّ الأوّل، ثمّ يليها نقاط الصف الثاني، فالثالث وهكذا...
افترض أنّ هذه هي خانات الصورة (سنكتب في كلّ خانة رقمها للإيضاح):

0	1	2	3	4
5	6	7	8	9

هكذا تخزّن في الذاكرة:

0	1	2	3	4	5	6	7	8	9

فإذا علمت أنّ كل نقطة تخزّن في ٤ وحدات ذاكرة، فإنّ هذا يعني أنّ أوّل نقطة في الصف الثاني من الصورة (لاحظ أنّ رقمه ١، لأنّ الصفّ الأوّل رقمه صفر) ستوضع في خانة الذاكرة التي تساوي:

عرض الصورة (٥ خانات في مثالنا هذا) × ٤

والتي تليها (لاحظ أنّ رقم عمودها ١، لأنّ العمود الأوّل رقمه صفر) ستكون في الخانة رقم: عرض الصورة × ٤ + ١ × ٤، وهكذا ..
وبصورة عامّة:

الخانة رقم ع في الصفّ الثاني ستوضع في خانة الذاكرة رقم:

٤ × (عرض الصورة + ع).

كما أنّ أوّل خانة في الصفّ الثالث ستوضع في الخانة رقم:

٤ × (٢ × عرض الصورة)

والتي تليها ستكون في الخانة: ٤ × (٢ × عرض الصورة + ١).
وبصورة عامّة: لمعرفة موضع نقطة في الذاكرة، تقع في الصورة في الصفّ ص والعمود ع، طبّق المعادلة:

رقم الخانة في الذاكرة = (ص × عرض الصورة + ع) × عدد الوحدات التي تمثّل النقطة (٤).

والآن تعال نكتب دالّة عكس الألوان بهذه الطريقة:

```
int W = Rect.Width;
Color Clr = default(Color);
for (int i = 0; i < Rect.Height; i++)
{
    for (int j = 0; j < W; j++)
    {
        // حساب موضع النقطة في الذاكرة
        int O = i * W * 4 + j * 4;
        // قراءة عدد صحيح من الذاكرة يمثّل اللون
        int C = Marshal.ReadInt32(pixels, O);
        // تحويل اللون من عدد صحيح إلى كائن لون
        Clr = Color.FromArgb(C);
        // عكس مكوّنات اللون وتكوين اللون المعكوس
        Clr = Color.FromArgb(Clr.A, 255 - Clr.R,
              255 - Clr.G, 255 - Clr.B);
        // تحويل كائن اللون إلى عدد صحيح
        C = Clr.ToArgb();
        // تخزين اللون المعكوس في الذاكرة
        Marshal.WriteInt32(pixels, O, C);
    }
}
```

ولا تنس في النهاية تحرير المساحة التي أغلقناها في الذاكرة، مع وضع كائن الصورة النقطيّة في مربّع الصورة وإنعاشه، حتّى ترى التغيير:

```
Bitmap.UnlockBits(BitmapData);
PictureBox1.Image = Bitmap;
```

PictureBox1.Refresh( );

ستجد هذا الكود كاملا في الأمر Inverse1 تحت القائمة Process في المشروع ImageProcessing.. حاول أن تقارن بين هذه الطريقة وبين الطريقة التقليديّة التي كتبنا بها الأمر Inverse في نفس القائمة.. ستجد الفارق في السرعة واضحا (وصل معي إلى خمسة أضعاف)، وسيتّضح أكثر، لو كتبت العمليّات الأربع الأخرى الموجودة في نفس القائمة بالطريقة الجديدة، فهي بالفعل تأخذ وقتا شنيعا!

ولو شئت تعديل كود عكس الألوان ليتعامل مع صور بتنسيقات مختلفة، تُخزَّن فيها النقطة في وحدة Byte أو اثنتين أو ثلاثة أو أربعة، فاستخدم الوسيلة "قراءة حجم تنسيق النقطة" GetPixelFormatSize، حيث يمكنك أن تعرف بها عدد الوحدات التي تخزّن فيها كلّ نقطة كالتالي:

N = Bitmap.GetPixelFormatSize(Bitmap.PixelFormat);

وبناء على هذا الرقم، عدّل الكود ليقرأ N من وحدات الذاكرة في كلّ مرّة بدلا من ٤.. لن يكون ذلك بسيطا، ولكنّه أيضا لن يكون معقّدا.. اعتبره تدريبا لك.. هيّا يا بطل!

تغيير لون معيّن في الصورة إلى لون آخر:

لحسن الحظّ، لن تحتاج إلى المرور على كل نقاط الصورة لونا لونا، بحثا عن اللون الذي تريد تغييره.. إنّ لدينا طريقةً جاهزةً لفعل ذلك.. اتّبع الخطوات التالية:

ابدأ بتعريف كائن الصورة النقطيّة، وضع به الصورة التي تريد تغييرها (ولتكن تلك الموجودة في مربّع الصورة):

```
var image = new Bitmap(PictureBox1.Image);
```

الآن سنعرّف كائنا يسمّى خريطة اللون ColorMap.. لهذا الكائن خاصيتان هامّتان:

اللون القديم OldColor: وهو الذي تريد إزالته من الصورة.

اللون الجديد NewColor: وهو الذي تريد استبداله باللون القديم.

وبإمكانك أن تعرّف مصفوفة من هذا الكائن، بحيث تستبدل مجموعة من الألوان بغيرها مرّة واحدة.. وفي مثالنا هذا سنعرّف مصفوفة من خانة واحدة، نستبدل بها اللون الأزرق (اللون الجديد) باللون الأحمر (اللون القديم):

```
var cm = new ColorMap( ) {OldColor = Color.Red,
                          NewColor = Color.Blue};
ColorMap[] colorMap = { cm };
```

ثمّ عرّف كائنا من نوع سمات الصورة ImageAttributes:

```
var imageAttributes = new ImageAttributes( );
```

الآن سنستخدم الخاصيّة "تغيير خريطة الألوان" SetRemapTable الخاصّة بكائن سمات الصورة لجعل مصفوفة خريطة التغيير جزءا من سمات الصورة:

```
imageAttributes.SetRemapTable(colorMap);
```

والآن لم يبقَ إلا أن تعيد رسم الصورة، حتّى تبدو التعديلات بها.. وسنستخدم في هذا الوسيلة DrawImage الخاصّة بكائن الرسوم.. ومن الطبيعيّ أن نرسم بها على الصورة النقطيّة، حتّى تظلّ الصورة المرسومة ثابتة كما شرحنا من قبل:

```
int W = PictureBox1.Width;
int H = PictureBox1.Height;
// إنشاء كائن رسوم من الصورة النقطيّة
Graphics G = G.FromImage(image);
// الرسم على الصورة النقطية بالسمات التي ضبطناها،
// وبذلك نغيّر اللون الأحمر إلى الأزرق
G.DrawImage(image, new Rectangle(0, 0, W, H),
    0, 0, W, H, GraphicsUnit.Pixel, imageAttributes)
// وضع الصورة في مربّع الصورة
PictureBox1.Image = image;
// إنعاش مربّع الصورة
PictureBox1.Refresh( );
```

هل تبدو صيغة وسيلة رسم الصورة DrawImage جديدة عليك؟.. هذه هي المعاملات بالترتيب:

- الصورة التي سنرسمها.
- المستطيل الذي يحدّد المساحة التي سنرسم فيها.
- المعاملات الأربعة التالية تحدّد المساحة التي سنرسمها من الصورة، وهي بالترتيب: الإحداثيين الأفقي والرأسي لزاوية المستطيل العليا اليسرى، وعرض المستطيل، وارتفاعه.
- وحدة قياس الأطوال، وفي حالتنا هذه هي النقطة Pixel.

- وأخيرا المعامل الذي يهمّنا والذي فعلنا كلّ ذلك من أجله: سمات الصورة، حيث أرسلنا كائن سمات الصورة الذي يحمل خريطة تغيير الألوان.

ويمكنك أن تجرّب هذا المثال في المشروع ImageProcessing، بضغط الأمر ChangeColor في القائمة Processing، بشرط أن تضع في مربّع الصورة صورة بها مساحة حمراء، حتّى تشعر بحدوث اختلاف.

لا بدّ أنّه قد ساءك أنّ المستخدم مجبر على تغيير اللون الأحمر إلى الأزرق.. فماذا لو أراد أن يغيّر أيّ لون في الصورة إلى أيّ لون آخر؟

كتدريب خاصّ، حاول تعديل الكود الخاص باستبدال الألوان، ليسمح للمستخدم باختيار اللون الذي يريد تغييره، واختيار اللون البديل، وذلك باستخدام الأداة ColorDialog.

وعلى فكرة: يمكنك تغيير أيّ لون ليصير شفافا Color.Transrarent، أو ليصير أي لون به نسبة شفافية (باستخدام معامل الشفافية في دالة تكوين اللون Color.FromArgb).

تغيير نقاط الصورة:

لكي تتمكّن من التعامل مع كلّ نقطة في الصورة على حدة، يمنحك كـــائن الصورة النقطيّة الوسيلتين التاليتين:

اقرأ النقطة GetPixel:

تأخذ هذه الوسيلة إحداثيي النقطة المطلوبة، وتعيد لك لونها:

```
Color Clr = Bitmap.GetPixel(X, Y);
```

وتستهلك هذه الوسيلة وقتا ملموسا، خاصّة مع تكرار استدعائها لكلّ نقاط الصورة.

تغيير النقطة SetPixel:

تأخذ هذه الوسيلة إحداثيي النقطة المطلوبة واللون الذي تريد تلوينها به:

```
Bitmap.SetPixel(X, Y, Clr)
```

وفي المثال التالي نرى كيف يمكننا عكس ألوان صورة ما للحصول على النسخة السلبية (النيجاتيف) منها.

لقد كتبنا في هذا الفصل دالة عكس الألوان RevColor.. والآن كلّ مـــا سنفعله، هو أن نطبّقها على كلّ نقطة في الصورة لعكس لونها.

ولقراءة نقاط الصورة وتغييرها، لا بد مـــن كتابـــة جملتـــين تكـــراريّتين متداخلتين، إحداهما لقراءة الصفوف والأخرى لقراءة الأعمدة.

هذا هو الكود الذي يعكس الألوان:

```
// تعريف متغير صورة نقطيّة، وتحميل صورة مربّع الصورة فيه
var Btmap = new Bitmap(PictureBox1.Image);
// ربط مربّع الصورة بكائن الصورة النقطيّة، حتّى يتأثّر بالتغيرات التي تحدث له
PictureBox1.Image = Btmap;
```

```
// جملة تكراريّة للمرور عبر صفوف الصورة
for (int R = 3 , R < Btmap.Width – 2)
{
    // جملة تكراريّة للمرور عبر نقاط كلّ صفّ
    for (int C = 0, C < Btmap.Height – 2)
        Btmap.SetPixel(R, C,
                RevColor( Btmap.GetPixel(R, C) ) );
}
PictureBox1.Refresh( );
```

ويقدّم لك المشروع ImageProcessing أربع عمليّات أخرى على الصور.. حاول تجربته.

رسم الدوالّ Plotting Functions:

افترض أنّ لدينا المعادلة: ص = ٢ س.

لرسم هذه الدالة، لا بدّ أن نوجد قيم ص المناظرة لكلّ قيم س (٠، ١، ٢، ٣، ... إلخ).

ثم نقوم برسم النقاط الناتجة، حيث إنّ كلّ نقطة هي عبارة عن الإحداثيين (س، ص).. فمثلا في دالتنا هذه ستكون النقطة على الصيغة (س، ٢س).. مثل: (٠، ٠) ، (١، ٢) ، (٢، ٤) ... إلخ.

ولا بدّ أن نصل هذه النقاط بخطوط مستقيمة، حتّى يبدو المنحنى متصلا.

وهنا ستبرز هذه المشاكل:

١- إذا كان معدّل تغيّر الدالة كبيرا، فستجد أنّ النقاط متباعدة، بحيث ستكون الخطوط الواصلة بينها واضحة للعيان، وسيبدو المنحنى متكسرا غير ناعم.. لحلّ هذه المشكلة نحتاج لتغيير مقياس رسم المحور السيني (الأفقيّ).. وللقيام بذلك نطبّق المعادلة التالية:

مقياس الرسم الأفقي = عرض مربّع الصورة ÷ (أكبر س – أصغر س).

فمثلا: لو كنا سنرسم الدالة لقيم س بين -١ و ٥، فسيكون:

مقياس الرسم = عرض مربّع الصورة ÷ (٥ – [-١])

= عرض مربّع الصورة / ٦.

٢- إذا كانت قيم ص الناتجة من الدالة كبيرة، فإنّ كثيرا منها لن يظهر في منطقة الرسم.. ولحلّ هذه المشكلة، نحتاج لتغيير مقياس رسم المحور الصادي (الرأسيّ).

مقياس الرسم الرأسيّ = ارتفاع مربّع الصورة ÷ (أكبر ص – أصغر ص).

٣- إذا كانت هناك قيم سالبة على المحور السينيّ أو الصاديّ، فإنّها لن تظهر في مساحة الرسم (لأنّ إحداثيات النموذج والأدوات تبدأ

من (٠، ٠).. ولحلّ هذه المشكلة، يجب نقل الإحداثيّات، حتّى تصبح أكبر قيمة سالبة منطبقةً على الصفر.

وفي حالتنا هذه، سنقوم برسم النقاط في كائن مسار الرسوم GraphicsPath، حيث سنصل بخطّ مستقيم بين كلّ نقطة جديدة نضيفها إليه والنقطة السابقة لها، ثمّ سنرسم كائن المسار مرّة واحدة.. هذا أسرع من رسم كلّ نقطة على حدة، كما أنّها تمكّنك من استخدام المتغيّر الذي يشير لكائن مسار الرسوم لإعادة رسم الدالة أكثر من مرّة بمقاييس رسم مختلفة.

ولكن كيف نقوم بهذه التحويلات؟

قلنا إنّ عمليات التحويل يتّم حفظها في مصفوفة التحويل Matrix المعرفة في النطاق System.Drawing.Drawing2D.. لهذا سنستخدم هذه المصفوفة في حالتنا هذه، وسنجري عليها التحويلات مباشرة، وأوّل ما سنفعله هو تعريف متغيّر من مصفوفة التحويل:

```
var World = new Matrix( );
```

ملحوظة:

هذه المصفوفة تتكوّن من ٣ صفوف و ٣ أعمدة، وهي تمتلك وسائل جاهزة لتدوير المصفوفة Rotate وعكسها Invert وضربها في مصفوفة أخرى Multiply.. ولقراءة محتويات مصفوفة التحويلات والحصول على مصفوفة عاديّة (3x3 Array) استخدم الخاصيّة Elements.

بعد ذلك سنغيّر مقياس الرسم أفقيّا ورأسيّا:

```
World.Scale( ((PictureBox1.Width - 4) / (Xmax - Xmin)),
-(PictureBox1.Height – 4) / (Ymax - Ymin));
```

لاحظ أنّنا طرحنا ٤ من عرض وارتفاع مربّع الصورة، حتّى نزيل عرض الإطار (٢ في كل حافة) ونترك هامشا بينه وبينها.. لاحظ أيضا أن أصغر وأكبر قيمتين على المحور الأفقي Xmin و Xmax اختياريتان، على حسب المدى الذي تريد رسم الدالّة فيه.. أمّا أصغر وأكبر قيمة على المحور الرأسي فلا بدّ من حسابهما أولا، سواء بالوسائل الرياضيّة التي تحسب القمّة العظمى والصغرى (سيحتاج هذا لإجراء عمليات تفاضل، وهذا خارج نطاقنا الآن)، أو بجملة تكراريّة تحسب كل قيم ص المناظرة لقيم س، بحيث تلتقط من بينها أصغر قيمة وأكبر قيمة.

بعد ذلك سنحرّك الإحداثيّات لتظهر كلّ النقط في منطقة الرسم:

```
World.Translate(-Xmin, -Ymax);
```

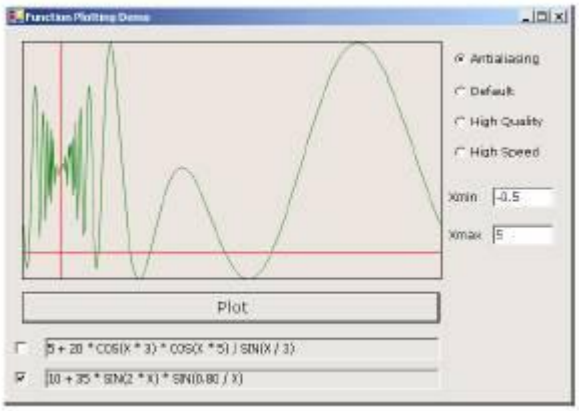
لاحظ أنّ الإحداثيّات في الكتب العلميّة والرياضيّة تفترض أنّ القيم على المحور الرأسي تزداد لأعلى وتقلّ لأسفل، لهذا فقد نقلنا الإحداثيّ الرأسيّ لأسفل بمقار أكبر قيمة له، بحيث تظهر هذه القيمة في أعلى الشاشة (ارتفاعها = ٠).

الآن أصبحت مصفوفة التحويلات تحتوي على كل التحويلات المطلوبة، ولكي تنطبق هذه التحويلات على كائن مسار الرسوم، فإنّنا نستدعي الوسيلة Transform الخاصّة به، ونرسل هذه المصفوفة كمعامل لها، ثمّ بعد ذلك نرسمه:

```
GraphicPath1.Transform(World);
G.DrawPath(plotPen, plot1);
```

لاحظ أنّ الفارق بين استخدام مصفوفة التحـويلات Matrix، ووسـائل التحويل الخاصّة بكائن الرسوم Graphics، هي أنّ مصفوفة التحـويلات خاصّة فقط بالكائن الذي يستخدمها (كائن مسار الرسوم هنا)، أمّا وسـائل التحويل الخاصّة بكائن الرسوم فهي عامّة لكل منطقة الرسم.

ولديك في مجلد برامج هذا الفصل، المشروع Plotting، وفيـه سـترى كيف يمكن رسم دالتين مختلفتين بواسطة المفاهيم التي شرحناها هنا.



أمّا لو أردت أن تسمح للمستخدم بتعريف الدالـة التـي يريـد رسـمها (بالصـيغة الإنجليزيّـة)، فيمكنـك اسـتخدام الأداة Script ActiveX Control.. ولإضافة هذه الأداة لبرنامجك، اضـغط بزر الفأرة الأيمن على كلمة المراجع References في متصفّح المشاريع Solution Explorer، ومن القائمة الموضعيّة اختر Add Reference، وفي مربّع الحوار الذي سيظهر اضغط الشـريط COM.. ومـن بـين العناصـر التـي تمـلأ القائمـة انقـر مـرتين علـى العنصـر Microsoft Script Control 1.0.. بعد ذلك أضغط OK لإغلاق مربّع الحوار.

