Graphics utility functionality in Freecode creator

For executor version 3

Before you start working go to the settings tab in the Freecode creator and check that your executor version is correct. This guide is created for executor version 3, but some parts might be applicable for other versions as well.

Configuring the canvas

Configuration of the canvas is done through the settings in the Freecode creator. In the graphics there is an option to determine if the implementation should make use of multiple steps or a single step. If you plan on making multiple graphics updates then make sure this option is checked. There is also a background color option which determines what the default background color of the canvas will be. Lastly there is also a resolution option, which determines both the maximum indexes available for placing graphics (possible to use decimals), and what dimensions the canvas will have. If making use of multiple steps, the step time is further used to determine the time spent on each step.

Setting up the API

In the Freecode creator there is an API tab, where one can add, edit and remove API methods. For it to be possible for the implementation to call a method in the solution, or for a solution to call a method in the Implementation through the API, those methods must be defined here. The comments and information added here will be available to anyone taking part of your challenge or exercise. To see previews of how it will look for them you can look in the preview tab. In order for the solution and implementation to remain compatible with any changes to the names of methods, arguments, or types, all those changes have to be made through this page. Of course you can create any number of methods that are not defined here, but you will only be able to call them internally in that file.

The solution

In the Freecode creator there is a solution tab, this is so you will have the possibility of creating a default solution used for verifying that the implementation works. When a user will attempt to make their own solution, they will be given the boiler plate code you can see in the preview tab.

The implementation

In the implementation tab you can set up the actual code that is used to run the challenge/exercise. The behaviour of the different levels are defined, the canvas data is configured, and scores are set here.

Printing to the console

Print from the solution

To print from the solution to the console just use the regular print statement used by your language. In python3 it would be "`print('hello world')`".

Print from the implementation

In order to write to the console from the implementation one has to access either the console from the context, or one can access a console linked to a specific solution. Printing to specific solutions can be useful when creating a tournament where multiple solutions compete against each other.

```
self._context.console.log('Everyone can always see this')
self._context.console.debug('Everyone can see this during development')
solution.console.log('Only the tagged solution can see this')
solution.console.debug('The tagged solution can only see this during development')
```

If multiple logs will be done it can be useful to temporarily save the console object and use that one.

```
console = self._context.console
console.log('Everyone can always see this')
```

Accessing the canvas object

The canvas object which is used to create graphical objects is part of the context. To create a simple red circle placed 5 units into the canvas, sized 2 canvas units, one can use

```
self._context.canvas.new_circle(2, 'red', x=5, y=5)
```

To avoid writing “`self._context.`” every time one can temporarily store the canvas as a separate variable.

```
canvas = self._context.canvas
canvas.new_circle(2, 'red', x=5, y=5)
```

Creating graphical objects

There are 5 main types of elements that can be used in the canvas. Rectangle, circle, polygon, text, bitmap, and spritesheet. All of the elements are created by accessing their respective method in the canvas object.

All color properties are set using CSS compatible names, such as "#FF6347" or "red".

All rotation values are set using degrees. Rotating halfway through a full rotation is 180 degrees.

The z_index is used to determine which object should overlap which. A higher z_index means the object will be drawn on top of objects with a lower value if they overlap.

All positional arguments are either relative to the upper left corner of the canvas, or relative to origo of its own position. If x and y are not set they will both be 0 by default.

If the edge argument is available and used the object will be an outline of the object with no fill color.

Rectangle

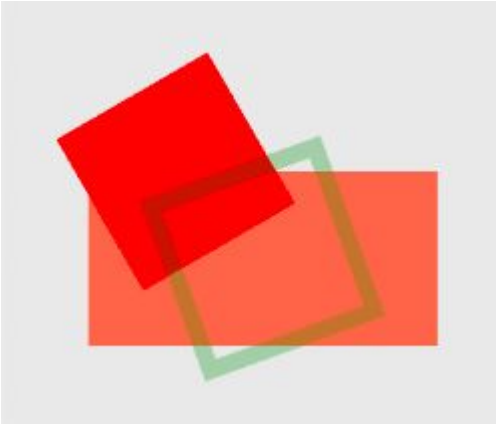
```
new_rectangle(width, height, color, x=0, y=0, scale_x=1, scale_y=1, opacity=1, rotation=0, z_index=0, edge=None, pivot_x=0, pivot_y=0)
```

Creates and returns a graphical object of the type rectangle.

Rectangles are defined by their width, height, and color. Their origo is in the middle of the shape and needs to be offset by half the width and height if you want to artificially move it to one of the corners.

