

Creación de animaciones 3D avanzadas con Silverlight 2.0

Declan Brennan

Si por algún infortunio increíble, no le ha llegado toda la publicidad que se ha hecho en los últimos meses de Silverlight™, permítame que le ponga al día: Silverlight es un nuevo complemento entre exploradores de Microsoft que aporta todo el potencial de Microsoft® .NET Framework para trabajar en un área que estaba reservada anteriormente para los applets de Flash y Java. Silverlight dispone de muchas y muy útiles características que están listas para usar desde el momento en que se adquiere el producto. Admite una versión eficiente de .NET Framework 3.5 que incluye, entre otras cosas, el lenguaje de marcado de aplicaciones extensible (XAML, Extensible Application Markup Language), colecciones genéricas, servicios web y LINQ. Silverlight admite también una gama de idiomas compatibles con .NET, pero aquí me centraré en C#.

Creo que la mejor manera de familiarizarse con una tecnología nueva es pasar un buen rato con ella. Así que cuando se lanzó la versión alfa de Silverlight 1.1 y vi la emocionante presentación realizada por Tim Sneath en la conferencia de IMT celebrada en Dublín, decidí crear una pequeña aplicación educativa que demostrara cómo se pueden ensamblar varias formas tridimensionales (llamadas poliedros) doblando una plantilla plana. Silverlight no admite 3D de forma predeterminada, así que hubo que crear una emulación de las bibliotecas matemáticas de DirectX® para salvar las dificultades asociadas al 3D.

Un poliedro es un objeto tridimensional con caras planas. Este ejemplo de Silverlight explora los poliedros regulares y semirregulares denominados poliedros platónicos y arquimedianos. Todas las caras de estos poliedros son polígonos regulares (todos sus lados presentan la misma longitud), como triángulos equiláteros o cuadrados. También son convexos, es decir, ninguna de sus partes sobresale. Como puede adivinar por los nombres griegos antiguos, estos objetos han fascinado a la humanidad durante mucho tiempo. Si está interesado, puede obtener mucha más información sobre ellos en el sitio web de George Hart: georgehart.com/virtual-polyhedra/vp.html.

Puede ver una demostración de la aplicación terminada en la Figura 1 o en picturespice.com/ps/Polyhedra/default.html. Básicamente, la aplicación le permite seleccionar una forma (un poliedro) moviendo el mouse sobre dicha forma. Entonces se le muestra información acerca de su selección en la esquina superior derecha de la ventana, así como una animación de la plantilla plana doblándose

para formar el poliedro elegido. Finalmente, si hace clic en el botón Cycle, el programa recorre automáticamente cada una de las formas.

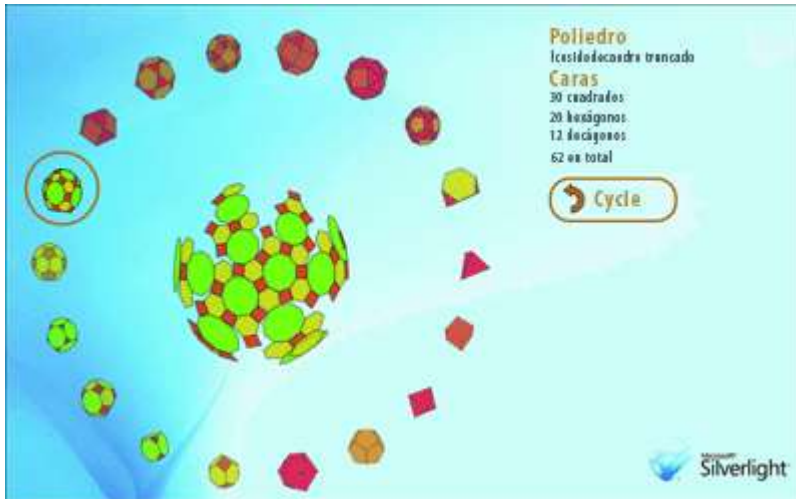


Figura 1 Demostración de Silverlight de Polyhedra (Hacer clic en la imagen para ampliarla)

Uso de XAML

Como muchas aplicaciones de Silverlight, Polyhedra hace un uso muy amplio de XAML, que es un idioma de definición de contenidos, una especie de equivalente de HTML, pero más flexible. Continuando con la analogía, aunque es posible crear una página HTML usando sólo el modelo de objetos de documento HTML (DOM, Document Object Model), esto no suele ser una manera sensata de producir contenidos, ya que se tarda mucho en codificar y se producen páginas de inicialización lenta. Casi siempre conviene mantener el mayor fragmento posible de página como marcado HTML y aumentarlo más tarde con JavaScript y DOM si es necesario ganar flexibilidad.

