Tutorial: Creating Speech Bubbles with the "Argo Bubbles" script

The **Argo Bubbles** script creates speech bubbles for talking characters in Visionaire Studio. If you want to use the script in your game, you don't have to read and understand this tutorial. Everything you need to know is explained inside the script.

For a deeper understanding of how the speech bubbles are created and how they replace the text displayed by Visionaire, keep on reading. This tutorial might help, if you want to modify the **Argo Bubbles** or build your own bubble (or non-bubble) script. Basic Lua knowledge is required. Note that the pieces of code are explained in a logical order, not necessarily in the order they appear in the final script.

Thanks to Visionaire user Turbomodus who worked out the basics. The Argo Bubbles are based on his work.

In case this file comes without the script itself, download it here (along with a small demo project):
https://wiki.visionaire-tracker.net/wiki/Compiled_Index_of_Lua_Scripts_for_Visionaire_Studio

1. Here's the plan

Our goal is to put a speech bubble graphic behind the text, whenever a character is talking. This graphic needs to adapt to the individual dimensions of that text. Additionally we want to add a pointer (that little handle at the bubble) pointing at the character.

What we will actually do is stop Visionaire from displaying the text at all. Instead we take it, create our bubble around it and then tell Visionaire to display the whole thing at the right position.

So we need to do the following (the numbers of the list do not correspond to the chapters of this tutorial):

1. Get each character text *just before* it gets displayed and save it
2. Prevent Visionaire from displaying the text
3. Create a speech bubble containing the saved text
4. Tell Visionaire to show that speech bubble
5. Tell Visionaire to hide the bubble as soon as the text is supposed to disappear

2. Get the text, hide and keep it

Whenever a text is about to be displayed in Visionaire, the `textStarted` event is triggered. By using the appropriate event handler, we can react to that event and execute a function. Let's call that function `show_argo_bubble`.

```
registerEventHandler("textStarted",
"show_argo_bubble")
```

The event automatically passes an object to our function. Since it is a text object, we call it `text`. Now before doing anything, we check whether this text is spoken by a character at all. Because if it isn't, we don't want to create a speech bubble and leave everything as it is. We can use the `Owner` property of the `text` object to check that.

If the text does belong to a character, we take it and save a table of three of the object's properties in our very own local table called `bubbles`:

- `Text`: the current text itself
- `Owner`: the name of the character
- `Background`: whether it's background text or not

Then we set `text.CurrentText` to an empty string, so Visionaire has nothing to display – no character texts will now show up anmyore.

```
local bubbles = {}

function show_argo_bubble(text)
 if text.Owner:getId().tableId == eCharacters then
  bubbles[text:getId().id] = {
   Text = text.CurrentText,
   Owner = text.Owner:getName(),
   Background = text.Background
  }

  text.CurrentText = ""
 end
end
```

The `bubbles` table holds all the texts that are supposed to be displayed in a speech bubble *right now*. Since it is possible for several characters to speak at the same time, this table may contain more than one item at once – hence it's a table. But most of the time there will probably be only one item. Or none.

A bubble shall be visible as long as the text is held in the `bubbles` table, so we need to remove it as soon as the text stops.

There is another event called `textStopped` which gets fired whenever a text is about to get hidden again. We can react to that event through an event handler, just as we did before. The `destroy_argo_bubble()` function deletes the text with the appropriate id from our `bubbles` table by setting it to `nil` and thus will stop displaying the bubble.

```
function destroy_argo_bubble(text)
 bubbles[text:getId().id] = nil
end

registerEventHandler("textStopped",
"destroy_argo_bubble")
```

3. "Infiltrate" the render pipeline

Until now, all we have done is hide the character text and save it in a table for as long as it's supposed to be displayed. Now we need to setup our own display routine.

