# OpArtGen

## An Op-Art Generator and Animator

**Michael Stuart-Matthews (mas04)**

**18/6/2008**

Project directory: /homes/mas04/IndividualProject/

## Abstract

Op-Art is an algorithmic style of art that should lend itself to being created with ease by computer programs. Current digital art programs do not simplify the process of creating Op-Art by much, and many do not exploit the algorithmic nature behind the artwork at all. This report documents the creation of a program that allows users to easily create and animate Op-Art, using a visual user interface layered over a modular, extensible rendering engine. A scripting interface to the rendering API is provided to allow users to script the creation of images from text files. The rendering engine uses vector image data so that images may be exported to the vector SVG file format for printing onto large format canvases.

## Acknowledgements

I would like to thank my supervisor Peter McBrien, and my second marker Simon Colton, for their useful assistance and advice that they provided during this project. I would like to acknowledge Bridget Riley and the other artists whose work I have used throughout this project for their work which provided the inspiration for this project.

## Table of Contents

## Introduction

Op-Art is an artistic movement that began to gain popularity in the USA in the early 1960s. Artists throughout history have experimented with various different ways to represent the depth and motion of our three dimensional environment in the constrained format of a two dimensional painting. Op-Art attempts to recreate this depth and movement through bold abstract forms that instil in the viewer's mind a sense of movement or vibration in their perception of the painting.

This artistic style originated before computers became a commonplace tool for artists to use and little Op-Art was created using digital means during the time of the movement itself, with each painting taking great forethought and care in execution. Even so, much of the existing work done in this style appears algorithmic or mathematical in nature, and therefore should lend itself to being easily created with the aid of computer programs.

Most computer programs that are designed to produce art tend to prove inadequate when used to create Op-Art. They usually rely on the user directly editing either the pixels, or the vector shapes that make up each layer of the image. As Op-Art comprises simple elements that are repeated and altered algorithmically over an image, the workflow of traditional art programs doesn't suit producing Op-Art.

How do we create a program that makes it easy to produce this algorithmic art? The best solution from existing programs is to create a programming language that allows users to describe the algorithms behind the images. This does however prevent users that unfamiliar with programming from using the software. As the target users for this software are artists, it is important that the program provides a visual mode of working along with a language based approach.

Other tools that do provide a visual interface for creating Op-Art are limited by their fixed rendering pipeline and lack of support for scripting: any design that cannot be produced using the provided rendering pipeline and interface may not be created by that program unless it is modified by its author.  A modular approach is needed that gives the user a flexible rendering pipeline and the ability to program and load new custom modules to cope with the demands of any artistic brief. One area of software that has adopted a similar approach to solve a similar problem is that of modular sound synthesis.

The software requirements for the sound design workflow map to our requirements for an Op-Art design workflow. The basic concept behind sound synthesis is to generate a repeating waveform using oscillator modules that is then shaped and modified by subsequent filter modules. The resulting sound produced depends on the parameters supplied to the various modules and the topology of the network that connects them together. When creating Op-Art we need to create repeating shapes that are then modified algorithmically to create geometric effects.

Computer based sound synthesis tools have grown from an purely language based solution (1) originally developed at the Music and Cognition Group at M.I.T.'s Media Laboratory  to modular visual graph based scriptable synthesisers (2). The intuition behind Reaktor allows users to visually connect various modules together in a graph and edit their properties rather than having to write code to describe how an instrument should create its sound wave. This intuition should prove helpful when applied to the problem domain of generating Op-Art.

Another shortcoming of many digital art programs is their lack of support for exporting images produced directly into a vector format. This is crucial for artists, as it enables them to print their work directly onto large format canvases without any perceived loss of quality.

In this project we present a solution derived from modular sound synthesis programs that enables artists to rapidly create unique, high quality, dynamic, digital artwork in an Op-Art style that can be printed onto large-format canvas. The program itself may easily be extended to support any artistic brief by loading new components at runtime. It allows you to easily and quickly create artwork using a visual interface, and also provides additional flexibility through scripting functionality for advanced users.

## Project Contributions

- We look at the approach taken by existing Op-Art software and also state-of-the-art modular sound synthesis programs to help develop an intuition for our modular solution to generating Op-Art. (See: *Background*, Page 10)
- We analyze a number of works of Op-Art in order to determine a set of useful core modules and the core data types that these modules should process. (See: *Analysis of Existing Op-Art to Determine API Modules,* Page 18)
- We construct an extensible API for an engine that renders Op-Art using a number of components that are dynamically loaded at runtime to create and process the image data algorithmically. (See: *The Rendering Engine*, Page 42)
- API Components may be built-in or external to allow for user generated components. We detail how you may extend the API with your own external components that are loaded at run-time. (See: *Creating external components for OpArtGen*, Page 76)
- Each API module encapsulates a different part of the image generation process; and by specifying different connection topologies between modules and altering the properties of each module the resulting graph acts as a high-level macro language that describes the algorithm the rendering engine should use to create the resulting image. (See: *Creating Art Using OpArtGen*, Page 27)
- We implement a program which provides a graphical user interface and a scripting interface to the API. The Op-Art rendering process may then be driven visually using the GUI, or programmatically using scripts. (See: *Using the Visual Structure Diagram Editor to Create Images*, Page 34; and: *Using Scripts to Create Images*, Page 37)
- We implement a number of the core API modules (See: *Implementation of Core Modules,* Page 62)

- We use the rendering engine and our core modules to generate and evaluate reconstructions of actual works of Op-Art, and other original art. (See: *Existing Works of Op-Art Recreated Using OpArtGen*, Page 83)

## Background

### The Op-Art Movement

The phrase Op-Art itself stems from an article in Time Magazine from 1964 that described a new geometric art form that was emerging at the time as 'Optical Art' (3). The movement first gained popularity amongst the public in 1965 with the exhibition called "The Responsive Eye" at the New York Museum of Modern Art.

Artists working in an Op-Art style use abstract geometric shapes or tessellations in their paintings combined with an illusory use of perspective and highly contrasting colours, often limited only to black and white. Where colour is used it tends to reinforce the illusion of depth in a painting, with warmer colours appearing to advance towards the viewer and cooler ones recede. Perspective is used to create dynamic energy and movement, by inducing visual tension as the viewer tries to reconcile the multiple contradictory geometric interpretations of the painting that exist.

Victor Vasarely, an artist considered to be one of the original proponents of the movement due to his 1938 work "Zebra"(4), has painted many works that make very effective use of such perspective. This is one style of Op-Art that I do not intend to cover: its construction is not so algorithmic in nature and reproducing such works would require the project to have too great a scope.

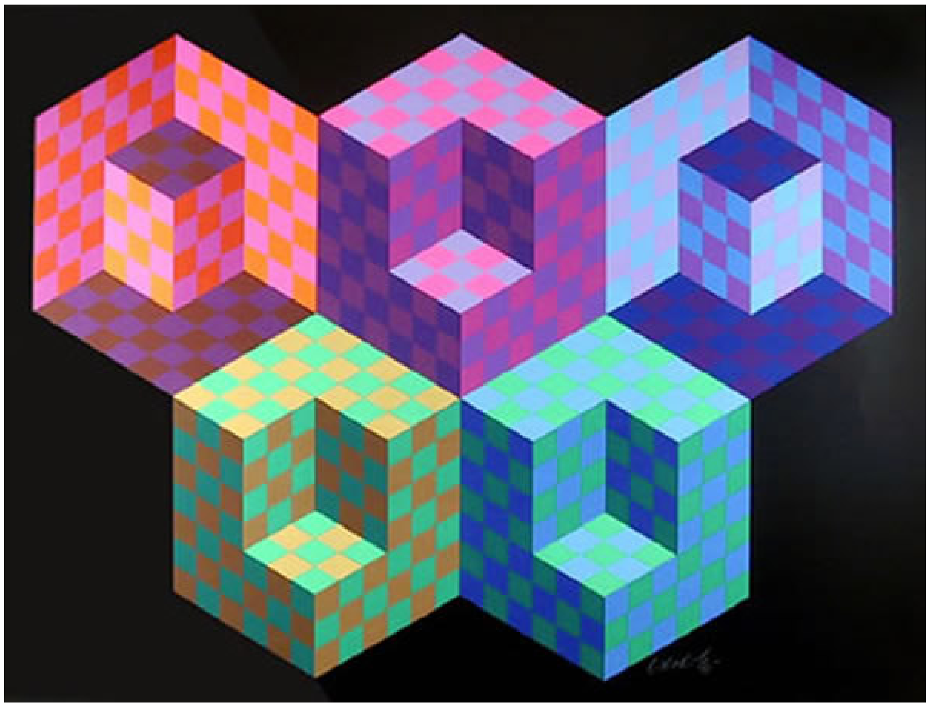


**Figure 1: Victor Vasarely, "Hexa 5", 1998.**

Another important artist from the Op-Art movement is Bridget Riley. Arguably one of England's most influential artists; Bridget Riley began creating her instantly recognisable black and white works in 1961, after repeated failed attempts to create an entirely black painting. (5) The resulting tension created by the minimal, contrasting parts of the work interested her enough that she continued creating works in a similar style for several more years. Her use of a precise composition of contrasting, modulating geometric shapes leads to a number of visual phenomena that occur when viewing her work.

The viewer of a Bridget Riley painting usually perceives a flicker or movement in their visual image created by the static picture. Research suggests that this may be due to a combination of perceptual judgements by the viewer and involuntary eye movements generating an 'incoherent distribution of motion signals' to the brain. (6) In a similar way that much of Vasarely's work uses perspective to create contradictory interpretations, perceptual tension in Riley's work is often due to contradictory interpretations of the figure-ground relationship. This is the relationship that governs our ability to distinguish elements based on contrast – allowing us to separate out foreground from background. (7)

Bridget Riley's early works (1961-1967) are mostly achromatic, and their style lends itself well to an algorithmic reproduction. The main principle behind Riley's early works was summed up by her as follows: "The basis of my paintings is this: that in each of them a particular situation is stated. Certain elements within that situation remain constant, others precipitate the destruction of themselves by themselves. Recurrently, as a result of the cyclic movement of repose, disturbance and repose, the original situation is restated."(8) Works such as her "Movement in Squares" (1961) exemplify this, and form the basis for the style of Op-Art that I would like the finished program to be able to generate.

After 1967 Bridget Riley began to explore the use of colour in her works through two styles. The first used contrasting colour series arranged in a sequence of stripes repeated throughout a painting, resulting in works such as "Cantus Firmus" (1972-3). Again, this is a style that lends itself to algorithmic generation and is one I would like my program to produce.  Other Op-Art artists that have made great use of colour in their works include Julian Stanczak and Richard Anuszkiewicz, (9) who tended to explore the visual relationships that exist between colours in their art. Combining her contrasting alternating colours with the mathematically based composition that featured in her earlier works resulted in Riley's second style, exemplified by "Cataract 3" (1967).

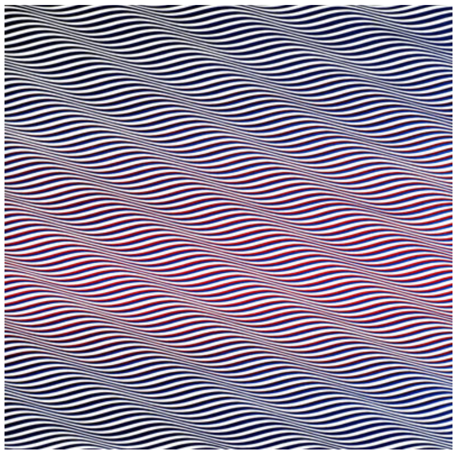


**Figure 2: Bridget Riley, "Cataract 3", 1967.**

In this work the colour sequence alters gradually throughout the image, and a set of repeated lines appears to be folded in a regular pattern. Producing such images in a vector format requires complex algorithms and is beyond the scope of this project. Eventually, Riley's experimentation with curved grids and colour resulted in her later 'Curvilinear' style. These works feature a far more complex layering of colour than her earlier curved grids. One can see the transition from her earlier colour sequence works to the 'Curvilinear' style of colour layering through works such as "New Day" (1988).

## User Requirements

In order to get a feel for what requirements an artist would have as a user of an Op-Art program, I discussed the concept with a number of artists and designers that I know. A number of important points were raised during these discussions. The first issue to come up was the need for the program to provide the user with sufficient creative scope. If the finished program is to be more than an Op-Art toy, there needs to be enough scope in the design to allow for the creation of unique and varying images. Simply having a number of parameters that can be altered to generate multiple versions of what is essentially the same image would not be sufficient.

Secondly, an element of visual algorithmic control was suggested. Since the generation of the images will to some extent be reliant on a set of equations, providing the artist with the ability to alter the equations themselves would allow for creative scope. Unfortunately this would require the artist to be comfortable with the mathematics behind the geometry. Instead, if some GUI based control were designed that allowed for visual editing, then the artist would still have the creative scope that they desired yet would have the ability to easily control the algorithms. Furthermore, if the artist could gradually alter parameters and see the effect it had on the generated output it would allow for greater creative output.

The last point that was raised was that of the program's output file format. Art of this kind is still often sketched out on graph paper and then realised directly onto canvas with paintbrush and paint. However, when it is created using digital means, if it is to be printed onto canvas rather than being painted then the artist's file is exported in a vector format. This means that once it has been sent to the printers, the file may be resized to the correct resolution for the size of the canvas without any aliasing occurring. If the finished program is to be used by artists then its output would need to be vector based so that the works produced can be printed out in varying resolutions at a high quality.

## Current State of the Art and Similar Existing Products

### *OPAGA (Jorden Lacey)*

I downloaded and compiled the source code for Jorden Lacey's program OPAGA, written as a previous solution to the Op-Art Generator and Animator problem. The approach taken by this software is to provide the user with a selection of images that have been derived from analysis of corresponding Bridget Riley works. The user can then alter the parameters used by the algorithms in reproducing the works using the controls situated to the side of the image. This provides a very quick and easy method of creating works of art in a similar style to some of Bridget Riley's art, however I feel that this method does somewhat limit the user's creativity.

The requirement of sufficient "creative scope" that was outlined earlier wouldn't be satisfied if I used a similar method as the basis for this project. To create an image that looks stylistically different to the ones that are provided would require the user to develop a new Java class file. Having said that I do feel that the output from this program is aesthetically pleasing as a result, and therefore some degree of empirical analysis of existing Op-Art works should be performed to derive default parameter values for the program.

The parameter control names are not very helpful, for example "Change one", "change two", etc. Trying to alter the image produced in a specific way is difficult as a result. It is important that the parameter names used in the program correspond to the aspects of the image that they change so that the program has a professional feel to it and is easy to use.

### *Op-Art Generator and Animator (Oliver Payne)*

Oliver Payne's Op-Art generator program effectively produces Op-Art in a similar style to much of Bridget Riley's early work. It is based around the concept of distorting a grid of repeated shapes and bending the surface on which the grid lies. It provides tools to progressively alter the size, colour and opacity of the shapes and reproduces works such as "Movement In Squares" well, however there is no scope to produce works like Riley's "Shift" (1963), which are created by progressively altering the geometry of the repeated shape over the course of the painting.

The rendering pipeline in Oliver Payne's program is fixed, so if it is unable to create artwork in a specific style then it is not possible to add any extra components to it that would enable that style to be rendered.

### *Photoshop and Illustrator (Adobe Systems)*

Adobe Photoshop and Illustrator are the industry standard digital graphics tools used by artists. They are not specific to Op-Art, and they are not generative art tools, however they contain a number of features that could be useful in an Op-Art generator program. The floating properties palette window gives immediate feedback on whatever elements the user has currently selected, and the layers feature means that complex compositions can be created out of multiple simple layered elements.

### *Opartica (Dan Zen)*

http://www.opartica.com/opartica/

Opartica is an online tool that has been designed specifically to create animations in an Op-Art style. It allows you to select from a small set of pre-drawn patterns that can be set to rotate clockwise or anticlockwise, or move up or down the screen. The speed of the movement can be altered, and the moving patterns can be layered over each other. All the patterns contain repeating areas of one identical colour and transparency, so the combined layered effect can create interesting visuals.

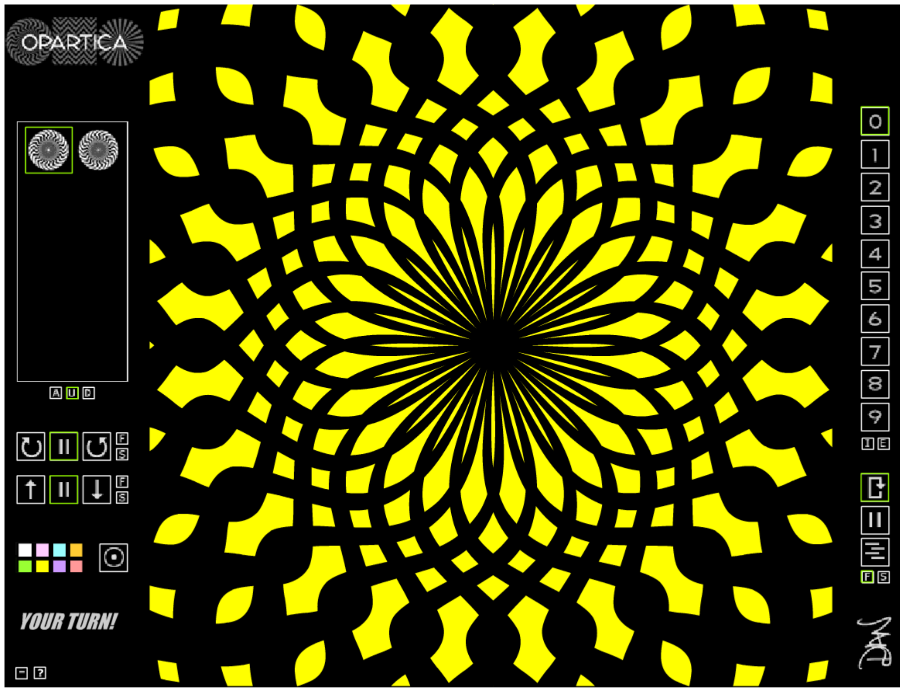


**Figure 3: Opartica Tool.**

Opartica makes much use of patterns that are based on a grid using polar coordinates. This allows scope for creating works that radiate out from the centre of the image, and such a feature could allow an Op-Art program to recreate works in a similar style to that of Riley's "Blaze" series.

### *Artlandia (Mathematica Plugin)*

http://www.artlandia.com/products/artlandia

Artlandia is a piece of commercial software that provides a Mathematica user with an extra set of functions related to graphics output and array processing. They enable simple generation of various curves, creation of distributions, wallpaper tiling, layer operations, and a variety of graphics transformations. These can then be coded up and run to produce visual output, with an emphasis on creating repeating patterns or visualisation of data and algorithms.

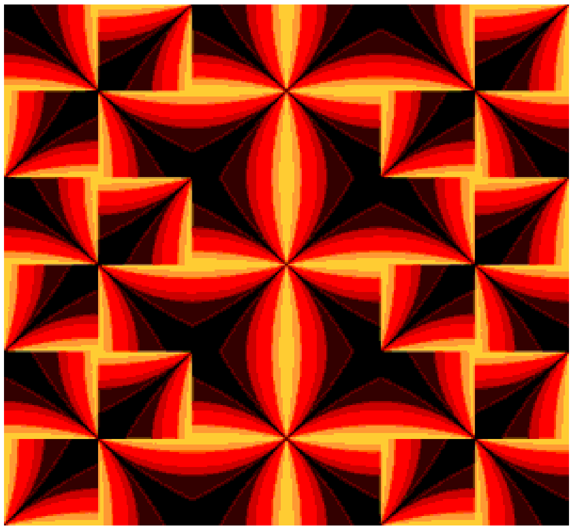


**Figure 4: Sample output generated by Artlandia.**

As the price of Artlandia is over £200, not including a Mathematica license, I was unable to try out the software to get a feel for any features that could be useful in an Op-Art generator. From the descriptions provided on the site however it seems that the creative scope provided is very broad due to the scripting language used to create an image. Eventually one can combine various complex rendered elements in different layers with ease; but the learning curve for the tool is steep as there is no visual editor. It would be good to recreate the flexibility of combining different layers and transformations that exists in Artlandia in a more visual user interface format.

The language based nature of the program does afford advanced users a great degree of flexibility in defining how their images are generated. Allowing OpArtGen to be scripted using a similar language would mean that the program would appeal both to new and advanced users. It would still enable artists that don't understand the intuition behind programming to use the visual editor to build the image's structure.

### *Advanced Visualization Studio (Winamp Plugin)*

The Winamp AVS plugin allows the user to create images that react to changes in the spectral content of a multimedia file that is being played at the same time in the Winamp media player. A number of rendering modules are provided that can freely be combined with transformation modules in effect lists. A user creates the animations by defining equations that modify variables on a per-frame, or per-beat basis. The objects rendered to screen are drawn according to equations containing those variables. A similar system could be used in the Op-Art program that would allow an artist to modify image parameters on a per frame basis.

The most impressive feature of the Advanced Visualisation Studio is the wealth of user generated content available at the community website. (10) This enables users to download a variety of presets to aid learning and satisfy casual users.

### *Flash (Adobe Systems)*

Adobe Flash is a vector based animation tool designed for authoring content for the web. The main animation techniques employed are “Keyframes” and “Motion Tweening”. This allows a user to create a number of vector drawings at certain points in time, and then have the program automatically “tween”, or create intermediate frames of vector content that smoothly blend between the Keyframes. This technique could be used to progressively alter shapes that are repeated throughout an image, without the artist having to write equations to do so as was found in the Winamp AVS plugin.

## Graphics Libraries

As the completed image files must be exported by the program in a vector format it is necessary to maintain the in memory image as a vector format. This means any transformations must be done to the constituent elements in an image in such a way that they remain stored in memory as vectors. This prevents the use of most 3D graphics libraries for Java such as JOGL or Java3D, as there is no capability to export the rendered 3D scenes out to file in a vector format. This is a shame as the 3D libraries would otherwise make creating the complex surface distortions that are found in Op-Art easier.

The Java 2D library contains helper classes for efficiently processing 2D graphics calculations for operations such as affine transformations. This would mean a engine could easily be written in Java to skew and rotate the image, and perform simple orthographic projection of a basic surface warped in 3D space onto a ‘canvas’ plane. The Java 2D library has a large community associated with it and is very well documented. Furthermore, a Java library called Apache Batik exists that allows programs that use the Java 2D library to easily render in memory images out to file in the SVG vector image format.

## Generative Art and Sound Synthesis

As we have seen, there are not many commercially available Op-Art software packages that exist. This is due to the limited market for Op-Art generating software. To build an Op-Art program that approaches the situation in a novel way it is necessary to find new ideas and methods of working, so it might therefore be useful to consider software packages that have a similar creative functionality, but are used to produce a different type of art.

The connection between visual and sonic generative art is strong - they are linked by the common need for an algorithmic construction. Much of Bridget Riley’s work has an inherent rhythm and flow, and observing one of her works can be likened to listening to the interaction between various sounds in a piece of music. Riley herself has noted this, here describing her later curvilinear painting style: “When played through a series of arabesques the curve is wonderfully fluid, supple and strong. It can twist and bend, flow and sway, sometimes with the diagonal, sometimes against, so that the tempo is either accelerated or held back, delayed” (11)

Russell Richards argues that an interactive computer system designed for creating art may exist in one of three modes of interactivity: consumer interactivity, processor interactivity or generator interactivity. The latter mode is one in which “the user is positioned as the creator of content within a system”. (12) This corresponds to the mode in which we can achieve our goal of “sufficient creative scope”, and the equivalent system that is easily available to analyse in sonic generative art world is that of sound synthesizers and sequencers.  Op-Art tends to have a central repeating element that is altered over the course of the canvas, and then often the entire canvas is shaped or distorted. A sound produced by computers tends to have gone through a similar construction process. If we consider an Op-Art animation to be equivalent to a musical passage of synthesized notes, then looking at how a software synthesizer enables a musician to produce, shape, and alter the timbre of the passage over time will give us ideas for how a novel type of Op-Art generator program could enable an artist to create an Op-Art animation.

There are a variety of common sound synthesis techniques that exist in software. One technique, known as modular synthesis, gives the musician an incredibly flexible and powerful tool to craft their sound. Most software synthesizers have a common fixed routing path for the sound signal from initial creation to output. The signal is usually created from a fixed number of oscillators, each possibly having a customizable waveform, the output of which is combined and passed through a number of filters and sound shapers. Modular synthesizers do away with the fixed number of oscillators and fixed signal path through a fixed set of sound processing units. Instead, the musician is presented with a framework in which they can build up their own set of oscillators, filters, and shapers, and a tool with which to route the audio signal and parameter control values between the various modules as they see fit. (13)

One of the most mature modular synthesis software packages available is Native Instruments' "Reaktor". (2)

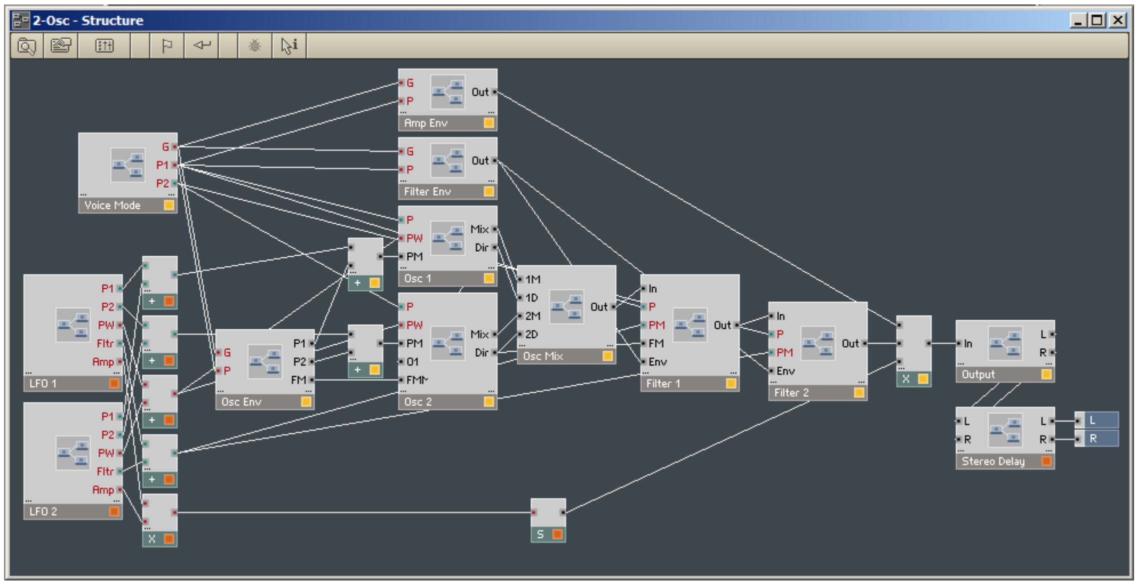


**Figure 5: Reaktor Module Structure Window showing interconnectivity.**

Reaktor's interface allows a user to easily see and alter the structure of an instrument via the connectivity of its various modules. Each module has its own internal structure window allowing the user to further customize how parameters are altered – either via a GUI window or via parameter output from another module. Altering the sound produced can be done by modifying a module's parameters via its GUI, or by reconfiguring the instruments' module structure.

A similar modular interface could be used for the Op-Art generator program, allowing the artist full control over how shapes and colours are generated, positioned and combined in each layer, and then how the layers themselves are transformed or interact with each other. In order to decide what individual modules should be, I will later look at a number of works of Op-Art to deduce the common elements.

The sound waves from real instruments are very complex, and musicians tend to vary the timbre of the sound as they play a passage of notes. To ensure that synthesised notes remain interesting and don't simply sound like a "tone organ", after the basic sound wave has been produced the sound is usually modulated. This involves changing a module parameter over time, for example a filter frequency might be modulated via a sine wave to produce a vibrato style effect. In Reaktor this can be achieved by connecting a LFO (Low Frequency Oscillator) module's output to the filter frequency parameter input on the target oscillator module.

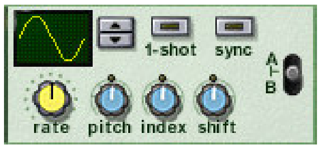


**Figure 6: LFO control from the Malstrom Synthesizer**

A LFO module on a synthesizer produces various customizable waveforms as its output. Such a module could be used in the Op-Art program to create simple animation, for example the sine wave output of a LFO module could be connected to the width and height inputs of the shape module in a reproduction of "Movement In Squares". This would result in an animation that appeared to pulse. Obviously, with a variety of LFO waveforms and more complex signal routings more interesting animations could be achieved.

In Granular synthesis oscillators as the sound source are removed and replaced by small chunks of sampled audio called grains. These grains are assembled in an array, and then when a note is played the audio grains are then played back in order. To add colour to the sound, a musician has the ability to control parameters such playback start point. (14) A similar technique could be used for varying series used by processing modules. This would allow an LFO module to be connected up to the colour index parameter and allow the colour used in an image element series to change over time.

## Analysis of Existing Op-Art to Determine API Modules

I will now look at a number of works to try and ascertain what common elements exist and develop methods for their reconstruction. This will lead us to a set of features that will be implemented as modules in the finished program so that it may reproduce similar works. Let us first look at Bridget Riley's "Movement In Squares".

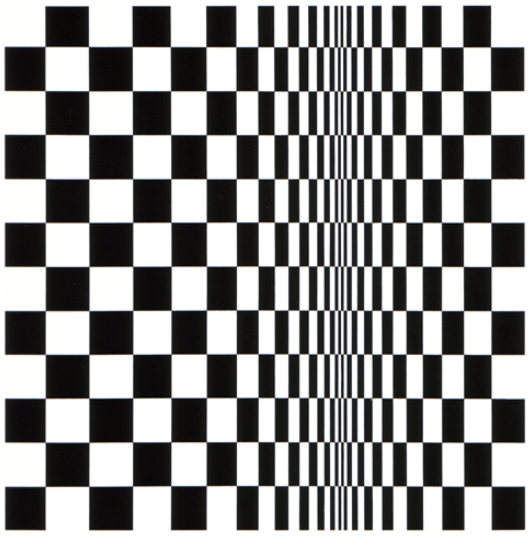
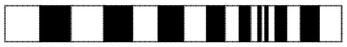
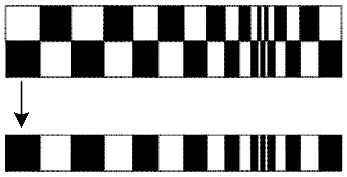


**Figure 7: Bridget Riley, "Movement In Squares", 1961.**

It seems that repeated shapes whose properties are varied across the canvas are a fundamental concept in most Op-Art. One required module (or group of modules) in the finished program should therefore be capable of generating a series of vector shapes, allowing the shape parameters such as position, size, colour, and opacity to vary over the image. There are a number of ways we could programmatically construct the above work. One way would be to draw a series of rectangles across the page, with fill colour alternating between white and black, and the starting colour of the series set to white. Each successive rectangle in the series is drawn with its leading edge touching the trailing edge of its predecessor. The width of a rectangle in the series is defined formulaically, with respect to its position in the sequence. The formula used would contain a number of parameters that could be altered, for example: the turnaround point, and the rate of change of width:

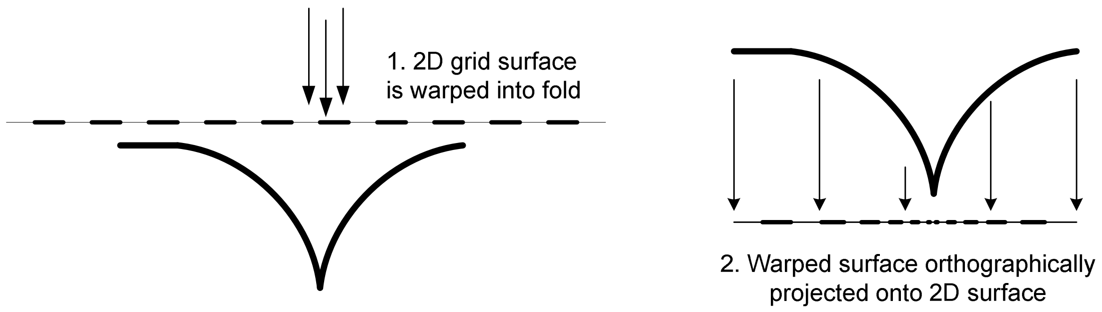


Once one series of rectangles has been drawn, another similar series should be constructed directly below the first, with the starting colour of the new series set to the opposite of the one used in the previous series. If a program continued outputting series of rectangles in this way until canvas of the desired size had been filled, then the image produced would be similar to that of "Movement In Squares".