Se puede aplicar un enfoque muy similar a XAML. La manera más rápida de reunir el contenido es usar el marcado de XAML tanto como sea posible y aumentarlo cuando sea necesario mediante un idioma compatible con .NET, como C# y la API de medios de Silverlight. El XAML puede estar codificado a mano, producido por un paquete de diseño como Expression Blend™, generado desde un programa que se ejecuta durante el proceso de desarrollo o incluso generado dinámicamente en el servidor. Esto puede requerir un cambio de actitud. Es demasiado tentador como programador de C# acabar codificando en tu entorno natural funcionalidad que estaría mejor en manos de XAML.

Cualquier alusión a XAML que no fuera breve, rebasaría el ámbito de este artículo. Sin embargo, Charles Petzold describe XAML minuciosamente en su libro Applications= Code+Markup.

Aquí está el equivalente XAML del ejemplo "Hola mundo" que todos hemos esperado alguna vez durante el aprendizaje de un nuevo idioma:

```
<UserControl x:Class="Polyhedra.Page"
  xmlns="http://schemas.microsoft.com/client/2007"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Width="400" Height="300">
  <Grid x:Name="LayoutRoot" Background="White">
    <TextBlock>Hello World</TextBlock>
  </Grid>
</UserControl>
```

El elemento raíz es un UserControl. Contiene una cuadrícula que a su vez contiene un elemento TextBlock con el texto "Hola Mundo".

Eche un vistazo rápido a los atributos UserControl. Sin entrar demasiado en detalle, esto define el equivalente de una clase subyacente para el XAML. La instancia de esta clase se crea al mismo tiempo que se analiza y carga el XAML. Puede realizar varios bits de inicialización en el constructor, pero para obtener un comportamiento más elaborado es a menudo necesario emplear controladores de eventos. Esto es una característica clave de Silverlight. Los controladores de eventos se pueden adjuntar a varios objetos XAML e implementarse en el idioma compatible con .NET de su elección; un idioma que, a diferencia de JavaScript, se compila y por lo tanto abre todo tipo de posibilidades que de otro modo serían poco viables.

Volviendo a la analogía HTML, los elementos a menudo se agrupan dentro de varios DIV para distribuir su ubicación en la página. De forma similar, en XAML, las formas se agrupan dentro de lienzos (u otros elementos como cuadrículas, que son tipos de lienzos). Del mismo modo que los DIV a menudo se anidan en HTML, los lienzos pueden anidarse en XAML. La mayoría de los elementos en HTML son rectangulares. Sin embargo, XAML admite una amplia gama de formas como bloques de texto, rectángulos, polígono, elipses y la muy flexible ruta de acceso, que tiene en cuenta las formas definidas por el usuario. Los elementos de HTML se identifican con un atributo de identificador. El atributo equivalente para identificar un elemento en XAML es x:Name, donde x es el alias del espacio de nombres de XAML.

Además de producir páginas estáticas, XAML hace mucho más. Una de sus características más eficaces es el uso de guiones gráficos como una manera de animar los cambios a la IU inicial especificados por el XAML. Algo similar denominado HTML+Time, se agregó a HTML en Internet Explorer® 5.0. Un ejemplo podría ser animar un cambio realizado al color, la visibilidad o la transparencia de un objeto. Combinados con transformaciones, los guiones gráficos también pueden girar, escalar o mover objetos.

Los guiones gráficos pueden generar todo tipo de efectos de animación con muy poco o ningún código convencional. Si se combinan con desencadenadores, es posible, en teoría, iniciar varias animaciones automáticamente cuando un evento como `MouseEnter` ocurre en un objeto. Ay, desde la versión beta de Silverlight 2.0 de marzo de 2008, `Loaded` es el único evento administrado por un desencadenador. Los otros eventos, requieren una cantidad pequeña de código estructural a modo de controlador de eventos. Espero que esta situación cambie pronto, si no lo ha hecho ya.

Una característica importante de los guiones gráficos de Silverlight es que se basan en el tiempo y no en marcos. El uso de varios guiones gráficos independientes enlazados a diferentes tiempos y eventos puede simplificar la implementación de un comportamiento complejo. A fin de cuentas, el mundo real no funciona en marcos (es una resaca de la forma en que se logró hacer películas en celuloide). La implementación de objetos independientes con comportamientos independientes simplifica muchísimo la vida.

