******

06 de Junio de 2016

***“INSTITUTO TECNOLOGICO DE VERACRUZ”***

***Asignatura***: Graficación.

***Carrera***: Ingeniería en Sistemas Computacionales

***Grupo***: A ***Semestre***: 6

***Trabajo:*** Manual Técnico.

***Integrantes:*** Hernández Noriega Luis Regino

Lázaro Velasco Raúl

Málaga Matías Luis Roberto

Trujillo García Luis Ángel

***Horario***: 11:00-12:00hrs.

***Catedrático***: Rodríguez León Abelardo

***PERIODO ESCOLAR***: ENERO–JUNIO 2016

**INDICE**

Introducción 4

Conocimientos/Herramientas utilizadas 5

Librerías 6

Variables/Constantes 7

Texturas 9

Metodos 10

* parseCubos() 10
* cargarTextura() 11
* cargarTexturas() 11
* reshape() 12
* dibujarSkybox() 13
* dibujarCubo() 15
* esValido() 17
* crearCamino() 17
* dibujarPiso() 18
* dibujarMapa() 19
* display() 20
* cargaMapa() 21
* mapaNuevo() 22
* init() 23
* moverJugador() 24
* copiarMapa() 26
* modoAutomatico() 26
* keyboard() 27
* main() 29

Manual de Usuario……………………………………………………………… 31

**INTRODUCCION**

Como parte del plan de estudios perteneciente a la materia de Graficación un conjunto de personas desarrollamos el siguiente proyecto. El proyecto presenta y contempla de forma integral cada uno de los módulos desarrollados y vistos durante el curso; lo cual permitió al equipo de trabajo una elaboración favorable.

El proyecto muestra una aplicación basada en el contexto de los videojuegos, la cual se creó desde cero, aplicando los conocimientos adquiridos en clases y de la experiencia en materias precedentes.

La aplicación tiene como propósito visualizar de forma grafica y digital una serie de laberintos que serán recorridos y superados por medio de una esfera como actor principal.

**Conocimientos y herramientas utilizadas**

Para la elaboración de la presente aplicación fueron necesarios los siguientes conocimientos adquiridos durante el curso:

* Manejo de primitivas.
* Manejo de la matriz de transformaciones al igual que las mismas (Traslación, Rotación y Escalación).
* Manejo del entorno 3D.
* Manejo de texturas.
* Manejo de la iluminación.

*Por otro lado fueron necesarios conocimientos extras adquiridos durante el transcurso de nuestra formación como ingenieros en sistemas computacionales.* Entre los destacados se encuentran:

* Manejo de grafos (Búsqueda en anchura – BFS “Breadth First Search”).
* Manejo en la lectura de archivos.
* Manejo de Macros.

Las herramientas utilizadas para el desarrollo de la aplicación son las siguientes:

* El motor gráfico de OpenGL (Geany).
* El programa de edición GIMP.

**Librerías**

Las siguientes librerías fueron utilizadas para poder realizar la aplicación y el funcionamiento de la misma:

* <GL/glut.h>: Esta librería contiene los elementos necesarios para el manejo de OpenGL.
* <math.h>: Esta librería contiene los elementos necesarios en la realización de operaciones y funciones matemáticas.
* <stdio.h>: Esta librería está destinada en los procesos de entrada y salida estándar.
* <vector>: Esta librería permite el manejo y control de los arreglos (vectores y matrices) usados en la aplicación.
* <sstream>: Esta librería permite el manejo y control en operaciones de strings usados por el grafo que permite el movimiento automático del actor principal.
* <string.h>: Esta librería permite el uso de strings dentro de la aplicación.
* <algorithm>: Librería que incluye funciones de ordenamiento y clases como pair que se utiliza en el algoritmo del cálculo del camino.
* <string>: Librería que permite el manejo de los strings de C++.
* <queue>: Librería que permite el uso de la clase queue, la cual es una estructura de datos de tipo FIFO utilizada para el cálculo del camino.

Al igual se usan los siguientes macros en la aplicación:

* MOV\_ARRIBA: Se usa en vez del numero 0.
* MOV\_ABAJO: Se usa en vez del numero 1.
* MOV\_IZQUIERDA: Se usa en vez del numero 2.
* MOV\_DERECHA: Se usa en vez del numero 3.
* MP: Se usa en vez del método make\_pair.
* \_F: Se usa en vez de la constante first
* \_S: Se usa en vez de la constante second

**Variables y Constantes**

Dentro de la aplicación se encuentran una gran cantidad de variables y constantes que se manejan de diversas formas por toda la aplicación y que van cambiando de acuerdo al funcionamiento de la misma cuando comienza a operar. Al igual se hace uso de punteros que permiten el manejo dinámico de los diferentes arreglos utilizados.