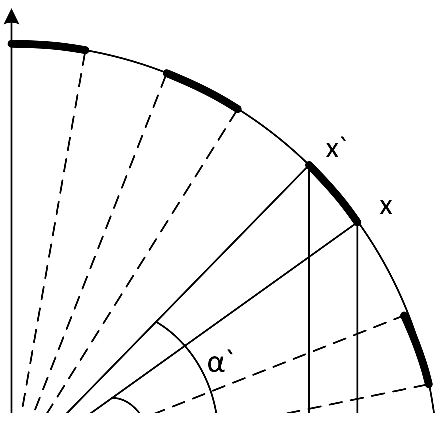
The problem with this approach is that it would be time consuming to set up the parameters so that the entire image fits into the canvas. This can be solved by constructing the image using an alternative approach. Instead of a series of shapes with changing width we use a grid of squares alternately filled with black and white lying on a flat surface. The white squares could even be empty, allowing the white canvas background to show through. Now, instead of specifying row height and column width for the grid we specify the number of rows and columns we want in the grid and have the program calculate the correct grid dimensions. The grid dimensions could then be stored proportionally, allowing us to combine multiple grids to form a composite image, ensuring that each grid filled up the space available to it.

The grid is then distorted by stretching the flat surface over two joined quarter circles, with the resulting surface looking like a piece of paper that has been folded and opened back out. The distorted 3D surface is then orthographically projected back onto a flat surface to give the finished image.


1. 2D grid surface is warped into fold
2. Warped surface orthographically projected onto 2D surface

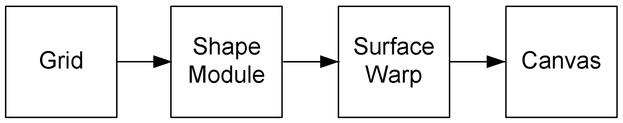
To achieve this in a general form we would need a module that could accept a set of vector shapes as its input, generate a set of points in 3D space corresponding to the shapes on the warped surface, and then calculate the x and y coordinates of the projected points as the dot product of the point in 3D space with each of the two orthonormal basis vectors of the final 2D surface. The 3D surface could be defined parametrically, and then the module could allow the surface to be changed dynamically. Unfortunately, arbitrarily warping a 3D surface requires vector based texture mapping, a technique that is beyond the scope of this project. (15)

Instead, it is still possible to warp a grid over simple curves and then calculate the projected points using some trigonometry, as long as it is possible to calculate the distance between two points on the curve. This means we could have a grid warping module that takes a grid as input and outputs a grid whose column sizes have been altered to represent a projection of the grid warped over a curved surfaced, however calculating the distance between two points on a curve is not always simple. One compromise that will still allow us to distort the surface without requiring the implementation of complex graphics algorithms is to base the distorted surfaces on circular arcs. (16)



From the parametric equation of a circle we can calculate the projected distance of a grid square, p:

Now we have a distorted grid we need a way of alternately filling the grid cells with black squares prior to the distortion being applied. To do this we need a module in which we could define a shape that could be tiled over the grid. The module structure diagram required to reproduce “Movement In Squares” might look something like this:



Looking at Riley’s “Shift” below, we see that the grid squares are not entirely filled by colour as we have had in previous works. Instead, there is a series of triangles, whose geometry changes over the course of the image. Looking at “Shift” is like viewing a stack of film strips, either laid on top of each other or side-by-side. We could then consider the grid to be similar to a series of still frames taken of an animation of the vector shape morphing. We have seen that the concept of Keyframes and Tweening are used in Adobe Flash to simplify the animation of vector shapes, so we could use a similar technique to create a series of vector shapes as an ‘animation’ along an image axis. This concept could then be extended to allow the shape series to be animated along time in conjunction with the LFO modules.

So, if we consider the grid to be an array of empty vector shape containers allowing us to insert any desired vector shape, then we could reproduce works like this. We can use the concept of keyframes and tweening in a control in the shape series module’s properties window, allowing us to define the way in which a basic vector shape changes over the entire grid. The module would then insert the interpolated shapes in the grid that is passed to it. “Movement In Squares” could still be reproduced by defining a shape series consisting of a single square, alternatively we could define a simple vector shape module that will insert the same shape into each grid cell.

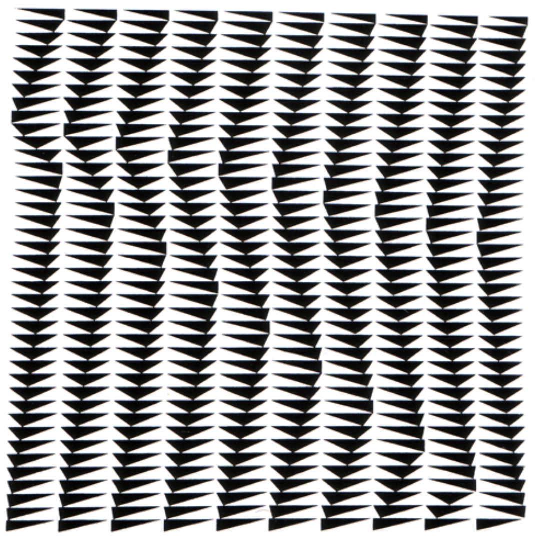
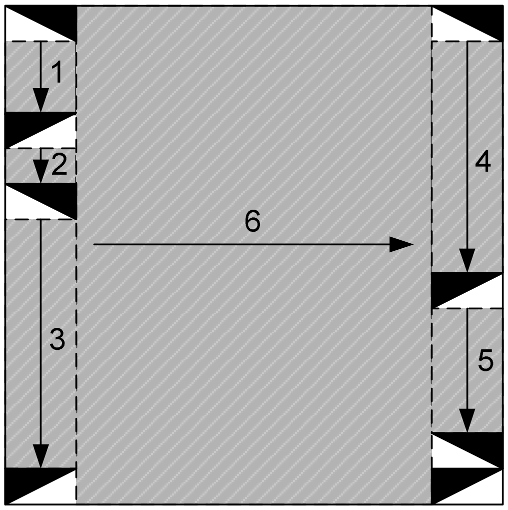
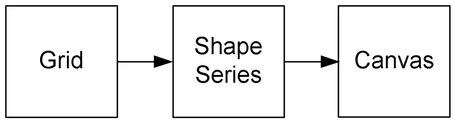


**Figure 8: Bridget Riley, "Shift", 1963.**

To reproduce a shape series like the one found in “Shift” a user would need to have the keyframes and tweening set up in the module’s control in a similar way to this:



The grey areas represent the shapes that are calculated by linearly interpolating the vector points from the surrounding key cells. The order in which interpolation is performed is indicated by the numbered arrows. The shape series module would need a control allowing the user to create a basic shape as the first keyframe, and then allow the user to define the location of the subsequent keyframes in the first and last columns, or first and last rows of the grid. The shapes would consist of a number of vector points defined using coordinates relative to the points' location in a cell. For example, the triangle in the top left corner of the diagram above could be defined as [(0%, 100%), (100%, 100%), (100%, 0%)]. The module structure diagram required to reproduce "Shift" might look something like this:



We now look at another of Bridget Riley's works, "Cantus Firmus". This work consists of a series of coloured vertical bars, however this time the size and spacing of the bars is defined by a regular, repeating pattern rather than a gradual curve.  To produce this we would need to be able to define an array of column proportions, which could then be translated into a repeating set of column widths that are applied to the grid. This distortion is similar to the surface warp module that we needed to produce "Movement In Squares"; given a column cell index the module returns a column width, and given a row index a height.

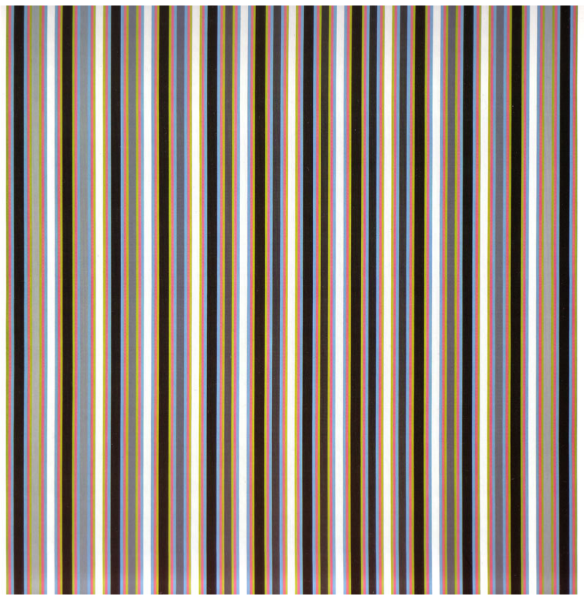


**Figure 9: Bridget Riley, "Cantus Firmus", 1972-3.**

From “Movement In Squares” we know we need a shape module to place rectangles in our grid with an option to choose the colour of the shape. We can refine this notion somewhat with “Cantus Firmus”. Here we need a way of applying a series of colours to those rectangles. We need a colour series module that allows us to cycle through colours in a list, applying each colour to the contents of each grid cell.

The core series in the work consists of three groups of 3 thin bars separated by two thick bars. The thick bars themselves change colours in a series: the rightmost bar changes from light grey, to dark grey, and back to light grey and the leftmost bar stays a constant dark grey. The two thick bars change their positions over 4 repetitions from the right edge of the painting. The thin bars also have their own series: GRB|BRG|GRB, BRG|GRB|BRG, GRB|BRG|GRB etc. We therefore need a module that will allow us to merge together individual grids containing shapes that have had a variety of colour series applied to them.

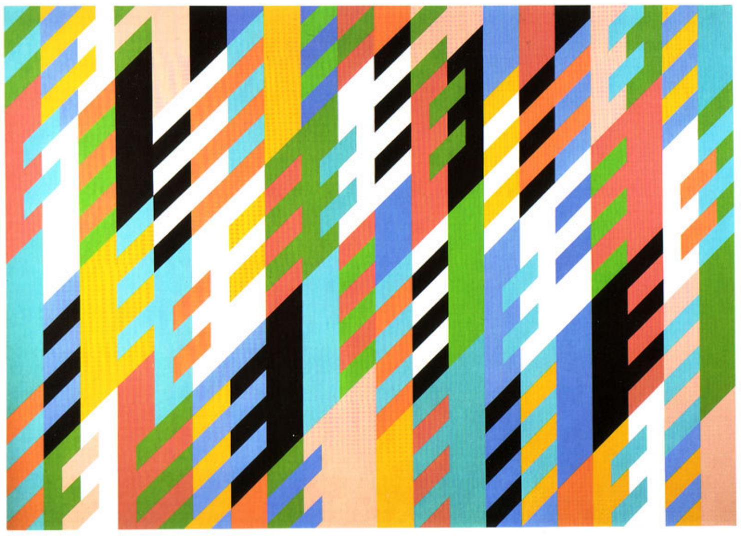


**Figure 10: Bridget Riley, "New Day", 1988.**

In “New Day” we can see there is still the concept of a square grid, though rather than the grid being distorted via a surface fold, it has been skewed to create parallelograms. This could be done in the program by a module that applied an affine transformation to the grid. Looking at the colouring of the image, instead of a regular sequence of colours as we have seen before, or shading over the image, there are multiple layers of shapes with each layer assigned a different colour. Not all the parallelograms in the grid in each layer are filled in however, so the multiple colours show through into the final image.

This introduces the concept of layers – to recreate works such as “New Day” it would be useful to be able to combine multiple layers of generated output into the final image. The merge module should therefore have a variety of merge types allowing you to join the input grids together in a row, or join them by stacking them on top of each other in layers.

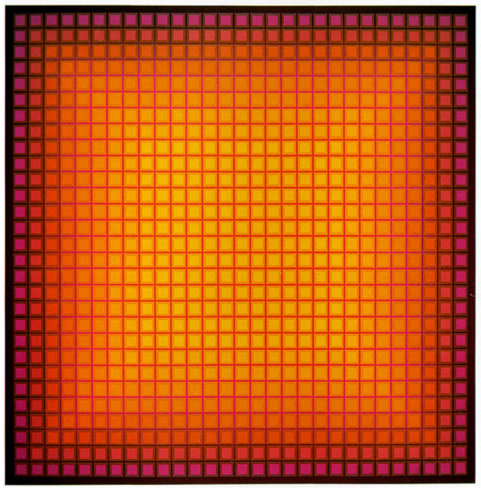
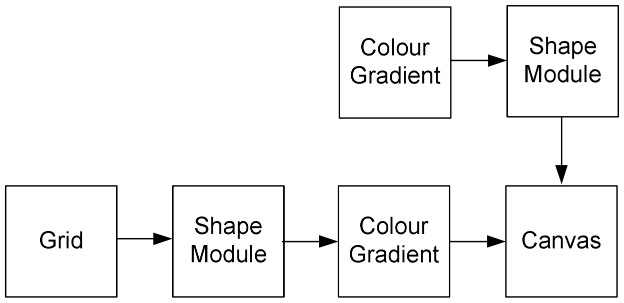


**Figure 11: Julian Stanczak, "CHARTRES", 1980.**

Looking at "CHARTRES" we see that the colour series module is insufficient for reproducing the shading of the rectangles in the picture. Also, the canvas background in this image is not a single colour but a gradient. To achieve this we need another colour module that allows us to create colour gradients. If the module is connected in line with the output of a grid module or a shape series module then a series of colours should be applied to the shape in each grid cell to create the impression of a gradient.

If instead the gradient module is connected to a shape module, then the gradient should be applied as shading to that object. This functionality would allow us to create the black to red gradient in the background of "CHARTRES" and at the same time create the red to orange colour change that exists in the small squares of "CHARTRES". The module structure diagram to reproduce this work might look something like this:



To complete the set of colour modules we look at Julian Stanczak's "Low Asteroid".

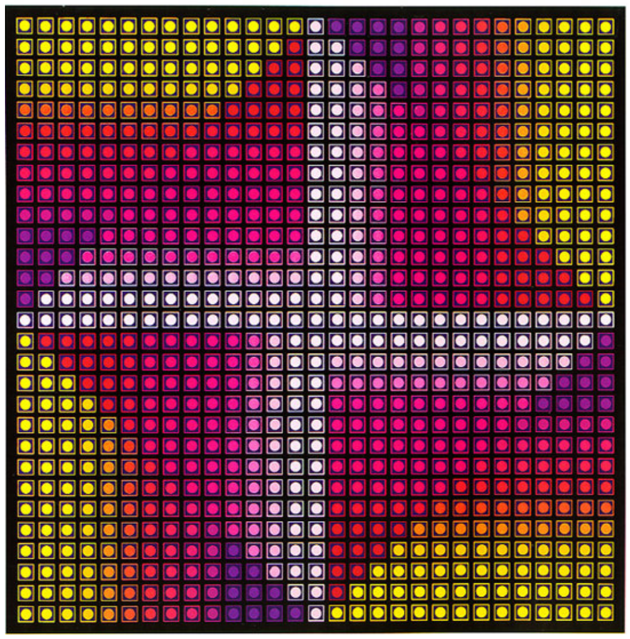
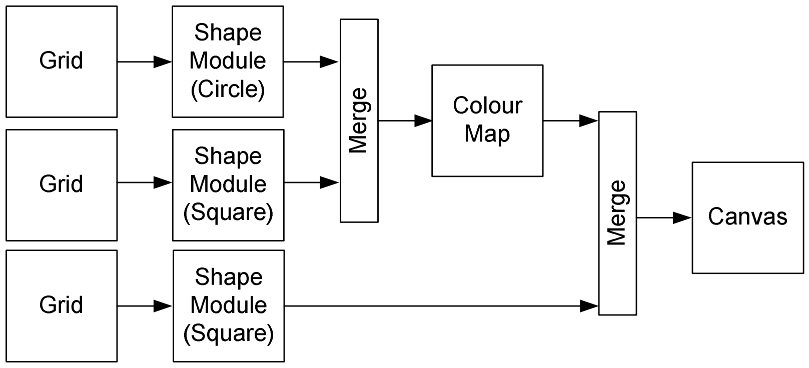


**Figure 12: Julian Stanczak, "Low Asteroid", 1983.**

Even with a colour series module and a colour gradient module it would still be very difficult to reproduce colouring similar to that found in "Low Asteroid". To ensure the user has full artistic control over the finished piece a module is needed that allows the user to apply a colour map over the image. The user could then create the desired colouring in an external graphics program such as Adobe Photoshop, and then load the bitmap into the colour map module. This would then take the relative location of each grid cell in the final image, extract the colour from the same location in the colour map bitmap, and apply the colour found to the contents of the grid cell.

This means "Low Asteroid" could be reproduced by combining 3 grids together onto a black canvas: one grid containing a large, colour mapped square; another containing a smaller, black square; and the last containing a smaller, colour mapped circle. The module structure diagram to reproduce "Low Asteroid" might look something like this:



## Software Design

The key priorities whilst developing this program were twofold:

- Develop a solution that allows rendering of artwork in an OpArt style both using a visual editor and GUI, or from a scripting interface and some domain based language.

- Employ a modular design so that user created components may be plugged-in at runtime without requiring the entire program to be rebuilt.

The design of the software was developed incrementally during the implementation phase of my project, as I did not have much previous experience in developing user interfaces or graphics based applications. I used semi-agile methods with incremental builds and sprint periods of roughly 1-2 weeks depending on what my current objectives were.

Java SE 6 was used to build the software as it has a mature 2D graphics library which is well documented and has a good community built up around it, making it easier to find solutions to common problems. This meant that even though my experience with graphics programming was limited I could quickly begin coding working programs that attempted to create simple graphics in an Op-Art style.

So that the image creation process could be scripted it was necessary to develop some kind of domain language to describe the process. The options available for this were to create either an internal or external Domain Specific Language (DSL), or an Application Programming Interface (API). Given the project scope and timeframe I decided against creating an external DSL due to the associated requirement of developing a compiler or interpreter for the language. Instead I have chosen to develop a hybrid of an internal DSL with an API.

The API provides an interface to a rendering engine and domain objects, and the internal DSL is achieved using interpreted or compiled scripts written in JavaScript or Java respectively that make calls to the API. The GUI has a layered design uses API calls to manage domain objects and render the images.

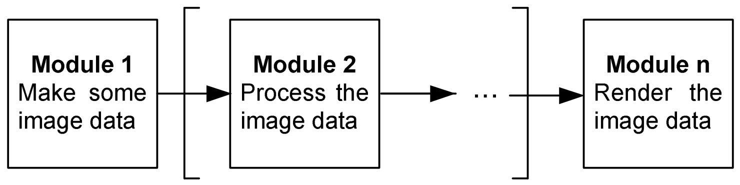
## Creating Art Using OpArtGen

Art is created with OpArtGen using a different workflow to most conventional 2D digital artwork programs. In most digital art programs, one or more vector or raster layers are stacked on top of each other to create the final canvas for output. The user then edits the data in the layers directly using manual tools. Some programs also allow you to modify the data algorithmically using effects and macros. The rendering pipeline for the image is fixed by the developer, and then can only be altered by changing the order of the layers that make up the image.

OpArtGen removes the need to directly edit the artwork data and instead allows the user to build an abstraction of the desired image, and then directly or algorithmically modify the parameters of that abstraction. In OpArtGen the rendering pipeline is not fixed; it is the custom rendering pipeline created by the user that defines the abstraction of the final image. Instead of editing the rendered image data in the GUI you edit the rendering pipeline that creates the final image.

This means that for the same abstraction there could be many variations of the final image produced. Each variation is linked to one combination of module property settings, with as many variations as there are combinations of property settings for the modules in the pipeline. Animation of the image is achieved by moving between these variations by progressively altering the property values and rendering the image each time a value changes.

Creating an image in OpArtGen involves creating a rendering pipeline that does the following:



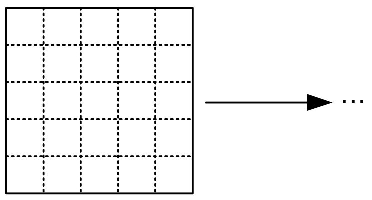
Let's consider an example: imagine trying to create an image consisting of a grid of chequered squares. If you were using an ordinary art program then your workflow would be similar to the following:

1. Create a new image canvas, size 400 pixels x 400 pixels.
2. Choose the desired foreground colour using the colour picker tool.
3. Select the square shape tool.
4. Use the mouse to draw a chequered grid of squares with chosen foreground colour in the canvas. Hopefully the program you are using has a feature such as snap to grid otherwise the grid may not look very accurate!

Now if you decide that you want to change the size or the colour of the squares, the last three steps need to be repeated in their entirety! If the program uses vector rather than raster data then you might be able to select the squares and update their size and colour without having to individually redraw each square in the entire grid. Now suppose that you want the squares to gradually expand and shrink between two sizes, or fade in-between two colours: how could you do this? With most conventional 2D art programs this is not normally possible.

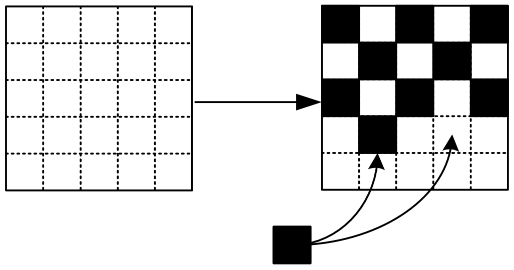
We now look at an outline of the equivalent workflow you would use if you were instead using OpArtGen to create the image. Detailed instructions on how to create the image using the visual editor or using a script may be found in the sections: *Using the Visual Structure Diagram Editor to Create Images* (Page 34) and *Using Scripts to Create Images* (Page 37). The workflow is as follows:

1. Create a new Grid Module and choose the grid size in its properties window (e.g. 5 x 5). This creates an empty grid into which other image elements may be placed.



2. Create a new Shape Repeater Module and connect the output of the Grid Module to the input of the Shape Repeater Module. This means that when the image is rendered, the Grid Module can send the grid to the Shape Repeater Module which then places shapes in each grid cell.

3. Open the properties window for the Shape Repeater Module and select Rectangle as the desired shape, choose its desired colour, and choose the desired size of the rectangle relative to the grid cell it sits in. Select to place the shape in alternate cells in the placement options tab.



4. Create a new Canvas Module and set its size to 400 pixels x 200 pixels using its properties window.

5. Connect the output of the Shape Repeater Module into the input of the Grid Module. The Shape Repeater Module now sends the grid containing the rectangles to the Canvas Module when the image is rendered. The Canvas Module then takes the grid of shapes and stretches it over the canvas size that you have specified with all shapes resized correctly and drawn accurately. If we now choose to display the canvas window for the Canvas Module then OpArtGen will render out our image to screen for us.

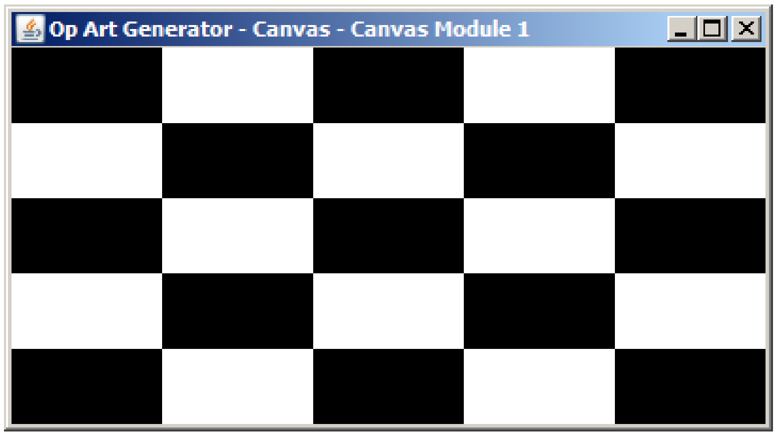


**Figure 13: Final Rendered Image**

Observe that we performed no direct editing of any of the image's data in this workflow. All we did was define a rendering pipeline for our canvas that instructed the program to place a rectangle into alternate cells of a 5x5 grid, and draw that grid stretched to fit our 400x200 canvas.

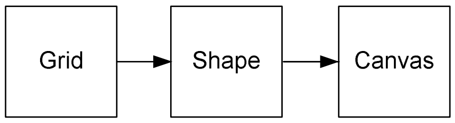


**Figure 14: Rendering Pipeline created by Workflow**

Our rendering pipeline can be thought of as a high level macro detailing how we wish the program to construct our image for us. Because we chose a grid with equal dimensions but a canvas with a width double its height our squares are rendered stretched as rectangles. The final size of the grid is determined by the Canvas Module, and all grid cells are resized so the grid fits the canvas.

We can now alter the parameters of the rendering pipeline we just defined to give one of many variations of our square image. For example, we may wish to have a red square that fills the image or a blue square that only occupies half of our image. We may even wish to have an image that oscillates between a larger and a smaller square. Altering module parameters to achieve this may be done in two ways: we can either manually edit them using the same properties windows that we just used in our workflow, or we can define how we wish the parameter values to vary over time using oscillator modules. This would allow us to define a type of high level macro that lets the shape's size parameter in our rendering pipeline vary algorithmically from say 50% to 100%.

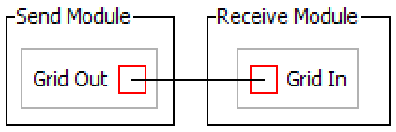
## Creating Rendering Pipelines

OpArtGen produces images using the rendering pipelines that you define.

Each rendering pipeline is a branching tree of connected modules stored by a graph. Each tree is rooted at a Canvas Module. Multiple rendering pipelines may be created in the same document, and each Canvas Module renders its own separate image using the modules at the nodes in that canvas' pipeline. Leaf node modules create data objects that are sent up the tree by the rendering engine. Modules at each intermediate node then process the data objects received from their child nodes until the final data object is received by the Canvas Module at the root node.

The Canvas Module then produces a Java *Canvas* object with an overridden *paint* method that can draw the grid data to any Java *Graphics* object. This allows the GUI to draw the image to screen or the Batik library to convert the calls to the *Graphics* object to an SVG file.

Modules are connected to each other using ports. A port provides an interface for a module allowing it to be connected to other modules present. When two modules are connected together by their ports the module objects are joined together in the graph of the rendering pipeline. Ports are of a specific type, and each type allows the sending and receiving of data objects of a given type. A port may either be a output (send) port or an input (receive) port with the obvious consequences for its ability to either send or receive data objects. Send and receive ports of the same type on different modules may be connected to each other allowing directed transmission of data objects through the modules in a rendering pipeline.



In the above screenshot of a rendering pipeline in a structure diagram we can see there are two connected modules. One module has an output port for sending grid data that is connected to an input port of another module for receiving grid data.

Rendering pipelines in OpArtGen may be constructed visually using the Structure Diagram editor window in the GUI, or created manually in a script file using calls to the OpArtGen API and then evaluated programmatically by the *start.RunScript* class.

There are two types of components present in OpArtGen: built-in and external. Built-in components are those that are compiled into the program and are always present when the program is run. These provide a set of core functionality for creating images.

External components are those that are written separately by users, and are then compiled and imported at run time to provide any extra functionality not provided by the built-in modules that is needed to create an image of a different style. This extra extensibility ensures that the art the program is able to create is not limited by the set of available modules. The program may be adapted to any given artistic brief.

When you are using the GUI to create images, external components are indistinguishable from built-in components. When using precompiled scripts to create images there are some differences in the way external components are accessed to those that are built-in. These differences are detailed in the section: *Using external components in scripts* (Page 38).

## *Creating a Rendering Pipeline That Works*

When creating a rendering pipeline for OpArtGen to create an image you should create and connect a set of modules using the following four steps as a guide:

1. At least one grid must be created to control the placement of elements in an image using a Grid Module
2. Subsequent modules in the pipeline add desired elements to the grid
3. Extra leaf node modules modify the properties of grid elements or combine grids
4. A final Canvas Module renders the completed grid

### *1. Creating grid data objects*

The first step is achieved by creating a number of Grid Modules. Each Grid Module creates an *m* x *n* grid data object that it sends to any module connected to its *Grid Out* port. A grid data object encapsulates a two dimensional array of cells. The grid dimensions *m* and *n* may be changed in the GUI via the Grid Module's properties window or using the *setNumCols* and *setNumRows* methods in a script.

Multiple grid data objects may be combined using *Merge* modules allowing complex grid layouts to be created. Each Merge module allows two grid data objects received from two other modules to be encapsulated in a single grid data object that is then sent along the rendering pipeline. There are several built-in Merge modules allowing the two input grids to be joined in the following configurations: Left-Right, Top-Bottom and Above-Below.

### *2. Adding elements to the grid*

Elements are added to the cells in a grid by the Shape Repeater Module or the Grid Repeater Module. Once you have created the correct module for the element you wish to add to the grid, then the Grid Out port of the Grid Module must be connected to the Grid In port of the repeater module.

The Shape Repeater Module allows predefined shapes to be added to cells in a grid. Some shapes are included with the program, and new shapes may easily be added to the program as detailed in the section: *Adding Extra Shapes* (Page 82). Shapes are stored as SVG image files and are extracted and converted to Java Shape objects using the Batik library. Once a shape is selected its dimensions proportional to its parent cell's dimensions may be set: e.g. 100% of the width and 50% of the height of the parent cell. A default colour may be chosen that is applied to all shapes added to the grid to save having to use a colour module further down the rendering pipeline.

The Grid Repeater Module allows you to take a grid and repeat it inside the cells of another grid. This allows you to combine shapes together in one grid to make a composite shape, and then tile that new shape across another grid.

Each cell in a grid may hold a renderer object. A renderer object provides a paint method that allows the rendering engine to draw the cell's contents onto a given Java Graphics object in the correct location. A number of built in renderers are provided that allow either Java Shapes or grids to be stored and output by each cell. To add other elements to the grid, such as SVG images or text, which are not covered by the built-in modules would require the creation of a new external renderer and external repeater module for each new element type. More details on how to do this may be found in the section:

Creating external components for OpArtGen (Page 76).

Additional instructions on how to use the built-in modules may be found in the sections: *Implementation of Core Modules* (Page 64) and *Appendix 5: API Reference for Scripting* (Page 106)

### *3. Modifying the grid spacing or properties of grid elements*

The grid spacing may be algorithmically modified by the Surface Warp Module to achieve effects that give the appearance of a distortion of the canvas surface in 3D. This module allows the selection of a Surface Warp. A number of built-in surface warps are included, and external ones may be created as detailed in the section: *Creating new Surface Warps.*

The colour of grid elements may be modified using the Colour Module or the Colour Series Module. Other properties of grid elements may be modified over time by connecting the Value Out ports of Oscillator Modules to the Value In ports of modules in the rendering pipeline. Further details on animating images are given in the section: *Animation of Images* (Page 57).

### *4. Rendering the completed grid*

Finally a canvas module must be created and connected to the Grid Out port of the last module in the rendering pipeline. The Canvas Module allows a background colour to be chosen along with the final image dimensions, and optionally for an empty margin to be added around the image. The image may then be rendered to screen in a Canvas Window or output in SVG format to a file. The method to do so varies depending on whether the image has been created in the visual editor or using a script file.

## Launching OpArtGen

Two main methods are provided to launch OpArtGen in its two modes. The first is found in the class *opArtGen.start.RunGUI*, takes no arguments, and launches the visual Structure Diagram editor. The second is found in the class *opArtGen.start.RunScript* and executes the script file located at the path supplied to the program as an argument.

OpArtGen uses three third-party Java libraries and also requires that the Java runtime **from the JDK** is present, with a version no less than Java SE 6. This is so the Java Compiler API may access the Java compiler contained within tools.jar. The four required libraries are Batik, Relative Layout, JCommon and the JDK tools.jar. The appropriate library jar files are bundled with the distribution of OpArtGen, and the jar files for then these libraries must all be present on the classpath when OpArtGen is run.

### Required Libraries

- **Batik 1.7:**
  http://xmlgraphics.apache.org/batik/
  *Handles SVG input and output*
- **JCommon 1.0.12:**
  http://www.jfree.org/jcommon/
  *Handles object cloning for Java2D API objects*
- **RelativeLayout:**
  http://www.andrew.cmu.edu/user/firebird/relativelayout/

Used to lay out module properties windows

- **JDK SE 6** (for tools.jar):
  http://java.sun.com/javase/downloads/
  *Required by Java Compiler API to compile code from on the fly*

Note that a Java runtime version 6 or greater is required to support the scripting language framework and the compiler API.

## Using the Visual Structure Diagram Editor to Create Images

The first window that opens when running OpArtGen using the main method in *RunGUI* is the visual Structure Diagram editor window. This editor provides a blank diagram to add modules to. The status bar at the bottom of the window provides you with some helpful information on whatever element in the structure diagram the cursor is hovering over. The Java console is also used to give you additional information on any modules that are created and removed, and any problems caused when trying to connect modules together.

The rendering pipeline is represented visually in the editor as a graph. Modules are arranged in the window with the leaf nodes towards the left side and the root canvas nodes on the right side. Data objects flow from left to right through the connected modules in the diagram, and connections are represented by lines drawn between modules.

There are two ways you can add modules to the structure diagram: via the Structure → Create Module submenu on the main menu or via the context menu opened by right clicking in the editor window at the location you wish the new module to be placed. Each module is displayed in the editor window as a rectangle containing the module name and any ports. The ports are split into two lists: one list is displayed on the left containing that module's input ports and one list on the right containing the output ports. As a visual aid, each port type has a corresponding port colour. An example module is shown below:

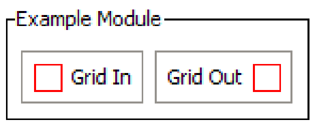


**Figure 15: Example module as displayed in a Structure Diagram**

Ports on different modules may be connected together by clicking on the coloured square found to the left or the right side of the port's name, and then dragging the cursor until it is over the port you wish to connect to and releasing. While the cursor is being dragged a red line appears between the source module and the cursor showing that a connection is currently being made. If the connection is a valid connection then a solid line will be drawn between the two ports once the mouse button has been released. Valid connections are those that satisfy the following constraints:

- The connection is between two ports of the same type. (i.e. same colour)
- Each port is on a distinct module
- The connection does not create cycles in the rendering pipeline's graph

Once a connection has been made it may be removed by right clicking on either constituent port and selecting *Disconnect* from the context menu that appears. The connection line will be removed and the rendering pipeline flushed.

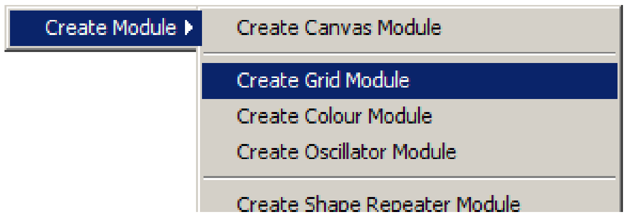
So that the structure diagram remains clear and legible, modules may be moved around the diagram by clicking and dragging on the area by the module name. The mouse cursor will change to a movement cursor to indicate when it is over a draggable area.

Module parameters may be changed manually using the GUI in the properties window for each module. To display the properties window for a module: right-click on the desired module and select *Properties* from the context menu that appears. The properties window for that module will be displayed. Any changes made to parameters in this window will not be saved to the module until either the *OK* or *Apply* button is clicked. Clicking *Cancel* allows all changes made to be discarded.

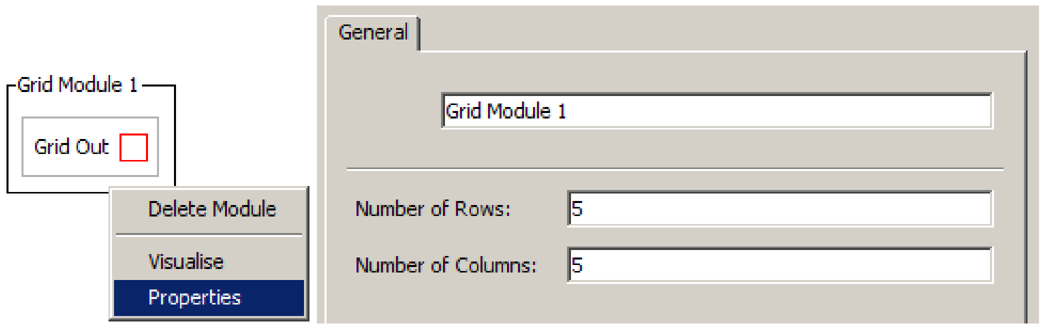
## *Example: Creating a chequered image*

We now apply the four steps previously outlined to create an image using the visual editor.

First we must create a grid that we can put the squares in. Open the Structure Diagram editor and right click on the empty diagram towards the left hand side of the window. Because data flows from the left to the right it helps keep things neat if you place leaf nodes towards the left of the diagram. Choose “Create Grid Module” from the Create Module submenu.



A new Grid Module named “Grid Module 1” will appear in the same location that you right clicked at. Right click on the background of the module panel next to its name label. Before you click the mouse cursor should be displaying a movement cursor with four arrows otherwise the cursor is not in the right location. Choose “Properties” from the context menu that appears.



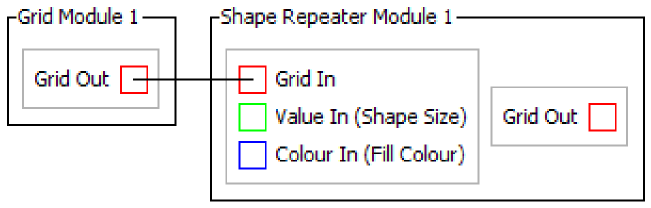
Grid Module 1

Number of Rows: 5

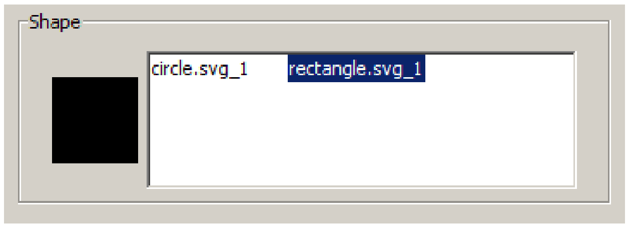
Number of Columns: 5

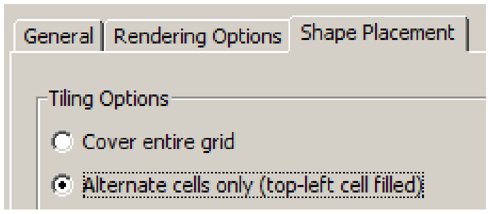
The properties window for the Grid Module will appear. In the General tab that is shown when the window opens you will see two text input fields where you can enter the number of rows and the number of columns that the grid produced by the module should have. Enter 5 in each field and press OK.

Second, use the same method as before to create a Shape Repeater Module in the structure diagram to the right of the Canvas Module. You now need to connect the "Grid Out" port on the Canvas Module to the "Grid In" port on the Shape Repeater Module. To do this move your cursor over the "Grid Out" port until it changes to a crosshair; then click and drag it until it is over the "Grid In" port and release it. If you were successful the red line between the two ports will turn into a black line:

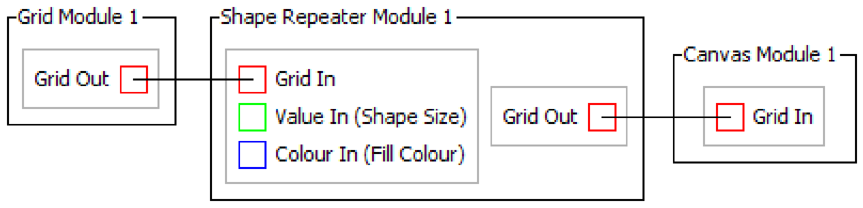


Open the properties window for the Shape Repeater Module. Choose a rectangle from the Shape box in the General tab, select black as the fill colour in the Rendering Options tab, and chose alternate cell shape tiling from the Shape Placement tab.

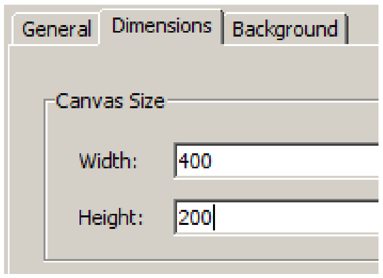




Finally we need to create a Canvas Module that we can pass our grid of shapes to so that it may be rendered to screen. Right click on the structure diagram to the right of the Shape Repeater Module, and create a Canvas Module. Connect the "Grid Out" port on the Shape Repeater Module to the "Grid In" port on the Canvas Module. The final structure diagram should now look like this:

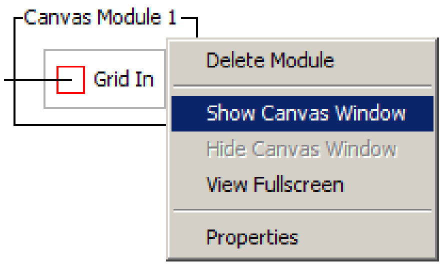


Use the Canvas Module's properties window to change size of the final image. Click on the Dimensions tab, change the height to 200, and select OK. To display the final image on screen in a Canvas Window, right click on the background of the Canvas Module and select “Show Canvas Window”.



Width: 400

Height: 200



The image that appears in the canvas window will look exactly like the final image we generated on page 29.

## Using Scripts to Create Images

OpArtGen scripts may be written either in JavaScript using the open source Mozilla Rhino language extensions and run in an interpreted mode; written as Java source files which are automatically compiled, loaded and run; or compiled to Java class files which are loaded and run.

In the above methods one makes calls from the script to the appropriate API methods in order to construct a rendering pipeline, and then render the image to screen or file. The benefit of using a script to create an image over using the visual Structure Diagram editor is that the rendering pipeline is exposed. This allows more complex animations to be created using Java timers where the pipeline itself is changed over time; or even the use of techniques such as Genetic Algorithms to produce generative art.

### Using Java to write scripts

The OpArtGen API provides an interface for writing scripts called *Script*, located in the package *opArtGen.API.scripts.* This interface defines a method *run* that takes no parameters and has a void return value. To write a script using Java code it is necessary to place your code inside a *run* method in a class that implements the *Script* interface. The script class may then either be compiled, and the compiled class file passed to RunScript as an argument; or the Java source file itself passed as the argument.

Precompiled class files provide the quickest execution speed as the only overhead is the initial class file loading and calling of the *run* method using Java reflection. Java source files incur an additional compilation overhead prior to the loading and calling of the *run* method. With Java source and compiled class files reflection must be used to access fields and methods of external components which results in increased code complexity. To get around this problem scripts may be written in JavaScript, however this will incur an extra execution overhead due to its interpreted nature.

### Using built-in components in scripts

When programming scripts using built-in components, it is only necessary to ensure that the OpArtGen jar file and required libraries are on the Java classpath. All component classes may then be imported and used in the scripts as normal. For example, using Java:

```
import opArtGen.API.modules.CanvasModule;
...
CanvasModule canvas = new CanvasModule();
```

### Using external components in scripts

External components that are loaded at runtime pose a problem. As it is not known prior to runtime which external components will be loaded and what their class information will be, it is not possible to compile a script written in Java that references an external component ahead of time directly, instead it is necessary to use the factory returned by the relevant component loader object's *getFactory* method to create an object instance. For example:

```
AbstractModule external = (AbstractModule)
  ComponentLibrary.getModuleFactory()
    .createComponentInstance("ExternalModule");
...
grid.getOutputPort(0).connect(external.getInputPort(0));
...
try {
      Class<?>[] types = {String.class};
      Method method = external.getClass()
        .getMethod("setPropertyValue", types);
      Object result = method.invoke(external,
                                       new Object[] {"newValue"});
} catch (Exception ignore) {}
```

Notice that you are able to easily access any methods defined by *Module* or *AbstractModule*; however you must use reflection to access property methods specified in the external module class unless you have the correct Java source or class file available to import when you compile your script.

### Using JavaScript and Rhino to write scripts

OpArtGen is capable of running scripts written in JavaScript in an interpreted mode rather than a compiled mode. As this feature is built around the Mozilla Rhino script engine it is still possible to access all the Java objects and methods alongside the OpArtGen API. Because compilation is no longer required it is possible to directly access any module types that have been loaded at run-time and removes any need to use reflection.

## *Example: Creating a chequered image*

We now apply the four steps previously outlined to create an image using the Rhino scripting engine.

In order to import class definitions into the Rhino engine calls must be made to the method *importPackage*. A top level object *Packages* is provided to allow you to specify package names. (Note that packages are not imported recursively.) In our case we need access to the classes in the Java awt; and OpArtGen API, modules, and rendering packages:

```
importPackage(Packages.java.awt);
importPackage(Packages.opArtGen.API);
importPackage(Packages.opArtGen.API.modules);
importPackage(Packages.opArtGen.API.rendering);
```

Now we have access to the classes that we need, we can create the three modules for our rendering pipeline:

```
// Create modules
grid = new GridModule();
shape = new ShapeRepeaterModule();
canvas = new CanvasModule();
```

Connecting the modules is done using the modules' *Port* objects. To retrieve the ports use the method *getInputPort(number)* or *getOutputPort(number)*. The numbers are 0 indexed and correspond to the order ports are displayed in the visual structure diagram editor. The port numbers and their functions are also listed for each module in *Implementation of Core Modules* (Page 64).

```
// Connect modules
grid.getOutputPort(0).connect(shape.getInputPort(0));
shape.getOutputPort(0).connect(canvas.getInputPort(0));
```

We also need to load the square *Shape* object ready to pass to our Shape Repeater Module. To do this, use the shape factory that you can retrieve from the *ComponentLibrary.* The name of the shape corresponds to the SVG file we have in the external/shapes directory, and the index number of the shape in the SVG file.

```
// Load shape
shapeFactory = ComponentLibrary.getShapeFactory();
rectangle =
shapeFactory.createComponentInstance("rectangle.svg_1");
```

Now we can set property values for our modules. First we need to specify to our Grid Module that we want it to output a 5x5 grid object:

```
grid.setNumRows(5);
grid.setNumCols(5);
```

Next we set the *Shape* object our Shape Repeater Module should use, and instruct it to only tile the shape on alternate cells ensuring that the top left cell in the grid is filled: we use the *TilingOptions* enum to specify this.

```
shape.setShape(rectangle);
shape.setTilingOptions(TilingOptions.ALTERNATE_CELLS_FIRST_FILLED
);
```

We also wish to ensure that our squares are drawn with a black fill: to do this we use a ShapeRenderingOptions object with the *fillPaint* field set to a black colour:

```
renderOpts = new ShapeRenderingOptions();
renderOpts.drawLine = false;
renderOpts.drawFill = true;
renderOpts.fillPaint = Color.BLACK;
shape.setRenderingOptions(renderOpts);
```

Finally we can set the desired size of our canvas to 400x200 and render it to screen:

```
canvas.setSize(400, 200);

// Display image
canvas.setCanvasWindowVisible(true);
```

Again, the image that appears in the canvas window will look exactly like the final image we generated on page 29.

## Software Architecture

The software is built around two main components, the rendering API and the user interface. The design of these is such that the rendering API may be used without the whole user interface being present, and if desired, the user interface could be used to control a new rendering API with different internal implementation.

All components are located in the root package opArtGen. The package *opArtGen* contains 4 sub-packages, two of which correspond to the GUI and the API and two of which contain utility classes. The root package additionally contains a class called *Config*, which contains a large number of constant object instances and values. It is used to externalize configuration options for the entire project and simplify maintaining compile-time variables.

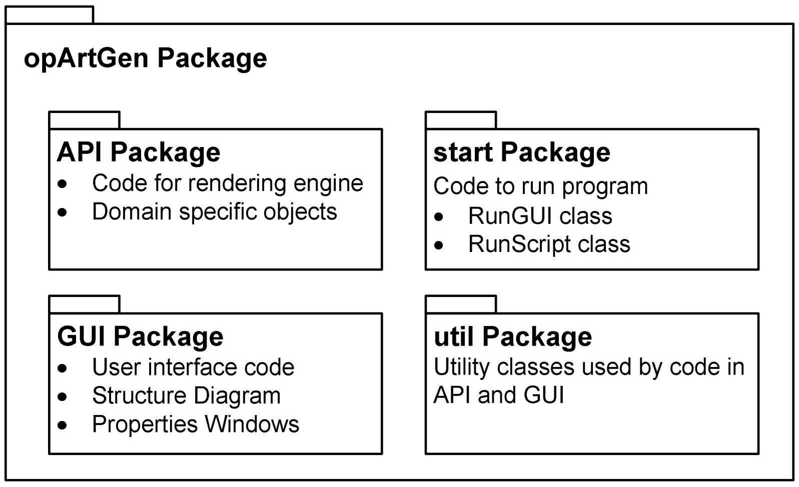


**Figure 16: OpArtGen Packages**

### API Package

The API is at the heart of OpArtGen, and contains the core code that the rest of the program is built around. The package contains:

- Classes for all the core components that have been implemented
- Any external component classes created by the user
- Domain classes used to model the rendering pipelines
- Classes that control the rendering engine
- Loader and Factory classes to allow external API components to be loaded at run-time
- Classes enabling scripts to be loaded and run

These classes are divided between the API package itself and a further 8 sub-packages:

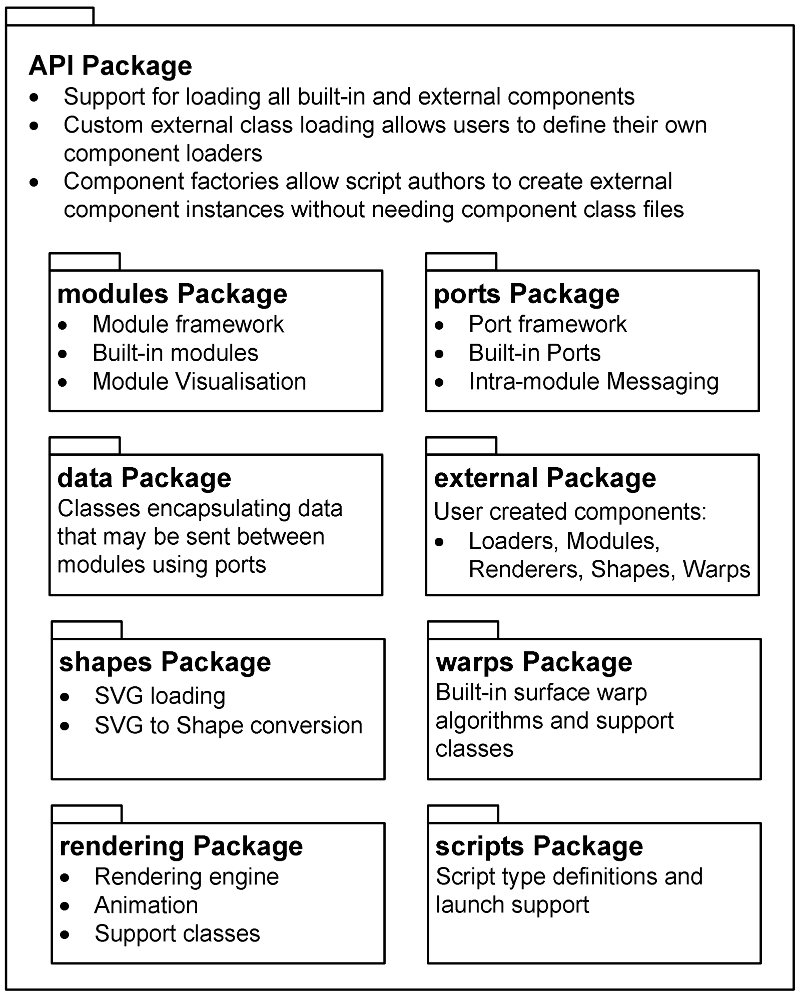

API Package
Support for loading all built-in and external components
Custom external class loading allows users to define their own component loaders
Component factories allow script authors to create external component instances without needing component class files
modules Package
Module framework
Built-in modules
Module Visualisation
ports Package
Port framework
Built-in Ports
Intra-module Messaging
data Package
Classes encapsulating data that may be sent between modules using ports
external Package
User created components:
Loaders, Modules, Renderers, Shapes, Warps
shapes Package
SVG loading
SVG to Shape conversion
warps Package
Built-in surface warp algorithms and support classes
rendering Package
Rendering engine
Animation
Support classes
scripts Package
Script type definitions and launch support

**Figure 17: API Packages**

## The Rendering Engine

As we have seen the rendering engine generates images using rendering pipelines. How does the engine actually use the pipelines to output images? We will now take a look at the various components that make up the engine and the pipelines in more depth to answer that question.

Producing an image from a pipeline is a two stage process. The first stage, the Refresh Cycle, is performed by the *RenderingManager* class and allows the modules in the pipeline to work together to produce a grid data object containing a specification of what should be drawn in the image. The second stage is then performed by the root Canvas Module in each rendering pipeline when it receives the grid data object produced by the Refresh Cycle. A Java *Canvas* object with a series of nested *paint* calls is produced from the grid object that can draw the image represented by the grid to any Java *Graphics* object. This *Canvas* is passed to the GUI which is instructed to repaint itself.

### The Refresh Cycle

The framework of the module and ports together contain a simple messaging protocol implemented using method calls. These method calls together allow the engine to perform the *Refresh Cycle*. Each Refresh Cycle causes fresh data to be pushed up from the leaf nodes in a pipeline's tree, through the intermediate nodes where it gets processed, up to the Canvas Module at the root node.

The RenderingManager class controls the rendering process and initiates the Refresh Cycle for each rendering pipeline. When the method *updateCanvases* is called, all *CanvasModule* instances get their asynchronous method *updateRenderingPipeline* called. This spawns a new worker thread that causes a refresh request to be sent down the pipeline.

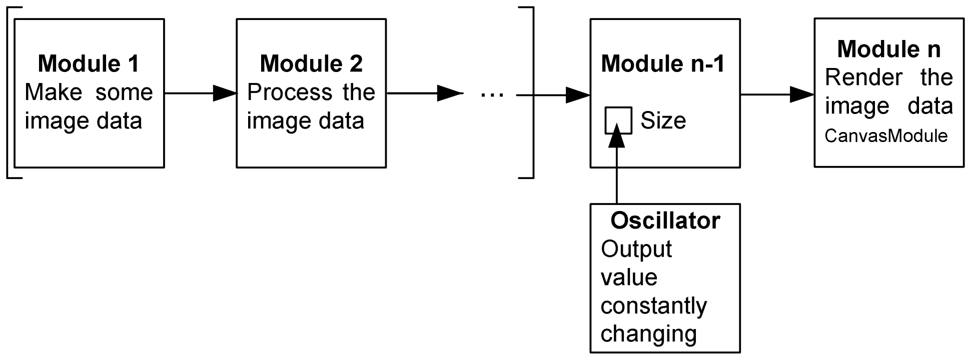
A refresh request is sent to a module by calling its *refresh* method. When a module receives a refresh request it executes the following algorithm:

```
ON RECEIPT OF A REFRESH REQUEST BY A MODULE:

     check whether the module's data is dirty:
     if dirty flag is set then do nothing
     else
          call refresh on any modules connected to input ports
          wait until refresh calls terminate
          process received data ready for output
          send data to any modules connected on output ports
          clear dirty flag

END
```

The refresh request is propagated backwards through the rendering pipeline causing each module to receive, process, and then output data. Because a module will propagate the request first, then wait until propagation is complete before processing the data stored in its data fields, the module is guaranteed to have received completely fresh data from all its child modules in the pipeline's tree once processing begins.

## Optimizing the Refresh Cycle

Notice that the refresh request will only cause a module to process its data if a **dirty flag** is set in the module. This allows the rendering engine to save processing any data unnecessarily. Consider the following rendering pipeline for large values of n:



We have a rendering pipeline similar to what we have seen before but with the last-but-one module having its size property varied constantly by an *Oscillator* module. The first time the image is rendered the engine must call the processing method for each module in the pipeline so that the CanvasModule can get the completed grid data object.

The second time the image is rendered the grid data sent out by Module n-2 will not have changed from the grid data it sent out the first time. This time then the engine does not need to call the processing method of any modules inside the square brackets as Module n-1 may reuse the data it previously received. The only new processing that must be done is for Module n-1 to reprocess that data based on the new value for its Size property and send it to the *CanvasModule* for output.

We call data that will be changed during a Refresh Cycle dirty data. If a module contains data that will change during a refresh request then its dirty flag should be set. When a module's data will change  it must notify the other modules further down the pipeline that they must set their dirty flag as they will be receiving fresh data. To achieve this, when a module's dirty flag is set using the method *setDirtyFlag,* a message is sent to any modules connected to its output ports requesting them to set their own dirty flags. This message will be propagated right through to the final *CanvasModule* in a pipeline.

Dirty flags may be set at any time independently of whether or not a Refresh Cycle is in progress at the time. If a refresh is in progress they will not be taken into account until the next refresh occurs as all methods involved with the Refresh Cycle are synchronized so must acquire locks on the module's object instance.

The method *setDirtyFlag* is called by setter methods for module properties and by the constructor method defined in the *AbstractModule* class. The convention is that property setter methods themselves do not call *updateCanvases* or *updateRenderingPipeline,* rather the client code that uses the setter method should call *updateCanvases* after all the desired changes to property values of modules in the pipeline have been made.

The following diagrams illustrate the sequence of messages that are sent and received by modules when first a property value is updated and then *updateRenderingPipeline* is called:

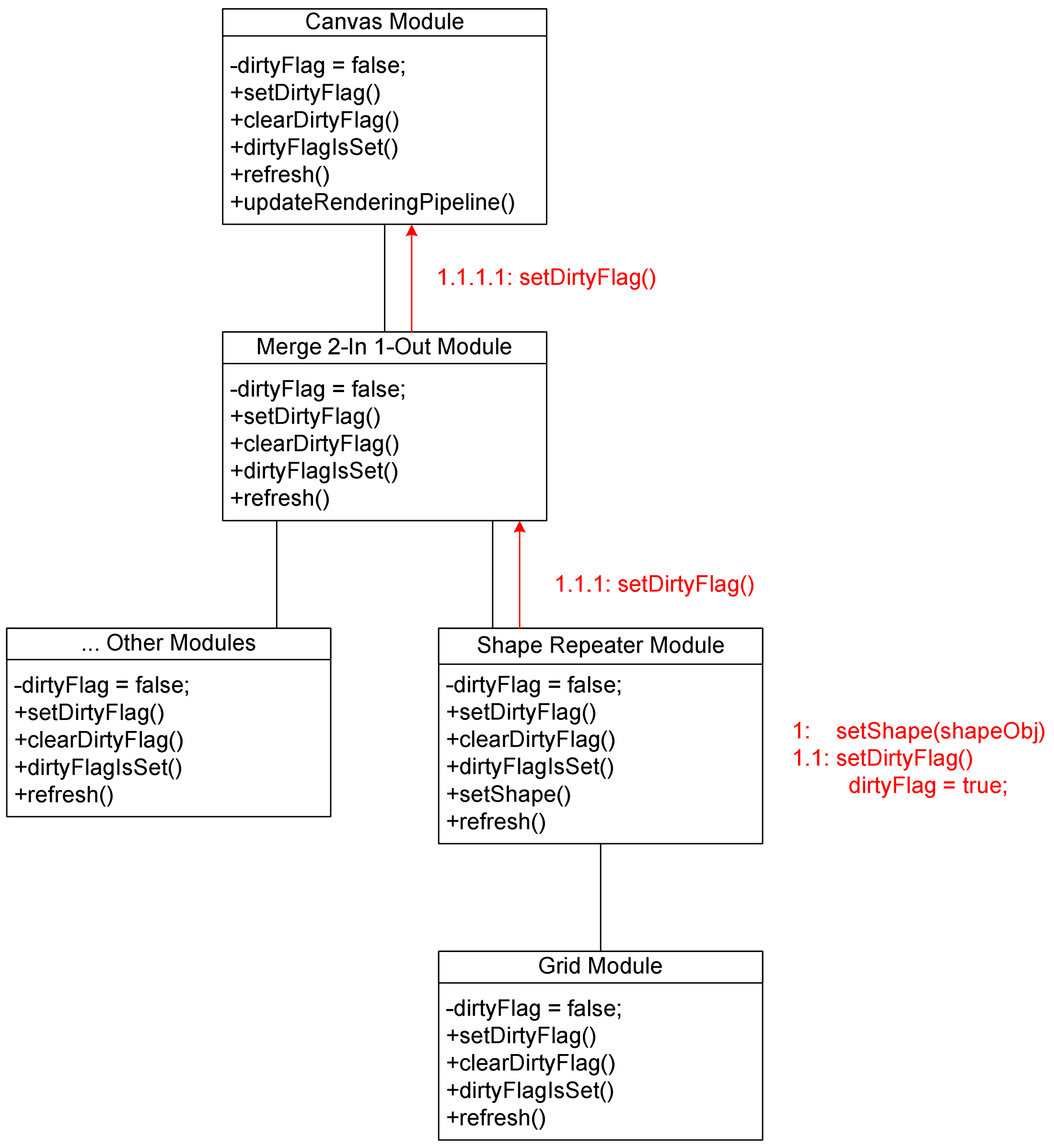


**Figure 18: Updating a Property Value for a Module - Setting the Dirty Flag**

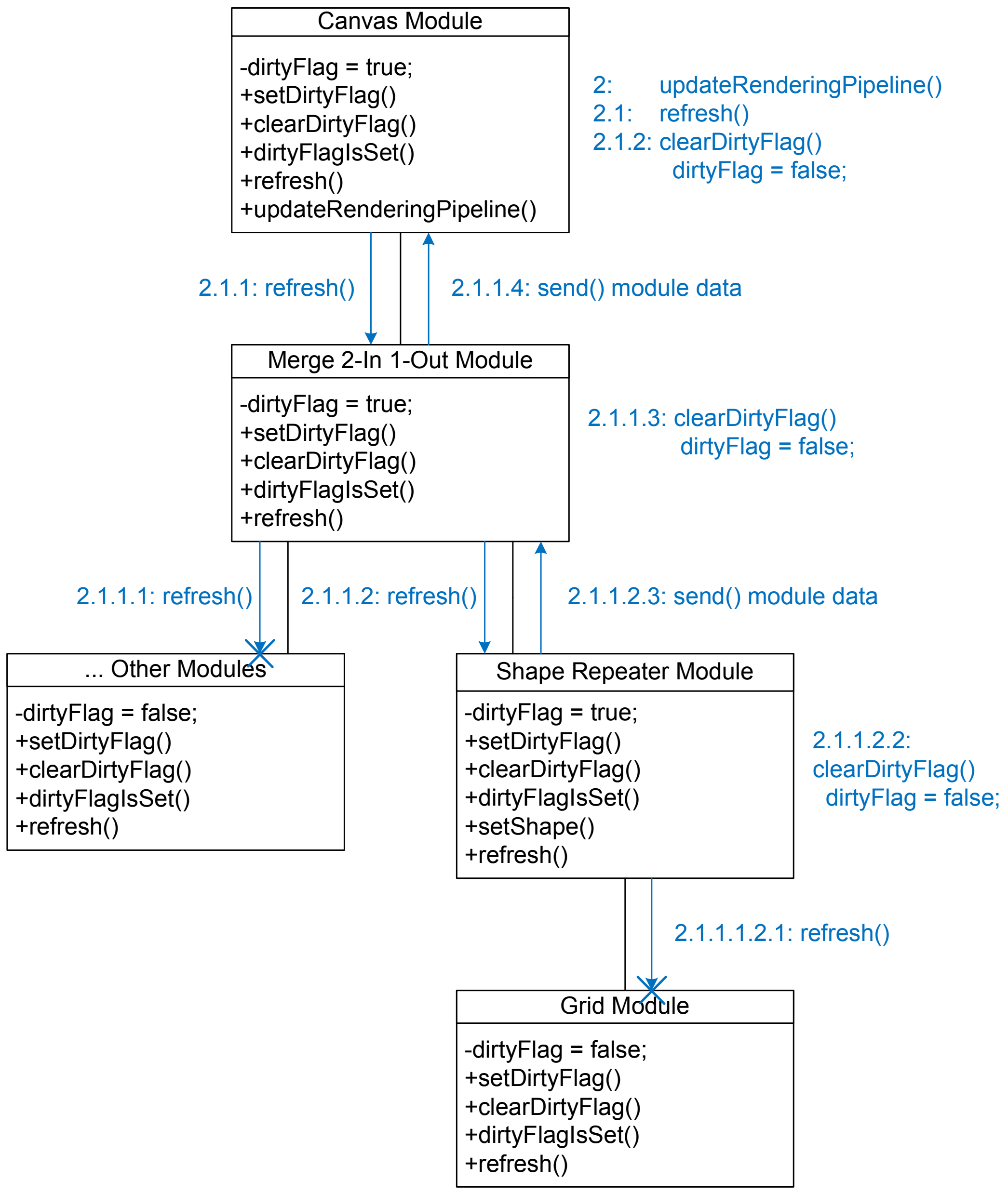


**Figure 19: The Refresh Cycle**

## Rendering Grid Data

The root data object for all image data in OpArtGen is the *GridData* object. Elements which may be placed in grids, such as shapes, are encapsulated in objects that implement the *CellRenderer* interface. These renderer objects store any size and position information for the elements and know how to draw the image data they contain onto a Java *Graphics* object. The *GridData* object itself contains an array of *Cell* objects, each of which may store a single *CellRenderer* object. The *Cell* objects act as placeholders for the elements stored in the *CellRenderer* objects.

When a *Cell* receives a *paint* request, it delegates the task to whatever *CellRenderer* object is currently stored in that cell. Two classes have been defined that inherit the *CellRenderer* interface and are able to store image elements: *GridCellRenderer* and *ShapeCellRenderer.* These are able to handle grids and shapes respectively. Another class has been implemented to act as a null renderer: *EmptyCellRenderer*. When a *Cell* object is first created, it is assigned an *EmptyCellRenderer*. The relationships between grids, cells, and renderers and the classes that they may render are summarized in the class diagram below.

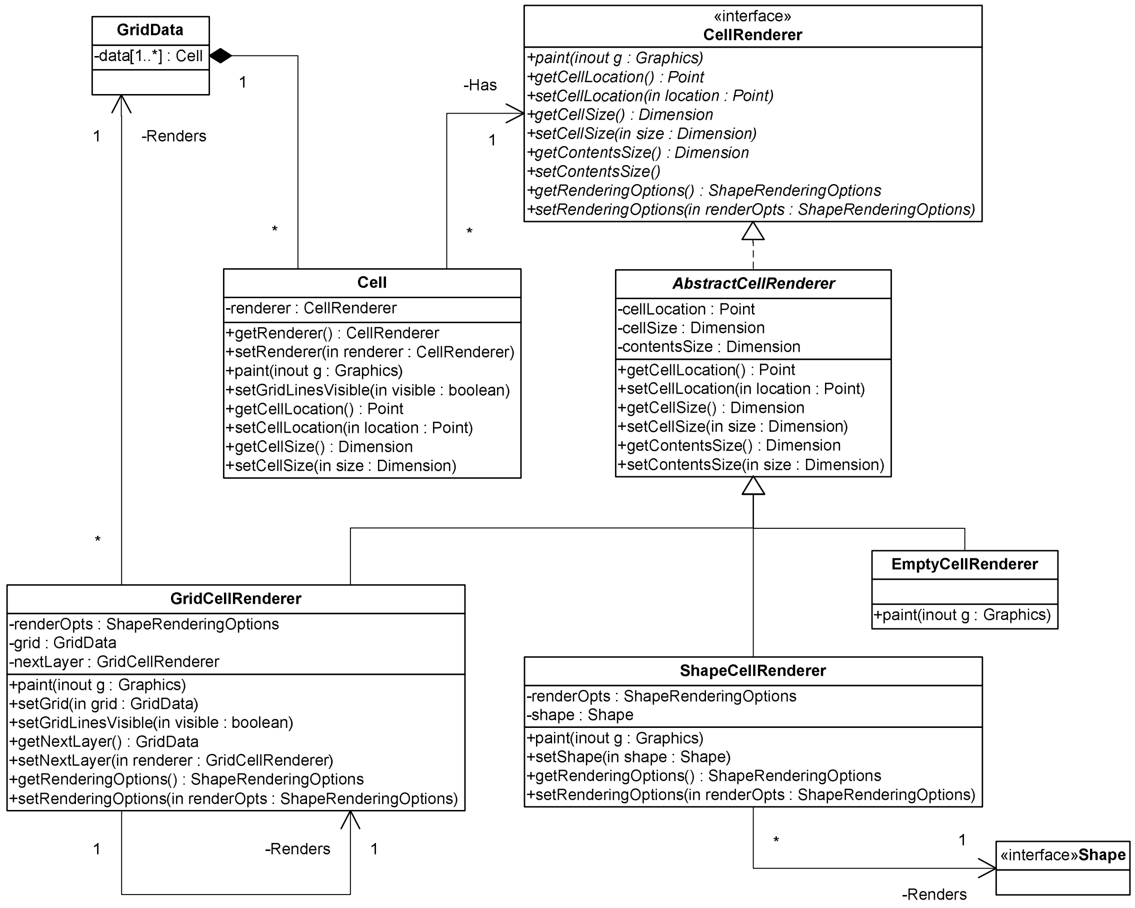

GridData
-data[1..*] : Cell
«interface»
CellRenderer
+paint(inout g : Graphics)
+getCellLocation() : Point
+setCellLocation(in location : Point)
+getCellSize() : Dimension
+setCellSize(in size : Dimension)
+getContentsSize() : Dimension
+setContentsSize()
+getRenderingOptions() : ShapeRenderingOptions
+setRenderingOptions(in renderOpts : ShapeRenderingOptions)
-Has
-Renders
Cell
-renderer : CellRenderer
+getRenderer() : CellRenderer
+setRenderer(in renderer : CellRenderer)
+paint(inout g : Graphics)
+setGridLinesVisible(in visible : boolean)
+getCellLocation() : Point
+setCellLocation(in location : Point)
+getCellSize() : Dimension
+setCellSize(in size : Dimension)
AbstractCellRenderer
-cellLocation : Point
-cellSize : Dimension
-contentsSize : Dimension
+getCellLocation() : Point
+setCellLocation(in location : Point)
+getCellSize() : Dimension
+setCellSize(in size : Dimension)
+getContentsSize() : Dimension
+setContentsSize(in size : Dimension)
EmptyCellRenderer
+paint(inout g : Graphics)
GridCellRenderer
-renderOpts : ShapeRenderingOptions
-grid : GridData
-nextLayer : GridCellRenderer
+paint(inout g : Graphics)
+setGrid(in grid : GridData)
+setGridLinesVisible(in visible : boolean)
+getNextLayer() : GridData
+setNextLayer(in renderer : GridCellRenderer)
+getRenderingOptions() : ShapeRenderingOptions
+setRenderingOptions(in renderOpts : ShapeRenderingOptions)
ShapeCellRenderer
-renderOpts : ShapeRenderingOptions
-shape : Shape
+paint(inout g : Graphics)
+setShape(in shape : Shape)
+getRenderingOptions() : ShapeRenderingOptions
+setRenderingOptions(in renderOpts : ShapeRenderingOptions)
«interface»Shape
-Renders
-Renders
1
*
1
*
1
*
1
1
*
1

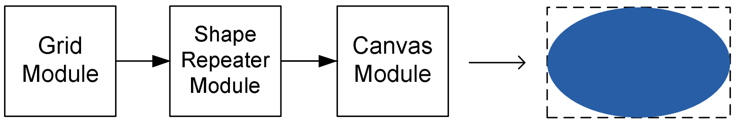
**Figure 20: Class Structure for Grids, Cells, and their Renderers**

Each *GridData* object is composed of a single layer of cells stored in a multi-dimensional array. Multiple image layers are supported by OpArtGen via the Merge Module. Rendering of multiple layers is handled by *GridCellRenderer* object, each of which may store a reference in the field *nextLayer* to another *GridCellRenderer* that stores the grid data for the layer above. The *paint* method for a *GridCellRenderer* once complete makes a call to *nextLayer.paint*. In this way, multiple layers can be chained together.

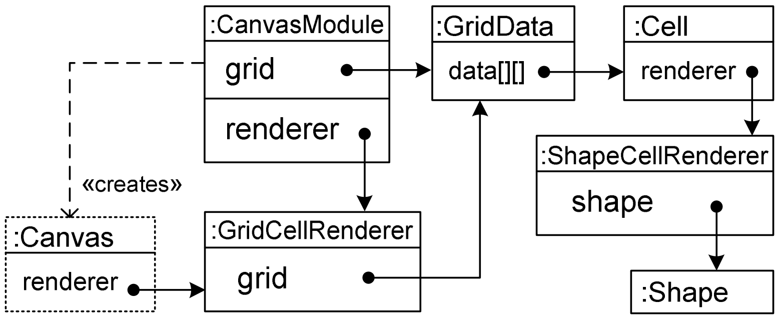
Once a refresh cycle has completed the root Canvas Module receives a *GridData* object containing the complete image data for its rendering pipeline. The Canvas Module stores the grid data in its *GridCellRenderer* object, and creates a new Java *Canvas* object with an overridden *pain*t *method* containing code to set the image's background colour, margins, and a single call to its renderer's *paint* method. This object is passed to the GUI, and when the GUI wishes to display the image it calls the *Canvas* object*'s paint* method. More information on the design of the GUI is provided in the section *GUI Package* (Page 60).

## *An Example*

To illustrate the rendering process, consider the following pipeline that draws a single shape in a grid with one cell. The image that should be output is shown to the right with the dashed box indicating the grid lines.



Once the refresh cycle has completed the simplified data structure of the *CanvasModule* object including the grid data received by the Canvas Module will be:



The method calls that create this data structure and then draw the image that it represents are illustrated in the following Interaction Sequence diagram:

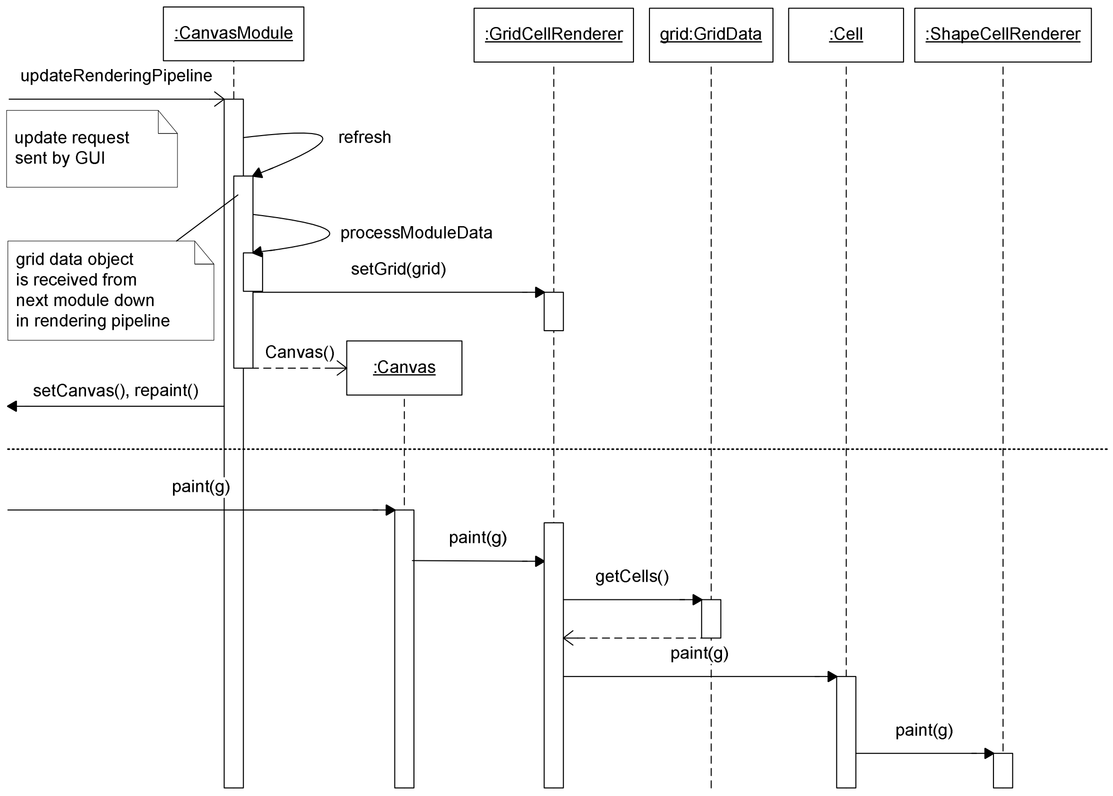


**Figure 21: Interaction Sequence for Image Rendering**

It is possible to save memory using this scheme with the flyweight design pattern. Instead of creating many shape objects with one per cell, a single shape object is created that gets shared between multiple cells. As the shape's position information is stored by its parent grid cell and its size and colour in each cell is determined by that cell's renderer object, only one instance of a *Shape* object is needed for each different shape used in the image.

### Visualisation of GridData

Certain modules support visualisation of the grid data that they output. When using the GUI this enables a user to preview what effect each individual module in the rendering pipeline is having on the final image output. Visualisation is achieved by the class *ModuleVisualiser*. When a *GridData* object is passed to a *ModuleVisualiser* object's *visualise* method, the Visualiser object creates a new Canvas Module, simulates a connection it and passes it the grid data to be rendered and displayed using the simulated connection.

## OpArtGen Modules

Modules are used to encapsulate algorithms for creating and processing image data that may be used by the OpArtGen rendering engine. All modules must implement the interface *Module*. This interface defines all the key methods modules must support so that they are able to interact with each other and are usable by both the engine and the GUI. The methods themselves fall into several groups:

- **Rendering Engine Support**

These methods allow the rendering engine to use the modules to process data. The most important method here is the *refresh* method, which is called during a Refresh Cycle. The other methods are for setting, clearing and testing a module's dirty flag. (*setDirtyFlag, clearDirtyFlag, dirtyFlagIsSet*).

- **Module Connectivity**
  Module connectivity methods provide a framework for connecting together module objects using their Ports. Methods are defined for returning the input and output ports of a module and for broadcasting cycle test messages to connected modules. These are messages designed to help check whether new connections between modules in a rendering pipeline would create any cycles in the graph.

- **User Interface Support Methods**
  Two methods are defined to enable the module to be used with a GUI: *getModulePanel* and *getPropertiesPanel*. The first is called by the GUI to retrieve a JPanel object that represents the module in the visual structure diagram editor; and the second to retrieve a JPanel that is able to change the property values of the module.

- **Other Methods**
  There are additional methods defined for getting/setting the module's name and a destructor method which is called when a module is deleted to aid clean-up.

No module classes should implement the Module interface directly; instead they should extend the abstract class AbstractModule. This provides an implementation for all the rendering engine support methods and instead only requires you to override a protected abstract method *processModuleData*. This method is called by its implementation of the *refresh* algorithm and is where the routine for converting input data into output data should be written. Further details on implementing new modules are given later in this report, please see: *Creating New External Modules* (Page 77).

### Module Data Types

Module data types encapsulate the data that modules send between each other in rendering pipelines to build the final image. They consist of classes that implement the *ModuleData* interface, encapsulating other data objects and providing getter and setter methods and a clone method. The data must be *cloneable* so that data that is sent out from each module is cloned. This prevents changes to data objects made by modules further up the rendering pipeline's tree from affecting the data being held and processed by those modules further down the tree.

Three core Module Data types have been defined for use by the core modules:

- *GridData* objects are used to encapsulate and clone the grids that make up images. Each object stores an array of cells, each of which may contain shapes or other grids.

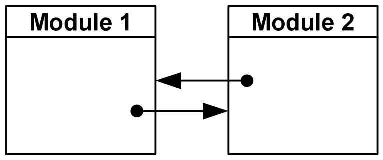
- ValueData objects are used to encapsulate a single *double* value. Accessor methods are provided to return the value as a *double* or an *int.*

- ColourData objects are used to encapsulate and clone a single Java Color value.

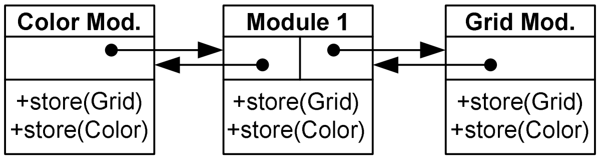
## Module Ports

Module ports are objects that allow you to connect modules together. They are the intermediary by which connected modules may communicate and send data objects between each other.

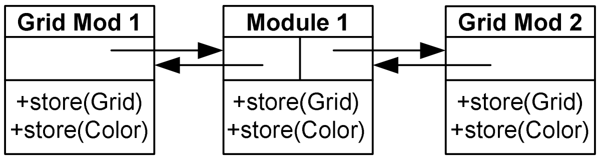
Why are ports necessary: why not directly connect together two modules using an object reference stored in a field in each module? Consider the following situation:



The two connected modules each have a reference to the remote module of the connection, and each module may send data objects to each other, via a method such as *store (dataObject)*. This scheme would work for simple modules that may be connected to one remote module. It could even deal with modules that may be connected to two remote modules, with each module sending a different type data and each data type being handled using method overloading:



Each overloaded method would store its data type in the correct local data field in Module 1. This is not sufficiently robust when we wish modules to be able to connect to an arbitrary number of other modules, with each connection sending/receiving an arbitrary data type:



Our scheme doesn't work with just two remote modules each wishing to transmit the same type of data, as both will end up calling the same overloaded *store* method and overwrite each other's data.

Instead we need intermediary objects to handle the connections. This is where *Ports* come in. Each module may create an arbitrary number of ports, with each port able to handle a specific type of data. Each port may be connected to one remote port, and send/receive data to/from the remote port. So that the ports know where in the module to read data from and write data to the module must supply the ports with *CallBack* objects. This scheme decouples modules from the implementation of communication in rendering pipelines, and decouples the ports from module specific data storage implementation.

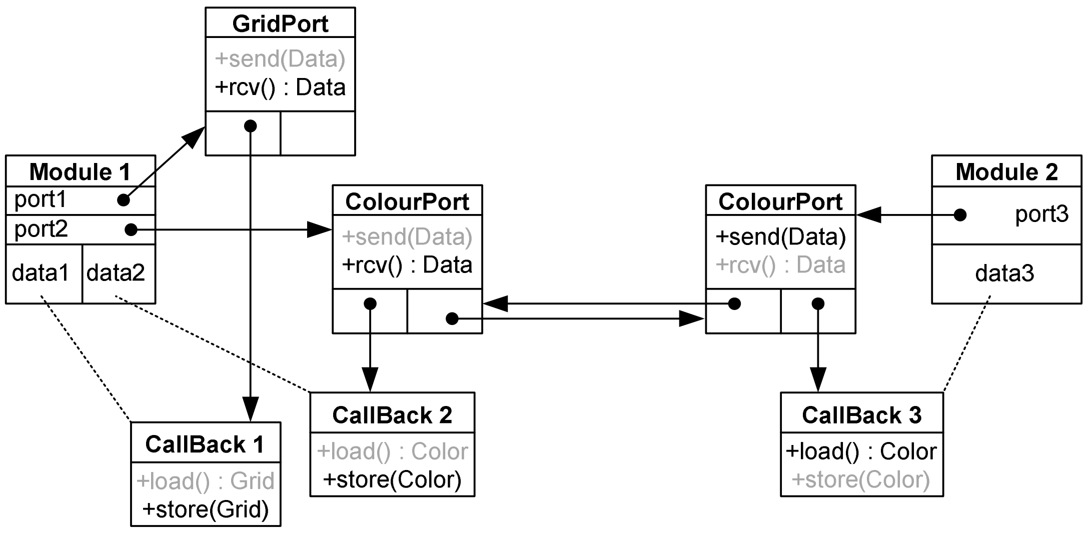


**Figure 22: Object Relationships Providing Module Connectivity**

Notice that ports may either send data, or receive data but not both. In a rendering pipeline, data always flows in one direction. Once created by a module's *initPorts* method, ports are stored using two protected List fields. These separate the ports used for sending data receiving data, and are called *outputPorts and inputPorts* respectively. They are defined in the *AbstractModule* class*.* Any ports a module creates must be placed in the correct list for their type.

Ports access the private module data fields using *CallBack* objects. CallBacks may either be read only or write only, this is determined by polymorphism: their objects are either an instance of *ReadOnlyCallBack* or *WriteOnlyCallBack*. CallBacks read and write data from a module using the methods *loadData* and *storeData. ReadOnlyCallBack* objects may only load data from their target module, and *WriteOnlyCallBack objects* may only store data in their target module. As you can see from the class diagram, the method *loadData* is only implemented in the *ReadOnlyCallBack* class, and *storeData* in the *WriteOnlyCallBack* class.

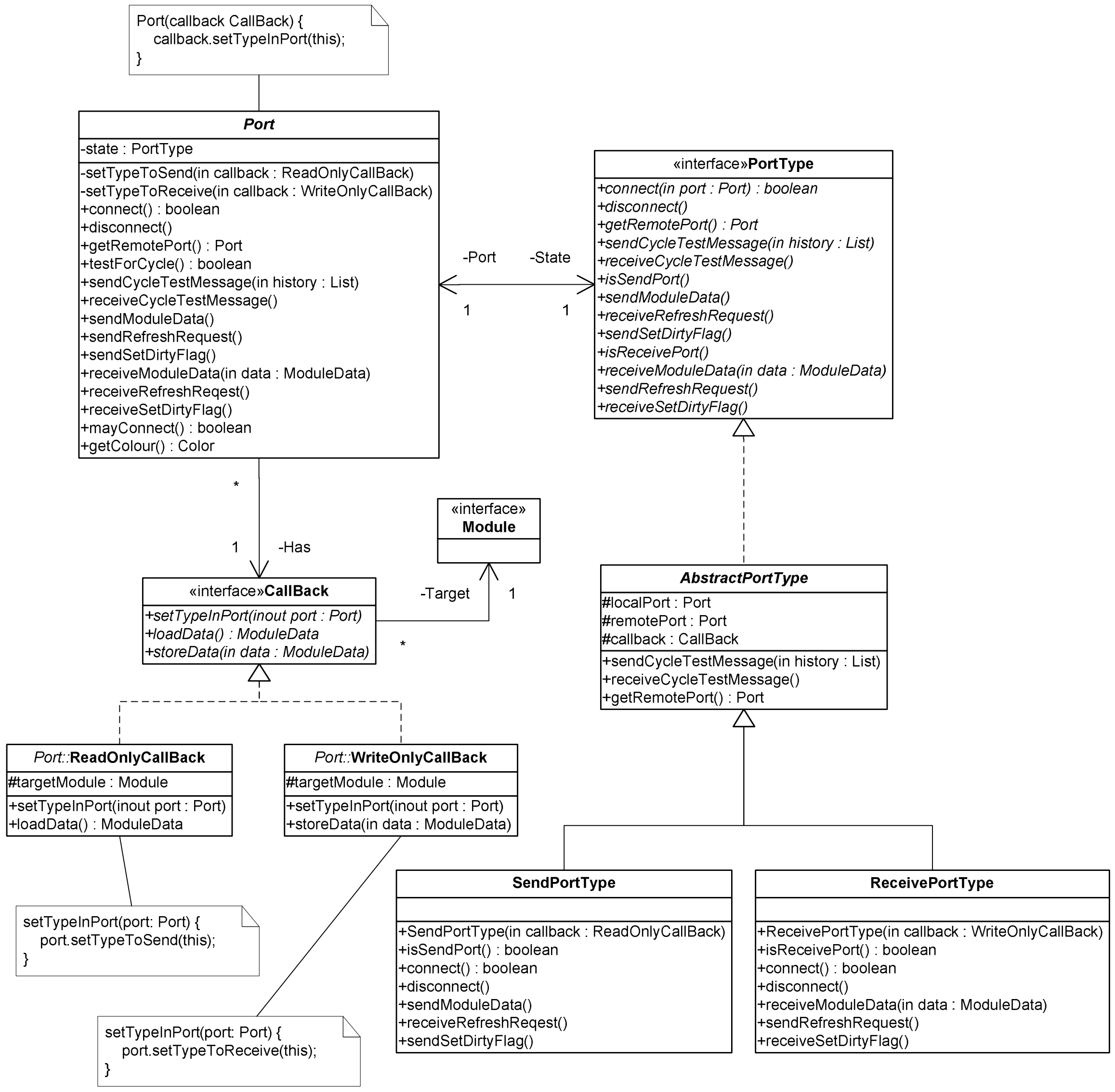


**Figure 23: Class Structure for Module Ports and their Internal States**

Port objects are designed using the State design pattern. Ports may be in one of two states: able to send, or able to receive. Their state is immutable and is determined when they are created by the CallBack object that is passed to the constructor. Notice that each class implementing the CallBack interface must define a method *setTypeInPort*. The constructor method calls the *setTypeInPort* method of the *CallBack* object it receives, which in turn either sets the port state to *SendPortType* or *ReceivePortType* depending on whether it is an instance of *ReadOnlyCallBack* or *WriteOnlyCallBack*.

The following interaction sequence diagram illustrates the calls that are made when a module creates a new port using a read only call-back and attempts to connect the new port to an existing remote port. The read only call-back sets the internal state of the new port to a send port. Note that a similar interaction sequence occurs if you create a port using a write only call-back, except you end up with a port that receives data only.

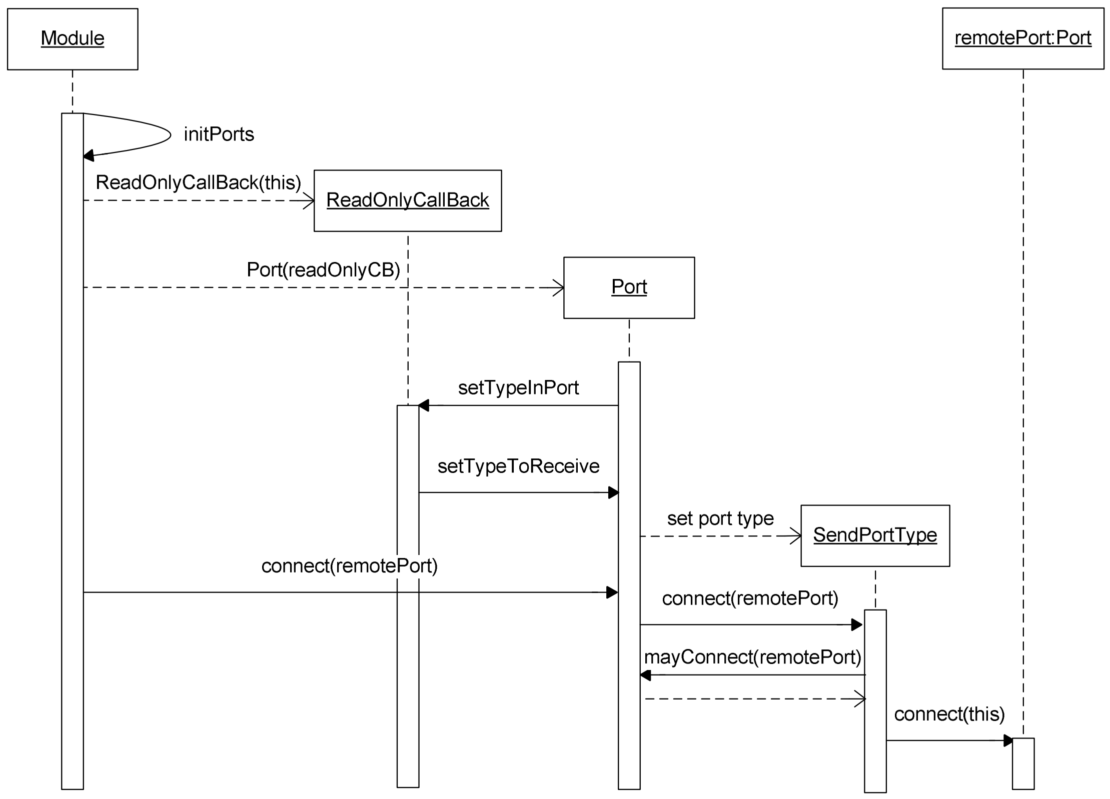


**Figure 24: Interaction Sequence for Creating Ports**

## Connecting Modules Using Ports

When an attempt is made to connect two ports together, the local port to the module that initiates the connection attempt must first check that the new connection would be legal. The steps that are taken before the connection is accepted are:

1. Check whether both ports are connectable: i.e. not already connected
2. Check that the remote port is on a different module (the target objects of each port's call-back objects are not equal)
3. Check that the remote port is of the same type: i.e. sends/receives same type of data
4. Check that the connection will not introduce a cycle in the pipeline's graph.
5. Check that the remote port's internal state is not the same as the local port's internal state (we must connect a send port to a receive port and vice versa)

When all tests have been satisfied then the local port then contacts the remote port to inform it a connection must be made. The local port stores a reference to the remote port, and the remote port stores a reference to the local port.

## Checking New Connections for Cycles

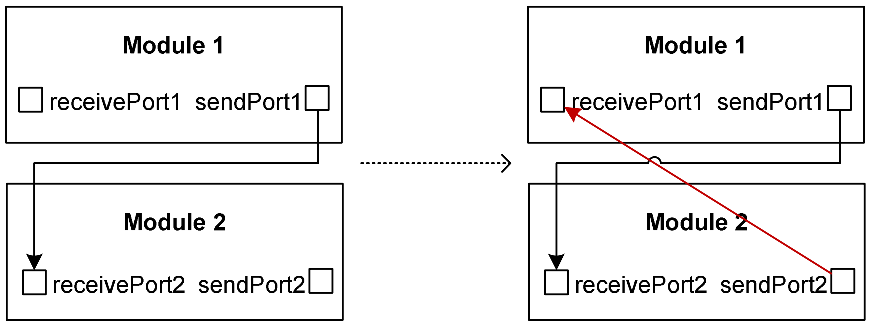
It is important to ensure that new connections between ports do not create cycles in the graph of the rendering pipeline. If a cycle is introduced then the refresh cycle will not terminate and calls to *refresh* will loop until a stack overflow error occurs.

The *Port* class defines a method *testForCycle* that takes a *Port* as an argument and checks whether connecting to it would create a cycle in the graph. The algorithm it uses to achieve this is outlined in pseudocode:

```
TO TEST WHETHER CONNECTING TO REMOTE_PORT INTRODUCES A CYCLE:
     testingForCycle := true

     send REMOTE_PORT: CycleTestMessage(history := new List)

     if CycleException is thrown then connection creates a cycle
     else a connection to REMOTE_PORT would not create a cycle

     testingForCycle := false
END

ON RECEIPT OF CycleTestMessage(history) BY A MODULE:
if history does not contain MODULE then
     add MODULE to history
     broadcast CycleTestMessage(history) to all ports in MODULE:
          if port.testingForCycle found during a broadcast
          then throw CycleException
END
```

The algorithm is a broadcast algorithm that may terminate in two ways. The first way is if a CycleException gets thrown. This only occurs if a module tries to get one of its ports to send a cycle test message, and that port was the port that originally initiated the cycle test. In this case we know that making a connection to the remote port would create a cycle in the pipeline's graph. The second way is if a cycle test message is sent to all ports successfully without any CycleException being thrown. In this case a connection could safely be made without creating a cycle.

To show what messages get sent during a call to *testForCycle* consider the following rendering pipeline consisting of two modules, each with one send port and one receive port. The send port of module 1 has already been successfully connected to the receive port of module 2. This is illustrated by the pair of modules on the left in the diagram below.



We now wish to connect the send port of module 2 to the receive port of module 1. A call is made to receivePort1.connect(sendPort2). The connect method runs through the first three checks: the ports are both connectable, the ports belong to different modules, and both ports send and receive the same type of data.

For the fourth check the following sequence of messages is sent between the objects:

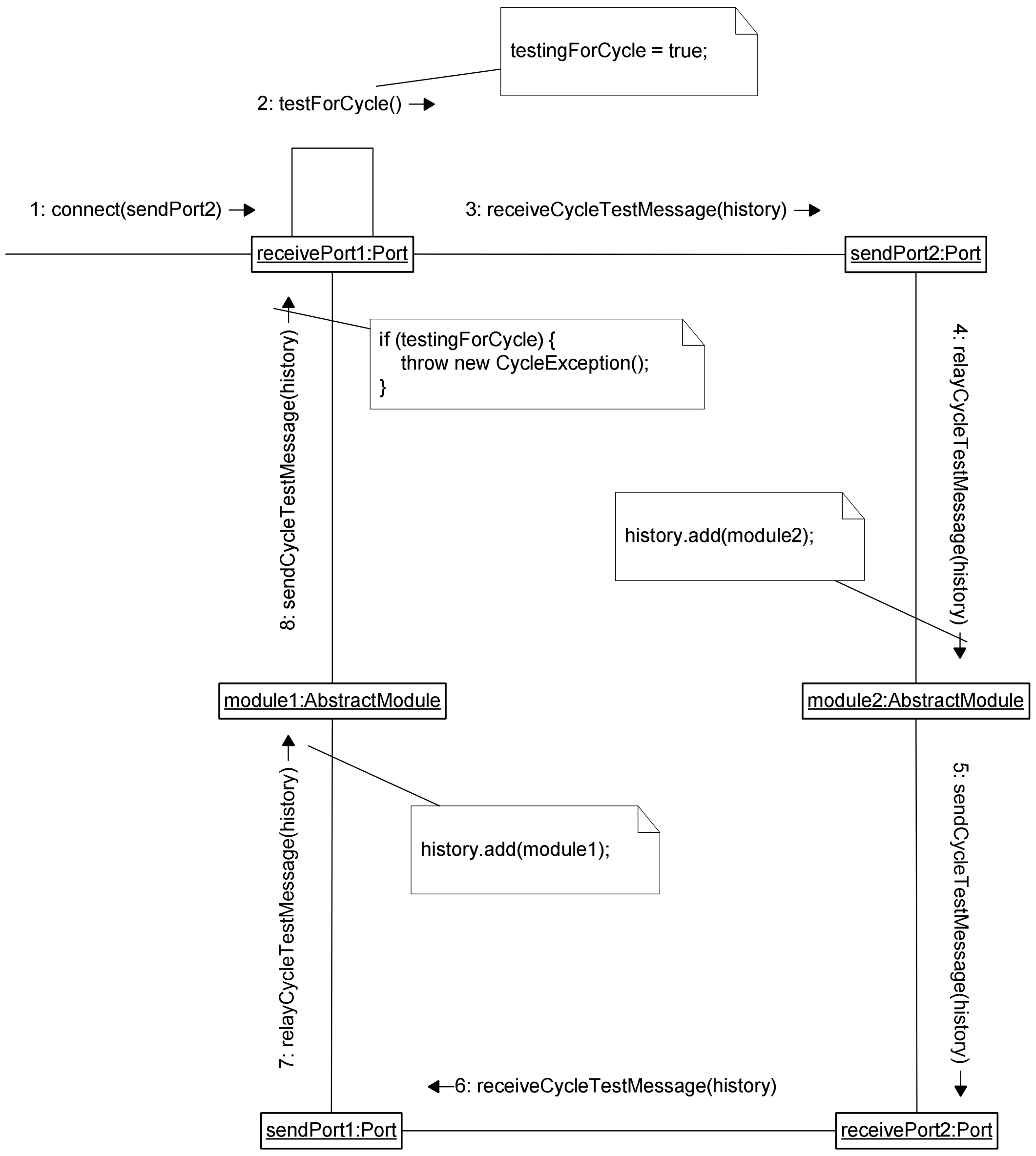

testingForCycle = true;
2: testForCycle() →
1: connect(sendPort2) →
3: receiveCycleTestMessage(history) →
receivePort1:Port
sendPort2:Port
if (testingForCycle) {
throw new CycleException();
}
8: sendCycleTestMessage(history) →
4: relayCycleTestMessage(history) →
history.add(module2);
module1:AbstractModule
module2:AbstractModule
7: relayCycleTestMessage(history) →
history.add(module1);
5: sendCycleTestMessage(history) →
←6: receiveCycleTestMessage(history)
sendPort1:Port
receivePort2:Port

**Figure 25: Collaboration Whilst Detecting Cycles**

As shown, the instance object for receivePort1 initiates the test from within its *connect* method by calling *testForCycle.* This sets a flag *testingForCycle,* and then *r*eceivePort1 sends a cycle test message directly to sendPort2 by calling sendPort2.receiveCycleTestMessage. At this point receivePort1 and sendPort2 are not yet connected, so if a cycle test message makes its way back from sendPort2 to receivePort1 then there must already be an indirect connection between receivePort1 and sendPort2. If this is the case, directly connecting receivePort1 and sendPort2 will create a cycle in the graph.

As you can see, the message is relayed by module2 through all connected ports, ending up at module1's sendPort1. This port requests module1 to relay the message again which it does. In doing this it tries to get receivePort1 to send a cycle test message, causing an exception to be thrown because the flag *testingForCycle* is set. The method *testForCycle* catches the exception and causes the connect method to return false.

## Animation of Images

Animation is achieved in OpArtGen in two ways: via Oscillator modules in a rendering pipeline, or by directly manipulating module property values in a script.

### Animation Using Oscillator Modules

Oscillator modules output a value that constantly changes in accordance with whatever waveform is defined by its property values. If their output port is connected to a value input port on a module in a rendering pipeline then this will cause the value of that module's property to oscillate. So that images are updated constantly to reflect any change in the values output by Oscillator modules in a pipeline, the Refresh Cycle must be performed repeatedly. The *RenderingManager* class contains code to start and stop animation of all rendering pipelines via calls to the methods *startAnimation* and *stopAnimation*.

To try and maintain a constant frame rate, the animation loop keeps track of the estimated delay required between each frame based on the actual time that each loop iteration occurs. (17)

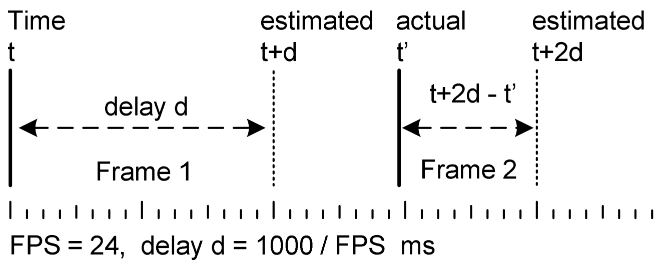


**Figure 26: Tracking Delay Required Between Frames**

Each time an animation loop iteration occurs the *RenderingManager* must first ensure that all Oscillator modules have updated their output value, and then initiate a Refresh Cycle for each rendering pipeline it is managing.

The first stage in a loop iteration is achieved using *AnimationEffector* objects. The *AnimationEffector* interface defines one method *nextFrame* which is called when an effector object should perform any frame dependent updating. Any object that implements the *AnimationEffector* interface may be registered with the *RenderingManager,* and each time a loop iteration occurs the *nextFrame* method of every *AnimationEffector* object registered with the manager is called. Oscillator Modules implement the *AnimationEffector* interface and automatically register themselves with the Rendering Manager.

All Canvas Modules also automatically register themselves with the Rendering Manager. For the second stage in a loop iteration the manager calls each Canvas Module's *updateRenderingPipeline* method to initiate a Refresh Cycle. Once this completes then the image for the iteration's frame gets displayed. As Refresh Cycles are asynchronous (the *updateRenderingPIpeline* method spawns a worker thread) then the animation loop is not held up. This means that during each iteration *nextFrame* gets called on time and the output of Oscillator modules remains current. Every time a *refresh* cycle commences it will be using up-to-date property values.

Further details on Oscillator modules and the waveforms that they output are given in the section: *Implementation of Core Modules*: *Oscillator Module* (Page 66).

### Animation Using Scripts

Animation may be performed manually in script files. This allows you to sequence timing of animation events and gives greater control over what image elements are changed: you are not limited to altering properties exposed by module ports. An example loop that causes a shape to expand and contract at different speeds is shown. The full script for this may be found in *Appendix 4: Manual Animation Example* (Page 104).

```
while (true) {
      for (int i = 1; i < 100; i++) {
            double size = 0.8 - 0.6 * i / 100;
            shape.setSize(size, size);
            try {
                  Thread.sleep(100);
            } catch (Exception e) {}
            canvas.updateRenderingPipeline();
      }
      for (int i = 100; i > 0; i--) {
            double size = 0.8 - 0.6 * i / 100;
            shape.setSize(size, size);
            try {
                  Thread.sleep(10);
            } catch (Exception e) {}
            canvas.updateRenderingPipeline();
      }
}
```

After the required modifications to property values are made for the next frame, a call is made to *updateRenderingPipeline* to render and display that frame's image. Here, the intra-frame delay is achieved using Thread.*sleep,* however it would be possible to create a sophisticated event model to trigger certain animation sequences at specific frame numbers.

## API Component Loading

External components are loaded at runtime from the subdirectories contained in the opArtGen/API/external directory. Five subdirectories exist: *loaders*, *modules*, *renderers*, *shapes*, and *warps*; each of these contains the external component source or class files for the relevant component type.

Components are loaded from each subdirectory by component loader objects. These objects are defined by classes that implement the *ComponentLoader<T>* interface. This defines a method *getComponents* which must return a Dictionary linking *Strings* with loaded *T* objects; and a method *getFactory* which returns a *ComponentFactory* object to allow easy creation of the loaded external components in scripts.

Two classes exist that implement the *ComponentLoader* interface: *AbstractClassComponentLoader* and *ShapeLoader*. *ShapeLoader* allows Java Shape objects to be loaded from SVG files contained within the shapes subdirectory, and extending the *AbstractClassComponentLoader* class provides an easy way to compile and load external component classes from source or class files contained in a package directory.

The class *ComponentLIbrary* provides a number of static methods that return the *ComponentLoader* objects for the five subdirectories: *getLoader, getModuleLoader, getRendererLoader, getShapeLoader, getWarpLoader.*

The automatic loading of OpArtGen components by these individual loaders at runtime is coordinated by the static method *loadAllComponents* in the *ComponentLibrary class*. This method is called as soon as the GUI is run and before a script gets executed. Any component source files are compiled, and all compiled class files for each component type are loaded into Java and may be accessed using reflection.

Factory objects are provided to spare script authors from having to write a lot of Java reflection code to access external components that have been loaded at runtime. Once a component loader has loaded its components they may be created via the factory object returned by that loader's *getFactory* method. A *ComponentFactory* object provides two methods for creating objects: *createComponentInstance* and *createComponentInstances.* The first method takes a component name as a parameter and returns a new instance of the named object, and the second returns a dictionary linking component names to instances for all the components the factory is able to create.

When creating new external components for an already defined component type it is sufficient to place the new components in the relevant external subdirectory for that component type, and allow the relevant component loader to automatically find and load your new external component. If you wish to create a new component type entirely, for example perhaps you need a way to allow users to define their own plugin classes for a new module that you have built; then you must create a new component loader for your new component type. The new component loader class must then be placed in the external *loaders* subdirectory.

## Loading Shapes

Shapes are loaded from SVG files by the *ShapeLoader* class. This uses an instance of the class *SVGToShapeTranscoder* to convert any shapes stored within those SVG files into Java *Shape* objects.

The *SVGToShapeTranscoder* class takes an URI to an SVG file and uses it to create a Batik *JSVGCanvas* object. A listener object is used to detect when the *JSVGCanavs* has finished building its internal GVT Tree representation of the SVG file. A GVT Tree is built using the following grammar:

```
<GraphicsNode> → <CompositeGraphicsNode>
<GraphicsNode> → ProxyGraphicsNode
<GraphicsNode> → RasterImageNode
<GraphicsNode> → ShapeNode
```

```
<CompositeGraphicsNode> → RootGraphicsNode
<CompositeGraphicsNode> → CanvasGraphicsNode
<CompositeGraphicsNode> → ImageNode
```

The *JSVGCanvas* object passes the listener object a *GVTTreeBuilderEvent* object when it finishes building the GVT Tree. From this event object we can call *getGVTRoot* retrieving the root node of the GVT Tree. The listener object calls the method *processGraphicsNode* in the transcoder object, passing it the *RootGraphicsNode* object. This method recursively processes the root object exposing each node in the tree: whenever it detects an instance of a *ShapeNode* object the *Shape* object for that node is retrieved. Finally, the set of shapes is returned to the loader object.

## GUI Package

The GUI package contains the classes that provide a visual editor for rendering pipes' structure diagrams and display rendered images in a window on screen. The GUI is layered over the top of the API, using the API objects and methods to compose and render images. The GUI itself is extendable via Module classes, as modules may define their own display panels for the GUI to use.

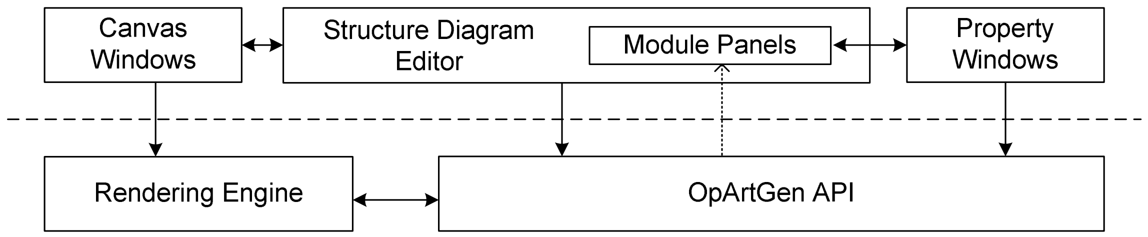


**Figure 27: GUI Architecture**

### Structure Diagrams

A structure diagram is a visual representation of the graphs of modules for a number of rendering pipelines. The structure diagram editor allows modules to be easily added to and removed from diagrams, and connections between modules to be made.

*StructureDiagram* objects are *JPanels* containers which hold module panels. A *DraggableManager* class manages the mouse listeners for any module panels that are added to the structure diagram when modules are created. A *ConnectionManager* class manages the mouse listeners for any port panels, sends connection requests to the API objects, and draws connection lines for any successful connections that are made.

#### *Creating New Modules in Structure Diagrams*

The GUI uses the Module loader object from the *ComponentLibrary* to retrieve a factory object for all built-in and external modules present at run-time. It builds the create module menus from the entries in the text file *ModuleNames.txt* in the modules package that the factory object is able to create. A mouse listener is added to the structure diagram so that the context menu displayed when right-clicking on the structure diagram allows you to add any available module.

When you click on an entry in the create module menu a *CreateModuleAction* object is fired. This uses the factory object to create a new instance of the correct module, retrieves the location point in the structure diagram that the user clicked on, and calls the Structure Diagram's *addModule* method passing it the module instance and point location.

The Structure Diagram calls the module's *getModulePanel* method to retrieve the correct JPanel object to display in the diagram. It then ensures the JPanel is correctly sized, and then adds it to itself at the requested location. The class *AbstractModule* provides a default implementation of the *getModulePanel* method that returns an instance of the class *DefaultModulePanel.* This provides a basic module panel that displays the module's name, input ports, and output ports. All implementations of core modules use this method to display themselves in the structure diagram.

### *State Information in Structure Diagrams*

The structure diagram itself doesn't maintain the state information for modules and ports present in the diagram: this is maintained by the API objects themselves. Their panel objects do store a reference to the actual Module or Port objects so that when you try to connect, disconnect, or delete them the correct methods can be called on the API object to update the rendering pipeline.

### *Connections  in Structure Diagrams*

When Port panels are retrieved from the Port API objects to build the default Module panels they are registered with the Structure Diagram's *ConnectionManager* object. This adds a mouse listener to the Port panels. When you wish to connect two module ports by pressing the mouse button on the first port, dragging the cursor until it is over the second port, and releasing the mouse button; the relevant mouse listener methods are fired.

The *mousePressed* method retrieves the Port panel object and notifies the manager that you have pressed the mouse button down over that port panel. It then begins to draw a red line between the port and the mouse cursor to provide a visual cue that you are attempting to make a connection. The *mouseDragged* method then checks with the manager that the connection is still able to continue (i.e. was the component the mouse button pressed over a port panel), and if so updates the red line to reflect the new position of the cursor.

Finally, when you release the button over a port the *mouseReleased* method is fired. This obtains a reference to the port panel object you released the mouse button over using *SwingUtilities* methods:

```
// get the actual component object the mouse was released over
Component target = SwingUtilities.getDeepestComponentAt(diagram,
pt.x, pt.y);

// try and find a parent port panel that contains the component
we just found
Component dest =
SwingUtilities.getAncestorOfClass(AbstractPortPanel.class,
target);
```

The listener then notifies the manager that the mouse button has been released, passing it a reference to the port panel object. The manager checks the port panel object and attempts to connect the two ports together using: firstPort.connect(dest). If the method returns true then the API has accepted the connection between the two ports, and the manager then adds another connector line to the structure diagram.

The manager also adds component listeners to module panels. The component listeners are fired when you drag the modules, and the listeners call the manager's method *repaintConnectionLines.* This updates the connection lines to reflect the new location of the port panels in the structure diagram.

### Canvas Windows

The GUI displays images rendered by pipelines from scripts or Structure Diagrams in Canvas Windows. These are displayed by the *CanvasWindowFrame* class that extends *JFrame.* The content pane of the canvas window contains one component: a *Painter* object which is able to display Canvas objects. Each Canvas Module has an associated Canvas Window. When a Canvas Module has created a *Canvas* object containing an overridden paint method able to draw its image it calls the *setCanvas* method on its Canvas Window's painter object and requests the Canvas Window to perform a *repaint*.

### Painter

The painter object extends *JPanel* and contains an overridden *paintComponent* method which sets up the correct rendering hints for the Java *Graphics* object it receives. It then calls the *paint* method of whatever canvas object it is currently storing. Get and set methods are provided to store and retrieve the *Canvas* object.

The default rendering hints that are used by OpArtGen to achieve high quality output are set in the *Config* class under the field DEFAULT_RENDERING_HINTS. These hints are:

| Rendering Hint | Value |
|---|---|
| **KEY_ALPHA_INTERPOLATION** | VALUE_ALPHA_INTERPOLATION_QUALITY |
| **KEY_ANTIALIASING** | VALUE_ANTIALIAS_ON |
| **KEY_RENDERING** | VALUE_RENDER_QUALITY |
| **KEY_STROKE_CONTROL** | VALUE_STROKE_NORMALIZE |

## Executing Scripts

Script loading and execution is supported by classes that implement the interface *ScriptType*. Three such classes are implemented: SourceScript, CompiledScript, and JavaScript. Respectively these support Java scripts in source files, in compiled class files, and JavaScript Rhino scripts. The *RunScript* class in the start package provides a main method that will load a file, attempt to create an instance of the correct *ScriptType* class based on the file extension, and if creation is successful then the script will be executed.

Ordinarily the Canvas Windows hide themselves when you try to close them rather than exiting the program. Because the Canvas Window is the only GUI component the user will see when using scripts, they need to be set to close the program when opened by a script. To achieve this a call is made to the static method CanvasWindowFrame.*setExitOnClose* by *RunScript.* If the script then creates any Canvas Modules, the associated *CanvasWindowFrame* objects will automatically set their behaviour to exit on close.

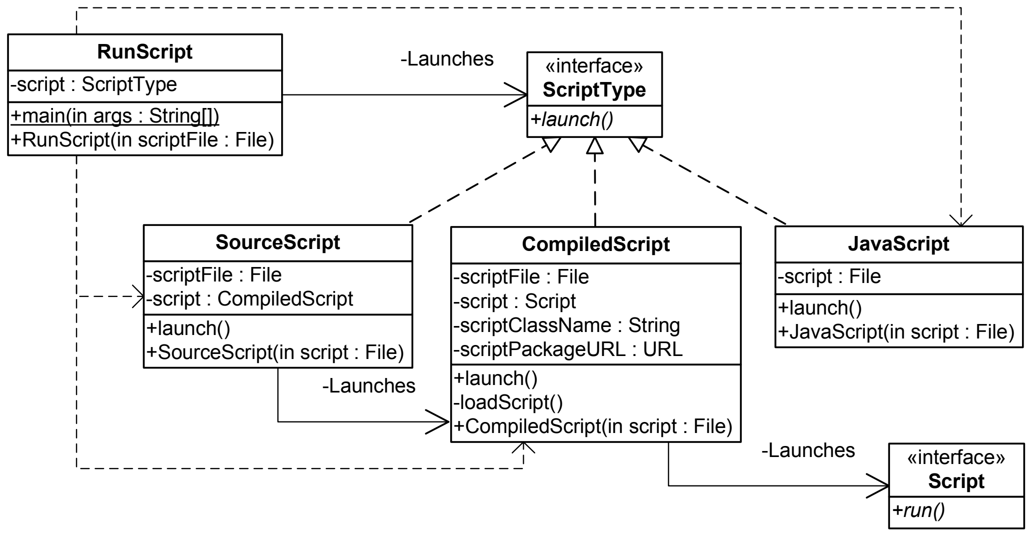


**Figure 28: Class Structure for Script Execution**

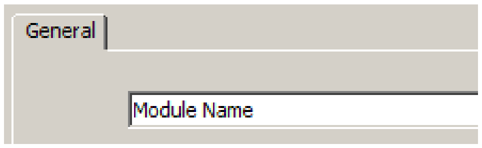
Each class extending *ScriptType* has a constructor that takes in a *File* object as a parameter and then attempts to load the given script. When the *launch* method is called, *SourceScript* uses the Java Compiler API to compile the source file supplied and then creates a *CompiledScript* instance using the class file produced by compilation. It then calls the launch method of the *CompiledScript* before removing any temporary files created.

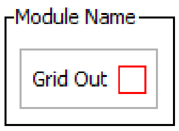
The constructor method for *CompiledScript* instances uses class loaders are used to load the script's class file and any other required class files. The class definition is then checked to ensure that it implements the *Script* interface, and if so a new instance of the script's class is created. When the *launch* method is called on a *CompiledScript* instance, the *ComponentLibrary* is used to load all components and the *Script* object has its *run* method called.

The constructor method for *JavaScript* instances simply stores the *File* reference. When the *launch* method is called, the Java Scripting API is used to interpret the contents of the file: a new script engine manager instance is created and the engine instance for ECMAScript is retrieved. The *ComponentLibrary* is used to load all components then the engine instance is used to evaluate the script.

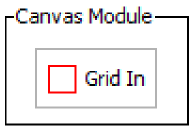
## Implementation of Core Modules

Reference tables of API objects and their methods for use when writing scripts are provided in *Appendix 5: API Reference for Scripting* (Page 106). In addition to any property values mentioned here, all modules also have an instance name which is displayed on the module's panel in the structure diagram. This may be changed using the text input field at the top of each module's General tab:





### Canvas Module



As we have seen, the Canvas Module is the root node of a rendering pipeline and allows you to stretch a grid over any sized canvas. It initiates a refresh cycle for its rendering pipeline and then performs the final stage of processing converting the grid data received to a Java *Canvas* object, which may be displayed on screen or exported to file.

The Canvas Module has four definable properties: canvas size, canvas margins, canvas background colour, and visibility of the canvas' window. The image defined in the grid data received by the Canvas Module is drawn to whatever inner size is left when the margins are subtracted from the canvas dimensions.

The properties window has three tabs allowing you to set these properties: General, Dimensions, and Background. The General tab allows you to show and hide the Canvas Window which displays the image rendered by the module; the Dimensions tab allows you to choose pixel sizes for the canvas height and width, and for the left, right, top and bottom margins; and the Background tab allows you to pick a colour that is filled over the entire canvas before anything else is drawn to it.

Each Canvas Module creates its own canvas window instance. To enable the user to resize the canvas window and automatically update the Canvas Module's canvas size property, the module adds a component listener to the window with an overridden *componentResized* method that automatically updates the module's size property

#### *Ports*

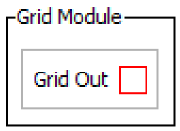
| Index | Input Port | Description |
|---|---|---|
| **0** | Grid In | Recieves grid data that should be drawn to screen or to file. |

#### Export to SVG

The Canvas Module allows you to save a rendered image to an SVG file. From the GUI this may be performed by selecting "Export to File..." from the Canvas Module's right-click context menu. The conversion of the image to SVG format is controlled by its method *exportToFile*. Drawing the image to screen is performed by the *Canvas* object's paint method, which receives a *Graphics* instance that can render image data to the screen using Java Graphics API calls. The Batik library is used to convert the calls made to the Graphics object into a SVG document tree.

The Batik library includes the class *SVGGraphics2D*. This provides an implementation of the Graphics2D abstract class which generates SVG documents instead of drawing to the screen. An instance of *SVGGraphics2D* is created to encapsulate an XML document, which is then passed to the *Canvas* object's *paint* method. Once this method has returned, the SVG tree contained in the *SVGGraphics2D's* XML document may then be written to file using an output stream.

## Grid Module

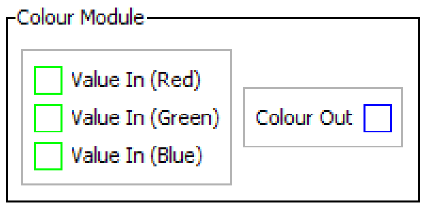


The Grid Module is a leaf node module that creates empty grid data objects for other modules in the rendering pipeline to process. The module has two properties: the number of rows and the number of columns the output grid should have. These are accessible under the "General" tab in the module's properties window.

### *Ports*

| Index | Output Port | Description |
|---|---|---|
| **0** | Grid Out | Outputs empty grid data with the specified number of rows and columns. |

## Colour Module



The Colour module is a leaf node or intermediate node module that outputs *ColourData* objects that encapsulate a specific Java *Color* value. The module provides you with one property value to store the *Color* value to use when creating the *ColourData* objects. This is accessible in the General tab in the module's properties window.

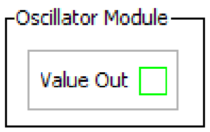
The module also exposes the colour property via three value input ports. When the module detects that a port is connected to another module, it uses the value received on that port to set the equivalent red green or blue part of the *ColourData* object it outputs. This allows you to connect up Oscillator Modules to allow animated fades in-between colours.

### *Ports*

| Index | Input Port | Description |
|---|---|---|
| **0** | Value In (Red) | Sets the red value of the colour output. Accepts values: 0 to 255 |
| **1** | Value In (Green) | Sets the green value of the colour output. Accepts values: 0 to 255 |
| **2** | Value In (Blue) | Sets the blue value of the colour output. Accepts values: 0 to 255 |

| Index | Output Port | Description |
|---|---|---|
| **0** | Colour Out | Outputs the colour value specified by the module's colour property, or by any modules attached to the Value In ports. |

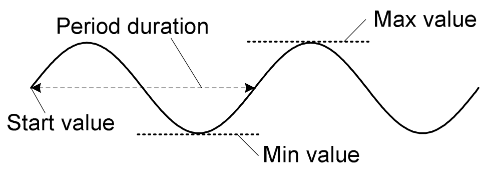
## Oscillator Module



The Oscillator Module is a leaf node module that outputs *ValueData* objects that encapsulate a single double value. Each time the module's *nextFrame* function is called by the animation loop in the *RenderingManager* object, the value output by the module changes following a sine wave pattern.

The module provides four properties that allow you to configure the sine wave produced. They are start value, min value, max value, and period duration, and all may be set in the Sine Wave Parameters box in the module's properties window. The module also keeps count of the current frame number.

The start value you specify allows you to configure the first value that the module outputs when animation is not running. The min and max values respectively set the highest and lowest values output by the module. The period duration sets the number of frames it takes the module to go from outputting the max value to the min value and back to the max value again.



