# ARTE E PROGRAMAÇÃO NA LINGUAGEM PROCESSING

Patricia Oakim

Capa desenvolvida pela autora. Imagem criada utilizando o código-fonte do Processing. A aplicação lê o código-fonte e seleciona algumas palavras e caracteres para produzir a imagem. Desenvolvida em Processing. Código-fonte da aplicação disponível no Apêndice A.

**PONTIFÍCIA UNIVERSIDADE CATÓLICA DE SÃO PAULO**
**PUC-SP**

**Patricia Oakim Bandeira de Mello**

# Arte e programação na linguagem Processing

**MESTRADO EM TECNOLOGIAS DA INTELIGÊNCIA**
**E DESIGN DIGITAL**

**SÃO PAULO**
**2015**

**Pontifícia Universidade Católica de São Paulo**
**Faculdade de Ciências Exatas e Tecnologia**

**Patricia Oakim Bandeira de Mello**

# Arte e programação na linguagem Processing

**MESTRADO EM TECNOLOGIAS DA INTELIGÊNCIA E DESIGN DIGITAL**

Dissertação apresentada à Banca Examinadora da Pontifícia Universidade Católica de São Paulo, como exigência parcial para obtenção do título de MESTRE em Tecnologias da Inteligência e Design Digital, sob a orientação do Prof. Dr. Marcus Vinicius Fainer Bastos.

**SÃO PAULO**
**2015**

**Banca Examinadora**

_____

_____

_____

*Àqueles que fazem coisas estranhas com software.*

## AGRADECIMENTOS

Em primeiro lugar, gostaria de agradecer ao meu orientador, Marcus Bastos, pelas aulas que me inspiraram à retornar ao Processing e a realizar esta pesquisa, pelo apoio e por indicar caminhos tão fascinantes.

Agradeço também ao Professor Alexandre Campos Silva pelas diversas recomendações e pelo apoio.

Ao Professor Hermes Hildebrand, pelas sugestões e pela entrevista concedida para a pesquisa.

Ao querido amigo Sinan, pela entrevista e pela generosidade com os dados da comunidade OpenProcessing.

Ao meu pai, Alexandre, pela revisão cuidadosa e pelas inúmeras correções.

À minha mãe, Sandra, pelas opiniões sobre a capa da dissertação.

À minha irmã, Juliana, por todo o apoio.

Ao Gustavo, pelas leituras, dicas, correções e por todo o apoio.

Ao Leo, pelas várias leituras, correções e apoio.

Ao amigos Fernando Cespedes, Judu e Caio, pelo apoio, sugestões e leituras.

Ao Matheus, por conferir o exemplo de código em Java e pelas dicas para a construção do algoritmo da capa.

Ao T3, pelas opiniões sobre a capa.

Ao Cabral, pela ajuda com a impressão da capa.

Ao amigo Clayton, por sentar ao meu lado nas aulas e por compartilhar as mesmas emoções.

E a todos aqueles que me apoiaram de tantas maneiras.

*Há muitas coisas que um computador não é. Ele não é um projetor de cinema interativo, nem uma máquina de escrever cara, nem uma enciclopédia gigante. Na verdade, ele é uma máquina para rodar software.*

**Noah Wardrip-Fruin**

**RESUMO**

Desde a década de 1960, diversos artistas vêm experimentando com a tecnologia computacional na criação de suas obras de arte. Também diferentes linguagens de programação surgiram com o objetivo de tornar a programação mais fácil para pessoas com formações e interesses diversos, como artistas ou crianças. O *Processing*, umas dessas linguagens, surgiu em 2001, no *Massachusetts Institute of Technology* (MIT), com o objetivo de facilitar a programação nas artes visuais. Ele foi amplamente adotado em vários países no ensino de programação para artistas e também para a produção final de obras de arte. Além disso, foi a inspiração para o *Arduino*, uma plataforma de prototipagem para interação física com o ambiente que se tornou bastante popular. Intitulada *Arte e Programação na linguagem Processing*, esta pesquisa se propõe a investigar a seguinte questão: "*Quais são as relações entre software, programação e arte na linguagem de programação Processing?*". A pesquisa debate, sob a perspectiva do *Processing*, o *software* na sociedade e na arte, além de examinar o papel do artista-programador – aquele que trabalha diretamente com linguagens de programação. Além disso, discute também o aprendizado de programação como parte de uma alfabetização digital. A metodologia de pesquisa se baseia em um levantamento bibliográfico acompanhado de uma análise crítica do material consultado e um estudo de caso da linguagem de programação *Processing*. A pesquisa é ainda complementada pela experiência direta da autora e por duas entrevistas. Esta investigação apresenta aspectos culturais da arte feita em *Processing*, englobando desde o contexto das linguagens de programação que facilitam a atividade de programar até características do *software* livre e da cultura *hacker* na comunidade de artistas-programadores do *Processing*.

Palavras-chave: arte computacional, artista-programador, *software*, cultura *hacker*, *software* livre, *Processing*

**ABSTRACT**

Since the 1960s, many artists have been experimenting with computer technology to create artwork. Different programming languages have also emerged since then with the goal of making programming easier for people with various backgrounds and interests, such as artists and children. Processing is one of these programming languages. It was created in 2001 at the Massachusetts Institute of Technology (MIT) with the objective of streamlining programming in the visual arts field. Processing was widely adopted as the programming language used for teaching programming to artists in several countries and it is also used for the final production of works of art. Besides, it was the inspiration for Arduino, a prototyping platform for physical computing that has become quite popular. Entitled *Art and Programming in the Processing language*, this research aims to investigate the following question: "*What are the relationships between software, programming, and art in the Processing programming language?*" The research debates, from the Processing perspective, software in society and art, in addition to examining the role of the artist-programmer - the artist who works directly with programming languages. It discusses still the learning of programming as part of computer and code literacy. The research methodology is based on a literature review and critical analysis, along with a case study of the Processing programming language. The research is complemented by the direct experience of the author and two interviews. This research presents cultural aspects of art made with Processing, encompassing from the context of programming languages that make the programming activity easier up to aspects of free software and hacker culture in the community of artists-programmers that use Processing.

Keywords: computational art, artist-programmer, software, hacker culture, free software, Processing

## LISTA DE FIGURAS

**SUMÁRIO**

# Apresentação

Meu primeiro contato com a linguagem de programação *Processing* aconteceu em 2006, enquanto eu cursava o mestrado *Interactive Telecommunications Program,* mais conhecido como ITP, da *Tisch School of Arts* na *New York University* (NYU). O programa é um grande centro de aprendizado prático em arte e tecnologia, uma espécie de pós-graduação *hackerspace*. Em uma das disciplinas que cursei no ITP, elaborada pelo Professor Daniel Shiffman (umas das atuais lideranças do *Processing*), a linguagem era utilizada no ensino de programação para novatos. E foi lá, com o *Processing*, que eu aprendi programação pela primeira vez.

Em 2013, retomei o contato com o *Processing* durante o mestrado no programa Tecnologias da Inteligência e Design Digital (TIDD), na Pontifícia Universidade Católica de São Paulo (PUC-SP), onde pude estudar arte e tecnologia com ênfase em teoria. E, apesar de já conhecer a linguagem há quase 8 anos, fiquei intrigada com sua expansão no Brasil. Descobri muitas pessoas usando a linguagem no país, além de cursos e oficinas de programação criativa em *Processing* no SESC e no Instituto Volusiano, em São Paulo.

Combinando a experiência prática do ITP, o interesse recém despertado pela perspectiva teórica da arte computacional no TIDD e a curiosidade sobre a expansão do *Processing* no Brasil, o escolhi como objeto de estudo.

O resultado é esta pesquisa.

# Introdução

Desde o início da computação nos anos 1950 e 1960, diversos artistas vêm realizando experimentos com computadores. Foi nessa época que surgiram as primeiras obras de arte computacional pelas mãos de pioneiros como Frieder Nake, Michael Noll e o brasileiro Waldemar Cordeiro. Recentemente, a arte computacional vem tomando novas formas nas obras de artistas como Casey Reas, Golan Levin, Aaron Koblin e dos brasileiros Jarbas Jácome e Fabrizio Poltronieri, dentre outros. Esses artistas utilizam o *software* como um material artístico e as linguagens de programação são seu instrumento de trabalho. Nesta pesquisa eles são chamados de artistas-programadores.

Desde seu surgimento, a computação e, consequentemente, o *software* se expandiram de maneira a permear diversos aspectos da sociedade. Até aqueles que não estão incluídos no mundo digital (por diferentes motivos) são influenciados pelo *software* mesmo que de maneira indireta, pois ele está presente no funcionamento da sociedade – no governo, na logística da circulação de materiais, nos mercados de ações, no entretenimento, na comunicação e em tantos outros setores. Vivemos, portanto, no que Manovich (2013) chama de Sociedade do *Software*. Todavia, nossa interação com o *software* se dá, na maior parte do tempo, nos telefones celulares e nos computadores por meio de interfaces que escondem o código-fonte. Assim, nos tornamos apertadores de botões dos diversos dispositivos que alguns poucos sabem como programar.

Contudo, diversos pesquisadores têm procurado, desde o início da computação, criar interfaces, ferramentas e linguagens para tornar a programação mais simples e levá-la para além dos limites da engenharia. Algumas iniciativas importantes são: o *Logo*, o *Max*, o *Pure Data*, o *Processing*, o *Arduino* e o *Scratch*. A linguagem *Logo*, de 1967, tinha por objetivo ensinar matemática para crianças. Na década de 1980, surgiu a linguagem *Max*, muito usada para composição musical e, principalmente, para performances ao vivo. Na década de 1990, foi lançado o *Pure Data*, uma versão *open source* do *Max*. Em 2001, surgiu a linguagem *Processing,* destinada às artes visuais. Em 2005, foi lançado o *Arduino*, um microcontrolador projetado para a programação de interação física entre o ambiente e o computador. Mais recentemente, em 2007, surgiu o *Scratch*, outra

linguagem de programação direcionada ao público infantil. Todas essas iniciativas compartilham o mesmo objetivo: facilitar a programação para pessoas de idades e formações diversas. Na presente pesquisa, essas linguagens são chamadas de linguagens de programação alternativas e se define o termo como linguagens de programação projetadas para pessoas com formações fora das Ciências Exatas e muitas vezes com um caráter educativo.

Os primeiros *hackers* surgiram no *Massachusetts Institute of Technology* (MIT) nas décadas de 1950 e 1960. Eles eram engenheiros criativos com paixão por experimentar as possibilidades dos computadores. Estabeleceram a base cultural da abertura da informação no universo computacional, que posteriormente se desdobrou no *software* livre. O trabalho em grupo, o aprendizado na interação com outros programadores e a abertura do código-fonte são alguns dos aspectos oriundos da cultura *hacker* que se fazem presentes em diversas linguagens de programação alternativas.

O *Processing* surgiu no *Media Lab do Massachusetts Institute of Technology* (MIT) em 2001. Ele foi criado por Casey Reas e Ben Fry, na época alunos de doutorado da instituição, com o objetivo de facilitar a programação para *designers* e artistas visuais. No *Processing*, computação e arte se encontram de uma maneira diferente do que havia acontecido anteriormente, pois não se trata de uma linguagem de programação comum. É uma linguagem concebida especificamente para as artes visuais. Assim como diferentes materiais possibilitam diferentes resultados artísticos na escultura, na pintura ou na música, também na arte computacional diferentes linguagens de programação permitem diferentes resultados estéticos (REAS; MCWILLIAMS, 2010). O *Processing* apresenta funções nativas e uma estrutura de processamento interno otimizadas para a produção de imagens. Ademais, tornou-se uma linguagem amplamente usada em escolas de arte de todo o mundo no ensino de programação para artistas e designers. Além disso, o *Processing* foi a inspiração de um outro projeto adotado em diversos *hackerspaces*, *makerspaces*, *fab labs* e *media labs* espalhados pelo mundo: o *Arduino*.

O *download* do *Processing* na internet já foi realizado mais de 2 milhões de vezes (PROCESSING, 2014). Existem 23 livros publicados por diversos autores (PROCESSING BOOKS, 2015) e mais de 11.300 tópicos de discussão foram postados no fórum oficial da

linguagem[1] (PROCESSING FORUM, 2015). A comunidade *OpenProcessing*[2], criada em 2008 em Nova York por Sinan Ascioglu, agrega artistas, professores e estudantes de todo o mundo. São 123.216 *sketches* (aplicações em *Processing*) publicados e 51.441 usuários cadastrados, dentre eles, além dos usuários comuns, 4.783 estudantes e 357 professores[3]. Foram criadas na comunidade mais de 1.000 salas de aula por professores de vários países, que são usadas como apoio para cursos presenciais (OPENPROCESSSING CLASSROOMS, 2015). Desde o início de 2015, o Brasil é o oitavo colocado no *ranking* mundial de acessos ao *OpenProcessing* e em agosto de 2015, esteve em segundo lugar (GOOGLE ANALYTICS, 2015). Em São Paulo, o Instituto Volusiano[4] e o Sesc São Paulo[5] oferecem cursos e oficinas de programação criativa com o *Processing*. A Escola de Arte e Tecnologia Oi Kabum![6], do Instituto Oi Futuro, em Belo Horizonte, também utiliza a linguagem, assim como a Universidade Tecnológica Federal do Paraná[7].