|  |
| --- |
| using namespace std; //Para usar todos los espacios de nombres de std  typedef pair<int,int> ii; //Pareja de enteros llamada ii  typedef pair<string,ii> sii; //Pareja de string y de ii (pareja anterior) llamada sii  GLuint \*texturas; //Arreglo dinámico para introducir las texturas  char cadena[128]; //Arreglo de char para leer el nombre del archivo del i-th mapa  int numMapa=1; //Variable entera para el contador del numero de mapa  int Width,Height; //Variables enteras para definir el ancho y alto de la ventana  int skyboxDist; //Variable entera para la distancia entre el centro del laberinto y el skybox  int \*\*mapa2; //Arreglo dinámico para almacenar una copia del mapa del laberinto y se utiliza para encontrar el camino más corto.  int dX[] = {0,0,-1,1}; //  int dZ[] = {-1,1,0,0}; //  bool dir =true; //Variable booleana para establecer en qué sentido va a girar la esfera  bool animacion = false; //Variable booleana que indica si se está realizando la animación del mapa  bool automatico=false; //Variable booleana que indica si esta activado el piloto automático  /\* Constantes para la luz \*/  const GLfloat light\_ambient[] = { 0.0f, 0.0f, 0.0f, 1.0f };  const GLfloat light\_diffuse[] = { 1.0f, 1.0f, 1.0f, 1.0f };  const GLfloat light\_specular[] = { 1.0f, 1.0f, 1.0f, 1.0f };  const GLfloat light\_spot\_dir[] = { 0.0f, 0.0f, 0.0f};  const GLfloat light\_attenuation[] = {1.0f};  GLfloat light\_position[] = { 2.0f, 5.0f, 5.0f, 0.0f }; //Arreglo que contiene las posiciones de la luz  const GLfloat maxCamY = 200.0f, minCamY = 60.0f; //Constantes para definir el máximo de altura y el mínimo de altura en el eje Y  GLfloat camX,camY,camZ; //Variables para la posición de la cámara  GLfloat deltay = 4; //Constante para el movimiento de la cámara  string camino; //Cadena que guarda el conjunto de direcciones (arriba, abajo, izquierda, derecha) del camino óptimo  float initX, initZ; //Coordenadas de la esquina superior izquierda del mapa en el espacio  int posX, posZ; //Posición actual del jugador en la matriz  int xActual,zActual; //Coordenada actual del jugador en el espacio  float rotx=0,roty=0; //Variables flotantes para la rotación de la esfera  int tamCubo=20; //Constante entera para el tamaño de cada cubo en el mapa  int tamSphere=tamCubo/2; //Constante para el tamaño de la esfera en el mapa  int altoMapa=0,largoMapa=0; //Variables que van a contener el alto y largo de cada mapa a dibujar  int \*\*mapa; //Arreglo dinámico para almacenar el mapa a dibujar  int \*\*\*cubos; //Arreglo dinámico para almacenar las coordenadas de los cubos a dibujar en el mapa |

**Texturas**

Para la aplicación se usaron 9 diferentes texturas. Las primeras tres son para el escenario principal; Los muros del laberinto, la esfera de actor principal y el piso dentro del laberinto. Las 6 restantes son para el skybox que se encarga de cubrir el laberinto, cada una enfocada a una cara (arriba, abajo, enfrente, atrás, izquierda y derecha) dentro del cubo.

|  |
| --- |
| struct textura\_512 {  unsigned int width;  unsigned int height;  unsigned int bytes\_per\_pixel; /\* 2:RGB16, 3:RGB, 4:RGBA \*/  unsigned char pixel\_data[512 \* 512 \* 3 + 1];  }; |

|  |
| --- |
| struct textura\_512 textura\_muro = {…}  struct textura\_512 textura\_piso = {…}  struct textura\_512 textura\_lava = {…}  struct textura\_512 skybox\_up = {…}  struct textura\_512 skybox\_front = {…}  struct textura\_512 skybox\_bottom = {…}  struct textura\_512 skybox\_right = {…}  struct textura\_512 skybox\_left = {…}  struct textura\_512 skybox\_back = {…} |

*Nota: Cada textura contiene código entendible por el lenguaje C para representar la imagen determinada, pero al ser muy extenso no se incluye en este apartado.*

**Métodos**

parseCubos()

Este método se encarga de cargar dentro del arreglo cubos las coordenadas donde se encontraran cada cubo que formara el muro del laberinto. Se utilizan las variables initX y initZ para encontrar la esquina superior y partir de allí, por medio de un ciclo for se recorre el mapa y se va guardando en el arreglo las coordenadas. Se utiliza una condición if para verificar si el código del mapa corresponde a un “2” (Significa que dibujara la esfera/actor principal en esa posición) y guarda los valores tanto para dibujar posteriormente la esfera en el espacio como para saber su posición dentro del arreglo.

