UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

VERTEX SHADER EVALUADOR DE SUPERFICIES B-SPLINE BICÚBICAS

T E S I S

QUE PARA OBTENER EL TÍTULO DE
INGENIERO EN COMPUTACIÓN

P R E S E N T A:

FRAGOSO ROJAS VÍCTOR MANUEL

DIRECTORA DE TESIS

M.I. Norma Elva Chávez Rodríguez

23 de noviembre del 2008

Este documento representa una parte muy importante de mi vida, tan grande lo es como para dedicarla a una gran persona. Esta tésis te la dedico Saiph Savage, has sido un grandísimo apoyo durante mi desarrollo personal y profesional.

Gracias Saiph porque has sido una persona que me ha inspirado mucho y que me ha llenado mi espíritu.

Agradecimientos

Gracias a Saiph Savage por su compañía y apoyo que han sido indudablemente lo mejor que me ha pasado en mi vida.

Gracias a mis padres, por haberme apoyado y formar la persona que soy. Gracias porque me han logrado llevar lejos y por la magnifica oportunidad de tenerlos como padres.

Gracias a la familia Savage, por ayudarme siempre.

Gracias a la Universidad Nacional Autónoma de México, porque me ha dado la mejor educación, y me ha permitido vivir la mejor etapa de mi vida.

Gracias también al M. en C. Daniel Alejandro Cervantes Cabrera quien fue la persona que me introdujo al tema de mi tésis.

Tabla de Contenidos

Índice de figuras

Resumen

El pesado cálculo que se realiza para la evaluación de superficies B-Spline bicúbicas utilizando el CPU resulta muy tardado y por consecuencia no da un buen efecto visual. Sin embargo, utilizando la tecnología de las tarjetas gráficas (GPU: Graphic Processor Unit) y mediante un lenguaje de alto nivel que permita utilizar el procesador de gráficos, para mejorar el desempeño del cálculo de dichas superficies.

Este proyecto de tesis muestra como generar superficies B-Spline bicúbicas utilizando un Vertex Shader. El software desarrollado fue escrito con el lenguaje de programación C++ y con ayuda de las bibliotecas OpenGL y CG de Nvidia ©.

La applicación que se describe en este documento, es capaz de dibujar la superficie B-Spline siguiendo sus puntos de control los cuales pueden variar en tiempo real y lo que implica un redibujo de la superficie. Para lograr un mejor desempeño y facilitar el cómputo de la superficie en tiempo real se utilizó el ”framework” de Shaders de Nvidia ©llamado CG, el cual permite utilizar el GPU o el AGP de la computadora. Esto implica una carga menor para el CPU debido a que la tarea de cómputo de esta superficie se ejecuta entre el CPU y el GPU.

Capítulo 1

Curvas y Superficies Spline

1.1. Introducción

En este capítulo se explica brevemente la historia y el trabajo matemático que desarrolló Pierre Bèzier También se describirán las definiciones matemáticas de las curvas Spline enfatizando en las curvas y superficies B-Spline. Además se hará un recapitulamiento acerca de las representaciones comunes matemáticas de curvas y superficies.

Pierre Bézier fue un eminente ingeniero francés, creador de las curvas y superficies que llevan su nombre. Fue un gran personaje en el área de Diseño Asistido por computadora (CAD), debido a sus aportaciones matemáticas. Bèzier trabajó para la compañía Renault en los años de 1933 a 1975 donde desarrolló su sistema de diseño asistido por computadora llamado UNISURF CAD [13].

El objetivo que se obtiene utilizando curvas de Bèzier o las curvas Splines es generar curvas suaves las cuales pasen por algunos puntos establecidos. Estas curvas también permiten ser modificadas utilizando puntos de control, los cuales van determinando el lugar geométrico de éstas.

El uso de estas curvas y superficies en el diseño asistido por computadora se debe a que estas permiten ser modificadas generando trazos suaves dado que estas pueden ser expresadas paramétricamente lo cual facilita el cálculo por una computadora.

1.2. Curvas Paramétricas

Para estudiar las curvas Spline se requiere recordar propiedades y características de las curvas paramétricas. La razón de recapitular las curvas paramétricas es porque las curvas y las superficies Spline se expresan regularmente de forma paramétrica. Existen dos maneras comunes de representar curvas y superficies que son: ecuaciones implícitas y funciones paramétricas.

$$f(x, y) = x^2 + y^2 - 1 = 0 \tag{1.1}$$

Una ecuación implícita de una curva en el plano XY tiene la forma $f(x, y) = 0$. Esta forma describe una relación implícita entre las coordenadas x y y de los puntos pertenecientes a la curva. Un ejemplo sencillo de este tipo de ecuación se presenta en la ecuación de una circunferencia 1.1 con radio unitario y centro en el origen [6].

Una curva paramétrica en el espacio tiene la forma que muestra la ecuación 1.2:

$$\overrightarrow{f} : [0, 1] \rightarrow (f(u), g(u), h(u)) \tag{1.2}$$

donde $f(u)$, $g(u)$ y $h(u)$ son tres funciones las cuales tienen como contradominio el conjunto de los números reales [14]. La función $\overrightarrow{f}$ mapea el parámetro u el cual varía en el intervalo $[0, 1]$ a un punto en el espacio. El dominio de la función $\overrightarrow{f}$ no necesariamente tiene que ser el intervalo $[0, 1]$, sino que también puede ser cualquier intervalo cerrado. Los intervalos suelen ser de $[0, 1]$ debido a que se prefiere tener la variable normalizada.

La definición de la función $\overrightarrow{f}$ en este ejemplo se definió para un espacio de dimensión tres. Sin embargo, se puede extender la definición a cualquier espacio. Por ejemplo, si quitamos la función $h(u)$ de la definición de la función, obtendríamos una función que mapea a un punto en el espacio, el cual pertenece a uno de dimensión dos.

$$x(u) = r \cdot \cos(2\pi u) + p \tag{1.3}$$

$$y(u) = r \cdot \sin(2\pi u) + q \tag{1.4}$$

Para cerrar esta subsección es conveniente hacerlo con un ejemplo. Supongamos que se requiere expresar una circunferencia en un espacio de dimensión dos paramétricamente. Sabemos que una circunferencia siempre tiene como propiedad un centro (p, q), y también tiene como característica un radio, el cual será denotado por r. La ecuación de una circunferencia puede ser expresada usando funciones trascendentes como lo muestran las ecuaciones 1.3 y 1.4:

$$\overrightarrow{f}(u) = (x(u), y(u)), \forall u \in [0, 1] \tag{1.5}$$

En donde x y y de dichas ecuaciones 1.3 y 1.4 , corresponden a las coordenadas de un punto en el plano XY. De tal forma que se puede expresar una función $\overrightarrow{f}$ tal que como resultado de evaluar la función se obtenga un punto en el espacio, el cual pertenezca a la circunferencia, por lo tanto, una circunferencia queda descrita paramétricamente como lo muestra la ecuación 1.5.

Esta función paramétrica* que describe el lugar geométrico de la circunferencia 1.5, evalúa el parámetro u con rango $[0, 1]$ y esta regresa como resultado un punto en el espacio, el cual pertenece a una circunferencia con centro (p, q) y radio r. De tal manera que si se recorre todo el rango parametral de u, se obtendrán todos los puntos de la circunferencia, lo que para un programador suena maravilloso, dado que se tiene control sobre la evaluación de algún lugar geométrico con solamente un parámetro.

Es importante reafirmar que estas funciones anteriormente mostradas tienen como dominio una variable real, la cual es el parámetro, y como contradominio tienen un conjunto de puntos en el espacio. Cabe mencionar también que la representación de curvas paramétricamente tiene ciertas ventajas como lo menciona Piegl en [6] y que se muestran a continuación:

- La forma paramétrica es más natural para el diseño y representación en una computadora.
- Se obtiene una mejor estabilidad en los algoritmos y en el diseño de métodos intuitivos.

*Es importante aclarar que no es la única forma de representar la circunferencia paramétricamente.

Figura 1.1: Segmento de Recta

1.3. Curvas de Bèzier

1.3.1. Algoritmo de Casteljau

Para comenzar a estudiar estas curvas se necesita comprender sencillamente el propósito que estas tienen. Estas curvas necesitan de puntos definidos por el usuario los cuales estén contenidas en estas curvas, en otras palabras, necesitan de puntos por los cuales la curva ”pase” o que la curva generada aproxime pasar por ellos. Se puede ver este problema como un problema de interpolación, donde se tienen puntos por los cuales se desea que algún lugar geométrico contenga aquellos puntos que se definan.

El algoritmo de Casteljau presenta una forma de realizar el proceso descrito anteriormente. El objetivo fundamental del algoritmo es tomar un punto C de un segmento de recta AB tal que el punto C divida al segmento en una razón $u : 1 - u$, con u menor que 1, como lo muestra la figura 1.1. La razón puede ser la distancia que existe entre el punto A y C y la distancia que existe entre el punto A y B.

$$C = u(B - A) + A \tag{1.6}$$

$$C = (1 - u)A + uB \tag{1.7}$$

El punto C se puede describir como lo muestra la ecuación 1.6 la cual toma como referencia al punto A. Si se trabaja la ecuación 1.6, podemos llegar a la ecuación 1.7 si se factoriza A, la cual muestra claramente una combinación entre los puntos A y B con ciertos coeficientes diferentes.

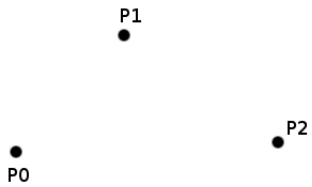


Figura 1.2: Puntos de Control.

Para ejemplificar el algoritmo con un ejemplo sencillo se propone el siguiente esquema. Se tienen dos puntos, como se expone en [4], por los cuales se pretende que la curva a generar ”pase”, y se tiene un tercer punto del cual la curva ”tira”, como se muestra en la figura 1.2.

Los puntos que se unirán serán el punto P_0 con el punto P_1 y después el punto P_1 con el punto P_2. Si cada segmento lo dividimos encontrando el punto correspondiente a la razón escogida obtemenos las ecuaciones siguientes:

$$P_0^1 = (1-u)P_0 + uP_1 \tag{1.8}$$

$$P_1^2 = (1-u)P_1 + uP_2 \tag{1.9}$$

tal y como se muestra en la figura 1.3. Los puntos amarillos en los dos segmentos de recta, representan los puntos encontrados P_0^1 y P_1^1 con cierto parámetro u.

Si se repite el proceso con los dos puntos nuevos que obtuvimos, es decir unir los puntos P_0^1 y P_1^2, obtenemos la ecuación 1.10:

$$P_0^2 = (1-u)P_0^1 + uP_1^1 \tag{1.10}$$

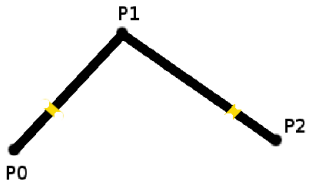


Figura 1.3: Puntos determinados por el algoritmo de Casteljau.

El punto P_0^2 pertenece a la curva que generaremos, que es una curva del tipo Spline que más adelante se explicará. Por lo tanto si u, el parámetro, se va variando en todo su dominio y se repite el proceso anterior para cada valor de u, se generarán todos los puntos de la curva producida por este algoritmo.

Si se juntan las ecuaciones 1.8 y la ecuación 1.9 en la ecuación 1.10, obtenemos una ecuación en función del parámetro y los puntos de control.

$$\overrightarrow{C}(u) = (1-u)[(1-u)P_0 + uP_1] + u[(1-u)P_1 + uP_2] \tag{1.11}$$

$$\overrightarrow{C}(u) = (1-u)^2 P_0 + 2u(1-u)P_1 + u^2 P_2 \tag{1.12}$$

Si se analiza la ecuación 1.12 se observa que esta ecuación es cuadrática, también se observa que para $u = 0$ la función es igual a P_0 y para $u = 1$ la función es igual a P_2; por lo tanto el algoritmo de Casteljau genera una curva la cual une los extremos, en este ejemplo anteriormente mostrado, los puntos P_0 y P_2, mientras que el punto P_1 ”jala” la curva hacia el. A esta propiedad se le conoce como **Interpolación en los extremos de la curva**.

Esta curva que se generó es cuadrática como lo muestra le ecuación 1.12 y se dice que es una curva Spline de grado dos y de orden tres, dado que tenemos tres puntos de control. Al conjunto de los puntos P_0, P_1 y P_3 se les conoce como el **polígono de control**. Estos puntos son los responsables

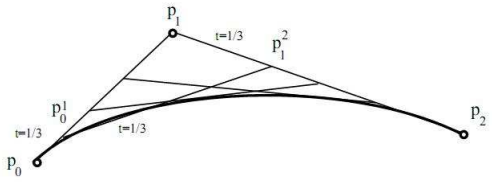


Figura 1.4: Poligono de Control, con algunos puntos calculados mediante el Algoritmo de Casteljau.

de la forma resultante de la curva.

En la figura 1.4 observamos el polígono de control para esta curva. En el caso de las curvas de Bèzier, estas siempre están dentro del polígono de control, por lo que este polígono recibe el nombre de **polígono envolvente** que en inglés se le conoce como la propiedad de **Convex Hull**. Existen algunas otras propiedades más que poseen estas curvas y que se pueden consultar en el primer capítulo del libro escrito por Piegl [6]. Es importante señalar que esto es válido si los puntos de control son cooplanares. Si no lo son, la propiedad se mantiene estando en este caso debajo del poliedro, el cual se forma a partir de los puntos de control.

1.3.2. Polinomios de Bernstein

En la subsección anterior se presentó un ejemplo en el cual se genera una curva de comportamiento suave con el algoritmo de Casteljau. También se presentó una ecuación paramétrica de la curva, la cual se conoce como **la forma de Bernstein de una curva de Bèzier**.

En las ecuaciones 1.10 y 1.12 se observa que los términos que aparecen multiplicando a los diferentes puntos de control, coinciden con los términos resultantes de aplicar el teorema del binomio a la suma $[u + (1 - u)]$, como se demuestra en [4].

$$[u + (1 - u)]^n = \sum_{i=0}^{n} C_i^n u^i (1 - u)^{(n-1)} = \sum_{i=0}^{n} B_i^n(u) \tag{1.13}$$

$$B_i^n(u) = C_i^n u^i (1 - u)^{(n-1)} \tag{1.14}$$

$$B_i^0(u) = 1 \tag{1.15}$$

De tal manera que los coeficientes que aparecen multiplicando a los puntos de control, son simplemente muestras de los polinomios de Bernstein $B_i^n(u)$, los cuales están definidos como los muestra la ecuación 1.14 y por su excepción como lo marca la ecuación 1.15. Estos polinomios, de grado n, forman una base para el espacio vectorial de polinomios de grado n [21].

La obtención de la base es simple, por ejemplo, supongamos que se quiere obtener una base polinomial para generar el espacio de polinomios de dimensión 1, lo que implica $n = 0$. De tal forma que si se sustituye en la ecuación 1.15 se obtiene $B_i^0(u) = 1$, tal y como se muestra en la figura 1.5

Si ahora se quisiera obtener una base para generar un espacio de polinomios de grado $n = 1$, simplemente se sustituye en la ecuación 1.14 y se evalúa, el resultado que se obtiene es $B_0^1(u) = u$ y $B_1^1(u) = 1 - u$, tal y como lo muestra la figura 1.5. En la figura mencionada se muestran las diferentes familias de polinomios generadas con la definición de los polinomios de Bernstein para

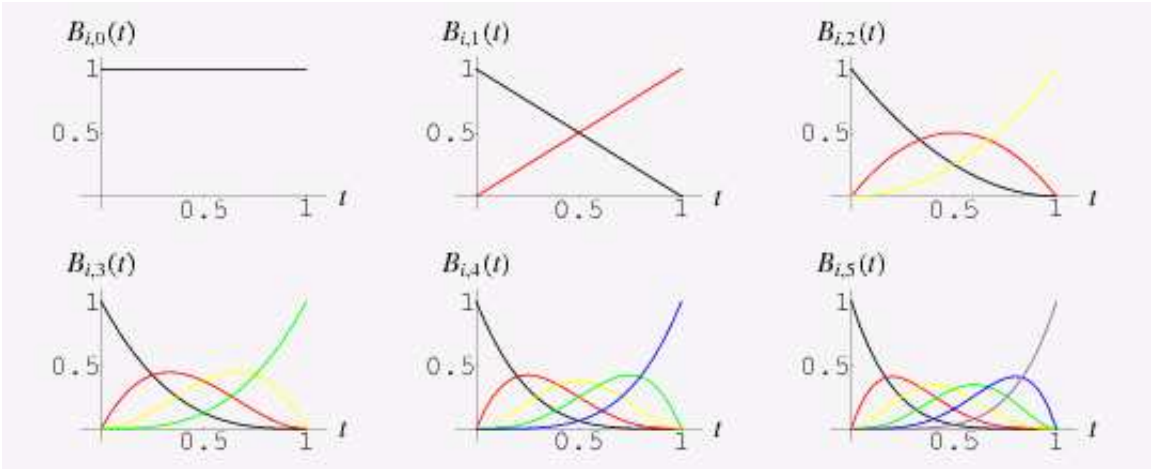


Figura 1.5: Bases polinomiales. Berntsein.

distintos grados.

Debido a que se pueden generar los coeficientes, utilizando la ecuación de Bèzier, se puede expresar entonces la curva obtenida en la subsección anterior como lo muestra la ecuación 1.16:

$$\overrightarrow{C}(u) = \sum_{i=0}^{n} B_i^n(u) \cdot \overrightarrow{P_i} \tag{1.16}$$

donde $\overrightarrow{P_i}$ es el i-ésimo punto de control y la i es el índice que varía de 0 hasta n, lo que implica que se necesitan $n + 1$ puntos de control. La ecuación 1.13 define a los polinomios de Bernstein, los cuales ayudan a definir de una manera muy elegante las curvas de Bèzier. Es importante resaltar que esta ecuación utiliza puntos en el espacio, lo cual implica que no necesariamente los puntos deben ser cooplanares.

Otra representación que probablemente sea útil de esta ecuación es la representación matricial. Para llegar a una expresión matricial a partir de esta ecuación debemos expandir la sumatoria tal y como se muestra en la ecuación:

$$\overrightarrow{C}(u) = B_0^n(u)\overrightarrow{P_0} + B_1^n(u)\overrightarrow{P_1} + ... + B_i^n(u)\overrightarrow{P_i} \tag{1.17}$$

De esta última ecuación 1.17, se puede expresar esa suma como el producto de matrices entre los coeficientes de los polinomios de Bernstein y el conjunto de puntos de control, por lo tanto:

$$\overrightarrow{C}(u) = \mathbf{B} \cdot \mathbf{P} \tag{1.18}$$

$$\mathbf{B} = \begin{bmatrix} B_0^n(u) & B_1^n(u) & \dots & B_i^n(u) \end{bmatrix}$$

$$\mathbf{P} = \begin{bmatrix} \overrightarrow{P_0} \\ \overrightarrow{P_1} \\ \vdots \\ \overrightarrow{P_i} \end{bmatrix}$$

Esta expresión matricial 1.18, maneja escalares, los cuales son representados por la matriz **B**, que son los coeficientes de los polinomios de Bernstein, mientras que la otra matriz **P** contiene puntos en el espacio, o los puntos de control. El producto entre estas dos matrices, finalmente es una combinación lineal de los puntos de control, escalados con un factor determinado por el coeficiente de Bernstein.

