Capítulo 3: Implementación y programación del software para visualizar

El software fue diseñado para poder cargar en memoria el campo de distancia acotado y voxelizado de un modelo tridimensional de una cabeza. Ya cargado el campo de distancia se superpondra el modelo de otra cabeza el cual sufrirá un texturizado en tiempo real dependiendo de la posición de cada vértice. El software permite que el modelo de la cabeza se pueda trasladar y rotar utilizando el botón izquierdo del mouse y asi mismo reducir el error extrínseco por alineación y facilitando dicha alineación. Con una regla de correspondencia entre la textura, el máximo y el mínimo valor de distancia del campo a este le asigna un color, con ello el vertice más cercano a ese voxel se texturiza con el color correspondiente.

La cámara se puede manipular con el botón derecho del mouse para poder observar cualquier ángulo de la escena. Tiene varios controles del lado derecho para modificar tanto la iluminación como el tamaño del campo de distancia el cual se necesita calibrar de acuerdo al tamaño real en centímetros (Figura 34).

Las diferencias simétricas de ambos modelos se verán en color azul cuando el vértice de la superficie del modelo esté en el interior de la superficie de referencia (distancia negativa), el color será rojo cuando el vértice esté fuera del modelo. Se usa una superfice plana compuesta por varios vértices para mostrar un corte transversal del campo de distancia.

Cuando a un vértice se le asigna un color éste tiene consigo un valor de distancia, en que se interpreta como un error. Si el vértice es texturizado con blanco entonces el error tiende a ser 0. El error total es el promedio de la distancia mínima que existe en cada vértice de un población muestral del modelo dentro del campo de distancia (se toman unicamente los vértices de la superficie de la cabeza en estudio)

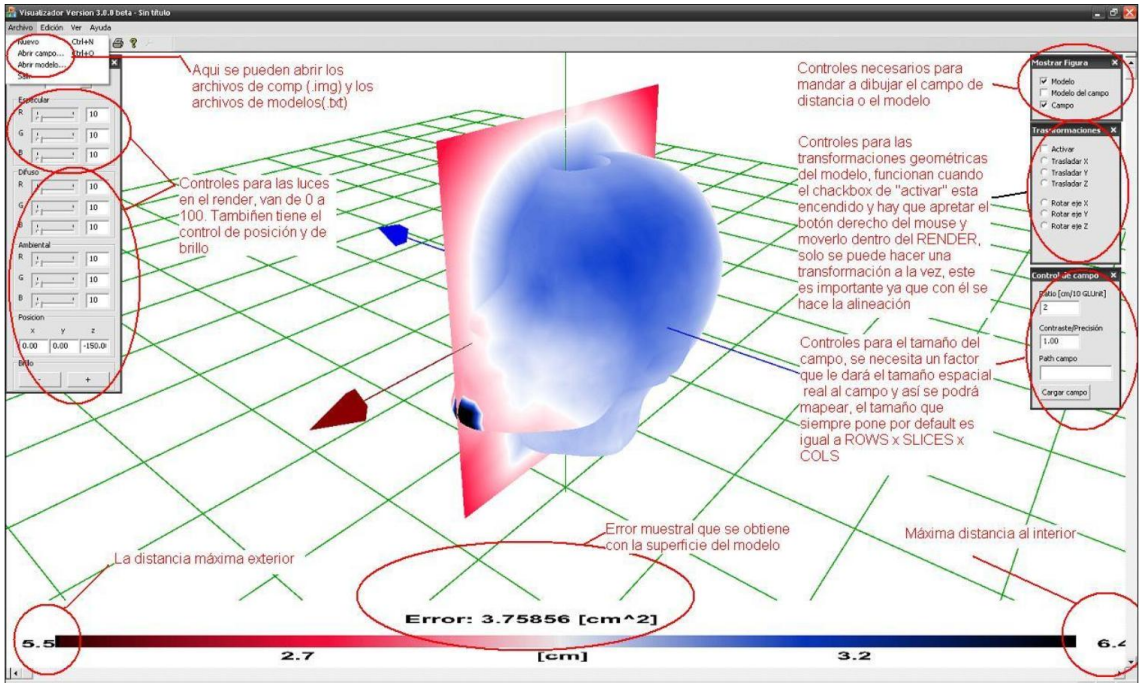

Figura 34: Descripción de la apliación

3.1.- Programación y diseño de la interfaz gráfica

El proyecto fue desarrollado como un híbrido de programación estructurada y de programación orientada a objetos y eventos. La parte estructurada es en la configuración del RENDER y su ejecución así como los tipos de datos a utilizar (estructuras). En general son 3 archivos el corazón de todo el proyecto:

- WinRender.h
- WinRenderRecursos.h
- WinRender.cpp

En WinRender.h estan incluidas todas las definiciones de las funciones de WinRender.cpp, en WinRender.cpp están las declaraciones y en WinRenderRecursos.h se encuentran todas las estructuras, numeraciones, tipos de datos etc. El resto del programa está orientado a objetos y se divide en 2 partes:

- Clases de gráficos para OpenGL
- Clases de interface (generado por el IDE de VisualStudio)

En la gráfica (proyecto bloques) se ilustra como interactuan los objetos, eventos y el renderizado.