Algunos ejemplos de XAML

La animación plegable de la aplicación Polyhedra se genera con código C#. Sin embargo, prácticamente el resto se ha definido en XAML. Esto incluye el círculo de poliedros de ejemplo, la forma en la que se enfatiza de varias maneras el poliedro seleccionado en la actualidad y el botón `Cycle`, que indica que está activado animando una flecha que gira.

Echemos un vistazo a un par de extractos de `Page.xaml` para ver cómo se consiguen estas animaciones, empezando por una mirada a la [Figura 2](#), que nos muestra el botón `Cycle`. Eche un vistazo a la ruta de acceso denominada `Cycle`. El atributo de datos especifica una serie de operaciones que incluyen `M` de mover, `A` de arco y `L` de línea. Como se trata de un glifo bastante sencillo, simplemente dibujé la forma que quería en una hoja de papel y realicé las operaciones necesarias manualmente. En la mayoría de los casos, le irá mejor si usa una herramienta como `Expression Design`.

Esta ruta de acceso tiene también un `RotateTransform` denominado `CycleRotate`. Inicialmente, esta transformación no hace nada, ya que su ángulo está establecido en 0. Sin embargo, hay un `Storyboard` de nombre `CycleLatched` que cuando está activo cambia continuamente el ángulo en incrementos pequeños, haciendo que la flecha gire.

La [Figura 3](#) muestra cómo se define una de las muestras de poliedro. En la parte inferior de este XAML, puede ver que la muestra consta de los cuatro polígonos que definen el tetraedro. Se encuentran dentro de su propio lienzo, denominado `Model0`, y están rodeados por un anillo o elipse de nombre `Ring0` que es un

principio es invisible. Hay dos animaciones Storyboard definidas, una para MouseEnter y otra para MouseLeave. La animación MouseEnter hace que el anillo sea visible inmediatamente, infla el tamaño del lienzo Model0 y lo hace más opaco durante un período de 7 segundos. La animación MouseLeave invierte cada uno de estos cambios.

Puesto que estos guiones gráficos pueden funcionar de manera independiente, todo sucede de la manera esperada sin que sea relevante la rapidez con la que el usuario mueve el mouse. Normalmente, la animación MouseLeave para una muestra y la animación MouseEnter para una muestra recientemente seleccionada, tendrán lugar al mismo tiempo. Sin embargo, no tiene que preocuparse de esto, ya que Silverlight administrará automáticamente la ejecución en paralelo de varios guiones gráficos activos.

Es posible que se pregunte cómo se han calculado los vértices de cada uno de los cuatro polígonos de la muestra del tetraedro, por no mencionar el muy superior número de polígonos de las otras muestras de la aplicación. Creo bastante en el enfoque perezoso que consiste en no hacer nada que un equipo pueda hacer más rápido. Así que todo lo que hice fue una versión modificada de Polyhedra durante el proceso de producción. Esta versión realizó un marco de la animación central para cada una de las muestras (el marco con la forma completamente cerrada). Coloqué cada uno de los grupos de polígonos resultantes en un conjunto de lienzos que estaban igualmente espaciados alrededor de un círculo y transmití el lote entero en secuencias a un archivo para que fuera usado por el programa principal.

Colocar objetos alrededor de un círculo implica aumentar el ángulo en incrementos iguales de $2 * \text{PI} / \text{NumSamples}$ y especificar a continuación el centro de cada muestra usando una coordenada de $x = \text{Radius} * \text{Cos}(\text{Angle})$, $y = \text{Radius} * \text{Sin}(\text{Angle})$. Además, puesto que el lienzo que posiciona los objetos usa las propiedades Left y Top, tuve que desplazar el centro la mitad de la anchura y la mitad la altura respectivamente.

Sugerencias para trabajar con XAML

Como puede ver, es posible lograr una gran IU con XAML y casi sin ningún código adicional. La inicialización es sorprendentemente rápida incluso con archivos grandes como Page.xaml. Pero antes de pasar a otra cosa, aquí tiene algunas sugerencias basadas en mi experiencia con la implementación actual (versión beta de marzo de 2008) de Silverlight 2.0.

