EL PROBLEMA DEL MILENIO

Roberto Charreton Kaplun

1 DE AGOSTO DE 2019

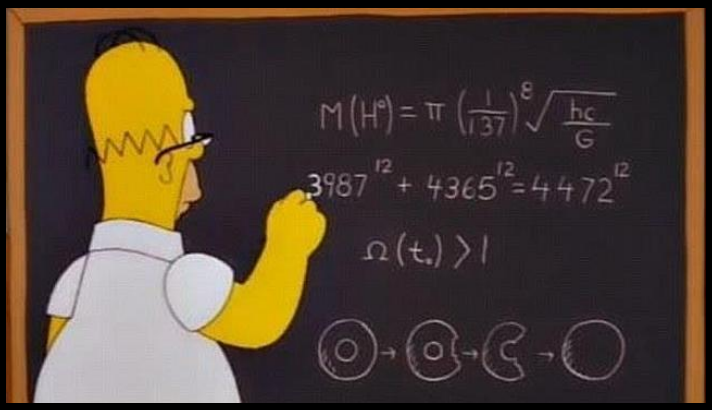
UNIVERSIDAD DE ARTES DIGITALES

Análisis y Algoritmos

INTRODUCCIÓN

¿Qué es un problema del milenio y que intenta resolver?

Los problemas del milenio son siete problemas matemáticos cuya resolución sería premiada, según anunció el Clay Mathematics Institute en el año 2000, con la suma de un millón de dólares cada uno.



Un científico asegura que la fórmula que aparece en un episodio de 1998 se acerca a la resolución de uno de los problemas 14 años después…

La lista contiene representación de todas las grandes áreas de la matemática: álgebra, geometría, teoría de números, análisis, física matemática, ... Pero también es sintomática la presencia de un problema que habría que catalogar más como de informática teórica que como de matemáticas: el problema de "P versus NP" cuya resolución (si es en el sentido P=NP) pondría en entredicho nuestras ideas de qué es un buen o un mal algoritmo, o de qué es un problema fácil o difícil de resolver por ordenador.

En futuras paginas se buscará resolver uno de estos problemas o acercase lo más posible a una solución factible del problema.

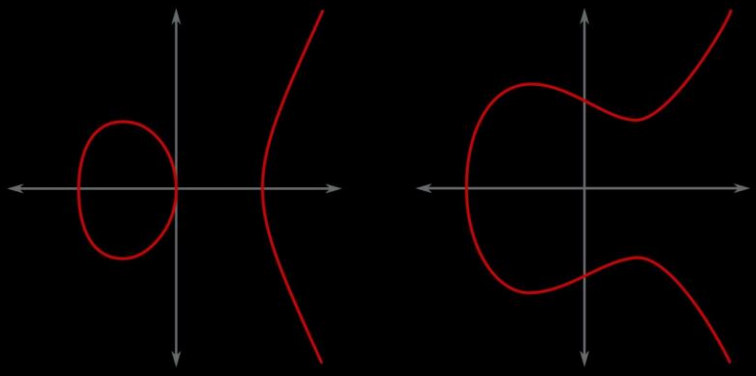
PROBLEMÁTICA

La conjetura de Birch y Swinnerton-Dyer, que une geometría algebraica y teoría de números, pide estudiar las soluciones racionales a ecuaciones que definen una curva elíptica. A principios de la década de 1960, Peter Swinnerton-Dyer usó la computadora EDSAC del laboratorio de informática de la universidad de Cambridge para calcular el número de puntos módulo p (denotado por Np) para un número largo de primos p sobre curvas elípticas cuyo rango era conocido. De esos resultados numéricos Bryan Birch y Swinnerton-Dyer conjeturaron que Np para una curva E con rango r obedecía una ley asintótica donde C es una constante.

$$\prod_{p \leq x} \frac{N_P}{p} \approx C log(x)^r \qquad cuando\ x \to \infty$$

Primera formula presentada sobre la conjetura de Birch y Swinnerton-Dyer

A este problema se le conoce como la conjetura de Birch y Swinnerton-Dyer, que une geometría algebraica y teoría de números, pide estudiar las soluciones racionales a ecuaciones que definen una curva elíptica.

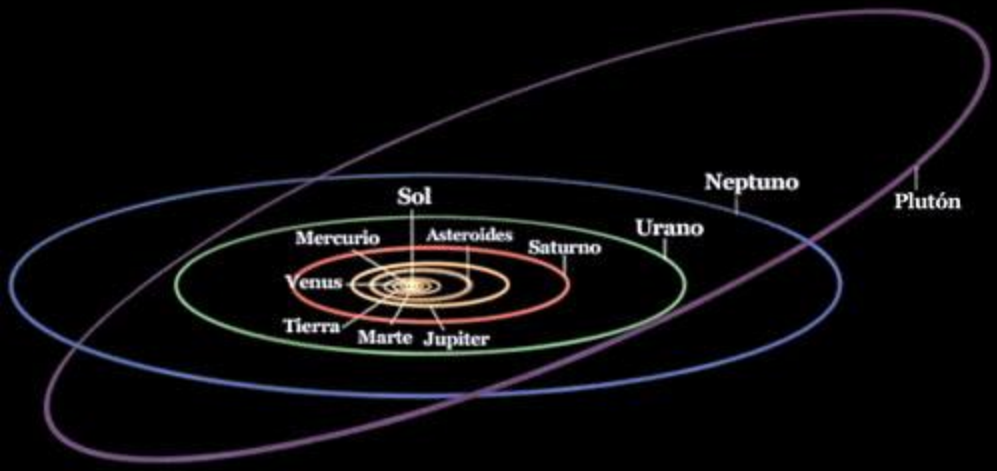


Representación grafica de la conjetura de Birch y Swinnerton-Dyer sobre curvas elípticas.