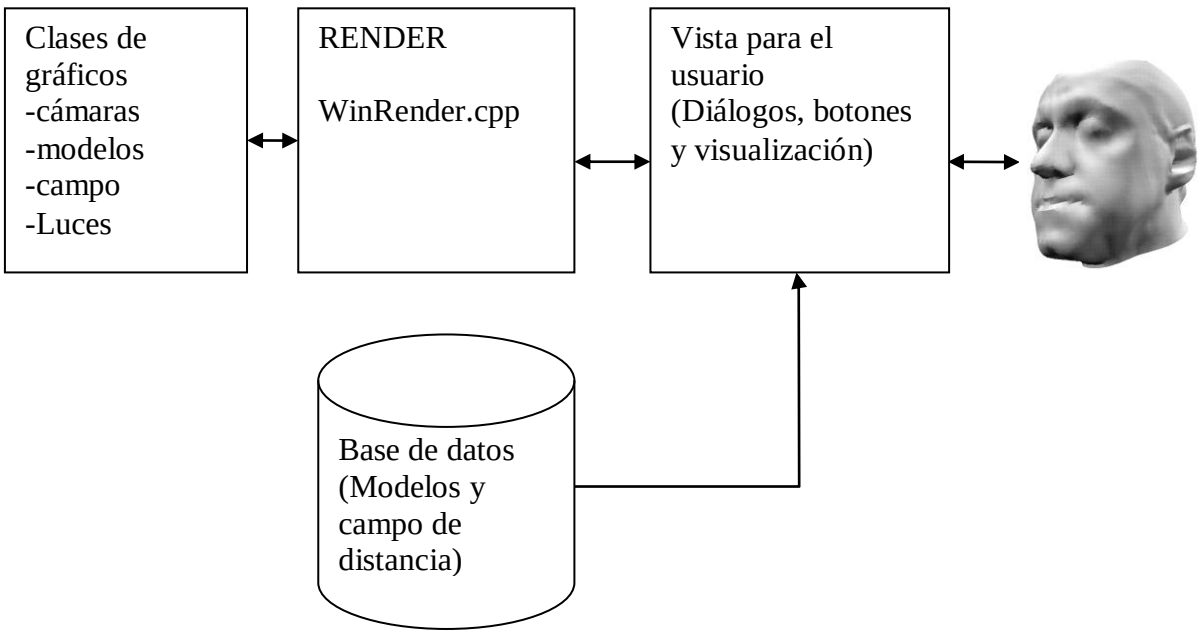

Figura 35: Diagrama de bloques del sistema gráfico

Las clases de gráficos son las que describen el comportamiento y datos de lo que se va a graficar. La clase “camaras” tiene los siguentes atributos y métodos:

```
private:
      Vector            Posicion;
      Vector            Up;
      Vector            Objetivo;
      RectanPuertoV     PuertoVista;
      bool              MouseButtonDown;
      float             alfa;
      float             beta;
      float             zoom;
      int               x1;
      int               x2;
      int               y1;
      int               y2;
      float             r;

public:
      Camaras(void);
      Camaras(float pvx, float pvy, float pvDeltax, float
pvDeltay);
      void RielEsferico(float Distancia);
      void RielEsferico();
      void Pan(int x, int y);
      void AngulosEsfericos(float x, float y);
      void Accion(void);
      void Zoom(void);

      void setZoom(float z);
      void setPosicion(float x, float y, float z);
      void setUp(float x, float y, float z);
      void setObjetivo(float x, float y, float z);
      void setPuertoVista(float x, float y, float dx, float dy);
      void setMouseButtonDown(bool setMBD);
      void setX1Y1(int X1,int Y1);
      void setR(float zDelta);

      Vector            *getPosicion(void);
      Vector            *getUp(void);
      Vector            *getObjetivo(void);
      RectanPuertoV     *getPuertoVista(void);
      float             getZoom(void);
```

```
bool              getMouseButtonDown(void);
float             getR(void);
```

Cada clase se divide en dos partes: una privada que son los atributos donde se almacenaran los datos y una parte pública donde van todas las funciones. La cámara por defecto estará observando el origen de todo el sistema de referencia y su movimiento es sobre una superficie esférica como se ve en la figura (Superficie de movimiento de cámara)