الآن يمكنك كتابة دالة تحسب لك قيمة الصيغة التي عرّفها المستخدم عنـــد قيمة معيّنة لـــ X، باستخدام لغة VBScript، كالتالي:

```
double FVal(double X, string Formula)
{
    var S = new MSScriptControl.ScriptControl( );
    S.Language = "VBScript";
    try
    {
        S.ExecuteStatement("X=" & X)
        return  S.Eval(Formula);
    }
    catch
    {
        Throw New Exception("لا يمكن حساب الدالة عند X=" & X)
    }
}
```

والآن جرّب استدعاء هذه الدالة كالتالي:

```
string F = "5 + 20 * Cos(X * 3) * Cos(X * 5) " +
           "/ Log(Abs(Sin(X)) / 10)";
MessageBox.Show(FVal(3, F));
```

عنـــد تنفيـــذ هـــذا الكـــود، ســـتظهر لـــك رســـالة تجمـــل الـــرقم: ١,٧٥٠٩١٠٣٢٧٣٠١٢٦.

جرّب وضع مربّع نصّ على النموذج، ليستقبل صيغة الدالة (بالإنجليزيّة) ومربّع نصّ آخر ليستقبل قيمة X التي تريد أن تحسب قيمة الدالة عندها، وزرا ينفذ الكود التالي:

```
double V = FVal( (Convert.ToDouble)(TextBox2.Text),
                 TextBox1.Text);
MessageBox.Show (V);
```

وفي ضوء هذا، حاول أن تطوّر مشروع رسم الدوال Plotting، ليسمح للمستخدم بتعريف صيغة الدالة التي يريدها.

عامّة ستجد هذا متحقّقا بالفعل في المشروع FunctionPlotting.
ولو أرد تدريبا جيّدا، فعليك أن تكتب دالة تحوّل الصيغ العربيّة إلى الصيغ الإنجليزيّة، مثل:

÷	/
×	*
جا	Sin
جتا	Cos
|.....|	Abs(.....)
لوج	Log

ويفضّل أن تستخدم في هذا محرّك "التعبيرات النمطيّة" Regular Expression الذي يمنحه لك إطار العمل، وهو مشروح بالتفصيل في مرجع "**من الصفر إلى الاحتراف: برمجة إطار العمل**".

الفصل الثاني

الأدوات الخاصّة
Custom Controls

القوّة.. والسهولة.. ومنتهى الروعة:

لا ريب أنّك ستستمع بهذا الفصل، ففيه ستتعلّم كيف تنشئ الأدوات Controls التي تروق لك، بحيث توفّر على نفسك وقت إعادة تصميم النماذج التي تستعملها كثيرا.

ويكفي أن أخبرك أنّك تستطيع أن ترث أيّ أداة من أدوات سي شارب، لتعدّل فيها أو تضيف بعض الخصائص والوسائل إليها.. هذا يضمن لك أنّك لن تبدأ من الصفر في كلّ مرّة!

تطوير الأدوات الموجودة

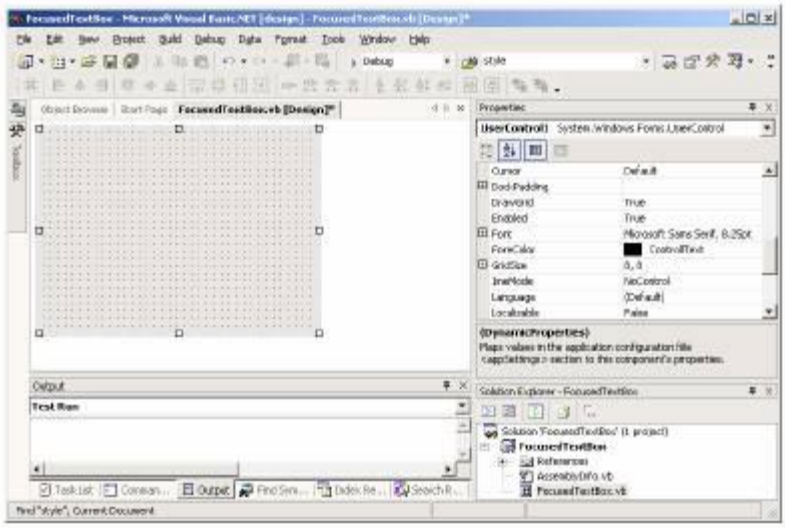
أبسط طريقة لبناء الأدوات، هي وراثة أداة موجودة وتطويرها.
افترض أنّك تريد أن تصمم نموذجا يستقبل البيانات من المستخدم عن طريق مجموعة من مربّعات النصّ.. وافترض كذلك أنّك تريد أن نمنع المستخدم من كتابة أيّ حروف في مربّعات النصّ، بحيث يقتصر فقط على كتابة الأرقام.. في هذه الحالة ستضطرّ لتكرار كتابة الكود الذي يمنع المستخدم من كتابة الحروف في كلّ مربّع نصّ لديك.. الحلّ الأفضل هو أن تنشئ أداة جديدة ترث مربّع النصّ، وتحتوي على خاصيّة تحدّد نوعيّة البيانات التي يقبلها مربّع النصّ: نصوص، أرقام، أعداد عشريّة، لا يقبل علامات ترقيم... إلخ.. إنّ هذه الأداة ستكون مفيدة لك كثيرا في تطبيقات إدخال البيانات.
تعال نرى:

تصميم الأدوات:

إن تصميم الأداة Control لن يكون جديدًا عليك كلّيّةً، فهي تتكوّن من شقّين:

١- واجهة الأداة:

ويمكنك تصميم هذه الواجهة كما تصمّم أيّ نموذج عاديّ، حيث تمنحك اللغة كائنا شبيها تماما بالنموذج، هو "أداة المستخدم" UserControl، التي يمكنك أن تضع عليها الأدوات أو الرسوم، بنفس الطريقة التي تفعل بها هذا مع النموذج.



٢- كود الأداة:

لا بدّ أن تقوم أداتك بوظيفة أو أكثر، كما لا بدّ لها أن تمنح من سيستخدمها العديد من الخصائص والوسائل والأحداث التي سيبرمجها بها، تماما كما تفعل الأدوات التقليديّة.. لا مشكلة: ستطبّق نفس القواعد التي تعلّمتها في إنشاء الفئات Classes، لإنشاء الخصائص والوسائل والأحداث اللازمة للأداة.

إذن يمكنك أن تقول باختصار: إنّ الأداة هي فئة Class لها واجهة استخدام (نموذج).

ولكنّ هناك بعض الفروق الهامّة بينَ الأداة والنموذج، منها أنّ الأداة لا يمكن أن تستخدم بمفردها، بل يجب أن توضع على نموذج أولا ليتمّ عرضها من خلاله.. ومنها أنّ برمجة الأداة يجب أن تراعي حالتي التشغيل التاليتين للأداة:

١ - حالة تعامل المستخدم مع الأداة في وقت التشغيل.
٢ - حالة استخدام المبرمج للأداة في وقت التصميم، بوضعها على نموذج وتعديل شكلها وخصائصها، فأنت المسئول أيضا عن سلوك الأداة في هذه الحالة، فرغم أنّه وقت تصميم للمبرمج، إلا إنّه وقت تشغيل للأداة!

من أجل هذا تمنحك أداة المستخدم UserControl الخاصيّة "طور التصميم" DesignMode.. هذه الخاصيّة تُرجع القيمة True إذا كانت الأداة في وقت التصميم، وتُرجع القيمة False إذا كانت في وقت التشغيل.. وبهذا تستطيع أن تحدّد ردّ الفعل المناسب الذي تستجيب به لأحداث المبرمج أو المستخدم.

إنشاء مربّع النصّ الفعّال FocusedTextBox:

في هذا المثال سنبدأ بتطوير مربّع النصّ، بحيث يتغيّر لونه عندما يستقبل مؤشّر الكتابة، ويستعيد لونه الأصليّ عندما يفقد مؤشّر الكتابة.

ابدأ مشروعا جديدا، وفي مربّع الحوار New Project اختر النوع Class Library.. سمِّ المشروعَ FocusedTextBox واضغط موافق.. سيحتوي المشروع على فئة اسمها Class1.. احذفها من متصفح المشاريع، ثم اضغط القائمة الرئيسة Project واختر الأمر Add User Control.. سمّ الأداة الجديدة FocusedTextBox واضغط موافق.

سيعرض لك مصمم النوافذ أداة شبيهة بالنموذج، ولكن ليس لها أيّ حدود أو شريط عنوان.. هذه الأداة هي أداة المستخدم UserControl.. هذه هي الأرضيّة التي ستصمّم عليها واجهة الأداة.

افتح نافذة الخصائص، وغيّر اسم أداة المستخدم إلى FocusedTextBox.. ولجعل هذه الأداة ترث مربّع النصّ، اضغط بزر الفأرة الأيمن على أداة المستخدم ومن القائمة الموضعيّة اختَر View Code.. ستظهر لك الفئة التي تمثّل الأداة، وستجد فيها الكود التالي:

```
public class FocusedTextBox: UserControl
{
}
```

تلاحظ أنّ السطر الأول ينصّ على أنّ هذه الأداة ترث فئة أداة المستخدم UserControl.. لا نريد ذلك.. قم بتغيير هذا السطر لجعل الأداة ترث مربّع النصّ كالتالي:

```
public class FocusedTextBox: TextBox
```

الآن احفظ المشروع، وعد إلى تصميم الأداة.. ستكتشف أنّ أداة المستخدم قد اختفت، وظهرت مكانها مساحة فارغة.

ملحوظة:

إذا كانت أداة المستخدم ما زالت معروضة، فأغلق واجهة التصميم وأعد فتحها مرّة أخرى، وستجد التغيير المشار إليه قد حدث.

ولاختبار هذه الأداة يجب عليك إضافتها إلى نموذج.. لهذا سنضيف مشروعا جديدا إلى التطبيق سنستخدمه لاختبار الأداة أثناء تصميمها.. اضغط القائمة File\Add Project\New Project.. وفي مربّع حوار "مشروع جديد" اختر النوع Windows Application وسمّ المشروع TestProject.

الآن يمكنك اختبار الأداة بوضع نسخة منها على النموذج Form1 في مشروع الاختبار.. ولفعل ذلك اتبع هذه الخطوات:

- حدد العنصر FocusedTextBox في متصفّح المشاريع Solution Explorer، ومن القائمة Build اضغط الأمر Build FocusedTextBox، وذلك لبناء ملف Dll للأداة يمكن استخدامه.
- حدد العنصر TestProject في متصفّح المشاريع، واجعل هذا المشروع هو مشروع بدء التشغيل، بضغط الأمر Set As Startup Project الموجود في القائمة الموضعية أو في القائمة الرئيسية Project.
- افتح النموذج Form1.
- في صندوق الأدوات ToolBox ستجد اسم الأداة FocusedTextBox موجودا أعلى الشريط.. ضع نسخة منها على النموذج.
- الآن حاول استخدام هذه الأداة.. ستكتشف أنّك تتعامل مع مربّع النصّ التقليديّ بدون أيّ اختلاف.

والآن سنضيف بعض الوظائف الجديدة لمربّع النصّ.. سنستخدم حدث دخول الأداة Enter Event.. لتغيير لون مربّع النصّ.. هذا الحدث ينطلق كلّما دخل مؤشّر الكتابة إلى مربّع النصّ.. لإضافة معالج لهذا الحدث، اعرض الأداة في وضع التصميم، وافتح نافذة الخصائص، واضغط زر عرض الأحداث، وانقر مرتين على الحدث Enter.. ثم اكتب الكود التالي في الإجراء المعالج للحدث:

```
private void FocusedTextBox_Enter(object sender,
                EventArgs e)
{
   this.BackColor = Color.Cyan;
}
```

لاحظ أنّ الكلمة this هنا تشير إلى أداة المستخدم، وهي في حالتنا هــذه مربّع النصّ.

كذلك يجب برمجة الحدث Leave، بحيث نعيد فيه لون مربّع النصّ إلى طبيعته.. هذا الحدث ينطلق عند مغادرة مؤشّر الكتابة للأداة:

```
private void FocusedTextBox_Leave(object sender,
          EventArgs e)
}
    this.BackColor = Color.White;
}
```

كتابة بعض الخصائص:

ما رأيك الآن أن نضيف بعض الخصائص الجديدة لمربّع النصّ الخــاص بنا؟.. هذه هي الخصائص التي سنضيفها:

- "إلزامي" Mandatory:

عند جعل قيمتها True سنغيّر لون مربع النصّ إذا كان فارغا، حتّــى نشعر المستخدم بحتميّة إدخال قيمة فيه.

- MandatoryColor:

تسمح للمبرمج باختيار اللون الذي سيتم عرضه إذا كان مربع الــنص فارغا.

- EnterFocusColor:

تحدد اللون الذي سيعرضه مربّع النصّ عند دخول مؤشّر الكتابة إليه.

- LeaveFocusColor:

تحدد اللون الذي سيعرضه مربّع النصّ عند خروج مؤشّر الكتابة منه.

في الحقيقة لا تحتاج هذه الخصائص لكتابة أي كود، لهذا سنجعلها ذاتيــة التعريف Auto Implemented:

```
public bool Mandatory {get; set;}
public Color EnterFocusColor { get; set; }
public Color LeaveFocusColor { get; set; }
public Color MandatoryColor { get; set; }
```

سنحتاج الآن إلى تعديل كود الحدثين Enter و Leave ليراعيا قيم هــذه الخصائص:

```
private void FocusedTextBox_Enter(object sender,
        EventArgs e)
{
    this.BackColor = EnterFocusColor;
}

private void FocusedTextBox_Leave(object sender,
        EventArgs e)
{
    // إذا كانت خاصية الإلزام صحيحة، وكان مربّع النصّ فارغا أو به مسافات
    // فغيّر لون مربّع النصّ إلى لون الإلزام
   if (this.Text.Trim( ).Length == 0 && this.Mandatory)
      this.BackColor =  this.MandatoryColor;
   else // غير لون مربّع النصّ إلى لون المغادرة
      this.BackColor =  this.LeaveFocusColor;
}
```

والآن لتجربة هذه الإضافات، أعد بناء الأداة من جديد، وانتقل إلى نموذج الاختبار.. ضع عليه عدّة نسخ من الأداة، وغيّر الخصائص الجديدة التــي

أنشأناها لكلّ منها باستخدام نافذة الخصائص.. ثمّ شغّل البرنامج واختبر هذه الخصائص.

تبويب الخصائص:

إذا كنت تعرض الخصائص في نافذة الخصائص مرتّبة على حسب النوع، فستجد أنّ الخصائص التي أنشأناها قد ظهرت كلّها تحت التصنيف العام Misc.. ولتغيير هذا الوضع الافتراضيّ، يمكن كتابة بعض السمات Attributes أمام تعريف الخاصيّة.. فمثلا لجعل الخاصيّة تظهر تحت التصنيف Appearance، استخدم السمة التالية:

```
[Category("Appearance")]
```

ويمكنك أن تضيف تصنيفا جديدا غير موجود سابقا في نافذة الخصائص، حيث سيتمّ إنشاؤه من أجلك، وذلك بمجرّد كتابة اسم النوع الجديد كالتالي:

```
[Category("Mine")]
```

ولاستخدام هذه السمات، يجب أن تكتب جملة الاستخدام التالية أعلى كود الأداة:

```
using System.ComponentModel;
```

وهناك سمة أخرى تهمّك، هي وصف الخاصيّة، ذلك الذي يظهر في الجزء السفليّ من نافذة الخصائص، ليشرح وظيفة الخاصيّة.. أمّا أهمّ سمة، فهي سمة القيمة الافتراضيّة للخاصيّة DefaultValue، حيث يمكنك كتابة القيمة الافتراضيّة بين القوسين.

انظر الآن كيف سنعدّل تعريف خاصيّة Mandatory، لنضيف هذه السمات:

```
[Description("توضّح إذا كان من الممكن ترك الأداة فارغة أم لا"),
 Category("Appearance"), DefaultValue(False)]
public bool Mandatory {get; set;}
```

وهناك سمات لفئة الأداة نفسها.. فمثلا، يمكنك أن تجعل الخاصيّة Mandatory هي الخاصيّة الافتراضيّة للفئة، التي يتم تحديدها تلقائيا في نافذة الخصائص عند عرض خصائص هذه الأداة.. عدّل تعريف الفئة كالتالي:

```
[DefaultProperty("Mandatorry")]
public class FocusedTextBox : TextBox
```

ويمكنك اختبار كلّ ذلك في المشروع FocusedTextBox في مجلّد برامج هذا الفصل.

حل مشكلة اليمين لليسار في بعض الأدوات:

في الفصل الحادي عشر، سبق أن تكلمنا عن عرض الشجرة من اليمين لليسار، وذلك باستخدام تقنية المرآة عبر دوال API.

لكن لحسن الحظّ أننا نملك طريقة أخرى أكثر أمانا، يمنحها لنا إطار العمل.. فأنت تعرف أن دوال API خاصة بالويندوز وتعتمد على نوع الإصدار.. لهذا فإن الكود الذي يستخدمها هو كود غير مدار Unmanaged Code.. بينما الكود الذي يعتمد على أدوات إطار العمل يسمّى كودا مدارا Managed Code، وهو أكثر أمنا وسهولة من الكود غير المدار.

هنا سنرى كيف نستخدم تقنية المرآة عبر إطار العمل.. المشكلة أنّنا هنا نحتاج لإنشاء أداة جديدة تستخدم هذه التقنية، على عكس ما كنا نفعله بدوال API التي تؤثر على الأدوات الموجودة مباشرة.

تعال نرى كيف نطور أداة الشجرة، لتعرض عناصرها من اليمين لليسار، عندما تتغير قيمة الخاصية RightToLeft إلى True.

ابدأ مشروعا جديدا من النوع Class Library، وسمّه RightToLeftTree (ستجد هذا المشروع في مجلد برامج هذا الفصل).

احذف الفئة Class1 وأضف أداة مستخدم User Control وامنحها الاسم RightToLeftTree.. ثم افتح نافذة محرّر الكود للأداة،وغير السطر:

public class RightToLeftTree : UseControl

إلى:

public class RightToLeftTree : TreeView

هذه الأداة الآن تمثّل الشجرة التقليدية.. نحتاج لإضافة الكود الذي يصحّح عمل الخاصية RightToLeft.

هذا هو الكود:

```
private const int LAYOUTRTL = 0x400000;
private const int NOINHERITLAYOUT = 0x100000;
protected override CreateParams CreateParams
{
  get
  {
     CreateParams MirrorExStyle = base.CreateParams;
     if (this.RightToLeft == RightToLeft.Yes)
        MirrorExStyle.ExStyle = MirrorExStyle.ExStyle |
              LAYOUTRTL | NOINHERITLAYOUT;
     return MirrorExStyle;
  }
}
```

أظنّ الكود واضحا.. لقد قمنا باستبدال Override خاصية إنشاء المعاملات CreateParams.. هذه الخاصية موجودة في النموذج والأدوات.. وهي تُستدعى عند تغيّر خصائص الأداة وعند رسمها.. كل ما سنفعله، هو أن نحصل على معاملات الأداة عن طريق الخاصية الأصلية:

```
CreateParams MirrorExStyle = base.CreateParams;
```

حيث base تشير إلى الفئة الأم التي نرث منها وهي هنا فئة الشجرة.. بعد هذا نتأكد من قيمة الخاصية RightToLeft للشجرة.. فإن كانت True، فعلينا أن نعرض الشجرة من اليمين لليسار، وذلك بإجراء عملية Or بين قيمة المعاملات الحالية للشجرة وبين الثابت LAYOUTRTL الذي عرّفناه ليدلّ على عرض الشجرة من اليمين لليسار.. ولا نستخدم عملية تساوي مباشرة، حتّى تحافظ على باقي قيم المعاملات بدون تغيير.