PLANTEAMIENTO DE SOLUCIÓN

Tras analizar el problema con el profesor Héctor Manuel Gómez Gutiérrez, se llegó a la conclusión de que se tomaran los siguientes temas en particular para intentar encontrar una solución:

- Movimiento circular
- Velocidad tangencial
- Aceleración Centrípeta
- Ecuación de la elipse



Cada tema seleccionado ha sido estudiado y analizado con el fin de realizar los cálculos necesarios para poder medir las distancias entre los planetas y el sol, con esto se podrá calcular el afelio y el perihelio, de tal modo que podamos calcular la velocidad tangencial de los planetas para posteriormente realizar la ecuación de la elipse y obtener datos concisos de cada planeta en un tiempo determinado N.

PLANTEAMIENTO DE SOLUCIÓN

Características del proyecto

Para el desarrollo del problema planteado previamente se tomarán a consideración los siguientes temas y herramientas que serán de utilidad para la implementación del proyecto.

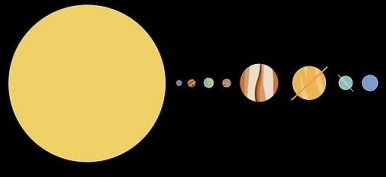
- Conocimiento en física aplicada a movimiento circular y uniforme
- Conocimiento en aceleración centrifuga y centrípeta.
- Conocimiento y aplicación de la ecuación de la elipse
- Uno de plataformas para registro de gran cantidad de datos (Se tomo Google Excel por su facilidad de manejo en equipo).
- Uso de la herramienta de visual studio.
- Conocimiento en el lenguaje c++.
- Conocimiento de la API de SFML.
- Uso de la lógica matemática para la implementación de ecuaciones.
- Conocimiento de estructura de datos.
- Conocimiento de red – black trees.
- Conocimiento en arboles binarios.

Objetivo principal

Llegar a una solución cercana al problema de la conjetura de Birch y Swinnerton-Dyer, con ayuda de los temas investigados y representando los resultados de una manera gráfica con la ayuda de una API (SFML) y programada en c++.

Objetivo secundario

Realizar un algoritmo que nos permita visitar y analizar de una manera eficiente los datos los planetas del sistema solar por medio de los conocimientos obtenidos en la clase de análisis y algoritmos, basándonos principalmente en arboles binarios, Red – Black Trees y estructura de datos.



FUNDAMENTOS DE LA SOLUCIÓN

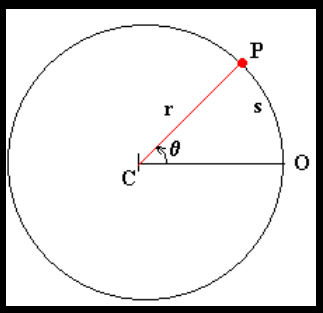
Para realizar un análisis practico de cómo se llegó a la solución cercana de las elipses, se tiene que explorar la teoría física y matemática de los cálculos a realizar.

Movimiento circular

Se define movimiento circular como aquél cuya trayectoria es una circunferencia. Una vez situado el origen O de ángulos describimos el movimiento circular mediante las siguientes magnitudes.

Posición Angular:

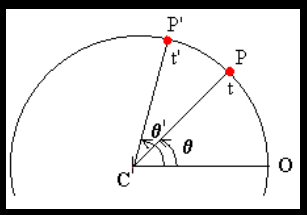
En el instante t el móvil se encuentra en el punto P. Su posición angular viene dada por el ángulo q, que hace el punto P, el centro de la circunferencia C y el origen de ángulos O.



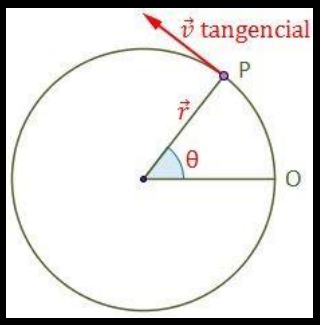
El ángulo q, es el cociente entre la longitud del arco s y el radio de la circunferencia r, $q = s/r$. La posición angular es el cociente entre dos longitudes y, por tanto, no tiene dimensiones.

Velocidad angular:

En el instante t' el móvil se encontrará en la posición P' dada por el ángulo q '. El móvil se habrá desplazado $Dq = q' - q$ en el intervalo de tiempo $Dt = t' - t$ comprendido entre t y t'.