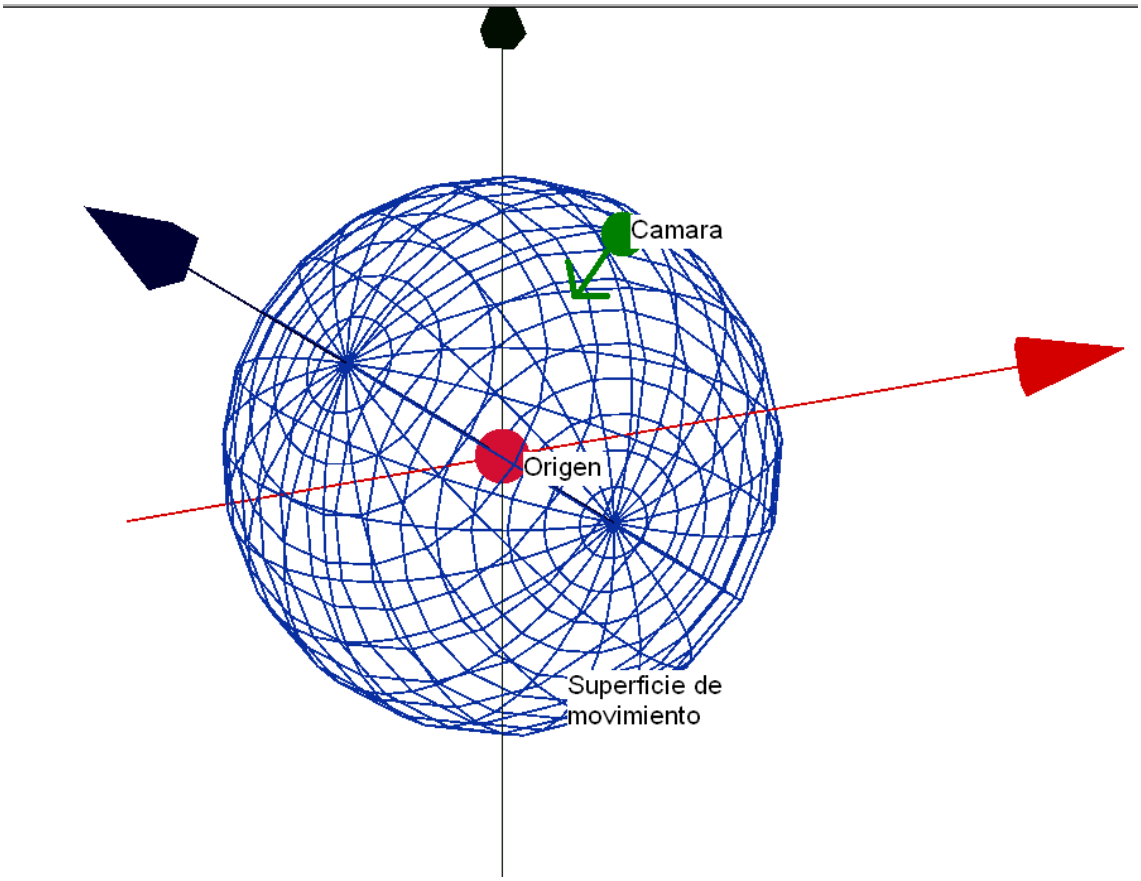

Figura 36: Superficie de movimiento de la cámara y siempre apuntando al origen del sistema coordenado

Entonces la cámara se mueve libremente por una superficie esférica usando una transformación de coordenadas esféricas a cartesianas y viceversa

$$x = r\cos(\alpha) \quad (39)$$

$$y = r\sin(\beta) \quad (40)$$

$$z = r\sin(\alpha)\cos(\beta) \quad (41)$$

```
Posicion.xi = Distancia * cos(alfa*PI/180);
Posicion.yi = Distancia * sin(beta*PI/180);
Posicion.zi = Distancia * sin(alfa*PI/180)*cos(beta*PI/180);
```

La cámara se manipula con el movimiento del mouse combinado con el botón izquierdo y la rueda central. Todos los movimientos de la cámara dependen de los eventos del mouse dentro del render. En general la parte de eventos para manipular las cámaras y toda transformación del modelo se generan en el bloque de Vista para el usuario con las siguientes ecuaciones:

$$\vec{v}_m = x_0 \hat{i} + y_0 \, j \qquad (42)$$

$$\overrightarrow{\Delta v}_{m0} = (x - x_0)\hat{i} - (y - y_0)\, j \qquad (43)$$

$$\overrightarrow{\Delta v}_{m0} = \Delta x_0 \hat{i} + \Delta y_0 \, j \qquad (44)$$

Utilizando una acumulación para Δx

$$\Delta x_1 = x - \Delta x_0$$

$$\Delta x_2 = x - \Delta x_1$$

.

.

.

$$\Delta x_i = x - \Delta x_{i-1} \qquad (46)$$

De igual forma con Δy

$$\Delta y_i = x - \Delta y_{i-1} \qquad (47)$$

$$\Delta x_i = x - \Delta x_{i-1} \qquad (48)$$

$$\overrightarrow{\Delta v}_{m0} = \Delta x_i \hat{i} + \Delta y_i \, \widehat{j} \qquad \mathrm{Dom}\{x, y \in \mathbb{Z}\} \qquad (49)$$

Estas ecuaciones describen los cambios que hay con respecto a la posición del puntero de mouse dentro del área de render (figura 36)

