

# Programació en C++: Atracció gravitatòria entre partícules

INST MANOLO HUGUÉ  
DEPARTAMENT DE MATEMÀTIQUES  
2 BATXILLERAT B  
6/2/2012

# Índex

1 Introducció.....	2
2 Introducció a la programació.....	3
3 El llenguatge C++.....	5
3.1 Origen del C++.....	5
3.2 Sintaxis del C++.....	7
3.2.1 Tokens.....	7
3.2.2 Tipus de dades.....	8
3.2.3 Exemple d'un programa en C++.....	10
3.2.4 Comparació If.....	12
3.2.5 Bucles.....	13
3.2.6 Classes.....	14
3.2.7 Estructures.....	17
4 SDL.....	18
4.1 Quant a SDL.....	18
4.2 Programació amb SDL.....	19
5 Simulador d'atracció gravitatòria entre partícules.....	21
5.1 Creació d'una partícula amb moviment.....	21
5.1.1 Part gràfica del programa.....	21
5.1.2 Creació de la partícula.....	24
5.1.3 Funció principal del programa.....	27
5.1.3 Control de la freqüència del programa.....	32
5.2 Atracció gravitatòria de les partícules.....	35
5.2.1 Estructuració del programa.....	35
5.2.2 Llibreria Vector.....	35
5.2.3 Llibreria Partícula.....	40
5.2.4 Programa principal.....	43
5.3 Atracció Entre varies partícules.....	44
5.3.1 Atracció entre totes les partícules.....	44
5.3.2 Introducció de dades.....	45
5.4 Millorant el programa.....	51
5.4.1 Dibuixar trajectòria.....	51
5.4.2 Posicionament de les partícules.....	51
5.4.3 Separació de la freqüència dels càlculs i gràfica.....	53
5.5 Simulació del Sistema Solar.....	55
6 Conclusions.....	57
7 Bibliografia.....	59

# 1 Introducció

He triat un treball sobre la programació perquè és un tema que em sembla molt interessant i que està molt relacionat amb els estudis que vull fer més endavant. Jo partia d'uns coneixements bàsics de programació, amb la qual cosa no sabia fins a quin punt podria desenvolupar el meu treball.

Gràcies a un curs que vaig fer a l'UPC sobre programació en C++ durant una setmana, vaig poder ampliar més els meus coneixements de programació per a poder desenvolupar el projecte. Una altra bona part de coneixements l'he adquirida als tutorials de la web [lazyfoo.net](http://lazyfoo.net).

L'objectiu del treball, és crear un programa informàtic que pugui simular l'atracció gravitatòria d'unes partícules introduïdes per l'usuari. És a dir, que l'usuari pugui introduir les dades que ell vulgui, i el programa representi gràficament l'atracció entre les partícules.

El treball està dividit en dos grans parts. La primera, on s'explica què és la programació, com ha evolucionat, i s'explica una mica com funciona el C++ i com escriure amb ell. La segona part consta de la part pràctica, que consta en programar un programa des de 0 que simuli l'atracció gravitatòria entre partícules donades per l'usuari.

## 2 Introducció a la programació

Avui dia els ordinadors son capaços de realitzar tot tipus de tasques diferents, des de navegar per Internet fins a editar vídeos, però tots els processos no els fa l'ordinador sol, només segueix les instruccions donades. Per ser més concrets, els ordinadors només entenen un llenguatge, el llenguatge màquina, basat en uns i zeros. L'ordinador només pot determinar si per un circuit passa corrent (1) o si no hi passa (0) així doncs, el llenguatge màquina es basa en llargues cadenes d'uns i zeros.

El problema del llenguatge màquina és que era massa complicat per a les persones, el codi era molt extens i per programar era completament necessari conèixer l'arquitectura de la màquina. Així que els programadors van decidir crear una traducció del llenguatge màquina que fos una traducció dels uns i zeros per paraules més fàcils d'entendre per l'home. Un dels llenguatges que es va crear, va ser l'assemblador.

El llenguatge assemblador permetia utilitzar diverses paraules que feien una certa funció, com sumar i restar. Així doncs, per sumar es podia escriure simplement "ADD" per substituir tota la cadena d'uns i zeros que feien la mateixa funció. Suposem que volguéssim escriure només l'article "al", en uns i zeros, correspondria a "10110000 01100001 ". En assemblador s'escriuria "mov al, 061h" Amb això la màquina entén que "al" s'ha d'assignar al valor en base 16, que correspon a 10110000 01100001 . Així moltes parts del codi podies ser simplifiades.

Però tot i que hi havia un llenguatge derivat d'un altre, feia falta algun programa que fes possible que l'ordinador entengués l'assemblador. Aquests programes, anomenats programes assembladors, permetien traduir del llenguatge assemblador, a codi màquina.

Posteriorment van anar sorgint diferents llenguatges de programació que cada vegada s'allunyaven del codi màquina i s'assemblaven més al llenguatge humà i, per tant, més fàcils de programar i més simples. Aquest tipus de llenguatge s'anomena llenguatges d'alt nivell. Un dels primers llenguatges d'alt nivell que van aparèixer és el Fortran. Proposat per John Backus a IBM com a substitut del llenguatge assemblador per a la computadora central IBM 704. Un altre llenguatge creat va ser el COBOL, data del 1960 i utilitzat encara en informàtica de gestió.

Com més complicades eren les tasques que havien de complir els ordinadors, més necessari era un llenguatge més simple i senzill per poder dur-les a terme. Tots aquests llenguatges són d'alt nivell, mentre que el llenguatge assemblador i el màquina corresponen a llenguatges de baix nivell.

Mentre que alguns llenguatges d'alt nivell com el BASIC (creat el 1980) permetia fer una operació com sumar, en una sola línia de codi, en llenguatge màquina per fer la mateixa operació, calien moltes més línies de codi.

Així doncs, sembla que els llenguatges d'alt nivell siguin molt millors que els de baix nivell, per la seva facilitat d'us al ser molt més intuïtius i curts, però no són tot avantatges, aquí estan les diferències entre un i l'altre

Alt nivell	Baix nivell
El llenguatge és més semblant al de les persones, més senzill	Llenguatge basat en uns i zeros, més complicat
El mateix codi serveix per a diverses màquines	El codi varia en funció de l'arquitectura del ordinador
El programes són més lents	Gran eficàcia al estar directament escrit perquè la màquina ho entengui
Els programes son més curts.	Els programes són llargs

Com es pot comprovar, el principal defecte és l'eficàcia del programa en alt nivell, això es debut a que un cop s'ha escrit el programa, cal seguir uns processos perquè la màquina pugui entendre el codi.

## 3 El llenguatge C++

### 3.1 Origen del C++

Durant els anys 60, van aparèixer nous llenguatges de programació. Un d'ells va ser el ALGOL 60, com alternativa del Fortran, inspirant-se en ell en alguns conceptes d'estructura de programació que més tard va inspirar a altres llenguatges de programació com CPL i els seus successors (C++ entre ells). Tot i així, ALGOL no era un llenguatge específic, i al ser bastant abstracte, va fer que sigués molt difícil programar tasques comercials amb ell.

Al 1963 va sortir el llenguatge CPL (Combined Programming Language o Llenguatge de programació combinat) que pretenia ser un llenguatge més concret que el Fortran i el ALGOL per a tasques de programació. Malgrat tot, això feia que el llenguatge fos realment extens i complex, i, per tant, difícil d'aprendre i d'implementar.

Al 1967 va aparèixer BCPL (Basic Combined Programming Language o Llenguatge de programació combinat bàsic) de la mà de Martin Richards. Aquest llenguatge tenia l'objectiu de simplificar el llenguatge CPL però mantenint les funcions. Tot i així, seguia sent un llenguatge massa extens i complicat.

Al 1970, Ken Thompson, va crear el llenguatge B, derivat del BCPL per a una màquina i un sistema específic (DEC PDP-7 amb UNIX) i va ser adaptat al seu gust. Encara que era dependent del sistema, era una simplificació del CPL molt més gran del que ho era el BCPL. Tenia certes limitacions, com que no es podia compilar en un fitxer executable, s'havia d'anar interpretant per un intèrpret mentre s'executava, així que el feia lent i no era viable per a fer sistemes operatius. Aleshores, un any després, al 1971, Dennis Ritchie, dels laboratoris Bell, va començar a treballar en un compilador de llenguatge B que permetés, entre altres coses, generar el codi executable de cop. Finalment, aquest nou "B" amb el compilador, es va acabar anomenant C, introduït també més funcions.

Al 1973, Dennis Ritchie, el creador del compilador del llenguatge B que va ser posteriorment el llenguatge C, mort recentment, va treballar en la base del C. Amb el avanç del llenguatge com la millora de matrius i punters, i la demostració de la capacitat que tenia el llenguatge sense ser del tot un llenguatge d'alt nivell, va fer que el llenguatge C s'expandís.

Al 1980, Bjarne Stroustrup, dels laboratoris Bell, va començar a desenvolupar el llenguatge C++, que rebria aquest nom a finals del 1983, quan es publicaria el primer manual del llenguatge. Al Octubre de 1985, va sortir la primera versió comercial de C++ com la primera edició del llibre “El llenguatge de programació C++” per Bjarne Stroustrup.

Durant els 80, es va anar polint el llenguatge C++ fins que va tenir una personalitat pròpia, que el distingien dels altres llenguatges. Tot això sense perdre pràcticament la compatibilitat del codi amb C ni les característiques més importants.

Del 1990 en endavant, el comitè d'ANSI, X3J16, va començar a treballar en un estàndard del C++. Fins que es va publicar l'estàndard al 1998, el C++ es va expandir espectacularment i, avui en dia, és el llenguatge preferit per desenvolupar aplicacions professionals en totes les plataformes.

## 3.2 Sintaxis del C++

### 3.2.1 Tokens

Un programa en C++ és un conjunt de caràcters que s'agrupen en components lèxics anomenats *tokens* que formen el vocabulari bàsic del llenguatge. Els components són els següents: paraules reservades, identificadors, constants, constants de cadena, operadores i signes de puntuació.

Els identificadors es poden formar amb els següents caràcters:

abcdefghijklmnopqrstuvwxyz

ABCDEFGHIJKLMNOPQRSTUVWXYZ

0123456789 \_

Els identificadors en el C++ estan formats pels noms de variables, constants etc. Han de començar amb una lletra del alfabet o amb una barra baixa. Els següents caràcters poden ser: lletres, dígitos o barres baixes. C++ distingeix entre majúscules i minúscules a l'hora de declarar identificadors. No estan permesos altres tipus de caràcter ni espais.

Podem utilitzar el nom *variable* com a identificador, i seria diferent a *Variable* amb majúscules. Podem utilitzar *variable2* , *variable\_2* i *\_variable* però no *2variable* ni *variable 2*.

En C++ hi ha unes paraules reservades que no es poden declarar com a identificadors ni es poden redefinir. Les següents paraules no es poden utilitzar en el C++ segons les regles del C++ estàndard.



asm	continue	float	new	signed	try
auto	default	for	operator	sizeof	typedef
break	delete	friend	private	static	union
case	do	goto	protected	struct	unsigned
catch	double	if	public	switch	virtual
char	else	inline	register	template	void
class	enum	int	return	this	volatile
const	extern	long	short	throw	while

Tot i així, hi ha diferents compiladors comercials de C++ que poden incloure noves paraules reservades, com el Borland, Microsoft i Sysmantec.

