CAPITULO IV

4.1 DISEÑO E IMPLEMENTACIÓN

En este capítulo se presenta el diseño y la implementación del tema propuesto, Modelado de Sistemas de control de un robot manipulador basado en Procesamiento Digital de Imágenes, el cual esta desarrollado en CBuilder bajo la plataforma WindowsXP; está formado por varios módulos que fueron desarrollados y probados independientemente antes de desarrollar una sola aplicación. Lo que se presenta en la imagen siguiente es un presentación del sistema ejecutando todos lo módulos en conjunto.

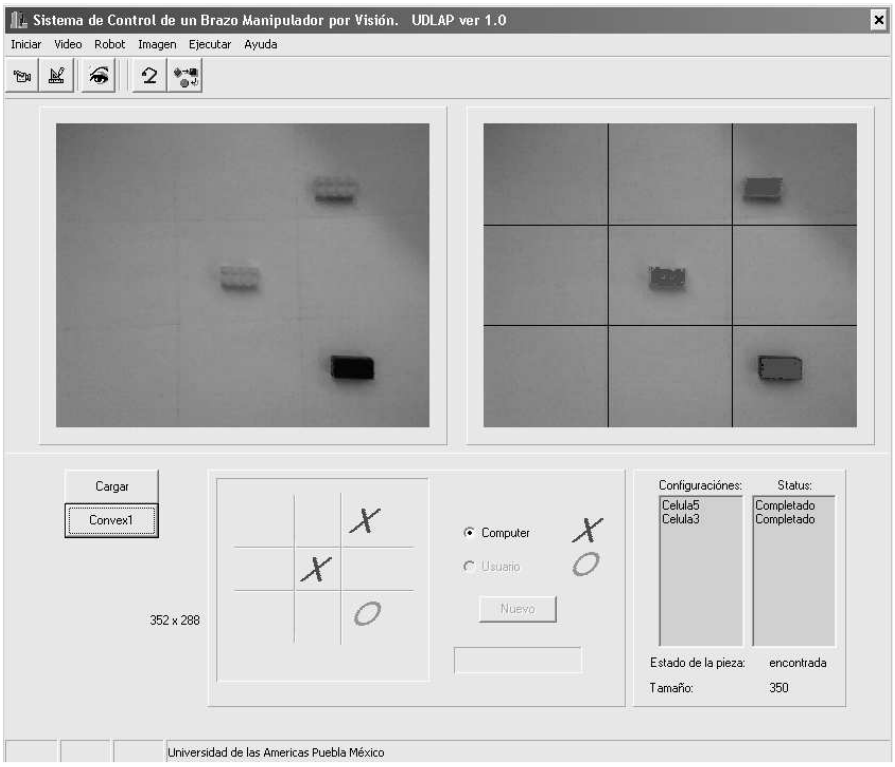

Figura 4.1: Imagen de ejecución del sistema Principal.

Dentro de la aplicación presentada, se encuentran el módulos de procesamiento de imágenes, que se encargan de analizar e interpretar los objetos que se encuentran dentro de las imágenes. El segundo módulo es el juego del Tic Tac Toe, el cual utiliza un algoritmo

Minimax para tomar las decisiones para hacer una mejor jugada por parte del computador, y en tercer lugar se encuentra el módulo de control, que recibe solamente configuraciones, este a su vez se encarga de cumplirlas, analizándolas y ejecutándolas. A continuación se presenta el diseño del proyecto y después su desarrollo.

4.2 DISEÑO DEL PROYECTO

Una vez que se tiene definidos los modelos que se utilizaran en el proyecto de procesamiento de imágenes, se plantea el siguiente diseño de sistema de control a través de procesamiento de imágenes, para esto se utilizan Diagramas de Flujo de Datos, los cuales dan un idea completa del proyecto.