Velocidad tangencial (universoformulas, 2019)

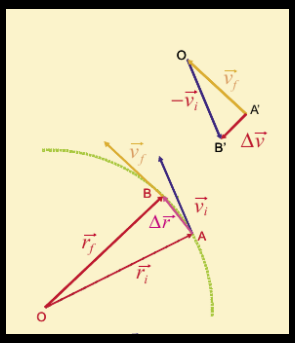


La velocidad tangencial es igual a la velocidad angular por el radio. Se llama tangencial porque es tangente a la trayectoria.

La velocidad tangencial es un vector, que resulta del producto vectorial del vector velocidad angular (ω) por el vector posición (r) referido al punto P.

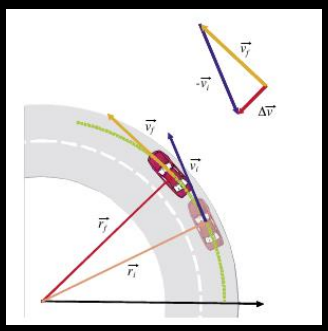
$$v = \omega \cdot r = \frac{2\pi}{T} \cdot r = 2\pi \cdot f \cdot r$$

Aceleración centrípeta (fisic, 2019)



En un movimiento circular cualquiera, la aceleración puede tener una componente en dirección tangencial a la circunferencia y otra componente en dirección radial y dirigida hacia el centro de la trayectoria. A la primera se le llama aceleración tangencial y a la segunda, aceleración centrípeta.

La aceleración tangencial se manifiesta como un cambio en el módulo de la velocidad tangencial, mientras que la aceleración centrípeta aparece como un cambio en la dirección y sentido de la velocidad.



En un movimiento circular uniforme, debido a que el módulo de la velocidad tangencial es constante, solo existe una aceleración que cambia la dirección y el sentido de la velocidad, es decir, la aceleración centrípeta.

El cambio del vector velocidad tangencial apunta hacia el centro de curvatura, al igual que la aceleración centrípeta.

Si se considera el cambio de velocidad, $\Delta v = v f - vi$, que experimenta un móvil en un pequeño intervalo de tiempo Δt, se ve que Δv es radial y está dirigido hacia el centro curvatura. La aceleración, por lo tanto, también tiene esa dirección y sentido, y por eso se denomina aceleración centrípeta.

Ecuación de la elipse

La elipse se define como una línea curva cerrada tal que la suma de las distancias a dos puntos fijos, F y F', llamados focos, es constante. Donde $d\,(P,F)$ y $d\,(P,F')$ es la distancia de un punto genérico P al foco F y al foco F' respectivamente.

$$\boldsymbol{d\,(P,F)\,+\,d\,(P,F')\,=2\cdot a}$$

Elementos de la elipse (Aga, 2019)

- Centro: Es el punto de intersección de los ejes. Es, además, centro de simetría.
- Eje principal o focal: Es el eje en el que se encuentran los focos. Es un eje de simetría.
- Eje secundario: Es el eje perpendicular al eje principal, mediatriz del segmento que une los focos.
- Vértices: Puntos de intersección de la elipse con los ejes.
- Distancia focal: Distancia entre los focos. Su longitud es $2 \cdot c$.
- Semi - distancia focal: Distancia entre el centro y cada foco. Su longitud es c.
- Semieje mayor o principal: Segmento entre el centro y los vértices del eje principal. Su longitud es 'a'.
- Semieje menor o secundario: Segmento entre el centro y los vértices del eje secundario. Su longitud es b y cumple: $b = \sqrt{a^2} - b^2$
- Radio vectores: Cada punto de la elipse cuenta con dos radio vectores que son los segmentos que unen dicho punto a cada uno de los focos. Para un punto $P\ (x, y)$ se cumple que $d\ (P, F)\ =\ a\ - e \cdot x$ y $d\ (P, F')\ =\ a + e \cdot x$

Desarrollo de teoría para la implementación de código

Después de realizar los cálculos físicos y matemáticos sigue implementar una fórmula que permita representar los datos en código.

$$\sqrt{1} - \frac{(posX)^2}{circleRadius^2} * (pharahelius \,||\, Aphelius)$$

Esta es la fórmula final que es la base para el funcionamiento de todo el programa, sin la misma ningún planeta generado en la API se desplazará y permanecerá inerte.

¿Que se tuvo que realizar para llegar a esta fórmula?

Afelio

Es el punto más distante de la órbita de un determinado planeta respecto al Sol. Esto quiere decir que, cuando un cierto planeta se encuentra en su afelio, está ubicado lo más alejado del Sol que llegará a estar mientras orbita.

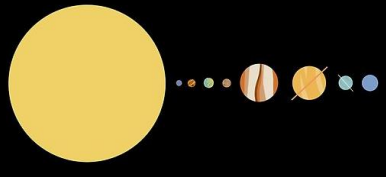
Perihelio

Es el punto en el cual un objeto celeste o un objeto que esté en el sistema solar que se encuentra girando alrededor del sol está a la mínima distancia de él.

