Uso del hardware gráfico en las Transformaciones Funcionales

Domingo Martín
Universidad de Granada
dmartin@ugr.es

Mark Segal
ATI
segal@ati.com

Resumen

En los últimos años ha habido un gran avance en el hardware gráfico. Las nuevas capacidades están permitiendo el desarrollo de nuevas implementaciones de algoritmos conocidos, teniendo como objetivo el conseguir versiones más rápidas. En otros casos, estas capacidades permiten el desarrollo de nuevos algoritmos.

Para estudiar las ventajas y desventajas del uso del hardware gráfico para las deformaciones, hemos optado por las Transformaciones Funcionales. Su fácil adaptabilidad a las nuevas capacidades nos ha permitido comprobar las limitaciones del hardware gráfico, y al mismo tiempo proponer soluciones a las mismas.

Palabras clave: deformaciones, hardware gráfico.

1. Introducción

En los últimos 4 años, el hardware gráfico ha sufrido un gran avance con la introducción de la programabilidad. Esta nueva capacidad implica flexibilidad y, en general, un aumento de la velocidad. Esta flexibilidad está permitiendo que, por un lado, algoritmos que ya estaban desarrollados sean adaptados, de tal forma que se obtengan un mejor rendimiento en velocidad, y que, por otro lado, se abra la posibilidad de desarrollar nuevas ideas y métodos.

Las Transformaciones Funcionales (TFs) son una clase de transformaciones geométricas cuya principal característica está en que el parámetro que controla las transformaciones depende de una función. Este tipo de transformación se usa especialmente para producir deformaciones orientadas hacia la animación de objetos blandos (*soft objects*), aunque sin ser físicamente exactas.

En este artículo se han investigado las ventajas y desventajas de implementar las Transformaciones Funcionales usando el hardware gráfico actual, teniendo como ventajas su sencillez, la capacidad de crear cualquier tipo de deformación, y su fácil adaptación a las capacidades del hardware gráfico, frente a la dificultad de las Deformaciones Libres de Forma (*Free Form Deformations*).

A pesar de basar el estudio en un solo tipo de deformaciones, los resultados obtenidos permiten extrapolar ciertas conclusiones, exponiendo las limitaciones del hardware gráfico, pero a la vez proponiendo soluciones que pueden ser adoptadas en futuros desarrollos. También se plantea la idoneidad de ejecutar el método en la *CPU* o en la *GPU, Graphics Processing Unit*.

2. Trabajos previos

Aunque la programabilidad en el hardware gráfico es relativamente nueva, existen ya numerosos trabajos que hacen uso de la misma. Para el caso de los trabajos relacionados con las transformaciones, el artículo en el que se describe por primera vez esta nueva capacidad es en el trabajo de Lindholm[8]. En el mismo se expone el concepto de los programas de vértices (*vertex programs*), y se incluye un ejemplo (*warp*) de ciertas deformaciones aplicadas a diferentes modelos. En este caso, las transformaciones presentadas son muy sencillas. En el trabajo de James[4] se implementan transformaciones más realistas y físicamente correctas, basándose en los modos de vibración y en la capacidad de la paleta de matrices para transformaciones (*matrix palette skinning*). La principal característica es que, aun siendo costosas, la aplicación de estas transformaciones es más rápida que en la *CPU*.

En relación con las transformaciones, los trabajos más importantes son las Transformaciones No-Lineales (*Non-Lineal Transformations*) de Barr[1] y las Deformaciones Libres de Forma (*Free Form Deformations*) de Sederberg[13]. Otros artículos relacionados con las deformaciones son los trabajos de Coquillart[2, 3], Lazarus[7], y Karan[5].

Transformaciones Funcionales es un nombre genérico para las Transformaciones No-Lineales Extendidas Jerárquicas desarrolladas por Martín[12, 9]. Este tipo de transformación se ha usado para producir deformaciones que reproduzcan las incoherencias de la animación bidimensional[11] y para la generación de lentes virtuales[10].

3. Transformaciones Funcionales

Aunque la Deformación Libre de Forma es un método de deformación ampliamente utilizado, para un caso general su complejidad lo convierte en un escollo difícil de solventar cuando se pretende implementar usando las capacidades del actual hardware gráfico. En cambio, las Transformaciones Funcionales permiten el estudio, ya que las mismas son muy fáciles de adaptar a estas nuevas capacidades, manteniendo un alto grado de flexibilidad.

Hagamos una breve revisión de los principales conceptos de las TFs. Las TFs son una extensión y formalización de las Transformaciones No-Lineales de Barr y de las curvas factor (*"factor curves"*) de Watt[15]. A su vez, las Transformaciones No-Lineales son una generalización de las transformaciones geométricas, traslaciones, rotaciones, escalados, para las que la magnitud de la transformación depende de una función y de las propias coordenadas del vértice que va a ser transformado. Esto se puede expresar más formalmente de la siguiente

manera:

$$(x', y', z') = T(x, y, z) \quad T : F(x, y, z)$$

Puesto que para transformar un vértice es necesario hacer referencia a sus propias coordenadas, ésto implica que para cada vértice se debe crear una transformación.

