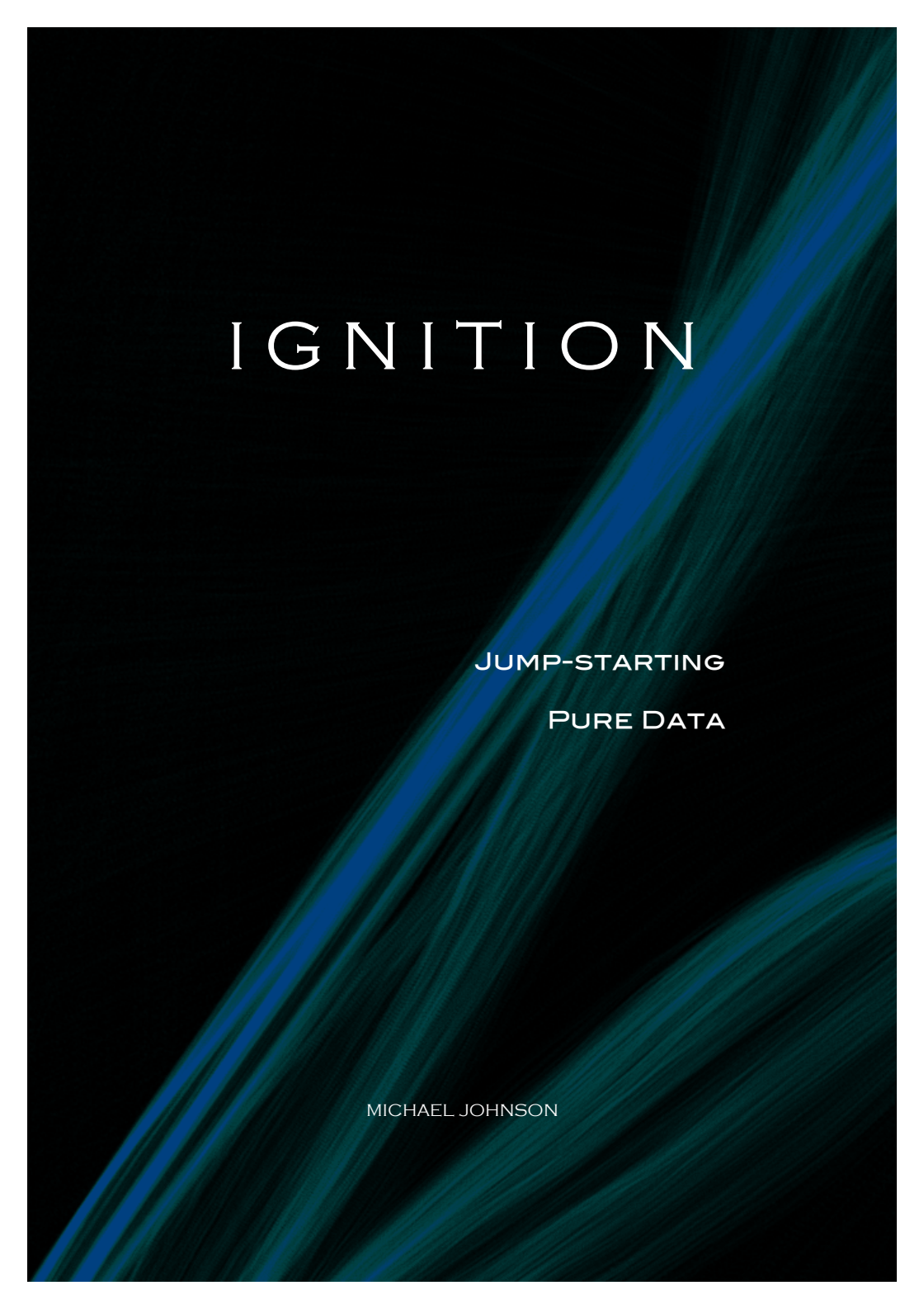




IGNITION

JUMP-STARTING

PURE DATA

MICHAEL JOHNSON

IGNITION
JUMP-STARTING PURE DATA

2012

Michael Johnson

Cover Design: Michael Johnson

Printing: touchpaper Graphics

0801480

RWCMD

Abstract

Pure Data is an open-source programming language used primarily for sound and visual applications. Ignition is the author's teaching method that offers a beginner's guide to the structure and use of the language, with step-by-step examples to create basic projects.

Chapter One presents an overview of the windows, menus and main functional components, and introduces the basic concepts of the language architecture.

Subsequent chapters guide the reader through the process of creating musical and graphical patches, combining into audio-visual applications. On completion of the examples and exercises, progressive un-aided assignments are set.

A data disc complements the method.

Table of Contents

Foreword	9
Rationale	11
Target User	13
The Method	14
 Chapter One: Pd At First Glance	 19
 Chapter Two: Sound	 35
Enabling Audio	35
2.1. Making Sound	39
2.2. Sequencing Sound	45
2.3. MIDI	51
2.4. Audio File Playback	53
Exercise: One	55
Exercise: Two	63
Assignment: One	66

Chapter Three: Visuals	69
3.1. The GEM Window	71
3.2. Objects Within GEM	77
3.3. GEM Object Images	89
3.4. Zoning the GEM Window	93
Exercise: Three	103
Assignment: Two	122
Assignment: Three	122
 Chapter Four: Audio-visuals	 125
4.1. Abstractions	125
Exercise: Four	129
Assignment: Four	157
 Debrief: The End Of Your Pd Jump-Start	 159
Where To Go Next	161
Acknowledgements	163

Foreword

Pure Data (Pd) is an object based programming system which is entirely open source. There are similar commercial programs available such as Max/MSP, Supercollider and AudioMulch, but Pd is free to download and use. Pd enables people who have no background in conventional programming to create all manner of different products, whether they be instruments, interactive applications, games, visualisations or data decoders. The list goes on and is only limited by the users' needs and imagination. An old English proverb sums up the diverse nature of the program:

“If there's a will, there's a way.”

Miller Puckette created Pd during the 1990s while he was at IRCAM (Institute for Research and Coordination in Acoustics and Music). It is part of a family of patcher programming languages, which includes Max/MSP, all of which continue to grow in appeal within the computer music world.

Pd continues to be maintained by Miller Puckette and as its community grows, so does the number of developers who contribute to its on-going development. These developments exist in both new releases of Pd, and external libraries that effectively bolt themselves on to the Pd application.

Rationale

I first came into contact with Pd in 2008 while at a university open day. At the time it appeared very daunting. Then in 2009 I was able to take part in an intensive week-long introductory course, but it was another year before I really started to utilise what I had learnt. I attended ‘PDCon 2011’, held in Weimar, Germany (a worldwide convention for developers and users to showcase and share their research), where my understanding of Pd in the computer music genre was reinforced.

My knowledge and experience of Pd derives from several sources; mostly self-teaching, with direction and guidance from Simon Kilshaw, a leading lecturer in Creative Music Technology and Pd specialist.

‘Patching’ – the process of writing in Pd – has evolved through developers and the Pd community. Pd contains several tutorials, however, given its open source nature, development has rarely been captured in formal documentation. Information found on the internet largely exists in Pd community forums. Here you can ask specific questions or observe videos of people working through patches. Few hardcopy books exist: a notable one is “Loadbang”¹.

The lack of practical written guidance led me to consider creating this teaching method for the purpose of nurturing newcomers to Pd.

¹ “Loadbang – Programming Electronic Music in Pure Data” – Johannes Kreidler

Target User

This method is for beginners wanting to program in Pd, especially students studying in further education, programming hobbyists, multimedia enthusiasts and music technologists.

Competent computer users won't be fazed by the method, and no prior programming skill is expected.

The method concentrates on sound and visual aspects of Pd, including their interaction. Having a general musical and mathematical understanding will enhance the learning outcomes.

The Method

Each chapter introduces fundamental information including specific boxes and illustrated programming steps. This is followed by a guided exercise which utilises the covered material. There is also an un-aided assignment at the end of each chapter with possible solutions provided on the accompanying disc.

Chapter One introduces Pd and presents an overview of its windows, menus and main functional components. It also introduces the basic concepts of the language architecture and where to look within Pd for more information.

Chapter Two looks at creating sound. It leads the reader step-by-step through the processes of

creating and using a sound wave generator to sequence a scale and melody. It also introduces the use of audio files and how to play them. It touches on MIDI, and how to send MIDI data to another program running on the same computer.

Two guided exercises enable the user to create an arpeggiated tone row backing and a solo melody, and a multiple audio file sound-scape generator. The assignment challenges the user to sequence and play a well known song.

Chapter Three gets to grips with Pd's graphical environment window. Again a step-by-step guide shows the user how to create, customise and manipulate graphical objects within this window.

It explains how to process positional information from the mouse, to produce triggers.

The exercise extends the learning and use of triggers, creating an interactive patch. The first assignment is to take control of and manipulate a movie file. The second assignment involves adding images to a trigger zone.

Chapter Four introduces, explains and utilises abstractions, consolidated within the final exercise. This exercise sees the creation of a simple interactive music patch, where abstractions are at the heart of the audio-visual amalgamation. The last assignment asks the user to create a two-tone synth.

This method is intended to be used as a practical aid. The user can recreate and work alongside the examples and exercises, creating and saving the patches for further experimentation. Example patches, exercises, useful links, and the whole method itself is also contained on an accompanying data disc.

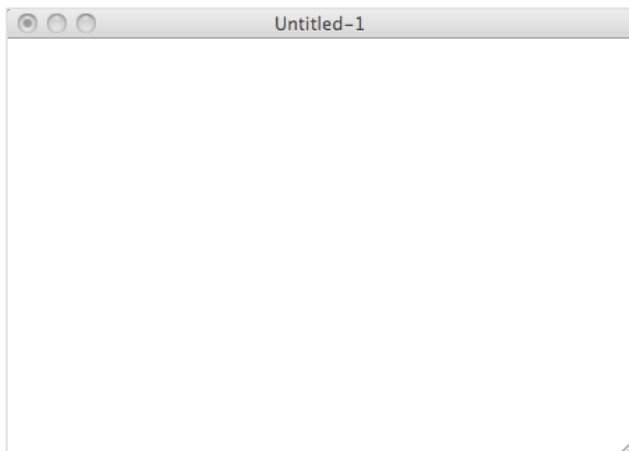
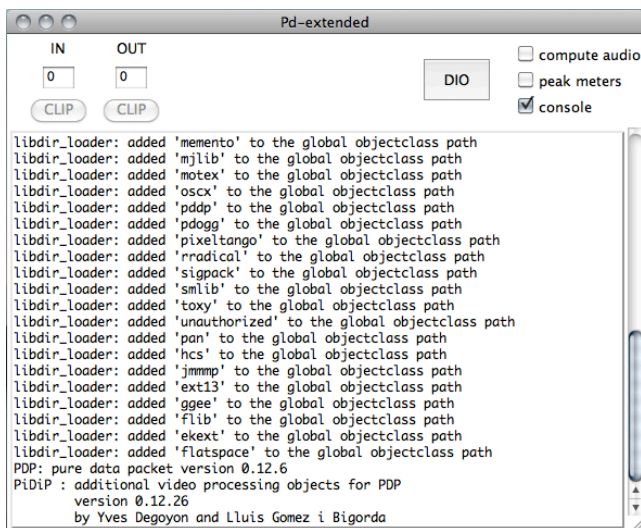
At the end of the method, users will feel confident to develop their own simple patches, to experiment and integrate Pd with their own creative practices, and converse intelligently with the wider Pd community.

Chapter One: Pd At First Glance

Chapter One introduces Pd and presents an overview of its windows, menus and main functional components. It also introduces the basic concepts of the language architecture and where to look within Pd for more information.

There are several current versions of Pd, all of which are viable options, but some are better equipped than others. ‘Pd Vanilla’ is the basic form of Pd and contains the standard set of coding found in all versions of Pd. ‘Pd-extended’ (version 0.42.5), as the name suggests, is a release that contains extended libraries, extensions and documentation. The newest release is another version of ‘Pd-extended’ (version 0.43), containing bug fixes, and is effectively a trial release. For the purposes of this method, ‘Pd-extended’ (version 0.42.5) is used.

Pd is able to run on most operating systems, but this method approaches Pd from a Mac perspective.



There are two main windows to observe while using Pd. The first is the ‘Pd window’ and this displays a stream of information about the patch, including error messages and print data. This window will tell you if something is going wrong.

When Pd is launched, only the ‘Pd window’ will be visible.

The ‘Parent window’, sometimes referred to as the ‘canvas’, is where all patching takes place. The ‘Parent window’ is created by navigating the menu and selecting:

File → New

bang 1

message

0

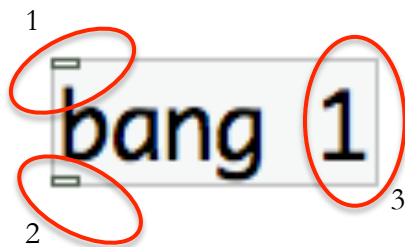
Patching in Pd consists of connecting different types of boxes together in certain patterns and resembles an 'Input → Process → Output' chain of events.

The three main types of boxes used to create patches in Pd are listed and explained below:

'object' box – this box is used more often than others, and is the building block of any patch. Each 'object' represents lines of code designed to do very specific tasks.

'message' box – the message box outputs information, either numbers or symbols, contained within the box, when called upon.

'number' box – a number box is a numerical display. It can be used to send or receive numerical data.



The anatomy of boxes is fairly coherent between the different types. Shown is an object box with important aspects labelled.

1: 'Inlet' – this is the part of the box that receives data. There can be more than one of these.

2: 'Outlet' – this is the part of the box that sends data. There can be more than one of these.

3: 'Argument' – an argument is used to control default values of a box. It can also affect the number of inlets and outlets a box can have.

Note: Only outlets are able to connect to inlets, and a box cannot connect directly to itself.

Put	Find	Media
Object	⌘1	bang 1
Message	⌘2	message
Number	⌘3	0
Symbol	⌘4	
Comment	⌘5	
Bang	⇧⌘B	
Toggle	⇧⌘T	
Number2	⇧⌘N	
Vslider	⇧⌘V	
Hslider	⇧⌘H	
Vradio	⇧⌘D	
Hradio	⇧⌘I	
VU	⇧⌘U	
Canvas	⇧⌘C	
Graph		
Array		

The process of creating boxes is conducted from the ‘Put’ menu. An example of how to create the “bang 1” object box is as follows:

‘Put’ → ‘Object’ → Click to place → Type “bang 1”

Every ‘Object’ must be named.

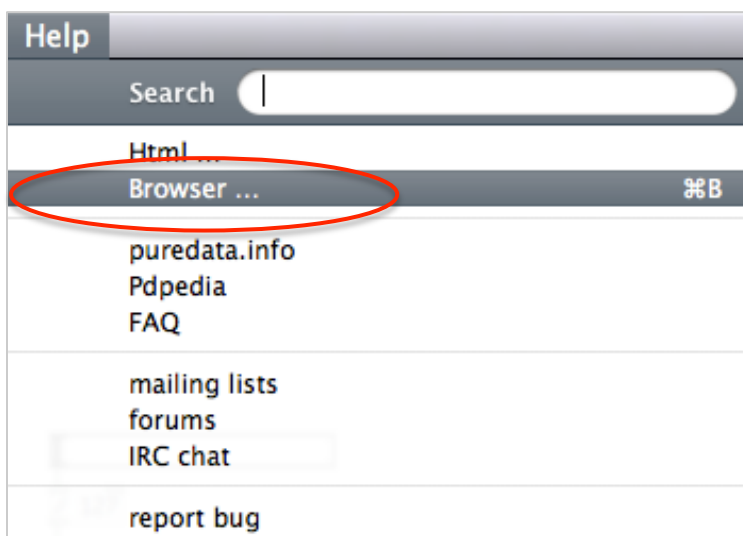
Three more useful objects found in the ‘Put’ menu are “bang”, “toggle” and “Hslider”.

