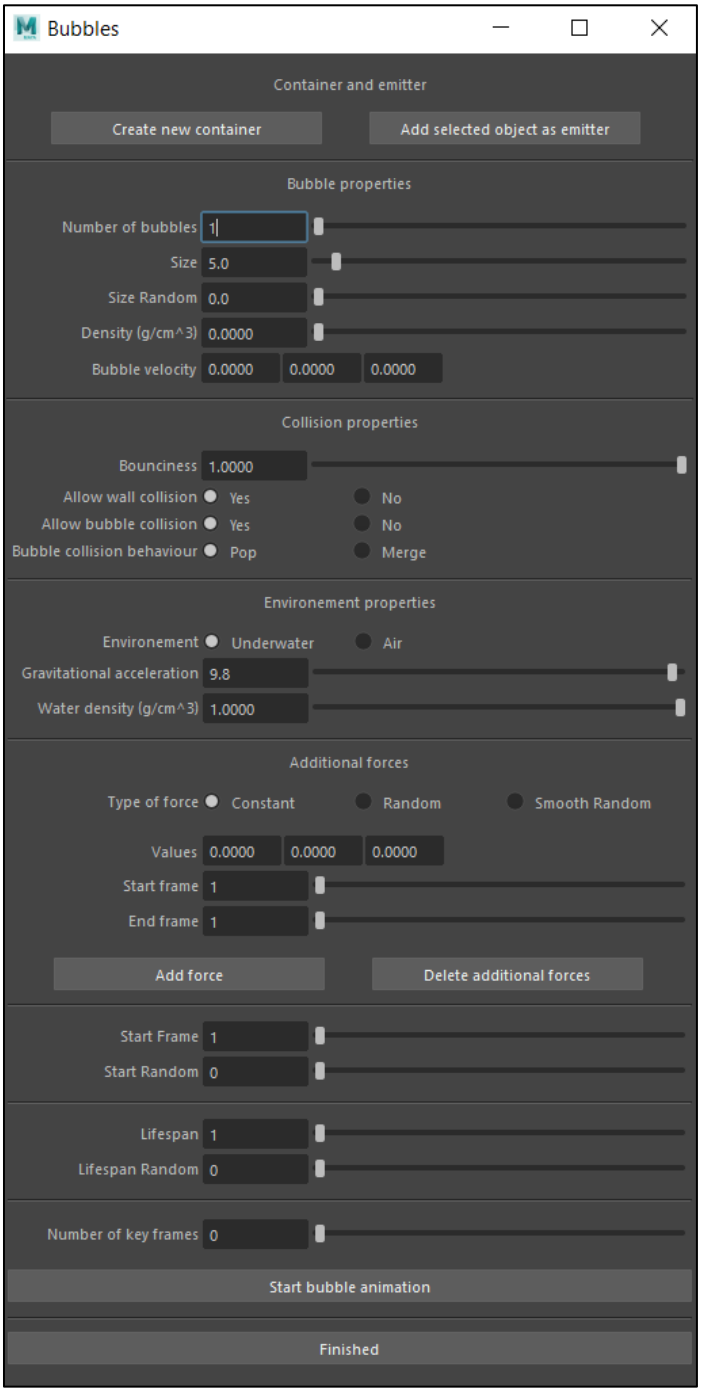
Scripting Report

Bubbles – Python script for Maya to generate bubble animation resembling real-world movement and interaction.



Manual

1. Open Maya
2. Open Bubbles.py, copy the content and paste it into the Maya script editor
3. Select all the code and run
4. The window on the right should pop up

How to use the UI

Button **Create new container** and creates a cube that should be resized and moved by the user before the animation starts. The button next to it adds any selected object(s) as emitter(s) from where the bubbles will appear. You need both a container and at least an emitter to start. You can reuse the emitter and the container, but the new bubbles will not interact with the old ones. To see the bubbles, you will need to use a transparent shading material on the container.

You can choose the environment and the bubble properties as you like.

If you want to add new forces, please do not forget to press Add force. If you do not like the forces, you can always delete **ALL** of them by clicking **Delete additional forces**.

Please do not forget to give the bubbles the **lifespan** and to introduce the **number of frames** of your animation, otherwise there will be no movement.