Radio del planeta

Se calculo el radio del planeta, esto con los datos almacenados en la tabla de Google Excel, expresando entonces:

$$circleRadius = \frac{(Aphelius + Parahelius)}{2}$$



FUNDAMENTOS DE LA SOLUCIÓN

Velocidad

El cálculo de la velocidad actual está conformado por la multiplicación de esta, con la división de la magnitud entre la posición del planeta menos la posición del sol y el radio calculado previamente.

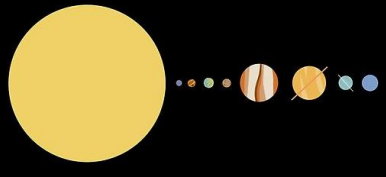
$$currentSpeed = currentSpeed * \frac{Magnitude(posPlanet - posSun)}{circleRadius}$$

A esta misma fórmula se le puede agregar una dirección, permitiendo de esta manera ejercer el movimiento, expresándolo de la siguiente manera:

$$posX = currentSpeed * \frac{Magnitude(posPlanet - posSun)}{circleRadius} * dirX$$

Al inicio de la explicación se mostró la formula final obtenida, puesto que el valor obtenido sirve para desplazarse en la aplicación, esta se atribuye a ser la posición en 'Y', esta fórmula se expresa de la siguiente manera:

$$posX = \sqrt{1} - \frac{(posX)^2}{circleRadius^2} * (pharahelius \,||\, Aphelius) * dirX$$



IMPLEMENTACIÓN

La implementación del proyecto se dividió en cuatro fases:

- Lógica del movimiento de los cuerpos
- Desarrollar estructura de datos que almacene la información de los cuerpos y que se comuniquen entre sí.
- Implementación de estructura desarrollada.
- Realizar búsqueda de planetas y lunas en un momento determinado.

Las primeras dos fases ya fuero implementadas y mostradas en las paginas anteriores, por lo que no se explayara más del tema en futuras páginas.

Implementación de estructura desarrollada

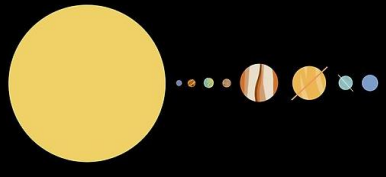
Se tomó la decisión de desarrollar e implementar la teoría enfocada en la API de SFML ya que es una biblioteca que ofrece recursos de una manera intuitiva y rápida, en comparación con otras herramientas más rebuscadas como Unity o Unreal Engine. Pese que estas dos últimas herramientas son las más útiles en el mercado, no nos servían del todo ya que tenían más cosas de las que se necesitaban para el proyecto a desarrollar.

Bases para desarrollar

El implementar una estructura específica para un proyecto nunca es sencilla por lo que se tomó a consideración un proyecto diseñado para IA, que serviría como base para la implementación de este proyecto.

MathCHK

Se tomo esta decisión ya que el proyecto ya contaba con una pequeña librería matemática que facilitaría las cosas a la hora de realizar cálculos vectoriales y poder implantar agentes de una manera sencilla.



IMPLEMENTACIÓN

La librería matemática incluye las siguientes operaciones de vectores:

- Producto punto.
- Magnitud.
- Normalización.
- Proyección.
- Resta.
- Suma.
- Multiplicación.
- Interpolación.

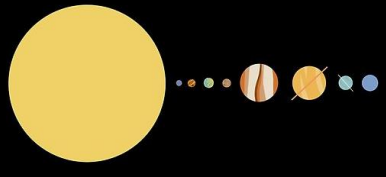
No hace falta explicar cada una de las operaciones necesarias, pero sirve de contexto para el código futuro.

Text

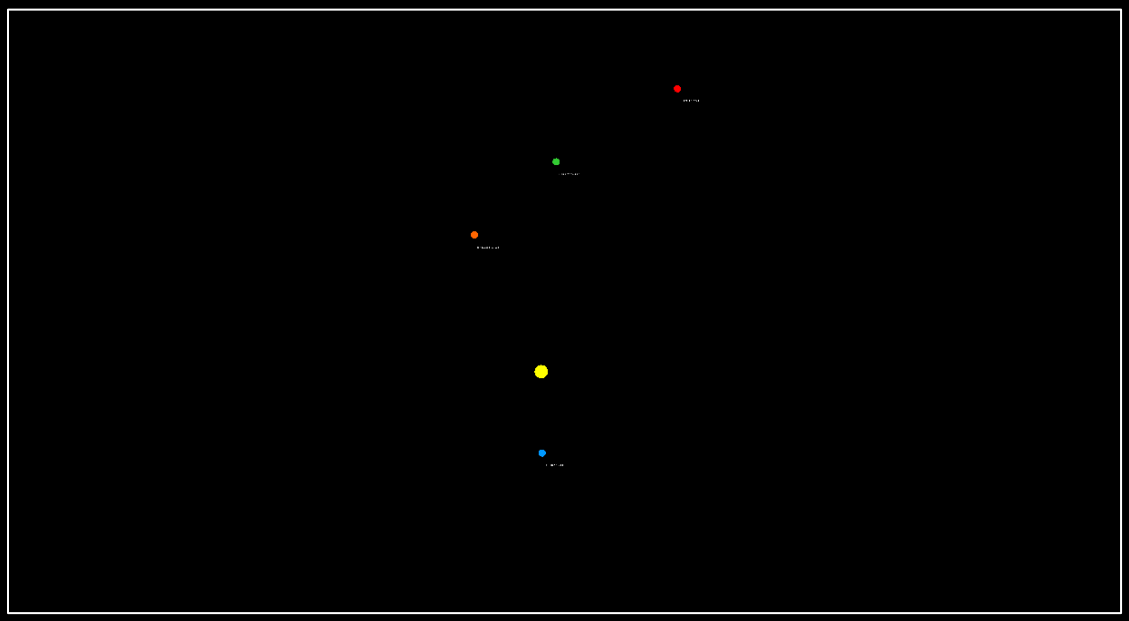
Otras características implementadas que servirán para la representación visual es una herramienta que genere texto para mostrar las coordenadas de los planetas en todo momento.

```
void Init(sf::Text text, sf::Font font, sf::Color color, float x, float y, int charSize, string str);
```