“bang” is used as a ‘do it’ button.

“toggle” works like a light switch and outputs 0 when off, and 1 when on.

“Hslider” is an object that can be used to select whole numbers from 0 to 127. “Vslider” is exactly the same but sits vertically on the canvas.

Whenever ‘Object’, ‘Message’, and ‘Number’ boxes are mentioned they will be colour coded as shown.

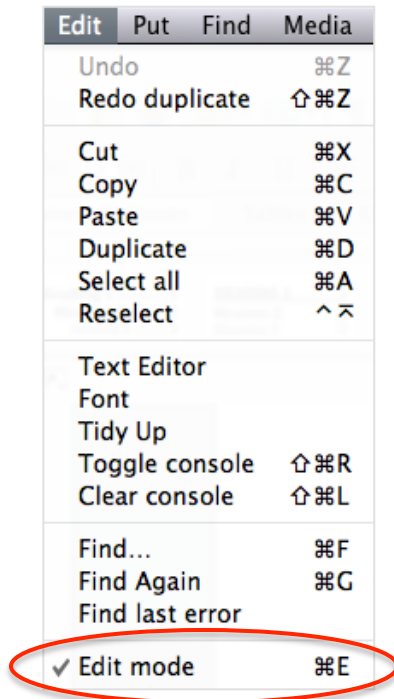


When an ‘object’ is left blank, or an incorrect function name is entered, the box will turn red, and an error message appears in the ‘Pd window’.

For more information about any ‘object’ you can right click the ‘object’ and select ‘help’. There is not a library of ‘objects’ listed in Pd, but examples can be found by navigating to:

Help → Browser

As with learning any new language, it takes time to build up a vocabulary.



The canvas has two modes: the ‘Edit’ mode, and the ‘Run’ mode. The ‘Edit’ mode is used to create and build patches, and the ‘Run’ mode is used to interact with boxes created in ‘Edit’ mode.

To switch between the two modes select:

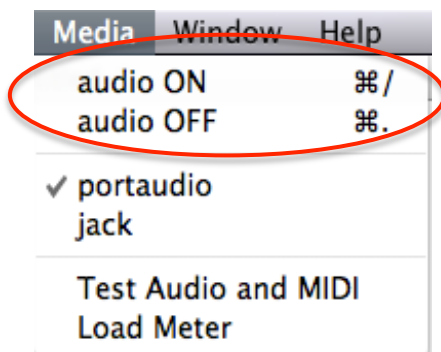
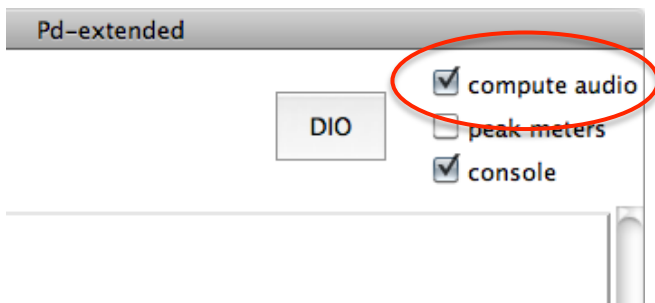
‘Edit → Edit mode’

Learning the Pd shortcuts for your operating system is strongly advised. It becomes very tedious going back and forth between the canvas and menu bar.

Chapter Two: Sound

Chapter Two looks at creating sound with common objects used in sound patches. It leads the reader step-by-step through the processes of creating and using a sound wave generator to sequence a scale and melody. It also introduces the use of audio files and how to play them. It touches on MIDI, and how to send MIDI data to another program running on the same computer.

Two guided exercises enable the user to create an arpeggiated tone row backing and a solo melody, and a multiple audio file sound-scape generator. The assignment challenges the user to sequence and play a well known song.

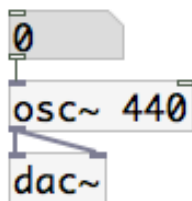
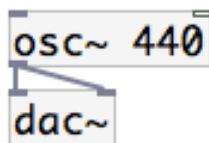


Enabling Audio

Before any sound can be heard from the computer, the ‘computer audio’ must be enabled within Pd. This can either be enabled/disabled using a checkbox in ‘Pd window’, or by selecting:

Media → Audio ON

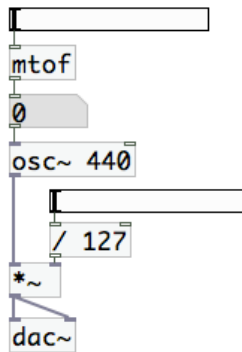
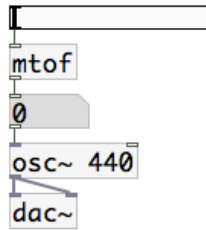
Media → Audio OFF



2.1. Making Sound

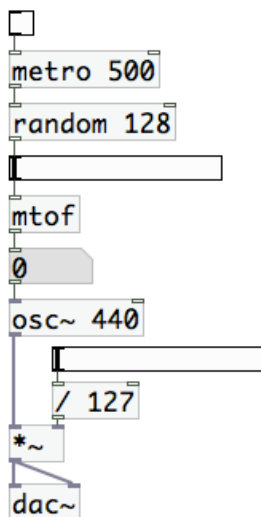
The easiest way to make Pd produce a sound is by using the “osc~” object. This is a simple sine wave generator and will oscillate at any desired frequency typed after ‘osc~’. In this example 440Hz is used, also known as A4, or tuning A. This is connected to both inlets of the “dac~” object. ‘dac’ stands for ‘digital analogue converter’, and is used to send audio from Pd to the computer speakers. The ‘~’ (tilde) is used because audio signals are being transmitted as opposed to data.

By adding a “number” box to the left inlet of the “osc~ 440” object, the frequency can be dynamically changed by dragging up and down with the mouse.



The slider objects, “[hslider](#)” and “[vslider](#)”, have a default output range of 0 – 127, the same values as MIDI. This means using a “[slider](#)” and an “[mtof](#)” object can produce tuned frequencies. The “[mtof](#)” object, ‘MIDI to frequency’, converts MIDI note numbers into frequencies. Now as the slider is moved, it will always produce a chromatic scale.

A slider can also be used to control volume, but there is no volume object. In this method, the “[*~](#)” object is used instead. In Pd, volume works on a scale of 0 - 1. The slider has a range of 0 – 127; it must be scaled to conform to the volume’s range. This is done by dividing the slider by its top value, 127. Any number divided by itself is 1 ($1/1=1$, $50/50=1$, $127/127=1$). This group almost always precedes the “[dac~](#)” object.



Congratulations!

You have created your first patch.

Don't forget to save.

To add some automation to the MIDI notes generated, a ‘metro/random’ combination is used. The “[random 128](#)” object is used to produce random whole numbers between 0 and 127 every time it receives a bang. (“[random 5](#)” will produce random integers between 0 and 4). A “[metro 500](#)” object is used to send a bang message every 500 milliseconds; it is Pd’s version of a metronome, and is started and stopped using a “[toggle](#)” object.

Note: MIDI note numbers represent western pitches (scales) in an electronic format.

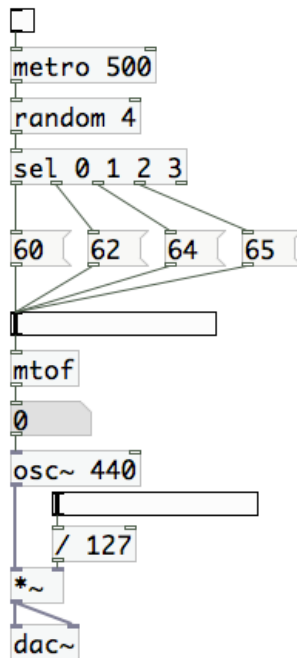
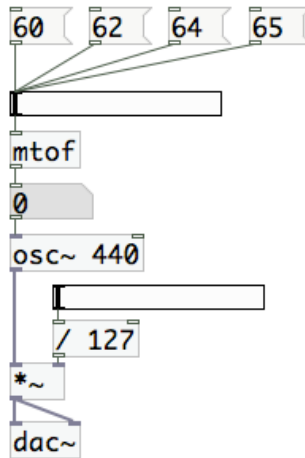
E.g.

60 = Middle C

62 = D

64 = E

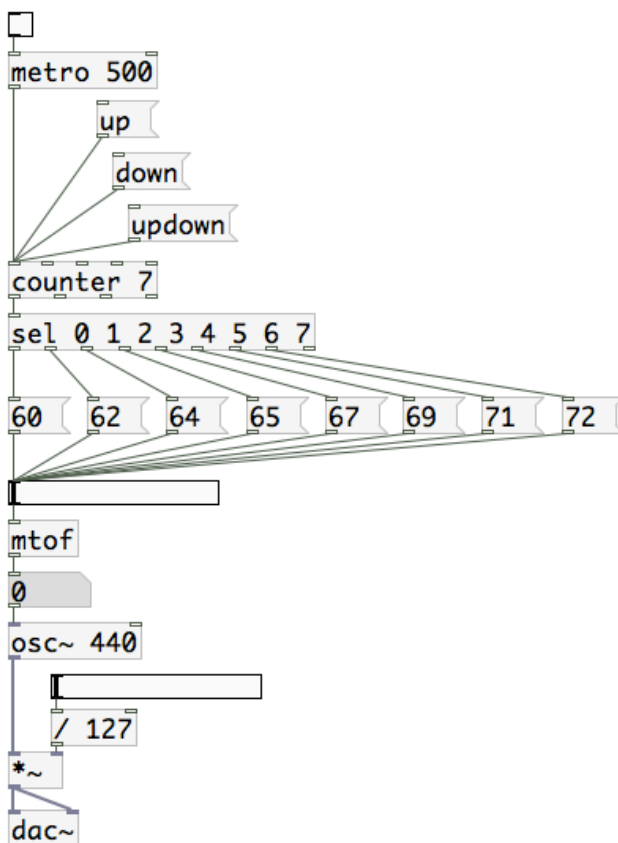
65 = F



2.2. Sequencing Sound

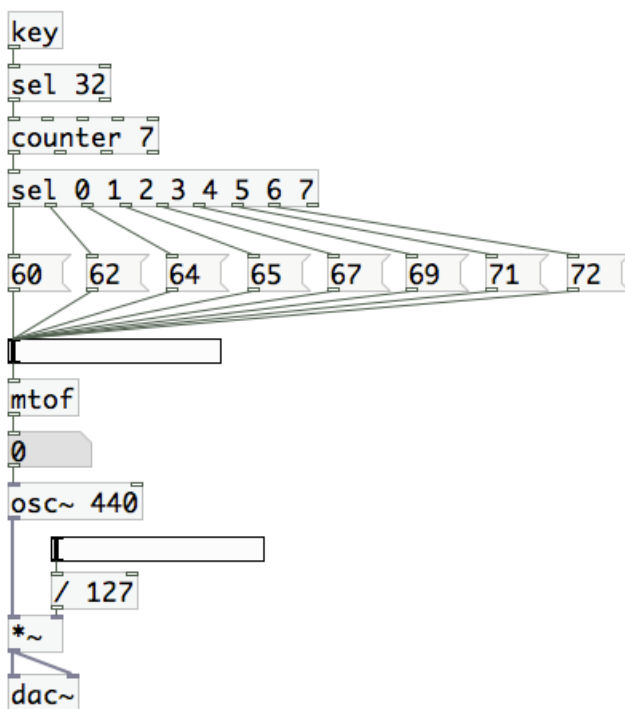
Using the previous patch, delete everything above the “[hslider](#)”, and create four “[message](#)” boxes. Sending numbers from message boxes means specific MIDI notes are played.

Re-create the ‘toggle/metro/random’ sequence, except set the “[random](#)” value to 4. This sequence can now be used to randomly bang messages using the “[sel](#)” object. “[sel](#)” stands for select, and will bang when the incoming number matches an argument within the “[select](#)” object. It has one more outlet than the number of arguments within it. The position of the argument determines which outlet bangs, and last outlet bangs when a number is received for which there is no matching argument. But that won’t happen in this scenario.

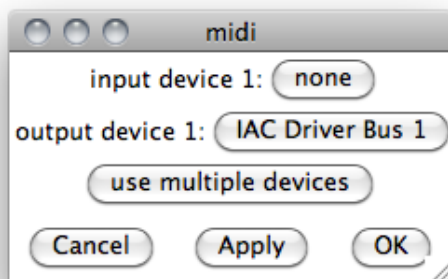
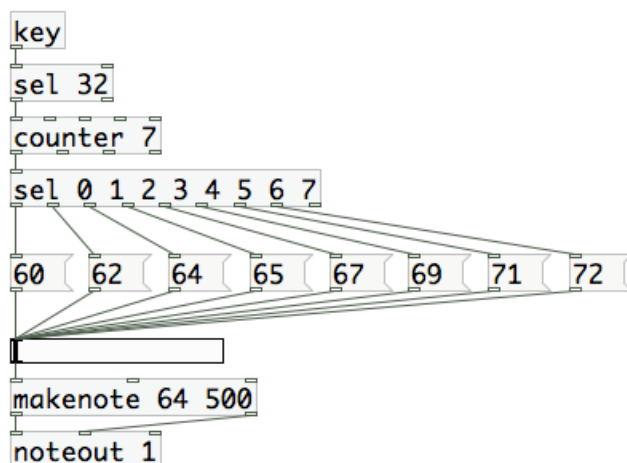


Replacing the “[random 4](#)” object with a “[counter](#)” object enables the patch to cycle through a series of MIDI notes. In this example, messages have been added to make up a full C major scale. If the “[counter](#)” object does not have an argument, it will continue to increase its number every time it receives a bang at the first inlet. The default count direction is up and will start on 0. The count direction can be dynamically changed with “[up](#)”, “[down](#)”, and “[updown](#)” messages sent to the first inlet.

Note: the “[updown](#)” message requires an argument in order to function.



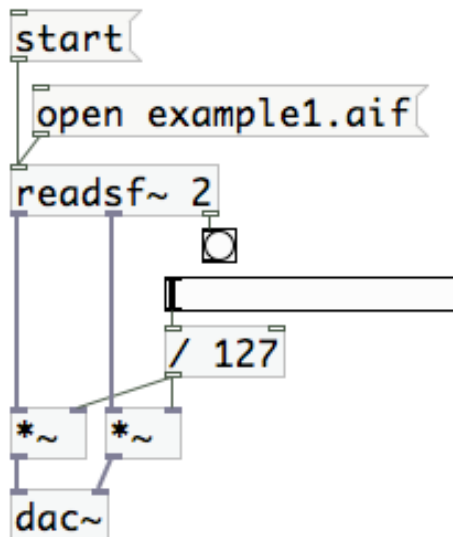
To regain **manual control** over the speed at which the messages are cycled, the “key” object is used. This means instead of clicking a bang object, the next message can be triggered using any key on the computer keyboard. Every key has its own unique number; this can be found by connecting a “key” object to a “number” box, and then pressing the desired key. This example uses the spacebar, which is the number 32. Using the “sel 32” object, the next message is banged every time the spacebar is pressed.



2.3. MIDI

Sending MIDI data from Pd to another piece of software only requires two objects. Replace everything after the “[hslider](#)” with “[makenote 64 500](#)” and “[noteout 1](#)” objects. The “[makenote](#)” object has two arguments: the first ‘64’ is the MIDI velocity (range 0 to 127), and the second ‘500’ is the duration in milliseconds. The “[noteout](#)” object has one argument determining the MIDI channel number.

