Universidad Nacional Autónoma de México

Facultad de Ingeniería

Visualización científica para comparar formas y evaluar métodos de registro

Tesis

PARA OBTENER EL TÍTULO DE INGENIERO EN COMPUTACIÓN PRESENTA:

JOSÉ LUIS DOMÍNGUEZ ROJAS

DIRECTOR DE TESIS:
DR. JORGE ALBERTO MÁRQUEZ FLORES

2010

Agradecimientos:

Quiero agradecerle a toda mi familia por su confianza y la fe que han tenido en mí y principalmente a mi madre Araceli Rojas por darme su inmenso apoyo y motivación. A mi hermana Montse que siempre cuento con ella y a mí padre, del que estoy muy orgulloso y le agradezco todo lo que me ha enseñado. Doy gracias a mis amigos Adrián, César, Couoh, Martha y Roque que han estado a mi lado en las buenas y en las malas, que con el paso del tiempo seguimos estando juntos y los considero como mis hermanos. A Rosario que es inmensamente especial le doy gracias por acompañarme y estar en mi vida en cada momento. Le doy gracias al Dr. Jorge Márquez por ser tan paciente y orientándome en cada paso que daba para este proyecto.

RESUMEN:

En esta tesis se reporta el diseño e implementación de un programa que permite visualizar errores morfológicos en cabezas humanas para la antropometría craniofacial; mediante *campos de distancia euclidiana* se obtienen los datos necesarios para describir las aproximaciones espaciales de otros cuerpos con el modelo de la cabeza en estudio, calculando así el error morfológico que existe entre estos dos modelos. Los errores extrínsecos se reducen alineando los modelos, utilizando un método alineación manual interactivo el cual hace uso del ratón de la computadora y el teclado para manipular la posición y orientación del modelo. La alineación en forma se hace con el algoritmo de *Análisis de Componentes Principales* tomando a los modelos como aproximaciones a elipsoides y alineando sus ejes. Los datos y el texturizado se mapean en tiempo real dependiendo de la intersección de la superficie del modelo con el campo de distancia del modelo de referencia. El mapeo se obtiene de un campo de distancia pre-calculado y cargado en la memoria física principal de la computadora. La programación es un hibrido entre programación estructurada y la programación orientada a objetos para su futura expansión y actualización. Esta tesis describe y muestra en una forma gráfica las diferencias entre objetos en general utilizando un formato parecido a VRML del cual se extraen los vértices de cada modelo. La parte estructurada del programa es la parte de renderizado, carga del campo y cálculos del mismo mientras que la parte orientada a objetos está dirigida a la interacción y visualización del ambiente de trabajo. Al integrar la medida de error morfológico, a lo largo de la superficie del objeto, se obtiene una medida de su similitud (o falta de alineación) con el objeto de referencia.

Índice:

Introducción y antecedente…….....6

Objetivos……...8
Relevancia…….9

Estructura de la tesis

1.- Ambiente de desarrollo y bibliotecas gráfica ……,……...10
1.1.- Software de dominio público……..10
1.1.1.- C++……...10
1.1.2.-DevC++…….……12
1.1.3.-VRML…….……..13
1.1.4.-OpenGL…….……....14
1.2.- Software comercial…….…..……..20
1.2.1.- IDE Visual Studio: Aplicación OpenGL con MFC…….20

2.- Métodos para comparación y visualización……...29
2.1.- Base de datos: Construcción de modelos de Cabezas con VRML……..29
2.2.- Campo de distancia con la transformada Euclidiana de distancia (EDT)……...33
2.3.- Medidas de comparación morfológicas……...37
2.3.1.- Diferencia simétrica entre 2 objetos…….37
2.4.- Métodos de visualización científica de morfometría……..40
2.4.1.- Visualización de cabezas con una paleta de pseudo-color……...40
2.5.- Emulación de cómo visualizar un campo de distancia……...……44
2.6.- Alineación……...50
2.6.1.- Análisis de componentes principales para alineación……..50

3.- Implementación y programación del software para visualizar.....…….……55
3.1.- Programación y diseño de interfaz gráfica……..56
3.2.- Campo de distancia en memoria…….63
3.3.- Visualización del campo de distancia…….68

4.- Resultados y discusión…….71

4.1.- Comparación entre 2 objetos…….. ……...71

Conclusiones……...77

Anexos A…….78

-Transformaciones en 3D

Bibliografía……..83

Introducción y antecedentes

En estructuras anatómicas como la cabeza, existen variaciones en su forma global, y en sus rasgos salientes (tanto forma como distribución en su posición y orientación). Por otro lado, es fundamental contar con una alineación o registro geométrico, y una normalización (contar con una misma escala) que permita realizar comparaciones cuantitativas y operaciones tales como promedios de formas, de medias y desviaciones estándar en forma y posición así como comparaciones cualitativas, tales como visualización de la distribución de diferencias intrínsecas (debidas a la variación y no al posicionamiento o a la adquisición) y similitudes, tanto entre individuos como respecto al promedio poblacional. En dichas comparaciones se deben minimizar las diferencias extrínsecas, debidas a una alineación parcial o errónea. Este último problema es el más difícil y en ocasiones la comparación y el registro (no lineal, con deformaciones) constituyen etapas de un mismo proceso.

En el caso de la cara de individuos de la población normal, es posible establecer promedios morfológicos (de la forma de la superficie externa, no de intensidades) [MARQUEZ2005], dado que las variaciones de los rasgos cranofaciales, a pesar de ser importantes, se presta a establecer correspondencias entre individuos. Esto permite el registro o alineación mediante transformaciones deformables y por lo tanto la normalización de un grupo de individuos, a partir del cuál es posible construir atlas antropométricos. En el laboratorio de Análisis de Imágenes y Visualización se cuentan con programas para dichos promedios morfológicos, pero aún no se tienen herramientas adecuadas para visualizar las diferencias en los resultados y el propio registro geométrico ha resultado una tarea compleja que requiere de dichas herramientas de visualización de errores para la evaluación de métodos de alineación.

Un paso previo a la tarea anterior es el *registro rígido*, mediante Análisis de Componentes Principales (PCA), del cuál se extrae un elipsoide equivalente cuya orientación determina la transformación [RAMIREZ2005]. En este caso, se puede hacer énfasis en la forma esferoidal de la cabeza, penalizando los rasgos salientes, o bien, al contrario, un registro iterativo que minimiza las diferencias entre rasgos que se

corresponden. En ambos casos es posible realizar comparaciones visuales mediante cálculos de la intensidad total de la diferencia entre campos de distancia extraídos de cada cabeza, una de referencia y la que se desea comparar o evaluar su alineación respecto a la primera. Un trabajo en que se usó una versión muy preliminar del que se presenta en esta tesis, pero con una aproximación de mapa de distancia discreta en *chaflan* (chamfer, en inglés), se reporta en [MARQUEZ2008], para ajuste de elipsoides a cabezas humanas. Otro método de registro entre dos objetos, que permite trabajar con muestras de sus superficies, es el de *ICP* (*iterated closest points*), en que se va iterando parejas de puntos correspondientes a los más cercanos en la referencia [BESL1992], y del cuál se cuentan con implementaciones que abordan el problema de la correspondencia óptima [BENJEMAA1998]. Finalmente, es posible tomar en cuenta deformaciones, mediante un registro "deformable", o no-rígido, en el cuál se incorporan puntos fiduciarios, e información de la anatomía conocida ("a priori"), para un alineación que incluya variaciones naturales en la forma de estructuras anatómicas. Un ejemplo de este proceso se describe en [CAMARA2007].

En el laboratorio de Análisis de Imágenes y Visualización (LAIV) del CCADET, UNAM, se cuenta con trabajos sobre la forma de la cabeza y rostro humano, y métodos para establecer promedios morfológicos de estructuras anatómicas cuya variación no es demasiado elevada. Entre dichos métodos se cuenta con experiencia en el uso de campos de distancia euclidiana para fines morfométricos [MARQUEZ2006]. En el caso de alineación con deformación, también se están realizando trabajos que requerirán medios de comparación de error morfológico. Un ejemplo del trabajo en curso en LAIV, CCADET, es reportado en [SULLI2010].

En el presente trabajo, se plantea el estudio y desarrollo de métodos de análisis comparativo, tanto de rasgos salientes (extraídos a mano) como de la cabeza en general, mediante visualización del campo de distancia tanto individual como diferencial, proyectado en la superficie de la cabeza de referencia o de la cabeza bajo estudio, y explorar alcances y limitaciones de al visualización paramétrica, en el caso de un banco digital de superficies cranofaciales de 40 individuos.

Objetivos:

-Visualizar en 3D, mediante escalas de color, la semejanza entre objetos (por ejemplo en violetas, azules y verdes para valores negativos o internos respecto a una superficie y en rosas, rojos y amarillos para valores positivos o externos respecto a la misma superficie).

- Explorar diversos esquemas de visualización, mediante color, patrones texturales, visualización de normales u otras líneas conceptuales, animación y superposición transparente de información diversa.

- Usar y adaptar el método de registro que penaliza rasgos, iniciando con Análisis de Componentes Principales, y los métodos de registro mediante minimización de la distancia cuadrática global entre rasgos correspondientes (capturados a mano).

- Usar las herramientas visuales para evaluar tanto métodos de registro como diferencias intrínsecas entre individuos.

Otras tareas a realizar.

- Mapeo en la superficie craniofacial de los valores de gris de un volumen con información escalar (campo de distancia diferencial, o sea del error de alineación mas las diferencias intrínsecas entre individuos).

- Uso de color para visualizar las variaciones del mapeo anterior.

- Uso de software para la extracción de rasgos salientes y puntos de referencia.

Se cuenta con:

- Software de apoyo para análisis tridimensional: AMIRA, Blender, VizUp (mallado) y programas desarrollados o en desarrollo en el laboratorio LAIV para alineación, captura de rasgos, triangulación, voxelización y cálculo de campos de distancia.

Relevancia:

En muchas aplicaciones biomédicas las formas biológicas presentan alta variación y su estudio requiere de contar con referencias representativas de la población de estudio. En el caso de la cabeza humana, en nuestro país no hay modelos específicos a la población mexicana. El contar con una visualización comparativa facilitaría, por un lado, la evaluación de métodos de registro, la construcción de modelos representativos y el establecimiento de diferencias respecto a dicho modelo; por otro lado facilitaría la comunicación e intercambio de ideas entre especialistas en antropometría craneofacial, médicos interesados en cirugía reconstructiva, por ejemplo, médicos forenses y científicos en visualización científica. Finalmente, este trabajo constituye un aspecto importante de proyectos de alta relevancia, en el Laboratorio de Análisis de Imágenes y Visualización, por ejemplo, un proyecto en curso, en colaboración con el Instituto de Neurobiología, es la construcción de promedios entre cerebros humanos, para extraer modelos de referencia (atlas) y la comparación entre cerebros y la referencia, con lo cuál sería posible por ejemplo establecer si una nueva instancia (un cerebro que no fue promediado) pertenece o no a la población, dado un criterio de distancia media, en determinadas regiones. Resultados preliminares de la alineación con deformación son reportados en [SULLI2010], y sobre atlas del cerebro, en [ORTIZ2009].

Capítulo I.- Ambiente de desarrollo y bibliotecas gráficas

1.1.- Software de dominio público

1.1.1.- C++

El lenguaje en C++ es la más adecuada para la programación gráfica debido a su flexibilidad en el uso y gestión de la memoria y de procesos a bajo nivel de la computadora además de contar con compiladores muy eficientes, lo que agiliza el desarrollo de proyectos complejos. En este lenguaje, a pesar de su flexibilidad en cuanto al uso de los recursos de la computadora, la programación se vuelve compleja y más cuando se trata de gráficos porque no solo es el uso de la memoria principal (RAM) sino también de la memoria de la tarjeta de video que hace todas las transformaciones en vértices, texturas y al renderizar una escena programada. En esta tesis las transformaciones geométricas del modelo se realizan en la memoria principal esto fue porque el texturizado depende del campo de distancia almacenado en memoria principal, entonces en vez de ir acumulando transformaciones para que al final las ejecute el GPU de la tarjeta de video, nosotros programamos esas transformaciones y las ejecutamos por cada evento dentro del programa. Un ejemplo de la programación de una transformación geométrica es la siguiente:

```
void Modelos::Trasladar(float x, float y, float z)
{


      for(int i=0;i<=nVertices;i++)
      {
            VertexArray[i].x+=x;
            VertexArray[i].y+=y;
            VertexArray[i].z+=z;
      }

}
```

La transformación geométrica fue programada de esta forma porque es más rápido y más eficiente para el procesador hacer una suma y una asignación que hacer las respectivas sumas y multiplicaciones que conlleva la multiplicación de matrices. El programa debe hacer transformaciones permanentes al modelo para que se pueda texturizar, si se hacen las transformaciones usando OpenGL en si el modelo se guarda tal y como es cargado y el texturizado no es dinámico porque OpenGL guarda el modelo en memoria pero no lo modifica, solamente hace una multiplicación de matrices con la matriz de transformación acumulada (ver anexo A) al final, antes de dibujar en el render. En C++ se dispone de muchas facilidades para programar estructuras de datos (arreglos no homogéneos) lo cual facilita la programación y permite al programador una programación más limpia y ordenada.

```
typedef struct
	{
		float x;
		float y;
		float z;
	}Vertice;

typedef struct
	{
		int slices;
		int rows;
		int cols;
		float minimoEnCampo;
		float maximoEncampo;
	}PropiedadesCampo;

typedef enum{ERROR0,ERROR1,ERROR2,ERROR3,ERROR4,ERROR5};
typedef enum{MODELO_ROTAR,MODELO_TRASLADAR,MODELO_ESCALAR};
typedef
enum{TRASLADAR_X,TRASLADAR_Y,TRASLADAR_Z,ROTAR_X,ROTAR_Y,ROTAR_Z
,SIN_TRASLADAR,SIN_ROTAR};
```

Es muy importante notar que al tener modelos de las cabezas en mallados, es decir en una estructura de vértices, sólo es necesario transformar sus coordenadas. Los campos

de distancia (información volumétrica) se calcularán sobre voxelizaciones de dichos mallados y re-voxelizaciones, cada vez que sufran alguna transformación geométrica. Si se intentara transformar un volumen de voxeles, sería necesario realizar al mismo tiempo la interpolación de los valores de cada voxel. Al trabajar las transformaciones sólo sobre los mallados, no es necesario lo anterior.

Las desventajas de la programación en C++ es la dificultad técnica para programar, al ser un lenguaje de menor abstracción que otros lenguajes, la programación se vuelve muy compleja y por ende dificulta el mantenimiento del código. . Por otro lado, la programación orientada a objetos permite la reutilización de código desarrollado en distintos proyectos, lo cuál ocurre frecuentemente en el Laboratorio de Imágenes y Visualización, donde se desarrolló el presente proyecto.

1.1.2.- DevC++

Bloodshed **Dev-C++** [DEVC2007] es un entorno de desarrollo integrado (IDE por sus siglas en inglés) para programar en lenguaje C/C++. Usa el compilador MinGW que es una versión de GCC (GNU Compiler Collection). Dev-C++ puede además ser usado en combinación con Cygwin y cualquier compilador basado en GCC.

DevC++ fue muy utilizado en el laboratorio de CCADET entre 2003 y 2009, debido a su facilidad de manejo y de ser software libre. Los programas para obtener el campo de distancia y alineación se hicieron mediante este entorno de desarrollo y parte del código fue adaptado al software de la aplicación de esta tesis. Un ejemplo de este código es una función para reservar memoria:

```
float ***alloc_float_volMem(void)
{     short i,j,k;
      float ***v;  /* if we set static, it will use the same */

      /* alloc a ptr to a slices-list of ptrs */
      /* TO_ADD:  if global and not NULL, free before realloc */
```

```
      if((v=(float ***) calloc( slices, sizeof(float **))
)==NULL)
      {     printf("Error of memory 1 ");
            exit(1);
      }

      /* for each slice ptr, alloc a row-list of float ptrs INIT
to 0 */
      for(k=0; k<slices; k++)
      {     /* could have been an alloc_ima( cols, rows, sizeof(
*float) ) */
            if((v[k]=(float **) calloc( rows, sizeof(float *))
)==NULL)
            {     printf("Error of memory 2 ");
                  exit(1);
            }

            /* for each row(=line) ptr-ptr, alloc a col-list of
floats */
            for(j=0; j<rows; j++)
            {     /* for each line ptr, alloc a col-list of
floats */
                  if((v[k][j]=(float *) calloc( cols,
sizeof(float)) )==NULL)
                  {     printf("Error of memory 3: volume TOO BIG
(%li bytes?)\n", cols*rows*slices);  /* Nc*Nl*Ns */
                        exit(1);
                  }
            };
      };
      /*** access: v[z][y][x]   k j i, slice k row j col i,  Ns
Nl Nc ***/
      return v;   /* caller sets dimens, if required */

}     /*** end  alloc_floa*/
```

1.1.3.- VRML

VRML (sigla del inglés Virtual Reality Modeling Language. "Lenguaje para Modelado de Realidad Virtual" [VRML2005])- formato de archivo normalizado que tiene como

objetivo la representación de escenas u objetos interactivos tridimensionales; diseñado particularmente para su empleo en la web.

El lenguaje VRML (*Virtual Reality Modeling Language*) posibilita la descripción de una escena compuesta por objetos 3D a partir de prototipos basados en formas geométricas básicas o de estructuras en las que se especifican los vertices y las aristas de cada polígono tridimensional y el color de su superficie. VRML permite también definir objetos 3D multimedia, a los cuales se puede asociar un enlace de manera que el usuario pueda acceder a una página web, imágenes, vídeos u otro fichero VRML de Internet cada vez que haga click en el componente gráfico en cuestión.

1.1.4.- Programación en OpenGL

OpenGL son librerías libres (o sea, de dominio público) a las que cualquier usuario puede tener acceso sin tener que comprar una licencia y las cuales sirven para la programación de gráficos en 3D. OpenGL es puramente estructurado y únicamente es para enviar mandos para dibujar gráficos, no soporta controladores de audio o interfaces de interacción con el usuario como lo es DirectX. Una referencia que nos resultó muy útil, en especial para los fundamentos matemáticos de graficos con OpenGL es [BUSS03].

La librería de OpenGL trabaja como una estructura de datos de pila donde las instrucciones de transformación y de dibujo se ejecutan en orden inverso es decir de la última instrucción del bloque glPushMatrix(); y glPopMatrix(); hasta la primera.

Ejemplo:

```
glPushMatrix();
      glScaled(.1,.1,.1);
      glTranslatef(xx,yy,zz);
      glTranslatef(-cx,-cy,-cz);
      glutSolitCube(D);
glPopMatrix();
```

glPushMatrix(); y glPopMatrix(); sirven como separadores para que otras transformaciones dentro del código no afecten a otras figuras geométricas. En el código anterior se ejecutaría de la siguiente manera:

1.- Se manda a dibujar el Cubo.

2.- Se traslada de su posición original a –cx,-cy,-cz

3.- Se traslada de igual forma xx, yy, zz.

4.- Se cambia el tamaño en un factor de .1 en ‘x’ en .1 en ‘y’ y en .1 en ‘z’

La unidad atómica para dibujar en OpenGL son los vértices que se mandan a dibujar con la instrucción `glVertex3f(x,y,z)` esta función es más que solo dibujar un punto en el espacio de 3D porque con ella se arman los triángulos o cuadrados para después formar un poligono más complejo o superficies de formas irregulares. Para dibujar un sencillo triángulo en OpenGL se debe poner entre 2 instrucciones que son glBegin(MODO); y glEnd(); donde MODO es de cómo se van a unir los vértices, esta unión puede ser desde que se unan los vertices en triángulo, cuadrado o conforme se vayan dibujando. La union entre vértices también le dan el carácter de un polígono solido, de malla o de puntos. Para mantener una congruencia se manda a dibujar triángulo por triángulo con sus respectivas uniones entre otros trángulos como se puede ver en la figura 1:

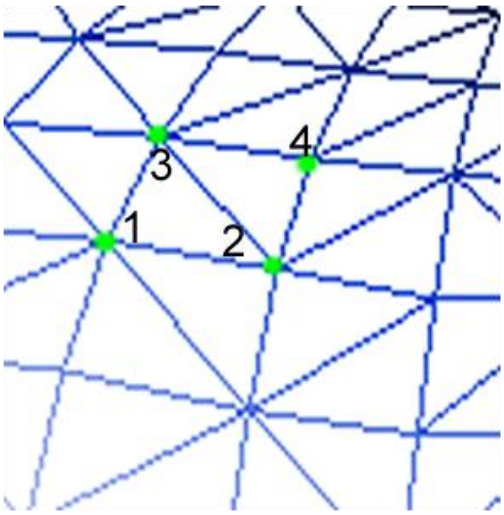


Figura 1: mallado por triángulos 1

En la figura anterior se puede ver que primero se manda a dibujar el vértice 1 y se une con el vértice 2 y 3 y el cuarto vértice se une con 3 y 2 y así se va armando toda la

figura. Entonces para figuras complejas se tienen 2 arreglos uno que es de los vértices y otro arreglo que es el de los índices de qué vértice se va a unir con cual otro.