També el comitè ANSI, ha afegit noves paraules al C++

bool	false	reinterpretcast	typeid
const_cast	mutable	static_cast	using
dynamic_cast	namespace	true	wchart

### 3.2.2 Tipus de dades

A l'hora de declarar variables i constants, és necessari indicar el tipus de dada que utilitza al llarg del programa. Hi ha dos grups diferents de dades, els de nombres enters (int ) i els de coma flotant (float).

La diferència entre els dos, és que els de coma flotant pertanyen als racionals

La quantitat que poden arribar a emmagatzemar és la següent:

## Enters:

Tipus de dada	Valor
unsigned char	Caràcters de 8 bits 0, de a 255 (segueixen sent enters)
char	Caràcters de 8 bits, de -128 a 127 (segueixen sent enters)
short int	16 bits, de-32.768 a 32.767
unsigned int	32 bits, de 0 a 4.294.967.295
int	32 bits, de -2.147.483.648 a 2.147.483.647
long	32 bits, de -2.147.483.648 a 2.147.483.647
unsigned long :	32 bits, de 0 a 4.294.967.295
long long	64 bits, de -9.223.372.036.854.775.808 a 9.223.372.036.854.775.807
unsigned long long	64 bits, de 0 a 18.446.744.073.709.551.615

## En coma flotant:

Tipus de dada	Valor
float	32 bits $3.4 \times 10^{-38}$ a $3.4 \times 10^{+38}$ (6 dec)
double	64 bits $1.7 \times 10^{-308}$ a $1.7 \times 10^{+308}$ (15 dec)
long double	80 bits $3.4 \times 10^{-4932}$ a $1.1 \times 10^{+4932}$

### 3.2.3 Exemple d'un programa en C++

Per veure d'una manera més clara el funcionament del llenguatge C++, aquí veurem un exemple del típic programa de mostra, que escriu un "Hola món" a la pantalla.

```
#include <iostream>
using namespace std;

int main()
{
    cout << "Hola món" << endl;
    return 0;
}
```

amb el `#include`, li diem al programa que ha de buscar la llibreria per a utilitzar les funcions que conté. `"using namespace std;"` serveix per aclarir que utilitzarem funcions de les llibreries estàndard com `"iostream"` en aquest cas, i al declarar-les no s'hagi de posar `"std::cout"`, sinó `"cout"` directament.

`"int main()"`

És la funció principal del programa, aquí comença el programa a funcionar.

`"cout << "Hola món" << endl;"`

el `cout` és una funció de la llibreria `iostream` per envia dades a la sortida, en aquest cas, la pantalla. els signes `"<<"` per separar cada paraula reservada de les dades, també per separar tipus de dades diferents al enviar-les juntes. El `"endl"` és per fer un salt de pàgina"

`"Return 0;"`

Fa que el programa principal retorni 0, i així acabi el programa. També es pot deixar en blanc, i el compilador serà responsable d'agregar la sortida adequada.

`"}"` tanca la funció principal que s'havia oberta amb `"{"`.

Ara que hem vist el programa més senzill, farem un una mica més complex, que treballi amb números per veure com fer càlculs amb C++.

Imaginem que volem un programa que simplement sumi dos números , a i b.

```
#include <iostream>

using namespace std;

int main()
{
    int a = 3, b =2;
    cout << "La suma de A + B es " << a +b << endl;
    return 0;
}
```

Aquí el programa no ha canviat massa de l'anterior, com es veu, es declaren els dos nombres a i b com a int, donant-li un valor de a =3 i b=2. Després s'imprimeix en la pantalla amb el comand cout. Aquí es pot veure com a l'hora d'imprimir no només es pot posar un número, sinó que es pot posar una operació per a que l'imprimeixi directament.

A part de la sortida "cout" també hi ha funcions per a tractar amb l'entrada: "cin".

quan hi hagi un cin, el programa llegirà tota la cadena d'entrada que s'introdueixi (fins a prémer la tecla Enter). així doncs, si volguéssim introduir els valors de a i de b manualment en el programa anterior, seria de la següent forma:

```
int a, b;
cin >> a >> b;
```

És important crear les variables primer, sinó donaria error. Després, amb la funció cin, s'introdueixen els valors primer a a i després a b. Aquí els operadors ">>" estan en sentit contrari que els de "cout", ja que ara es tracta d'entrada.

### 3.2.4 Comparació If

Però els programes no són sempre tan estàtics, fent la mateixa operació cada vegada. Alguns programes necessitaran fer una operació o una altra, depenent d'alguna condició. Les comparacions es fan amb la sentència **if**

La sentència **if** compara una condició, i si es verdadera, realitza una operació, sinó, no.

Per exemple, podem fer un programa que introduint l'usuari 2 números, si són iguals ho indiqui en la pantalla, sinó, que digui que són diferents:

```
#include <iostream>
using namespace std;

int main()
{
    int a, b; //els dos enters
    cout << "Introdueix dos números" << endl;
    cin >> a >> b;
    if( a == b ) cout << "Son iguals" << endl;
    else cout << "Son diferents" << endl;
    return 0;
}
```

S'ha d'aclarir, que a l'hora de comparar amb **if**, dins la condició al ser iguals va un doble signe d'igualació ( **==** ) ja que el signe simple significa que s'assigna el valor d'un a l'altre.

Apart dels **if**, es poden utilitzar altres sentències seguits d'un **if**, con el "else if" que vindria a ser: En cas que no s'hagi complert la primera condició, si... " I s'establiria una nova condició. Per tant, si es compleix la primera, ja no es pot complir la segona. Si no es compleix la primera condició, però es compleix la segona, es fa la funció de la segona.

Dins dels ifs es poden fer nombroses comparacions, no només d'igualtat.

Nom del operador	Sintaxis
Més petit que	$a < b$
Més petit o igual que	$a \leq b$
Més gran que	$a > b$
Més gran o igual que	$a \geq b$
Diferent de	$a \neq b$
Igual que	$a == b$
Negació lògica	$!a$
AND lògic	$a \&\& b$
OR lògic	$a \parallel b$

### 3.2.5 Bucles

Un bucle, és una repetició d'una part del programa mentre es doni una condició.

Hi ha dos principals, el for, i el while. El for ve donat per tres declaracions, primer, es declara la variable, segon, es declara la condició, i últim, es dóna l'efecte. És a dir, un exemple que es repeteixi 5 vegades:

```
for ( int i = 0; i < 5; ++i )
```

Aquest for diu: Per a  $i = 0$ , es repetirà el bucle mentre la  $i$  sigui més petita que 5, i a cada passada, s'incrementarà en 1 la variable  $i$ . Els bucles for s'utilitzen per a recórrer números.

L'altre bucle és el while, que únicament té una condició, i és que es repeteixi mentre una condició sigui verdadera.

Per exemple, farem un programa que puguis escriure qualsevol programa fins que s'escrigui la paraula "sortir", que aleshores el programa es tancarà.

Per tant, dins del while posarem dos condicions. La primera, un "cin" i l'altre es compararà que la paraula introduïda no sigui "sortir". Per tant, mentre s'introdueixin paraules, i aquestes no siguin "sortir", es repetirà el codi, i seguirà demanant paraules. Quan finalment s'introdueixi "sortir", el programa escriure "Sortint del programa" i finalitzarà.

```
#include <iostream>
using namespace std;

int main()
{
    string paraula;
    while ( cin >> paraula and paraula != "sortir" )
    {
        cout << paraula << endl;
    }
    cout << "Sortint del programa" << endl;
}
```

La complexitat d'un programa augmenta a mesura que anem combinant els les diferents declaracions per anar fer operacions.

### 3.2.6 Classes

Les classes són una característica que diferencia el C++ del C. Forma part de la programació orientada a objectes ( P.O.O).

La P.O.O és un paradigma de programació que es basa en els conceptes classe i objecte.

Classe: Especificació de les característiques d'un conjunt d'objectes

Objecte: Una entitat autònoma amb una funcionalitat concreta i definida. Un objecte és una instància d'una classe.

Així doncs, podem crear qualsevol classe que faci unes funcions predeterminades amb les característiques, i després crear tants objectes com vulguem d'aquesta classe.

També es pot crear una classe i tot seguit, una segona classe derivada d'aquesta primera, que heretarà les seves característiques. Per exemple, podem crear una classe que es digui Animals, amb les característiques que vulguem, i podrem crear els objectes que vulguem. Després podem crear una altre classe, que derivi de Animals, i que es digui Mamífers. La classe Mamífers tindrà les característiques de Animals, i apart podrà tenir unes altres pròpies.

Les classes tenen especificadors d'accés, que controlen l'accés per que permetin accedir als membres de la classe, o no.

Hi ha 3 especificadors:

1. public: els membres declarats en public, seran accessibles tant des de dintre com fora de la classe.
2. Protected: els membres només són accessibles des de funcions de la pròpia classe i derivades, però no d'altres classes.
3. Private: L'accés queda únicament accessible a la funcions membre de la pròpia classe. És com la Protected, però exclouent les classes derivades.

Si es declaren membres sense especificador, per defecte seran private.

Per a veure un exemple, crearem una classe que sigui un rectangle. Aquest rectangle, tindrà la seva base i la seva altura en private. Després, tindrà dues funcions membres públiques. Una per introduir les dades i una altre per saber la seva àrea.



```

#include <iostream>

using namespace std;

class Rectangle{
private:
    int x, y;
public:
    void definir_valors( int a, int b);
    int retornar_area();
};

void Rectangle::definir_valors( int a, int b)
{
    x = a;
    y = b;
}
int Rectangle::retornar_area()
{
    return (x*y);
}

int main()
{
    int n = 4;
    Rectangle rectangle;

    rectangle.definir_valors( 6, 3);
    cout << "Area: " << rectangle.retornar_area(); << endl;
    return 0;
}

```

En el programa creem la classe rectangle, amb la seva x i y.

En public, declarem les funcions membres de “definir\_valors” i “retornar\_area”. Al declarar-les, hem de el tipus de funció que és, i els tipus de paràmetres que s'introdueixen. Un cop creada la classe, definim les funcions amb la següent estructura “NomClase::NomFunció”.

Un cop definides les funcions, podem crear el nostre objecte en la funció principal del programa. La forma de declarar un objecte, és igual que la manera de declarar qualsevol altre tipus de variable: primer escrivint el tipus, i després el nom de la variable. Així, posant `Rectangle rectangle`, estem creant el objecte “rectangle” de la classe “Rectangle”.

Per accedir a una funció d'una classe, com la de definir els valors, es fa posant el nom del objecte, un punt, i el nom de la funció. “`rectangle.definir_valors( a, b )`” i “`rectangle.retornar_àrea()`”.

### 3.2.7 Estructures

Un element que és similar en C++ a les classes, són les estructures.

Les estructures en sí són una agrupació de diferents tipus de dades sota un mateix nom.

Es declaren de la següent forma:

```
Nom_Estructura
{
    Tipus_Membre Nom_Membre;
};
```

## 4 SDL

### 4.1 Quant a SDL

La llibreria SDL ( Simple DirectMedia Layer ) és una llibreria multi plataforma que proporciona accés a l'àudio, teclat, ratolí, joystick, 3D hardware per OpenGL, i vídeo en 2D.

