Gliwice, 26.06.2019

Semester: 2

Group: 2

COMPUTER PROGRAMMING LABORATORY

Author: Krzysztof Sauer

E-mail: krzysau648@student.polsl.pl

Tutor: dr inż. *Piotr Fabian*

1. Task topic

Simulation of gravity. Write a program simulating the movement of bodies in their gravitational field. Results should be presented as an animation.

This problem required some physical as well as algebraical knowledge. The first step was to design a UI for the user. I decided to help myself with SFML. My first goal was to make things move so after the basic setup, I started creating an object for a simulated entity, as well as tried to work out how to simulate vector relations. To do this I had to remind a bit of vector calculus as well as relations between mass, acceleration, velocity and unit/direction vector. Next, was to work out the simulation of gravity and how what the final force vector is (as well as it's direction). I came up with the idea of creating 2 nested loops for(i;i<number of bodies;i++) where the first loop would indicate the object which force is being calculated and the second loop, which would calculate and add individual forces of other bodies to the body indicated in the previous loop. This was the hardest part. Rest was actually a matter of rewriting and adding existing functions. I did experimet a little with sf::View, which gave me an abilityto zoom in and zoom out, created a fucntion updating a position, which was basically AccelerationVector*timeElapsed, made color of the objects pedendent on the mass of circles, as well as few of miscellaneous functions, which gave user more control.

2. External specification

To create an object use RMB (Right Mouse Button).

While the RMB is pressed, if you don't move a mouse, you will create a static object, which does not move, what could be used as a point of gravity.

While the RMB is pressed and you do move the mouse, a line will be drawn from the point of the click to the cursor's tip. This line indicates the object direction and the line lenght, determines the speed at which the object will be heading towards that direction.

To change mass of object use mouse scroll:

Scrolling up/down will result in increasing/decreasing the mass of the object by 50.
If during that you will use CTRL + Scroll, the increment will be multipied by 20.

There are also some minor key configs:

Q and E – zoom out/zoom in.
Space – pause/resume simulation
Z and X – decrease/increase speed of simulation.
Note: The lower the time is, the more accurate the simulation becomes
F1 – toggle the display of help infromation.
R – restart the simulation.

3. *Internal specification*

The “core” of the simulation. It calculates and updates new position of simulated bodies.

```
//Physics core
if (clear)
{
	{
		//Calculate gravitational vectors
		if (time != 0)
		{
			for (int i= 0; i < ID; i++) {
				for (int j= 0; j < ID; j++) {
		//This if statement prevents from simulation of object A - object A which would result in undefined behavior
					if (i != j && !Celestial[i].isStatic) {
			double FScalar;
				//Celestial[i].ForceVector		= sf::Vector2f(0, 0);
				Celestial[i].ForceDirection	= sf::Vector2f(0, 0);
				Celestial[i].AccelerationVector	= sf::Vector2f(0, 0);
				//For notation purposes dx		= d.x2 - d.x1
				double dx				= Celestial[j].position.x - Celestial[i].position.x;
				double dy				= Celestial[j].position.y - Celestial[i].position.y;
				//Normalize
				Celestial[i].ForceDirection.x	= (dx) / sqrt(((dx) * (dx)) + ((dy) * (dy)));
				Celestial[i].ForceDirection.y	= (dy) / sqrt(((dx) * (dx)) + ((dy) * (dy)));
				//Obtain Gravitational Force F	= GmM/r^2
				double m				= Celestial[i].ObjMass; double M = Celestial[j].ObjMass;
				double r				= sqrt((dx * dx) + (dy * dy));
FScalar  = 6.67408 * (Celestial[i].ObjMass * Celestial[j].ObjMass) / (sqrt((dx * dx) + (dy * dy)) * sqrt((dx * dx) + (dy * dy)));
				//Convert to F(orce)Scalar by multiplying AccelerationVector * ForceDirection vector * Force scalar
			Celestial[i].AccelerationVector.x = Celestial[i].AccelerationVector.x + Celestial[i].ForceDirection.x * FScalar / m;
			Celestial[i].AccelerationVector.y = Celestial[i].AccelerationVector.y + Celestial[i].ForceDirection.y * FScalar / m;
				//Change Velocity Vector v		= v0 + at
			Celestial[i].VelocityVector.x	= Celestial[i].VelocityVector.x + Celestial[i].AccelerationVector.x * time;
			Celestial[i].VelocityVector.y	= Celestial[i].VelocityVector.y + Celestial[i].AccelerationVector.y * time;
					}
				}
			}
		}
	Vector2f Delta = Vector2f(0, 0);
	//Update graphics
	for (int i = 0; i < ID; i++)
	{
		if (!Celestial[i].isStatic && time != 0)
		{
			Delta.x = (time * Celestial[i].VelocityVector.x + time * Celestial[i].AccelerationVector.x);
			Delta.x = (time * Celestial[i].VelocityVector.x + time * Celestial[i].AccelerationVector.x);
			Delta.y = (Celestial[i].VelocityVector.y + time * Celestial[i].AccelerationVector.y);
			Celestial[i].position.x = Delta.x + Celestial[i].position.x;
			Celestial[i].position.y = Delta.y + Celestial[i].position.y;
		}
		for (int i = 0; i < ID; i++)
		{
			celestialsprite[i].setPointCount(resPT);
			celestialsprite[i].setOrigin(sf::Vector2f(Celestial[i].ObjSize, Celestial[i].ObjSize));
			celestialsprite[i].setPosition(Celestial[i].position);
			celestialsprite[i].setRadius(Celestial[i].ObjSize);
		}
	}
}
}
```

Collision detection:

```
//Collision detection
quo:
	for (int i = 0; i < ID; i++)
	{
		for (int j = 0; j < ID; j++)
		{
			if (j != i)
			{
				double dx = Celestial[j].position.x - Celestial[i].position.x;
				double dy = Celestial[j].position.y - Celestial[i].position.y;
				double r = Celestial[i].ObjSize + Celestial[j].ObjSize;
				double d = sqrt((dx * dx) + (dy * dy));
				if (r >= d && Celestial[i].ObjSize >= Celestial[j].ObjSize) {
					object child(Celestial[i], Celestial[j]);
					Celestial.at(i) = child;
					Celestial.erase(Celestial.begin() + j);
					ID--;
					goto quo; //Breaks the loop and forces to check again for colisions (otherwise vector would throw exception)
				}
				if (r >= d && Celestial[i].ObjSize < Celestial[j].ObjSize) {
					object child(Celestial[j], Celestial[i]);
					Celestial.at(j) = child;
					Celestial.erase(Celestial.begin() + i);
					ID--;
					goto quo; //Breaks the loop and forces to check again for colisions (otherwise vector would throw exception)
				}
			}
		}
	}
```

It checks if the sum of checked circles radius is greater that their distance.

Obsolete object detection:

```
quo2:
for (int i = 0; i < ID; i++) {
	if (Celestial[i].position.x > 6000 || Celestial[i].position.x < -5000 || Celestial[i].position.y > 4000 || Celestial[i].position.y < -3000)
		{
		Celestial.erase(Celestial.begin() + i);
		ID--;
		goto quo2; //Breaks the loop and forces to check again for obsolete objects (otherwise vector would throw exception)
		}
}
```

This function checks if object is outside the screen, that is if -3000[px]<y<4000[px] and if -5000[px] and -3000[px]
If it's true, it erases the vector, decreases total object count and breaks the loop with goto command.

Two object collision.

It's worth to take a look at one of constructors of class, to be specific `object(object a, object b)`

```
class object
{
public:
	bool isStatic;
	sf::Vector2f position;
	sf::Vector2f start;
	sf::Vector2f end;
	double ObjMass;
	double ObjSize;
	double Velocity;
	sf::Vector2f VelocityVector;
	sf::Vector2f UnitVector;
	sf::Vector2f ForceDirection;
	sf::Vector2f AccelerationVector;
	sf::Vector2f ForceVector;
	object(double, double, sf::Vector2f, sf::Vector2f);
	object(bool, double, double, sf::Vector2f);
	object(object, object);
	object() { isStatic = false; }
	~object();
	double calculateVelocity();
	sf::Vector2f calculateDirection();
private:
};
object::object(object a, object b) : object() {
	sf::Vector2f VelocityVector;
```

```
	sf::Vector2f UnitVector;
	sf::Vector2f ForceDirection;
	sf::Vector2f AccelerationVector;
	sf::Vector2f ForceVector;
	this->position = a.position;

	if (a.isStatic || b.isStatic)
		this->isStatic = true;
	else
		this->isStatic = false;

	this->ObjMass = a.ObjMass + b.ObjMass;
	if (a.ObjSize > b.ObjSize) {
		this->ObjSize = sqrt(a.ObjMass + b.ObjMass / 3.142);
	}
	else {
		this->ObjSize = sqrt(a.ObjMass + b.ObjMass / 3.142);
	}

	this->VelocityVector.x = (b.VelocityVector.x + a.VelocityVector.x) * (1 - a.ObjMass / this->ObjMass);
	this->VelocityVector.y = (b.VelocityVector.y + a.VelocityVector.y) * (1 - a.ObjMass / this->ObjMass);
}
```

It reads the values of first (bigger/heavier) which will be replaced with a new object ,and second (smaller/lighter) object which will be deleted. The new object has a mass equal to sum of previous object's masses and it takes a part of the heavier object's velocity vector (a negative relation of heavier object to the newly created one) and automatically determines the direction, which the object will take when operation is complete. Note that the loop has to be deleted here, since the vector which is being iterated has it's size reduced, what usually results in getting an exception of access violation.

4. Source code

Whole code has been included in separately, due to its size. Names of files are:

Main.cpp (main part of the code)

Object.h (a class which stores simulated object information)

5. Testing

You can see that in the source code, there is debugFlag variable, which is responsible for toggling the debug mode of the program. Due to dynamic nature of the program, I could not use Visual Studio's debug mode, thus I have written a few if(){} statements which allow the program to print calculation variables directly in the window. With the help of it, I could keep track of critical values like DirectionVector or AccelerationVector, which were crucial for the simulation.