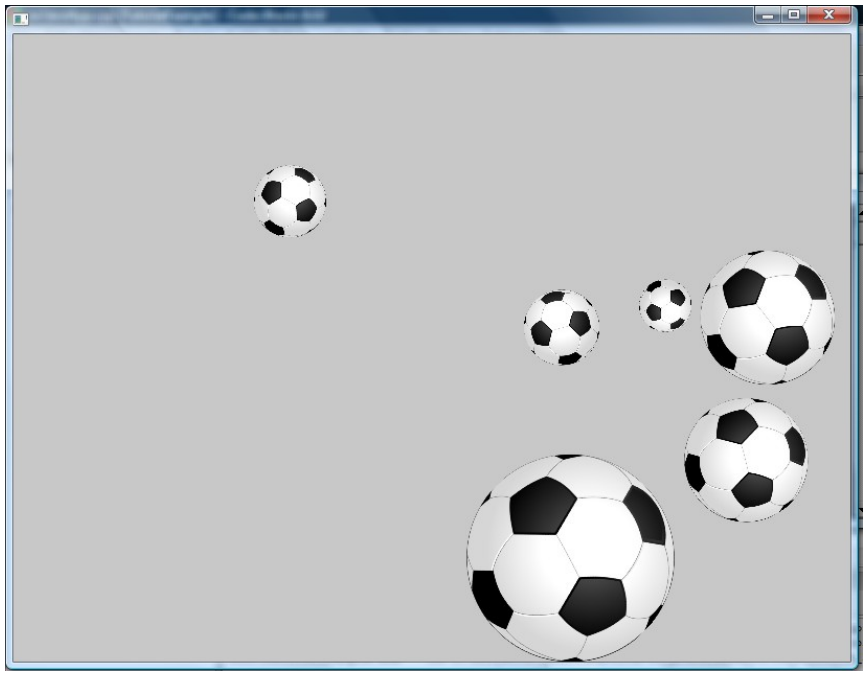
Tutorial Simple Physics (2d)

by Sophie van der Sijp and Fin Kingma.

In this tutorial we are going to create numerous round moving objects which collide with each other and the surrounding walls. This will be done with images, each with it's own explanation of the steps which should be taken. It is recommended to test the application as often as you like, to understand what you are programming.
Also try to experiment during each step of the tutorial, these pages are merely guidelines to create your own physics engine.

Some lines are made italic, meaning these lines are specific codelines.
Other lines are made in grey, meaning they explain the step you just took, in case you were wondering.

This tutorial is divided into these following steps:



Nov 1 2009

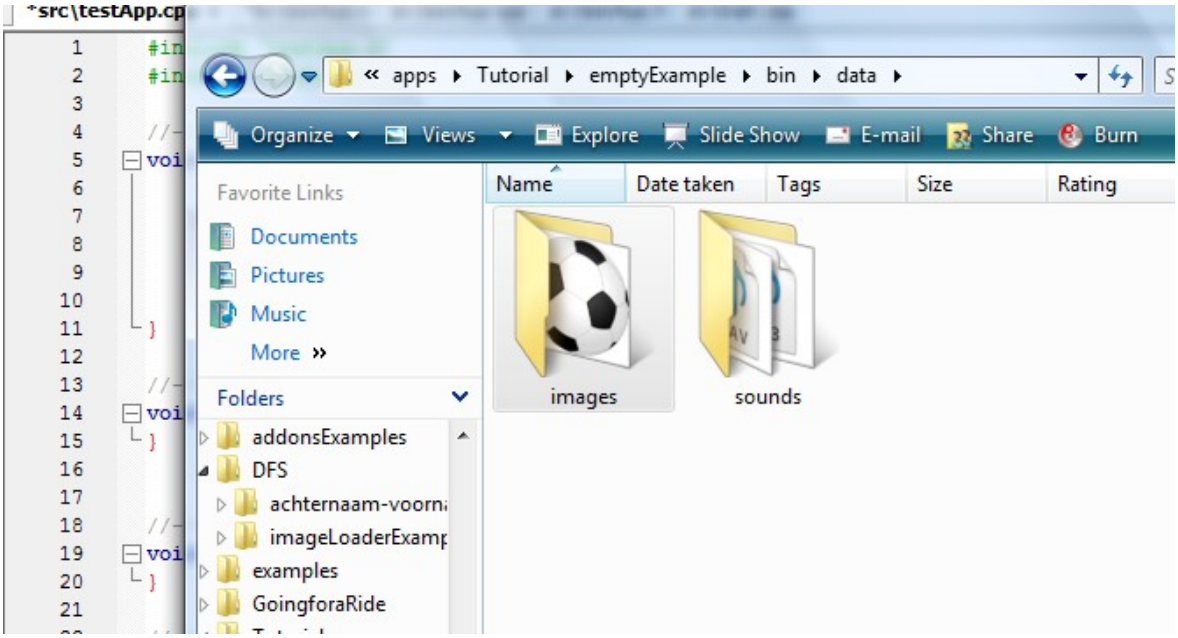
Drawing

Start by opening “empty example” (downloadable from openframeworks.cc).
Once you have done this we can import the ball. To do this go to the header (if you have the .cpp file open, press on f11 to open the header). Here we will give the image it's own variable which we can link to an image.

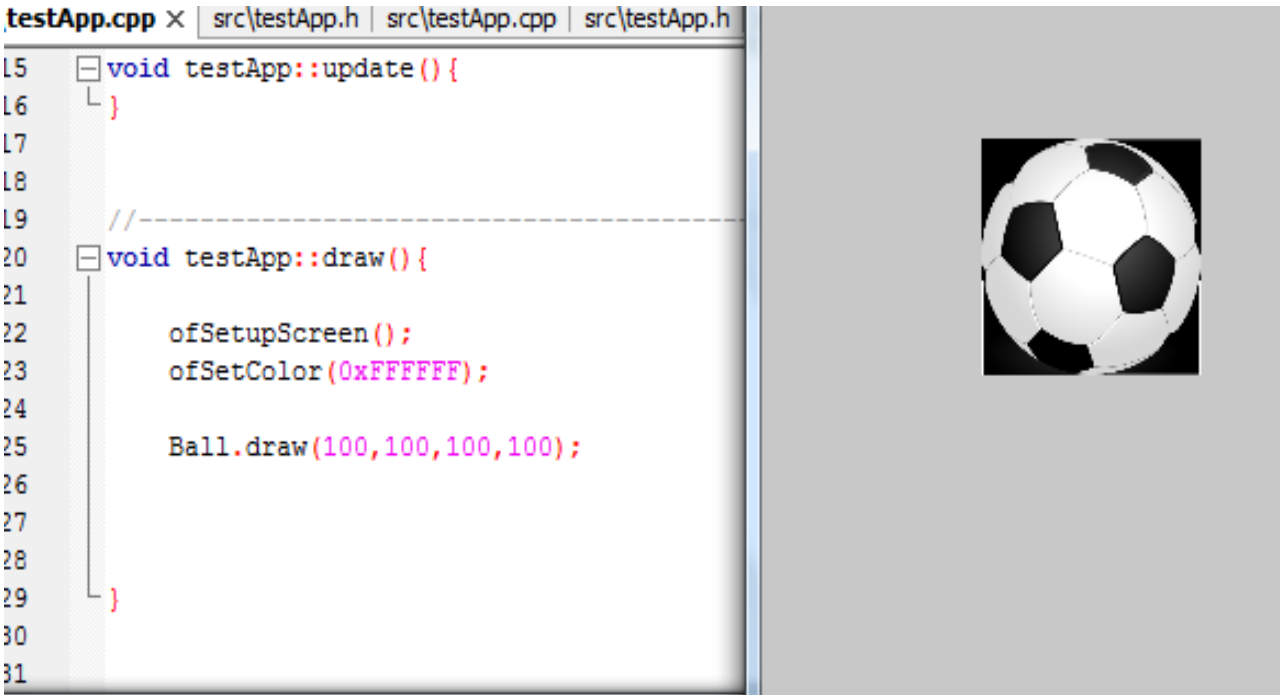
```
void setup();
void update();
void draw();

void keyPressed  (int key);
void keyReleased(int key);
void mouseMoved(int x, int y );
void mouseDragged(int x, int y, int button);
void mousePressed(int x, int y, int button);
void mouseReleased(int x, int y, int button);
void resized(int w, int h);

ofImage Bal;
```

Now we can import the image “ball.png”. Make sure you note carefully where you have saved the image.
To load the image we'll use the code: “load.Image”. This code is placed in “setup” of the .cpp file.