Como ya he mencionado, actualmente los desencadenadores sólo pueden iniciar un guión gráfico automáticamente (a través del método Begin) para el evento

Loaded. Los demás eventos, requieren una cantidad pequeña de código estructural a modo de controlador de eventos:

```
public void MouseEnterHandler(
    object o, EventArgs e) {
    this.MouseEnterStoryboard.Begin();
}
```

Si adopta una convención de nomenclatura para los guiones gráficos, como el nombre de objeto seguido del nombre de evento, podrá reducir en gran medida el número de controladores de eventos separados que necesita codificar. Por ejemplo, este método se usa en Polyhedra como un controlador de eventos compartido para todas las muestras del círculo:

```
public void MouseEnterHandler(
    object o, EventArgs e) {
    this.triggerStoryboard(o, "MouseEnter");
}
private bool triggerStoryboard(
    object o, string eventType) {
    Canvas el = o as Canvas;

    string name = el.GetValue(NameProperty) as String;
    Storyboard sb = el.FindName(name + eventType) as Storyboard;
    if (sb != null)
        sb.Begin();
    return (sb != null);
}
```

Con la versión beta de Silverlight 2.0, la inicialización principal (llamando a InitComponents) ha pasado de un controlador de eventos Loaded al constructor del objeto subyacente. Esto es más elegante, pero tenga en cuenta que no todo es posible en el constructor. Por ejemplo, no es posible llamar a Begin o Pause en un guión gráfico, así que esto hay que seguir haciéndolo en un controlador de eventos.

Tal y como descubrí en la demoledora muestra de Silverlight Rocks! de Andy Beaulieu (www.andybeaulieu.com/Home/tabid/67/EntryID/73/default.aspx), una buena manera de lograr una animación basada en código es usar un conjunto de guiones gráficos durante un período corto de tiempo y tener un controlador de eventos Completed que haga un marco de animación y, a continuación, reinicie el guión gráfico:

```
public Page() { // Constructor for "code-behind"
    // Required to initialize variables
    InitializeComponent();
    this.animationTimer.Completed +=
        new EventHandler(animationTimer_Completed);
}
```

```

void animationTimer_Completed(object sender, EventArgs e) {
    [ Do a frame of animation ]
    this.animationTimer.Begin();
}

```

La actualización de septiembre de la versión alfa de Silverlight 2.0 cambió los requisitos para guiones gráficos. Así, una animación debía tener un destino incluso si no se usaba:

```

<Canvas.Resources>
  <Storyboard x:Name="animationTimer">
    <DoubleAnimation Duration="00:00:00.01"
      Storyboard.TargetName="bogusTimerTarget"
      Storyboard.TargetProperty="Width" />
  </Storyboard>
</Canvas.Resources>
<Canvas Name="bogusTimerTarget">
</Canvas>

```

No intente tener demasiados controles Silverlight individuales en la misma página HTML. Mi primera implementación de Polyhedra usaba un control separado para cada muestra del círculo y, realmente, devoraba la memoria. Esto puede traducirse, a veces, en la necesidad de mover el contenido de HTML a XAML para reducir el número de controles que se deben usar.

Una de las ventajas de XAML es que descarga mucho material rutinario en la IU, permitiéndole concentrarse en el código del dominio con problemas. En este ejemplo, el dominio con problemas implica doblar las plantillas para componer las formas tridimensionales, lo cual nos dirige a la siguiente sección.

Cómo doblar un poliedro

Sospecho que el libro de Charles Petzold, mencionado anteriormente, es una adaptación de un libro mucho más antiguo, anterior a la orientación a objetos, escrito por Niklaus Wirth y titulado Algorithms+Data Structures=Programs. Incluso después de todos estos años sigue siendo uno de los libros más influyentes que he leído jamás, y, a pesar de todos los cambios relacionados con los idiomas y los paradigmas que han tenido lugar desde entonces, conserva toda su vigencia. El principio básico del libro es una aproximación al desarrollo que consiste en identificar qué estructuras de datos pueden afrontar mejor el problema y, a continuación, identificar qué algoritmos pueden procesar o modificar estas estructuras de datos. Éste es un enfoque que adopto a menudo cuando se trata de abordar un programa no estándar.

Probé varios enfoques antes de decidirme finalmente por el que se usa en Polyhedra. Quería ver con qué poca información tendría que contar para lograr la animación que usamos para plegar formas. Resulta que puesto que todos los lados tienen la misma longitud, es posible hacerlo casi todo sabiendo simplemente qué caras están conectadas entre sí en un poliedro dado. Esto sugiere una estructura de datos de gráfico. Antes de alcanzar la producción final de XAML, se usa un conjunto de algoritmos para procesar el gráfico a través de dos estructuras de datos de árbol separadas. Explicaremos todo esto más tarde.