SDL dóna suport a molt Sistemes Operatius: Linux, Windows CE, BeOS, MacOS, Mac OS X, FreeBSD, NetBSD, OpenBSD, BSD/OS, Solaris, IRIX i QNX. Apart, el codi també conté suport per AmigaOS, Dreamcast, Atari, AIX, OSF/Tru64, RISC OS, SymbianOS i OS/2, però no estan suportats oficialment

SDL Està escrit en C, però funciona amb C++ nativament i te enllaços per a funcionar amb molts més llenguatges, incloent: Ada, C#, D, Eiffel, Erlang, Euphoria, Go, Guile, Haskell, Java, Lisp, Lua, ML, Objective C, Pascal, Perl, PHP, Pike, Pliant, Python, Ruby, Smalltalk, i Tcl.

SDL està distribuït sota la llicència GNU LGPL version 2. Aquesta llicència permet utilitzar les llibreries SDL de manera lliure en programes comercials, mentre s'enllaci amb les llibreries dinàmiques.

## 4.2 Programació amb SDL

Farem un programa molt simple amb SDL per a mostrar un exemple de com funciona amb C++. El programa en si imprimirà una imatge en la pantalla durant uns segons.

Per començar a programar amb SDL, el primer de tot serà incloure la llibreria SDL. Així com incloïem fins ara la <iostream> per escriure a la pantalla, ara també inclourem la llibreria SDL.

```
#include <SDL/SDL.h>
```

Tot seguit començarem amb la funció principal. Aquest cop, apart de ser tipus int, tindrà uns paràmetres addicionals.

```
int main( int argc, char* args[] )  
{  
    SDL_Surface* imatge= NULL;  
    SDL_Surface* pantalla = NULL;
```

Els arguments "int argc" i "char\* args[]" recullen els events amb la finestra, com per exemple, la minimització de la mateixa.

"SDL\_Surface\* imatge" és a una superfície de SDL. Com que de moment no apunten a res, estan buits amb NULL.

```
SDL_Init( SDL_INIT_EVERYTHING );  
pantalla = SDL_SetVideoMode( 640, 480, 32, SDL_SWSURFACE );  
imatge = SDL_LoadBMP( "imatge.bmp" );
```

"SDL\_Init( SDL\_INIT\_EVERYTHING );" inicia els sistemes de SDL per començar a fer ús dels gràfics.

Després s'inicia la pantalla amb "SDL\_SetVideoMode". Els paràmetres a continuació són: amplada, alçada, bits per píxel, i l'últim paràmetre "SDL\_SWSURFACE" estableix la superfície en la memòria.

Un cop la superfície de la pantalla està llesta, podem carregar la nostra imatge en la superfície "imatge" amb "SDL\_LoadBMP". Aquesta funció, però, només permet carregar imatges en .bmp.

Ara, ja només ens queda imprimir.

```
SDL_BlitSurface( imatge, NULL, pantalla, NULL );  
SDL_Flip( pantalla );  
SDL_Delay( 2000 );
```

Amb "SDL\_BlitSurface" imprimim una imatge en la pantalla, seguint els següents parametres: "imatge", superfície de la imatge que vulguem imprimir. "pantalla" superfície a on s'imprimeix. Els dos nulls són per imprimir només una part indicant les dimensions de la imatge a agafar.

SDL\_Flip( pantalla ); serveix per actualitzar la pantalla i mostri el que hi ha.  
SDL\_Delay( 5000 ); Amb això farem una pausa de 5000 milisegons. Sinó, el programa acabaria massa ràpid com per veure'l.

La imatge ja estaria impresa i el programa acabaria en 5 segons. Per acabar, s'allibera la imatge i es tanca el sistema SDL.

```
SDL_FreeSurface( imatge );  
SDL_Quit();  
return 0;
```

Una característica destacable de SDL és que el sistema de coordenades no segueix els eixos cardinals. Sinó que l'eix de la Y està invertit. Augmenta a mesura que baixa, i disminueix a mesura que va cap amunt.

## 5 Simulador d'atracció gravitatòria entre partícules

### 5.1 Creació d'una partícula amb moviment

#### 5.1.1 Part gràfica del programa

Començarem amb un programa simple, que contingui coses bàsiques, i després a mesura que vagi avançant i modificant codi i afegint funcions.

El primer que farem, serà crear una partícula, i que es mogui per la pantalla, per entendre les bases del moviment en la programació.

Començarem el programa incloent les llibreries necessàries.

```
#include <iostream>
#include <SDL/SDL.h>
#include <math.h>
```

Al principi del programa crearem la majoria de variables que necessitem al llarg del programa. Primer podem crear les característiques de la pantalla en constants.

```
const int PANTALLA_AMPLADA = 640;
const int PANTALLA_ALTURA = 480;
const int PANTALLA_BPP = 32;
```

Les superfícies a utilitzar, seran la de la pantalla, i la partícula.

```
SDL_Surface *particula = NULL;
SDL_Surface *pantalla = NULL;
SDL_Event event;
```

SDL\_Event, crea un event SDL, bàsicament servirà per a recollir les interaccions amb la finestra així com el teclat i retolí.

Aquest cop, moltes de les funcions que hem fet servir abans directament, les agruparem en funcions per a poder utilitzar-les més còmodament i sense haver de repetir tant de codi.

Per a la partícula, utilitzarem una imatge, i per a carregar-la, en comptes de carregar-la directament, crearem la nostre funció.

```
SDL_Surface *cargar_imatge( string arxiu )
{
    SDL_Surface *imatgeCargada = NULL;
    SDL_Surface *imatgeOptimitzada = NULL;

    //es carga la imatge
    imatgeCargada = LoadBMP( arxiu.c_str() );

    if( imatgeCargada != NULL )
    {
        imatgeOptimitzada = SDL_DisplayFormat( imatgeCargada );

        SDL_FreeSurface( imatgeCargada );
    }
    return imatgeOptimitzada;
}
```

En la funció es carga la imatge en una superfície, i es s'optimitza en una altre amb "SDL\_DisplayFormat( Superfície ). L'optimització aquesta serveix per a poder imprimir imatges que tinguin uns bpp diferents als de la pantalla. Si l'intentéssim imprimir directament, obtindríem uns resultats defectuosos, així que al carregar-la l'optimitzem directament, i la retornem.

Ara ja tenim la imatge carregada, però queda imprimir-la.

```
void imprimir_imatge( int x, int y, SDL_Surface* origen, SDL_Surface* desti)
{
    //rectangle per imprimir les imatges
    SDL_Rect coordenades;

    //coordenades del rectangle
    coordenades.x = x;
    coordenades.y = y;

    //imprimir

    SDL_BlitSurface( origen, NULL, destí, &coordenades );
}
```

En aquesta funció, introduint les coordenades x i y, la superfície d'origen i la de destí imprimirem una imatge. Les coordenades es passen a un rectangle "SDL\_Rect" que es posa en l'últim paràmetre de "SDL\_BlitSurface" per a indicar en quina posició de la superfície s'imprimeix la imatge.

Ara necessitem una funció per a iniciar tot el sistema SDL.

```
bool iniciar()
{
    //S'inicia tots els subsistemes SDL
    if( SDL_Init( SDL_INIT_EVERYTHING ) == -1 )
    {
        cerr << "No s'ha pogut iniciar SDL" << endl;
        return false;
    }

    //parametres de la pantalla
    pantalla = SDL_SetVideoMode( PANTALLA_AMPLADA, PANTALLA_ALTURA,
    PANTALLA_BPP, SDL_SWSURFACE );

    if ( pantalla == NULL )
    {
        cerr << "Error en ajustar parametres de la pantalla" << endl;
        return false;
    }

    //ajustar títol finestra
    SDL_WM_SetCaption( "Programa de partícules", NULL );

    //si tot s'ha iniciat be
    return true;
}
```

Bàsicament és una funció booleana que retornarà cert o fals. Per a iniciar cada apartat, comprovem si ha donat algun error amb la condició "NULL" o -1 en el cas de SDL\_init. Si alguna donés un error, la funció retornaria fals. Si tot es correcte, al final retornarà true. Això ens permetrà carregar tot i comprovar que s'ha carregat bé. Els cerr són per imprimir com el cout, i així poder identificar millor l'error en cas que n'hi hagi i " SDL\_WM\_SetCaption( "Programa de partícules", NULL );" és per a establir el títol de la finestra.



L'ultima funció que queda és la de finalitzar el sistema SDL al final.

```
void netejar()
{
    //alliberar la superfície de la imatge
    SDL_FreeSurface( particula );

    //sortir de SDL
    SDL_Quit();
}
```

### 5.1.2 Creació de la partícula

Ara ja tenim les funcions per al que és la part gràfica. Ara falta tota la part de la partícula. Per a la partícula, crearem una classe, on tindrà els seus valors i les seves funcions.

```
class Particula {
private:

    float x, y, vx, vy;

public:

    Particula();
    void entrada();

    void moviment();

    void imprimir();

    bool activat( int i );
};
```

Per a la part privada, tindrem els valors de la posició ( x, y ) i velocitat ( vx, vy );

A la part pública es declara el constructor "Particula();" que serà una funció que iniciarà l'objecte ja amb uns valors predeterminats.

Les diferents funcions de la partícula seran "void entrada();" per a tot referent al teclat i ratolí; "void moviment()" per als càlculs per moure la partícula; "void imprimir();" per a imprimir la partícula.

- **Constructor**

El constructor simplement estableix com a posició iniciar el mig de la pantalla.

```
x = PANTALLA_AMPLADA/2;  
y = PANTALLA_ALTURA/2
```

- **Entrada de la partícula.**

Aquesta funció tindrà totes les funcions per manipular la partícula amb el teclat.

La funció començarà amb la següent expressió:

```
if( event.type == SDL_KEYDOWN )  
{  
    switch( event.key.keysym.sym )  
    {  
        case TECLA: ACCIÓ; break;  
    }  
}
```

Es comprova la condició "SDL\_KEYDOWN", que comprova si alguna tecla és premuda. I després, en comptes de fer un "if" per a cada tecla, utilitzarem la funció switch. Switch agafa una variable, i podem donar una funció per als valors que vulguem. En aquest cas agafem "event.key.keysym.sym" que és l'entrada per teclat, i després es passa cada condició de cada tecla amb el "case". Les tecles venen donades amb el nom "SDLK\_NOMTECLA"

Si volguéssim moure la partícula cap a la dreta, apretaríem la fletxa de la dreta, i la condició quedaria de la següent manera:

```
switch( event.key.keysym.sym )  
{  
    case SDLK_RIGHT: vx += 2; break;
```

En aquest cas, si es premés la tecla, s'incrementaria la velocitat x en 2.

- **Moviment de la Partícula.**

En aquesta part es farà moure la partícula. Bàsicament, s'aplicarà un canvi de posició segons la seva velocitat.

Per una part tenim el moviment en el eix de les X:

```
x += vx;  
if( x < 0 || x > ( PANTALLA_AMPLADA - particula-> w ))  
{  
    x -= vx;  
    vx = -vx;  
}
```

Simplement se li suma la velocitat a la posició a cada cicle. Sent la velocitat en píxels/cicle.

Després, es limita que surti de la pantalla, comprova que si la posició d'aquesta és més petita que 0 (se surt per l'esquerra) o més gran que l'amplada de la finestra menys l'amplada de la partícula (un apart de la partícula se surt de la finestra per la dreta) es fa que en comptes de sortir, vagi en direcció contrària restant la velocitat en comptes de sumant-la, i la velocitat es posa negativa, així en comptes d'anar cap a la dreta, anirà cap a l'esquerra fins que torni a xocar. Així es pot simular el rebot amb parets. El mateix amb les Y.