Esta herramienta está conformada por una clase con los principales atributos de dibujado posicionado de texto.



ITERACION #1



Primera representación visual del resultado esperado para el sistema solar, esta versión está sujeta a cambios…

Esta iteración muestra el funcionamiento del sistema de movimiento de los planetas, este no cuenta con un sistema de búsqueda, pero si tiene el uso de las fórmulas para la representación visual y lógica de movimiento orbital.

IMPLEMENTACIÓN

Universe

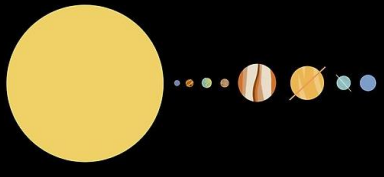
La clase universe se encarga de administrar todos los planetas que conforman al sistema solar, esta clase se conforma por tener una inicialización de los planetas, una actualización de estos donde la posición de los planetas se cambiara relativo al sol, y para finalizar una función que permita pintar al sistema solar.

Body

Esta es la clase más importante del proyecto ya que cuenta con toda la información requerida para la actualización de los planetas en tiempo de ejecución, además de dar los atributos a cada planeta, también actualiza su información para poder moverse alrededor del sol.

Esta clase cuenta con atributos como:

- Masa
- Posición
- Radio
- Afelio
- Perihelio
- Velocidad de movimiento
- Color
- Dirección



IMPLEMENTACIÓN

La clase body cuenta con una función de inicialización la cual obtiene los datos para generar un planeta o cuerpo.

```
CBody::CBody(Vector2 pos, float mass, float radius, float perihelius, float aphelius, float cirlceSpeed, float
circleRadius, sf::Color color, CBody * parent)
{
  m_parent = parent;
  m_direction_X = 1;
  m_pos = pos;
  m_mass = mass;
  m_radius = radius;
  m_perihelius = perihelius;
  m_aphelius = aphelius;
  m_circleSpeed = cirlceSpeed;
  m_currentSpeed = cirlceSpeed;
  m_circleRadius = circleRadius;
  ////////////////////////////////////////
  // Init circle attributes for planet
  Circle.Init(m_shapes, color, radius, pos.X, pos.Y, radius, radius);
  if (!font.loadFromFile("accid__.ttf"))
  {
    cout << "can't load font" << endl;
  }
}
```

Se puede apreciar que en esta función solamente se toman los atributos ya que en la función de update será donde se realizará el cálculo matemático.

Posteriormente, body cuenta con una funcion que actualiza la informacion de los planetas, esto funciona de la siguiente manera:

```
void CBody::Update(CBody * parent, float deltaTime)
```

La funcion update recibe dos parametros, el primer parametro es de tipo body el cual se define como el padre del planeta mismo, tomando los valores respectivos a cada uno, el segundo parametro es un delta Time el cual se encarga de mantener la informacion actualizada frame por frame.

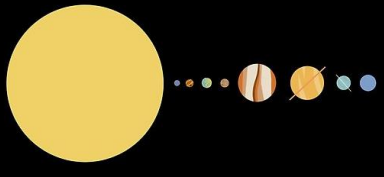
```
m_pos.X -= m_circleSpeed * deltaTime;
```

Se actualiza la posicion del planeta referentemente al tamano de la figura creada:

```
m_pos.X -= m_circleSpeed * deltaTime;
  if (parent)
    Circle.Update(m_Math.Add(parent->m_pos, m_pos));
  else
    Circle.Update( m_pos );

  if (m_parent)
    Move(deltaTime);
```

Se puede observar que cuando se emparenta al planeta, se llama a la funcion Move que es la escargada de realizar los movimientos entorno a su padre.



IMPLEMENTACIÓN

La funcion Move toma las ecuaciones previamente realizadas para poder ejercer la ecuacion de la elipse, de esta manera las posiciones del planeta cambaran referentes a su padre, pero respectivo a su propia orbita establecida.