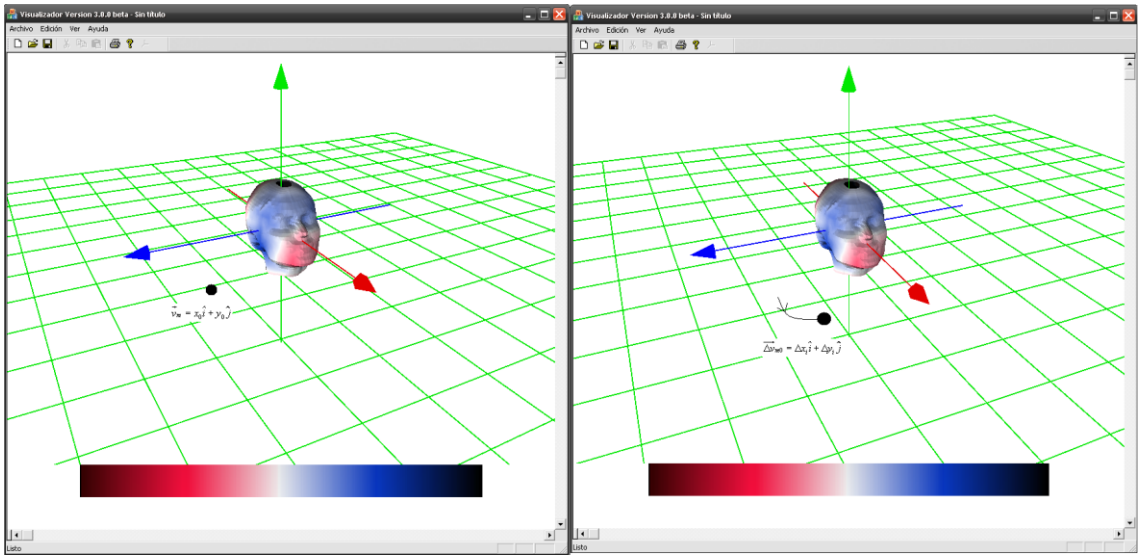

Figura 37: rotación de la cámara

La clase de modelos utiliza principalmente 3 atributos esenciales y las funciones de transformación así como las de las normales y de texturizado; los atributos son 3 arreglos de vértices que son los índices, los vértices y las normales de cada triangulación. Las funciones de transformación son de rotación y traslación en cada uno de los ejes para así completar los 6 grados de libertad para alinear la cabeza conforme al campo de distancia, el comportamiento de estas funciones es semejante al de la cámara, cuando hay un evento de click derecho del mouse y activado el checkbox de transformación realiza la transformación elegida muy similar a los programas de edición de modelos 3D tales como *3D StudioMax, Blender etc*.

```
class Modelos
{
private:
	Vertice		VertexArray [4000];
	Normal		NormalArray [4000*3];
	Indice		IndexArray  [4000*3];
	Centroide	centroide;
	Vector		Origen;

	int		nTriangulos;
	int		nVertices;
	int		GLModo;
	bool		mouseButtonDown;

public:
```

```
                    Modelos(void);
      bool        Recorte(Vertice *v1,Vertice *v2, Vertice *v3,
Vector *camaraPos);
      void        Dibujar(Vector *camaraPos);
      void        Dibujar(Vector *camaraPos,float
***campo,PropiedadesCampo pc);
      void        Texturizar(float x, float y, float z);
      void        Texturizar(float x, float y, float z, float
***campo, PropiedadesCampo pc);
      int               CargarModelo(char *modelo,char *indice);
      void        CalculaCentroide(void);
      void        Rotar(float angulo,int x,int y,int z);
      void        Trasladar(float x,float y,float z);
      void        Escalar(float x, float y, float z);
      void        CalculaNormales(void);
      void        dibujarEjesRotacion(Vector *camaraPos);
      void        dibujarEjesTraslacion(void);
      void        dibujarEjesEscalamiento(Vector *camaraPos);
      bool        iluminaEjeTraslacion(Vector
*camaraPos,RectanPuertoV *pv,int x, int y);
      void        trasladarConPunteroX(float x);
      void        trasladarConPunteroY(float y);
      void        trasladarConPunteroZ(float Z);
      void        trasladarX(void);
      void        trasladarY(void);
      void        trasladarZ(void);
      void        rotarConPunteroX(float x);
      void        rotarConPunteroY(float y);
      void        rotarConPunteroZ(float Z);
      void        rotarX(void);
      void        rotarY(void);
      void        rotarZ(void);



      Centroide   *getCentroide(void);
      Vertice           *getVertexArray(void);
      Normal            *getNormalArray(void);
      Indice            *getIndexArray(void);
      int               getnTriangulos(void);
```

```
        int                 getGLModo(void);
        bool          getMouseButtonDown(void);

        float         getTx1(void);
        float         getTy1(void);

public:
        ~Modelos(void);
};
```

La función de texturizar es la función que nos va a dar el mapeo del modelo de acuerdo con el campo de distancia, para esto se necesita leer el volumen de campo el cual es un arreglo de 3 dimensiones que pertenece a la clase de Campo.

La clase Campo es muy sencilla, tan solo tiene un arreglo de 3 dimensiones para guardar el campo y unas funciones para leerlo desde un archivo.

Para la interpretación del campo de distancia primero se realiza una voxzelización esto significa que se muestrea en 3 dimensiones y se guarda en un arreglo (es el equivalente 3D de una *pixelización*). El arreglo se encuentra en formato denominado "acceso de índice marginal" [FOLEY1992] y se implementa por un triple apuntador al cual debe asignarse sucesivamente memoria para listas de arreglos de apuntadores:

```
float       ***volCampo;
```

Como ejemplo de uso, véase el código de la función `alloc_float_volMem()`, de la sección 1.1.2.- DevC++.

3.2- Campo de distancia en memoria

En C la declaración anterior significa que es un apuntador a apuntadores de apuntadores de tipo flotante, en la memoria de la computadora, una vez asignada la memoria, se interpretaría como se muestra en la figura 37.