Per a imprimir la partícula, utilitzarem la funció que hem creat abans per a imprimir imatges.

```
void Particula::imprimir()  
{  
    //s'imprimeix  
    imprimir_imatge( x, y, particula, pantalla );  
}
```

Amb la funció, imprimirem la imatge de la partícula, en la pantalla, en les posicions x i y de la pròpia partícula.

### 5.1.3 Funció principal del programa

Un cop ja tenim totes les funcions necessàries i la classe per a les partícules creades, començarem amb el programa principal carregant tot

```
int main( int argc, char* args[] )
{
    if (iniciar() == false)
    {
        cerr << "Error al iniciar" << endl;
        return 1;
    }

    cargar_imatge( "particula.bmp" );
```

Declarem la funció `int main` principal, amb els paràmetres necessaris, i tot seguir cridem la funció `iniciar()` que ja teníem creada. En aquest cas per a cridar-la, ho farem amb una comparació. En cas de que `iniciar()` retorni `false`, vol dir que hi ha hagut algun error. En els cas que hi hagi un error, el que farem serà utilitzar un `cerr` per escriure on hi ha hagut el error ( en aquest cas al iniciar ) i retornarem 1 per a finalitzar el programa.

Tot seguit, carreguem la nostre imatge "particula.bmp" amb la funció "cargar\_imatge()" que ja teníem. Al carregar aquesta imatge, la imatge ha d'estar en la mateixa direcció que el programa.

Ara, podem crear la nostre partícula.

```
Particula particula;
```

Ara ja tenim l'objecte "particula" de la classe "Particula". Aquesta partícula s'ha creat amb les propietats especificades en el constructor, centrada al mig de la pantalla i amb velocitat 0.

Per a poder imprimir una imatge i que hi hagi moviment, cal fer càlculs i imprimir la imatge contínuament. Per tant, necessitem un bucle while que es repeteixi tota l'estona mentre vulguem que el programa funcioni, per aconseguir-ho, posarem com a condició del while, mentre "sortir" sigui fals, es repeteixi el bucle, i quan es vulgui tancar el programa, es canviarà el valor de sortir a true, i s'acabarà el bucle.

Dins del bucle hi haurà les següents parts:

1. Llegir entrada
2. Càlculs
3. Imprimir

En llegir entrada, es llegeix l'entrada de teclat, i es mira quines tecles s'han polsat, i es fan les operacions corresponents a cada entrada.

Càlculs: Es fan tots els càlculs del programa, així com l'atracció com el canvi de posició de les partícules en el moviment.

Imprimir. S'imprimeix a la pantalla les partícules en les seves respectives posicions.

Per tant, farem el nostre bucle que es repeteixi contínuament mentre no vulguem sortir, per tant, crearem un booleà "sortir" i mentre "sortir" sigui fals, es repetirà el bucle infinitament.

```
Bool sortir = false;
```

```
while ( sortir = false )  
{
```

Ara dins del bucle, començarem a processar l'entrada, per a processar l'entrada, s'utilitza `SDL_PollEvent ( &event )`, que recull les accions que succeeixen dins del nostre event creat. Per tant, es diu al programa que mentre reculli una acció, es processa, i ho farem amb un while.

```
while( SDL_PollEvent( &event ) )
{
    if( event.type == SDL_KEYDOWN )
    {
        particula.entrada()
    }
}
```

Per tant, mentre es recullen events, si alguna tecla és premuda ( SDL\_KEYDOWN), es cridarà la funció particula.entrada, i farà les funcions d'entrada de teclat.

També dins del event, posarem la condició que si es prem la “x” de la finestra per a sortir, que es tanqui el programa, ja que el programa per si sol, no podria tancar-se amb la creu.

```
if( event.type == SDL_QUIT )
{
    sortir = true;
}
}
```

I per sortir, assignem sortir a verdader, i així el bucle ja no es repetirà més i al arribar al final i finalitzarà.

Amb això ja hauríem acabat la part d'entrada.

La part del moviment, com que totes les funcions del moviment les tenim la funció que hem creat anteriorment, n'hi haurà prou en cridar-la perquè s'executi cada vegada.

```
Particula.moviment();
```

Finalment, queda la part d'imprimir. Per a imprimir i veure que la partícula es mou, no n'hi ha prou en imprimir la imatge a cada cicle, ja que cada vegada que s'imprimeix, no es borra l'imatge anterior, i es veu com el recorregut. Si volem que es vegi la partícula sola movent-se, haurem de imprimir cada vegada tota la pantalla de blanc, i seguidament, imprimir les partícules que vulguem.

Per a pintar cada vegada la pantalla de negra, utilitzarem la funció "FillRect"

```
SDL_FillRect( pantalla, &pantalla->clip_rect, SDL_MapRGB( pantalla->format, 0, 0, 0 ) );
```

Amb això s'omple la superfície pantalla amb els colors RGB 0, 0, 0, que corresponen al negre.

Després ja només cal imprimir la partícula, i com abans, ho farem cridant la nostre funció.

```
Particula.imprimir();
```

Finalment s'actualitza la pantalla per a mostrar el contingut.

```
if( SDL_Flip( pantalla ) == -1 )
{
    cout << "impossible fer flip screen" << endl;
    return 1;
}
```

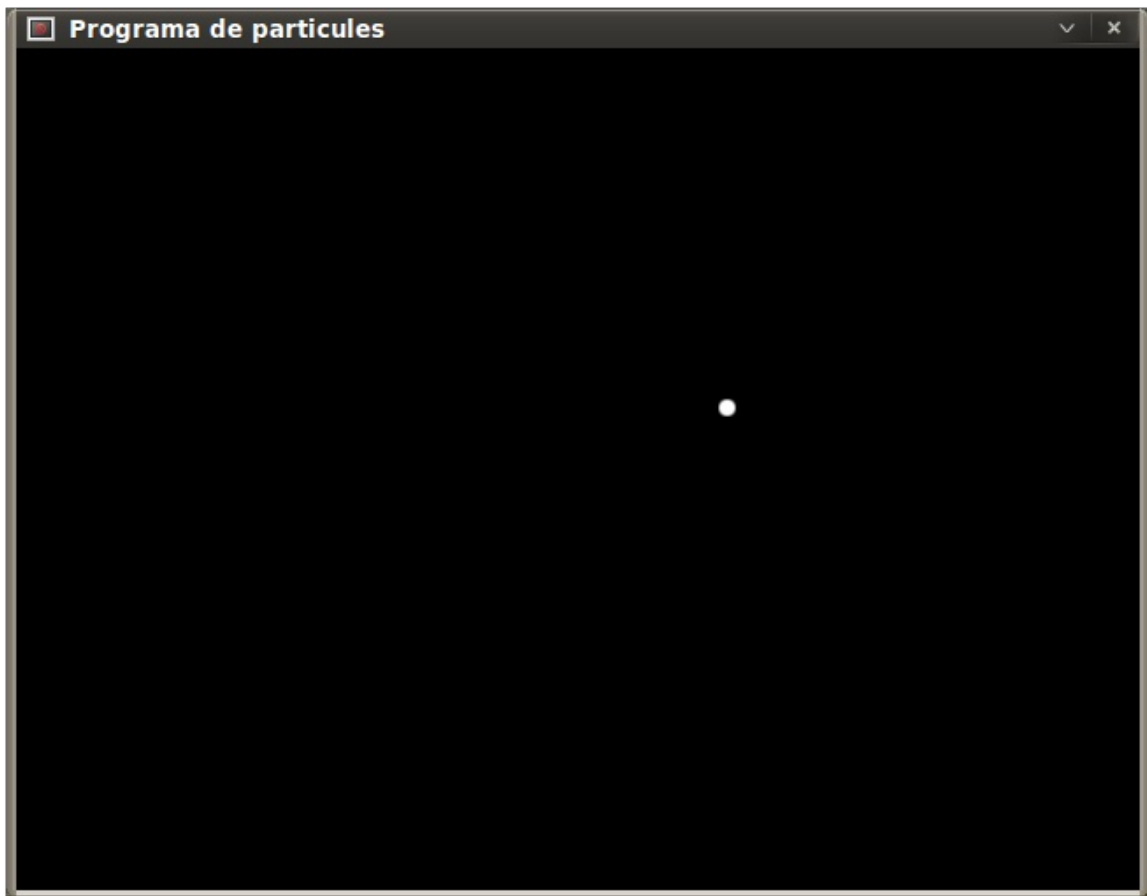
Quan s'acabi el bucle, el que farem serà cridar la funció per finalitzar els sistemes SDL i sortir del programa retornant 0.

```
    }
    netejar();

    return 0;
}
```

El primer claudàtor tanca el bucle principal, i després de finalitzar els sistemes i finalitzar el programa, es tanca la funció principal.

Ara tindríem un primer programa funcional, el resultat seria el següent.



*Captura del programa*

Tenim una finestra amb el títol "Programa de partícules", amb el fons negre i la imatge d'una partícula. Aquesta partícula la podem moure amb les tecles de direcció i anirà rebotant amb les parets.



### 5.1.3 Control de la freqüència del programa

Ja tenim el nostre primer programa finalitzat. Tenim un programa que ens crea una partícula i la podem controlar amb les tecles, i rebotarà amb la paret. El primer problema que veiem, però, és que el programa pot anar a salts, ja que treballa al màxim, intentant fer tants càlculs per segons com pot, i això no és adequat perquè la velocitat a la que va el programa és inestable i pot variar de manera dràstica d'un ordinador a un altre. Cal regular els cicles de programa per segons.