O *Processing* se tornou também uma linguagem de grande popularidade entre artistas para a produção final de suas obras. Artistas prestigiados como Casey Reas, Aaron Koblin e Golan Levin o utilizam. Além disso, não é raro encontrar obras feitas em *Processing* em exposições de arte e tecnologia no Brasil e no mundo. Alguns exemplos são o FILE (Festival Internacional de Linguagem Eletrônica)[8], que acontece no Brasil todo ano, o Emoção Art.ficial[9], que acontecia bianualmente no Itaú Cultural em São Paulo até 2012, e a competição *online* DevArt[10], patrocinada pelo Google em 2014 e que apresenta obras feitas a partir de programação. Outrossim, se observa o uso da linguagem em projetos digitais que nascem no programa de residência artística LABMIS (Laboratório de Novas Mídias do Museu da Imagem e do Som)[11], em São Paulo.

---

[1] forum.processing.org
[2] openprocessing.org
[3] Dados de 25 de agosto de 2015 fornecidos por Sinan Ascioglu, criador da comunidade.
[4] institutovolusiano.com.br
[5] sescsp.org.br/aulas/1528_COMPUTACAO+CRIATIVA+COM+PROCESSING+E+ARDUINO
[6] oifuturo.org.br/educacao/oi-kabum
[7] utfpr.edu.br
[8] file.org.br
[9] emocaoartficial.org.br
[10] devart.withgoogle.com
[11] mis-sp.org.br/labmis

Assim sendo, com o seu crescimento no Brasil e no mundo, faz-se necessário pensar como o *Processing* está alterando o cenário da criação de arte computacional. Mais do que explorar o conteúdo da produção, é crucial examinar o contexto cultural em que ele está inserido. Segundo Manovich (2010), vivemos em uma cultura do *software*. Em nossa sociedade a produção, a distribuição e a recepção da maior parte do conteúdo cultural são mediadas por *software*. Logo, para refletir sobre essa produção cultural, é preciso também examinar o *software*.

> Se não abordarmos o software em si, corremos o perigo de sempre lidar somente com suas consequências, em vez de causas: o produto que aparece na tela do computador, e não os programas e as culturas sociais que geram esses produtos (MANOVICH, 2010, p.186).

Nesta dissertação, entende-se *software* como o amplo universo cultural de utilização e produção de programas de computador. De acordo com Roberts (2008), a criação de um programa de computador requer dois passos: primeiro se projeta a estratégia de funcionamento do programa (o algoritmo) e depois se expressa o algoritmo em código-fonte utilizando uma linguagem de programação. Fazem parte do universo do *software*, portanto, os programas, os algoritmos, o código-fonte, as linguagens de programação, as interfaces, os usuários e os programadores. O debate sobre a conceituação do que é *software* faz parte do Capítulo 1. Usualmente, os termos *software* e programa são utilizados como sinônimos para se referir a um programa de computador e assim se faz nesta dissertação.

A presente pesquisa busca debater, sob a perspectiva do *Processing*, o *software* na sociedade e na arte, além de examinar o papel do artista-programador e discutir o aprendizado de programação como parte de uma alfabetização digital. Esta investigação visa a um entendimento dos aspectos culturais da arte feita em *Processing*, englobando desde o contexto das linguagens de programação alternativas até aspectos do *software* livre e da influência da cultura *hacker* na sua comunidade de artistas-programadores. Desse modo, procurando compreender o atual processo de criação de arte computacional a partir da programação, a seguinte questão é explorada:

*Quais são as relações entre software, programação e arte na linguagem de programação Processing?*

Apesar de existirem livros, dissertações e artigos publicados que ensinam programação em *Processing* ou que abordam a linguagem (i.e. GREENBERG, 2007; NETO; 2010; RIBEIRO, 2007; SHIFFMAN, 2008), uma análise como a aqui proposta ainda não foi realizada. A investigação parte de duas hipóteses:

1. O sucesso do *Processing* se deve à licença de *software* livre e à cultura *hacker*, com base principalmente na abertura da informação, o que gerou uma grande comunidade *online* de desenvolvedores e de artistas que trocam experiências e colaboram entre si, contribuindo para o seu crescimento.
2. O *Processing* surgiu em uma época em que os artistas computacionais passaram a aprender programação para criar suas obras, tornando-se, nesse processo, artistas-programadores.

Para a realização desta pesquisa foi feito um levantamento bibliográfico acompanhado de uma análise crítica do material consultado e um estudo de caso da linguagem de programação *Processing*. Além disso, esta investigação é complementada pela experiência direta da autora e por duas entrevistas com personagens relevantes para o tema: Sinan Ascioglu, criador da comunidade *OpenProcessing,* e Hermes Renato Hildebrand, que além de lecionar *Processing* na Universidade Estadual de Campinas (UNICAMP) é também coordenador do programa de pós-graduação TIDD na PUC-SP.

A presente dissertação está dividida em cinco partes. A seguir encontra-se o Capítulo 1, *Software e Arte*, que está dividido em duas seções. Na seção 1.1, *O mundo roda em software*, aborda-se o crescimento da importância do *software* na sociedade, sua definição como prática cultural e a interação cotidiana das pessoas com as interfaces computacionais, instigando o debate sobre o aprendizado de programação. A finalidade da Seção 1.2, *Software na Arte*, é expor de maneira abrangente a arte feita a partir do *software*. São apresentados um breve histórico da arte computacional internacional e seu

aparecimento no Brasil, no final da década de 1960, pelas mãos do artista Waldemar Cordeiro e do poeta Erthos Albino de Souza, além de outros artistas mais recentes.

No Capítulo 2, *Programação e Arte*, o debate segue em direção às linguagens de programação, pois elas são o instrumento de trabalho do artista que se expressa através do *software*. Na seção 2.1, *O artista-programador*, o papel do artista que se apropria da tecnologia, prototipa e cria suas obras a partir do código-fonte é examinado. Em seguida, na seção 2.2, *Linguagens de Programação Alternativas*, a discussão sobre o aprendizado de programação é aprofundada e o contexto das linguagens alternativas em que o *Processing* se encontra é apresentado através de uma exposição sobre as linguagens *Logo*, *Scratch*, *Max*, *Pure Data* e *Arduino*. Por último, na Seção 2.3, *Cultura hacker,* os aspectos culturais de abertura da informação, livre troca de ideias e aprendizado em comunidade oriundos do grupo dos primeiros *hackers* no MIT são abordados. Faz-se uma relação entre essas características e o *software* livre, além de uma análise sobre a cultura *hacker* na educação.

O Capítulo 3, *O Processing*, apresenta um estudo de caso sobre a linguagem. Na seção 3.1, *A linguagem*, o *Processing* é exposto em detalhes: seu surgimento, sua estrutura e seus objetivos. Na seção 3.2, *Software livre e cultura hacker na programação de arte em Processing*, os conceitos relativos à cultura *hacker* discutidos na seção 2.3 são aplicados em uma análise sobre a criação de arte que utiliza a prática de *Tweak* (bifurcação) na comunidade *OpenProcessing* como exemplo. Além disso, a troca de informação sobre o *Processing* em fóruns *online* é apresentada. Por último, na seção 3.3, *Arte em Processing*, os conceitos de funcionário e artista experimental, expostos no Capítulo 1, são abordados pela perspectiva do *Processing*.

Finalmente, na *Conclusão*, com base na trajetória de pesquisa descrita acima, são elaboradas algumas considerações finais sobre as hipóteses iniciais em relação à licença de *software* livre do *Processing* e em relação ao papel do artista-programador. Ademais, são indicados alguns desdobramentos possíveis para pesquisas futuras.

# 1 Software e Arte

## 1.1 O mundo roda em software

A expansão da computação nas últimas cinco décadas provocou inúmeras transformações na sociedade. As mudanças na comunicação, no ensino e no aprendizado, no surgimento das redes sociais e das comunidades *online*, nas novas formas de entretenimento e de organizações empresariais, entre tantas outras, são evidentes. Há 50 anos o computador pessoal nem existia e, atualmente, há 3,2 bilhões de usuários de internet no mundo, o equivalente a 45% da população total do planeta (ICT FACTS & FIGURES, 2015).

Esse avanço da computação significou também um aumento da presença do *software* na sociedade. As lógicas e os códigos do *software* cada vez mais penetram em diferentes aspectos do cotidiano, como se pode notar, por exemplo, na telefonia celular: são 7 bilhões de aparelhos no mundo com 97% de penetração global (ICT FACTS & FIGURES, 2015). É claro que isso não significa que o acesso à computação é irrestrito. A exclusão digital ainda é vasta, porém mesmo as pessoas que não estão incluídas digitalmente de maneira direta são influenciadas de maneira indireta pela computação e pelo *software*, pois os sistemas que gerem a sociedade moderna funcionam em *software*, como explica Manovich (2013, p.8, tradução nossa):

> a escola e o hospital, a base militar e o laboratório científico, o aeroporto e a cidade – todos os sistemas sociais, econômicos e culturais da sociedade moderna – rodam em software. Software é a cola invisível que amarra tudo. Apesar dos vários sistemas da sociedade moderna falarem línguas diferentes e de terem objetivos diferentes, todos eles compartilham as sintaxes do software: parâmetros de controle como “if then” e “while do”, operadores e tipos de dados (como caracteres ou números floating), estrutura de dados como listas e convenções para a interface que englobam menus e caixas de diálogo.

Os meios de comunicação baseados em *software* estão crescendo. A comunicação digital é regida por regras determinadas nos algoritmos das ferramentas que utilizamos no cotidiano tanto para falar com outras pessoas quanto para o entretenimento. Em maio

de 2015, por exemplo, quase 37% de todo o tráfego *downstream* (volume de dados sendo requisitados pelos usuários) da internet nos Estados Unidos era Netflix (LUCKERSON, 2015). Enquanto a televisão analógica é um meio de comunicação eletrônico por meio do qual um sinal de vídeo chega às casas transmitido por satélite, por antena ou por cabos, um provedor de conteúdo audiovisual sob demanda como Netflix funciona em *software* através da internet. Assim como houve a predominância dos meios eletrônicos analógicos no século 20, o pilar central da comunicação atual se baseia cada vez mais nos meios de comunicação que utilizam *software*.

Mas, afinal, o que é *software*? Existem algumas maneiras de se entender o que é o *software*. Pode-se pensar, por exemplo, no *software* sob uma perspectiva técnica. Nessa visão, *software* é definido em contraste ao *hardware*. Enquanto o *hardware* é material, o *software* é uma entidade virtual e abstrata, uma sequência de comandos declarados em uma linguagem que o *hardware* consegue interpretar e executar (ROBERTS, 2008). Essa série de passos e instruções escritas numa linguagem de programação que formam o programa de computador é chamada de algoritmo.

> A palavra *algoritmo* vem até nós do nome do matemático persa do século IX Abu Ja 'far Mohammed ibn Mûsa al-Khowârizmî, que escreveu um tratado sobre matemática intitulado *Kitab al jabr w'al-muqabala*, cujo título deu origem à palavra inglesa *álgebra*. Informalmente, você pode pensar em um algoritmo como uma estratégia para solucionar um problema (ROBERTS, 2008, p.8, grifo do autor, tradução nossa).

No entanto, contrário a essa concepção de *software* como uma entidade abstrata que funciona em conjunto com o *hardware*, Kittler (1995), no ensaio *There Is No Software* (Não Há *Software*), argumenta que o *software* em si não existe, pois todo o funcionamento do computador está baseado no processamento eletrônico e material dos dados. Utilizando como exemplo um programa de edição de texto, o autor afirma que os nossos gestos de produção de texto correspondem meramente a códigos construídos em silício e a impulsos elétricos. E sob essa perspectiva, o texto escrito no computador não existe do ponto de vista material, pois só existe como processamento eletrônico das sequências de código-fonte que o formam.

> Todas as operações de código, apesar dessas faculdades metafóricas como chamada e retorno, se resumem absolutamente a manipulações locais de linhas de código, ou melhor, temo dizer, de significantes de diferenças de voltagem (KITTLER, 1995, tradução nossa).