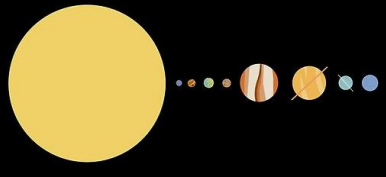
```
  m_pos.X += m_currentSpeed * (abs(m_pos.Y + 0.1f) / m_circleRadius) * m_direction_X;

  ////Y position
  if (m_pos.X > 0.0f)
  {
    m_pos.Y = sqrt(abs(1 - (pow(m_pos.X, 2) / pow(m_perihelius, 2)))) * m_circleRadius
* m_direction_X;
  }
  else
  {
    m_pos.Y = sqrt(abs(1 - (pow(m_pos.X, 2) / pow(m_aphelius, 2)))) * m_circleRadius *
m_direction_X;
  }
```

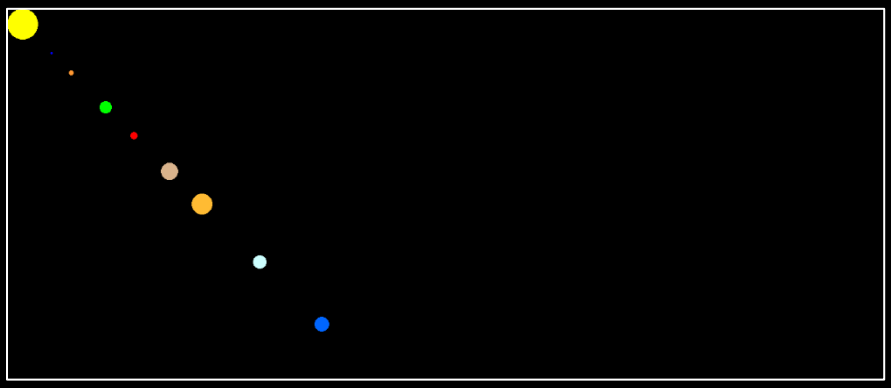
Retomando la ecuacion que se habia definido en un inicio, se puede observar en el codigo que el unico valor que cambia es relativo a su posicion en X o en Y, cabe mencionar que el radio tambien cambia para que de esta manera los datos tengan mas sentido, asi que en ves de realizar $\frac{(posX)^2}{circleRadius^2}$, se va cambiar por la siguiente ecuacion:

$$posX = \sqrt{\frac{1-(posX)^2}{(pharahelius \;||\; Aphelius)}} * circleRadius * dirX$$

Con estos cambios, el movimiento de los planetas es preciso y cumple con la simulacion de la segunda ley de kepler, de este modo se puede observar que la simulacion funciona adecuadamente resperto al afelio y el perihelio.



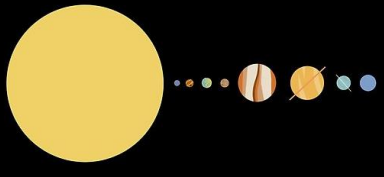
ITERACION #2



Segunda iteración enfocada en la búsqueda de datos de cada planeta…

Esta iteracion se basa totalmente en el funcionamiento del algoritmo de busqueda y los datos que se pueden obtener del mismo, en esta iteracion no se comentaran topocos de implementacion de movimiento, ya que se explico su funcionamiento en la iteracion numero 1.

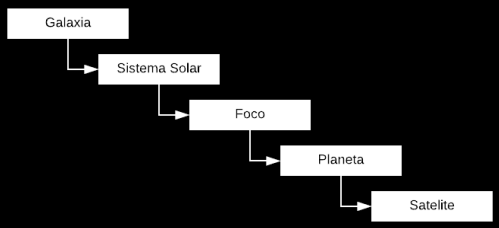
Si se van a tomar a considerar funciones como la inicializacion de los planetas en un sistema solar y como es que estos conforman una galaxia.



IMPLEMENTACIÓN

Estructura de clases por capas

Para la creacion de la nueva iteracion se decidio crear una nueva estructura en la que se crean los planetas como se hace en la vida real, esta estructura esta conformada por galaxias, en las cuales pueden existir sistemas solares (previamente definida como universos), y en los sistemas solares existen planetas, los cuales pueden tener satelites naturales.



Asi es como cada clase hereda de su padre y pueden alamacenar informacion del mismo, se pueden inicializar todos los datos y atributos que se deseen de cada planeta.

Nodo

Todas las clases cuentan con atrubutos mixtos que se relacionan entre si, esto permitiendo poder heredar de una clase varios atributos del mismo.

Al basar todo el sistema en una busqueda se puede definir que la galaxia esta compuesta por nodos, que en este caso son representados como los planetas y satelites. Asi cada planeta o nodo, tienen atributos propios que sirven a la hora de realizar una busqueda, estos atributos son:

- El padre del cual hereda.
- Los hijos del padre.
- La posicion del nodo o planeta.
- La capa en la que se encuentran.
- El nombre que tiene cada nodo.