Aunque Windows® Presentation Foundation (WPF) puede admitir 3D en XAML, Silverlight sólo admite 2D de forma predeterminada, ya que la compatibilidad entre exploradores es mucho más fácil de lograr cuando no hay que preocuparse por el hardware de la unidad de procesamiento de gráficos (GPU) con que cuenta el equipo. Por supuesto, el 3D en la pantalla de un equipo no es más que una ilusión. Independientemente de las manipulaciones que puedan suceder, al final se muestra un conjunto de polígonos ordinarios en 2D en la superficie del monitor. Si está dispuesto a hacer las manipulaciones de 3D en su propio código para calcular las coordenadas de estos polígonos, es posible hacer una representación en 3D no acelerada por hardware. Siempre que se limite a unas cuantas docenas de polígonos, el rendimiento es aceptable. Subjetivamente, el rendimiento de marco ha mejorado al pasar de la versión alfa a la versión beta.

Cuando se selecciona una forma, se crea una plantilla desplegada bidimensional denominada Net (nada que ver con .NET). Esto se hace en dos fases, como ilustra la Figura 4. Primero se crea un gráfico en la memoria con un GraphNode correspondiente a cada cara de la forma (consulte Graph.cs). Este gráfico se crea transmitiendo en secuencia un conjunto de información de conectividad (indicando qué caras están conectadas entre sí) desde uno de los recursos .shp que están incrustados en el ensamblado de la aplicación. Por ejemplo, aquí está el contenido de cube.shp:

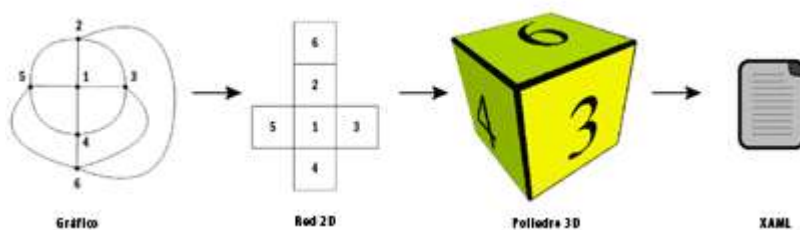


Figura 4 Creación de la forma, de gráfico a XAML (Hacer clic en la imagen para ampliarla)

- 1:3,4,5,2
- 2:1,5,6,3
- 3:2,6,4,1
- 4:3,6,5,1
- 5:1,4,6,2
- 6:2,5,4,3

Cualquier GraphNode del gráfico se puede elegir como el primer nodo desde el que empezar a crear la Net (consulte Net.cs en la descarga de código para obtener más detalles). Se dispone entonces un FlatFace bidimensional con tantos lados como vecinos tiene el GraphNode. Se elige entonces cualquier GraphNode vecino para repetir el proceso, disponiendo el siguiente FlatFace de manera que comparta un lado con el FlatFace actual. Cada GraphNode se visita sólo una vez y el proceso no se detiene hasta que se hayan visitado todos los GraphNodes, teniendo como resultado una estructura de árbol de FlatFaces.

En cada marco de la animación de 3D, se crea un poliedro tridimensional (es posible que parcialmente cerrado) desde la Net (consulte Polyhedron.cs en secciones posteriores de este artículo para obtener los detalles de esta operación). Esto implica copiar el árbol de FlatFaces en un árbol de Faces reemplazando los vértices 2D con vértices 3D. La Figura 5 muestra los dos sistemas de coordenadas. Como quiero empezar en el plano horizontal, Y debe iniciarse como cero. Así que para asignar de 2D a 3D configuro $X=X$, $Y=0$ y $Z=Y$.