While the game is running, the Visionaire player continuously re-renders the screen (about 50 times per second). We can add a custom draw function to this render process, which will take the `bubbles` table and display a speech bubble for each text item inside. So as soon as a text is saved into the table, it gets thrown into the rendering, and as soon as it's removed, the speech bubble will disappear again.

```
graphics.addDrawFunc("bubble_below_interface()", 0)
graphics.addDrawFunc("bubble_above_interface()", 1)
```

As you can see, we're not only adding one but two functions to the render process here:

- Regular character texts will be rendered on top of the game's interfaces (parameter 1): When the player opens the inventory and looks at an item, that speech bubble must overlay the inventory, of course, or you might not see it.
- Background texts however will be rendered below interfaces (parameter 0): Your cursor is visible when executing background texts. Cursors belong to interfaces and must show up in front of speech bubbles, not below them.

The two functions `bubble_below_interface()` and `bubble_above_interface()` are basically identical. They loop through the `bubbles` table and call our main bubble creation function `create_bubble()` (which we will work on in the next step) for each text item they find. The difference is: the first one only displays background texts, the second one only non-background texts.

```
function bubble_below_interface()
 for key, val in pairs(bubbles) do
  if val.Background then
   create_bubble(key, val)
  end
 end
end

function bubble_above_interface()
 for key, val in pairs(bubbles) do
  if not val.Background then
   create_bubble(key, val)
  end
 end
end

function create_bubble(key, val)
  -- this is where the magic happens...
end
```

4. Bubble creation

With that process set up, all that's left to do is create and draw the bubbles. This happens inside the `create_bubbles()` function, which receives a text item from the `bubbles` table as the `val` parameter.

The Argo Bubbles script offers the ability to define a default bubble style and custom styles, so you can switch between different bubble designs and positionings during the game. We'll skip that for now and focus on the speech bubble creation mechanism first. The handling of multiple styles will be explained in chapter 5.

For now we'll work with a single `bubble_style` table. These are options the game developer (you) has to define for his game when using the **Argo Bubbles** script. How should the bubbles look like, where should they appear …

```
local bubble_style = {
 align_h = "center",
 align_v = "top",
 bubble_offset_x = 0,
 bubble_offset_y = -25,
 color = 0xffffff,
 text_align = "center",
 padding = {
  top = 15,
  right = 20,
  bottom = 12,
  left = 20
 },
 file_bubble = "vispath:gui/bubble.png",
 ninerect_x = 20,
 ninerect_y = 15,
 ninerect_width = 30,
 ninerect_height = 20,
 file_pointer_bottom_right =
  "vispath:gui/bubble_pointer_br.png",
 file_pointer_bottom_left =
  "vispath:gui/bubble_pointer_bl.png",
 pointer_bottom_right_offset_x = -20,
 pointer_bottom_left_offset_x = 0,
 pointer_bottom_offset_y = 3,
 file_pointer_top_right = "",
 file_pointer_top_left = "",
 pointer_top_right_offset_x = 0,
 pointer_top_left_offset_x = 0,
 pointer_top_offset_y = 0
}
```

4.1 Character information

First we get the character (the owner of the text) and the text position Visionaire has calculated. This position – above the character's head – will be our reference position when calculating the position of the bubble. We save it in the *pos* variable.

```
local char = Characters[val.Owner]
local pos = graphics.getCharacterTextPosition(char)
```

Next step is determine the facing direction of the character. The positioning of the speech bubble as well as the choice which bubble pointer to attach may depend on where he looks (left or right).

However, if the horizontal alignment for the bubble (*align_h*) is set to "left" or "right", we will ignore the "real" facing direction of the character. Because with the bubble placed on one side of the character, it doesn't matter, which way he looks. The pointer always has to point in the opposite direction, towards the character (see examples 3 and 4 in the picture on page 6). That's why we override the *char_facing* variable in these cases, pretending the character always looks towards the bubble.

```
local char_facing = "right"

if bubble_style.align_h == "left" then
 char_facing = "left"
elseif bubble_style.align_h ~= "right" then
 if char.Direction > 90 and char.Direction < 270
 then
  char_facing = "left"
 end
end
```

4.2 Text and bubble dimensions

Now we calculate the text dimensions. That is height and width of the (rectangular) space the lines of text take up.