Si se analiza esta expresión matricial, se puede deducir fácilmente que la evaluación de un punto de la curva de Bèzier presenta una complejidad de O(n), dado que para evaluar un punto necesitamos hacer una combinación lineal que involucra a todos los n puntos de control.

El conocer las curvas de Bèzier facilitará mucho más la comprensión y la generación de las superficies de Bèzier, dado que las superficies de Bèzier se generan utilizando los mismos conceptos matemáticos que para generar las curvas.

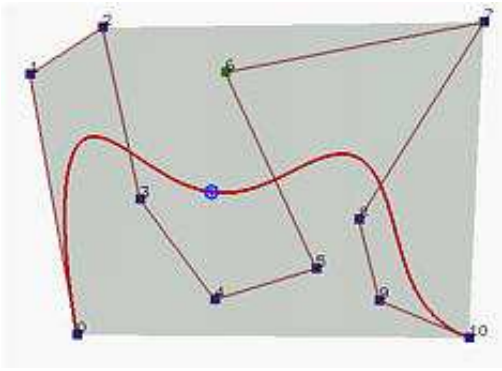


Figura 1.6: Curva de Bèzier.

1.4. Superficies de Bèzier

En esta subsección se explica como generar superficies de Bèzier, las cuales también reciben el nombre de parches dado que se pueden generar superficies complejas juntando diferentes superficies de Bèzier. Las superficies de Bèzier se describen mediante una función paramétrica la cual tiene como argumentos dos parámetros los cuales varían independientemente.

Una superficie representada paramétricamente tiene la forma como lo muestra la ecuación 1.19.

$$\overrightarrow{f}(u,v) = (x(u,v), y(u,v), z(u,v)) \tag{1.19}$$

Para nuestros fines se asumirá que los parámetros u y v varían en el rango de $[0, 1]$, es decir que están normalizados. Si se analiza la ecuación 1.19 se observa que la función $\overrightarrow{f}$ mapea la tupla (u, v) a un punto perteneciente a la superficie tal y como lo muestra la figura 1.7.

Una superficie paramétrica, de este tipo, puede considerarse como la unión de un número infinito de curvas [15]. La forma más simple de unir estas curvas es la de **curvas isoparamétricas**. Para comprender esta unión de curvas supongamos que se tiene una función $\overrightarrow{f}(u, v)$, supongamos también que el valor del parámetro u es fijo y que el valor del parámetro v es variable. Este esquema genera una curva sobre la superficie en donde su parámetro u es constante y la curva se extiende sobre la dirección donde varía el parámetro v. De igual forma se puede fijar el valor del parámetro v

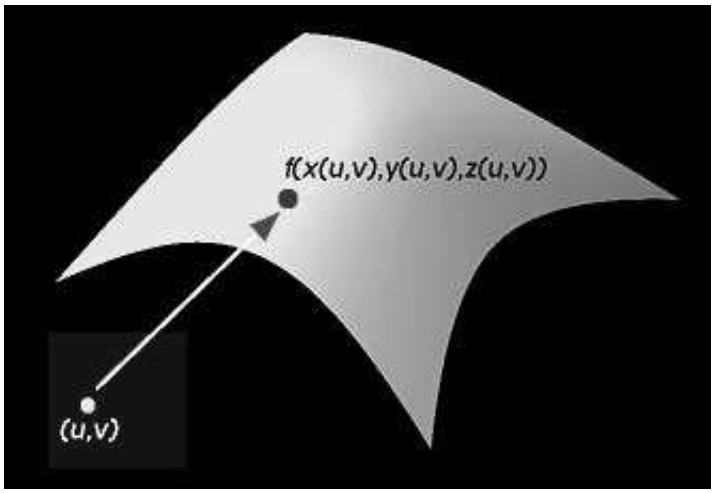


Figura 1.7: Mapeo de la función f.

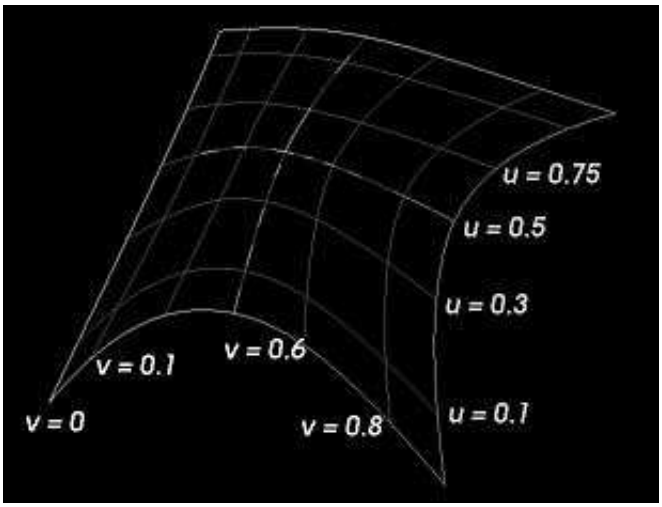


Figura 1.8: Curvas Isoparamétricas.

y dejar el parámetro u variar, esto implicaría una curva sobre la superficie en la dirección u con un valor constante para el parámetro v. Un ejemplo de este esquema se presenta en la figura 1.8.

$$\overrightarrow{S}(u,v) = \sum_{i=0}^{m}\sum_{j=0}^{n} B_i^m(u)B_j^n(v)\overrightarrow{P_{ij}} \tag{1.20}$$

La definición de las superficies de Bèzier es la mostrada en la ecuación 1.20. Donde u y v son nuestros parámetros, m es el grado de la curva isoparamétrica en la dirección u, mientras que n es el grado de la curva isoparamétrica en la dirección v, i y j son los índices y $\overrightarrow{P_{ij}}$ son los puntos de control.

Esta definición representa a la superficie como el producto entre dos curvas. A esta construcción de superficies se le conoce como **Superficies de Producto Tensorial** [6]. El procedimiento consiste en multiplicar funciones bases de la primer curva por las funciones bases de la segunda curva y el producto se usa como la función base para un conjunto de puntos de control.

De la ecuación 1.20, mediante algunos movimientos algebráicos, se pueden observar las curvas isoparamétricas. Se nota que los coeficientes $B_i^m(u)$ no dependen del índice j, por lo que nos es posible extraer ese factor de la suma, de tal forma que la ecuación queda como lo muestra la ecuación 1.21.

$$\overrightarrow{S}(u,v) = \sum_{i=0}^{m} B_i^m(u)\sum_{j=0}^{n} B_j^n(v)\overrightarrow{P_{ij}} \tag{1.21}$$

Si se extrae la suma más interna de la ecuación 1.21, es decir $\sum_{j=0}^{n} B_j^n(v)\overrightarrow{P_{ij}}$, notamos que esta suma es una curva de Bèzier en la dirección v para un cierto valor de u. De esta manera podemos identificar que efectivamente la superficie de Bèzier se conforma de curvas isoparamétricas.

De igual manera que con las curvas, es posible representar la definición de las superficies de Bèzier matricialmente, siguiendo el mismo procedimiento que se utilizó para las curvas de Bèzier. Para fines prácticos se presenta la expresión matricial de estas superficies en la ecuación 1.22.

$$\overrightarrow{S}(u,v) = \mathbf{B_i} \cdot \mathbf{P} \cdot \mathbf{B_j} \tag{1.22}$$

$$\mathbf{B_i} = \begin{bmatrix} B_0^n(u) & B_1^n(u) & \dots & B_i^n(u) \end{bmatrix}$$

$$\mathbf{P} = \begin{bmatrix} P_{00} & P_{01} & \dots & P_{0j} \\ P_{10} & P_{11} & \dots & P_{1j} \\ \vdots & \vdots & \dots & \vdots \\ P_{i0} & P_{i1} & \dots & P_{ij} \end{bmatrix}$$

$$\mathbf{B_j} = \begin{bmatrix} B_0^m(v) \\ B_1^m(v) \\ \vdots \\ B_j^m(v) \end{bmatrix}$$

Para cerrar esta subsección se presentan imágenes 1.9, 1.10 de una superficie de Bèzier vista desde diferentes ángulos. Con los puntos de control en Blanco y los puntos generados en color Rojo.

1.5. Curvas Spline

Para generar curvas o superficie con mayor detalle utilizando curvas o superficies de Bèzier se necesita un grado polinomial muy alto. El hacer esto tiene como algunas desventajas que Piegl [6] menciona en su capítulo 2 *B-Spline functions* y que se citan a continuación:

- El tener una curva con alto grado es difícil de procesar y además numéricamente inestable.
- Para generar formas complejas se necesita un grado polinomial alto.
- Aún cuando las curvas o superficies de Bèzier pueden ser controladas en su forma a partir de sus puntos de control, estas no presentan un control local. Esto significa que si se altera un

Figura 1.9: Superficie de Bèzier bicúbica.

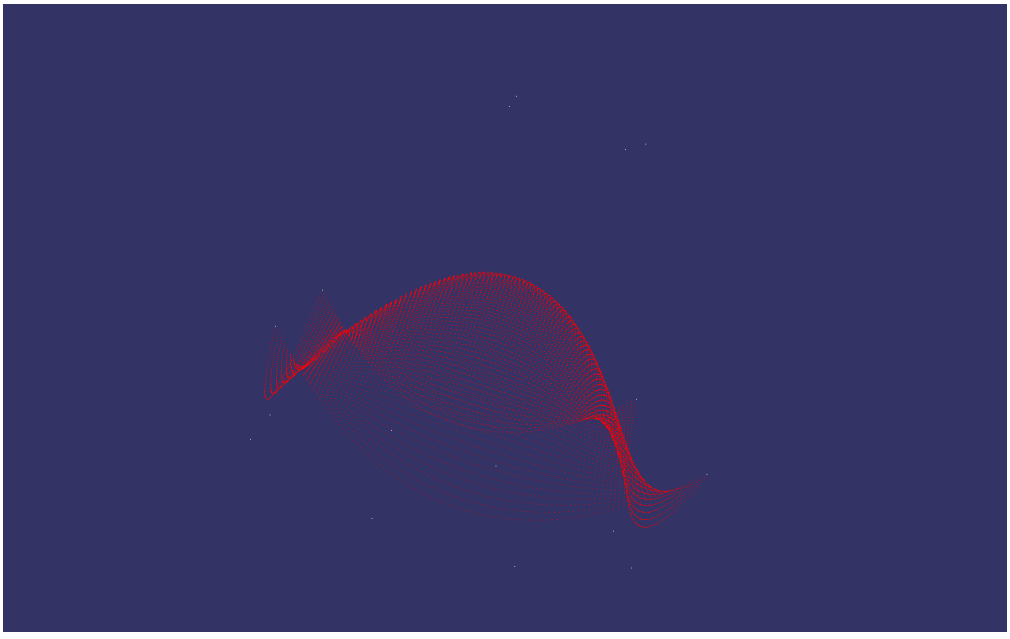


Figura 1.10: Superficie de Bèzier bicúbica.

punto de control, este puede afectar la forma de toda la superficie.

Por estas razones, se describe en esta sección a las curvas Spline, dado que pueden ser utilizadas para ganar control local y reducir un alto grado polinomial. Una curva Spline en resumen es una función polinomial a pedazos [23]. En su forma más general podemos escribir una función Spline como lo muestra la ecuación 1.23:

$$\mathbf{S(t)} = \begin{cases} P_0(t) & \text{si} \quad t_0 < t < t_1 \\ P_1(t) & \text{si} \quad t_1 < t < t_2 \\ P_2(t) & \text{si} \quad t_2 < t < t_3 \\ \vdots & \\ P_{k-2} & \text{si} \quad t_{k-2} < t < t_{k-1} \end{cases} \tag{1.23}$$

El vector $t = (t_0, t_1, \ldots, t_{k-1})$ es llamado el vector de nodos, dado k puntos de control. Si los nodos son equidistantes entre sí, entonces se dice que la Spline es **uniforme**, de otra manera se dice que es **no uniforme**. Si alguno de los polinomios, que constituyen la curva, tiene el grado n más grande que los demás grados de los demás polinomios, entonces la curva recibe el grado de menor o igual a n o de orden $n + 1$.

Dada la definición de curvas Spline y si se recuerda a las curvas de Bèzier podemos entonces construir curvas Spline juntando distintas curvas o parches de Bèzier, ya que como anteriormente se demostró, una curva de Bèzier es una curva polinomial. Existen algunas consideraciones que se deben tomar para realizar la unión de diferentes parches de Bèzier.

Dos curvas de Bèzier se pueden unir siempre y cuando la primera y última arista, del polígono de control de cada una de las curvas, tengan la misma dirección. Gracias a esta condición, se puede asegurar una condición G^1 de continuidad dado que los vectores tangenciales tienen la misma dirección pero no la misma longitud [16].

La primer pregunta que podría surgir es la siguiente, habrá alguna manera de construir curvas Spline, de tal forma que no tengamos que preocuparnos por la condición de continuidad entre curvas de Bèzier. Para nuestra fortuna, existe una manera de realizar esta unión, la cual consiste en utilizar

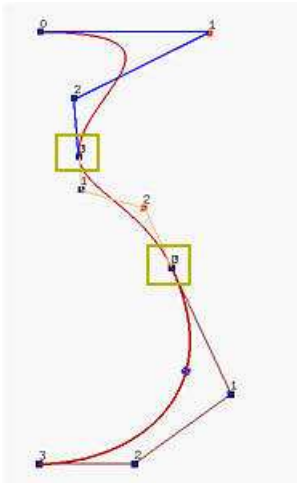


Figura 1.11: Curva Spline. Esta curva se constituye de 3 curvas de Bèzier. Enmarcando en amarillo las uniones de las curvas.

las curvas B-Spline, las cuales son una generalización de las curvas de Bèzier.

Existen distintos algoritmos para calcular curvas o superficies Splines, en este trabajo de tesis solamente se presenta alguno de ellos, en especial aquél que por su simpleza es claro y fácil de comprender. Una buena referencia si es que el interés por conocer más algoritmos es grande, se pueden encontrar distintos algoritmos en el libro de Späth [19].

1.6. Curvas B-Spline

1.6.1. Base B-Spline

Las curvas B-Spline son una generalización de las curvas de Bèzier, estas curvas responden a la pregunta que se planteó en la sección anterior, la cual cuestionaba si existe alguna manera de unir curvas de Bèzier sin preocuparse por la condición de continuidad. Los polinomios de Bernstein eran utilizados en las curvas de Bèzier como pesos para los puntos de control, en las curvas B-Spline, se utilizará la base B-Spline para el mismo propósito. Cabe mencionar que esta base es más compleja que la base utilizada en las curvas de Bèzier.

Existen dos propiedades importantes [17] que podemos mencionar antes de exponer la definición, las cuales no se deben perder de vista acerca de estas bases:

- El dominio de las bases está subdividido por nodos
- La Base B-Spline puede ser cero en un intervalo

A continuación se explicará el vector de nodos que se utiliza junto con esta base. Sea **U** el vector de nodos $u_0 \leq u_1 \leq u_2 \leq \ldots \leq u_{m+1}$. Los elementos u_i reciben el nombre de **nodos**, al conjunto de los nodos se le conoce como el **vector de nodos**, y cada $u_i \leq u_{i+1}$ recibe el nombre del i-ésimo intervalo. Es importante notar que algunos intervalos no existen dado que podemos repetir nodos en el vector, debido a que los nodos pueden tener multiplicidad, es decir que se repitan más de una vez en el vector.

El propósito de los vectores de nodos es subdividir la curva en varias, de tal manera que podamos ganar control local de cada división. Por lo tanto, una curva B-Spline será dividida según el vector de nodos utilizado. Se puede visualizar esta idea en la figura 1.12. Los puntos amarillos en la curva, representan los puntos en donde se unen dos curvas de Bèzier, y las líneas punteadas en verde proyectan a un eje. La intersección de estas líneas con el eje representan a un nodo, pertenenciente al vector de nodos.

En resumen, para diseñar una curva B-Spline, se necesita un conjunto de nodos y un conjunto de coeficientes, uno por cada punto de control, de tal forma que todos los segmentos de la curva estén unidos cumpliendo con la condición de continuidad. El cálculo de los coeficientes es en parte el paso más complejo dado que deben de garantizar la continuidad en las uniones.

Continuando con la descripción de los vectores de nodos se presentan a continuación ciertos tipos de vectores. Como se había mencionado anteriormente, si los nodos del vector son equidistantes entre sí, este recibe el nombre de vector uniforme, de otra manera es un vector no uniforme.

Para poder definir la base B-Spline se necesita de un parámetro u, el grado de la curva p y el vector de nodos U. La base B-Spline se define de manera recursiva como lo muestra la ecuación 1.24

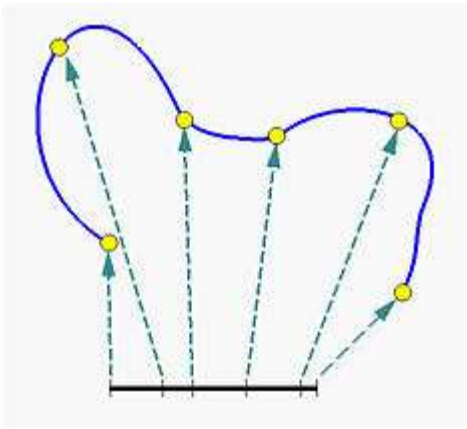


Figura 1.12: Curva Spline dividida en el dominio del Vector de Nodos.

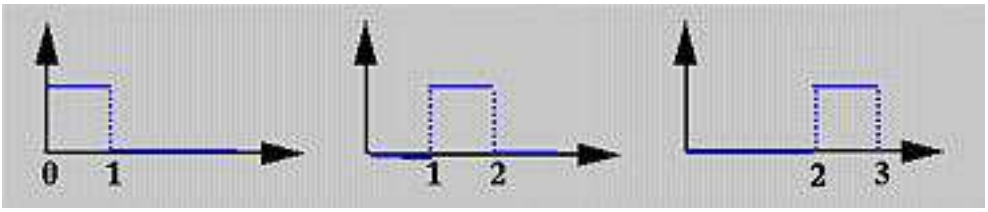


Figura 1.13: Funciones escalón para $N_i^0(u)$

$$N_i^0(u) = \begin{cases} 1 & \text{si} \quad u_i \leq u < u_{i+1} \\ 0 & \text{si} \quad \textit{de otra manera} \end{cases} \tag{1.24}$$

$$N_i^p(u) = \frac{u - u_i}{u_{i+p} - u_i} N_i^{p-1}(u) + \frac{u_{i+p+1} - u}{u_{i+p+1} - u_{i+1}} N_{i+1}^{p-1}(u)$$

A la ecuación 1.24 se le suele llamar la fórmula recursiva de **Cox - de Boor**. Si se analiza esta ecuación, notamos que para el cálculo de $N_i^0(u)$ se tiene una familia de funciones escalón para el intervalo $u_i \leq u < u_{i+1}$ sólo si u está en el intervalo, tal y como lo muestra la figura 1.13. Mientras que el análisis para el cálculo de $N_i^p(u)$ con $p > 0$ es recursivo, por lo que se puede generar una especie de árbol o un esquema triangular tal y como lo muestra la figura 1.14.