Bal.loadImage(“images/Ball.png”);



To make sure the object can be seen we must “draw” it (see image below).

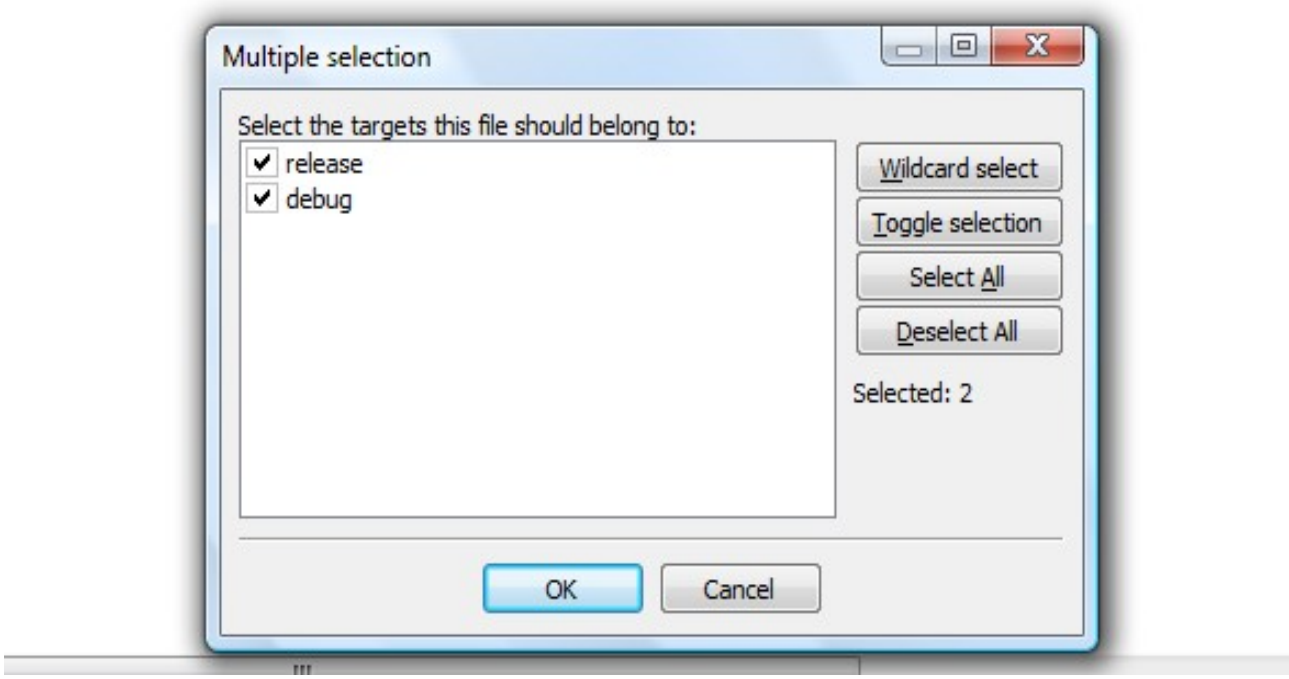


We mustn't forget to give the image the correct settings; in this case we need to remove the box surrounding the ball. This is done in the “setup” of the .cpp with the code:
ofEnableAlphaBlending();

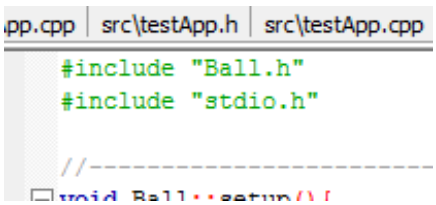
Classes

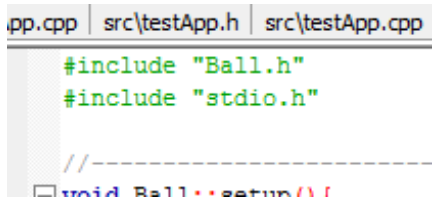
Now we have an object we can give it it's own class. To do this you need to open a new file. Codeblocks will then ask you if you want to save this with the project currently open. Do this, and then give the file a name like “Ball.h”. A window will pop up asking you to select the targets the file should belong to. Select both release and debug and click on OK.