Example:

```
canvas.new_rectangle(4, 2, '#FF6347', x=3, y=3)
canvas.new_rectangle(2, 2, 'red', x=2, y=2, rotation=60)
canvas.new_rectangle(2, 2, 'green', x=3, y=3, rotation=-20, edge=0.2, opacity=0.3)
```



Circle

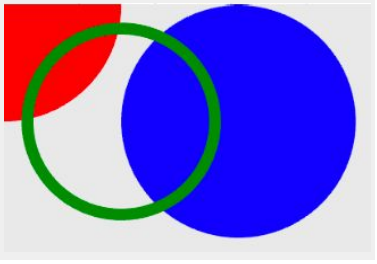
```
new_circle(radius, color, x=0, y=0, scale_x=1, scale_y=1, opacity=1, rotation=0,
z_index=0, edge=None, pivot_x=0, pivot_y=0)
```

Creates and returns a graphical object of the type circle.

Circles are defined by their radius and their color. Their origo is in the middle of the shape and needs to be offset by the radius if you want to artificially move it to one of the corners.

Example:

```
canvas.new_circle(1, 'red')
canvas.new_circle(1, 'blue', x=2, y=1)
canvas.new_circle(0.8, 'green', x=1, y=1, edge=0.1)
```



Polygon

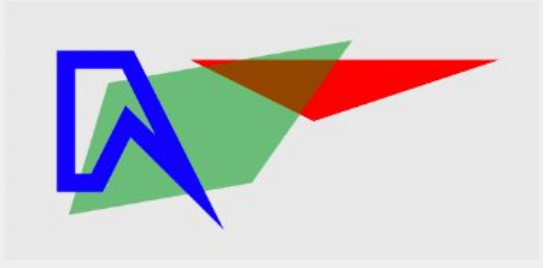
```
new_polygon(points, color, x=0, y=0, scale_x=1, scale_y=1, opacity=1, rotation=0,
z_index=0, edge=None, pivot_x=0, pivot_y=0)
```

Creates and returns a graphical object of the type polygon.

Polygons are abstract shapes defined by a list of coordinates and a color. The list contains pairs of x and y coordinates but without any special separation, e.g. [x1, y1, x2, y2, x3, y3].

Example:

```
canvas.new_polygon([0, 0, 5, 0, 2, 1], 'red', x=3, y=1, z_index=1)
canvas.new_polygon([0, 0, 0, 2, 0.5, 2, 1, 1, 2, 2, 1, 0], 'blue', x=1, y=1, edge=0.3,
z_index=3)
canvas.new_polygon([0, 0, -2, 2, 2, 2, 3, 0], 'green', x=4, y=3, opacity=0.5, rotation=170,
z_index=2)
```



Text

```
new_text(text, size, color, x=0, y=0, text_align='left', scale_x=1, scale_y=1, opacity=1, rotation=0, z_index=0, pivot_x=0, pivot_y=0)
```

Creates and returns a graphical object of the type text.

Text is defined by the text, size, and color. Origo is to the left of the text by default but can be changed with the `text_align` argument. By default `text_align` is "left", other possible values are "right", and "center".

Example:

```
canvas.new_text('default left alignment', 0.8, 'blue', x=7, y=1)
canvas.new_text('alignment also moves origo', 0.8, 'black', x=7, y=2, text_align='center')
canvas.new_text('right alignment', 0.8, 'red', x=7, y=3, text_align='right')
```

Bitmap

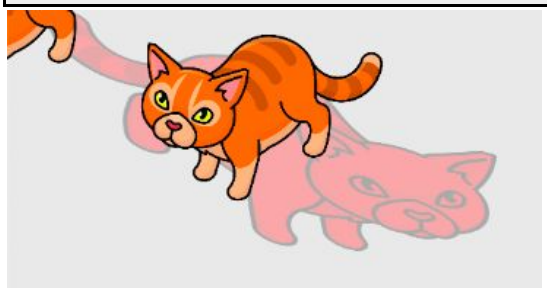
```
new_bitmap(width, height, image, x=0, y=0, scale_x=1, scale_y=1, opacity=1, rotation=0, z_index=0, color='white', pivot_x=0, pivot_y=0)
```

Creates and returns a graphical object of the type bitmap.

Width and height defines the size of the image, and will stretch the image if the original dimensions are not the same. The image argument should be the name of the image as it is written in the graphics tab.

Example:

```
canvas.new_bitmap(4, 4, 'striped_cat.png')
canvas.new_bitmap(4, 4, 'striped_cat.png', x=4, y=1.5)
canvas.new_bitmap(8, 4, 'striped_cat.png', x=4, y=2, rotation=20, color='red', opacity=0.3, z_index=-1, scale_x=-1)
```



Sprite sheet

```
new_spritesheet(width, height, image, animation='default', frame_index_start=None,
frame_index_end=None, x=0, y=0, scale_x=1, scale_y=1, opacity=1, rotation=0,
z_index=0, color='white', pivot_x=0, pivot_y=0)
```

Creates and returns a graphical object of the type sprite sheet, intended to be used for animations.

Width and height defines the size of the object to be drawn, if the dimensions are not the same in the sprite sheet then the image will be stretched. Each individual image in an animation should be the same size or they will stretch weirdly. The image argument should be the name of the sheet as it is written in the graphics tab.