Para poder mandar a dibujar se necesita configurar el puerto vista (Viewport) que es la ventana a través de la cual puede observar la escena. Para el viewport solo se necesita poner una posición inicial dentro del "render" y luego indicar los desplazamientos en las coordenadas x,y

```
glViewport(Vx,Vy,(GLsizei)(width),(GLsizei)(height));
```

Algo muy importante es el uso de cámaras en OpenGL. Una cámara en OpenGL simula una cámara real. Las cámaras también sufren cambios cuando se aplica una transformación 3D por ello se debe tener cuidado al poner alguna. Para expresar una cámara en OpenGL se necesitan de varios parámetros dentro de la función:

```
gluLookAt (Cx,Cy,Cz, Px, Py, Pz, Ux,Uy ,Uz);
```

Los tres primeros paramétros son las coordenadas de la posición de la cámara, los siguientes 3 son las coordenadas del punto a observar (punto de fuga) y por último se especifica un vector unitario "Up" que es el vector que orienta a la cámara con respecto a su propio sistema coordenado.

$\bar{c} = c_x \hat{i} + c_y j + c_z k$ *Vector de posición de la cámara*

$\bar{p} = p_x \hat{i} + p_y j + p_z k$ *Vector del punto del punto p*

$\overline{Up} = u_x \bar{i} + u_y \bar{j} + u_z \bar{k} \leftrightarrow \left|\overline{Up}\right| = 1$ *Vector Up*

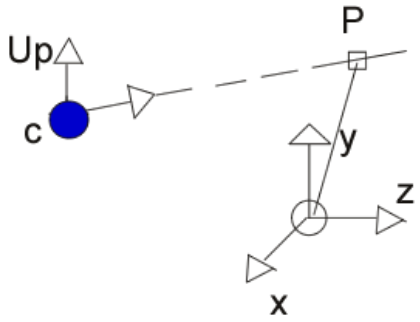


Figura 2: Representación geométrica de la cámara

La programación de luces en OpenGL requiere de 4 parámetros que 3 son para los tipos de luz (Ambiental, difusa, especular) y una posición de la fuente de luz. Los tipos de luces se construyen dentro del espacio de color RGB y que va de 0 a 1 en punto flotante esto para que la luz ambiental, difusa y especular se programe su comportamiento conforme al color.

```
GLfloat     light_specular1[]={1.0f,1.0f,1.0f,1.0f};
GLfloat     light_diffuse1[]={1.0f,1.0f,1.0f,1.0f};
GLfloat     light_ambient1[]={1.0f,1.0f,1.0f,1.0f};
GLfloat     light_position1[]={0.0f,0.0,-150.0f,1.0};

glLightfv(GL_LIGHT0,GL_POSITION,light_position1);
glLightfv(GL_LIGHT0,GL_AMBIENT ,light_ambient1);
glLightfv(GL_LIGHT0,GL_DIFFUSE ,light_diffuse1);
glLightfv(GL_LIGHT0,GL_SPECULAR,light_specular1);
glEnable (GL_LIGHT0);

glEnable(GL_LIGHTING);
```

Las luces se van agrupando deacuerdo a una numeración {GL_LIGHT0, GL_LIGHT1,…} se agregan los parámetros, se activa la luz y después se “encienden” las luces. Las luces son importantes para poder visualizar sombras, contornos y principalmente texturas.

El texturizado es un mapeo de los vertices a una textura. En OpenGL las texturas se mapean con la instrucción `glTexCoord2f(x,y);` x,y van de 0 a 1 en punto flotante. Se utilizan imágenes para texturizar y estas imágenes deben de tener un tamaño de multiplos de 2 si no, OpenGL no interpreta bien la imagen y produce efectos de deformación en la imagen. Lo que hace OpenGL para texturizar es tomar la imagen guardada en memoria y luego crea una tabla en punto flotante donde mapea la imagen a esa tabla en una razón de 1 entre la longitud de la imagen en 'x' y de 1 entre la longitud de la imagen en 'y' para que después se haga el mapeo con los vértices asignándolo a cada vértice un pixel de la imagen e interpolando la secuencia de pixeles con respecto al siguiente vértice con su asignación de textura. La textura de la figura 3 es una escala de color pensada para poder representar visualmente valores de distancia positivos (rojos) y negativos (azules) respecto al valor de referencia "0" en blanco. Servirá para cotejar objetos muy semejantes al superponerse en el espacio, o bien para desplegar un sólo objeto en cuya superficie se mapea la distancia al punto más cercano del segundo objeto (que no será visible, pero se encuentra en memoria).

Figura 3: Textura a mapear 256 x 256 pixeles

Ejemplo del texturizado de un poligono plano rectangular y que es mapeado simétricamente y rotado con respecto a la textura (véase figura: 4)

```
glBegin(GL_POLYGON);
      glTexCoord2f(.1,0);glVertex3f(-(width)/30,0,0);
      glTexCoord2f(.1,0);glVertex3f(-(width)/30,500,0);
      glTexCoord2f(.1,1);glVertex3f((width)/30,500,0);
      glTexCoord2f(.1,1);glVertex3f((width)/30,0,0);
glEnd();
```

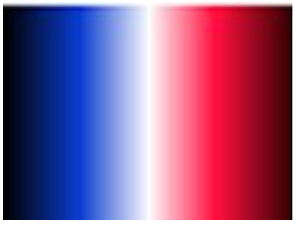


Figura 4: texturizado con simetría horizontal

El siguiente ejemplo es de un texturizado asimétrico con respecto a la textura y rotado (véase figura 5)

```
glBegin(GL_POLYGON);
      glTexCoord2f(.1,0);glVertex3f(-(width)/30,0,0);
      glTexCoord2f(.1,0);glVertex3f(-(width)/30,500,0);
      glTexCoord2f(.1,.7);glVertex3f((width)/30,500,0);
      glTexCoord2f(.1,.7);glVertex3f((width)/30,0,0);
glEnd();
```

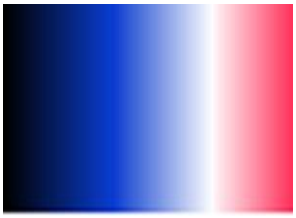


Figura 5: texturizado asimétrico horizontal

Ejemplo de un texturizando rotando en 180º la referencia de las texturas (véase figura 6):

```
glBegin(GL_POLYGON);
      glTexCoord2f(.1,1);glVertex3f(-(width)/30,0,0);
      glTexCoord2f(.1,1);glVertex3f(-(width)/30,500,0);
      glTexCoord2f(.1,0);glVertex3f((width)/30,500,0);
      glTexCoord2f(.1,0);glVertex3f((width)/30,0,0);
glEnd();
```

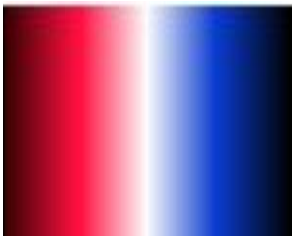


Figura 6: texturizado con una rotación de 180º

Existen varias formas de crear una ventana que contenga el render de nuestra aplicación una de ellas es usando las librerías de GLUT (OpenGL Utility ToolKit) que es una herramienta para crear la interface del render y la interacción de la escena con el teclado o mouse. GLUT es una buena herramienta para la portabilidad en distintos sistemas operativos. En esta tesis la aplicación de OpenGL se desarrolla en Windows y por ello se utilizará un ambiente de desarrollo como lo es Visual Studio 2005 con C++ y con ello crear la aplicación; se puede hacer comunmente de 2 formas; una es como una aplicación Win32 estandar y otra es como una aplicación MFC (Microsoft Foundation Class) . Aunque se comenzó el desarrollo de software en la plataforma con DevC++, la dificultad para programar las interfaces, no solamente en éste sino en otros proyectos de laboratorios del CCADET, nos llevó a buscar otras alternativas. La liberación de las licencias de Microsoft para académicos y estudientes ha permitido, desde fines de 2008 el uso de la plataforma Visual Studio. En el laboratorio de imágenes se están probando las plataformas de wxDevC++ y Visual Studio, la cuál seleccionamos para continuar el desarrollo, dado que portar los programas de DevC++ y librerías OpenGL resultó sencillo y rápido.

1.2.- Software comercial

1.2.1.- IDE Visual Studio: Aplicación OpenGL con MFC

MFC son unas librerias orientadas a objetos para la creación de ventanas en un entornos de Windows. Para nuestra aplicación de OpenGL con MFC se necesita crear una ventana de dialogo. Utilizando Visual Studio 2005 se crea un nuevo proyecto

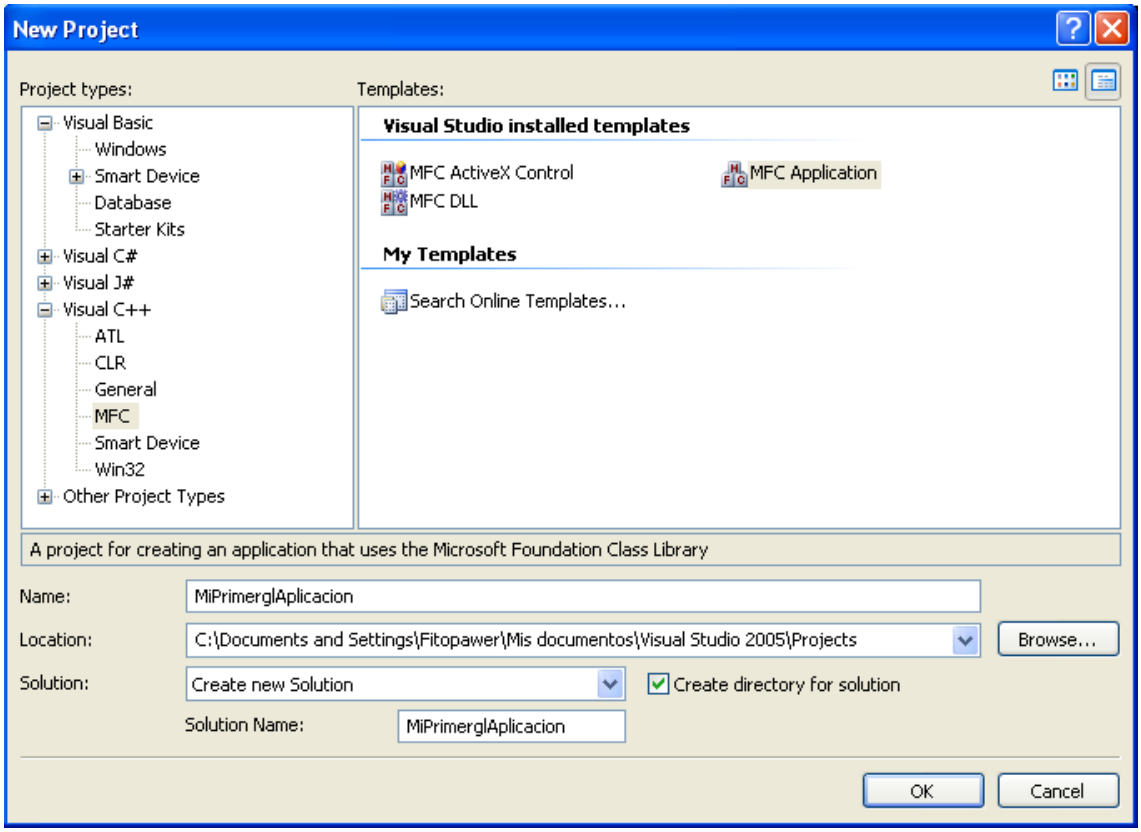


Figura 7: Ventana para crear un nuevo proyecto usando el framework de MFC

Se selecciona MFC y después se configura el tipo de aplicación. Para realizar nuestras las pruebas se escogió una ventana de Diálogo base y después se dejó que el MFC Wizard creara todas las clases y un único objeto que será la ventana principal. Obteniendo la ventana de diálogo de la Figura 8:

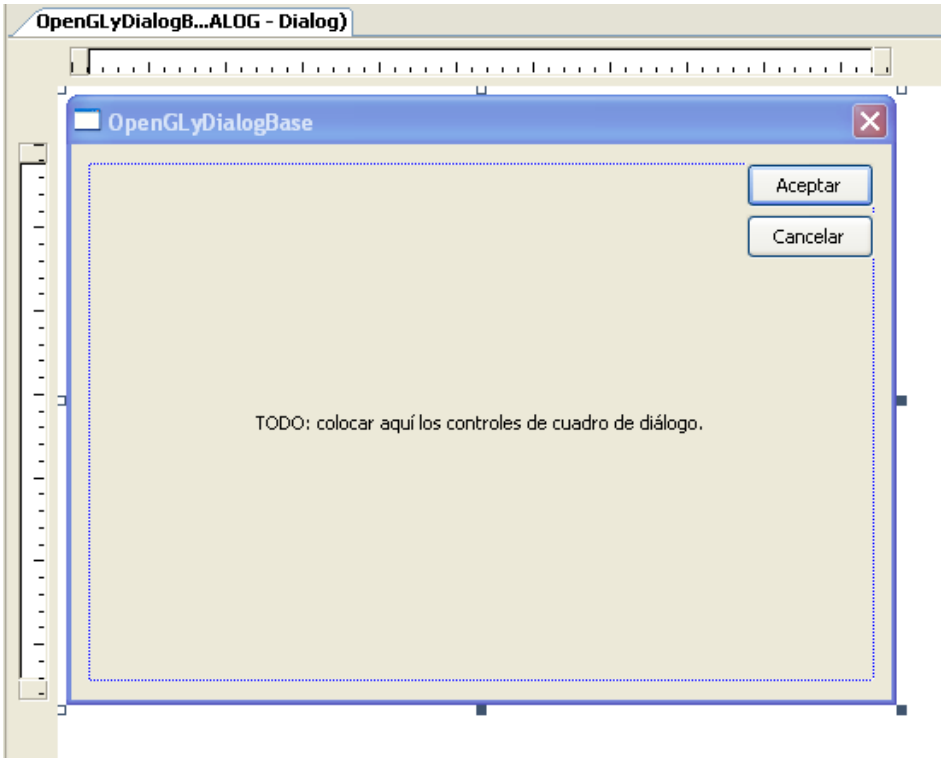


Figura 8: Ventana del tipo cuadro de diálogo base

Se agrega un tipo de control llamado Group Box en donde se va a renderizar la escena. Para lograr el renderizado se necesitan 3 funciones para la escena las cuales son las siguientes:

```
bool CrearGLWindow(HWND hWnd)
{
	GLuint	PixelFormat;
	hInstance = GetModuleHandle(NULL);
	hDrawWnd = hWnd;

	if(hDC=GetDC(hWnd),!hDC)
	{
		return FALSE;

	}

	pfd.nSize	=sizeof(PIXELFORMATDESCRIPTOR);
	pfd.nVersion	=1;
	pfd.dwFlags
	=PFD_DOUBLEBUFFER|PFD_DRAW_TO_WINDOW|PFD_SUPPORT_OPENGL;
	pfd.iPixelType	=PFD_TYPE_RGBA;
```

```
	pfd.cColorBits		=32;
	pfd.cRedBits		=0;
	pfd.cRedShift		=0;
	pfd.cGreenBits		=0;
	pfd.cGreenShift		=0;
	pfd.cBlueBits		=0;
	pfd.cBlueShift		=0;
	pfd.cAlphaBits		=0;
	pfd.cAlphaShift		=0;
	pfd.cAccumBits		=0;

	pfd.cAccumRedBits	=0;
	pfd.cAccumGreenBits	=0;
	pfd.cAccumBlueBits	=0;
	pfd.cAccumAlphaBits	=0;
	pfd.cDepthBits		=32;
	pfd.cStencilBits	=0;
	pfd.cAuxBuffers		=0;
	pfd.iLayerType		=PFD_MAIN_PLANE;
	pfd.bReserved		=0;
	pfd.dwLayerMask		=0;
	pfd.dwVisibleMask	=0;
	pfd.dwDamageMask	=0;

	if(PixelFormat=ChoosePixelFormat(hDC,&pfd),!PixelFormat)
	{
		MessageBox(NULL,"Unable to find the required pixel
format","ERROR",MB_OK|MB_ICONERROR);
		return FALSE;

	}

	if(!SetPixelFormat(hDC,PixelFormat,&pfd))
	{
		//Failed
		MessageBox(hWnd,"Unable to set the required pixel
format","ERROR",MB_OK|MB_ICONERROR);
		return FALSE;

	}
```

```
	if(hRC=wglCreateContext(hDC),!hDC)
	{
		//Failed
		MessageBox(NULL,"Unable to create Rendering
Context","ERROR",MB_OK|MB_ICONERROR);
		return FALSE;
	}

	if(!wglMakeCurrent(hDC,hRC))
	{
		//Failed
		MessageBox(NULL,"Unable to bind Rendering with Device
Context","ERROR",MB_OK|MB_ICONERROR);
		return FALSE;
	}
	RECT  WindowRect;

	GetClientRect(hWnd,&WindowRect);

	//Calculate drawing area
	width  = WindowRect.right - WindowRect.left;
	height = WindowRect.bottom - WindowRect.top;

	if (!InitGL())    // Initialize the OpenGL workspace
	{
		//Failed
		MessageBox(NULL,"Initialization
		Failed.","ERROR",MB_OK|MB_ICONERROR);
		return FALSE;

	}



	return TRUE;
}
```

En la función de `CrearGLWindow(HWND hWnd)` es donde se va a inicializar y configurar todas las caracterizticas de la ventana donde se va a renderizar nuestra escena, esta se compone primero de obtener el manejador del control “Group Box” que es el parámetro

que entra a la función y donde dibujaremos, lo segundo es llenar una estructura del tipo PIXELFORMATDESCRIPTOR que es la configuración de el rasterizado para los pixeles, después se inicializa el "Group Box" con esa configuración utilizando su manejador para que por último se inicie los valores que tendrá por default la escena y esta función es `InitGL();`

```
int InitGL(void)
{
	/*Initialize the OpenGL workspace*/

	glLoadIdentity ();

	gluPerspective(45.0f,(GLfloat)width/(GLfloat)height,0.1f,10
0.0f;
	glMatrixMode (GL_MODELVIEW);

	glLoadIdentity ();
	//Enable features
	glEnable(GL_DEPTH_TEST);
	glEnable(GL_POLYGON_MODE);
	glEnable(GL_POLYGON_SMOOTH);
	glEnable(GL_POLYGON_STIPPLE);
	glDepthFunc(GL_LEQUAL);
	glClearColor(0.0f,0.0f,0.0f,0.0f);
	return TRUE;}
```

La tercera función indispensable para dibujar es una llamada `DibujarEscena(void)` la cuál es donde agregamos los vértices, polígonos, figuras, luces, trasnformaciones y todo aquello que queremos mandar a dibujar. Esta función es muy importante ya que por cada cambio que hagamos a un control de nuestra ventana de diálogo esta se debe llamar para ejecutar de nuevo las instrucciones de dibujo y se vea el cambio en el despliegue.

En la función:

```
BOOL CVisualizadorVersion20betaDlg::OnInitDialog()
```

Se llama a la función CrearGlWindow(..) y lo más importante para que funcione es que el parámetro que entra a la función es el manejador del "Group Box" y se obtiene con la función `GetDlgItem(RENDER)->m_hWnd` en la cual GetDlgItem toma el objeto "Group Box" utilizando si ID=RENDER (en este caso el ID del "Group Box" lo llamamos RENDER) y como GetDlgItem devuelve un apuntador al objeto entonces necesitamos que apunte a m_hWnd que es el manejador del objeto "Group Box"

```
CrearGLWindow(GetDlgItem(RENDER)->m_hWnd);
```

Ya para mandar a dibujar por primera vez nuestra escena llamamos a la función DibujarEscena(); en la función:

```
void CVisualizadorVersion20betaDlg::OnPaint()
{
	if (IsIconic())
	{
		CPaintDC dc(this); // Contexto de dispositivo para
dibujo

		SendMessage(WM_ICONERASEBKGND,
reinterpret_cast<WPARAM>(dc.GetSafeHdc()), 0);

		// Centrar icono en el rectángulo de cliente
		int cxIcon = GetSystemMetrics(SM_CXICON);
		int cyIcon = GetSystemMetrics(SM_CYICON);
		CRect rect;
		GetClientRect(&rect);
		int x = (rect.Width() - cxIcon + 1) / 2;
		int y = (rect.Height() - cyIcon + 1) / 2;

		// Dibujar el icono
		dc.DrawIcon(x, y, m_hIcon);
	}
	else
	{
		CDialog::OnPaint();
	}
	DibujarEscena();
}
```

Con esto se dibuja por primera vez nuestra escena. Como queremos agregar controles a nuestra ventana y los valores que se obtengan sean enviados a nuestra escena, primero debemos agregar un control ya sea un botón, checkbox, scroll, etc… se arrastra el control desde el toolbox de Visual Studio

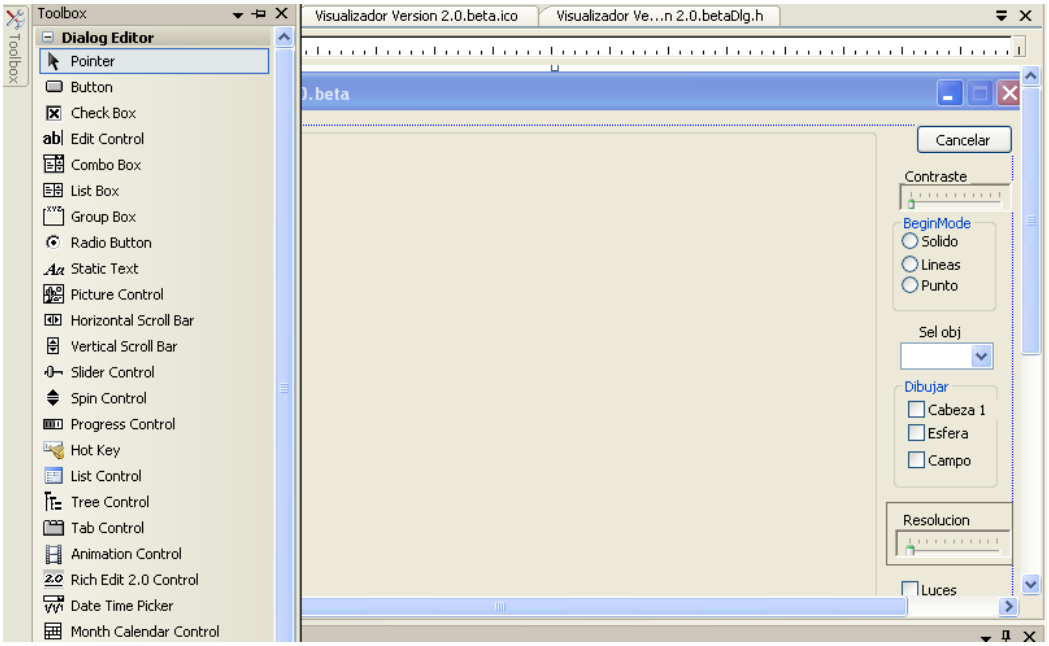


Figura 9: Ventana con una caja de grupo donde se renderizará la escena y con sus controles para transformar el modelo

Ya colocado el control con el botón derecho del mouse se de un click encima y se elije la opción “add variable” y se le da un nombre a la nueva variable, el tipo de variable lo da automáticamente el “Add Member Variable Wizard” así con esta variable podemos acceder a los datos del control.

Estos controles son muy útiles para poder controlar nuestra aplicación en OpenGL, cabe señalar que por cada cambio en el control se debe mandar a dibujar de nuevo con la función DibujarEscena(); por ejemplo si queremos hacer rotaciones o traslaciones con la posición del mouse y un click de alguno de sus botones el código quedaría de la siguiente forma:

```
void CVisualizadorVersion20betaDlg::OnMouseMove(UINT nFlags,
CPoint point)
{
```

```
        // Aqui se recibe los mensajes de la posición del MOUSE

        switch(nFlags){
              case MK_LBUTTON:
                    GetRotacion(point.x,point.y);
                    DibujarEscena();
                    break;
              case MK_RBUTTON:
                    GetTraslacion(point.x,point.y,0);
                    DibujarEscena();
                    break;
              case MK_MBUTTON:
                    GetTraslacion(0,0,point.y);
                    DibujarEscena();
                    break;
              default:GetBandera(false);
        }
        CDialog::OnMouseMove(nFlags, point);
}
```

OnMouseMove es una función asociada a la clase que se ejecuta cuando existe un mensaje del sistema operativo sobre la ventana del tipo `ON_WM_MOUSEMOVE()` y esta función recibe 2 parámetros que uno es nFlag el cual es el que indica si se ha apretado un botón del mouse y el segundo es CPoint que es un objeto con 2 atributos enteros que son las coordenadas en pixeles de la posición del cursor del mouse.

Los mensajes en Windows son los eventos que suceden al interactuar el usuario con el sistema operativo. Estos eventos se van encolando deacuerdo a su aparición, los eventos van desde apretar un botón del teclado, dar un click encima de un botón, un cambio de posición del mouse etc…

Capítulo 2.- Métodos para comparación y visualización

En la comparación de formas de objetos hay distintos métodos recientes [MARQUEZ2006], ya sea por diferencias simétricas o bien por mapeo de un objeto en un campo de distancia del objeto de referencia y asignando el valor leído a la superficie del primero (objeto a comparar), para poder efectuar una comparación exitosa se debe tener en cuenta los parámetros y los atributos extrínsecos e intrínsecos de ambos objetos en cuestión porque de ello dependerá error o diferencia que entre ellos. Dentro de los parámetros y atributos nos hallamos con la alineación de dos objetos en la cual si dos objetos en el espació no están alineados las diferencias que se miden aumentan. En esta tesis la alineación se realiza haciendo transformaciones geométricas [ver anexo A] por parte del usuario. En el laboratorio de LAIV de CCADET ya se desarrolló software que efectúa este trabajo automáticamente utilizando varios métodos como por ejemplo: Por componentes principales o penalización de rasgos. Los rasgos de un objetos que en nuestro caso son cabezas humanas, las diferencias intrínsecas son los rasgos diferentes que tienen dos cabezas como son la posición y/o tamaño de la oreja, volumen de la cabeza, mentón etc.

2.1.- Base de datos: Construcción de modelos de Cabezas con VRML

VRML (Virtual Reality Modeling Languaje) es un formato para aplicaciones de Realidad Virtual (VR). Con este formato se pueden contruir mundos virtuales tales como casas, vehículos, personas, escenas, robots, etc. La base de datos de las cabezas humanas están hechas en formato VRML.

La adquisición numérica de cabezas humanas se realizó con un escáner láser 3D de *Cyberware*. Detalles sobre antecedentes del proyecto y construcción de la base de datos se dan en [MARQUEZ2000] . El escáner rota alrededor de la cabeza produciendo información 3D de la distancia, como profundidad (en inglés “range image”). Esta adquisición de datos consiste en tomar los perfiles meridianos y variando el número de puntos dependiendo de los relieves.

También hay paso de incremento Δy entre los puntos 1mm en la altura 'y' de proyección y un paso .5mm en la posición angular θ. Las diferencias de profundidad $\Delta\rho$ pueden variar mucho especialmente detrás de la oreja. Después para la construcción del modelo se utilizan imágenes resultado de la acción del escaner en una resolución de 480×580×5 bytes en punto flotante; las imágenes se filtran para quitar el ruido y los artefactos (zonas de información faltante o espúrea), preservando la nitidez para después localizar con procesamiento de imágenes en coordenadas cilíndricas el objeto escaneado en el espacio[MARQUEZ2000]. La figura 10 y 11 muestra el concepto de escaneo y de interpretación con un cilindro e imágenes en escala de grises

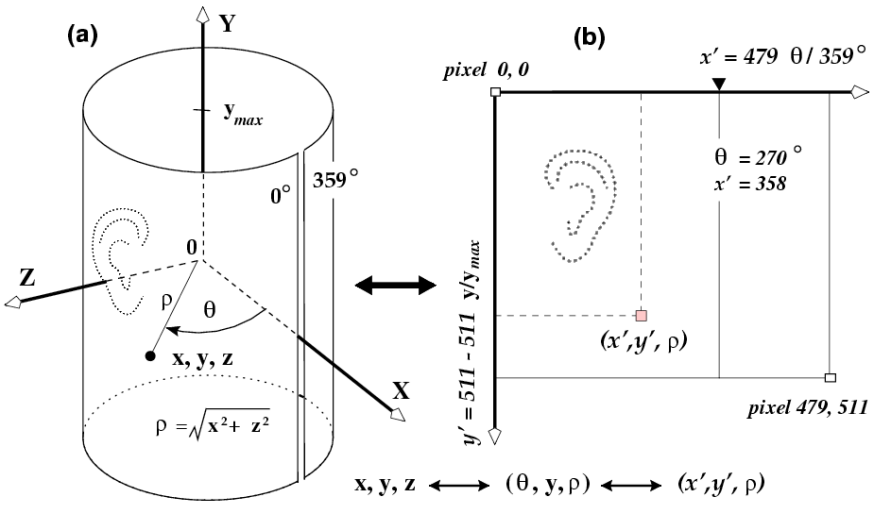


Figura 10: Representación e interpretación del escáner láser

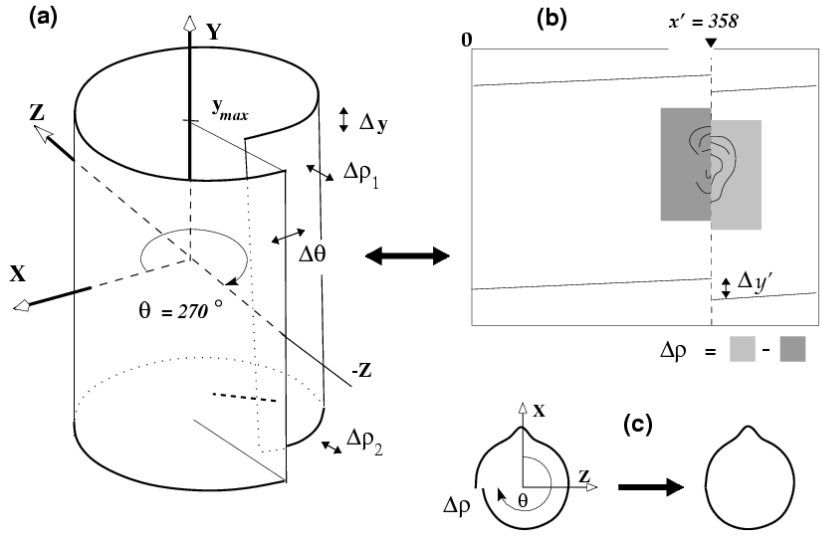


Figura 11

La construcción del *phantom* (mallado triangular) incluye formatos de conversión y técnicas de procesamiento de imágenes, obteniendo así un código de niveles de grises que va cambiando conforme va rotando el escaner. [MARQUEZ2000]

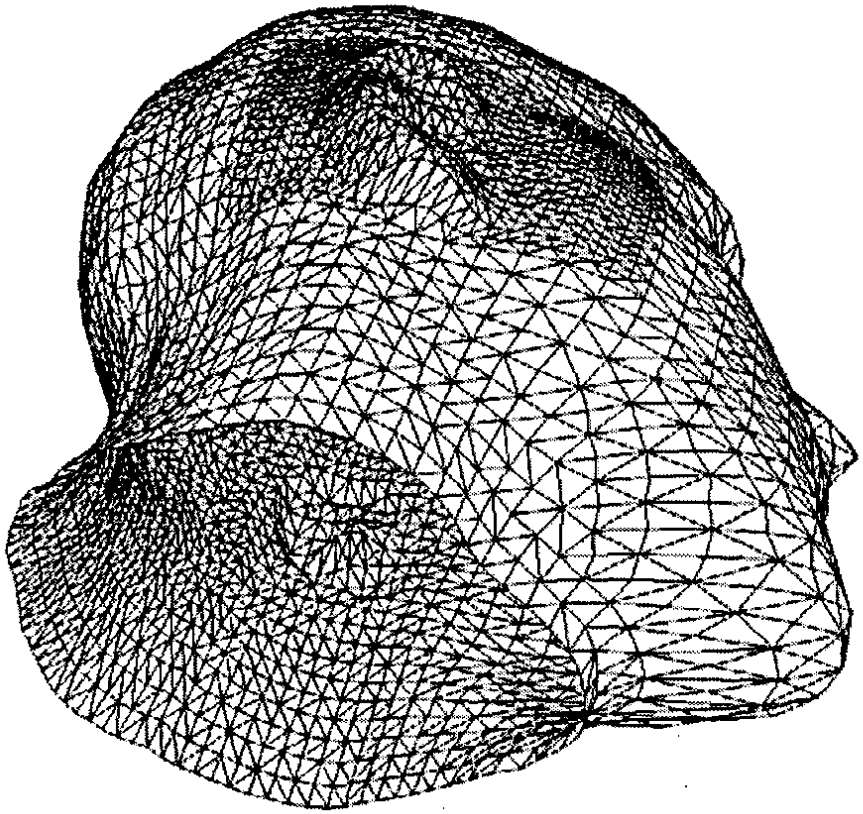


Figura 12: modelo obtenido del escáner láser. La zona alrededor de la oreja se malló a mayor resolución por ser de interés particular en el proyecto original.

El mallado de las cabezas es llevado a un archivo de formato VRML 1.0 el cual lleva una parte que son las coordenadas de los vértices y otra que lleva los índices para mandarlo a dibujar.

En formato VRML 1.0 para describir los vértices se utiliza la palabra “Coordinate3” el cual significa que se pondrán en una matriz de $n \times 3$ todos los vértices de la superficie de interés. Los vértices están relacionados entre sí para hacer una triangulación esta relación se crea con la instrucción “IndexedFaceSet” que al igual a la anterior es una matriz en la cual relaciona los vértices mientras estén antes un valor de “-1”. La matriz de vértices también se interpretan como una lista de ternas en un sistema coordenado de

tres dimensiones y se relacionan con una lista de índices necesarios para triangularizar los vértices y así construir la superficie del mallado.

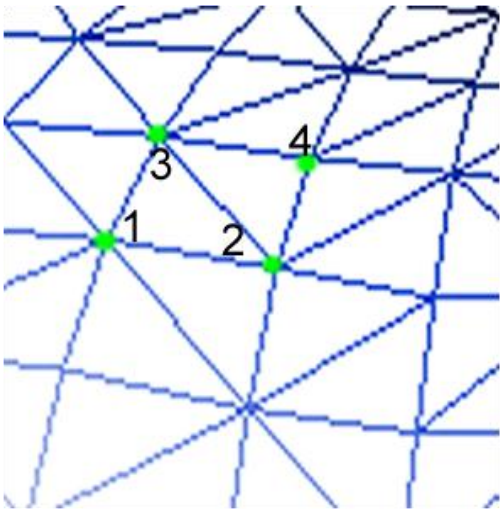


Figura 13: Agrupación de índices y vértices en VRML

Vertices:

```
Coordinate3 {
        point [
                73.381905 -91.289062 -70.172577,
                74.798843 -97.695312 -73.695381,
                        .
        ]

        }
```

Indices:

```
IndexedFaceSet {
        coordIndex [
                1,2,3,-1,
                2,3,4,-1,
                .
                .
                ]
        }
```

2.2.- Campo de distancia con la transformada Euclidiana de distancia (EDT)

Consideremos un objeto binario A, en 3D, el cuál puede tener varias componentes conexas. El *campo de distancia Euclideana* (EDT) en 3D (y mapa, en 2D), o también *transformada de distancia Euclideana* [BORGEFORS86] obtenido a partir de A, es una imagen 3D, o volumen donde el atributo del voxel (o pixel, en 2D) es la distancia Euclideana de ese punto al punto más cercano del objeto A, y de hecho es la distancia al punto más cercano del borde o frontera de A. Matemáticamente, definimos más en general el *EDT con signo del objeto A* como:

$$\mathbf{D}_{\pm}(\partial \mathrm{A}) = \{(\mathbf{p}, d(\mathbf{p})) \mid d = \mathrm{sgn} \cdot \min_{\mathbf{q} \in \partial \mathrm{A}} \|\mathbf{p} - \mathbf{q}\|\} \qquad (1)$$

$$\mathrm{sgn} = \begin{matrix} +1 & if \ \mathbf{p} \in \mathrm{A}^{\mathrm{c}} \\ -1 & if \ \mathbf{p} \in \mathrm{A} \end{matrix} \qquad (2)$$

donde **p**, **q** tienen coordenadas (*x*,*y*,*z*), en el caso tridimensional y "∂A" es la frontera (superficie externa) o "borde tridimensional" del objeto A, y puede tener varias componentes conexas. La superficie externa ∂A de un objeto discreto y binario, en 3D, se encuentra fácilmente directamente de la definición de "frontera" *6-conexa* [HERMAN1998]: es la lista de voxeles de A que tienen al menos un vecino 18-conexo (conectividad por borde) con el fondo (A^c). Si se busca una frontera *18-conexa*, entonces es la lista de voxeles de A que tienen al menos un vecino 6-conexo (por cara) con el fondo (A^c).

Existen varios algoritmos para obtener el campo de distancia en una imagen discreta los cuales se pueden adaptar para obtener el campo de distancia de un objeto tridimensional, entre esos algoritmos hallamos los siguientes:

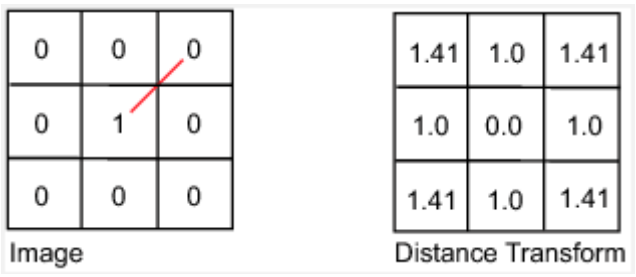


Figura 14

City block

$$V = (x, y) \quad (3)$$

$$D(V_0, V) = (x - x_0) + (y - y_0) \quad (4)$$

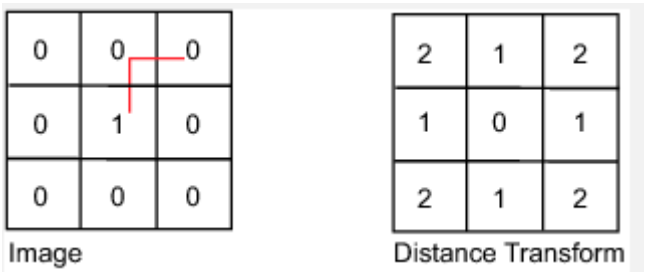


Figura 15

Chessboard

$$V = (x, y) \quad (5)$$

$$D(V_0, V) = \max((x - x_0), (y - y_0)) \quad (6)$$

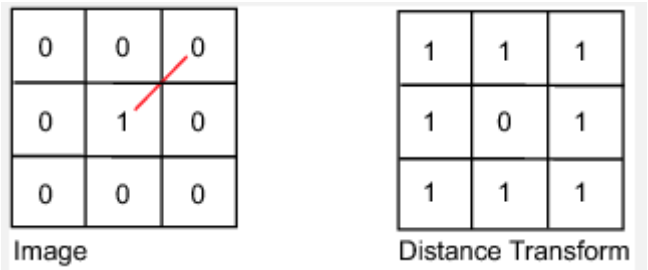


Figura 16

Quasi-Euclidean

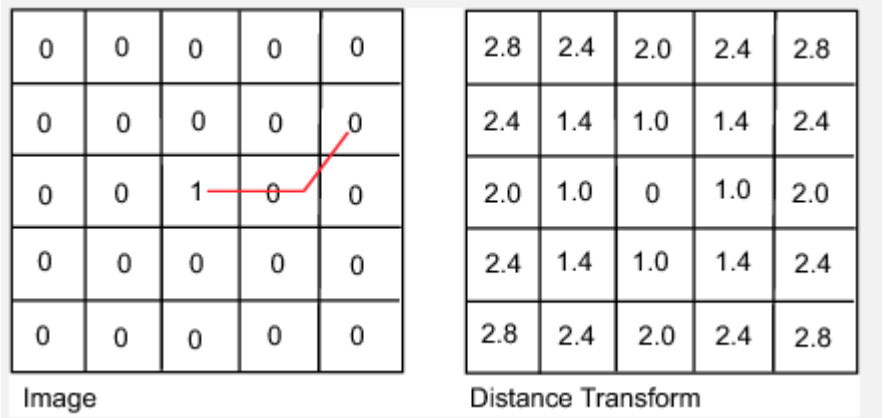


Figura 17

Se debe adaptar a 3 dimensiones algún algoritmo para obtener el campo de distancia, si ahora en vez de ver los pixeles en el plano utilizamos voxeles, entonces el algoritmo se repetiría *n* veces más en un eje de profundidad, como se muestra en la figura 43

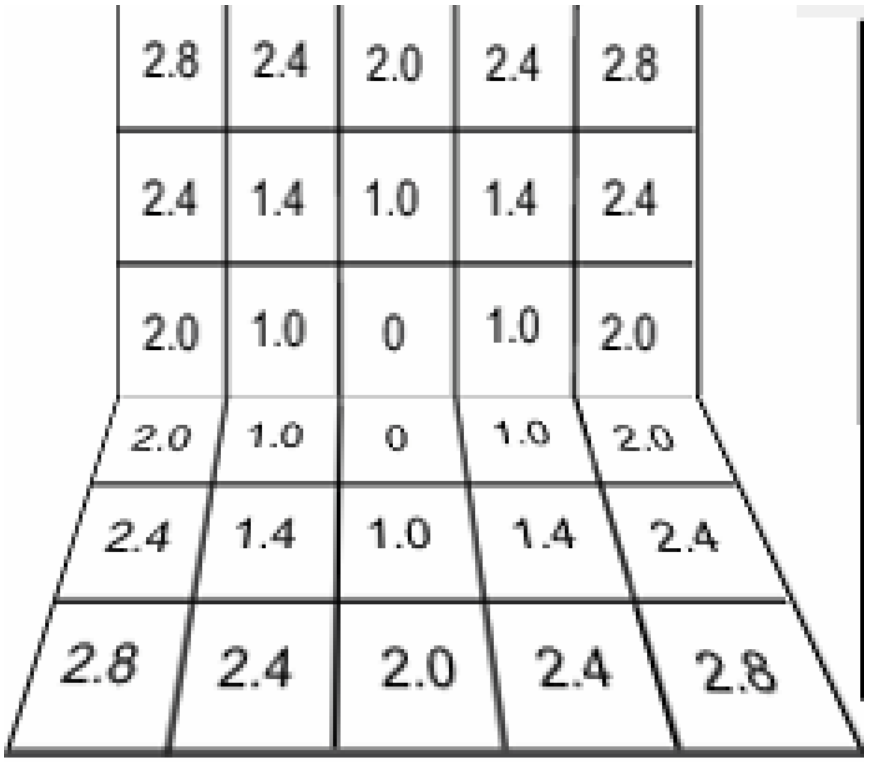


Figura 18: Campo de distancia tridimensional

Usamos en nuestro caso una implementación rápida realizada por el Dr. Márquez, en la cuál se utiliza la versión Euclideana, más exacta y aprovechando tablas de búsqueda que pre-calculan la distancia de una diferencia entre puntos. Dicha implementación, en el caso 2D, se describe en detalle en la tesis [SIMONandDAVIGNON2009].

Lo que se visualiza en la ecuación 1 es la *restricción del campo de distancia (con signo) de un objeto A a un objeto B* (o más bien, a su superficie o frontera *∂B*, representada por un mallado triangular):

$$\mathbf{D}\ \partial A\ \big|_{\partial B} \tag{7}$$

donde *la restricción de f a B, denotada como* $f|_B$ indica considerar sólo aquellos valores de *f* restringidos al conjunto *B* (o sea, *f* queda definida solamennte en (*dom f*) ∩ *B*). Como (1) asigna un valor escalar a cada punto de *∂B*, obtenemos una *frontera atribuida*.

Como los objetos en imágenes binarias, sus fronteras y el campo escalar **D** son también funciones (con atributo “1” donde hay objeto y “0” donde no), podemos escribir:

$$\mathbf{D}\ \partial A \quad x, y, z \ \Big|_{\partial B\ x,y,z} \tag{8}$$

y referirnos a puntos $(x,y,z)\in\partial A$ ó $(x,y,z)\in\partial B$. Equivalentemente, podemos considerar:

$$\mathbf{D}\ \partial B \quad x, y, z \ \Big|_{\partial A\ x,y,z} \tag{9}$$

A y *B* pueden ser cabezas distintas o puede ocurrir que simplemente $A=\mathbf{T}(B)$, o más explícitamente: $A(x,y,z)=B(\mathbf{T}(x,y,z))$, donde **T** es una transformación geométrica, por ejemplo una rotación con traslación, reducción, o en general cualquier deformación no-lineal. Si **T** es la identidad, en el caso anterior tenemos $B=A$ y por tanto:

$$\mathbf{D}\ \partial A \ \Big|_{\partial B} = \mathbf{D}\ \partial A \ \Big|_{\partial A} = 0, \quad \text{para todo}\,(x, y, z)\in\partial A\,, \tag{10}$$

pues por definición, los valores del campo de distancia son justamente cero a lo largo de ∂A. Si **T** es una rotación muy pequeña o cualquier trasformación muy ligera, los valores de $\mathbf{D}\ \partial A \ \Big|_{\partial B}$ serán también de magnitud reducida. La escala de color que se diseñó, permite distinguir valores positivos y negativos de **D,** fuera y dentro de ∂A.

En el caso más general de dos objetos distintos *A*, *B,* lo anterior permite definir un criterio del *grado de alineación* (correspondencia, registro o "matching", en inglés) entre *A* y *B,* permitiendo hallar aquella transformación $\mathbf{T}_{\min}$, aplicada a *B* que minimiza la suma de los valores absolutos del campo de ∂A restringido a ∂B:

$$\mathbf{T}_{\min} = \arg\min_{\mathbf{T}} \sum_{(x,y,z)\in\partial(\mathbf{T}(B))} \text{abs}\ \mathbf{D}\ \partial A \quad x, y, z \ \Big|_{\partial(\mathbf{T}(B))} \tag{11}$$

donde $(\mathbf{T}(B(x, y, z))) = B\ \ \mathbf{T}(x, y, z)\ \ = B(x', y', z')$, de modo que si cambia **T**, se visitan y suman puntos distintos de **D**. Nótese que en el caso de objetos distintos, $A \neq \mathbf{T}_{\min}(B)$. Normalmente la minimización se refiere al conjunto de parámetros de **T** (ángulos, traslación, escalas, etc.) que permiten minimizar la suma anterior, optimizando la alineación entre *A* y *B*.

Una vez alineados lo más posible, el valor mínimo puede interpretarse como una medida de disimilitud entre *A* y *B*, siendo cero cuando son iguales:

$$disim(A,B)=\min_{\mathbf{T}}\left\{\frac{1}{card(\partial(\mathbf{T}(B)))}\sum_{(x,y,z)\in\partial(\mathbf{T}(B))}\text{abs}\ \ \mathbf{D}\ \ \partial A\quad x,y,z\ \Big|_{\partial(\mathbf{T}(B))}\right\}\qquad(12)$$

donde se introdujo una normalización respecto al objeto a comparar, pues la suma es sobre su frontera, tras una transformación. En otras métricas, la normalización se hace respecto al objeto de referencia *A*, el cuál se mantiene fijo, pero, usando la expresión alternativa (3), se calcularía **D**(∂(**T***B*)) con cada cambio en **T,** lo cuál resultaría impráctico. Finalmente, el "grado de alineación" del que se habló, dada cualquier **T**, no necesariamente óptima, es la expresión entre corchetes, en (6), es decir:

$$match(A,B,\mathbf{T})=\frac{1}{card(\partial(\mathbf{T}(B)))}\sum_{(x,y,z)\in\partial(\mathbf{T}(B))}\text{abs}\ \ \mathbf{D}\ \ \partial A\quad x,y,z\ \Big|_{\partial(\mathbf{T}(B))}\qquad(13)$$

2.3.- Medidas de comparación morfológicas

Para medir las diferencias morfológicas un objeto se necesita un objeto de referencia, en este caso se utiliza una cabeza del cual se obtiene su campo de distancia para depués superponer otra cabeza y la superficie de la misma sea mapeado con una paleta de color. Los colores nos muestra que tan diferente es un objeto con respecto a otro y la gama de colores representa una escala de medición de que tan alejado está un puto de la superficie con el puto más cercano a la superficie de referencia.

2.3.1.- Diferencias simétricas de dos cabezas:

La diferencia simétrica de dos conjuntos se expresa como:

Sea el conjunto A y B

$$A \subseteq v_1..,v_k,...,v_n\} \qquad (14)$$

$$B \subseteq v_1..,v_k,...,v_n\} \qquad (15)$$

La diferencia simétrica es:

$$D = A \cup B - A \cap B \qquad (16)$$

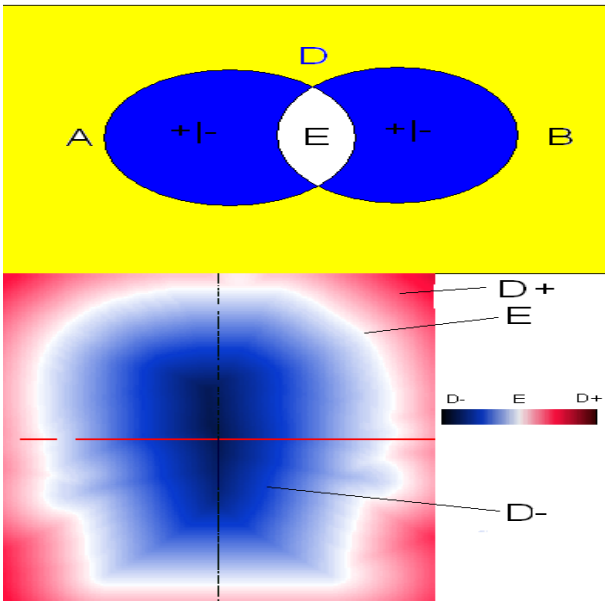


Figura 19: diagrama de Venn para diferencias simétricas de dos conjuntos en azul y la diferencia simétrica de un campo de distancia con un campo de distancia

La diferencia simétrica se usa entre dos objetos binarios, y al resultado se le obtiene su campo de distancia; este campo se puede visualizar de dos formas: como una proyección del campo en un corte transversal y la otra es visualizando la cabeza dentro de un campo de distancia y dependiendo del color representa la distancia que hay entre una superficie del modelo con la superficie de otro modelo que genera el campo de distancia en cuestión; como lo muestra la figura 20.

Si se realiza la suma cuadrática del campo de distancia diferencial en cada uno de sus componentes acotados se obtiene el error cuadratico de diferencias entre dos modelos y de la misma forma el error muestral cuadratico.

$$error^2 = \sum \frac{(\mathbf{D}_{\pm}(\partial \mathrm{A}) - \mathbf{D}_{\pm}(\partial B))^2}{N} \qquad (17)$$

$$error^2{}_{muestral}{}' = \sum \frac{(\mathbf{D}_{\pm}(\partial \mathrm{A}) - \mathbf{D}_{\pm}(\partial B))^2}{n-1} \qquad (18)$$

Si nuestra muestra del campo de distancia en cuestión, va a ser la intersección de la superficie a comparar con el campo de distancia del modelo de referencia obtenemos que el error cuadratico es:

$$error^2_{muestral}{}' = \sum \frac{(\mathbf{D}_{\pm}(\partial \mathrm{A}) \cap \partial B)^2}{n-1} \qquad (19)$$

Donde se puede observar esta intersección en la figura 20 con ayuda de una paleta de psudo-color mapeando el modelo dentro del campo de distancia $\mathbf{D}_{\pm}(\partial \mathrm{A})$.

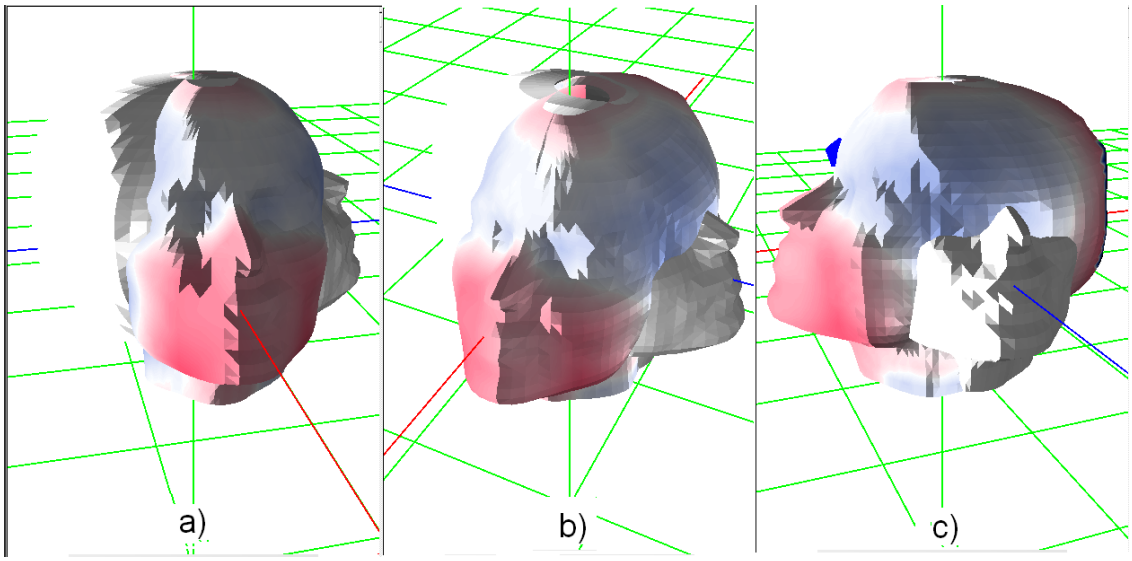


Figura 20: Modelo en diferentes alineaciones y superpuesto dentro de un campo de distancia

La diferencia simétrica entre dos objetos tri-dimensionales depende de su posición, porque puede que sean dos mismos modelos pero sin estar alineados y así aumentar la diferencia simétrica entre ambos. Para alinear los modelos en el laboratorio utilizamos el método de componentes principales para obtener una aproximación de una elipsoide y poder alinearlas con sus respectivos semi-ejes. Nuestro trabajo actual es la de visualizar dichas diferencias simétricas y el cual nos dará un error estadístico para mediciones.

El volumen de la diferencia simetrica de los dos objetos A, B, si están perfectamente alineados, sirve como una medida de la diferencia (o de similitud, si es pequeña) entre ambos, y sirve también como medida de error de alineación, al poder minimizarse durante un proceso de registro geométrico. Tenemos así dos aplicaciones de la visualización de la diferencia entre campos de distancia: (a) evaluar métodos de alineación (y de hecho como parte de un proceso iterativo de alineación) y (b) comparar dos cabezas de individuos diferentes, una vez que están alineadas.

2.4.- Métodos de visualización científica de morfometría

2.4.1.- Visualización de cabezas en un campo de distancia con una paleta de colores

La base de datos de las cabezas que se tienen están en formato VRML así que sólo se utiliza el arreglo de vértices y el de índices que se cargan en memoria para después mandar a dibujar la cabeza.

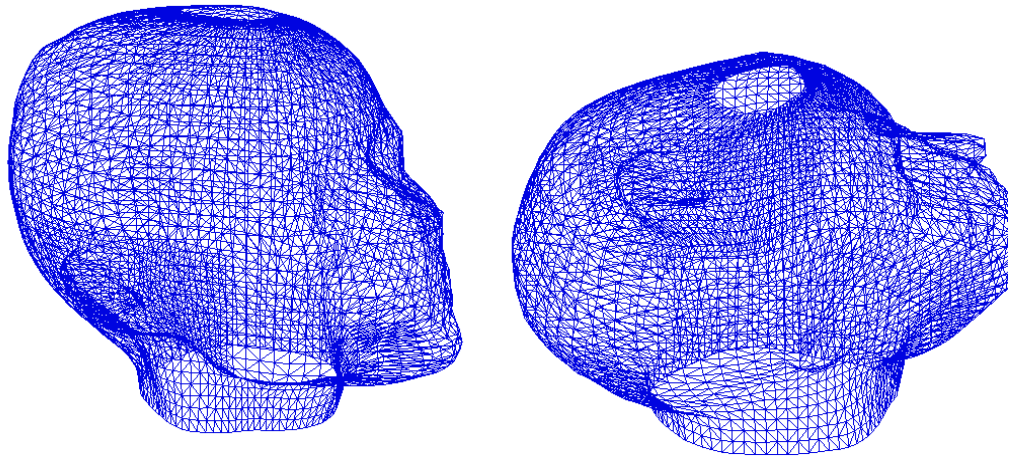


Figura 21: Modelo construido a base del archivo VRML

Ya después de visualizar las cabezas se texturizaron conforme a una paleta de colores, LUT (de *Look-Up Table*) o función de transferencia para *pseudo-color*, a partir del valor de distancia leido del campo EDT calculado para el objeto de referencia (con el que se compara). Para validar el método, se usa una esfera de referencia como objeto de control: el primer algoritmo de prueba es texturizar la cabeza dependiendo de la distancia de sus vértices con respecto a una esfera de radio *r* con centro en el origen. La esfera de referencia tendrá un radio aproximado a que cubra gran parte del phantom. [SIMONandDAVIGNON2009]

$x^2 + y^2 + z^2 = r^2$ *Ecuación de la esfera*

h_{error} : *Valor para el rango de error*

$D_{error}(x, y, z)$: *Campo a mapear*

$$D_{error}(x, y, z) = \begin{cases} x^2 + y^2 + z^2 \le (r + h_{error})^2 \\ x^2 + y^2 + z^2 \ge (r - h_{error})^2 \end{cases} \quad \forall\, x, y, z, r, h_{error} \in \Re \qquad (20)$$

Entonces, de la función anterior se necesita obtener otra función para mapear con una paleta de colores (LUT).

$$\bar{v} = x_v \hat{i} + y_v\, j + z_v k \qquad \textit{vector de posición de vértices}$$

$$f_{mapeo}(x_v, y_v, z_v) = \begin{cases} 0 & si \quad \dfrac{\sqrt{x_v^2 + y_v^2 + z_v^2} - r}{(r + h_{error})} + \dfrac{1}{2} < 0 \\ \dfrac{\sqrt{x_v^2 + y_v^2 + z_v^2} - r}{(r + h_{error})} + \dfrac{1}{2} & \\ 1 & si \quad \dfrac{\sqrt{x_v^2 + y_v^2 + z_v^2} - r}{(r + h_{error})} + \dfrac{1}{2} > 1 \end{cases} \qquad x_v, y_v, z_v \in \Re \qquad (21)$$

Del módulo del vector de posición de los vértices del phantom se resta el radio de la esfera que se está usando como referencia y es dividida entre la suma del radio de la esfera (se toma el radio de manera arbitraria para usarla como valor máximo) con la *h* de error y se hace esto para que los valores queden dentro del intervalo de la paleta de colores o sea entre 0 y 1. Se suma 0.5 al final por 2 razones

1.- Para evitar errores negativos que pueden ser mapeados
2.- Para desfasar el error 0 a 0.5 y quede a la mitad de la paleta de colores

En la figura se muestra el intervalo o gama de la textura de la paleta de colores y en la siguiente se muestra el mapeo para el error. Se muestra también, arriba de la primera una escala de grises de blanco a negro; notar que el valor de 0.5 de la escala en color corresponde a un gris medio (valor 128, en una escala de 0 a 255), en la paleta de grises:

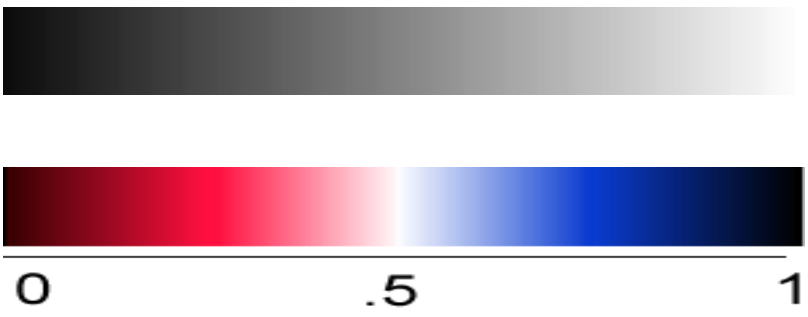


Figura 22: (Arriba) Escala de grises. (Abajo) escala en color como textura normalizada por OpenGL al intervalo [0,1].

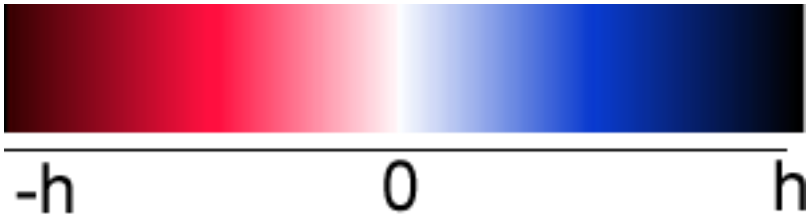


Figura 23: textura mapeada en "h" unidades de distancia con signo

Entonces al mandar a dibujar la cabeza y la esfera se puede ver que en blanco se visualiza la intersección de ambas (distancia Euclidiana cercana al valor 0.0) y los demás valores dependen de la distancia que hay entre la esfera de referencia con los vertices es el color tanto externos (distancia "positiva", en azules) como internos (distancia "negativa", en rojos), de acuerdo a la definición de la ecuación (2.1).

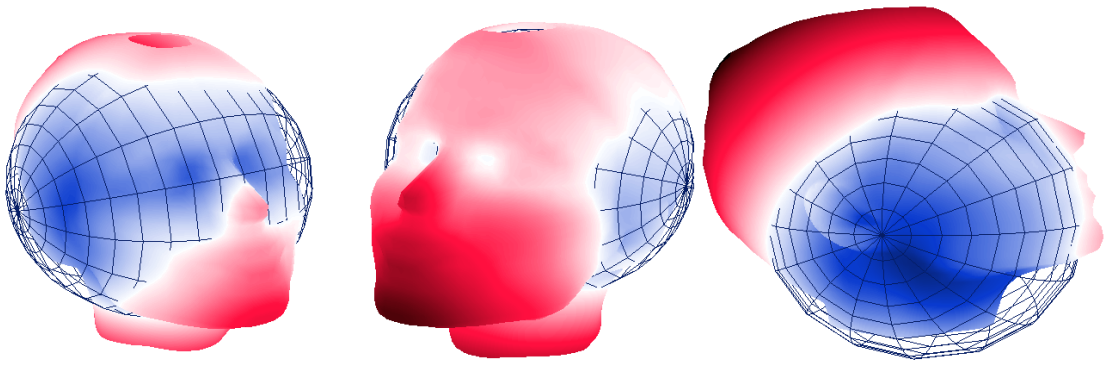


Figura 24: Representación gráfica de la diferencia simétrica utilizando una paleta de colores; de un modelo de cabeza con una esfera, renderizada en mallado transparente.

Usando MFC se colocó un "Slider Control" para aumentar o reducir el intervalo dinámico de error y con ello se obtiene un mayor contraste para el mapeo de color sobre la cabeza (Figura 20), permitiendo apreciar la distribución de distancias muy pequeñas. Notar que el negro, como valor de saturación de distancias mayores a la escala, tanto positivas como negativas, ayuda a visualizar mejor la zona cercana a la intersección.