Necessitarem tenir un cronometre que calculi el temps que passa, per això crearem una classe nova.

```
class Cronometre {
private:
    int tempsInici;
    int tempsActual;
public:

    Cronometre();

    void iniciar();

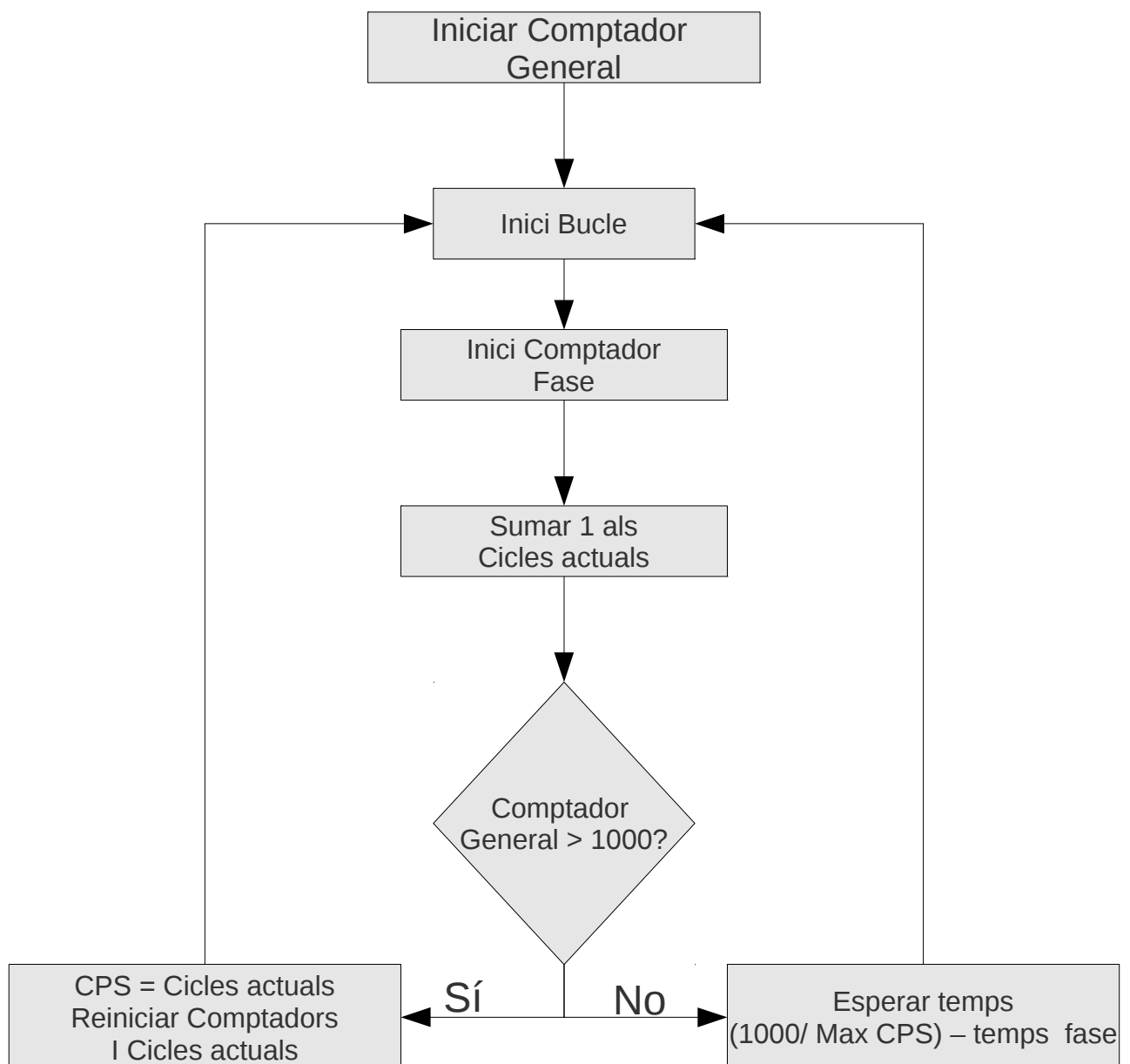
    int temps();

};
```

El cronometre constarà de dos membres privats, el temps inicial i el temps actual, ja que per a calcular el temps que passa, hem de calcular la variació del temps. I en public tindrem el constructor, la funció iniciar(), que li donarà un valor al tempsInici, i la funció temps(), que ens retornarà el temps que ha passat des de que s'ha iniciat calculant la diferència.

Com que el temps s'obté amb *SDL\_GetTicks()*, en la funció iniciar, es donarà el valor del temps en aquell moment a *tempsInici* amb *SDL\_GetTicks()*. La funció *temps()* calcularà la diferència: *tempsActual = SDL\_GetTicks() - tempsInici*; I la retornarà. Així podem calcular el temps que passa entre dos moments.

Si volem que el nostre programa vagi a 30 Cicles per segon, per a cada cicle, hauran de passar (  $1000 \text{ mil·lisegons} / 30 \text{ cicles} = 33,33 \text{ mil·lisegons}$  cada cicle. Si en cada cicle no han passat 33.33 mil·lisegons, es farà una pausa del temps que quedi per arribar a 33.33 mil·lisegons (  $1000 / 30 - \text{temps de cicle}$  ). Si això es repeteix durant un segon, i es va comptant els cicles que ha fet en una altre variable( CPSmitjana ), es poden obtenir els cicles per segon a que funciona el programa.



*Diagrama de flux del limitador de Cicles per Segon*

Ara tenim el programa a uns certs CPS, però no rebem cap informació de si està realment funcionant a aquests cicles per segon. Com imprimir text en la pantalla amb SDL és bastant complex perquè necessita llibreries auxiliars. Una manera còmode de mostrar la informació és pel títol de la finestra, amb "SDL\_WM\_SetCaption". Així, podem escriure el número de CPS al que va el programa, en el mateix títol de la finestra. Aquesta funció, però, només accepta cadenes de text. Per a poder imprimir un nombre enter, farà falta passar-ho a cadena de text, i ho podem fer amb la llibreria "sstream".

```
stringstream dades;  
dades << "CPS: << CPSmitjana;  
SDL_WM_SetCaption( dades.str().c_str(), NULL );
```

Creem un "stringstream" que es diu dades. A dades introduïm el text i les dades que vulguem passar a text, i ho acabem posant a "SDL\_WM\_SetCaption". Amb això podrem anar rebent informació de manera senzilla i tenir alguns paràmetres del programa controlats en tot moment..

## 5.2 Atracció gravitatòria de les partícules

### 5.2.1 Estructuració del programa

Ara que ja tenim el nostre programa iniciar, volem expandir-lo més, crearem l'atracció gravitatòria entre les partícules i un punt. Per altre banda, per millorar el rendiment del programa, en comptes de utilitzar una imatge en sí, utilitzarem una funció per pintar un quadrat blanc, i el resultat no variarà gaire. Per a fer l'atracció, necessitarem treballar les posicions i les velocitats, ja no com a variables, sinó com a vectors, per tenir en un mateix tipus de dada, les dues components.

Com que necessitarem tenir varies funcions per a treballar amb vectors, i la classe de la partícula s'anirà expandit, per a organitzar-ho millor i no tenir-ho tot barrejat, dividirem el programa en 3 parts.

1. Llibreria partícula. Contindrà la classe de la partícula
2. Llibreria vector: Es crearan els vectors i les seves funcions per a treballar amb ells.
3. Programa Principal. Serà el programa sencer, aprofitant les llibreries vector i partícula.

Les llibreries es guardaran en format .hpp, i després s'inclouran en el programa principal com si fossin qualsevol altre llibreria, amb `#include <nom llibreria>`

### 5.2.2 Llibreria Vector

Per a poder treballar al nostre gust amb els vectors, crearem els nostres propis. Ho farem mitjançant una estructura.

```
struct vector
{
    double x;
    double y;
};
```

Així doncs, ara tenim els nostres propis vectors, amb la component x i y corresponent. Ara podem declarar qualsevol vector, amb el nom de la estructura seguit del nom de la variable:

```
vector ExempleVector;  
ExempleVector.x = ExempleVector.y = 1;
```

Un cop creat el vector, assignem un valor (en aquest cas 1) a cadascuna de les components. Per accedir a una component d'una estructura, només cal escriure el nom de la variable que hem creat, un punt, i després el nom de la component.

Ara que ja tenim el vector, podem escriure funcions per a fer càlculs amb els vectors, tals com la suma, multiplicació etc. per a utilitzar-les més endavant.

Una funció que serà necessària, serà afegir el valor d'un vector, a un altre sumant-lo. És a dir, tenint el vector A i B, que A sigui  $A + B$ . Com el valor s'afegirà a un vector ja existent, i realment no retorna cap valor, sinó que realitza càlculs, la funció serà void, que no retorna res.

```
void SumaVectors( vector& v1, vector v2 )  
{  
    v1.x += v2.x;  
    v1.y += v2.y;  
}
```

En els paràmetres que s'entren en la funció, la primera és vector& v1. El signe "&" indica que el valor de la variable fora de la funció es modifiqui. Així, quan es cridi la funció amb aquests dos vectors, s'aplicaran els canvis de v1, al vector que s'ha introduït, i així no caldrà retornar res.

Crearem una altra funció per a multiplicar un vector donat per un número. Aquest cop, no modificarem el vector introduït, farem que el vector retorni un de nou. Com que retorna un vector, la funció serà de tipus vector.

```
vector MultiplicacioVectors( vector v1, double k )
{
    vector v2;
    v2.x = v1.x*k;
    v2.y = v1.y*k;

    return v2;
}
```

Després d'aquestes funcions molt bàsiques, necessitarem funcions per arribar a l'atracció gravitatòria. Dos funcions necessàries seran la distancia entre dos vectors, i el mòdul d'un vector.

```
vector DistanciaVectors( vector v1, vector v2)
{
    vector v3;

    v3.x = v2.x - v1.x;
    v3.y = v2.y - v1.y;
    return v3;
}
```

Distància vectors: Donats dos vectors ( v1 i v2 ), retorna un tercer vector (v3) que es la resta del segon menys el primer.

```
double ModulVector(vector v)
{

    return sqrt( (v.x*v.x) + (v.y*v.y));
}
```

El modul del vector, res més que l'arrel quadrada de les sumes de les components al quadrat.

Ara que podem obtenir el modul d'un vector amb una simple funció, podem crear una funció que crei el vector unitari.

```
vector VectorUnitari ( vector v )
{
    vector vUnitari; //vector unitari
    vUnitari.x = v.x / ModulVector(v);
    vUnitari.y = v.y / ModulVector(v);

    return vUnitari;
}
```

N'hi ha prou en dividir cada component del vector pel seu modul, i així la funció retornarà el vector unitari del introduït.

Amb aquestes funcions, ja podem crear la funció principal de l'atracció. Realment, per a les simulacions no és necessària la força d'atracció gravitatòria en si. Així que per a les atraccions calcularem directament l'atracció per a estalviar processos.

L'acceleració amb que es atret un cos amb massa 1 ( $m_1$ ) per un altre de massa ( $m_2$ ), es pot treure a partir de la formula de gravitació universal.

$$F = G \frac{m_1 \cdot m_2}{d^2} \cdot \vec{u}$$

busquem l'acceleració, i com que l'acceleració es  $a = \frac{F}{m}$

Així que la F serà  $a = G \frac{m_1 \cdot m_2}{d^2 \cdot m_1} \cdot \vec{u}$

Simplificant

$$a = G \frac{m_2}{d^2} \cdot \vec{u}$$

Així doncs, en la formula d'acceleració amb que es veu atreta una partícula de  $m_1$  per una altre de  $m_2$ , no hi interfereix la massa del primer objecte. Per exemple, l'atracció amb que es veu atreta la lluna per la terra, no importa la massa de la lluna, seria igual si peses 1kg.

Així doncs, podem crear la funció que calculi l'acceleració. Aquesta acceleració vindrà donada en un vector, i també declarem la constant G com a double.

```
double G = 6.67 * pow(10, -11); // constant de gravitació
vector AtraccioVectors ( vector v1, vector v2, unsigned long long m )
{
    vector v3; //vector de la atracció
    vector vdistancia;
    vdistancia = DistanciaVectors( v1, v2 );
    double distancia = ModulVector( vdistancia );
    v3.x = (( G * m ) / ( distancia * distancia ) ) * VectorUnitari( vdistancia ).x;
    v3.y = (( G * m ) / ( distancia * distancia ) ) * VectorUnitari( vdistancia ).y;
    return v3;
}
```

En aquesta funció s'introduirà el vector de cada partícula, i la massa del cos que l'atrau. La massa vindrà donada per unsigned long long, per emmagatzemar un nombre natural tan gran com es pugui.

vector v3; Es crea el vector de l'atracció que es retornara.

vdistancia = distanciaVectors( v1, v2 );

double distancia = ModulVector( vdistancia ); Es guarda en una variable el valor escalar de la distància, calculant el mòdul de la distància.