Press f11 and repeat the proces, now naming the file .cpp.



We now need to add the class to the file testApp. This needs to be done in both the testApp.h and testApp.cpp files, on top of all the used functions like setup().





Movement

In the header of the class, which is named ball.h in the example, we need to make variables to give the object x&y positions and a speed to move at. Since we are building this engine in 2d, we need 2 variables for each.

```
//x & y position for Bal
        float yBal;
        float xBal;

//Bal's speed in directions x&y
        float vyBal;
        float vxBal;
```

In the .cpp file of the class you can define the values of these variables. These values will be overwritten later on, but in order to test properly they are added right now in the setup() function.

```
//value given to the x&y position of Bal
    yBal=0;
    xBal=0;
//value given to the speed of Bal
    vyBal=1;
    vxBal=1;
```

In the update() of the testApp.cpp (where all calculations are done) we'll establish the fact that every time an update is carried out 1 will be added to the current position of the object. This ensures the movement of the ball.

```
void testApp ::update(){

    yBal+=vyBal;
    xBal+=vxBal;

}
```

In order to actually make the ball move, you need to replace the first 2 values of the draw function by “yBal” and “xBal”.

Now the object can move, we want to make it bounce off the walls. Start by filling in the dimensions of the screen. This can be done in the main.cpp file.

```
5   //========================================================
6   int main( ){
7
8       ofAppGlutWindow window;
9       ofSetupOpenGL(&window, 800,600, OF_WINDOW);          // <
10
11      // this kicks off the running of my app
12      // can be OF_WINDOW or OF_FULLSCREEN
13      // pass in width and height too:
14      ofRunApp( new testApp());
15
16  }
17
```

Bouncing

First we'll establish the fact that when the y position of the object is larger than the width of the screen the object reverses it's direction (it “bounces”). We do this by multiplying the speed of the object by -1, reversing it.

However, because the ball is 100x100, and we want it to “bounce” back as soon as the edge of the ball hits the wall, we need to play about with the numbers a bit. The center point of the ball is the left top corner of the total image, which is shaped like a box. Because the ball is 100x100, we take 100 off of the hit point of the wall.

e.g. The screen used in the example is 800x600. Therefore our hit points are 700 and 500.

```
32      if (yBal>700)
33      {
34          vyBal*=-1;
35      }
36
37      if (xBal>500)
38      {
39          vxBal*=-1;
40      }
41
```

Don't forget to use these bounces for the left and top borders as well (which are simply “0”), or you will lose your ball anyway.

Now we have a bouncing object, we can have some fun. Let's say a sound is played every time the ball hits a wall, each wall with it's own sound.

To do this you first need to import the sounds. Use the code loadSound. Don't forget to refer to the correct folder.

You need to mention them in the header as well, the code used for this is: ofSoundPlayer [name]; *ofSoundPlayer SoundN;*

Then you need to use the code play in update to ensure the sound is played when the wall is hit. SoundN.play();

apps\Tutorial\emptyExample\src\Ball.h | **apps\Tutorial\emptyExample\src\Ball.cpp** × | src\m

```
7       ofEnableAlphaBlending();
8
9       SoundN.loadSound("sounds/Wall_1.wav");
10      SoundW.loadSound("sounds/Wall_2.wav");
11      SoundS.loadSound("sounds/Wall_3.wav");
12      SoundE.loadSound("sounds/Wall_4.wav");
13      Bal.loadImage("images/Ball.png");
14
15      yBal=2;
16      xBal=3;
17      vyBal=0.2;
18      vxBal=0.2;
```

Friction

Because we don't want the ball to keep on bouncing forever we're going to put in some deceleration.

First give the object a new variable, like dvBal, to represent the deceleration. This variable will be placed in the setup() of your testApp.

apps\Tutorial\emptyExample\src\Ball.h **apps\Tutorial\emptyExample\s**

```
15      yBal=2;
16      xBal=3;
17      vyBal=1;
18      vxBal=1;
19      dvBal=1.0005;
20
```

Now divide the value of the x&y of the object with the value of dvBal, in update(). This way the speed of your ball will be slightly affected by friction.

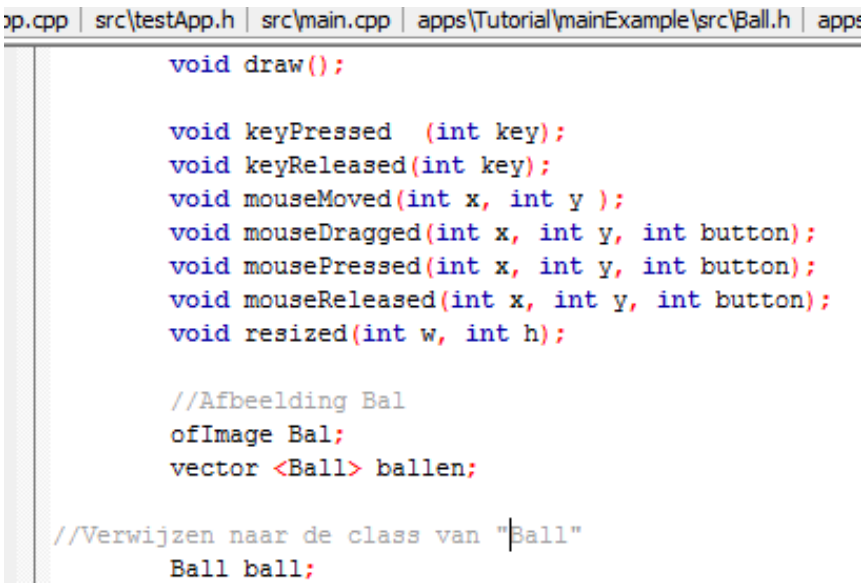
```
29
30      yBal+=vyBal;
31      xBal+=vxBal;
32      vyBal/=dvBal;
33      vxBal/=dvBal;
34
35
```

Vector array

For anyone unfamiliar with arrays, arrays are like simple tables. You put an ammount of values in the table and you can call them by requesting the right number. (The first value gets number 0, then 1, 2, 3, etc).
Normal arrays can only hold simple data types, like numbers or words. In order to hold an ammount of classes, we need to use a vector array.

Now we start adding more objects to the scene. Do this using the code vector<[*name class*]> [*name array*];
vector <Ball> ballen;

Also, in order to duplicate the class we made a few pages ago, we need to define the class in the header with a new name [name class] [new name];
Ball ball;



Now in the setup(); of your testApp, we can add as many classes as we want, using the code assign().
ballen.assign(2,ball);

This will make 2 new classes, which will be categorized in the vector array.

Loops

Now you are able to create more objects. You just have to make sure each new object receives it's own start point and velocity.

Start using a “for” loop; for (int i=0; i<ballen.size();i++)

int=0; with this you're saying that you are using an integer, in this case with a value of 0 (the loop therefore starts at 0). “i” is a temporary variable, made just for the loop.

i<ballen.size(); With this you're saying that the loop keeps on repeating as long as i is smaller than the amount of objects which are archived in the array ballen.

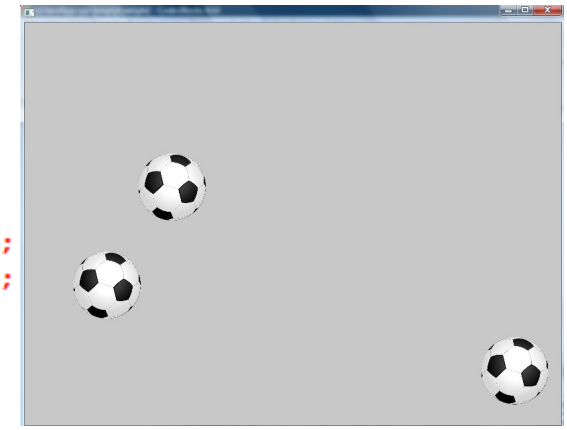
i++; this ensures that the loop adds one at the end of each loop; the counter.

```
    for(int i=0;i<ballen.size();i++)
    {
```

Every time the loop is executed, the value “i” is 1 more. This way we can use “i” to call to the right class within the array and give each class new values.