First we need to take care of line-breaks. Visionaire saves them as *
* tags – we have to replace them with "real" line-breaks (*\n*) using Lua's global substitution method *gsub*.

```
local txt = val.Text:gsub("<br/"..">", "\n")
```

You'll notice that we split the *
* string up into two pieces and immediately glued them together again with the two dot string operator (*..*). That doesn't seem to make much sense. However, the Visionaire editor treats *
* as a line-break, even when typed within a script. So if you do that, the editor messes up your script. Avoid writing the *
* string as a whole.

Next we use a special function to split our text into lines utilizing those line-breaks. Thus we get a table called *lines* holding our text line-by-line.

```
local lines = graphics.performLinebreaks(txt)
```

By looping through that *lines* table, we can determine the width (longest line) of our text through the *graphics.fontDimension()* function. The height is calculated using the font-size, the line gap (*VerticalLetterSpacing*), and the number of lines.

```
local text_dim = {x = 0, y = 0}

for k, line in ipairs(lines) do
 local tempdim = graphics.fontDimension(line)

 if text_dim.x < tempdim.x then
  text_dim.x = tempdim.x
 end
end

text_dim.y = #lines * (char.Font.Size +
char.Font.VerticalLetterSpacing) -
char.Font.VerticalLetterSpacing
```

By adding the padding values (as defined in *bubble_style*), we get the dimensions of the speech bubble (without pointer).

```
local bubble_dim = {x = 0, y = 0}

bubble_dim.x = text_dim.x +
 bubble_style.padding.right +
 bubble_style.padding.left
bubble_dim.y = text_dim.y +
 bubble_style.padding.top +
 bubble_style.padding.bottom
```

4.3 Bubble position

Now that we know how much space our bubble will take up, we may calculate its desired position. The script allows positioning through various style properties.

Horizontal bubble position

There are four horizontal alignment options (*align_h*) available in the script: "left" of the character, "right" of the character, "centered" above the character, and "char_facing", which aligns the bubble according to the direction the character is currently facing. That last one results in left or right, too. One may also define an offset (*bubble_offset_x*) to further adjust the position.

The initial position for our calculations is the default character text position, stored in the *pos* variable, which is measured from the top left corner of the *scene*. For drawing our bubble, we need to calculate its top left corner in relation to the *screen* (≠ *scene*) though, so – to also make it work for scrollable scenes – we first need to substract the current *ScrollPosition*. The rest of the math depends on which of the alignment options applies.

"Right" is the easiest, because that's where the bubble appears, if we just keep the initial position: the

bubble's left edge matches the character's position. If we want to align the bubble to the "left", we have to move it by its width from the initial position to the left. In order to "center" the bubble, we only have to move it half its width to the left.

```
if bubble_style.align_h == "right"
or (bubble_style.align_h == "char_facing"
and char_facing == "right") then
 pos.x = pos.x - game.ScrollPosition.x
elseif bubble_style.align_h == "left"
or (bubble_style.align_h == "char_facing"
and char_facing == "left") then
 pos.x = pos.x - game.ScrollPosition.x -
 bubble_dim.x
else -- center
 pos.x = pos.x - game.ScrollPosition.x -
 bubble_dim.x / 2
end
```

This position gets adjusted by the bubble offset. It is either added or substracted, depending on the facing direction of the character. By definition, positive values move the bubble away from the character (in the facing direction).

```
if char_facing == "left" then
 pos.x = pos.x - bubble_style.bubble_offset_x
else
 pos.x = pos.x + bubble_style.bubble_offset_x
end
```

To avoid drawing our bubble partially outside of the screen or too close to the edge, we check the distance to the left and right edges of the screen. If the bubble gets too far to the left, we change the position and put it as far left as possible. The same goes for the right edge. There's a `min_distance` setting available in our script to prevent the bubble from sticking to the edge.

```
if pos.x < min_distance then
 pos.x = min_distance
elseif pos.x > game.WindowResolution.x -
bubble_dim.x - min_distance then
 pos.x = game.WindowResolution.x -
 bubble_dim.x - min_distance
end
```

Vertical bubble position

We added two vertical alignment options (`align_v`): "top" and "bottom". This setting also defines, whether the bubble pointer will be placed at the bottom edge of the bubble, pointing downwards, or on the top edge, pointing upwards.