En les següents línies s'assigna el valor de l'atracció de cada component. En comptes d'assignar el valor multiplicant l'atracció pel vector unitari, assignarem cada component, multiplicant-la també per la component del vector unitari.

```
v3.x = (( G * m ) / ( distancia * distancia ) ) * VectorUnitari( vdistancia ).x;
```

Bàsicament el codi es pot veure que és la formula. La constant G multiplicada per la massa, entre la distància escalar al quadrat. Tot això, multiplicat per la seva component del vector unitari de la distància

Finalment, es retorna el vector resultant.



### 5.2.3 Llibreria Partícula

Com que la manera de manipular la posició i la velocitat ara es fa amb vectors, caldrà canviar una mica el programa. Així mateix, la partícula tindrà més propietats, com la massa, i més funcions, per a l'atracció gravitatòria.

En comptes de declarar  $x$ ,  $y$ ,  $v_x$  i  $v_y$ , ho farem en dos vectors separats.

```
vector posicio, v;
```

Per a la massa, un int tan gran com puguem, ja que la proba principal està en fer una simulació del sistema solar.

```
unsigned long long massa;
```

Ara en el constructor, en comptes de situar la partícula en mig de la pantalla, com que crearem varies, ens interessa que apareguin a l'atzar per a la pantalla. Per fer-ho ho farem amb `rand()`, que dona un número aleatori. I perquè aquest número quedi dintre de la pantalla, útilitzarem el signe %, que serveix per a trobar el residu d'una divisió. Per tant, fent `rand()%PANTALLA_AMPLADA`, ens donarà un nombre natural al atzar més petit que l'amplada de la pantalla. Així, aplicant tant a l'amplada com a l'altura, es pot posar una partícula en una situació aleatòria.

Amb vectors, la part del moviment se simplifica bastant, ja que ara no farà falta que reboti la partícula amb les parets de la finestra del programa. Que surti de la finestra no serà cap problema.

```
SumaVectors( posicio, v ):
```

Així doncs, per trobar l'acceleració en que una partícula 1 és atreta per una altre 2 és aquesta:

```
( AtraccioVectors( posicio, posicio2, massa2 );
```

I retornaria un vector amb l'acceleració amb que és atreta. Igual que a la posició li sumem la velocitat a cada cicle, l'acceleració la sumariem a la velocitat a cada cicle.

```
SumaVectors( v, AtraccioVectors( posicio, posicio2, massa2 ));
```

Un problema que tindria el programa es que en el cas de la velocitat i la velocitat, la posició s'incrementaria per a cada cicle el mateix. És a dir, si tenim una velocitat de 5. A cada cicle la posició s'incrementaria en 5. El que volem per a que el programa simuli a temps real, és a dir, que si la velocitat és 5 metres/segon, que s'incrementi la posició 5 metres a cada segon i no a cada cicle com actualment passa.

Per solucionar això, es pot calcular el temps que passa a cada cicle, i a l'hora de aplicar la velocitat i l'acceleració es fa en funció del temps per cicle multiplicant la velocitat pel temps que ha passat. Per la multiplicació podem utilitzar la funció ja creada "MultiplicacioVectors" que multiplica un vector amb un *double*.

```
SumaVectors( posicio, MultiplicacioVectors( v, (TempsPerCicle)) );
```

A més, com que la prova del programa serà amb els planetes del sistema solar, que la simulació treballi a un ritme igual que el real, farà que no notem cap diferència a simple vista. Per solucionar això, hauríem de fer que un segon real, equivalgués a un mes del programa, mig any, o fins i tot un any. Així, en un segon podríem observar una volta sencera de la Terra al voltant del sol. Per fer-ho, només hem de multiplicar el temps per fase per al seu valor equivalent en la unitat que vulguem.

Si volguéssim que un segon equivalgués a una hora del programa, caldria multiplicar-ho per 3600, que són els segons que té una hora. Per tant, crearem una variable "FactorVelocitat" que multipliqui al temps per cicle en funció de la velocitat a la que es vol que vagi el simulador.

```
SumaVectors( posicio, MultiplicacioVectors( v, (TempsPerCicle * FactorVelocitat)) );
```

Farem que es pugui triar diferents velocitats del programa, on un segon real equivalgui des de un segon del programa, fins a un any del programa. Així , els diferents "FactorVelocitat" seran el següents

Factor Velocitat	Equivalència
1	1 segon
3600	1 hora
84600	1 dia
1296000	15 dies
2628000	1 mes
15768000	mig any
31536000	1 any

Per a la acceleració, però, com que necessitem les dades d'altres partícules que s'han d'introduir als paràmetres de la funció, com la massa i la posició, crearem una altre funció que sigui només per a l'atracció. Aquesta funció aplicarà directament l'atracció a la velocitat.

```
void Particula::atraccio( vector posicio2, unsigned long long massa2 )
{
    SumaVectors( v, MultiplicacioVectors ( AtraccioVectors( posicio, posicio2, massa2 ),
    (TempsPerCicle * FactorVelocitat)));
}
```

Per imprimir la partícula, com hem dit abans, ja no ho farem amb una imatge, sinó pintant rectangles.

```
void Particula::imprimir()
{
    SDL_Rect ImatgeParticula;
    ImatgeParticula.x = posicio.x;
    ImatgeParticula.y = posicio.y;
    ImatgeParticula.w = 1
    ImatgeParticula.h = 1;
    SDL_FillRect( pantalla, &ImatgeParticula, SDL_MapRGB( pantalla->format, 255, 255, 255 ));
}
```

En aquest cas, es crea un rectangle SDL, amb les dimensions de l'Amplada i l'altura que serà 1, i les coordenades x i y correspondran a la seva posició.

Amb l'ultima línia, s'imprimeix el rectangle de color blanc a la pantalla.

## 5.2.4 Programa principal

En el programa principal, començarem incloent les llibreries partícula i vector. La llibreria vector la podem incloure al principi del programa, però la llibreria partícula l'hem d'incloure després d'haver declarat el event de SDL, la Superfície pantalla, i variables com FactorVelocitat i TempsPerCicle, ja que estan incloses en la llibreria, i si no es declaren abans, donarien error.

Ara per crear la partícula, no crearem només una, sinó que podem crear tantes com vulguem en forma de vector dinàmic, que són el estàndard del C++.

Crearem 10 partícules:

```
Particula particula[10];
```

I en les altres parts del programa on hi havia una funció de la partícula, per exemple, en els events que hi havia particula.entrada(). Doncs ara posarem la funció de cada partícula. Per recórrer totes les partícules utilitzarem bucles for.

```
for (int i = 0; i < NUMERO_PARTICULES; ++i)
{
    particula[i].entrada();
}
```

Així cridarem la funció entrada() de les 10 partícules, que van de la 0 a la 9.

A la part de moviment, ara també fa falta afegir la funció de particula.atraccio( vector posicio, int massa ). Podem crear un vector posició en un punt que vulguem, junt amb una massa, i al cridar aquesta funció, cada partícula es veurà atreta per aquell punt i es mourà cap a ell.

```
for (int i = 0; i < NUMERO_PARTICULES; ++i)
{
    particula[i].atraccio( PuntAtraccio, MassaAtraccio);
}
```

El vector `PuntAtraccio` i `MassaAtraccio` només son variables d'exemple per a ensenyar com es cridaria la funció. Creant un vector que sigui un punt, i introduint una massa, funcionaria.

De la mateixa manera imprimiríem les imatges, amb un bucle for totes les partícules.

## 5.3 Atracció Entre varies partícules

### 5.3.1 Atracció entre totes les partícules

La part final, serà que totes les partícules s'atraguin entre elles i aconseguir un mètode per a introduir les dades de les partícules que vulguem, així tenir una simulació amb dades reals.

El que necessitem perquè s'atreguin entre elles, és aplicar la funció d'atracció per a cada combinació entre les partícules. Si hi ha tres partícules, la primera amb la segona i amb la tercera; la segona amb la primera i la tercera. i finalment, la tercera amb la primera i la segona.

Com que la funció necessita valors com la massa i posició d'altres partícules que els seus membres són privats, accedirem a ells per noves funcions.

Crearem una funció simple que serveixi per retornar la massa i la posició. Aquestes funcions seran públiques i així es podrà accedir a la posició i massa de cada partícula.

```
vector Particula::retornarposicio()
{
    return posicio;
}

unsigned long long Particula::retornarmassa()
{
    return massa;
}
```

Així, ara podem accedir a la massa d'una partícula des de qualsevol part del programa, amb `particula.retornarmassa()`.

Ara podem aplicar la part d'atracció entre totes les partícules.

```
for (int i = 0; i < NUMERO_PARTICULES; ++i)
{
    particula[i].moviment();

    for (int j = 0; j < NUMERO_PARTICULES; ++j)
    {
        if ( j != i )
        {
            particula[i].atraccio( particula[j].retornarposicio(),particula[j].retornarmassa() );
        }
    }
}
```

Aquí hi ha dos bucles for, un dintre de l'altre, el primer recorrerà totes les partícules, i s'aplicarà la funció de cada una com ja havíem vist. I en el segon, es recorrerà per a cada una de les primeres partícules, la resta, per a fer totes les combinacions. Es crida la funció atracció de una partícula (i) i s'introdueix la massa i la posició de la resta de partícules (j).

### 5.3.2 Introducció de dades

El problema que trobem, es que les partícules que tenim es creen aleatòriament, i l'objectiu del programa es poder fer les practiques que desitgi l'usuari, per tant necessitarem crear un mètode per que l'usuari pugui dir quantes partícules vol, i quins valors inicials vol assignar-li. El mètode que utilitzarem serà llegit un fitxer extern on hi hagi escrites les dades, i llegir-les, així cada vegada que vulguem fer la mateixa simulació, n'hi haurà prou de tenir el fitxer i executar el programa.

Per a l'accés a fitxer, utilitzarem la llibreria *fstream*. L'arxiu que carregarem es dirà "config.txt". Per tant, en aquest fitxer sempre s'introduiran les dades que carregará el programa.

Per obrir un arxiu amb *fstream* utilitzarem la funció: `ifstream arxiu ( "config.txt" )`. Això ens carregarà l'arxiu "config.txt" en la variable arxiu.

Amb el operador ">>" podem passar cadenes de text de l'arxiu a una variable del nostre programa. Aquest procés ignorarà els espais i els salts de línia. Així doncs, si volem agafar tres cadenes de text diferents del arxiu, si les tenim separades per un espai o un salt de línia, podem utilitzar tres vegades l'operador ">>" i ens les agafarà perfectament.

Amb això no només podem introduir els valors de les partícules, sinó també alguns paràmetres del programa, com per exemple, Les dimensions de la finestra, el número de partícules que introduïrem, i els cicles per segon als que vulguem que treballi el programa.