Then give the objects a random start position and speed. In the brackets behind random fill in the minimum and maximum of the random values.

```
//Image van Ball importen
    Bal.loadImage("images/Ball.png");


//Waarde van deceleratie vaststellen
    dvBal=1.0005;
    ballen.assign(10,ball);

    for(int i=0;i<ballen.size();i++)
    {
        ballen[i].yBal=int(ofRandom(50,650));
        ballen[i].xBal=int(ofRandom(50,450));
        ballen[i].vyBal=ofRandom(0.5,2);
        ballen[i].vxBal=ofRandom(0.5,2);
    }
}
```



Placed in testApp.cpp under setup

The same for loop must be used in the update function, where all current variables must be given the name of the array, like:
ballen[i].xBal += ballen[i].vxBal;

Also the draw function must get this for loop, and the values of the draw function must be given these same names:
img.draw(ballen[i].yBal, ballen[i].xBal, 100, 100);

Keypressing

Now we'll make it that we add an object everytime a key is pressed.

Start by making the variable amountBall in testApp.h. In the .cpp we'll give amountBall a value, in this case 3.

```
//Image van Ball importen
    Bal.loadImage("images/Ball.png");
    amountBall=3;

//Waarde van deceleratie vaststellen
    dvBal=1.0005;
    ballen.assign(amountBall,ball);
```

Placed in testApp.cpp under setup()

Now in the testApp.cpp under keyPressed copy the code for the start positions and speed used. To make sure a ball is added everytime you press on an key, add the code amountBall++. This adds one on the value of the variable (which in our case is 3).

```
//--------------------------------------------------------------
void testApp::keyPressed  (int key){
    ballen.push_back(ball);
    amountBall++;

    ballen[ballen.size()-1].yBal=int(ofRandom(50,650));
    ballen[ballen.size()-1].xBal=int(ofRandom(50,450));
    ballen[ballen.size()-1].vyBal=ofRandom(0.5,2);
    ballen[ballen.size()-1].vxBal=ofRandom(0.5,2);
```

Instead of “i”, we are using the size of the array minus 1 (if there is 5 classes in the array, the last one answers to the number 4)

Gravity

To make it slightly more realistic, before we start working on the collision, we need to give everything the newton makeover; gravity and weight (or correctly speaking mass).

We'll start by gravity. Create a variable called gravity in the header. In the .cpp file give the gravity a value; as it may react rather sensitively you need to play around with numbers. We used 0.006.

```
        int amountBall;
/variabel voor gravity
        float gravity;
```

```
ballen.assign(amount

gravity=0.006;
```

Now add the gravity to the x movement of the object, in the update() function.

```
 //gravity
     ballen[i].vxBal+=gravity;
```

Now the ball will move/fall downwards.

Weight

Let's move on to the weight of the object.

As with the gravity, first you need to create a variable for the weight. Do this, as ever, in the header, of the class this time, since every ball will get a different weight.

```
//variabel voor gewicht
    int weight;
```

In the loop used earlier to create random start positions and speeds add the fact that each weight will be random.

```
for(int i=0;i<ballen.size();i++)
{
    ballen[i].yBal=int(ofRandom(50,650));
    ballen[i].xBal=int(ofRandom(50,450));
    ballen[i].vyBal=ofRandom(0.5,2);
    ballen[i].vxBal=ofRandom(0.5,2);
    ballen[i].weight=ofRandom(50,200);
}
```