أمّا الثابت NOINHERITLAYOUT فهو يمنع توريث هذا المظهر للأدوات المحتواة داخل هذه الشجرة إن وجدت.. هذا ليس مفيدا هنا، لكن

يمكن أن يكون مفيدا لو كنت تتعامل مع أداة حاوية Container Control مثل النموذج واللوحة Panel.

لاحظ أن هذه الطريقة صالحة لاستخدامها مع أيّ أداة أخرى بنفس الكود بدون تغيير أي حرف، سوى اسم الأداة فقط!.. بهذا يمكنك تطوير العديد من الأدوات التي لا يتم عرضها من اليمين لليسار، مثل ProgressBar و ToolBar وغيرها.

الأمر بسيط كما ترى.

بقي شيء صغير.. عند عكس عرض الشجرة، سيفسد عرضها بإطار مجسم 3D Border، وذلك نتيجة عكس موضع الخط الأبيض مع الخط الأسود الذي يوحي بالعمق.. لهذا فمن الأفضل ألا تعرض هذه الأداة مجسمة.. لفعل هذا أضِف هذا السطر لمنشئ الفئة Constructor:

```
this.BorderStyle = BorderStyle.FixedSingle;
```

ولو أردت، يمكنك استخدام وسائل الرسم والتلوين في الحدث Paint لرسم خط أسود على الحافة اليسرى للشجرة، وآخر على الحافة العليا، وخطين رماديين على الحافتين اليمنى والسفلى، ليوحي هذا بالعمق، فتبدو الشجرة مجسمة.

قم ببناء المشروع، وأضِف مشروع اختبار للأداة (واجعله هو مشروع البداية Startup Project)، وضع نسخة من الشجرة المعدلة على النموذج، وأضف إليها عدة عناصر (باستخدام الخاصية Nodes)، ثم غيّر قيمة الخاصية RightToLeft إلى True ثمّ إلى False ثمّ إلى True، وانظر كيف سيتغير عرض الشجرة أثناء التصميم.. ثمّ شغّل المشروع وانظر لأداتنا الرائعة.

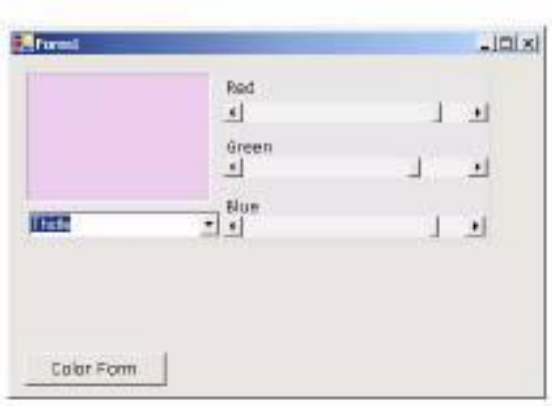
إنشاء أدوات مركّبة

في هذا المقطع لن نرث أداةً ونطوّرها، ولكنّنا سنستخدم مجموعة من الأدوات تقوم بوظيفة معيّنة، لإنشاء أداة جديدة مركّبة.

تعال ننشئ أداة تسمح للمستخدم باختيار اللون.. هذه الأداة ستتكوّن من الأجزاء التالية:

- ثلاثة منزلقات أفقيّة، تمثّل نسب الألوان الأساسيّة: الأحمر والأخضر والأزرق، بحيث يستطيع المستخدم تكوين اللون من نسبه.
- قائمة مركّبة منسدلة، يختار المستخدم منها اسم اللون مباشرة.
- لافتة سنجعل لون خلفيّتها هو اللون الذي سيركّبه المستخدم أو يختاره، حتّى يستطيع معاينة اللون.

أنشئ مشروع أداة ويندوز جديدا بنفس الطريقة التي تعلمناها في المشروعين السابقين، وسمّ المشروع ColorEdit، وصمّم أداة المستخدم لتبدو كما في الصورة.



سمّ المنزلقات الأفقيّة: RedBar، و GreenBar و BlueBar، واجعل قيمة الخاصيّة MinimumValue لكلّ منها صفرا، وقيمة الخاصيّة

MaximumValue لكلّ منها ٢٥٥.. سنستخدم لافتة أو لوحة Panel لمعاينة اللون.. أعطها الاسم pnlColor، وسمّ القائمة المركّبة CmbNamedColors.. ولملء هذه القائمة بأسماء ألوان النظام وأسماء الألوان الإنجليزية المعروفة، سنستخدم حيلة بسيطة، وهي الحصول على جميع هذه الأسماء من المرقم KnownColor.. لفعل هذا سنستخدم الوسيلة Enum.GetNames التي تستقبل نوع المرقم المراد الحصول على جميع مسميات الأرقام التي يحتويها، وتعيد مصفوفة نصية تحتوي على هذه الأسماء.. ولملء القائمة المركبة بهذه الأسماء، سنرسل هذه المصفوفة إلى الوسيلة Items.AddRange الخاصة بالقائمة المركبة.. ضع هذه الكود في حدث تحميل النموذج Load:

```
String[ ] Colors = Enum.GetNames(typeof(KnownColor));
cmbNamedColors.Items.AddRange(Colors);
```

ولتسهيل وصول المستخدم إلى اللون الذي يريده، عليك بجعل القائمة المركبة تعرض الأسماء مرتبة:

```
cmbNamedColors.Sorted = true;
```

والآن دعنا نضيف إلى الأداة الخاصة بنا خاصيتين جديدتين، هما "اللون المحدّد" SelectedColor و"اللون المسمّى" NamedColor، ها هو ذا كود الخاصيّة الأولى:

```
public Color SelectedColor
{
   get
   {
      تعيد هذه الخاصيّة لون اللافتة //
      return pnlColor.BackColor;
   }
```

```
    set
    {
        // عند تغيير قيمة هذه الخاصيّة
        // يجب أن تظهر مكوّنات اللون الجديد على المنزلقات
        RedBar.Value = value.R;
        GreenBar.Value = value.G;
        BlueBar.Value = value.B;
    }
}
```

أما الخاصيّة NamedColor فسنجعلها للقراءة فقط، وكودها كالتالي:

```
public string NamedColor
{
    get
    {
        // هذه الخاصيّة ترجع قيمة العنصر المحدّد حاليّا في القائمة
        return (string)cmbNamedColors.SelectedItem;
    }
}
```

وعندما يختار المستخدم اسم لون من القائمة، فعليك أن تكوّن اللون المقابل للاسم، باستخدام الوسيلة Color.FromName.. أضف هذا الكود للحدث SelectedIndexChanged الخاص بالقائمة المركبة:

```
if (cmbNamedColors.SelectedIndex != -1)
{
    string ColorName = (string)
                        cmbNamedColors.SelectedItem;
    var Clr = Color.FromName(ColorName);
    pnlColor.BackColor = Clr;
    RedBar.Value = Clr.R;
    GreenBar.Value = Clr.G;
    BlueBar.Value = Clr.B;
}
```

بقيت آخر خطوة، وهي كتابة إجـــراء يســـتجيب للحـــدث Scroll فـــي المنزلقات الثلاثة.. لفعل هذا حدد المنزلقات الثلاثة على النافـــذة، وافـــتح نافذة الخصائص، واجعلها تعرض أسماء الأحداث، واعثر علـــى الحـــدث Scroll واكتب في الخانة المجاورة له ColorScroll واضغط Enter.. اجعل كود هذا الإجراء يبدو كالتالي:

```
pnlColor.BackColor = Color.FromArgb(
        RedBar.Value, GreenBar.Value,
        BlueBar.Value);
```

الآن لم يبقَ إلا اختبار أداة الألوان.. ولفعل ذلك، أضف مشروع وينـــدوز جديدا لاختبار الأداة، واجعله مشروع بدء التشغيل.. قم ببناء الأداة وأضف نسخة منها لنموذج الاختبار.. شغّل التطبيق وانظر كيف يمكنـــك اختيـــار الألوان باستخدام المنزلقات أو القائمة.

أوقف التشغيل وأضف زرّا للنموذج، واكتب فيه الكود التالي، لتغيير لون خلفيّة النموذج تبعا للون الأداة:

```
this.BackColor = colorEdit1.SelectedColor;
```

بناء أدوات بأشكال خاصّة

في هذه المرّة لن نضع أيّ أداة على أداة المستخدم، بل سنقوم نحن بالرسم عليها، بإعادة كتابة وسيلة الرسم OnPaint لتستبدل Overrides وسيلة الرسم الخاصّة بالأداة.. تعال نرى ذلك عمليّا:

اللافتة المجسّمة

سنقوم هنا بإنشاء لافتة تعرض نصّا مجسّما.. وأنصحك هنا بمراجعة فصل التلوين، فقد قمنا فيه بإنشاء المشروع Text Effects الذي يرسم نصّا مجسّما على النموذج.. سنقوم هنا بنفس الأمر، لكنّنا سنرسم في هذه المرّة على أداة المستخدم وليس على النموذج.

ابدأ مشروعا جديدا من النوع Class Library وسمّه Label3D.. احذف الفئة Class1 وأضف UserControl وامنحها الاسم Label3D أيضا.

إضافة الخصائص:

من الخصائص التي سنضيفها لهذه اللافتة، خاصيّة المحاذاة، وهي تأخذ عدّة قيم، سنضعها في المرقّم Align.. ابدأ بتعريف هذا المرقّم في بداية فئة الأداة:

```
public enum Align
{
    TopLeft,
    TopMiddle,
    TopRight,
    CenterLeft,
    CenterMiddle,
    CenterRight,
    BottomLeft,
    BottomMiddle,
    BottomRight
}
```

أمّا النصّ الذي ستعرضه اللافتة، فسنسمح للمبرمج بتحديد تأثير عرضه، وذلك من خلال الخاصيّة Effect3D التي تأخذ إحدى قيم المرقّم التالي:

```
public enum Effect3D
{
    None, // تأثير بدون
    Raised, // مرتفع
    RaisedHeavy, // سميك مرتفع
    Carved, // محفور
    CarvedHeavy // سميك محفور
}
```

الآن سنكتب خاصيتي المحاذاة والتأثير المجسّم:

```
private Align mAlignment;
public Align Alignment
{
   get
   {
       return mAlignment;
   }
```

```
    set
    {
        mAlignment = value;
        this.Invalidate( );
        if (AlignmentChanged != null)
            AlignmentChanged(this, new EventArgs( ));
    }
}

private Effect3D mEffect;
public Effect3D Effect
{
    get
    {
        return mEffect;
    }
    set
    {
        mEffect = value;
        this.Invalidate( );
        if (EffectChanged != null)
            EffectChanged(this, new EventArgs( ));
    }
}
```

لاحظ استخدامنا للوسيلة Invalidate، حتّى تعيد رسم النصّ من جديد على سطح اللافتة، وبهذا يتمّ تطبيق القيمة الجديدة للخاصيّة على النصّ.

أضف مشروع اختبار للتطبيق، وحاول أن تجرّب الأداة.. ضع نسخة منها على النموذج، وانظر لخصائصها في نافذة الخصائص.. ستكتشف أنّ النافذة لا تحتوي فقط على الخاصيتين اللتين عرّفناهما، بل إنّ هناك عددا هائلا من الخصائص الجاهزة التي تتعامل مع شكل وموقع اللافتة وألوانها

وخطوطها.. هذه الخصائص لم تكتبها أنت، ولكنّ أداة المستخدم منحتهـا لك لحسن الحظّ.

ولكن رغم هذا ما زلنا بحاجة للخاصيّة Text3D التـي سـيكتب فيهـا المستخدم النصّ الذي يريد عرضه في اللافتة.. تعال نكتب هذه الخاصيّة:

```
private string mText;
public string Text3d
{
    get
    {
        return mText;
    }

    set
    {
        mText = value;
        this.Invalidate( );
        if (Text3DChanged != null)
              Text3DChanged(this, new EventArgs( ));
    }
}
```

لاحظ أننا في كود تغير كل خاصية set أطلقنا حدثا يخبر المبرمج لتغير قيمة هذه الخاصية.. هـذه الأحـداث Events ومنـدوباتها Delegates معرفة على مستوى الفئة كالتالي:

```
حدث تغير المحاذاة //
public delegate void AlignmentChangedEvHnd(
        object sender, EventArgs ev);
public event AlignmentChangedEvHnd AlignmentChanged;
```

```
// حدث تغير تأثير التجسيم
public delegate void EffectChangedEEvHnd(
        object sender, EventArgs ev);
public event EffectChangedEvHnd EffectChanged;
// حدث تغير النص المجسم
public delegate void Text3DChangedEvHnd(
        object sender, EventArgs ev);
public event Text3DChangedEvHnd Text3DChanged;
```

رسم النصّ المجسّم:

نحتاج الآن لكتابة الوسيلة OnPaint.. هذه الوسيلة ستستدعيها الأداة تلقائيّا كلّما احتاجت لأن يعاد رسمها وعند استخدام الوسيلة Invalidate.. ابدأ بوضع الجملة التالية في بداية الملفّ:

```
using System.Drawing.Drawing2D;
```

والآن تعال نرى كيف سنرسم النصّ بتأثيراته ومحاذاته:

```
protected override void OnPaint(PaintEventArgs e)
{
   Font lblFont = this.Font;
   SolidBrush lblBrush = new SolidBrush(Color.Red);
   int X = 0;     int Y = 0;
   // يجب معرفة أبعاد النصّ حتّى نستطيع محاذاته
   SizeF TextSize = e.Graphics.MeasureString(mText,
                        lblFont);
   switch (this.mAlignment)
   {
      case Align.BottomLeft:
         // رسم النصّ في الركن السفليّ الأيسر
         X = 2; // نقطتان لمراعاة حافة الأداة
         Y = (int)(this.Height - TextSize.Height);
         break;
```

```
case Align.BottomMiddle:
   // رسم النصّ في أسفل المنتصف
   X = (int)(this.Width - TextSize.Width) / 2;
   Y = (int)(this.Height - TextSize.Height);
   break;
case Align.BottomRight:
   // رسم النصّ في الركن السفليّ الأيمن
   X = (int)(this.Width - TextSize.Width - 2);
   Y = (int)(this.Height - TextSize.Height);
   break;
case Align.CenterLeft:
   // رسم النصّ في منتصف اليسار
   X = 2;
   Y = (int)(this.Height - TextSize.Height) / 2;
   break;
case Align.CenterMiddle:
   // رسم النصّ في مركز اللافتة
   X = (int)(this.Width - TextSize.Width) / 2;
   Y = (int)(this.Height - TextSize.Height) / 2;
   break;
case Align.CenterRight: // رسم النصّ في منتصف اليمين
   X = (int)(this.Width - TextSize.Width - 2);
   Y = (int)(this.Height - TextSize.Height) / 2;
   break;
case Align.TopLeft: // رسم النصّ في أعلى اليسار
   X = 2;
   Y = 2;
   break;
case Align.TopMiddle: // رسم النصّ في أعلى المنتصف
   X = (int)(this.Width - TextSize.Width) / 2;
   Y = 2;
   break;
```

```
    case Align.TopRight: // رسم النصّ في أعلى اليمين
        X = (int)(this.Width - TextSize.Width - 2);
        Y = 2;
        break;
}
// المتغيّران التاليان سنضع فيهما
//مقدار إزاحة النصّ رأسيّا أو أفقيّا حتّى يبدو مجسّما
int dispX = 0;
int dispY = 0;
switch (mEffect)
{
    case Effect3D.None:
        dispX = 0;
        dispY = 0;
        break;
    case Effect3D.Raised:
        dispX = 1;
        dispY = 1;
        break;
    case Effect3D.RaisedHeavy:
        dispX = 2;
        dispY = 2;
        break;
    case Effect3D.Carved:
        dispX = -1;
        dispY = -1;
        break;
    case Effect3D.CarvedHeavy:
        dispX = -2;
        dispY = -2;
        break;
}
```

```
    رسم النصّ بلون أبيض //
    lblBrush.Color = Color.White;
    e.Graphics.DrawString(mText, lblFont, lblBrush, X, Y);
    رسم النصّ مرّة أخرى بلون أسود، //
    مع إزاحته قليلا رأسيا وأفقيّا ليبدو التأثير المجسّم //
    lblBrush.Color = this.ForeColor;
    e.Graphics.DrawString(mText, lblFont, lblBrush,
      X + dispX, Y + dispY);
}
```

الآن حاول تجربة ما فعلناه.. ستكتشف كم هو رائع!

حاول كذلك أن ترى أحداث اللافتة المجسّمة.. ستكتشف أنّ لها عددا كبيرا من الأحداث الجاهزة التي لم تكتبها أنت.. منتهى الراحة!

جعل خلفيّة الأداة شفّافة:

ما رأيكُ أن نُضيفَ إمكانيّة جديدةً للافتة المجسّمة، بحيثُ يمكنك أن تجعل خلفيّتها شفّافة؟.. سيضمنُ هذا مظهرا أفضلَ للنموذج عند وضع صورة في خلفيّته، حيثُ لن تحجب اللافتة ذلك الجزء من الصورة الذي يقع وراءها.

الأمرُ في غاية البساطة: اكتب الجملة التالية في منشئ الفئة Constructor الخاص بالأداة:

```
SetStyle(ControlStyles.SupportsTransparentBackColor,
         true);
```

ستسمح لك هذه الجملة بجعل خلفيّة النموذج أو الأداة شفّافة.. ويمكنك أن تجرّب ذلك.. قم ببناء الأداة من جديد، ثمّ انتقل لنموذج الاختبار.. حدّد اللافتة المجسّمة، وفي نافذة الخصائص حدّد الخاصيّة BackColor، وفي خانة القيمة اضغط زرّ الإسدال.. في مربّع الألوان اضغط

الشريط Web.. ستجد أنّ أول قيمة في هـذا الشـريط هـي "شـفّاف" Transparent.. اختر هذه القيمة.. وحتّى تتأكّد بالفعلِ أنّ خلفيّة اللافتـة شفّافة، غيّر لون النموذج، أو أضف لخلفيّته صورة (عن طريق الخاصيّة BackgroundImage).

وضع القيم المبدئيّة للأداة:

لتحديد القيم المبدئيّة للأداة، تلك التي ستأخذها خصائصها عنـد وضـعها على النموذج لأوّل مرّة، استخدم منشـئ الفئـة Constructor (وهـو الإجراء الذي يحمل نفس اسم الفئة).. اكتب فيه ما يلي:

```
public Label3D( ) : base( )
{
    InitializeComponent( );
    mText = "Label3D";
    mAlignment = Align.CenterMiddle;
    mEffect = Effect3D.Raised;
    SetStyle(ControlStyles.ResizeRedraw, true);
    SetStyle(
        ControlStyles.SupportsTransparentBackColor,
        true);
     this.BackColor = Color.Transparent;
     this.Font = new Font(this.Font.Name, 14,
        FontStyle.Bold);
}
```

ومن الجدير ذكرا، هو أنّ أداة المستخدم User Control تمتلك خاصـيّة جاهزة تريحك من استدعاء الوسيلة SetStyle.. هـذه الخاصـيّة هـي ResizeRedraw، والتي عند جعل قيمتهـا True تجبـر الأداة علـى

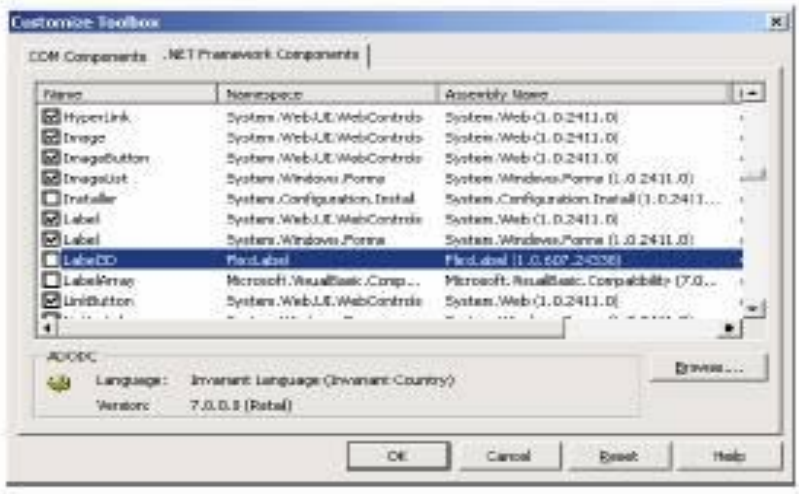
استدعاء حدث الرسم عندما يتغيّر حجمها.. ويمكنك استبدال الجملة التالية بجملة SetStyle الأولى في الكود السابق:

this.ResizeRedraw = true;

استخدام اللافتة المجسمة في مشاريع أخرى:

لكي تستخدم اللافتة المجسّمة في مشروع آخر اتبع هذه الخطوات:

- ابدأ مشروعا جديدا، واعرض نموذجه.
- اضغط صندوق الأدوات بزر الفأرة الأيمن، ومن القائمة الموضعيّة اضغط الأمر Customize Toolbox.



- في مربّع الحوار الذي سيظهر لك، اضغط الشريط .NET Framework Components..
- اضغط زرّ التصفّح Browse، وفي مربّع حوار "فتح ملفّ" افتح الملفّ Label3D.dll (ستجده في المجلّد Bin في مجلّد مشروع اللافتة المجسمة).

- ســـتجد أنّ اللافتـــة المجسّـــمة قـــد أضـــيفت للقائمـــة NET Framework components... تأكّد أن مربّع الاختيـــار الموجود أمام اسمها عليه العلامة (**ü**)، واضغط OK.
- الآن ستظهر أيقونة اللافتة المجسّمة في صندوق الأدوات، حيـث يمكنك استخدامها بالطريقة التقليديّة.

تصميم أدوات بأشكال غير تقليديّة:

رأيت في الفصل الخامس كيف أنشأنا نموذجا بيضاويّا.. لا يوجد ما يمنع من تنفيذ نفس الأمر مع أداة المستخدم لتبدو بيضاويّة أو تتخذ أيّ شكل تريده.

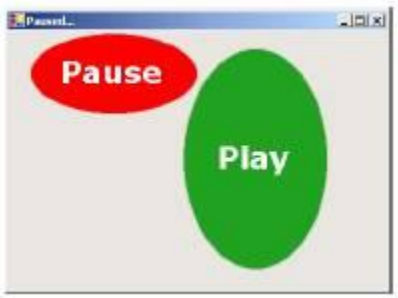
سنقوم هنا بالمثل، ولكن ليس باستخدام الخاصيّة TransparencyKey، ولكن باستخدام الخاصيّة "منطقة الرسم" Region الخاصّة بأداة المستخدم، والتي تتيح لنا تحديد منطقة رسم الأداة.

ملحوظة:

يمتلك النموذج أيضا الخاصيّة Region، حيث يمكنك استخدامها لتغيير شكل النموذج، بنفس الطريقة التي سنشرحها مع أداة المستخدم هنا.. ولكنّ هذه الخاصيّة لا توجد في قائمة الإكمال التلقائي (التي تظهر أثناء كتابة الكود).. ويمكنك أن تعرض كلّ خصائص ووسائل النموذج في قائمة الأعضاء، باتّباع التالي: افتح القائمة Tools واضغط الأمر Options.. في النافذة التي ستظهر لك، اضغط العنصر Text Editor، ومن عناصره الفرعيّة اختر Basic ومن عناصره الفرعيّة اختر General.. على اليسار ستجد مربّعات اختيار.. أزل العلامة من الاختيار التالي "إخفاء الأعضاء المتقدّمة" Hide Advanced Members.. سيؤدّي هذا لظهور كلّ أعضاء النموذج والأدوات في قائمة الأعضاء، ومن بينها الخاصيّة Region.

انظر للمثال التالي:

```
protected override void OnPaint(PaintEventArgs e)
{
   GraphicsPath roundPath = new GraphicsPath( );
   Rectangle R = new Rectangle(0, 0, this.Width,
                this.Height);
   Graphics G = this.CreateGraphics( );
   roundPath.AddEllipse(R);
   this.Region = new Region(roundPath);
   // الخطوة التالية اختياريّة، وهي ترسم حافة رمادية حول منطقة الأداة
   G.DrawEllipse(new Pen(Color.DarkGray, 3), R);
}
```



وللتدريب، لديك في مجلّد برامج هذا الفصل المشروع RoundButton.. وقتا ممتعا.

الأدوات من النوع ActiveX:

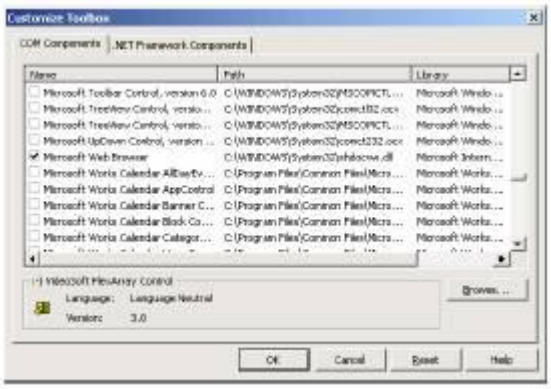
لعلّك لاحظت أنّ الأدوات الجديدة التي تنشئها في سي شارب لها الامتداد dll. .. لم يكن هذا هو الحال سابقا، حيث كانت التقنية المسمّاة COM هي المستخدمة، وكانت الأدوات تسمّى ActiveX ولها الامتداد ocx. .

ولكن.. هل معنى هذا أن تقنية COM قد أقصيت نهائيّا؟

لا، فهناك آلاف الأدوات التي تمّ إنتاجها من تلك النوعيّة، ممّا يعني أنّ إهمالها مرّة واحدةً سيكون خسارة فادحةً للمبرمجين.

لقد كانت (ميكروسوفت) تطلق على تقنية Net. في بدايتها COM2، حيث قامت بإعادة بناء تقنية COM من جديد، لحلّ المشاكل التي كانت تلازمها.. لهذا ما زال باستطاعتك استخدام أدوات COM في سي شارب.

ولإضافة هذه الأدوات لمشروعك، اضغط صندوق الأدوات بزر الفأرة الأيمن، ومن القائمة الموضعيّة اضغط Customize Toolbox.. وفي مربّع الحوار اضغط الشريط COM Components، وأضِف الأداة التي ترغب فيها.



ولقد رأيت في فصل الرسم والتلوين، كيف استخدمنا الأداة Script control لرسم أيّ دالّة يكتبها المستخدم.

إنّ عالم أدوات ActiveX رحيب، ويمتلئ بالإمكانيّات الرائعة.. وإن كان تعاملك معها يستلزم أن تحصل على ملفّات الإرشادات التي تشرح كيفيّة استخدامها، فهذه المعلومات لن تجدها بالطبع في إرشادات سي شارب.

آخر ما أحبّ أن ألفت نظرك إليه، هو أنّ سي شارب تسمح لك بتحويل الفئات والأدوات التي تنشئها إلى تقنية COM، لتتوافق مع اللغات التي تتعامل مع هذه التقنية.. للأسف.. هذا موضوع معقّد ويحتاج لمرجع أكثر تقدّما.

الفصل الثالث

مشروع تعليميّ لـ Direct3D

مقدمة حول Microsoft DirectX 9.0:

منذ فترة، وميكروسوفت معنيّة بتطوير أدواتها الخاصّة بالرسوم والألعاب والوسائط المتعدّدة Multimedia.. وفي هذا الصدد، قامت ميكروسوفت بتطوير تقنية DirectX.. ويعنينا هنا أن نتوقّف قليلا أمام هذه التسميّة.

أمّا بخصوص الكلمة "مباشر" Direct، فهي تشير إلى أنّ هذه التقنية تستخدم مجموعة منخفضة المستوى Low Level من واجهات برمجة التطبيقات Application Programming Interfaces (APIs)، التي تتيح لمن يستخدمها أن يتعامل ((**مباشرةً**)) مع مكونات الحاسوب المادية، بدون وسيط من تقنية COM، ممّا يضمن سرعةً وكفاءةً وتحكّما أعلى في تنفيذ البرنامج.. لهذا تحتاج حوالي ٧٥% على الأقّل من الألعاب لإصدار من إصدارات DirectX المختلفة لكي تعمل.. إنّ هذا يشي بوضوح بأهمّيّة هذه التقنية، ومدى الإمكانيّات التي تقدّمها.

وقبلَ أن ينقبضَ صدرك مع سماعك لتعبيرات Low Level و API و COM، يجب أن أخبركَ أنّ شيئا لن يختلف بالنسبة لك كمبرمج سي شارب، فقد قامت ميكروسوفت بإنتاج نسخة من DirectX في صورة فئات Classes، بحيث يمكنك استخدامها من داخل سي شارب بنفس الطريقة التي تستخدم بها باقي فئات إطار العمل Framework.. تنفّس الصُعَداء إذن.

أما الحرف X، فهو يقوم عوضا عن أيّ مجهول، يمكن أن يتّخذ مجموعة مختلفة من القيم.. هذا يعني أنّ DirectX تحتوي على مجموعة مختلفة من الأدوات، مثل Direct3D و DirectDraw و DirectInput.

مفاهيم أساسيّة

عالم المجسّمات:

اكتسبَ الكمبيوتر في الفترةِ الأخيرة قوّةً هائلة، حتّى لقد صار بإمكانك أن تصمّم وتعرض الرسوم المجسّمة على جهازك الشخصيّ.. حتّى لقد وصل الأمر إلى إنتاج أفلام كاملة بالرسوم المجسّمة، التي تذهل من يشاهدها من فرط مقاربة تفاصيلها من الواقع.

ولكن لا تظنّ الأمر تافها، فما زالت عمليّة تكوين المجسّمات على أجهزة الكمبيوتر Rendering بطيئة، نظرا للكمّ الهائل من العمليّات التي يتمّ إجراؤها، حتّى إنّ تكوين صورة مجسّمة واحدة (أي تحويلها من طور التصميم إلى ملفّ فيديو)، قد يستغرق عدّة دقائق ــ على أحسن الفروض.

طبعا ستقول في نفسك مستخفّا: وماذا في عدّة دقائق؟

ولكي تعرف الإجابة، يجب أن أذكّرك أنّ الرسوم المتحرّكة عبارة عن مجموعة هائلة من الصور، يتمّ عرضها متتابعة بسرعة (أكثر من ٢٠ لقطة في الثانية)، بحيث تنخدع عين الرائي فترى الرسوم تتحرّك.. هل تخيّلت إذن كم صورة في دقيقة واحدة؟.. أكثر من ١٢٠٠ صورة.. حاول أن تتخيّل إذن الوقت الذي ستستهلكه عمليّة تكوين مشهد رسوم مجسّمة مدّته دقيقة واحدة!

ولكن لا تجزع.. إنّ التطوّر السريع لإمكانيات الكمبيوتر يجعل هذا الوقت يتقلّص باستمرار.

ثمّ إنّنا ــ كمبرمجين ــ لن نقوم بهذه العمليّة.. إنّ كلّ ما سنفعله هو استلام المجسّمات من مصمّميها (بالاستعانة بتطبيقات مثل 3DS Max

و TrueSpace و Lightwave و Maya وغيرها)، واستخدامها في تطبيقاتنا كما يحلو لنا.

وفي هذا الفصل، سنركّز جهودنا لتعلّم بعض القدرات المدهشة التي يمنحها لنا DirectX عبر تقنية Direct3D.

إنّ Direct3D يقف بيننا وبين كارت الشاشة Video Card مباشرةً، وذلك منعا لإضاعة أيّ وقت، فكما ذكرنا سابقا، تستهلك الرسوم المجسّمة وقتا ملموسا بالفعل.

ملحوظة:

لا علاقة لك كمبرمج بنوع مكوّنات الجهاز الماديّة Hardware، فهي لن تؤثّر علي الكود الذي تكتبه.. إنّ Direct3D سيتّولى عنك هذه المسئوليّة، فهو يعرف كيف يتحاور مع أنواع المكوّنات المختلفة.

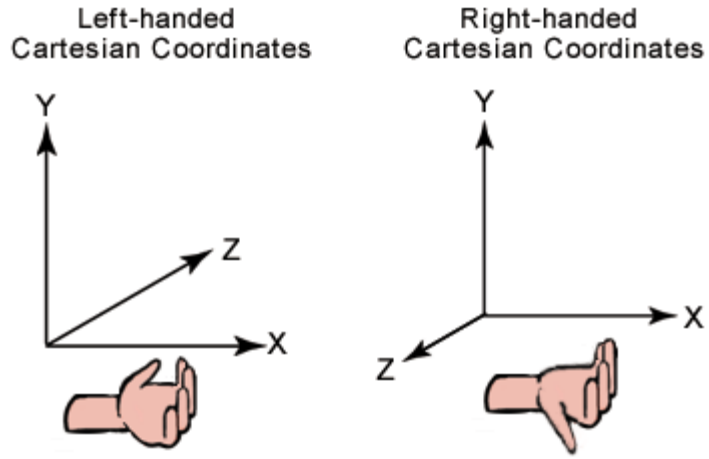
نظام الإحداثيات ثلاثي الأبعاد 3-D Coordinate Systems:

تستخدم الرسوم المجسّمة نوعين من نظم الإحداثيّات: اليساريّ Left-handed واليميني Right-handed.. وفي كلا النظامين، يشير محور السينات الموجب Positive x-axis إلى اليمين، ومحور الصادات الموجب Positive y-axis لأعلى.

ملحوظة:

انتبه لهذه الاتجاهات جيّدا، ولا ترتبك بينها وبين إحداثيّات الشاشة في النظام ثنائيّ البعد 2D.. (تذكّر أنّ محور الصادات الموجب Positive y-axis في سي شارب يشير لأسفل).

في أيّ اتجاه إذن يشير محور العينات الموجب Positive z-axis؟
كلّ ما عليك هو أن تطبّق قاعدة اليد اليسرى في النظام اليساريّ، وقاعدة اليد اليمنى في النظام اليمينيّ!



اجعل أصابعك الثلاثة الإبهام والسبّابة والوسطى متعامدة، واجعل سبّابتك تشير إلى اتجاه محور السينات الموجب، ووسطاك تشير إلى اتجاه محور

الصادات الموجب.. في هذه الحالة سيشير إبهامك إلى اتجاه محور العينات الموجب.

وطبعا في نظام الإحداثيّات اليمينيّ يجب أن تستخدم أصابع يدك اليمني، وفي نظام الإحداثيّات اليساريّ يجب أن تستخدم يدك اليسرى.

وكما هو واضح من الرسم، يشير اتجاه محور العينات الموجب إلى خارج الشاشة في نظام الإحداثيّات اليمينيّ، وإلى داخل الشاشة في نظام الإحداثيّات اليساريّ.

ويستخدم Direct3D نظام الإحداثيّات اليساريّ Left-handed coordinate system، أي أن محور العينات الموجب يشير إلى داخل الشاشة.

طبعا ستتساءل في ضجر، عن سبب كلّ هذه الثرثرة.. إنّ هذا يعود لسبب جوهريّ.. هو أنّك لن تستخدم Direct3D بمفرده، فلا بدّ أنّك ستستعين ببعض تطبيقات الرسوم ثلاثيّة الأبعاد الشهيرة، لتصميم الشكل الذي تريده وذلك للاستفادة من التسهيلات التي تقدّمها لك هذه التطبيقات.. في هذه الحالة قد تصطدم بأنّ بعض هذه التطبيقات يستخدم نظام الإحداثيّات اليمينيّ!!... فماذا ستفعل إذن يا ترى في هذه الحالة؟

طبعا يجب عليك أن تحوّل من النظام اليمينيّ إلى النظام اليساريّ.. لا تقلق.. هناك من الدوال ما سيساعدك على القيام بهذا.

الرءوس Vertices:

لا ريب أنّك تتساءل عن كيفيّة تمثيل عالم ثلاثيّ الأبعاد على شاشة مسطّحة؟!

بإمكانك تخيّل ذلك، لو علمت أنّ كلّ شيء نرسمه على الشاشة ما هو في النهاية إلا مثلث.. نعم.. كلّ الأشكال والتقاطعات يمكن تمثيلها على شاشة الكمبيوتر بمجموعة من المثلّثات.

ولكن لماذا المثلّث بالذات؟

أولا: لأنّه شكل مغلق.

ثانيا: لأنّه يتكوّن من أقلّ عدد من الرءوس.

ثالثا: وهو الأهمّ، أنّنا نضمن دائما أنّ رءوسه موجودة في مستوى واحد، فمن الممكن عند توصيل أربع نقاط أو أكثر ألا تشكّل شكلا مستويا، لأنّ نقطةً أو أكثر قد تكون موجودة في المستوى العموديّ على مستوى باقي النقاط.

وطبعا لن نتدخّل نحن في تمثيل المجسّمات على الشاشة المسطّحة، فهناك معادلات رياضيّة معقّدة تقوم بتحويل إحداثيّات الأشكال الخاصّة بنا من النظام ثلاثيّ البعد إلى النظام ثنائيّ البعد، دون أن تفقد العين إحساسها بالتجسيم.

إنّ الشاشة تجبرك دائما على أن تنظر للشكل من زاوية واحدة (فأنت لا تتوقّع أن تدور حول الشاشة لتشاهد الشكل من خلفيّته!!).. هذا معناه أنّ بإمكاننا أن نريك صورة ثنائيّة البعد تمثّل الشكل من هذه الزاوية، مع إحساس بوجود العمق.. فإذا كان الشكل يتحرّك، فمعنى هذا أنّ الزاوية التي نراه منها ستختلف.. هنا سيقوم Direct3D بإجراء المعادلات الرياضيّة اللازمة لرسم صورة جديدة

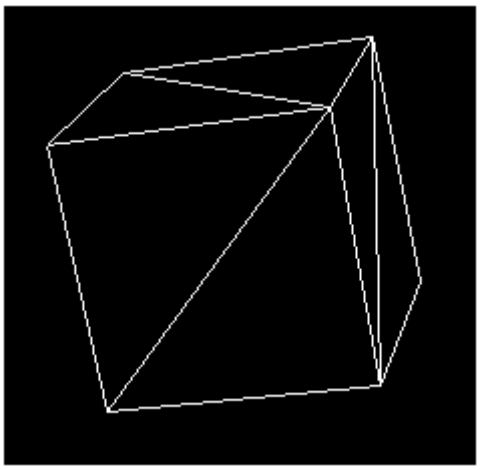
للشكل تبرزه من هذه الزاوية.. وبهذا ستشعر أنت أنّك في عـــالم مجسّـــم حقيقيّ، لأنّ كلّ التفاصيل التي تريدها موجودة!