Els valors que obtinguem, els posarem a cada partícula amb una nova funció, `particula.valors()`, que modificarà els valors que ja tenien, pels que introduïm. Els valors a modificar seran: posició x, posició y, velocitat x, velocitat y, massa i el diàmetre.

```
void Particula::valors( double posx, double posy, double velx, double vely,
unsigned long long massaentrada, float diametre )
{
    posicio.x = posx;
    posicio.y = posy;

    v.x = velx;
    v.y = vely;

    massa = massaentrada;

    PARTICULA_AMPLADA = PARTICULA_ALTURA = diametre;
}
```

Ara ja podem llegir els valors per a introduir-los. Primer, s'haurà de carregar l'arxiu i comprovar que el programa l'ha llegit correctament, sinó, passariem l'error amb un `cerr` i finalitzaríem el programa.

```

ifstream arxiu ( "config.txt" );
if ( !arxiu )
{
    cerr << "no es pot llegir l'arxiu" << endl;
}

```

En cas que s'hagi llegit correctament, procedim a llegir les dades. En aquest cas, llegirem les dimensions de la finestra, els cicles per segon, i el número de partícules a introduir

```

else
{
    arxiu >> PANTALLA_AMPLADA;
    arxiu >> PANTALLA_ALTURA;
    arxiu >> NUMERO_PARTICULES;
    arxiu >> CICLES_PER_SEGON;

    cout << "Pantalla X: " << PANTALLA_AMPLADA << " Pantalla Y: " <<
PANTALLA_ALTURA << " N partícules: " << NUMERO_PARTICULES << " Cicles:
" << CICLES_PER_SEGON << endl;
}

```

Un cop tenim llegides les dades, les imprimirem amb un cout per si volem comprovar que les hem introduït correctament.

Un cop ja sabem el número de partícules, podem crear el nostre vector de partícules. Abans, però, comprovem si el número de partícules a introduir sigui positiu i més gran de 0, sinó no tindria sentit, i ho indicariem amb un cerr

```

if ( NUMERO_PARTICULES <= 0 )
{
    cerr << "Número de partícules invàlid" << endl;
    return 1;
}

```

Ara sí, creem les partícules

```

Particula particula[NUMERO_PARTICULES];

```



Ara hem de llegir les dades de cada partícula, que seran: Posició (x i y), Velocitat inicial ( x i y), Massa i Diàmetre.

La manera d'estructurar les dades en el arxiu perquè sigui de la forma més clara possible serà la següent.

Valor Nom\_valor  
Valor Nom\_valor  
etc.

Així es pot veure clar, per exemple podríem introduir una partícula amb els següents valors:

40 Particula\_X  
50 Particula\_Y  
0 Particula\_Vx  
10 Particula\_Vy  
500 Particula\_Massa  
5 Particula\_Diametre

Això és important, perquè si la configuració no seguís aquesta estructura, no es llegirien bé les dades, ja que el programa està pensat per llegir el valor, i una cadena de text que serà per fer més llegible el fitxer "config.txt"

```
arxiu >> posx >> restes;  
arxiu >> posy >> restes;  
arxiu >> velx >> restes;  
arxiu >> vely >> restes;  
arxiu >> massa >> restes;  
arxiu >> diametre >> restes;
```

Així doncs, agafarem cada variable, i amb un "string" agafarem les cadenes de text que són només per aclarir.

És important, que cada tipus de valor correspongui amb el de la variable que utilitzem. Aquest procés per introduir les dades doncs, el repetirem amb un bucle "for" per a totes les partícules.

La funció al final quedaria així.

```
for (int i = 0; i < NUMERO_PARTICULES; ++i )
{
    double posx, posy, velx, vely;
    unsigned long long massa;
    float diametre;
    string restes;
    arxiu >> posx >> restes;
    arxiu >> posy >> restes;
    arxiu >> velx >> restes;
    arxiu >> vely >> restes;
    arxiu >> massa >> restes;
    arxiu >> diametre >> restes;
    cout << i
    << ": Pos X: " << posx
    << " | Pos Y: " << posy
    << " | Vel X: " << velx
    << " | Vel Y: " << vely
    << " | Massa: " << massa
    << " | Diametre: " << diametre << endl;
    partícula[i].valors( posx, posy, velx, vely, massa, diametre );
}
```

Ja només cal tancar el fitxer que estàvem llegint.

```
arxiu.close();
```

Ara el programa estaria casi acabat, només hi ha un problema amb les dades, i és les unitats.

Quan introduïm les dades com la posició, velocitat i massa, haurien d'estar en les unitats del SI, ja que concorda així amb les unitats de la  $G(6,6742(10) \times 10^{-11} \text{ m}^3/\text{kg} \cdot \text{s}^2)$  I si utilitzem les unitats del sistema internacional, tindríem números excessivament grans que no cabrien en les variables, ja que per exemple el tipus unsigned long long, només permet un número fins a 18.446.744.073.709.551.615.

Per a la introducció més adient de les dades, hauríem de permetre introduir la constant  $G$  al usuari amb les unitats que vulgui, i així les demés dades al estar en unes altres unitats, poden tenir uns valors més adients.

Si volguéssim introduir la massa del sol en unitats del sistema internacional, la seva massa seria de  $1.99 \cdot 10^{30}$  kg, número que sobrepassa amb diferència el límit. El mateix passa amb la distància, que si hi ha nombres molt grans, no apareixerien les partícules en la pantalla. Si volguéssim representar el sol i la terra, estan separats per  $1.5 \cdot 10^8$  m. Si fem un canvi de la G:  $6.6742 \cdot 10^{-11} \frac{m^3}{kg \cdot s^2}$

a G:  $6.6742 \cdot 10^{-26} \frac{Gm^3}{Pg^1 s^2}$ , és a dir, treballar en Gigametres ( $10^9$  metres) en comptes de metres, i en Petagrams ( $10^{15}$  grams) en comptes de Kilograms ( $10^3$  grams) podem aconseguir les següents dades:

$$\text{Massa del sol} = 1.9891 \cdot 10^{30} kg \cdot \frac{1 Pg}{10^{12} Kg} = 1.9891 \cdot 10^{18} Pg$$

$$\text{Distància Terra-Sol} = 149,597 \cdot 10^9 m \cdot \frac{1 Gm}{10^9 m} = 149.597 Gm$$

En la distància, podríem agafar aquest valor "149.597" i imprimir el Sol i la Terra i demés planetes amb una distància a escala més fàcilment.

Per a què es pugui modificar la G, n'hi haurà prou en crear una variable que representi l'exponent de la G, en aquest cas serà -26.

```
int GExp = -26
double G = 6.6742* pow(10, GExp);
```

Aquí es quan es defineix, en la llibreria partícula, que es on es fa ús de la G per primer cop, però en la llegida del arxiu, podem agafar aquesta dada, i tornar a definir la G amb el nou exponent, així l'usuari podrà fer la simulació amb les unitats de la G que vulgui.

Ara el programa estaria bastant complet. L'usuari pot introduir qualsevol número de partícules, i es podria veure com es mouen per l'atracció gravitatòria entre elles.

## 5.4 Millorant el programa

### 5.4.1 Dibuixar trajectòria

Ara que el programa és funcional, el que aniria bé és fer el programa més útil de cara a l'usuari en la representació de la simulació. Una funció que es pot afegir, és que el programa dibuixi les trajectòries que van descrivint les partícules. La tasca és senzilla.

Per imprimir, les que es fa és pintar tota la pantalla de negra i després la partícula, per a que no es vegi repetida. Si ignorem la part de pintar la pantalla, a cada cicle, es dibuixarà la partícula, i com el procés es fa varies vegades per segon, deixarà el rastre del moviment que fa. Si a l'hora de pintar cada vegada la pantalla de negra es va una comprovació d'un booleà, si s'està dibuixant la trajectòria, es podrà dibuixar la trajectòria a gust de l'usuari en prémer una tecla.

```
if (DibuixarTrajectoria == false)
{
    SDL_FillRect( pantalla, &pantalla->clip_rect, SDL_MapRGB( pantalla->format, 0,
0, 0 ) ); }

```

A la part d'events, quan es comprova si una tecla és premuda, es posa la condició que si es prem la tecla t, si no s'estava dibuixant la trajectòria, que es dibuixi, i si ja ho estava fent, que deixi de fer-ho.

```
case SDLK_t: if (DibuixarTrajectoria == true) DibuixarTrajectoria = false; else
DibuixarTrajectoria = true; break;

```

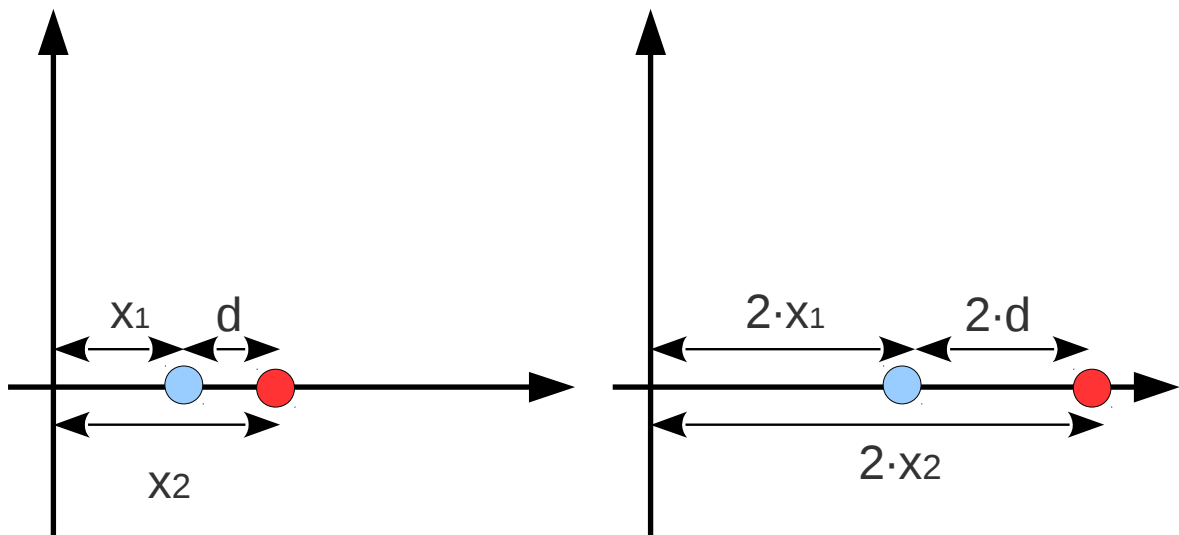
### 5.4.2 Posicionament de les partícules

També sobre la part gràfica, caldria millorar la manera d'imprimir les partícules. Actualment, s'imprimeixen en el punt que tinguin, i per visualitzar-ho bé caldria centrar-ho al mig. També, caldria poder fer zoom al nostre gust, per veure-ho de més aprop o més lluny. Per això haurem de millorar la funció de imprimir.

Per a centrar-ho tot, agafarem la meitat de la pantalla com a punt de referència, i imprimirem totes les posicions respecte el punt de referència, no respecte l'origen de coordenades. Per exemple, podem considerar la posició (4, 8). La posició està sumada al seu punt de referència, que al ser l'origen de coordenades, és el 0,0 (0+4, 0+8). Si agaféssim el nostre punt de referència el (5, 5), la posició seria (5+4, 5+8) = (9, 13). Si això ho apliquem a totes les partícules, totes es veuran desplaçades cap al punt de referència, i si alguna estava en la posició (0, 0), al agafar el punt de referència el centre de la pantalla, aquella partícula quedarà al mig de la pantalla.

Un cop dit això, podem fer un efecte de zoom, si apart de sumar el punt de referència, multipliquem la posició de cada partícula per les vegades que vulguem fer zoom.

Si tenim dues partícules, amb les respectives posicions (0, 3) i (0, 5), si volem ampliar el doble, multiplicaríem per 2, i quedarien així les posicions (0,6) i (0,10). Les distàncies respecte al punt de referència són ara dues vegades més gran, i entre elles també són dues vegades més gran. Si volguéssim allunyar la imatge, multiplicant per 0.5, aconseguiríem el mateix efecte però disminuint les distàncies.