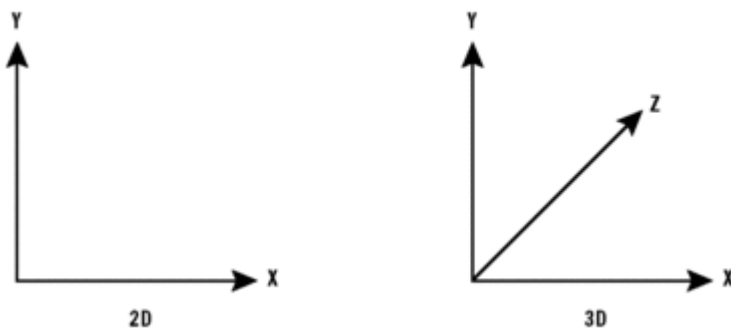


Figura 5 Transformación de vértices de 2D a 3D (Hacer clic en la imagen para ampliarla)

Finalmente, hacemos uso de la recursividad para visitar cada unión del árbol de Faces y realizar un pliegue. La cantidad del pliegue es una fracción del ángulo (denominado ángulo diedro) que las caras formarán entre sí cuando la forma esté completamente cerrada. Es posible, en teoría, calcular los ángulos diedros usando directamente la información de conectividad incluida en el archivo .shp.

Esto es bastante fácil en casos especiales como, por ejemplo, tres caras cualesquiera que se unen en un vértice o un número indeterminado de caras del mismo tipo que se reúnen en un vértice. Sin embargo, el caso general es algo más difícil, así que decidí almacenar estos ángulos en recursos .dihedrals por separado. Por ejemplo, aquí tiene un extracto de cube.dihedrals:

```
1,3:1.5707965056551
1,4:1.57079661280793
1,5:1.57079614793469
1,2:1.57079604078186
```

2,1:1.57079604078186
2,5:1.57079628466395
2,6:1.57079661280793
2,3:1.57079636892585
3,2:1.57079636892585
3,6:1.57079614793469
3,4:1.57079628466395
...

Las pequeñísimas diferencias entre estos números, que deberían ser todos $\pi/2$ en este caso, son solamente consecuencia de la manera en que se calcularon.

El proceso final implica una serie de transformaciones para proyectar una vista del poliedro en el monitor como un conjunto de polígonos. Puede que haya advertido que durante la animación que pliega la forma, el poliedro también gira sobre un eje vertical. Para lograr esto, traduzco el punto de pivote al origen, realizo el giro y , a continuación, traduzco al punto de vista. Entonces hago una transformación de la perspectiva para asegurarme de que los objetos lejanos (en el eje Z) aparecen más pequeños. Explicaré un poco más cómo hacer las transformaciones en la siguiente sección.

Finalmente, se genera un conjunto de polígonos de XAML que corresponden a la proyección de la forma 3D en la superficie del monitor. Se fuerza a Silverlight para que disponga estos polígonos en orden del último al primero mediante la configuración de la propiedad XAML `ZIndex` para cada polígono usando la coordenada Z del centro del polígono 3D (escalado para ajustar un flotante en un int). Ésta es una forma bastante simplista de lograr 3D que sólo funcionará si los polígonos se portan bien, por ejemplo, no formando intersecciones entre sí (algo que me puedo permitir en este ejemplo). Las formas más sofisticadas de 3D incorporan mecanismos como búferes de profundidad. Como éstos requieren acceso a GPU, quedan fuera del ámbito del recinto de seguridad de Silverlight .NET.

Como toque final, XAML permite que los polígonos sean parcialmente transparentes (usando la propiedad `Opacity`), que da el efecto artístico de un poliedro en parte translúcido. Y como se suele decir, esto es todo lo que hay.

Emulación de las matemáticas de DirectX

Pasaré levemente por las matemáticas usadas para lograr la animación. Si desea obtener información más detallada, hay muchos sitios en la Web que le ayudarán, como wikipedia.com, euclideanspace.com, mathworld.wolfram.com, gamasutra.com y gamedev.net.

Ya he introducido algunas transformaciones 2D, como `RotationTransform` y `ScaleTransform`, en los ejemplos de XAML descritos anteriormente. En estos casos, Silverlight hace todo el trabajo por usted. Si usa XAML en un entorno de WPF, también tendrá acceso a transformaciones 3D, pero éstas no están disponibles en Silverlight. WPF depende de DirectX para hacer el trabajo de bajo nivel. Esto tiene sentido, ya que DirectX tiene un buen conjunto de clases para hacer las matemáticas necesarias para las transformaciones 3D. Para hacer algunas de las mismas transformaciones en Silverlight, he producido una emulación de las matemáticas de DirectX de muchas de estas clases para que se puedan usar en el recinto de seguridad de Silverlight.

Si tiene experiencia de codificación en `Direct3D®`, se sentirá bastante cómodo con las implementaciones de Microsoft de `Vector2`, `Vector3`, `Matrix` y otros conceptos similares (consulte la Figura 6). De lo contrario, le haré una introducción general muy breve y muy superficial y le daré algunos ejemplos.