إذن فكلّ ما نريد أن نعرفه هنا، هو كيفيّة تمثيل الأشكال المجسّمة.

طبعا أنت تعرف أنّ للمثلّث ثلاثة رءوس 3 vertices.

والرأس vertex هو أبسط وأصغر وحدة سنستخدمها للتعبير عن البيانات في الرسوم المجسّمة.. هذا الرأس ما هو إلا نقطة فـــي النظـــام ثلاثـــيّ الأبعاد، ممّا يعنـــي أنّ تعريفـــه يحتـــاج لمعرفـــة الإحـــداثيّات الثلاثـــة: (س، ص، ع).. ولكنّ الإحداثيّات ليست هي كلّ المعلومات التي تخـــصّ النقطة.. هناك أيضا لونها، درجة إضاءتها، خامتها.. إلخ.. وكلّها عوامل تؤثّر في سلوك هذه النقطة عند تحرّكها وتفاعلها مع ما حولها، كما سنرى فيما بعد.

تعال نرى مثالا لكيفيّة تمثيل أحد المجسّمات، وليكن المكعّب:



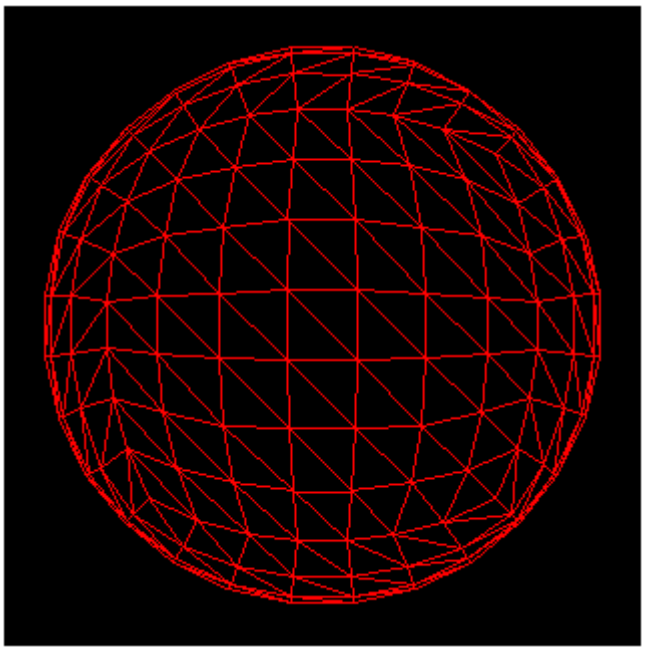
للمكعّب ستّةَ أوجه، كلّ منها عبارة عن مربّع.. وأنت تعرف أنّ قطر المربّع يقسمه لمثلّثين متماثلين.. معنى هذا أنّ المكعّب يتكوّن من ١٢ مثلّثا.

هل يعني هذا أنّنا نحتاج لـ ٣ × ١٢ = ٣٦ رأسا لتمثيل المكعّب؟

لا بالطبع!.. فكلّ مثلثين متجاورين يشتركان في أحد الأضلاع (أيّ يشتركان في رأسين!)

نحتاج إذن إلى ٨ رءوس فقط، حيث يمكن الحصول على كلّ المثلثات بتوصيل كلّ ثلاثة من هذه الرءوس معا.

أعرف أنّ الشكّ سيراودك حول قدرة المثلّثات حلى تمثيل كلّ المجسّمات.. الشكل التالي سيقنعك أنّ المثلثات قادرة حتّى على تمثيل الأسطح المحدّبة والمنحنية:



الخامات Textures:

لن يكون عالمك المجسّم قريبا من الحقيقة، ما لم تراعِ نوعيّة الخامات الداخلة في تكوين عناصره.. وإلا فكيف يمكنك مثلا رسم فقّاعة صابون تعلو في الفضاء.. هذه الفقّاعة ستعكس بعض المشاهد، وستشفّ عن بعض المشاهد في خلفيّتها.. طبعا من العبث أن تعتقد أنّك ستبرمج كلّ هذا!

كلّ ما عليك فعله، هو أن تستعين بأحد محترفي الرسوم المجسّمة ليعطيك صورة مجسّمة للفقاعة والمكان من حولها.. ثمّ باستخدام Direct3D يمكنك تحديد خامات كلّ مادة ودرجة شفافيتها ولونها وإضاءتها... إلخ...

وعندما تقوم بتحريك الفقاعة، سيتولّى Direct3D إحداث التأثيرات التي تشعرك بأنّ المشهد حقيقيّ تماما، وذلك عبر خوارزميّات معقّدة للغاية.

ولكن كيف يتمّ تمثيل الخامات؟

الخامة هي صورة ثنائيّة البعد (مخزّنة كمصفوفة نقاط Bitmap) يتمّ لصقها على مثلّث أو مجموعة من المثلثات، لإعطاء جزء من الشكل المجسّم التأثير المطلوب.

وسنتعرّف على ذلك بالتفصيل فيما بعد.

التحويلات Transformations:

قلنا من قبل إنّ المجسّمات التي نمثّلها في الإحداثيّات ثلاثيّة البعد، يجب أن يتمّ تحويلها إلى رسوم ثنائيّة البعد حتّى يمكن عرضها على الشاشة.. هذه العمليّة تسمّى "التحـــويلات المتتابعـــة" Transformation Pipeline.. وطبعا هناك رياضيّات مرعبة خلف هذه العمليّة، وإن كنّا سنستعين عليها بالعديد من الدوالّ الجاهزة، التي ستجعل الأمر بالنسبة لنا أبسط كثيرا.

تحويل الإحداثيّات World Ttransform:

افترض أنّ لدينا خمسة مكعّبات في أماكن مختلفة، ونريد عرض لقطة لها معا على الشاشة.. في البداية يمكننا تمثيل هذه المكعّبات بـــــ ٤٠ رأســـا (٨ لكلّ مكعّب).. بعد هذا سنطلب من Direct3D أن يرسمها لنا.

هذا صحيح وسيفي بالغرض.. ولكنّ هناك مشكلة: ماذا لو كانت المكعّبات الخمسة متماثلة، وكلّ ما هناك هو أنّها في مواضع مختلفة؟.. فـــي هـــذه الحالة نكون قد احتفظنا بخمسة أضعاف المعلومات المطلوبة، ففي حقيقـــة الأمر ليس لدينا سوى مكعّب واحد، تمّ رسمه في خمسة أمـــاكن مختلفـــة بزوايا مختلفة!

إذن فالأفضل أن ننشئ نسخة واحدة من المكعّب، مركزها نقطة الأصـــل (٠،٠،٠).. بعد ذلك يمكننا أن نقوم بالتحويلات اللازمة لتغيير موضع هذا المكعّب، مع إمكانيّة تكبيره وتصغيره وتدويره.. بهذه الطريقة يمكننـــا أن ننشئ أيّ عدد من النسخ من هذا المكعّب بمنتهى البساطة والسرعة.

تحويلات الكاميرا Camera Transforms:

كيف سيشعر مستخدم الكمبيوتر بأنّ ما يشاهده مجسّم، إذا لم يكن باستطاعته أن يتحرّك بحريّة بداخله؟

في هذه الحالة يجب أن تكون هناك وسيلة لتغيير زوايا المشهد.. هذه الوسيلة هي الكاميرا، التي تتيح لك اختيار الموقع الذي تنظر منه للمشهد.

تحويلات المنظور Projection Transforms:

لن يكتمل تجسيم المشهد، ما لم يكن باستطاعتنا تحديد درجة تناسب الأحجام مع المسافات، ودرجة التركيز على المشهد، ودرجة تحدّب زاوية الرؤية... إلخ.

العناصر الهندسيّة Meshes والنماذج Models:

يعبّر المصطلح Mesh عن أيّ عنصر هندسيّ، مثل الرءوس Vertices والمثلّثات Triangles... إلخ..

ويمكن جمع هذه العناصر معا لتكوين مجسّم معقّد (مثل جسم الإنسان).. وكما ذكرنا من قبل، سيكون من العبث محاولة كتابة معادلات لرسم الأشكال المعقّدة، والأسهل أن يتمّ تصميمها في تطبيقات الرسوم المجسّمة، واستيرادها.

وهناك مصطلح آخر يجب أن تعرفه، ذلك هو المصطلح Model، الذي يعبّر عن عنصر هندسيّ وخصائصه المختلفة (اللون، الإضاءة، الخامة، ... إلخ).

استخدام Direct3D9

مرحبا في سي شارب:

قديما كانت الوسيلة المثلي لاستخدام Direct3D هي استدعاء دوال API من لغة C++.. كان ذلك كذلك، إلى أصدرت ميكروسوفت DirecX7، وسمحت فيه للمبرمجين بالتعامل مع DirectX بطريقة مشابهة لتقنية COM.

والآن مع DirectX9 تطوّر الأمر كثيرا، حيث صار بإمكان مبرمج سي شارب أن يستخدم DirextX9 بطريقة مشابهة لاستخدام فئات إطار العمل Framework Classes، باستخدام الكود المدار Managed-code، وهو ذلك الذي تديره بيئة VS.Net حتّى لا يكون مرتبطا بنوع نظام التشغيل أو مكوّنات الجهاز، وحتّى يكون آمنا في تعامله مع الذاكرة المؤقّتة RAM.

وللمقارنة، فإنّ استدعاء دوال API هو كود غير منظّم Unmanaged-code، وذلك لاعتماده على إصدار نظام التشغيل.. كذلك فإنّ حجز مساحة من الذاكرة مباشرةً هو كود غير منظّم، لأنّه قد يؤدّي إلى استهلاك مساحة الذاكرة لو لم يتمّ تحرير المساحات المحجوزة بعد الانتهاء من استخدامها.

كيف تعدّ جهازك للتعامل مع DirectX9:

ادخل موقع ميكروسوفت على الإنترنت:

www.msdn.microsoft.com

وادخل قسم إنزال البرامج Downloads، وقم بتنزيل DirectX SDK.

ملحوظة١:

يجب أن يكون إطار العمل Framework معدّا أولا على جهازك (يتمّ ذلك عند إعداد VS.Net)، قبل محاولة إعداد DirectX SDK، وإلا فلن يتمّ إعداد كلّ الملفّات المطلوبة على جهازك، ولن يعمل DirectX من خلال سي شارب، حتّى لو أعددت إطار العمل بعد ذلك.. إنّ الترتيب مهمّ جدّا.

ملحوظة٢:

لكي تعمل تطبيقاتك على جهاز العميل، يجب أن يقوم برنامج الإعداد بالخطوتين التاليتين بالترتيب:

١- إعداد إطار العمل على جهاز العميل.. تسمح ميكروسوفت بتداول نسخة مجانية من إطار العمل، يجب أن يتمّ تضمينها ببرنامج الإعداد.

٢- إعداد DirectX Runtimes (ليس SDK كلّها) على جهاز العميل.

مشروع تعليميّ: مكعبان دوّران

أعرف أنّك مللت من كلّ هذا الكلام النظريّ الجافّ.. تعال إذن نبدأ رحلة المتعة.

وكبداية، سنحاول أن نتعلّم، عن طريق إنشاء مكعّبين يدوران على الشاشة، كما في الصورة التالية:

ابدأ مشروعا جديدا، وأسمه Cs_D3D9.. أضف لهذا المشروع فئة Class وأسمها .CSampleGraphicsEngine.

إضافة مرجع لـــ Direct3D:

لتتمكن من استخدام Direct3D في مشروعك، يجب أن تضيفه كمرجـــع للمشروع.

اضغط القائمة الرئيسيّة Project، واختـــر الأمـــر Add Reference.. ستظهر لك نافذة تحتوي على كلّ المراجع التي يمكن إضافتها.. لو كنـــت قد أعددت DirectX9 SDK، فستجد العناصر التالية في القائمة:

Microsoft.DirectX
Microsoft.DirectX.Direct3D
Microsoft.DirectX.Direct3DX

اختر هذه العناصر، بحيث تظهر في القائمة السفليّة، ثمّ اضغط موافق.

بعد ذلك عليك باستيراد هذه المراجع في بداية كلّ ملفّ ستستخدمها فيه:

```
using Microsoft.DirectX;
using Microsoft.DirectX.Direct3D;
using System.Math;
```

لاحظ أنّ السطر الثالث يستورد فئة الرياضيّات، وهي تنتمي لإطار العمل، وذلك لأنّنا سنحتاج للدوال الرياضيّة في مشروعنا.

تعريف المتغيرات:

الخطوة التالية هي أن نقوم بتعريف متغيّرات عامّة للفئة.. فلنبدأ بالكائنات الأساسيّة:

```
private Manager D3DRoot;
private Device D3DDev;
private D3DX D3DHelp;
```

إنّ كائن الإدارة Manager يمثّل مكتبة Direct3D، وبالتالي فهو نقطـــة انطلاقنا التي يجب أن نبدأ منها عملنا.. هذا الكائن يتحكّم فـــي الأجهـــزة

Devices، عن طريق العديد من الوسائل التي تنشئ وتغلــق الأجهــزة وتتحكّم في عملها.

ولكن ما هو الجهاز Device؟

إنّه ببساطة، كائن يمثّل أحد مكوّنات الحاسب الماديّة التي تستخدمها.. هذا معناه أنّنا نتحكّم في مكوّنات الجهاز مباشرة.

وأخيرا لدينا مكتبة المساعدة D3DX، وهي مجموعة من دوال الخــدمات التي ترافق كلّ إصدار من إصدارات DirectX، لتمنح المبرمج العديــد من التسهيلات.

بعد ذلك سنعرّف مصفوفات التحويل Transformation Matrices:

```
private Matrix matCube1;
private Matrix matCube2;
private Matrix matView;
private Matrix matProj;
```

وهي تستخدم في عمليات تحويل الإحداثيات المختلفة.

يلي ذلك تعريف الخامات Textures:

```
private Texture texCube1;
private Texture texCube2;
private Texture texMenu;
```

ثمّ الأشكال الهندسيّة Geometry:

```
private VertexBuffer vbCube;
private VertexBuffer vbMenu;
```

ثمّ الخطوط Fonts:

```
private Font fntOut;
```

وهناك مجموعة أخرى من المتغيّرات، أفضّل أن نشرحها في حينها.

وضع القيم الابتدائيّة لـــ Direct3D9:

أوّل خطوة في أيّ مشروع تنشئه بـــ Direct3D، هي أن تعــدّ أجــزاء الجهاز المختلفة لتناسب الوظيفة التي ستؤدّيها، وذلك بوضع قيم ابتدائيّــة لها، في خطواتٍ تكاد تكون ثابتة في كلّ مرّة:

١- تحديد طور العرض display mode والتنسيق Format.

٢- ضبط مخزن البعد الثالث (العمق) Depth Buffer.

٣- ضبط أيّ اختيارات إضافيّة.

٤- إنشاء جهاز Device.

٥- إعداد حالات الرسم Render states.

٦- إعداد مصفوفات التحويلات المبدئيّة.

طور العرض Display Mode والتنسيق Format:

توجد طريقتان في Direct3D لتكوين الأجهزة: فإمّــا أن تحتــلّ إحــدى النوافـــذ Windowed Mode أو تحتـــلّ الشاشـــة كاملـــة Full Screen Mode.

يهمّنا كذلك المساحة التي سنستخدمها من الشاشة.. هنا يجب أن تراعــي طور العرض Display Mode، الــذي يمثّــل مــدى دقّــة العــرض Resolution على الشاشة، فهو سيؤثّر كثيرا على الكود الذي سنكتبه.

ونظرا لأنك تصمّم مجسّماتك في طور عرض معيّن، في حــين يســتطيع المستخدم تغيير طور عرض جهازه كما يحلو له، فإنّ أمامك اختيارين:

- إمّا أن تطلب من المستخدم تغيير طور العرض قبل استخدام برنامجك.

- وإمّا أن تقوم أنت بتغيير طور العرض برمجيّا قبل بدء برنامجك، ثــمّ إعادته لما كان عليه بعد انتهاء البرنامج.. في هذه الحالة ســتقع فــي مشكلة، وهي أنّ أطوار العرض ترتبط بنوع كارت الشاشة، وقــد لا يسمح الكارت الموجود بطور العرض الذي تريده.. لذا عليك أن تتأكّد من قدرات كارت الشاشة في البداية.

تعال نكتب أوّل دالة في مشروعنا، وهي الدالة isDisplayModeOkay التي تفحص إن كان طور العرض مناسبا، وهــي تســتقبل منــك ثلاثــة معاملات:

- عرض الشاشة (المعامل iWidth).
- وارتفاعها (المعامل iHeight)
- وعمق الألوان Depth (المعامل iWidth).

وهي المقاييس التي تريد أن تتحقّق من أن جهــاز المســتخدم يمكــن أن يتعامل معها بلا مشاكل.

في البداية ستتحقّق الدالة من التنسيق Format الذي يناسب عمق الألوان. إنّ Direct3D9 يقدّم العديد من التنسيقات الجديدة لمعلومات الألــوان.. وعليك أن تقرّر إذا كنت تفضّل استخدام ١٦ خانة ثنائيّة 16 bits أم ٣٢، لحفظ معلومات اللون لكلّ نقطة Pixel.. طبعا استخدام ١٦ خانة أوفر في المساحة وأسرع نوعا في التنفيذ، ولكنّ استخدام ٣٢ خانة أكثــر كفــاءة (على حساب مساحة الذاكرة فهو يستهلك ضعف المســاحة).. وكمثــال: الشاشة التي مساحتها ١٠٢٤×٧٦٨ بتنسيق ٣٢ خانة لكلّ نقطة تحتاج إلى مساحة ذاكرة تساوي ١٠٢٤×٧٦٨×٣٢ = ٣ ميجا بايت.

ما يهمّنا الآن هو نوعان من التنسيقات:

R5G6B5	في هذا التنسيق يتمّ تخزين مكوّنات لون كلّ نقطة في ١٦ خانة: ٥ لتخزين نسبة لأحمر، و ٦ للأخضر و ٥ للأزرق.
X8R8G8B8	في هذا التنسيق يتمّ تخزين مكوّنات لون كلّ نقطة في ٣٢ خانة: ٨ خانات غير مستخدمة، و ٨ لتخزين نسبة لأحمر، و ٨ للأخضر و ٨ للأزرق.
A1R5G5B5	في هذا التنسيق يتمّ تخزين مكوّنات لون كلّ نقطة في ١٦ خانة: خانة واحدة لتخزين درجة الشفافية، و ٥ لتخزين نسبة لأحمر، و ٥ للأخضر و ٥ للأزرق.

لاحظ أنّ هناك حوالي ٤٠ تنسيقا مختلفا، ولكنّك تستطيع أن تفهمها كلّها بمجرّد النظر، فهي تحتوي على الحروف A (ويرمز لدرجة الشفافية)، و R (ويرمز للأحمر)، و G (ويرمز للأخضر) و B (ويرمز للأزرق)، و X (ويرمز للخانات غير المستخدمة).. ويلي كلّ حرف عدد الخانات التي يستخدمها.