A visão de Kittler é fundamental para que não se perca a referência de que há um aparato material tecnológico que roda o *software*. Isso leva a uma reflexão política, pois a visão abstrata do *software* como algo existente somente dentro da lógica de funcionamento do computador camufla o contexto social e econômico em que se encontra o seu desenvolvimento. O *software* e a sua produção comercial estão inseridos em uma conjuntura econômica, com organizações empresarias que têm objetivos financeiros gerindo todo o processo. Cox e McLean (2013, p.20-21, tradução nossa) esclarecem que

> essa visão [de Kittler] também reconhece o aparato material mais amplo da programação e os vários agentes de produção envolvidos no processo, os quais incluem engenheiros que projetaram as máquinas e os operários que as montaram assim como os programadores que escreveram os programas. Sentidos são produzidos através das interações desses agentes em todos os níveis de atuação

Do ponto de vista econômico, principalmente quando se trata do *software* proprietário, a produção de sentido está atrelada à lógica comercial, evidenciando até mesmo relações de hegemonia econômica e cultural entre países. Isso se torna perceptível quando se pensa no Vale do Silício, nos Estados Unidos, por exemplo, como centro de produção intelectual de *software* e nas fábricas da FoxConn na China como centro de manufatura de dispositivos eletrônicos para grandes marcas da indústria digital – uma lógica que se repete em diversas outras indústrias.

Todavia, nenhuma das duas concepções sobre *software*, seja a abstrata ou a material, está equivocada. Conceitualmente, o *software* é a entidade abstrata por meio da qual o programador consegue controlar a máquina, utilizando linguagens de programação para determinar as sequências de comando que são executadas através de diferenças de voltagem no *hardware*. Ademais, em uma reflexão política sobre o *software*, não se pode perder de vista o fato de que existe um aparato material amplo, ou seja, político e econômico, que possibilita a existência do *software* em nossa sociedade.

Contudo, apesar de o *software* não existir materialmente e de ser imprescindível se refletir sobre os meios digitais levando-se em consideração o contexto político e econômico em que estão inseridos, talvez seja possível afirmar que o *software* realmente existe, mas não materialmente e sim por uma perspectiva cultural. Na presente pesquisa, por cultura se entende – como define Castells (2003) – um conjunto de valores e crenças que formam os padrões de comportamento repetitivos de um grupo.

Cramer (2005), em sua definição de *software*, vai além da oposição ao *hardware.* Segundo o autor, o *software* é acima de tudo uma prática cultural. A literatura, por exemplo, não é somente literatura quando está escrita, mas envolve narração, tradição e poesia declamada. Da mesma maneira, do *software* nascem práticas culturais como "navegar" na internet, ferramentas de *chat*, *download*, *streaming* de filmes, entre outras. A criação de *software* é uma prática cultural que pode ser observada em listas de discussão, revistas, livros e na troca de código-fonte nas redes. Quando o código-fonte está escrito em um livro, em uma revista ou em um fórum *online*, e não necessariamente sendo executado em uma máquina, ainda é código-fonte e ainda é uma prática cultural de produção de *software*.

Essa visão abrangente do *software* como prática cultural proposta por Cramer oferece uma base interessante para a presente pesquisa e para o estudo do *Processing*. Se o *software* não é somente uma sequência abstrata de linhas de comando sendo executadas em uma máquina, como explica Roberts, ou diferenças elétricas de voltagens, como afirma Kittler, mas uma prática cultural inserida em um contexto social como defende Cramer, para compreender o *software* na arte é necessário ir além do resultado estético da obra e pensar em toda a sua conjuntura. O *Processing* é uma linguagem de programação e uma ferramenta de desenvolvimento de *software* para artistas e designers. Porém, é também parte de um contexto de facilitação da programação, como será visto no Capítulo 2, e apresenta uma cultura colaborativa de programação, de interação e de troca *online* influenciada pela cultura das comunidades de *software* livre e pela cultura *hacker*, como será examinado no Capítulo 3. Logo, pode-se dizer que o *software* escrito em *Processing* faz parte de uma prática cultural.

O *software* está presente no cotidiano da sociedade, porém as pessoas estão majoritariamente usando programas criados por programadores profissionais em vez de estarem criando os seus próprios programas. Isso os torna consumidores de *software*. Todavia, o domínio total do meio deveria incluir a possibilidade de escrever ferramentas próprias e para isso é necessário saber programação. Segundo Rushkoff (2010, p.7, tradução nossa), aprender programação é uma questão de alfabetização digital:

> quando seres humanos adquiriram a linguagem, nós aprendemos não somente a ouvir mas também a falar. Quando ganhamos a alfabetização, aprendemos não somente a ler mas também a escrever. E na medida em que entramos numa realidade digital cada vez maior, nós precisamos aprender não somente a usar programas, mas a como fazê-los.

Os programas que utilizamos no dia-a-dia apresentam interfaces que facilitam o uso do computador. O surgimento da interface gráfica de usuário – GUI (*Graphic User Interface*, em inglês) – na década de 1960 inaugurou possibilidades de interação entre seres humanos e computadores. Aliás, interfaces de interação com computadores não cessam de aparecer. O *Kinect*, lançado pela Microsoft para o videogame Xbox em 2010, é um exemplo de nova interface que permite que o jogador interaja com o computador através dos movimentos do seu corpo. A pulseira *FuelBand* da Nike, lançada em 2012, é um exemplo de interface vestível (um dispositivo *wearable*) que monitora desde o número de passos até a qualidade do sono do usuário. Interfaces como essas, no entanto, acabam "escondendo" o real funcionamento da máquina (KITLER, 1995; RUSHKOFF, 2010). O entendimento de como funciona o computador é frequentemente substituído por processos mágicos, como é o caso do *Wizard* (palavra inglesa que significa mago) do Windows, que auxilia na instalação e na configuração de programas. Atualmente, a indústria da tecnologia cria produtos cada vez mais "mágicos". Essa tendência se acentua com a utilização dos serviços de computação na nuvem em que os usuários nem ao certo sabem o que são os programas que usam, nem onde eles realmente se encontram fisicamente. Dentro dessa perspectiva, Machado (2010, p.46, grifo do autor) destaca que

> somos, cada vez mais, operadores de rótulos, apertadores de botões, "funcionários" das máquinas, lidamos com situações programadas sem nos darmos conta delas. Pensamos que podemos escolher e, como decorrência disso, nos imaginamos criativos e livres, mas nossa liberdade e nossa capacidade de invenção estão restritas a um software, a um conjunto de possibilidades dadas a priori e que não podemos dominar inteiramente. Esse é justamente o ponto em que a *Filosofia* [da Caixa Preta] de Flusser quer intervir: ela quer produzir uma reflexão densa sobre as possibilidades de criação e liberdade numa sociedade cada vez mais programada e centralizada pela tecnologia.

Como assinalado na citação acima por Machado, o pensamento de Flusser em *A Filosofia da Caixa-Preta* fornece uma base conceitual valiosa para a reflexão sobre a interação entre o homem e os aparelhos tecnológicos produtores de sentido. Na obra, escrita em 1985, Flusser utilizou a fotografia como objeto de análise para expor uma teoria sobre aparelhos e as possibilidades de interação com eles.

Segundo Flusser, existem três períodos históricos distintos: a pré-história, a história e a pós-história. Costa (2008) explica que a pré-história seria o tempo das imagens e da magia. A história estaria relacionada ao texto, linear, com relações de causa e efeito e que teria proporcionado o pensamento racional e científico. E por último, a pós-história seria o momento atual, das imagens técnicas, produzidas por uma combinação de texto e imagens, pois são criadas por dispositivos inventados com base no texto científico – os aparelhos.

Sem o pensamento científico não seria possível a invenção do aparelho, como, por exemplo, a câmera fotográfica. Desse modo, as imagens técnicas são produtos indiretos do texto, pois são intermediadas por aparelhos. Tais imagens são, por conseguinte, a expressão dos conceitos científicos, ainda que não de maneira explícita, pois sem esses conceitos elas não existiriam. Não são imparciais. São conceitos científicos transformados em imagens e projetam um determinado sentido à realidade.

Diferente de outros meios, como a pintura ou a escultura, a fotografia é feita com um dispositivo tecnológico que chega às mãos do fotógrafo programado por seus criadores. Esse entendimento sobre a programação original do aparelho é crucial. Quem utiliza o aparelho é, sem dúvida, capaz de produzir imagens diversas, porém "o fotógrafo somente pode fotografar o fotografável, isto é, o que está inscrito no aparelho" (FLUSSER, 1985, p.19). Em relação a esse ponto, Orben (2013, p.121) expõe que

> considerando que a programação do aparelho já define previamente, a partir de determinados conceitos, as imagens produzidas, então as escolhas possíveis sempre estarão já determinadas. A liberdade expressa nas imagens técnicas, por mais astuciosas e inovadoras que elas possam ser, sempre vai ser apenas uma liberdade de escolha, nunca uma verdadeira liberdade. Sua escolha não é livre, mas sim uma escolha programada.

Quem utiliza o aparelho é chamado por Flusser de funcionário. Essa pessoa, apesar de estar usando o dispositivo e acreditando ter o domínio sobre o que está produzindo, é, na verdade, parte da engrenagem necessária para a execução do programa do aparelho e, por isso, o termo funcionário. Em decorrência dessa ideia surge a questão sobre quem está usando quem: se é o fotógrafo quem domina o aparelho ou se é o aparelho que domina o fotógrafo:

> o fotógrafo domina o *input* e o *output* da caixa: sabe como alimentá-la e como fazer para que ela cuspa fotografias. Domina o aparelho, sem no entanto, saber o que se passa no interior da caixa. Pelo domínio do input e do output, o fotógrafo domina o aparelho, mas pela ignorância dos processos no interior da caixa, é por ele dominado (FLUSSER, 1985, p.15, grifo do autor).

Flusser apresenta, então, o conceito de caixa-preta: um aparelho que pode ser utilizado sem a compreensão do seu funcionamento – ou do seu programa. A câmera fotográfica, programada pelos seus criadores, pela fábrica, pelo parque industrial, pela economia e pela sociedade, nas mãos de meros funcionários, nada mais é do que uma caixa-preta que domina ao se simular dominada.

Diante do pensamento de Flusser sobre a caixa-preta, é possível fazer uma analogia entre a câmera fotográfica e algumas interfaces digitais com as quais interagimos frequentemente. Assim como o fotógrafo somente pode fotografar o fotografável, nas interfaces digitais somente podemos produzir algumas das finitas possibilidades programadas. Um exemplo é o programa de editoração de imagens Photoshop, que apresenta filtros e efeitos pré-programados. Segundo Greenberg (2007), apesar das inúmeras possibilidades oferecidas pelo Photoshop, ainda assim são possibilidades limitadas; cada filtro, por exemplo, segue uma linha de raciocínio com o objetivo de produzir resultados previsíveis. Sobre os aplicativos para criação artística, Machado (2010, p.12) destaca que

> mesmo os aplicativos explicitamente destinados à criação artística (ou, pelo menos, àquilo que a indústria entende por criação), como os de autoria em computação gráfica, hipermídia e vídeo digital, apenas formalizam um conjunto de procedimentos conhecidos, herdados de uma história da arte já assimilada e consagrada. Neles, a parte "computável" dos elementos constitutivos de determinado sistema simbólico, bem como as suas regras de articulação e os seus modos de enunciação, é inventariada, sistematizada e simplificada para ser colocada à disposição de um usuário genérico, preferencialmente leigo e "descartável", de modo a permitir a produtividade em larga escala e atender a uma demanda de tipo industrial.

Segundo Rushkoff (2010), o mundo digital é tendencioso e nos leva a constantes escolhas pré-programadas, pois tudo tem que ser expresso através de uma linguagem simbólica e binária do sim-ou-não. Portanto, é necessário compreender o posicionamento tendencioso das máquinas para que se possa diferenciar a intenção do indivíduo da intenção dos programas. E é nesse sentido que o conceito de funcionário, ou de apertador de botões, pode ser transposto para o mundo digital para se pensar como os usuários interagem com as diversas interfaces programadas. E mais ainda, quando Flusser afirma que somente é possível fotografar o "fotografável", ele auxilia na compreensão de muitas das ferramentas digitais com as quais somente se consegue criar o que já está previamente programado. A liberdade de criação no mundo digital é, muitas vezes, uma liberdade programada.

De acordo com Flusser, escapar dessa programação seria o objetivo de fotógrafos experimentais, aqueles que buscam a liberdade ao jogar contra o aparelho. Ao usar o aparelho, buscam encontrar novas possibilidades não previstas no seu programa inicial:

> fotógrafos assim chamados experimentais; estes sabem do que se trata. Sabem que os problemas a resolver são os da *imagem, do aparelho, do programa e da informação*. Tentam, conscientemente, obrigar o aparelho a produzir imagem informativa que não está em seu programa. Sabem que sua práxis é a estratégia dirigida contra o aparelho (FLUSSER, 1985, p.41, grifo do autor).

De acordo com Machado (2005, 2009, 2010), em uma sociedade tecnológica, este é um dos papéis da arte: corromper o funcionamento dos aparelhos na recusa sistemática a se submeter aos seus projetos industriais, desafiando a sua lógica com o objetivo de evidenciar a sua programação e a dominação que exercem. Ao mesmo tempo em que

existem usuários funcionários, existem também artistas que buscam subverter o funcionamento do aparelho: os artistas experimentais.