Galaxia

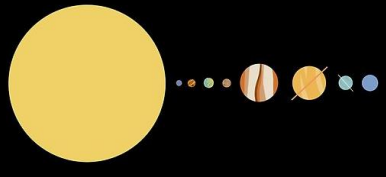
La galaxia hereda de la clase nodo por lo que cuenta con todos los atributos del mismo, por lo que las inicializaciones que esta cree, seran parecidas a la de las otras clases.

La clase galaxia tiene varias clases que se encargan de asignar los atributios tanto a los planetas del sistema solar y al nodo que le corresponde.

Esto quiere decir que la clase galaxia puede crear un sistema solar, darle un nombre, una posicion entre otras cosas.

A su vez la clase tambien cuenta con funciones basicas para el funcionamiento como es init, update, delete y draw, por cuestiones de repeticion estas estaran implicitas en todas las clases subsecuentes.

```
void GALAXIA::addSistem(SISTEMASOLAR *newsistem)
{
	newsistem->parent = this;
	childrens.push_back(newsistem);
	sistemas.push_back(newsistem);
}
```



Sistema Solar

El sistema solar es el encargado de asignar tanto los planetas como los focos y satelites de los mismos.

```
void SISTEMASOLAR::addPlanet(PLANETA * newPlanet)
{
	newPlanet->parent = this;
	childrens.push_back(newPlanet);
	planetas.push_back(newPlanet);
}

void SISTEMASOLAR::addFoco(FOCO *newfoco)
{
	newfoco->parent = this;
	childrens.push_back(newfoco);
	focos.push_back(newfoco);
}
```

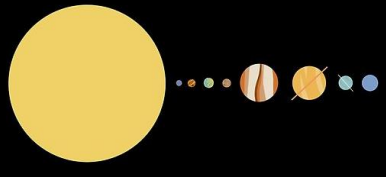
En el caso de nuestro sistema solar solo existe un foco que es referente al sol, mientras que satelites existen varios de ellos correspondientes a casa planeta.

Esta clase tambien se encarga de pintar al foco, los planetas y sus respectivos satelites con la funcion de draw.

```
void SISTEMASOLAR::drawSistem(sf::RenderWindow &wnd)
{
	for (int i = 0; i < focos.size(); i++)
	{
		focos[i]->draw(wnd);
	}
	for (int i = 0; i < planetas.size(); i++)
	{
    planetas[i]->draw(wnd);
	}
}
```

Foco

La clase foco a diferencia de la clase planeta, solamente se pinta, ya que los atributos son asignados por medio de la clase de galaxia.



IMPLEMENTACIÓN

Planeta

La clase planeta es idéntica a la clase foco, con una variante en la cual se pueden emparentar satélites al planeta deseado.

```
void PLANETA::addSatelite(SATELITE * newSatelite)
{
        lunas.push_back(newSatelite);
}
```

El derivar todas las clases de nodo, solo es necesario añadir a un vector el objeto de tipo satélite.

Satélite

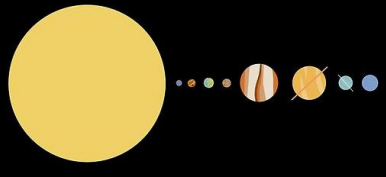
La clase satélite no cambia en nada a la clase foco, ya que solamente sirve para establecer sus atributos esenciales y la clase galaxia se encarga de definir las características de cada satélite.

Manager

La clase mánager es de las mas importantes de este proyecto ya que es la que inicializa la galaxia que contiene el sistema solar, de modo que tras asignar la información es posible realizar una búsqueda en el sistema solar.

Para la creación de una galaxia es necesario inicializar todo lo que contendrá la misma. Por lo tanto, se toma como partida, la creación de un sistema solar, el cual permitirá almacenar a los otros planetas.

```
SISTEMASOLAR* System1 = new SISTEMASOLAR;
```



IMPLEMENTACIÓN

Con la formación de un sistema solar, se pueda inicializar cada uno de los planetas que este pueda contener, con sus respectivos atributos. Para ello, se puede seguir un orden el cual se base en el foco dominante, en este caso el sol.

```
FOCO* sol = new FOCO;
sis1->addFoco(sol);
sol->setName("sol");
sol->setRatio(25);
sol->setPosition(sf::Vector2f(0,0));
sol->setShapeColor(sf::Color::Yellow);
```

El foco cuenta con varios atributos como lo es su etiqueta, su radio, su posición inicial y su color. Con esto el foco se agrega al sistema solar, siendo así la estrella dominante.

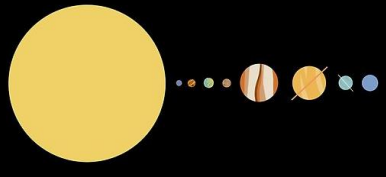
Posteriormente se inicializan todos los planetas que tiene el sistema, el cual al igual que el sol tienen los mismos atributos que el foco, con la diferencia de que se dan los datos de afelio y perihelio.

```
PLANETA* tierra = new PLANETA;
sis1->addPlanet(tierra);
tierra->setMasa(10);
tierra->setName("tierra");
tierra->setRatio(10);
tierra->setDistToFoco(50, 60);
tierra->setPosition(sf::Vector2f(150,150));
tierra->setShapeColor(sf::Color::Green);
```

Este proceso se puede repetir constantemente con la cantidad de planetas deseados.