Las TFs usan una idea similar a las Transformaciones No-Lineales y a las curvas factor, pero definiendo las transformaciones de una manera más general y formal. Para ello se introducen los conceptos de *eje de selección* (la coordenada que se usa como variable independiente), *función de control* (la función que establece la relación entre la variable independiente el parámetro de la transformación), *eje de aplicación* (coordenada a la que se aplica la transformación), y las capacidades de poder componer las transformaciones y de estructurar las transformaciones jerárquicamente. Otro cambio es que la animación se produce al transformar las funciones de control de una manera más flexible que la propuesta por Watt, ya que se hacen depender de otras funciones de control de una manera genérica.

La introducción del eje de aplicación puede producir resultados parciales. Este problema es resuelto mediante la posibilidad de combinar varias transformaciones. Para ello, se usa el método de la *lista inicial* y la *lista final*, estableciendo una relación de posición entre vértices. Para aplicar una transformación se toman las coordenadas de los vértices de la *lista inicial*, pero se aplican a los vértices de la *lista final*. Esto es:

$$Vertice_{lista\ final} = T(Vertice_{lista\ final}) \quad T : F(Vertice_{lista\ inicial})$$

En algunas ocasiones es necesario actualizar el contenido de la *lista inicial* con los valores de la *lista final*, generalmente cuando se produce un cambio de sistema de coordenadas.

El hecho de usar para las transformaciones las coordenadas de la *lista final* permite mantener la coherencia en la definición de las mismas. Por ejemplo, se pueden definir una rotación y un escalado de manera sencilla, pues el usuario conoce el objeto al que se aplican dichas transformaciones, el objeto original. En otro caso, el usuario debería conocer a priori el resultado de cada transformación para poder saber cómo será la siguiente.

4. Transformaciones constantes y programas de vértices

Generalmente la creación de un objeto se realiza mediante estructuras jerárquicas, las cuales establecen un orden de aplicación de las transformaciones. Para ello se utiliza una pila, en la cual se van incluyendo y acumulando las transformaciones geométricas, de tal manera que la transformación resultante se aplica a todos los vértices.

Este es el caso normal, en el que las transformaciones geométricas son constantes (el parámetro de la transformación no varía para los vértices). La aplicación principal se encarga de modificar el contenido de la pila, y OpenGL y el hardware gráfico se encargan de calcular y aplicar las transformaciones. Con la llegada de la programabilidad, aparecen los programa de vértices, procedimiento mediante el cual se flexibiliza la aplicación de las transformaciones.

4.1. Programas de vértices

Un programa de vértices es un programa que se ejecuta en el procesador gráfico, en el que dados como entrada un vértice y otros atributos asociados (p.e. el vector normal), permite aplicarles operaciones para obtener los valores de salida, sin la posibilidad de crear nueva geometría.

Para ejecutar los programas de vértices, el hardware expone el siguiente modelo de máquina (GeForce4 y Radeon8500):

- Atributos de vértice: $\geq$ 16 registros de 4 valores. Contienen los valores del vértice, como la posición, color, etc.
- Programa: $\geq$ 128 instrucciones.
- Resultados: $\geq$ 8 registros de 4 valores. Contienen los resultados obtenidos, como la posición, color, etc.
- Variables de entorno: $\geq$ 96 registros de 4 valores. Mantienen valores comunes para todos los programas de vértices.
- Variables locales: $\geq$ 96 registros de 4 valores. Mantienen valores propios de cada programa de vértice.
- Variables temporales: $\geq$ 12 registros de 4 valores.
- Registros de direccionamiento: $\geq$ 1 registro de 4 valores.

Veamos un ejemplo sencillo de programa de vértices, en cual, las coordenadas de mundo son transformadas a coordenadas de vista:

```
!ARBvp1.0
ATTRIB Posicion_entrada=vertex.position;
ATTRIB Color_entrada=vertex.color;
OUTPUT Posicion_salida=result.position;
OUTPUT Color_salida=result.color;
PARAM Matrix_modelview[4]={state.matrix.mvp};

DP4 Posicion_salida.x, Matrix_modelview[0], Posicion_entrada;
DP4 Posicion_salida.y, Matrix_modelview[1], Posicion_entrada;
DP4 Posicion_salida.z, Matrix_modelview[2], Posicion_entrada;
DP4 Posicion_salida.w, Matrix_modelview[3], Posicion_entrada;
MOV Color_salida, Color_entrada;
```

En este ejemplo se puede observar que en las primeras líneas se establecen relaciones entre variables del programa y valores disponibles de OpenGL. Por ejemplo, a la posición del vértice se le asocia una variable llamada `Posicion_entrada`. Esta posición es multiplicada (con un producto escalar mediante la instrucción DP4 (*Dot Product*)), y se obtiene la posición en coordenadas de vista, asignada a la variable `result.position`, la cual será utilizada en otras fases del proceso de visualización (*pipeline*) (para una introducción a los programas

de vértices ver[6, 16]). Este programa se aplica a cada vértice que es definido en OpenGL con `glVertex`.

