Informe 3 Simulación

Harvey David Cortes
u1201712@unimilitar.edu.co
Carlos Andrés Gómez
u1201717@unimilitar.edu.co
Carlos Andrés Mendieta
u1201932@unimilitar.edu.co

***Abstract*—El programa se encarga de mostrar la simulación de tres partículas interconectadas entres sí por medio de un resorte. A su vez cada partícula esta anclada al techo por otro resorte, esta simulación hace uso de la librería de OpenGL y la solución de ecuaciones diferenciales por método analítico.**

***Index Terms*—Ecuaciones diferenciales, OpenGL, resorte, partícula.**

Objetivos

- Realizar la simulación de tres esferas unidas entre ellas por un resorte formando una base triangular. Cada esfera estará sujeta al techo por medio de un resorte.

Metodología

Lo primero que se hizo fue crear la clase vector, porque se iban a utilizar para la posición, velocidad, fuerza y los operaciones que se realizarían entre estos. Se crea una clase partículas que contiene métodos para el cálculo de fuerzas, pintar la partícula, y los enlaces entre ellas. Luego en *resorte.cpp* se inicializan las partículas y se grafican.

Clase Vector

En el Algoritmo 1 se declara el vector p de tres posiciones flotantes *x, y, z*, donde cada vez que se crea un vector estos valores flotantes tendrán un valor de 0.

Algoritmo 1 *vector.h*

```
#pragma once
class vector
{
public:
  float p[3];
  vector(float x = 0, float y = 0, float z = 0);
  ~vector();
};
```

En el Algoritmo 2 cuando el constructor recibe tres flotantes lo que hace es cambiar los valores que estaban en 0, por los recibidos en el vector en su respectiva posición.

Algoritmo 2 *vector.cpp*

```
#include "vector.h"
vector::vector(float x, float y, float z)
{
  p[0] = x;
  p[1] = y;
  p[2] = z;
}
vector::~vector()
{}
```

Clase partícula

En el Algoritmo 3 se crea la clase partícula, que contiene un flotante *m* que es la masa, *rad* el radio de la partícula, un vector *posición, velocidad, fuerza.* El constructor recibe un vector posición, un vector velocidad, un flotante *M* (masa) y un flotante *Rad* (radio). Se crea el método *Calcular_Fuerza* que se encarga de encontrar la sumatoria de las fuerzas, a este se le pasa un vector de tres posiciones flotante, un flotante *kd, ks* y *H* (paso de la simulación).

Se crea el método *Calcular_Resorte* que se encarga de encontrar la fuerza de la partícula cuando es conectada a otra partícula.

Algoritmo 3 *particula.h*

```
#pragma once
class particula
{
public:
  float m, df[6], f[6], rad, norml = 0, delta = 0;
  vector posicion, posini, velocidad, velini, Fuerza
    , ele, elevel;
  particula(vector pos, vector vel, float M, float
    Rad);
  particula();
  ~particula();
  void pintar();
  void enlace(particula ancla, particula part);
  void Calcular_ele(particula ancla);
  void Calcular_fa(particula ancla);
  void Calcular_Fuerza(float g[3], float kd, float
    ks, float H);
  void Calcular_Resorte(float g[3], float kd, float
    ks);
};
```

En el Algoritmo 4 se reciben dos vectores uno de posición y uno de velocidad, un flotante *M* y *Rad.* El *for* se encarga

de inicializar el vector fase que son dos vectores en uno, los primeros tres valores se llenan con la posición y los siguientes tres con al velocidad inicial.

Algoritmo 4 Cabecera y declaración de clase de *particula.cpp*

```
#include <math.h>
#include <stdlib.h>
#include <iostream>
#include <windows.h>
#include <string.h>
#include <stdio.h>
#include "particula.h"
particula::particula(vector pos, vector vel, float M
    , float Rad){
  posicion = pos;
  velocidad = vel;
  posini = pos;
  velini = vel;
  m = M;
  rad = Rad;
  for (int i = 0; i < 3; i++){
    f[i] = posicion.p[i];
    f[i + 3] = velocidad.p[i];
  }
}
```