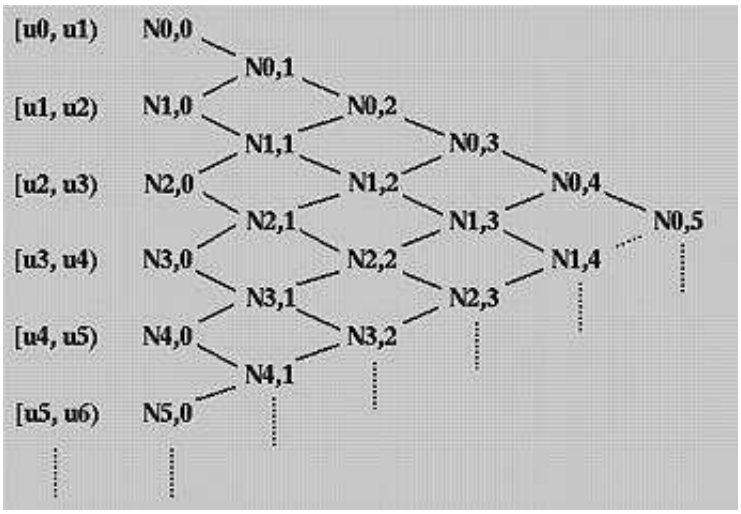


Figura 1.14: Esquema triangular para el cálculo de $N_i^p(u)$

1.6.2. Propiedades de la Base B-Spline

Algunas propiedades importantes que se mencionan en [18] acerca de la base B-Spline son las siguientes:

- $N_i^p(u)$ es un polinomio de grado p en u.
- Para toda i, p y u, $N_i^p(u)$ es positivo.
- Soporte Local: $N_i^p(u)$ es un polinomio distinto de cero sobre el intervalo $[u_i, u_{i+p+1})$
- En cualquier intervalo $[u_i, u_{i+p+1})$, existen a lo máximo $p + 1$ funciones de grado p distintas de cero.
- La suma de las funciones distintas a cero de grado p en un intervalo $[u_i, u_{i+p+1})$ es uno.
- Si se tienen $m + 1$ nodos, el grado es p y el número de funciones base de grado p es $n + 1$, entonces $m = n + p + 1$.
- La base $N_i^p(u)$ es una curva polinomial de grado p con puntos de unión en el intervalo $[u_i, u_{i+p+1})$.

- En un nodo con multiplicidad k, la función base $N_i^p(u)$ tiene un nivel de continuidad de C^{p-k}. En otras palabras, si la multiplicidad de un nodo aumenta, el grado de continuidad de la curva disminuye.

1.6.3. Multiplicidad de un nodo

La multiplicidad de un nodo [18] tiene una repercusión importante en el cómputo de la base. Pero aún más importante tiene una repercución en la trayectoria que la curva toma. El que un nodo tenga multiplicidad implica dos cosas importantes:

- Cada nodo de multiplicidad k reduce a lo más el dominio que no es cero de $k - 1$ funciones base.
- En cada intervalo de multiplicidad k, el número de funciones base que no son cero es a lo máximo $p - k + 1$, donde p es el grado de la función base.

El efecto que esta subsección tiene como propósito mostrar es el efecto visual que tiene la multiplicidad de un nodo. La multiplicidad provoca la tendencia a contener el punto de control al cual le corresponde el intervalo en el que el nodo tiene su multiplicidad.

En la figura 1.15 se muestran dos curvas B-Spline. La curva de la izquierda no contiene en el vector de nodos multiplicidad. Notamos que las bases B-Spline, las que se encuentran a la derecha de la curva, tienen forma suave y no presentan alteración dado que no hay multiplicidad de algún nodo. La curva de la derecha, tiene multiplicidad en un nodo, lo cual implica cambio en la base B-Spline que se encuentra a su derecha, notamos como la curva resultante tiende a contener al punto de control y también notamos como la base se modifica.

1.6.4. Curva B-Spline

La definición de la curva B-Spline require de ciertos parámetros extra que para las curvas de Bèzier no son necesarios. Dados $n + 1$ puntos de control y un vector de nodos $\mathbf{U} = \{\mathbf{u_0}, \mathbf{u_1}, \ldots, \mathbf{u_m}\}$, la curva B-Spline de grado p se define como lo muestra la ecuación 1.25. Donde $N_i^p(u)$ es la base

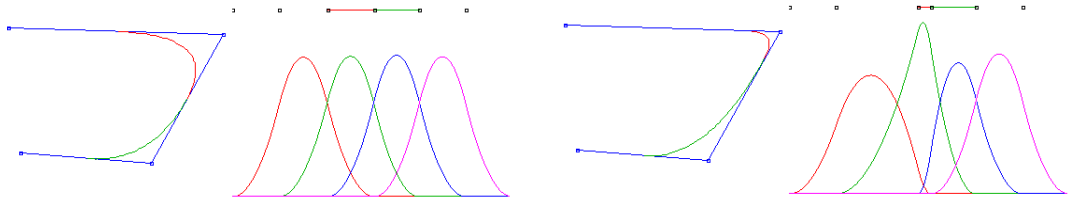


Figura 1.15: Efecto visual de la multiplicidad de un Nodo

B-Spline de grado p. Existe una relación, la cual determina el tamaño del vector de nodos que se necesita para construir la curva. Dado el grado de la cuva p, y con $n+1$ puntos de control, entonces se necesita un vector de nodos de tamaño $m = n + p + 1$.

$$\overrightarrow{C}(u) = \sum_{i=0}^{n} N_i^p(u)\overrightarrow{P}_i \tag{1.25}$$

El punto $\overrightarrow{C}(u_i)$ correspondiente al nodo u_i se le llama **Punto Nodo**, los cuales subdividen a la curva, como lo muestran los puntos amarillos de la figura 1.12. Estos puntos se encuentran separados de acuerdo al intervalo en el vector de nodos. Cada segmento de la curva, entre un punto nodo y otro, es una curva de Bèzier de grado p.

Si se analiza la ecuación 1.25, notamos que la definición es muy similar a la definición de las curvas de Bèzier. El grado p en las curvas B-Spline puede considerarse como una entrada necesaria para el cálculo de esta curva, mientras que el grado de la curva de Bèzier depende del número de puntos de control.

El vector de nodos es tan importante que incluso si este no tiene una estructura en particular, la curva generada no contendrá al primer punto ni así como el último punto de control. Este tipo de curvas reciben el nombre de **B-Spline flotantes**. También existe la manera de hacer que estos puntos se incluyan en la curva. Para lograr esto se debe tener el primer y último nodo con multiplicidad $p + 1$, esto significa repetir el nodo $p + 1$ veces en el vector. A este tipo de curvas generadas con multiplicidad en los extremos se les conoce como **B-Spline abiertas**. La figura 1.16 muestra los

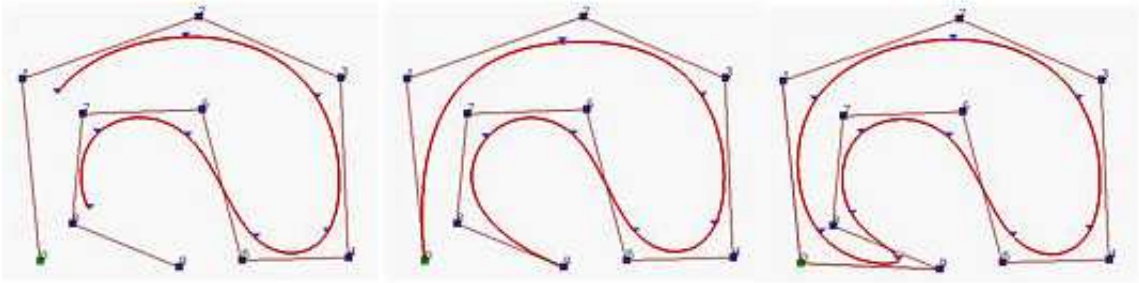


Figura 1.16: B-Spline flotante, abierta y periódica.

tres tipos de curvas B-Spline, el tercer tipo de curva es una curva cerrada y recibe el nombre de **B-Spline periódica**.

1.6.5. Algoritmo de Boor

El algoritmo de Boor [22] propone una nueva expresión para generar curvas B-Spline de una manera recursiva y aprovecha la propiedad de localidad de las curvas B-Spline. En pocas palabras la propiedad de localidad se refiere a que se pueda modificar la posición de algun punto de control y como consecuencia de este movimiento solamente repercuta en la forma de cierta parte de la curva o superficie. De tal forma que se puede expresar a la curva utilizando solamente la propiedad de localidad como lo muestra la ecuación 1.26.

$$\overrightarrow{C}(u) = \sum_{i=l-p}^{l} N_i^p(u)\overrightarrow{P_i} \tag{1.26}$$

La ecuación 1.26 genera una curva B-Spline de grado p utilizando únicamente la propiedad de localidad y obviamente a la base B-Spline. Si se observa esta ecuación, notamos que aparece una nueva variable l a diferencia de la definición 1.25 antes mostrada. Esta nueva variable indíca el índice del intervalo en el cual se encuentra u en el vector de nodos, propiamente $u \in [u_l, u_{l+1})$.

El algoritmo de Boor propone una nueva notación para el cálculo de esta curva. Dado u tal que $u \in [u_l, u_{l+1})$ y sea $\overrightarrow{d_i^{[0]}} = \overrightarrow{d_i} = \overrightarrow{P_i}$ para $i = l - p, \ldots, l$.

$$d_i^{[k]} = (1 - \alpha_k^i) d_{i-1}^{[k-1]} + \alpha_k^i d_i^{k-1}$$

$$k = 1, \ldots, p$$

$$i = l - p + k, \ldots, l$$

$$\alpha_k^i = \frac{u - u_i}{u_{i+p+1} - u_i}$$

$$\overrightarrow{c}(u) = \overrightarrow{d_l^{[p]}} \tag{1.27}$$

Esta última ecuación 1.27 expresa a la curva B-Spline eficientemente utilizando recursión y aprovechando la propiedad de localidad que poseen estas curvas. Este algoritmo tiene una complejidad de $O(n^2)$, donde n es el grado de la curva, lo cual contradice un poco que su eficiencia es tan buena como se presume. De alguna manera hay una mejora computacionalmente, dado que esta definición ahorra operaciones que en las definiciones anteriores se hacían sin tomar en cuenta propiedades, específicamente se ahorran multiplicaciones por cero.

1.7. Superficies B-Spline

La generación de superficies B-Spline es, como de esperar, muy similar a la generación de las superficies de Bèzier. Las superficies B-Spline también se consideran superficies que se pueden generar con el producto de dos curvas, es decir superficies de producto tensorial.

De igual manera que las superficies de Bèzier estas superficies también dependen de dos parámetros u y v. Dados los parámetros, sabemos, como se vio en la sección anterior, que cada parámetro es acompañado por un vector de nodos **U** de tamaño $h + 1$ y **V** de tamaño $k + 1$. Los puntos de control $\overrightarrow{P_{ij}}$ con $m + 1$ filas y $n + 1$ columnas, p y q los grados para las bases para cada dirección.

$$\overrightarrow{s}(u, v) = \sum_{i=0}^{m} \sum_{j=0}^{n} N_i^p(u) N_j^q(v) \overrightarrow{p}_{ij} \tag{1.28}$$

Las superficies B-Spline se definen como lo expresa la ecuación 1.28. Donde $N_i^p(u)$ es la base B-Spline de grado p en la dirección u y $N_j^q(v)$ es la base B-Spline de grado q en la dirección v. Cada vector de nodos debe cumplir una relación matemática entre el grado de la base y el número de columnas o filas según sea el caso. El tamaño para cada vector de nodos para el caso que se expone es la siguiente $h = m + p + 1$ y $k = n + q + 1$.

Es importante recalcar que a diferencia de las superficies de Bèzier, las superficies B-Spline poseen la propiedad de localidad, o soporte local. La cual en pocas palabras implica que no se necesitan de todos los puntos de control para generar segmentos de las curvas isoparamétricas de esta superficie.

Dado que estas superficies poseen la propiedad de soporte local y si se analiza la ecuación 1.28 notamos que los índices de las dos sumatorias recorren todos los puntos de control para calcular solamente un punto de la superficie. Se sabe por una de las propiedades de las curvas que en ciertos intervalos la base B-Spline es cero, lo que provoca pensar en alguna definición de la superfice que aproveche esta localidad para ahorrar operaciones.

Si se observa la ecuación 1.26 notamos que esta ecuación genera la curva B-Spline utilizando la propiedad de localidad. De tal forma, que si se retoma el concepto de superficies de producto tensiorial y esta ecuación, entonces se puede generar una superficie B-Spline.

$$\overrightarrow{S}(u, v) = \sum_{i=l_u-p}^{l_u} \sum_{j=l_v-q}^{l_v} N_i^p(u) N_j^q(v) \overrightarrow{P_{ij}} \tag{1.29}$$

La ecuación 1.29 es la más importante para esta tesis. La razón es que a partir de esta ecuación podemos generar una expresión matricial y la cual más adelante se utilizará para el diseño del software. Esta misma ecuación también puede expresarse matricialmente como se muestra en la ecuación 1.30.

$$\overrightarrow{S}(u,v) = \mathbf{A} \cdot \mathbf{P} \cdot \mathbf{B} \tag{1.30}$$

$$\mathbf{A} = \begin{bmatrix} N^p_{l_u-p}(u) & N^p_{l_u-p+1}(u) & \dots & N^p_{l_u}(u) \end{bmatrix}$$

$$\mathbf{B} = \begin{bmatrix} N^q_{l_v-q}(v) \\ N^p_{l_v-q+1}(v) \\ \vdots \\ N^q_{l_v}(v) \end{bmatrix}$$

$$\mathbf{P} = \begin{bmatrix} P_{l_u-p,l_v-q} & P_{l_u-p,l_v-q+1} & \dots & P_{l_u-p,l_v} \\ P_{l_u-p+1,l_v-q} & P_{l_u-p+1,l_v-q+1} & \dots & P_{l_u-p+1,l_v} \\ \vdots & \vdots & \dots & \vdots \\ P_{l_u,l_v-q} & P_{l_u,l_v-q+1} & \dots & P_{l_u,l_v} \end{bmatrix}$$

La matriz **A** representa las bases B-Spline en la dirección u de grado p, mientras que la matriz **B** representa la base B-Spline en la dirección v de grado q. La matriz **P** representa los puntos de control que se encuentran involucrados en el cálculo de un punto perteneciente a la superficie. Es decir, que es una matriz que contiene los puntos de control necesarios para el cálculo del punto.

Esta expresión matricial de la superficie será de gran utilidad para diseñar el software que más adelante se presentará. La razón de utilizar esta expresión es porque fácilmente se identifican los agentes matemáticos que participan para la generación de la superficie.

Capítulo 2

Unidad Procesadora de Gráficos y el Proceso de Dibujo de OpenGL

2.1. Introducción

Este capítulo tiene como propósito explicar la arquitectura general de una Unidad procesadora de Gráficos (GPU) y la tecnología que se utiliza al utilizar los llamados Shaders. Así como también analizar el proceso que se lleva a cabo cuando se utiliza OpenGL como biblioteca de dibujo; específicamente las etapas que se realizan para dibujar algo en pantalla.

Regularmente los llamados Shaders, que en este capítulo se explicarán, son muy utilizados en la industria de videojuegos debido a que permiten manipular etapas del proceso de dibujo y de esta forma, manipular el detalle visual de los objetos dibujados. Existen diferentes lenguajes* de programación para desarrollar Shaders, los cuales han sido desarrollados por diferentes empresas y organizaciones. DirectX, la cual es una biblioteca de Microsoft ©facilita el uso de shaders con su llamado lenguaje HLSL. OpenGL también tiene su propio lenguaje para desarrollar Shaders el cual se llama GLSL y Nvidia ©que también cuenta con un lenguage el cual tiene por nombre CG y el

*Estos lenguajes que se mencionaron en la introducción, son lenguajes de alto nivel que de alguna manera abstraen algunos procedimientos comunes que se realizan en el lenguaje ensamblador del procesador de gráficos. Es muy similar a la abstracción que lenguajes como C facilitan para programar el CPU de la computadora.

que se utilizó para desarrollar el producto final de esta tesis.

2.2. Proceso de Dibujo de OpenGL

2.2.1. Breve Historia de OpenGL

OpenGL es una interface API, la cual tiene como propósito facilitar el dibujo en pantalla en una computadora de tal manera que ésta sea multiplataforma [11]. La especificación para esta API fue terminada en 1992 y su primera implementación fue hecha en 1993. La compañía Sillicon Graphics había desarrollado una API la cual era muy compatible con la especificación hecha en 1992; para establecer en la industria un estándar, Sillicon Graphics colaboró con otras compañías de Hardware para Gráficos con el motivo de crear un estandar abierto el cual derivó en **OpenGL**.

OpenGL comparte mucho del diseño con IrisGL, de Sillicon Graphics. La intención de esta API es darle al usuario la capacidad de acceder al Hardware de gráficos a un nivel bajo, es decir permitir aprovechar al máximo el Hardware sin usar una API con alto nivel de abstracción.

OpenGL presume de ser una API multiplataforma, esto se refiere a que la API puede ser utilizada en distintos Sistemas Operativos y también presume que ha podido ser utilizada en diferentes arquitecturas de Hardware Gráfico. OpenGL está constantemente siendo revisada por su comité conformado por distintas empresas. Desde el primer lanzamiento de OpenGL versión 1.0 han habido revisiones frecuentemente que han agregado nueva funcionalidad a esta API.

2.2.2. El Framebuffer

El API de OpenGL se diseñó para dibujar gráficos usando la computadora. Por lo que datos enviados por una aplicación externa al API deben ser desplegados en pantalla. A este proceso mencionado se le conoce como **render** [11]. Este proceso puede ser realizado por Hardware especializado aunque a veces algunas o todas las operaciones que conlleva realizar esta acción las puede hacer un software.