Al igual se calculan los valores para definir la distancia de la cámara y el foco de iluminación dependiendo del tamaño del mapa; también se define la distancia que tendrá el skybox respecto al mapa.

|  |
| --- |
| void parseCubos(){  initX = -(tamCubo\*largoMapa)/2.0;  initZ = -(tamCubo\*altoMapa)/2.0;  cubos = new int\*\*[altoMapa];  for(int i=0;i<altoMapa;i++){  cubos[i] = new int\*[largoMapa];  for(int j=0;j<largoMapa;j++){  cubos[i][j] = new int[2];  cubos[i][j][0] = initX+(tamCubo\*j);  cubos[i][j][1] = initZ+(tamCubo\*i);  if(mapa[i][j]==2){  posZ=i;  posX=j;  xActual = initX+(tamCubo\*j);  zActual = initZ+(tamCubo\*i);  }  }  }  double w = (largoMapa/2.0);  double h = (altoMapa/2.0);  int hipotenusa = ceil(sqrt(w\*w + h\*h));  skyboxDist = (hipotenusa+3) \* tamCubo;  hipotenusa = (hipotenusa+2) \* tamCubo;  camX = light\_position[0] = 0.0f;  camY = light\_position[1] = minCamY;  camZ = light\_position[2] = hipotenusa;  } |

cargarTextura()

Este método se encarga de establecer las configuraciones predeterminadas para cargar las texturas y que se logren visualizar en el ambiente grafico, igual verifica si se va a texturizar el skybox o no.

|  |
| --- |
| void cargarTextura(GLuint textura, struct textura\_512 textuN,bool sky){  glBindTexture(GL\_TEXTURE\_2D,textura);  glTexEnvf( GL\_TEXTURE\_ENV, GL\_TEXTURE\_ENV\_MODE, GL\_MODULATE);  if(sky){  glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP);  glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_CLAMP);  }  gluBuild2DMipmaps(GL\_TEXTURE\_2D, textuN.bytes\_per\_pixel, textuN.width, textuN.height, GL\_RGB,GL\_UNSIGNED\_BYTE, textuN.pixel\_data );  } |

cargarTexturas()

Este método se encarga de cargar todas las texturas que se usaran en el ambiente grafico y las activa para su futura visualización

|  |
| --- |
| void cargarTexturas(){  texturas = new GLuint[9];  glGenTextures(9,texturas);  cargarTextura(texturas[0],textura\_muro,false);  cargarTextura(texturas[1],textura\_lava,false);  cargarTextura(texturas[2],textura\_piso,false);  cargarTextura(texturas[3],skybox\_up,true);  cargarTextura(texturas[4],skybox\_front,true);  cargarTextura(texturas[5],skybox\_left,true);  cargarTextura(texturas[6],skybox\_right,true);  cargarTextura(texturas[7],skybox\_bottom,true);  cargarTextura(texturas[8],skybox\_back,true);  glEnable(GL\_TEXTURE\_2D);  } |

reshape()

Este método es esencial en el manejo de OpenGl, en esta aplicación también se usa para definir la profundidad que tendrá la perspectiva de visualización dependiendo del tamaño de mapa (Laberinto) a visualizar.

|  |
| --- |
| void reshape(int width, int height) {  glViewport(0, 0, width, height);  glMatrixMode(GL\_PROJECTION);  glLoadIdentity();  Height = height; Width = width;  int maxi = (sqrt(largoMapa\*largoMapa + altoMapa\*altoMapa)+10) \* tamCubo;  gluPerspective(90.0,width/height,1,maxi\*2);  glMatrixMode(GL\_MODELVIEW);  glLoadIdentity();  } |

dibujarSkybox()

Este método se encarga de dibujar el skybox que contendrá al laberinto. Se encarga de crear cada cara del cubo a formar y carga su textura correspondiente. Al igual se desactivan ciertas funciones como la profundidad y la luz para su creación, una vez hecho se vuelven activar.