Algoritmo 5 Primeras funciones de la clase en *particula.cpp*

```
void particula::Calcular_ele(particula ancla){
  vector L;
  float R;
  float norml;
  for (int i = 0; i < 3; i++)
    L.p[i] = ancla.posicion.p[i] - posicion.p[i];
  norml = sqrt(pow(L.p[0], 2) + pow(L.p[1], 2) + pow
    (L.p[2], 2));
  R = ancla.posicion.p[1] - posini.p[1];
  for (int i = 0; i < 3; i++)
    ele.p[i] = L.p[i] / norml;
  delta = norml - R;
}
void particula::Calcular_fa(particula ancla){
  vector L, Lvel;
  float R;
  for (int i = 0; i < 3; i++)
  {
    L.p[i] = ancla.posicion.p[i] - posicion.p[i];
    Lvel.p[i] = ancla.velocidad.p[i] - velocidad.p[i
    ];
  }
  norml = sqrt(pow(L.p[0], 2) + pow(L.p[1], 2) + pow
    (L.p[2], 2));
  //R = ancla.posicion.p[1] - posini.p[1];
  R = 50;
  for (int i = 0; i < 3; i++)
  {
    ele.p[i] = L.p[i] / norml;
    elevel.p[i] = (Lvel.p[i] * L.p[i]) / norml;
  }
  delta = norml - R;
}
void particula::Calcular_Resorte(float g[3], float
    kd, float ks){
  for (int i = 0; i < 3; i++)
    Fuerza.p[i] += (ks*delta + kd*elevel.p[i])*ele.p
    [i];
}
```

En el Algoritmo 5 se calcula la fuerza de la partícula *B,* el método recibe una partícula que será el ancla, se definen dos vectores y un flotante *R* que determinará el largo del resorte. En el primer *for* lo que se hace es llenar vector *L* y el vector *Lvel.* Luego se encuentra la norma del vector *L*, se le da un valor a *R,* en este caso fue 50.

El segundo *for* da valores a *ele* y *elevel.* Finalmente se define el miembro delta. Calcular resorte sumar a las fuerzas que se tienen el valor de la fuerza resultante de el resorte generado entre la partícula ancla y la partícula trabajada, para ello se utilizan los valores que se hallaron en el método anterior.

En el Algoritmo 6 se tiene el método *Calcular_Fuerza* que calcula la suma de fuerzas, a *df* (derivada de la fase) los primeros tres valores van a ser la velocidad de la partícula, y los siguientes tres valores son la aceleración. Se resuelve la ecuación diferencial teniendo en cuenta el paso *H,* finalmente la fuerza se reinicia en cero. El método *pintar* dibuja la partícula. El método *enlace* recibe dos partículas y dibuja la línea que las une.

Algoritmo 6 Segundas funciones de la clase en *particula.cpp*

```
void particula::Calcular_Fuerza(float g[3], float kd
    , float ks, float H){
  for (int i = 0; i < 3; i++)
  {
    Fuerza.p[i] += m*g[i]; //F=mg
    Fuerza.p[i] += -kd*velocidad.p[i];
    /*Fuerza.p[i] += delta*ks*ele.p[i];*/
  }
  // calcular derivada
  for (int i = 0; i < 3; i++)
  {
    df[i] = velocidad.p[i];
    df[i + 3] = Fuerza.p[i] / m;
  }
  // resover eq diferencial
  for (int i = 0; i < 6; i++)
    f[i] += (df[i] * H);
  // Actualizar variable
  for (int i = 0; i < 3; i++)
  {
    posicion.p[i] = f[i];
    velocidad.p[i] = f[i + 3];
    cout << posicion.p[i] << " ";
    Fuerza.p[i] = 0;
  }
  cout << endl;
}
void particula::pintar(){
  glColor3f(1, 0, 0);
  glutSolidSphere(rad, 20, 20);
}
void particula::enlace(particula ancla, particula
    part){
  glLineWidth(5);
  glBegin(GL_LINES);
  glColor3f(1, 1, 1);
  glVertex3fv(ancla.posicion.p);
  glVertex3fv(part.posicion.p);
  glEnd();
  glFlush();
}
particula::particula(){}
particula::~particula(){}
```

En el Algoritmo 7 se inicializan las constantes y las posiciones de las partículas que son tratadas como anclas (en el la gráfica no se ven), se inicializan velocidades de las partículas. masas y radios.

Algoritmo 7 Cabecera y declaración de variables de *resorte.cpp*

```
#include <math.h>
#include <stdlib.h>
#include <iostream>
#include <windows.h>
#include <string.h>
#include <stdio.h>
#include "vector.h"
#include "particula.h"
using namespace std;
#ifdef _WIN32
#include "glut.h"
#elif defined(__APPLE__)
#include <GLUT/glut.h>
#else
#include <GL/glut.h>
#endif
using namespace std;
float r = 5, t = 0, g[3] = { 0, -9.8, 0 }, m = 10, h
      = 0.1, Kd = 0.5, Ks = 3;
int i;
double movZ = 0, movY = 0, giro = 0, giro2 = 0;
void TransformacionesCamara();
vector pos = { 0, 100, 77.94f };
vector pos2 = { 45, 100, 0 };
vector pos3 = { -45, 100, 0 };
vector vel = { 0, 0, 0 };
vector vel2 = { 0, -100, 100 };
particula part(pos, vel, m, r);
particula part2(pos2, vel, m, r);
particula part3(pos3, vel, m, r);
vector va = { 0, 150, 77.94f };
vector va2 = { 45, 150, 0 };
vector va3 = { -45, 150, 0 };
particula ancla(va, vel, m, r);
particula ancla2(va2, vel, m, r);
particula ancla3(va3, vel, m, r);
```

En el Algoritmo 8 se dibujan los ejes del plano, se establece la posición de la cámara y la perspectiva de la simulación.

En el Algoritmo 9 la función *Graficar* se encarga de mostrar en pantalla las posiciones actualizadas de las partículas simuladas como se observa en la Figura 1.

En el Algoritmo 10 la función *OnKey* se encarga controlar la rotación y traslación del escenario, además de permitir asignar una fuerza extra en el eje *Y* de cada partícula con los números 1, 2 y 3.

En el Algoritmo 11 se le dan a cada partícula su ancla y se calcula la fuerza del resorte, a sus respectivas partículas, al final se agrega la fuerza que no se tienen que recalcular como la gravedad y la resistencia del aire.

En el Algoritmo 12 se encuentra la función *OnTimerGL* que se encarga de actualizar los valores cada cierto tiempo, también

Algoritmo 8 Configuración de cámara y dibujo ejes coordenados de *resorte.cpp*

```
void IniciarCG(){
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho(-300, 300, -300, 300, -300, 300);
  glMatrixMode(GL_MODELVIEW);
}
void TransformacionesCamara(){
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(30, 1, 0.001, 10000);
  glTranslated(0, movY, movZ);
  gluLookAt(400, 400, 400, 0, 0, 0, 0, 1, 0);
  glMatrixMode(GL_MODELVIEW);
}
void ejes(){
  glLineWidth(3); glBegin(GL_LINES);
  glColor3f(0.0, 1.0, 0.0);
  glVertex3d(0, -180, 0); glVertex3d(0, 180, 0);
  glColor3f(0.0, 0.0, 1.0);
  glVertex3d(0, 0, -180); glVertex3d(0, 0, 180);
  glColor3f(1.0, 0.0, 0.0);
  glVertex3d(-180, 0, 0); glVertex3d(180, 0, 0);
  glEnd();
}
```

Algoritmo 9 Función graficar de *resorte.cpp*

```
void Graficar()
{
  glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
  glMatrixMode(GL_MODELVIEW);
  glEnable(GL_DEPTH_TEST);
  glLoadIdentity();
  glRotated(giro, 0, 1, 0);
  TransformacionesCamara();
  ejes();
  glPushMatrix();
  glTranslatef(part.posicion.p[0], part.posicion.p
    [1], part.posicion.p[2]);
  part.pintar();
  glFlush(); glPopMatrix(); glPushMatrix();
  glTranslatef(part2.posicion.p[0], part2.posicion.p
    [1], part2.posicion.p[2]);
  part2.pintar();
  glFlush(); glPopMatrix(); glPushMatrix();
  glTranslatef(part3.posicion.p[0], part3.posicion.p
    [1], part3.posicion.p[2]);
  part3.pintar();
  glFlush(); glPopMatrix();
  part.enlace(ancla, part);
  part2.enlace(ancla2, part2);
  part3.enlace(ancla3, part3);
  part.enlace(part2, part);
  part2.enlace(part3, part2);
  part3.enlace(part, part3);
  glutSwapBuffers();
}
```

se encuentra la función *main* que llama las funciones de teclado, graficar, tiempo, el tamaño de ventana y la posición.

Algoritmo 10 Función teclado de *resorte.cpp*

```
void OnKey(unsigned char key, int x, int y)
{
  switch (key){
  case 'i': case 'I':
    movZ += 1;
    break;
  case 'k': case 'K':
    movZ -= 1;
    break;
  case 'o': case 'O':
    movY += 0.4f;
    break;
  case 'l': case 'L':
    movY -= 0.4f;
    break;
  case 'a': case 'A':
    giro += 0.4f;
    break;
  case 'd': case 'D':
    giro -= 0.4f;
    break;
  case 's': case 'S':
    giro2 -= 0.4f;
    break;
  }
  if (key == 27)
    exit(0);
  if (key == '1')
    part.Fuerza.p[1] += -200;
  if (key == '2')
    part2.Fuerza.p[1] += -200;
  if (key == '3')
    part3.Fuerza.p[1] += -200;
}
```

Algoritmo 11 Cálculos de *resorte.cpp*

```
void calc(){
  part.Calcular_fa(ancla);
  part.Calcular_Resorte(g, Kd, Ks);
  part.Calcular_fa(part2);
  part.Calcular_Resorte(g, Kd, Ks);
  part.Calcular_fa(part3);
  part.Calcular_Resorte(g, Kd, Ks);
  part.Calcular_Fuerza(g, Kd, Ks, h);
}
void calc2(){
  part2.Calcular_fa(ancla2);
  part2.Calcular_Resorte(g, Kd, Ks);
  part2.Calcular_fa(part);
  part2.Calcular_Resorte(g, Kd, Ks);
  part2.Calcular_fa(part3);
  part2.Calcular_Resorte(g, Kd, Ks);
  part2.Calcular_Fuerza(g, Kd, Ks, h);
}
void calc3(){
  part3.Calcular_fa(ancla3);
  part3.Calcular_Resorte(g, Kd, Ks);
  part3.Calcular_fa(part);
  part3.Calcular_Resorte(g, Kd, Ks);
  part3.Calcular_fa(part2);
  part3.Calcular_Resorte(g, Kd, Ks);
  part3.Calcular_Fuerza(g, Kd, Ks, h);
}
```

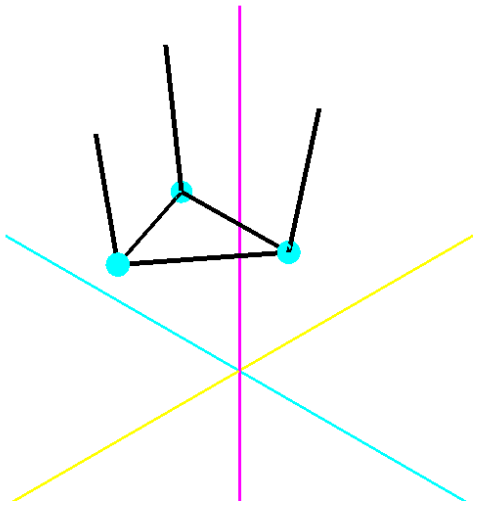

Fig. 1. Resultado de la simulación con colores invertidos.

Algoritmo 12 Función de tiempo y main de *resorte.cpp*

```
void OnTimerGL(int id){
  calc(); calc2(); calc3();
  glutPostRedisplay();
  glutTimerFunc(10, OnTimerGL, 1);
  t += h;
}
int main(int argc, char **argv){
  glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB |
    GLUT_DEPTH);
  glutInitWindowSize(800, 800);
  glutInit(&argc, argv);
  glutInitWindowPosition(200, 200);
  glutCreateWindow("Programa Basico");
  glutKeyboardFunc(OnKey);
  IniciarCG();
  glutDisplayFunc(Graficar);
  glutTimerFunc(1, OnTimerGL, 1);
  glutMainLoop();
  system("pause");
  return 0;
}
```