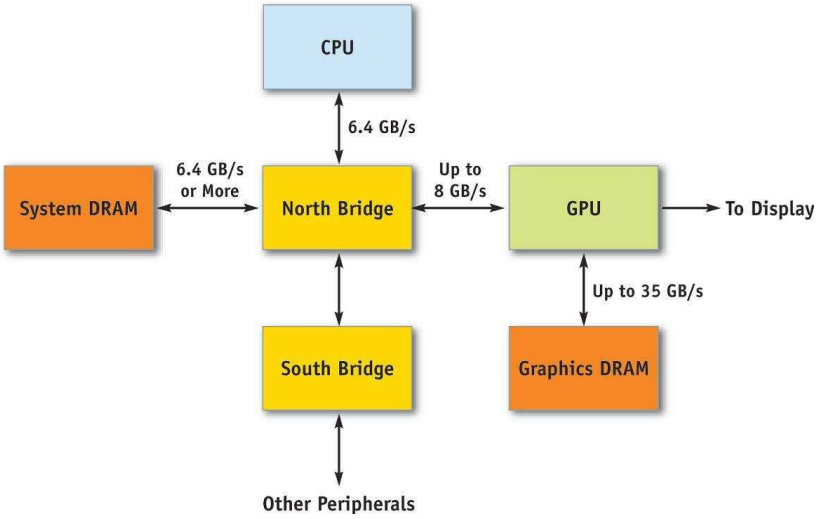


Figura 2.1: Arquitectura dentro de una PC

La organización de una computadora, la cual puede desplegar información visual tiene una arquitectura como se muestra en la figura 2.1, como la expone Robinson [1]. Como se puede observar, existe una porción de la memoria que está destinada a contener la información que se despliega en pantalla. A esta porción de la memoria en la cual se dibujará, se le llama **FrameBuffer** y la cual es accesada mediante algún Bus para desplegar finalmente en pantalla. La tarjeta de video accesa el FrameBuffer para refrescar la imagen en pantalla, esta acción se lleva a cabo 60 veces por cada segundo [3].

El sistema de ventanas actualmente es el encargado de manejar el alojamiento y el acceso al FrameBuffer, por lo tanto éste es el encargado de asignar memoria a utilizar por OpenGL [11]. En cada sistema operativo existe un conjunto de funciones las cuales unen a OpenGL y al ambiente en el que se está ejecutando OpenGL. Por ejemplo, en Microsoft Windows este conjunto de funciones reciben el nombre de WGL, en el ambiente X Window System este conjunto de funciones tiene por nombre GLX, y en las computadoras Macintosh que no utilizan X Window System, recibe el nombre de AGL. La función de este conjunto de funciones es manejar el alojamiento y desalojamiento de estructuras llamadas Contextos Gráficos (GRAPHIC CONTEXTS) las cuales mantienen el estado

de OpenGL [11]. Estas funciones seleccionan el contexto gráfico, así como también seleccionan la región de memoria en la cual se dibujará. Estas funciones son encargadas de sincronizar los comandos entre OpenGL y el sistema de ventanas.

Cada pixel representado en pantalla tiene un modelo de datos asociado a la memoria de la computadora, es decir en el FrameBuffer. Una imagen en escala de grises podría tener una representación por pixel con tan sólo usar 1 byte, mientras que una imagen a color necesitaría 3 bytes para un pixel, 1 byte para cada componente de color. El color es decodificado usando alguna paleta de color definida. De esta manera es como se organiza una computadora y como es que el Hardware Gráfico funciona en ella. Con esta breve explicación, se puede comprender mejor la finalidad de alguna biblioteca de dibujo. En breve, las bibliotecas de gráficos, se encargan de escribir en el FrameBuffer de tal forma que se obtenga la imagen a desplegar en pantalla.

2.2.3. El proceso de Dibujo de OpenGL

Esta subsección explica como está organizado el proceso de dibujo de OpenGL, en otras palabras como es que OpenGL escribe en el FrameBuffer para desplegar modelos, polígonos, etc. Se recomienda mucho entender el proceso de dibujo para ser capaz de escribir Shaders porque están ligados estrechamente.

El proceso de dibujo[7] que se utiliza en OpenGL refleja estado. Esto significa que existen etapas del proceso que reflejan un estatus de lo que se dibujará en pantalla. En la figura 2.2 se presenta un diagrama de bloques de las etapas que presenta dicho proceso. Los comandos pueden ser enlistados en una lista o pueden ser procesados inmediatamente. Dicha lista, la cual recibe el nombre de Lista de Despliegue, es útil porque permite optimizar y reusar comandos, pero no todos los comandos pueden ser enlistados. A continuación se describen brevemente las etapas que se llevan a cabo en el proceso de dibujo [5].

Como primer etapa del proceso, se tiene el Evaluador el cual toma cualquier comando del tipo polinomial y lo evalúa utilizando los vértices correspondientes y atributos del comando. Como segunda etapa se tienen transformaciones a los vértices. En esta etapa se realizan operaciones por

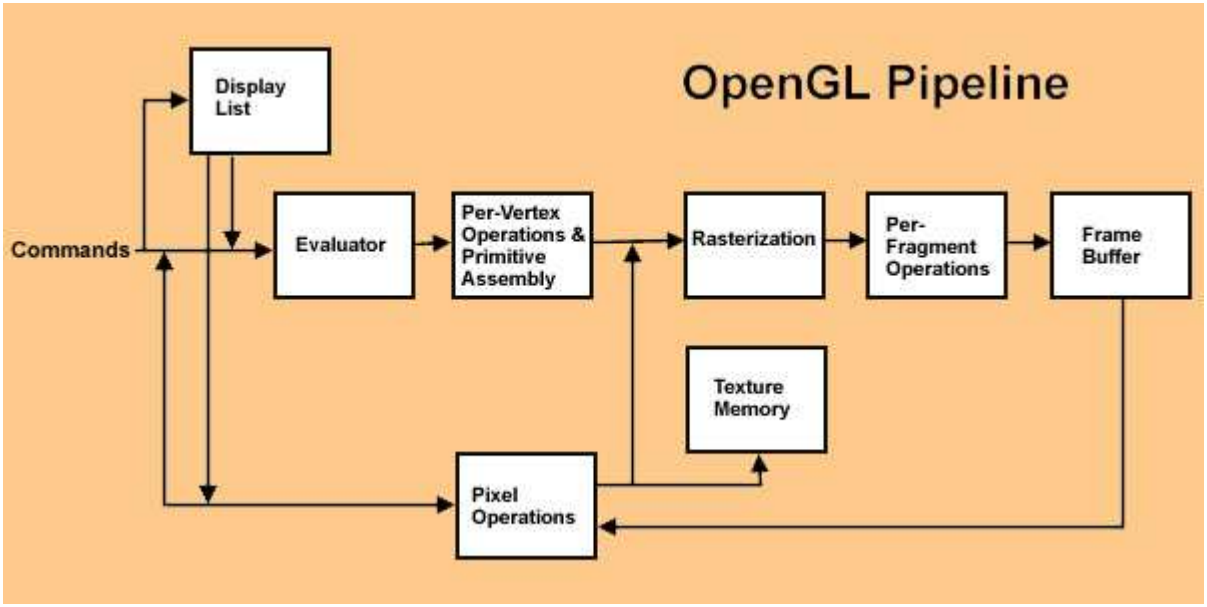


Figura 2.2: Proceso de Dibujo de OpenGL

cada vértice. Cada vértice tiene como atributo su posición en el espacio, un color, una normal, coordenadas de alguna textura, entre otras más. Las operaciones que se realizan en esta etapa son transformaciones a la posición del vértice, iluminación por vértice y generación o transformación de las coordenadas de textura.

En la tercera etapa del proceso se realiza el proceso de rasterización, el cual produce fragmentos. Los fragmentos son series de direcciones de memoria del FrameBuffer y algunos valores. En esta etapa se recibe como entrada vértices y su información de conectividad entre si. Con esta información de conectividad entre vértices es posible formar primitivas, como líneas, triángulos, puntos, polígonos, cuadrados, etc. En esta etapa específicamente es donde se ensamblan las primitivas.

Un fragmento es una pieza de información que será utilizada para actualizar el valor de un pixel en el FrameBuffer en determinada localidad [5]. Un fragmento no sólo contiene color, también contiene información de normales, coordenadas de textura y cualquier otra información necesaria para actualizar el color de un nuevo pixel. Como salida de esta etapa, se obtienen las posiciones de los fragmentos en el FrameBuffer y los valores interpolados de cada fragmento que fueron calculados

en la etapa de procesamiento de vértices.

En la cuarta etapa se realizan operaciones por cada fragmento. Antes de mandar al FrameBuffer un fragmento, este es sujeto a operaciones tales como **Blending** y **Z-Buffering** [5]. La operación Blending realiza una operación con colores, de tal manera que como salida de la operación se obtiene un color. La operación Z-Buffering o también conocida como depth-buffering se encarga de calcular la profundidad de alguna imagen en 3D. En resumen, esta operación consiste en guardar en un Buffer la coordenada Z de algún pixel, cuando algún otro objeto también utilice ese pixel la tarjeta gráfica compara las dos profundidades y escoge la profundidad más cercana al observador. Finalmente se realizan operaciones por pixel, y se despliega en pantalla.

Hasta este punto es posible cuestionar si es posible de modificar este proceso de dibujo y la respuesta es afirmativa. Muchas tarjetas recientes de video permiten modificar algunas etapas del proceso de dibujo, a los programas que describen la modificación se les conoce como **Shaders**. Es importante resaltar que la tarjeta de video es la encargada de ejecutar Shaders para modificar el proceso de dibujo.

Existen diferentes tipos de Shaders, los cuales están asociados a cada etapa del proceso de dibujo; por ejemplo los VertexShaders son encargados de modificar la parte de operaciones por vértice, por lo que el programa debe realizar operaciones con vértices de entrada y producir vértices [12]. Es muy importante tener en cuenta, que para no alterar el proceso de dibujo se debe tener claro lo que produce cada etapa así como las entradas por cada una de estas. Una regla importante que se utiliza cuando se programan Shaders, es que siempre se debe tener bien claro la etapa que se modifica, dado que se está sobrescribiendo la funcionalidad por defecto que se tiene. Por este motivo es importante también conocer como se constituye arquitectónicamente una unidad procesadora de gráficos, ya que esta pieza de Hardware es la encargada de ejecutar dichos programas.

2.3. Unidad Procesadora de Gráficos GPU

La unidad procesadora de Gráficos, llamada también por sus siglas en inglés GPU (Graphic Processing Unit), se encuentra últimamente en las computadoras modernas. Su función principal es acelerar el despliegue de los componentes gráficos. Este componente se encuentra en el dispositivo de video de la computadora, la cual se compone además del GPU de un BIOS de Video, memoria de Video, RAMDAC y salidas de video como VGA, DVI y S-Video.

Como sabemos, el GPU trabaja con imágenes, por lo tanto es válido cuestionarse, qué tipo de datos trabaja el GPU. Como entrada al GPU se tienen los siguientes datos: primitivas geométricas en 3d llamadas *mallas*, o en inglés "Meshes", algunas imágenes llamadas *texturas*, algunas matrices, regularmente de 4x4 y casi siempre algunos otros comandos[1].

El elemento primordial en esta tesis el cual no se tiene que perder de vista es el vértice. Este vértice forma parte de las primitivas, y por lo tanto en las mallas de entrada. En conjunto los vértices forman primitivas, y el conjunto de primitivas forma las mallas. Un vértice regularmente necesita de datos como posición, vector normal, y una coordenada de textura [1].

La propiedad que nos interesa en este trabajo es la posición, aunque como breviario cultural es conveniente explicar brevemente la utilidad del vector normal, y la coordenada de la textura. El vector normal regularmente se usa para realizar cálculos de iluminación, estos cálculos ayudan a dar efectos visuales muy interesantes, como lo es el Bump Mapping. La coordenada de la textura indica como es que la imagen será mapeada en la geometría, que se está dibujando [1].

Los GPUs modernos traen muchos procesadores de vértices así como procesadores de pixeles los cuales calculan el color de los vértices o de los pixeles en paralelo. Estos procesadores son originalmente diseñados para el cálculo del color de los vértices o de los pixeles aunque también pueden ser utilizados para cálculos de propósito general.

Las ventajas de utilizar el GPU son distintas, una de las cuales, que es siempre muy importante en aplicaciones de video, es el desempeño. El separar la carga de trabajo, entre el CPU y el GPU puede ayudar a utilizar la memoria en un uso más eficiente, dado que la transmisión de datos que

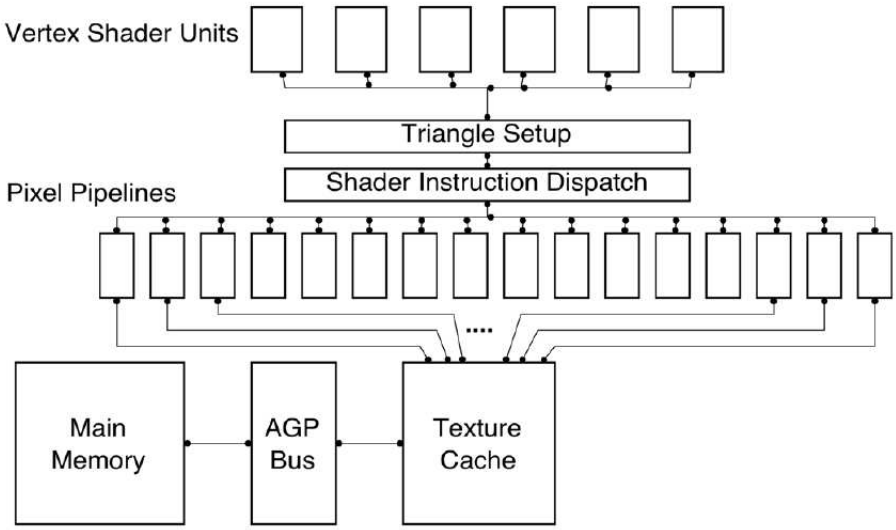


Figura 2.3: Diagrama de Bloques general del GPU

deben ser refrescados en pantalla mediante algún bus pueda ser disminuida [1].

2.3.1. Arquitectura de la Unidad Procesadora de Gráficos

El GPU implementa en Hardware el proceso de dibujo, esto implica que como todo sistema recibe una entrada y produce una salida. Como entrada el GPU recibe la geometría que se va a dibujar, realiza las transformaciones geométricas necesarias sobre la entrada, como rotaciones, traslaciones, etc, para después calcular el color y finalmente desplegar el resultado[9].

La arquitectura general de una unidad procesadora de gráficos se presenta en la figura 2.3. Como se puede ver en la figura 2.3 un GPU consta de un número fijo de unidades que procesan vértices (Vertex) y pixeles, los cuales pueden trabajar en paralelo. Estas unidades obtienen los datos a procesar de memoria reservada para textura. El bus que puede conectar al GPU con la memoria usualmente es el AGP aunque también mediante PCI es posible realizar esta conexión. El proceso es el siguiente, los datos se extraen de memoria principal y estos viajan a la memoria de textura del

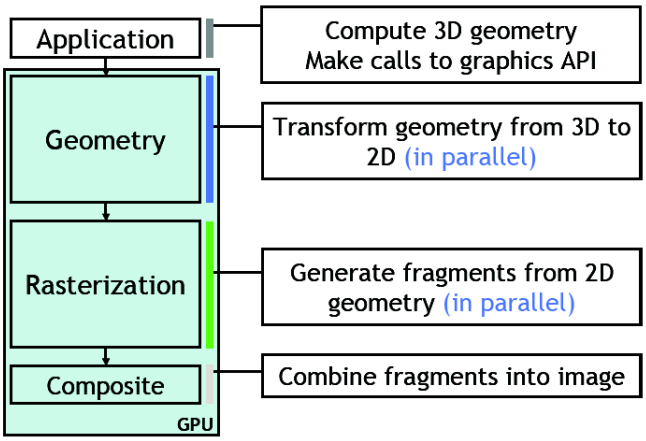


Figura 2.4: Proceso de Despliegue

GPU mediante algún BUS, ya sea AGP o PCI dependiendo del dispositivo de video. De esta manera los datos están disponibles para los procesadores (shaders) de vértices o pixeles para realizar alguna tarea asignada.

Es importante mencionar la importancia que tiene el BUS que permite la transferencia de datos hacia el GPU, dado que puede ser un factor a considerar si se busca un rendimiento en el cómputo. Como se muestra en el artículo escrito por Pushkar [10], hay dos puntos que se deben tomar en cuenta para desarrollar aplicaciones en un GPU para propósito general.

- La carga de transferencia que se pasará a los Shaders a procesar debido a la capacidad del BUS.
- Los GPUs modernos cuentan con una implementación propietaria del punto Flotante, la cual se sabe que no es tan precisa.

Para cerrar esta subsección se muestra la figura 2.4 la cual muestra muy fácilmente como es que se lleva acabo el proceso gráfico y en dónde interviene el GPU. Como primer bloque se tiene a la aplicación la cual hace llamadas a la Intefraz de gráficos API, en nuestro caso OpenGL. Esta

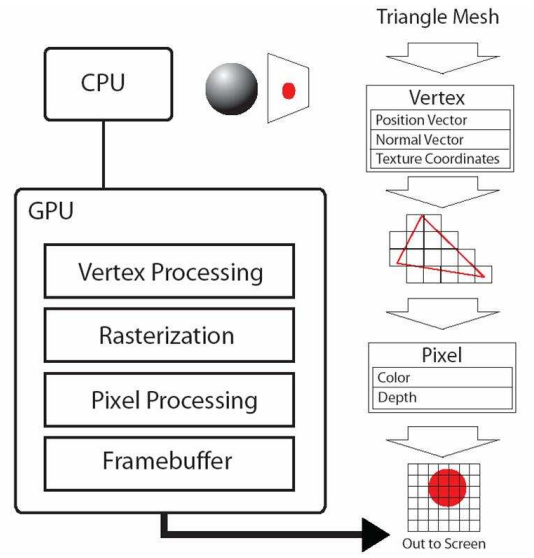


Figura 2.5: Proceso de Dibujo en el GPU

aplicación calcula la geometría en 3D y esta es pasada mediante la interfaz al GPU para ser procesada. Como segundo bloque esta geometría es transformada de 3D a una proyección en 2D, debido a que nuestra pantalla sólo muestra fiugras planarmente. Se generan los fragmentos de la geometría en 2D, lo cual compone la parte de Rasterización y finalmente se combinan los fragmentos para formar una imagen y ser desplegada.

Se observa en la figura 2.5 que existen cuatro últimas etapas las cuales son ejecutadas por el GPU, por lo que se concluye que es de gran importancia esta pieza dado que realiza tareas importantes para lograr un despliegue en pantalla. Es interesante saber, que por el año de 1990 este proceso era fijo, es decir que no se le permitía al usuario modificar las etapas de dibujo. Como se puede observar la figura 2.5 muestra el proceso de dibujo identificando las acciones específicas que el GPU debe realizar.