The value that the module outputs on each frame whilst the animation loop is running is calculated as follows. For frame number f and period duration d:

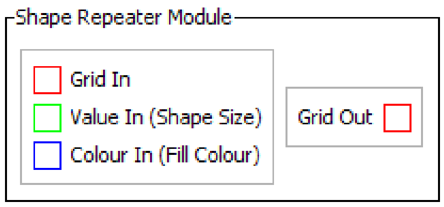
$$amplitude = \frac{\max value - \min value}{2}$$

$$output = \sin\left(2\pi\left(\frac{f\ mod\ d}{d}\right)\right) \times amplitude$$

### *Ports*

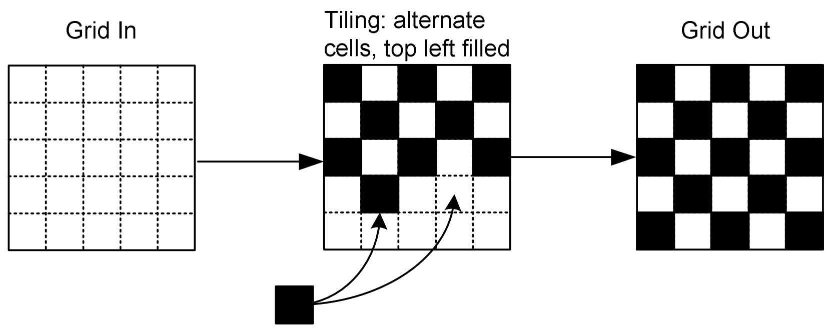
| Index | Output Port | Description |
|---|---|---|
| **0** | Value Out | If the animation loop is not running outputs the value defined by the start value property. Otherwise outputs the correct value for the current frame number. |

## Shape Repeater Module



The Shape Repeater Module allows you to place Java shape objects in the cells of the grid object received on its grid input port. The module has four options that allow you to control the placement of shapes in the grid: shape, size, tiling options and rendering options.

The shape property stores a Java *Shape* object ready to place into each cell. The size property allows you to specify the size of the shape placed into each cell as a proportion of each individual's cells size, once the grid has been stretched out over the final canvas) The tiling options property allows you to choose which cells in the grid the shape gets placed into; and the rendering options allows you to choose whether or not to draw the line and/or the fill, and which colour to use for each.



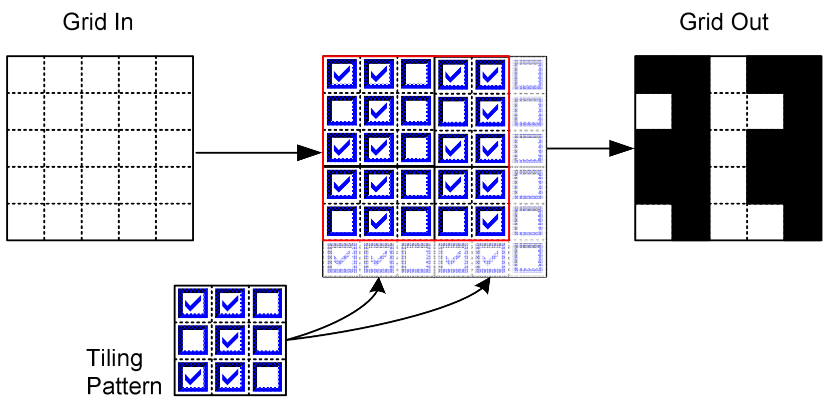
Shapes are placed in grid cells by creating a *ShapeCellRenderer* object for each cell a shape should be placed in and linking each cell to its own renderer. The shape field for each renderer is set to point to the shape object stored by the module. This allows later modules to change the way each shape gets rendered whilst saving on memory by using the same shape object.

## Tiling Options

The tiling options property value gets set in the Shape Placement tab of the module's properties window. The Tiling Options box allows you to choose from a number of predefined options, these are:

- Cover entire grid
- Alternate cells only (top-left cell filled)
- Alternate cells only (top-left cell empty)
- Random placement
- Custom tiling pattern

If the custom tiling pattern option is chosen then you may create a pattern of selected and unselected checkboxes in the Custom Tiling Pattern box. Each checkbox represents a cell, and those that are checked will have a shape placed in them. If the size of the checkbox pattern that you create is smaller than the size of the grid, the pattern is repeated over the grid in the following way:



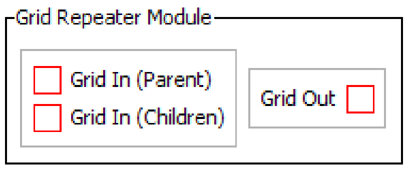
The size of the tiling pattern may be altered by clicking on the + and – buttons next to the rows and columns labels in the Custom Tiling Pattern box.

### *Ports*

| Index | Input Port | Description |
|---|---|---|
| **0** | Grid In | Sets the grid object that this module places shapes into |
| **1** | Value In (Shape Size) | Sets the size of each shape as a proportion of its parent cell's size. Accepts values in-between 0 and 1. |
| **2** | Colour In (Fill Colour) | Overrides the fill colour specified by the module's rendering options property. |

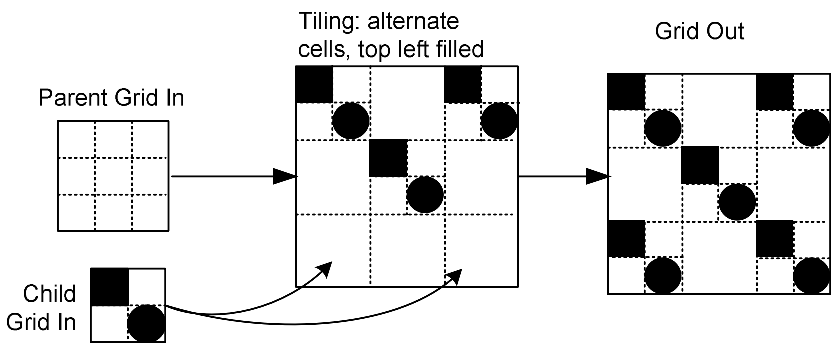
| Index | Output Port | Description |
|---|---|---|
| **0** | Grid Out | Outputs a clone of the input grid containing the correct placement of shapes in cells as specified by the module's properties. |

## Grid Repeater Module



The Grid Repeater Module behaves in a similar way to the Shape Repeater Module. It allows you to place copies of a child input grid into cells of a parent input grid. In this way you can build up large, complex patterns of shapes using small child grids that contain basic shapes. The module has one property for tiling options. This property behaves identically to the same property found in the Shape Repeater Module. Two grid input ports are provided: one for the parent grid, and one for the child grid.

The module places copies of the child grid in the parents cells using multiple *GridCellRenderer* objects the same way shapes were placed in cells by the Shape Repeater Module.

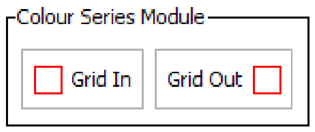


### *Ports*

| Index | Input Port | Description |
|---|---|---|
| **0** | Grid In (Parent) | Sets the grid object that this module places child grid copies into. |
| **1** | Grid In (Child) | Sets the grid object that this module makes copies of to place in the parent grid. |

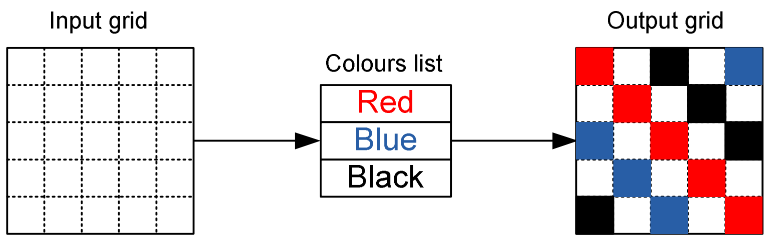
| Index | Output Port | Description |
|---|---|---|
| **0** | Grid Out | Outputs a clone of the input grid containing the correct placement of child grid copies as specified by the module's tiling options property. |

## Colour Series Module



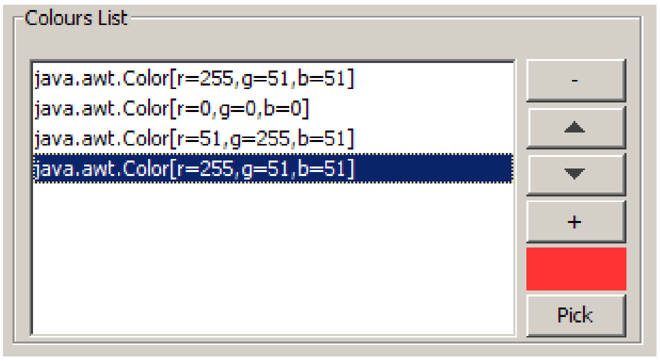
The Colour Series Module allows you to repeat a series of colours over cells in a grid, setting either the line colour or the fill colour of any shapes present in that grid. The module has four properties: colours list, random order, alter line colour, and alter fill colour. The colours list stores an ordered list of colours that are repeated over the input grid. The random order property specifies whether the module should use a randomly shuffled copy of the colours list each time it is repeated over the grid. The properties alter line colour and alter fill colour specify whether the module should alter the line colour and/or the fill colour of any shapes stored in the input grid's cells.

Colour values are stored in each cell's renderer regardless of which renderer type it is. This means empty cells will still take up one of the colour values in the series.



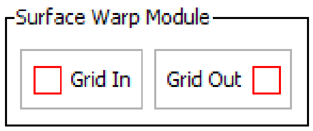
### *Colours List*

The Colours List box in the module's properties window allows you to pick and edit a list of colour values for the module to use. The + and – buttons add and remove colours from the list, and the arrow buttons change the order of the colours in the list. The Pick button allows you to select a different colour to add to the list using the + button.

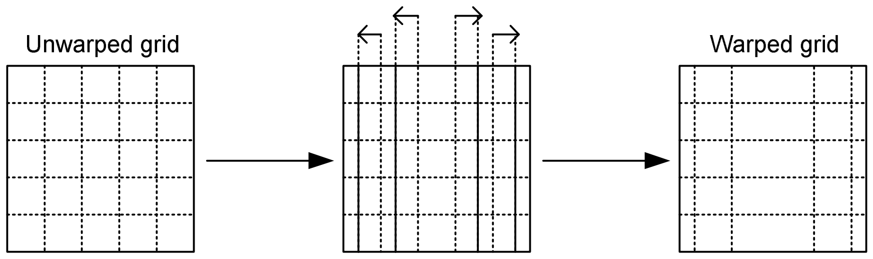


java.awt.Color[r=255,g=51,b=51]
java.awt.Color[r=0,g=0,b=0]
java.awt.Color[r=51,g=255,b=51]
java.awt.Color[r=255,g=51,b=51]

## Surface Warp Module



The Surface Warp module applies an algorithmic resizing to the dimensions of the grid data that it receives. Each different algorithmic resizing it can apply is called a warp; each warp may resize the widths and heights of the rows and columns of the grid. As elements contained in grid cells are sized proportional to the size of the parent cell, the elements also get stretched or compressed.



In order to ensure that the grid output from module retains its original total dimensions it is necessary that the various warp algorithms that may be applied only alter the grid cells' proportional widths and heights, and that the total proportional width and height of all the grid cells are both equal to 1 after a warp has been applied. Each warp has a parameter list allowing you to alter the value of parameters used by different warp algorithms.

The module has one property that stores the current surface warp object being used. The module's properties window allows you to configure the warp type to use, whether to warp the grid's X-axis by altering column widths, whether to warp the grid's Y-axis by altering row heights, and the value of any parameters that warp type has.

### *Ports*

| Index | Input Port | Description |
|---|---|---|
| **0** | Grid In | Sets the grid object that this module changes the row heights and column widths of. |

| Index | Output Port | Description |
|---|---|---|
| **0** | Grid Out | Outputs a clone of the input grid whose row heights and column widths have been altered as specified by the module's warp property. |

## Two Quarter Circ le Arcs Warp

The first warp built for the Surface Warp Module consists of a projection of an evenly spaced grid onto two quarter circle arcs, and is similar to "Surface Type 4" in Oliver Payne's program. The un-warped grid surface is stretched along the line made by the two quarter circle arcs and is then projected back onto the axis being warped. The projected grid cell widths or heights are then used to resize the grid. As each circle is centred at the two ends of the axis the warp is being applied to, and the original grid is evenly stretched over the two arcs, we ensure that the proportional grid sizes output by the warp total 1.

The warp has two parameters that may be altered: *xArcIntersection* and *yArcIntersection*. These alter the distance along each axis that the two arcs intersect for the X-axis and the Y-axis respectively.

The geometry used to calculate the warp applied to the x-axis of a grid is shown below:

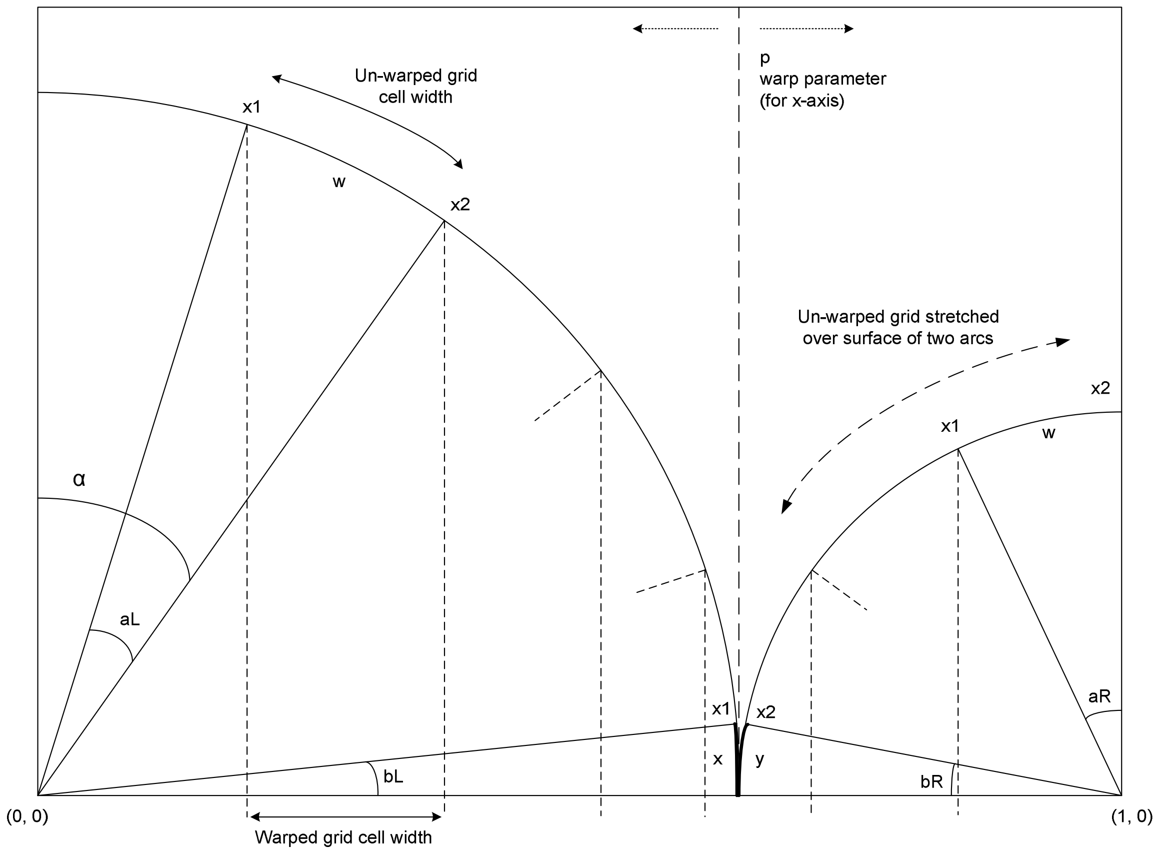


**Figure 29: Geometry of Two Quarter Circle Arcs Warp**

### *Calculating the warped grid cell widths*

As the grid is stretched evenly over the two arcs we know that the initial width w of each grid cell is equal to the total proportional grid width divided by the number of grid columns:

$$w = \frac{1}{cols}$$

The parametric equation of a circle allows us to easily calculate the x-coordinates of the edges of each grid cell if we know the central angles $a$ of the sectors whose arcs are equal to the grid cells. The ratio of the arc length of each circle to the right angle must be the same as the ratio of the width of a grid cell to the central angle of its sector. We can calculate these angles for the left and right circles as follows:

| | |
|---|---|
| Left circle's radius: | $r_L = pw$ |
| Right circle's radius: | $r_R = pw$ |
| Central angle for each grid cell: | $a_L = \frac{\pi w}{2r_L},\ a_R = \frac{\pi w}{2r_R}$ |

Once the central angles are calculated for each grid cell the x-coordinate of each grid line can also be calculated using the parametric equation for a circle and a value for the warped cell width obtained:

Warped cell width for left arc:

$$w' = x_2 - x_1$$
$$\alpha = angle\ ray\ makes\ with\ y\ axis$$
$$x_n = r_L.\sin\left(\frac{\pi}{2} + \alpha\right)$$
$$w' = r_L.\left\{\sin\left(\frac{\pi}{2} + \alpha_2\right) - \sin\left(\frac{\pi}{2} + \alpha_1\right)\right\}$$

The cell widths for the right arc are calculated in a similar way, except with $x_n = 1 - r_L.\sin\left(\frac{\pi}{2} + \alpha\right)$.

As is shown in the geometry diagram, the stretched grid will not always lie with a grid line located at the point of intersection of the two arcs. If this is the case a grid cell must be split across the two arcs, requiring additional calculations to be performed to obtain the x-coordinates of the split cell's grid lines. First, the length of the un-warped grid cell present on each of the two arcs must be found. We know the arc length of the left quarter circle, and how many full grid cells should be stretched along the left circle: $\lfloor p \times cols \rfloor$. The arc length of the left part of the split cell is obtained by subtracting the total length of the full grid cells from the arc length of the left quarter circle. A similar method is used to obtain the right arc length of the split cell.

| | |
|---|---|
| Split cell arc length on left circle: | $x = r_L - \left(\frac{1}{cols}.\lfloor r_L.cols \rfloor\right)$ |
| Split cell arc length on right circle: | $y = r_R - \left(\frac{1}{cols}.\lfloor r_R.cols \rfloor\right)$ |
| Central angle for left and right parts: | $b_L = \frac{\pi x}{2r_L},\ b_R = \frac{\pi y}{2r_R}$ |

It is then simple to obtain the x-coordinates of the grid lines using the parametric equation for a circle and calculate the warped cell width as before.

### Cubic Function Warp

The second warp resulted from the quantitative analysis of Bridget Riley's "Movement In Squares" as discussed in the section *Existing Works of Op-Art Recreated Using OpArtGen* (Page 83). The Cubic Function Warp resizes the grid according to a cubic function that relates the un-warped location of grid lines to their warped locations. So that the total proportional width of the grid remains unchanged the warp function used has the following properties:

- It is a continuous function
- Its gradient is always positive
- The result of applying the function to 0 and 1 is 0 and 1 respectively.

The function obtained from the anaylsis is

$$g \coloneqq x \to \frac{3.2206x^3 - 5.4091x^2 + 3.1979x}{1.0094}$$

So that the intensity of the effect may be varied, this function was rearranged and the coefficient in $x^3$ made into a parameter p:

$$warp \coloneqq x \to \frac{p.x^3 - 5.4091x^2 + 3.1979x}{p - 2.2112}$$

Ideally we would like the range of the parameter values we supply the warp function with to be between two sensible values, such as between 0 and 1. From experimentation, the range of suitable $p$ values that produce a warp function that satisfies our three desired properties was found to be: $2.8 \leq p \leq 5$.

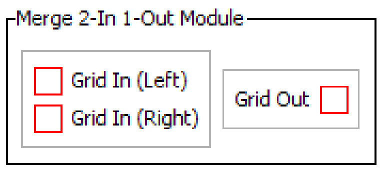
The warp parameter conversion function should linearly adjust the value of p, the cubic component multiplier in the function, with a parameter value $p' = 0$ corresponding to a multiplier value of 2.8 and a parameter value $p' = 1$ corresponding to a multiplier value of 5. The function that performs this conversion is easily derived from the equation for a straight line passing through the points (0, 2.2) and (1, 5):

$$convert \coloneqq p \to 2.2\,p + 2.8$$

Our converted warp function then is:

$$warp \coloneqq x \to \frac{convert(p').x^3 - 5.4091x^2 + 3.1979x}{convert(p') - 2.2112}$$

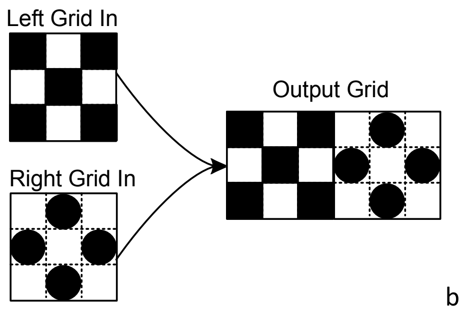
## Merge Module



The Merge Module allows you to combine two input grids into one output grid. This is useful for creating uneven grids, giving you greater flexibility when constructing the layout of your image. The module has one property that controls the merge type used. Three merge types are provided: Horizontal Merge, Vertical Merge, and Overlay Merge. Depending on which merge type is currently selected the input ports are renamed to indicate where they will be placed in the output grid.

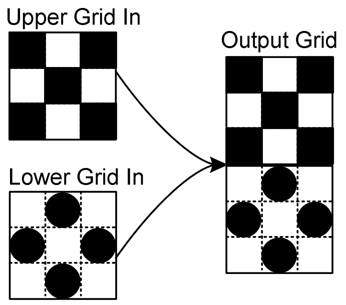
### *Horizontal Merge*

The Horizontal Merge type allows you to place each input grid into one cell of an output grid that has one row and two columns. An empty 2x1 grid is created that has each cell's renderer set to a *GridCellRenderer,* and *e*ach grid is stored in one of the renderers.



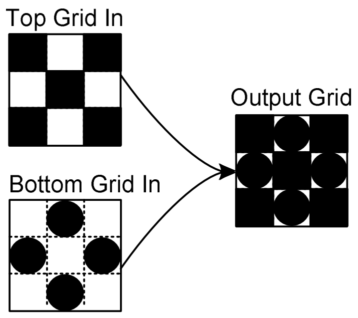
### *Vertical Merge*

The Vertical Merge is similar to the Horizontal Merge, except the output grid has two rows and one column.

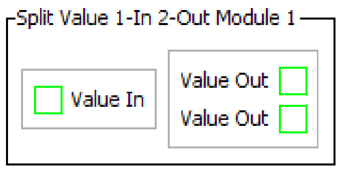


### *Overlay Merge*

The Overlay Merge makes use of the *setNextLayer* method in grid renderers to create an output grid with one cell that contains both input grids.



## Split Modules



Split modules allow you to make an additional copy of the data received by the input port. Three modules are provided to allow you to split grid data, colour data, and value data.

Splitting value data allows to you control parameters of different modules with the same Oscillator Module, so that different elements in an animated image change in sync with each other. Split modules may be chained to make more than one extra copy of the input data received.

## Creating external components for OpArtGen

### Custom Component Loading

When building external components it is necessary that you are aware of how your components will be loaded by the program at runtime. If you are creating an additional component for an already defined component type, for example a new module, you will be able to drop the class file in the correct directory and the program will automatically load it at runtime. If you have created a new component type that needs to load external components at runtime then you will need to create your own component loader.

For example: your might want your new module to load in preset objects that allow users to plug in their own processing functionality to your module.

**Steps to create a custom component class loader:**

1. Create a class that extends the class *AbstractClassComponentLoader<T>*. This class uses generics to ensure that internal collection types are checked, so make sure to parameterise the inheritance by the name of the interface that is common to your component type, e.g.:

```
public class MyCustomComponentLoader extends
AbstractClassComponentLoader<MyComponent> {
```

   Custom component loader classes must end with *Loader*. The loader that loads the custom component loaders searches for files ending in Loader.java or Loader.class.

2. Override the abstract methods that are inherited from *AbstractClassComponentLoader*. See *Component Loading Classes* in *Appendix 5: API Reference for Scripting* for method reference (Page 113).

3. For the method *getFactory* the easiest way to return the correct factory object is to create an instance of *ClassComponentFactory* if the components you are loading are Java classes:

```
public ComponentFactory<MyComponent> getFactory() {
return new ClassComponentFactory<MyComponent>(this);
}
```

4. Finally, ensure any new external component classes are loaded at runtime using your new custom component class loader by placing the loader's class file in the opArtGen/API/external/loaders directory.

Your component loader will not load any components until an instance of it is created. You can access your component loader and create an instance of it using the following code:

```
...
ComponentFactory<ComponentLoader> factory =
      ComponentLibrary.getLoaderFactory();
factory.createComponentInstance("MyCustomComponentLoader");
...
```

## Creating New External Modules

If the functionality present in the core built-in models is not sufficient to create images in a certain style, then a new module may be built that performs the required processing of data in the rendering pipeline.

Modules are Java classes that implement the *Module* interface. Since many of the methods defined in this interface are part of the intra-port messaging protocol that the rendering engine uses, and as such are common to all modules, this code is already implemented in the abstract class *AbstractModule*.

To simplify the task of writing a new module a template module code file has been provided that contains all the necessary abstract method implementation stubs and some boilerplate code to help you understand how to add ports, call-backs etc. to your module. This may be found in the report in: *Appendix 1: Module Template Code* (Page 93).

1. Create a new class for your module and paste in the code contained in **ModuleTemplate.java** as a template.

2. Change the class name to name of desired new module ensuring that the name of the constructor is also changed. Try to choose a helpful name; for example use TextPrinterModule rather than StringOut.

3. Edit the stub for the method *getModuleName* to return the string that is to be displayed in the structure diagram for the module. Note that this is the name returned by calls to *getName* before *setName* is called, and is not the name used in the GUI menus.

4. To define the name and position used for the module in GUI menus add a new line to the file **ModuleNames.txt**. If you wish to add a separator to the menus place a single”-“ character on a line. Blank lines in this file are ignored, and comment lines start with a # character.

5. Add any fields needed to store data for your desired module properties.

6. Create any get or set methods needed to provide API access to the new module’s properties. If the property controlled by a set method has an effect on the data output by the module then the set method should also include a call to the method *setDirtyFlag.* This ensures that the rendering engine knows that data from this module has changed and can refresh all parent modules in the rendering pipeline’s tree.

7. Create any additional constructors to preset module property values to help when writing scripts. Note that all constructors should call the *super* constructor first, as this contains code to ensure the module is initialised in the rendering pipeline correctly, and *initModule* last after property values have been set.

8. Put any additional module initialization code in the *initModule* method.

9. Change type of module's data fields to the desired data type that implements the *ModuleData* interface. A data field of the correct type is needed for each port: e.g. a

GridData field is needed for a GridPort. When the rendering engine is pushing data through the pipeline the ports read and write data from/to these fields.

10. Add code to create the correct ports in the *initPorts* method. Each port requires a callback object so that it may access private module data fields. The port's internal state (send or receive) is determined by the read/write permissions the callback object used to create it has.

    It helps to create private helper methods that create the relevant callback objects using anonymous inner classes, eg:

```
protected void initPorts() {
      inputPorts = new LinkedList<Port>();
      inputPorts.add(new GridPort(getInCallBack()));
      outputPorts = new LinkedList<Port>();
      outputPorts.add(new GridPort(getOutCallBack()));
}

private ReadOnlyCallBack getOutCallBack() {
      return new ReadOnlyCallBack(this) {
            public ModuleData loadData() {
                  if (dataObject == null) {
                        return null;
                  }
                  return dataObject.clone();
            }
      };
}

private WriteOnlyCallBack getInCallBack() {
      return new WriteOnlyCallBack(this) {
            public void storeData(ModuleData data) {
                  if (data instanceof GridData) {
                        dataObject = (GridData) data;
                  } else if (data == null) {
                        dataObject = null;
                  } else {
                        throw new IllegalStateException();
                  }
            }
      };
}
```

11. Edit the stub for the method *processModuleData*:

    This method is the main processing routine for your module. It is called automatically by the rendering engine when needed to process data from input ports. The code must perform any processing on the data received automatically by the ports in the *dataObject* field. Once processing is complete the resulting data is then cloned and sent down the rendering pipeline automatically by the output port.

12. Edit the method stub for *getPropertiesPanel:*

    If you want your module to have a properties window when used in the structure diagram editor, the stub for the method getPropertiesPanel must be completed. It should return a *JPanel* object containing all the necessary user interface components to receive user input and store the values in the module's properties fields using the get and set methods. A quick way to do this is to extend the basic panel class *PropertiesWindowPanel.* Use the *addTab* method in the constructor to add any desired pages to the dialog.

13. If your module processes GridData objects then it may have a visualisation option that enables users to preview the effect of your module in the rendering pipeline. This is achieved by the following code

```
public class MyModule implements Visualisable {
...
     public void visualise(ModuleVisualiser visualiser) {
          refresh();
          visualiser.visualiseData(dataObject);
}

}
```

    If your module does not process GridData objects then remove the *Visualisable* declaration and the *visualise* method.

## Creating a Custom Module Properties Window

When the user clicks on the module's context menu to open the properties window, a new window frame is created that contains the *JPanel* returned by *getPropertiesPanel.* The easiest way to create the *JPanel* object is to create a new class that extends *PropertiesWindowPanel* using the template code provided in *Appendix 2: Properties Window Template Code* (Page 97).

To simplify this task a template properties panel file has been provided that contains all the necessary abstract method implementation stubs and some boilerplate code to create a panel containing OK, Cancel, and Apply buttons; and a tabbed properties pane. This may be found in the report in: *Appendix 2: Properties Window Template Code* (Page 97).

1. Create a new class for your properties panel and paste in the code contained in **ModulePropertiesWindowPanelTemplate.java** as a template.

2. Change the class name to name ensuring that the name of the constructor is also changed. Try to choose a helpful name; for example use TextPrinterModulePropertiesWindowPanel.

3. Add fields to the class to store each property object retrieved from the module, and the relevant GUI object that edits each property value.

4. Edit the constructor so that the current property value for each property is retrieved from the module and stored in the relevant field each time the user opens a new properties window.

5. Use builder methods called from the constructor to create *JPanel* objects that allow the user to edit groups of properties. Add each *JPanel* object returned as a new tab in the properties window using a call to *addTab(*“Tab Name”, panelObject*)*

6. Alter the method stub provided for *getPreferredSize* so that it returns a new *Dimension* object of the desired size for your properties window.

7. Alter the method stub provided for *setModuleProperties*. This is called whenever the user clicks the OK or Apply buttons, and should check the current value in each GUI object for each property and compare it to the value stored in the field by the constructor. If it has changed, then the correct property setter method for your module should be called to update the property value stored in your module.

To add a the new properties window to your external module, replace the stub *getPropertiesPanel* method in the module’s class file with the following code:

```
public JPanel getPropertiesPanel() {
   return new YourModulePropertiesPanelClass(this);
}
```

## Creating External Port Types

If your module needs to send and receive a new data type, a new port type must be created to enable this. You must create a new inner class inside the class for your external module that extends the abstract class *Port.* The constructor of your new class should have a single parameter accepting a call back object. It should then call the *super* constructor, passing it the call back object and then set the *name* field as follows:

```
public NewPortType(CallBack callback) {
      super(callback);
      name = "New Port " + state.toString();
}
```

Appending the value of *state* to the ports name enables users to differentiate between input and output ports in the GUI. Additionally, by overriding the method *getColor* you allow users to distinguish your port by its colour in the GUI:

```
@Override
public Color getColor() {
      return Color.YELLOW;
}
```

### *Creating ModuleData classes*

A new module data class must be created that implements the interface *ModuleData* to encapsulate the data type you wish your new port to send. The interface requires you to implement a *clone* method so that the rendering engine can clone the data objects. This class should be placed in the package opArtGen.API.external.data, and the file should be placed in the external folder so that it gets loaded at run-time.

