Mi primer sistema de particulas

una aproximación “divertida” a clases y herencias en C++

Patricio Gonzalez Vivo.com
procesos creativos integrados

Introducción

Este archivo da por sentado que se han leído los anteriores, se ha instalado satisfactoriamente openFrameworks, creado los proyectos y compilado el core de librerías. A su vez también se recomienda que antes de arrancar se halla compilado los proyectos de ejemplo que se pueden encontrar en el apps/examples y apps/addonsExamples .

También damos por sentado que se ha aprendido a compilar en cada plataforma utilizando correctamente el IDE correspondiente.

1er paso: dibujar algo

Empecemos dibujando algo escribiendo en el testApp.cpp:

```
#include "testApp.h"

void testApp::setup(){

}

void testApp::update(){

}

void testApp::draw(){
      ofBackground(0);
      ofSetColor(255,0,0);          // ofSetColor() altera el color de
                                    // todas las figuras que se dibujen
                                    // luego de ser declarada
      ofCircle(100, 100, 30);
}
```

Aquí sólo estamos dibujando es: primero el fondo (`ofBackground(int)`) y luego un círculo (`ofCircle(float,float,float)`) rojo (`ofSetColor(int,int,int)`) en 100 de x y 100 de con 30 pixels de radio.

Si quisiéramos dibujar un rectángulo la función que necesitaríamos sería `ofRect(x,y,width,height)`.

Jugando con los valores x e y podemos comprender rápidamente cuales son los valores 0,0 en pantalla. Al mismo tiempo que modificando los valores de color en ofSetColor y ofBackground podemos comprender las diferentes formas de declarar un color de acuerdo a la luminancia monocromática o en cada canal.

Al mismo tiempo podemos explorar el uso de los métodos de `ofFill()` y `ofNoFill()` los cuales establecen si las figuras se dibujarán rellenas o tan sólo el borde de las mismas.

Otros valores para cambiar es el ancho de la linea: `ofSetLineWidth( float ancho )`

2do paso: moverlo

Para agregarle un poco de movimiento podemos dibujar en cada itineración el mismo círculo en la posición donde se encuentra el mouse. Esto se realiza con dos variables propias del ofBaseApp que poseen la posición en X (mouseX)y en Y (mouseY) del mouse.

```
void testApp::draw(){
      ofBackground(0);
      ofSetColor(255, 0, 0);
      ofCircle(mouseX, mouseY, 30);
}
```

Si en vez de dibujar un círculo dibujamos un rectángulo notaremos que el modo de posicionar el rectángulo es distinto para uno y otro. En el primero el dibujo se ancla en el centro del círculo, mientras que en el segundo se dibuja desde la esquina superior izquierda.

```
void testApp::draw(){
      ofBackground(0);
      ofSetColor(255, 0, 0);
      ofRect(mouseX, mouseY, 30, 30);
}
```

Esto puede cambiarse con el método `ofSetRectMode( OF_RECTMODE_CENTER )` / `ofSetRectMode( OF_RECTMODE_CORNER )`

Si quisiéramos volver este código más prolijo podríamos actualizar la posición de la figura en el update() y dibujar la figura en el draw(). Para poder preservar esa información necesitaríamos agregar una variable global a toda la clase del testApp.

En el testApp.h:

```
#pragma once
#include "ofMain.h"

class testApp : public ofBaseApp{
public:
      void setup();
      void update();
      void draw();

      void keyPressed  (int key);
      void keyReleased(int key);
      void mouseMoved(int x, int y );
      void mouseDragged(int x, int y, int button);
      void mousePressed(int x, int y, int button);
      void mouseReleased(int x, int y, int button);
      void windowResized(int w, int h);
      void dragEvent(ofDragInfo dragInfo);
```


```
	void gotMessage(ofMessage msg);

	int	x,y;
};
```

En el testApp.cpp:

```
#include "testApp.h"
void testApp::setup(){
	x = ofGetWindowWidth()/2;	// El circulo comienza en el centro
	y = ofGetWindowHeight()/2;	// de la ventana
}

void testApp::update(){
	x = mouseX;
	y = mouseY;
}

void testApp::draw(){
	ofBackground(0);
	ofSetColor(255, 0, 0);
	ofCircle(x, y, 30);
}
```

Con los métodos `ofGetWindowWidth()` y `ofGetWindowHeight()` podemos saber cuanto mide de alto y largo nuestra ventana. De igual manera con los métodos `ofGetScreenWidth()` y `ofGetScreenHeight()` podríamos saber el alto y largo del monitor.