DFD 0, nivel contextual

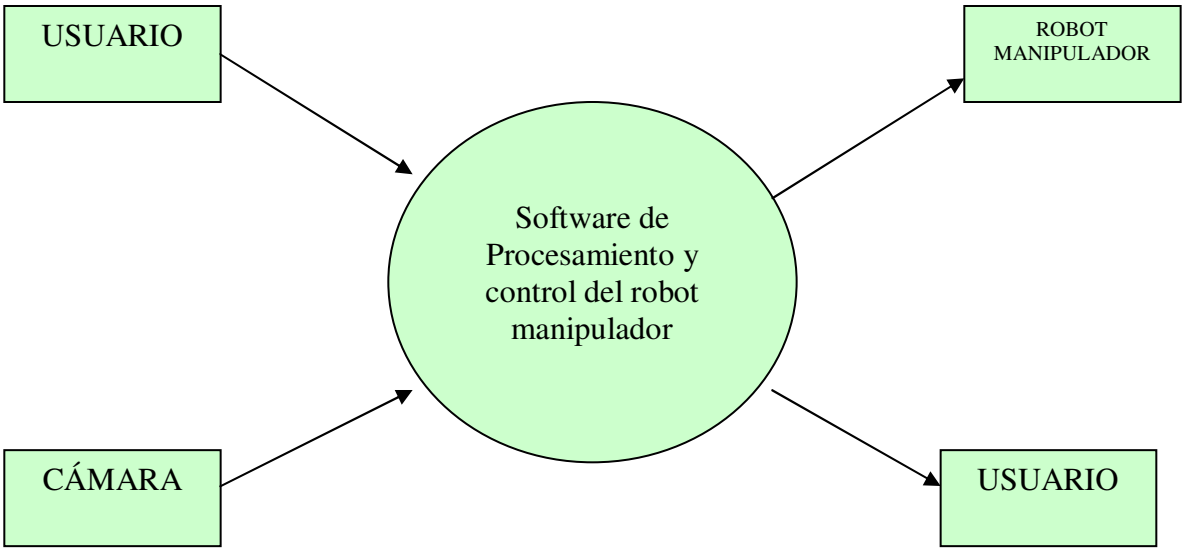

Figura 4.2: DFD 0

DFD nivel 1

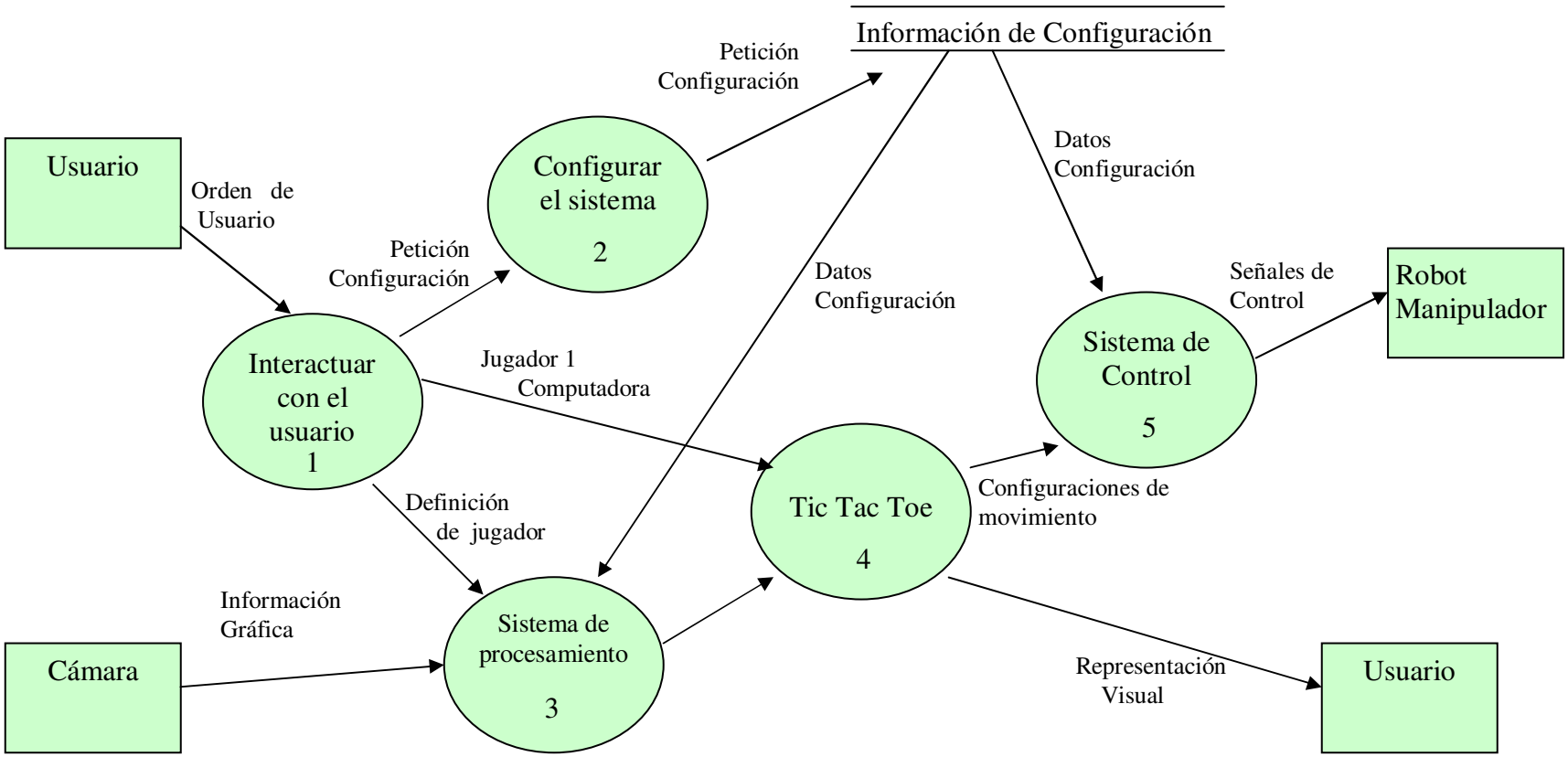

Figura 4.3: DFD 1

DFD 2 Sistema de Procesamiento

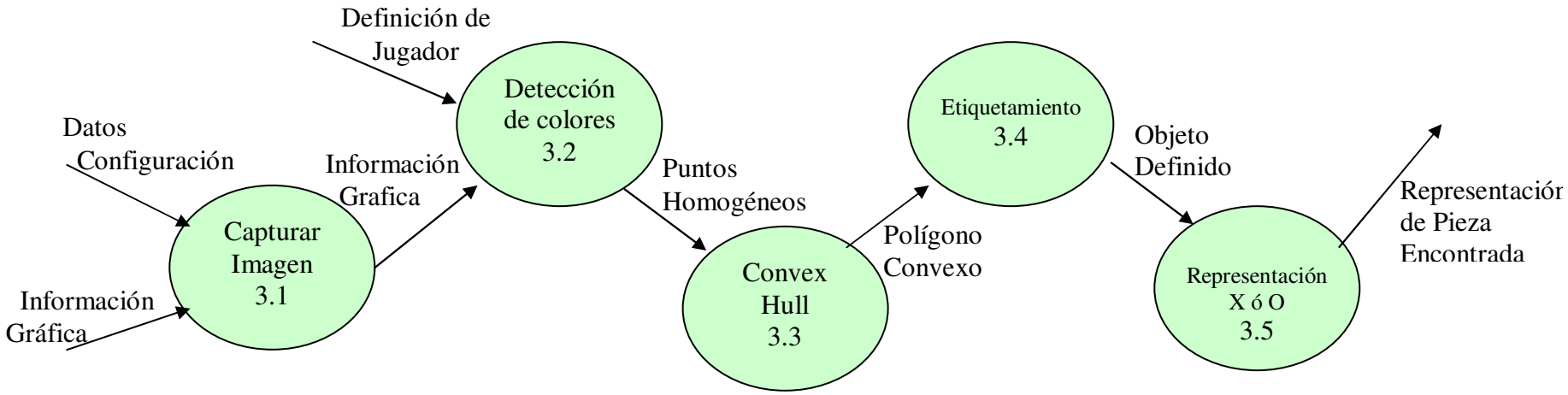

Figura 4.4: DFD 2

DFD 2 Sistema Control

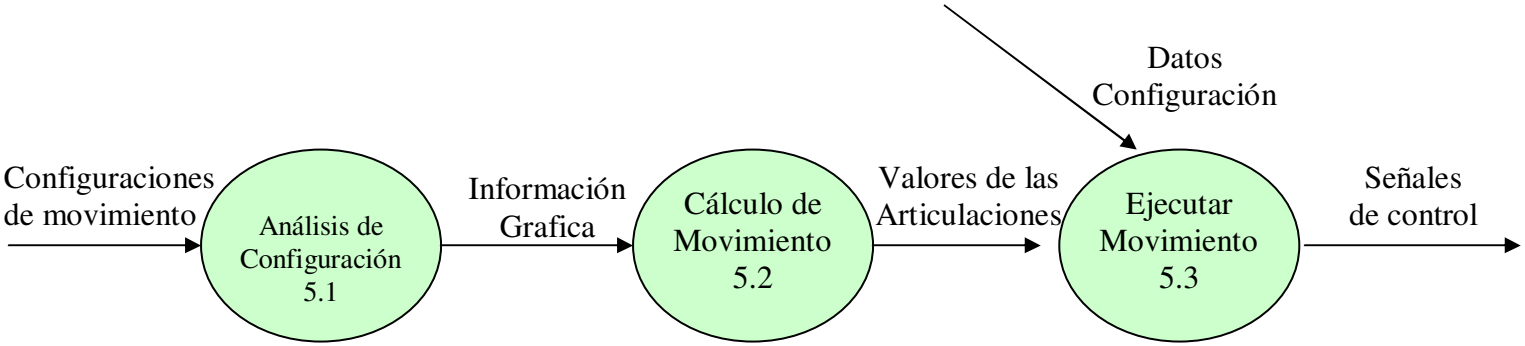

Figura 4.5 DFD 2

En el apéndice B se presenta los diagramas detallados del Diseño orientado a Objeto

4.3 IMPLEMENTACIÓN

SISTEMA DE PROCESAMIENTO DE IMÁGENES

La imagen 5.1 muestra una ejecución del módulo de procesamiento de imágenes. Este módulo lleva una conjunción de pasos para que funcione adecuadamente que se describe a continuación.

Captura de Imagen

Inicialmente se tiene que tomar en cuenta el tipo de dispositivo físico para obtener la información grafica. Para el desarrollo de este proyecto se utilizo una cámara de Lectura por línea, o mejor conocida como WebCam.

Esta cámara puede ser manipulada desde un lenguaje de alto nivel como es C++, a través de su ambiente de programación como es Cbuilder, y para lo cual se instaló una librería tomada desde la pagina Web, de la empresa donde fue hecha la cámara.

Especificaciones:

- Sensor CMOS
- Lente de enfoque manual
- Tamaño de imagen: 352 x 288
- Frecuencia de cuadro: hasta 30 fps (cuadros por segundo)

Las cámaras de lectura por línea funcionan diferentes que las cámaras de lectura por área. En una cámara de lectura por área, una matriz de CCD (generalmente rectangular en forma) en píxeles proporciona una vista de un objeto que contiene longitud y anchura. Con una cámara de lectura por línea, el CCD contiene solamente una sola fila de píxeles. Esta imagen casi unidimensional requiere generalmente que el objeto esté movido y que una serie de cuadros esté tomada para proporcionar los datos útiles para una inspección. Las

cámaras de lectura por línea prevén típicamente la exploración muy rápida de los píxeles para poder asumir el control de muchos cuadros por un período del tiempo corto mientras que el objeto se mueve en el campo visual de la cámara.

CALIBRADO DE LA CÁMARA.

Es una de las partes fundamentales de todo sistema de visión, ya que de este proceso dependen los resultados que se obtengan. Se divide en dos partes: la primera es la orientación de la cámara en el plano. La segunda es la definición de medidas que se utilizarán dentro del plano.

Orientación de la cámara

Aquí es donde se establecen las coordenadas XYZ de la imagen. Inicialmente se coloca la cámara en un lugar fijo, dirigida hacia el área de trabajo establecido, se hace una toma de 1 imagen, se calcula el centro de la imagen, y se le define la distancia en que se encuentra el lente de la cámara, hasta el área de trabajo, que es el eje Z.

Calculo del centro de la imagen

$$C_x = \frac{X_{\max} - X_{\min}}{2} \quad C_y = \frac{Y_{\max} - Y_{\min}}{2} \tag{2.19}$$

Esto orienta la cámara e inicializa las coordenadas XY Z de la imagen,

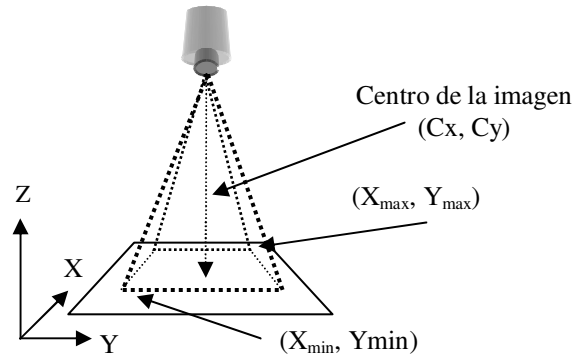

Figura 4.6: Orientación de la cámara a calibrar

Definición de medidas

La definición de la distancia se establecerá en el mismo sistema, haciendo una asignación de equivalencia, entre la imagen y la distancia. Es decir a la imagen tomada, se le define una distancia real en centímetros que es el eje Z.

Una vez definida la distancia con respecto a la imagen, se define un determinado número de píxeles y se hace una equivalencia o escala de distancia en cm.

$$\text{Distancia aproximada} = \frac{d(P,Q)}{Escala}$$

Donde: d (P,Q)= distancia Euclidiana

$$\text{d (P,Q)} = \sqrt{(y_1 - x_1)^2 + (y_2 - x_2)^2} \quad \text{y} \quad Escala = \frac{\text{Eie Z}}{3} \tag{2.20}$$

La escala formada es la siguiente:

1 línea de 20 píxeles = 1 cm. en una imagen con una distancia focal de 45cm

Con esto se puede encontrar el tamaño aproximado del objeto y su posición dentro de área de trabajo. Estos datos serán usados para localizar los objetos en el mundo real, y poder indicar al robot en donde debe poner las piezas exactamente.

Se toma en cuenta que está limitado para calcular mas datos de los objetos, debido a que se esta trabajando con visión no estéreo.

Esta calibración es fundamental, para definir los movimientos del robot Lego MindStorms.

Modificaciones de Propiedades de video.

Las propiedades de la cámara pueden ser manipuladas desde el lenguaje de programación como son la fuente y formato, la cual sirve para un mayor calibrado de la iluminación recibida.

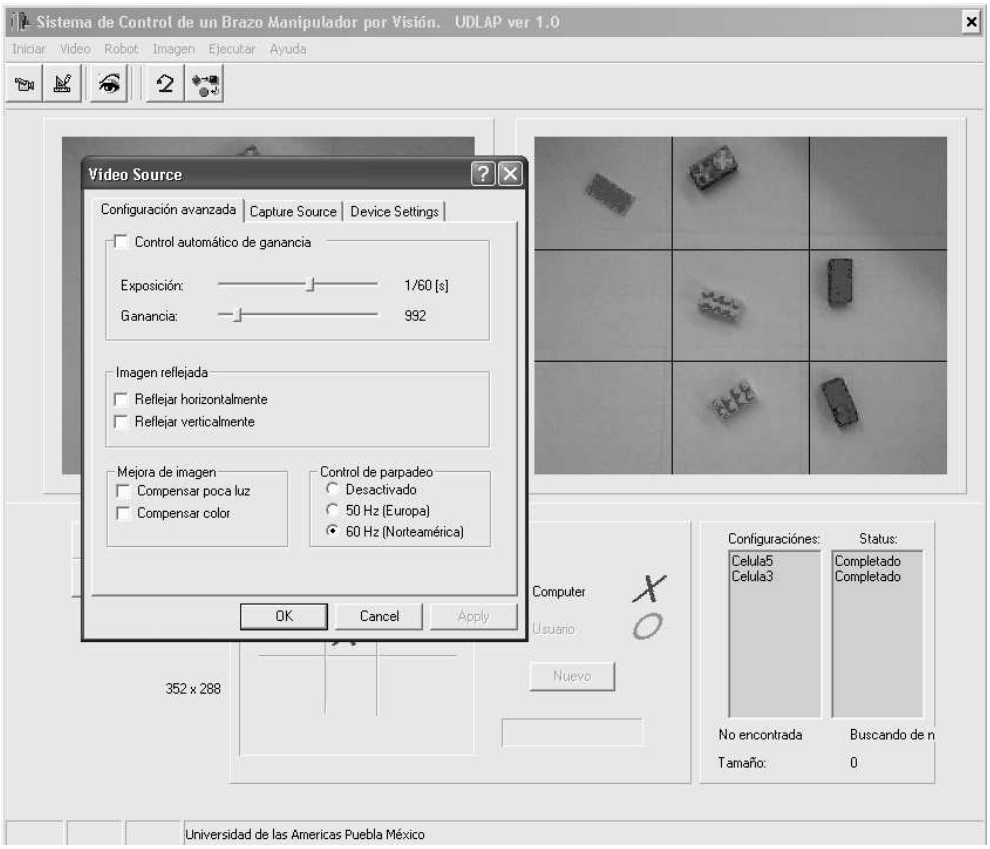

Figura 4.7: Manipulación de características de la cámara

Segmentación

Este método se encarga de dividir en partes o células una imagen, cada célula es independiente una de otra, lo cual permite un mejor análisis por parte del sistemas. Sin embargo, las líneas presentadas en la imagen, son informativas al usuario del estado del sistema.

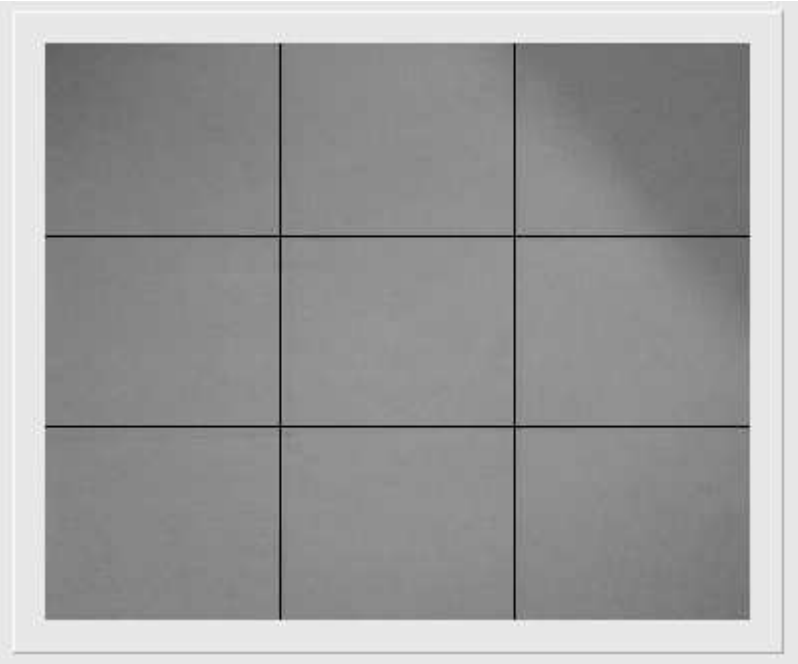

Figura 4.8: Segmentación y creación de las células

La segmentación está definida, tomando el ancho y largo de la imagen. Estos atributos están divididos en 3, y de igual forma están implícitos en el código del programa

Detección de Color

Se define un valor de umbral por objeto a encontrar, debido a que los objetos son afectados por la iluminación de ambiente, y tomando en cuenta que se utiliza un formato RGB, en lugar del HSI.

Inicialmente se hace una lectura de la imagen, de izquierda a derecha y de arriba hacia abajo, conforme se va leyendo la imagen, se obtiene un píxel, el cual se descompone en sus tres principales componentes básicas RGB.

Figura 4.9: Detección del color, mediante un formato RGB.

Este modelo RGB, define en forma sensorial un objeto de un color especifico, el cual al ser detectado por el umbral definido, es convertido a un valor o color RGB uniforme, y de esta forma se define la forma de un objeto. Obteniendo una proximidad de volumen de un objeto. Con el fin de que se pueda manipular esta información para obtener más características en métodos siguientes

Algoritmo de Obtención de Color en RGB y conversión Uniforme.

```
for(Y=0;Y<largo;Y++)  //recorrido de la imagen dentro del picture..
  for(X=0;X<ancho;X++)
    { c=foto1->Canvas->Pixels[X][Y];  //Se obtienen los pixels, y se descompone en sus
      r=c & 255;                      // 3 principales colores..
      g=(c & 56280)>>8;
      b=(c& 16711680)>>16;
      Busca_Color(r,g,b,X,Y );
    }
```

El algoritmo presentado, recorre la imagen de entrada y obtiene el valor del píxel, el cual es transformado en sus principales colores primarios que lo conforman y lo envía a un proceso de conversión a un color homogéneo, siempre y cuando este dentro del valor de umbral definido

Proceso de Conversión a un color homogéneo

```
int TFPrincipal::Busca_Color(int rC,int gC,int  bC,int Xx,int Yy)
{int p=0;
    if ((rC > 115 && rC < 170 && gC > 62 && gC < 132 && bC <20)
          ||    (rC > 205 && rC < 240 &&  gC > 129 && gC < 151 && bC > 32 && bC <42))
     { foto1->Canvas->Pixels[Xx][Yy]=clRed; //Si encuentra el color, actualiza las banderas
       p=1;                                                    // indicando que es la primera
       existe++;                                  //pasada
     }                                          // pero tambien checa la vuelta..
   else
       if((rC == 0  && gC >= 0 && gC < 28 && bC>=29 && bC <135)||
           (rC == 0  && gC ==0 &&  bC >    55     f&&bC <133))
        {foto1->Canvas->Pixels[Xx][Yy]=clGreen;
          p=2;
          existe++;
        }
     return p;                //regresa si fue amarillo o azul o uno diferente..
}
```

Este método recibe cinco parámetros, los primeros tres se refieren al valor de las principales definiciones del color recibido, y las otras dos a la posición donde se encuentren para transformarla. La transformación es en base de un valor de umbral ya definido del objeto y una vez encontrado se convierten a valores homogéneos. En el caso de objetos amarillos, los convierte en colores rojos, y los objetos azules, en verdes.

R > 115 < 170 y >205 <240	G >62 <132 y >205<240	B > 0 <20 y >32 <42	Amarillo
R2=0	G>=0 <28	B >= 29 < 135 y >55 <133	Azules

Tabla 4.1. Valores de umbral de los objetos amarillos y azules.

A continuación se presenta el Diagrama de flujo del método Obtención de Color en RGB

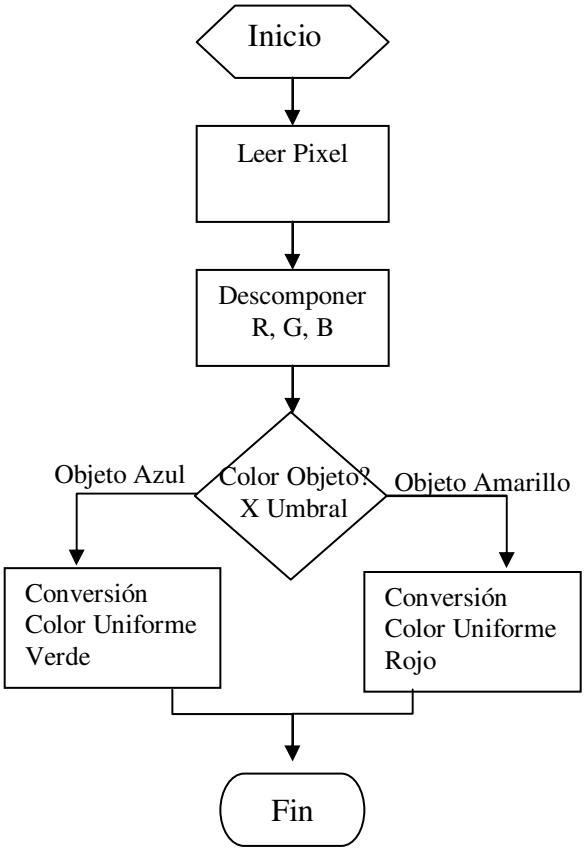

Figura 4.10: Diagrama de Flujo de Obtención de Color en RGB

Convex Hull

Partiendo de que todo sistema de procesamiento de imágenes es diferente, y que existen pocos métodos para detectar bordes de objetos en imágenes a color y en formato RGB, se aplica este algoritmo para definir un polígono convexo, tomando en cuenta que en este proyecto se esta utilizando objetos convexos.

Este método trabaja con los puntos de color uniforme que fueron obtenidos en la detección de colores que se desarrollo anteriormente, estos son los que se utilizan para formar la envolvente convexa y dibuja el área.

Figura 4.11: Aplicación del Método Convex Hull

En la imagen se presenta la forma de la envolvente convexa que define un volumen del objeto encontrado. Tomando en cuenta que existen métodos para definir un borde de piezas, pero en formato a color no dan un resultado contundente, es por eso que se tiene que buscar otro tipo de métodos, para este se definió el Convex Hull

A continuación se presenta el diagrama de flujo del Método Convex Hull así como su Algoritmo.

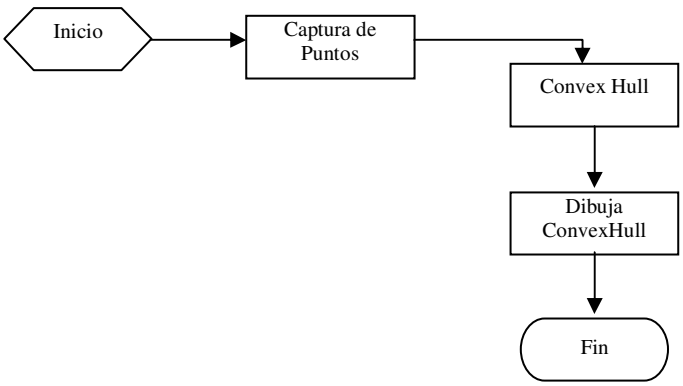

Figura 4.12: Diagrama de Flujo del Método Convex Hull

Algoritmo del Convex Hull

Declaración de estructura:

```
void dibujaConvex();
 typedef struct TipoPunto
  { int x;
    int y;
  }TipoPunto;

TList *Puntos=new TList;
TList *Convex=new TList;
```

Inicialmente se define una estructura para almacenar los puntos que utilizara el método Convex Hull, estos puntos tienen valores de posición X,Y.

```
void __fastcall THerramientas::CargaPtsClick(TObject *Sender)
{  int r2,g2,b2;
   RGBTRIPLE *ptr,*ptr2;
   int ancho,largo;
   ancho =foto2->Width;
   largo=foto2->Height;
   //Puntos=new TList; //llamada a la lista de puntos..
    struct TipoPunto *Pto1;
    Puntos->Clear();
    //Falta liberar la memoria
   for(int j=0;j<largo;j++)
   {  ptr=(RGBTRIPLE *)foto->Picture->Bitmap->ScanLine[j];
     for (int i=0;i < ancho  ;i++)
      {  b2=ptr[i].rgbtBlue;
         g2=ptr[i].rgbtGreen;
         r2=ptr[i].rgbtRed;
         if(r2==255 && g2==0&& b2==0)  //Aqui tengo los puntos..
          {   Pto1 =new TipoPunto;
              Pto1->x=i;
              Pto1->y=j;
              Puntos->Add(Pto1);
           }
       }
    }
   Label13->Caption=IntToStr(Puntos->Count);
   N=Puntos->Count;
}
```

CargarPtos: Este método, se encarga de cargar los puntos válidos, que en este caso son los colores homogéneos, para esto se hace un recorrido por la imagen de salida, la cual fue modificada anteriormente por el método de obtención de color, una vez que detecta el color homogéneo, lo almacena en una estructura de Puntos, el cual tiene la posición XY.

```
void __fastcall THerramientas::BConvexClick(TObject *Sender)
{ struct TipoPunto *Pto1, *Pto2;
  int nn;
  for (int i=0; i<N; i++)
    for(int j=i+1; j<N; j++)
      {Pto1 = (TipoPunto*)Puntos->Items[i];
       Pto2 = (TipoPunto*)Puntos->Items[j];
              if(Checa_Ptos(Pto1->x, Pto1->y, Pto2->x, Pto2->y))
            { Convex->Add(Pto1);
              Convex->Add(Pto2);
            }
       }
  dibujaConvex();
  delete Pto1;
  }
```

BConVex: Es invocado el Método principal del Convex Hull que se encarga de buscar la envolvente convexa del objeto con colores definidos homogéneos. Hace la lectura de dos en dos puntos, donde cada dos puntos los compara para ver si pertenecen a la envolvente convexa, una vez encontrados los agrega a una estructura donde solo van los puntos que están dentro de la envolvente convexa

Una vez encontrada la envolvente convexa, se dibuja está dentro del cuadro de salida y se limpia la estructura que contiene los puntos encontrados, esto se hace para calcular las siguientes envolventes convexas.

```
void __fastcall THerramientas::dibujaConvex()
{   struct TipoPunto *Pto1, *Pto2;
    Pto1 = new TipoPunto;
    Pto2 = new TipoPunto;
    for (int i=0; i<Convex->Count-1; i=i+2)
     { Pto1 = (TipoPunto*)Convex->Items[i];
       Pto2 = (TipoPunto*)Convex->Items[i+1];
       Linea(Pto1->x, Pto1->y, Pto2->x, Pto2->y);  }
   }
```

Dibuja: Se encarga de dibujar dentro de la imagen de salida un polígono convexo. Los puntos a dibujar son obtenidos de la estructura de Convex.

Checa_Ptos: Este método se encarga de comparar cada par de puntos y define en forma afirmativa o falsa su existencia en una envolvente convexa. Para esto se define un conjunto de operaciones que se le aplica a los valores X,Y de los puntos, obteniendo diferencias entre ellos, estas diferencias, son almacenadas en variables globales, con el fin de comparar si el siguiente punto entrante pertenece a la envolvente convexa.

```
bool __fastcall Checa_Ptos(int x0, int y0, int x1, int y1)
{ int A, B, C, Ax0, By0, Pto;
  int band1 = 0, band2 = 0;
  struct TipoPunto *Punto;
  A = y0 - y1;  B = x1 - x0;  C = (x0 * y1) - (x1 * y0);
  Punto = new TipoPunto;
  for(int i=0; i<N;i++)
  {    Punto = (TipoPunto*)Puntos->Items[i];
       Ax0 = A * Punto->x;
       By0 = B * Punto->y;
       Pto = Ax0 + By0 + C;
      if(Pto != 0)
       { if(Pto > 0)
            band1++;
          else
           band2++;
        }
       if( ((band1>0)&&(band2==0)) || ((band1==0)&&(band2>0)) ||  ((band1==0)&&  (band2 = =0)) )
           {  continue;  }
        else
          return(false);
       }
   return(true);
}
```

Es importante señalar que cada par de puntos es comparada con toda la nube de puntos encontrados, es por eso que se puede definir una envolvente convexa

Visión General del Modulo de Procesamiento de Imágenes

La aplicación en conjunto de los métodos de procesamiento de imágenes, dan una interpretación del mundo real, sirviendo para desarrollar una aplicación en forma compleja. A continuación se presenta la forma general del modulo de procesamiento de Imagen en una secuencia de Frames.