Depending on the number of bubbles and the number of animation frames, the program can run for a few seconds or for a really long time, so I recommend keeping the number of bubbles under 250 and the animation frames under 1000.

Implementation

My program is based on two main functions:

- **startSimulation –** calls other important functions as fillListOfBubbles, disperseBubbles, createBubble and prepares the scene for the animation;
- **startKeyframing –** it is called from inside startSimulation and it is the most compex function in the program as it takes care of the frame-by-frame animation; it represents the center of the flow of control, being rich in while loops and if statements, but it also calls many functions like updateForces, checkWallCollision, checkCollision, mergeBubbles, popBubble;

The simulation starts by filling the **listOfBubbles** as it is always important to have a well-defined data structure early on, especially when it is needed during the whole runtime. This list contains information about the state of the bubble (alive/dead), the geometry, the radius, the position in world space, the velocity, the starting and the ending frames for that bubble and its mass. It is very important to update the values and to check how their new ones may affect the scenes. This list is updated every frame for all the bubbles as long as they are still visible.

Next comes **dispersing the bubbles**, where I decided to let the user choose objects as emitters and then used their vertices' positions in space to place **polySpheres** as the geometry for the bubbles. In the meantime, the program is checking every bubble if it is inside the container and if it doesn't touch other bubbles. An improvement to the algorithm could be dispersing the bubbles emission over time by looking for the position of each bubble only when they become visible and start their lifespan. This way it is possible to disperse more bubbles because otherwise it is too crowded for big numbers and so the dispersion becomes impossible.

The bubbles are then created and parented to their own group.

The function **startKeyframing is** called and it begins by preparing a set of values that will be helpful later. It goes frame by frame into updating the resultant force obtained by adding all the additional forces that are active at that frame and then it starts checking every single bubble that is active. Depending on the user's input, the function looks for wall collision, bubble collision (respectively the type of bubble collision) and the type of environment. Data is compared between

immediate previous values and the ones from the current frame for the collision to be done properly, otherwise the bubble will be stuck to the wall.

The movement of the bubble is calculated frame by frame. To get the position of the bubble we calculate the displacement/ distance vector dividing the velocity at that frame by the number of frames per second and adding the result to the position in the previous frame as shown in Eq 1.

$$\boldsymbol{p(t) = p(t-1) + \frac{v(t)}{fps}}$$

$t = \ current\ frame$

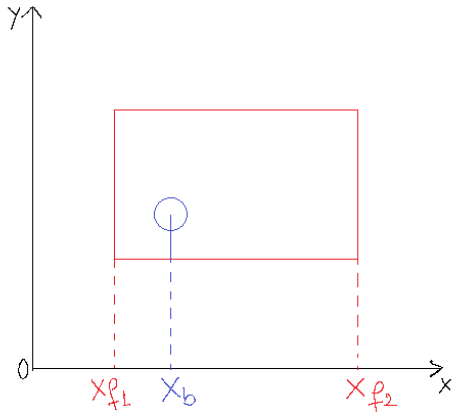
$p = \ position\ vector$

$v = velocity\ vector$

$fps = frames\ per\ second$

Equation **1**

For **checking the container-bubble collision** I opted for a very simple method that would take little time for the computer to process. However, this comes with its own constrains. It is required that the container is not rotated at all so that all his faces are parallel to the 3 planes formed by pairing the axis. Despite its name, the function used actually verifies if the bubble is not outside the container and if it intersects the container because for big velocities and small radiuses for the bubbles could make them step outside and not touch the container at all so we need to consider that to be able to bring it back immediately. Check Fig 1 to see the method.