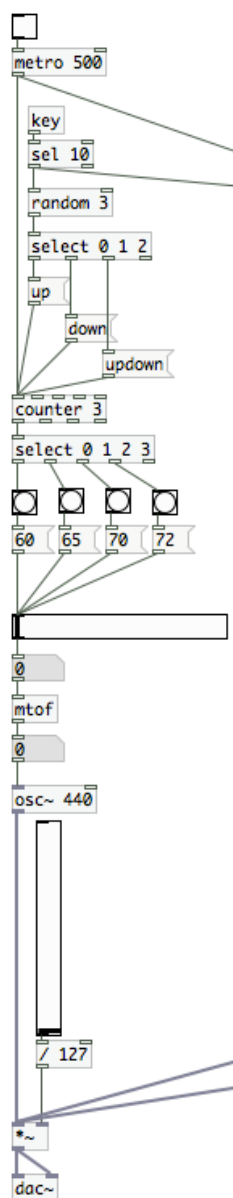
For Mac users: Make sure the ‘IAC Driver Bus1’ is the designated MIDI output device in Pd (found in Pd MIDI settings), and the input device for the destination software.



2.4. Audio File Playback

It is possible to use Pd to play audio files using the “`readsf~ 2`” object. This has a number of arguments, but the only one used in this example is the number of playback channels. There are two “`*~`” objects, to control volume for each audio channel.

Note: Save the Pd patch and the desired audio file in the same folder, otherwise Pd won't be able to locate the audio file. Create an “`open example1.aif`” message, replace ‘example1.aif’ with desired ‘*filename.filetype*’, then use a “`start`” message to play the file. Once the file has finished playing, the last outlet will bang. The file will not play again until the “`open`” message has been banged.

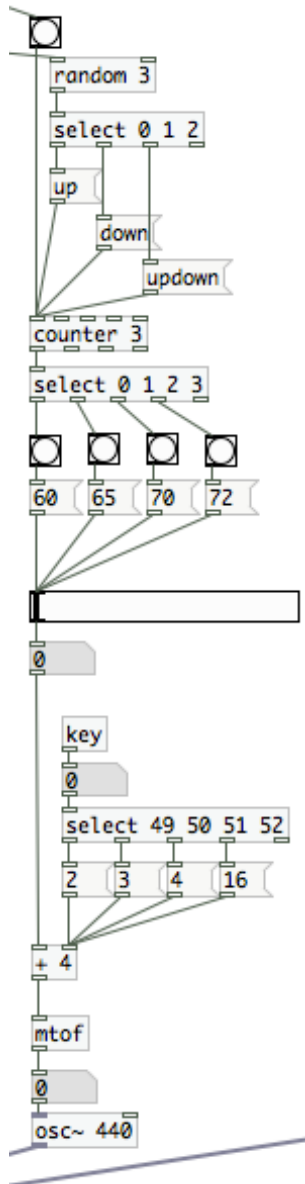


Exercise: One

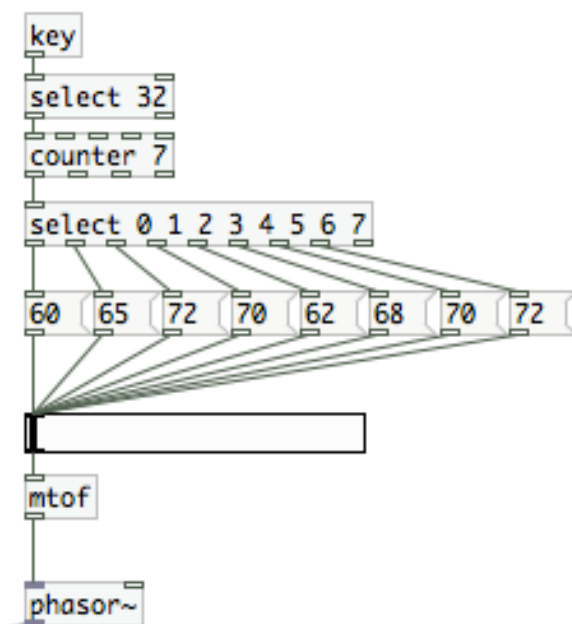
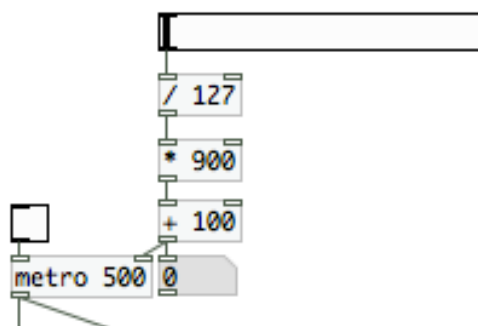
The aim of this exercise is to create an arpeggiated tone row backing and a solo melody.

The first step is to decide on a MIDI tone row. This can be any set of four numbers from 0-127, contained in messages. Similar to examples earlier in the chapter, create a “counter” object to cycle through the tone row.

Instead of the counter always counting the same way, and cycling through the tone row in the same direction, a “random 3” and “sel 0 1 2” combination can be used to randomly change the counter’s direction. To control it, the “key” object is used again. This time the keyboard ‘enter’ button will be used to bang the “random 3” object.

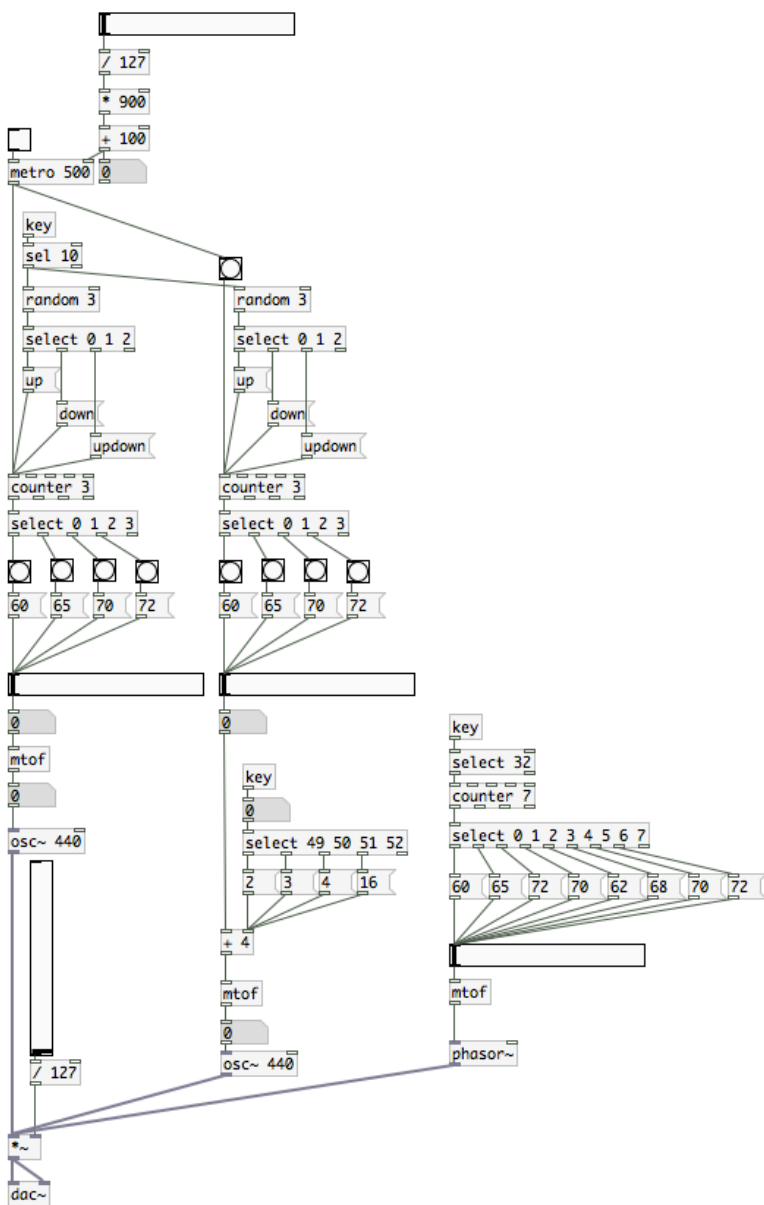


To create harmony from the arpeggiator, select and duplicate everything between “random 3” and “osc~ 440” objects. Now between the “number” box and “mtof” object a harmony routine will be added. This is controlled using the “key” object again. Using the “key” object attached to a “number” box, the unique numeric values of the QWERTY keyboard can be found. Numbers 1-4 on the QWERTY keyboard have unique numbers from 49-52 respectively. Selecting these numbers and attaching them to messages containing numbers, and then to a “+ 4” object, increases the MIDI interval, and when played with the first part of the patch, creates harmonies based on the message numbers.



The speed at which the “metro” object bangs can be changed dynamically, and can be controlled using a slider. As stated before the range of the slide is between 0 and 127. Dividing the slider output by 127, “/ 127”, multiplying by 900, “* 900” and then adding 100, alters the range to go between 100 and 1000 milliseconds.

To create the solo melody line, the “key”, “select” and “counter” objects are used in the same fashion, except this time extend the tone row out to eight notes. Change the progression of notes so the melody flows. This time the ‘spacebar’ will be used to bang the next note. To give the melody line a more distinguishable sound, use a “phasor~ 440” instead of an “osc~ 440” object.

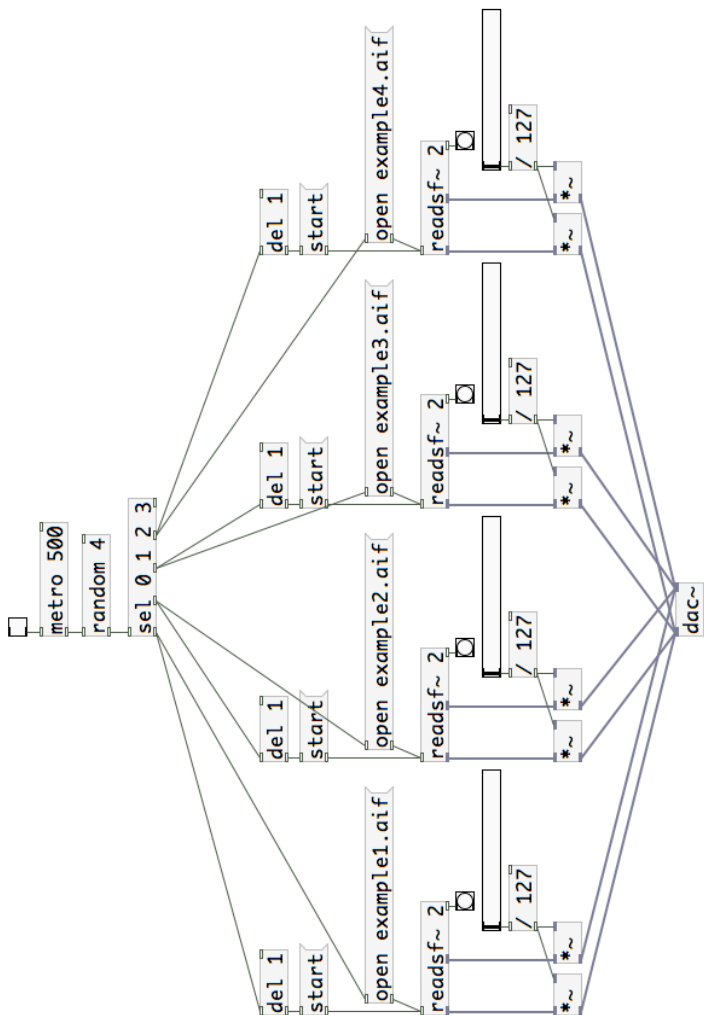


When each part is combined together this is how the full patch will look. There is only one volume control, “*~”, and each of the wave generators goes to it.

Remember to click the “toggle” above the “metro” to go.

Remember to turn the volume slider up.

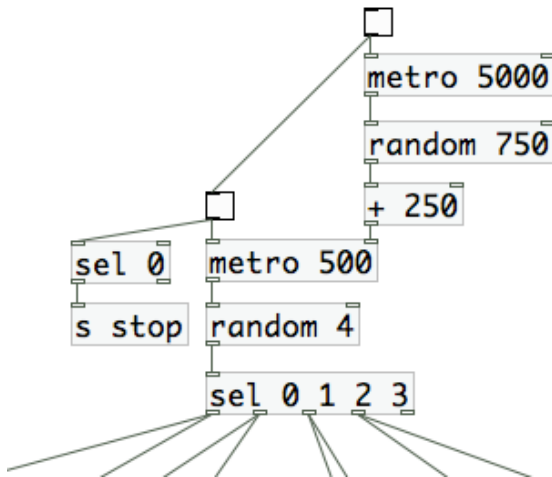
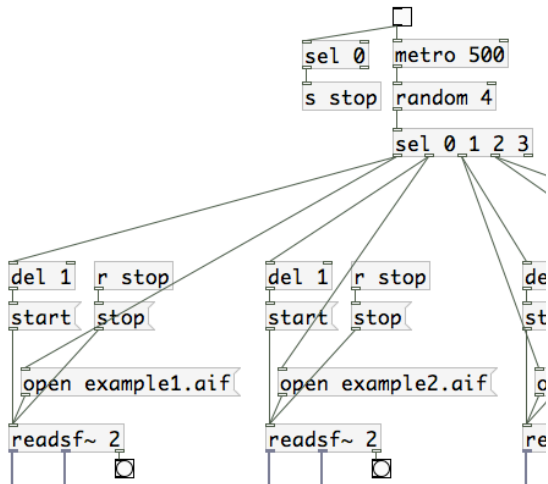
Remember to turn the computer audio on from the ‘Media’ menu in Pd.



Exercise: Two

The aim of this exercise is to create a sound-scape using audio samples.

Re-create the file player cluster from the ‘Audio File Playback’ section in this chapter, and duplicate until there are four clusters. Change the audio files so that they are all different, and connect them to the same “[dac~](#)”. Create a ‘metro/random/select’ cluster from the ‘Sequencing Sound’ section, to randomise the audio files. Before connecting the “[select](#)” object to the “[start](#)” messages, create a “[delay 1](#)” object and connect it to the “[start](#)” message. This is used to delay a bang by one millisecond, because the “[open](#)” message always needs to be banged before the “[start](#)” message in order for an audio file to play.



To ensure the files stop playing when the “metro” object is turned off, connect a “sel 0” object to the “toggle” controlling the “metro”. Then create a “send” object, “s stop”. The send object “s *name*” sends any data it receives, to a receive object “r *name*” with the same name. Create “r stop” objects, connect them to the “stop” messages, and connect these to the “reads f~ 2” object.

At the moment the “metro” is set to bang at regular intervals. To randomise the metro speed between 250 and 1000 milliseconds, create a “metro 5000”, and a “random 750” objects. Then connect this to a “+ 250” object, and then connect this to the second inlet of the first metro. The “toggle” objects have been connected so there is only one master go/stop “toggle”.

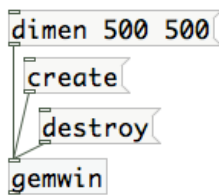
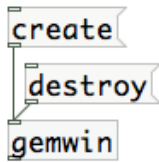
Assignment: One

Using the techniques learnt within this chapter, sequence the notes of 'Happy Birthday' and play them back using the 'spacebar' to progress to the next note.

Chapter Three: Visuals

Chapter Three gets to grips with Pd's graphical environment window. Again a step-by-step guide shows the user how to create, customise and manipulate graphical objects within this window. It explains how to process positional information from the mouse, to produce triggers.

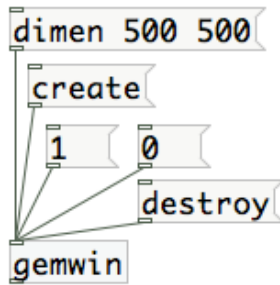
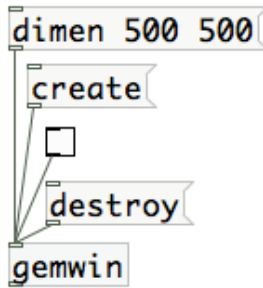
The exercise extends the learning and use of triggers, creating an interactive patch. The first assignment is to take control of and manipulate a movie file. The second assignment involves adding images to a trigger zone.



