Topics for DANM 220 Winter 2013.

*Optional readings

Date	topic	Objectives	Reading	Assignment
Jan 7	Essential Background	Coordinates, colors, Define frame structure	Intro to Max, Pre-jitter studies, Messages and structure*	Set up Max
Jan 9	Basic drawing	Code structure, functions,	Basic Drawing, Drawing with mgraphics*	
Jan 14	Animated drawings	The draw loop	Simple Animation	Random drawing
Jan 16	Movie munging	pixel color, color processing	Dealing with Movie, Gallery, op spotter*, effect spotter*	
Jan 21	Holiday		debugging	
Jan 23	Compositing, repos	Image combination Image distortion	Compositing Meet Repos*	Filter a movie
Jan 28	Drawing with trigonometry	Function based images	Trigonometry Notes	
Jan 29	Feedback tricks	Iterative processing, fractals, flamer	Feedback, Fractals, Flamer*	triangle
Feb 4	Live video	Video capture & analysis	Zen Mirror*	
Feb 6	object tracking	Trigger events from motion	Motion tracking, net cameras*	Follow the post-it
Feb 11	Sound	Playback, capture, MSP	Sound Playback	
Feb 13	visualization	Audio to image	Visualization of audio*, Lissajous*	Visualizer
Feb 18	Holiday			
Feb 20	Input	Keyboard, mouse & external devices	Controllers, Firmata*	Work on project
Feb 25	Output	External control	Firmata*, DMX	""
Feb 27	Networking	Webcams, data mining	Webcams*	""
Mar 4	The 3D world	Intro to openGL	OpenGl in Jitter*	""
Mar 6	Motion in 3D	Vectors and collisions	Vectors in jitter*	""
Mar 11	Optimization	Gen, shaders	Shaders in Jitter*	
Mar 13	Wrapup			
Mar 18	Crit	Student projects		

Assignments:

Assignments may be done with Max or Processing. However, the lectures and sample programs will be in Max. You may email completed assignments to elsea@ucsc.edu. You may email me with questions at the same address.

- Random Drawing: create an image with some aspects based on chance.
- Filter a Movie: make a movie unrecognizable but interesting
- Triangle: create a program that builds layers of Sierpenski triangles
- Follow post-it: use findbounds to trace an object's motion.
- Visualizer: write a patch that responds to music.

To include a patch in an email:

1. Use Select All on the patch.
2. Choose Copy Compressed from the edit menu.
3. Paste that directly into an email.

If I send a patch to you, it will look like this:

<pre><code>
----------begin_max5_patcher----------
211.3ocSPsjBBCCDccxoHLqqhsXUvcdCDboHRZcrFIeJookhh2cal1ptYBuO
yjGuWbFT35wFPrSbRvXu3LFQEIXSXFXj8kZYCYCJcFCZCPxnV.6CD+QooVih
CxP48YQaqQY0Xf1Lch7lyFZTOQhKa4pedcsgYyyr27UED9qu53GnrUW7XYXL
3o4oCxhz04zSdzrHaXJNOsj5JkQWwiEYve4vJMTNf8dkTCQg2bdbPqBx55Nz
2nbVpQh8wPa7v4ivMIDTYGgT.AO1ol8ukGu1a9G.3F6TqC
-----------end_max5_patcher-----------
</code></pre>

Copy everything from <pre> to </pre> to the clipboard.
In Max, choose New from Clipboard.

What exactly can a computer do for the artist?

- Art shown with projections and screen display.
- Control artistic contraptions.
- Change the audience experience with responsive systems.
- Generate a lot of variations on a theme.
- Surprise us.

The realities of the digital canvas.

The digital canvas is some sort of computer screen or a projection. The colors you get will be somewhat different from model to model. (Imagine what kind of painting you'd get if the color of the canvas kept changing.) In such a world, it is better to worry about contrast and color range instead of a specific shade of teal.

Many projectors have a native pixel architecture that is different from computer screens. Often it is actually different from the published specifications. Converting images with differing architecture will require rescaling, which often introduces lateral lines or other distortion. Distortion will be even worse if the aspect ratios do not match. You should be prepared to resize your images if necessary.

The level of detail is limited by the screen resolution. It is also limited by the power of the computer hosting your application. There are millions of pixels in a single frame of video, and calculating a frame will take several machine cycles per pixel. All of a sudden a billion operations per second does not seem very fast. Computers have both a CPU (central processing unit) and GPU (graphics processing unit, generally part of the display card.) The GPU is about a thousand times faster than the CPU, but it is often awkward to use. For instance, GPU programming is done in a different language. (We can program the GPU in Max, but not in processing.) In all cases there is a tradeoff between resolution and time.

The time/resolution tradeoff means the frame rate is often less than the optimum 30 fps. We usually don't notice until the rate goes below 20 fps, but our perception change is sudden. This happens at different rates for different people. Slow rates may be tolerated for images that don't change much.

The third dimension.

The world has become accustomed to very real 3D images from features like "Finding Nemo" and "Avatar". Of course those are produced on render farms with 1000 machines producing a few minutes a day. A more realistic goal is game level realism. That means simple textures, less subtlety in lighting, and so on.

We specify 3D images through a language called openGL. Both Max and processing support openGL, and try to help with the more tedious aspects of working in that language. Max 6 offers important improvements in game style graphics.

Image and sound.

Our works of art need not be silent. Processing will allow cued playback of recorded material and samples, as well as some elementary synthesis. Max/MSP is a full fledged synthesis and DSP environment. The two can coexist and communicate, so using Processing for images and Max for sound is perfectly feasible. Either can also communicate with most audio platforms such as Reactor, Supercollider, ChucK and Csound. Max can host your choice of plug-ins.

Creating images out of sounds is also feasible in either language.

Transformation of pixels

A lot of video art is based on transforming existing video footage. At one time there were modular video synthesizers to complement the more famous Moog and Buchla music synthesizers. These performed processes like threshold switching and slew limiting, all things that can be done with an analog image waveform. In digital video processing, it is the color of pixels that is modified. The modifications can be based on several things. For instance, if there are two video sources, the color of a pixel may be compared to a specified color. If they match, the output is taken from the second source. Otherwise the first source is passed through. Thus everything colored green in the A signal is replaced by colors from the B signal. In other processes, the color used for a pixel may be taken from a different pixel in the original image. This can change the size of an image, or twist it in arbitrary ways.

Algorithmic drawing

Tools like Illustrator let us use the computer as a canvas, transferring ideas from our head to the screen. Max and Processing move some of the decisions to the computer. All we have to do is invent rules for the computer to follow. The rules are necessarily based on mathematics, but the math need be no more complex than what's needed to balance a checkbook. One beauty of algorithmic drawing is the rules are controlled by parameters supplied by the artist. This means many images can be made from the same algorithm.

- *Geometry* produces straight line figures. You can do a lot with straight lines, especially if they make up tiny rectangles. The underlying concept is the vector, but that is only an instruction of where to draw.
- *Trigonometry* produces curves. These curves are defined by angles and functions based on the angles. These functions are deep math, but the computer will perform them for us.
- *Iterative function systems* apply functions to input, then apply the same function to the results. When we get an image we like, we quit. This kind of thing can produce remarkable images including plants, mountain ranges and snowflakes.

- *Chance operations* inject random parameter values into a process so each run produces a different image.
- *Path based graphics* work from a script of drawing instructions similar to the actions available in Illustrator or other vector drawing programs.

Dynamic Drawing

Computer art does not have to stand still. If we use time as parameter the image can be redrawn with subtle changes or drawn fast enough that the eye is fooled into perceiving motion.

- *Loops* perform the same pattern of operations over and over again. This produces simple animation or a painting that follows the time of day.
- *Physical models* follow some of the rules of the real world. This produces action like bouncing balls or growing vines. This also includes chaotic systems with repeatable but surprising behavior.
- *Particle-systems* take modeling down to the molecular level. They can be used to create fire and smoke and flocking birds.

Noticing the world

Computer programmers often speak of real time. This is code that runs fast enough that response seems simultaneous with input. Even if there is noticeable delay, an art program can respond to changing environment or requests by users (whom we define as someone who is operating the program, but did not write it). This input takes many forms:

- *Camera input* captures images. These images can be used as source material for image processing or analyzed to provide control parameters for the program.
- *Physical sensors* can detect practically anything. We can build a system that is played like a violin or tracks the humidity over several days.
- *Network links* expand the input range to include a large part of the world. Again the foreign data can provide raw material or control. We can also use networks to remotely control our art by arbitrary gestures like tipping an iPhone.

Controlling the world

The program output is not limited to images on a screen. Anything that is electrical can be controlled, either directly from an Arduino style circuit or a standardized network protocol. A program can generate instructions to a performer or mechanic or another computer program. Some very powerful protocols available to us include:

- *MIDI* Originally a system for transmitting musical performance, MIDI has been expanded into surprising areas, such as lighting control.
- *DMX* is a control system for theatrical lighting systems.
- *Open Sound Control or OSC*. Another musical system, with more power at the expense of less generality.
- *RTSP* lets us receive or transmit images over the internet.

The Way of Code

The only true mastery of a computer comes through the ability to write code. When we use canned software, we are bound to a certain world view, accepting the priorities and paradigms of the software's author. With millions of programs available, it seems we should be able to find something that matches our needs and methods, but we soon realize the search takes longer than the task, and we settle for anything that will (sort of) get the job done.

It is absurd to consider writing any application from the ground up. Fully formed applications require complex operations for screen display, file storage and retrieval, communications and numerous other functions. These problems have been solved again and again-- tackling them is a waste of our time. Instead, we build on other's work, adding our code to a framework of routines that satisfy the standard needs. We will work with one of two frameworks:

Max/MSP/Jitter lets us assemble complex programs from building blocks of working code. The process seems something like Legos at first. As with Lego building, you are limited to the objects already provided, and these objects force a certain overall shape. However, there are so many objects available that just about anything can be done within the system. Unlike Lego, it is possible to invent new objects. There are enough people doing this now that Max has reached a critical mass, and the possibilities are expanding faster than any one person can reach the limits.

Processing provides a nearly complete application called a sketch. It is your job to complete the application by writing two or three unfinished sections of code. The result is a Java applet, suitable for adding to a web site or running an installed art work. The scope of Processing is not limited to computation and screen display. It can generate audio and MIDI or communicate with Arduino to control external hardware.

Both languages provide an ever-expanding set of capabilities and have ample resources for learning and discovery. Each has an impressive on-line community willing to give advice to the newcomer (after the basic manuals have been read, of course) and provide advice for advanced projects. There are fine books of instruction and vast libraries of examples.

The standard approach to writing code

The traditional development process goes through a series of logical steps, the same steps necessary to many complex undertakings. They are specification, design, coding and testing.

Specification

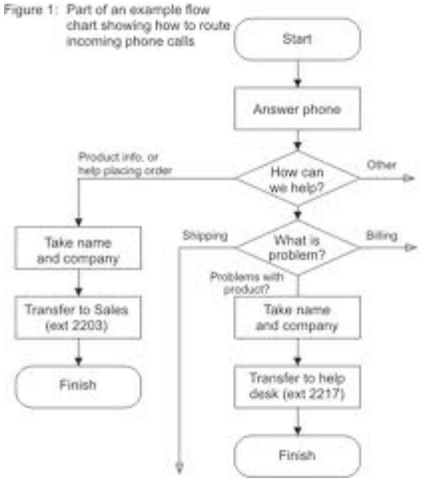
An application begins with a problem to be solved or an operation to be performed. The more precisely the problem or operation can be defined, the easier the code will be to

write. It is possible to change your mind, to add features or remove them, but the later this happens the more wasted work. Changing specs midway can also lead to internal inconsistencies that will wreck the entire program. Specification is not as formal or complex as it sounds. It's just a list of inputs and outputs with a description of how they relate. If you think of it an instruction manual, it becomes a much less academic undertaking.

Design

Design is the creative phase, the invention of algorithms to get the work done. A lot of these algorithms have already been invented, so your first task is to research the field and find them, or something close enough to be modified. Resist the urge to write code during the design phase, except little experiments to test something. The result of the design phase is a flow chart showing how one action leads to another. Make sure every input and output in the specification is included.

Figure 1: Part of an example flow chart showing how to route incoming phone calls



Coding

Writing code is the tedious part of the process. If you work in Max, the code resembles the flow chart. In Processing, every object in the chart is represented by a paragraph of code that defines a function and the structure is defined by calling functions in the draw routine. Coding is made difficult by the terse vocabulary and rigid syntax of Processing or the subtle gotchas and annoying inconsistencies of Max.

Testing

Testing actually begins as soon as there is enough code to test. If the flow chart is complete, the input and output of the functions will be clear enough to try each out as they are written. Once the whole thing is operational, give it to someone else. It will immediately break. Find out how your tester broke it and figure out the problem. Repeat the process until your tester cannot break it. It is tempting to test it yourself, but you will be amazed at how your intimate knowledge allows you to subconsciously avoid dangerous operations.

Revision

Testing and use will inevitably lead to revisions of the program. When you do this, don't just jump in and modify code. Go back and change the specifications, then see how wide ranging the effects of the change will be.

Documentation

The initial specification can be revised into instructions for using the program. You should also include comments in your code that describe the operations of each section. Make these clear enough for a stranger to understand[1].

The artist's method for writing code

We aren't in it for the money, so we don't really need to worry about efficient production strategies or meeting customer's specifications. In most cases, all we want to do is produce something beautiful and surprising. This means we don't need to obsess about "correct code", it just needs to run. So the coding process for art can be more free-wheeling (although you can do the other way if you want to.) Art coding is based on playing, stealing, modifying existing code, critique, and yes documentation.

Play

The first thing to do is get comfortable with the code world. You don't need to learn it all at once--all you really need is enough knowledge to get output in a usable format. We'll go through a lot of examples together, and if some process seems interesting, play with it enough to see where it takes you. Go ahead and try things that don't make sense. Some of my favorite pieces come from mistakes and foolishness.

Steal

Since the art is the results of our programming efforts, and not the code itself, we are less bound by the issues of patents and copyright[2] that plague the computing world. When you see something by another artist that intrigues you, see if you can figure out how they did it. Then play with that process and see what you can produce.

Modify

Never settle for the first version of anything. Your first try at a process may give awesome results, but a slight change may make it even better. Keep copies of all versions-- some paths may be a blind alley, but if we back up a bit and try different variations, we may find even neater things.

---

[1] That stranger is you, in about 6 months.

[2] Of course, we have our own copyright issues, but they are more straightforward. Somebody may own the copyright for an image of a person, but no one will ever patent

[2] Of course, we have our own copyright issues, but they are more straightforward. Somebody may own the copyright for an image of a person, but no one will ever patent the idea of portrait.

Critique

Once you have output, look at your work, and decide how it rates. There are three possible categories: something you were looking for and can use now, not what you were looking for but still useable in another context, and learning experience. The criteria that put the work into these categories are your own, but after enough exposure to other works, you should be able to develop a sense of how others will react.

Documentation

Boring as it is, we have just as much need for documentation in the art business as in any other programming field. There is no worse feeling in the world than being asked to do some more of what you produced two years ago and having no idea how you did that. Keep a notebook.

What Programmers Really Do

I mentioned earlier that no application is ever written from scratch. The basic functions that load files, connect to the internet and so forth are all supplied as part of the operating system. Furthermore, if you are programming for Windows or Apple computers, you have thousands of utility routines to do things like manage window display and text editing. Coding in those environments is mostly research-- I often spend a day of manual reading to write one line of code, but what that line does is amazing. For instance, in Mac OSX, there is a function called NewMusicPlayer. This creates everything needed to play a multitrack MIDI sequence. Once this is done, it only takes a few more lines of code (with equally powerful functions) to load a sequence and play it.

If a programmer needs something that is not supported directly in the OS, the code may be available in a library somewhere. Libraries are sets of routines that address common problems. Once a programmer (or company) has written code that works well, they are often willing to share (or make some extra $$). For instance, a search for "fractal fern code" returned 132,000 hits. If I wanted to write a fractal fern routine, I need look no further. In addition to this, there are hundreds of textbooks that consist of nothing but examples of code. "Fractal programming" turns up 14 pages of books on Amazon.

Working coders also take advantage of application frameworks. Frameworks are nearly complete applications- sort of DIY kits for code. There are a few hundred of these. Max and Processing are frameworks, as are Flash, MacFlux and Mathematica. Frameworks may be extensions to an existing language (Processing is an extension of Java) or something completely unique like Max.

All of this means programmers don't often get to invent things, most applications are paste-ups of existing code. Of course, they do need the knowledge necessary to understand how things fit together.