Um exemplo relevante para ilustrar o conceito de artista experimental que se recusa a seguir o programa do aparelho é Nam June Paik, que já na década de 1960 explorava as possibilidades de expressão com o uso de televisores. Na obra *TV Cello* de 1964 (Figura 1), ele criou um instrumento para ser usado pela violoncelista Charlotte Moorman em uma performance artística[12]. O objeto utilizava três televisores e na medida em que a musicista tocava o instrumento, imagens dela mesma e de outros violoncelistas apareciam nas telas e sons eletrônicos eram produzidos. Na obra *TV Cello*, o televisor é apresentado com um uso diferente da sua função original, que é passar o conteúdo televisivo emitido pelas estações de TV.

Figura 1 – Charlotte Moorman performando com o *TV Cello* de Nam June Paik (1964).
Fonte: Lussac (2009).

[12] Vídeo disponível em: youtu.be/-9lnbIGHzUM (Acesso em 29 ago. 2015).

Uma prática do mundo do *software* que também exemplifica esse conceito são os *codeworks*, algoritmos performáticos escritos em código-fonte (executável ou não) que têm o objetivo de serem interpretados por pessoas e não necessariamente por computadores. Segundo Cox e McLean (2013), *codeworks* são um bom exemplo de uma prática de programação que combina lógica e poesia e que também tem a função de questionar a comum ideia de que código-fonte de computador somente serve para ser interpretado pelo *hardware*. O exemplo abaixo, do website *de programming.us*[13], mostra um programa de computador escrito na linguagem de programação Perl com o nome de *Prozac* (marca de um medicamento antidepressivo) que pode ser executado criando um *loop* infinito até o computador travar:

```
#!/usr/bin/perl
# prozac.pl
# it will greet your system to death.
# but you go down in a cheerful endless loop.

while (1) {
       print "Hello World!\n";
       system ("$0"); # this line replicates it.
}
```

Enquanto o *hardware* consegue executar o programa *Prozac*, o faz sem a capacidade de compreensão conotativa. Um programador, no entanto, é apto a entender a camada de crítica sobre a sociedade e sobre as consequências do uso de antidepressivos presente tanto nas linhas de comentários (que aparecem depois do caractere "#" e que são ignoradas pelo computador) quanto nas linhas de código executáveis. Dessa forma, o programa *Prozac* subverte o funcionamento "normal" do *software*, que é criar algo útil para o usuário, fazendo exatamente o contrário: travando o computador.

Uma outra prática de programação que desafia o aspecto funcional do *software* como entidade abstrata que funciona nos bastidores do computador são as linguagens de programação esotéricas – linguagens experimentais que são criadas e utilizadas por diversão, e não para o uso prático. Ainda de acordo com Cox e McLean (2013), as linguagens de programação esotéricas são capazes de demonstrar a possibilidade de expressão cultural na prática de programação. As instruções na linguagem de

[13] deprogramming.us

programação esotérica Piet, por exemplo, são dadas pela relação de cores entre cada célula do programa, colocando o código-fonte em posição protagonista e deixando o *output* do programa em papel coadjuvante. A Figura 2 apresenta o código-fonte de um programa que produz a frase "*Hello, world!*" em Piet, um exemplo de *software* visualmente mais interessante do que o resultado final. Tanto o programa em Piet na Figura 2 quanto o programa *Prozac* (apresentado acima) são exemplos de práticas culturais de *software* que subvertem a sua função mais comum: a de ser executado pelo *hardware* para produzir sentido no *output* do programa.

Figura 2 – Código-fonte escrito em Piet de Thomas Schoch, que imprime a frase *"Hello, world!"*.
Fonte: Cox e McLean (2013).

Com base nos exemplos acima mencionados, fica claro como o pensamento de Flusser fornece uma base conceitual valiosa para se refletir sobre a produção de sentidos através da tecnologia. No entanto, transpor todos os conceitos do livro *Filosofia da Caixa-preta* para o computador de uma maneira generalizada e inflexível, principalmente em uma análise sobre arte, pode ser incerto. Apesar de muitos de seus processos não serem visíveis e serem incompreensíveis para os usuários, o computador não é como uma câmera fotográfica que com um clique produz fotografia. O computador muitas vezes expõe o que ocorre em seus processos internos, mostrando mensagens de erro,

travando, sendo contaminando com vírus e revelando suas falhas ou *bugs*. Além disso, atualmente, o computador não é um único dispositivo. A computação se apresenta em formas diferentes tais como *smarthphones*, *tablets*, *laptops*, *desktops*, aparelhos de GPS, até os mais recentes dispositivos *wearable* e a internet das coisas. E esses dispositivos dispõem de interfaces diversas e apresentam usos heterogêneos. O computador permite diferentes níveis de complexidade nas aplicações, desde interfaces e aparelhos simplificados, que conseguem gerar resultados instantâneos, até a elaboração de programas complexos através da escrita de *software*. Daí o argumento sobre a importância do aprendizado da programação (ou de pelo menos uma noção básica de programação) para uma real alfabetização no meio computacional – ideia defendida por diversos autores, como Casey Reas (2003, 2004, 2006, 2010) e Douglas Rushkoff (2010). A câmera fotográfica, por outro lado, é um aparelho que esconde sua complexidade (ótica, mecânica, química ou digital) e permite um uso simplificado por parte do usuário: produzir fotografias ao apertar um simples botão[14].

Essa ressalva é fundamental, principalmente em uma análise sobre arte feita em *software*, pois o artista, ao utilizar a produção de *software* como ferramenta de trabalho cria por meio da programação, que é a maneira como ele acessa e controla a máquina. Para McLean (2011), o artista que não cria a partir do código-fonte utiliza a tecnologia como uma caixa-preta, com foco somente no resultado final, enquanto que a arte criada em código seria aquela que tem o seu eixo central na cultura tecnológica. No entanto, uma visão dicotômica apresenta lacunas quando se busca compreender o universo do artista-programador, pois existem diferentes níveis possíveis de conhecimento e domínio da programação. Algumas linguagens que buscam facilitar a programação, como é o caso do *Processing*, podem ser usadas tanto por um cientista da computação quanto por um artista que acaba de iniciar seu aprendizado. Portanto, ao se refletir sobre arte feita em programação e sobre o *Processing* torna-se extremamente útil presumir que existe um espectro contínuo de possibilidades entre os dois extremos: o do funcionário e do artista experimental, como abordado na Seção 3.3, *Do artista-funcionário ao artista-programador*.

[14] Flusser expõe o conceito sobre sistemas complexos com usos simples e sistemas simples com usos complexos em uma entrevista de 1988, disponível em: youtu.be/lyfOcAAcoH8 (Acesso em 17 jul. 2015).

## 1.2 Software na Arte

Com o surgimento do computador, alguns artistas começaram a se interessar pelo novo dispositivo como ferramenta de criação e de expressão. A maioria das pessoas que se envolveram na criação dos primeiros projetos de arte computacional eram engenheiros e cientistas, pois tinham acesso a computadores em laboratórios de pesquisas e em universidades. Entre alguns desses pioneiros estão os matemáticos Georg Nees e Frieder Nake, na Alemanha, e o engenheiro Michael Noll, nos Estados Unidos.

As primeiras obras de arte feitas com o computador envolviam o processo de criação de um algoritmo por um programador, pois no início da computação não havia programas para a criação artística. Na verdade, já era um desafio produzir imagens impressas com o computador, pois o dispositivo não tinha sido projetado para esse fim. Desse modo, é interessante ressaltar que a arte computacional no seu início era uma espécie de aventura com o objetivo de fazer o computador produzir resultados que não eram parte do seu projeto inicial. Frieder Nake (2005, p.56, tradução nossa), um dos principais pioneiros da arte computacional, deixa claro o desafio de fazer o computador desenhar, na década de 1960, quando criou as suas primeiras obras:

> nunca antes na minha vida eu soube que um computador podia desenhar. Ele podia calcular e era bastante eficiente nisso. Mas desenhar – como? A tarefa parecia ser um desafio que surgiu por acaso. Eu o aceitei e isso mudou minha vida.

A aleatoriedade como processo de criação artística era muito presente nas primeiras obras de arte computacionais. Isso significava que parte das decisões sobre a obra não era feita pelo artista, mas pelo computador. Na Figura 3 é possível observar o recurso de aleatoriedade em três obras computacionais da década de 1960: *23 Verticles,* de Georg Nees (1965), *Vertical-Horizontal number 3* – parte de uma série de imagens produzidas com linhas verticais e horizontais –*,* de Michael Noll (1964) e *Random Polygon,* de Frieder Nake (1965).

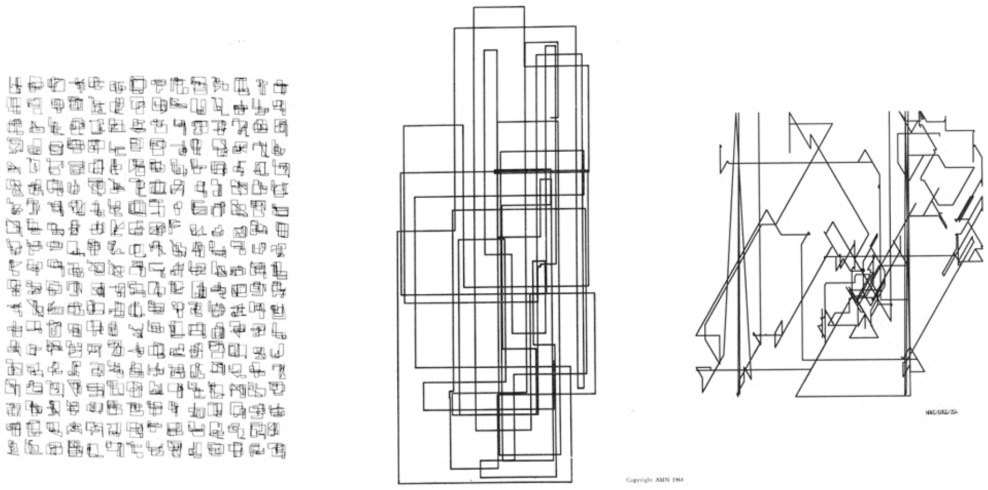


Figura 3 – Uso do recurso de aleatoriedade nas primeiras obras computacionais. Esquerda: 23 Vertices (G. Nees, 1965); Centro: Vertical-Horizontal no. 3 (A.M. Noll – © AMN 1964); Direita: Random polygon (F. Nake, 1965). Fonte: Nake (2005).

No final da década de 1960, alguns outros artistas, como Manfred Mohr e Vera Molnar, adotaram o computador como meio de expressão e a arte computacional começou a crescer nos Estados Unidos e na Europa. Foi também nessa época que começaram a surgir exposições e eventos dedicados à arte computacional, como, por exemplo, a exposição *Cybernetic Serendipity* realizada em Londres no ano de 1968.

Também nessa época a arte computacional surgiu no Brasil. O mais famoso pioneiro da arte computacional em território brasileiro foi Waldemar Cordeiro. Artista plástico participante do movimento da arte concreta no Brasil, Cordeiro foi líder do Grupo Ruptura, fundado em 1952, e curador da exposição Arteônica, realizada em São Paulo no ano de 1971 na Fundação Armando Álvares Penteado (FAAP).

Em 1968, ele iniciou pesquisas sobre arte computacional em parceria com o físico Giorgio Moscati, pesquisador da Universidade de São Paulo. A colaboração entre o artista e o físico durou cerca de dois anos e contou com um grande intercâmbio de conhecimento e experiência dos campos de trabalho de cada um. A dupla buscava reinventar o uso do computador produzindo arte:

> a abordagem girava sempre no sentido de perceber as possibilidades de cada técnica para gerar novas formas de expressão artística, fugindo sempre do simples uso de uma nova técnica para substituir uma técnica antiga sem renovar a mensagem (MOSCATI, 1993, p.9).

Apesar de Cordeiro ter sido uma das principais lideranças da arte concreta no Brasil e da tendência geométrica da arte computacional que naquela época era produzida na Europa e nos Estados Unidos, suas obras no computador não seguiam uma estética geométrica. A primeira obra de Cordeiro em parceria com Moscati foi o Beabá, um gerador aleatório de palavras com sonoridade de palavras reais da língua portuguesa – interessante notar o uso do recurso da aleatoriedade também nesse caso. A segunda obra, *Derivadas de uma Imagem* (Figura 4), é uma releitura de imagens através do computador. Segundo Moscati (1993), Cordeiro queria produzir com o computador uma imagem com forte conteúdo emotivo, já que a máquina era vista como fria e calculista. Cordeiro e Moscati criaram para cada imagem original três imagens derivadas. A cada derivada, o nível de definição e semelhança com o original diminui.

Figura 4 – Derivadas de uma Imagem de Waldemar Cordeiro (1969), transformação em grau 0 (Esquerda), grau 1 (Centro) e grau 2 (Direita). Fonte: Moscati (1993).