هذا هو الكود الذي يحدّد التنسيق المناسب:

```
Format fmt;
اختيار تنسيق مناسب لعمق اللون //
if (iDepth == 16)
   fmt = Format.R5G6B5;
else if (iDepth == 32)
   fmt = Format.X8R8G8B8;
```

بعد هذا تستخدم الدالة كائن الإدارة Manger للحصول على كلّ الموصلات Adapters التي تستخدم للتحكّم في الشاشة، وذلك باستخدام

الخاصيّة Adapters، التي تعيد مجموعة Collection تحتوي على كائنات من النوع "معلومات الموصّل" AdapterInformation، يمثّل كلّ منها أحد الموصّلات المعدّة على النظام.

ثمّ تقوم الدالة بالمرور عبر كلّ كائنات معلومات الموصل واحدا تلو آخر:

```
Manager D3Dr;
foreach (AdapterInformation AdapterInfo in D3Dr.Adapters)
{
}
```

وفي داخل حلقة التكرار، سنحاول الحصول على أطوار العرض التي تسمح بالتنسيق المطلوب (عمق اللون المطلوب) في كلّ موصّل، وذلك باستخدام الوسيلة SupportedDisplayModes، والتي تستقبل نوع التنسيق كمعامل، وتعيد مجموعة Collection تحتوي على كائنات من النوع "طور العرض" DisplayMode، يمثّل كلّ منها أحد أطوار العرض الملائمة للتنسق المطلوب.

```
foreach (DisplayMode DispMode in
        AdapterInfo.SupportedDisplayModes(fmt) )
{
}
```

آخر خطوة هي اختبار مساحة العرض التي تقدّمها لنا كلّ أطوار العرض المتاحة، وذلك باستخدام الخاصيتين Width و Height لكائن طور العرض.

```
if (DispMode.Width == iWidth &&
            DispMode.Height == iHeight )
    return true;
```

كلّ ما نريده هو أن نتأكّد أنّ هناك طور عرض واحدا على الأقل في أيّ موصّل، يمتلك السمات المطلوبة.. فإذا وجدنا واحدا، فلا داعي لإكمال

التحقّق من باقي الأطوار في باقي الموصلات، وعلينا أن نغادر الدالة ونعيد القيمة True، دليلا على أنّ كلّ شيء على ما يرام.
وستجد كود هذه الدالة كاملا في المشروع المرفق بأمثلة هذا الفصل.

المخزن الخلفيّ BackBuffer:

يظهر مفهوم المخزن الخلفيّ BackBuffer، في أكثر من خاصيّة من خصائص سجلّ "المعاملات الحاليّة" PresentParameters Structure، والذي سنستخدمه في الكود بعد قليل.
أنت تعرف أنّ حركة المجسّم على الشاشة تتمّ بإعادة رسمه في مواضع متتالية، بسرعة لا تسمح للعين بملاحظة هذه العمليّة (أكثر من ٢٠ مرة في الثانية الواحدة).. ولكي يتمّ ذلك بكفاءة، يجب رسم المجسّم كاملا أولا في مخزن خفيّ عن عين المستخدم، ثمّ إظهار اللقطة على الشاشة كاملة.. بخلاف هذا، سيتمّ رسم المجسّم نقطةً نقطةً أمام المستخدم، وهي عمليّة تتّسم بشيء من البطء، ممّا سيجعل المجسّم يبدو لعين المستخدم وكأنّه يرتعش.
إذن فعليك في كلّ مرّة تستخدم فيها Direct3D، تعريف مخزن خلفيّ BackBuffer بنفس سمات الشاشة التي سترسم عليها.
دعنا نرى.

حدث إنشاء الفئة Constructor:

سننشئ تعريفين لهذا الحدث.. أحدهما يستقبل معاملا واحدا، يمثّل النموذج الذي سيتمّ الرسم عليه، بينما الآخر يستقبل المزيد من المعاملات، لتمثّل عرض وارتفاع الشاشة وعمق الألوان.

ونظرا لأنّ الصيغة الأولى لا تمدّ الفئة بأيّ معلومات حول طريقة العرض، فسنفترض أنّ مستدعيها يريد عرض الرسوم داخل نافذة Windowed-mode، بينما في الصيغة الثانية سنفترض أنّ العرض سيكون في كلّ الشاشة Fullscreen.

دعنا أولا نعرّف بعض المتغيّرات الخاصة على مستوى الفئة، لنضع فيها بعض القيم التي يستقبلها حدث إنشاء الفئة:

```
private Form rTarget; // النموذج الذي سيتمّ الرسم عليه
private bool bWindowed; // طور الشاشة (كاملة/نافذة)
// سنكتب في هذا المتغيّر معلومات حول طريقة العرض
private string sDispInfo;
// متغيّر يمثّل نجاح وضع القيم الابتدائيّة أم لا
private bool bInitOkay = false;
```

ها هي ذي الصيغة الأولى:

```
public CSampleGraphicsEngine(Form Target)
{
   try
   {
      var d3dPP = new PresentParameters( );
      d3dPP.Windowed = true;
      bWindowed = true;
      d3dPP.SwapEffect = SwapEffect.Discard;
      d3dPP.BackBufferCount = 1;
      d3dPP.BackBufferFormat = Manager.Adapters[
          0].CurrentDisplayMode.Format;
      d3dPP.BackBufferWidth = Target.ClientSize.Width;
      d3dPP.BackBufferHeight = Target.ClientSize.Height;
      sDispInfo = "[WINDOWED] " +
          Target.ClientSize.Width.ToString( ) + "x" +
          Target.ClientSize.Height.ToString( ) + " " +
          d3dPP.BackBufferFormat.ToString( );
      rTarget = Target;
      initialiseDevice((Control)Target, d3dPP);
   }
   catch
   {
      bInitOkay = false;
      throw new Exception("initialization error.");
   }
}
```

معظم ما فعلناه في هذا الإجراء، هو ملء سجلّ "المعاملات الحاليّة" PresentParameters بالمعلومات اللازمة، وسنرى فيم سنستخدمه، وذلك عند شرح الدالة initialiseDevice، التي بدورها تمّ استدعاؤها في هذا الإجراء.

والآن، تعال نرى الصيغة الأخرى لحدث الإنشاء، تلك الخاصة بـــالعرض على كل ّالشاشة:

```
public CSampleGraphicsEngine(Form Target,
        int iWidth, int iHeight, int iDepth)
{
   try
   {
      var d3dPP = new PresentParameters( );
      if (!(isDisplayModeOkay(iWidth, iHeight,
           iDepth)))
        throw new Exception("دقة العرض غير مناسبة");

      bWindowed = false;
      d3dPP.BackBufferWidth = iWidth;
      d3dPP.BackBufferHeight = iHeight;
      d3dPP.BackBufferCount = 1;
      d3dPP.SwapEffect = SwapEffect.Copy;
      d3dPP.PresentationInterval =
              PresentInterval.Immediate;
      switch (iDepth)
       {
          case 16:
            d3dPP.BackBufferFormat = Format.R5G6B5;
            sDispInfo = "[Fullscreen] " +
                    iWidth.ToString( ) + "x" +
                    iHeight.ToString( ) +
                    " 16bit R5G6B5";
            break;
```

```
            case 32:
              d3dPP.BackBufferFormat = Format.X8R8G8B8;
              sDispInfo = "[Fullscreen] " +
                    iWidth.ToString( ) + "x" +
                    iHeight.ToString( ) +
                    " 32bit X8R8G8B8";
              break;
            default:
              throw new Exception("iDepth is not valid");
        }

        rTarget = Target;
        initialiseDevice((Control)Target, d3dPP);
    }
    catch (Exception err)
    {
        bInitOkay = false;
        throw new Exception("Initialization error");
    }
}
```

إضافة إلى عدد المعاملات، هناك اختلافان جوهريان بين الصيغتين:

الأوّل يتمثّل في عدم استخدام أبعاد النمـــوذج، واســـتخدام المواصـــفات المرسلة كمعاملات بدلا من ذلك، طبعا بعد التأكّد من صلاحيتها للعرض، باستخدام الدالة isDisplayModeOkay.

والثاني يتمثّل في استخدام خاصيّة "مدة العرض" PresentationInterval الخاصّة بالسجلّ PresentParameters لإجبار Direct3D على عرض المجسّم مباشرة على الشاشة بدون أيّ تأخير، بمجرّد اكتمال رسمه.

مخزن البعد الثالث Depth Buffer:

كل ما يتمّ رسمه على الشاشة ما هو إلا صورة ثنائيّة البعد.. فإذا رسمت مجسّما على الشاشة، فإنّه قد يتقاطع مع مجسّم آخر مرسوم من قبل، ممّا يعمل على إخفاء جزء منه.. هذا يضع على عاتقك عبئا ثقيلا، وهو حتميّة رسم المجسّمات بترتيب بُعدها: الأبعد عن الكاميرا، فالأقرب، فالأقرب.... وذلك حتّى لا يحدث أيّ تأثير غير منطقيّ عندما يخفي أحد المجسّمات جزءا من مجسّم آخر.

هنا يأتي دور مخزن البعد الثالث Depth Buffer (ويسمّى أيضا Z-Buffer)، فهو يعمل على إضافة بعد ثالث تخيّليّ، يحتفظ فيه بمعلومات عن عمق كلّ مجسّم يتمّ رسمه.. وعند رسم أيّ مجسّم جديد، يقوم Direct3D بالتحقّق من البعد الثالث لكلّ نقطة.. فإذا كانت النقطة المرسومة سابقا أقرب للكاميرا من النقطة المراد رسمها، تظلّ النقطة القديمة كما هي، ولا يتمّ رسم النقطة الجديدة على الشاشة.. ولو كانت النقطة الجديدة أقرب للكاميرا من النقطة القديمة، يتمّ رسمها بدلا منها، بحيث يبدو أنّ المجسّم الجديد قد أخفى كلّ أو بعض المجسّم القديم.

يستلزم هذا عددا كبيرا من عمليات المقارنة، لهذا يستخدم Direct3D بعض الخوارزميّات لتوقّع الأوجه الخلفيّة للمجسّمات (التي لا تواجه الكاميرا) ويتجاهل رسمها، لأنّها في كلّ الأحوال لن تظهر أمام الرائي من هذه الزاوية.

معنى هذا أنّ استخدام مخزن البعد الثالث يسمح لنا برسم المجسّمات دون اعتبار ترتيبها، ففي كلّ الأحوال سيبدو المشهد النهائيّ تماما كما أردناه.. والآن كلّ ما علينا هو استخدام الجملة التالية لتفعيل هذه الإمكانيّة:

```
D3DDev.RenderState.ZBufferEnable = true;
```

نوع الجهاز Device Type:

قلنا إنّ الكائن Manager هو نقطة انطلاقنا، حيث سنستخدمه لإنشاء الأجهزة التي سنتعامل معها.

وهناك أربعة أنواع من الأجهزة، تحدّد الطريقة التي سيتمّ بها إجراء العمليّات على نقاط الصورة:

Software	المشغّل الدقيق الخاصّ بالكمبيوتر هو المسئول عن إجراء عمليّات التحويل والإضاءة.
Hardware	كارت الشاشة هو المسئول عن إجراء عمليّات التحويل والإضاءة.
Mixed	مزيج من النوعين السابقين: المشغّل الدقيق وكارت الشاشة سيشتركان معا في معالجة الرسوم.
Pure	كارت الشاشة هو المسئول عن معالجة الرسوم لأقصى درجة ممكنة.

ويمكن ترتيب الأنواع السابقة من حيث الأفضليّة (السرعة وكفاءة الأداء) كالتالي: Software ــ Mixed ــ Pure ــ Hardware.

ويجب أن تنتبه إلى أنّ بعض هذه الأنواع قد لا يكون متاحا على جهاز العميل.. لهذا يجب أن تستخدم الوسيلة "اقرأ قدرات الجهاز" GetDeviceCaps الخاصّة بكائن الإدارة Manger.. هذه الوسيلة تستقبل معاملان: رقم الموصّل Adapter، ونوع الجهاز.. وتعيد هذه الوسيلة كائنا من النوع Caps:

```
D3DCaps = D3DRoot.GetDeviceCaps(0,
      DeviceType.Hardware;
```

بعد ذلك استخدمنا هذا الكائن لنتأكّد من أنّ كارت الشاشة يستطيع معالجة التحويلات والإضاءة، فإذا كان الشرط متحقّقا، يمكن إسناد معالجة التحويلات والإضاءة لكارت الشاشة، وإلا فسنسند هذه المهمّة للمشغّل الدقيق:

```
if (D3DCaps.DeviceCaps.
      SupportsHardwareTransformAndLight)
   DevCreate = CreateFlags.HardwareVertexProcessing |
         CreateFlags.MultiThreaded;
else
   DevCreate = CreateFlags.SoftwareVertexProcessing |
         CreateFlags.MultiThreaded;
```

لاحظ كذلك أنّنا سنرسل المتغيّر DevCreate كمعامل لحدث إنشاء الجهاز، وبذلك تؤثّر القيم التي وضعناها به على عمل الجهاز:

```
D3DDev = new Device(0, DeviceType.Hardware,
         Target, DevCreate, win);
```

دالة وضع القيم الابتدائيّة:

سنعرّف الآن المزيد من المتغيّرات الخاصة على مستوى الفئة:

```
// حجم الخط الذي سيتم رسم النصوص به
private int iFontSize;
private int lastFrameUpdate;
private string sDevInfo; // اسم كارت الشاشة
```

نصل الآن إلى كود الدالة initialiseDevice.. لا تجعل طوله يفزعك، فقد تعرّفنا بالفعل على كلّ أجزائه.

تكوين المجسّمات في الذاكرة

كيفية تخزين المجسّمات:

يمكنك حفظ رءوس المجسّمات vertices بإحدى طريقتين:

١- مخازن الرءوس Vertex Buffers:

وهي مصفوفة يوفرّها لك Diurect3D لمعالجة الرسوم بكفاءة عالية، حيث يتحكّم في كيفيّة تخزين الرسوم (إمّا في ذاكرة الكمبيوتر أو ذاكرة كارت الشاشة).. وحتّى تحصل على أفضل أداء من هــذه الطريقــة، استخدم مخزن الرءوس Vertex Buffer لتخزين عدد من الــرءوس ما بينَ ١٠٠ و ٥٠٠٠.. فإذا زاد العدد عن ذلك، فاقسمه على أكثر من مخزن.. وإذا قلّ العدد عن ذلك، فضع رءوس أكثر من مجسّم معا في نفس المخزن.

٢- باستخدام المصفوفات التقليديّة:

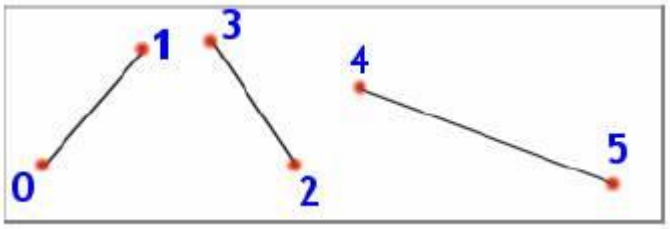
ولا ننصحك بهذه الطريقة، إلا إذا كنت تحتاج لقراءة وتغيير بيانــات الرسوم مرارا وتكرارا، وهو ما يتّسم بنوع من البطء فــي الطريقــة الأولى.

كيفيّة رسم المجسّمات:

هناك ٦ طرق للتعامل مع الرءوس التي احتفظنا بها:

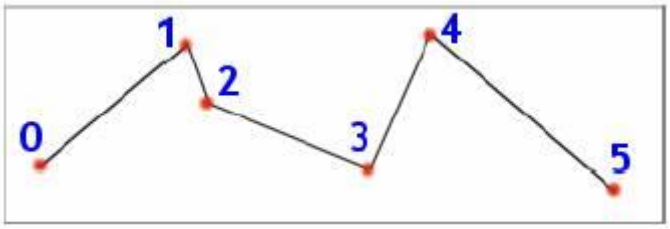
١ - قائمة الخطوط Line List:

يتمّ رسم خط مستقيم بين كلّ زوجين متتاليين من الـــرءوس (دون أن تشترك رأس بين أكثر من خطّين).



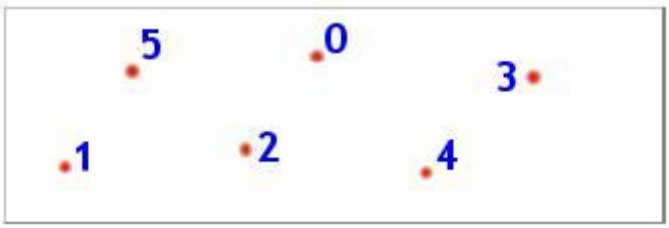
٢ - شريط الخطوط Line Strip:

يتمّ رسم خطّ مستقيم بين الرأس الحاليّة والرأس السابقة، بحيـــث يـــتمّ توصيل كلّ الرءوس معا بخطّ واحد متّصل، بدون المرور بأيّ نقطـــة أكثر من مرّة.



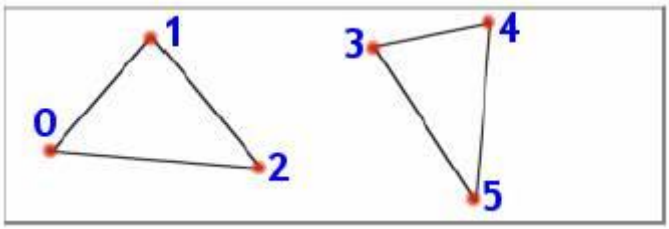
٣ - قائمة النقاط Point List:

يتمّ رسم كلّ الرءوس كنقاط مستقلّة.. ويمكنك استغلال هذه الإمكانيّة لرسم الأشكال الشبحيّة، كالدخان والماء والنار... إلخ.



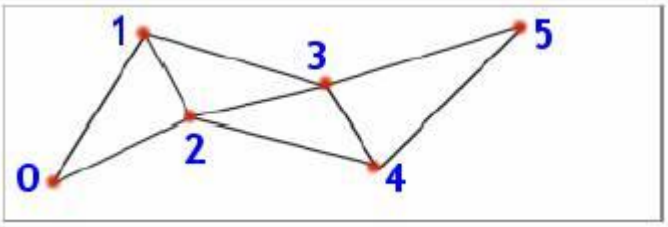
٤ - قائمة المثلثات Triangle List:

يتمّ توصيل كلّ ثلاث رءوس لتشكّل مثلّثا، دون أن تستخدم أيّ رأس في أكثر من مثلث.



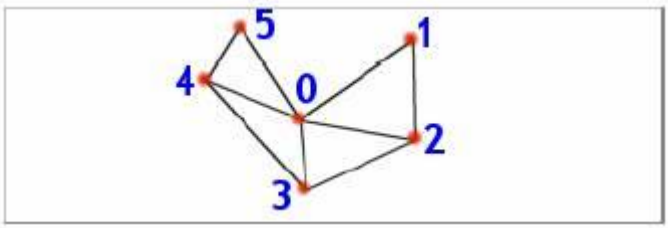
٥ - شريط المثلثات Triangle Strip:

يتم رسم مثلثات بين كلّ ثلاثة رءوس، مع اشتراك كلّ مثلثين متجاورين في أحد الأضلاع.



٦ - مروحة المثلات Triangle Fan:

يتمّ رسم كلّ المثلثات الممكنة، بحيث يكون الرأس الأوّل (رقم صفر) مشتركا بينها جميعا.