|  |
| --- |
| void dibujarSkybox(){  glEnable(GL\_TEXTURE\_2D);  glDisable(GL\_DEPTH\_TEST);  glDisable(GL\_LIGHTING);  glDisable(GL\_BLEND);  glColor4f(1.0,1.0,1.0,1.0);  //Cara de Enfrente  glBindTexture(GL\_TEXTURE\_2D,texturas[4]);  glBegin(GL\_QUADS);  glNormal3f(0.0f, 0.0f, 1.0f);  glTexCoord2d(0.0,0.0); glVertex3f(-skyboxDist,skyboxDist,-skyboxDist);  glTexCoord2d(0.0,1.0); glVertex3f(-skyboxDist,-skyboxDist,-skyboxDist);  glTexCoord2d(1.0,1.0); glVertex3f(skyboxDist,-skyboxDist,-skyboxDist);  glTexCoord2d(1.0,0.0); glVertex3f(skyboxDist,skyboxDist,-skyboxDist);  glEnd();  //Cara Izquierda  glBindTexture(GL\_TEXTURE\_2D,texturas[5]);  glBegin(GL\_QUADS);  glNormal3f(1.0f, 0.0f, 0.0f);  glTexCoord2d(0.0,0.0); glVertex3f(-skyboxDist,skyboxDist,skyboxDist);  glTexCoord2d(0.0,1.0); glVertex3f(-skyboxDist,-skyboxDist,skyboxDist);  glTexCoord2d(1.0,1.0); glVertex3f(-skyboxDist,-skyboxDist,-skyboxDist);  glTexCoord2d(1.0,0.0); glVertex3f(-skyboxDist,skyboxDist,-skyboxDist);  glEnd();  //Cara de Atrás  glBindTexture(GL\_TEXTURE\_2D,texturas[8]);  glBegin(GL\_QUADS);  glNormal3f(0.0f, 0.0f, -1.0f);  glTexCoord2d(0.0,0.0); glVertex3f(skyboxDist,skyboxDist,skyboxDist);  glTexCoord2d(0.0,1.0); glVertex3f(skyboxDist,-skyboxDist,skyboxDist);  glTexCoord2d(1.0,1.0); glVertex3f(-skyboxDist,-skyboxDist,skyboxDist);  glTexCoord2d(1.0,0.0); glVertex3f(-skyboxDist,skyboxDist,skyboxDist);  glEnd();  //Cara Derecha  glBindTexture(GL\_TEXTURE\_2D,texturas[6]);  glBegin(GL\_QUADS);  glNormal3f(-1.0f, 0.0f, 0.0f);  glTexCoord2d(0.0,0.0); glVertex3f(skyboxDist,skyboxDist,-skyboxDist);  glTexCoord2d(0.0,1.0); glVertex3f(skyboxDist,-skyboxDist,-skyboxDist);  glTexCoord2d(1.0,1.0); glVertex3f(skyboxDist,-skyboxDist,skyboxDist);  glTexCoord2d(1.0,0.0); glVertex3f(skyboxDist,skyboxDist,skyboxDist);  glEnd();  //Cara Arriba  glBindTexture(GL\_TEXTURE\_2D,texturas[3]);  glBegin(GL\_QUADS);  glNormal3f(0.0f, -1.0f, 0.0f);  glTexCoord2d(0.0,0.0); glVertex3f(-skyboxDist-2,skyboxDist-2,skyboxDist+2);  glTexCoord2d(0.0,1.0); glVertex3f(-skyboxDist-2,skyboxDist-2,-skyboxDist-2);  glTexCoord2d(1.0,1.0); glVertex3f(skyboxDist+2,skyboxDist-2,-skyboxDist-2);  glTexCoord2d(1.0,0.0); glVertex3f(skyboxDist+2,skyboxDist-2,skyboxDist+2);  glEnd();  //Cara Abajo  glBindTexture(GL\_TEXTURE\_2D,texturas[7]);  glBegin(GL\_QUADS);  glNormal3f(0.0f, 1.0f, 0.0f);  glTexCoord2d(0.0,0.0); glVertex3f(-skyboxDist,-skyboxDist+2,-skyboxDist);  glTexCoord2d(0.0,1.0); glVertex3f(-skyboxDist,-skyboxDist+2,skyboxDist);  glTexCoord2d(1.0,1.0); glVertex3f(skyboxDist,-skyboxDist+2,skyboxDist);  glTexCoord2d(1.0,0.0); glVertex3f(skyboxDist,-skyboxDist+2,-skyboxDist);  glEnd();  glEnable(GL\_DEPTH\_TEST);  glEnable(GL\_LIGHTING);  glEnable(GL\_BLEND);  } |

dibujarCubo()

Este método se encarga de dibujar un cubo el cual será parte de los muros del laberinto. Recibe las coordenadas y dependiendo de la posición dibuja en ese punto en el espacio grafico dicha figura, al igual se le aplica su textura correspondiente.