O conceito de reprodutibilidade técnica, de Benjamin (2012), nos auxilia a compreender a obra de Cordeiro e Moscati, pois o que o artista e o físico fizeram foi utilizar a própria possibilidade de reprodução técnica para criar a obra. É a reprodução técnica das imagens no *Derivadas de uma Imagem* que permite a linguagem artística das

versões subsequentes com menor definição. Para Benjamin, a reprodutibilidade técnica inaugurada pela fotografia no século 19 colocava em xeque a questão da autenticidade da obra, pois antes da fotografia, a obra de arte pertencia a locais físicos específicos, como museus ou igrejas. Para usufruir de uma obra era necessário estar com ela no lugar dela – no "aqui e agora". No entanto, a reprodutibilidade técnica da fotografia abalou essa tradição de autenticidade da obra de arte, pois contrapunha-se à sua unicidade. Segundo Seligmann-Silva (2009, p.15, grifo do autor), "solta do tempo e do espaço, a reprodução pode circular livremente. O preço dessa liberdade é o *fim da autenticidade* da obra: enquanto reprodução ela não é mais *algo idêntico a si*".

Dessa maneira, é interessante observar como *Derivadas de uma Imagem* levanta o debate sobre a reprodução técnica de uma imagem ao criar imagens altamente emotivas com o computador, subvertendo o senso comum de que a reprodução tecnológica é fria e calculista. Além disso, o conceito de cópia e original são desafiados nessa obra. Qual das três imagens é a original? Ou é a original a fotografia inicial que foi utilizada para produzir essas três imagens? No *Derivadas de uma Imagem*, essas perguntas são quase incoerentes, pois os conceitos de cópia e original não estão presentes. Todas as imagens são cópias e originais ao mesmo tempo. Aliás, no computador de uma maneira geral, o conceito de original perde parte de seu sentido porque todas as versões são originais idênticos, ou são todas cópias idênticas. Na verdade, ao criar algo no computador, automaticamente ocorrem duplicações do arquivo seja na memória, em arquivos temporários, na rede ou na nuvem. Logo, podemos afirmar que tudo que é criado digitalmente já é cópia, pois é reproduzido em *bits* em vários locais.

Um outro nome importante na arte computacional no Brasil, porém menos conhecido, foi Erthos Albino de Souza. Erthos era engenheiro da Petrobrás quando os computadores chegaram ao Brasil e logo se especializou em operar as novas máquinas. De acordo com Paros (2014), interessado por poesia e por ter uma boa renda, ele financiou (inteira ou parcialmente) alguns trabalhos de poesia concreta de Augusto de Campos, Décio Pignatari e Haroldo de Campos, na década de 1970. Também editou a revista *Código*, importante publicação de poesia concreta no país, que era um espaço de experimentação para ousadas posturas artísticas. Erthos, inclusive, foi um

experimentador e para isso utilizava o seu instrumento de trabalho: o computador. Apesar de não ter publicado muitos dos seus poemas, alguns podem ser encontrados em edições das revistas *Código*, *Polem*, *Poesia em Greve*, *Corpo Estranho*, *Artéria* e *Atlas*. Um exemplo da poesia computacional de Erthos, o poema *Sousândrade*, é apresentado na Figura 5.

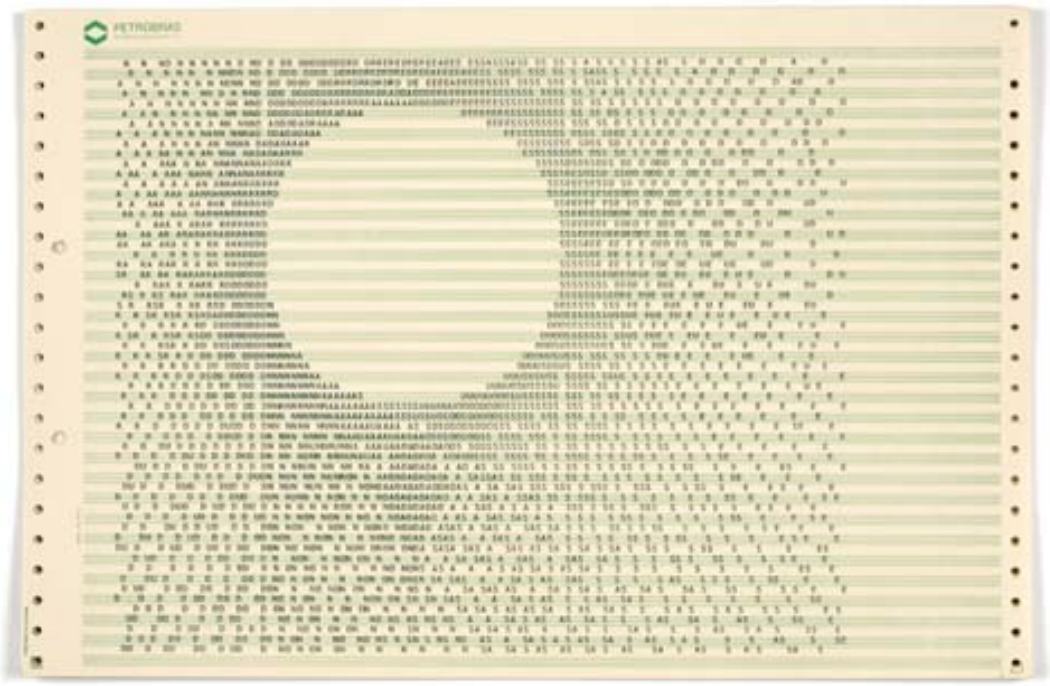


Figura 5 – Poema Sousândrade de Erthos Albino de Souza. Fonte: Folder (2010).

A estética espacial do poema computacional *Sousândrade* de Erthos revela a relação de proximidade entre arte digital e poesia concreta nos primeiros experimentos de arte computacional no Brasil. Essa afinidade não é casual. Tanto a estética da arte concreta quanto a estética da arte digital trabalham com a quantificação dos elementos da obra de arte, ou seja, com o processamento numérico da obra. Cordeiro (1993, p.25) destaca que essa relação acontece devido ao processo de criação artística:

> a arte computadorizada, enquanto metodologia, se identifica, em última análise, com as tendências da arte contemporânea chamadas, genericamente ‘‘construtivas’’ e que visam à quantificação e à digitalização dos elementos da obra de arte.

A proximidade entre arte digital e poesia concreta é importante para a reflexão sobre como a linguagem computacional existente na arte pode ser anterior à própria criação que utiliza o computador. Segundo Paros (2014), Erthos, por exemplo, experimentou com métodos computacionais – no sentido literal de computação – para calcular manualmente alguns poemas antes de usar o computador. Contudo, mais do que isso, mesmo antes do surgimento do computador, em diversas épocas e sociedades é possível encontrar exemplos de estética e linguagem computacionais e algorítmicas na poesia, na música e na arte (BARRIÈRE, 2010; CRAMER, 2005).

É fato que a tecnologia amplifica as possibilidades de linguagem, mas a computação em si não é dependente nem posterior ao computador. O computador é uma máquina de computar, mas a computação não foi inventada com o computador. Não é o objetivo desta pesquisa realizar um levantamento histórico detalhado das linguagens artísticas algorítmicas antes do computador e sua influência na arte computacional, nem fazer uma análise sobre a história do *software* antes do *software*[15]. No entanto, para ilustrar a amplitude do tema e assinalar possíveis caminhos para uma investigação histórica sobre arte computacional vale expor alguns exemplos.

Um caso do uso de aleatoriedade antes da invenção do computador são os jogos de dados para composições musicais. Segundo Barrière (2010), o primeiro exemplo conhecido de uso de dados para compor é de 1787 e atribuído a Mozart, os chamados *Musikalisches Würfelspiel*[16]. Ao jogar dados o compositor selecionava trechos musicais, que depois eram colocados em uma sequência a fim de criar uma composição. Com os mesmos trechos, o compositor podia montar diferentes combinações. A afinidade dessa técnica de composição com uma estética computacional é tamanha que é possível encontrar recentes aplicações multimídia que simulam o processo, como é o caso do CD-ROM criado em 2001 por Christoph Reuter (Figura 6).

[15] Cf. CRAMER, 2005.
[16] Vídeo disponível em: https://youtu.be/Jtpb1DkPx7I (Acesso em 10 jul. 2015).


Figura 6 – *Musikalisches Würfelspiel* de Christoph Reuter (2001), CD-ROM com adaptação do jogo musical de dados para um interface multimídia. Fonte: Wergo (2015).

Esse exemplo de jogo de composição de Mozart e sua versão multimídia expõem como a linguagem artística computacional não é determinada exclusivamente pelo aparato tecnológico. A concepção de que uma nova tecnologia inaugura uma nova linguagem artística, apesar de comum, é na verdade uma visão simplificada da relação entre arte e desenvolvimento tecnológico. Se o que chamamos de *software* é uma série de instruções que são computadas para se executar algo, então é possível chamar o jogo de composição de dados de Mozart de *software* ou de algoritmo.

Uma outra técnica muito comum em arte computacional é a criação de padrões visuais por meio de repetição. Apesar de muitas vezes associarmos esse tipo de estética com a arte feita no computador, é possível encontrar padrões visuais baseados em repetição em diversas sociedades, em diferentes épocas e lugares. Na arte islâmica, por exemplo, esse tipo de método de criação de imagens está muito presente e pode ser observado na Figura 7.

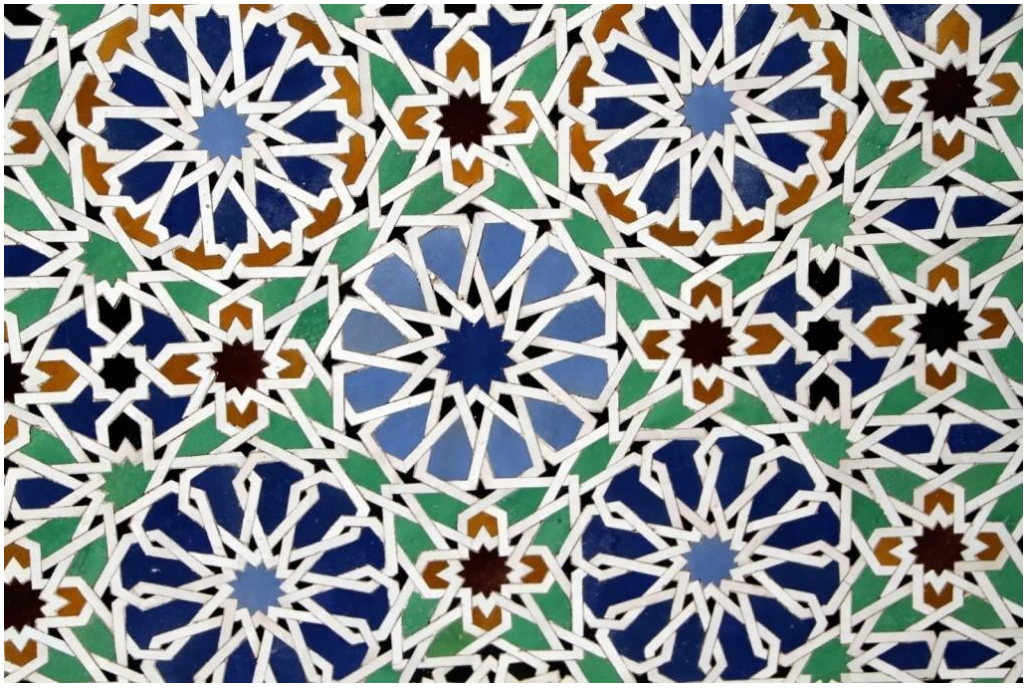


Figura 7 – Padrão visual de repetição em azulejo islâmico. Fonte: Islamic (2015).

Padrões visuais baseados em repetição são criados a partir de regras e, por isso, são também facilmente criados no computador a partir de algoritmos – como mostra a Figura 8. Assim sendo, na arte criada a partir de *software* padrões visuais se tornam uma forma artística bastante interessante a ser explorada, como explicam Reas e McWilliams (2010, p.59, tradução nossa):

> todos os padrões visuais e mosaicos no seu núcleo são compostos por algoritmos. Até mesmo padrões que existem há séculos, como os tartãs escoceses, seguem regras de composição rígidas que podem ser codificadas em software. Escrever código é uma maneira estimulante de abordar padrões visuais.

Entretanto, apesar de existirem formas artísticas de diferentes culturas e de diferentes épocas que apresentam semelhanças estéticas à arte criada em *software* tal qual conhecemos atualmente, a linguagem artística computacional contemporânea não é isenta de inovações. Ela não é consequência direta do computador, mas a tecnologia possibilita novas formas de expressão e de experimentação. Assim como seria um discurso radical afirmar que a estética computacional é completamente consequência do desenvolvimento tecnológico, seria um discurso simplista declarar que a arte

desenvolvida em *software* já tinha sido feita antes da computação no século 20. Existem raízes antes da invenção do computador e desdobramentos posteriores à tecnologia computacional.