3.1. The GEM Window

GEM in Pd is an acronym for ‘Graphical Environment for Multimedia’, and is used to display graphics. To create a GEM window only two boxes are needed: a “**create**” message, connected to a “**gemwin**” object. A GEM window is created when the “**create**” message is clicked; there can only ever be one GEM window. To close the GEM window, a “**destroy**” message is connected to the “**gemwin**” object.

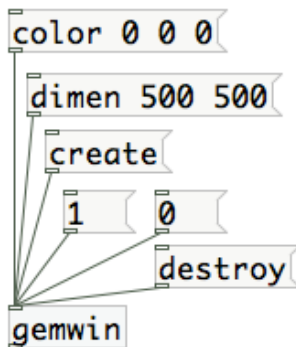
The default size of the GEM window is 500 x 500 pixels. To change the GEM window size a “**dimen x y**” message, where x and y are the number of pixels wide and tall respectively, is connected to the “**gemwin**” object in the same vein as before.



Once the GEM window has been created, in order for any graphics to be displayed rendering must be enabled. Rendering a GEM window means an internal connection is set up between the GEM window and any GEM objects created within the patch. If rendering is not enabled, nothing will be displayed. Rendering uses values of 1 and 0 denoting on/off states, and is sent to the “[gemwin](#)” object.

To ensure rendering is turned on, either create a “[toggle](#)” object and connect it to the “[gemwin](#)” object; or create two messages containing “0” and “1”.

It is possible to create and render a GEM window larger than your screen size.



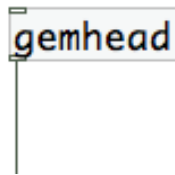
```
color 0 0 0, dimen 500 500, create, 1  
destroy, 0  
gemwin
```

The order in which the messages to the “gemwin” are clicked is important. Always click the “create” message before rendering. The GEM window’s size cannot be changed once it has been created.

The default colour of the GEM window is black, but can be changed by sending a “color n n n” message to the “gemwin” object, where “n n n” are red, green and blue (RGB) values from 0 to 1.

To simplify the cluster, all messages sent to the “gemwin” object can be chained together in one message. Each message is separated by a comma, and is processed from left to right. So instead of lots of separate messages, there are only two going to the “gemwin” object:

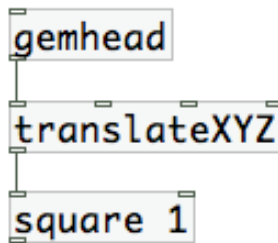
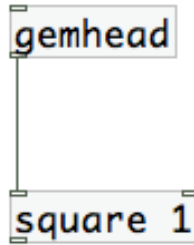
“dimen x y, color n n n, create, 1” and “destroy, 0”.



3.2. Objects Within GEM

In order for GEM to display graphics, a framework chain has to be created. This framework chain will be very similar for anything displayed within the GEM window. Starting at the top of the chain is a “gemhead” object. This denotes the rendering layer of the graphic. The default layer value for a “gemhead” object is 50, and this can be changed by adding the desired layer value: “gemhead n”. When multiple “gemhead n” objects are created the layering process renders positive lower values first, meaning higher render value graphics will sit on top of lower render value graphics. Negative value graphics are rendered after all positive value graphics, and lower negative value graphics will sit on top of higher negative value graphics.

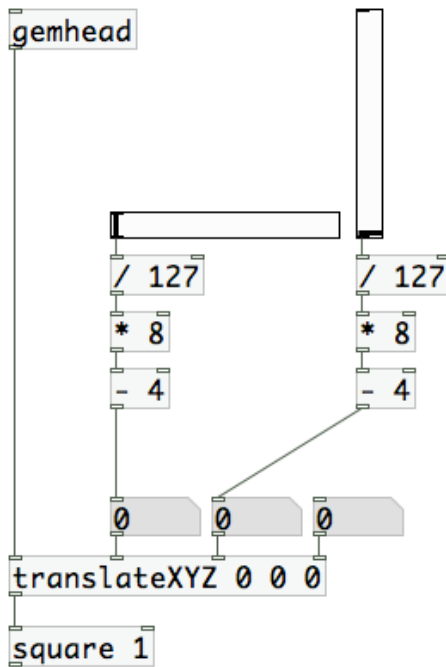
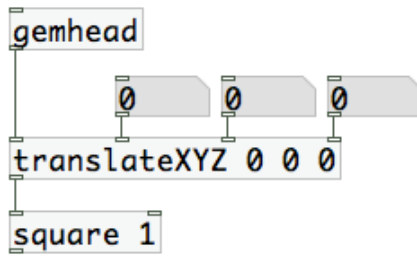
E.g. ‘1’ before ‘50’, before ‘-10’, before ‘-23’



At the bottom of the chain is the graphic itself. This can be a number of objects, normally 2D and 3D shapes. In this example a square will be used. To render a square in the GEM window create a “[square n](#)” object. The “n” denotes the size of the square, also adjustable by the second inlet.

The size and position of the shapes within a GEM window are not measured in pixels. The centre of the GEM window is always (0,0), and the y value always ranges from -4 to +4; the x value changes depending on the aspect ratio of the GEM window. When a graphic is created, it will default to (0,0).

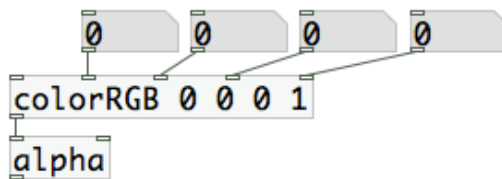
To change the position of the graphic add a “[translateXYZ](#)” object in between the “[gemhead](#)” and “[square](#)” objects.



The “[translateXYZ](#)” object takes three numeric arguments. The first two correspond to the x y position in the GEM window, and the third, z, is where it is placed in the depth of field. These can be set within the object, or altered using the corresponding inlets.

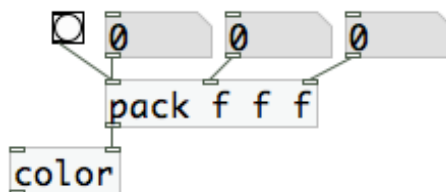
Create one “[hslider](#)” object, and one “[vslider](#)” object, and under each put a “[/ 127](#)” object. Then a “[* 8](#)” object, followed by a “[- 4](#)” object. Now connect these to the second and third inlets. Also connect the outlets from the “[- 4](#)” objects to “[number](#)” boxes. This gives the GEM window coordinate for the centre of the graphic. Moving the sliders dynamically changes the “[number](#)” box values.

color colorRGB

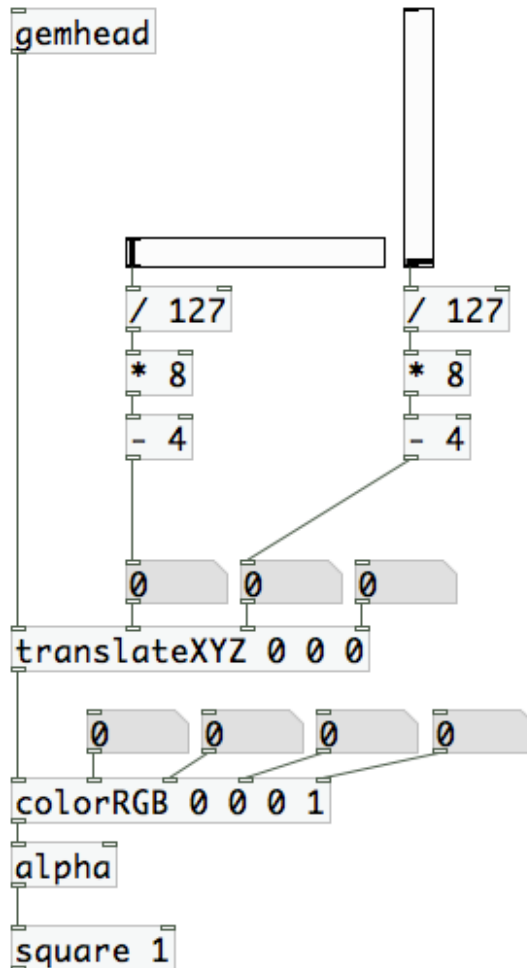


Changing the colour of an object works in a similar way to changing the colour of the GEM window. There are two objects that can change the colour of a GEM window object, “colorRGB” and “color”, but they work in slightly different ways. Both use RGB values (0 to 1), but they differ in the way they receive arguments.

The “colorRGB” object has five inlets. The first is for the gemlist, second, third and fourth are arguments for the RGB values, and the fifth is the alpha content. The alpha content is how much transparency the GEM object has. Again it ranges between 0 and 1, with 0 being completely transparent, and 1 being opaque. The alpha content will not be sent to the final image unless an “alpha” object is included.



The “color” object only has two inlets, and only receives packed data from the “pack” object. The “pack” object takes many streams of data and compresses them down to one data list. It can contain as many arguments as necessary, however these arguments have to correspond to the type of data being packed. There are three types of data that can be packed: float, symbol and pointer. The main two are float and symbol. A float ‘f’ refers to a number value, and a symbol ‘s’ refers to a word. A pointer is used when dealing with scalars. To send “color” a message, connect a “pack f f f” to the second inlet. The “pack f f f” object now has three inputs, and these correspond to the RGB values of “color”, again ranging between 0 and 1. The pack object will only send data when it has received a bang on the first inlet.



On the whole, the “colorRGB” object gives greater control over the colour of GEM window objects. It can be placed anywhere between the “gemhead” and “square” objects.

`pix_image`

`pix_video`

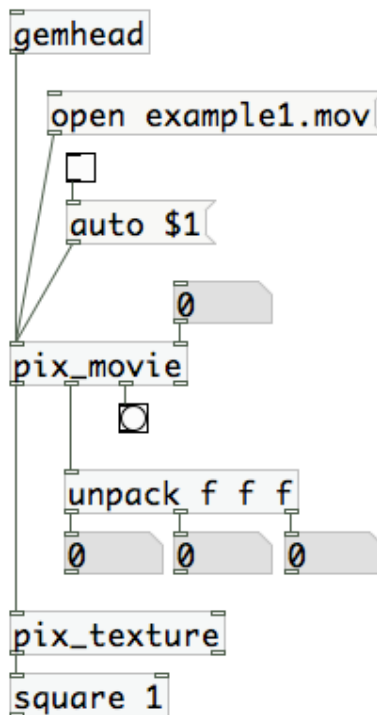
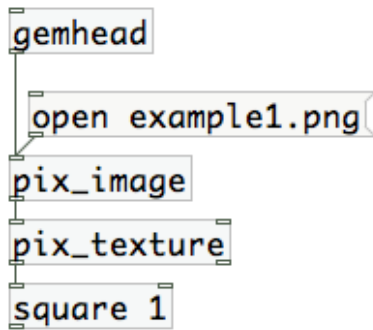
`pix_movie`

`pix_texture`

3.3. GEM Object Images

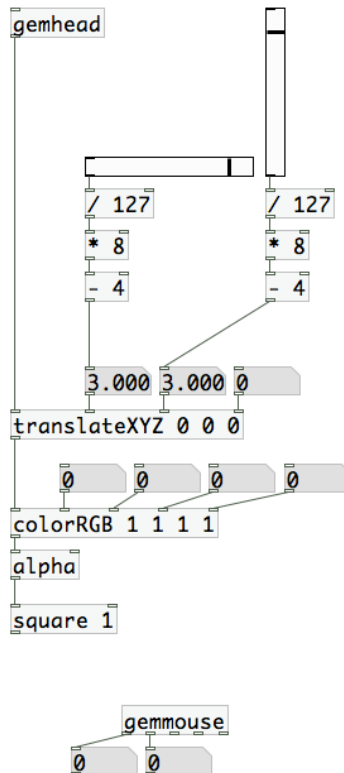
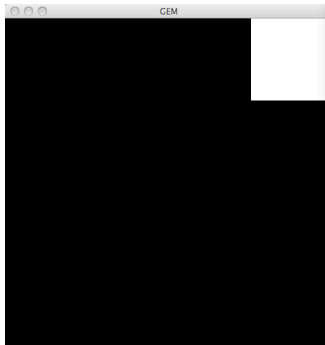
It is possible to render images, movies, and even webcam onto GEM objects. The objects that deal with imaging all start with “[pix_](#)”. The “[pix_image](#)” object will enable the use of still image files, the “[pix_video](#)” object enables the use of a webcam, and the “[pix_movie](#)” object enables the use of movie files (no sound). All of these require a “[pix_texture](#)” object to be included in the gemlist. The gemlist is always sent and received at the first inlet and outlet of “[pix_](#)” objects. Generally, the image creation objects (“[pix_image](#)”, “[pix_video](#)”, “[pix_movie](#)”) come straight after the “[gemhead](#)” object, and the “[pix_texture](#)” comes prior to shape object.

Both “[pix_image](#)” and “[pix_movie](#)” objects need to be sent a message telling them what to open.



Similar to loading audio files, the images and movie files need to be located in the same folder the patch is saved in. To open an image, use the “*pix_image*” object, and send an “*open filename*” message to the first inlet. The filename must include the file type as well.

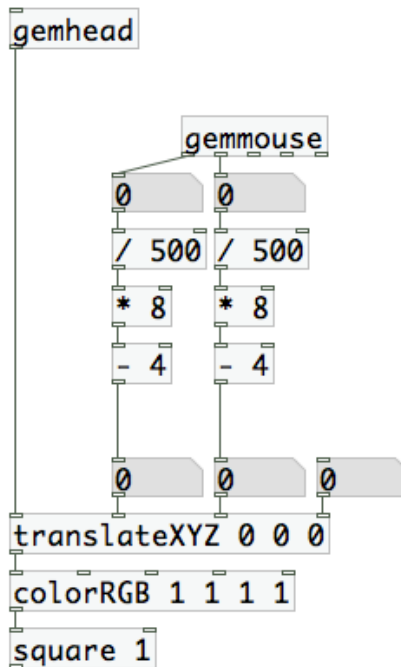
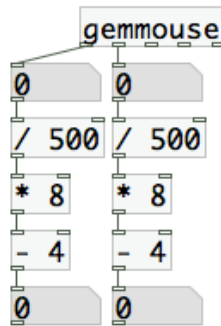
Opening a movie file is similar again. Send an “*open filename.filetype*” message to the first inlet to load the movie. Attach a “*toggle*” object to an “*auto \$1*” message to the first inlet to control the start/stop of playback. The second inlet sets the frame number using a “*number*” box. The second outlet unpacks giving three float values using an “*unpack f f f*” object. First is number of frames. Second and third are width and height respectively. The third outlet bangs when the video has ended.



3.4. Zoning the GEM Window

To trigger actions based on the mouse's position in the GEM window, a cluster must be created so that Pd knows where the mouse is, and what to do when it is there. Take the final example from chapter 3.2. and move the square using the sliders so it sits entirely in the top right corner of the GEM window. This square is the first trigger area.

To see where the mouse is positioned in the GEM window, a “[gemmouse](#)” object is utilised. The first two outlets of the “[gemmouse](#)” object are the x and y positions of the mouse in the GEM window, measured in pixels. Connecting a “[number](#)” box to each of the outputs will show the position. Notice the number values only change when the GEM window is the active window.

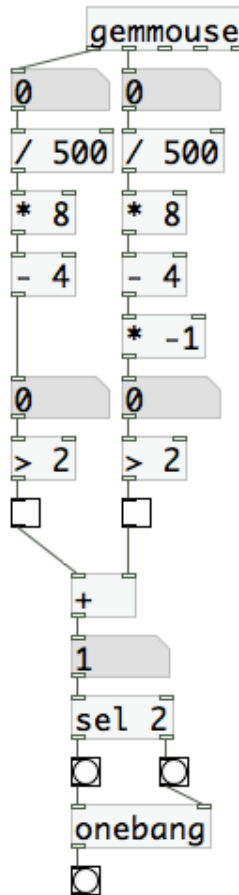


Because the “[gemmouse](#)” object outputs its positioning values in pixels, the scale must be changed to the same scale used by the GEM objects. This is similar to changing the scale of sliders to the GEM object scale. The same sequence of mathematical operations will be used, except instead of dividing by 127 (for sliders), the range of the GEM window will be used instead. The default size of the GEM window is 500 x 500 pixels, so the range will be 500. This can now be connected to a gemlist and be used to control the position of a GEM object.