This step also needs to included in the draw and the keypressed.

```
//--------------------------------------------------------------
void testApp::draw(){
for(int i=0;i<ballen.size();i++)
    {
        ofSetupScreen();
        ofSetColor(0xFFFFFF);

        //Bal toevoegen & start positie vastleggen
        Bal.draw(ballen[i].yBal,ballen[i].xBal,ballen[i].weight,ballen[i].weight);

    }
}


//--------------------------------------------------------------
void testApp::keyPressed  (int key){
    ballen.push_back(ball);
    amountBall++;

    ballen[ballen.size()-1].yBal=int(ofRandom(50,650));
    ballen[ballen.size()-1].xBal=int(ofRandom(50,450));
    ballen[ballen.size()-1].vyBal=ofRandom(0.5,2);
    ballen[ballen.size()-1].vxBal=ofRandom(0.5,2);
    ballen[ballen.size()-1].weight = int(ofRandom(50,200));
    ballen[ballen.size()-1].nullX = ballen[ballen.size()-1].weight / 2;
    ballen[ballen.size()-1].nullY = ballen[ballen.size()-1].weight / 2;
}
```

Now each object has it's own weight, we need to make it react appropriately according to it's weight. So the larger, thus heavier the object, the faster it needs to decelerate.

```
    ballen[i].vyBal/=dvBal+(ballen[i].weight/198);
    ballen[i].vxBal/=dvBal+(ballen[i].weight/198);
```

You can play around with the number 198 to change the deceleration to your taste.

We also need to change the way the objects bounce off the walls.

```
if (ballen[i].yBal>800-balls[i].weight)
{
    ballen[i].vyBal*=-1;
    ballen[i].yBal = 798 - balls[i].weight;
    //SoundE.play();
}

if (ballen[i].xBal>600-balls[i].weight)
{                                                    ---------------------->
    ballen[i].vxBal/=1+(balls[i].weight/400);
    ballen[i].vyBal/=1+(balls[i].weight/200);

    ballen[i].vxBal*=-1;

    if (ballen[i].vyBal >0) {
        ballen[i].vyBal-= 0.001;
    } else {
        ballen[i].vyBal+= 0.001;
    }

    if (ballen[i].vxBal<=0.3 && balls[i].vxBal >= -0.3){
        ballen[i].vxBal = 0;
        ballen[i].xBal= 600-balls[i].weight;
    }

    else {
        ballen[i].xBal=598-balls[i].weight;
        ballen[i].vxBal+=0.1;
```

Here we say if the x position of the ball is larger than the edge of the wall minus the size of the ball (see the trick we used earlier for bouncing off walls) then we add some more friction.
The deceleration will be divided by 1+ the weight of the object divided by 400 or 200, depending if it's about the x or y direction

And finally we displace the ball 1 pixel against the wall to deny any errors from happening.

Collision

Now we have realistically moving object we can let them bounce off each other; the collision.

First a short explanation how this is achieved:
When two objects move towards each other, the distance between the edges of the objects needs to be calculated. When the two edges “touch” each other, they bounce off in the direction according to the angle the two objects collided at.
What's important is to calculate the correct distances. As these differ per object, each now having it's own size, we need to make a few formulas. We've now gotten to the good bit.

It's important to know that as soon as you start the calculations for the collision you cannot test it until the entire code is complete.

Make a variable for the the distance between the two colliding objects. We've also made a variable (called newdist) which will make the values used in the calculations positive, thus making the said calculations easier.

```
float distX;
float distY;
float dist;
```

Also make a variable for the centre point of the object in the header of the object's class.

```
float nullX;
float nullY;
```

These null values must also be created in the setup function of the array and the keypress. And in the update you will need this line:
ballen[i].nullX = ballen[i].xBal + ballen[i].weight / 2;

In the old update loop we'll now make a new loop for the collision. This way we can calculate the distance between all the different classes, no matter how many there are.

```
void testApp::update(){
    for(int i=0;i<ballen.size();i++)
    {
    //loop om de distance te berekenen
        for (int j=0;j<ballen.size();j++) {
```

As “i” is already taken in this loop for, let's say object1, we need a new letter, in this case “j”, to represent say object2. In short they represent both the objects in the collision.

```
    for (int j=0;j<ballen.size();j++) {
        if (j!=i) {
            distX = ballen[i].nullX - ballen[j].nullX;
            distY = ballen[i].nullY - ballen[j].nullY;
```

Here it says that if j does not equal I (so if both balls are different from eachother, we don't want the balls to calculate the distance to themselves). The distances between the two can be calculated

by looking at the differences in both x and y.

Now the differences need to be calculated by drawing a straight line from the centre of the objects.
*dist = sqrt((distX*distX) + (distY*distY));*

```
if (dist <= (ballen[i].weight / 2) + (ballen[j].weight / 2)) {
    ballen[i].vyBal *= -1;
    ballen[i].vxBal *= -1;
    ballen[j].vyBal *= -1;
    ballen[j].vxBal *= -1;
```

Here, if the distance is smaller or equal to the sum of the radii of the colliding objects divided by two, the directions of the movement of the objects will be reversed. This is the collision. To make it more fun we'll make the balls react differently depending on the weight of the ball they're colliding into.

In order to do this, we have used the following code:
*ballen[i].vxBal += (ballen[i].vxBal * (ballen[i].weight/200)) + (distX/(ballen[i].weight) * (200-ballen[i].weight)/200);*

This line means that the old speed has been added the a percentage of itself, and the remaining percentage of the distance, in which the percentage has been formed out of the weight of the ball.

It is also recommended to add the same friction as you added when you bumped against the wall.

Rotating

We're now going to let the object rotate around it's centre point.

We need to let numerous images be able to move independently from each other. To do this we need to use the code ofPushMatrix, which creates a temporary window in which certain events can take place, which are later forgotten. PullMatrix then stops this proces.

src\Ball.h | *src\testApp.cpp | src\testApp.h | src\main.cpp | src\Ball.h | src\Ball.cpp

```
130
131
132     //--------------------------------------------------------------
133     void testApp::draw(){
134     for(int i=0;i<ballen.size();i++)
135         {
136             ofPushMatrix();
137                 ofSetColor(0xFFFFFF);
138
139                 //Bal toevoegen & start positie vastleggen
140                 Bal.draw(ballen[i].yBal,ballen[i].xBal,ballen[i].weight,ballen[i].weight);
141             ofPopMatrix();
142         }
143
144     }
145
146
147     //--------------------------------------------------------------
148     void testApp::keyPressed  (int key){
149         ballen.push_back(ball);
150         amountBall++;
```

You now need to create a variable in the class header which we have named degrees. You then need to give degrees a value of 0 in the setup of testApp.cpp. This variable is then paired to the speed of the object.

To let the object rotate around it's centre point you first need to create a new centre point for the new matrix. To do this use ofTranslate(), in which you can add the y and x values of the ball (for an example, look on the next page).

OfRotate is then used to ensure that everything that follows uses this rotation around the centre point.

SetAnchorpoint helps by making sure the centre point of the image is in fact in the middle instead of at the top left corner of the image, so it will be correctly placed on the screen.

The last page shows an example of these final steps.

src\Ball.h | **src\testApp.cpp** × | src\testApp.h | src\main.cpp | src\Ball.h | src\Ball.cpp

```
132     //--------------------------------------------------------------
133     void testApp::draw(){
134     for(int i=0;i<ballen.size();i++)
135         {
136             ofPushMatrix();
137                 ofSetColor(0xFFFFFF);
138                 ballen[i].degrees+= ballen[i].vyBal+ballen[i].vxBal/20;
139                 ofTranslate(ballen[i].nullY, ballen[i].nullX);
140                 ofRotate(ballen[i].degrees);
141                 Bal.setAnchorPercent(.5,.5);
142 
143                 //ofTranslate(ballen[i].xBal+100, ballen[i].yBal+100);
144 
145                 Bal.draw(0,0,ballen[i].weight,ballen[i].weight);
146                 //Bal toevoegen & start positie vastleggen
147                 Bal.draw(0,0,ballen[i].weight,ballen[i].weight);
148             ofPopMatrix();
149         }
150     }
151 
152 
153     //
```

Eventually the image will be drawn and rotated around the correct centre point. It's positionwill then also be 0,0.