$x_B - x_{f1} < r => collision\ is\ true$

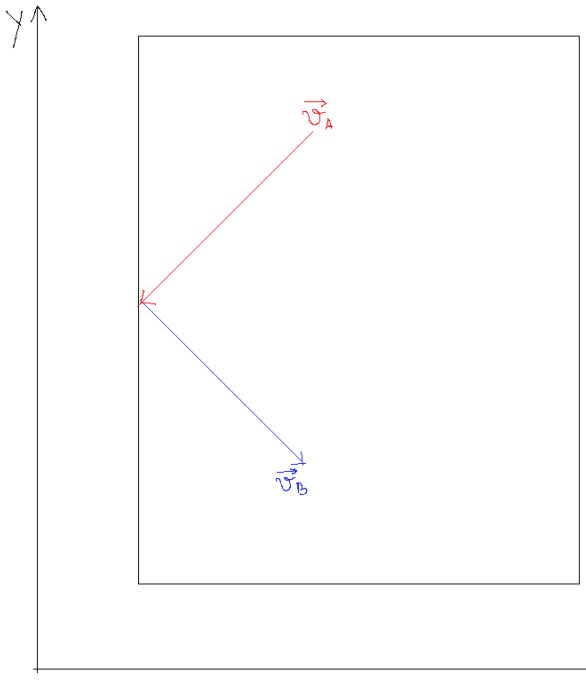
$x_{f2} - x_B < r => collision\ is\ true$

The same checking has to be done for y and z coordinates.

Fig **1**

When collision with one of the walls has occurred, the bubble can:

- **pop** – it becomes invisible and *dead*
- **collide** – see Fig 2



$$\boldsymbol{v_A = (x, y, z)}$$

$$\boldsymbol{v_B = (-x, y, z)}$$

$v_a = velocity\ before\ collision$

$v_a = velocity\ after\ collision$

In this case, the bubble collides with the face for which axis (-x, x) is perpendicular (the face's normal is parallel to this axis). To make the collision we just must change the sigh for the x coordinate.

We will also multiply the result by the bounciness of the bubbles. If it is 0 the bubble will not bounce back at all, if it is 1 it will look like the image on the left.

Fig **2**

Checking bubble-bubble collision is done by calculating the distance between the centers of the bubbles (Eq 2) and then comparing it with the sum of the radiuses of the bubbles.

$$(x_A + x_B)^2 + (y_A + y_B)^2 + (z_B + z)^2$$

Equation **2** – distance between 2 points in 3S space

Regarding bubble collision, we have two types:

- **pop collision** where both bubbles which collided become invisible and *dead*
- **merge collision** where the bubbles disappear like last time, but a new bubble will take their place (Fig 3).

Fig. **3**

For the **merging** to be smooth and natural we must consider the new bubble's radius which can be easily found as shown below (Eq 3):

$$\boldsymbol{V_C = V_A + V_B \Rightarrow r_C = (r_A^3 + r_B^3)^{\frac{1}{3}}}$$

$V_A = volume\ of\ the\ first\ bubble$

$V_B = volume\ of\ the\ scond\ bubble$

$V_C = volume\ of\ the\ bubble\ resulted\ from\ merging$

$r_A = radius\ of\ the\ first\ bubble$

$r_A = radius\ of\ the\ second\ bubble$

$r_C = radius\ of\ the\ bubble\ resulted\ from\ merging$

Equation **3**

To find of the resulting velocity we use the formula for perfect plastic collision (Eq 4). We preserve the momentum because we will alter the velocity of the resulting bubble in the next frame. It will be affected by gravity or by the force caused by the density difference, depending on the case.

$$\boldsymbol{v_c(t) = (v_A(t) \cdot r_A^3 + v_B(t) \cdot r_B^3)/(r_A^3 + r_B^3)}$$

$t =$ *current frame* (*at which the merging takes place*)

$v_A =$ *velocity vector of the first bubble*

$v_B =$ *velocity vector of the scond bubble*

$v_C =$ *velocity vectorof the bubble resulted from merging*

$r_A =$ *radius of the first bubble*

$r_A =$ *radius of the second bubble*

$r_C =$ *radius of the bubble resulted from merging*

Equation **4 –** formula for **perfect plastic collision**

We can use the same formula to find out the position in space of the new bubble just by substituting the volume vectors with the position vectors (Eq 5).

$$\boldsymbol{p_c(t) = (p_A(t) \cdot r_A^3 + p_B(t) \cdot r_B^3)/(r_A^3 + r_B^3)}$$

$t =$ *current frame* (*at which the merging takes place*)

$p_A =$ position *vector of the first bubble at frame t*

$p_B =$ position *vector of the scond bubble at frame t*

$p_C =$ position *vectorof the bubble resulted from merging at frame t*

$r_A =$ *radius of the first bubble*

$r_A =$ *radius of the second bubble*

$r_C =$ *radius of the bubble resulted from merging*

Equation **5 –** formula for **perfect plastic collision** adapted for the position vector

An improvement to the project would be making some of the bubbles split. It would be done by using the same method for merging, but backwards and with the help of a random number to split the bubbles in two different volumes.