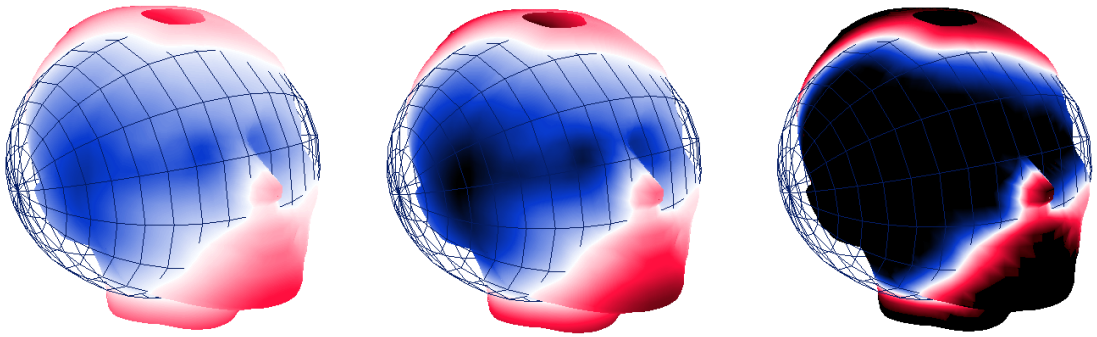


Figura 25: Resolución en las medidas de distancia utilizando diferentes contrastes

Con las ecuaciones anteriores se puede hacer una nueva ecuación más general para mapear otro tipo de mediciones y parámetros, no únicamente la distancia que hay entre los vértices de una figura con una esfera, Tal formulación quedaría como sigue:

$$f_{mapeo}\ \ g(x,y,z) = \begin{cases} 0 & si \quad \dfrac{g(x,y,z)}{\left|\max\ \ g(x,y,z)\right| + h_{rango}} + \dfrac{1}{2} < 0 \\ \dfrac{g(x,y,z)}{\left|\max\ \ g(x,y,z)\right| + h_{rango}} + \dfrac{1}{2} & \\ 1 & si \quad \dfrac{g(x,y,z)}{\left|\max\ \ g(x,y,z)\right| + h_{rango}} + \dfrac{1}{2} > 1 \end{cases} \tag{22}$$

$h_{rango} \in \Re$

$\mathrm{Im}(g(x,y,z)) \subset \Re$

Se utiliza el máximo valor que tiene la función *g* para no umbralizar vértices menores al máximo de *g* y con h se puede aumentar o reducir el contraste al mapear la figura. Con esta última ecuación solo queda definir a $g(x,y,z)$ que más adelante será el valor de la transformada de distancia Euclidiana, o una función de la misma (raíz, cuadrado o logaritmo, por ejemplo, para realzar ya sea valores pequeños o grandes). El código siguiente muestra como se programó el campo de distancia esférico y únicamente se manda a mapear los vertices antes de dibujarlos

```
bool CampoDeDistanciaEsferico(float x,float y,float z)
{
	float v_Modulo;
	float error;

	v_Modulo=sqrt(pow(x,2)+pow(y,2)+pow(z,2));
	error=(v_Modulo-r)/Rango;

	error=error+.5;

	if(error<0)	{ glTexCoord2f(0.1,0);	return true;}
	else if(error>1)	{ glTexCoord2f(0.1,1);	return true;}
	else		{ glTexCoord2f(0.1,error); return true;}
}
```

2.5.- Emulación de cómo visualizar un campo de distancia

Aunque se cuenta con un algoritmo muy eficiente para calcular EDTs, el tiempo de cálculo es muy elevado para su uso durante el desarrollo de las aplicaciones gráficas, que requieren muchas pruebas.

Se trató de emular cómo se vería un campo de distancia utilizando una paleta de colores y además que sea en objetos 3D y en tiempo real. Esto no es un campo de distancia como tal pero siguiendo el concepto de campo de distancia se emuló gráficamente como una aproximación razonable y sobretodo muy rápida. Para esto lo que se hizo fue tomar cualquier figura geométrica tridimencional y mandarla a dibujar un gran número de veces escalándola. Para mandar a dibujar el campo se hizo uso de una sucesión aritmética multiplicando cada término de la sucesión por una matriz unitaria 4 por 4 restandole una matriz de corrección para obtener así una matriz de escalamiento:

d_{gama} : *gama de propagación del campo*

$k_{resolución}$: *resolución*

$$s = i\frac{d_{gama}}{k_{resolucion}} \quad \forall\, i = 1,2,3,4,...,k_{resolucion} \mid i \in \square \ \wedge d_{gama} \in \square$$

$$I = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$ *Matriz identidad*

$$M = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$ *Matriz de corrección*

$$P_{4\times m} = \begin{pmatrix} x_0 & \cdots & x_m \\ y_0 & \cdots & y_m \\ z_0 & \cdots & z_m \\ 1 & \cdots & 1 \end{pmatrix}$$ *Matriz de vértices de la figura*

$$S_i = s(I - M) + M$$

$$C_i = S_i P_{4\times m}$$ *Conjunto de figuras emulando el campo*

Ahora la matriz C *i*-esima es una superficie que dista en una factor de $i\frac{d_{gama}}{k_{resolucion}}$ en forma radial a la superficie de la figura original.Mandando a dibujar todas las nuevas superficies obtiene una emulación del campo de distancia como lo muestra la figura

Figura 26: Aproximación de un corte transversal del campo de distancia

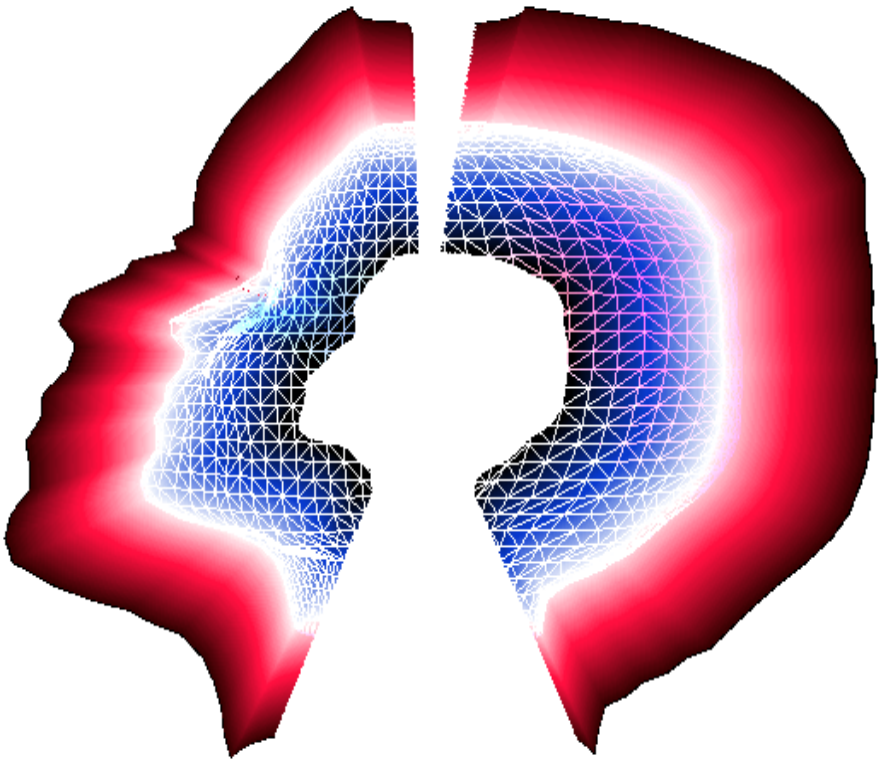


Figura 27: Aproximación de un corte transversal del campo de distancia con su respectivo modelo

El campo de distancia no se podría observar de esta manera porque la superficie más alejada hacia exterior taparía a las menos alejadas entonces para poder ver el campo se requiere hacer un corte transversal y dependiendo de la posición de la cámara, se hace dicho corte. Si se tiene la posición de la cámara y siempre está observando al origen, utilizamos ese vector, que es el vector de superficie de donde hacemos el corte además tenemos el vector de posición de cada vértice y deben ser operados de la siguiente forma:

$\bar{c} = c_x \hat{i} + c_y j + c_z k$ *Vector de posición de la cámara*

$\bar{v} = v_x \hat{i} + v_y j + v_z k$ *Vector de posición del vértice*

$\bar{p} = \bar{v} - \bar{c}$ *Vector de la cámara al vértice*

si:

$$\bar{c} \cdot \bar{p} = c_x p_x + c_y p_y + c_z p_z \quad (23)$$

$$\bar{c} \cdot \bar{p} = |\bar{c}||\bar{p}|\cos(\alpha) \quad (24)$$

y si la proyección del vector $\bar{p}$ *en* $\bar{c}$ *es*:

$$|\bar{p}|\cos(\alpha) \quad (25)$$

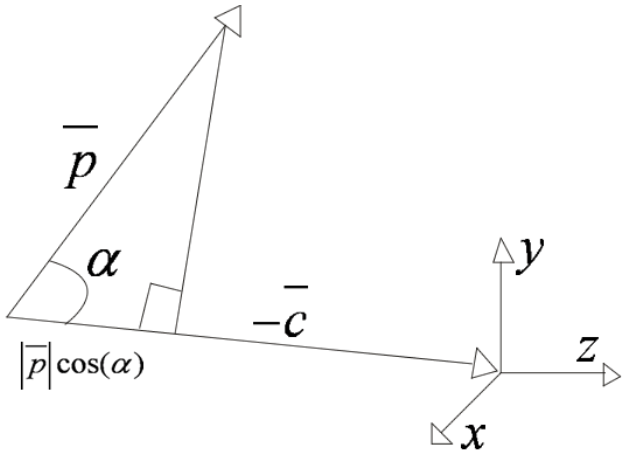


Figura 28: Gráfica de un plano de la región de corte

Entoces usando el producto punto se hacer un recorte entre 2 planos y son ello solo se manden a dibujar los vértices que pertenecen a ese intervalo entre los planos

$$f_{recorte}(\bar{p},\bar{c}) = \begin{cases} 1 & si \quad |\bar{c}| + t_{recorte} < \dfrac{|-\bar{c}\cdot\bar{p}|}{|\bar{c}|} \\ 0 & para\ otro\ caso \\ 1 & si \quad |\bar{c}| - t_{recorte} > \dfrac{|-\bar{c}\cdot\bar{p}|}{|\bar{c}|} \end{cases} \quad (26)$$

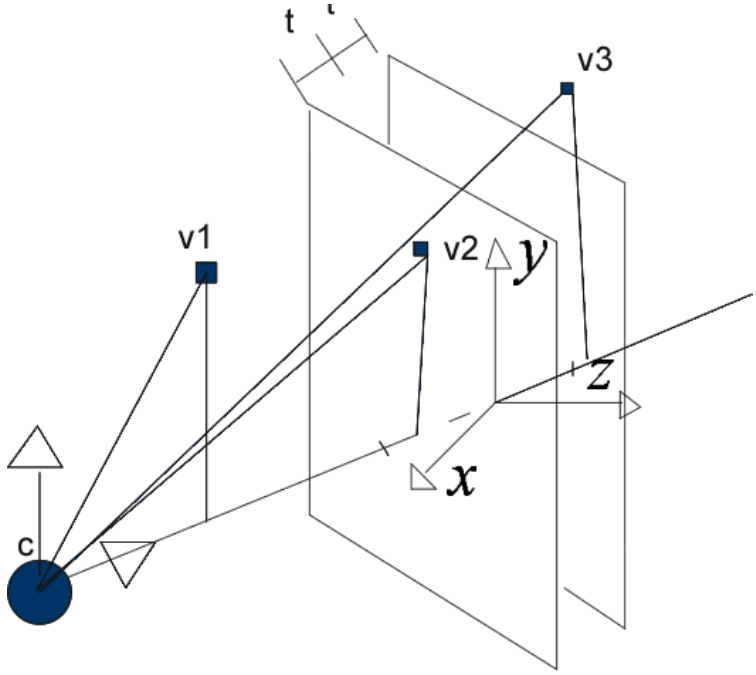


Figura 29: Modelo gráfico del algoritmo de recorte

La función resultante es una función booleana que sólo, devuelve valores verdaderos o falsos (0,1) con los cuales se va a determinar cuales vértices se mandan a dibujar y cuales no. Se muestra el código para hacer el recorte entre los dos planos.

```
V1[0]=((VertexArray[a][0]-cx)-Camara[0]/.1);
V1[1]=((VertexArray[a][1]-cy)-Camara[1]/.1);
V1[2]=((VertexArray[a][2]-cz)-Camara[2]/.1);

V2[0]=(VertexArray[b][0]-cx-Camara[0]/.1);
V2[1]=(VertexArray[b][1]-cy-Camara[1]/.1);
V2[2]=(VertexArray[b][2]-cz-Camara[2]/.1);

V3[0]=(VertexArray[c][0]-cx-Camara[0]/.1);
V3[1]=(VertexArray[c][1]-cy-Camara[1]/.1);
V3[2]=(VertexArray[c][2]-cz-Camara[2]/.1);

p1=(  V1[0]*Camara[0]*(-1)+
      V1[1]*Camara[1]*(-1)+
      V1[2]*Camara[2]*(-1))/modulo;

p2=(  V2[0]*Camara[0]*(-1)+
      V2[1]*Camara[1]*(-1)+
      V2[2]*Camara[2]*(-1))/modulo;
```

```
p3=(  V3[0]*Camara[0]*(-1)+
      V3[1]*Camara[1]*(-1)+
      V3[2]*Camara[2]*(-1))/modulo;


if(   p1>=modulo/.1-10 &&
      p2>=modulo/.1-10 &&
      p3>=modulo/.1-10 &&
      p1<=modulo/.1+10 &&
      p2<=modulo/.1+10 &&
      p3<=modulo/.1+10)
{
glVertex3f (VertexArray[a][0], VertexArray[a][1],
VertexArray[a][2]);
glVertex3f (VertexArray[b][0], VertexArray[b][1],
VertexArray[b][2]);

glVertex3f (VertexArray[c][0], VertexArray[c][1],
VertexArray[c][2]);

}
```

Todo esto da una idea de cómo se observa el campo de distancia de cualquier objeto y además se muestra en 3D y en tiempo real (el algoritmo exacto, aunque es muy eficiente, puede tardar más de media hora para una escena de 400×400×400).

2.6.- Alineación

2.6.1.- Análisis de componentes principales para alineación

El Análisis de Componentes Principales (ACP) en general es una técnica estadística de síntesis de la información, o reducción de la dimensión (número de variables). Es decir, ante un banco de datos con muchas variables, el objetivo será reducirlas a un menor número perdiendo la menor cantidad de información posible. Para ello, el resultado de aplicar ACP da eigenvalores en orden de importancia, y en una aplicación de reducción de información, se conservan aquellos cuya magnitud es mayor.

Los nuevos componentes principales o factores serán una combinación lineal de las variables originales, y además serán independientes entre sí. Un aspecto clave en ACP es la interpretación de los factores, ya que ésta no viene dada a priori, sino que se dedujo tras observar la relación de los factores con las variables iniciales.

El Análisis de Componentes Principales, en nuestro caso se utiliza para la alineación de dos modelos disminuyendo de esta forma el error simétrico. Obteniendo los eigenvalores y los eigenvectores de la distribución de vértices del modelo y estos representan los semiejes de un elipsoide ajustado al modelo de la cabeza. Por ejemplo se tiene las siguientes nubes de puntos como lo muestra la figura 30

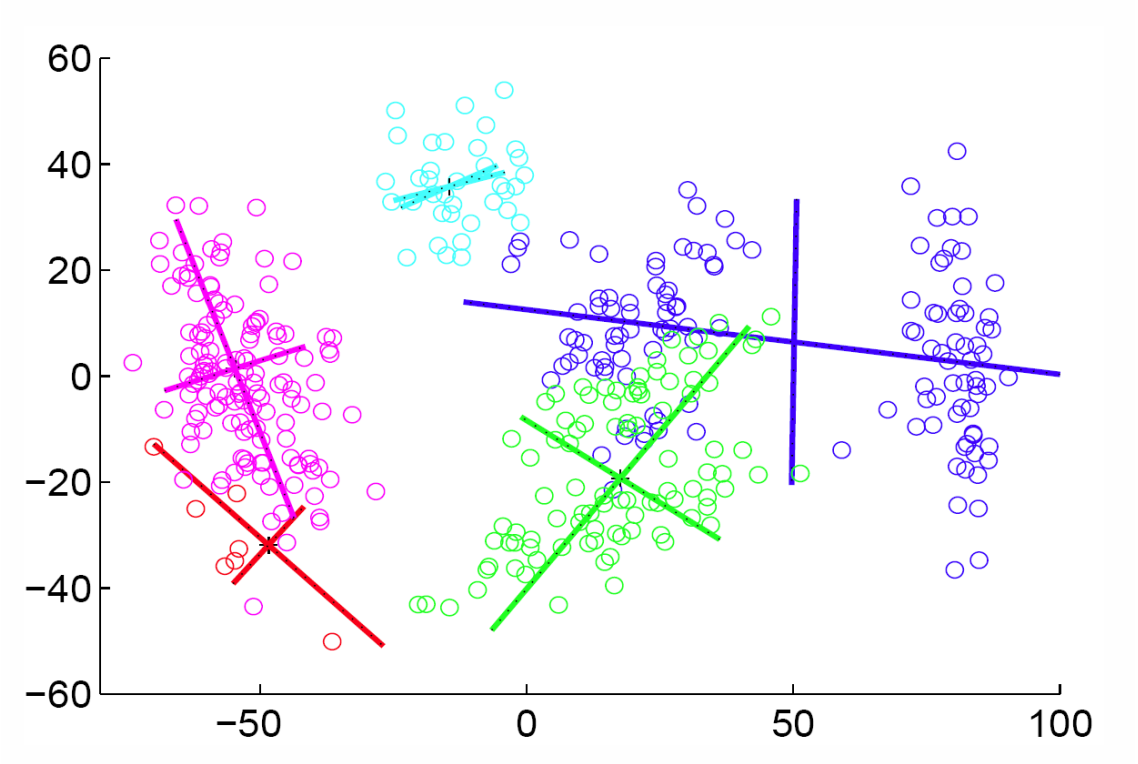


Figura 30: nubes de puntos y sus componentes principales

De la figura 30 se extraen una nube de puntos y se traza la elipsoide que representaría o ajustaría a la nube de puntos

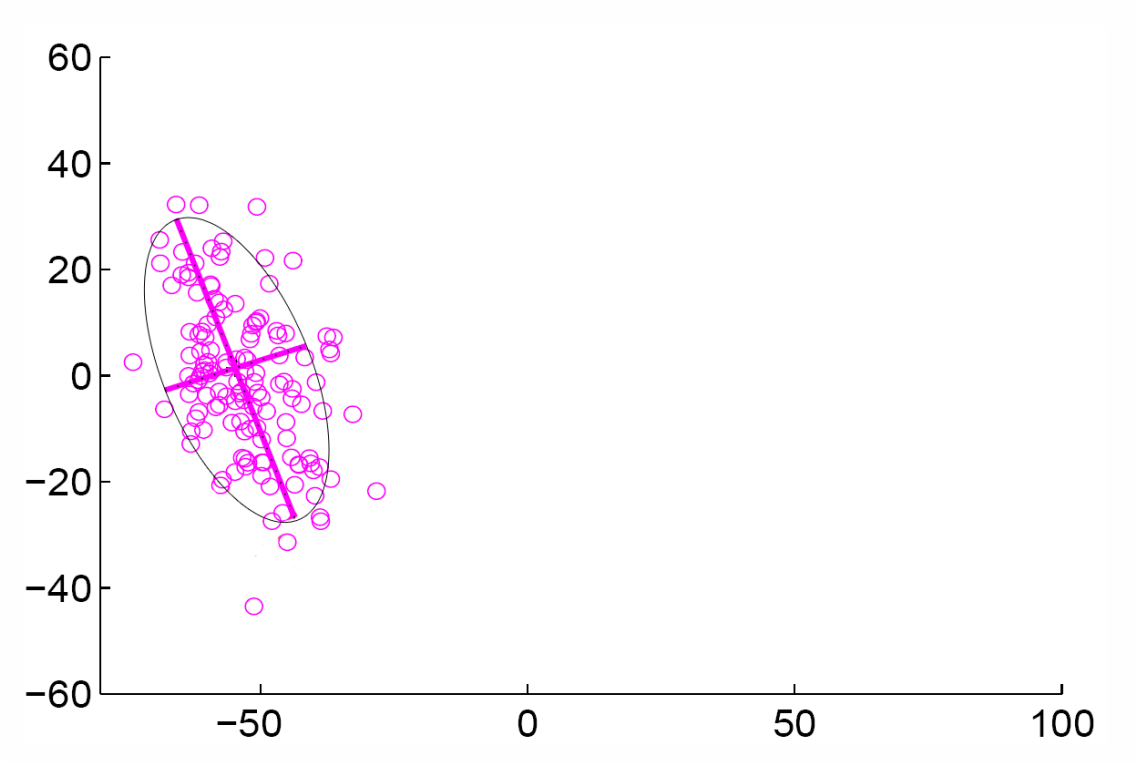


Figura 31

Se realiza lo mismo con el modelo de la cabeza con la diferencia de que será de 3 dimensiones y en vez de ser una elipse será una elipsoide como se muestra en la figura 32.

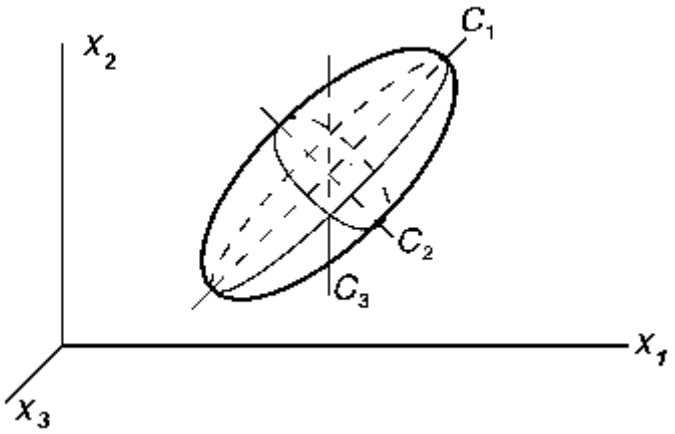


Figura 32: interpretación geométrica de las componentes principales

En la figura 33 se muestra la alineación entre la elipsoide y el modelo de la cabeza

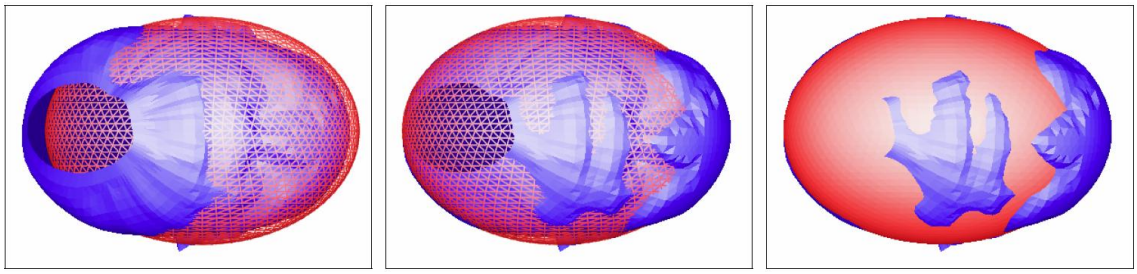


Figura 33: elipsoide que ajusta a un modelo de la cabeza humana. Figura extraída de [RAMIREZ2005].

Sea la siguiente matriz de vértices:

$$V = \begin{pmatrix} x_0 & \dots & x_n \\ y_0 & \dots & y_n \\ z_0 & \cdots & z_n \end{pmatrix}$$ Matriz de vértices

$$C = \begin{pmatrix} \bar{x} \\ \bar{y} \\ \bar{z} \end{pmatrix}$$ Centroide

El centroide se puede representar como el punto medio de una nube de vértices

$$C = E\ V \qquad (27)$$

$$E\ V\ = V \cdot \frac{1}{n} \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix}_{1 \times n} \qquad (28)$$

Para aplicar el ACP necesitamos la matriz de covariancia de los vértices

$$\mathrm{cov}(x, y, z) = \begin{pmatrix} \mathrm{cov}(x,x) & \mathrm{cov}(x,y) & \mathrm{cov}(x,z) \\ \mathrm{cov}(y,x) & \mathrm{cov}(y,y) & \mathrm{cov}(y,z) \\ \mathrm{cov}(z,x) & \mathrm{cov}(z,y) & \mathrm{cov}(z,z) \end{pmatrix} \qquad (29)$$

La covarianza se define como:

$$\mathrm{cov}(x, y) = \sum_{i=0}^{n} \frac{(x_i - \bar{x})(y_i - \bar{y})}{n-1} \qquad (30)$$

$$Q = \begin{pmatrix} \bar{x} & 0 & 0 \\ 0 & \bar{y} & 0 \\ 0 & 0 & \bar{z} \end{pmatrix} \qquad (31)$$

$$M = Q\begin{pmatrix} 1 & \dots & 1 \\ 1 & \dots & 1 \\ 1 & \cdots & 1 \end{pmatrix} \qquad (33)$$

$$V - M = \begin{pmatrix} x_0 - \bar{x} & \dots & x_n - \bar{x} \\ y_0 - \bar{y} & \dots & y_n - \bar{y} \\ z_0 - \bar{z} & \cdots & z_n - \bar{z} \end{pmatrix} \qquad (34)$$

$$\text{cov}(x, y, z) = \frac{1}{n-1}(V - M)(V - M)^T \qquad (35)$$

Obtenemos los eigenvalores y los eigenvectores que serán las componentes principales tales que representarán los semiejes de la elipsoide que ajuste al modelo

$$\text{cov}(x, y, z)v = \lambda v \qquad (36)$$

$$(\text{cov}(x, y, z) - I\lambda)v = 0 \qquad (37)$$

$$\det(\text{cov}(x, y, z) - I\lambda) = 0 \qquad (38)$$

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)

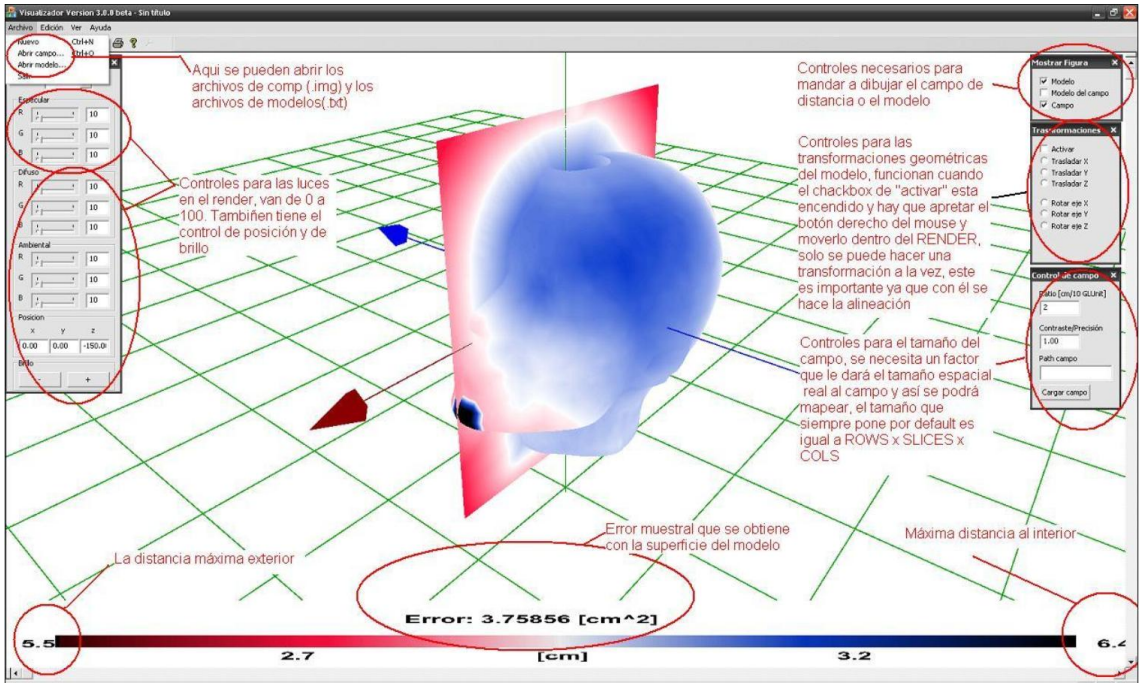


Figura 34: Descripción de la apliación

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.

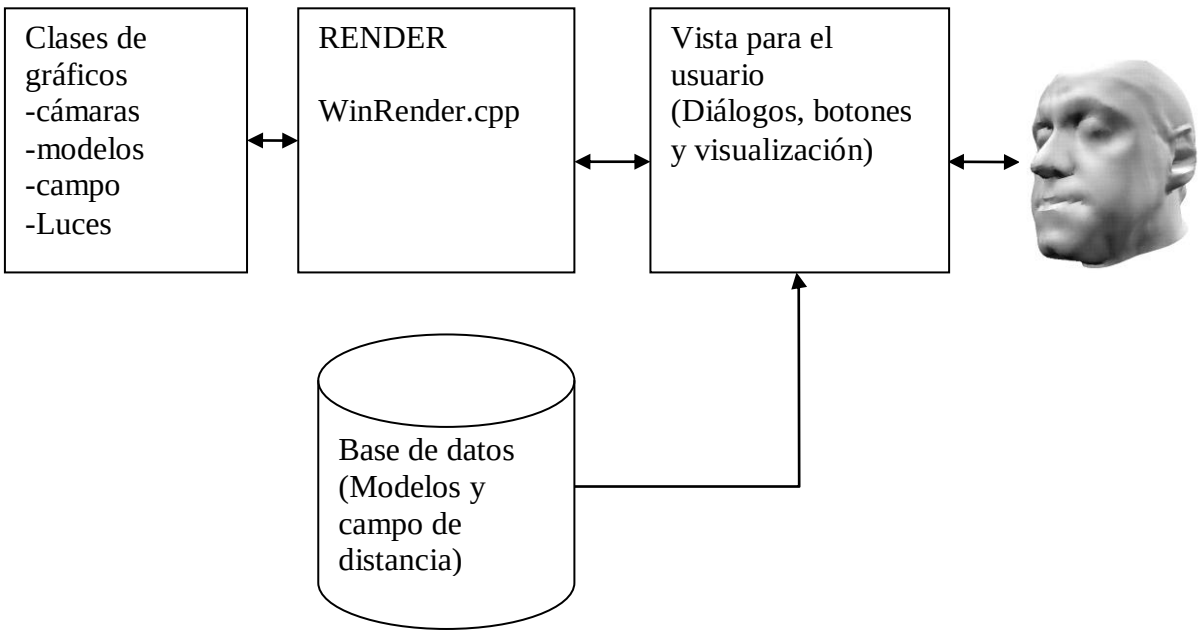


Figura 35: Diagrama de bloques del sistema gráfico

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)

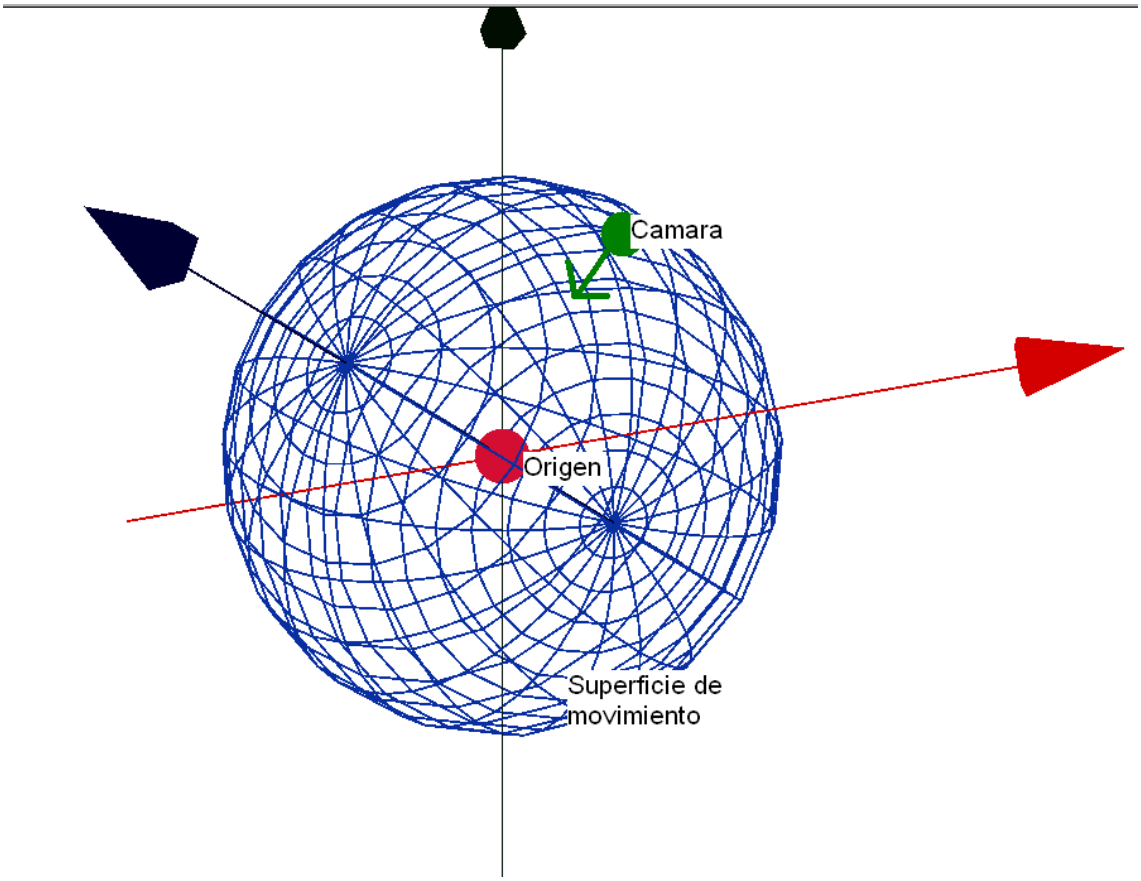


Figura 36: Superficie de movimiento de la cámara y siempre apuntando al origen del sistema coordenado

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:

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

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

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

Utilizando una acumulación para Δx

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

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

.

.

.

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

De igual forma con Δy

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

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

$$\overrightarrow{\Delta v}_{m0} = \Delta x_i \hat{i} + \Delta y_i \widehat{j} \qquad \text{Dom} \left\{ y \in \mathbb{Z} \right\} \quad (49)$$

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

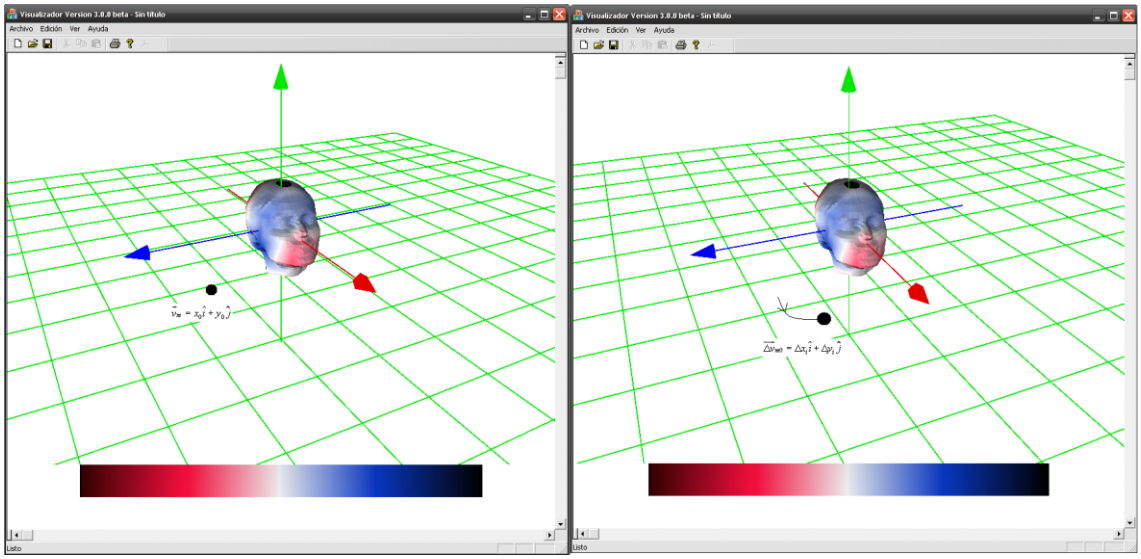


Figura 37: rotación de la cámara

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.

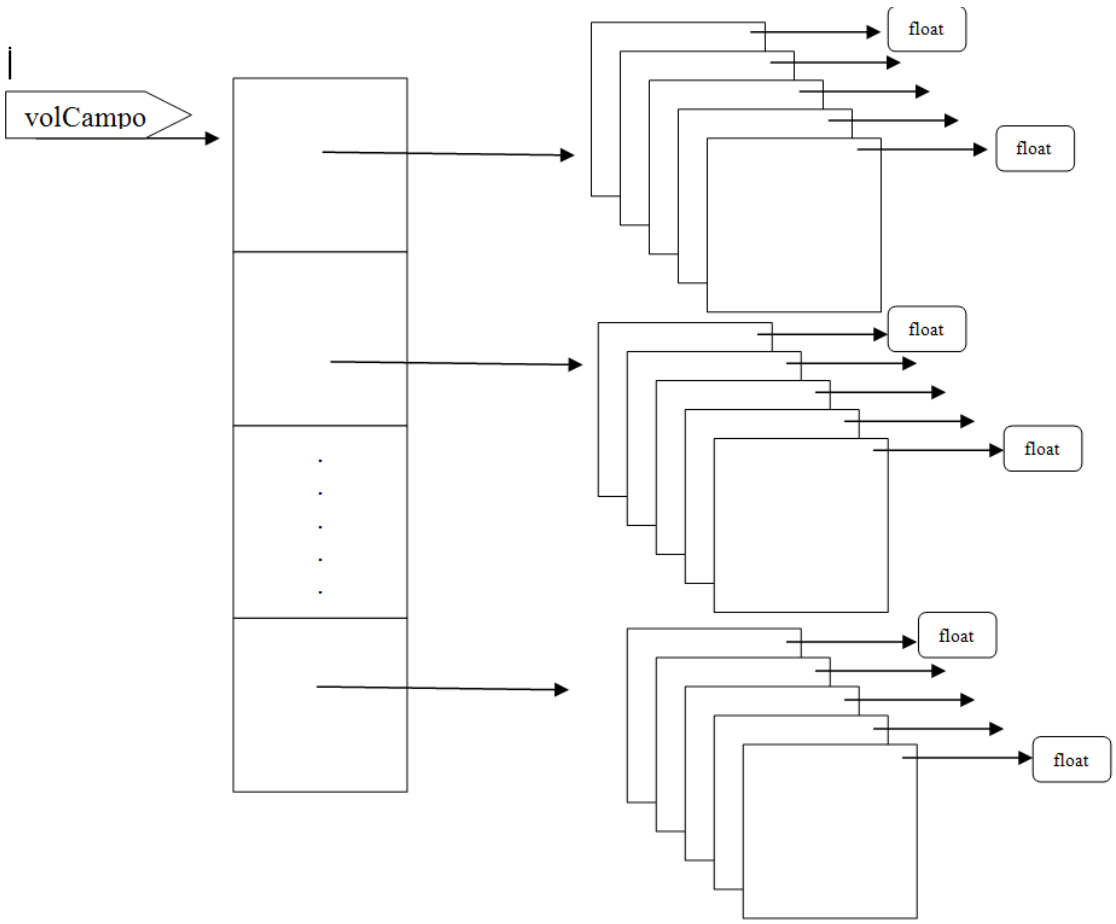


Figura 38: estructura de datos de apuntadores a apuntadores para la voxzelización. Los últimos apuntadores son a listas de floats correspondientes a renglones

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)

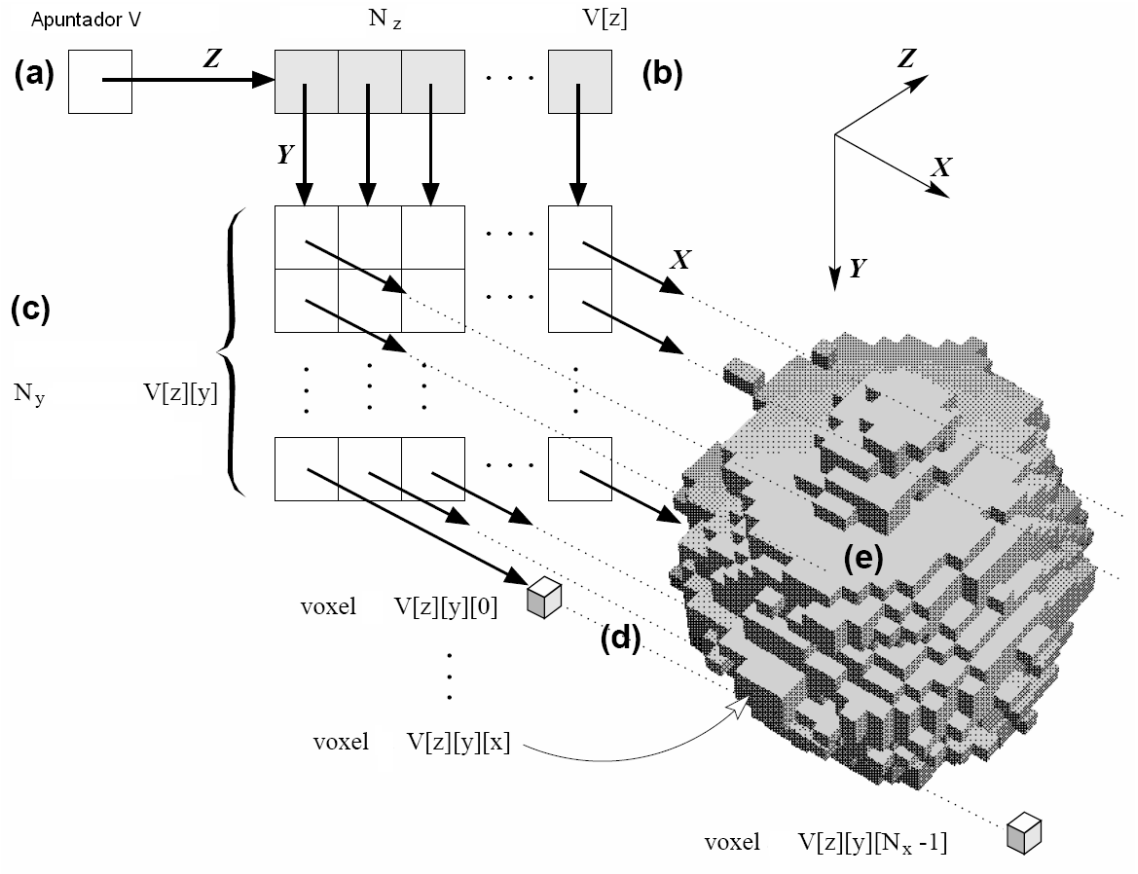


Figura 39: a) apuntador V del volumen (*V) ; b) apuntador contenido en V[z] para la componente en z; c) V[z][y] componente y del sistema coordenado; d) V[z][y][x] componente en x y devuelve el valor de la distancia en ese punto coordenado (x, y, z); e) gráfica de la voxelización**

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:

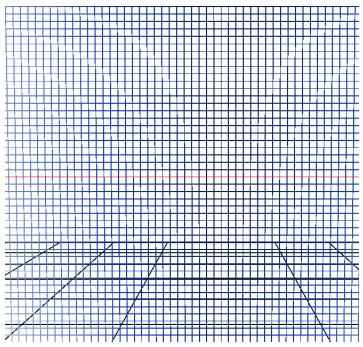


Figura 41: mallado de los puntos a texturizar

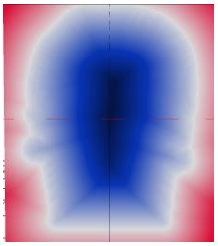


Figura 42: mallado con un texturizado solido (GL_POLYGON)

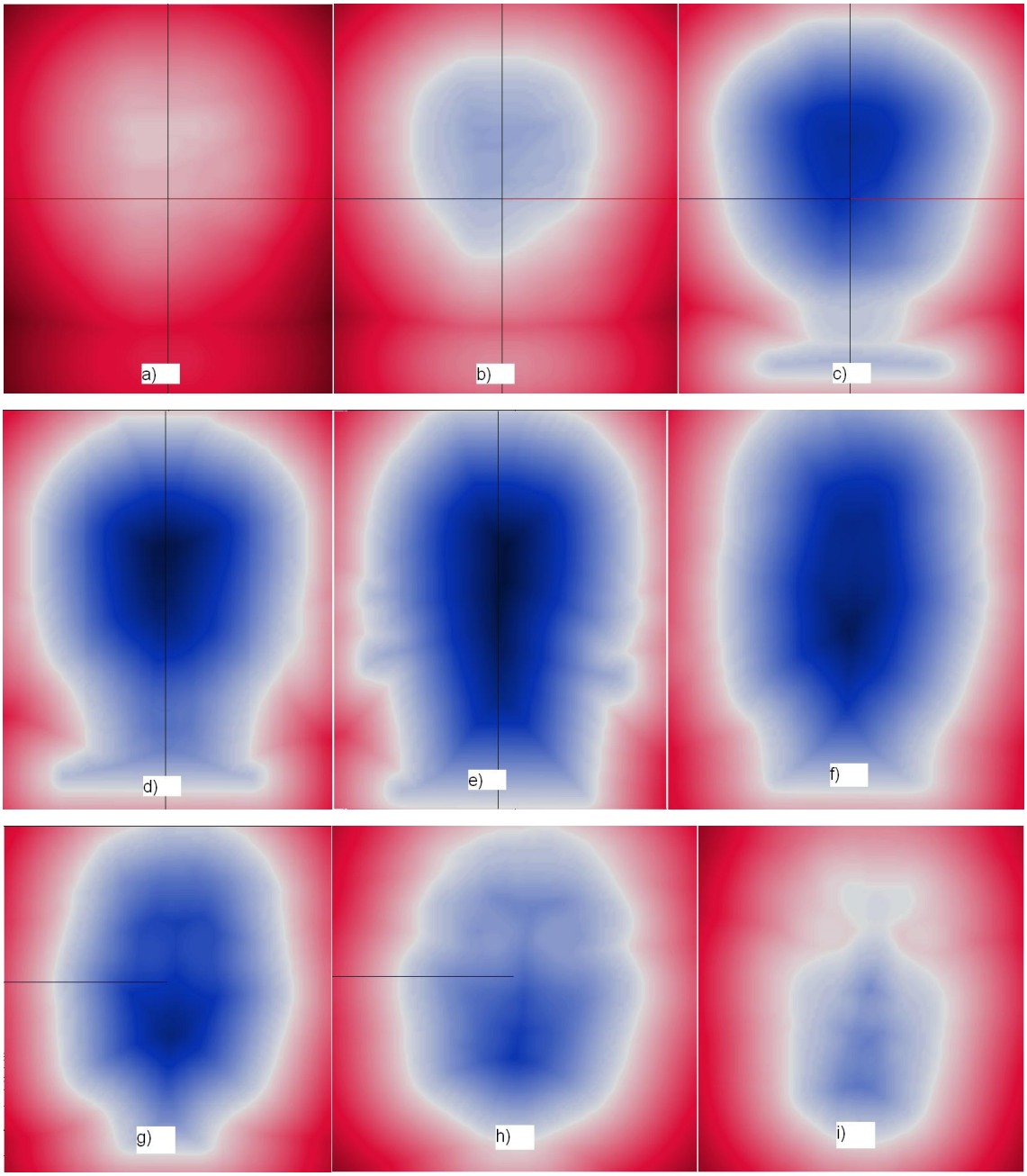


Figura 43: Múltiples cortes transversales del frente de una cabeza humana en su campo de distancia: a) tangente al parietal; b) parietal y el occipital; c) temporal y el parietal; d) frontal, mandíbula y parte del parietal; e) esfenoides, frontal y mandíbula; f) frontal y cigotomático; g) etmoides y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

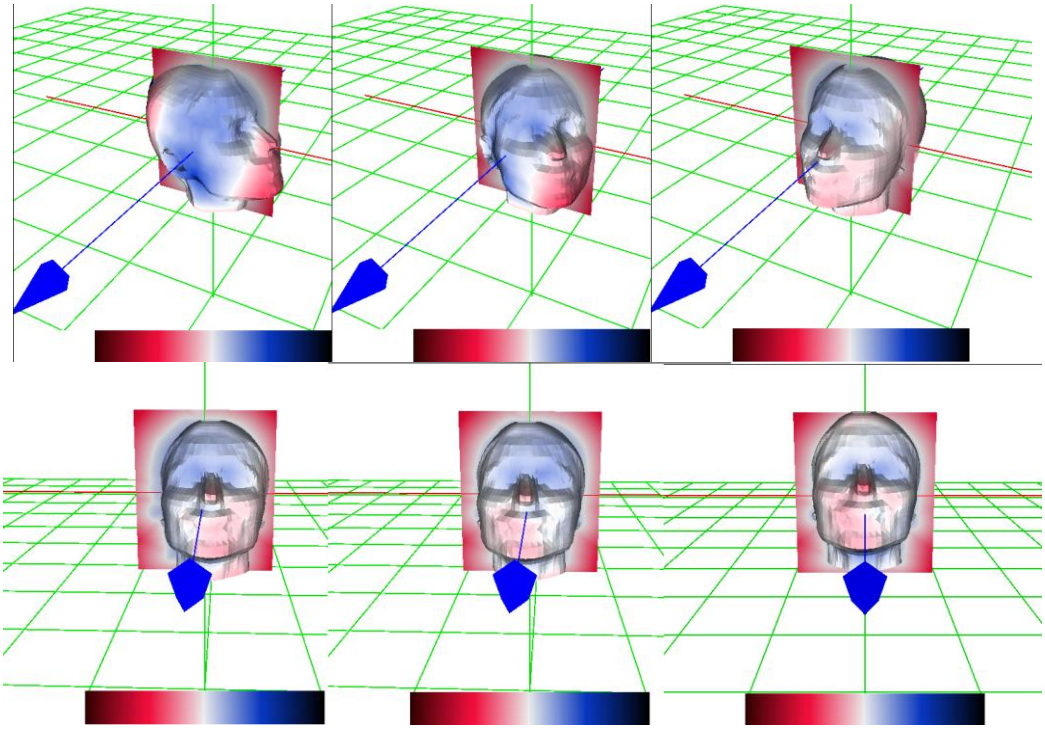


Figura 44: Múltiples vistas del modelo y un corte transversal del campo de distancia

Capítulo 4.- Resultados y discusión

El resultado generado con el trabajo realizado fue un software que permite visualizar las diferencias morfológicas de dos objetos, utilizando campos de distancia, alineación, diferencia simétrica y texturización. En este capítulo se documenta los resultados que se obtuvieron utilizando el software desarrollado aplicado a la antropometría craneofacial. Esto es, se comparan diferentes cabezas y notando las diferencias por rasgos entre diferentes modelos.

4.1.- Comparación entre 2 objetos.

La primera comparación va a ser de una cabeza consigo misma, en la que observaremos que entre mejor alineada queda la cabeza más blanca se verá y el error cuadrático tiende a ser cero.

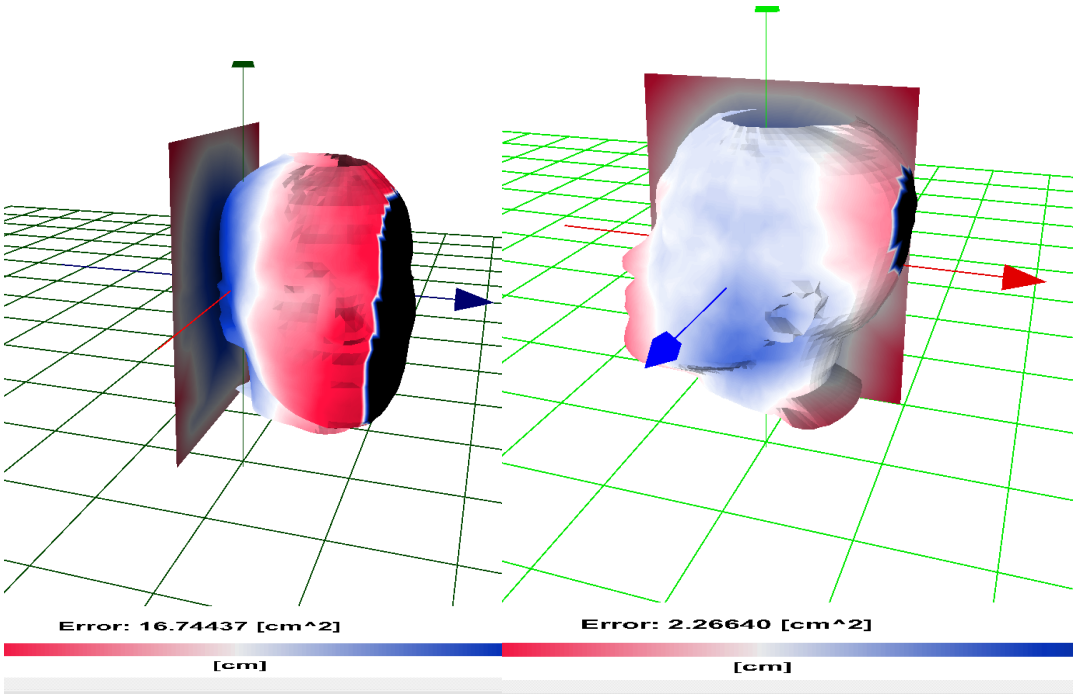


Figura 45: A la izquierda el modelo sin alinear, a la derecha la primera iteración para alinear

En las figuras 44 y 45 muestra el error en cm^2 y como se puede observar aunque el campo de distancia pertenece a esa cabeza, si no está alineado, entonces el error aumenta. El corte transversal del campo ayuda a que se pueda alinear la cabeza. El error que se muestra en las figuras es un error muestreal donde solo se toman los puntos que

interseca la superficie del modelo al campo. Este error se obtiene en tiempo real dentro la aplicación calculándose al mismo tiempo que se va mapeando el color sobre la superficie del modelo de la cabeza.

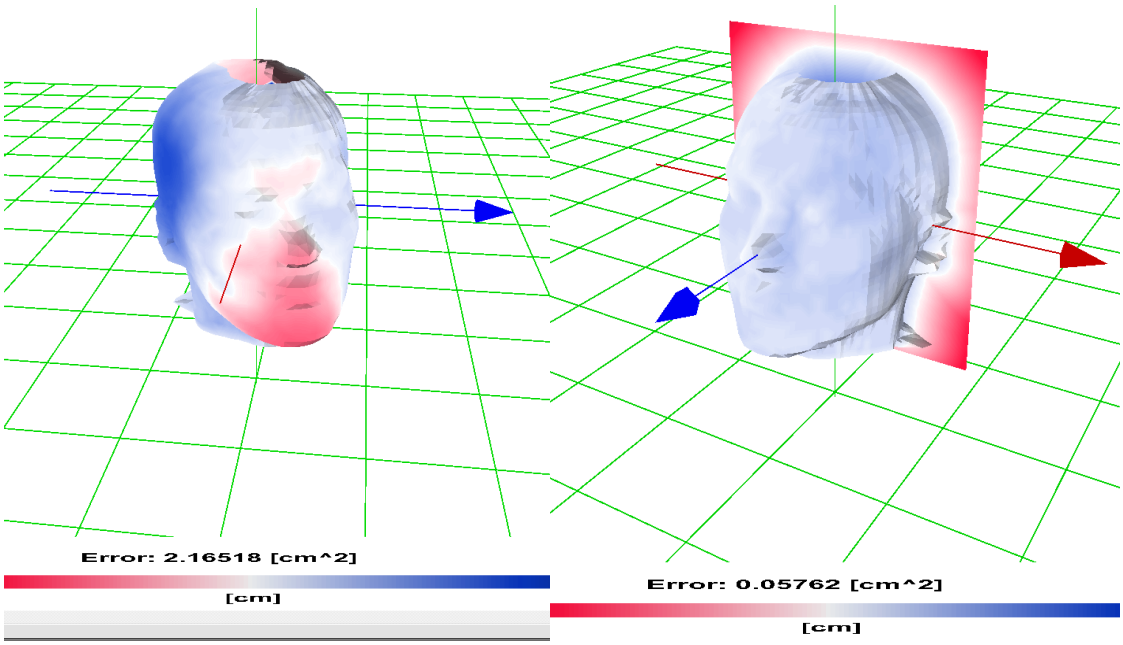
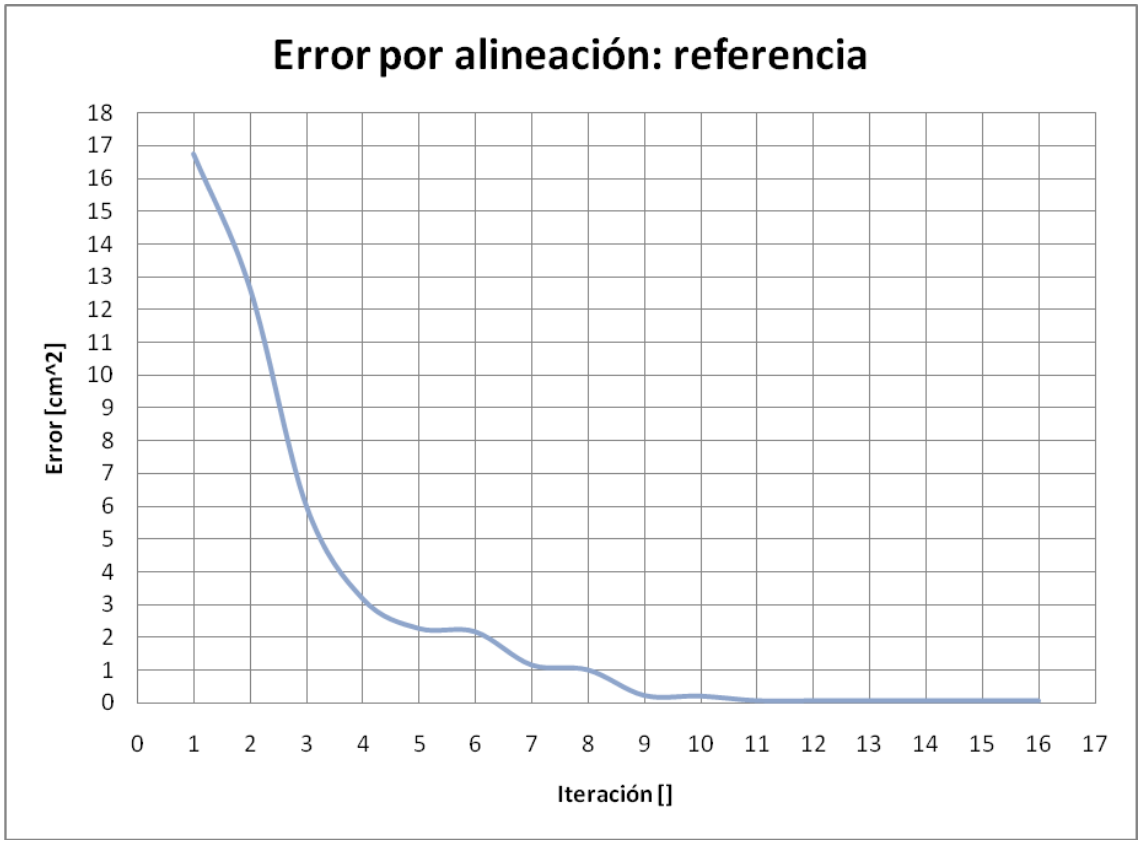


Figura 46: Iteraciones de alineación acercando al error a 0

En la tabla 1 se muestra con 16 iteraciones alineando la cabeza y el error se aproxima a cero el error mientras más alineada esté.

1	16,74437
2	12,65768
3	6,01911
4	3,17564
5	2,2664
6	2,16518
7	1,15263
8	1,00156
9	0,22147
10	0,20351
11	0,06234
12	0,06023
13	0,0598
14	0,058283
15	0,05802
16	0,05762

Tabla 1: Error por iteración de una cabeza en su propio campo de distancia



Gráfica 1: Alineación de una cabeza con su propio campo de distancia

Enseguida se compara una cabeza diferente con la cabeza anteriormente expuesta. Utilizando el campo de distancia se va a texturizar la cabeza y se obtuvo un valor de error mayor.

1	19,65191
2	11,86322
3	9,12837
4	8,28919
5	7,78735
6	7,00495
7	3,99732
8	2,8369
9	2,8004
10	2,8855
11	2,74359
12	2,64276
13	1,7649
14	1,78453
15	1,75916
16	1,76064

Tabla 2: error por iteración de una cabeza en un campo de distancia

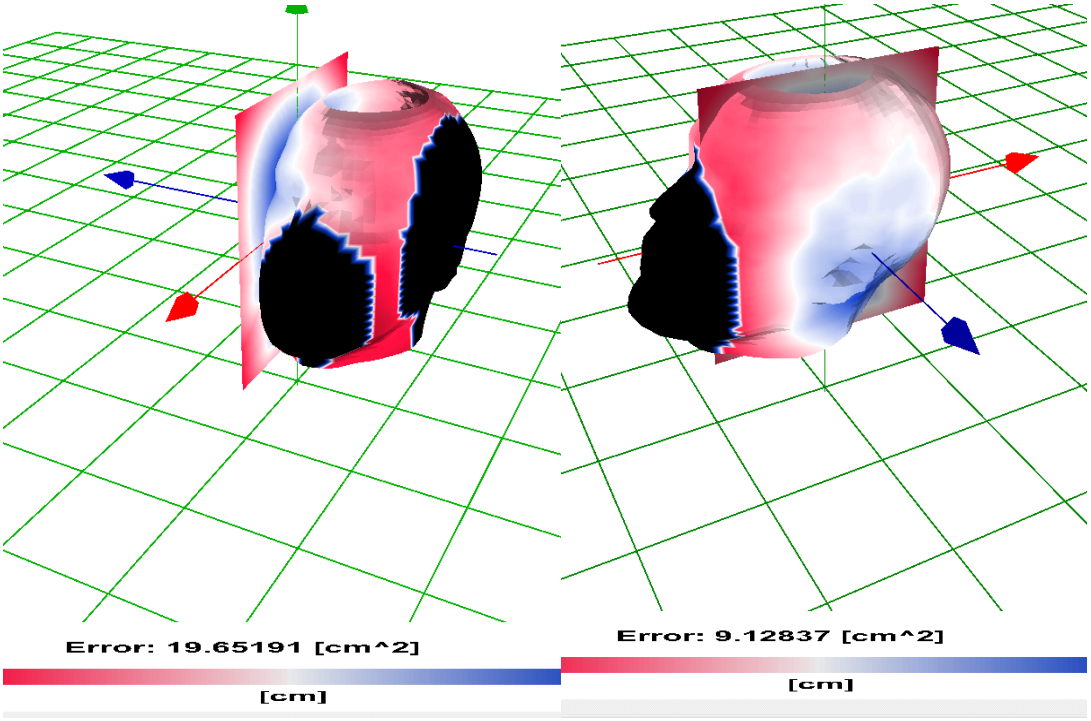


Figura 47: A la izquierda el modelo sin alinear, a la derecha la primera iteración para alinear con diferente cabeza

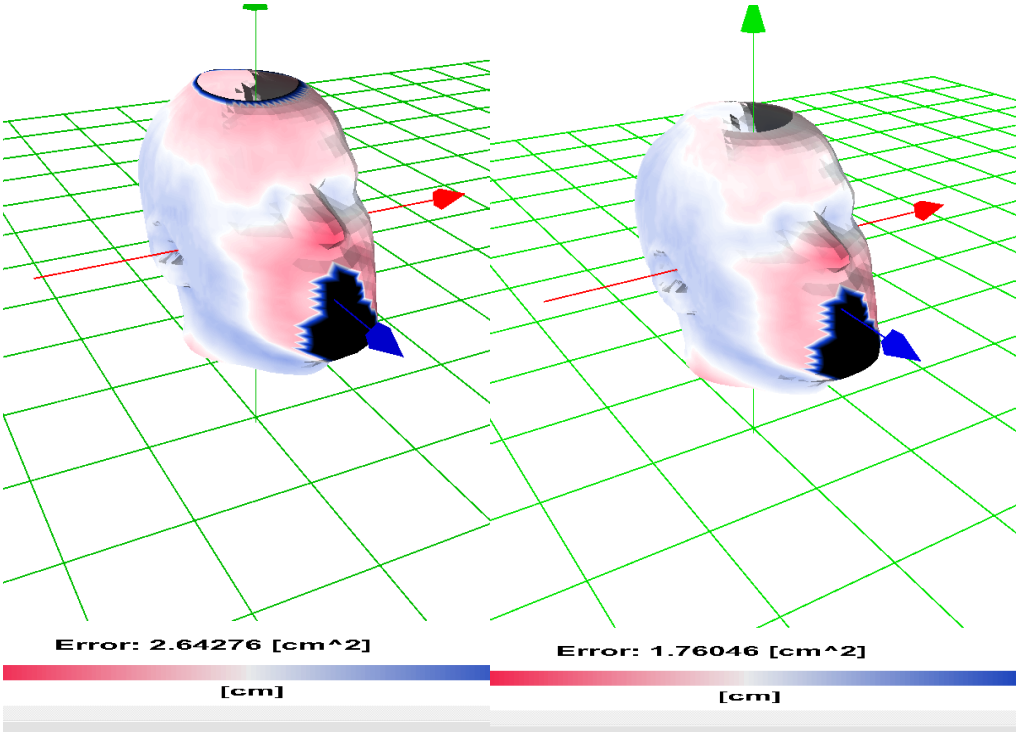


Figura 48: Iteraciones de alineación acercando al error a 0 con diferente cabeza

Gráfica 2: Alineación de una cabeza en un campo de distancia

Se vé claramente en la gráfica 2 que el error es muy grande a comparación de la referencia, esto significa que existe una gran diferencia entre las dos cabezas aproximadamente de 2[cm^2] de error. En la gráfica 3 se puede ver la diferencia que existe entre amabas cabezas.

Gráfica 3: Comparación de los errores en con una cabeza de referencia (azul) y una cabeza de muestra (roja)

Conclusiones:

El campo de distancia ha sido un método fundamental para el desarrollo de la comparación antropológica de las cabezas humanas, lograr interpretar las diferencias extrínsecas mediante una paleta de color ha ayudado a visualizar las diferencias entre los modelos mediante su campo de distancia. Utilizando el color para mostrar esas diferencias se pueden crear índices de deformación y en el área médica poder calcular de manera más eficiente y más exacta la deformación de un modelo y con ello hacer una corrección plástica más exacta.

Con el error se puede obtener datos estadísticos de las diferentes formas de rostro y cabeza que hay en las distintas razas entre humanos la cual ayudará a obtener una biblioteca antropológica, que requiere de modelos representativos cuya obtención implica además una alineación correcta, de modo que la herramienta cumple con un doble fin: calcular y visualizar error de alineación y similitud entre objetos alineados.

El error podrá tender a cero aunque es muy poco probable que llegue a ser cero aunque sea una comparación de una cabeza consigo misma y esto es debido a que se pierde información al voxelizar el campo de distancia; si se utiliza tecnología de 64 bits se logrará agregar memoria física para direccionar a la PC y con ello se tendrán más recursos de memoria para cargar un campo de distancia con mayor definición. Un campo de distancia no se podrá comparar con otra forma si estas dos tienen una gran diferencia de tamaños porque al hacer las operaciones pertinentes unas van a tender a cero en la diferencia y no se mostrará la verdadera diferencia. Otro posible problema es que si el campo de distancia está trasladado lejos del origen también se puede perder información por los decimales que son truncados.

El sistema va a facilitar en el laboratorio la observación y la interpretación de las diferencias entre cabezas además de mostrar y corregir la alineación de una manera interactiva el cual reduce tiempos en cálculos y procesamiento de imágenes.

Anexo A

Transformaciones en 3D

Las transformaciones en 3D son operaciones para modificar la posición geométrica de puntos o vértices en el espacio. Un punto en el espacio se representa como una terna de números para obtener su localización en un espacio coordenado. Estos puntos también pueden ser representados como una suma vectorial o una matriz como lo muestran las siguientes ecuaciones:

$$1)\ \bar{v} = x\bar{i} + y\bar{j} + z\bar{k}$$

$$2)\ \bar{v} = (x, y, z)$$

$$3)\ \bar{v} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

La primera expresión es una suma vectorial de 3 componentes donde las letras testadas $\bar{i}, \bar{j}, \bar{k}$ son los vectores unitarios (1,0,0), (0,1,0) y (0,0,1) en un sistema coordenado rectangular, la segunda es una terna de números y la tercera forma es la representación matricial de dichos vectores de posición. La tercera forma es la que se usará para realizar las operaciones matriciales de transformación geométrica.

Las tres transformaciones más comunes son la de traslación, rotación y escalamiento. La traslación es una suma vectorial del vector de posición del punto mas el vector de traslación, el escalamiento es una multiplicación por 3 factores en cada una de las componentes del vector y la rotación también es una multiplicación de los elementos del vector pero los factores con los que se multiplican son el resultado de aplicar una función trigonométrica a un ángulo de rotación.

Si se quieren hacer múltiples transformaciones a un punto la mejor forma de hacerlo es agregando un valor $h=1$ al vector y cambiar la traslación a una multiplicación de matrices, estas constituyen las ecuaciones en *coordenadas homogéneas*, y permite convertir en transformación lineal el conjunto de rotaciones y escalamientos mas la traslación.

$$V=\begin{pmatrix} x \\ y \\ z \\ h \end{pmatrix}=\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \qquad \textit{Nuevo vector con } h=1$$

$$T=\begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad \textit{Matriz de traslación}$$

$$R_x=\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad \textit{Matriz de rotación en x}$$

$$R_y=\begin{pmatrix} \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad \textit{Matriz de rotación en y}$$

$$R_z=\begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad \textit{Matriz de rotación en z}$$

Esto facilita mucho las operaciones al ser solamente multiplicaciones de matrices y si se va a hacer a muchos puntos las mismas transformaciones se puede pre calcular una matriz de transformación y se reduce el número de operaciones. Para aplicar las transformaciones 3D a un polígono se deben tener todas las coordenadas de cada uno de sus vértices para representarlos en una matriz y realizar las operaciones matriciales.

El efecto que producen las transformaciones en 3D es apreciable cuando se habla de múltiples puntos o múltiples vértices como se muestra a continuación

$$P_{4\times m} = \begin{pmatrix} x_0 \cdots x_m \\ y_0 \cdots y_m \\ z_0 \cdots z_m \\ 1 \ \cdots \ 1 \end{pmatrix} \qquad \textit{Matriz de vértices de un polígon}$$

$$T_{4\times 4} = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \textit{Matriz de traslación}$$

$$P_T = (T_{4\times 4})(P_{4\times m})$$

$$P_T = \begin{pmatrix} x_0 + t_x \cdots x_m + t_x \\ y_0 + t_y \cdots y_m + t_y \\ z_0 + t_z \cdots z_m + t_z \\ 1 \qquad \cdots \qquad 1 \end{pmatrix}$$

Por ejemplo una traslación de un prisma queda como sigue:

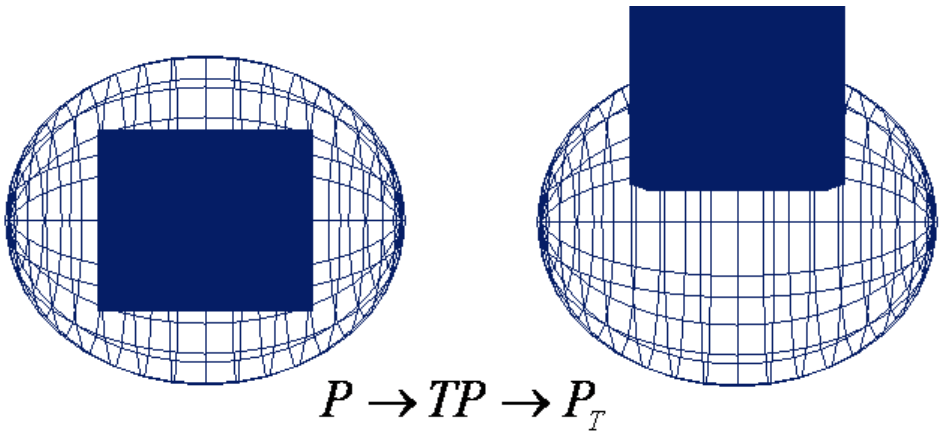


Figura 49: Traslación

Una rotación se aplica el mismo principio pero con otra matriz al igual que el escalamiento

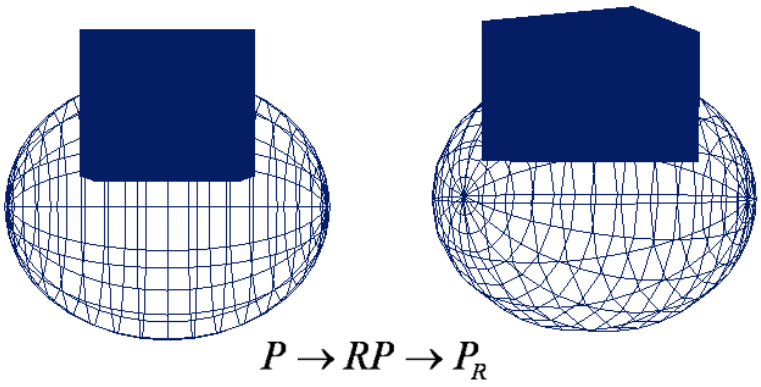


Figura 34: Rotación

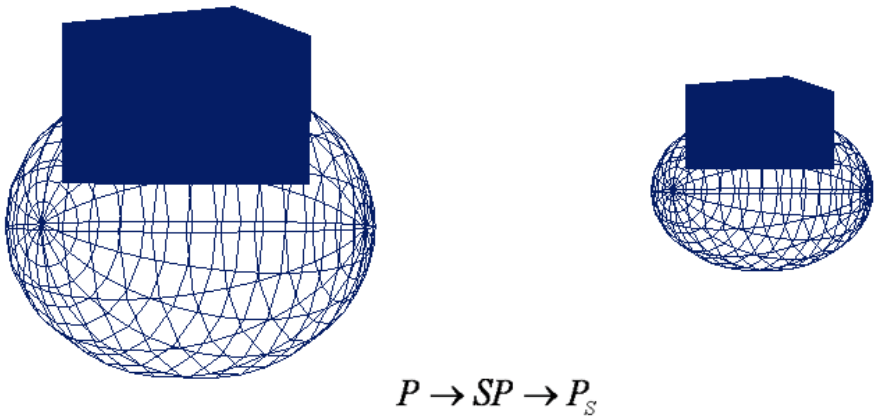


Figura 50: Escalamiento

Cabe aclarar que la multiplicación de matrices no es conmutativa (en la mayor parte de los casos) así que el resultado final varía si se hacen las mismas transformaciones en orden diferente

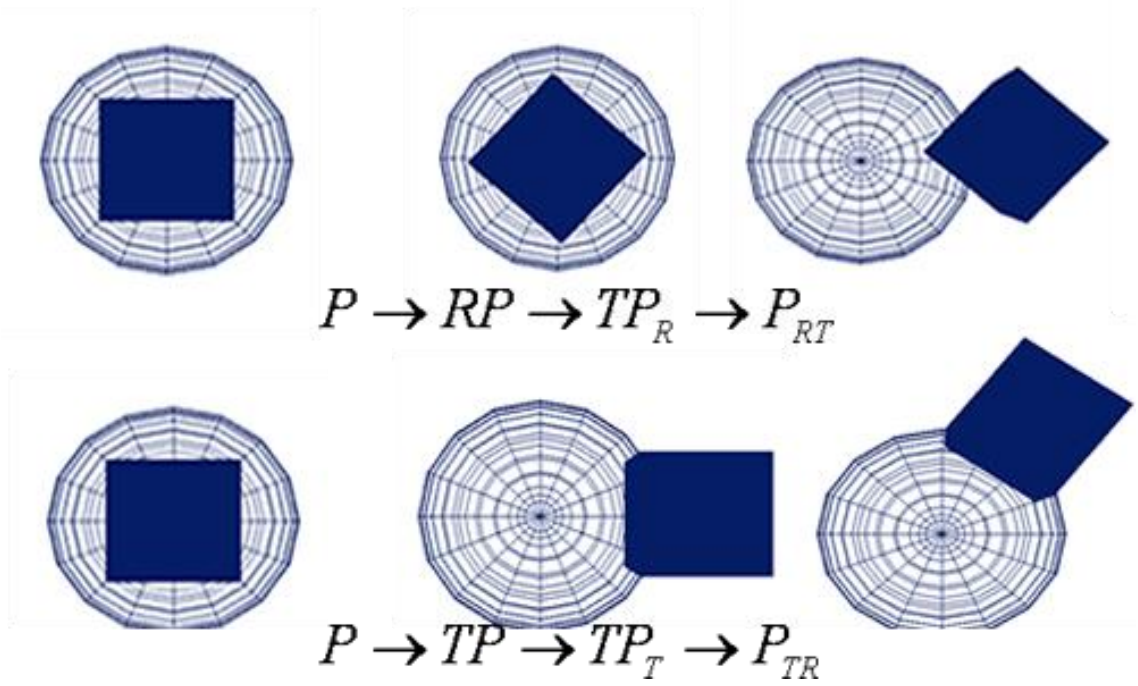


Figura 51: Tranformaciones acumuladas

BIBLIOGRAFÍA

[BENJEMAA1998] Benjemaa Raouf and Schmitt Francis. "A Solution for the Registration of Multiple 3D Point Sets Using Unit Quaternions." *Proc., European Conference on Computer Vision.* 1407:34-50,1998

[BESL1992] P. J. Besl and N. D. McKay. "A method for registration of 3-D shapes." *IEEE Transactions on Patterrn Analysis and Machine Intelligence,* 14(2):239-256, Feb. 1992.

[BORGEFORS86] G. Borgefors, "Distance transformations in digital images", *Computer Vision, Graphics, and Image Processing,* Vol 34, No. 3, (1986), pp. 344–371.

[BULAN2001] G. Bulan, C. Ozturk. "Comparison of two distance based alignment method in medical imaging." *Proceeding of the 23rd Abbual EMBS Internacional Conference.* 2001.

[BUSS03] Buss, Samuel, 3-D Computer Graphics: "A Mathematical Introduction with OpenGL", *Cambridge University Press,* US, 2003, pp. 1-229.

[CAMARA2007] O. Camara, G. Delso, O. Colliot, etal. "Explicit incorporation of prior anatomical information into a non-rigid registration of thoracic and abdominal CT and 18-FDG whole-body emission PET images." *IEEE Transactions on Medical Imaging,* 26(2):164-278, Feb. 2007.

[DEVC2007] Descripción del compilador en http://en.wikipedia.org/wiki/Dev-C, sitio oficial en http://www.bloodshed.net/index.html, fuentes (última versión del compilador) en http://sourceforge.net/projects/dev-cpp/. Actualmente (2010) substituido por http://wxdsgn.sourceforge.net/.

[FARKAS1996] L. G. Farkas, C. K. Deutsh. *Anthropometric determination of craniofacial morphology*. Wiley-Liss, Inc., A Wiley Company, 1996.

[FOLEY1992] J. D. Foley, A. Van Dam, S. K. Feiner, and J. F. Hughes, *Computer Graphics: Principles and Practice*, Addison-Wesley, second edition, Reading, MA, 1990.

[HERMAN1998] G. T. Herman, *Geometry of Digital Spaces*, Boston: Birkhauser, 1998.

[MARQUEZ1999] J. Márquez I. Bloch, F. Schmitt. 1999. « IPCYL : Logiciels de traitement de données cylindriques et images en profondeur, MakeTri et Voxelize: maillage triangulaire et voxelization pour la création de fantômes antropométriques. Déscriptif et documentation technique ». *Département TSI ENST, CNRS URA 820, Dosimétrie et Antennes de ALCATEL-CIT Alsthom Recherche, S.A.* octobre, 1999..

.[MARQUEZ2000] J. Márquez, T. Bousquet, I. Bloch, F. Schmitt and C. Grangeat. "Construction of Human Head Models for Anthropometry and Dosimetry Studies of Hand-Held Phones." *Revista Mexicana de Ingeniería Biomédica.* Vol. XXI, No. 4, diciembre de 2000. pp. 120-128.

[MARQUEZ2005] Jorge Márquez. I. Bloch, T. Bousquet, F. Schmitt and C. Grangeat, "Shape-Based Averaging for Craniofacial Anthropometry", *ENC´2005: Sixth Mexican International Conference on Computer Science, SMCC*, paper 137, memorias en extenso publicadas por la IEEE, 26-30 de septiembre, Puebla, México, 2005. pp. 314 – 319. DOI: 10.1109/ENC.2005.41

[MARQUEZ2006] Márquez Jorge, Patrice Delmas, Isabelle Bloch and Francis Schmitt. "Morphological Averaging of Anatomical Shapes Using Three-Dimensional Distance Transforms", *IVCNZ'06, P. Delmas, J. Morris(ed.), International Vision Computing New Zealand 2006*, Great Barrier Island, NZ, Nov 27-29, 2006, pp. 337-342.

[MARQUEZ2008] Márquez Jorge, Ramirez Waldo, Boyer Loïc and Delmas Patrice. "Robust Ellipsoidal Model Fitting of Human Heads", work 155, *2nd International Workshop "Robot Vision", RobVis'08*. Proceedings Lecture Notes in *Computer Science (LNCS),* Vol. 4931, Springer-Verlag; Auckland, New Zealand, 18-20 febrero, 2008, pp. 381-390.

[ORTIZ2009] J.J. Ortiz, L. González-Santos, Ll. Rodríguez, J Márquez**,** F A., Barrios, "Brain atlas of children six to eight years old". Poster 577 SU-AM presentado en el 15th. *Annual Meeting of the Organization for Human Brain Mapping* (OBHM), San

Francisco, California, USA, 18-23 de junio, 2009. Memorias publicadas en Volume 47, Supplement 1 to *NeuroImage, Elsevier*, 2009.

[RAMIREZ2005] Waldo Ramírez. "Construcción de Modelos Simplificados de la Cabeza Humana". Tesis de Licenciatura en Ingeniería de la Computación, Fac. de Ingeniería, UNAM. Examen presentado y aprobado el 11 de septiembre, 2005.

[SIMONandDAVIGNON2009] SLT (Soulieutenant) Clément Davignon y SLT Richard Simon, "Morphological Averaging of Fluctuating Shapes and Feasibility study of a fast implementation with Field Programming Gate Arrays"("training course report", equivalente a tesis maestría), *CCADET-UNAM y Department of Engineering Sciences, Ecole Militaire de Saint- Cyr, Coëtquidan, Francia, Promotion Chef de Bataillon Segrétain 2006-2009*, Course Director: Doctor Márquez Jorge, Tutor: Mrs. Ababou Rachel. Examen aprobado en Enero 9, 2009.

[SULLI2010] SLT (Soulieutenant) Nicolas Sulli, "Non-linear Distortion of Images for building and Atlas of the Human Brain", maestría internacional, ("training course report", equivalente a tesis maestría), *CCADET-UNAM y Department of Engineering Sciences, Ecole Militaire de Saint-Cyr, Coëtquidan, Francia, Promotion CES Francobille 2008-2010*, Course Director: Doctor MÁRQUEZ Jorge, Tutor: Mr. MOTSCH Jean.

[TOKAZI1996] T. TOKAZI, Y. KATAWATA, N. NIKI. "3D image analysis of the lung area using thin section CT images and its application to diferential diagnosis." *Proc., International Conference on Image Processing.* 2:281-284,1996.

[VRML2005] - "Virtual Reality Modeling Language", VRML 97 (ISO/IEC DIS 14772-1). Especificación del lenguaje en http://www.web3d.org/x3d/specifications/vrml/, descripción en http://www.web3d.org/x3d/vrml/index.html (VRML Archives), y en http://xml.coverpages.org/vrml-X3D.html; artículo con ligas y descripción general en http://en.wikipedia.org/wiki/VRML.