Notice what happens on the y-axis. The object’s position is inverted to the mouse’s position. This can easily be fixed by adding a “[* -1](#)” object after the “[- 4](#)” object on the y-axis string.

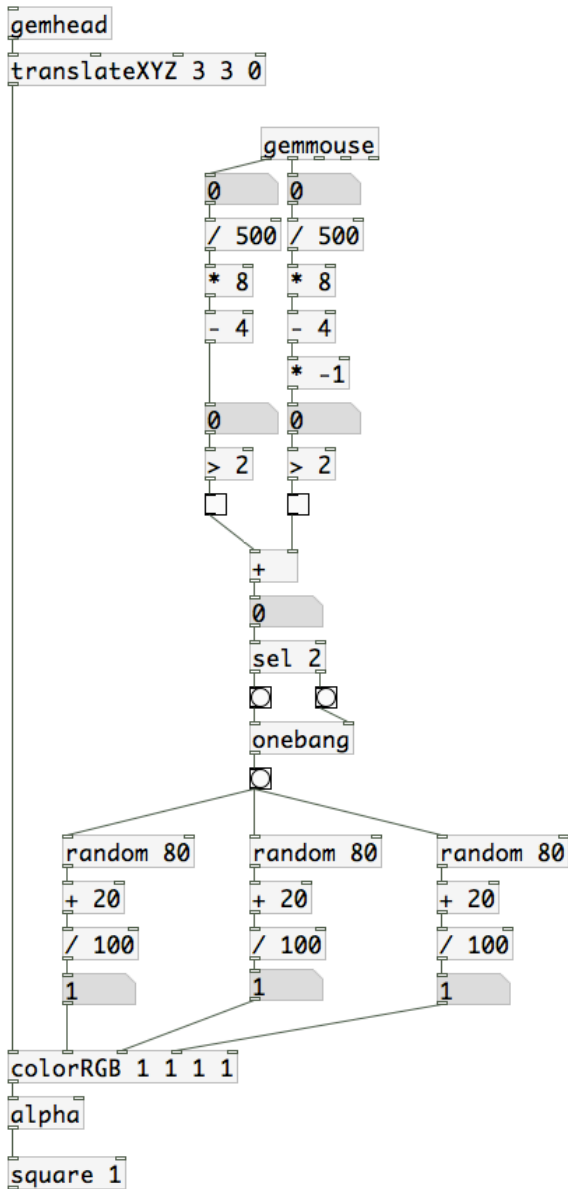
This is an easy fix because any positive number multiplied by a negative number becomes negative, and any negative number multiplied by another negative number becomes positive.

To zone the area of the square in the GEM window, its boundaries need to be found using the new “[gemmouse](#)” scale. Move the mouse to the left edge of the square, and then take note of what happens to the x-axis number when the mouse moves further into the square. It increases. Knowing this we can use the “[>](#)” object, the mathematical symbol for ‘greater than’. This object works by comparing the value received at the first inlet to its initial argument, or the value received at the second inlet. It will output a ‘1’ if the first inlet value is greater, and a 0 if not.

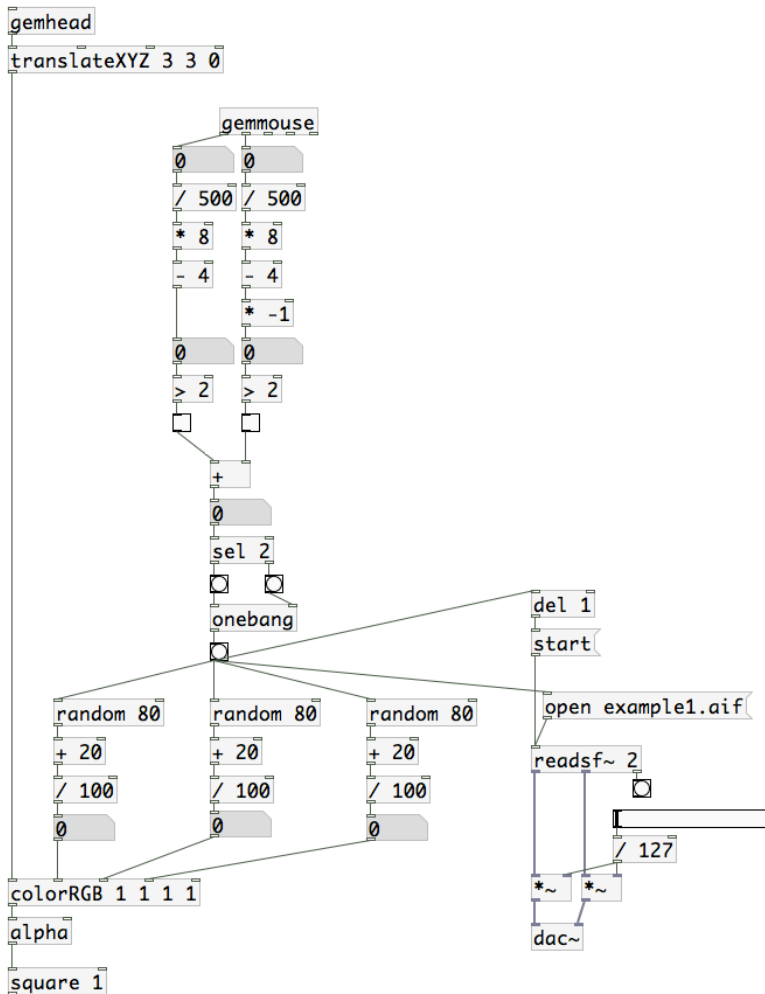


Using this object we can use the values found at the edges of the square, then convert the outputs to a “bang”.

Adding the outputs of the “>” objects, and then using the “sel” object to bang when both “>” objects output a 1, means a bang will be sent every time the mouse enters, and moves within, the square area. To eliminate the continuous stream of bangs, the “onebang” object is used. The “onebang” object only outputs a bang received at the first inlet, if it has previously received a bang at the second inlet. Using the second outlet of the “sel” object as the priming bang, “onebang” will only bang once on entry to the square area, and not again until the mouse has left and re-entered the area.



Put the positioning values for the square into the “[translateXYZ](#)” object as its default arguments, “[translateXYZ 3 3 0](#)”. Connect the zoning string to three “[random 80](#)” objects. These three objects will be used to randomise the colour of the square every time the mouse enters this zone. As stated before the “[colorRGB](#)” object takes red, green and blue values between 0 and 1. To make sure there is always some visible colour (not black), the range of values will go from 0.2 to 1, a difference of 0.8. The random object only produces integers so some re-ranging is needed. A “[random 80](#)” will randomise numbers between 0 and 79. Adding 20 will produce random numbers between 20 and 99. This can then be divided by 100 to give a range of 0.2 to 0.99.

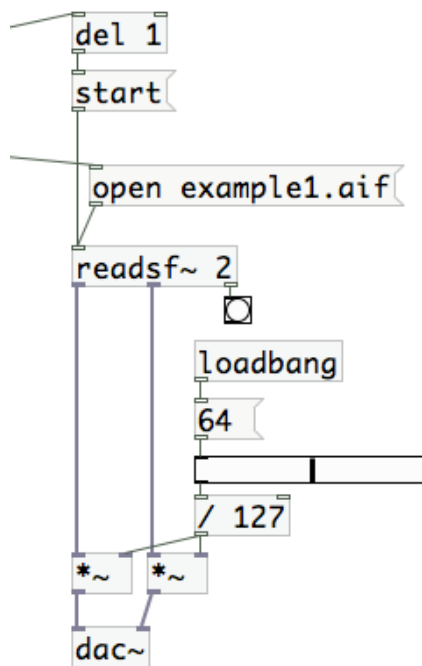


Exercise: Three

The objective of this exercise is to create nine trigger zones in the GEM window, controlled with the mouse.

The main patching lines have already been written in previous sections, but they will need to be duplicated and have some values altered to ensure the triggers work properly.

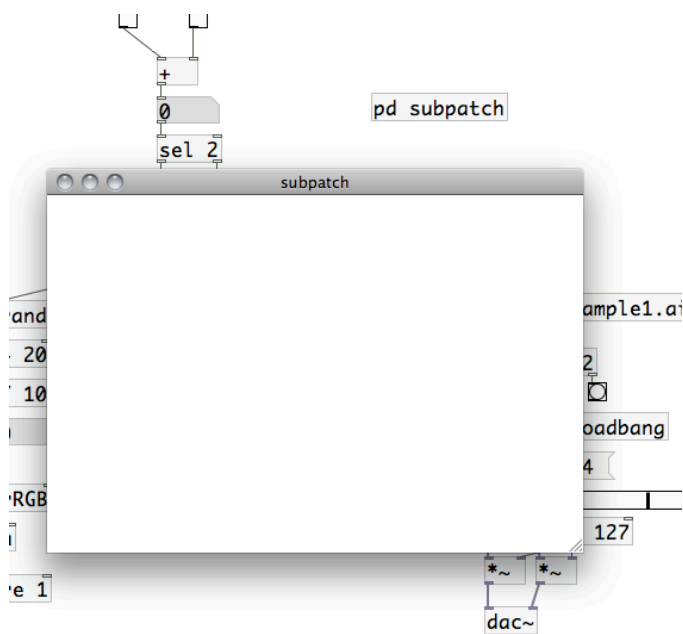
To begin with, duplicate or re-create the audio file playback cluster from section 2.4. and position it next to the zoning patch from the previous section. Add a “del 1” object to the “start” message to ensure the file is loaded before it plays. Connect the “bang” from underneath the “onebang” object to the “del 1” object, and “start” message.



At the moment the volume of the audio file will default to 0. It can become very tedious to turn all the volumes up individually, so the volumes will be sent a default value when the patch is first opened using an object called “loadbang”.

The “loadbang” object outputs a ‘bang’ when the patch is first opened, and can send default values when combined with a message. It is a very useful object.

To send a default value of half volume to the audio file playback volume slider, create a “loadbang” object, connect it to a “64” message, and then connect that to the slider. The “loadbang” will not ‘bang’ when it is created, so make sure to click the “64” message to send the data to the slider.

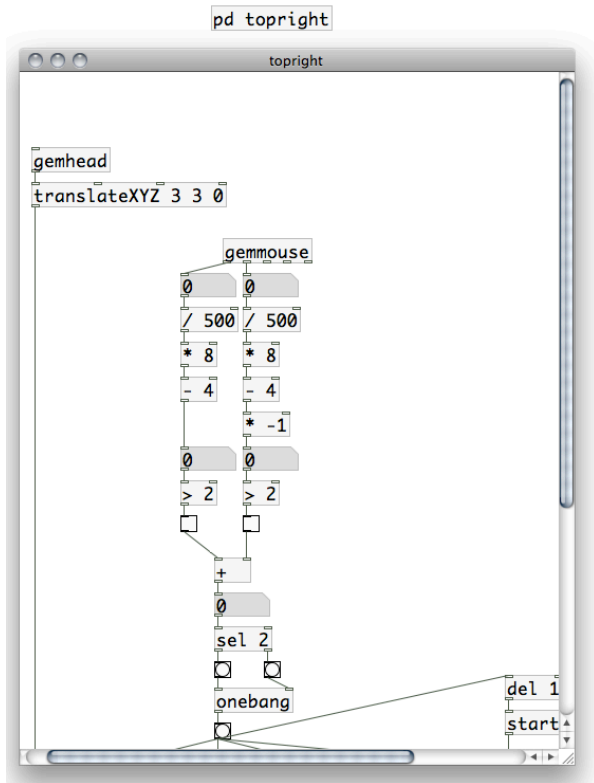


Patches can become large, and subsequently very messy if not managed properly. An easy way to keep patches tidy is to create sub-patches. This is a way of opening another canvas inside a patch. Sub-patches are created from objects by using a ‘pd’ prefix to the desired sub-patch name, for example, “pd subpatch”. When created, the new sub-patch canvas will become the active window, with the sub-patch name as its window name. To re-open a sub-patch, switch to ‘Run’ mode, and click the sub-patch object.

```

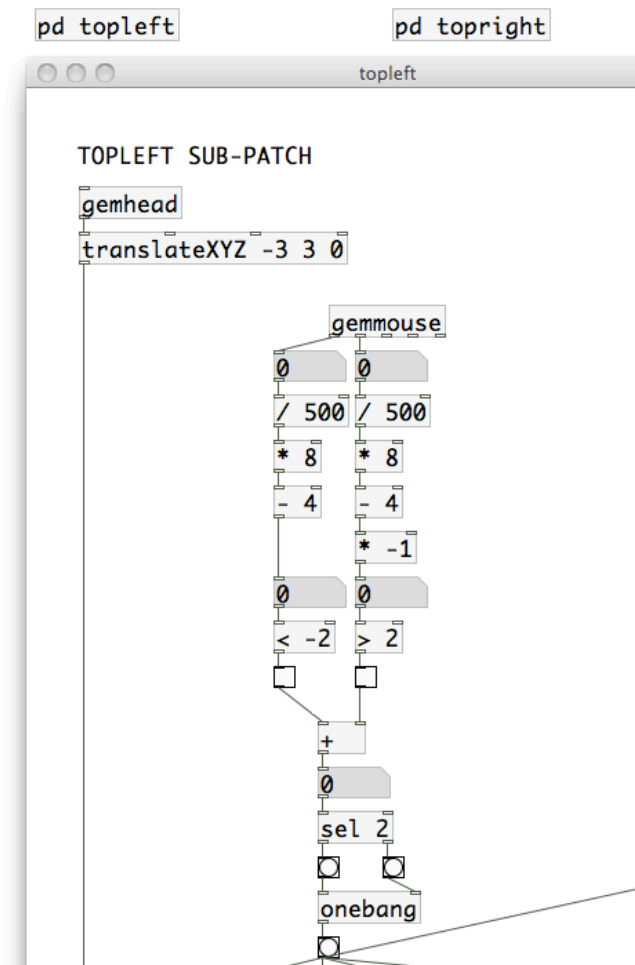
dimen 500 500, color 0 0 0, create, 1
destroy, 0 reset
gemwin

```



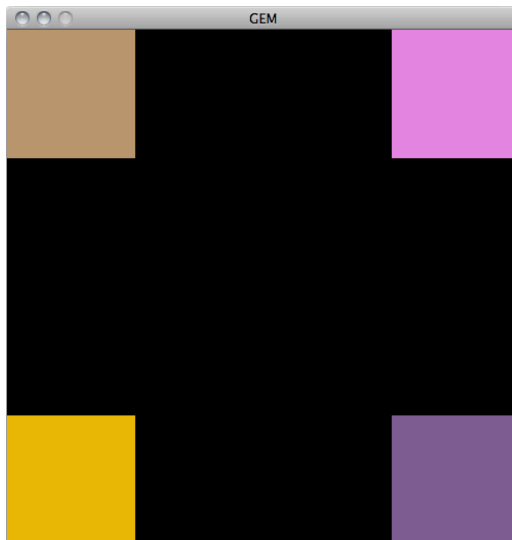
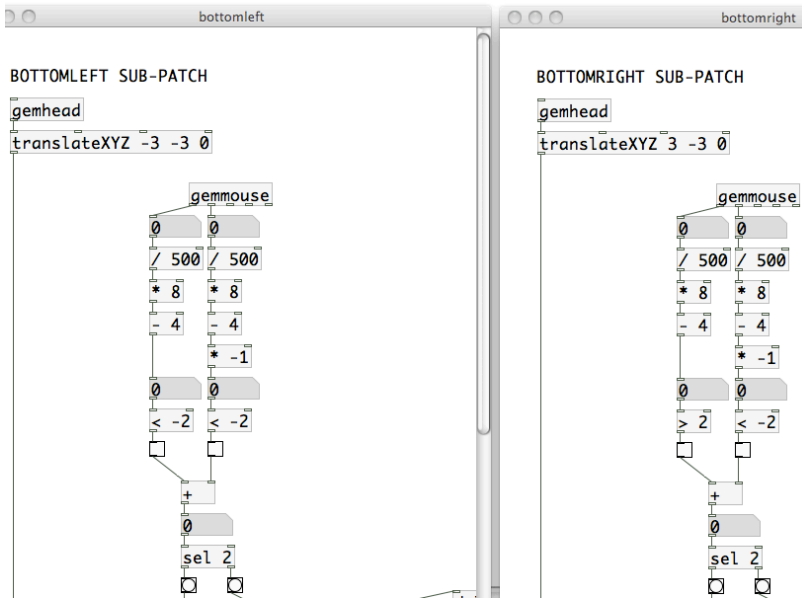
This exercise contains several instances of the zoning cluster, and file player strings, so putting each of them into a sub-patch keeps them organised. Simply copy the section of the patch, paste into the sub-patch canvas, then delete the original section. **Make sure the sub-patch object is not deleted.**

Rename “`pd subpatch`” to “`pd topright`”. The name can be changed at any time without deleting the contents of the sub-patch. There can also be numerous sub-patches with the same name with different contents. Sub-patch objects can also be duplicated without losing any data, and are great way of organising the workspace, and encapsulating streams of data.