Este ejemplo es muy sencillo y en el mismo se está haciendo lo mismo que normalmente hace OpenGL, pero expone claramente que la modificación de los vértices y sus atributos deja de ser algo estático y se convierte en algo dinámico, en el sentido de poder ser cambiado según las necesidades del usuario.

La forma de crear y utilizar un programa de vértices es la siguiente: el usuario define el programa, la aplicación lo lee y lo compila quedando disponible para su uso, y cada vez que es necesario, se habilita para ser aplicado y se deshabilita al terminar.

Planteemos ahora un ejemplo en el que sólo se va a utilizar un programa de vértices para aplicar dos transformaciones constantes, un escalado y una rotación, en las que sólo van a cambiar el ángulo y los factores de escala. La pila estaría descrita de la siguiente forma (el número indica el orden, siendo 1 el *top*):

```
1 escalado, ángulo
0 traslación, (escala_x, escala_y, escala_z)
```

Una posibilidad sería la de compilar únicamente un programa de vértices y luego, para cada vértice, pasar como parámetros los elementos que cambian, el ángulo y los factores de escalado, calcular las transformaciones, combinarlas y aplicarlas. La segunda posibilidad consistiría en modificar el programa de vértices cada vez que se visualiza el objeto, teniendo en cuenta los nuevos valores del ángulo y los factores de escalado, y compilarlo.

Aunque la segunda solución parece mejor, a pesar de que el programa de vértices debe ser recompilado cada vez, esto se compensa ya que se aplicaría a todos los vértices. En cambio, la primera posibilidad implica aplicar repetidamente los mismos cálculos para cada uno de los vértices (dentro de esta posibilidad, también cabría hacer el cálculo de la combinación de las transformaciones y pasarlo como parámetro, pero no nos interesa). Aunque parece una mala solución, y lo es, para el caso de tener sólo transformaciones geométricas constantes, es el mecanismo perfecto para integrar las TFs. Esto es así debido a que cuando se usan transformaciones variables, según se ha expuesto anteriormente, es necesario calcular una transformación diferente para cada vértice.

5. Transformaciones Funcionales y programas de vértices

Por tanto, hemos reducido el problema del uso de las TFs a la obtención de un programa de vértices. Antes de su exposición, es necesario comentar la definición de la jerarquía de transformaciones y si la misma es constante o no.

Ya que se desea poder combinar tanto transformaciones constantes como variables, no se puede usar directamente la pila de OpenGL para mantener la jerarquía del modelo, ya que la misma sólo mantiene los valores de la matriz de transformación. Es necesario crear una nueva pila de más alto nivel, la cual mantenga el tipo y los parámetros de las transformaciones, los cuales se usarán para actualizar los parámetros que se pasan a los programas de vértices. Por ejemplo, podríamos tener un modelo cuya jerarquía quedara representada por la siguiente pila de alto nivel:

```
2 variable,no,scaling,z_axis,y_axis,...
1 variable,no,scaling,z_axis,x_axis,...
0 constant,no,translation,...
```

Mientras que la definición de las transformaciones constantes es simple, en el caso de las transformaciones variables hay que añadir el eje de selección, la función de control, etc. Para ambos tipos se incluye un campo que indica si se debe o no actualizar la lista inicial (en el ejemplo tiene el valor `no`). En este ejemplo se combinan dos transformaciones variables y una transformación constante.

La jerarquía de transformaciones puede ser constante, sin poder cambiar ni orden ni tipo ni número, o variable, pudiendo cambiar. En el primer caso, cuando el modelo es definido por el programa, la jerarquía de transformaciones es conocida, y por tanto, puede ser "compilada" para ser convertida en un programa de vértices. En el segundo caso, cuando hay cambios en la jerarquía, es necesario recompilar los programas de vértices cada vez que estos se produzcan.

5.1. Compilando las transformaciones

Según se ha comentado, es necesario un compilador que convierta la jerarquía de transformaciones en uno o varios programas de vértices. La estructura de un programa de vértices en el que se incluyan TFs es la siguiente:

```
Inicializacion
Definición de los parámetros
Definición de las variables temporales
Transformación 1
Transformación 2
Transformación ...
Transformación n
Transformación modelview
```

Es importante hacer notar que la propuesta de implementación del programa de vértices, existe una etapa para cada transformación. Aunque en el caso de tener dos o más transformaciones constantes seguidas se pueden combinar, para el caso de las transformaciones variables no siempre es posible (actualización de la lista inicial), y además complica el código (falta de instrucciones condicionales `SI ... ENTONCES ...`). En cada etapa se producen unos resultados que son pasados a la siguiente etapa.