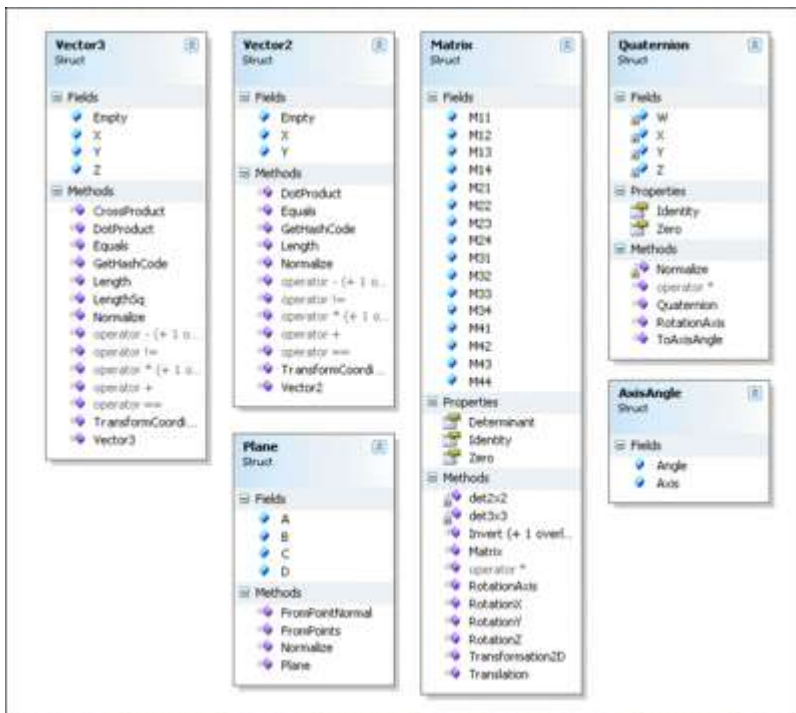


Figura 6 Clases emuladas de las matemáticas de DirectX (Hacer clic en la imagen para ampliarla)

Localizar un punto en 2D requiere dos coordenadas: X e Y. En 3D se necesitan tres coordenadas: X, Y y Z. Aunque siendo rigurosos, los puntos no sean vectores, los puntos 2D se pueden almacenar en un objeto `Vector2` y los puntos 3D en un objeto `Vector3`. Los vectores son realmente entidades con una magnitud y una dirección que podemos ilustrar con una flecha que empieza en el origen y termina en el punto. De algún modo, se puede pensar que todos los vectores empiezan en el origen.

Las formas en tres dimensiones se pueden identificar como un conjunto de caras. Las caras, por su parte, se identifican como un conjunto de puntos que corresponden a los vértices. Casi todo que lo quizás desee hacer implica transformar estos puntos en otro conjunto de puntos aplicando una o más operaciones como el giro, la traducción (o el movimiento) o la escala. Casi todas las operaciones que desee realizar podrán representarse en una cuadrícula 4x4 de números denominada matriz. En general, las matrices pueden tener cualquier número de filas y columnas, pero sólo nos interesan las matrices especiales que se usan para manipular los puntos 3D con las llamadas coordenadas homogéneas. Estas operaciones se conocen como transformaciones lineales porque no distorsionan (excesivamente) la forma del objeto al que se aplican.

De manera muy conveniente, existe una manera de combinar ambas matrices en una sola que tenga el mismo efecto. Esta operación se llama multiplicación de matriz y se puede repetir varias veces. Esto significa que puede encadenar un conjunto entero de transformaciones en una sola matriz que tenga el mismo efecto. Esto es mucho más eficaz que aplicar cada transformación por separado. No hay que preocuparse por cómo se va a implementar la multiplicación de matriz o cómo ésta va a hacer sus trucos de magia con las transformaciones. Simplemente úsela como una herramienta.

Para transformar un punto de origen en un punto de destino, hay también una operación de multiplicación definida entre una matriz y un objeto de vector. Para evitar confusiones (con la multiplicación matriz-matriz), esto se implementa en Vector3 usando el método TransformCoordinate.