Spritesheets need to be generated from individual images with a separate image atlas defining where in the sheet each individual image is (atlas needs to be compatible with pixi-js). A free tool to do this is https://github.com/krzysztof-o/spritesheet.js. In the freecode platform sprite sheets are mainly intended to be used for animations. When adding a sprite sheet and its atlas it is possible to automatically generate a default animation which will be defined by the last available number in the individual images names. It is also possible to personally define other animations in the same sheet and they can be utilized by setting the animation argument when creating the new_spritesheet in the code.

Each animation is defined as a list of frame names, using the frame_index_start and frame_index_end arguments it is possible to only run part of an animation, or to reverse the animation.

Example:

```
canvas.new_spritesheet(2, 2, 'weird_sheet.png', x=1, y=1)
canvas.new_spritesheet(2, 2, 'weird_sheet.png', x=3, y=1, frame_index_start=2,
frame_index_end=0, scale_x=-1)
canvas.new_spritesheet(1.5, 2, 'weird_sheet.png', x=5, y=1, animation='just_display_guy')
```

The "weird_sheet.png" used in the example only contains 3 very different images so it will be easy to see the difference in a static context, normally you want to use variations of the same image in each sheet in order to get nice animations.

Manipulating graphical objects

Once a graphical object has been created there are various methods that can be used to manipulate them. All graphical objects have all methods, but they might not work for all types. Such as why rotate a circle, or why set a text on something that is not a text?

The time argument is always a percentage ranging from 0 to 1, and represents when during a step the update will occur. For some operations it is necessary to first set the object to its original value and then to the new one. For a smooth movement you need to set the current position at time 0.0, and to the new position at the desired time. If you want to make an object suddenly change opacity at a specific time, then you need to set it to the current opacity just before making the change, or the change will be gradual from the start of the step time.

```
get_name()
```

Returns the name of the object. The name will be generated as a unique identifier.

```
set_x(x, time=None)
```

Sets the x coordinate of the graphical object (relative to the upper left corner).

```
get_x()
```

Returns the x coordinate of the graphical object (relative to the upper left corner).

```
set_y(y, time=None)
```

Sets the y coordinate of the graphical object (relative to the upper left corner).

```
get_y()
```

Returns the y coordinate of the graphical object (relative to the upper left corner).

```
set_position(x, y, time=None)
```

Sets the x and y coordinate of the graphical object (relative to the upper left corner).

```
get_position()
```

Returns the x and y coordinate of the graphical object (relative to the upper left corner).

move(x=0, y=0, time=None)

Changes the x and y coordinate of the graphical object by the specified amount, relative to the objects previous coordinates. For a smooth movement animation you first need to use `move(0, 0, 0.0)` followed by `move(1, 0, 1.0)`, or it will just set the new position.

set_z_index(z, time=None)

Sets the z index of the graphical object (higher z index objects are on top of lower ones).

get_z_index()

Returns the z index of the graphical object (higher z index objects are on top of lower ones).

set_scale_x(scale_x, time=None)

Sets the x scale of the graphical object (1 being default and 2 being twice the width). Useful for changing a pictures x orientation without setting a negative width.

get_scale_x()

Returns the x scale of the graphical object (1 being default and 2 being twice the width).

set_scale_y(scale_y, time=None)

Sets the y scale of the graphical object (1 being default and 2 being twice the height). Useful for changing a pictures y orientation without setting a negative height.

get_scale_y()

Returns the y scale of the graphical object (1 being default and 2 being twice the height).

set_scale(scale, time=None)

Sets the scale of the graphical object, both x and y (1 being default and 2 being twice the width and height).

get_scale()

Returns the x and y scale of the graphical object (1 being default and 2 being twice the width and height).

set_color(color, time=None)

Sets the color of the graphical object (CSS compatible colors, e.g. #FF4D4D or red).

```
set_opacity(opacity, time=None)
```
Sets the opacity of the graphical object (0 being completely transparent and 1 being opaque).

```
get_opacity()
```
Returns the opacity of the graphical object (0 being transparent and 1 being opaque).

```
set_rotation(rotation, time=None)
```
Sets the rotation of the graphical object (in degrees).

```
get_rotation()
```
Returns the rotation of the graphical object (in degrees).

```
rotate(rotation, time=None)
```
Rotates the graphical object by the specified amount, relative to the objects previous rotation (in degrees).

```
set_pivot(x=0, y=0, time=None)
```
Set the point around which an object should rotate. (relative to the regular origo)

```
get_pivot()
```
Return the point around which an object should rotate. (relative to the regular origo)

```
set_text(text, time=None)
```
Sets the text of the graphical object (only available on text objects).

```
get_text()
```
Returns the text of the graphical object (only available on text objects).

```
destroy()
```
Destroys the graphical object and removes it from the canvas (irreversible). If you try to update an object after it has been destroyed your implementation might crash.

Manually add animation step

It is possible to manually add animation steps without going through the natural step cycle. This can be useful if you want to perform an extra step such as drawing a final score or perform some elaborate animation.

```
canvas.finish_step()
```

All graphical operations performed after finish_step will occur in a new step.