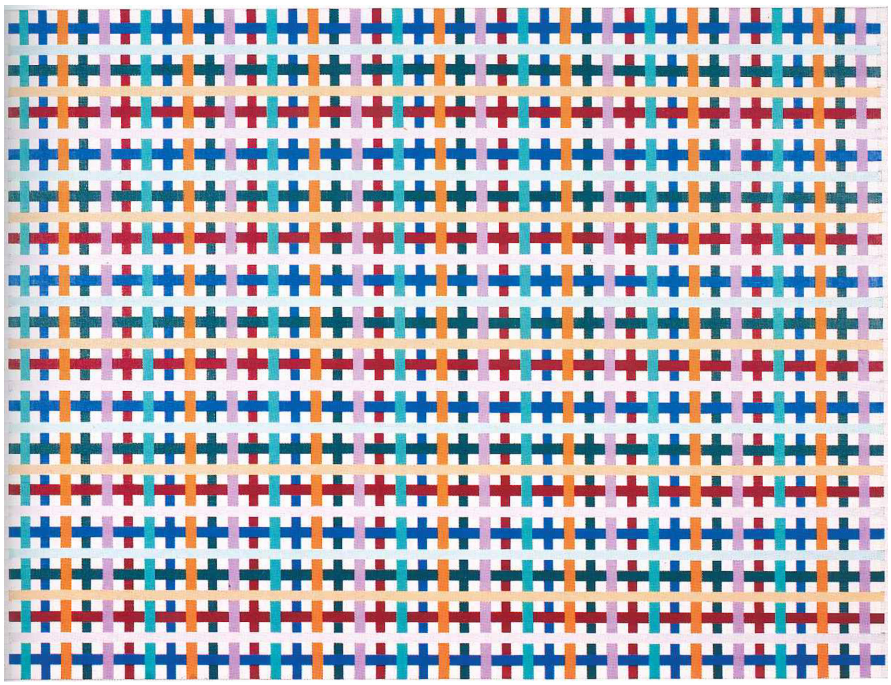


Figura 8 – Padrão visual de repetição na arte computacional. *Painting #207* de Vasa Minich (2004). Fonte: Reas e McWilliams (2010).

Uma técnica dependente do computador, por exemplo, é a geometria fractal, que trata de formas não contempladas na geometria euclidiana e que abriga o conceito de que todas as formas da natureza contêm dentro de si cópias menores de si mesmas. Mandelbrot (1993), matemático que criou a geometria fractal, explica que a arte fractal não tem como ser dissociada do computador, pois ela não seria possível antes da existência do *hardware* e do *software,* na década de 1970. A dependência não se dá por ser impossível calcular tudo manualmente, mas devido ao tempo necessário e à magnitude da empreitada. A Figura 9 mostra uma obra da artista Mary Thornton submetida à uma competição de arte fractal em 2007, a *Fractal Art Contest*.

Figura 9 – Arte fractal; *Golden Slope* de Mary Thornton (2007). Fonte: Thornton (2007).

Um outro exemplo de obra de arte criada a partir de *software* é a obra *Electric Sheep*[17], de Scott Draves (Figura 10), de 1999. *Electric Sheep* é um *software* que recria o fenômeno biológico da evolução através da matemática. É um sistema distribuído que roda nos computadores de mais de 450.000 colaboradores, que guiam o "processo de evolução" das imagens produzidas (*sheeps*) votando nas que mais gostam. Cada *sheep* apresenta uma árvore genealógica, pois, em seu processo colaborativo de evolução, vai se transformando em novas *sheeps.* O projeto foi lançado em 1999 e continua a se desenvolver (DRAVES, 2015).

*Eletric Sheep* utiliza fractais e a colaboração de participantes na internet para criar uma animação contínua. É uma obra de arte que se utiliza da colaboração *online* e do *software* como métodos de criação, assinalando possibilidades artísticas interessantes com o uso da internet. Quando instalada em exposições, a animação produzida por *Eletric*

[17] Vídeo disponível em: youtu.be/jVD67pMdv9k (Acesso em 10 jul. 2015).

*Sheep* é projetada nas paredes e seus efeitos visuais se tornam o foco da obra. Contudo, o processo de criação da obra, ou seja, o fato de ser um *software* que cria formas visuais que se reproduzem através de "cruzamentos digitais" com a ajuda de mais de 450.000 colaboradores na internet é sua característica mais interessante. Outro aspecto a ser assinalado é que o resultado – as imagens – são expostos nas telas dos colaboradores como *screensavers*, como se fosse uma enorme exposição distribuída.

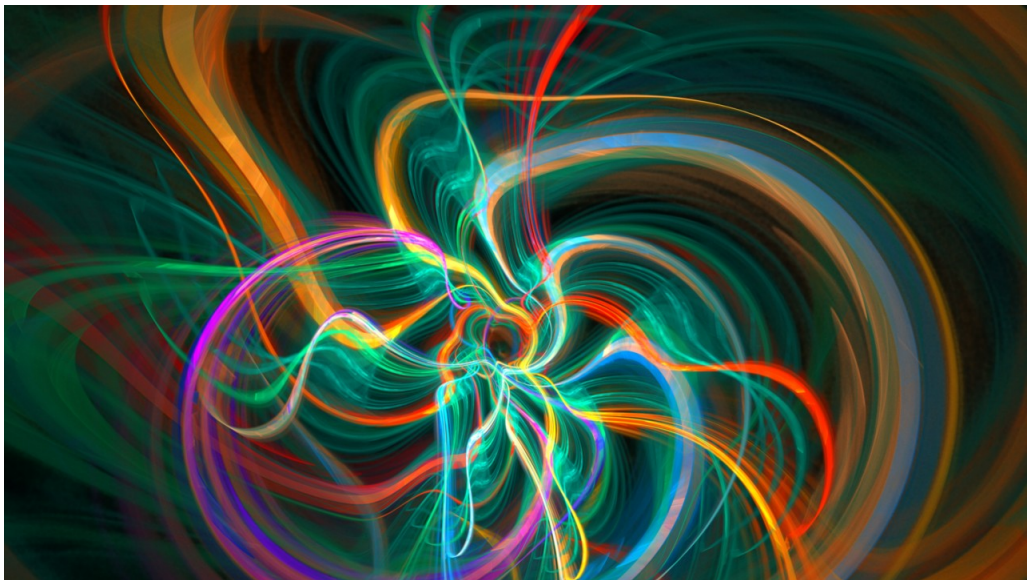


Figura 10 – *Electric Sheep - Generation 245 - Sheep 2706*. Fonte: Electric (2015).

Jarbas Jácome, artista brasileiro, expôs em junho de 2015, na cidade de São Paulo, a instalação *Vitalino*[18] (Figura 11) no Festival FILE. A obra é uma proposta interativa na qual o visitante pode fazer uma escultura virtual com as mãos. O artista (VITALINO, 2015) explica que "a escultura é sintetizada por processamento de imagem através de uma função de intersecção das extrusões das silhuetas dos dedos capturados por cada câmera e desenhada utilizando-se o conceito de voxel, isto é, pixel em três dimensões". *Vitalino* é uma instalação de arte interativa que, através da tecnologia, coloca o visitante como cocriador da escultura virtual.

[18] Vídeo disponível em: youtu.be/LvlbDbMsgfQ (Acesso em 10 jul. 2015).

Não faltam exemplos de artistas no mundo todo que utilizam *software* como material de criação. Alguns nomes importantes são: Aaron Koblin, Casey Reas e Ben Fry (criadores do *Processing*), Golan Levin, Harold Cohen, Amit Pitaru, Daniel Rozin, além dos brasileiros Fernanda Viegas, Fabrizio Poltronieri, Gilbertto Prado, Suzete Venturelli, entre outros.

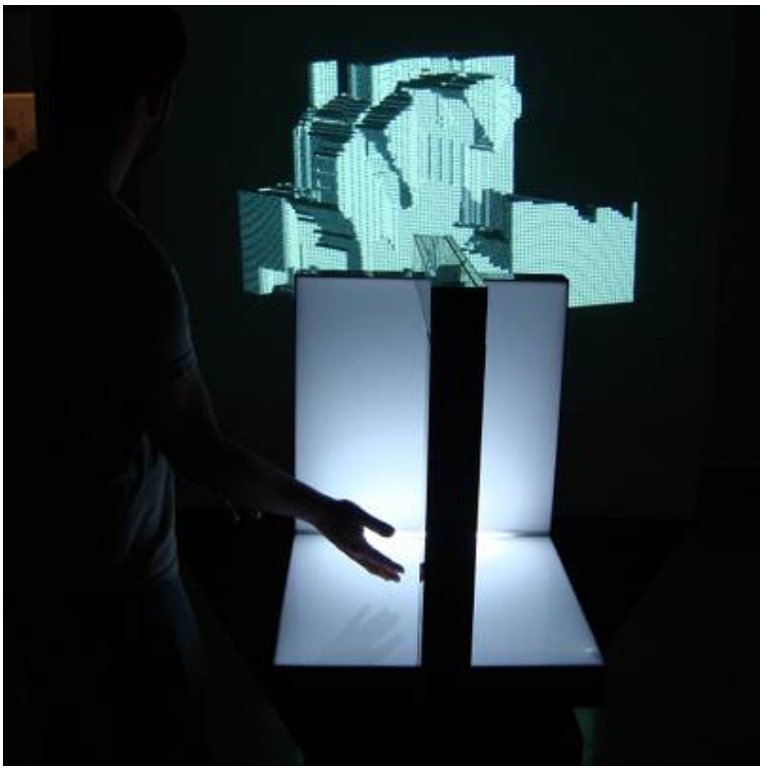


Figura 11 – Instalação *Vitalino*, de Jarbas Jácome (2010). Fonte: Vitalino (2015).

# 2 Programação e Arte

## 2.1 O artista-programador

Apesar de o *software* ser inerente a toda arte produzida ou reproduzida digitalmente, ele é comumente esquecido como um material artístico e um fator na estética da obra. Segundo Cramer e Gabriel (2001), isso acontece devido à progressão do uso dos computadores, que passaram de máquinas que só podiam ser utilizadas por programadores a interfaces gráficas, que camuflam o código que está realmente operando o computador – como foi discutido anteriormente.

Há, no entanto, artistas que trabalham diretamente com linguagens de programação, utilizando o código-fonte como material artístico. Esse tipo de artista que não utiliza somente ferramentas prontas, mas que trabalha diretamente no algoritmo da arte computacional, é chamado de artista-programador por alguns autores, como Caetano (2010), McLean (2011) e Neto (2010). Este último (p.16) esclarece que

> o artista programador é aquele que ao explorar a software arte investiga os processos de criação com foco principalmente na poética artística da reconstrução dos algoritmos. Permite-se inserir no código-fonte a sua poética, ou sobre a execução do código e as imagens e processos artísticos automaticamente gerados, ou ainda sobre o uso das linguagens de programação e a estética visual gerada na construção das interfaces computacionais, mas em qualquer abordagem o código e a construção algorítmica é [sic] parte da poética e do processo criativo da obra.

Paros (2014) utiliza o termo computador-artista para se referir ao poeta Erthos Albino de Souza, pois para ele, Erthos não era apenas um engenheiro comum, "era também um engenheiro-poeta: dentro de sua cabeça conviviam o programador e o poeta, e estes se uniram com o objetivo por demais nobre de 'enganar' o computador, para que este fosse poeta também, mas sem o saber" (p. 94). Assim, em uma reflexão sobre a contribuição de Erthos para a arte computacional no Brasil, não se pode deixar de assinalar o seu papel fundamental como artista-programador. Erthos e Waldemar Cordeiro foram importantes nomes da arte computacional no Brasil, contudo, enquanto

Cordeiro contava com a parceria com o físico Moscati para criar sua obra, Erthos trabalhava programando sua poesia computacional diretamente no computador. Desse modo, Erthos pode ser considerado um precursor dessa geração atual de artistas nacionais que se apropria do *software* e do *hardware*, que experimenta, cria e convive com os aspectos de artista e de programador no seu processo criativo.

> Juntamente com Waldemar Cordeiro, Erthos pode ser considerado um dos grandes pioneiros da arte por computador no Brasil e foi, provavelmente, o primeiro a se aventurar em experimentos com a função poética da linguagem utilizando-se de máquinas dessa natureza, em uma época em que os poucos computadores existentes em território nacional ocupavam salas inteiras e estavam confinados em algumas poucas indústrias e universidades, no geral inacessíveis tanto para o artista como para o poeta. Por tudo isso, foi em grande parte através de suas mãos que veio a nascer a primeira geração de "computadores-artistas" no Brasil (PAROS, 2014, p.92).