El modelo de cómputo del GPU es diferente al del CPU. El GPU fuerza una estrecha relación entre el código y los datos; esta relación es muy diferente hablando del CPU. En el CPU, las instrucciones son cargadas por el hardware cuando se incrementa el contador de programa, y esas instrucciones son las encargadas de cargar los datos de memoria que necesiten. En el GPU el har-

dware carga automáticamente los datos así como también las instrucciones escritas en el búffer de comandos por el CPU [1].

Otra diferencia importante entre el CPU y el GPU es el uso de la ALU(Unidad Aritmética Lógica). El CPU es responsable de encargar un cálculo al ALU, mientras que en el GPU la tarea de asignar tareas al ALU es mas larga, como se menciona a continuación.

1. Cuando el CPU escribe comandos al GPU, este hace referencia a localidades en donde se encuentran un número de programas para el GPU ”Shaders”. Estos Shaders corren en el GPU de acuerdo a la etapa que le corresponda.

2. Cuando una tarea es agendada en el GPU, este asigna un Shader a la vez a cada una de las etapas correspondientes. Estas etapas, corren simultáneamente con diferentes Shaders y diferentes datos.

3. Dentro de cada etapa, el GPU ejecuta diferentes hilos (threads). Estos hilos son manejados por el propio GPU, no por el programa mismo.

4. Cuando el Shader termina, se liberan recursos, y por lo tanto el GPU utiliza estos recursos llenándolos con más información a procesar.

Este modelo se acerca mucho al formalmente modelo llamado ”stream processing”. Que puede ser traducido como procesamiento de flujo. La característica principal de este modelo es que un pedazo de código es llamado una y otra vez cada que nuevos datos llegan y que existan recursos para procesar. A diferencia con el CPU, que necesita de incrementar el contador de programa para ejecutar una instrucción la cual es la encargada de extraer la información a procesar[1].

2.3.2. Vertex Shaders

Como se describió en la subsección anterior, existe una etapa en la cual se procesan los vértices a ser dibujados. Es imporante comprender bien estos programas, dado que serán utilizados para generar las superficies B-Spline.

Los programas encargados de procesar vértices son los Vertex Shaders. Estos Shaders tienen como entrada exáctamente un vértice y un vértice como salida. Un vértice tiene atributos tales como un vector normal, un color primario y secundario, coordenadas de textura o cualquier otro valor necesario para algún calculo para procesar el vértice. Estos programas no pueden quitar vértices de alguna primitiva así como tampoco agregar vértices. Tampoco es posible operar en múltiples vértices al mismo tiempo [8].

Cuando un Vertex Shader está en uso, algunas funciones establecidas del procedimiento de vértices no están activas, por lo que si se intenta utilizar alguna de estas funciones no tendrán efecto alguno sobre el Shader [2]. Estas funciones se enlistan a continuación:

- Transformaciones de Corte (Clipping).
- Normalización.
- Iluminación y Materiales.
- Generación de coordenadas de Texturas.
- En algunos ambientes de shaders, los planos de corte definidos por el usuario son deshabilitados.

Algunos lenguajes de ”Shading”, especialmente los de bajo nivel, definen su propio contexto de ejecución. Aunque podemos generalizar una arquitectura del contexto de ejecución del Vertex Shader como se muestra en la figura 2.6. El Vertex Shader tiene acceso a un cierto número de registros, los cuales algunos son solamente de lectura o de escritura. En recientes arquitecturas estos registros son especiales para punto flotante aunque también algunas otras arquitecturas presentan registros capaces de manejar registros booleanos y enteros [10].

Los vértices poseen atributos, los cuales son almacenados en los registros anteriormente mencionados. El vertex shader tiene una limitante en cuanto al uso de registros temporales para el procesamiento, pero dispone de una gran cantidad de registros de sólo lectura para extraer los atributos del vértice. Finalmente el Shader escribe su producto en registros de sólo escritura. Estos registros

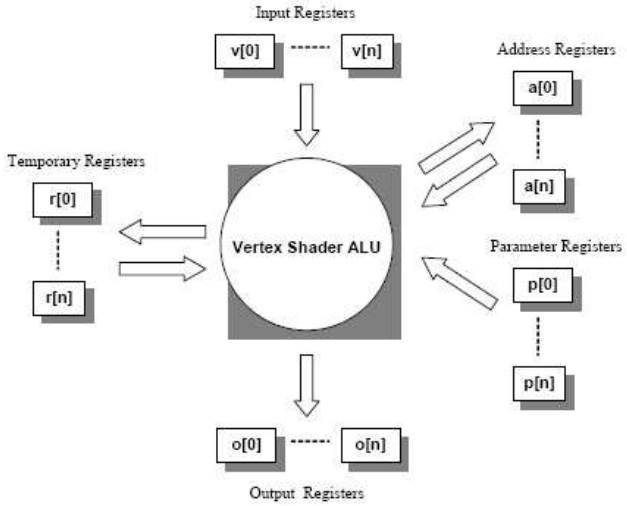


Figura 2.6: Contexto de Ejecución de un Vertex Shader

poseen una estructura definida, la cual contiene la posición transformada homogénea, coordenadas de textura y colores de vértices. Finalmente estos resultados son pasados a la siguiente etapa del proceso de dibujo.

Capítulo 3

Arquitectura de la Aplicación Evaluadora de Superficies B-Spline Bicúbicas

3.1. Introducción

El presente capítulo presenta la arquitectura que se ideó y se utilizó para llevar acabo el desarrollo del software encargado de generar una superficie B-Spline bicúbica, utilizando Vertex Shader con el lenguaje CG de Nvidia ©y C++. Este capítulo también describe la solución a problemas que se presentaron durante el desarrollo, algoritmos utilizados, diagrama de clases y demás artefactos necesarios para explicar la implementación del software dirigido a calcular y dibujar dicha superficie. Este capítulo muestra principalmente los componentes por separado, que en conjunto serán utilizados para calcular la superficie.

Podemos dividir la aplicación completa en dos partes principales: **Aplicación principal** encargada de llevar el flujo de dibujo, y el **Vertex Shader**, el cual evalúa por cada vértice de entrada un punto en el espacio de la Superficie. Esta separación tiene como particular caracteristica los actores que procesan cada parte; para la Aplicación principal el CPU es el encargado de ejecutar las

instrucciones de ese programa, mientras que el Vertex Shader tiene como actor principal el GPU.

3.2. Análisis del Problema y Arquitectura del Software

Para poder definir una arquitectura de solución a algún problema, es necesario comprender el problema o la meta. El objetivo principal para este documento de tesis en concreto es desarrollar un Vertex Shader capaz de evaluar una superficie B-Spline bicúbica y dibujarla en pantalla, por lo que para esta sección es necesario tener un buen entendimiento de la matemática necesaria para las superficies B-Spline y la arquitectura de Hardware de una tarjeta gráfica las cuales pueden ser revisadas en el capítulo 1 y 2.

3.2.1. Arquitectura de la Aplicación Principal

Para comenzar a definir la arquitectura que la aplicación utilizó, es necesario retomar la definición de las supercies B-Spline. Como se menciona en el capítulo 1 en la sección **Superficies B-Spline**, la definición de esta superficie se expresa como la ecuación 3.1 lo muestra.

$$\overrightarrow{S}(u,v) = \sum_{i=l_u-p}^{l_u} \sum_{j=l_v-q}^{l_v} N_i^p(u) N_j^q(v) \overrightarrow{P_{ij}} \tag{3.1}$$

Como se menciona en la sección **Superficies B-Spline** la ecuación 3.1 puede ser expresada matricialmente de la siguiente forma:

$$\overrightarrow{S}(u,v) = \mathbf{A} \cdot \mathbf{P} \cdot \mathbf{B} \tag{3.2}$$

$$\mathbf{A} = \begin{bmatrix} N_{l_u-p}^p(u) & N_{l_u-p+1}^p(u) & \dots & N_{l_u}^p(u) \end{bmatrix}$$

$$\mathbf{B} = \begin{bmatrix} N^{q}_{l_v-q}(v) \\ N^{p}_{l_v-q+1}(v) \\ \vdots \\ N^{q}_{l_v}(v) \end{bmatrix}$$

$$\mathbf{P} = \begin{bmatrix} P_{l_u-p,l_v-q} & P_{l_u-p,l_v-q+1} & \cdots & P_{l_u-p,l_v} \\ P_{l_u-p+1,l_v-q} & P_{l_u-p+1,l_v-q+1} & \cdots & P_{l_u-p+1,l_v} \\ \vdots & \vdots & \cdots & \vdots \\ P_{l_u,l_v-q} & P_{l_u,l_v-q+1} & \cdots & P_{l_u,l_v} \end{bmatrix}$$

Analizando la ecuación 3.2 podemos identificar que debemos contar con alguna rutina para calcular la base B-Spline de grado **p** y con parámetro **u**. Retomando que la definición de las superficies puede ser expresada matricialmente, se concluye que se necesitará de otra rutina encargada de generar los vectores **A** y **B**.

$$N^{0}_{i}(u) = \begin{cases} 1 & \text{si} \quad u_i \leq u < u_{i+1} \\ 0 & \text{si} \quad \textit{de otra manera} \end{cases} \tag{3.3}$$

$$N^{p}_{i}(u) = \frac{u - u_i}{u_{i+p} - u_i} N^{p-1}_{i}(u) + \frac{u_{i+p+1} - u}{u_{i+p+1} - u_{i+1}} N^{p-1}_{i+1}(u)$$

Si se retoma la definición de las Bases B-Spline del primer capitulo ya que es importante tenerla presente, y se expresa con la ecuación 3.3 tal y como se presenta en el primer capítulo, se observa de la definición que requerimos de un Vector de Nodos u, el cual bien puede ser expresado utilizando alguna estructura de datos como un arreglo, o un Vector. Gracias a la programación Orientada a Objetos se puede definir una estructura de datos que se adecue a nuestras necesidades, por lo que se propone como primer Objeto a crear la clase que represente un vector de nodos. Cabe mencionar que el nombre para esta clase fue **KnotVector** y fue programada en C++.

Clase KnotVector

Esta clase tiene como propósito definir la estructura que un objeto de este tipo debe poseer. En esta sección dedicada a esta clase se describirán atributos así como también métodos los cuales podamos utilizar para realizar operaciones necesarias para el cálculo de la base B-Spline.

Un vector de nodos no es mas que una colección de números ordenados ascendentemente. Recordemos que el vector de nodos puede contener números con multiplicidad, esto significa que podemos tener números repetidos dentro de la colección. Si suponemos el caso de que se desea generar una curva abierta, necesariamente implicaría que el primer elemento tanto como el último elemento del vector tengan multiplicidad. Esta multiplicidad al principio del vector como al final, implica analizar bien donde comienza el intervalo tal que el argumento u de la base B-Spline cumpla con la siguiente condición 3.4 con elementos del vector K.

$$K_i \leq u < K_{i+1} \tag{3.4}$$

Por ejemplo, si se tiene un vector de nodos normalizado definido de la siguiente manera:

$$\mathbf{K} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1/3 & 2/3 & 1 & 1 & 1 & 1 \end{bmatrix}$$

y se desea evaluar la base B-Spline de grado 3, con un argumento $u = 1/4$, es decir $N_i^3(1/4)$; entonces necesariamente se debe encontrar el índice i de tal forma que $K_i \leq 1/4 < K_{i+1}$. Para este caso el índice tiene un valor de $i = 3$ ya que $K_3 = 0$ y $K_4 = 1/3$ por lo que la condición se cumple. Es importante mencionar esta condición porque de otra manera se podrían presentar divisiones entre 0, mientras se calcula la base B-Spline de algún número.

Como se puede observar, el encontrar el índice el cual cumpla la condición necesaria de orden, no es mas que una búsqueda dentro de una colección de números ordenados. El algoritmo de búsqueda que se empleó en la aplicación fue **Binary Search** o **Búsqueda Binaria** modificado ligeramente.

El algoritmo de búsqueda binaria tiene como entrada un vector V de números ordenados de

longitud l y una llave a buscar k. El algoritmo consiste en buscar el índice tal que $V[indice] = key$. Este algoritmo busca el elemento dentro de la colección y regresa el índice donde lo encontró, pero el caso que se tiene es ligeramente diferente, ya que no se pretende buscar la llave *key* dentro de la colección, se pretende encontrar el índice tal que $V[indice] \leq key < V[indice + 1]$ [20].

El algoritmo de la búsqueda binaria iterativa consiste de los siguientes pasos:

Entrada: Vector V de longitud N y una llave *key* a buscar.

Salida: Índice tal que $V[indice] = key$

```
BinarySearch( V[0 ... N-1],  key)
    low  <- 0
    high <- N-1

    while( low <= high )
        mid  <- (low + high)/2

        if ( V[mid] > key )
            high <- mid
        else  if( V[mid] < key )
            low <- mid
        else
            return mid //Elemento Encontrado

    return -1 // Elemento no Encontrado
```

Existen diferentes maneras de expresar el algoritmo de Binary Search, se presenta la que al parecer es la forma más clara. Existen algunas otras representaciones interesante como la que presenta Niklaus [24].

Un ejemplo del uso de este algoritmo se presenta a continuación, tomemos el vector V:

$$\mathbf{V} = \begin{bmatrix} 0 & 1/6 & 2/6 & 3/6 & 4/6 & 5/6 & 1 \end{bmatrix}$$

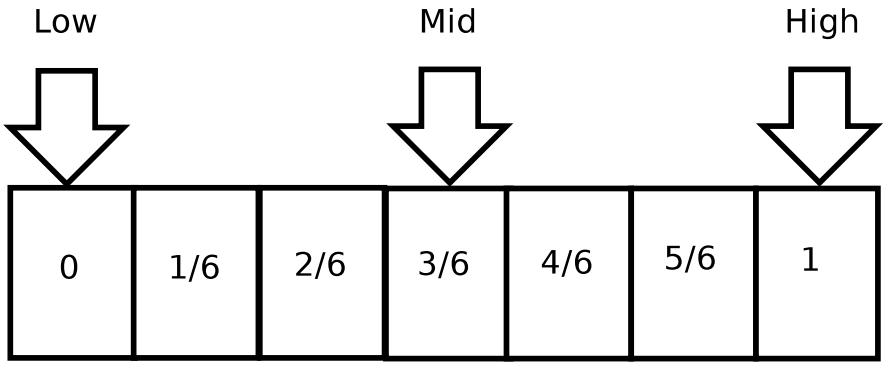
y supongamos que el valor a buscar, es decir la llave, sea $key = 1/6$. Siguiendo los pasos del pseudocódigo se obtendrían las siguientes etapas:

Primera Iteración:

low = 0

high = 6

mid = 3

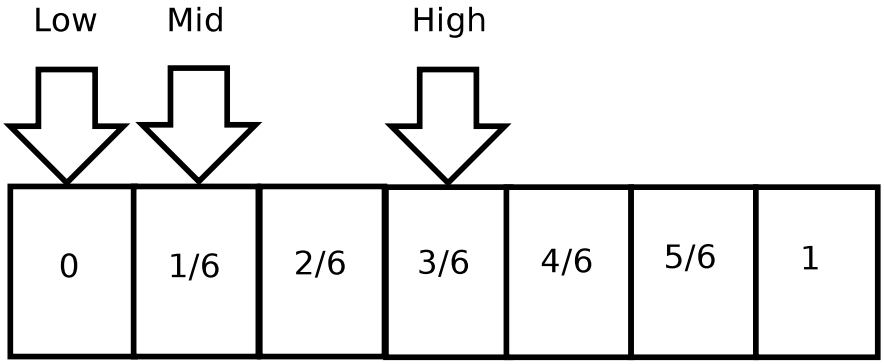


Segunda Iteración:

low = 0

high = 3

mid = 1



Como se mencionó anteriormente, este algoritmo debe ser modificado ligeramente para encontrar el intervalo en el cual la llave cumpla la condición $V[indice] \leq key < V[indice+1]$, y adecuando los actores participantes en la generación de las superficies y específicamente para este proceso, se tiene al vector de Nodos. Es posible visualizar a este vector como una colección de números ordenados ascendentemente el cual para este algoritmo se denotará como V, y al argumento de la función de la base B-Spline recibirá el nombre de *key*. El algoritmo* modificado y denominado ”indexOf”que se utilizó en esta clase se describe a continuación:

*El algoritmo que se utilizó asume que el Vector de Nodos está normalizado, es decir que sus elementos se encuentran en el rango $[0, 1]$

```
indexOf( V[0 ... N-1],  key)
    if( key == 1.0 )
        return upperIndex;
    else if ( key == 0 )
        return lowerIndex;

    low  <- 0
    high <- N-1
    mid  <- (low + high)/2

    while( key < V[mid] || key >= V[mid + 1] )
        if ( key < V[mid] )
            high <- mid
        else
            low <- mid
        mid  <- (low + high)/2

    return mid;
```

En este algoritmo se introducen las variables **lowerIndex** y **upperIndex**, las cuales representan los índices, en donde existe una transición de 0 a 1 y en donde existe una transición de algún número x contenido en el vector menor que 1 y diferente de 0 respectivamente. En pocas palabras estas variables son los índices que indican en donde termina o comienza la multiplicidad de los nodos extremos, tanto al inicio como al final.

La clase KnotVector por lo tanto debe contener métodos los cuales determinen el inicio y final de los elementos, dado que nuestro vector de nodos puede contener multiplicidad en los elementos en los extremos. Para esta tesis se utilizó un vector de nodos con multiplicidad en los extremos, es decir en el primer elemento tanto como en el último elemento con el propósito de generar una

superficie abierta en las dos direcciones. Dada esta necesidad, la clase debe contener dos métodos, los cuales determinen los índices donde comienza el vector y el índice que termina el vector.

Por ejemplo, supongamos que se tiene el siguiente vector como ejemplo:

$$\mathbf{K} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1/3 & 2/3 & 1 & 1 & 1 & 1 \end{bmatrix}$$

La variable *lowerIndex* para este caso debe ser $lowerIndex = 3$, mientras que la variable *upperIndex* debe ser en este caso $upperIndex = 6$. Para encontrar estos índices, se creó un método el cual calculará los índices en donde se presenta esta característica y guardará su valor en un atributo de esta clase. A continuación se describe la rutina que calcula estos índices:

```
computeIndexes( V[0 ... N-1] )
    for( i <- 0 ; i < N ; i++ )
        if( V[i + 1] - V[i] )
            lowerIndex = i
            break
    end

    for( i <- N - 1 ; i > 0 ; i-- )
        if( V[i] - V[i-1] )
            upperIndex = i
            break
    end
```

Este algoritmo supone que el vector ya esta ordenado, y que existen al menos dos elementos diferentes. Este algoritmo es muy simple dado que nuestro vector utilizado tiene la forma de un vector uniforme con multiplicidad en los extremos, es decir al principio y al final. El algoritmo busca de manera lineal diferencias entre elementos.