Veamos que esta forma de operar y el uso de registros temporales resuelve también la implementación del mecanismo de las listas inicial y final. Sólo se necesitan tres registros temporales: uno se usa para mantener las coordenadas iniciales, el registro de la lista inicial (p.e. Temp2) y los otros dos registros mantienen las coordenadas finales, registros de la lista final (p.e. Temp0 y Temp1). Los registros de la lista final son necesarios para pasar los resultados de una transformación a la siguiente, de manera ordenada. Esto es, uno de ellos funciona como registro de entrada (p.e. Temp1) y el otro como salida (p.e. Temp0). Al principio Temp0 y Temp2 son inicializados con las coordenadas originales. Cada vez que comienza el conjunto de instrucciones de una transformación, el resultado de la anterior es copiado (Temp1=Temp0). El resultado de cada transformación se deja en Temp0. En caso de

que sea necesario actualizar la lista inicial con la lista final, simplemente se copia el registro de salida Temp0 en Temp1.

Por último, veamos cómo se compila cada tipo de transformación. El número de registros que pueden usarse como parámetros está limitado, así como el número de instrucciones. Para facilitar la compilación, cada clase de transformación tiene asignado un número fijo de instrucciones.

Para una transformación constante, de las clases traslación y escalado, sólo es necesario una instrucción:

```
Translate: ADD Temp0, Temp1, Input_parameter;
Scale: MUL Temp0, Temp1, Input_parameter;
```

El caso de las rotaciones es más complejo ya que no existen funciones seno y coseno. Por tanto, es necesario implementarlas mediante una aproximación usando series de Taylor. Este tipo de aproximación sólo es valida entre $-\pi$ y π. La falta de estas dos instrucciones implica el uso de varias instrucciones (aprox. 20) cada vez que un seno o un coseno son calculados.

Para las transformaciones variables las principal diferencia es la necesidad de tener en cuenta el eje de selección, el eje de aplicación y la función de control. Los ejes de selección y aplicación son implementados haciendo uso de la capacidad de enmascaramiento de las instrucciones (p.e. Temp0.x)

El intervalo de definición de las abscisas de la función de control puede ser definido bien mediante valores numéricos o bien mediante variables simbólicas relacionadas con la caja frontera del objeto(p.e. MAX_X, CENTER_Y, etc.). En el programa de vértices los valores del intervalo se toman como el punto inicial y final de una línea definida paramétricamente. El valor intermedio de esta línea, t ($0 \leq t \leq 1$), se obtiene a partir del valor de la coordenada indicada por el eje de selección. Este valor t se usa como variable independiente de la función de control para obtener el valor de la transformación. Se han implementado dos tipos de función de control: senos y curvas de Bezier 2D. Las curvas de Bezier se calculan mediante interpolaciones lineales. Para cada curva son necesarios 3 registros para pasar parámetros: uno para las abscisas y dos para las coordenadas de los puntos de control. En la actual implementación el número máximo de puntos de control es cuatro, pero es fácilmente extensible. Para las funciones seno son necesarios 2 registros: uno para las abscisas y otro para el ángulo.

Es posible implementar funciones de control bidimensionales y tridimensionales pero el número de instrucciones que son necesarias implica que el número de transformaciones que se puede definir sea muy reducido.

6. Resultados

Hemos comprobado la eficacia de los programas de vértices aplicados por la *GPU* y por la *CPU*. Para ello hemos utilizado tres modelos con distinto número de vértices y tres conjuntos de transformaciones: C1 compuesto de 2 transformaciones constantes, C2 compuesto por las transformaciones de C1 más un escalado usando una función de control senoidal, y C3 compuesto por las transformaciones de C1 más dos escalados usando funciones de control

Modelo	vértices	C1	C2	C3
hormiga	486	301.29/201.20/(1.86 %)	300.30/300.12/(2.29 %)	300.21/284.41/(3.31 %)
Betoven	2,521	174.85/174.88/(8.35 %)	174.85/174.85/(7.66 %)	174.82/146.86/(6.67 %)
coche	8,477	81.81/81.78/(23.34 %)	82.10/82.02/(-5.21 %)	81.67/62.12/(-0.69 %)

Tabla 1: Media de los valores (en fotogramas por segundo) para tres modelos, usando 4 transformaciones constantes (C1), 4 transformaciones constantes y 1 variable (C2) y 4 transformaciones constantes y dos variables (C3). Para cada caso, el primer valor es usando la *CPU* y el segundo la *GPU*. El tercer valor muestra la mejora porcentual para la segunda configuración. Los tiempos se han obtenido con imágenes de 800x800 de resolución.

senoidales. El conjunto de transformaciones de C1 equivale a hacer un cambio de sistema de coordenadas y después deshacerlo. Esto es necesario para la generación de las lentes virtuales. Para la obtención de los tiempos se han visualizado los modelos sólo en modo de alambre. Los tres modelos son un coche (Figura 1, a), Betoven (Figura 2, a), y una hormiga (Figura 3, a). En los casos b, c y d se muestran los modelos en modo sólido en conjunción con el modo de alambre.