The math is basically the same as for the horizontal position. First we substract the current `ScrollPosition` from our initial position. In addition, we have to move the bubble upwards by its height, if the alignment is set to "top". Then we add the vertical offset. Positive values are defined to move the bubble down, so we don't have to differentiate between the alignments.

And finally we make sure the bubble is drawn onscreen by adjusting the position if needed.

```
if bubble_style.align_v == "bottom" then
 pos.y = pos.y - game.ScrollPosition.y +
 bubble_style.bubble_offset_y
else
 pos.y = pos.y - game.ScrollPosition.y -
 bubble_dim.y + bubble_style.bubble_offset_y
end

if pos.y < min_distance then
 pos.y = min_distance
elseif pos.y > game.WindowResolution.y -
bubble_dim.y - min_distance then
 pos.y = game.WindowResolution.y -
 bubble_dim.y - min_distance
end
```

4.4 Main bubble graphics

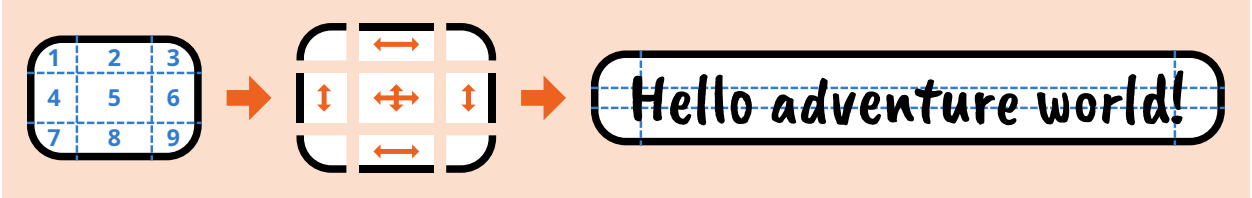
The game developer has to provide the graphics for the bubble and add the file path to the `bubble_style` table. We can now load this file.

```
local sprite =
graphics.loadFromFile(bubble_style.file_bubble)
```

Since each text has different dimensions, the speech bubble has to be variable in size. But just stretching and shrinking the graphic would distort it.

Instead we use the "ninerect" technique. That means, the graphic file gets split up into nine pieces (tiles), which are individually scaled and then reassembled to match the calculated bubble size. The advantage of this technique is that the corners won't get stretched but keep their original aspect ratio, and outlines keep their width, too (see the picture at the bottom of this page).

It also means, that there are limitations to what the bubbles can look like. It is not possible with this script to have elliptical or fancy-shaped bubbles. They need to be more or less rectangular, but may have rounded or chamfered corners.



The ninerect technique: the bubble graphic is split up, the tiles (except the corners) are scaled to match the desired dimensions, and everything is put together again.

The *graphics* object offers a special function for the ninerect technique. We need to tell it two things: how to split up the sprite and how and where to put it back together. The latter is defined in the *dest_rect* variable: it gets the speech bubble position and dimensions we just calculated. The splitting of the ninerect graphic depends on how the graphic had been created and thus has to be defined in the *bubble_style* table: "x" and "y" are width and height of the top left corner (tile no. 1), "width" and "height" mean width and height of the center tile (no. 5).

The *drawSpriteWithNineRect()* function expects two more parameters: a color value to tint the graphic and an alpha value. We added the tint option to our *bubble_style*, but did not make use of the alpha (both can be achieved by creating the graphics appropriately anyway).

```
local dest_rect = {
 x = pos.x,
 y = pos.y,
 width = bubble_dim.x,
 height = bubble_dim.y
}

local nine_rect = {
 x = bubble_style.ninerect_x,
 y = bubble_style.ninerect_y,
 width = bubble_style.ninerect_width,
 height = bubble_style.ninerect_height
}

graphics.drawSpriteWithNineRect(sprite, dest_rect,
nine_rect, bubble_style.color, 1.0)
```

At this point of development the drawing of the bubble itself is already working. It just misses the text and the pointer.