Todavia, pensar que um artista é capaz de ser também programador pode, por vezes, causar um estranhamento ou soar como um conceito inusitado. Por que a necessidade de denominá-lo artista-programador e não somente artista, colocando ênfase na técnica utilizada? Uma chave para essa questão pode estar nos conceitos de arte, técnica e tecnologia. Normalmente, o termo "programador" é utilizado para denominar uma atividade técnica e objetiva e, de maneira geral, arte e técnica são tratadas como capacidades de pouca compatibilidade, até mesmo com embasamento em pesquisas científicas sobre os lados do cérebro utilizados para cada tipo de atividade. Entretanto, talvez uma pista sobre essa questão esteja no conceito de *téchne* grega, termo que deu origem às palavras "técnica" e "tecnologia" [19], como explica Machado (2009, p.182-183):

[19] A palavra *técnica* tem sua raiz etimológica na palavra grega *téchne*, que significa técnica, arte ou ofício, e a palavra *tecnologia* tem origem nas palavras gregas *téchne* e *lógos* – esta última significa razão, discurso ou pensamento (DICTIONARY, 2015). Assim, tecnologia tem sua origem na combinação dos conceitos de *téchne* e *lógos*, sendo, então, uma sistematização de uma arte, de um ofício ou de uma técnica. De acordo com Oliveira (2008), o surgimento do conceito de tecnologia tem relação direta com a com a ascensão da burguesia no século 18, com a Revolução Industrial e com o emprego do conhecimento científico para a elaboração de técnicas de produção em larga escala aplicadas ao capitalismo. Gama (1985) esclarece, no entanto, que não é raro, o uso da palavra tecnologia para se referir ao modo de se fazer alguma coisa, a um conjunto de técnicas, sem necessariamente apresentar um recorte geográfico e temporal, emprego da palavra muito comum entre autores de língua inglesa.

> sabemos, por exemplo, que a palavra grega *téchne*, da qual deriva tecnologia, se referia a toda e qualquer prática produtiva e abrangia inclusive a produção artística. Os gregos não faziam nenhuma distinção de princípio entre arte e técnica, e esse pressuposto atravessou boa parte da história da cultura ocidental, até pelo menos o Renascimento (Dufrenne, 1980, p.1965). Para um homem como Leonardo da Vinci, pintar uma tela, estudar a anatomia humana ou a geometria euclidiana e projetar o esquema técnico de uma máquina constituíam uma única atividade intelectual.

Na citação de Machado exposta acima, fica claro que nem sempre arte e técnica foram tratadas como conceitos apartados. A separação entre elas está muito atrelada ao surgimento da visão romântica do artista no final do século 18. No romantismo, a arte passou a se referir à subjetividade e à vida interior, enquanto que a técnica passou a ser percebida como mecânica e objetiva (MACHADO, 2009). Com uma visão de mundo centrada no indivíduo, nas emoções subjetivas, no sonho e na fantasia, o conceito romântico da arte se opunha à racionalidade e à objetividade. Essa dicotomia tem reflexos até hoje. Não é raro o pensamento de que um engenheiro não tem habilidade para a arte ou de que um artista não sabe matemática. De acordo com Cramer (2005), a separação entre o que é técnico e o que é a inteligência humana subjetiva, ou entre o "gênio" e o "engenheiro", abriu caminho para as controvérsias que ainda persistem sobre a arte e até que ponto ela pode ser formalizada e automatizada. Daí o estranhamento que pode ocorrer em relação ao conceito de artista-programador, esse artista que domina tanto a estética quanto a técnica computacional.

Um exemplo interessante do século 16 para pensar sobre como a criação de arte e o domínio de técnicas, que atualmente pertenceriam ao campo das Ciências Exatas, podem estar conectadas são os perspectógrafos de Albrecht Dürer (1471-1528) – máquinas para facilitar a percepção da perspectiva (Figura 12). De acordo com Flores (2007), Dürer começou a estudar pintura artística aos quinze anos e interessado pelos fundamentos teóricos da arte, dedicou-se também a pesquisar ótica e matemática. Após dominar esses conhecimentos, criou os perspectógrafos com o objetivo de facilitar o aprendizado da perspectiva por artesãos e artistas na Alemanha. Interessante notar o aspecto de facilitação do entendimento da perspectiva nas máquinas de Dürer e a semelhança de objetivo com a linguagem de programação *Processing.* Ambos os projetos apresentam a finalidade de facilitar a compreensão de conceitos matemáticos para

aprendizes ou pessoas sem experiência na área. Mais detalhes sobre o *Processing* serão apresentados no Capítulo 3.

Figura 12 – Perspectógrafo de Dürer de 1525. Fonte: Albrecht (2015).

Os perspectógrafos de Dürer eram máquinas de desenhar que englobavam aspectos tanto da arte quanto da matemática e da ótica, estando, portanto, muito mais próximos do conceito de *téchne* grega do que da visão romântica da arte, conforme discutido anteriormente. Dürer não era somente um engenheiro capaz de criar uma máquina para aprimorar o entendimento da perspectiva ou apenas um pintor. Ele era os dois, assim como Erthos era tanto poeta quanto programador.

Harold Cohen é outro artista-programador que trabalha com computação desde a década de 1970. Cohen criou no início da sua carreira como artista digital o programa AARON, que utiliza princípios de inteligência artificial para pintar. Cohen programou o AARON para desenhar diferentes formas, desde abstrações até formas figurativas, como elementos naturais e humanos, como é possível observar na Figura 13, que mostra o programa AARON em execução, e na Figura 14.

Figura 13 – Programa AARON de Harold Cohen criando uma pintura. Fonte: ACM (2014).

Figura 14 – Obras produzidas com o programa AARON de Harold Cohen. Em cima: 040502 (2004) e embaixo: *Two Friends with Potted Plant* (1991). Fonte: Cohen (2004) e Cohen (1995).

É pertinente comparar os perspectógrafos de Dürer e o programa AARON de Cohen, pois cada um utiliza técnicas ou tecnologias próprias de seus tempos para auxiliar no processo artístico: a perspectiva no século 16 e a computação no século 20. De certa maneira, e preservando as diferenças inclusive de objetivos entre os dois projetos, ambos são máquinas de desenhar e ambos têm relação com o conceito de *téchne*, que engloba tanto arte quanto técnica no fazer artístico.

Na obra de Cohen, a parceria com o programa AARON é sua essência por mais de quatro décadas e exemplifica um grande debate existente na arte computacional: a questão da autoria. Apesar de o debate sobre autoria não ser foco desta pesquisa, é válido ressaltar a tensão gerada por essa relação criativa entre homem e máquina. Uma parte do processo o artista domina, mas a outra parte a máquina realiza "independentemente". A questão da automatização do processo artístico muitas vezes desperta questionamentos sobre a validade da obra de arte. Como esclarece McLean (2011, p.118, tradução nossa), essa é uma questão recorrente:

> a perene pergunta sobre autoria está sempre conosco: se o computador produz arte, quem a criou: o humano ou a máquina? As opiniões sobre a criatividade através da programação de computadores tendem a polos opostos, com a total negação da criatividade de um lado e com excêntricas afirmações sobre uma criatividade libertária na arte generativa do outro.

Uma possível resposta para a questão da autoria requer examinar a arte computacional a partir da seguinte divisão: 1) a arte criada a partir de programas já prontos e 2) a arte criada a partir do código escrito pelo artista-programador. Segundo McLean (2011), na arte criada a partir de programas prontos a autoria seria compartilhada com os desenvolvedores do programa, pois o artista não tem domínio ou conhecimento sobre os algoritmos que estão sendo utilizados no processo de criação, enquanto que na arte criada a partir do *software*, o artista seria autor tanto do *software* quanto do resultado final.

Na citação a seguir, Nake (2014, p.108, tradução nossa) expõe a importância da criação do algoritmo para a autoria da arte computacional e argumenta que o trabalho automático de produção da máquina perde a relevância quando se enfatiza o papel do algoritmo:

> arte algorítmica começa com a criação de um algoritmo. Isso é trabalho humano. O processo generativo termina em um objeto material, como tinta no papel. Isso é trabalho da máquina. O que a máquina realiza é uma possibilidade de potencialmente infinitos conjuntos de partes. O artista, entretanto, descreveu um conceito: toda a classe dessas partes. Dependendo do poder expressivo dos parâmetros contidos na descrição algorítmica, as diferenças na aparência visual das partes podem ser imensas. Não há limites para a nossa capacidade descritiva.

Apesar da importância do debate sobre autoria e dos diferentes níveis de controle do artista-programador quando usa programas já prontos e quando cria o seu próprio, não existe uma maneira mais apropriada de se criar arte com *software*. Ao trabalhar com *software*, cada artista tem suas preferências sobre quais programas ou quais linguagens de programação usar. Algumas são mais simples e fáceis de usar e outras mais complexas, permitindo um maior controle da máquina. Porém, em qualquer caso, para ser capaz de criar um algoritmo no computador, é necessário ter alguma noção de programação e saber utilizar alguma linguagem de programação.

## 2.2 Linguagens de programação alternativas

O material de trabalho do artista-programador é o *software*. Na pintura, na escultura e na música, diferentes tipos de tintas, de materiais e de instrumentos musicais produzem resultados artísticos diversos. Tinta a óleo e aquarela, por exemplo, produzem pinturas distintas. No trabalho do artista-programador com o *software*, diferentes linguagens de programação possibilitam diferentes resultados estéticos:

> pode ser útil pensar em cada linguagem de programação como um material com qualidades e limitações únicas. Diferentes linguagens são apropriadas dependendo do contexto. Algumas linguagens são fáceis de usar, porém ofuscam o potencial do computador, e outras linguagens são muito complicadas, mas proporcionam controle total fornecendo acesso completo à máquina. Por exemplo, algumas linguagens de programação são flexíveis e outras são rígidas. Linguagens flexíveis como Perl e Lingo são boas para criar programas curtos rapidamente, mas quando o programa se torna longo elas frequentemente se tornam difíceis para manter e compreender. Programar com linguagens rígidas como 68008 Assembly ou C exige um enorme cuidado e uma atenção entediante ao detalhe, porém os resultados são eficientes e robustos. Da mesma maneira que as madeiras Pinho e Carvalho são diferentes, programas de software escritos em diferentes linguagens também têm gestalts estéticas distintas (REAS, 2003, tradução nossa).

É importante ressaltar a relevância da explicação acima feita por Reas sobre como diferentes linguagens de programação permitem processos e resultados variados. Segundo Cox e McLean (2013), as linguagens de programação podem ser analisadas como linguagens humanas. E sendo linguagens, seus limites são também os limites do mundo: o que nós não conseguimos pensar, não conseguimos falar; a extensão do nosso mundo depende da potência da nossa linguagem. Dessa maneira, do mesmo modo que alguns conceitos são melhor expressados em certos idiomas, dentre as linguagens de programação também existem diferenças entre o que é possível realizar ou expressar com cada uma delas.

Logo, ao se refletir sobre arte computacional e sobre o *software* como material de trabalho do artista-programador, é importante também considerar as linguagens de programação. Falar de *software* de uma maneira generalizada para explicar o trabalho do artista-programador não é suficiente, pois seria uma visão simplificada do seu processo de criação. De acordo com Greenberg (2007), as linguagens de programação são o instrumento de trabalho do artista-programador, a matéria-prima da computação, o lugar onde ele pode colocar a mão na massa.

Muitos artistas digitais são, de fato, engenheiros, cientistas da computação e programadores profissionais, como é o caso de muitos dos pioneiros da arte computacional mencionados no Capítulo 1. No entanto, também existem artistas digitais com formações variadas, inclusive em Artes e outras áreas das Ciências Humanas, que começam a programar com linguagens projetadas para tornar a programação uma

atividade possível para pessoas com formações não técnicas. Muitas iniciativas desse tipo vêm surgindo desde a década de 1960 e, entre elas, encontram-se linguagens criadas para diferentes públicos, como artistas ou crianças. E é nesse exato contexto que o *Processing* se encontra. Assim sendo, para compreender os aspectos culturais do *Processing* relacionados ao mundo da programação, é relevante contextualizá-lo em relação a outras iniciativas de facilitação da programação, inclusive do ponto de vista histórico.

O *Processing* não é primeira e nem a mais recente linguagem de programação alternativa, porém é a primeira linguagem voltada para as artes visuais que foi amplamente adotada. Os programadores dessas linguagens têm formas de pensar diversas que nem sempre se enquadram na forma de pensar dos programadores das linguagens de programação clássicas da Engenharia da Computação e das Ciências Exatas, as quais costumam ser textuais e lineares. Entretanto, é fundamental ressaltar que o domínio da programação não é algo banal. Aprender programação exige esforço e dedicação e, mesmo nas linguagens alternativas, programar continua a ser uma atividade complexa. Não obstante, essas linguagens facilitam a entrada no mundo da programação para pessoas com interesse no assunto mas sem uma educação formal na área.