Un par de fragmentos pequeños de código aclararán las cosas. Primero, se usa el código siguiente en polyhedron.cs para doblar dos caras por el lado que tienen en común:

```
Vector3 axis = axisTo - axisFrom;
Matrix foldTransform =
    Matrix.Translation(-axisFrom) *
    Matrix.RotationAxis(axis, proportion * _dihedralAngle) *
    Matrix.Translation(axisFrom);
Vector3[] p= Vector3.TransformCoordinate(
    face.Points, foldTransform);
```

Aquí, el lado en común se define como una línea que va desde axisFrom hasta axisTo. Los giros siempre se definen alrededor de una línea que atraviesa el origen, de modo que antes de iniciar el giro tengo que mover un punto del lado al origen y devolverlo a su posición más adelante. Para hacer esto, voy a encadenar tres transformaciones lineales para crear la matriz que voy a usar para transformar la coordenada de la cara: mover al origen, girar y devolverlo a su posición.

En este punto recurrimos a un pequeño truco: el vector que representa el eje de rotación también tiene que empezar desde el origen, así que resto axisFrom, moviendo (axisFrom,axisTo) a (origen, axisTo-AxisFrom).

Si no tiene la cabeza demasiado llena de matemáticas a estas alturas, podemos probar otro ejemplo (basado en projector.cs) que se usa para transformar los polígonos del poliedro en lo que espera ver desde el punto de vista de pantalla:

```
Matrix projection =  
  Matrix.Translation(-pivot) *  
  Matrix.RotationY(yaw) *  
  Matrix.RotationX(angle) *  
  Matrix.Translation(viewPoint) *  
  Geometry.Perspective(5);  
Vector3[] p= Vector3.TransformCoordinate(face.Points,foldTransform);
```

Recuerde que, al tiempo que se pliega, la forma está girando en el plano horizontal. Para hacer esto, se mueve un punto del eje de pivote al origen. Entonces tiene lugar un giro en torno al eje vertical (Y) a través del ángulo de rotación. Si ha pilotado un avión en la realidad o en modo de simulación, se habrá topado con las otras dos posibles rotaciones independientes, denominadas pitch y roll (longitudinal y lateral).

Esta vez, en lugar de retroceder a la posición de inicio, se produce un movimiento hacia el punto de vista. Finalmente, tiene lugar una transformación de la perspectiva para reducir los objetos más alejados sobre el eje Z. Todo lo que está ocurriendo aquí se puede combinar en una sola matriz 4x4, algo que resulta bastante impresionante.

Más cosas que explorar

Aunque no se usan en Polyhedra, también hay cuaterniones disponibles en la biblioteca de emulación. Los cuaterniones constituyen una buena manera de combinar un conjunto de rotaciones 3D en un solo giro. Su uso es muy útil también para cambiar de una orientación a otra de forma poco traumática. Los descubrió mi paisano dublinés Sir William Rowan Hamilton mientras paseaba por la calle. Sir William garabateó inmediatamente la ecuación en el puente Broome Bridge para no olvidarla (vaya a www.maths.tcd.ie/pub/HistMath/People/Hamilton/Quaternions.html). Quiero aclarar que todos nuestros grafitis comparten el mismo carácter erudito, pero supongo que no me creará.

Los cuaterniones evitan también el denominado problema de gimbal. Esto jugó un papel importante en la navegación giroscópica durante las misiones Apolo a la luna, rescatadas por el cine con la película Apolo 13.

Hay mucho más material en el que no me he detenido, como los productos vectoriales de punto y de cruz, pero espero haberle dado una idea de lo que la emulación de las matemáticas de DirectX nos permite hacer. Originalmente desarrollé esta emulación para hacer transformaciones en un servidor hospedado que ejecute ASP.NET y en el que no se pueda instalar DirectX. Puede encontrar otras aplicaciones para este fin en este tipo del entorno.

Polyhedra debe darle una idea de cómo Silverlight proporciona una herramienta útil y extensible para interfaces de usuario web que realmente llaman la atención del usuario y que obtienen y distribuyen la información de forma rápida y memorable. Puede crear su experiencia .NET de muchas formas creativas y, a diferencia de JavaScript, no tiene que obsesionarse con el número de líneas de código que se ejecutan en el cliente.

Declan Brennan tiene la edad suficiente para recordar el primer microprocesador y todavía no se ha acostumbrado a tener su propio genio de la lámpara maravillosa. Declan no puede creer lo afortunado que es de estar en un mundo cuyos límites no vienen dictados por la tecnología, sino por la imaginación. Obtenga más información acerca de Declan en declan.brennan.name.

© 2007 Microsoft Corporation and CMP Media, LLC. Reservados todos los derechos; queda prohibida la reproducción parcial o total sin previa autorización.