|  |
| --- |
| void dibujarCubo(int X, int Z){  glBindTexture(GL\_TEXTURE\_2D,texturas[0]);  glColor4f(1.0,1.0,1.0,1.0);  glBegin(GL\_QUADS);  //Cara de enfrente  glNormal3f(0.0f, 0.0f, 1.0f);  glTexCoord2d(0.0,0.0); glVertex3f(X-tamSphere,tamSphere,Z+tamSphere);  glTexCoord2d(0.0,1.0); glVertex3f(X-tamSphere,-tamSphere,Z+tamSphere);  glTexCoord2d(1.0,1.0); glVertex3f(X+tamSphere,-tamSphere,Z+tamSphere);  glTexCoord2d(1.0,0.0); glVertex3f(X+tamSphere,tamSphere,Z+tamSphere);  //Cara izquierda  glNormal3f(-1.0f, 0.0f, 0.0f);  glTexCoord2d(0.0,0.0); glVertex3f(X-tamSphere,tamSphere,Z-tamSphere);  glTexCoord2d(0.0,1.0); glVertex3f(X-tamSphere,-tamSphere,Z-tamSphere);  glTexCoord2d(1.0,1.0); glVertex3f(X-tamSphere,-tamSphere,Z+tamSphere);  glTexCoord2d(1.0,0.0); glVertex3f(X-tamSphere,tamSphere,Z+tamSphere);  //Cara de atrás  glNormal3f(0.0f, 0.0f, -1.0f);  glTexCoord2d(0.0,0.0); glVertex3f(X+tamSphere,tamSphere,Z-tamSphere);  glTexCoord2d(0.0,1.0); glVertex3f(X+tamSphere,-tamSphere,Z-tamSphere);  glTexCoord2d(1.0,1.0); glVertex3f(X-tamSphere,-tamSphere,Z-tamSphere);  glTexCoord2d(1.0,0.0); glVertex3f(X-tamSphere,tamSphere,Z-tamSphere);  //Cara derecha  glNormal3f(1.0f, 0.0f, 0.0f);  glTexCoord2d(0.0,0.0); glVertex3f(X+tamSphere,tamSphere,Z+tamSphere);  glTexCoord2d(0.0,1.0); glVertex3f(X+tamSphere,-tamSphere,Z+tamSphere);  glTexCoord2d(1.0,1.0); glVertex3f(X+tamSphere,-tamSphere,Z-tamSphere);  glTexCoord2d(1.0,0.0); glVertex3f(X+tamSphere,tamSphere,Z-tamSphere);  //Cara Arriba  glNormal3f(0.0f, 1.0f, 0.0f);  glTexCoord2d(0.0,0.0); glVertex3f(X-tamSphere,tamSphere,Z-tamSphere);  glTexCoord2d(0.0,1.0); glVertex3f(X-tamSphere,tamSphere,Z+tamSphere);  glTexCoord2d(1.0,1.0); glVertex3f(X+tamSphere,tamSphere,Z+tamSphere);  glTexCoord2d(1.0,0.0); glVertex3f(X+tamSphere,tamSphere,Z-tamSphere);  glEnd();  } |

esValido()

Este método se encarga de verificar si la posición en la matriz del actor principal (esfera) está dentro de los rangos (dimensiones del arreglo), si es así regresa una respuesta true en caso contrario false.

|  |
| --- |
| bool esValido(int X, int Z) {  return X>=0 && X<largoMapa && Z>=0 && Z<altoMapa;  } |

crearCamino()

Este método retornará una cadena de caracteres donde cada carácter significa un movimiento. Utiliza el algoritmo conocido como Búsqueda por Anchura (BFS) para encontrar el camino más corto dentro de un grafo no ponderado. En este hace uso de una queue y de los objetos de tipo pair.

|  |
| --- |
| string crearCamino(){  queue<sii> q; q.push(MP(string(""),MP(posX,posZ)));  while(!q.empty()){  sii f = q.front(); q.pop();  string path = f.\_F;  int x = f.\_S.\_F, z = f.\_S.\_S;  for(int i=0;i<4;i++){  int a = x+dX[i], b = z+dZ[i];  stringstream sstm;  string r;  sstm << path << i;  r= sstm.str();  if(esValido(a,b)){  if(mapa2[b][a]==3){  return r;  }else if(mapa2[b][a]==0){  mapa2[b][a] = 1;  q.push(MP(r,MP(a,b)));  }  }  }  }  return string("");  } |

dibujarPiso()

Este método se encarga de dibujar un piso que será parte del laberinto. Crea un cuadrado en las coordenadas especificadas y le aplica su textura correspondiente.

|  |
| --- |
| void dibujarPiso(int X, int Y, int Z){  glBindTexture(GL\_TEXTURE\_2D, texturas[2]);  glColor4f(1,1,1,1);  glBegin(GL\_QUADS);  glNormal3f(0.0f, 1.0f, 0.0f); //Hacia arriba  glTexCoord2d(0.0,0.0); glVertex3f(X-tamSphere,Y,Z-tamSphere);  glTexCoord2d(0.0,1.0); glVertex3f(X-tamSphere,Y,Z+tamSphere);  glTexCoord2d(1.0,1.0); glVertex3f(X+tamSphere,Y,Z+tamSphere);  glTexCoord2d(1.0,0.0); glVertex3f(X+tamSphere,Y,Z-tamSphere);  glEnd();  } |

dibujarMapa()

Este método se encarga de recorrer por medio de ciclos for anidados el arreglo que tiene almacenado el mapa para formar el laberinto. Dentro de esta función se encuentra un condicional switch que verificara que elemento encuentra en la matriz, los valores especificados dentro del arreglo son los que ayudan a formar el mapa, 0 para los espacios vacios, 1 para los muros, 2 para el inicio de la esfera (actor principal) y 3 para especificar el final del juego (cubo de victoria). Para los casos 0 y 2 va a proceder a llamar el método dibujarPiso(), para el caso 1 llama al método dibujarCubo() y para el caso 3 crea un cubo de color azul y por ayuda de una transformación lo traslada a la coordenada especificada.

