THE CANVAS MODE: RAPID PROTOTYPING FOR THE DECIBEL SCOREPLAYER

Aaron Wyatt
Monash University
Sir Zelman Cowen School of Music

Lindsay Vickery
Edith Cowan University
Western Australian Academy of Performing Arts

Stuart James
Edith Cowan University
Western Australian Academy of Performing Arts

ABSTRACT

This paper talks about recent developments in the Decibel ScorePlayer project, notably the introduction of a canvas mode that allows for other applications to send drawing commands to the player via OSC. It outlines the object model that has been developed to allow for the creation of hierarchies of drawn objects, the commands that can be used to create and control these, and the way in which scores can be developed to take advantage of this new mode. It is hoped that this addition will give composers the flexibility to experiment with new score paradigms while being able to leverage the existing strengths of the platform.

1. INTRODUCTION

The Decibel ScorePlayer was always designed to be a modular platform (Wyatt and Hope 2013). While the main player window is responsible for controlling the user interface and networking functions, the task of drawing the score is passed off to other modules which can be swapped in and out to allow for the implementation of many different types of scores, from simple scrolling scores to far more complicated, non-linear ones. On a more open platform, additional rendering modules could easily be compiled as libraries to be added by the user as necessary, but unfortunately the highly closed off nature of iOS (and in fact most tablet computing environments) means that this is simply not possible[1]. The only way for new types of scores to be implemented is for the necessary rendering code to be included in the main package that is made available from the App Store. The creation of a canvas mode for the ScorePlayer is an attempt to overcome this limitation by allowing composers to directly control the drawing surface of the player from an external application using commands sent via the Open Sound Control (OSC) protocol (Wright 2002; Hope, Wyatt and Vickery 2015). In this way, new score paradigms can be quickly developed, prototyped, and even distributed without the need for any code to be accepted into the App Store. This paper will discuss the first publicly released implementation of this new feature, detailing how to interface with it from outside the app, and outlining possible future paths for further development.

2. OBJECT MODEL

While an early proof of concept version of the canvas mode had been thrown together in 2016 during an exchange in Hamburg between the *Decibel New Music Ensemble* and the *ZM4 Research Group x* (James et al. 2017), this version was very limited in scope and hadn't been designed in a way that allowed for easy future expansion. While layers could be created and removed, and images could be loaded into them, there was no ability to create layer hierarchies, and little provision for the implementation of other types of objects. The revised canvas mode overcomes these shortcomings by having a much more strongly defined object model, where commands are addressed to specific objects, and where objects can be placed within other, parent objects.

In its current iteration, the canvas mode implements six different types of objects: layers, scrollers, text, glyphs (a special case of the text object used as a convenient shortcut to display symbols from the bravura music font (Steinberg Media Technologies GmbH. 2018)), staves, and lines. With the exception of lines, which are simply drawn within the coordinate space of their parent object, all of these objects can also be used as containers for other objects. Changing the position of the parent object on the canvas will move all of the objects contained within it as one. Likewise, changing the opacity of the parent object will affect the rendering of the entire group, and removing the parent removes all of the child objects along with it. This makes it possible to very easily manipulate large groups of objects using a relatively small number of network commands.

While some of the more complex objects will be discussed in more depth later, the most basic object is the layer object. It consists of a simple rectangular region on the canvas which is empty when first created, but which can be set to contain either a flat colour or an image. As with most of the objects, its location on the canvas or within its parent object's coordinate space is made with reference to its upper left corner. Figure 1 shows the result of placing a red layer at (10,10) on the canvas, and an image layer at (20, 20) on the red layer. An important thing to note is that the contents of a child layer can spill over the boundary of a parent layer, although there may

[1] While it is technically possible to dynamically load a library in iOS in some circumstances, attempting to do so would see an app rejected from the App Store (DeVille 2104).

be a command added in future releases to allow the composer to change this behaviour. Until then, if you want to prevent this from happening, the easiest workaround is to instead use a scroller as a container without setting it in motion. This way the contents will be masked to the boundary of their parent.