Duplicate the “pd topright” sub-patch, and rename the duplicate “pd topleft”. This is the first step in creating another trigger area. There are several values that need to be altered to ensure the trigger area is situated in the top left of the GEM window. The first is the GEM object. This is moved using the “translateXYZ” object, and is altered from “translateXYZ 3 3 0” to “translateXYZ -3 3 0”. Only the x value has changed, and now the objects sit equidistant from the centre of the GEM window.

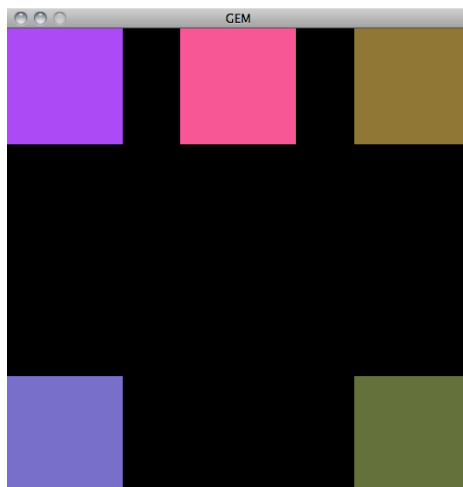
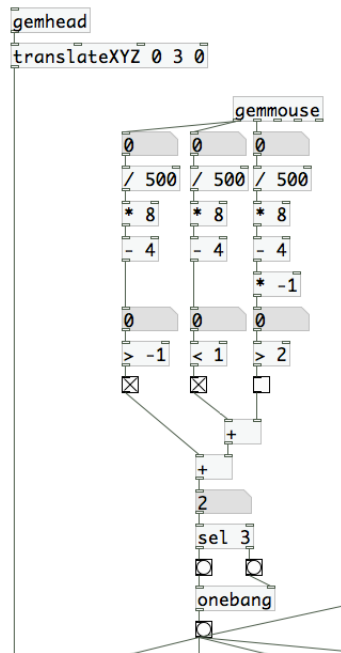
However the zoning area is still the top right hand corner for both sub-patches. To change the zoning area, the “>” object needs to be changed. Instead of “> 2” (greater than two), it now needs to be “< -2” (less than minus two). Change the “pd topleft” audio file so it plays a different file.



Duplicate both “pd topright”, and “pd topleft” sub-patches, and rename them “pd bottomright” and “pd bottomleft” respectively. Alter the values within the new sub-patches so that four trigger zones will now be visible and triggering in each of the four corners of the GEM window. This time, only the y-axis values need to be altered, along with the change in audio file.

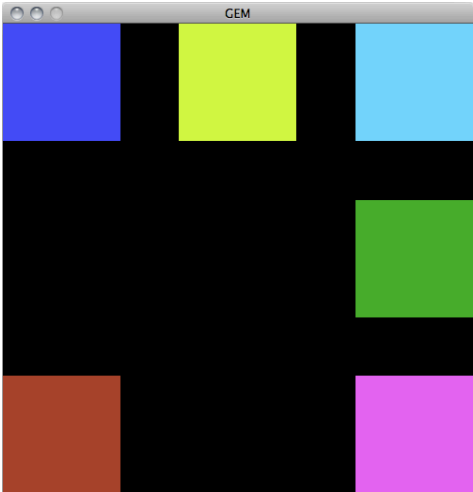
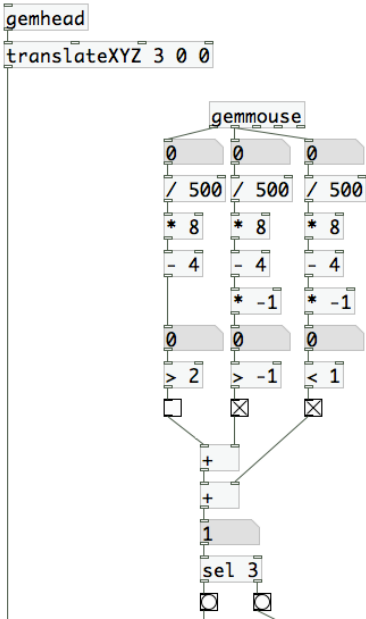
Adding more zones between the four corners requires extra coding because there will be more exposed boundaries for each square. The ‘middle top’ and ‘middle bottom’ areas will need an extra x-axis argument, whereas ‘middle right’ and ‘middle left’ require an extra y-axis argument.

MIDDLTOP SUB-PATCH



These can be found in the same way as before. Start by duplicating the “pd topright” sub-patch, and rename it “pd middletop”. The next step is moving the square to the middle of the top edge of the GEM window by changing its position to “translateXYZ 0 3 0”. There needs to be an extra argument for the x position of the mouse because there are two exposed edges along the x-axis. This means there needs to be a “> -1” object, greater than, and a “< 1” object, less than, contributing to the trigger. The three “toggle” objects all need to add together, and the “sel 2” changed to “sel 3” to make “onebang” object output a bang when all three toggles output 1.

MIDDLERIGHT SUB-PATCH

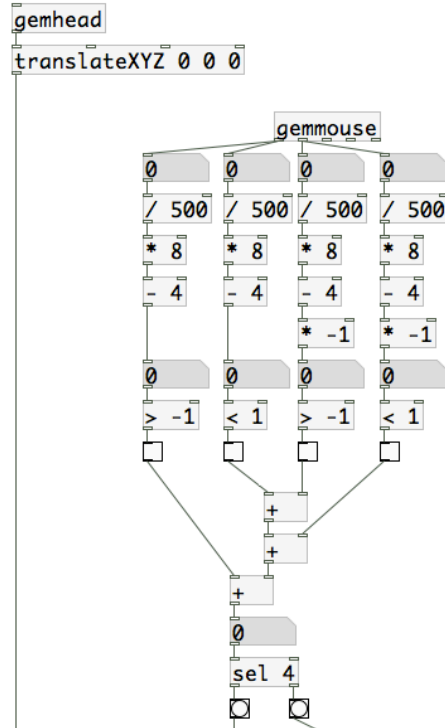


Adding zoned areas along the middle of the side edges is similar again, but this time there needs to be an extra y-axis argument.

Start by duplicating the “`pd topright`” sub-patch, and rename it “`pd middleright`”. As before, change the position of the square to “`translateXYZ 3 0 0`” so it sits on the middle of the right hand side. Then as before, add an extra argument but this time for the y-axis. Change the y-axis values to “`> -1`” and “`< 1`” objects. Add all the “`toggle`” outputs together and change the “`sel 2`” object to a “`sel 3`”.

Repeat this process for the middle of the bottom edge and middle of the left edge, giving eight trigger zones all together.

MIDDLE SUB-PATCH



Adding the final trigger zone in the centre of the GEM window requires a further argument because there are four exposed boundaries. There must be two x-axis arguments, and two y-axis arguments.

Duplicate the “pd middleright” object, and rename it “pd middle”. Then change the squares position to “translateXYZ 0 0 0”, so it sits in the centre. Then add an extra x-axis argument, and alter the values so for both the x and y-axis there are “> -1”, “< 1” objects. Then add all toggle together, and change “sel 3” to “sel 4”.



Once all nine trigger zones have been created, this patch can generate random colours and trigger audio files, all controlled with the mouse.

Assignment: Two

Using the “[pix_movie](#)” object, create a patch that can control the frame position and brightness of a movie, with the x and y position of the mouse in the GEM window.

Assignment: Three

Using the zoning technique, create a four trigger grid. Assign sounds to each zone, and place an associated picture with each sound.

Chapter Four: Audio-Visuals

4.1. Abstractions

This chapter utilises abstractions as objects. Abstractions are not part of the original Pd program content, but have been created by Pd users. Abstractions can be downloaded from the internet, and quickly implemented within Pd.

There are a few general points worth noting in order for abstractions to work successfully:

- As with audio and movie files, abstractions have to be saved within the same folder as the patch utilising them, otherwise Pd will not recognise them as objects.

- When naming the abstraction, other object names within Pd have to be ruled out ensuring there is no name duplication. If you are unsure, type the desired name into an object box to check.
- If there is name duplication, the abstraction will be ignored in favour of the original object. There is no error message displayed in the 'Pd window'.
- Unlike media files, abstractions do not require the inclusion of the filetype extensions when creating abstraction objects on the canvas. If the filetype is included, Pd will see it as an unknown object and produce an error message in the 'Pd window'.

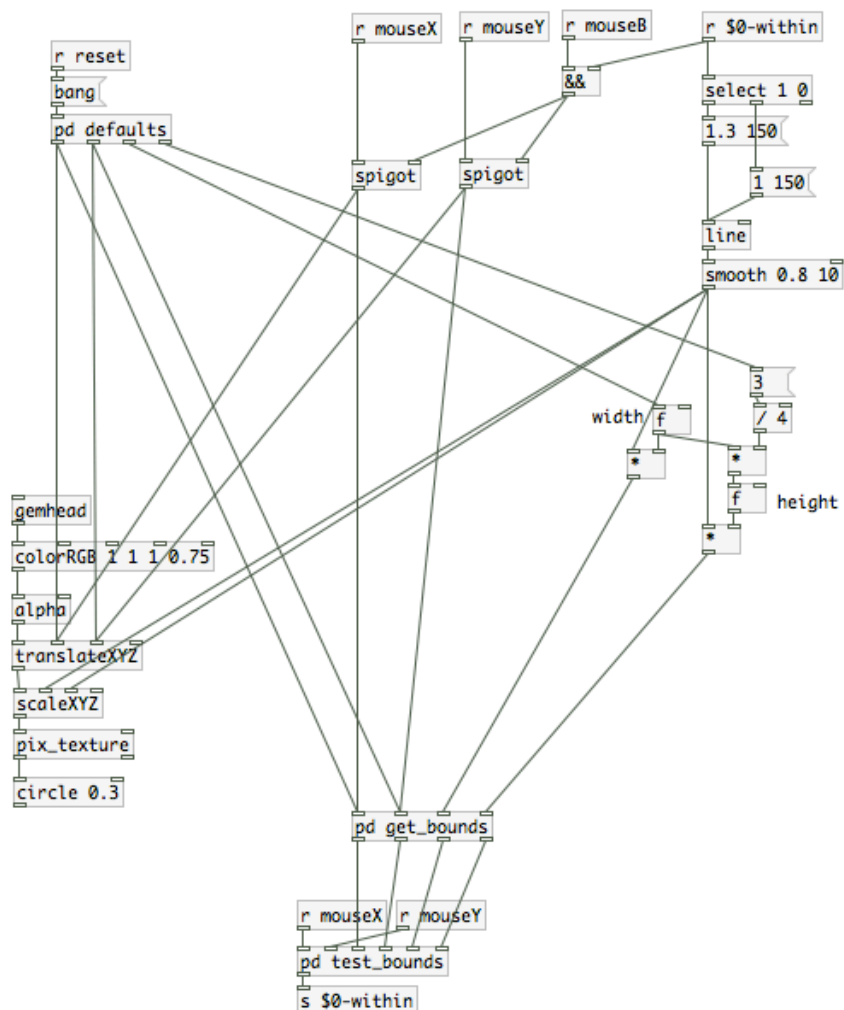
Sometimes there will be scenarios when multiple instances of one abstraction are desired. If any part of any instance of the abstraction content is altered (even repositioning a box), every other instance will be changed when the altered abstraction is saved.

Always ‘save as’ and rename an altered abstraction in order to preserve the other instances.

Abstractions need to be saved individually – they are not saved when saving the main patch.

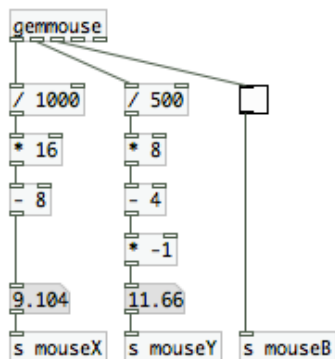
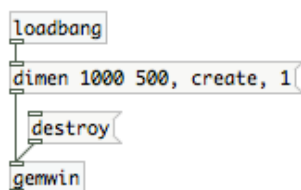
It is good practice to keep a separate library of all original abstractions, which is unconnected to the folder you are patching from.

video_object

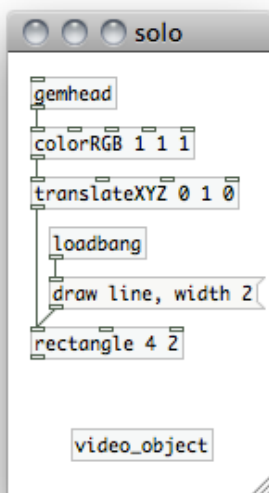


Exercise: Four

This exercise implements abstractions to manipulate and control audio from the GEM window to create an interactive music patch. The abstraction used is “[video_object](#)”, and is provided on the data disc. This abstraction creates a GEM object that can be moved using mouse in the GEM window. The main areas of “[video_object](#)” are the “[pd defaults](#)” located top left, the GEM object cluster located bottom left, and the “[receive](#)” objects located at the top. The “[pd default](#)” sub-patch contains default positions for this instance of the “[video_object](#)”. The GEM object is the visual representation of the “[video_object](#)”. The “[receive](#)” objects collect data about the mouse position, and decide whether the pointer is inside the GEM object, and if it has been repositioned.



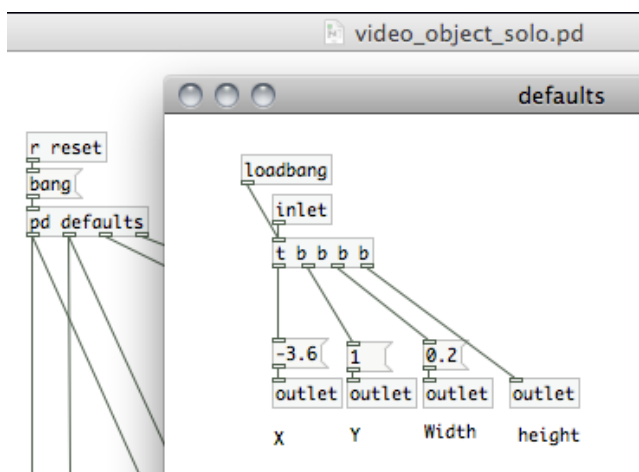
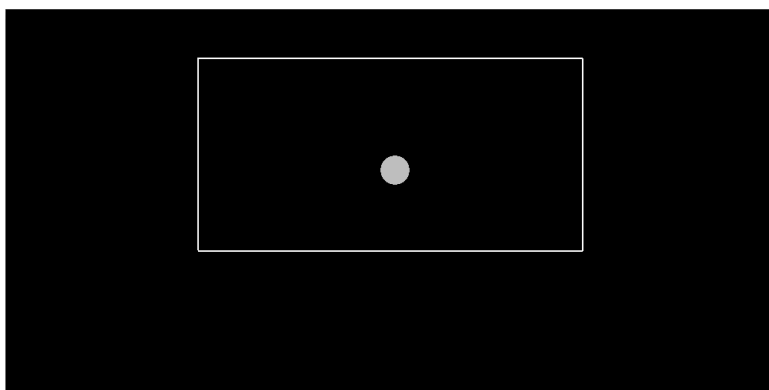
pd solo



To begin, create a GEM window in the same way as previous chapters, except change the size to “`dimen 1000 500`”. Then create the “`gemmouse`” object with the relevant scaling cluster. The x, y, and button data from the “`gemmouse`” object will be attached to “`send`” objects, “`s mouseX`”, “`s mouseY`” and “`s mouseB`” so this sequence only has to be written once.