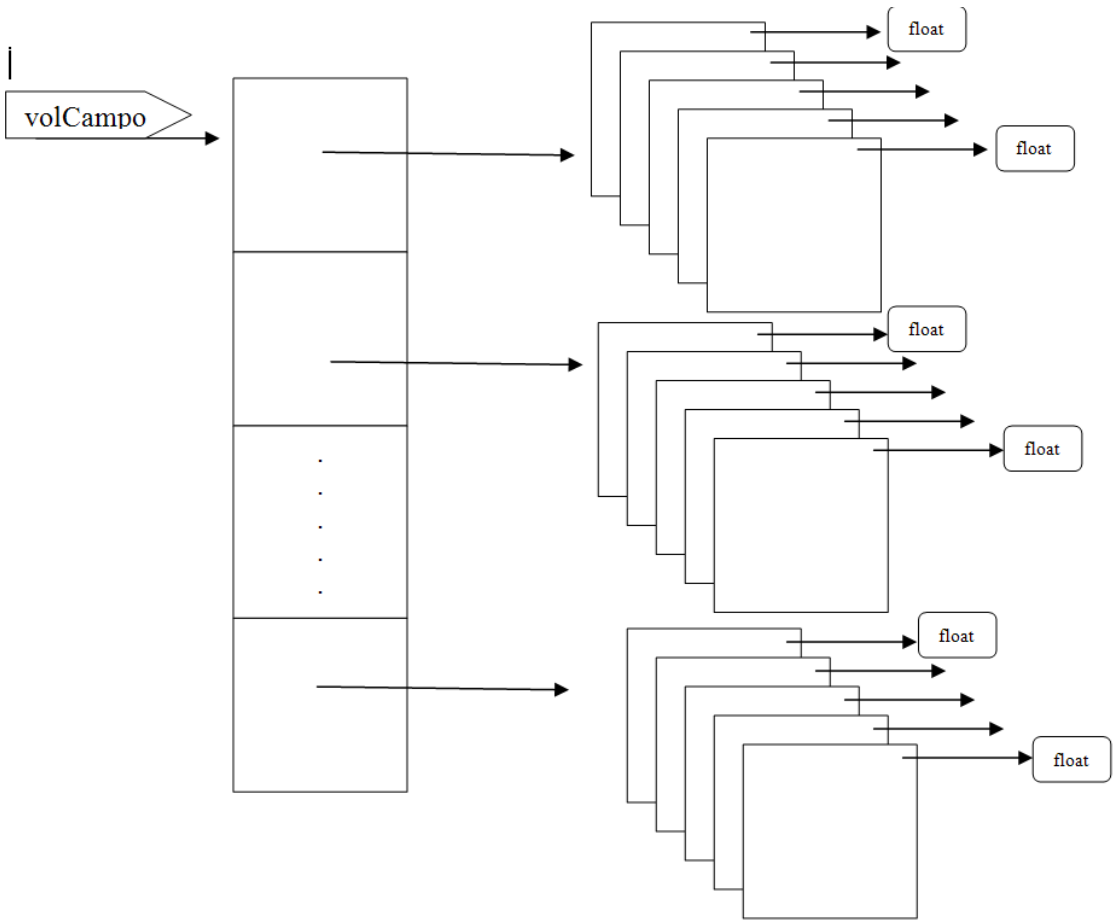

Figura 38: estructura de datos de apuntadores a apuntadores para la voxzelización. Los últimos apuntadores son a listas de floats correspondientes a renglones

Entonces, ya en memoria, el volumen de campo de distancia cada valor float es un valor de distancia que hay en el espacio de la figura que genera el campo. El campo se interpreta entonces como una posición en x, y, z de toda la escena como se muestra en la figura (voxzelización)