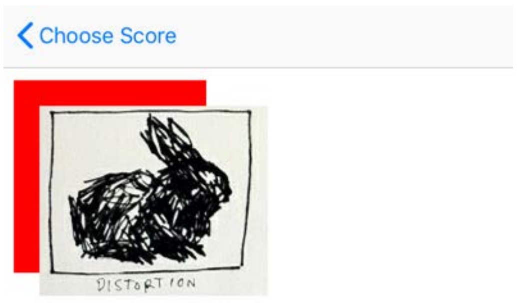

Figure 1. Nested layers in the player.[2]

It is possible for individual parts to be created so that each musician can see their own, specific material. The creation command of all objects, regardless of their type, requires the assignment of a unique name, and an assignment to one of these parts. (The number of available parts is determined in the score file, as described in the next section.) Anything assigned to part number 0 will appear on all of the parts. Unless a parent object is set to appear on all parts in this manner, any child object will inherit the path number setting from its parent object, regardless of the part that the composer attempts to assign to it. Moving between these parts in the ScorePlayer itself is as simple as swiping up or down. (This is the same mechanism used to switch between parts in other types of scores, so it is one that should be familiar to the end user.)

3. SCORE FILE

As with other score types, scores that use canvas mode consist of a standard zip file with its extension changed to .dsz (Wyatt and Hope 2013). This file contains all of the image resources needed by the score, in jpeg or png format, as well as a couple of xml files (W3C 2013) that define the score's metadata, and set any additional options. This file is then imported into the player via iTunes' file sharing feature. The main xml file that defines the score is named opus.xml and a typical example of one is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE opus SYSTEM "opus.dtd">
<opus>
	<score>
		<name>Canvas Test</name>
		<composer>Aaron Wyatt</composer>
		<type>Canvas</type>
		<duration>0</duration>
		<prefsfile>canvas.xml</prefsfile>
	</score>
</opus>
```

[2]The bunnies used in the canvas test score have been taken from Kim Krans' artwork for the cover of Tape Op. issue #88 (Krans 2012).

Most of the settings are common to other score types and are fairly self explanatory, but there are a couple of things worth noting. Obviously, the <type> is set to "Canvas", but the other point of interest is the <duration> tag: this can either be set to zero or any number of seconds. If zero (or negative), the player only displays the Reset button to the user, and the navigation bar and status bar remain visible. If set to a positive value, then the Play button is also displayed, and pressing it sends the usual /Control/Play command over the network, starts the clock, and hides the navigation bar until the specified time limit is reached. The obvious advantage of this is that it makes a larger drawing surface available, even if a composer has no intention of making use of the timing functions of the player. (The size of the canvas is 1024x768 when in landscape mode, and the status and navigation bar clip 70 pixels off this height.)

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE canvas SYSTEM "canvas.dtd">
<canvas>
	<parts>2</parts>
</canvas>
```

The preferences file pointed to by the opus.xml file only has a limited number of possible options. The most important defines the number of available parts that can be drawn to. (In the player, you can move between these by swiping up or down, just as you do with other types of score.) The only other option is the <clearonreset> setting. This determines whether the canvas is cleared by pressing the Reset button in the player. If undefined, it is set to "yes."

4. COMMAND STRUCTURE

Once connected to the player, drawing commands can be sent from an external device via OSC. All of these have an OSC address starting /Renderer/Command, with the name of the object that you wish to manipulate, and the command that you wish to send to it being added as the final two components. So, for example, to add a layer to the canvas you would send a command with the address "/Renderer/Command/canvas/addLayer" to the player. The arguments would include the name of the layer to be created, the part for it to be assigned to (or 0 for the layer to be placed on all parts) and then coordinate data. A full list of drawing commands can be found in the appendix at the end of this paper.