Una vez inicializado el sistema solar, se puede añadir el sistema a la galaxia desea, permitiendo así, tener una galaxia con un sistema solar y sus planetas correspondientes.

```
galaxia = new GALAXIA;
galaxia->addSistem(sis1);
galaxia->onInit();
```



Búsqueda en el espacio

La búsqueda en el sistema solar, esta definida por una función de tipo nodo recursiva la cual se encarga de recibir el nombre del planeta y el nodo a buscar.

¿Qué es un algoritmo de búsqueda? (ecured, 2019)

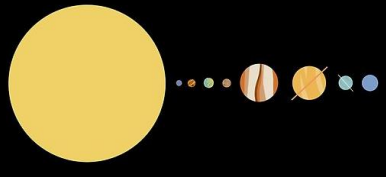
Un algoritmo de búsqueda es aquel que está diseñado para localizar un elemento concreto dentro de una estructura de datos. Consiste en solucionar un problema de existencia o no de un elemento determinado en un conjunto finito de elementos, es decir, si el elemento en cuestión pertenece o no a dicho conjunto, además de su localización dentro de éste.

¿Qué es una función recursiva? (Fernando, 2019)

Se dice que una función es recursiva cuando el cuerpo de la función utiliza a la propia función.

¿Qué es un árbol? (Blancarte, 2019)

Los Árboles son las estructuras de datos más utilizadas, pero también una de las más complejas, Los Árboles se caracterizan por almacenar sus nodos en forma jerárquica y no en forma lineal como las Listas Ligadas, Colas, Pilas, etc., de las cuales ya hemos hablado en días pasados.



IMPLEMENTACIÓN

Un árbol está estructurado de la siguiente manera:

- Nodos: Se le llama Nodo a cada elemento que contiene un Árbol.
- Nodo Raíz: Se refiere al primer nodo de un Árbol, Solo un nodo del Árbol puede ser la Raíz.
- Nodo Padre: Se utiliza este término para llamar a todos aquellos nodos que tiene al menos un hijo.
- Nodo Hijo: Los hijos son todos aquellos nodos que tiene un padre.
- Nodo Hermano: Los nodos hermanos son aquellos nodos que comparte a un mismo padre en común dentro de la estructura.
- Nodo Hoja: Son todos aquellos nodos que no tienen hijos, los cuales siempre se encuentran en los extremos de la estructura.
- Nodo Rama: Estos son todos aquellos nodos que no son la raíz y que además tiene al menos un hijo.

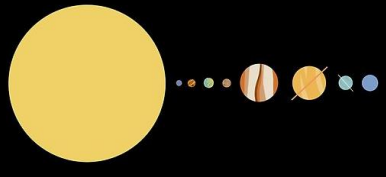
¿Como funciona?

Al igual que en muchas búsquedas dentro de un árbol, o una estructura, se tiene un nodo el cual es el que contiene la información, y es el cual pude ser encontrado de distintas maneras, contemplando su complejidad y el algoritmo a utilizar. Para esto es necesario analizar paso a paso el funcionamiento de este algoritmo.

```
CNODO * MANAGER::search(std::string name, CNODO * node)
```

La función de búsqueda es de topo nodo, ya que servirá para la recursión de esta. Se inicia la función comparando el nombre del planeta deseado con los nodos almacenados.

```
if (node->nombre != name)
```



Posteriormente se entra en un ciclo que recorre todos los hijos del nodo en búsqueda del nodo deseado.

```
for (int i = 0; i < node->childrens.size(); i++)
{
	CNODO *NODE = search(name, node->childrens[i]);
	if (node!=nullptr)
	{
		return NODE;
	}
}
```

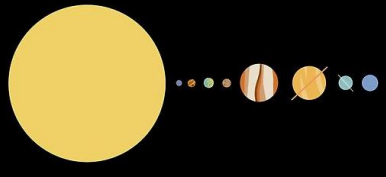
Dentro del recorrido se busca recursivamente el nodo en base a los hijos del mismo, si se encuentra el nodo deseado se, se regresa el nodo resultante. De lo contrario, se regresa el nodo actual para realizar una nueva búsqueda.

Complejidad

Existen dos casos para el algoritmo de búsqueda que se implementó, los cuales son los siguientes:

Mejor Caso: El mejor caso que se puede representar es que se busque exclusivamente una galaxia ya que será el más cercano. $\Omega(1)$

Peor Caso: El peor caso que se puede representar es que se busque la última luna del ultimo planeta del ultimo sistema. $O(n)$



BIBLIOGRAFÍAS

http://dspace.ucbscz.edu.bo/dspace/bitstream/123456789/4379/3/1847.pdf - libro

https://dialnet.unirioja.es/servlet/articulo?codigo=3694888

http://cslibrary.stanford.edu/110/BinaryTrees.html

https://www.includehelp.com/data-structure-tutorial/red-black-tree.aspx

https://o6ucs.files.wordpress.com/2012/11/data-structures-and-algorithms-in-c.pdf - libro