Una vez que finaliza el condicional switch se procede a configurar el actor principal. Se crea una esfera usando los elementos necesarios para su configuración y se le aplican transformaciones (rotación y traslación) para su movimiento dentro del juego. Para realizar la rotación se usa una condición (variable dir) para saber el tipo de movimiento que se llevara a cabo (arriba/abajo o izquierda/derecha) y posteriormente se traslada a la coordenada actual donde se encontrara la esfera.

|  |
| --- |
| void dibujarMapa(){  for(int i=0;i<altoMapa;i++){  for(int j=0;j<largoMapa;j++){  switch(mapa[i][j]){  case 0:  case 2:  dibujarPiso(cubos[i][j][0],-tamSphere,cubos[i][j][1]);  break;  case 1:  dibujarCubo(cubos[i][j][0],cubos[i][j][1]);  break;  case 3:  glPushMatrix();  glColor3f(0,0,1);  glBindTexture(GL\_TEXTURE\_2D,0);  glTranslatef(cubos[i][j][0],0,cubos[i][j][1]);  glutSolidCube(tamCubo);  glPopMatrix();  break;  }  }  }  GLUquadric \*qobj = gluNewQuadric();  glPushMatrix();  gluQuadricTexture(qobj,GL\_TRUE);  glBindTexture(GL\_TEXTURE\_2D,texturas[1]);  glColor3f(1,1,1);  glTranslatef(xActual,0,zActual);  if(dir){  glRotatef(roty,0,0,1);  glRotatef(rotx,1,0,0);}  else{  glRotatef(rotx,1,0,0);  glRotatef(roty,0,0,1);  }  //glutWireSphere(tamSphere,tamSphere,tamSphere);  gluSphere(qobj,tamSphere,tamSphere,tamSphere);  gluDeleteQuadric(qobj);  glPopMatrix();  glBindTexture(GL\_TEXTURE\_2D,0);  } |

display()

El método display es necesario en aplicaciones OpenGL. Para el uso de esta aplicación aquí se especifica el aspecto de visualización de la cámara y la función para su movimiento y la función para el movimiento de la iluminación. También se llaman las funciones dibujarSkybox() y dibujarMapa() para el repintado de las mismas en el ambiente grafico.

|  |
| --- |
| void display() {  glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);  gluLookAt(camX,camY,camZ,0,0,0,camX,camY+1,camZ);  glLightfv(GL\_LIGHT0, GL\_POSITION, light\_position);  dibujarSkybox();  if(animacion){  glTranslatef(0,yMas,0);  }  dibujarMapa();  glLoadIdentity();  glutSwapBuffers();  } |

cargarMapa()

Este método se encargara de abrir el archivo .map que contendrá el mapa a dibujar. Usando una función snprintf concatenamos el nombre del archivo (mapa+numeroN) para abrir, posteriormente se abre usando la variable \*pFile y se verifica que no esté vacio. Luego se recuperan los valores de alto y largo del mapa para establecer las dimensiones del arreglo dinámico mapa y posteriormente con ciclos for anidados se van ingresando los valores del archivo al arreglo. Por último se cierra el archivo y se llama la función reshape() para redimensionar el campo de visualización.

|  |
| --- |
| void cargarMapa(int n){  snprintf(cadena, sizeof(cadena), "mapa%d.map", n);  FILE \*pFile = fopen(cadena,"r");  if(pFile==NULL){  exit(-1);  }  fscanf(pFile,"%d %d",&altoMapa,&largoMapa);  mapa = new int\*[altoMapa];  for(int i=0;i<altoMapa;i++){  mapa[i] = new int[largoMapa];  for(int j=0;j<largoMapa;j++){  fscanf(pFile,"%d",&mapa[i][j]);  }  }  fclose(pFile);  reshape(Width,Height);  } |

mapaNuevo()

Realiza la animación que mueve el mapa hacia arriba, carga el siguiente mapa que encuentre y este lo renderiza debajo para moverlo de igual forma hacia arriba.