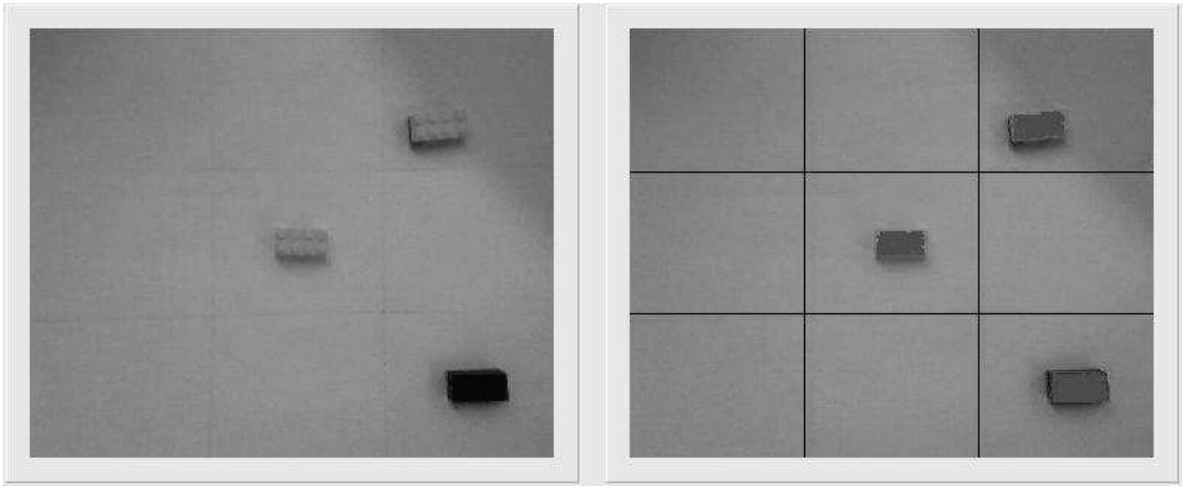

Figura 4.13: Visión general del Juego Tictac toé aplicando Procesamiento de imágenes.

En esta figura se presenta una ejecución del procesamiento de imágenes, con la imagen de entrada a izquierda, y en la imagen derecha se encuentra una imagen de salida, que presenta inicialmente la segmentación de la imagen, en segundo se encuentra definido el volumen aproximado de los objetos representados por sus colores uniformes dentro de la imagen, en tercer lugar cada objeto tiene definido una envolvente convexa, y por ultimo ya se encuentran etiquetados internamente. De esta forma se finaliza el procesamiento de imágenes. A continuación se presenta el diagrama de flujo y su respectivo algoritmo.