تحميل المجسمات:

سننشئ الآن الدالة loadGeometry، لتكون مسئولة عن تحميل الأشكال، وهي تنقسم إلى قسمين:

- الأوّل يقوم بإنشاء قائمة بسيطة ثنائيّة البعد (مجرّد مستطيل سنعرض فيه بعض المعلومات عن وظائف بعض الأزرار).
- والثاني يقوم بإنشاء المكعّب.

في كلتا الحالتين سنبدأ بحجز مخزن الرءوس VertexBuffer:

```
vbCube = new VertexBuffer(typeof(
            CustomVertex.PositionTextured), 36,
            D3DDev, 0,
            CustomVertex.PositionTextured.Format,
            Pool.Managed);
```

حيث يمثّل المعامل الأوّل نوع الرءوس، والثاني عددها، والثالث الجهاز الذي سيرسمها، والخامس نوع التنسيق (وسنستخدم هنا إحداثيات الموضع والخامات)، والأخير يعطي السلطة لـ Direct3D لإدارة الذاكرة المؤقّتة، سواء الخاصة بالحاسوب أو بكارت الشاشة.

بعد هذا سنغلق مخزن الرءوس أمام استخدامه من قبل أيّ تطبيق آخر، وسنحصل على مؤشّر Pointer يشير إلى بياناته، وذلك باستخدام الوسيلة vbCube.Lock(0, 0)، التي تعيد مصفوفة تقليديّة.. ونظرا لأنّنا نتعامل مع Vertex Buffer، فسنحوّل القيمة العائدة إلى مخزن رءوس من النوع الذي نستخدمه CustomVertex.PositionTextured:

```
var vCube = (CustomVertex.PositionTextured[ ])
            vbCube.Lock(0, 0);
```

يمكننا الآن التعامل مع الخانات من ٠ إلى ٣٥ (٣٦ خانة كما عرّفناها) في المصفوفة (المخزن) vCube.. علينا أن نملأ هذه المصفوفة بالإحداثيّات.. أحضر ورقة وقلما، وصمّم المكعّب كما تريد، واحسب إحداثيات رءوسه.

لاحظ أنّنا في هذا المثال استخدمنا طريقة قائمة المثلثات Triangle-List للتوصيل بين رءوس المكعّب.

آخر خطوة في هذه العمليّة، هي تحرير المخزن الذي أغلقناه أمام التطبيقات الأخرى.. هذه الخطوة ضروريّة، لأنّ Direct3D لو حاول أن يرسم المكعّب على الشاشة في حين أنّ مخزن الرءوس مغلق، فسيحدث خطأ ولن ينفّذ العمليّة:

```
vbCube.Unlock( );
```

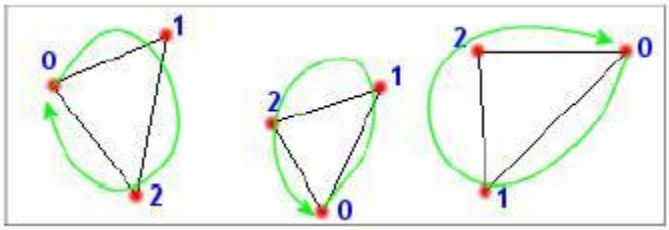
وستجد كود الدالة loadGeometry كاملا في المشروع Cs_D3D9.

ترتيب الرءوس في المثلث:

لكي تحصل على الشكل الذي تريده، يجب عليك مراعاة الترتيب الذي تخزّن به الرءوس.. هذا الترتيب سيؤثّر على تحديد الأوجه الخلفيّة للمجسّم (تلك التي لا تظهر للرائي من الزاوية الحاليّة).. لقد ذكرنا من قبل أن Direct3D يتخلّص من الأوجه الخلفيّة، وهي عمليّة تعرف باسم "غربلة الأوجه الخلفيّة" Back Face Culling، وذلك ليوفّر وقت رسمها. وبإمكانك استخدام الخاصيّة Device.RenderState.CullMode ــ كما سنرى لاحقا ــ لتخبر Direct3D بكيفيّة تحديد الأوجه الخلفيّة، وذلك عن طريق واحدة من قيم المرقّم Cull التالية:

CounterClockwise	القيمة الافتراضيّة.. إزالة الأوجه الخلفيّة الموجودة في عكس اتجاه عقارب الساعة.
Clockwise	إزالة الأوجه الخلفيّة الموجودة في اتجاه عقارب الساعة.
No culling	لن يتمّ محو الأوجه الخلفيّة.. هذا مفيد عند التعامل مع الأسطح شبه الشفافة، حيث تتمّ رؤية الخلفيات في هذه الحالة.

وفي الصورة التالية، إذا استخدمنا CounterClockwise، فلن يتمّ رسم المثلّث الأوسط، لأنّ ترتيب رسم نقاطه في عكس اتجاه عقارب الساعة.



وعليك بالاحتراس عند تخزين رءوس المجسّمات، فعلى حسب الترتيب الذي ستضعها به في الذاكرة سيتمّ رسم أو محو المثلثات.

أعرف أنّك ما زلت متعجبا من احتياج Direc3D لهذه التقنية!!

حسنا.. إنّ غربلة الأوجه الخلفيّة Culling توفّر ٤٠% من الوقت اللازم لرسم المجسّم بدون استخدام هذه التقنية.

لاحظ أنّ هناك وجها واحدا فقط (من وجوه المكعّب الستّة) هو الذي يسهل تحديد كونه وجها خلفيّا، لأنّه يقع في المستوى (س ــ ص)، لهذا عليك أن تخبر Direct3D بكيفية إزالة ثلاثة أوجه من الوجوه الخمسة المتبقية.

والآن تعال نواصل مشروعنا.

الخامات Textures

تنسيقات الخامات:

ذكرنا من قبل أنّ الخامة هي صورة نقطيّة Bitmap يتمّ لصقها على جزء من المجسّم.

ولكي تحصل على أفضل أداء للخامة، اجعل مقاييسها ثنائيّة (أعداد مرفوعة للأس ٢)، مثل ١٢٨×١٢٨، ٢٥٦×٢٥٦، ٥١٢×١٠٢٤.... إلخ.. هذا سيسهّل على معظم أنواع كروت الشاشة تسريع التعامل مع هذه الخامات.

وليس عليك أن تحفظ الخامات بهذه الأبعاد بنفسك، فـ Direct3D سيقوم بتحويلها تلقائيّا (ما لم تطلب أنت منه العكس).

لكن يجب أن تتأكّد أولا من حجم الخامة الذي يستطيع كارت الشاشة التعامل معه.. معظم الكروت تتعامل مع خامات مساحتها أصغر من أو تساوي ١٠٢٤×١٠٢٤، وهو كاف بالفعل في معظم الحالات.

ويتمّ حفظ الخامة على الكمبيوتر كصورة بالامتدادات المعروفة (BMP, TGA, GIF, JPG... إلخ).

وهناك امتداد آخر هو DDS (سطح الرسم المباشر Direct Draw Surface)، وهو يحفظ الصورة بنفس التنسيق الذي يتمّ تمثيلها به في الذاكرة بواسطة كارت الشاشة.. هذا يجعل تحميل الصورة من الملفّ أسرع، كما يمنح مصمّم الصورة القدرة على تحديد عدد الخانات التي تمثّل درجة الشفافية كما يريد، كأن يجعلها خانتين 2-Bits بدلا من ٨، دون أن يضيّع Direct3D وقتا طويلا في التحويلات.

فإذا كنت ترغب في إنشاء ملفّات بامتداد DDS، فيمكنك استخدام الأداة DXTex الموجودة مع DirectX9 SDK.. من سطح المكتب اضغط:
Start Menu/Programs/Microsoft DirectX 9.0 SDK /DirectX Utilities ،DirectX Texture Tool.

وعند تحميل الخامة في ذاكرة كارت الشاشة، يتمّ حفظها بتنسيق معيّن.. ويمنحنا Direct3D ٤٠ تنسيقا مختلفا، أسماؤها على شاكلة X8R8G8B8، وقد تعرّفنا من قبل على ما تعنيه الحروف R و G و B، وما تعنيه الأرقام التالية لكلّ منها.

لاحظ أن هناك مساحة مخصّصة من كارت الشاشة لتحميل الخامات (لو كان كارت الشاشة ٣٢ ميجا، فسيوفر مساحة من ٢٠ إلى ٢٥ ميجا للخامات)، وعليك ألا تتجاوز هذه المساحة، وذلك بتحميل الخامات الضروريّة في الوقت المناسب، والتخلّص من الخامات التي انتهى دورها.

إحداثيّات الخامات Texture Coordinate:

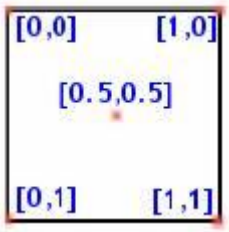
لكلّ رأس Vertex موقع ولون وإضاءة وإحداثيّات خامة.

وتكون الخامة ثنائيّة البعد 2D كما اتفقنا.. لهذا سيكون لكلّ رأس إحداثيان ثنائيا البعد، سنرمز لهما بالرمزين U و V.

هذا هو السبب الذي جعلنا نرسل خمس إحداثيّات أثناء ملء مصفوفة الرءوس في دالة تحميل الرسوم: ٣ إحداثيّات الموضع، وإحداثيّين الخامة.

مع ملاحظ أنّ وحدة قياس الخامة تختلف عن وحدة قياس الموضع (يقاس بالنقطة Pixel)، حيث تقاس الخامات بأرقام عشريّة تنحصر بين ٠,٠ و ١,٠، وأيّ أعداد خارج هذا النطاق لا يتمّ تمثيلها.

والصورة والجدول التاليين يوضّحان لك إحداثيّات بعض المواضع في الخامة:



الموضع	V	U
أعلى اليسار	0.0	0.0
أعلى اليمين	0.0	1.0
أسفل اليسار	1.0	0.0
أسفل اليمين	1.0	1.0
منتصف الوسط.	0.5	0.5

ولعدم استخدام الوحدات التقليديّة حكمة.. افترض أنّ المثلّث الذي سيتمّ لصق الخامة عليه يتحرّك بعيدا عن الكاميرا.. معنى هذا أن حجمه الظاهريّ سيقلّ.. في هذه الحالة لا جدوى من تحميل خامة ذات كفاءة عالية على مثلّث ضئيل المساحة.. لهذا يُنتج Drect3D نسخا مختلفة من الخامة بدقّة مختلفة (٢٥٦×٢٥٦، ١٢٨×١٢٨، ٦٤×٦٤.... ٢×٢، ١×١)، ويختار مستوى الدقّة تبعا لحجم المثلّث وبعده عن الكاميرا.. لهذا فإنّ التعامل بإحداثيّات نسبيّة للخامة أفضل من التعامل مع إحداثيّات الشاشة، حيث سيقوم Direct3D بإجراء التحويلات المناسبة.

كما يسمح لك هذا أيضا بتغيير مساحة الخامة بدون تغيير الكود.. افترض أنّ لديك خامة ١٠٢٤×١٠٢٤ ولم يستطع كارت الشاشة التعامل معها.. في هذه الحالة يمكنك تحميلها بأبعاد أصغر، دون أن تخشى من حدوث مشكلة في تنفيذ الكود، فالإحداثيّات التي تستخدمها نسبيّة وليست حقيقيّة.
شيء آخر يجب أن تعرفه، هو أنّه من الممكن أن يكون للمثلّث الواحد أكثر من خامة (قد يصل الأمر إلى ٨ خامات للمثلّث)، وطبعا يمكن ألا تستخدم الخامات أساسا.

تحميل الخامات:

نصل الآن إلى كود الدالة loadTextures، وهو بسيط للغاية.
لاحظ أنّنا سنستخدم ثلاث خامات، اثنتين للمكعّبين الذين سنرسمهما، وواحدة للقائمة.. ستجد كود هذه الدالة كاملا في المشروع Cs_D3D9، وهو واضح ومباشر.. ربما ما يحتاج لقليل من الإيضاح هو الوسيلة Manager.CheckDeviceFormat، التي نتأكّد بها إذا ما كان تنسيق الخامة صالحا للاستخدام مع كارت الشاشة أم لا.. وتأخذ هذه الدالة عدّة معاملات، هي بالترتيب:

- رقم الموصّل Adaper: وهو هنا أوّل موصّل (رقم صفر).
- نوع الجهاز: ونحن نستخدم في هذا المشروع النوع Hardware.
- تنسيق الموصّل: ونحن نستقبله في الدالة loadTextures كمعامل.. تذكّر أنّنا وضعناه في إحدى خصائص السجلّ PresentParameters من قبل.

- خيارات الاستخدام: سنستخدم القيمة صفر، حيث لا يوجد أيّ استخدام خاصّ.
- نوع المصدر: وهو هنا ResourceType.Textures.
- نوع التنسيق: وهو التنسيق الذي نريد التأكّد من صلاحيته للاستخدام.. وسنجرّب أولا استخدام التنسيق X8R8G8B8، فإذا لم يكن متاحا فسنجرّب التنسيق R5G6B5.

وتستخدم الدالة loadTextures أيضا الوسيلة TextureLoader.FromFile.. وكما هو واضح من اسمها فهي تقوم بتحميل الخامة من ملف.. لاحظ أن آخر معامل من معاملات هذه الوسيلة يستقبل أحد الألوان.. إنّ هذا المعامل يقوم بنفس دور خاصيّة مفتاح الشفافية TransparencyKey في النموذج، حيث لا يتمّ رسم النقاط التي تحمل لون مفتاح الشفافية، وذلك لجعل منطقة من الرسم شفافة.

مصفوفات التحويل Transformation Matrices:

سنرى الآن كيف نستخدم مصفوفات التحويل لعمل نسختين مختلفتين من المكعّب الذي أنشأناه.. ولكن يجب عليك أولا أن تتذكّر هذه القاعدة الرياضيّة الهامّة للتعامل مع المصفوفات، وهي أنّ الترتيب مهمّ عند ضرب المصفوفات، بمعنى أنّ A×B لا يساوي B×A.. تذكّر هذا جيّدا عند قيامك بعمليّات تحويل متتابعة على أحد المجسّمات، فاختلاف الترتيب قد يؤدّي إلى اختلاف النتيجة.

وهذه هي أهمّ العمليّات التي يمكن إجراؤها على المجسّمات:

١- تغيير الحجم Scale.

٢- التدوير Rotation حول أيّ محور (س، ص، ع).

٣- تغيير الموضع Translation.

ولكن... هل علينا أن نقوم بإجراء كلّ العمليّات الرياضيّة على المصفوفات بأنفسنا؟

لحسن الحظّ لا.. إنّ لدينا مكتبة رياضيّات Math Library يمدّنا بها Direct3DX، وهي ستقوم بمعظم العمل المعقد بنفسها.. إنّ سجلّ المصفوفة Matrix Structure يمدّنا بأكثر من ٢٥ دالة تسهّل علينا التعامل مع مصفوفات التحويل.. والجدول التالي يوضّح لك سبعة من أهمّ هذه الدوال:

Multiply	لضرب مصفوفتين معا (لتكوين مصفوفة جديدة تـــدمج عملتي التحويل معا).
Identity	مصفوفة الوحدة (وهي مصفوفة كلّ عناصرها أصفار ما عدا قطرها، فكلّه آحاد).. عند ضرب هذه المصفوفة في أيّ مصفوفة أخرى لا يحدث أيّ تأثير (مثلما تضـــرب الواحد في أيّ عدد).
RotateX	لتدوير المجسّم حول محور السينات.
RotateY	لتدوير المجسّم حول محور الصادات.
RotateZ	لتدوير المجسّم حول محور العينات.
Scaling	تغيّر حجم المجسّم بالنسبة التي ترسلها إليها كمعامل.
Translation	لتغيير موضع المجسّم.

انظر كيف نغيّر حجم المكعّب بالنسبة لجميع المحاور:

```
matCube1 = Matrix.Multiply(matCube1,
                Matrix.Scaling(3, 3, 3))
```

حيث matCube1 هي مصفوفة التحويل التي عرّفناها على مستوى الفئة من قبل:

```
private Matrix matCube1;
```

وكل ما فعلناه أننا ضربناها في مصفوفة تكبير للحجم بنسبة ٣ في جميـــع المحاور.

ما يجب أن تعرفه هنا، هو أنّ أيّ تغيير في المصفوفة يؤثّر علـــى كـــلّ المجسّمات التي سيتمّ رسمها.

إذن كيف يمكن إحداث تأثيرات مختلفة على المجسّمات؟

لفعل هذا، كوّن مصفوفة التحويل الأولى وارسم المجسّم الأوّل.. هذا يعنى أنّه سيتأثّر بهذه التحويلات.. بعد ذلك كوّن مصفوفة التحويل الثانية وارسم المجسّم الثاني.. سيتأثّر المجسّم الثاني فقط بـــالتحويلات الجديـــدة، بينمـــا المجسّم الأوّل ثابت كما هو.. ويمكن أن ترسم أكثر من مجسّم معا بـــنفس مصفوفة التحويل.. بعد أن تنتهي ستحصل على اللقطة التي تريدها، وبها كلّ مجسّم في الوضع المطلوب.

والآن تعال نكمل مشروعنا، لنرى كيف سنرسم المكعّبين ونحرّكهما.

تحديث الإطار Frame Update

قلنا إنّ الحركة تنتج عن عرض العديد من الصور المتتابعة بسرعة مناسبة.. وتسمّى كلّ صورة من هذه الصور إطارا Frame.. تعال الآن نرَ كيف نكوّن هذه الإطارات، بحيث يبدو لمن يراها وكأنّ المكعبين يدوران:

رسم الإطار:

تعال نرَ كود الإجراء oneFrameUpdate، وهو مسئول عن تكوين الإطار التالي للإطار المعروض حاليّا على الشاشة.. ولكننا سنعرّف بعض المتغيّرات على مستوى الفئة أولا:

```
private Single cube1Angle;
private Int32 lastFrameUpdate;
private float cube1Angle;
private float cube2Angle;
private const float cube1Speed = 50.0F;
private const float cube2Speed = 75.0F;
private const float cube1Size = 2.0F;
private const float cube2Size = 4.0F;
private bool bInitOkay = false;
private Int32 iLastFPSCheck;
private const int iFPSProfileSpeed = 200; //ms
private int iCurrCnt;
private int iFrameRate;
private string sDevInfo;
private string sDispInfo;
```

وستجد كـود الإجـراء oneFrameUpdate كـاملا فـي المشـروع Cs_D3D9.. الفكرة في هذا الكود، هي تغيير زاويـة دوران المكعـب بمقدار ثابت في كلّ ثانية (٥٠ درجة في الثانية مـثلا).. هـذه الطريقـة تضمن أداء ثابتا، فمهما زاد أو قلّ عدد الإطارات التي يتمّ رسـمها فـي الثانية، فإنّ دورانا مقداره ٥٠ درجة فقط هو الذي يحدث للمجسّم.. أمّا لو كنّا نعتمد على استخدام سرعة معيّنة للدوران (عدد الإطارات في الثانية)، وعلى أساس ذلك يتحدّد مقدار التغيير في زاوية الدوران، فقد يؤدّي هـذا إلى عدم استقرار الحركة، فقد تكون سريعة جدّا على الحواسب السريعة، وقد تكون بطيئة جدّا ومتقطّعة إذا كانت هناك عملية أخرى تعطّل الحاسب أو كانت سرعته بطيئة.