*Explicació del zoom en les posicions*

Aplicant les dues funcions a la vegada, la funció d'imprimir quedaria de la següent manera:

```
void Particula::imprimir( vector PuntReferencia, float Zoom)
{
    SDL_Rect ImatgeParticula;
    ImatgeParticula.x = (PuntReferencia.x + (posicio.x * Zoom));
    ImatgeParticula.y = (PuntReferencia.y + (posicio.y * Zoom));
    if ( PARTICULA_AMPLADA * Zoom > 1 ){
        ImatgeParticula.w = PARTICULA_AMPLADA * Zoom;
        ImatgeParticula.h = PARTICULA_ALTURA * Zoom;
    }
    else
    {
        ImatgeParticula.w = 1;
        ImatgeParticula.h = 1;
    }
    SDL_FillRect( pantalla, &ImatgeParticula, SDL_MapRGB( pantalla->format,
255, 255, 255 ));
}
```

El zoom també s'aplica al diàmetre de la partícula, però en cas de que el diàmetre final resulti més petit que 1, s'hauria de seguir imprimint com a 1, ja que sinó no es veuria.

### 5.4.3 Separació de la freqüència dels càlculs i gràfica

Un altre aspecte important a millorar, es el rendiment del programa. Fins ara, podíem limitar els càlculs per segon que feia el programa, però els cicles per segon anaven igual que els fotogrames per segon, és a dir, que a cada cicle, s'imprimia tot, i això consumeix molts recursos i limita els cicles màxims en que pot anar el programa. Per tenir un bon rendiment, hem de separar la freqüència de càlculs amb la gràfica.

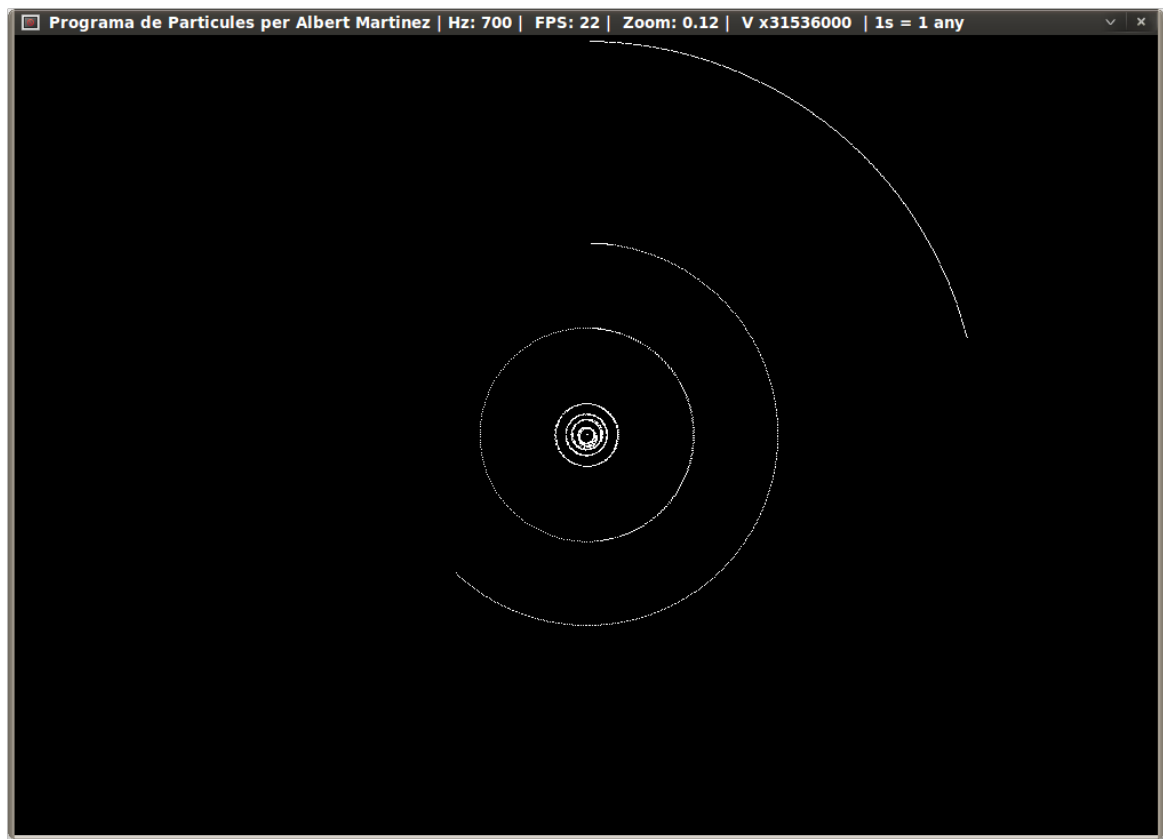
Igual que teníem abans un comptador que calculava cada segon que passava, i un comptador que calculava el temps entre càlculs, aquí necessitarem un tercer comptador, per calcular el temps entre cada vegada que s'imprimeix. Per regular els cicles per segon, es feia una pausa del temps que havia de passar entre cicle. Aquí amb la part gràfica, més que pauses, imprimirem nomes cada X vegades. Si volem que el programa ens vagi a 25 fotogrames per segon (FPS), el temps que ha de passar entre cada impressió en milisegons és de  $1000/25$ . Per tant, si imprimim cada  $1000/25$  segons, i reiniciem el comptador, i tornem a contar fins a  $1000/25$ , tindrem la regulació dels FPS que volem. Per a comprovar els FPS a què va el programa, farem igual que amb el cicles per segon, cada vegada que s'imprimeixi, incrementarem en 1 una variable. Quan hagi passat un segon, comprovarem quantes vegades s'ha incrementat i així obtenim els FPS.

## 5.5 Simulació del Sistema Solar

La prova principal que faré amb el programa ja creat, serà fer una simulació dels planetes del sistema solar. Introduint les dades, i comprovant que fan òrbites al voltant del sol, i que els períodes d'aquests són similars als reals.

Les dades introduïdes no corresponen del tot amb la realitat, ja que les velocitat les he calculat suposant que fan òrbites circulars, ja que la seva excentricitat és

molt petita. La velocitat l'he calculada amb la següent fórmula. 
$$v = \sqrt{\frac{G \cdot M}{r}}$$



*Captura del programa simulant el sistema solar*

Aquí es pot observar una captura del programa, funcionant a 700hz, 22 fps, un zoom de 0.12 i a la velocitat d'un any/segon real. En el programa es veu les òrbites que descriuen els diferents planetes. Tots els planetes es veuen de la mateixa mida, perquè a escala, mesurarien menys d'un píxel, cosa que és impossible d'imprimir.



Mesurant amb un cronòmetre i observant les trajectòries del programa, he pogut calcular el període dels planetes del meu programa i comparar-los amb el període real dels mateixos planetes.

Planeta	Període del programa	Període real
Mercuri	88.5 dies	87.97 dies
Venus	221.4 dies	224.7 dies
Terra	365.4 dies	365.256 dies
Mart	1,85 anys	1,882 anys
Júpiter	11.82 anys	11.86 anys
Saturn	28.4 anys	29.46 anys
Urà	83.2 anys	84.1 anys
Neptú	161.6 anys	164.6 anys

Realment, els períodes del programa són molt acurats tenint en compte la manera en que les dades han estat mesurats.

## 6 Conclusions

La prova principal que he fet ha estat la dels planetes del Sistema Solar i el Sol, per posar en pràctica el meu programa amb un exemple real. Amb un cronòmetre he analitzat els períodes dels planetes del programa i els he comparat amb els reals, i en general, crec que els resultats són bastant satisfactoris, ja que són molt aproximats.

Les principals causes en la variació dels resultats del programa respecte els reals són:

1. El fet que es simuli els moviments amb números molt grans i pocs càlculs per segon. En varis casos, s'ha hagut de posar la velocitat a 1 any/segon, es a dir, que un any del programa equivalgui a un segon real, i això fa s'acumulin marges d'error.
2. La poca precisió al mesurar el temps, ja que es van fer a vista comprovant les trajectòries amb un cronòmetre.
3. Les dades de les velocitats dels planetes estan calculades suposant que fan orbites circulars en comptes d'el·líptiques.
4. Hi ha un problema per controlar freqüències properes a 1000, ja que la unitat mínima de mesura del temps es 1 milisegon.

Varies solucions a aquests problemes poden ser:

- Treballar a una freqüència tan gran com sigui possible
- Mesurar les dades del simulador afegint una funció que ho faci, en comptes de a simple vista amb un cronòmetre
- Agafar dades inicials més aproximades a la realitat
- Utilitzar alguna llibreria auxiliar per a treballar amb números tan grans
- Millorar l'apartat del control de la freqüència.

El programa compleix la seva funció, ja que ha pogut simular el Sistema Solar a partir de les dades inicials dels planetes, donant orbites molt similars a les reals.

A més, amb aquest programa, variant les dades inicials es poden fer moltes més simulacions diferents a gust de l'usuari.

Per a fer el programa, vaig començar aprenent, gràcies a un curs de C++ de l'UPC d'una setmana i a un tutorial de SDL de lazyfoo.net. Mentre vaig anar adquirint coneixements, anava fent proves senzilles. Un cop tenia suficients coneixements, vaig començar a desenvolupar el meu programa, partint d'un molt simple, i anar afegint les funcions que necessitava i modificant-les.

Un problema que he tingut, és que per a fer el programa, no tenia ningun model a seguir en cas de dubte, i si alguna part em donava problemes, moltes vegades havia de resoldre'ls sense poder consultar-ho i tardava en solucionar-ho.

En general estic molt satisfet amb el treball, perquè no només he aconseguit el meu objectiu, sinó que també he pogut aprendre molt sobre la programació, que és un tema que m'interessa i molt i tinc pensat seguir estudiant en el futur.

Vull donar les gràcies a la meva tutora, per haver-me permès dur a terme aquest treball amb bastanta llibertat, sabent que al començament no hi havia ni un objectiu molt clar, ni estava gaire relacionat amb el departament assignat. També per haver-me apuntat al curs de l'UPC sense el qual, aquest treball hagués estat molt més complicat, i per ajudar-me a estructurar millor el treball i l'objectiu.

## 7 Bibliografia

### Llibres/Recerca:

- Fundamentos de Programación: Guía de sintaxis del lenguaje c++, Luis Aguilar.
- Manual básico de Programación en C++ , Apoyo a la Investigación C.P.D. ,Servicios Informáticos U.C.M.

### Internet:

- [http://lazyfoo.net/SDL\\_tutorials/index.php](http://lazyfoo.net/SDL_tutorials/index.php)
- <http://www.cplusplus.com/info/>
- <http://es.wikipedia.org/wiki/Programacion>
- <http://es.wikipedia.org/wiki/C%2B%2B>
- <http://dis.unal.edu.co/~fgonza/courses/2003/poo/c++.htm>
- <http://www.tecnun.es/asignaturas/Informat1/AyudaInf/aprendainf/cpp/basico/cppbasico.pdf>
- <http://www.nebrija.es/~abustind/Informatica/Metodologia/Archivos.pdf>
- <http://www.xtec.es/~rmolins1/solar/cat/planetes.htm>