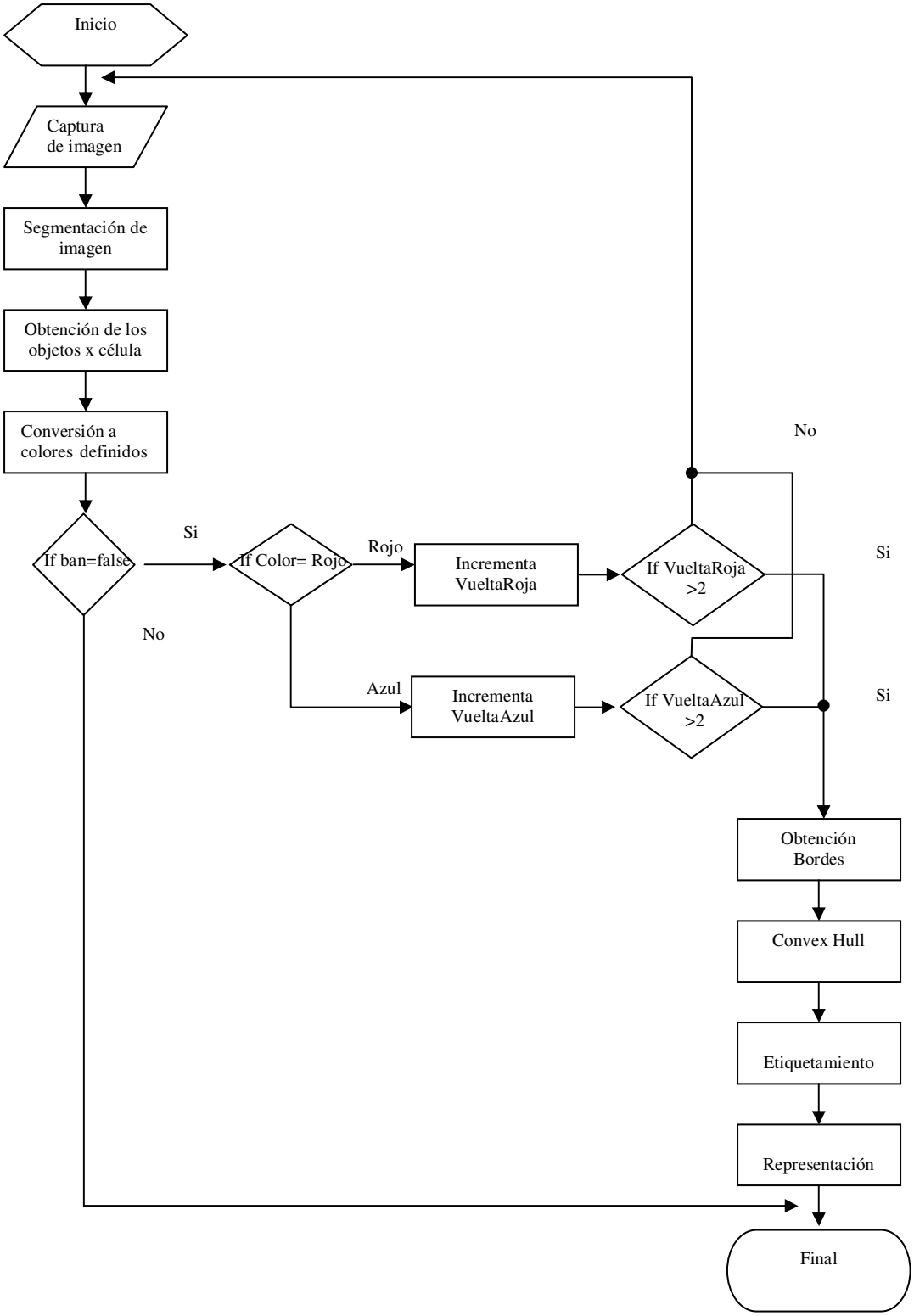

Figura 4.14: Diagrama de Flujo del Módulo de procesamiento de imágenes

Algoritmo General del procesamiento de imágenes

```
void TFPrincipal::EjecutaTodo( )
{int ancho,largo;        // el color de la figura
  int c,r,g,b;          // Tomando en este caso el
  int X,Y, x1,y1,fx, fy;
  int contt=0;
  int BC;
  int contAma=0,contAzu=0;
  existe=0;
  ancho=foto1->Width;  //saca el ancho y largo de la figura
  largo=foto1->Height;//analizar si es necesario sacarlo... se puede disminuir vales
  x1 = ancho;
  y1 = largo;
  fx = 0;
  fy = 0;
    for(Y=0;Y<largo;Y++)  //recorrido de la imagen dentro del Picture.
      for(X=0;X<ancho;X++)
         { c = foto1->Canvas->Pixels[X][Y];  //Se obtienen los píxeles, y se descomponen en sus
           r = c & 255;                                                                    // 3 principales colores..
          g = (c & 56280)>>8;
          b = (c& 16711680)>>16;
           if (Y<96 && X <117)  //rango del primer cuadro..
             {BC=Busca_Color(r,g,b,X,Y);  //busca el color y lo cambia..
               if (vuelta<2)    // solo hace 2 lecturas para definir que se encuentra un objeto en el mismo lugar
                 switch(BC)
                   {case 1: contAma++;
                            break;
                    case 2: contAzu++;
                             break;
                   }
            if((Y==96-1&& X==117-1)&&(contAma>15 ||contAzu>15))
                 {Label3->Caption=IntToStr(contAma);
                   if (contAma>15) //Disminución de piezas falsas.
                     { contAma2=contAma;vuelta++;
                       if(vuelta==2 ) //Pieza amarilla y pieza existente
                            this->Click1();Convex->click();Etiqueta->click;
                      }
                    if(contAzu>15) //Pieza Azul
                      {contAzu2=contAzu; vuelta++;
                       if(vuelta==2)
                          this->Click1();Convex->click();Etiqueta->click ;
             }          }    }
if(Y==96-1&& X==117-1 && contAma2>15 && contAma<15 && vuelta < 2)
           {vuelta=0;  Label2->Caption="Buscando de nuevo";     }
          else
            if((Y==96-1&& X==117-1)&& contAzu2>15 && contAzu<15 && vuelta <2)
              {vuelta=0;Label2->Caption="Buscando de nuevo";}
       } //fin del if del primer rango..
     } //fin de los for..
     dibujaMarco();
      foto1->Refresh();
}
```

Este algoritmo general, es el encargado de analizar cada célula y aplicarle los métodos de procesamiento, inicialmente este algoritmo solo presenta el análisis de una sola célula, sin embargo son nueve células analizadas.

Este método, se ejecuta 3 veces por célula, ya que descarta movimientos de la mano del robot o del usuario al pasar la mano con el objeto, es decir, utiliza una validación para cada pieza, a movimientos del objeto y define la posición de un objeto hasta que esta se encuentra en un lugar fijo, en caso de que se este en movimiento, la descarta.

Una vez definida la posición de la pieza, se procede a aplicar los métodos de procesamiento de imágenes definidos.

Módulo del juego Tic Tac Toe

Esta parte del proyecto, se encarga de ser la interacción visual entre un usuario y la computadora, y es la encargada de generar movimientos que son enviados al robot manipulador. Para esto se presenta la entrada de datos con el usuario, solamente que esta en función de los datos aportados por el módulo de procesamiento de imágenes.

Figura 4.15: Pantalla del Juego Tic Tac Toe

Este módulo recibe inicialmente quien va a empezar un juego de tic tac toe, una vez definido, se procede a ejecutarse, todas la entradas de datos, por parte del usuario, es solamente a través de imágenes de objetos procesados independientes y que corresponden a una determinada jugada.

En la Figura 4.15, se muestra una escena del juego del Tictac toé, no finalizada. Ésta es la interfaz que el usuario ve, siendo la única forma de jugar a través de objetos y procesamiento de imágenes. Las X y O's, tienen todavía su color normal, debido a que no ha finalizados el juego. En la siguiente figura se presenta por que los colores desaparecen.

Figura 4.16 Juego Tictac Toé, donde se encuentra un empate.

En esta figura, se presenta una escena terminada del juego Tictac Toé, donde se generó un empate, tomando en cuenta que el Usuario fue el que inicio el juego. Además, se muestra un relieve sobre las X y O's indicando que el juego ha finalizado y por lo tanto se desactivan los colores.

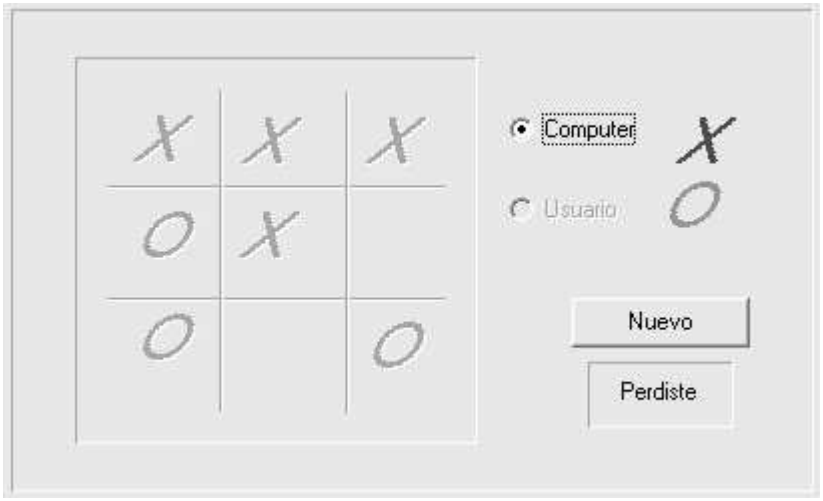

Figura 4.17 Escena del juego Tictac Toé, donde la computadora gana.

Esta escena finalizada, presenta un juego ganado por la computadora, la cual también fue la que inicio el jugo del Tictac Toé; al igual que cuando se crea un empate, los colores de las X y O's se desactivan y se crea una figura con relieve, indicando que el juego ha finalizado. De esta forma se presenta las escenas que se utilizan dentro del Juego del Tictac Toé.

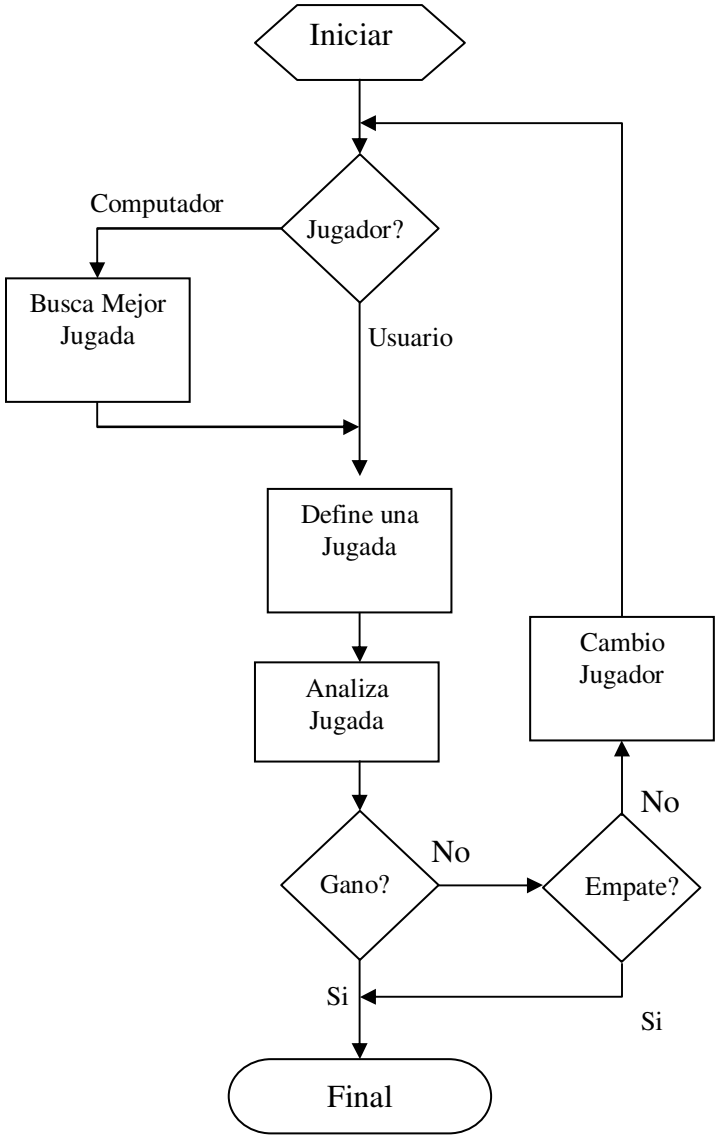

Figura 4.18: Diagrama de Flujo del Módulo del Juego Tic Tac Toe

Declaración de la estructura

Inicialmente se declara una estructura para definir el árbol de Minimax, así como una arreglo bidimensional que guardan las jugadas ganadoras, así también un arreglo que lleva los movimiento que se van desarrollando durante el juego.

```
typedef char Square_Type;
typedef Square_Type Board_Type[Squares];
const Square_Type Empty = ' ';

const int Infinity = 10;
const int Maximum_Moves = Squares;        /* Máximos movimientos en el juego */

int Total_Nodes;                          /* Nodos buscados con  Minimax */

Square_Type Player;
 char Answer[String_Length];
 Board_Type Board;
 int Move_Nbr = 1;

#define Possible_Ganador 8
const int Arbol_en_una_Columna[Possible_Ganador][3] = {
    { 0, 1, 2 },
    { 3, 4, 5 },
    { 6, 7, 8 },
    { 0, 3, 6 },
    { 1, 4, 7 },
    { 2, 5, 8 },
    { 0, 4, 8 },
    { 2, 4, 6 }
};
```

Movimiento principal del Tic Tac Toe

```
void Mover(Board_Type Board, Square_Type Player, int Move_Nbr)
 { int Square;
   int resp;
    if (Player == 'X')
    {           Total_Nodes = 0;
        resp=Best_Move(Board, 'X', &Square, Move_Nbr, -Infinity, Infinity);
        presenta(Square);
                Play(Board, Square, 'X');
    } else
      {         Square--;
                Play(Board, Square, 'O');
      }
    }
```

Mover: Método que se encarga de hacer un movimiento por parte del computador, buscando su mejor movimiento dentro de la arreglo que contiene la jugada actual y la presenta. En caso de que encuentre una jugada por parte del Usuario, solamente recibe donde jugo el usuario.

Mejor Movimiento dentro del Juego Tic Tac Toe

```
int Best_Move(Board_Type Board, Square_Type Player, int *Square, int Move_Nbr,  int Alpha, int Beta)
{    int Best_Square = -1;
     int Moves = 0;
     int I;
     Move_Heuristic_Type Move_Heuristic[Squares];
   Total_Nodes++;
   /*Busca la heuristica para cada movimiento y ordena los movimientos en orden decendientes*/
     for (I = 0; I < Squares; I++) {
                if (Board[I] == Empty) {
                   int Heuristic;
                   int J;
                   Play(Board, I, Player);
                   Heuristic = Evaluate(Board, Player);
                   Play(Board, I, Empty);
                   for (J = Moves-1; J >= 0 &&  Move_Heuristic[J].Heuristic < Heuristic; J--)
                              {Move_Heuristic[J + 1].Heuristic = Move_Heuristic[J].Heuristic;
                               Move_Heuristic[J + 1].Square = Move_Heuristic[J].Square;    }
                    Move_Heuristic[J + 1].Heuristic = Heuristic;
                     Move_Heuristic[J + 1].Square = I;
                     Moves++;
  }              }
     for (I = 0; I < Moves; I++) {
                int Score;
                int Sq = Move_Heuristic[I].Square;
                Square_Type W;
                Play(Board, Sq, Player);        /*marca un movimiento y es capturado por *score*/
                W = Winner(Board);
                if (W == 'X')
                   Score = (Maximum_Moves + 1) - Move_Nbr;
                else if (W == 'O')
                   Score = Move_Nbr - (Maximum_Moves + 1);
                else if (W == 'C')
                   Score = 0;
                else
                   Score = Best_Move(Board, Other(Player), Square, Move_Nbr + 1, Alpha, Beta);
                   Play(Board, Sq, Empty);
                    if (Player == 'X') {      /*Realiza el pruning ( poda ) en alfa-beta*/
                       if (Score >= Beta)
                              {*Square = Sq;
                                return Score;
                            } else if (Score > Alpha)
                                        {Alpha = Score;
                                         Best_Square = Sq;
                                        }
                             } else {  if (Score <= Alpha)
                              {  *Square = Sq;
                                  return Score;
                             } else if (Score < Beta)
                                         {Beta = Score;
                                          Best_Square = Sq;
  }              }                      }

     *Square = Best_Square;

  if (Player == 'X')
              return Alpha;
   else
              return Beta;}
```

Best_Move: Este método se encarga de buscar en su árbol de Minimax, la mejor jugada, buscando minimizando los movimientos.

Modulo del sistema de control del Robot Manipulador

Este módulo se encarga de interpretar y ejecutar las configuraciones puestas por el módulo de juego del Tic Tac Toe, la ejecución hace un envío de señales de control a un robot manipulador, cada configuración es independiente y tiene un tiempo de desarrollo definido.

Figura 4.19: Módulo de Sistema de Control

En esta imagen se presenta una ejecución de 2 configuraciones, cada una representa un conjunto de señales de control, con el fin de que el robot manipulador Lego MindStorms desarrolle una tarea. Una vez que ejecuta su tarea, el robot regresa a una posición inicial y envía una señal al computador para definir el status de su tarea, que en la imagen muestra un estatus correcto.

Conclusión: Se Muestra una forma estructurada para controlar el robot manipulador a través de procesamiento de imágenes, donde se explican las partes fundamentales del proyecto.