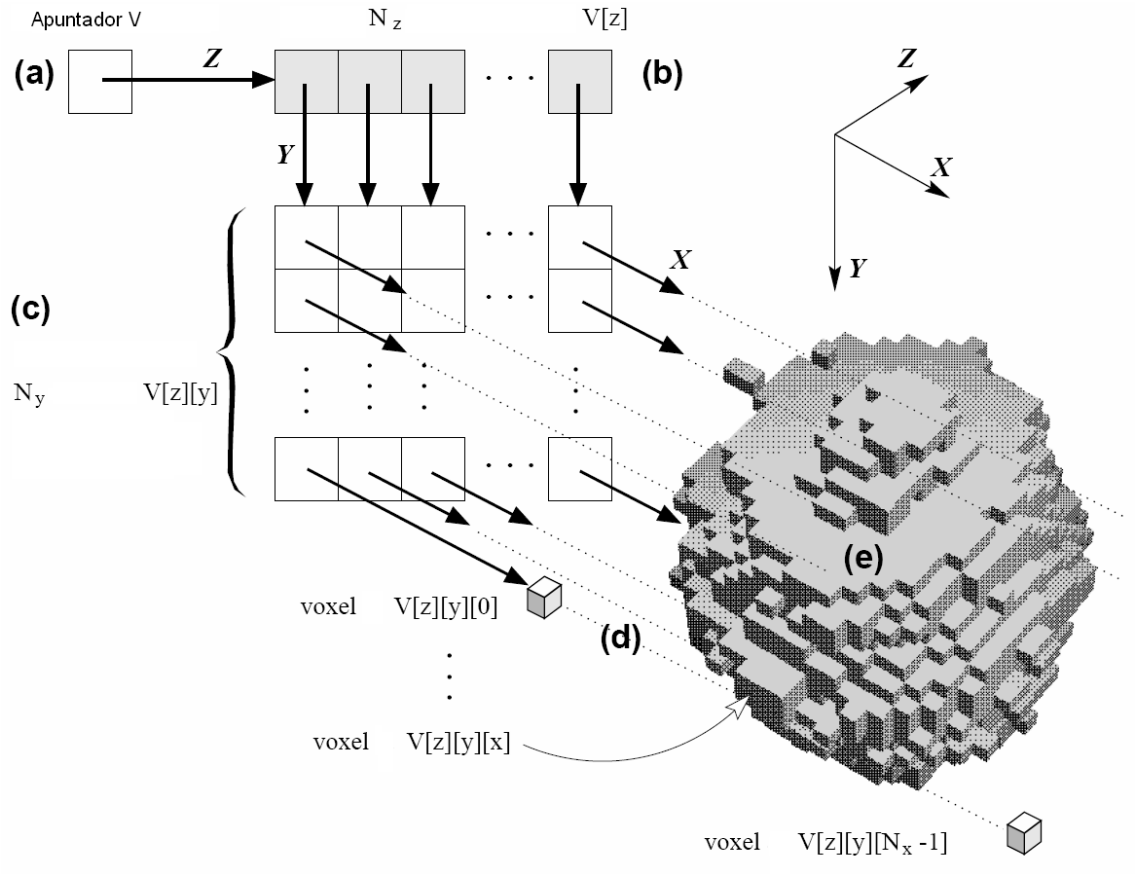

Figura 39: a) apuntador V del volumen (*V) ; b) apuntador contenido en V[z] para la componente en z; c) V[z][y] componente y del sistema coordenado; d) V[z][y][x] componente en x y devuelve el valor de la distancia en ese punto coordenado (x, y, z); e) gráfica de la voxelización**

El apuntador es utilizado como un arreglo de 3 dimensiones donde cada parámetro de offset que utiliza el apuntador es una coordenada. El direccionamiento es muy rápido, con esta forma de representar el campo 3D y con ello se hace muy dinámica la manipulación del modelo dentro del campo de distancia y su texturizado al grado de poder hacerlo en tiempo real. Al no ocupar un bloque continuo, sino renglones dispersos en el "heap" de memoria, es más fácil e inmediata su localización y la propia asignación de memoria, dado que es más probable que el sistema encuentre muchos renglones libres, que un sólo bloque continuo de las mismas dimensiones. Al texturizar el modelo con el campo de distancia (vea texturas a mapear) se necesitan los valores de las posiciones (x, y, z) de cada vértice y se asocia al valor del campo. Finalmente, el acceso (lectura y escritura) es totalmente transparente, como arreglo 3D: V[z][y][z] = 255, voxel = V[z][y][z].

Se da formato a los vértices y al campo ya que el campo de distancia al estar guardado como un triple apuntador en memoria solo permite valores enteros positivos para su direccionamiento y además se deben normalizar la distancias para que queden en el intervalo de 0 a 1

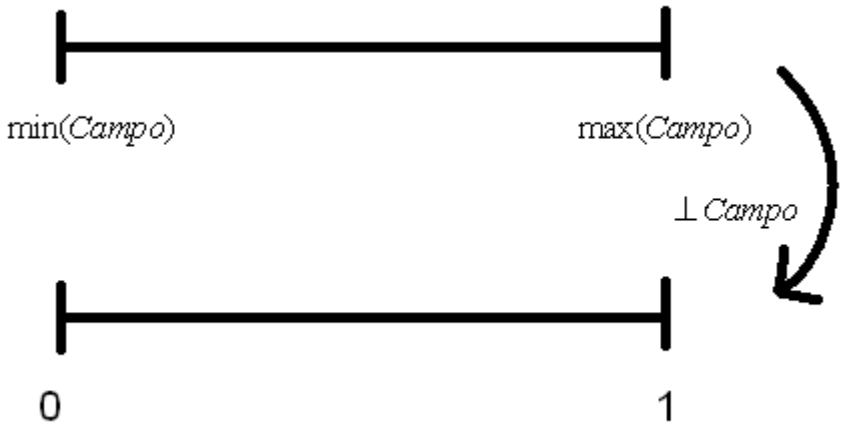

Figura 40: normalización de la máxima distancia y la mínima distancia del campo

Superficie del modelo w
Campo de distancia V(w)
Normalización $\perp\{V(w)\}$

$$V_{\perp}(w) = \perp\{V(w)\}$$

$$V_{\perp}(w) = \left\{ \frac{V(w) + \min\{V(w)\}}{\max\{V(w)\} - \min\{V(w)\}} \right\}$$

NOTA: Debido a que las distancias del campo tienen un offset, debe ser restado para que no se pierdan decimales al hacer operaciones de número muy grandes con números decimales.

```
void Modelos::Texturizar(float x, float y, float z, float
***campo, PropiedadesCampo pc)
{
      int xInt=(int)x + pc.slices/2;
      int yInt=(int)y + pc.rows/2;
      int zInt=(int)z + pc.cols/2;
```

```
	if(xInt>= pc.slices)xInt	= pc.slices-1;
	if(yInt>= pc.rows)	yInt = pc.rows-1;
	if(zInt>= pc.cols)  zInt	= pc.cols-1;

	if(pc.minimoEnCampo<0)pc.minimoEnCampo    =
pc.minimoEnCampo*(-1);



	if(xInt<pc.slices && yInt<pc.rows && zInt<pc.cols &&
	xInt>=0 && yInt>=0 && zInt>=0)
		glTexCoord2f(0.1,(campo[xInt][yInt][zInt]-
128+pc.minimoEnCampo)/(pc.maximoEncampo+pc.minimoEnCampo));

	else
	{
		glTexCoord2f(0.1,0);
	}
}
```

Ya normalizados los valores de distancia del campo, hay que mover el modelo al centroide del campo, porque el campo se despliega con únicamente valores positivos en sus coordenadas.