## Creating External Renderers

If you wish your module to be able to place data other than shapes or grids in grid cells then you must create a new cell renderer object that is able to store and draw the data. This can easily be achieved by extending the class *AbstractCellRenderer*. You must implement four methods: *getRenderingOptions*, *setRenderingOptions*(ShapeRenderingOptions), *paint(*Graphics*),* and *clone,* in addition to any getter and setter methods needed for the data.

The get and set methods for the rendering options should store the fill and line colour used by your renderer object. These may be retrieved from the rendering options object fields:

| Type | Field | Description |
|---|---|---|
| **boolean** | drawLine | Sets whether or not your renderer should draw lines for its object |
| **boolean** | drawFill | Sets whether or not your renderer should draw a fill for its object |
| **Color** | lineColor | Color to use when drawing lines |
| **Stroke** | lineStroke | Stroke to use when drawing lines |
| **Paint** | fillPaint | Paint to use when drawing fills (Note: this is set to a Color value) |

The paint method must draw the data object to the *Graphics* device, scaled correctly for the size stored in the renderer:

```
public void paint(Graphics g) {
      Graphics2D g2d = (Graphics2D) g.create();
      Rectangle cellBounds = getBounds();

      // Scale object according to cell contents scaling factor
      DimensionDouble contentsSize = getContentsSize();
      double sx = contentsSize.width;
      double sy = contentsSize.height;

      // Translate shape to ensure it stays centered
      double tx = 0.5 * ((1 – sx) * cellBounds.width);
      double ty = 0.5 * ((1 – sy) * cellBounds.height);

      // Perform transformations
      g2d.translate(tx, ty);
      g2d.scale(sx, sy);

      // Draw object to graphics device
      g2d.draw( ... )
      ...

      // Restore original graphics context
      g2d.dispose();
}
```

## Adding Extra Shapes

Shapes that are used in OpArtGen are stored in the external/shapes folder as SVG files. Use a visual SVG editor or a text editor to create new ones. Any shape elements defined in SVG files are loaded up as individual shapes by the shape loader at run-time using the following naming convention:

name of shape = <ShapeFileName.svg>_<ID>

The value of <ID> starts at 1 and is incremented each time a new shape definition is found in the SVG file. The new shapes may be accessed in scripts using the following code:

```
shapeFactory = ComponentLibrary.getShapeFactory();
rectangle =
  shapeFactory.createComponentInstance("ShapeFileName.svg_1");
                              //  or "ShapeFileName.svg_2" etc.
```

## Creating new Surface Warps

To create a new surface warp type you must create a new class in the package opArtGen.API.external.warps that extends the abstract class *SurfaceWarp.*

1. You must add any required parameters in the constructor:

```
addParameter(new WarpParameter("Name", "Description"));
```

2. Override the *clone* method.

3. Override the *getWarpName* method to return a string containing the name of your surface warp.

4. Override the *applyWarp*(int rows, int cols) method. This must return a *GridSpacing* object specifying the new grid line positions as a proportion of the total grid height or width, for the given grid size. A grid spacing object may be constructed from an array containing the warped grid column widths and an array containing the warped row heights. As an example, the following code returns a grid spacing object that sets all the grid line positions to be equally distributed:

```
@Override
public GridSpacing applyWarp(int rows, int cols) {
      double[] widths = new double[cols];
      double[] heights = new double[rows];
      Arrays.fill(widths, 1.0 / cols);
      Arrays.fill(heights, 1.0 / rows);
      return new GridSpacing(widths, heights);
}
```

## Evaluation of OpArtGen

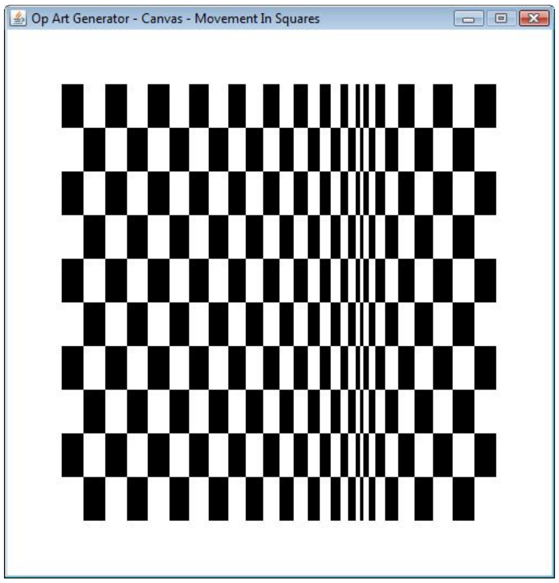
Most users initially found it hard to understand how to use OpArtGen unless the intuition behind the program was explained. Once explained, users found it easy to create artwork using the program. As such, the status bar tool tips are not sufficient to help new users. A new-user walkthrough feature or set of tutorials needs to be built into the program to aid new users in becoming accustomed to the workflow needed to create art in OpArtGen.

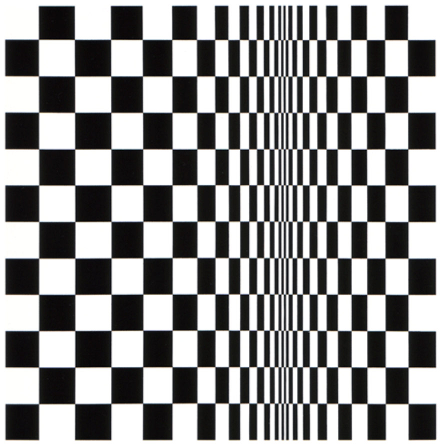
## Existing Works of Op-Art Recreated Using OpArtGen

We now recreate existing works of Op-Art using the program and compare the recreated images to the originals. Script files for these images may be found in Appendix 3: Scripts for Recreations of Existing Op-Art (Page 99).

### *Movement In Squares (Bridget Riley)*

The first test rendering of *Movement In Squares* using a script file linking a grid module to a shape module, surface warp module, and then the canvas module, resulted in the image on the left below. The surface warp type used for this rendering was the *Two Quater Circle Arcs* warp.

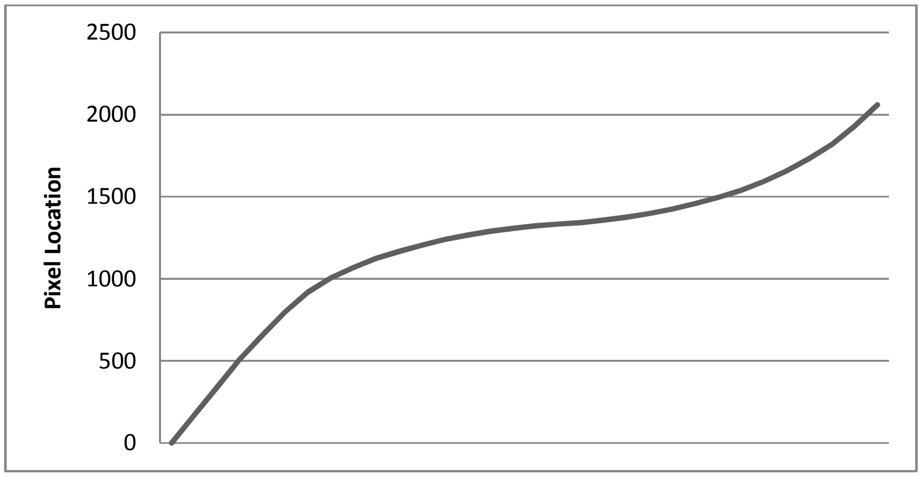




Stylistically the image does resemble Bridget Riley's work (shown on the right), with eye movement around the canvas generating the visual disturbances that are common to Op-Art. The motion effect is not as strong as the one generated by Riley's work, and looking at the rendering on more of a pixel-by-pixel basis it seems that this could be due to an incorrect surface warp being applied.

The squares are rectangular at the left of the canvas as the warp has not "squashed up" the grid enough in the middle, although the rough pattern of the warp seems accurate. In both our reconstructed image and the original work the grid spacing from left to right initially stays constant, and then gradually decreases with an increasing rate of decrease, and then reverses back up to its original spacing. This pinch still looks stronger in the original image so a different surface warp needs to be used to achieve this.

To construct the new warp the actual grid spacing in the original image was extracted from a scan made of a print of the work. The grid line positions from the scanned image were recorded and graphed in Excel:



From this graph it appears that the warped grid line positions should vary across the x-axis by a function similar to a cubic in x.

To try and fit a curve to the data I converted the pixel valued grid line locations of the scanned image to proportional locations with the leftmost line having a coordinate of 0 and the rightmost a coordinate of 1. I also converted the grid column index number values to their proportional location coordinates on the x-axis of the un-warped evenly spaced square grid, assuming the image has a 31x10 grid.

These values were then transferred from Excel to Maple, and the *LeastSquares* function in the *CurveFitting* package was used to obtain a cubic curve: $ax^3 + bx^2 + cx + d$ to fit the data.

The equation of the resulting curve was:

$$f := x \to 3.2206x^3 - 5.4091x^2 + 3.1979x - 0.0076493$$

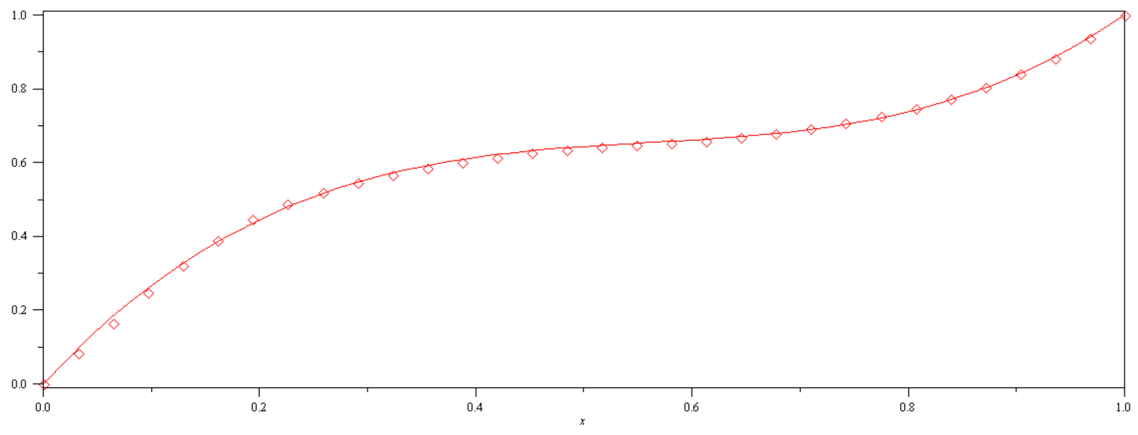
This was unsuitable to use directly as a function for a surface warp due to these results:

$$f(0) = -7.6493 \times 10^{-3}$$
$$f(1) = 1.0017$$

This would result in the warp altering the total grid dimensions as the total proportional width of the grid is changed from 1 to 1.0017, so the function must be normalized by adding 0.0076493 so $f(0) = 0$, and then divided by the new resulting value when $x = 1$ so that $f(1) = 1$.

$$g := x \to \frac{3.2206x^3 - 5.4091x^2 + 3.1979x}{1.0094}$$

When we plot this curve along with the points representing the original grid line positions we get the following:



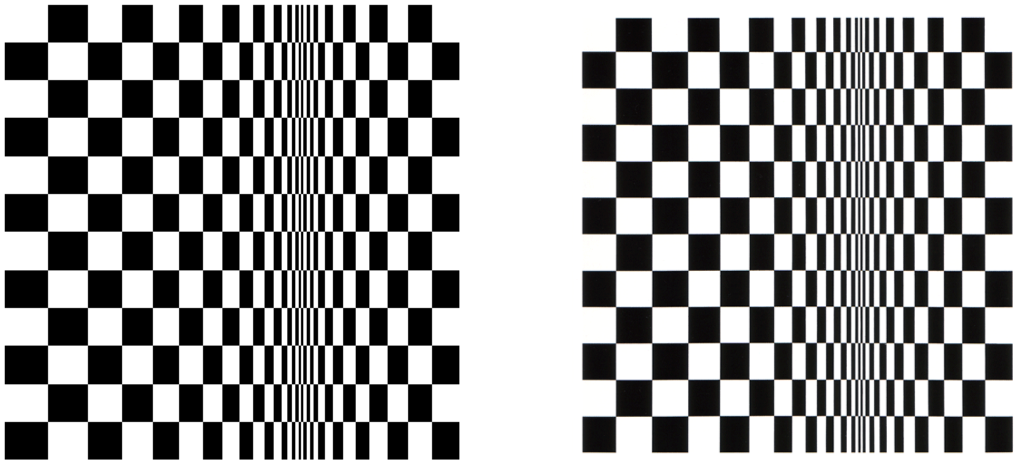
Notice that our curve approximates the grid line locations for the *Movement In Squares* image effectively.

As a final check that this function is a suitable approximation we must ensure that the gradient remains positive. A negative gradient implies that grid lines with increasing index are not always guaranteed to have warped coordinates that increase in value; this is not the case with Movement In Squares.

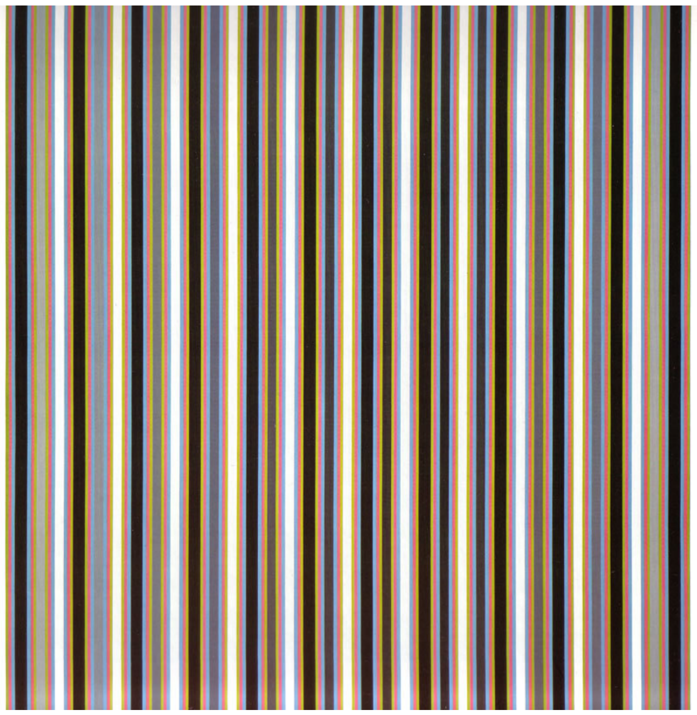
We find that

$$\frac{d}{dx}(3.1907x^3 - 5.3589x^2 + 3.1682x) = 9.5720x^2 - 10.7178x + 3.1682$$

With the Surface Warp Module changed to use the *Cubic Function* warp type in the script the rendered image now looks far closer to the original (original image on right):



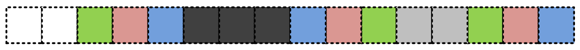
***Cantus Firmus (Bridget Riley)***



I scanned in a print of this work and then used an image editing program to extract the RGB colour values for each colour used in the image. To ensure that each bar contained a consistent colour value and scanning artifacts were removed I applied a vertical motion blur to the image. The colour values found were:

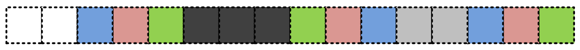
| Colour | RGB Value |
|---|---|
| **Light Green** | 189, 188, 41 |
| **Pink** | 221, 111, 113 |
| **Light Blue** | 112, 168, 197 |
| **Dark Grey** | 34, 26, 23 |
| **Medium Grey** | 101, 100, 100 |
| **Light Grey** | 167, 168, 163 |
| **White** | 248, 251, 249 |

The first set of colours moving right from the leftmost side of the image is as follows:

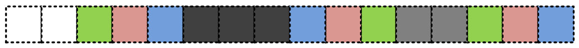


The thick dark grey strip is roughly three times the width of the thin coloured strips, and the light grey and white strips are equal and roughly twice the width of the thin coloured strips. This can be achieved in OpArtGen using an equal grid, and setting the colour series so that dark grey is repeated three times in a row, white twice and light grey twice.

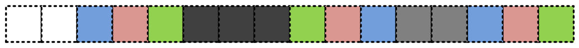
The next set of colours has the order of the thin colour strips reversed:



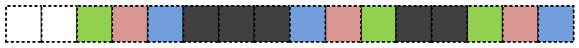
We now have a third set, but this time it is just the first set repeated. The fourth set is the second set repeated. Now the fifth set is the same as the first set but with the light grey replaced by the medium grey:



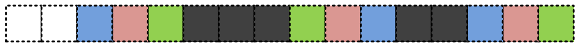
The sixth set is the same as the second set but also with its light grey replaced by the medium grey:



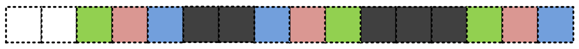
The seventh set is the same as the first set, but its light grey is this time replaced by the dark grey:



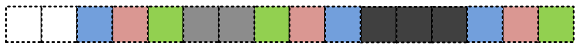
The eighth set is the same as the second set, but its light grey is this time replaced by the dark grey:



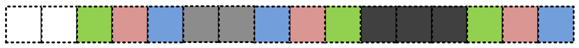
The ninth set has a changed pattern, and the thick dark grey bar from the eighth set is swapped with the medium sized dark grey bar:



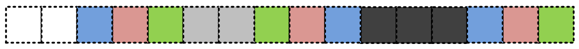
The tenth set has the same pattern as the ninth set, except the order of the thin colour strips is reversed and the leftmost dark grey bar replaced by a medium grey bar:



The eleventh set is the same as the ninth set but with the leftmost dark grey bar replaced by a medium grey bar:



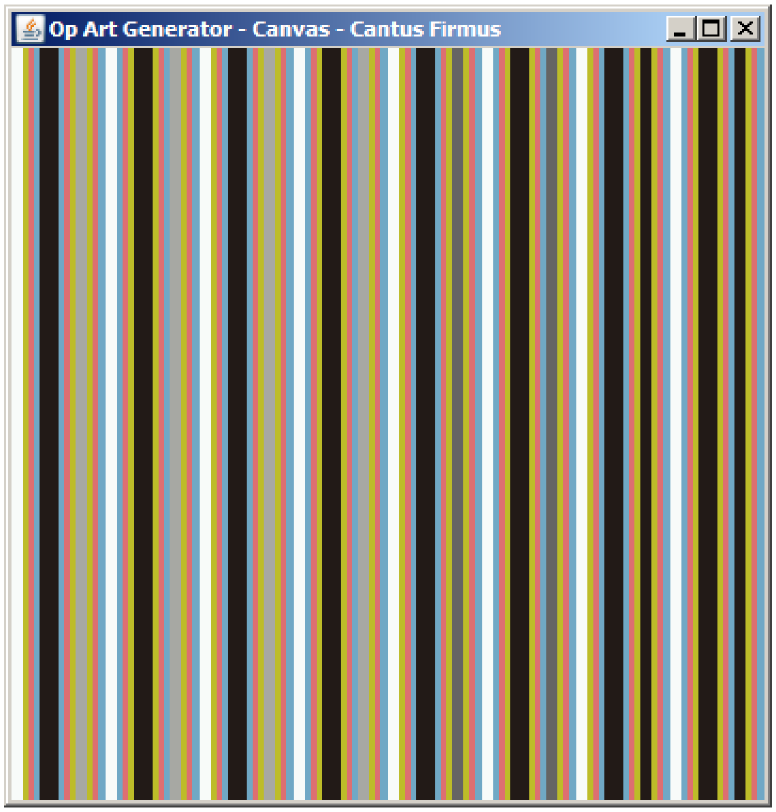
The twelfth set is the same as the tenth set but with the leftmost dark grey bar replaced by a light grey bar:



Each colour set is produced in OpArtGen using a Colour Series Module linked to each colour set. If all the sets are defined in a script using a list, then the rendering pipeline can be built algorithmically inside a for loop.

It would be useful to modify the merge module so that you can programmatically select how many input ports are merged into the one output port. This would have saved having to write a separate function in the script to recursively merge together a number of grid outputs.

The final image generated by the script which may be found in the appendix is shown below. A similar effect was achieved however the colours seem slightly off, perhaps due to slightly incorrect colour readings taken from the scanned image.



## Conclusions and Future Work

This project has shown that an extensible modular rendering engine provides a viable platform for creating Op-Art and other geometric art. The modular approach is able to recreate existing works of Op-Art as effectively as previous solutions have been able to. For any artwork styles that the program cannot reproduce, the extensibility of the rendering engine allows users to later add in the missing functionality themselves.

Combining the rendering engine with a visual graph based interface allows simple and intuitive use of the software by users not familiar with programming. Those comfortable with programming may easily extend the program in any way they see fit. I now suggest a number of further extensions to the base concept developed during this project that stem from the engine's extensibility and scripting capabilities.

### Real-Time Audio Input and Analysis

The Advanced Visualisation Studio (AVS) discussed in the Background section of this report uses a similar modular rendering system to OpArtGen. The purpose of AVS is not to create still art but moving visuals that respond to music. OpArtGen could be extended to provide a real-time audio input and analysis feature that allowed the artwork generated to change in response to the audio signal being received.

Given the modular nature of OpArtGen it would be possible to create another module similar to the Oscillator module that could hook into the sound in soundcard device using the Java Sound API. Any sound data captured by the Java Sound API would be analyzed and converted into values that could be output by the module. The outputs of the new module would then respond to events occurring in the sound wave being received by the module: for example a *ValuePort* could output either 0 or 1 depending on whether a beat had been detected in the sound; or a value in a range corresponding to the current VU level of the sound.

A feature similar to this could open up several commercial opportunities for using this software:

- Proprietors of bars and pubs could use the software to generate moving visuals for display on flat screen TVs at their premises to create ambience.

- Musicians could use the software to create an engaging backdrop at concerts that would be able to dynamically respond to their performance, linking the visuals to the sound.

## OpArtGen as a Media Player Plugin

Rather than using the Java Sound API to obtain sound data for analysis, the OpArtGen program could be wrapped up as a plugin for a media player that supports third party plugins, such as Winamp. This would allow users to create artwork that responds to their favourite music. Doing so would increase user base to a wider demographic than artists interested in creating Op-Art.

## Combining Sound Synthesis with Op-Art

As noted in the background section, a visual, graph based, modular structure has been employed as a successful design for computer based sound synthesis programs. OpArtGen could be extended using the Java Sound API to provide a set of modules and data types that allow the user to create sounds at the same time as images.

These new sound modules would work alongside the existing modules and data types already used for creating images. Modules that output data types common to image and sound modules could be connected to both. For example, an Oscillator Module that gradually altered the hue of elements in the image over time could also be used to control the timbre of a sound wave in parallel.

The rendering engine would require some changes in order to guarantee a sufficient frame rate for sound data. Currently the rendering engine only updates a pipeline when a *setDirtyFlag* message has been received by the root Canvas Module and a call to *updateCanvases* is made. Even with animation running there is no guarantee of a minimum update rate: the refresh is performed by a swing worker thread so under high processor usage requests for modules to *refresh* may stack up.

Allowing users to create both sonic and visual art from the same set of parameters would create immersive works that could engage the senses of viewers/listeners in a powerful way. I have not been able to find any software commercially available or otherwise that allows the combined generation of art in this manner.

## Real-Time Video Input and Analysis

It would also be possible to extend OpArtGen to respond to images captured by a video camera. A new video input module could be created that used algorithms to output value data based on images sampled in real-time from the video. A computer vision library such as Intel's OpenCV (18) could be used to aid in processing the captured image data. This could lead to interesting new ways for people to interact with Op-Art works created by users.

For example, when displaying art created by the program at an exhibition, images could be projected onto screens instead of having the artwork printed onto canvas and mounted. Video cameras could be set up to capture real-time video of the people viewing the artwork. A video input module could use a computer vision library to extract data based on facial features of people viewing the artwork, and supply the data as parameters to the other modules used to create the images.

## Generative Art: Using Artificial Intelligence Algorithms in Scripts

Artificial Intelligence techniques such as Genetic Algorithms could be used in scripts to evolve module parameters and rendering pipelines that result in entirely computer generated visually appealing art. Genetic algorithms are algorithms modelled on evolution theory that deliver solutions to difficult optimization or search problems. They are useful for situations where the fitness function for candidate solutions is complex or cannot be easily defined mathematically.

The problem of generating visually appealing art algorithmically may be modelled as an optimization problem. For artwork rendered using OpArtGen the problem is optimizing both the nodes and edges used in the graph of the rendering pipeline; and the set of parameter values used to set the properties of the module in each node of the pipeline.

Each candidate solution to this problem may be modelled by a string containing a binary encoding of the rendering pipeline and parameter values. A script could then be used to generate a population of individuals (candidate solutions) and evolve new generations of better solutions. Here, the fitness function would give an aesthetic rating for the image generated by each candidate solution. This evaluation of individuals would best be performed by an expert human: the image produced by each individual would need to be evaluated and assigned a fitness value.

## Extending the API to Provide Better Scripting Support for Animation

It is possible to animate images produced by OpArtGen in scripts either using Oscillator modules or by creating your own animation loop which alters property values and renders the image each iteration. It would be useful to extend the support provided in the API for animation by adding some kind of event sequencing functionality to the *RenderingManager*. This would allow you to add frame events that would fire once specific frame numbers were reached by the manager. Each frame event would perform a once off or periodically recurring set of tasks such as modifying module properties or even modifying the rendering pipeline itself.

When used in combination with an updated Oscillator Module that allows users to choose from a variety of waveforms, or construct their own, complex animation sequences could be built up.

The frame event model could be extended further to allow animation sequences to be constructed without Oscillator Modules. By allowing you to specify events that directly alter the property values of a module over a time period the script language could be extended to cover animation without having to manually create many nested loops. Fluent interfaces could allow a very elegant, natural language to be used:

```
// Decrease value of property from current value to 1, starting
// at frame no. 450, and ending at frame number 500
(new event()).atFrame(450).decrease(property).to(1).overNext(50)
```

## Community Site for Users

To increase the appeal of the program for its users it is necessary for the users to be able to easily adapt it without having to code additional modules themselves. This requires a large body of custom modules, scripts and components that the users can choose from depending on their requirements. The best way to ensure that such a body is developed and that it also reflects the requirements of most users is to create an online community site for OpArtGen.

A community site could host user developed contributions such as scripts, custom built modules and effects; and also user created walkthroughs and tips. A similar community was developed for AVS at www.winamp.com by Nullsoft (the creators of Winamp and AVS). This community alone has generated 3592 plugins for Winamp, of which 2457 are preset collections for AVS. (10) That provides for a level of customization that would have been very difficult to achieve by Nullsoft alone without the contribution made by community users.

## Bibliography

1. **CSound.** [Online] http://csound.sourceforge.net/.

2. **Native Instruments.** Reaktor. [Online] http://www.native-instruments.com/index.php?id=reaktor5_us.

3. **Borgzinner, Jon.** Op Art. *Time.* 23 October 1964.

4. **Esaak, Shelley.** Op Art- Art History 101 Basics. *About.com.* [Online] http://arthistory.about.com/cs/arthistory10one/a/op_art.htm.

5. **Moorhouse, Paul.** *Bridget Riley.* s.l. : Tate, 2003. p. 14.

6. **Zanker, Johannes and Walker, Robin.** A new look at Op art: towards a simple explanation of illusory motion. *Naturwissenschaften.* 2004, Vol. 91, pp. 149-156.

7. **Wikipedia.** Figure-ground (perception). [Online] http://en.wikipedia.org/wiki/Figure-ground_%28perception%29.

8. **Riley, Bridget and Kudielka, Robert.** *The Eye's Mind: Bridget Riley - Collected Writings, 1965-99.* s.l. : Thames & Hudson Ltd, 1999. pp. 66-67.

9. **Wikipedia.** Artists known for their op art. [Online] http://en.wikipedia.org/wiki/Op_art#Artists_known_for_their_op_art.

10. **Nullsoft.** Winamp Plugin Page. *Winamp.com.* [Online] http://www.winamp.com/plugins.

11. **Kudielka, Robert.** Supposed to be Abstract. *Parkett.* 2001, 61, p. 26.

12. **Richards, Russell.** Generative Art: Music Generation, Digital Art Production and Nebula. *Nebula 1.3.* January 2005, p. 165.

13. **Future Publishing.** The Fast Guide To Modular Synthesis. *Computer Music Special.* Vol. 26, pp. 36-37.

14. —. The Fast Guide To Wavetable Synthesis. *Computer Music Special.* Vol. 26, p. 19.

15. *Texture Mapping with Vector Graphics: A Nested Mipmapping Solution.* **Zhang, Wei, Yang, Yonggao and Xing, Song.** 2006. International Conference on Computer Graphics and Virtual Reality (CGVR'06). pp. 97-103.

16. **Payne, Oliver.** *Op Art Generator And Animator.* 2006. pp. 29-30.

17. **Arthur Van Hoff, Kathy Walrath.** Animation in Java Applets. *JavaWorld.* [Online] http://www.javaworld.com/javaworld/jw-03-1996/jw-03-animation.html?page=1.

18. **Intel.** *Open Source Computer Vision Library.* [Online] http://www.intel.com/technology/computing/opencv/.

## Appendix 1: Module Template Code

### *ModuleTemplate.java*

```
package opArtGen.API.modules;

import java.util.LinkedList;

import javax.swing.JPanel;

import opArtGen.API.data.GridData;
import opArtGen.API.data.ModuleData;
import opArtGen.API.modules.AbstractModule;
import opArtGen.API.modules.ModuleVisualiser;
import opArtGen.API.modules.Visualisable;
import opArtGen.API.ports.GridPort;
import opArtGen.API.ports.Port;
import opArtGen.API.ports.Port.ReadOnlyCallBack;
import opArtGen.API.ports.Port.WriteOnlyCallBack;

public class ModuleTemplate extends AbstractModule implements
Visualisable {

      /*
       * Module Properties
       */

      // Field to hold value of a module property
      private Object property;

      /*
       * Rendering Support Fields
       */

      // Data sent and received by ports.
      //  - should be a subclass of ModuleData
      private GridData dataObject;

      /*
       * User Interface Support Fields
       */

      private static int instanceNumber = 1;


      /*************************************************************
**
       * Constructors & Destructor

*************************************************************/

      /**
       * Default constructor. Ensure that you call the super
       * constructor and the method initModule
       */
      public ModuleTemplate() {
            super();
            initModule();
      }

      /**
       * Perform any module initialization tasks
       */
      private void initModule() {
            instanceNumber++;
      }

      /*
       * Optional method: used to perform any housekeeping tasks
       * required when module is deleted from the structure
diagram.
       */
      /*
      @Override
      public void destructor() {}
      */


      /*****************************************************************
**
       * Module Properties Support Methods

*************************************************************/

      public Object getProperty() {
            return property;
      }

      public void setProperty(Object property) {
            this.property = property;
      }


      /*****************************************************************
**
       * Rendering Support Methods

*************************************************************/

      /*
       * Processes data in dataObject field based on value of any
       * property fields ready for output by module's ports.
       * Called by the rendering engine
       */
      @Override
      protected void processModuleData() {
            if (dataObject != null && property != null) {
                  // TODO method stub
                  // Process dataObject based on value of property
            }
      }
```

```
	/*******************************************************
**
	 * Module Connectivity Methods
	 
*******************************************************/

	/**
	 * Creates and adds the correct ports for our module
allowing
	 * us to connect it to other modules
	 */
	@Override
	protected void initPorts() {
		// TODO method stub
		inputPorts = new LinkedList<Port>();
		inputPorts.add(new GridPort(getInCallBack()));
		outputPorts = new LinkedList<Port>();
		outputPorts.add(new GridPort(getOutCallBack()));
	}

	/**
	 * Returns a write only CallBack object that enables the
input
	 * port to write data to our dataObject field
	 */
	private WriteOnlyCallBack getInCallBack() {
		return new WriteOnlyCallBack(this) {
			public void storeData(ModuleData data) {
				if (data instanceof GridData) {
				   dataObject = (GridData) data;
				}  else if (data == null) {
				   dataObject = null;
				}  else {
				   throw new IllegalStateException();
				}
			}
		};
	}

	/**
	 * Returns a read only CallBack object that enables the
output
	 * port to read data from our dataObject field
	 */
	private ReadOnlyCallBack getOutCallBack() {
		return new ReadOnlyCallBack(this) {
			public ModuleData loadData() {
				if (dataObject == null) {
					return null;
				}
				return (GridData) dataObject.clone();
			}
		};
	}
	/*******************************************************
**
	 * User Interface Support Methods

********************************************************/

	/*
	 * Returns the module name displayed in the structure
diagram
	 * Note: the name used in context menus is determined by the
	 * module's entry in ModuleNames.txt
	 */
	@Override
	protected String getModuleName() {
		// TODO method stub
		return null;
	}

	@Override
	protected int getInstanceNumber() {
		return instanceNumber;
	}

	/*
	 * Returns a JPanel object containing the components to
	 * display in the module's properties window.
	 */
	public JPanel getPropertiesPanel() {
		// TODO method stub
		return null;
	}

	/*
	 * Remove this method along with the "implements
Visualisable"     * declaration if this module doesn't output any
visualisable      * data.
	 */
	public void visualise(ModuleVisualiser visualiser) {
		refresh();
		visualiser.visualiseData(dataObject);
	}

}
```

## Appendix 2: Properties Window Template Code

*ModulePropertiesWindowPanelTemplate.java*

```
package opArtGen.GUI.propertiesWindow;

import java.awt.Dimension;

import javax.swing.BorderFactory;
import javax.swing.JPanel;
import javax.swing.JTextField;

import opArtGen.API.modules.Module;
import opArtGen.GUI.GUIFactory;

public class ModulePropertiesWindowPanelTemplate extends
PropertiesWindowPanel {

	/**
	 * Version number for serialisation
	 */
	private static final long serialVersionUID = 1L;

	/**
	 * Reference to data object of the module we are editing
	 * properties for
	 */
	private Module moduleObject;

	/**
	 * Example property: module name
	 */
	private String name;

	/**
	 * User interface field for module name
	 */
	private JTextField nameField;


	/**
	 * Default constructor
	 *
	 * @param module: reference to module we are editing
	 * properties for
	 */
	public ModulePropertiesWindowPanelTemplate(Module module) {
		this.moduleObject = module;
		name = module.getName();
		JPanel generalPanel = makeGeneralPanel();
		addTab("General", generalPanel);
		validate();
	}

	@Override
	public Dimension getPreferredSize() {
```

```
            return new Dimension(300, 400);
      }

      /**
       * Helper function called by constructor to aid in building
a
       * tab for the properties window
       *
       * @return JPanel containing user interface objects for
       * General tab
       */
      private JPanel makeGeneralPanel() {
            nameField = new JTextField(name);

            JPanel panel = new JPanel();
            panel.setBorder(BorderFactory.createTitledBorder(
                        BorderFactory.createEtchedBorder(),
                        "Module Properties"));

            //
            // panel.add(component) calls to build rest of General
            // tab
            //

            return
GUIFactory.makePropertiesWindowGeneralPanel(nameField, panel);
      }

      /**
       * Method to check whether the user has changed any property
       * values and update module using set methods accordingly.
       * Called by the parent window when either an OK or Apply
       * button is pressed.
       */
      @Override
      protected void setModuleProperties() {
            if (name.compareTo(nameField.getText()) != 0) {
                  name = nameField.getText();
                  moduleObject.setName(name);
            }
      }

}
```

# Appendix 3: Scripts for Recreations of Existing Op-Art

## Movement In Squares

*MovementInSquaresScript.js*

```
// Import class definitions into Rhino
importPackage(Packages.java.awt);
importPackage(Packages.opArtGen.API);
importPackage(Packages.opArtGen.API.modules);
importPackage(Packages.opArtGen.API.warps);

// Create modules
canvas = new CanvasModule("Movement In Squares");
grid = new GridModule();
shape = new ShapeModule();
warp = new SurfaceWarpModule();

// Load shape
shapeFactory = ComponentLibrary.getShapeFactory();
rectangle =
shapeFactory.createComponentInstance(“rectangle.svg_1”);

// Connect Modules
grid.getOutputPort(0).connect(shape.getInputPort(0));
shape.getOutputPort(0).connect(warp.getInputPort(0));
warp.getOutputPort(0).connect(canvas.getInputPort(0));

// Configure module parameters

grid.setNumRows(10);
grid.setNumCols(31);

canvas.setSize(500, 500);
canvas.setBackground(Color.WHITE);
canvas.setMargins(new Insets(50, 50, 50, 50));

shape.setShape(rectangle);
shape.setSize(1, 1);
shape.setTilingOptions(
ShapeTilingOptions.ALTERNATE_CELLS_FIRST_FILLED);
renderOpts = new ShapeRenderingOptions();
renderOpts.drawLine = false;
renderOpts.drawFill = true;
renderOpts.fillPaint = Color.BLACK;
shape.setRenderingOptions(renderOpts);

cubicWarp = new CubicFunctionWarp();
cubicWarp.setWarpXAxisEnabled(true);
cubicWarp.getParameter("xIntensity").setValue(0.1911777168);
warp.setSurfaceWarp(cubicWarp);

// Display image
canvas.setCanvasWindowVisible(true);
```

## Cantus Firmus

*CantusFirmusScript.java*

```
import java.awt.Color;
import java.awt.Shape;
import java.util.Collections;
import java.util.LinkedList;
import java.util.List;

import opArtGen.API.ComponentFactory;
import opArtGen.API.ComponentLibrary;
import opArtGen.API.modules.AbstractModule;
import opArtGen.API.modules.CanvasModule;
import opArtGen.API.modules.ColourSeriesModule;
import opArtGen.API.modules.GridModule;
import opArtGen.API.modules.Merge2In1OutModule;
import opArtGen.API.modules.ShapeRepeaterModule;
import opArtGen.API.modules.SurfaceWarpModule;
import opArtGen.API.modules.Merge2In1OutModule.MergeType;
import opArtGen.API.scripts.Script;


public class CantusFirmusScript implements Script {

	// Create colours
	final Color green = new Color(189, 188, 41);
	final Color pink = new Color(221, 111, 113);
	final Color blue = new Color(112, 168, 197);
	final Color darkGrey = new Color(34, 26, 23);
	final Color medGrey = new Color(101, 100, 100);
	final Color lightGrey = new Color(167, 168, 163);
	final Color white = new Color(248, 251, 249);

	// Create colour lists
	final List<Color> GPB = new LinkedList<Color>() {
		{add(green);
		 add(pink);
		 add(blue);}
	};
	final List<Color> BPG = new LinkedList<Color>() {
		{add(blue);
		 add(pink);
		 add(green);}
	};
	final List<Color> DDD = new LinkedList<Color>() {
		{add(darkGrey);
		 add(darkGrey);
		 add(darkGrey);}
	};
	final List<Color> DD = new LinkedList<Color>() {
		{add(darkGrey);
		 add(darkGrey);}
	};
	final List<Color> MM = new LinkedList<Color>() {
		{add(medGrey);
```

```
             add(medGrey);}
      };
      final List<Color> LL = new LinkedList<Color>() {
            {add(lightGrey);
             add(lightGrey);}
      };
      final List<Color> WW = new LinkedList<Color>() {
            {add(white);
             add(white);}
      };

      List<AbstractModule> colourModules;

      public List<Color> buildSetWideOnLeft(final boolean
                  greenOnLeft, final List<Color> twoGreys) {
            return new LinkedList<Color>() {
                  {addAll(WW);
                   addAll((greenOnLeft)? GPB : BPG);
                   addAll(DDD);
                   addAll((greenOnLeft)? BPG : GPB);
                   addAll(twoGreys);
                   addAll(GPB);}
            };
      }

      public List<Color> buildSetWideOnRight(final boolean
                  greenOnLeft, final List<Color> twoGreys) {
            return new LinkedList<Color>() {
                  {addAll(WW);
                  addAll((greenOnLeft)? GPB : BPG);
                   addAll(twoGreys);
                   addAll((greenOnLeft)? BPG : GPB);
                   addAll(DDD);
                   addAll(GPB);}
            };
      }

      public List<AbstractModule> merge(List<AbstractModule>
layer) {
            AbstractModule prev = null;
            AbstractModule next = null;

            List<AbstractModule> ret = new
                  LinkedList<AbstractModule>();

            for (int i = 0; i < layer.size() / 2; i++) {
                  prev = layer.get(2*i);
                  next = layer.get((2*i) + 1);
                  Merge2In1OutModule merge = new
                        Merge2In1OutModule();
                  merge.setMergeType(MergeType.HORIZONTAL);

      prev.getOutputPort(0).connect(merge.getInputPort(0));

      next.getOutputPort(0).connect(merge.getInputPort(1));
```

```
                        ret.add(merge);
                }

                System.out.println("Merge");

                return ret;
        }

        public void run() {

                // Create colour lists
                List<List<Color>> colourSets = new
                        LinkedList<List<Color>>();
                colourSets.add(buildSetWideOnLeft(true, LL));
                colourSets.add(buildSetWideOnLeft(false, LL));
                colourSets.add(buildSetWideOnLeft(true, LL));
                colourSets.add(buildSetWideOnLeft(false, LL));
                colourSets.add(buildSetWideOnLeft(true, MM));
                colourSets.add(buildSetWideOnLeft(false, MM));
                colourSets.add(buildSetWideOnLeft(true, DD));
                colourSets.add(buildSetWideOnLeft(false, DD));
                colourSets.add(buildSetWideOnRight(true, DD));
                colourSets.add(buildSetWideOnRight(false, MM));
                colourSets.add(buildSetWideOnRight(true, MM));
                colourSets.add(buildSetWideOnRight(false, LL));

                // Load shape
                ComponentFactory<Shape> sFactory =
                        ComponentLibrary.getShapeFactory();
                Shape rectangle =
                sFactory.createComponentInstance("rectangle.svg_1");

                Merge2In1OutModule lastMerge = null;

                // Create rendering pipeline
                colourModules = new LinkedList<AbstractModule>();

                for (List<Color> list: colourSets) {
                        System.out.println("colourset");
                        GridModule grid = new GridModule(1, 16);
                        ShapeRepeaterModule shape = new
                                ShapeRepeaterModule(rectangle);
                        ColourSeriesModule colour = new
                                ColourSeriesModule(list);
                        colour.setAlterFillColour(true);

        grid.getOutputPort(0).connect(shape.getInputPort(0));

        shape.getOutputPort(0).connect(colour.getInputPort(0));

                        colourModules.add(colour);
                }

                while(colourModules.size() != 1) {
                         colourModules = merge(colourModules);
```

```
            }

            // Render image
            CanvasModule canvas = new CanvasModule("Cantus
Firmus");
            canvas.setSize(400, 400);

      colourModules.get(0).getOutputPort(0).connect(canvas.getInpu
tPort(0));

            canvas.setCanvasWindowVisible(true);

      }

}
```

## Appendix 4: Manual Animation Example

*AnimationScriptExample.java*

```
import java.awt.Shape;

import opArtGen.API.ComponentFactory;
import opArtGen.API.ComponentLibrary;
import opArtGen.API.modules.CanvasModule;
import opArtGen.API.modules.GridModule;
import opArtGen.API.modules.ShapeRepeaterModule;
import opArtGen.API.scripts.Script;

public class AnimationScriptExample implements Script {

      public void run() {

            // Create modules
            CanvasModule canvas = new CanvasModule("TestScript");
            GridModule grid = new GridModule(1, 1);
            ShapeRepeaterModule shape = new ShapeRepeaterModule();

            // Load shape
            ComponentFactory<Shape> sFactory =
              ComponentLibrary.getShapeFactory();
            Shape circle =
              sFactory.createComponentInstance("rectangle.svg_1");

            // Connect modules
            grid.getOutputPort(0).connect(shape.getInputPort(0));

      shape.getOutputPort(0).connect(canvas.getInputPort(0));

            // Configure module parameters
            canvas.setSize(400, 400);
            shape.setShape(circle);

            // Display image
            canvas.setCanvasWindowVisible(true);

            while (true) {
                  for (int i = 1; i < 100; i++) {
                        double size = 0.8 - 0.6 * i / 100;
                        shape.setSize(size, size);
                        try {
                              Thread.sleep(100);
                        } catch (Exception e) {}
                        canvas.updateRenderingPipeline();
                  }
                  for (int i = 100; i > 0; i--) {
                        double size = 0.8 - 0.6 * i / 100;
                        shape.setSize(size, size);
                        try {
                              Thread.sleep(10);
                        } catch (Exception e) {}
                        canvas.updateRenderingPipeline();
                        }
                }
        }
}
```

# Appendix 5: API Reference for Scripting

This section provides an abridged API reference to aid you when writing scripts for OpArtGen.

## ModuleData Classes

### *ColourData*

| Constructors | Description |
|---|---|
| **ColourData()** | Default constructor. Creates a new ColourData object encapsulating a black colour value |
| **ColourData(**int r, int g, int b**)** | Creates a new ColourData object encapsulating a colour that has the given red, green and blue colour values |
| **ColourData(**Color colour**)** | Creates a new ColourData object encapsulating the given colour value |

| Public Methods | Description |
|---|---|
| Color **getColour()** | Returns the Color value encapsulated by this object |
| ColourData **clone()** | Returns an identical copy of this ColourData object. |

### *GridData*

| Constructors | Description |
|---|---|
| **GridData()** | Default constructor. Creates a new GridData object encapsulating a new empty 1x1 grid. All rows and columns are initially equally spaced. |
| **GridData(**int rows, int cols**)** | Creates a new GridData object encapsulating a new empty grid with the specified dimensions. All rows and columns are initially equally spaced. |
| **GridData(**Cell[][] grid**)** | Creates a new GridData object encapsulating the specified array of Cells. All rows and columns are initially equally spaced. |

| Public Methods | Description |
|---|---|
| static Cell[][] **generateCells(**int rows, int cols**)** | Returns a multidimensional array of new empty Cell objects with the specified dimensions. |
| Cells[][] **getCells()** | Returns the multidimensional array of cells representing the grid encapsulated by the GridData object. |
| **setCells(**Cells[][] grid**)** | Sets the grid encapsulated by the GridData object to the multidimensional array of cells specified. If the dimensions of the new grid are different to the previous grid, then the grid spacing is reset to equal. |
| **setEqualGridSpacing()** | Sets the row heights and column widths such that all rows and columns are equally spaced. |
| **setGridSpacing(**GridSpacing spacing**)** | Sets the row heights and column widths to equal those contained within the supplied GridSpacing object. |
| double **getRowHeight(**int row**)** | Returns the proportional height of the row with given index as a fraction of the total height of the grid. |

| | |
|---|---|
| double **getColWidth(**int col**)** | Returns the proportional width of the column with given index as a fraction of the total width of the grid. |
| int **getNumRows()** | Returns the row count for the grid. |
| int **getNumCols()** | Returns the column count for the grid. |
| Dimension **getAbsoluteSize()** | Returns the absolute size of the grid in pixels (device coordinates). |
| **setAbsoluteSize(**Dimension size**)** | Sets the absolute size of the grid in pixels. Recursively updates the absolute sizes for all cells and any child grids. |
| boolean **getGridLinesVisible()** | Gets the value of a flag indicating whether or not grid lines should be drawn for this object. |
| **setGridLinesVisible(**boolean visible**)** | Sets the value of a flag indicating whether or not grid lines should be drawn for this object. Recursively sets the value of the same flag for any child grids. |
| GridData **clone()** | Returns an identical deep copy of this GridData object. |

### *ValueData*

| Constructors | Description |
|---|---|
| **ValueData()** | Default constructor. Creates a new ValueData object encapsulating the value 0. |
| **ValueData(**int value**)** | Creates a new ValueData object encapsulating the given int value. The value is stored internally as a double. |
| **ValueData(**double value**)** | Creates a new ValueData object encapsulating the given double value. |

| Public Methods | Description |
|---|---|
| int **getInt()** | Returns the stored value cast to an int |
| int **getDouble()** | Returns the stored value |
| **setInt(**int value**)** | Stores the given int value. The value is stored internally as a double. |
| **setDouble (**double value**)** | Stores the given value. |
| ValueData **clone()** | Returns an identical copy of this ValueData object. |

## Module Classes

### *AbstractModule*

Note that all modules extend the AbstractModule class.

| Public Methods | Description |
|---|---|
| String **getName()** | Returns the name for this module instance |
| **setName(**String name**)** | Sets the name for this module instance |
| **setDirtyFlag()** | Sets the dirty flag for this module indicating that its output needs to be recalculated by the rendering engine. |
| **clearDirtyFlag()** | Clears the dirty flag for this module |
| boolean **dirtyFlagIsSet()** | Returns the current state of the dirty flag for this module |
| Port **getInputPort(**int index**)** | Returns the input port with the given index |

| | |
|---|---|
| Port **getOutputPort(**int index**)** | Returns the output port with the given index |
| List<Port> **getInputPorts()** | Returns a list holding all the input ports for this module |
| List<Port> **getOutputPorts()** | Returns a list holding all the output ports for this module |

### *CanvasModule*

| Public Methods | Description |
|---|---|
| Dimension **getSize()** | Returns the current image canvas size |
| **setSize(**Dimension size**)** | Sets the size to use for the output image canvas |
| Insets **getMargins()** | Returns the current margins for the output image |
| **setMargins(**Insets margins**)** | Sets the margins to use for the output image canvas |
| Color **getBackground()** | Returns the current image background colour |
| **setBackground**(Color background) | Sets the colour to use for the image background |
| boolean **getCanvasWindowVisible()** | Returns the current visibility of the canvas window |
| **setCanvasWindowVisible(**Boolean visible**)** | Sets the visibility of the canvas window |
| **getCanvas()** | Returns a canvas object containing an overridden paint method able to draw the image specified by the current rendering pipeline |
| **updateRenderingPipeline()** | Updates the rendering pipeline, creating a new canvas object with an overridden paint method able to draw the image specified by the current rendering pipeline |

### *GridModule*

| Public Methods | Description |
|---|---|
| int **getNumRows()** | Returns the number of rows in the output grid |
| **setNumRows(**int rows**)** | Sets the number of rows in the output grid |
| int **getNumCols()** | Returns the number of columns in the output grid |
| **setNumCols(**int cols**)** | Sets the number of columns in the output grid |

### *ColourModule*

| Public Methods | Description |
|---|---|
| Color **getColour()** | Returns the colour value output by this module |
| **setColor(**Color colour**)** | Sets the colour value output by this module |

### *OscillatorModule*

| Public Methods | Description |
|---|---|
| double **getStartValue()** | Returns the value initially output by this module before animation commences |
| **setStartValue(**double value**)** | Sets the start value initially output by this module before animation commences |
| double **getMinValue()** | Returns the minimum value output by this module |
| **setMinValue(**double value**)** | Sets the minimum value output by this module |
| double **getMaxValue()** | Returns the maximum value output by this module |

| | |
|---|---|
| **setMaxValue(**double value**)** | Sets the maximum value output by this module |
| int **getPeriodDuration()** | Gets the duration in frames for each wave cycle output by this module |
| **setPeriodDuration(**int duration**)** | Sets the duration in frames for each wave cycle output by this module |
| **nextFrame(**int FPS**)** | Causes the module to change its output value based on the new frame count and the given animation speed value in frames per second |

### *ShapeRepeaterModule*

| Public Methods | Description |
|---|---|
| Shape **getShape()** | Returns the shape object this module is using |
| **setShape(**Shape shape**)** | Sets a new shape object for this module to use |
| DimensionDouble **getSize()** | Returns the size of the shape placed into each grid cell as a proportion of the total cell size |
| **setSize(**DimensionDouble size**)** | Sets the size of the shape placed into each grid cell as a proportion of the total cell size |
| TilingOptions **getTilingOptions()** | Gets the tiling options this module is using to decide which cells to place shapes into |
| **setTilingOptions(**TilingOptions o**)** | Sets the tiling options this module should use to decide which cells to place shapes into |
| ShapeRenderingOptions **getRenderingOptions()** | Gets the rendering options this module is using for each cell that a shape is placed into |
| **setRenderingOptions(** ShapeRenderingOptions o**)** | Sets the rendering options this module should use for each cell that a shape is placed into |

### *GridRepeaterModule*

| Public Methods | Description |
|---|---|
| TilingOptions **getTilingOptions()** | Gets the tiling options this module is using to decide which cells to place copies of the child grid into |
| **setTilingOptions(**TilingOptions o**)** | Sets the tiling options this module should use to decide which cells to place copies of the child grid into |

### *ColourSeriesModule*

| Public Methods | Description |
|---|---|
| List<Color> **getColoursList()** | Returns the current list of colours that this module repeats over grid data |
| **setColoursList(**List<Color> cs**)** | Sets the list of colours that this module should use to repeat over grid data |
| boolean **isRandomOrder()** | Returns whether the module shuffles the colours list each time it repeats it over grid data |
| **setRandomOrder(**boolean b**)** | Sets whether the module should shuffle the colours list each time it repeats it over grid data |
| boolean **isAlterLineColour()** | Returns whether the module alters the line colour used in each cell of the grid data |

| | |
|---|---|
| **setAlterLineColour(**boolean b**)** | Sets whether the module should alter the line colour used in each cell of the grid data |
| boolean **isAlterFillColour()** | Returns whether the module alters the fill colour used in each cell of the grid data |
| **setAlterFillColour(**boolean b**)** | Sets whether the module should alter the fill colour used in each cell of the grid data |

### *SurfaceWarpModule*

| Public Methods | Description |
|---|---|
| SurfaceWarp **getSurfaceWarp()** | Returns the surface warp object this module is currently using to alter grid spacing |
| **setSurfaceWarp(**SurfaceWarp w**)** | Sets the surface warp object this module should use to alter grid spacing |

### *Merge2In1OutModule*

| Public Methods | Description |
|---|---|
| MergeType **getMergeType()** | Returns the current type of merge this module is performing |
| **setMergeType(**MergeType m**)** | Sets the type of merge that this module should perform |

| MergeType enum Field | Description |
|---|---|
| **HORIZONTAL** | Horizontal merge (see page 74) |
| **VERTICAL** | Vertical merge (see page 74) |
| **OVERLAY** | Overlay merge (see page 74) |

## Port Classes

| Port Type | Description |
|---|---|
| **ColourPort** | Used to send ColourData between modules |
| **GridPort** | Used to send GridData between modules |
| **ValuePort** | Used to send ValueData between modules |

### *Port abstract class*

| Public Methods | Description |
|---|---|
| boolean **isSendPort()** | Returns true if this port object is capable of sending module data objects |
| boolean **isReceivePort()** | Returns true if this port object is capable of receiving module data objects |
| boolean **connect(Port remote)** | Attempts to connect this port to the remote port object. Returns true if the connection was successful, false otherwise |
| **disconnect()** | Disconnects this port from any remote port |
| boolean **isConnected()** | Returns true if this port is currently connected to a remote |

| | |
|---|---|
| | port |
| Port **getRemotePort()** | Returns the remote port object that this port is connected to. Returns null if it is not currently connected to any port |
| Module **getModule()** | Returns the module instance this port belongs to |
| String **getName()** | Returns the name of this port instance |
| **setName(**String name**)** | Sets the name of this port instance |

## Rendering Classes

### *RenderingManager*

| **Public Methods** | **Description** |
|---|---|
| static RenderingManager **getInstance()** | Returns the singleton instance of RenderingManager |
| **startAnimation(**int fps**)** | Starts running the animation loop at the given speed |
| **stopAnimation()** | Stops running the animation loop |
| **updateEffectors()** | Causes all registered effector objects to update via a call to their nextFrame() method |
| **updateCanvases()** | Updates the rendering pipeline on all Canvas Modules |
| **registerEffector(**AnimationEffector e**)** | Registers the given effector object with the manager |

### *CellRenderer*

| **Public Methods** | **Description** |
|---|---|
| Rectangle **getBounds()** | Returns a rectangle object indicating the location of this cell on the output image |
| DimensionDouble **getContentsSize()** | Returns the size of the renderer's contents as a proportion of the parent cell's size |
| **setContentsSize(**DimensionDouble d**)** | Sets the size of the renderer's contents as a proportion of the parent cell's size |
| ShapeRenderingOptions **getRenderingOptions()** | Returns the rendering options this renderer uses while drawing to a graphics object |
| **setRenderingOptions(** ShapeRenderingOptions opts**)** | Sets the rendering options this renderer should use while drawing to a graphics object |
| CellRenderer **clone()** | Returns a clone of this renderer |

### *GridCellRenderer*

| **Public Methods** | **Description** |
|---|---|
| **setGridLinesVisible(**boolean b**)** | Sets a flag indicating whether the grid stored in this renderer should be drawn with the grid lines visible |
| GridCellRenderer **getNextLayer()** | Gets the renderer object for the next layer above this one |
| **setNextLayer(**GridCellRenderer r**)** | Sets the renderer object for the next layer above this one |

### *ShapeCellRenderer*

| Public Methods | Description |
|---|---|
| **setShape(Shape shape)** | Sets the shape object drawn by this renderer |

### *TilingOptions*

| Public enum Fields | Description |
|---|---|
| **COVER_ENTIRE_GRID** | Indicates object should be placed in every grid cell |
| **ALTERNATE_CELLS_FIRST_FILLED** | Indicates object should be placed in alternate grid cells so that the top left grid cell contains an object |
| **ALTERNATE_CELLS_FIRST_EMPTY** | Indicates object should be placed in alternate grid cells so that the top left grid cell does not contain an object |
| **RANDOM** | Indicates object should be placed into grid cells randomly |
| **CUSTOM** | Indicates object should be placed into grid cells specified by a true value in enum's custom mask |

| Public enum Methods | Description |
|---|---|
| boolean[][] **getCustomMask()** | Returns a boolean array indicating which cells an object should be placed into |
| **setCustomMask(b**oolean[][] m**)** | Sets a boolean array indicating which cells an object should be placed into |

## Surface Warp Classes

### *SurfaceWarp*

| Public Methods | Description |
|---|---|
| boolean **isWarpXAxisEnabled()** | Returns a value indicating whether the surface warp should modify grid column widths |
| **setWarpYAxisEnabled(**boolean b**)** | Sets a value indicating whether the surface warp should modify grid column widths |
| boolean **isWarpYAxisEnabled()** | Returns a value indicating whether the surface warp should modify grid row heights |
| **setWarpYAxisEnabled(**boolean b**)** | Sets a value indicating whether the surface warp should modify grid row heights |
| Set<String> **getParameters()** | Returns a set containing all parameter names for this surface warp |
| WarpParameter **getParameter(**String name**)** | Gets the warp parameter object for the given parameter name. The parameter value may be altered by modifying the object that this method returns. |
| GridSpacing **applyWarp(**int rows, int cols**)** | Returns a grid spacing object containing the new grid spacing values for a grid of the given size |

### *WarpParameter*

| Constructors | Description |
|---|---|
| **WarpParameter(**String name, String description**)** | Creates a new warp parameter with the given name and description |

| | |
|---|---|
| **WarpParameter(**String name, String description, double minValue, double maxValue, double defaultValue**)** | Creates a new warp parameter with the given name and description; whose value is initially equal to defaultValue and always between minValue and maxValue |

| Public Methods | Description |
|---|---|
| String **getName()** | Gets the name of this warp parameter object |
| String **getDescription()** | Gets the description of this warp parameter object |
| double **getValue()** | Gets the current value of this warp parameter object |
| **setValue(**double value**)** | Sets the new value for this warp parameter object |

## Component Loading Classes

### *ComponentLibrary*

Note all methods return either a *ComponentLoader* object, or a *ComponentFactory* object

| Public static Methods | Description |
|---|---|
| **getLoader()** | Returns the loader that loads component loaders |
| **getLoaderFactory()** | Returns a factory able to create external loader objects |
| **getModuleLoader()** | Returns the loader that loads modules |
| **getModuleFactory()** | Returns a factory able to create external module objects |
| **getRendererLoader()** | Returns the loader that loads cell renderers |
| **getRendererFactory()** | Returns a factory able to create external cell renderer objects |
| **getWarpLoader()** | Returns the loader that loads surface warps |
| **getWarpFactory()** | Returns a factory able to create external surface warps |
| **getShapeLoader()** | Returns the loader that loads shapes |
| **getShapeFactory()** | Returns a factory able to create shape objects |

### *ComponentLoader<T>*

| Public Methods | Description |
|---|---|
| Dictionary<String, T> **getComponents()** | Returns a dictionary linking component names to component objects that were loaded by this component loader at run-time. If components have not yet been loaded then calling this method will cause the loader to attempt to load them. |
| ComponentFactory **getFactory()** | Returns a factory object able to create component instances for all components loaded by this loader. If components have not yet been loaded then calling this method will cause the loader to attempt to load them. |

### *ComponentFactory<T>*

| Public Methods | Description |
|---|---|
| T **createComponentInstance(** | Returns a new instance of the component with given |

| | |
|---|---|
| String name**)** | name |
| Dictionary<String, T> **createComponentInstances()** | Returns a dictionary linking component names to new instances for all components this factory is able to create |

## *AbstractClassComponentLoader*

| Method | Description |
|---|---|
| protected String **getComponentTypeName(int multiplicity)** | Returns the name of your custom components in English for the given multiplicity: i.e. for 1: "My Component", or for 5 : "My Components" |
| protected String **getComponentSourceFileMask()** | Returns a string containing a Java style RegEx pattern that identifies java source files for your custom components. For example, if the various classes for are named as: "Type1MyComponent.java, Type2MyComponent.java ..." then this method should return the Java style RegEx string "MyComponent.java$". |
| protected String **getComponentClassFileMask()** | Returns a string containing a Java style RegEx pattern that identifies compiled java class files for your custom components. For the above example this method should return: "MyComponent.class$" |
| protected String **getExternalComponentsPackageName()** | Returns the name of the package that contains either the source or compiled class files for your custom components. For the above example this method may return: "opArtGen.API.external.mycomponents" which would correspond to the directory opArtGen/API/external/mycomponents. *-Any source files are automatically compiled at runtime* |
| protected int **loadBuiltInComponents()** | This method should add any built-in classes for your custom components to the inherited field *components*. If you are writing a component loader that will only load external components at runtime, then this method should simply return. |
| public ComponentFactory<MyComponent> **getFactory()** | This method should return a factory instance object for creating new instances any external classes loaded at runtime. |