El tamaño del vector de Nodos está determinado por una relación entre el grado de la base B-Spline y del número de puntos de control, por lo que es necesario validar que el vector de nodos cumpla con esta regla, por lo que también se programó un método el cual validara esta regla. La regla que determina el tamaño queda expresada en la ecuación 3.5.

$$m = p + n + 1 \tag{3.5}$$

Donde m es el tamaño del Vector de Nodos, p es el número de puntos de control y n es el grado que se utilizará para el cálculo de la base B-Spline. De tal forma que la tarea del método simplemente es validar que esto se cumpla, por lo que el pseudo código de éste método queda expresado como sigue:

```
isValid( V[0 .. K], p, n )
  if( K + 1 == p + n + 1 )
    return true

  return false
```

A continuación se presenta un modelo UML de clase en la figura 3.1, con la finalidad de resumir el diseño de esta entidad; no se explica a detalle demás métodos posibles, como constructores, dado que los algoritmos principales que se expusieron se consideraron los más importantes en el cálculo de la superficie.

El constructor utilizado en esta aplicación es aquel que como argumentos tiene el número de puntos de control y el grado de la curva a generar. Con estos datos, y asumiendo que nuestro vector de nodos será uniforme y con multiplicidad a los extremos, es posible generar un vector de nodos que cumpla con las características deseadas de la curva.

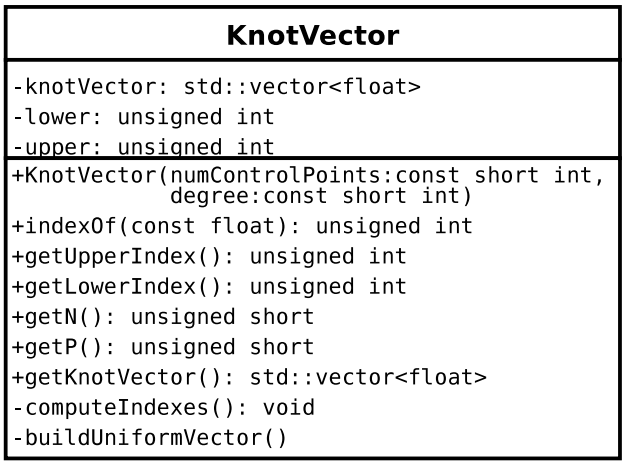


Figura 3.1: Diagrama de Clase UML para KnotVector

Por lo tanto, la construcción de un objeto tendrá la siguiente secuencia:

- 1. Construir el vector de nodos Uniforme ordenado ascendentemente y con multiplicidad en los extremos.
- 2. Calcular índices e inicializar atributos.

Clase ControlPoints

Esta clase es muy simple, dado que su finalidad es guardar los puntos de control que definirá la forma de la superficie a generar. Recordemos que la superficie a generár será bicúbica, esto significa que en ambas direcciones el grado de las curvas isoparamétricas generadas tendrán grado 3.

Los puntos de control deben cumplir ciertas reglas de acuerdo al grado que se utilize para generar la base B-Spline. Existe una relación entre el grado y el número de puntos de control que determina el tamaño del contenedor de los puntos de control.

Es conveniente analizar esta relación especificamente para una dirección, porque de esta manera podemos validar una consistencia de datos necesaria para generar la curva. Dado que la superficie a generar se construye como una superficie de producto tensorial es suficiente analizar la propiedad que nos ayuda a especificar el tamaño del vector en solamente una dirección.

La relación es la siguiente, dado p que representa el grado de la curva B-Spline a generar, entonces:

$$n = p + 1 \tag{3.6}$$

Por lo tanto, para generar una curva B-Spline se requiren $p+1$ puntos de control en una dirección determinada por el parámetro u. Recordemos la definición de la superficie mostrada en la ecuación 3.7.

$$\overrightarrow{S}(u,v) = \sum_{i=l_u-p}^{l_u} \sum_{j=l_v-q}^{l_v} N_i^p(u) N_j^q(v) \overrightarrow{P_{ij}} \tag{3.7}$$

Partiendo de la definición mostrada 3.7, notamos que existen dos parámetros u y v, para los cuales se les asocia un eje en el sistema de coordenadas, es por eso que se dice que en la dirección u se deben tener $p + 1$ puntos de control y para la dirección v se deben tener $q + 1$ puntos de control. Se deduce por lo tanto, que la estructura necesaria para guardar los puntos de control, que no son mas que puntos en el espacio, puede ser una Matriz de puntos en el espacio, en nuestro caso los dos grados en ambas direcciones son cúbicos, por lo que la matriz tendría un tamaño de 4x4, es decir 16 puntos de control.

Una matriz por lo tanto puede ser representada como un arreglo de arreglos, o bien utilizando una estructura similar a un arreglo tal como un vector de vectores. Esta última opción fue la que se utilizó para inicializar un contenedor de Puntos de Control.

La clase queda expresada en un Diagrama de clase como lo muestra la figura 3.2, de acuerdo a lo mencionado en esta sección.

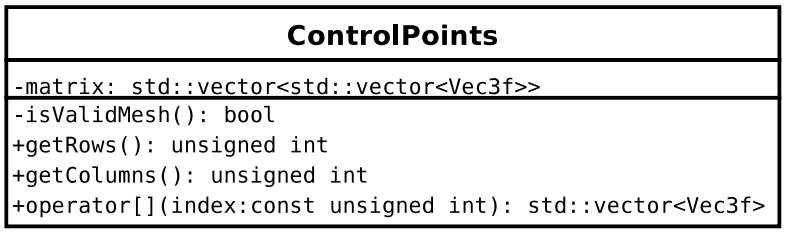


Figura 3.2: Diagrama de Clase UML para los Puntos de Control

Clase Vector

En este punto es conveniente mostrar como es que se representaron los vértices en el espacio para esta aplicación. Se desarrolló una clase la cual representa un punto en el espacio de dimensión tres. La idea es muy simple, un punto dentro del espacio $\Re^3$ contiene tres atributos x, y y z. Las operaciones más comunes sobre un vector en el espacio son las siguientes: escalamiento, combinación lineal, calcular su tamaño y proyeción sobre otro vector. Un diagrama simple de esta clase se presenta en la figura 3.3.

Esta clase fue programada utilizando Template en C++, es decir que se utilizó programación genérica, para especificar el tipo de dato que mejor represente las propiedades de un vector, sobre todo si se busca mayor precisión numérica. Cabe mencionar, que para nuestra aplicación se definió un tipo llamado **Vec3f** el cual no es mas que un Vector parametrizado con elementos del tipo float, es decir que $Vec3f = Vector < float >$.

Clase BsplineInterpolator

Esta clase presenta métodos los cuales serán de gran ayuda cuando se comience a calcular la Superficie. En esta clase presentan funciones para el cálculo de la base B-Spline, así como para calcular el vector de aquellas evaluaciones de la base B-Spline, que si se recuerda la expresión matricial de la superficie en la ecuación 3.2, los vectores **A** y **B** son evaluaciones de la base B-Spline en cada dirección. También existe una función la cual calculará la matriz **P**. En el diagrama 3.4 se puede apreciar mejor las funciones de esta clase.

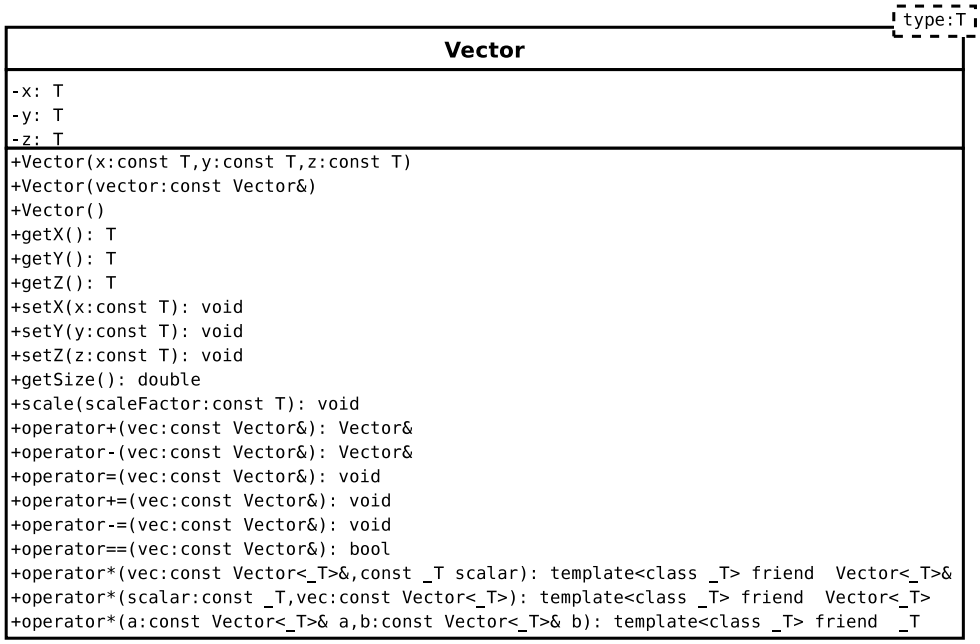


Figura 3.3: Diagrama de Clase UML para la clase Vector

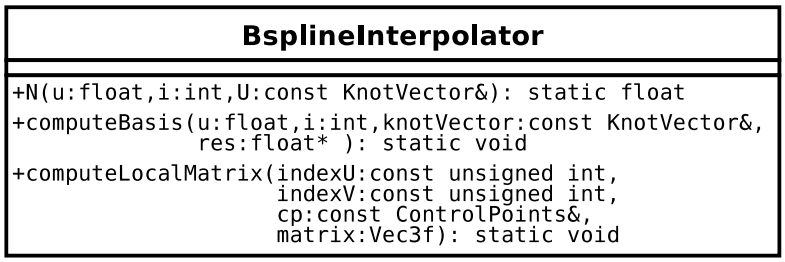


Figura 3.4: Diagrama de Clase UML para la clase con utilerias B-Spline

La función que lleva por nombre **N** evalúa la base B-Spline. Esta función implementa la definición para el cálculo de dicha Base. Se puede ver, partiendo de la definición 3.3, que existe un caso base, y después existe un caso el cual puede ser programado recursivamente. La implementación desarrollada utilizó la manera iterativa, para evitar la recursión y de esta forma tener un mejor control de este cálculo. A continuación se presenta el pseudocódigo que evalúa la base, este pseudocódigo asume que la base a calcular tiene grado tres.

- **Entrada**: Parametro a evaluar u, i indice de la base B-Spline y el vector de nodos U.
- **Salida**: Evaluación de la definición de la base B-Spline

```
N( u, i, U )
 j      <- 0
 Uleft  <- 0.0
 Uright <- 0.0
 saved  <- 0.0
 temp   <- 0.0

 if( i = 0 && U[0] = 0 ||
    (U.getUpperIndex() == i && u == 1.0 ) )
   return 1;

 N[5] //Buffer de 5 elementos

 for( j <- 3; j >= 0; --j)
   if( u >= U[i+j] && u < U[i + j + 1])
     N[j] <- 1.0
   else
     N[j] <- 0.0
```

```
for( k <- 1; k <= 3; k++)
  if( N[0] == 0.0 )
    saved <- 0.0
  else
    saved <- ( (u - U[i])*N[0] )/( U[i + k] - U[i] )

  for( j <- 0; j < 3 - k + 1; j++ )
    Uleft  <- U[i + j + 1]
    Uright <- U[i + j + k + 1]
    if( N[j + 1] == 0.0 )
      N[j]  <- saved
      saved <- 0.0
    else
      temp  <- N[j+1]/ (Uright - Uleft)
      N[j]  <- saved + (Uright - u)*temp
      saved <- (u - Uleft)*temp

 return N[0]
```

La siguiente función llamada *computeBasis* calcula los vectores los cuales contienen las bases B-Spline calculadas para cierto índice. Su función principal es calcular los vectores **A** y **B**. Estos vectores son de gran importancia ya que podemos evitar muchas multiplicaciones por cero si seguimos al pie de la letra la definición del cálculo de la superficie. El pseudocódigo que genera dichos vectores asume que el grado que se maneja es cúbico.

- **Entrada** : Parámetro *u*, índice *i*, el vector de Nodos $knotVector$.
- **Salida** : *res* un apuntador a un búffer en donde se escribirá el resultado del vector.

```
computeBasis( u, i, knotVector, res )
  temp   <- 0
     l   <- i - 3 //Porque el grado es cúbico

  //Se calculan 4 componentes del vector
  for( index <- 0; index < 4; index++ )
    res[index] = N(u, l++, knotVector) //Evaluando la Base B-Spline
    temp += res[index]

  //Se calcula el último componente restante
  temp  -= res[0]
  res[0] = 1 - temp
```

Esta rutina calcula los 4 componentes del vector, lamentablemente en este punto existe un problema, el primer elemento se calcula de manera erronea, pero se corrigió utilizando una propiedad que cumple la base. La propiedad es simple, la suma de los elementos de la base suman 1, es por eso que en las últimas líneas se corrige este problema.

La última función llamada *computeLocalMatrix* es la encargada de calcular la matriz **P**, la cual contiene aquellos puntos que influyen en el cálculo de sólo un vértice. Esta matriz también reduce multiplicaciones innecesarias, y esta matriz tanto como los vectores de las Bases serán entrada para el Vertex Shader. Podemos ver a esta función como aquella función que selecciona de los puntos de control aquellos puntos que verdaderamente influyen en el cálculo del vértice.

El siguiente pseudocódigo muestra como calcular la matriz **P**.

- **Entrada**: *indexU* el índice en la dirección U, *indexV* en la dirección V, *cp* son los puntos de control.
- **Salida**: *matrix* es en donde se escribirá la matriz resultante.

```
computeLocalMatrix( indexU, indexV, cp, matrix )
   u  <- 0
   v  <- 0

   for(i <- indexU - 3; i <= indexU; i++){
     for(j <- indexV - 3; j <= indexV; j++)
       matrix[u][v++] <- cp[i][j];

   u++;
   v = 0;
}//End of for
```

Con estas clases y sus métodos es posible programar la secuencia lógica que se requiere para generar la superficie. Más adelante se mostrará un diagrama de actividades el cual mostrará el flujo que llevará la aplicación.

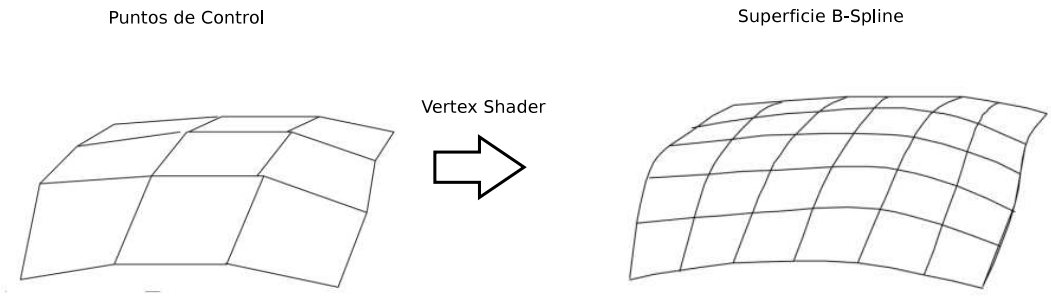


Figura 3.5: Diagrama de Bloques para el proceso de generación de la Superficie B-Spline.

3.2.2. Arquitectura del Vertex Shader

Como se explicó en el segundo capítulo, los shaders pueden ser vistos como una modificación al proceso de dibujo. En este caso, se modificará la parte en donde se procesan los vértices. Podemos visualizar al Shader como una función la cual recibirá la matriz **P**, los vectores de las bases en ambas direcciones **baseU**, **baseV**, y el vértice a transformar. Como salida se espera un vértice representando un punto evaluado de la superficie.

Una manera más simplista de ver este proceso de evaluación es ver al Shader como una caja negra la cual transforma una malla conformada de vértices en una superficie B-Spline bicúbica como lo muestra la imagen 3.5. Es importante tener presente que como entrada al Vertex Shader en general se tiene que una malla de vértices, los cuales serán transformados para generar la superficie. Finalmente el Shader evalúa la superficie como lo muestra la expresión matricial 3.2. Este shader simplemente es el encargado de multiplicar las matrices para obtener un punto de la superficie representado como vértice.

3.3. Desarrollo del Vertex Shader

En el periodo de desarrollo, se tuvieron bastantes problemas en diferentes operaciones. Uno de ellos que vale la pena mencionar en este punto, es el paso de parámetros de entrada entre la aplicación principal y el Shader. Como se vio en el capítulo 2, los recursos del GPU son a veces muy limitados para algunas operaciones.

El problema que se tuvo, hablando del paso de parámetros, fue la manera de pasar la matriz **P**, los vectores **A** y **B**. Dado que en el lenguaje CG de Nvidia para shaders, no cuenta con apuntadores, una primera idea fue la de pasar como arreglo bidimensional la matriz **P**, la cual tendría tamaño de $4x4$, mientras que los vectores podrían fácilmente pasarse como un arreglo de tamaño 4. Sintácticamente y gramaticalmente en el lenguaje de CG no se tuvo problema alguno, es decir que el compilador de CG no generó problema alguno, el problema se presentó en el tiempo de ejecución, ya que el resultado visual no fue el esperado.

Tras un largo esfuerzo de encontrar el error, se llegó a la conclusión de que el paso de parámetros no funcionaba del todo bien, dado que el valor que el Shader poseía de las variables era nulo. Cabe mencionar, que para hacer ”debugging” en CG es muy difícil, dado que en su momento no se contaban con herramientas de depuración ni de una consola la cual al menos despliegue valores impresos por el desarrollador.

La solución a este problema fue representando cada valor de la matriz **P** como un $float3$, el cual es un tipo de dato en CG que representa un vector de 3 componentes flotantes, por lo que el shader cuenta con 16 variables como entrada de este tipo representando cada elemento de dicha matriz. De igual manera se pasaron los vectores de las bases B-Spline pero utilizando el tipo de dato $float4$ el cual es un vector con cuatro componentes flotantes, regularmente utilizados para color. Se muestra a continuación la forma de la función principal del vertex shader:

```
vertex main( vertex IN, uniform float4x4 ModelViewProj,
             uniform float3 p00,
             uniform float3 p01,
             uniform float3 p02,
             uniform float3 p03,
             uniform float3 p10,
             uniform float3 p11,
             uniform float3 p12,
             uniform float3 p13,
             uniform float3 p20,
             uniform float3 p21,
             uniform float3 p22,
             uniform float3 p23,
             uniform float3 p30,
             uniform float3 p31,
             uniform float3 p32,
             uniform float3 p33,
             uniform float4 baseU,//Base B-Spline U direction
             uniform float4 baseV )//Base B-Spline V direction
```

En donde *vertex* se define como:

```
struct vertex{
  float4 position : POSITION; //Vertex Variables
  float4 color    : COLOR;    //Vertex Variables
};
```

Analizando la firma de la función principal del Shader, se puede observar que contiene las entradas descritas anteriormente. Los argumentos de la función son: *IN* que es el vértice de entrada,

el cual será transformado, $ModelViewProj$ que es la matriz de proyección, la matriz **P** se encuentra representada por aquellas 16 variables con nombre pXX y finalmente las dos bases $baseU$ y $baseV$. Un vértice posee posición y color; es por esta razón que se muestra la estructura *vertex*, para mostrar la representación de un vértice en CG. Estas porciones de código están escritos en CG, el lenguaje para desarrollar Shaders se basa mucho en C y C++, incluso el nombre CG proviene de **C for Graphics**.