ولحساب الوقت استخدمنا فئة البيئة Environment Class، وهي تابعة لفضاء الاسم System في إطار العمل، ومهمّتها إمدادنا بمعلومات عـن نظام التشغيل.

ونحن هنا نستخدم الخاصيّة "عدد الأجـزاء مـن الألـف مـن الثانيـة" Environment.TickCount، وهي تحسب الوقت المنقضي منذ تشغيل الويندوز إلى اللحظة الحاليّة، بدقّة أجزاء من الألف من الثانية.

لاحظ أنّنا نحـتفظ بـآخر وقـت حـدّثنا فيـه الإطـار فـي المتغيّـر lastFrameUpdate، وبطرحه من الوقت الحالي يمكن معرفـة الوقـت المنقضي.. وطبعا نقسم على ١٠٠٠ حتّى نقرّب الناتج لأقرب ثانية.

انظر إلى كيفيّة حساب زاوية الدوران للمكعّب الأوّل في الإطار الحاليّ:

```
cube1Angle += (float) (cube1Speed *
  (Environment.TickCount - lastFrameUpdate) / 1000.0);
```

هذه المعادلة تقول ببساطة، إنّ الزاوية الجديدة تزيد عن الزاوية القديمة (لاحظ علامة +=)، بمقدار الوقت الذي انقضى (بالثانية) منذ رسم الإطار السابق، مضروبا في سرعة الدوران (معبّرا عنها بثابت، ليسهل عليك تغييرها في أيّ لحظة.. هذا الثابت هنا يساوي ٥٠).

هذا معناه أنّ دوران المجسّم سيكون مستقّرا وبسرعة ثابتة، فلو حدث شيء عطّل الحاسب لثلاث ثوانٍ مثلا، فستزيد زاوية الدوران بمقدار ٣×٥٠ = ١٥٠ مرّة واحدة، بحيث يبدو وكأنّ المجسّم كان يواصل حركته بطريقة طبيعيّة.. طبعا ليس من الشائع أن ينشغل الحاسب عن تنفيذ برنامجنا لثلاث ثوان دفعة واحدة، والأحرى أن نتوقّع أن يتعطّل لأجزاء من مئات الأجزاء من الثانية على أسوأ تقدير!

وبعد أن نحسب زاويتي دوران المكعّبين، يجب أن نخزّن الوقت الحاليّ، لأنّ هذه هي اللحظة التي حدّثنا فيها الإطار الحاليّ، والتي ستدخل في حساب الإطار القادم:

```
lastFrameUpdate = Environment.TickCount( );
```

الآن علينا أن نجرى التحويلات على المكعّبين.. تعال نرى ما سيحدث للمكعّب الأوّل:

١- تحجيم هذا المكعّب لتصبح أبعاده ٣×٣×٣.. تذكّر أنّ أبعاد المكعّب الأصليّ الذي أنشأناه كان ١×١×١.

٢- تدوير هذا المكعّب حول محور السينات ثمّ محور الصادات بمقدار زاوية الدوران التي حسبناها من قبل.. لاحظ أنّنا قمنا بتحويل الزاوية إلى القياس الدائريّ وذلك بالضرب في النسبة التقريبيّة ط والقسمة على ١٨٠:

```
cube1Angle * (Math.PI / 180)
```

لاحظ كذلك أنّ زاوية الدوران تبدأ من صفر حتّى ٣٦٠ (دورة كاملة) ثمّ تعود إلى الصفر مرّة أخرى.. ولن نحتاج للتدخّل في هذا، فهو من أبسط قواعد الرياضيّات و Direct3D يفهمه!

ولن ننقل هذا المكعّب من موضعه في مركز الشاشة، حيث سنكتفي بتدويره حول مركزه.

أما المكعّب الثاني، فهذا ما سيحدث له:

١- سنغيّر أبعاد هذا المكعّب.. ونحن هنا لن نتركه مكعّبا، بل سنحوّله لمتوازي مستطيلات، طوله وعرضه وارتفاعه مختلفة عن بعضها.

٢- بعد هذا سننقل هذا المجسّم بعيدا عن مركز المكعّب الأصليّ بمقدار (-٨، -٤، ٠).

٣- ثمّ سندير هذا المجسّم حول محور العينات Z-axis بمقدار زاوية الدوران التي حسبناها له من قبل.. سيجعل هذا المجسّم يدور حول نقطة الأصل.. لماذا؟.. لأنّنا نقلناه من موضعه أولا.. إنّ الدوران سيتمّ حول المحور عين، وليس حور مركز المجسّم نفسه.. ولو أردت أن تجعل المجسّم يدور حول محوره هو، فقم بعمليّة الدوران أولا قبل نقل المجسّم من موضعه.. فهمت الآن فائدة الترتيب؟

ويمكنك الحصول على أنماط حركة مختلفة، بالتلاعب بمعاملات الدوران والانتقال للمكعّبين.. جرّب.

اعتبارات يجب مراعاتها عند الرسم:

قبل أن نقوم بالخطوة الأخيرة، يجب أن تضع في اعتبارك أنّ أكثر من ٩٩% من وقت تنفيذ برامج الرسوم يضيع في رسم المجسّمات، وليس في تحميل الصور والخامات ووضع القيم الابتدائيّة.. لهذا فإنّ خبرتك في البرمجة ستتمرّكز حول كيفيّة تحسين أداء تطبيقك، بحيث لا تضيّع وقتا أطول من اللازم في رسم المجسّمات.. وفيما يلي بعض المطبّات التي يجب أن تتلافاها:

١ - الرسم زيادة عن الحدّ Overdraw:

لو اختبرت أيّ نقطة في الشاشة، فستجد أنّها ترسم مرتين أو ثلاثة للإطار الواحد، نتيجة لأنّ بعض المجسّمات يحجب البعض الآخر، فيتمّ رسم أجزاء من المجسّم الأول على الثاني.. وللأسف، لا يستطيع مخزن البعد الثالث Depth Buffer حلّ هذه المشكلة، فهو لا يعرف إذا كان هناك مجسّم سيتمّ رسمه مستقبليّا فوق المجسّم الذي يقوم حاليّا برسمه أم لا!.. معنى هذا أنّ هذه مسئوليّتك بالدرجة الأولى، وإن كانت هناك بعض الحلول من Direct3D لا مجال هنا لذكرها.

٢ - **المجسّمات شبه الشفّافة**:

لكي يعطي Direct3D تأثير الشفافيّة، فإنّه يطبّق معادلات الخلط Blending Equations على المجسّمات الموجودة حاليّا على الشاشة، حتّى تظهر عبر السطح شبه الشفّاف.. فماذا لو تأخّرت أنت في رسم أحد المجسّمات التي يجب أن تكون في خلفيّة السطح شبه الشفّاف؟.. طبعا سيظهر هذا المجسّم المتأخّر عاديّا تماما، دون أن يتأثّر بشفافية السطح الذي أمامه!.. هنا أيضا ترتيب الرسم مهمّ.

٣ - **الرسم بالقوّة الشرسة Brute-force Rendering**:

يحدث مع حركة المجسّمات أن تخرج من نطاق الرؤية.. فلماذا إذن نضيّع وقتا في رسمها لتشغل حيّزا من الذاكرة وتضيّع الوقت دون أن تؤثّر في شيء؟.. هناك خوارزميّات لحلّ هذه المشكلة، مثل Frustum culling، occlusion culling، portal culling، BSP Tree's، وغيرها، ولكنّ المقام لا يتّسع هنا لشرحها!

رسم المكعّبين على الشاشة:

آخر خطوة أمامنا الآن هي أن نرسم المكعّبين والقائمة على الشاشة (رسم الإطار).. سيتمّ هذا عن طريق الإجراء oneFrameRender.

لاحظ أنّ هذا الإجراء هو المكان الذي يجب أن تكتب فيه حلولا للمشاكل التي ناقشناها سابقا، مثل التخلّص من المجسّمات الموجودة خارج نطاق الرؤية.

ها هو ذا الهيكل العامّ للكود:

```
public void oneFrameRender( )
{
   if (!bInitOkay)
      throw new Exception("…");
   // محو الإطار السابق ومخزن البعد الثالث
   D3DDev.Clear(ClearFlags.Target |
            ClearFlags.ZBuffer, Color.FromArgb(255,
            0, 0, 64), 1.0F, 0);
    // بداية الإطار
    D3DDev.BeginScene( );
    //*****************************
          // ضع كود الرسم هنا
    // *****************************
    // إنهاء الإطار
    // D3DDev.EndScene( );
    // تنفيذ رسم المجسّمات
    // D3DDev.Present( );
}
```

والآن فلننتقل لكتابة كود الرسم.. ونظرا لأنّ قائمة التعليمات قد تعلو فتُخفي أجزاء من المكعبين أثناء حركتهما، فيجب هنا أن نراعي الترتيب التالي في الرسم: المكعّبين، القائمة، النصوص التي سنكتبها على القائمة.

سنبدأ بإعداد الجهاز لاستخدام بيانات الرءوس الخاصة بالمكعّب الأصليّ (ممّا سيؤثّر على النسختين المشتقتين منه):

```
// تحديد مصدر الرسم عن طريق تمرير مصفوفة رءوس المكعّب
D3DDev.SetStreamSource(0, vbCube, 0);
// تحديد نوع تنسيق البيانات في مصفوفة الرءوس
D3DDev.VertexFormat =
    CustomVertex.PositionTextured.Format;
```

وأخيرا سنرسم المكعّب الأوّل:

```
if (bRenderTextures)
    // رسم الخامة على أوجه المكعّب
    D3DDev.SetTexture(0, texCube1);
else
    // عدم رسم الخامة على أوجه المكعّب
    D3DDev.SetTexture(0, null);

// وضع مصفوفة التحويل الخاصة بالمكعّب الأوّل في مصفوفة الموضع
D3DDev.Transform.World = matCube1;
// تحديد طريقة الرسم (قائمة مثلثات).. //
// يتكوّن المكعّب من ١٢ مثلثا سنبدأ رسمها من أولها (رقم ٠) //
D3DDev.DrawPrimitives(PrimitiveType.TriangleList,
    0, 12);
```

لاحظ أنّ bRenderTextures هو متغيّر خاصّ بالفئة يحتوي على قيمة الخاصيّة useTextures، وهي خاصيّة يجب أن تضيفها للفئة لتسمح

للمبرمج الذي يستخدمها بتحديد إذا ما كان يرغب في استخدام الخامات أم لا (خاصيّة منطقيّة bool).. أعتقد أنّك تستطيع أن تكتب كود هذه الخاصيّة، فهو كود تقليديّ لا علاقة له بـ DirectX على الإطلاق.. (راجع الكود في التطبيق المرفق بهذا الفصل).

وسيتمّ رسم المكعّب الثاني بنفس الطريقة، مع استبدال texCube2 و matCube2 بـ texCube1 و matCube1.

الآن علينا أن نرسم القائمة ثنائيّة البعد:

```
// حفظ قيمة الخاصيّة في متغيّر احتياطي،
// لاستعادة قيمتها الأصليّة منه بعد رسم القائمة
bool bPrev = wireframe;
// يجب تعطيل إمكانيّة رسم الأشكال كشبكة خطوط،
// لأنّنا لا نريد أن تبدو القائمة كذلك
wireframe = false;
// تفعيل الشفافيّة، حتّى يتمّ استخدام لون الشفافية الذي اخترناه من قبل
// سيزيل هذا اللون القرمزيّ من خامة القائمة، ممّا سيمنحها حوافّ منحنية
D3DDev.RenderState.AlphaBlendEnable = true;
// مصدر الرسم (القائمة)
D3DDev.SetStreamSource(0, vbMenu, 0);
// تنسيق الرسم
D3DDev.VertexFormat =
    CustomVertex.TransformedTextured.Format;
// رسم الخامة على القائمة
D3DDev.SetTexture(0, texMenu);
// طريقة الرسم (شريط مثلثات).. تذكّر أنّ المستطيل يتكوّن من مثلّثين
D3DDev.DrawPrimitives(PrimitiveType.TriangleStrip,
    0, 2);
```

```
// إيقاف فاعلية الشفافية بعد رسم القائمة
D3DDev.RenderState.AlphaBlendEnable = false;
// لا تنس إعادة قيمة خاصيّة شبكة الخطوط لقيمتها الأصليّة
wireframe = bPrev;
```

لاحظ أنّ الخاصيّة wireframe أيضا هي خاصيّة يجب تعريفها داخل الفئة، لتسمح للمبرمج بأن يحدّد إذا ما كان يريد رسم المجسّم مغلقا، أم في صورة شبكة من الخطوط.. سأترك لك كتابة هذه الخاصيّة، ولكنّي سأرشدك للجزء الخاص بالرسوم فيها.. إذا اختار المستخدم رسم المجسّم مغلقا، فاستخدم الجملة:

```
D3DDev.RenderState.FillMode = FillMode.Solid;
```

وإذا اختار أن يرسم المجسّم في صورة خطوط، فاستخدم الجملة:

```
D3DDev.RenderState.FillMode = FillMode.WireFrame;
```

وأخيرا، يجب علينا أن نرسم النصوص التي ستظهر في أعلى النموذج.. وسنستخدم في هذا الوسيلة DrawText الخاصّة بكائن الخطّ.. لاحظ أنّ هذا الكائن خاص بـ DirectX وهو يختلف عن كائن الخطّ الذي نستخدمه في نماذج الويندوز، والذي ينتمي لفضاء الاسم Drawing.

```
Sprite Spt = new Sprite(D3DDev);
var Rec = new System.Drawing.Rectangle(5,
    5 + 2 * (iFontSize + 6) , 0, 0);
fntOut.DrawText(Spt, "Frame Rate: " +
    iFrameRate.ToString( ) + "fps", Rec,
    DrawTextFormat.Left | DrawTextFormat.Top,
    System.Drawing.Color.FromArgb(255, 200, 128, 64));
```

وعليك أن تحدّد مساحة المستطيل لكي يحتوي الخطّ.. لا تحاول أن تجعله أكبر ممّا ينبغي، لأنّ ذلك يهبط بكفاءة تطبيقك.. ويمكنك استخدام بعض وسائل GDI، لمعرفة حجم المستطيل الذي يحتوي الخط.

استخدام الفئة:

الآن صارت لدينا فئة جاهزة.. نريد إذن استخدامها.

كلّ ما سنفعله هو أن نستخدم نموذجا لنرى الرسوم عليه.. لن نحتاج لأيّ أدوات على هذا النموذج، وإن كان باستطاعتنا أن نضع ما نشاء من الأدوات (قد يؤدي ذلك لبعض المشاكل).

ابدأ بتعريف متغيّر من الفئة على مستوى النموذج:

```
private CSampleGraphicsEngine c3DEngine;
```

وفي حدث تحميل النموذج أنشئ نسخة جديدة من الفئة وضعها في هذا المتغيّر:

```
c3DEngine = new CSampleGraphicsEngine(this);
```

واضح أنّ هذا خاصّ بالرسم في نافذة.. أمّا إذا أردت أن ترسم على الشاشة بأكملها، فعليك باستخدام الصيغة الأخرى لحدث إنشاء الفئة.. في هذه الحالة عليك أن تتأكّد أولا من طور العرض الذي يسمح لك به كارت الشاشة.. ها هو ذا الكود:

```
int iW = 0; int iH = 0; int iD = 0;
if (CSampleGraphicsEngine.isDisplayModeOkay(1024,
    768, 32))
{
      iW = 1024;
      iH = 768;
      iD = 32;
}
else if (CSampleGraphicsEngine.isDisplayModeOkay(1024,
     768, 16))
{
      iW = 1024;
      iH = 768;
      iD = 16;
```

```
}
else if (CSampleGraphicsEngine.isDisplayModeOkay(640,
     480, 32))
{
     iW = 640;
     iH = 480;
     iD = 32;
}
else if (CSampleGraphicsEngine.isDisplayModeOkay(640,
     480, 16))
{
     iW = 640;
     iH = 480;
     iD = 16;
}
else
{
   // لا يوجد طور عرض متاح!!
   throw new Exception("No display modes found");
}
c3DEngine = new CSampleGraphicsEngine(this, iW,
     iH, iD);
```

بعد ذلك يمكننا ضبط خصائص الفئة:

```
c3DEngine.backFaceCulling = false;
c3DEngine.drawFrameRate = true;
c3DEngine.useTextures = true;
c3DEngine.wireframe = false;
bRunning = true;
```

لاحظ أنّنا لم نشرح هذه الخصائص، ولكن يمكنك الرجوع إليها في كـــود المشروع.. ليس فيها شيء صعب.

الآن كلّ ما علينا هو أن نكتب جملة تكراريّة يتمّ فيها تحديث الإطـــارات ورسمها.. سنعرّف أولا متغيّرا على مستوى النمـــوذج لنســـتخدمه فـــي الخروج من الجملة التكراريّة:

```
bool bRunning = true;
```

لا تنسَ أن تستخدم حدث ضغط الأزرار KeyDown لتجعل هذا المتغيّر False عندما يضغط المستخدم زرّ ESC.

```
while (bRunning)
{
    c3DEngine.oneFrameUpdate( );
    c3DEngine.oneFrameRender( );
    // يجب استخدام الجملة التالية
    //للسماح للنموذج بالاستجابة لضغطات الأزرار
    Application.DoEvents( );
}
```

وأخيرا سنكتب كـــود الحـــدث KeyDown، لنعطـــي فيـــه للمســـتخدم الإمكانيّات، الموضحة في الجدول التالي:

الزر	وظيفته
Esc	إنهاء حركة المكعبين.
F1	الانتقال من عرض الأجسام مغلقة إلى عرضها بالخطوط والعكس.
F2	استخدام الخامات أو منع استخدامها.
F3	محو الأوجه الخلفيّة أو تركها.
F4	كتابة معدّل رسم الإطارات أو عدم كتابته.

الكود الذي ينفذ هذه الوظائف مباشر جدا، فكل ما يفعله هو تغيير قيم بعض الخصائص.. ستجده في المشروع التعليمي.

وهكذا أخيرا تمّ هذا المشروع التعليميّ بحمد الله.. استمتع بتجربته، مع ملاحظة أنّ بإمكانك تطوير هذه الفئة لاستخدامها في مشاريعك القادمة.

ما الذي تعلّمناه في هذا المشروع:

بعد هذا الفصل التعليميّ، يجب أن تكون قد ألممت بما يلي:

١- أهمّ مفاهيم العالم ثلاثيّ الأبعاد.

٢- تكوين المشهد باستخدام الرءوس Vertices والمثلّثات.

٣- كيف يعمل Direct3D9 بالتوافق مع نظامك.

٤- إنشاء المجسّمات يدويّا، بدون الاستعانة بتطبيقات الرسوم المجسّمة.

٥- تحميل واستخدام الخامات.

٦- إعداد مصفوفات التحويل (تغيير الحجم أو الموضع أو زاوية الدوران)

٧- كيفيّة رسم المجسّمات على الشاشة، وأهمّ الاعتبارات التي يجب مراعاتها عند عمل ذلك.

كانت هذه خطوة صغيرة على بداية الدرب.

والله وليّ التوفيق.