|  |
| --- |
| void mapaNuevo(){  animacion = true;  for (yMas = 0.0; yMas < 500.0; yMas+=1.0)  {  display();  }  numMapa++;  cargarMapa(numMapa);  parseCubos();  for (yMas = -100.0; yMas <= 0; yMas+=1.0)  {  display();  }  animacion = false;  } |

init()

Este método es muy usado en aplicaciones de OpenGL. En esta aplicación se usa para cargar desde un inicio todas las texturas a utilizar (cargarTexturas()) y de esta forma conservar memoria y no hacer tan robusta la aplicación, se carga el primer mapa cargarMapa(1) recibiendo como parámetro que será el archivo map+numeroN, donde numeroN será 1. Al igual se crean las coordenadas de donde estarán dibujados los cubos/muros del laberinto en el espacio grafico, parseCubos() y se activa la profundidad. Se inicializan valores de texturas e iluminación.

|  |
| --- |
| void init() {  glClearColor(0,0,0,0);  cargarTexturas();  cargarMapa(1);  parseCubos();  glEnable(GL\_DEPTH\_TEST);  glEnable(GL\_LIGHT0);  glEnable(GL\_NORMALIZE);  glEnable(GL\_COLOR\_MATERIAL);  glEnable(GL\_CULL\_FACE);  glEnable(GL\_LIGHTING);  glLightfv(GL\_LIGHT0, GL\_SPOT\_DIRECTION,light\_spot\_dir);  glLightfv(GL\_LIGHT0, GL\_AMBIENT, light\_ambient);  glLightfv(GL\_LIGHT0, GL\_DIFFUSE, light\_diffuse);  glLightfv(GL\_LIGHT0, GL\_LINEAR\_ATTENUATION, light\_attenuation);  glLightfv(GL\_LIGHT0, GL\_SPECULAR, light\_specular);  glLightfv(GL\_LIGHT0, GL\_POSITION, light\_position);  } |

moverJugador()

Este método se encargara de recibir una dirección que especificara el movimiento del jugador, ya sea arriba/abajo o izquierda/derecha. Dependiendo de la dirección ingresara a un condicional switch y verificara que este permitido el movimiento, se llamara a la función esValido() y verificara que no sea un muro al lugar donde se quiera mover sino un espacio vacío. Dentro de cada caso, dependiendo del movimiento se cambiaran las variables para el movimiento del actor principal (esfera) y asignara la posición dentro del ambiente grafico; al igual se mandara llamar la función display() que repintara el escenario.

Al final del método se verifica si el jugador llego al fin (cubo de la victoria) si es así se llamara la función nuevoMapa() para cargar el siguiente escenario.

|  |
| --- |
| void moverJugador(int direccion){  switch(direccion){  case MOV\_ARRIBA://Arriba  if(esValido(posX,posZ-1) && mapa[posZ-1][posX]!=1){  for (int i = 0; i < tamCubo; i++)  {  zActual-=1;  rotx-=4.5;  dir=false;  display();  }  posZ--;  }  break;  case MOV\_ABAJO://Abajo  if(esValido(posX,posZ+1) && mapa[posZ+1][posX]!=1){  for (int i = 0; i < tamCubo; i++)  {  zActual+=1;  rotx+=4.5;  dir=false;  display();  }  posZ++;  }  break;  case MOV\_IZQUIERDA://Izquierda  if(esValido(posX-1,posZ) && mapa[posZ][posX-1]!=1){  for (int i = 0; i < tamCubo; i++){  xActual-=1;  roty+=4.5;  dir=true;  display();  }  posX--;  }  break;  case MOV\_DERECHA://Derecha  if(esValido(posX+1,posZ) && mapa[posZ][posX+1]!=1){  for (int i = 0; i < tamCubo; i++){  xActual+=1;  roty-=4.5;  dir=true;  display();  }  posX++;  }  break;  }  if(mapa[posZ][posX]==3) mapaNuevo();  } |

copiarMapa()

Este método se encarga de copiar el mapa almacenado en el arreglo mapa al arreglo mapa2 para su uso en el método de crearCamino() y encontrar el camino optimo para llegar al fin (cubo de la victoria).

|  |
| --- |
| void copiarMapa(){  mapa2 = new int\*[altoMapa];  for(int i=0;i<altoMapa;i++){  mapa2[i] = new int[largoMapa];  for(int j=0;j<largoMapa;j++){  mapa2[i][j] = mapa[i][j];  }  }  } |

actualizar()

Este método se encarga de hacer un movimiento del camino óptimo en un hilo aparte, con el fin de que el usuario pueda mover y rotar la cámara durante el movimiento automático del actor principal. Este realiza un movimiento cada 250 milisegundos.

|  |
| --- |
| void actualizar(int value){  moverJugador(camino[value]-'0');  value++;  if(value<camino.size()){  glutTimerFunc(250,actualizar,value);  }else{  automatico = false;  }  } |

modoAutomatico()