Tarea: investigar que otros parámetros se puede objetener de la ventana activa en el header ofAppRunner.h dentro del directorio openFrameworks/libs/app

3er paso: volverlo una clase

Ahora que nuestro “programa” posee cierta lógica interna respecto a como inicia, como acutaliza y como se dibuja, podríamos transformar este círculo en un objeto. Más bien en una clase Pelota.

Por un lado creamos la clase (esto quiere decir empezar creando el archivo header o .h y el source o .cpp correspondiente)

Comencemos con Pelota.h

```
#ifndef PELOTA    // recordemos que esto previene declarar la misma clase
#define     PELOTA      // dos veces

#include "ofMain.h"     // Aqui agregamos los metodos y clases de oF

class Pelota {
public:                 // Todos los métodos aquí son públicos por los que
cualquiera puede
                  // verlos y acceder a ellos.

      Pelota();   // Constructor indispensable en toda clase

      void update(int _x, int _y);
      void draw();

      int x,y;
};
#endif
```

En el Pelota.cpp:

```
#include "Pelota.h"           // Referencia al header donde los siguientes
métodos estan
                        //declarados
Pelota::Pelota(){
      x = ofGetWindowWidth()/2;
      y = ofGetWindowHeight()/2;
}

void Pelota::update(int _x, int _y){
      x = _x;
      y = _y;
}

void Pelota::draw(){
      ofSetColor(255, 0, 0);
      ofCircle(x, y, 30);
}
```

Ahora tan sólo falta agregar este objeto a nuestro testApp.h

```
#pragma once

#include "ofMain.h"
#include "Pelota.h"           // es importante agregar el header donde se
declara el objeto

class testApp : public ofBaseApp{
public:
	void setup();
	void update();
	void draw();

	void keyPressed  (int key);
	void keyReleased(int key);
	void mouseMoved(int x, int y );
	void mouseDragged(int x, int y, int button);
	void mousePressed(int x, int y, int button);
	void mouseReleased(int x, int y, int button);
	void windowResized(int w, int h);
	void dragEvent(ofDragInfo dragInfo);
	void gotMessage(ofMessage msg);

	Pelota	p;
};
```

En el el testApp.cpp tan sólo debe escribirse:

```
#include "testApp.h"
void testApp::setup(){
}

void testApp::update(){
	p.update(mouseX,mouseY);
}

void testApp::draw(){
	ofBackground(0);
	p.draw();
}
```


4to paso: aplicarle cierta física

Desde la versión 007 oF cuenta con clases para realizar cálculos vectoriales (de dos, tres o cuatro dimensiones) de forma sencilla. Hechándole una mirada al directorio `openFrameworks/libs/openframeworks/math` podemos encontrar que también existen clases nativas para realizar calculos de matrices y quaterniones.
En este paso vamos a hacer uso de esas clases reemplazando los `int x` e `y` por un vector de dos dimensiones con valores de punto flotante (numero con coma) que guarde la posición de nuestro objeto, para eso nos vamos a valer de la clase `ofVec2f()` definida en `ofVec2f.h`
Además agregaremos dos vectores más para calcular la aceleración y la velocidad.
Así mismo crearemos dos variables más. Una contendrá el tamaño del objeto, mientras que la última será una variable de tipo `ofColor()` para contener el color del mismo. Este tipo de variables nos permite realizar operaciones de color a la vez que converciones. Vale la pena hecharle una mirada a `libs/openFrameworks/types/ofColor.h` para comprender cómo se las declara y cuantas funciones útiles posee.

En la rutina de update haremos que la aceleración actualice la velocidad. La cual dependerá de la resistencia o densidad del espacio, la cual moverá la posición del objeto.

Para que nuestro objeto se mueva en vez de pasarle una posición le pasaremos un vector de fuerza hacia donde debe moverse.

Una vez hecho todo esto nuestra clase Pelota.h debería lucir algo así:

```
#ifndef PELOTA
#define PELOTA
#include "ofMain.h"

class Pelota {
public:
	Pelota();

	void agregarFuerza(ofVec2f fuerza); // aquí le pasamos la fuerza

	void update();
	void draw();

	ofColor	color;
	ofVec2f	pos, vel, acc;

	int	tamanio;
};
#endif
```

Mientras que el Pelota.cpp:

```
#include "Pelota.h"

Pelota::Pelota(){
```

```
	tamanio = 30;
	color.set(255,0,0);

	pos.set( ofGetWindowWidth()*0.5 ,   // es un buen hábito
		 ofGetWindowHeight()*0.5 ); // usar más matemáticas,
					    // sobretodo normales para
					    // hacer ajustes
	vel.set(0,0);
	acc.set(0,0);
}

void Pelota::agregarFuerza(ofVec2f fuerza){
	acc += fuerza;
}

void Pelota::update(){
	vel += acc;		// Suma la aceleración a la velocidad
	vel *= 0.03;		// Le agregamos algo de resistencia o densidad
	pos += vel;		// Suma la velocidad a la posición
	acc *= 0;		// Vuelve a cero la aceleración
}

void Pelota::draw(){
	ofSetColor(color);
	ofCircle(pos.x, pos.y, tamanio);
}
```

Mientras que en el testApp.cpp ahora actualizamos la posición de la siguiente manera:

```
void testApp::update(){
	ofVec2f destino, haciaDestino;

	destino.set(mouseX,mouseY);
	haciaDestino = destino - p.pos;

	p.agregarFuerza(haciaDestino);
	p.update();
}
```

Al compilar podemos ver como la pelota es atraída por el cursor. Sin embargo esto dista mucho a comportarse como una pelota.

5to paso: un poco de onda

Como parte de algunas funciones matemáticas incorporadas al framework podemos agregarle valores random para hacer las cosas un poco más interesante. Para esto a la hora de inicializar utilizaremos la clase ofRandom de dos maneras. Un con un sólo parámetro explicitando el máximo y otra por medio de dos especificando el minimo y máximo de los valores random que devuelva.
Para este ejemplo le agregaremos una función más a nuestra clase Pelota para saber cuando ha salido de la pantalla de tal forma que podamos volverla a crear. También dejaremos que las pelotas caigan por gravedad.

Pelota.h:

```
#ifndef PELOTA
#define     PELOTA
#include "ofMain.h"

class Pelota {
public:
      Pelota();

      void agregarFuerza(ofVec2f fuerza);
      bool pasoElBorde();
      void update();
      void draw();

      ofColor     color;
      ofVec2f pos, vel, acc;

      int   tamanio;
};
#endif
```

Pelota.cpp:

```
#include "Pelota.h"

Pelota::Pelota(){
      tamanio = ofRandom(20);
      color.set(ofRandom(255),ofRandom(255),ofRandom(255));

      pos.set( ofGetWindowWidth()*0.5 , ofGetWindowHeight()*0.5 );
      vel.set( ofRandom(-1,1),ofRandom(-1,1));
      acc.set( ofRandom(-1,1),ofRandom(-1,1));
}

void Pelota::agregarFuerza(ofVec2f fuerza){
      acc += fuerza;
}

void Pelota::update(){
```

```
	vel += acc; // Suma la aceleración a la velocidad
	pos += vel; // Suma la velocidad a la posición
	acc *= 0;   // Vuelve a cero la aceleración
}

void Pelota::draw(){
	ofSetColor(color);
	ofCircle(pos.x, pos.y, tamanio);
}

bool Pelota::pasoElBorde(){
	bool rta = false;
	if ( pos.x < 0 || pos.y < 0 || pos.x > ofGetWindowWidth() || pos.y >
ofGetWindowHeight())
		rta = true;

	return rta;
}
```

en el update() del testApp.cpp:

```
void testApp::update(){
	p.agregarFuerza(ofVec2f(0,0.0098));
	p.update();

	if (p.pasoElBorde()){
		p = Pelota();			// las pelotas se vuelven a
		p.pos.set(mouseX,mouseY);	// crear donde esta el cursor
	}
}
```


6to paso: multiplicarlo

En cuanto sistema de partículas se refiere, más es mejor. Así que hagamos un array de estos objetos que estamos usando. Los arrays en C++ son estáticos. Eso quiere decir que una vez creados con cierto tamaño no pueden modificarse, por lo menos en principio.
Vamos a utilizar el pre-compilador para crear una variable constante donde guardar el número de objetos en nuestro array. Esto nos ayudará luego a itinerar por cada uno de ellos fácilmente. Pudiendo cambiar la cantidad total de forma sencilla.