Los resultado de la Tabla 1 se han obtenido usando un PC con un procesador Pentium IV a 2 GHz, con una tarjeta gráfica GeForce4 4200 Go, sobre Linux 2.4.21, gcc 3.2, y OpenGL. Se indican en cada caso el tiempo de la ejecución en la *CPU* y en la *GPU*. El tercer valor son los resultados obtenidos usando un procesador Pentium IV a 2.4 GHz, con una tarjeta gráfica ATI 9800 PRO. Los resultados muestran los porcentajes de mejora.

La geometría se ha dibujado usando el modo inmediato. Aunque ciertamente esta es la peor de las posibilidades para mostrar las ventajas de las GPUs, probablemente sea la condición más paritaria para realizar la comparación. Con todo, resulta destacable que las GPUs, máquinas SIMD, con un bus especial AGP, etc., no mejoren substancialmente los resultados de las CPUs (de bajo nivel, por otro lado).

7. Limitaciones

Como se ha visto anteriormente, las FTs pueden ser adaptadas muy fácilmente a las nuevas capacidades del hardware gráfico, obteniéndose unos resultados no tan buenos como cabría esperar, especialmente cuando el número de transformaciones variables comienza a crecer. Creemos que estos resultados se deben a las serías limitaciones que posee el hardware actual para los programas de vértices, limitaciones que deberán ser resueltas en el futuro. Para ello, planteamos posibles soluciones.

Una de las principales limitaciones es el reducido número de registros y, especialmente, de instrucciones. Esta ultima limitación implica que el número máximo de transformaciones tiene un límite superior, el cual depende del tipo y clase de transformaciones que se usen. Aunque se han podido producir todos los ejemplos que se han mostrado anteriormente, algunos pocos no han podido ejecutarse debido ha que tenían muchas transformaciones. Este

problema está principalmente relacionado con la falta de las funciones seno y coseno, aproximadas mediante series de Taylor, y el ajuste de los ángulos entre los límites en el que dichas series son validas. Una posible solución con el hardware actual consiste en la división de los programas de vértices en varios y reciclar los vértices después de que se aplica cada parte. Otra solución, mucho mejor pero que depende de futuras implementaciones del hardware gráfico, sería la implementación de las funciones seno y coseno. En tal caso, el número de transformaciones, dado el límite de 128 instrucciones (GeForce4, Radeon8500), sería de más de 10 (en un entorno mixto de transformaciones constantes y variables de cualquier tipo). Otra mejora en el mismo sentido sería la posibilidad de usar subrutinas, evitando la repetición de código. La disponibilidad de una instrucción condicional simplificaría los programas, haciendolos mucho mas cortos y legibles.

Otra limitación es el conjunto tan reducido de instrucciones disponibles en un programa de vértices. Un ejemplo ya comentado son las funciones seno y coseno (¡aunque estas instrucciones están disponibles para los programas de fragmentos (otra capacidad del hardware gráfico actual)!). También habría que incluir la falta de la interpolación lineal (1D, 2D y 3D; también existen para los programas de fragmentos), o el acceso rápido a la memoria local de la tarjeta gráfica (en los programas de fragmentos esto es posible con las texturas).

Una de las posibilidades más interesantes que se han comprobado con este trabajo ha sido el acceso a memoria desde los programas de vértices. Si tal posibilidad se implementara en el hardware gráfico, sería posible implementar las deformaciones (TFs, DLFs, etc.) de una manera sencilla y eficiente. En la versión actual, las funciones de control tienen que ser evaluadas para cada transformación y para cada vértice. En caso de que se pudiera usar la memoria en los programas de vértices, de la misma forma que la memoria de las texturas en los programas de fragmentos, una función de control (1D, D2, 3D o una DLF) podría ser discretizada e indizada directamente. La transformación se reduciría a calcular el valor del parámetro t y usarlo para indizar la memoria y obtener el valor que se usa en la transformación. Los mismos mecanismos, o similares, que son usados para eliminar problemas con las texturas se podrían usar con funciones que estuvieran discretizadas con un número de muestras reducido.

8. ¿*CPU* o *GPU*?

Por último, quisiéramos dar algunas ideas con respecto a la pregunta de en qué procesador se debe ejecutar cierto método, ¿en el gráfico o en el de propósito general?. En principio nos parece que el argumento de la afinidad puede servir como criterio. Por ejemplo, en el caso de las Transformaciones Funcionales, que en definitiva son transformaciones geométricas, parece que lo más lógico es que sea el procesador gráfico el responsable, con la esperanza de que además sea más rápido, debido a su especialización. Tras evaluar los resultados obtenidos, y en base al hardware gráfico actual, la respuesta no es positiva. La implementación de las TFs ha mostrado la rigidez que aún poseen los procesadores gráficos. En nuestro caso, frente a sus capacidades SIMD y de proceso vectorial, las limitaciones en el juego de instrucciones y de acceso a memoria hacen que, al menos en los programas de vértices, sea más lento que un procesador general.

Otro criterio posible es el de la máxima velocidad. Aunque en general los procedimientos gráficos se ejecutan más rápidamente en un procesador gráfico, habría que indicar que esto se produce especialmente en los procesos de bajo nivel (discretización de primitivas), y que no está tan claro en procesos de más alto nivel, como el en caso de las deformaciones. Indicar, además, que el criterio de la velocidad debería ser relativizado al del coste.