While any application that can send and receive OSC messages can be used to control the ScorePlayer in canvas mode, a reference Max patch is available from the Decibel New Music website that can be used for this purpose (Decibel New Music 2018). Additionally, a library has been developed to allow externals to be quickly and easily written in python, with a plan towards releasing it for distribution via the Python Package Index, or PyPI for short (Python Software Foundation 2018). Both of these solutions use Bonjour (Apple Inc 2013) for service discovery, and are able to find any iPads running the ScorePlayer on the local network, as long as multicast traffic is not blocked.[3]

4.1. Creating Externals in Python

The python scoreplayer_external library defines two python classes: scorePlayerExternal and scoreObject. The first class is used to make a connection to the iPad, opening a UDP listening socket and letting the iPad know which port to send its replies to. The second object is designed to represent the objects that populate the canvas. It acts as a wrapper so that OSC commands can be sent in a python like way. So a command like /Renderer/Canvas/clear becomes simply canvas.clear().

Connecting to an iPad requires only a few lines of code, as demonstrated below:

```
#!/usr/bin/env python
from scoreplayer_external import scoreObject,
scorePlayerExternal
from threading import Event

finished = Event()
external = scorePlayerExternal()
external.selectServer()
canvas = external.connect(onConnect)
finished.wait()
external.shutdown()
```

After creating a new scorePlayerExternal object, the selectServer() method checks for any iPads on the network and presents the following prompt:

```
Choose an iPad to connect to
1: bojack (Canvas Test)
Or
2: Refresh List
Enter Selection:
```

Each line lists the device name of an iPad found running the ScorePlayer, along with the name of the score that it currently has loaded. Upon making a selection, the connect(connectionHandler) method is called. This passes a function to the method as a variable, which is run once the scorePlayerExternal receives a positive response from the iPad. The method also returns a scoreObject named "canvas" which can then be used as a starting point to spawn more objects. The penultimate line keeps the script running until the onConnect function signals to the main thread that it has finished, at which point it's safe to shut down the external.

If it's not possible to use Bonjour for service discovery, and you know the IP address of the iPad that you wish to connect to, then in place of the selectServer() and connect(connectionHandler) methods you can call the connectToAddress(address, port, connectionHandler) method. This also returns the appropriate scoreObject pointing to the canvas.

The connection handler is where the bulk of the drawing code should reside. The following sample connection handler code clears the canvas, creates a basic scroller (called 'scroll' placed in part number 1) into which an image is loaded, and places a bass clef onto the scroller. The line, placed on the canvas instead of the scroller, remains stationary and acts as a playhead.

```
def onConnect():
    canvas.clear()
    scroll = canvas.addScroller('scroll', 1, 10,
10, 300, 200, 500, 20.0)
    scroll.loadImage('modulation.png')
    line = canvas.addLine('line', 1, 25, 10, 25,
210, 5)
    clef = scroll.addGlyph('clef', 1, 50, 100)
    clef.setGlyphSize(72)
    clef.setColour(0, 0, 128)
    clef.setGlyph('fClef')
    input("Press Enter to start...")
    scroll.start()
    finished.set()
```

After the initial set up is completed, the script waits for a keyboard prompt from the user before setting the scroller in motion. As soon as that is done, it sets the finished event so that the main thread can finally complete and the script can exit. Below is the result of the script as seen in the ScorePlayer.

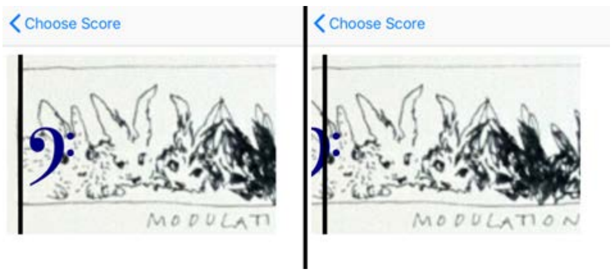

Figure 2. The result of the above code on initial set up and just after setting the scroller in motion.