4.5 The pointer

Depending on the horizontal and vertical alignments of the bubble as well as the facing direction of the character, our script will attach one of four different pointers to the bubble, either pointing downwards right, downwards left, upwards right or upwards left. The game developer needs to provide those four graphic files (if he wants to make use of all of them). Our script has to decide, which file to use and where to put it.

Select and load the pointer

If the bubble is aligned to the top, we need the pointer pointing downwards and vice versa. The horizontal pointing direction (left/right) is determined by the facing direction of the character: if he looks left, the pointer points to the right and vice versa … unless the horizontal alignment of the bubble is set to "left" or "right". For these cases we have already forced the *char_facing* variable to face the bubble, even if the character actually looks the other way (see chapter 4.1).

Based on facing direction and vertical bubble alignment, we can load the appropriate pointer file and save it in our *pointer* variable. The game developer has the option of not having a pointer attached to his speech bubbles though. He may set the file path in his *bubble_style* to nil or attach an empty string. To avoid a script error, we wrap an if condition around our file loading code to check for that.

We provided each of the four pointers with its own horizontal offset setting. When loading the selected file, we also save this offset from the *bubble_style* table in the *pointer_offset* variable to use it later when positioning the pointer. Like for the bubble itself, positive offset values will move the pointer in the facing direction. That's why we reverse the sign of the value for right pointing pointers.

```
local pointer = nil
local pointer_offset = 0

if char_facing == "left" then
 if bubble_style.align_v == "bottom" then
  if bubble_style.file_pointer_top_right ~= nil and
  bubble_style.file_pointer_top_right ~= "" then
   pointer = graphics.loadFromFile(
    bubble_style.file_pointer_top_right)
   pointer_offset =
   -bubble_style.pointer_top_right_offset_x
  end
 else
  if bubble_style.file_pointer_bottom_right ~= nil
  and bubble_style.file_pointer_bottom_right ~= ""
  then
   pointer = graphics.loadFromFile(
    bubble_style.file_pointer_bottom_right)
   pointer_offset =
   -bubble_style.pointer_bottom_right_offset_x
  end
 end
else
 if bubble_style.align_v == "bottom" then
  if bubble_style.file_pointer_top_left ~= nil and
  bubble_style.file_pointer_top_left ~= "" then
   pointer = graphics.loadFromFile(
    bubble_style.file_pointer_top_left)
   pointer_offset =
   bubble_style.pointer_top_left_offset_x
  end
 else
  if bubble_style.file_pointer_bottom_left ~= nil and
  bubble_style.file_pointer_bottom_left ~= "" then
   pointer = graphics.loadFromFile(
    bubble_style.file_pointer_bottom_left)
   pointer_offset =
   bubble_style.pointer_bottom_left_offset_x
  end
 end
end
```

Horizontal pointer position

If we were able to load a pointer file, we continue with calculating its position. If not, we can skip the rest of the pointer code.

```
if pointer ~= nil then
 -- Calculate pointer position
 -- Draw the pointer
end
```

If our bubble is aligned to the right, we place the pointer at the leftmost position, i. e. at the left edge of the bubble (see examples 3 and 4 in the picture on the right). That's easy, because our x coordinate then matches the `pos.x` value of the bubble – we just have to add the `pointer_offset` variable.