The graphical user interface (GUI) for the solo control is the first segment to be developed for the audio-visual application. It uses the “`video_object`” to manipulate volume and pitch of a “`phasor~`” object, within a pre-defined scale.

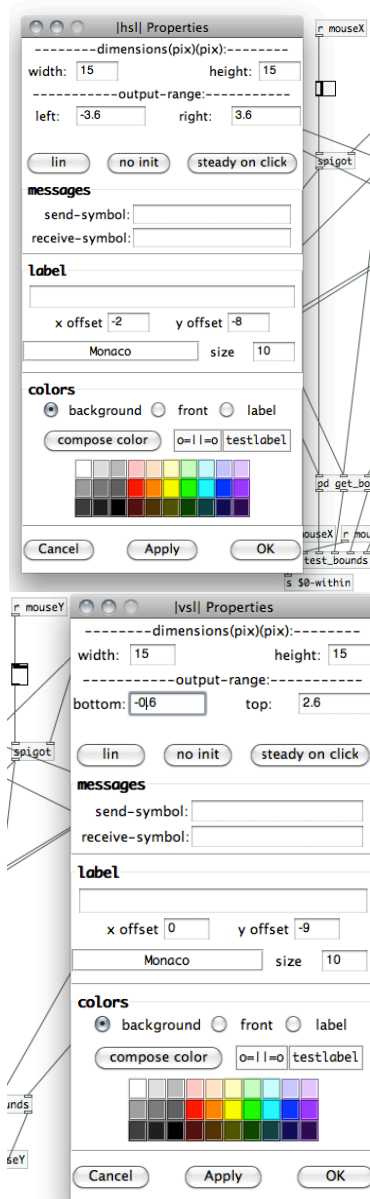
Creating a sub-patch for the “`video_object`” and other related boxes keeps things tidy.



The GEM window should now look like this. Pitch will be controlled by movement along the y-axis, and volume using the x-axis.

To preserve the content of “video_object”, a new abstraction must be created. To do this, open the “video_object” and **immediately** ‘save as’ “video_object_solo”. Then recreate the “video_object_solo” in place of “video_object”. This is now a new abstraction specifically for the solo control. At the moment it will look exactly the same as the “video_object”.

Now change the default position by opening the “pd defaults” sub-patch within “video_object_solo”, and change the x and y message values to “-3.6” and “1” respectively.



The solo GUI will be confined to a specified area.

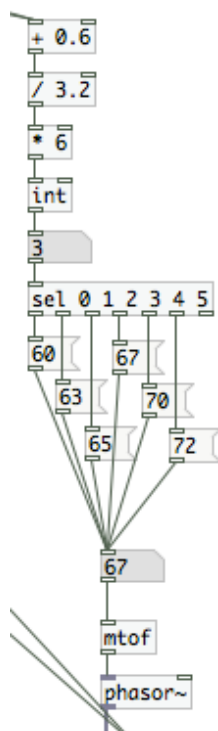
The easiest way to do this is to include sliders in the positioning data chain, after “[r mouseX](#)” but before “[spigot](#)”, within “[video_object_solo](#)”.

Sliders only output data that is within their range.

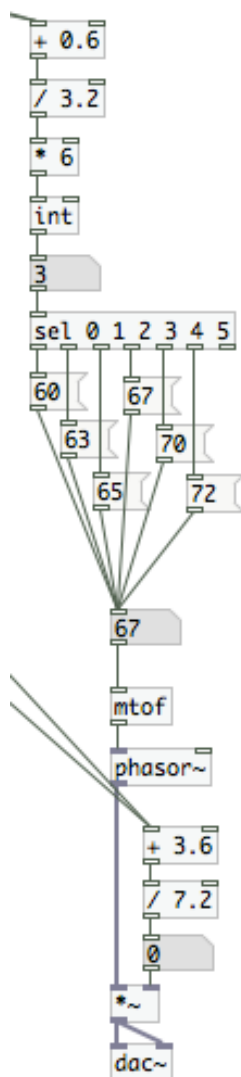
To alter the range, right click the slider, select properties, and change the left and right values.

The “[video_object_solo](#)” default x position is -3.6, and because the rectangle is centred, its right position will be 3.6. Enter these values into the properties of the “[hslider](#)” object, and also change its width to 15, reducing its size on the canvas.

Similarly add a “[vslider](#)” object after “[r mouseY](#)”, find a suitable top and bottom value by dragging the “[video_object_solo](#)” up and down and, keeping an eye on the “[gemmouse](#)” object output, enter the values in the “[vslider](#)” object properties.



The audio side of “[video_object_solo](#)” uses a pentatonic scale, with six notes including the octave, controlled by the y-axis. The y-axis data must be transformed to integers (whole numbers) before notes can be banged. A “[select](#)” object with six sequential arguments is required to ‘bang’ the notes. The range of the y-axis needs to be positive, so a “[+ 0.6](#)” object is used to shift the lowest value to 0. A “[/ 3.2](#)” object rescales the number to vary between 0 and 1. A “[* 6](#)” object now produces numbers between 0 and 5, and the “[int](#)” object is used to create integers. So now the range is composed of whole numbers from 0 to 5, six steps. The “[sel](#)” object coupled to messages containing the MIDI note values of the pentatonic scale feeds the “[mtof](#)” and “[phasor~](#)” objects.

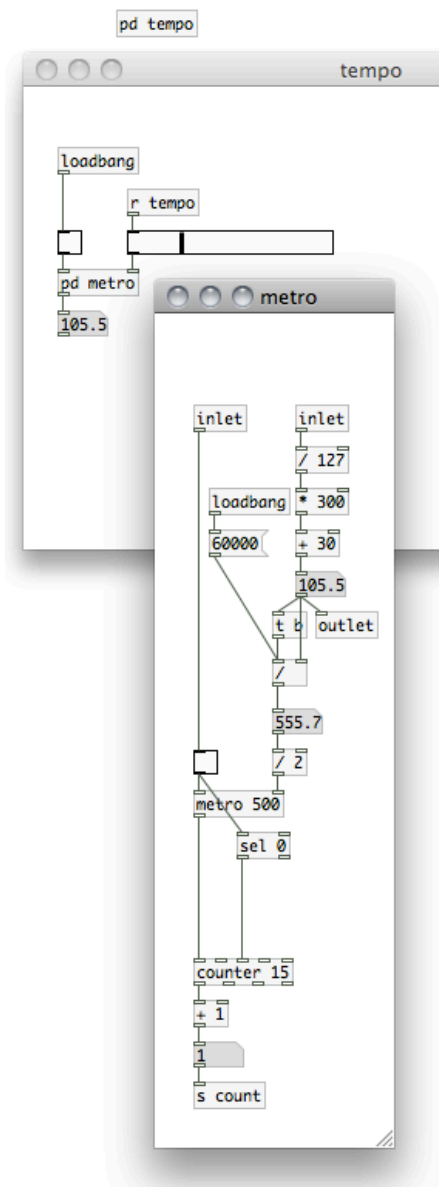


The x-axis controls the volume, so in the same way as the y-axis, shift the range so that the scaling takes place in the positive. The total range is 7.2, but needs to be between 0 and 1, so dividing it by itself, a “/ 7.2” object, achieves this.

The y-axis feed comes from both the y-axis “spigot”, and the second outlet of “pd defaults”. The x-axis feed comes from the x-axis “spigot”, and the first outlet of “pd defaults”.

A “spigot” object has two inlets. The left inlet only lets data pass to the outlet when a 1 is received at the right inlet.

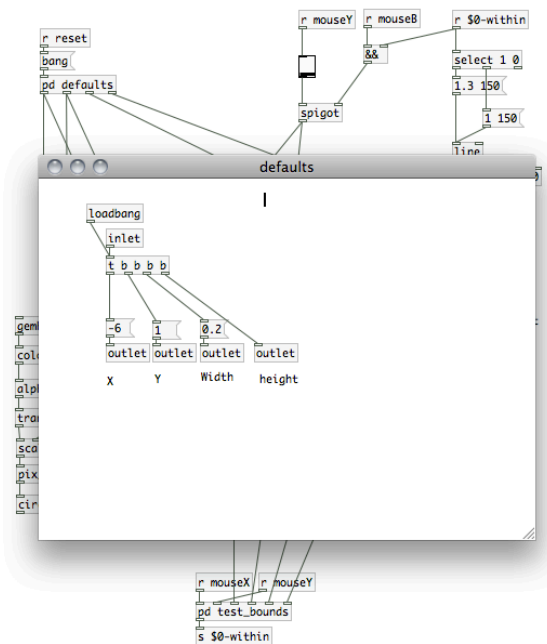
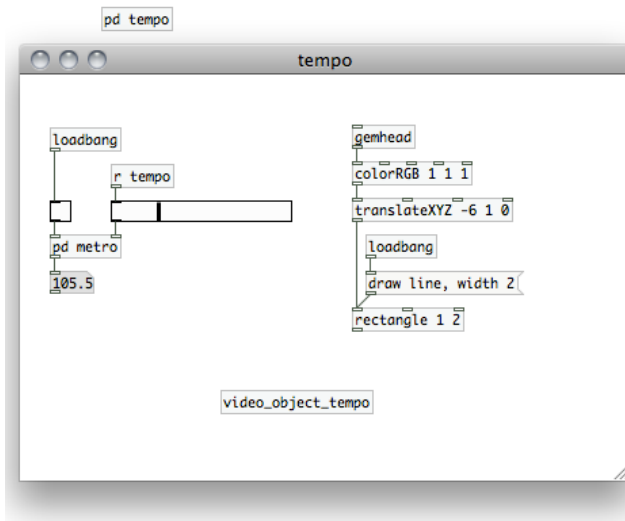
Turn the audio on, and drag the “video_object_solo” around the GEM window.



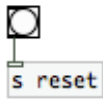
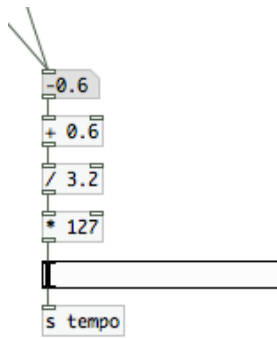
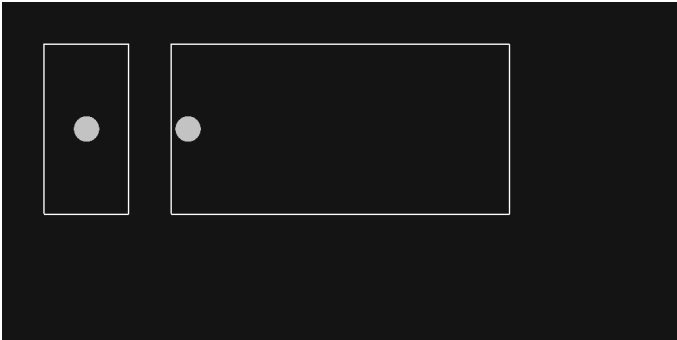
The next part is the first step for the integration of chords, drums, and tempo control. There is only one “metro” object keeping everything in time.

On the main canvas create a new sub-patch and call it “pd tempo”. Inside, create another sub-patch called “pd metro”. This is where a bpm (beats per minute) value changes into milliseconds, and a 16 beat count is produced. Create two “inlet” objects and the feed from the second “inlet” gets rescaled to go between 30 and 330 bpm.

The millisecond value is divided by 2 so that the “metro” bangs twice as fast as desired, giving the option of half beats. Connecting a “counter 15” and “+ 1” object gives a 16 beat count of 1 to 16.



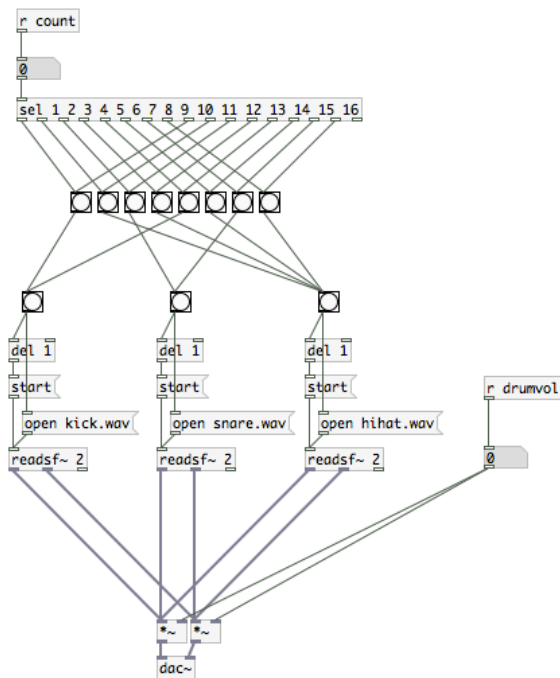
In the “pd tempo” sub-patch create a rectangle area for the tempo controls to be situated. Create a “video_object” as before and again before any changes are made, ‘save as’ “video_object_tempo”, and in the patch replace “video_object” with this. The tempo controls only require the y-axis, so the x-axis feed can be deleted from “video_object_tempo”, and because the y-axis is working on the same range as “video_object_solo”, implement the same slider-limiting cluster. The defaults will be set to “-6” and “1” for x and y-axis respectively so it defaults to the centre of the rectangle.



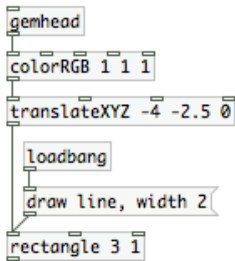
Now the GEM window should look like this. The “[video_object_tempo](#)” should only move along the y-axis, and both should stay inside their boxes.

To connect the “[video_object_tempo](#)” to the global “[metro](#)”, take a feed from the y-axis data stream, and the second outlet from “[pd defaults](#)” and connect them to the same “[number](#)” box. Rescale the data to go between 0 and 127. Then create a “[s tempo](#)” object. This is then received by the slider connected to the “[pd metro](#)” sub-patch.

All the “[video_object](#)” abstractions have a reset feature, which sets all values to the default values. Creating a “[bang](#)” object connected to “[s reset](#)” is used to reset to default values.



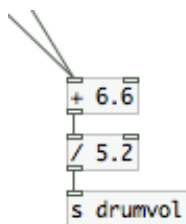
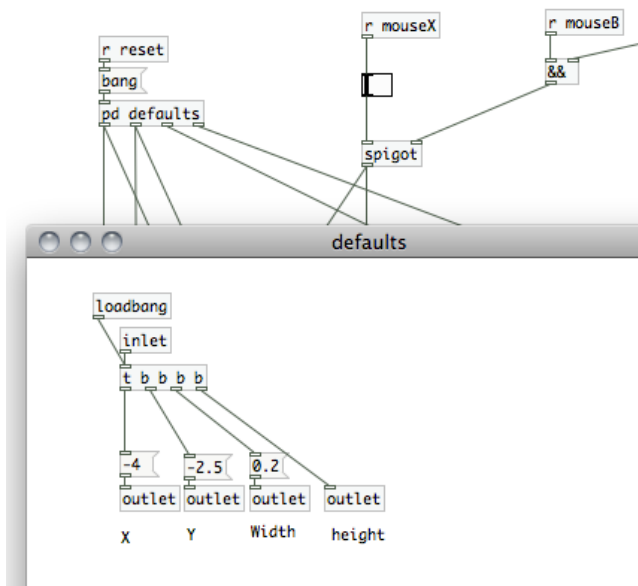
video_object_drumvol



Sequencing the drums is similar to triggering any other sound, except that the patterns and time intervals are more precise. Here is a simple two bar drum rhythm, with four beats per bar. The 16 steps from the “sel” object double up so that there is a total of eight beats per count cycle. Play around with the drum patterns by connecting different “sel” outlets to different “bang” objects until there is a satisfactory rhythm.

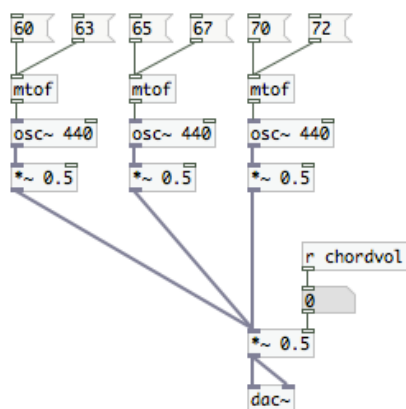
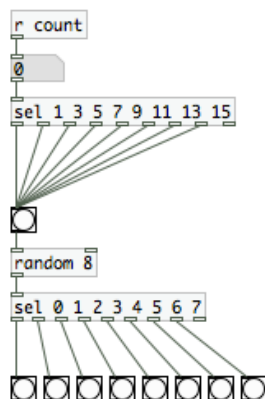
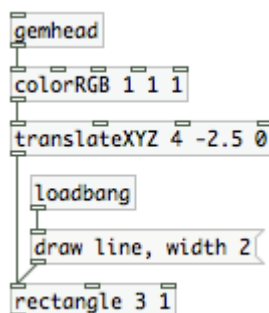
The volume of the drums is controlled from another “video_object”. As before, create a “video_object”, then immediately ‘save as’ “video_object_drumvol” and replace “video_object”.

Create an area for the “video_object_drumvol” to sit in.



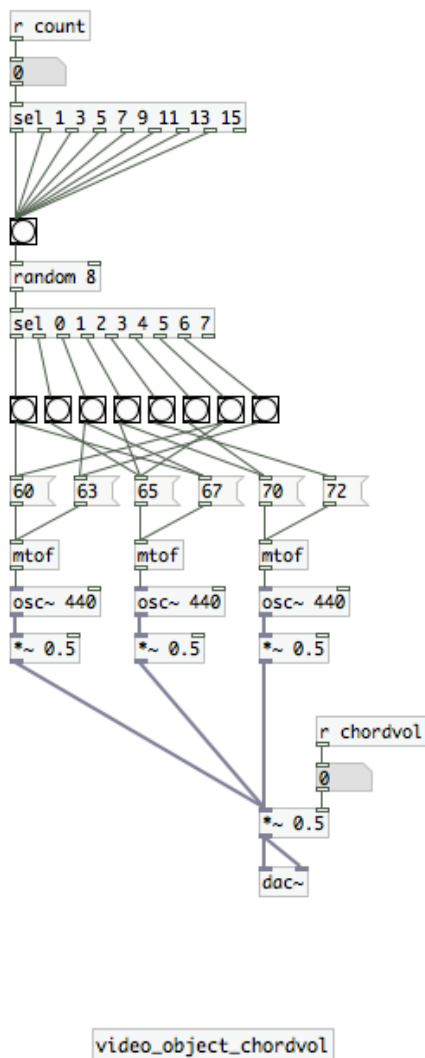
Change the “pd default” values of “video_object_drumvol” to be the same as the area’s position, where x is “-4”, and y is “-2.5”. The volume will default to half with these values. The “video_object_drumvol” abstraction does not require a y-axis data stream so that can be deleted straight away.

The x-axis needs to be limited. So like before, add a “hslider” object between the “r mouseX” and “spigot” objects and change the left and right properties to ‘-6.6’ and ‘-1.4’ respectively. Knowing volume works on a range from 0 to 1, the data must again be rescaled. Connect the x-axis data stream, and first outlet from “pd defaults” to a “+ 6.6” object so the lowest value becomes 0. Then divide by the highest value with a “/ 5.2” object, and connect to a “s drumvol” object.



The chords' volume control area will sit opposite the drum volume control area. Again start by creating a sub patch called “pd chords” on the main canvas. Then re-create the control area as shown, and reposition it.

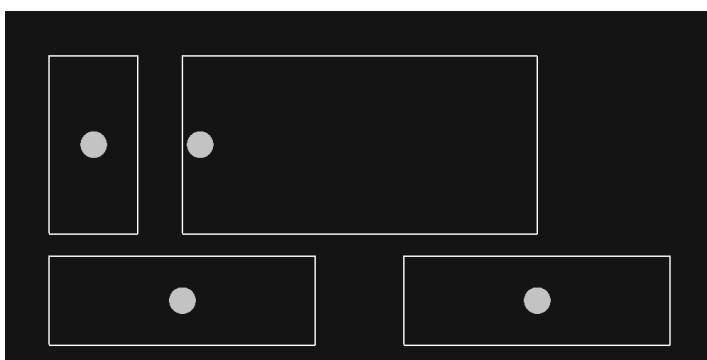
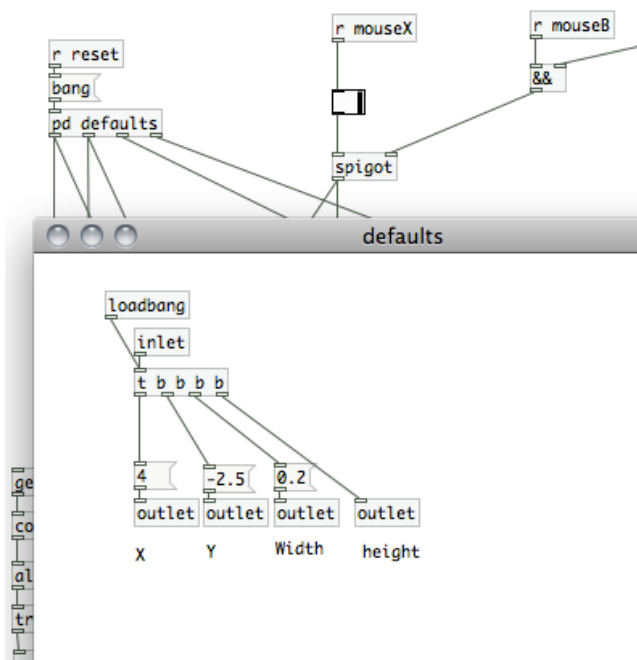
The chords will progress at half the speed of the drums. They will also be randomly selected. A ‘bang’ is needed every other beat. So, create the desired half speed progression using the “sel” object to bang a “random 8” object from every other number from the 16 beats. Then using another “sel” object, eight outputs become available and are used to control the chord selection. The chords come from the same pentatonic scale, and are grouped in pairs. This means only three notes can be played at one time.



Connect the “bang” objects to the number messages so that different messages are selected for each ‘bang’. There are no right or wrong ways to do this, just connect them up, and then listen to the chord by clicking one of the “bang” objects. The only thing to keep in mind is not connecting one “bang” to both messages in a pair.

Each pair has been set to half volume before going to the chord volume control to lessen the chance of distortion.

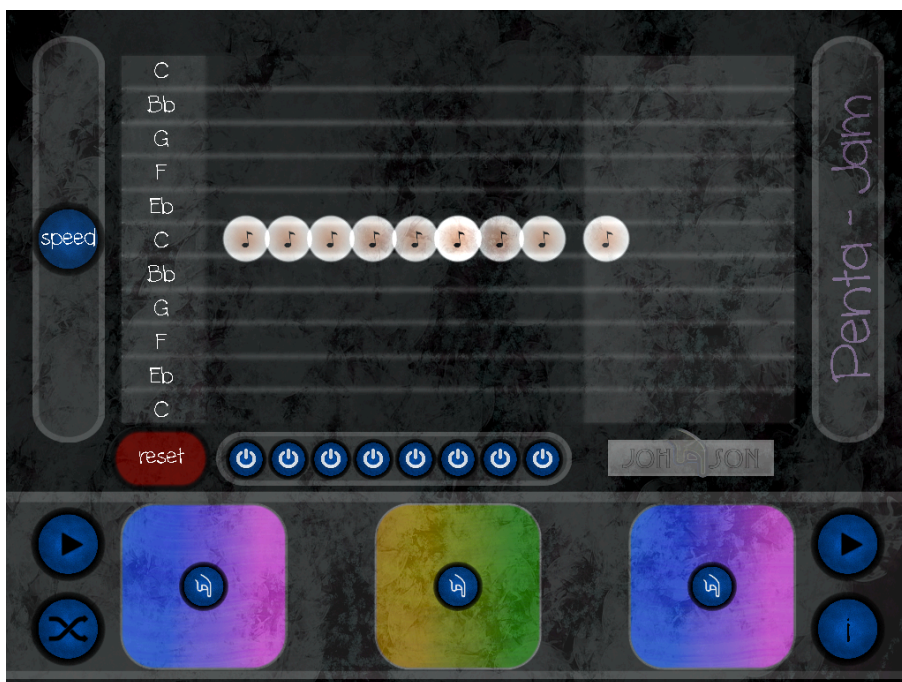
Now to create the “video_object_chordvol” abstraction. Again start with the “video_object” and ‘save as’ “video_object_chordvol”, and replace the “video_object” abstraction.



Similar to before, change the default message values to “4” and “-2.5” for x and y respectively so that they are the same as the chords’ volume control area. Also, delete the y-axis data stream, and implement the limiting slider for the x-axis.

To scale the x-axis data to control the chord volume, use a “- 1.4” object to bring the lowest value down to 0, and the a “/ 5.2” object to change the range to go between 0 and 1. Then connect this to a “s chordvol” object.

The main controls are now completed, and the GEM window should look like this when everything is set to its default position. Alter the colours and shapes to make your patch unique.



For a finished application using the same elements with a few added features, open “Penta-Jam” from the data disc.

Assignment: Four

Using the basic “video_object” abstraction, create a two-tone synth where the y-axis data from two “video_object” abstractions control the frequencies of two separate sound wave generators. Include a third “video_object” to control the global volume using the x-axis data. All objects should be able to roam freely within the GEM window.

Debrief: The End Of Your Pd Jump-Start

If you have been successful with the exercises and assignments you will be able to:

- **Develop your own patches.**

Practising your new-found skills will consolidate your learning, and you can get your pet projects underway.

- **Experiment and implement Pd with your own creative practices**

Try integrating Pd by creating instruments, interactive applications, games, visualisations or data decoders, resulting in digital harmony.

- **Converse intelligently with the wider Pd community.**

Using internet forums is a great way to further your knowledge of Pd. They are also the place to start sharing your patches and getting feedback.

Where To Go Next

Forums are a bountiful source of information about conferences, conventions, for meeting like-minded people in local patching circles, and for responding to calls for works.

Even if you don't submit patches to a conference or convention, they are well worth attending to discover the latest developments and applications.

To advance to expert level, there is no substitute for direct interaction with other Pd users, especially in practical workshop environments.

Final thoughts ...

EA's 'Spore' and 'Dark Spore' use Pd to control the sound effects, audio and musical elements of the games.

Could you do that now?

Acknowledgements

I am grateful to Simon Kilshaw, a leading lecturer in Creative Music Technology and Pd specialist, who guided and shaped my understanding of Pd.

The following resources have supplemented my learning.

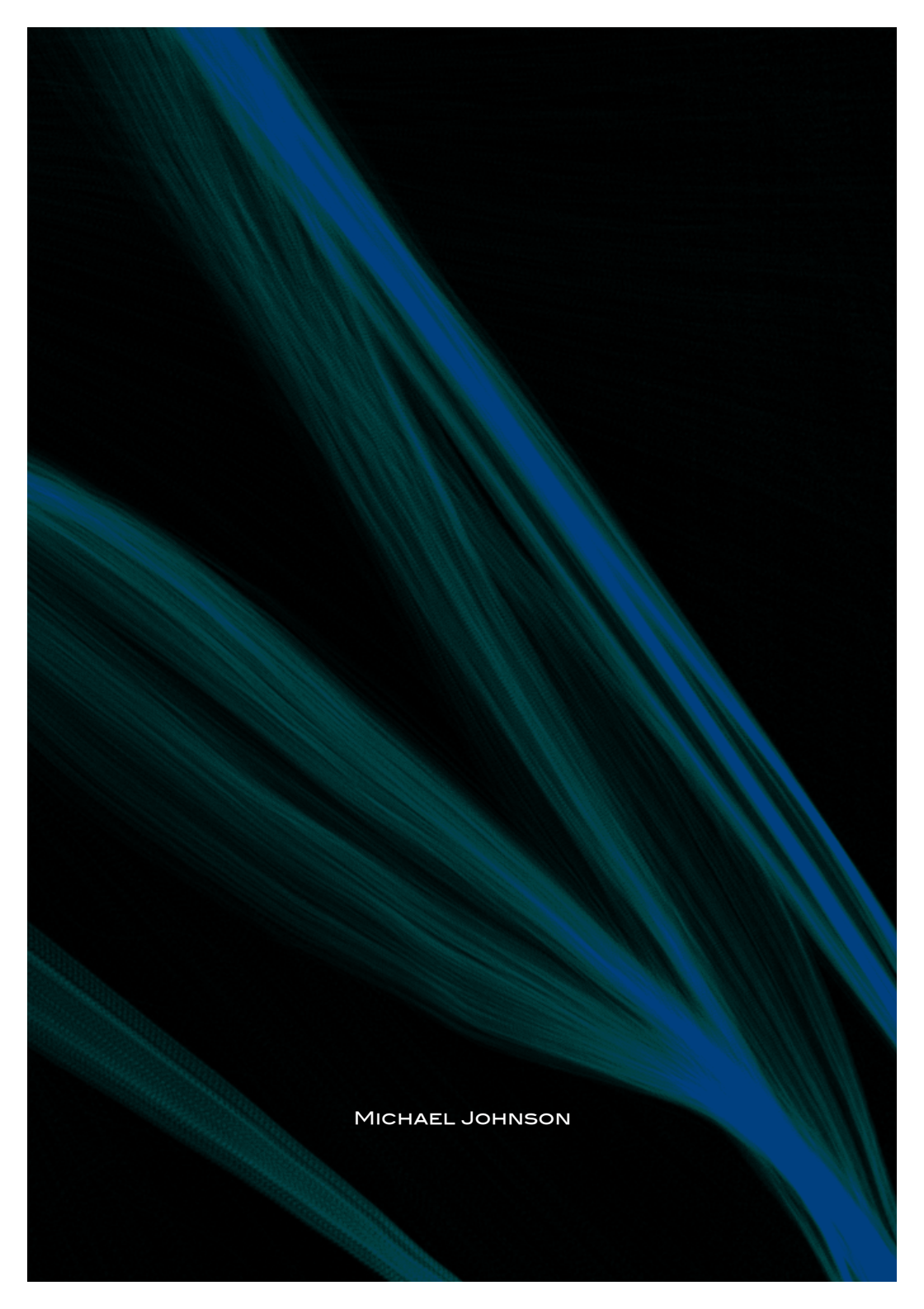
Pd Community Website
<<http://puredata.info/>>

‘Pure Data’ from Floss Manuals
<<http://www.flossmanuals.net/pure-data/>>

Kreidler, J., 2009: ‘*Loadbang: Programming Electronic Music in Pure Data*’

Abstracts from works presented at the first Pd convention, 2006: ‘*Bang: Pure Data (1. International PD-Convention Graz)*’

Abstracts from works presented at the Pd convention, Weimar - Berlin, 2011: ‘*Proceedings*’



MICHAEL JOHNSON