A note about z-order: when objects are created they are placed above existing objects that share the same parent object. This is why, in the above example, the playhead line appears above the scroller object. As the clef glyph belongs to the scroller, it renders below the playhead, just like its parent. There is currently no way to change where objects are located in the stack once they have been created, but it

[3]I'm looking at you, Monash eSolutions.

should be relatively trivial to implement a moveAbove(object) and moveBelow(object) method in future releases.

5. ANIMATION FEATURES

While the scroller is the most sophisticated animated component of the player that is currently available, it is possible to apply simple animations to most of the other object types. The "move" and "fade" commands allow you to specify a change in an object's location or opacity that will occur over a specified period of time. The fade command can be applied to any object, while the move command can be applied to any object except for lines. If you need to have a line move across the screen, you can achieve this by drawing the line into a layer, and then applying an appropriate move command to that.

The scroller object allows for a larger image to be scrolled horizontally at a set rate through a fixed viewing window. The viewing window itself is defined by the usual position and size coordinates that are passed to the object on creation. A few additional parameters, also passed at creation, are used to define the behaviour of the scroller while animated. The scrollerWidth parameter determines the width of the layer to be scrolled through the viewing window, while the scrollerSpeed defines the rate at which this occurs in pixels per second. Setting a negative value for scrollerSpeed causes the scroller content to move backwards, to the right of screen. These values can be changed at any time, and the scroller can be set in motion or stopped using the "start" and "stop" commands.

As the scroller layer moves towards the limits of its motion, any material underneath the scroller will become visible. (This is represented by the shaded area in Figure 3.) Once the layer makes it outside the frame of the viewing window, animation is stopped and the layer remains fixed, just out of view. The animation state of the scroller isn't changed though: changing the scrollerPosition value so that the layer is brought back into view will resume animation without the need to send an additional "start" command. Changing the scrollerSpeed to a negative value in this case would also cause animation to automatically resume, with the contents of the scroller moving back in the opposite direction.

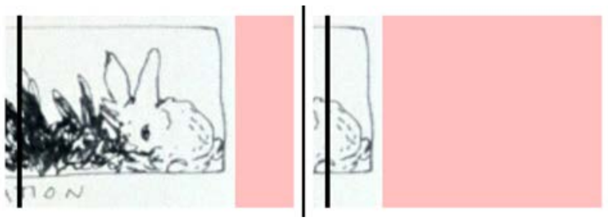

Figure 3. The scroller object as the scroller layer moves towards the limit of its motion.

6. THE STAVE OBJECT

The stave object brings limited access to traditional notation to the canvas mode. Since the ScorePlayer is geared more towards graphical notation, a greater emphasis is placed on the proportional placement of notes (particularly useful when scrolling a stave past a playhead) than on traditional bar structure. The size of the object defines the length of the stave lines and the height from the top stave line to the bottom one. Placing one or more clefs on the stave allows for the easy placement of notes using a syntax based on scientific pitch notation (C4 is middle C), with some additional modifiers: # is a sharp, + is quarter sharp, - is a quarter flat, b is flat, and n is a natural. So A#+4 would be the A three quarter sharp above middle C.

Notes and clefs are placed by specifying their position in pixels from the left hand edge of the stave object. At present, the placement code isn't particularly smart: the player will do exactly what it's told, regardless of whether that means there will be a collision of notes or accidentals. Future releases may do something to address this, at least on the accidental front. The option to honour the exact placement of noteheads, no matter how crowded the stave may be, should be maintained in the future. (Most likely as the default option to retain backwards comparability with existing scores.)