However, if the character is standing in close distance to the right edge of the screen, the bubble may get shifted to the left (remember, we adjusted the bubble position to keep the `min_distance`). In that case the pointer might end up left of the character pointing away from him. If that happens, we set the pointer to the character's position instead.

```
local pointer_pos = {x = 0, y = 0}

if bubble_style.align_h == "right" or
(bubble_style.align_h == "char_facing" and
char_facing == "right") then
 pointer_pos.x = pos.x + pointer_offset

 if pointer_pos.x < char.Position.x -
 game.ScrollPosition.x + pointer_offset then
  pointer_pos.x = char.Position.x -
  game.ScrollPosition.x + pointer_offset
 end
end
```

The code for the left aligned bubble looks similar, just that we need to take the bubble and pointer widths into account when placing the pointer rightmost. The pointer width is a property of the `pointer` object.

```
pointer_pos.x = pos.x + bubble_dim.x -
pointer.width + pointer_offset
```

Last thing are the centered bubbles. They just get their pointers centered at the character's position.

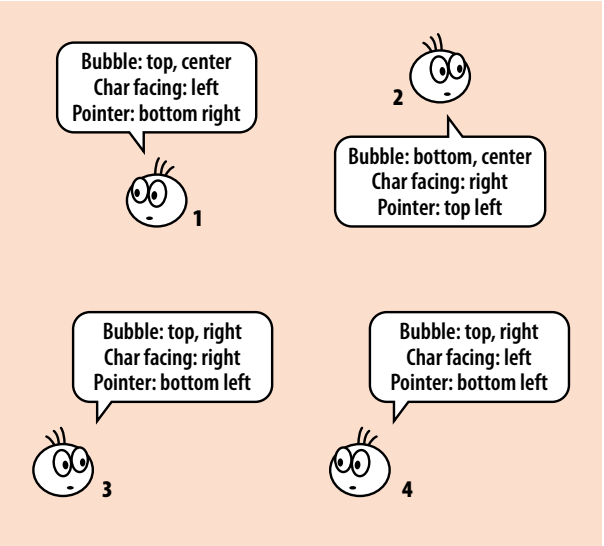
Here's the full code for the horizontal position:

```
local pointer_pos = {x = 0, y = 0}

if bubble_style.align_h == "right" or
(bubble_style.align_h == "char_facing" and
char_facing == "right") then
 pointer_pos.x = pos.x + pointer_offset

 if pointer_pos.x < char.Position.x -
 game.ScrollPosition.x + pointer_offset then
  pointer_pos.x = char.Position.x -
  game.ScrollPosition.x + pointer_offset
 end
elseif bubble_style.align_h == "left" or
(bubble_style.align_h == "char_facing" and
char_facing == "left") then
 pointer_pos.x = pos.x + bubble_dim.x -
 pointer.width + pointer_offset

 if pointer_pos.x > char.Position.x -
 game.ScrollPosition.x + pointer_offset then
  pointer_pos.x = char.Position.x -
  game.ScrollPosition.x + pointer_offset
 end
else -- center
 pointer_pos.x = char.Position.x -
 game.ScrollPosition.x + pointer_offset
end
```



Some bubble positioning scenarios with different pointers

Vertical pointer position

There are two possible values for the vertical alignment: top bubbles will have their pointers at the bottom, bottom bubbles will have them at the top. So we need the y coordinate of the top and bottom edges of the speech bubble. For pointers that go on top, we need to substract their height.

For both options there is a vertical offset value defined in the `bubble_style` table. That's because when you have outlines around bubble and pointer, you don't want to put the two graphics just next to each other. The pointer has to overlap the bubble in order to cover the bubble outline. That makes both graphics look like one object.

```
if bubble_style.align_v == "bottom" then
 pointer_pos.y = pos.y - pointer.height +
 bubble_style.pointer_top_offset_y
else
 pointer_pos.y = pos.y + bubble_dim.y -
 bubble_style.pointer_bottom_offset_y
end
```

Draw the pointer

Last thing to do is draw the pointer through the `drawSprite()` function. The `position` is a property of the `pointer` object itself.

```
pointer.position = {
 x = pointer_pos.x,
 y = pointer_pos.y
}

graphics.drawSprite(pointer, 1.0,
bubble_style.color)
```

4.6 Adding the text

Now we have the perfect empty speech bubble. It's at the right position and has the right size, nice pointer included – it just lacks the text.