Quizás surga en este momento la duda del significado de la palabra reservada **uniform**, el significado de esta palabra radica en el paso de parámetros externa, es decir de alguna aplicación ajena que invoque la ejecución del shader, por lo tanto de esta manera se indica que esos parámetros serán pasados por la aplicación principal para poder evaluar la superficie con los parámetros necesarios.

La evaluación de la superficie consta simplemente en realizar la multiplicación de matrices, tal y como se muestra en la ecuación 3.1. Existen diferentes algoritmos de multiplicación de matrices, como el de Strassen o como el convencional, y regularmente algunos de ellos pueden ser desarrollados utilizando ciclos para calcular esta operación. De primera instancia, el programar una operación de este tipo suena simple para un programador común y obviamente con un lenguaje convencional, sin embargo el lenguaje utilizado CG no cuenta con ciclos, hablando específicamente de Vertex Shaders; al parecer en otros tipos de Shader sí es posible utilizar ciclos. Es por esta razón que la implementación de la evaluación se vió obligada a ser desarrollada sin ciclos, y con más líneas de código como consecuencia inmediata. A continuación se muestra el código que realiza la multiplicación de estas matrices sin utilizar ciclos.

```
vertex OUT;


float3 row[4];


//Evaluando en dirección V
row[0] = p00*baseV[0] + p01*baseV[1] +
         p02*baseV[2] + p03*baseV[3] ;
```

```
row[1] = p10*baseV[0] + p11*baseV[1] +
         p12*baseV[2] + p13*baseV[3] ;

row[2] = p20*baseV[0] + p21*baseV[1] +
         p22*baseV[2] + p23*baseV[3] ;

row[3] = p30*baseV[0] + p31*baseV[1] +
         p32*baseV[2] + p33*baseV[3] ;

//Evaluando en dirección U
IN.position.x =   row[0].x*baseU[0] + row[1].x*baseU[1] +
                  row[2].x*baseU[2] + row[3].x*baseU[3];

IN.position.y =   row[0].y*baseU[0] + row[1].y*baseU[1] +
                  row[2].y*baseU[2] + row[3].y*baseU[3] ;

IN.position.z =   row[0].z*baseU[0] + row[1].z*baseU[1] +
                  row[2].z*baseU[2] + row[3].z*baseU[3];

//Proyectando
OUT.position  = mul(ModelViewProj, IN.position);
//Estableciendo Color Azul
OUT.color     = float4(0.0f,0.0f,0.75f,0.0f);

return OUT;
```

Como se puede observar en el código, se realiza la multiplicación de las matrices multiplicando escalar por vector del tipo $float3$ resaltando que esta operación se traduce gracias al compilador de CG. Es importante aclarar que las dimensiones de las matrices están establecidas gracias a que la superficie a generar es de grado cúbico, de otra manera se observa que si se hubiése deseado hacer un Shader capaz de evaluar cualquier superficie B-Spline, habría costado mucho trabajo, y tal vez sin muy buenos resultados, dadas las limitantes del lenguaje. Recordemos que el Vertex Shader debe cumplir el objetivo de la etapa correspondiente al proceso de dibujo, es por eso que al final del código se encuentren dos líneas las cuales establecen la posición del vector asi como también su color.

Capítulo 4

Flujo Principal y Resultados de la Aplicación

4.1. Introducción

Este capítulo explica como es que se calcula un vértice o mejor dicho un punto de la superficie B-Spline utilizando los componentes descritos en el capítulo anterior. Se presentará un diagrama de actividad, el cual explicará todo el ciclo completo para calcular un punto de la superficie, este diagrama ejemplificará todo el ciclo y además mostrará que actores intervienen y en que momento lo hacen durante el proceso.

El desarrollo que se presenta en este trabajo consta de ciertos componentes fijos que se utilizaron para generar la superficie, que serán descritos en este capítulo con el motivo de exponer que vector de nodos y que puntos de control se utilizaron. También se mostrarán los resultados obtenidos, presentando algunas imágenes y diferentes perspectivas de la superficie generada por este software.

4.2. Flujo de la Aplicación

Esta sección muestra como es que se lleva a cabo el cálculo de un vértice representando un punto de la superficie. Para comenzar se exponen el grado, los vectores de nodos y los puntos de control que se utilizaron durante el desarrollo de este trabajo.

El grado de la superficie para ambas direcciones es cúbico, por lo tanto se necesita por cada dirección 4 puntos de control, lo que implica que la malla de entrada, representando a los puntos de control **P**, será de 16 vértices.

Para comenzar el ejercicio, se muestra a continuación los vectores de nodos que fueron utilizados:

En dirección V:

$$\mathbf{V} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1/5 & 2/5 & 3/5 & 4/5 & 1 & 1 & 1 & 1 \end{bmatrix}$$

En dirección U:

$$\mathbf{U} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1/5 & 2/5 & 3/5 & 4/5 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Estos vectores de nodos son uniformes y con multiplicidad en los extremos con el motivo de generar la superficie abierta y uniforme. Se propone la siguiente matriz conteniendo los puntos de control, los cuales representan un plano, con fines de explicar el proceso:

$$\mathbf{P} = \begin{bmatrix} (0,1,0) & (0,1,1) & (0,1,2) & (0,1,3) & (0,1,4) & (0,1,5) & (0,1,6) & (0,1,7) \\ (1,1,0) & (1,1,1) & (1,1,2) & (1,1,3) & (1,1,4) & (1,1,5) & (1,1,6) & (1,1,7) \\ (2,1,0) & (2,1,1) & (2,1,2) & (2,1,3) & (2,1,4) & (2,1,5) & (2,1,6) & (2,1,7) \\ (3,1,0) & (3,1,1) & (3,1,2) & (3,1,3) & (3,1,4) & (3,1,5) & (3,1,6) & (3,1,7) \\ (4,1,0) & (4,1,1) & (4,1,2) & (4,1,3) & (4,1,4) & (4,1,5) & (4,1,6) & (4,1,7) \\ (5,1,0) & (5,1,1) & (5,1,2) & (5,1,3) & (5,1,4) & (5,1,5) & (5,1,6) & (5,1,7) \\ (6,1,0) & (6,1,1) & (6,1,2) & (6,1,3) & (6,1,4) & (6,1,5) & (6,1,6) & (6,1,7) \\ (7,1,0) & (7,1,1) & (7,1,2) & (7,1,3) & (7,1,4) & (7,1,5) & (7,1,6) & (7,1,7) \end{bmatrix}$$

Con estos datos mostrados, y suponiendo que se generará una superficie con una malla de entrada con $100x100$ vértices, se puede entonces proceder al cálculo de un vértice, en otras palabras mostrar una iteración del cálculo.

Es importante retomar la ecuación que define la evaluación de un vértice matricialmente, esta ecuación tiene como referencia el número 1.30 y su deducción de esta ecuación puede ser revisada en el primer capítulo.

$$\overrightarrow{S}(u,v) = \mathbf{A} \cdot \mathbf{P} \cdot \mathbf{B} \tag{4.1}$$

$$\mathbf{A} = \begin{bmatrix} N^p_{l_u-p}(u) & N^p_{l_u-p+1}(u) & \dots & N^p_{l_u}(u) \end{bmatrix}$$

$$\mathbf{B} = \begin{bmatrix} N^q_{l_v-q}(v) \\ N^p_{l_v-q+1}(v) \\ \vdots \\ N^q_{l_v}(v) \end{bmatrix}$$

$$\mathbf{P} = \begin{bmatrix} P_{l_u-p,l_v-q} & P_{l_u-p,l_v-q+1} & \cdots & P_{l_u-p,l_v} \\ P_{l_u-p+1,l_v-q} & P_{l_u-p+1,l_v-q+1} & \cdots & P_{l_u-p+1,l_v} \\ \vdots & \vdots & \cdots & \vdots \\ P_{l_u,l_v-q} & P_{l_u,l_v-q+1} & \cdots & P_{l_u,l_v} \end{bmatrix}$$

Recordando, estas superficies son del tipo paramétricas, por lo que a cada parámetro se le asocia una dirección en el sistema de coordenadas. De la ecuación 4.1 se observa que se necesita el valor de dos parámetros u y v, por lo que para definir un punto de esta superficie es necesario especificar el valor de estas dos variables, supongamos que $u = 1/2$ y que $v = 1/3$, lo que significa estar evaluando $\overrightarrow{S}(1/2, 1/3)$.

Esta evaluación implica calcular tres matrices, la matriz **A** y **B** que representan los coeficientes de las bases B-Spline, que en realidad son muestras de las funciones B-Spline que fueron explicadas en el primer capítulo, y la matriz **P** que representa una submatriz de los puntos de control; esta matriz contiene aquellos puntos que forman parte de la vecindad de los puntos de control que aportan a la generación del punto de la superficie respectivo a los parámetros.

El proceso comienza con el cálculo de las matrices relacionadas con las bases. La matriz **A** tiene como vector de nodos al vector **U** y la matriz **B** tiene como vector de nodos al vector **V**. Si se comienza con la matriz **A** y se analiza la definición de esta matriz, notamos que debemos de encontrar el valor l con el algoritmo modificado de la búsqueda binaria, para encontrar el intervalo que le corresponde como se explicó en el capítulo anterior. Por lo tanto, se encuentra el punto en donde el método *indexOf* de la clase **KnotVector** debe ser llamado. Hay que recordar que al momento de la construcción de alguna instancia de la clase KnotVector se llama el cálculo en donde se determinan los índices *lower* y *upper*, por lo tanto los valores de estos atributos son $lower = 3$ y $upper = 8$; es importante recalcar que esto es necesario dado que en el cálculo de la base se pueden llegar a presentarse casos en donde el denominador de alguna división puede ser cero.

Si se retoma el cálculo de la matriz **A**, observamos que el primer elemento de esta matriz es $N^p_{l_u}(u)$, por lo tanto se debe encontrar entonces el valor de l_u, para lo cual llamaremos al método del

vector de nodos *u* indexOf el cual regresará el índice en donde la condición del intervalo se cumple. Éste índice será necesario para la llamada a la función llamada *computeBasis*, la cual debe regresar el valor del primer elemento. El tamaño de la matriz **A** es de $1x4$ dado que el grado $p = 3$, por lo tanto se deben calcular 3 elementos más, para completar el tamaño de la matriz, de tal forma que la matriz **A** queda como se muestra a continuación:

$$\mathbf{A} = \begin{bmatrix} N^3_{5-3}(1/2) & N^3_{5-3+1}(1/2) & N^3_{5-3+2}(1/2) & N^3_{5-3+3}(1/2) \end{bmatrix}$$

Evaluando índices resulta:

$$\mathbf{A} = \begin{bmatrix} N^3_2(1/2) & N^3_3(1/2) & N^3_4(1/2) & N^3_5(1/2) \end{bmatrix}$$

y evaluando la función de la base respectivamente finalmente para $u = 1/2$ la matriz **A** queda de la siguiente forma:

$$\mathbf{A} = \begin{bmatrix} 0{,}020833 & 0{,}479167 & 0{,}479167 & 0{,}020833 \end{bmatrix}$$

Para la matriz **B** se repite el mismo procedimiento, solamente teniendo en cuenta el valor de la variable *v*. El índice que regresa el método *indexOf* para el valor de *v* es $l_v = 4$. Con este valor se puede comenzar el cálculo de las bases como se muestra a continuación:

$$\mathbf{B} = \begin{bmatrix} N^3_{4-3}(1/3) & N^3_{4-3+1}(1/3) & N^3_{4-3+2}(1/3) & N^3_{4-3+3}(1/3) \end{bmatrix}^{\mathbf{T}}$$

Evaluando índices resulta:

$$\mathbf{B} = \begin{bmatrix} N^3_1(1/3) & N^3_2(1/3) & N^3_3(1/3) & N^3_4(1/3) \end{bmatrix}^{\mathbf{T}}$$

y evaluando la función finalmente para $v = 1/3$ la matriz **B** queda de la siguiente forma:

$$\mathbf{B} = \begin{bmatrix} 0{,}049346 & 0{,}009273 & 0{,}367391 & 0{,}573991 \end{bmatrix}^{\mathbf{T}}$$

Resta calcular la matriz local, la cual contiene a los puntos que influyen en el cálculo de este punto. Para calcular esta matriz, se debe llamar al método *computeLocalMatrix*. Éste método calcula una matriz con la siguiente forma:

$$\mathbf{P} = \begin{bmatrix} P_{l_u-3,l_v-3} & P_{l_u-3,l_v-2} & P_{l_u-3,l_v-1} & P_{l_u-3,l_v} \\ P_{l_u-2,l_v-3} & P_{l_u-2,l_v-2} & P_{l_u-2,l_v-1} & P_{l_u-2,l_v} \\ P_{l_u-1,l_v-3} & P_{l_u-1,l_v-2} & P_{l_u-1,l_v-1} & P_{l_u-1,l_v} \\ P_{l_u,l_v-3} & P_{l_u,l_v-2} & P_{l_u,l_v-1} & P_{l_u,l_v} \end{bmatrix}$$

donde, P_{l_u,l_v} representa un elemento de la matriz que contiene a los puntos de control. Por lo tanto, sustituyendo las variables $l_u = 5$ y $l_v = 4$ se obtiene la siguiente matriz:

$$\mathbf{P} = \begin{bmatrix} P_{2,1} & P_{2,2} & P_{2,3} & P_{2,4} \\ P_{3,1} & P_{3,2} & P_{3,3} & P_{3,4} \\ P_{4,1} & P_{4,2} & P_{4,3} & P_{4,4} \\ P_{5,1} & P_{5,2} & P_{5,3} & P_{5,4} \end{bmatrix}$$

que sustituyendo los puntos correspondientes

$$\mathbf{P} = \begin{bmatrix} (2,1,1) & (2,1,2) & (2,1,3) & (2,1,4) \\ (3,1,1) & (3,1,2) & (3,1,3) & (3,1,4) \\ (4,1,1) & (4,1,2) & (4,1,3) & (4,1,4) \\ (5,1,1) & (5,1,2) & (5,1,3) & (5,1,4) \end{bmatrix}$$

Hasta este punto el encargado de calcular todas estas matrices es el CPU. Para este momento se tienen todos los componentes listos para evaluar la superficie, y como se explicó en el capítulo anterior el encargado de evaluar la superficie es el Vertex Shader. Por lo tanto, la evaluación se reduce a multiplicar las matrices.

Si se evalúa la primer multiplicación, se obtiene la siguiente matriz **C**

$$\mathbf{C} = \mathbf{P} \cdot \mathbf{B}$$

$$\mathbf{B} = \begin{bmatrix} 0{,}049346 & 0{,}009273 & 0{,}367391 & 0{,}573991 \end{bmatrix}^{\mathbf{T}}$$

$$\mathbf{P} = \begin{bmatrix} (2,1,1) & (2,1,2) & (2,1,3) & (2,1,4) \\ (3,1,1) & (3,1,2) & (3,1,3) & (3,1,4) \\ (4,1,1) & (4,1,2) & (4,1,3) & (4,1,4) \\ (5,1,1) & (5,1,2) & (5,1,3) & (5,1,4) \end{bmatrix}$$

$$\mathbf{C} = \begin{bmatrix} (2,1,3{,}466029) \\ (3,1,3{,}466029) \\ (4,1,3{,}466029) \\ (5,1,3{,}466029) \end{bmatrix}$$

La matriz **C** representa la evaluación en la dirección *v*, por lo tanto realizando la última evaluación, que representa la evaluación en la dirección *u* se obtiene

$$\overrightarrow{S}(\mathbf{u}, \mathbf{v}) = \mathbf{A} \cdot \mathbf{C}$$

$$\overrightarrow{S}(\mathbf{1/2}, \mathbf{1/3}) = \begin{bmatrix} 0{,}020833 & 0{,}479167 & 0{,}479167 & 0{,}020833 \end{bmatrix} \cdot \begin{bmatrix} (2,1,3{,}466029) \\ (3,1,3{,}466029) \\ (4,1,3{,}466029) \\ (5,1,3{,}466029) \end{bmatrix}$$

$$\overrightarrow{S}(\mathbf{1/2}, \mathbf{1/3}) = (\mathbf{3{,}5}, \mathbf{1}, \mathbf{3{,}466029})$$

Finalmente, después de todo este proceso se obtuvo un punto de la superficie correspondiente a los parámetros u y v. Es importante mencionar que para cada combinación de nuestros parámetros debe existir un vértice de la malla de entrada (u, v) a transformar, dado que se trabaja con un espacio discreto. Se puede visualizar este proceso como una transformación de la malla entrante a una superficie, además hay que recordar que para usar Vertex Shaders se debe operar sobre vértices, lo que implica transformar u operar vértice por vértice. Si se repite este proceso por cada vértice que conforma la malla de entrada, se obtendrá la superficie B-Spline bicúbica uniforme.

A continuación se presenta un diagrama de actividad en la figura 4.1, que puede verse como un diagrama de flujo, el cual representa el proceso de cálculo de toda la superficie identificando que actor ejecuta cada parte del proceso.

4.3. Resultados en Tiempo Real

Esta sección muestra algunas tomas de la superficie generada con la aplicación desarrollada. Cabe mencionar en este punto una ventaja de utilizar Vertex Shader. El modificar en tiempo real la malla de entrada al Vertex Shader implicaría generar diferentes superficies de acuerdo a los puntos de control en ese instante, es decir que si se deseara generar una forma diferente en tiempo real de la superficie, implicaría modificar los puntos de control. Esto permitiría formar diferentes formas en distintos tiempos según el usuario lo desee. Este tipo de aplicaciones, podrían utilizarse en efectos de oleaje de agua, o superficies moldeables en tiempo real para simulaciones de cualquier tipo.

A continuación se muestra una imagen la cual muestra una Superficie B-Spline bicúbica, con los mismos vectores de nodos utilizados en este capítulo, los puntos de control utilizados no formaban parte de un plano para ese instánte en el que se tomó la imagen. En la figura 4.2 podemos observar los puntos de control de color Rojo, mientras que los vértices transformados se presentan en color Azul.