Following is some example python code that shows the basic usage for the stave object:

```python
def onConnect():
    canvas.clear()
    stave = canvas.addStave('stave', 0, 10, 100,
600, 72, 2)
    stave.setClef('treble', 40)
    stave.addNotehead('C#+4', 120, 1)
    stave.addNote('Gbb3', 200, 32)
    stave.setClef('alto', 400)
    stave.addNotehead('D-5', 500, 0)
    finished.set()
```

The addNote and addNotehead commands have a very similar syntax: the first two arguments are the note to be drawn and it's horizontal position along the stave respectively. The final argument for addNotehead is an optional argument that defines whether the notehead is filled or not. (The default is that it is filled.) The addNote command takes the note duration as it's final argument, with 0 being a breve, 1 being a semibreve, 2 a minim, 4 a crotchet, 8 a quaver and so on. If the size of a stave is changed, all of the glyphs are scaled accordingly, with the horizontal position of notes preserved as is: no scaling is performed along the horizontal axis. The output of the above code is shown below in Figure 4.

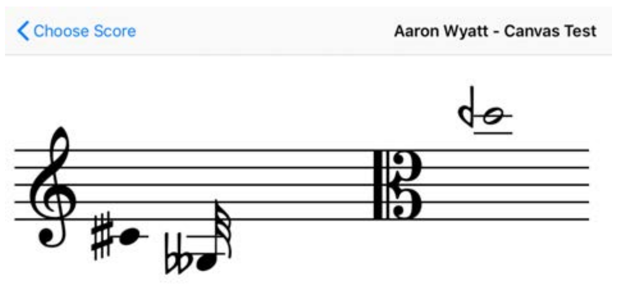

Figure 4. The stave object.

In addition to the methods included as part of the stave object, it's possible to manually place musical symbols on the canvas using glyph objects. These are a special case of the text object: instead of displaying a string, they display a lone character from the bravura music font. The type of glyph to display can be selected by using the setGlyph command, which takes a single argument: the canonical name of the desired glyph as defined by the SMuFL standard (Spreadbury and Piéchaud 2015). Currently only a limited set of glyphs are implemented, but this can easily be expanded in the future. The geometry of the glyphs is designed to make them as easy to use as possible. The size of the glyph object should be set to the height of a stave that it is meant to fit on, and the position of the glyph is defined not by its top left corner as with other objects, but by a sensible middle point. In the case of notes, this is the centre of the notehead. For clefs, it is at the height of the note defined by the clef (C, F or G). Some examples of this can be seen in Figure 5 below: the centre of the crosshairs points to the location of the object as defined by its position coordinates, and the horizontal lines either side of the centre show the distance between stave lines.

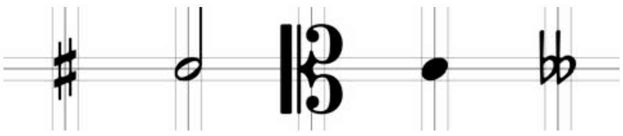

Figure 5. A sample of some of the available glyphs showing their centre points.

Any glyphs that haven't been implemented yet can of course be placed on the score using a simple text object instead, sent the appropriate unicode character. The only drawback to this being that placement of the object won't be quite as simple. That said, it should be possible to find an appropriate vertical centre point at a y value of twice the height of the font.

7. CONCLUSIONS

While the canvas mode is still in a relatively early stage of development, there should already be sufficient drawing commands available to make it a useful tool to those who want to extend the abilities of the ScorePlayer beyond the extant score types. One of the most commonly asked questions involving the ScorePlayer that the members of Decibel have received over the years has been whether it's possible to change the speed of a scrolling score at certain points in a work. Now, with the more fine grained controls offered by the canvas mode, it is. Those who've felt constrained by the conditions imposed by the built in forms now have the freedom to experiment, and with the existence of a python library, this should be achievable with minimal programming experience. While greater technical knowledge is required to take advantage of these new abilities, the learning curve shouldn't be too steep, and it is hoped that this paper, along with the resources that it references, will serve as a solid starting point for those who wish to tinker.

8. REFERENCES

Apple Inc. "Bonjour Overview." Last Modified April 23 2018. https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/NetServices/Introduction.html.

Decibel New Music. "The Decibel ScorePlayer." Accessed October 26, 2018. http://www.decibelnewmusic.com/decibel-scoreplayer.html.

DeVille, D. "Dynamic linking on iOS." Last modified April 2, 2014. http://ddeville.me/2014/04/dynamic-linking.

Hope, C., Wyatt, A., and Vickery, L. "The Decibel ScorePlayer: New Developments and Improved Functionality." *Proceedings of the 2015 ICMC Conference.* Denton, USA: International Computer Music Association, pp. 314-317.

James, S., Hope, C., Vickery, L., Wyatt, A., Carey, B., Fu, X., and Hadju, G. 2017. "Establishing connectivity between the existing networked music notation packages Quintet.net, Decibel ScorePlayer and MaxScore." *Proceedings of the Tenor 2017 Conference*. A Coruña, Spain: TENOR, pp. 171-183.

Krans, K. "How a Bunny Sounds: About." Accessed October 26, 2018. https://howabunnysounds.com/pages/about.

Python Software Foundation. "Python Package Index." Accessed October 26, 2018. https://pypi.org/.

Spreadbury, H., and Piéchaud, R. "Standard Music Font Layout (SMuFL)." *Proceedings of the Tenor 2015 Conference*. Paris, France: TENOR, pp. 147-154.

Steinberg Media Technologies GmbH. "SMuFL: Standard Music Font Layout." Accessed October 26, 2018. https://www.smufl.org/fonts/.

WC3. "Extensible Markup Language (XML) 1.0 (Fifth Edition)." Last modified February 7, 2013. https://www.w3.org/TR/xml/.

Wright, M. "The Open Sound Control 1.0 Specification." Last modified March 26, 2002. http://opensoundcontrol.org/spec-1_0

Wyatt, A., and Hope., C. 2013. "Animated music notation on the iPad (or: Music stands just weren't designed to support laptops)." *Proceedings of the 2013 ICMC*

Conference. Perth, Australia: International Computer Music Association, pp. 201-207.

9. APPENDIX

Here is a full list of commands for the canvas mode of the ScorePlayer. This is accurate for version 1.9.2 of the software. Future versions will add more commands, but should retain backwards compatibility with this current list.

If sending OSC messages directly, remember to prepend these commands with /Renderer/Command/*objectname*. If using the python library these are the methods called on your object. In python, any numerical values given as arguments will automatically be cast to the variable type expected by the player. In other environments, you will have to manually take this into consideration. All arguments are either strings (s), integers (i), or floats (f). Optional variables are listed with their defaults values.

addGlyph(name(s), part(i), x(i), y(i), glyphSize(f)=36.0)
Applies to: canvas, layer, scroller, text, glyph, stave
Adds a music symbol from the bravura font to the object. Name must be a unique identifier: the glyph won't be created if an object of that name already exists. Part specifies the part to be drawn into, or 0 for the object to be placed in all parts. (If the parent object doesn't exist in all parts, then this is ignored and this value is instead inherited from the parent object.) Unlike other objects, the x and y position coordinates refer not to the top lefthand corner of the glyph but rather to a sensible point within it (like the centre of a notehead). The size of the glyph is set to 36 if omitted. Creating the glyph does not display anything immediately: the setGlyph command must be used to specify the actual glyph to be loaded.

addLayer(name(s), part(i), x(i), y(i), width(i), height(i))
Applies to: canvas, layer, scroller, text, glyph, stave
See the addGlyph command for information about the name and part arguments. Adds a layer to the parent object. The x and y position coordinates refer to the location of the top left corner of the layer within the parent object's coordinate space. Width and height are in pixels. When first loaded, the layer is empty: the loadImage or setColour commands can be used to change this.

addLine(name(s), part(i), x1(i), y1(i), x2(i), y2(i), lineWidth(i))
Applies to: canvas, layer, scroller, text, glyph, stave
See the addGlyph command for information about the name and part arguments. Draws a line on the parent object from the start point (x1, y1) to the end point (x2, y2). The width of the line is given by the lineWidth argument, and the default colour is black.

addNote(pitch(s), position(i), duration(i))
Applies to: stave
Draws a note at the horizontal point on the stave given by the position argument. The pitch is of the form Ax4, where x can be omitted for a not without any accidental or one of the following: ## double sharp, #+ three quarter sharp, # sharp, + quarter sharp, n natural, - quarter flat, b flat, b- three quarter flat, bb double flat. C4 is middle C. The duration can be one of the following: 0 breve, 1 semibreve, 2 minim, 4 crotchet, 8 quaver, 16 semiquaver, 32 demisemiquaver.

addNotehead(pitch(s), position(i), filled(i)=1)
Applies to: stave
See the addNote command for information about the pitch and position arguments. The filled argument determines whether the notehead is filled (1) or not (0). If omitted the default is a filled notehead.

addScroller(name(s), part(i), x(i), y(i), width(i), height(i), scrollerWidth(i), scrollerSpeed(f))
Applies to: canvas, layer, scroller, text, glyph, stave
See the addGlyph command for information about the name and part arguments. Adds a scroller to the object with a viewing window whose location and size is determined by the x, y, width, and height arguments. The scrollerWidth sets the width of the scrolling layer that can be set in motion from right to left through the viewing window. The scrollerSpeed specifies how fast the scrolling layer will move in pixels per second. Setting a negative value for this will reverse the direction of the scroller.

addStave(name(s), part(i), x(i), y(i), width(i), height(i), lineWidth(i))
Applies to: canvas, layer, scroller, text, glyph, stave
See the addGlyph command for information about the name and part arguments. Adds a stave to the object. The x and y position coordinates refer to the left end of the top stave line. The width and height specify the length of the stave lines and the distance from the top to bottom stave line respectively. The lineWidth argument sets the width of the stave lines.

addText(name(s), part(i), x(i), y(i), fontSize(f)=36.0)
Applies to: canvas, layer, scroller, text, glyph, stave
See the addGlyph command for information about the name and part arguments. Adds text to the object. The x and y position coordinates refer to the top left corner of the bounding box of the text. If not specified, the fontSize is 36. No text is drawn immediately: the setText command should be used to achieve this.

clear()
Applies to: canvas, stave
When applied to the canvas, this method removes all of the existing objects and any reference to them. When applied to the stave, this method removes all notes and clefs, leaving a blank stave.

clearImage()
Applies to: layer, scroller

Removes the currently loaded image. (This has no effect on any child objects in the layer.)

fade(opacity(f), duration(f))
Applies to: canvas, layer, scroller, text, glyph, stave, line
Sets the opacity of an object and animates the transition over a period of time specified in seconds by the duration argument.

loadImage(imageFile(s), autoSize(i)=0)
Applies to: layer, scroller
Loads an image into the object. The source of the image must be a file that was placed in the current score's .dsz file at the time of its creation. The autoSize argument is disabled by default. If set to 1 then the object's size will be altered to accommodate the image. In the case of a scroller object, the height of the scroller and the width of the scrolling layer will be adjusted to fit this. (There is no change made to the width of the viewing window.)

move(x(i), y(i), duration(f))
Applies to: canvas, layer, scroller, text, glyph, stave
Sets the position of an object and animates the transition over a period of time specified in seconds by the duration argument.

remove()
Applies to: canvas, layer, scroller, text, glyph, stave, line
Removes the object from its parent object. All references to the object are removed and it cannot be reused.