The text is stored in the `lines` table. After setting the `font` property of the `graphics` object to match the character's font, we loop through that table and draw each line at the intended position.

```
graphics.font = char.Font

for k, line in ipairs(lines) do
 -- Calculate the position
 -- Draw the line
end
```

Again we have to do some position calculations. We know the position of the top left corner of the bubble, stored in the `pos` variable, and need to add the top and left padding, respectively, to get the position of our text box.

If there is only one line of text, it will fit perfectly in our bubble. If there are multiple lines, we have to take the `text_align` property of the `bubble_style` table into account.

Possible text alignments are "left", "right" and "center". If aligned to the "left", the line will start at the text box position. If it's "right", we have to move it by the difference of text box width and line width to the right. And for "centered" lines we move it half that way. The width of the current line is calculated through the `fontDimension()` function again.

```
local tempdim = graphics.fontDimension(line)
local text_pos = {x = 0, y = 0}

if bubble_style.text_align == "left" then
 text_pos.x = pos.x + bubble_style.padding.left
elseif bubble_style.text_align == "right" then
 text_pos.x = pos.x + bubble_style.padding.left +
 text_dim.x - tempdim.x
else -- center
 text_pos.x = pos.x + bubble_style.padding.left +
 (text_dim.x - tempdim.x) / 2
end
```

The vertical position depends on the counting of lines. The first line goes to the top, the second goes one line height plus one line gap below that and so on. The `k` parameter of the loop is counting the lines, starting with 1.

```
text_pos.y = pos.y + bubble_style.padding.top +
(k - 1) * (char.Font.Size +
char.Font.VerticalLetterSpacing)
```

The `drawFont` function expects integers for the position, so we round our values with `math.floor`.

```
graphics.drawFont(line,
 math.floor(text_pos.x),
 math.floor(text_pos.y),
 1.0
)
```

And that's basically it!

5. Custom bubble styles

In our bubble creation process we have worked with the `bubble_style` table, where the game developer defines the look and positioning of his bubbles. But the **Argo Bubbles** script offers a way to define more than one style.

The table we called `bubble_style` so far is actually called `default_style`. In addition to that, there is another table called `custom_styles`. This table (if not empty) may contain an unlimited number of style tables with the same properties as the `default_style` table. The difference is the additional `char` property, which is mandatory and has to match the name of a character in the Visionaire project. Only for this character will the particular style apply.

```
-- example data
local custom_styles = {
 {
  char = "Hero",
  align_h = "char_facing",
  pointer_bottom_left_offset_x = 20,
  pointer_bottom_right_offset_x = 20
 },
 {
  char = "Witch",
  align_v = "bottom",
  bubble_offset_y = 100
 },
 {
  char = "Hero",
  color = 0xfbefb6,
  file_pointer_bottom_left = ""
 }
}
```

As you can see in the example above, not all properties from the original `default_style` table are defined, but only those that differ from the default values. And you can see that multiple styles can be defined for the same character. The developer may switch the style during the game.

5.1 Rebuilding the custom styles table

We built the `custom_styles` table the way you see above, because we wanted to make it as easy to use for the developer as possible. For internal use in the script we create a new table called `c_styles` from it that looks a bit different.

In that new table we collect all styles of the same character, number them and put them in a table table with the character name as key. That way, we can later reference each style of a certain character by a number. And we fill up the styles with the missing values from the `default_style` table.

So the new table with the above example data would look like this:

```
-- example data
c_styles = {
 ["Hero"] = {
  [1] = {
   char = "Hero",
   align_h = "char_facing",
   pointer_bottom_left_offset_x = 20,
   pointer_bottom_right_offset_x = 20,
   -- filled up with the default properties
  },
  [2] = {
   char = "Hero",
   color = 0xfbefb6,
   file_pointer_bottom_left = "",
   -- filled up with the default properties
  }
 },
["Witch"] = {
  [1] = {
   char = "Witch",
   align_v = "bottom",
   bubble_offset_y = 100,
   -- filled up with the default properties
  }
 }
}
```

For creating the *c_styles* table from the *custom_styles* table, we loop through the latter, add a table for the character, if it doesn't already exist and another table inside it that gets a number. Then we loop through the *default_style* table and add all its properties to the new character table. Last step is loop through the current custom style and overwrite the default values with the custom ones.