En el testApp.h:

```
#pragma once

#include "ofMain.h"
#include "Pelota.h"

#define TOTAL 100       // Esta bueno explorar hasta que numero puede llegar

class testApp : public ofBaseApp{
public:
            void setup();
            void update();
            void draw();

            void keyPressed  (int key);
            void keyReleased(int key);
            void mouseMoved(int x, int y );
            void mouseDragged(int x, int y, int button);
            void mousePressed(int x, int y, int button);
            void mouseReleased(int x, int y, int button);
            void windowResized(int w, int h);
            void dragEvent(ofDragInfo dragInfo);
            void gotMessage(ofMessage msg);

            Pelota      p[TOTAL];
};
```

En el testApp.cpp:

```
#include "testApp.h"
void testApp::setup(){
      for(int i = 0; i < TOTAL; i++){
            p[i].pos.set(ofRandom(ofGetWindowWidth()
),ofRandom(ofGetWindowHeight()));
      }
}

void testApp::update(){
      for(int i = 0; i < TOTAL; i++){
            p[i].agregarFuerza(ofVec2f(0,0.0098));
            p[i].update();
```


```
            if (p[i].pasoElBorde()){
                  p[i] = Pelota();
                  p[i].pos.set(mouseX,mouseY);
            }
      }
}

void testApp::draw(){
      ofBackground(0);

      for(int i = 0; i < TOTAL; i++){
            p[i].draw();
      }
}
```


7mo paso: buscando sutilezas

Otra forma de jugar con valores random es utilizar `ofNoise`. Este método devuelve valores más orgánicos. Una de las variantes de la misma clases es `ofSignedNoise()`.
En este paso intentaremos darle movimientos y efectos más interesantes a nuestra clase.
Para eso le agregaremos un tiempo de vida a nuestra Pelota. Al mismo tiempo que estaremos atentos a la cantidad de itineraciones del loop principial del testApp. Si su procesador es rápido quizá hayan notado que por momentos las partículas se movían sumamente rápido y por otros momentos se enlentecían. Eso era causado porque no habíamos fijado la cantidad de frames por segundos en las que nuestro loop principal trabaja. Lo fijaremos en 60 con el método `ofSetFramerate(60);` . También habilitaremos el uso de colores con canal alpha mediante `ofEnableAlphaBlending()`
A nuestra clase le agregarémos un par de funciones y variables de tal modo que queden así el header Pelota.h:

```
#ifndef PELOTA
#define     PELOTA

#include "ofMain.h"

class Pelota {
public:
      Pelota();

      void agregarFuerza(ofVec2f fuerza);
      void agregarNoise(float _angulo, float _turbulencia);
      //agregará un movimiento aleatorio suave

      void agregarAlphaFade(bool _fadeOut);     // hará que se desvanezcan
      void agregarScaleFade(bool _melt);        // hará que se achiquen

      bool pasoElBorde();
      bool estaViva();                    // esta función chequea si esta viva

      void update(float _limiteDeVelocidad);
      // el update quitará vida y limitará el movimiento

      void draw();

      ofColor     color;

      ofVec2f pos, vel, acc;
      float alphaF, escalaF;
      int vida,vidaInicial, tamanio;
};
#endif
```

En el source Pelota.cpp:

```
#include "Pelota.h"
```

```
Pelota::Pelota(){
	tamanio = ofRandom(20);
	color.set(ofRandom(255),ofRandom(255),ofRandom(255));

	pos.set( ofGetWindowWidth()*0.5 , ofGetWindowHeight()*0.5 );
	vel.set( ofRandom(-1,1),ofRandom(-1,1));
	acc.set( ofRandom(-1,1),ofRandom(-1,1));

	alphaF	= 1;
	escalaF = 1;

	vida = vidaInicial = ofRandom(200,1000);
}

void Pelota::update(float _limiteDeVelocidad = 0.0){
	vel += acc; // Suma la aceleración a la velocidad

	if (_limiteDeVelocidad != 0)
		vel.limit(_limiteDeVelocidad);
	pos += vel; // Suma la velocidad a la posición
	acc *= 0;   // Vuelve a cero la aceleración

	vida--;
}

void Pelota::agregarFuerza(ofVec2f fuerza){
	acc += fuerza;
}

void Pelota::agregarNoise(float _angulo, float _turbulencia){
	float angulo = ofSignedNoise(pos.x * 0.005f, pos.y *0.005f) * angulo;
	ofVec2f noiseVector( cos( angulo ), sin( angulo ) );
	acc += noiseVector * _turbulencia * (1.0 - ofNormalize(vida, 0,
vidaInicial));
}

void Pelota::agregarAlphaFade(bool _fadeOut = true){
	if (_fadeOut)
		alphaF = 1.0f-ofNormalize(vida, 0,vidaInicial);
	else
		alphaF = ofNormalize(vida, 0,vidaInicial);
}

void Pelota::agregarScaleFade(bool _melt = true){
	if (_melt)
		escalaF = 1.0f-ofNormalize(vida, 0,vidaInicial);
	else
		escalaF = ofNormalize(vida, 0,vidaInicial);
}

void Pelota::draw(){
	ofSetColor(color, color.a * alphaF );
	ofCircle(pos.x, pos.y, tamanio * escalaF );
}
```


```
bool Pelota::pasoElBorde(){
	bool rta = false;

	if ( pos.x < 0 || pos.y < 0 || pos.x > ofGetWindowWidth() || pos.y >
ofGetWindowHeight())
		rta = true

	return rta;
}

bool Pelota::estaViva(){
	bool rta = true;

	if (vida <= 0)
		rta = false;

	return rta;
}
```