La malla de entrada, la cual se observa en azul, consta de 4096 vértices, lo cual significa que por cada lado de esta malla se tienen 64 vértices. Es muy evidente el pesado cálculo que se tiene

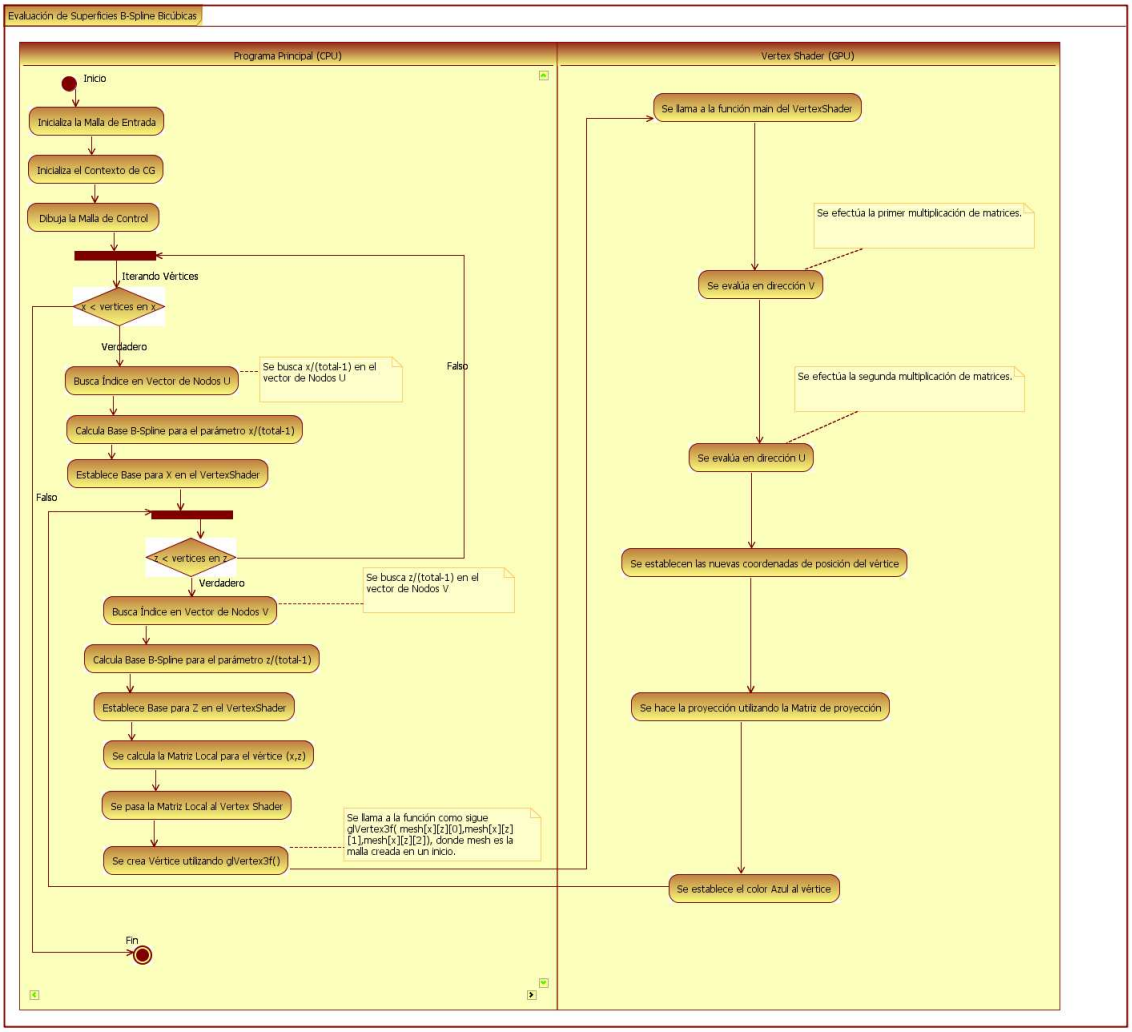


Figura 4.1: Diagrama de Actividad de la evaluación de la Superficie B-Spline Bicúbica.

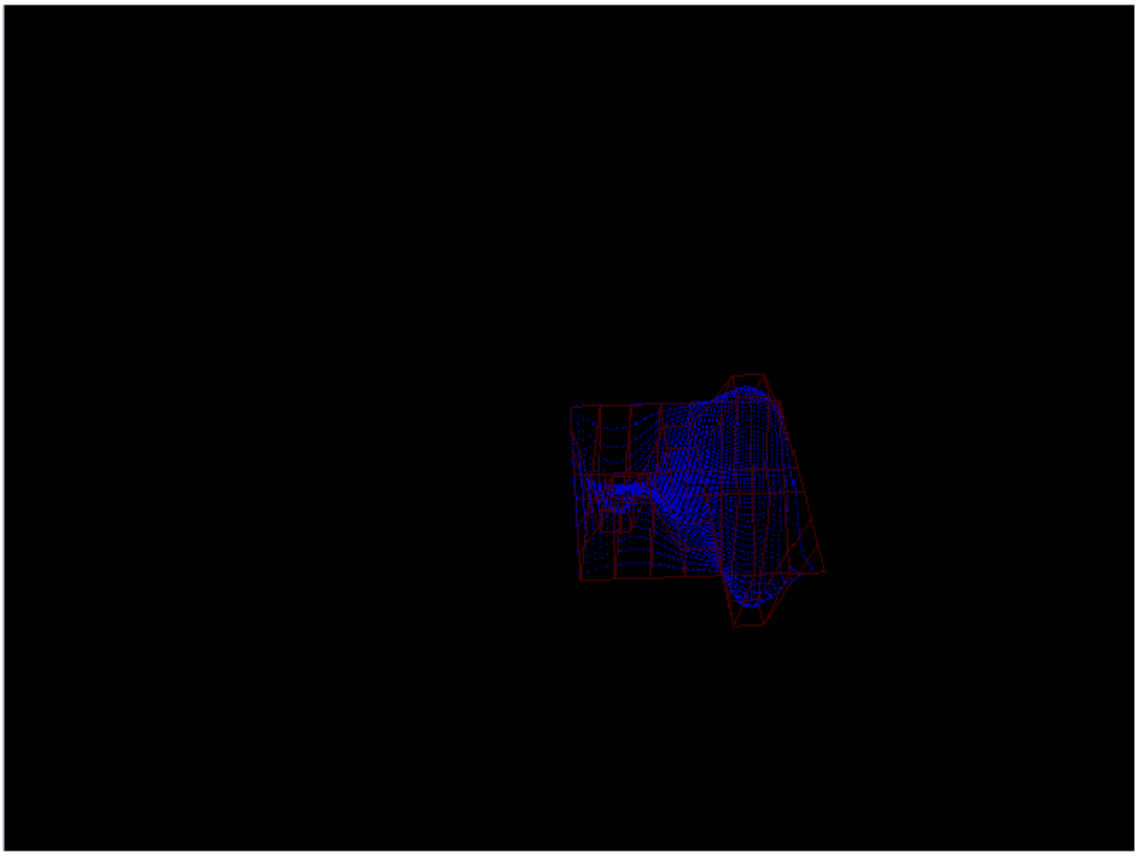


Figura 4.2: Superficie B-Spline Bicúbica con puntos de control en Rojo.

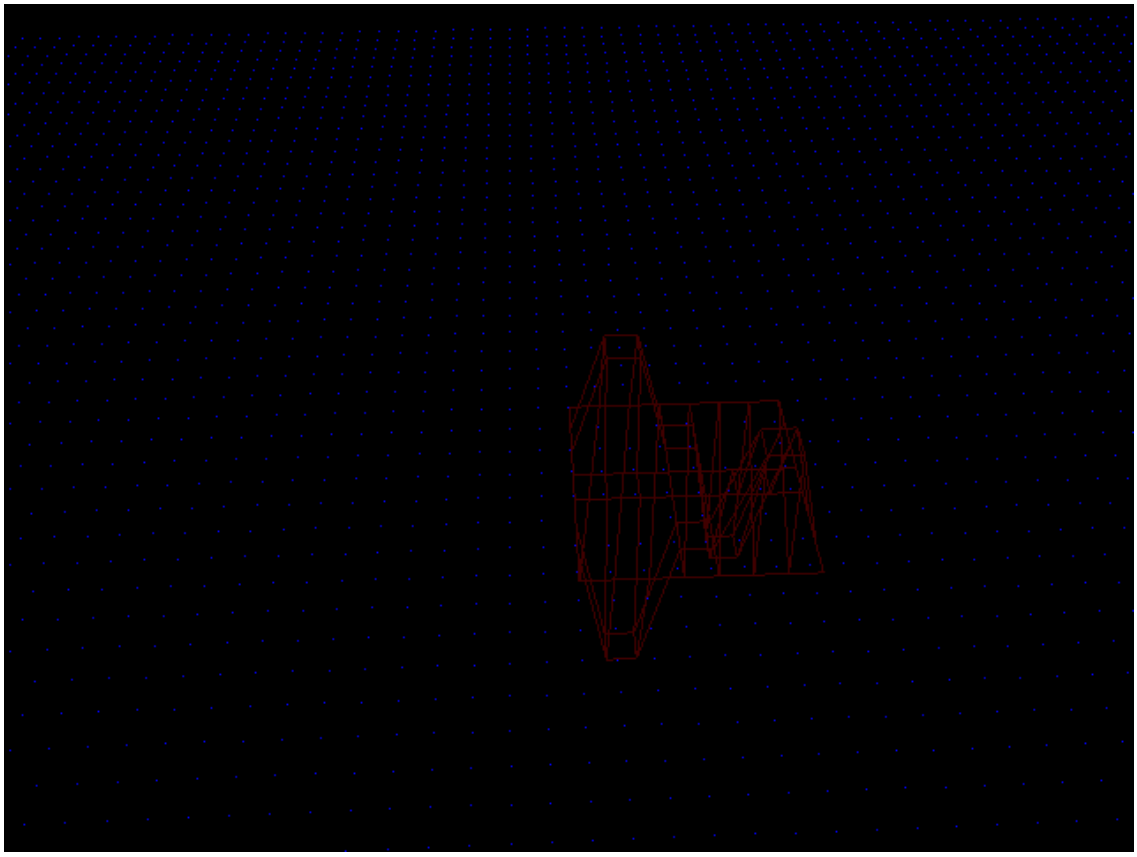


Figura 4.3: Puntos de control en Rojo y la malla de entrada

que realizar para calcular esta superficie. La computadora debe realizar todo el cálculo descrito en la sección anterior por cada vértice de la malla de entrada.

Este pesado cálculo como se puede deducir, es reducido si dos entes colaboran entre sí para distribuirse la carga de distintos procesos para la evaluación de la superficie, en nuestro caso el GPU y el CPU. En la figura 4.3 se presenta la malla de entrada utilizada en la aplicación asi como la malla formada por los puntos de control en rojo.

En la siguiente figura 4.4, se alcanza a apreciar mejor las uniones entre los parches de Bèzier, en esta figura se alcanzan a notar perfectamente 4 parches que en conjunto forman la superficie B-Spline. La aplicación que genera estas superficies, a propósito modifica aquellos puntos centrales de cada parche para mostrar que una superficie B-Spline no es mas que una superficie conformada

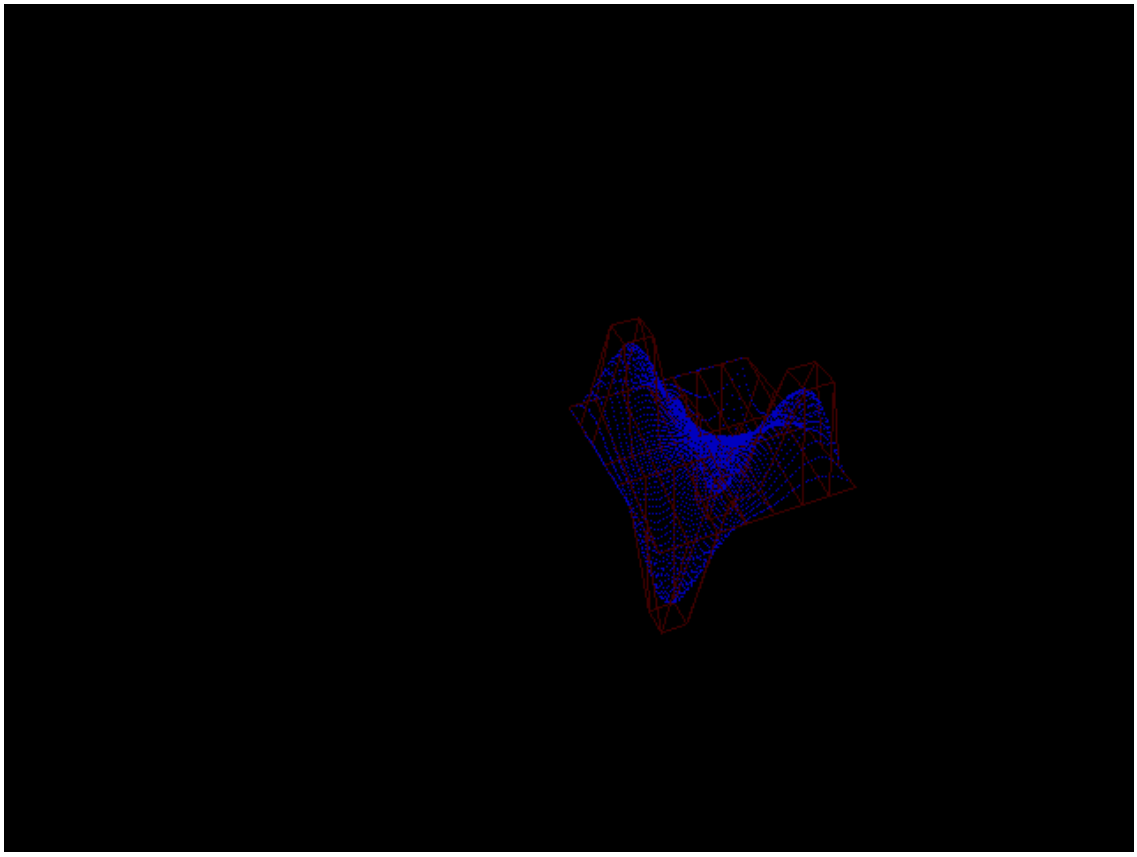


Figura 4.4: Superficie B-Spline con 4 parches de Bèzier.

de distintos parches, y mostrando así la propiedad de localidad.

La aplicación que evaluó estas superficies, constantemente modifica los puntos de control, logrando con esto un efecto bastante interesante. Las modificaciones a los puntos de control fueron hechos de tal manera que la evaluación de dicha superficie presuma sus parches de Bèzier, como ya se habia mencionado.

Para finalizar este trabajo, se comentan algunas ventajas de utilizar superficies B-Spline. Principalmente en aquellos desarrollos de Software encargados en CAD, los cuales utilizan bastante este tipo de curvas y superficies, para desarrollar formas deseadas en modelos de diseño. Existen algunas técnicas para el procesamiento de imágenes con curvas o bases B-Spline. En la rama de procesamiento de señales se utilizan también estas curvas para algunos filtros.

Las curvas B-Spline presentan bastante utilidad en distinas ramas, gracias a diferentes propiedades y su fácil control para generarlas. Muchas técnicas de interpolación utilizan esta matemática para aproximar o para unir puntos polinomialmente. Gracias a esto se concluye que las curvas y superficies B-Spline resultan muy útiles para distintas ramas de la ciencia e ingeniería.

Bibliografía

[1] Abe Winter Austin Robison. An overview of graphics processing hardware. Recuperación Septiembre 2008 http://people.cs.uchicago.edu/~robison/src/gpu_paper.pdf.

[2] Martin Ecker. Xengine programmable graphics pipeline architectures. Recuperación Mayo 2008 http://xengine.sourceforge.net.

[3] James D. Foley. *Computer Graphics Principles and Practice*. Addison-Wesley, 1997.

[4] Julio Fernández Hermida. Modelado geométrico de curvas. Recuperación Marzo 2008 http://webs.uvigo.es/xuliofh/Web-TMM/Clase8Splines/c4\%20cur\%20sp2.pdf.

[5] Worcester Polytechnic Institute. Opengl overview. Recuperación Abril 2008 http://web.cs.wpi.edu/~matt/courses/cs563/talks/OpenGL_Presentation/OpenGL_Presentation.html.

[6] Wayne Tiller Les A. Piegl. *The NURBS Book*. Springer, 1997.

[7] Lighthouse3d. Glsl tutorial. Recuperación Abril 2008 http://www.lighthouse3d.com/opengl/glsl/index.php?pipeline.

[8] Lighthouse3d. Glsl tutorial. Recuperación Abril 2008 http://www.lighthouse3d.com/opengl/glsl/index.php?vertexp.

[9] John Owens. Gpu architecture overview. Recuperación Abril 2008 http://www.gpgpu.org/s2007/slides/02-gpu-architecture-overview-s07.pdf.

[10] Leslie Ikemoto Pushkar Joshi. Harnessing the gpu for general purpose computation: A case study. Recuperación Abril 2008 http://www.cs.berkeley.edu/%7Eppj/publications/gpu_paper.pdf.

[11] Barthold Lichtenbelt Randi J. Rost, John M. Kessenich. *OpenGL Shading Language*. Addison-Wesley, 2004.

[12] Mark J. Kilgard Randima Fernando. *The Cg Tutorial: The Definitive Guide to Programmable Real-time Graphics*. Addison-Wesley, 2003.

[13] David F. Rogers. *An Introduction to NURBS: With Historical Perspective*. Morgan Kaufmann Publishiers, 2001.

[14] Dr. Ching-Kuang Shene. Computing with geometry course notes. Recuperación Marzo 2008 http://www.cs.mtu.edu/~shene/COURSES/cs3621/NOTES/curves/curves.html.

[15] Dr. Ching-Kuang Shene. Computing with geometry course notes. Recuperación Marzo 2008 http://www.cs.mtu.edu/~shene/COURSES/cs3621/NOTES/surface/basic.html.

[16] Dr. Ching-Kuang Shene. Computing with geometry course notes. Recuperación Abril 2008 http://www.cs.mtu.edu/~shene/COURSES/cs3621/NOTES/spline/B-spline/bspline-motiv.html.

[17] Dr. Ching-Kuang Shene. Computing with geometry course notes. Recuperación Abril 2008 http://www.cs.mtu.edu/~shene/COURSES/cs3621/NOTES/spline/B-spline/bspline-basis.html.

[18] Dr. Ching-Kuang Shene. Computing with geometry course notes. Recuperación Marzo 2008 http://www.cs.mtu.edu/~shene/COURSES/cs3621/NOTES/spline/B-spline/bspline-curve-prop.html.

[19] Helmut Späth. *One Dimensional Spline Interpolation algorithms*. AK Peters, Ltd, 1995.

[20] Ronald L. Rivest Clifford Stein Thomas H Cormen, Charles E. Leiserson. *Introduction to Algorithms, Second Edition*. The MIT Press, 2003.

[21] Eric W. Weisstein. Bernstein polynomial. From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/BernsteinPolynomial.html.

[22] Wikipedia. De boor algorithm. Recuperación Abril 2008 http://en.wikipedia.org/wiki/De_Boor's_algorithm.

[23] Wikipedia. Spline. Recuperación Marzo 2008 http://en.wikipedia.org/wiki/Spline_(mathematics).

[24] Niklaus Wirth. *Algorithms and Data Structures*. Prentice-Hall, 1986.