Well, that sounds complicated, but it's difficult to describe. Have a look at the actual code …

```
local c_styles = {}

if custom_styles ~= nil then
 for key, styles in pairs(custom_styles) do
  local num_char_styles = 1

  if c_styles[ styles["char"] ] ~= nil then
   num_char_styles = #c_styles[ styles["char"] ] + 1
   c_styles[ styles["char"] ][num_char_styles] = {}
  else
   c_styles[ styles["char"] ] = {{}}
  end

  for k, v in pairs(default_style) do
   c_styles[ styles["char"] ][num_char_styles][k] = v
  end

  for k, v in pairs(styles) do
   c_styles[ styles["char"] ][num_char_styles][k] = v
  end
 end
end
```

5.2 Choosing the style

Before starting to create our bubble, we have to decide which bubble style to use: the default one or one of the custom styles. This must happen after setting the *char* variable (because we need to know whose custom styles to look for) and before setting the *char_facing* variable (because that depends on settings in the style). So the following code block belongs between the two blocks in chapter 4.1.

First we set our well-known *bubble_style* variable to the *default_style*. If there is no custom style defined at all or at least no custom style for that particular character, we're done. The default style will then be used.

If there is a custom style for the current character, we look for a Visionaire value called "argo_bubble" that comes with the character. The developer can use this to choose between multiple styles for the character by setting a number corresponding to the *c_styles* table. If that value does not exist, we take the first custom style for the character and put it in our *bubble_style* variable. If it does, we check for the number and get the corresponding custom style. If the value is set to a number that does not correspond to a style, we keep the default style. This is a way for the developer to have a character use both the default and a custom style during the game: just switch between an existing and a non-existing number.

```
local bubble_style = default_style

if c_styles ~= nil and
c_styles[char.name] ~= nil then
 if Characters[char.name].Values["argo_bubble"] ~=
 nil then
  if c_styles[char.name][ Characters[char.name].
  Values["argo_bubble"].Int ] ~= nil then
   bubble_style = c_styles[char.name][
   Characters[char.name].Values["argo_bubble"].Int ]
  end
 else
  bubble_style = c_styles[char.name][1]
 end
end
```

After that we continue with the table stored in the *bubble_style* variable, as we did in the whole of chapter 4.

6. Some handler options

To make it easier to integrate the **Argo Bubbles** script into a Visionaire project, we added the ability to deactivate the bindings to the event handlers.

```
local ab_bind_to_handlers = true -- or false
```

```
if ab_bind_to_handlers then
 registerEventHandler("textStarted",
 "show_argo_bubble")
 registerEventHandler("textStopped",
 "destroy_argo_bubble")
end
```

Since it is only possible to register event handlers once in a project, one may choose to do that elsewhere, not in this script. The *ab_bind_to_handlers* boolean is a convenient way to do this without messing around with the main code. The developer would then have to call *show_argo_bubble(text)* and *destroy_argo_bubble(text)* manually from that other place.

And for anyone who wants do do it the other way around, there is an option to add external functions to those two event handlers through this script.

```
function ab_on_text_started(text)
  -- add your functions here
end

function ab_on_text_stopped(text)
  -- add your functions here
end
```

```
function show_argo_bubble(text)
 ... -- see chapter 2 for rest of the function

 if ab_on_text_started ~= nil then
  ab_on_text_started(text)
 end
end

function destroy_argo_bubble(text)
 ... -- see chapter 2 for rest of the function

 if ab_on_text_stopped ~= nil then
   ab_on_text_stopped(text)
 end
end
```

Written by Einzelkämpfer, September 2022
Based on Argo Bubbles version 2.1.2

Sorry for any quirky English, I tried my best.

"The Argonauts" have created adventure games with Visionaire Studio for fun since 2020:

https://the-argonauts.itch.io/