En el testApp.h hace falta agregar un integer llamado tiempo, mientras que en el testApp.cpp:

```
void testApp::setup(){
	ofEnableAlphaBlending();
	ofSetFrameRate(60);

	for(int i = 0; i < TOTAL; i++)
		p[i].pos.set( ofRandom(ofGetWindowWidth()),
				ofRandom(ofGetWindowHeight()));

	tiempo = 0;
}


void testApp::update(){
	for(int i = 0; i < TOTAL; i++){
		p[i].agregarFuerza(ofVec2f(0,0.00098));
		p[i].agregarNoise(2.7, 0.76);
		p[i].agregarAlphaFade(false);
		p[i].agregarScaleFade(false);
		p[i].update(1);


		if (p[i].pasoElBorde() || !p[i].estaViva() ){
			p[i] = Pelota();
			p[i].color.setHue( (tiempo%3000)*0.1 );
			p[i].pos.set(mouseX,mouseY);
		}
	}

	tiempo++;
}
```


```
void testApp::draw(){
	ofBackground(0);

	for(int i = 0; i < TOTAL; i++)
		p[i].draw();
}
```


8vo paso: variedad (herencia y polimorfismo)

Unas de las novedades que C++ trajo al mondo de la programación allá a mediados de los ‘80 fue el concepto de programación orientada a objetos. Hasta aquí hemos estado trabajando con el objeto Pelota. Sin embargo, queda muchísimo del potencial de la programación orientada a objetos por descubrir.

Una de sus principales características se denomina herencia. Podemos utilizar la clase Pelota como padre de otras. Estas heredarán todas las variables y funciones públicas que esta tengan.

Por ejemplo:

```
#ifndef PELOTITA
#define PELOTITA

#include "ofMain.h"
#include “Pelota.h”     // Es importante incorporar el .h a la clase padre

class Pelotita : public Pelota{
public:
      Pelotita();
      void draw();
};

#endif
```

En este caso la clase Pelotita hereda todas las características de Pelota sin embargo podremos definir un estilo propio de modo de dibujarla a esto se denomina polimorfismo . Una cuestión muy útil de todo esto es que en un array de Pelotas podríamos insertar Pelotitas . De tal forma que en un sólo for podríamos itinerar por Pelotas y Pelotitas actualizándolas o dibujándolas a todas.

9no paso: cargando Imágenes

El último paso de este recorrido tiene que ver con remplazar nuestro humilde círculo coloreado por algo más interesante. Para eso vamos a bajar una imagen (http://github.com/patriciogonzalezvivo/OF05/blob/master/bin/data/bola.png) copiarla al directorio bin/data de nuestro proyecto y levantarla desde nuestra App.
El plan es el siguiente, envez de generar 100 copias, una por cada pelota, vamos a cargar en memoria una sólo de estas imágenes y en cada objeto le vamos a pedir que la "dibuje" en su respectivo color y tamaño.
Para levantar la imagen tan sólo necesitamos crear una variable `ofImage()` y cargar el contenido con el método `.loadImage(string _archivo)`
A cada objeto vamos a pasarle la dirección en memoria de esta imagen, utilizando el comando & a un la función de cada objeto draw(). Esta va estar esperando un puntero. Osea va a estar mirando hacia el contenido de esa dirección la cual nos pasa. Así va aquedar la función draw de la clase Pelota:

```
void Pelota::draw(ofImage* _imagen){ // espera un puntero de tipo ofImage
      ofSetColor(color, color.a * alphaF );
      _imagen->draw( pos.x - _imagen->getWidth() * escalaF,
                     pos.y - _imagen->getHeight() * escalaF,
                     _imagen->getWidth() * escalaF,
                     _imagen->getHeight() * escalaF);
}
```

Podemos ver que el objeto se comporta como debería, con la única diferencia que en vez de utilizar puntos (".") estamos usando flechas ("->").

