Programming Graphics I: Introduction to Generative Art

Joshua Davis

ASSIGNMENT

Your assignment is to program graphics and create a generative pattern. You will be able to apply this pattern anywhere- but I am going to teach you to prepare it for print. By understanding the programming behind some of the most interesting generative art, you will be able to grow your concept and craft skills to create a beautiful work of art.

DELIVERABLE

Upload an image of your final generative art piece. A completed project includes process images of how you arrived at your final piece.

PROJECT STEPS

Environment Setup

1. **Download Processing**
   Download Processing from http://processing.org
2. **Download Sublime Text 2**
   Download Sublime Text 2 from http://www.sublimetext.com/2
3. **Add Sublime Text 2 Package Control**
   Find information about Sublime Text 2 Package Control here https://sublime.wbond.net
   In Sublime Text 2 open the console - View / Show Console - and in the input field at the bottom paste :
   import urllib2,os; pf='Package Control.sublime-package'; ipp = sublime.installed_packages_path(); os.makedirs(ipp) if not os.path.exists(ipp) else None; urllib2.install_opener(urllib2.build_opener(urllib2.ProxyHandler( ))); open(os.path.join(ipp, pf), 'wb').write(urllib2.urlopen('http://sublime.wbond.net/' +pf.replace(' ','%20')).read()); print('Please restart Sublime Text to finish installation')
   Hit return and RESTART Sublime Text 2 for changes to take effect.
4. **Install Processing Package**
   Within Sublime Text 2 navigate to:
   Sublime Text 2 / Preferences / Package Control
   From newly opened sub-menu click :
   Package Control : Install Package
   From newly opened sub-menu / input field type "processing" and click to install to install the Processing bundle into Sublime Text 2 (7:30 min on the "Working with Sublime Text 2" video)
   Processing should now be an available language when you click bottom right "Plain Text" area of Sublime Text 2.
5. **Install "processing-java"**
   We now need to add processing-java to our computer.
   To do this launch processing the application and navigate/click :
   Tools / Install "processing-java"

This will install processing-java on your machine so that code written in Sublime Text 2 actually runs.
(10:50 min on the "Working with Sublime Text 2" video)

6. **Change "Build" in Sublime Text 2**
   To change "Build" in Sublime Text 2 so that it mimics "Build" in Processing navigate in Sublime Text 2 to :
   Sublime Text 2 / Preferences / Key Binding - Default
   and
   Sublime Text 2 / Preferences / Key Binding - User
   in Default press find (command + f) and type "build" around line 232 copy :
   { "keys": ["super+b"], "command": "build" }
   and paste into User and make change from "super+b" to "super+r"
   [
   { "keys": ["super+r"], "command": "build" }
   ]
   Hit save and now Tools / Build should now be tied to "command + r" on the keyboard
7. **OSX users install "SizeUp"**
   Need help organizing windows? Use SizeUp to help position windows in certain quadrants of the screen. Windows users already have this functionality.
   http://www.irradiatedsoftware.com/sizeup
8. **OSX users install "Caffeine"**
   Looking to make pauses or walk away a bit without your computer going to sleep? OSX users can install "Caffeine" from the App Store.
   https://itunes.apple.com/us/app/caffeine/id411246225?mt=12
9. **Working with Sublime Text 2 Snippets**
   Download the attached snippet.zip from the "Final Environment Setup Tips" video.
   Open your Library folder :
   Finder / Go / Go to Folder and type ~/Library and click "Go"
   From this hidden folder navigate to :
   Library / Application Support / Sublime Text 2 / Packages / Processing / Snippets
   Copy "hype_setup.sublime-snippet" to this Snippets folder
   Close finder. Now within Sublime Text 2, select Processing from bottom right language selector and start typing "hype" and "hype_setup" should be available.
   (10:00 min on the "Final Environment Setup Tips" video)

Intro to Processing and HYPE

1. **Download HYPE**
   All of the source code for HYPE lives on Github. The most current build can be found inside of the staging branch:
   https://github.com/hype/HYPE_Processing/tree/staging
   In the right hand sub-menu click "Download ZIP"
   This zip file contains all of the necessary code that we will need in order to create our projects.
2. **Check out ZIP file contents**
   Examples are posted on the http://hypeframework.org website, the contents of the website are also mirrored in the "website" folder in the ZIP.

HYPE is a library of code that perform certain tasks - an isolated view of each of the classes and what they do, can be found in the "pde" folder. All of these individual files have been combined and concatenated into 1 single MASTER file, HYPE.pde.
As we write new classes I try to make as many examples as possible illustrating the functionality of that class. By digging around the "examples" folder you can find some pre-made blocks of code I have written showcasing some possible uses. If a folder has several examples usually the first few examples... example_001, example_002, etc... are as basic as possible. However as the examples progress an example might become more difficult as I showcase mixing that class with other classes to create even more robust interaction, animations and/or effects.
When first diving into examples be sure to start with : examples / H_BASICS as it tries to introduce core concepts using HYPE.

3. **Locate HYPE.pde**
   Inside of the zip file you will find HYPE.pde, this contains the entire HYPE universe! This file will be copy and pasted into every project we create.
   - projectName (folder)
   --- build (folder)
   ------ build.pde
   ------ HYPE.pde
   Every project will have this base structure.
4. **.JAR versus .PDE**
   Processing and it's community of developers have created several external libraries that perform a whole multitude of tasks from audio synthesis to using the Kinect. Most of these libraries are packaged into 1 single .JAR file.
   With the introduction of Processing 2 and ProcessingJS, we can actually publish sketches to HTML5 using the canvas object. However you cannot publish to ProcessingJS using external libraries using .JAR files.
   Since I like the idea of some of our projects being displayed and interacted with via the web and a web browser. HYPE was developed using the .PDE format.
5. **HYPE AS3**
   The first version of HYPE was created for Adobe Flash and its scripting language ActionScript with my good friend Branden Hall. When I decided to move to Processing I wanted to take some of the core functionality of the AS3 version and fix some of the issues and feedback that existed with the AS3 version of HYPE.
   for example, lets say we wanted to create a composition of 25 assets in a 5 x 5 grid. In the old version of HYPE I might have written something like this:
   var numAssets:int = 25;
   var layout:GridLayout = new GridLayout(50, 50, 100, 100, 5);
   a variable for how many assets I want to paint to screen and also use the GridLayout class to give the code some information about how to draw that grid.
   but what is 50, 50, 100, 100, 5 ? There's no information about what these numbers do. so I had to write the code like this with a comment :
   var numAssets:int = 25;
   // xStart, yStart, xSpacing, ySpacing, columns
   var layout:GridLayout = new GridLayout(50, 50, 100, 100, 5);
   Now you have some information what those numbers are… start the grid 50 on the x axis, 50 on the y axis, what's the spacing between the assets, and how many columns should there be. So major feedback was how rigid and obscure the arguments were in the code. If I was going to start over

with a new language and a new library, I wanted to make changes to the HYPE processing version so that it could be as clear and as flexible as it could be.

6. **HYPE processing**
   As stated before…
   If I was going to start over with a new language and a new library, I wanted to make changes to the HYPE processing version so that it could be as clear and as flexible as it could be.
   So we looked at 2 concepts that we think would make this new re-write as best as it could be… using a Fluent interface and Method Chaining.
   http://en.wikipedia.org/wiki/Fluent_interface
   http://en.wikipedia.org/wiki/Method_chaining
   so lets look at writing the same code like we did in the previous HYPE AS3 version :

```
int numAssets = 25;
HGridLayout layout = new HGridLayout();
layout.startX(50);
layout.startY(50);
layout.spacing(100,100);
layout.cols(5);
```

   so here you can see in this example we have no need for a comment. When we create our HGridLayout which we assign to layout… the next few lines handle the job of talking to layout and passing it some information like startX, startY spacing and columns. We can re-arrange the order… organize it however you'd like. But josh, you're saying, this seems like a lot more code from the previous example? Well lets look at Method chaining you can actually write this exact code also like this :

```
HGridLayout layout = new HGridLayout().startX(21).startY(21).spacing(26,26).cols(24);
```

   Pretty awesome! you can create HGridLayout and then start "chaining" the information, one right after the other. Or sometimes, to make it more readable, I can add hard returns and tabs like this :

```
HGridLayout layout = new HGridLayout()
        .startX(50)
        .startY(50)
        .spacing(100,100)
        .cols(5)
;
```

   these concepts I think make it much easier to write code… but are also major departures from how people have typically written code in Processing and Java. So remember that if you're new to programming this way of writing code is ONLY specific to HYPE and does not work with processing code writing tutorials and textbooks.

Drawing Visual Assets

1. **Find a theme to make art assets**
   Turkish Warriors, Japanese Kimonos, Kittens and Chainsaws ?
   Watch the "Drawing Visual Assets" video… in this video I have picked forms from the book "Mongolian National Ornaments". Think about what kind of forms you want to use as assets that will populate the code that we write in the next section.
   Look at some the forms I have collected and where they come from. They can come from internet searches, books (older the better), free drawing (sketchbooks), architecture, flea markets, etc. There's a whole world of form and texture that's going to get a fresh new look by abstracting it into code.

2. **Break apart, embrace abstraction**
   Watch the "Drawing Abstraction" video. Remember that a single master drawing can be broken apart into several parts. Those abstracted parts can do some unexpected things, because they might be taken from something familiar, they might bring some visual comfort to the viewer. Yet the viewer may not be able to pin-point why.
   Taking abstracted assets, placed into code and generated on screen will create new form and texture.
   For example:

http://pinterest.com/pin/180495897536927022/

This is a Mayan form I drew. When abstracted and placed into code it still feels Mayan, but becomes something new. A reinterpretation of something familiar in aesthetic.

http://pinterest.com/pin/180495897536927018/
http://pinterest.com/pin/180495897536927010/

3. **Draw a visual asset**
   Watch the "Drawing Time-lapse" video. Yes I love to draw in Flash! it's the closest I'll come to the old Macromedia Freehand style of drawing.

Draw a circle
Use circle to "create a symbol"
Make four layers
Place your circle on all four layers
Rotate each layer by 90 degrees (rotation 0,90,180,270)
You ONLY need to draw one side (I draw the right half)
Copy the half, paste and flip
Now one prong of the drawing is complete, but because we duplicated and rotated, all prongs are done!
Break apart (command+b) so that the drawing is not 4 duplicated symbols but is one vector drawing on stage (no symbols, just vectors).
Use paint bucket to fill areas with grays

4. **Prepare your final asset**
   Watch the "Final Asset Prep" video. How we save our vector based drawing into SVG is super important. Let's take a moment to look at my steps…
   Select artwork in Flash
   Click File / Export / Export Selection
   Save as .FXG
   Open .FXG in Illustrator
   In Illustrator select a compound path
   Click Object / Compound Path / Release
   This will break the vectors into separate "Paths" as opposed to being grouped into one larger object. This means the random coloring in our sketch will have the possibility of random colors as opposed to one universal color.
   Select and "GROUP" artwork in illustrator that I want to get the same color. For example grouping two eyes in a face so that both eyes receive the same random color.

Painting to the Screen

1. **Write your basic code**
   Lets start with the video "Basics / Line and Rect". This is the very basic amount of code you'll write in processing. This also demonstrates some of the basic snippet functionality of Sublime Text 2. The code written in this video section is in the attached "basics.zip" file.
2. **Experiment with moving and rotation**
   Moving on to the video "Basics / Rotation". If you're new to processing this helps get a basic understanding of how things get painted to screen. For example, we don't actually rotate the rect we rotate the entire canvas of the sketch.
   A good analogy is drawing on a sheet of paper… when you're drawing you might, move the position of the paper… up and down. You might also rotate the paper while drawing boxes. So the boxes themselves weren't rotated… the paper was rotated.
   The code written in this video section is in the "basics.zip" file from the previous video tutorial. Use these examples to help visualize how stage rotation is working.
3. **Using the Matrix Stack**
   In the next video "Basics / Matrix / push and pop" I illustrate how using the Matrix Stack can be used to position things on screen. The code written in this video section is in the attached "matrix.zip" file.
   http://processing.org/reference/pushMatrix_.html
   pushMatrix()
   Do some positioning stuffs

popMatrix()
Reset everything back
Within pushMatrix and popMatrix we can use translate(x,y,z) to move the x,y, and z coordinates of the stage and we can use rotate() to rotate the stage. When all is said and done… after the popMatrix… translate gets reset back to 0,0,0 for x,y,z and rotate back to 0. Remember our drawing on a piece of paper analogy: we don't draw the rect at a 45 degree angle. We turn the paper 45 degrees and simply draw a perfect square. popMatrix then rotates the sheet of paper back to 0. If you're now viewing the sheet of paper the rect "appears" rotated 45 degrees.

4. **Use code for rotation and color**
   In "rotation.zip" I simply want to add some animation to my sketch by having the squares rotate. This file helps us understand the "setup" and "draw" functions and some of the things we have to do when painting animation the screen. Notice if we don't add background() to the draw function, the screen doesn't clear and we are repeatedly painting the rect to the screen (leaves trails).
   In "color.zip" I add code for random color to the fill of my rect. But since this code is in the draw function and the draw function being a loop, it thinks every frame of the animation I want a new random color. In order to get only one color I would have to pre-pick a random color in setup and store it somewhere so that when it paints the rect in the draw function it applies the pre-picked color.
   All of these annoyances would lead me to build things a certain way in the processing port of HYPE. In the next project steps/videos, we start exploring using HYPE and how we can circumvent some of these problems.
5. **Start processing with HYPE**
   We're now moving on to the "HYPE / Basics" video tutorial and the associated "HYPE_basics.zip" file. Now we'll start using HYPE.pde in our sketches and show how HYPE circumvents those previous roadblocks.
   As we write some code let's think about some important things to remember :
   1. in setup() we need to tell processing that we're using an external library by adding :
   H.init(this);
   and we can chain some arguments… like a background color. Are we using 3D? should we clear or repeat paint the background? etc.
   H.init(this).background(#202020).use3D(false).autoClear(true);
   2. create some HYPE specific drawables like using HRect instead of processing's rect() :
   HRect d;
   d = new HRect();
   d.size(50).rotation(45).anchorAt(H.CENTER).loc(100, height/2);
   we can then chain some arguments for the rect like strokeWeight(), stroke(color), fill(color), noStroke(), noFill(), anchor(x,y), anchorAt(HConstants), scale(), size(width,height), width(), height(), loc(x,y,z), alpha(), rotation(), visibility(true/false), etc check / pde / HDrawable.pde for a list of all the things you can do.
   and then finally after we have set all of the properties we want this HRect to have we need to :
   H.add(d);
   think of it like this… .add() stores all of the properties for your HRect. so that we ask hype to paint what it stored to screen :
   3. H.drawStage();
   it will paint that visual asset with all of the properties that you set. Even if H.drawStage(); lives in the loop draw() function. If we picked a random color, that random color is stored and is carried over to the draw() function. If you're new to programming this saves a huge headache of having to understand the idea of pre-picking colors and storing them in setup() manually. Simply HYPE handles all of this stuff for you.

Can I get an Amen ?

--

step 2 introduces HRotate and makes rotating visual assets painless.

step 4 saves the randomly chosen color auto-magically when H.drawStage() is called in the draw() function.

6. **Paint drawables with HYPE**

In the next video "HYPE / Drawables" and associated code examples in "HYPE_objects.zip" we run through the many different types of things you can paint with on screen. Things like HRect, HEllipse, HPath, HBox, HSphere, HImage, and HShape.

All of the code example use the same basic structure:

1. H.init()
2. for loop / run something a set number of times / how many things do we want to paint on screen?
3. draw with something and set some properties
4. H.drawStage()

While it's great that I showcase a lot of the drawables in HYPE… it's best to really digest HShape. HShape is what will use to paint SVG art in our project. So lets take some time to focus on some of the properties associated with HShape.

First lets notice… we have a folder called "data" and this is where we will place any and all of our SVG artwork.

then will point/load in our SVG artwork by writing the following :

d = new HShape("filename.svg");

next lets look at the property of .enableStyle() by default .enableStyle(true), this means… whatever styling I did in Illustrator this will get carried over into our processing sketch… so any stroke and fill colors, etc. If I set .enableStyle(false) this means disable any styling done in illustrator because we're going to override the styling from within processing.

Some of the things we can override are :

.strokeWeight(1)
.strokeJoin(ROUND) / ROUND or MITER (square)
.strokeCap(ROUND) / ROUND or MITER (square)
.stroke() / color
.noStroke / don't display a stroke
.fill / color
.noFill / don't display a fill
.size() / set width and height
.width() / set only width
.height() / set only height
.scale() / scale artwork based on artwork's original size
.rotate() / set rotation 0 to 360
.loc(x,y,z) / set position of artwork on screen / z only works when using P3D
.anchor(x,y) / set the artwork's registration point / this point is used in .loc() and .rotate()
.anchorAt(HConstants) / use a pre-set registration position
- H.LEFT
- H.RIGHT
- H.CENTER_X
- H.TOP
- H.BOTTOM
- H.CENTER_Y
- H.CENTER

- H.TOP_LEFT
- H.TOP_RIGHT
- H.BOTTOM_LEFT
- H.BOTTOM_RIGHT
- H.CENTER_LEFT
- H.CENTER_RIGHT
- H.CENTER_TOP
- H.CENTER_BOTTOM

for example if you wanted the registration anchored in the "bottom / right" of the artwork
.anchorAt(H.BOTTOM_RIGHT)
Again this idea of registration anchoring is something borrowed from working in Flash and positioning/animating artwork around this point using either .anchor() or .anchorAt() I hope is useful to everybody.

Working with Color

1. **Commit color theft**
   In the video "The Color Thief's" I'm sharing some of my thoughts on where to hunt for color inspiration. A great starting point is to mine the color out of images. We simply search for .JPG's which can display millions of colors and save them to .GIF's which have a color table associated with them. The steps are as follows :
   1. Search for an image that has good colors.
   2. Open the image in Photoshop
   3. Select file / save for web
   4. In top right select the preset "GIF 32 No Dither"
   5. Use eyedropper tool and trash can in Color Table to remove unwanted colors
   6. Save GIF
   7. Download ColorPickingTool and open GIF
   8. You can sort color chips, hold "control" and click chips to delete
   9. When finished click "View Color List"
   10. Then click "View For Processing"
   11. Click "Copy Data To Clip Board"
   12. Paste colors, HEX data, in our processing sketches

   Start to use the trick and this tool to start building some great color combinations !
   Don't forget to give / https://twitter.com/michaelSvendsen a huge Internet High Five for packaging up the ColorPickingTool as a downloadable Air App.
2. **Use Kuler to create colors**
   In the video "Expanding Kuler" we don't have to work with photographs, we can use a Kuler Screenshot and work with gradients to build images that we can "Save for Web".
   Use Kuler's base 5 colors and expand them into a much larger range of colors.
   Save them as a "master.gif" and load them into the color picking tool.
3. **Create a color pool**
   In the video "HColorPool" we start using the first of three color capable classes within the HYPE framework. This code best suits the "Color Theft" from images video as we will start to build collections of colors. The code written in the video section is in the "HColorPool.zip" downloadable zip file.
   The HColorPool class is best defined in 3 easy steps :
   Create an instance of "HColorPool" and give it the name of "colors"

1. HColorPool colors;
Define a selection of colors
2. colors = new HColorPool(#FFFFFF, #F7F7F7, #ECECEC, #333333, #0095a8, #00616f, #FF3300, #FF6600);
Map randomly picked colors to our objects being painted on screen.
3. .fill(colors.getColor())
With these three easy steps, when the "for" loop runs 100 times, the fill is getting assigned a random color. Notice that we have eight colors. This means every color has a 1/8 chance of getting randomly picked. Using this process means the distribution of color, while random, should appear evenly distributed across the 100 objects.
Lets move on to "step 2" within the HColorPool folder and re-think item two in our three easy steps. If the code was written like this :

```
colors = new HColorPool()
.add(#FFFFFF, 10)
.add(#ECECEC)
.add(#CCCCCC)
.add(#333333)
.add(#0095a8)
.add(#00616f)
.add(#FF3300)
.add(#FF6600)
;
```

What's actually happening is we're building a list of 17 colors. The first ten colors in this list of 17 colors is actually the same color of WHITE #FFFFFF. So when the code runs, and picks a random number from 0 to 16, if the random number that selected is 0,1,2,3,4,5,6,7,8 or 9, the artwork is going to get filled with white.
This is "Color Weighting".
A very simple way for us to define which colors have a higher probability of getting picked over other colors. In our "Step 2" example the code should look like this :

```
colors = new HColorPool()
.add(#FFFFFF, 9)
.add(#ECECEC, 9)
.add(#CCCCCC, 9)
.add(#333333, 3)
.add(#0095a8, 2)
.add(#00616f, 2)
.add(#FF3300)
.add(#FF6600)
;
```

A dominance of the light colors, #FFFFFF, #ECECEC, and #CCCCCC, a small amount of dark gray #333333, and subtle tapering of teal blues #0095a8 and 00616f, and then our oranges being the rarest colors to hit the screen, #FF3300 and #FF6600. Most of my files are structured this way, as I tend to want to use randomness but control the probability.

4. **Apply color with HPixelColorist**
   In the video "HPixelColorist" we start using the second of three color capable classes within the HYPE framework. The code written in the video section is in the "HPixelColorist.zip" downloadable zip file.
   I have to thank Erik Natzke / http://www.natzke.com / for showing me this trick so many years ago… sitting on a beach in Sa Tuna, Spain in the beautiful Costa Brava region.
   Simply put: rather than stealing color OUT of photograph, why not use the photograph itself as a map of the movement of color across a specific width and height? Since an image is made up of pixels, if I were to randomly generate an art asset and that asset got attached at an x axis of 120 and a y axis of 42 (120,42) I could go to the photograph, move to the x and y of (120,42) and we would find a single pixel at that location, sample what color that pixel is and pull that color into the fill of our art asset.
   Doing so would mean that we could use a wide array of random techniques to paint stuff on screen -but sample the color from the color structure of a photograph. For example, a flower or a portrait, like Erik does in the following images.

In HYPE, the HPixelColorist class is also best defined in 3 easy steps :
Create an instance of "HPixelColorist" and give it the name of "colors"

1. HPixelColorist colors;
Tell HPixelColorist which image to use for the color sampling.
2. colors = new HPixelColorist("kelp.jpg").fillOnly();
Any of our color classes can be told what to color using .fillOnly(), .strokeOnly(), or fillAndStroke()
The last step, slightly different than in HColorPool, after the art asset has been attached, sample the pixel on the photograph in the exact same x and y axis that the "Anchor" got randomly affixed to.
3. colors.applyColor(d);
--
Don't forget using a Gaussian Blur on a photograph from within Photoshop is going to smooth out the blending of colors and may produce a more harmonious flow of colors on screen.
--
This zip file has five steps showcasing different sampling scenarios. Only step five is a bit different by using the following code :
.stroke(colors.getColor(d.x(),d.y()))
.fill(colors.getColor(d.x(),d.y()), 100)
This is only used if you are looking to apply color to both the Stroke and the Fill, but have an alpha of 100 applied to the Fill assets only. Please notice that when using d.x() and d.y() that .loc() would need to be moved above any d.x() and d.y() calls or it will return 0 for both values every time.

5. **Blend colors with HColorField**
In the video "HColorField" we start using the last of the three color capable classes within the HYPE framework. The code written in the video section is in the "HColorField.zip" downloadable zip file.
This class allows us to specify radial gradients in space. You'd see colors blending if we add two points and the radius of the radial gradients are large enough to overlap.
In HYPE, the HColorField class is also best defined in three easy steps :
Create an instance of "HColorField" and give it the name of "colors"
1. HColorField colors;
Tell HColorField the width and height of the field. Pass it a point of location (x,y). Then pass it a color to use, and finally the radius of the gradient. The last thing we want to use is what to color, .fillOnly().
2.
colors = new HColorField(width, height)
.addPoint(width/2, height/2, #FF3300, 0.3) // x, y, color, radius
.fillOnly()
;
Any of our color classes can be told what to color using .fillOnly(), .strokeOnly(), or fillAndStroke()
The last step, after the art asset has been attached, sample the color field, which is looking at this x, y, color, and radius information to provide a color which gets pulled into the art asset.
3. colors.applyColor(d);
--
Don't forget that using .fill(#000000) or .fill(#ffffff) affects what color the gradient transitions to.
Don't forget that using multiple points with larger radius's will cause the radial gradients to interact and blend colors. For example, a point of red and a point of blue that overlap would blend purple where they overlap with each other.

HGridLayout

1. **Specify your drawing with HDrawablePool + HRect**

In the video "HDrawablePool + HRect", before we can start working with layout it's best for us to cover the "glue" that's going to tie all of these things we're assembling together. The code written in the video section is in the "HDrawablePool.zip" downloadable zip file.
While HDrawablePool has an abundance of features, what's important in the context of this class, is it allows us to do the following :
1. Specify the number of assets being painted to screen
2. Specify which assets are getting painted to screen
3. Like HColorPool, weight the probability of certain assets getting painted more than others
4. Assign layout classes
The attached downloadable zip file, has three steps, which is used across the next three videos. This video uses only step one, which paints with HRect. Let's take some time to break down HDrawablePool.
First create an instance of HDrawablePool, calling it "pool".
1. HDrawablePool pool;
Now lets define how many assets the pool will draw with.
2. pool = new HDrawablePool(100);
Where should the pool paint these assets? In the case of these file were going to be painting assets to the stage.
3. pool.autoAddToStage()
What assets is the pool using to paint with?
.add(new HRect(), 20)
.add(new HRect().rounding(10))
Here we're painting with 2 HRect's, the second having rounded corners with a radius of 10. However, the first HRect is weighted, adding itself 20 times. Running the sketch should paint the screen with a dominance of HRect without rounded corners.
Next we define what happens to each of the individual HRect's that get painted to screen.
4.

```
.onCreate(
  new HCallback() {
    public void run(Object obj) {
      HDrawable d = (HDrawable) obj;
      d
        .strokeWeight(1)
        .stroke(#FF3300)
        .fill(#111111)
        .size( (int)random(25,125) )
        .rotate( (int)random(360) )
        .loc( (int)random(width), (int)random(height) )
        .anchorAt(H.CENTER)
      ;
    }
  }
)
```

Finally we ask for ALL of our 100 objects to get painted on screen at once.
5. .requestAll()
--

We'll continue to use this base structure for every example moving forward. HDrawablePool becomes the foundation/glue that ties many of our separate blocks of code into one finely tuned running machine of awesome.
Hooray for awesome!

2. **Apply SVG assets with HDrawablePool + HShape**
   In the video "HDrawablePool + HShape 1", we'll continue to work with HDrawablePool but add working with external SVG assets which we make in Illustrator. The code written in the video section is in the "HDrawablePool.zip" downloadable zip file, folder / step 2.
   Just like the step 1 example, the foundation of the code is the same with a few minor details.
   1. Create a "data" folder
   2. Add our 6 SVG assets within this folder
   3. Change our .add() to work with HShape instead of HRect

```
.add(new HShape("svg1.svg"))
.add(new HShape("svg2.svg"))
.add(new HShape("svg3.svg"))
.add(new HShape("svg4.svg"))
.add(new HShape("svg5.svg"))
.add(new HShape("svg6.svg"), 20)
```

   With this change, running the sketch should be painting on screen with our 6 external SVG assets… however, "svg6.svg" should have dominance on screen since 20 instances of it were weighted in the HDrawablePool.
   Finally we also add :
   4. HColorPool colors; / d.randomColors(colors.fillOnly());
   which should be painting all of the individual fills of the attached SVG with random colors.
   --
   Take note this artwork is simple vector assets, geometric shapes with a width of 50 pixels and a height of 50 pixels. Painting many "simple" shapes on screen will not become too busy. Keep this in mind as we move on to the next exercise.
   --
   Take note that typically when working with SVG assets I use the following code :

```
.strokeJoin(ROUND)
.strokeCap(ROUND)
```

   As this will display better visual results on screen and when outputting to a vector based file format at the end of this class.

3. **Apply complex SVG assets with HDrawablePool + HShape 2**
   In the video "HDrawablePool + HShape 2", we'll continue again to work with HDrawablePool but add working with external SVG assets which we make in Illustrator. The code written in the video section is in the "HDrawablePool.zip" downloadable zip file, folder / step 3.
   Just like in the step two example, the foundation of the code is the same with a few minor details.
   1. pool = new HDrawablePool(50);
   Because this set of assets is much more complex than in the previous step 2 example, it's best to lower the number of assets getting painted as we might run into some issues with crashing or out of memory errors.
   2. change our .add() to work with the new set of SVG assets
   .add(new HShape("mongo1.svg"))
   .add(new HShape("mongo2.svg"))
   .add(new HShape("mongo3.svg"))
   .add(new HShape("mongo4.svg"))
   .add(new HShape("mongo5.svg"))
   .add(new HShape("mongo6.svg"))
   Finally we also edit :
   3. .size((int)random(200,400))
   You'll find you need to tailor the creation properties with the artwork getting used. Since these SVG assets have lots of details, these details don't tend to read well at smaller scales on screen.

   Take note this artwork has much more complex vector assets. This idea of creating complexity from simplicity really starts to resonate at this point. If the assets being used are very complex in nature and you make the mistake of painting to many of them on screen. The complexity within complexity is going to start to complicate or muddy the clarity of the composition.
   Unless you're a more is more as a life mantra, the more complex the artwork gets, the less you'll want to paint on screen.

4. **Control display with HGridLayout class**

In the video "HGridLayout", we'll finally be adding the final piece of the puzzle by adding a layout class to help us control how things can displayed on screen. The code written in the video section is in the "HGridLayout.zip" downloadable zip file.

So starting with the adjustments implemented in HDrawablePool, we'll attach a layout class to the code. As long as you don't specify any .loc() within .onCreate() HDrawablePool will use ANY layout class to set .loc() auto-magically.

in order to use a layout class, only 1 block of code needs to be added to your sketch. I've added this block of code right after .add() of my SVG assets.

```
1.
.layout(
  new HGridLayout()
  .startX(25)
  .startY(25)
  .spacing(50,50)
  .cols(11)
)
```

When attaching this particular class, we have some arguments that need to be passed with the class in order to describe the instructions for visual display on screen.

1. .startX() // where does this grid, the whole thing, start on the x axis?
2. .startY() // where does this grid, the whole thing, start on the y axis?

NOTE : that this is based off .anchor() or .anchorAt(). If your artwork is .anchorAt(H.CENTER), and in the case of this file the artwork is 50x50, then using a .startX(0) and a .startY(0) your grid would be -25 on BOTH x and y since .anchor has moved -25 pixels in order to center.

Next lets describe the spacing of each cell :

3. .spacing(50,50) // what's the amount of pixels of each cell for the width and height.

Since our artwork is 50 x 50, we build a grid that has no spacing between the visual assets. If our artwork was 100 x 100 then using .spacing(50,50) cell2 of the grid would overlap 50 pixels of cell1, and so on. If our artwork was 50 x 50 and we used .spacing(60,60) then there would be a 10 pixel gutter between visual assets.

The last thing we describe in our layout is :

4. .cols(11) // how many cells per row

since we can look at :

pool = new HDrawablePool(121);

121 / 11 = 11 So we know that in this sample file were building a perfect 11 x 11 grid.

--

NOTE : We could still have pool = new HDrawablePool(100); and layout will still build the grid it will just be 21 cells short of being a perfect grid.

--

folder / step 2

In this modification, we only paint with .add(new HShape("svg6.svg")) and play with the idea of randomly rotating each cell by 90 degree jumps. .rotate((int)random(4) * 90) which will rotate either 0, 90, 180, or 270 (please don't forget that a rotation of 0 is the same as a rotation of 360)

--

folder / step 3

In this modification we paint with all SVG assets but introduce random sizes based on 50 pixel jumps, .size(50 + ((int)random(4) * 50)) which would give us 50, 100, 150, and 200

To clarify why, lets break it down…

Focus on this code (int)random(4) * 50) this would give us this :
0 * 50 = 0
1 * 50 = 50
2 * 50 = 100
3 * 50 = 150
Since setting a size of 0 (0 * 50 = 0) would be invisible on screen… this is why we start the code with a 50 + randomStuff
0 * 50 = 0 + 50 = 50
1 * 50 = 50 + 50 = 100
2 * 50 = 100 + 50 = 150
3 * 50 = 150 + 50 = 200
--
Be sure to check out these project which all use the code you now have:
http://pinterest.com/praystation/244-the-social-grid/ this project even uses the SVG assets provided in HGridLayout / step 2 - Group HUG
and
http://pinterest.com/praystation/194-aim-high-keep-moving/
http://pinterest.com/praystation/158-grid-filigree/
For you textile / pattern freaks… I anxiously await the endless patterns you shall dazzle us with. Using randomness within structure is a trick I have used for years to make things that are aesthetically pleasing on screen while still embracing the unexpected.
--
Future thinking :
http://pinterest.com/pin/180495897536926335/
It's a HGridLayout - the x spacing is tight, the y spacing is loose and an HOscillator uses a Sine wave to twist each row.

HShapeLayout

1. **HShapeLayout**

In the video "HShapeLayout", we'll be working with our last layout class. The code written in the video section is in the "HShapeLayout.zip" downloadable zip file.
This is simply one of my favorite layouts to work with… we'll again be embracing random but defining where that random happens. HShapeLayout uses a .target() to pass in a shape to use for the attaching and sticking of assets. .target() can be passed many things HRect, HEllipse, etc… but my favorite is HImage, and specifically transparent PNG's.
so lets break down our new set of steps :
1.

```
.layout(
  new HShapeLayout()
    .target(
      new HImage("shapeMap.png")
    )
)
```

Here we create an instance of HShapeLayout, pass it an argument .target() and define our external PNG to use new HImage("shapeMap.png")
2. toggle over to photoshop and create a new image 600 x 600

3. create a new layer
4. hide the background layer
5. define some shapes, you could use squares, circles, type, paintbrush to free draw, whatever.
6. file / save for web
7. select the upper right preset PNG-24 which supports transparency
8. save as "shapeMap.png" within your data folder of your sketch
9. toggle back to Sublime Text 2 and build sketch
10. watch head explode.

So what's happening here ? Well… HShapeLayout, when passed a transparent PNG is randomly picking x and y values on your image… and if the x and y value returns a transparent pixel… it re-picks a new x and y coordinate. It will keep doing this until the randomly selected x and y coordinate is a NON-transparent pixel. If it hits a NON-transparent pixel it simply attaches the SVG .anchor to the successful x and y coordinate.

What were left with on screen is a method of randomly "STICKING" assets to what ever forms and shapes you defined within your PNG.

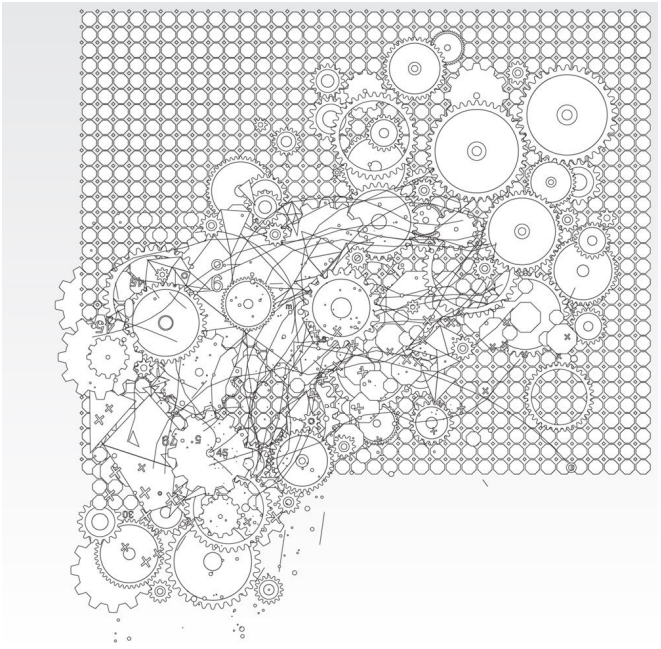
HUZZAH !

--

I can't begin to tell you how much I use this in the generative posters I do for clients, for example… Deadmau5, aka Joel Zimmerman asked me to some graphics for a European Tour Poster… the theme ? cut the head in half and show the inner gears and workings of a mau5 head.

http://pinterest.com/praystation/230-deadmau5-gear-head/

So I drew up some gears and stuff… and used HShapeLayout to attach those assets to the form of the mau5 head.



Outputted this back to a vector file and masked the generated output to said mau5 head.

I then sent over the grayscale composition to Joel… which I always do… never color… always grayscale. Clients can get super hung up on color, by presenting the work in grayscale, Joel can offer feedback on texture and not get hung up on my color selections.
Then I send over a round of compositions using HColorPool and different banks of colors… blue was the winner.

Finally we Internet High Five each other, our chakras align, the universe sings… and a unicorn dies. (or so I've been told)
--
some other examples :
the assets / http://pinterest.com/pin/180495897536882294/

the shape / http://pinterest.com/pin/180495897536882291/
the assets / http://pinterest.com/pin/180495897536881931/
the shape / http://pinterest.com/pin/180495897536881934/
and some of my favs using type :
http://www.flickr.com/photos/joshuadavis/3420913987/in/set-72157613080257958
http://www.flickr.com/photos/joshuadavis/3420912953/in/set-72157613080257958
http://www.flickr.com/photos/joshuadavis/3420911767/in/set-72157613080257958

Output Files

1. **letsRender / bad**
   In the video "letsRender / bad", we'll be looking at ways of getting our randomly generated compositions out of processing and into a format we can edit either in Photoshop or Illustrator. The code written in these video sections is in the "letsRender.zip" downloadable zip file.
   Sometimes it's best to look at the bad way of doing things to fully understand the limitations.
   In this step 1 example we only add 1 line of code :
   1. saveFrame("render.png");
   the default use of rendering is easy, but leaves much to be desired. While the use of saveFrame() supports many different image formats…
   saveFrame("render.jpg");
   saveFrame("render.png");
   saveFrame("render.tif");
   saveFrame("render.tga");
   Some formats are better than others… compression versus no compression… but I'm drawn to the PNG format despite not getting any initial transparency. Let's fix some of these issues in the next video segment.
2. **letsRender / better**
   In the video "letsRender / better", we'll be looking at ways of getting our randomly generated compositions out of processing and into a format we can edit either in Photoshop or Illustrator. The code written in these video sections is in the "letsRender.zip" downloadable zip file.
   Let's enhance our ability to render to a pixel based format with a few additions to the code.
   1. within setup() call a new function saveHiRes(2); where we pass a scaleFactor
   2. noLoop(); only fire the draw() function once
   3. have a draw() function with a call to H.drawStage();
   4. create a new function called saveHiRes()

```
void saveHiRes(int scaleFactor) {
  PGraphics hires = createGraphics(width*scaleFactor, height*scaleFactor, JAVA2D);
  beginRecord(hires);
  hires.scale(scaleFactor);
  if (hires == null) {
    H.drawStage();
  } else {
    H.stage().paintAll(hires, false, 1); // PGraphics, uses3D, alpha
  }
  endRecord();
  hires.save("render.png");
}
```

If we're rendering to Photoshop and a pixel based image then this is the file for you. Even though our sketch is 600 x 600 saveHiRes(int scaleFactor) allows us to create a larger image 1200 x 1200 by calling saveHiRes(2);
we can also render to PNG WITH transparency by calling H.init(this).background(H.CLEAR);
Rendering this to a transparent larger image just means we have more endless possibilities within photoshop… we can layer items and create larger compositions than we could with step 1.

3. **letsRender / BEST**
   In the video "letsRender / BEST", we'll be looking at ways of getting our randomly generated compositions out of processing and into a format we can edit either in Photoshop or Illustrator. The code written in these video sections is in the "letsRender.zip" downloadable zip file.
   There's no denying… this is my JAM!
   If were working with vector based assets then I want to get back to a vector based render. While similar to the pixel based output in step 2, there are a few modifications we need to run through.
   1. import processing.pdf.*; we'll be outputting to PDF so we need to import processing PDF library in order for this to work.
   2. H.init(this).background(#202020); no need for H.CLEAR
   3. create a new function call called saveVector(); again no need for a scaleFactor
   4. make changes to our new function :

```
void saveVector() {
  PGraphics tmp = null;
  tmp = beginRecord(PDF, "render.pdf");
  if (tmp == null) {
    H.drawStage();
  } else {
    H.stage().paintAll(tmp, false, 1); // PGraphics, uses3D, alpha
  }
  endRecord();
}
```

   With all of this in place we can work with vector based assets and get back to a vector based output render. For me, this is the most ideal scenario, because it offers further modification in illustrator… color changes, additions, deletions, etc. Anything can be modified. While in the step 2 example, once something is rendered to a pixel based image… making changes to that image is not as flexible.
   Hooray for vectors !
4. **letsRender / final edits**
   In the video "letsRender / final edits", we'll be looking at ways of getting our randomly generated compositions out of processing and into a format we can edit either in Photoshop or Illustrator. The code written in these video sections is in the "letsRender.zip" downloadable zip file.
   In this example, our base file is from HGridLayout / step 3. It uses some random rotation and random sizing that would require some further edits within illustrator.
   for example :
   1. selecting all the same stroke colors within illustrator and give them 1 universal stroke weight. as the SVG attached in our sketch changes sizes, so does the size of the stroke weights.
   2. use the appearance palette to "double up" your strokes 1 at a stroke weight of 1 and another at a stroke weight of 3 but with a 20% transparency… this soft blending of weight's when being printed adds a softness that I use in all of my outputted prints.
   3. adding colors and making color groups, because…

4. the "recolor artwork" button in the toolbar allows us to experiment with re-randomizing the color mappings.
With the already existing colors or maybe an entirely new set of colors added into a color group.
--
Obviously the more comfortable you are with illustrator… the better your visual results are going to be.
This is just the beginning for some of you in an entirely new world of random possibility.

Project Inspirations

1. **Student Gallery**
   Based on some of the projects that you are working on and posting in the Student Gallery... I thought I would create some videos and .ZIP files that show some things you can try.
   Keep Posting the Awesome!

Additional Resources

- Suggestion:
- At this point, I would complete:
- Section four "Painting to the screen" and section five "Working with Color".
- In these sections I will provide some sample SVG files to work with. I felt it was important to cover drawing assets first, even though it's probably the last thing you'll do.
- Why?
- Because one block of code using a random layout class and a random color classs is artwork agnostic. It doesn't care what the drawings are. In fact you could take one project, feed it different colors and artwork and never touch the "base" code but you'd have an infinite number of possibilities.
- So lets write some CODE !