Additional forces are optional, and they give the user more control of the movement of the bubbles. They can make it look like there is wind or other turbulences in the environment. If the bubbles hit one of the walls the force will have the corresponding coordinate 0 for that frame. For example, if there is a collision like in Fig 2 the forces will be ignored (equal to 0) for the x coordinate. They are not added to the bubble velocity directly as they would alter the bubble velocity for the rest of the animation. Instead, by adding just displacement obtained from them to the final displacement for each frame separately it gives the user the option to se for how long the new force will affect the animation.

When making this project I had two types of behavior in mind: **soap bubbles in the air** and **air bubbles in the water**. They can be achieved by changing the **environment type**.

In the case of **soap bubbles in the air** the environment will have gravitational acceleration, so we calculate the weight vector. To do so, we need to find out the mass of the bubble, which is possible because we have the density and the radius of the bubble (Eq 5). Next, we need to add it to the velocity for every frame (Eq 6). It will only move the bubble in the y direction and when the bubble hits the bottom face it will **pop** when wall collision is activated.

$$m = \varphi \cdot V$$

$m = mass$

$\varphi = density$

$V = volume$

$V \cong r^3 \cdot 4.19$

$r = radius$

Equation **6**

$$v(t) = v(t-1) - g \cdot m$$

$t = current\ frame$

$v = velocity$

$g = gravitational\ acceleration$

$m = mass\ of\ the\ bubble$

Equation **7**

In the case of **air bubbles in the water** we ignore the gravity, but we add into the equation the density difference between bubbles and water using Eq 8. In real life, the air density is much smaller than the water density and therefore they raise to the surface. However, I made the density of the bubble to take values bigger than the ones for the water too because they can resemble other round objects that sink instead of floating. To get the value of this force, In the other case we consider the difference between the density of the bubbles and the density of the water and then we multiply it by the volume of the sphere. Because I did not find a formula to calculate the acceleration caused by the difference in

density, I decided to follow my instincts and divide the result by 10. Therefore, the bubble goes up when its density is smaller than the one of the water and down if it is bigger. In this case the top face will make the bubbles pop, not the bottom ones.

$$v(t) = v(t-1) + \left(\varphi_{H_2O} - \varphi_B\right) \cdot V_B : 10$$

$$t = current\ frame$$

$$v = bubble's velocity\ vector$$

$$\varphi_{H_2O} = density\ of\ water$$

$$\varphi_B = density\ of\ the\ bubble$$

$$V_B = volume\ of\ the\ bubble$$

Equation **8**

In the end, to get the position where the bubble has to be moved, we take the final velocity for the current frame and divide it by the number of frames per second and then add this result to the previous position in space (Eq 9).

$$p(t) = p(t-1) + \frac{v(t)}{f}$$

$$t = current\ frame$$

$$p = possition\ vector$$

$$v = velocity\ vector$$

$$f = number\ of\ frames\ per\ second$$

Equation **9**

Because my project is mainly about procedural animation, it contains many iterations (for, while) and decision making (if, if…else). There are also a few relatively short functions which are, in fact used a lot, like **checkWallCollision** and **checkCollision** (used in **disperseBubbles** and in **startKeyframing**), **moveObject**, **popBubble**.

It was very helpful to make those smaller functions for simple tasks because they made the script easier to read and to develop, leaving space for more attention to detail, which I consider being the key when it comes to creating a realistic movement.

Although my project seems a bit incomplete at this stage because the bubbles do not change their shape, I plan on taking it further in my spare time because I got captivated by the idea of "recreating" what surrounds us. Even if it is still far from mimicking the real-world, I like the result as it is too. It makes me realize how amazing the world is and how much I can learn from it just by observing it carefully.

Here are a few frames from the animations done with my tool that I rendered using Arnold's aiStandardSurface.