removeClef(position(i))
Applies to: stave
Removes the clef at the specified horizontal position on the stave. This is only allowed if doing so doesn't leave any ambiguous notes or noteheads on the stave. (The leftmost clef cannot be removed unless there are no notes between it and the next clef.)

removeNote(pitch(s), position(i))
Applies to: stave
Removes the note or notehead of the given pitch located at the specified horizontal position on the stave.

setClef(clef(s), position(i))
Applies to: stave
Sets the clef for the given horizontal position on the stave. If a clef already exits at this point then it is replaced, otherwise a new clef is added. The current available clefs are treble, bass, and alto.

setColour(r(i), b(i), g(i), a(i)=255)
Applies to: canvas, layer, scroller, text, glyph, stave, line
Sets the colour of the object. For the canvas and for layers, this is the background colour. For the scroller, it is the background colour of the scrolling layer. For text, glyphs and lines it is the foreground colour of the object. For staves, it currently only sets the colour of the stave lines, although this behaviour may change in future. The colour is specified with red, green, and blue parameters between 0 and 255, with an optional alpha parameter (set to full opacity if omitted.)

setEndPoint(x(i), y(i))
Applies to: line
Sets the end point of the line to the given coordinates.

setFont(fontName(s))
Applies to: text
Sets the font to be used by the text object. A full list of font names on iOS can be found here: https://github.com/lionhylra/iOS-UIFont-Names.
Additionally "Bravura" can be supplied as the fontName argument to use the Bravura music font.

setFontSize(fontSize(f))
Applies to: text
Sets the font size of the text object.

setGlyph(glyphType(s))
Applies to: glyph
Sets the music symbol displayed by the glyph. The argument should be the canonical name of the desired glyph as defined in the SMuFL specifications. Only a small subset of these are currently implemented. (Mostly basic clefs, accidentals, and notes.) If the wanted glyph is not available an unknown glyph message will be returned.

setGlyphSize(glyphSize(f))
Applies to: glyph
Sets the size of the glyph. The command will make the glyph an appropriate size for a stave of the height given by the glyphSize argument.

setLineWidth(lineWidth(i))
Applies to: stave
Sets the width of the stave lines in pixels.

setOpacity(opacity(f))
Applies to: canvas, layer, scroller, text, glyph, stave, line
Sets the opacity of the object. 0 is entirely transparent, 1 is fully opaque.

setPosition(x(i), y(i))
Applies to: canvas, layer, scroller, text, glyph, stave
Sets the position of the object within the coordinate space of the parent object.

setScrollerPosition(scrollerPosition(i))
Applies to: scroller
Sets the position of the scrolling layer. This value specifies which horizontal point on the scrolling layer should align with the leftmost edge of the viewing window. When set to 0, the left edge of the scrolling layer is aligned with the left edge of the viewing window. Increasing the value offsets it to the left. The maximum value this can take is the width of the scroller layer: at this value, the layer is hidden just off to the left of the viewing window. The minimum value it can take is the

negative of the viewing window width: at this value the scroller layer is hidden just off to the right of the viewing window.

setScrollerSpeed(scrollerSpeed(f))
Applies to: scroller
Sets the rate at which the scrolling layer moves through the viewing window, in pixels per second. A positive value sees the scrolling layer move to the left. A negative value sees it move to the right.

setScrollerWidth(scrollerWidth(i))
Applies to: scroller
Sets the width of the scrolling layer.

setSize(width(i), height(i))
Applies to: layer, scroller, stave
Sets the size of the object, given by the width and height arguments.

setStartPoint(x(i), y(i))
Applies to: line
Sets the starting point of the line to the given coordinates.

setText(text(s))
Applies to: text
Sets the string of text to be displayed by the text object. Unicode characters are permitted.

setWidth(width(i))
Applies to: line
Sets the width of the line in pixels.

start()
Applies to: scroller
Sets the scroller layer in motion.

stop()
Applies to: scroller
Stops the animation of the scroller.