Matriz de vértices del modelo:	M
Dominio del Campo:	$Dom(V(w))$
Matriz de offser:	K_{offset}
slices:	s
rows:	r
cols:	c
Redondeo:	$round(M_{offset})$

$$Dom(V(w)) = \{x, y, z \mid 0 < x < s; 0 < y < r; 0 < z < c \mid x, y, z, s, r, c \in \mathrm{N}\} \quad (50)$$

$$M = \begin{bmatrix} x_1 \ldots x_n \\ y_1 \ldots y_n \\ z_1 \ldots z_n \\ 0 \ldots 0 \end{bmatrix} \quad (51)$$

$$K_{offset} = \begin{bmatrix} \frac{s}{2} \\ \frac{r}{2} \\ \frac{c}{2} \end{bmatrix} \quad (52)$$

$$M_{offset} = M + K_{offset} \quad (53)$$

$$round(M_{offset}) = (\mathrm{int}) \left(M_{offset} + \begin{bmatrix} .5 \\ .5 \\ .5 \end{bmatrix} \right) \quad (54)$$

3.3.- Visualización del campo de distancia

Ya con estos elementos se observan los resultados del mapeo de textura conforme a un campo de distancia y un modelo. En la figura 41 se muestra un corte transversal del campo de distancia. El corte es un mallado de vértices dibujados como GL_QUADS. El mallado en este proyecto sirve principalmente como referencia para ver la localización del Campo en el espacio y como es proyectada en un plano de corte:

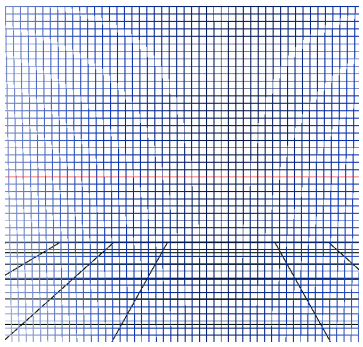

Figura 41: mallado de los puntos a texturizar

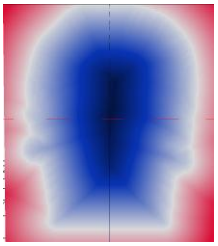

Figura 42: mallado con un texturizado solido (GL_POLYGON)