9. Conclusiones

El hardware gráfico actual ha aportado la programabilidad. En general, esta flexibilidad ha supuesto también una ganancia en velocidad. En busca de esa velocidad se ha adaptado el método de las Transformaciones funcionales a las nuevas capacidades.

Las Transformaciones Funcionales pueden ser consideradas como una herramienta genérica para la deformación de modelos geométricos, proporcionando un marco general, formalizando el uso de las deformaciones, permitiendo la creación de estructuras jerárquicas y la combinación de transformaciones constantes y variables. En comparación con las Deformaciones Libres de Forma, las Transformaciones Funcionales permiten producir el mismo tipo de deformación, pero con la característica de que se puede modular la complejidad usando funciones de control 1D o 2D cuando es posible, en vez de 3D.

Una ventaja adicional es la fácil adaptación a las capacidades del nuevo hardware gráfico, lo cual ha permitido convertir las Transformaciones Funcionales en secuencias de instrucciones de programas de vértices de manera automática con un sencillo compilador.

El hardware gráfico actual ha aportado la programabilidad. Siendo *a priori* una gran ventaja, en el caso del procesado de alto nivel, programas de vértices, la misma no se ha podido reflejar de forma contundente. Hemos mostrado que sigue estando muy limitado, ya que en el caso de los programas de vértices, las operaciones se han definido pensando en procesos de transformación básicos, no preveyendo otras posibilidades, como por ejemplo las deformaciones (Transformaciones Funcionales, Deformación Libre de Forma, etc.). Aunque el uso de otros mecanismos, que actualmente están presentes en las GPUs como los vectores de vértices, o extensiones de reciente aparición, como los *Vertex Buffer Objects* [14], deben permitir una mejora de prestaciones, probablemente algunas o bastantes de las limitaciones de diseño comentadas se irán resolviendo en próximas versiones del hardware gráfico (OpenGL 2, Direct3D 10).

Agradecimientos

Este trabajo ha sido parcialmente subvencionado por el MECT y con los fondos Feder a través de del proyecto TIC2001-2099-C03-02. Gracias a ATI por su colaboración. Gracias a los comentarios de los revisores que han permitido mejorar este trabajo.

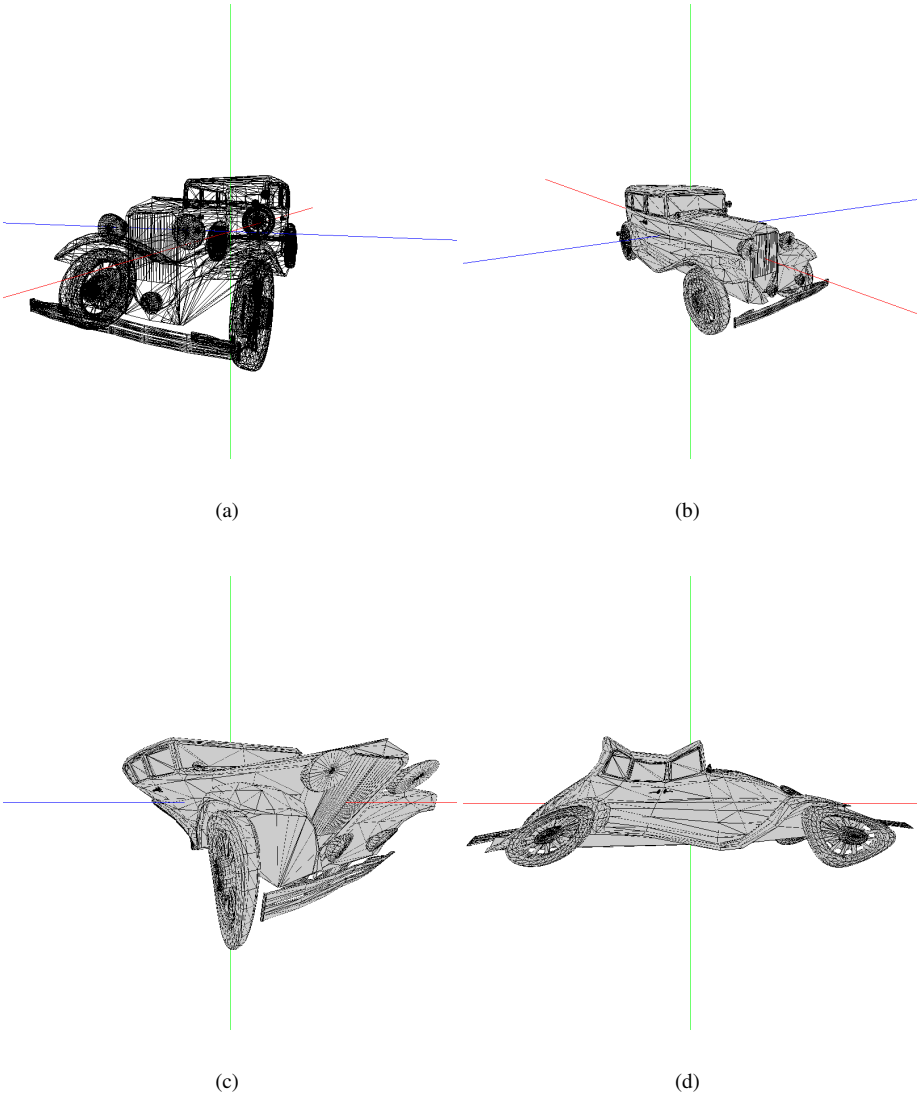

Figura 1: (a) Modelo de un coche. (b) Aplicación de C1. (c) Aplicación de C2. (d) Aplicación de C3.

Referencias

[1] A. Barr. Global and local deformations of solid primitives. *Proceedings of SIGGRAPH*, 18(3):21–30, 1984.

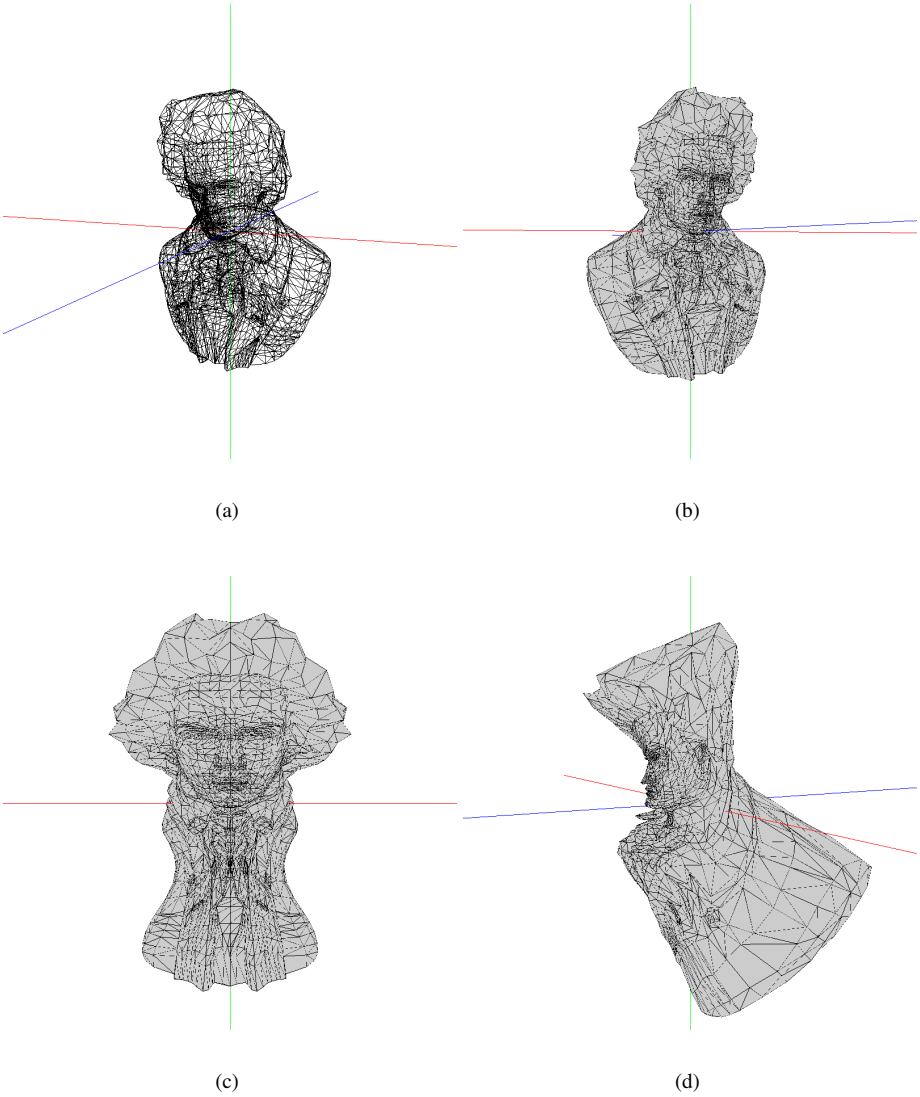

Figura 2: (a) Modelo de Betoven. (b) Aplicación de C1. (c) Aplicación de C2. (d) Aplicación de C3.

[2] S. Coquillart. Extended free form deformation: A sculpturing tool for 3D geometric modeling. *ACM Computer Graphics*, 24(4):187–196, August 1990.

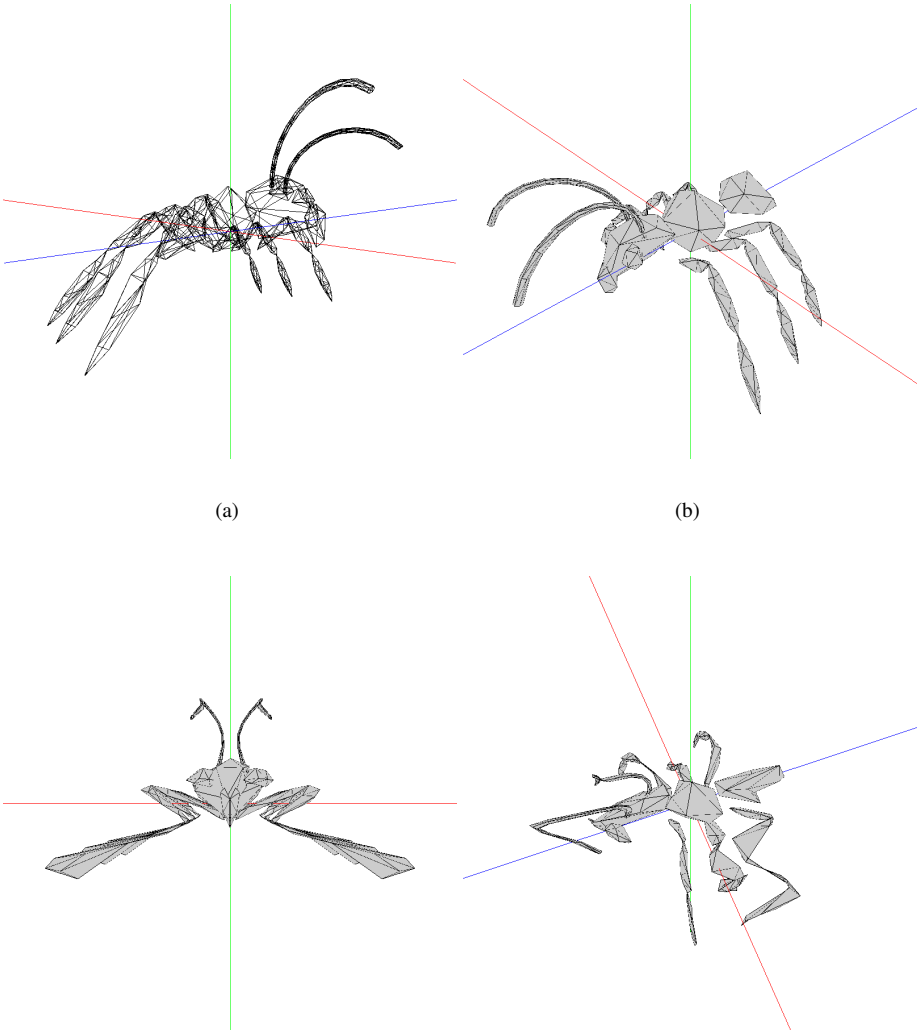

Figura 3: (a) Modelo de una hormiga. (b) Aplicación de C1. (c) Aplicación de C2. (d) Aplicación de C3.

[3] S. Coquillart and P. Jancene. Animated free form deformation: An interactive animation technique. *ACM Computer Graphics*, 25(4):23–26, July 1991.

[4] Doug L. James and Dinesh K. Pai. Dyrt: Dynamic response textures for real time deformation simulation with graphics hardware. *Proceedings of SIGGRAPH*, 21(3):582–585, July 2002.

[5] S. Karan and E. Fiume. Wires: A geometric deformation technique. *Proceedings of SIGGRAPH*, pages 405–414, 1998.

[6] Martin Kraus. Low-level vertex programmin. Technical report, Eurographics tutorial (Programming graphics hardware), 2003.

[7] Francis Lazarus, Sabine Coquillart, and Pierre Jancène. Axial deformations: An intuitive deformation technique. *Computer-Aided Design*, 28(8):607–613, 1994.

[8] Erik Lindholm, Mark J. Kildgard, and Henry Moreton. A user-programmabel vertex engine. *Proceedings of SIGGRAPH*, pages 149–158, August 2001.

[9] D. Martín. Transformaciones no-lineales jerárquicas. *9 Congreso Español de Informática Gráfica CEIG99*, pages 77–90, Junio 1999.

[10] D. Martín. Deformaciones globales dependientes del observador: Lentes sintéticas. *X Congreso español de informática Gráfica CEIG'00*, June 2000.

[11] D. Martín, S. García, and J. C. Torres. Observer dependent deformations in illustration. *Proceedings of NPAR*, pages 75–82, June 2000.

[12] D. Martín and J. C. Torres. *Alhambra*: A system for producing 2D animation. *Computer Animation 99*, pages 38–47, 1999.

[13] T. Sederberg and S. R. Parry. Free form deformation of solid geometric models. *Proceedings of SIGGRAPH 86*, 20(4):151–160, 1986.

[14] Mark Segal and Kurt Akeley. *The OpenGL Graphics System: A Specification (Version 1.5)*. 2004.

[15] A. Watt and M. Watt. *Advanced Animation And Rendering Techniques. Theory And Practice*. Addison-Wesley, 1992.

[16] Chris Wynn. Opengl vertex programming on future-generation gpus. Technical report, Nvidia, 2002.