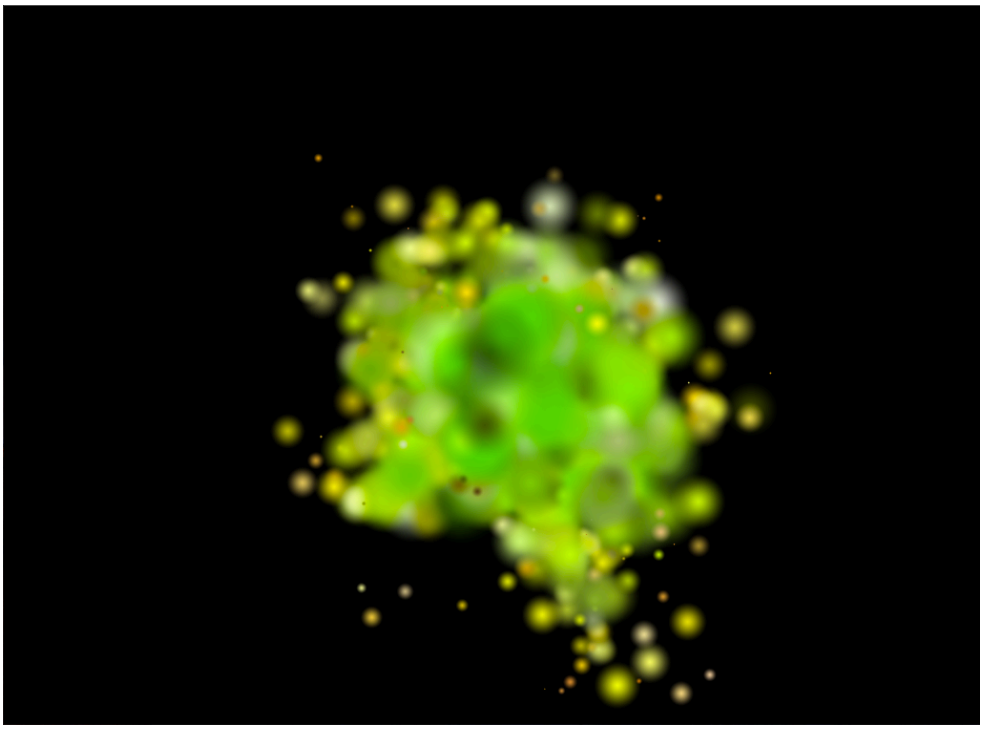
Desde la testApp.cpp tan sólo debemos:

```
void testApp::draw(){
      ofBackground(0);

      for(int i = 0; i < TOTAL; i++){
            p[i].draw(&bola);     // Le pasa la dirección en memoria
      }
}
```

Punteros es un tema importante en C++ el cual volveremos a explicar con mayor detalle en siguiente tutorial. Hasta entonces recomendamos hecharle una leida a los siguientes links: http://es.wikibooks.org/wiki/Programaci%C3%B3n_en_C%2B%2B/Punteros
http://es.scribd.com/doc/19125272/POO-Punteros-en-C
http://elvex.ugr.es/decsai/c/apuntes/punteros.pdf

A continuación una imagen de como debería verse seguida de un link con los códigos



Código:
http://github.com/patriciogonzalezvivo/OF05

Tarea:
- Transformar esta clase en una sub-clase que herede de Pelota
- Investigar sobre arrays dinámicos y aplicar el tipo <vector> (parte la librería estandar de c++) a nuestro sistema de partículas.
http://ronnyml.wordpress.com/2009/07/04/vectores-matrices-y-punteros-en-c/
http://es.wikibooks.org/wiki/Programaci%C3%B3n_en_C%2B%2B/Biblioteca_Est%C3%A1ndar_de_Plantillas/Vectores
- Investigar sobre las diferencia entre <vector> y <list>.
http://es.wikibooks.org/wiki/Programaci%C3%B3n_en_C%2B%2B/Librer%C3%ADa_Est%C3%A1ndar_de_Plantillas/Listas