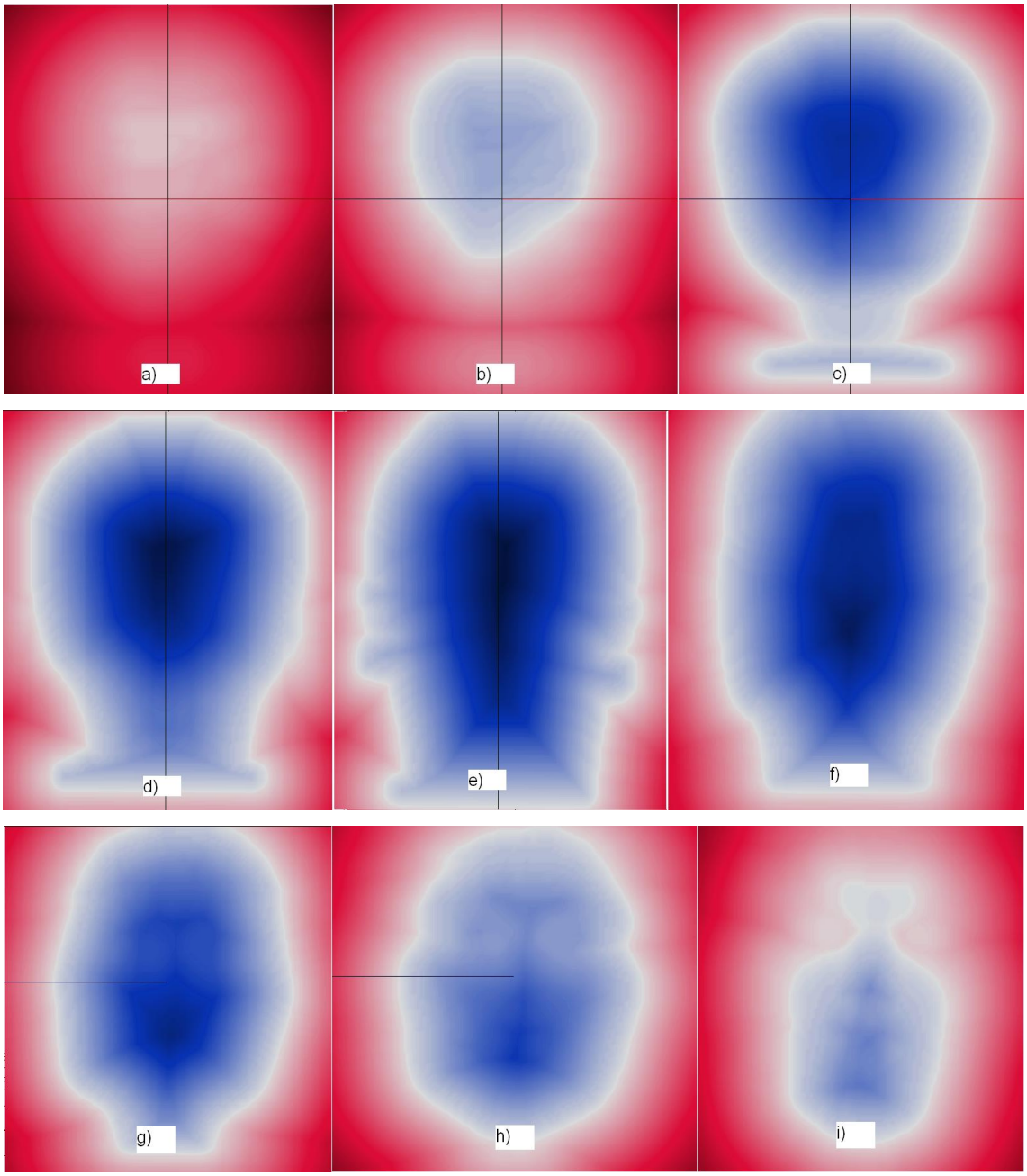

Figura 43: Múltiples cortes transversales del frente de una cabeza humana en su campo de distancia: a) tangente al parietal; b) parietal y el occipital; c) temporal y el parietal; d) frontal, mandíbula y parte del parietal; e) esfenoides, frontal y mandíbula; f) frontal y cigotomático; g) etmoides y frontal; h) frontal, etmoides y maxilar; i) nasal, maxilar y mandíbula

Al igual que con los modelos y la cámara se puede interactuar con los cortes coronales, sagitales y transversales (o axiales) del campo de distancia. En el corte transversal del campo se ven en color las partes internas y externas de la superficie del modelo que genera el campo, usando la escala de la Figura (22), página 40. La línea blanca es la intersección del plano proyectante con el modelo. Las tonalidades indican la distancia que hay de cualquier punto en el espacio con respecto al modelo generador, entre más oscuro sea el color más alejado del modelo se encuentra. Al mapear el modelo con el campo, la distancia disminuye entre ambas superficies al aumentar el parecido y al aumentar la alineación se interpreta que entre más blanco se texturice más parecido es al modelo de referencia (modelo generador del campo). Para poder comparar formas con el campo de distancia primero se necesita alinear lo mejor posible y así se podrá observar el error intrínseco o más bien las diferencias simétricas que hay entre el modelo y el modelo de referencia, o bien, las diferencias debidas a una alineación parcial. Una medida numérica de tales diferencias (o similitudes) es el integrar el volumen diferencia entre campos de distancia, pero para ello convendrá establecer un factor de normalización, por ejemplo en base al volumen total.

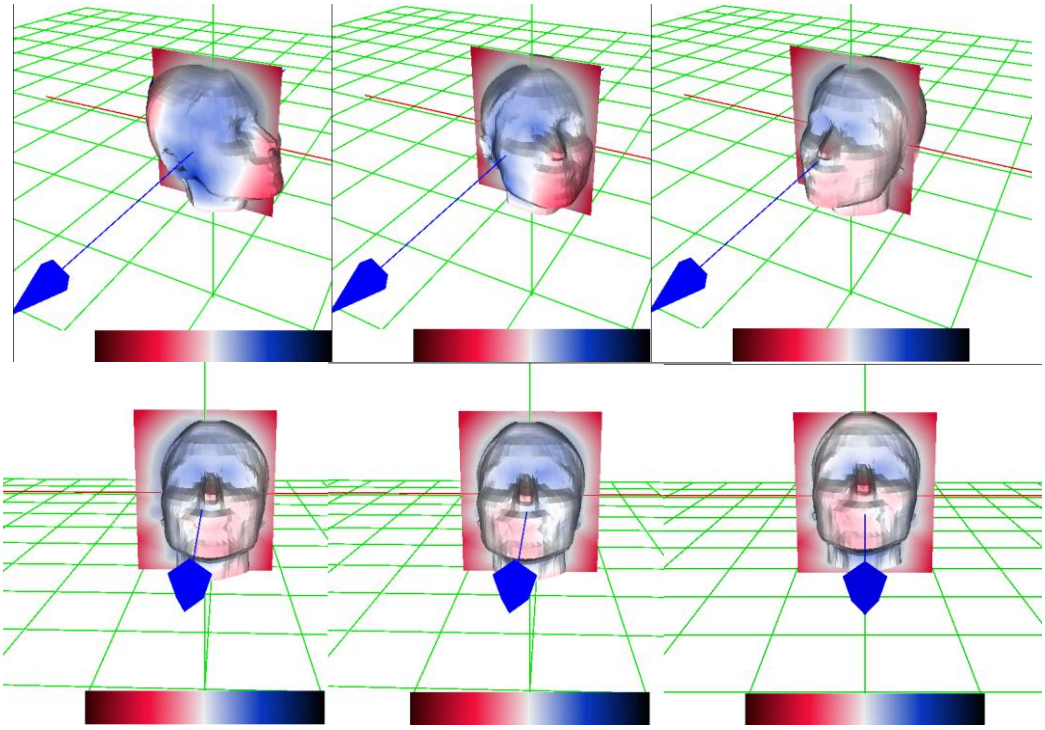

Figura 44: Múltiples vistas del modelo y un corte transversal del campo de distancia