Este método se encarga de realizar el movimiento automático del actor principal dentro del laberinto. Primero copia el mapa al arreglo mapa2 con la función copiarMapa(), posteriormente crea un string que contendrá la cadena recibida por el método crearCamino() la cual tendrá los movimientos a seguir por parte del actor para encontrar el camino optimo al fin (cubo de la victoria). Se verifica que el camino exista de ser lo contrario no se moverá la esfera y regresa el control al usuario; si existe el camino se llama la función actualizar() la cual recibirá las direcciones de movimiento (arriba/abajo e izquierda/derecha) para llegar al final.

|  |
| --- |
| void modoAutomatico(){  automatico = true;  copiarMapa();  camino = crearCamino();  if(camino.empty()){ //Si no hay camino posible  automatico = false;  return;  }  glutTimerFunc(0,actualizar,0);  } |

keyboard()

Método que permite la interacción entre el juego y el usuario. Aquí se encuentran asignadas las teclas para el movimiento de la esfera y la cámara en el ambiente grafico así como la salida de la aplicación. Esta parte esta mas especificada en el manual de usuario.

|  |
| --- |
| void keyboard(unsigned char key, int x, int y){  float t;  switch(key) {  case 'z':  case 'Z':  if(!automatico){  modoAutomatico();  }  break;  case 'w':  case 'W': //Arriba  moverJugador(MOV\_ARRIBA);  break;  case 's':  case 'S': //Abajo  moverJugador(MOV\_ABAJO);  break;  case 'a':  case 'A': //Izquierda  moverJugador(MOV\_IZQUIERDA);  break;  case 'd':  case 'D': //Derecha  moverJugador(MOV\_DERECHA);  break;  // ROTAR CAMARA LA IZQUIERDA  case '1':  t=camX;  light\_position[0]=camX=t\*cos(deltay\*3.1416/180)-camZ\*sin(deltay\*3.1416/180);  light\_position[2]=camZ=camZ\*cos(deltay\*3.1416/180)+t\*sin(deltay\*3.1416/180);  display();  break;  // ROTAR CAMARA LA DERECHA  case '2':  t=camX;  light\_position[0]=camX=t\*cos(-deltay\*3.1416/180)-camZ\*sin(-deltay\*3.1416/180);  light\_position[2]=camZ=camZ\*cos(-deltay\*3.1416/180)+t\*sin(-deltay\*3.1416/180);  display();  break;  // MOVER CAMARA HACIA ARRIBA  case '3':  if(camY<maxCamY){  camY+=2;  light\_position[1] += 2;  }  display();  break;  // MOVER CAMARA HACIA ABAJO  case '4':  if(camY>minCamY){  camY-=2;  light\_position[1] -= 2;  }  display();  break;  case '5':  camZ-=5;  light\_position[2] -=5;  display();  break;  case '6':  camZ+=5;  light\_position[2] +=5;  display();  break;  case 27:  delete texturas;  exit(0);  break;  }  } |

main()

Método principal que se encarga de cargar la aplicación. Aquí se asignan las funciones principales para el manejo y control de todo el programa, init(), display(), reshape() y keyboard(). Al igual se especifica el modo de visualizar en este caso con profundidad y las dimensiones de la ventana.

|  |
| --- |
| int main(int argc, char \*\*argv){  glutInit(&argc, argv);  glutInitDisplayMode(GLUT\_DOUBLE | GLUT\_DEPTH | GLUT\_RGB);  glutInitWindowPosition(50, 50);  Height = 800; Width = 1200;  glutInitWindowSize(Width, Height);  glutCreateWindow("Laberinto Magico");  init();  glutDisplayFunc(display);  glutReshapeFunc(reshape);  glutKeyboardFunc(keyboard);  glutMainLoop();  return 0;  } |

1.-Controles del Juego

|  |  |
| --- | --- |
| Tecla | Acción |
| **W** | Se mueve hacia adelante. |
| **S** | Se mueve hacia atrás. |
| **A** | Se mueve hacia la izquierda. |
| **D** | Se mueve hacia la derecha. |
| **Z** | Movimiento automático. |
| **1** | Rota la cámara hacia la izquierda. |
| **2** | Rota la cámara hacia la derecha. |
| **3** | Mueve la cámara hacia arriba. |
| **4** | Mueve la cámara hacia abajo. |
| **5** | Mueve la cámara sobre el eje Z negativo. |
| **6** | Mueve la cámara sobre el eje Z positivo. |

2.- Objetivo del juego

El objetivo del juego es hacer llegar la pelota desde donde inicia hacia el cuadrado azul para así pasar al siguiente mapa.

Cuando usted llegue al cubo de la victoria pasará al siguiente mapa como se muestra a continuación:





