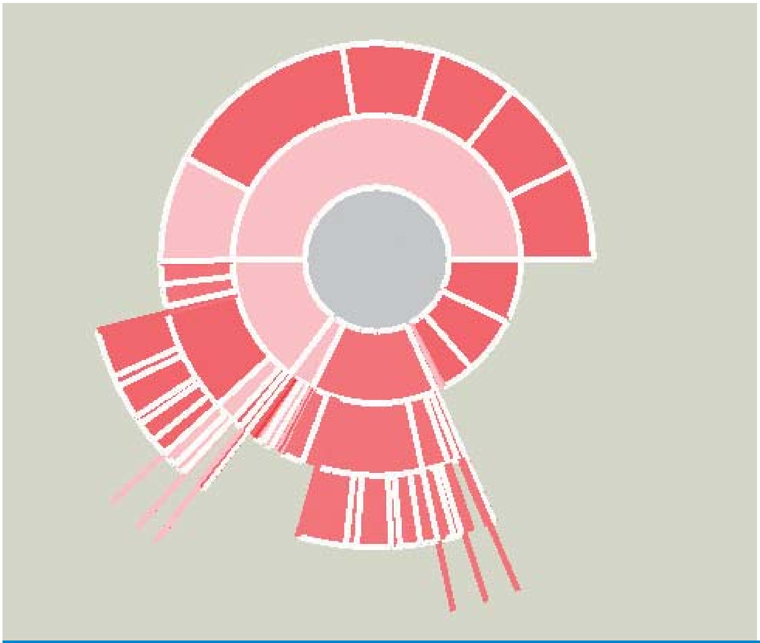


Visually Deconstructing Code

Ben Fry

Visually Deconstructing Code is a series of experiments looking at the form of code. This collection searches for an alternative perspective of text, machine language, and binary software codes, making visual the abstract structures and processes buried within.

The evolution of software projects

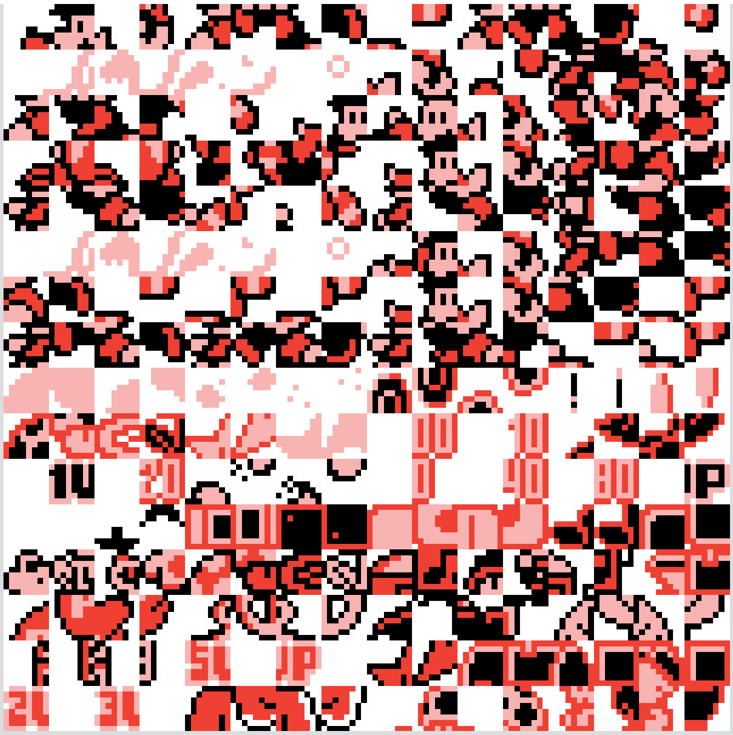
While it's obvious that the code in a software project changes over time, less obvious is the nature of how individual changes have taken place in a broader context. Projects are typically structured as a collection of files that are added, removed, and reorganized throughout the course of development. The contents of the individual files are modified, line by line or in large pieces for every fix and feature.
Reasonably large projects, or those that are shared between several authors, are often tracked using a version control system. One of the most common is called CVS (or Concurrent Versions System), and is freely available. Understanding how a project has evolved is a matter of understanding the data stored by the version control system, which keeps track of changes as incremental events on the modification to files or how they're organized.
The first experiment in this set is an interactive application that shows the evolution of the structure and content of the Processing *(http://Proce55ing.net)* project over time, from its initial inception through fifty releases. The experiment consists of large-format printed pieces to depict broader changes over time, while an interactive application allows the user to walk through the individual events that led to differences between the releases.
The result is a depiction of the organic process in which even the smallest pieces of software code become mature through the course of its development, as they are passed between developers, revisited for later refinement, merged, removed, and simplified.

From machine-like language to the language of the machine

For most programming languages, code written by the programmer is translated to a more abstract form that is directly understandable by the machine on which it will be run. The initial code, which ranges from unintelligible to beautifully expressive, is made undoubtedly more arcane in its “machine language” form. But the complexity of this obfuscation process often hides the beauty in the translation, where some parts are made more terse, others more verbose, and all of it given a more rigid structure.
When developing in the Java programming language, the stream of instructions in the resultant machine language code (which is actually written for a “virtual” machine which does not exist) also stores information about what line in the original human-readable code produced each machine-readable instruction. This experiment brings the two kinds of code back together, using a visual image to lay bare the translation process of compilation.

Packaging of data within code

Any piece of executable code is also commingled with data, ranging from simple sentences of text for error messages to entire sets of graphics for the application. In older cartridge-based console games, the images for each of the small on-screen images (the “sprites”) were often stored as raw data embedded after the actual program’s instructions.
This third piece examines the unpacking of a Nintendo game cartridge, decoding the program as a four-color image, revealing a beautiful soup of the thousands of individual elements that make up the game screen.


packaging data within code

Seeing time in the operation of code

It's common to use a "profiler" on code while it's running to see where the machine is spending its time. Functions are shown with percentages marked for how long is spent within each area. Better tools also show the hierarchy of functions that have been used one after another, a hierarchy that shows the "call stack" of the successive methods. This piece examines the output from a profiler using an active diagram, where the functional relationships are laid out spatially, and the percentage of time is shown as a series of thicknesses. The diagram makes apparent the bloat of areas within the code that are poorly written or are simply doing the majority of the work.

Visual mathematics in code algorithms

There is a class of software algorithms that includes cryptography, error checking, and serial number testing that work like the mathematical equivalent of the ridges found on an intricate key. Their operation is a series of gymnastics performed with a group of numbers that comprise the key.

This piece examines such an algorithm that simulates the process of how the serial numbers for software products by the manufacturer Adobe are generated and tested. It begins with a simple seed number, and then walks through several mathematical steps, mostly simple addition or multiplication, to generate a multi-digit key for the product. An application such as PhotoShop or Illustrator will use such an algorithm to test whether the key entered by the user is proper; while the same algorithm can also be used to generate a myriad of fake but working keys for anyone who wants their own. The nature of this experiment is to illustrate the elegance and simplicity of a process kept intentionally as opaque as possible to the end-user.

Each of these experiments begins with the question: "How can this aspect of code be understood visually?" As singletons, they are simplistic ideas, but as a collection, they begin to provide a visual perspective on how code works and behaves, introducing a mental model for code that is more organic than common tools of depiction like text, tables, and graphs.