O debate sobre aprendizado de programação é extremamente atual. Diferentes projetos e organizações estão surgindo com o objetivo de ensinar programação, como é o caso da Code Academy[20], um projeto norte-americano que ensina programação *online* e que conta com o apoio de Douglas Rushkoff; da Code.org[21], uma organização norte-americana dedicada à ampliação do ensino de programação nas escolas como parte do currículo obrigatório; e do britânico Code Club[22], uma rede de clubes de programação para crianças, que conta com uma versão brasileira desde 2013, o Code Club Brasil[23]. Segundo Schachman (2012, tradução nossa), está surgindo uma nova geração de programadores alternativos com diferentes interesses:

[20] codecademy.com
[21] code.org
[22] codeclub.org.uk
[23] codeclubbrasil.org

> esses programadores "alternativos" são pessoas que não se identificam como programadores, mas que regularmente programam computadores para alcançar seus objetivos. Programadores alternativos podem incluir músicos, performers, escritores, artistas visuais, designers, cientistas e ativistas.

É significativo notar como essa definição de Schachman sobre programadores alternativos é diversificada. Além de atividades relacionadas ao mundo das artes, ela engloba também cientistas e ativistas. Isso reflete a relevância e a expansão do *software* na nossa sociedade, como explorado no Capítulo 1. Na citação abaixo, Reas elucida a importância do aprendizado de programação para o artista que trabalha com *software*, mas não seria incoerente ampliar esse argumento para englobar também cientistas e ativistas como indicado acima por Schachman.

> Na minha opinião, todo artista utilizando software deveria ser alfabetizado em software. O que alfabetização significa no contexto do software? Alan Kay, um inovador no pensamento sobre o computador como um meio, escreveu: A habilidade de 'ler' um meio significa que você pode acessar os materiais e as ferramentas criadas pelos outros. A habilidade de 'escrever' em um meio significa que você pode gerar materiais e ferramentas para os outros. Você precisa dos dois para ser alfabetizado. Na escrita, as ferramentas de que você precisa são retóricas; elas demonstram e convencem. Na escrita computacional, as ferramentas que você gera são processos; elas simulam e decidem (REAS, 2003, tradução nossa).

Indo além, é possível argumentar que aprender programação, ou pelo menos os seus conceitos fundamentais, também é importante na educação de crianças e adolescentes. Como defende Rushkoff (2010) e como foi abordado no Capítulo 1, a sociedade atual roda em *software* e aprender programação é uma questão de alfabetização. Já existem alguns exemplos dentro e fora do Brasil de inclusão do ensino de programação nas escolas, como parte da grade curricular. A matéria publicada na *Folha de São Paulo* em maio de 2014 (RICHTEL, 2014), "Programação vira disciplina em escolas infantis nos EUA", traz a notícia da inclusão do ensino da programação em diversos estados norte-americanos. No Brasil, a iniciativa Computação na Escola[24] do Departamento de Informática e Estatística (INE), da Universidade Federal de Santa Catarina (UFSC), defende o ensino da computação para crianças. Utilizando o *Arduino* e a

[24] computacaonaescola.ufsc.br

linguagem *Scratch*, a organização realiza oficinas para ensinar robótica para estudantes e seus pais. Em entrevista concedida ao *Diário Catarinense* (LUMINI, 2014), o coordenador do projeto, Aldo von Wangenheim, afirma que "o ensino tradicional de informática mostra o computador para a criança como um substituto à máquina de escrever e não como uma ferramenta para criar tecnologia, programas de computador e jogos".

Não se faz na presente pesquisa, porém, o argumento de que não deveria haver facilitação no uso do computador para pessoas não-técnicas ou muito menos se defende a ideia de que todos deveriam se tornar cientistas da computação e operar os computadores por linhas de comando. A questão central a ser debatida é a relevância do entendimento do meio em um mundo intermediado por *software*. Sem dúvida, é importante que as interfaces facilitem a utilização das máquinas, mas também é crucial que exista a compreensão do funcionamento delas e um acesso mais amplo para a "escrita" de programas, ou seja, uma alfabetização de código – termo popular em inglês: *code literacy*.

As linguagens de programação alternativas são interfaces de programação projetadas com a intenção de facilitar a atividade de programar. Então, apesar de as interfaces "esconderem" do usuário como o computador funciona, um argumento contrário às interfaces de maneira generalizada seria improdutivo. Na verdade, da mesma maneira que a criação de interfaces facilita o uso do computador, linguagens de programação alternativas facilitam a atividade de programar. Desse modo, muitas linguagens de programação alternativas, como o *Processing*, funcionam como portas de entrada para o mundo da programação. Depois de consolidado o interesse e alguns conhecimentos básicos, o programador alternativo – ou o artista-programador – pode se sentir mais confortável para explorar outras possibilidades mais complexas da computação ou até mesmo se aventurar em outras linguagens. Se algumas interfaces podem ser consideradas caixas-pretas, como analisado no Capítulo 1, as linguagens de programação alternativas, como o *Processing* – apesar de apresentarem um enorme potencial nas mãos de programadores experientes – podem ser consideradas algo como "caixas-cinzas": facilitam a criação do programa através de uma interface e, ao mesmo tempo, transformam o usuário – ou funcionário – em programador.

Uma importante referência histórica para o universo das linguagens de programação alternativas é o projeto *Dynabook* de Alan Kay e de sua equipe na XEROX PARC, iniciado na década de 1970[25]. O projeto serviu como inspiração direta tanto para os criadores do *Processing* quanto para os desenvolvedores da linguagem *Scratch*, que será examinada mais adiante (REAS; FRY, 2006; RESNICK et al., 2009). Kay explorava a possibilidade de que o computador pudesse ser uma ferramenta de criatividade utilizado na produção de música, animação, desenhos, poesia, entre outros – algo que na época não era tão comum quanto é hoje. O *Dynabook* era um projeto de *hardware* e *software* integrados, um novo computador, no qual o usuário – que poderia ser criança, artista ou adulto – tinha o controle da máquina.

O computador utilizava a linguagem de programação *Smalltalk* (Figura 15), a primeira linguagem orientada a objeto com uma interface gráfica, janelas, documentos integrados e editor com função de copiar e colar (SMALLTALK, 1980). Assim, o Dynabook tornava o computador mais do que uma máquina que realiza cálculos matemáticos com rapidez, pois colocava o computador no centro da produção criativa. Além disso, o computador podia ser reprogramado pelos próprios usuários, passando de uma ferramenta multiúso para uma máquina na qual era possível criar novas ferramentas. Segundo Manovich (2013), Kay e sua equipe na XEROX PARC reinventaram o computador, transformando-o em um sistema interativo para o pensamento criativo. O *Dynabook* trazia uma nova proposta de computador pessoal: um computador acessível que permitia que o usuário fosse também o programador.

A seguir são expostas cinco linguagens de programação alternativas a fim de contextualizar o *Processing*. São elas: *Logo*, linguagem educativa da década de 1960; *Scratch*, projeto educacional do MIT mais recente, de 2007; *Max* e *Pure Data*, linguagens voltadas para a produção musical, das décadas de 1980 e 1990; e *Arduino*, projeto de *hardware* livre inspirado no *Processing*, de 2005. Uma linha do tempo que mostra o histórico de criação dessas linguagens se encontra na Figura 16. Vale ressaltar que não é a intenção dessa linha do tempo apresentar todas as linguagens de programação alternativas existentes, mas mostrar de forma visual simples a cronologia das linguagens selecionadas para apresentar o contexto do *Processing*.

[25] Cf. KAY; GOLDBERG, 2003.

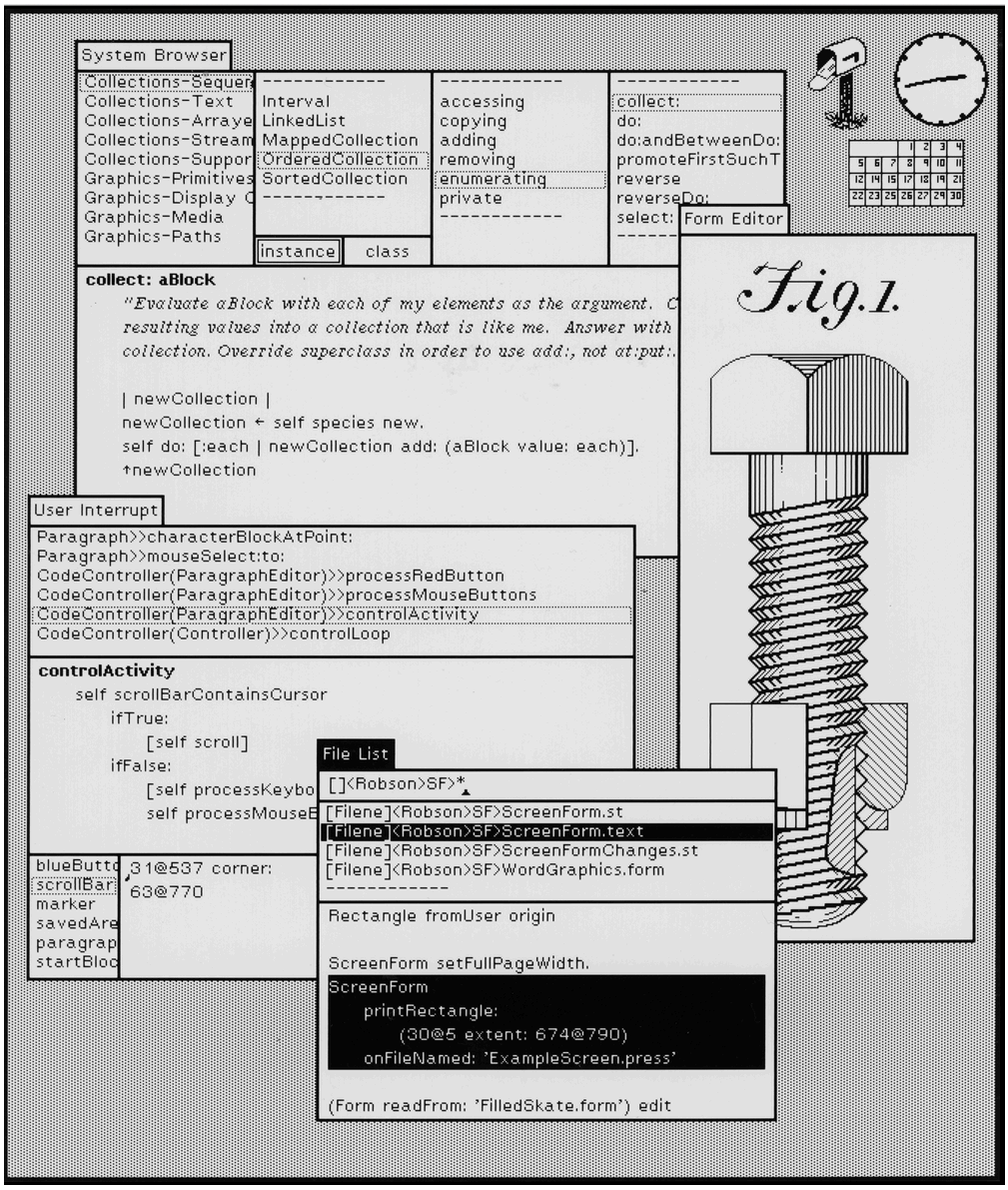



Figura 15 – Linguagem de programação *Smalltalk*. Fonte: Smalltalk (1980).

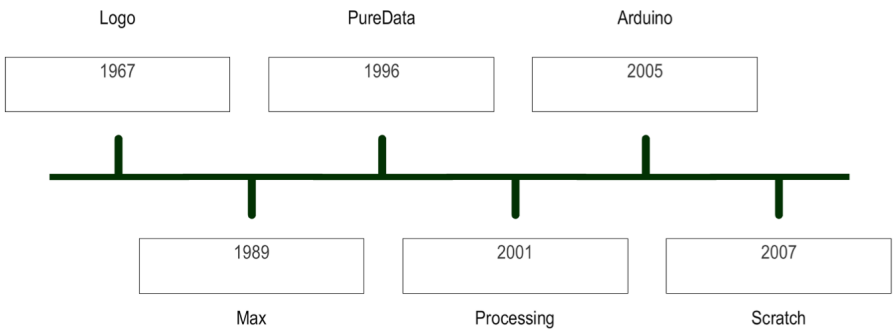



Figura 16 – Linha do tempo de linguagens de programação alternativas para contextualização do *Processing.* Criada pela autora.

### 2.2.1 *Logo*

A linguagem *Logo* foi desenvolvida no MIT e sua primeira versão foi publicada em 1967. O início do projeto ocorreu no *Artificial Intelligence Laboratory* do MIT e o objetivo era criar uma ferramenta para o ensino de matemática para crianças nas escolas, fazendo com que elas colocassem o conhecimento em prática usando a programação e que, desse modo, melhorassem o seu aprendizado (FEURZEIG et al., 1969). O ambiente de programação mais conhecido do *Logo* é o da tartaruga, que era inicialmente um pequeno robô controlado por comandos no computador, como pode ser observado na Figura 17.

Figura 17 – O robô "tartaruga" da linguagem *Logo*. Fonte: Papert (1980).

Mais tarde a tartaruga migrou para a tela do computador na versão mais conhecida da linguagem. Porém, o uso do *Logo* só aumentou significativamente com a chegada dos computadores pessoais, no final da década de 1970.

> O LOGO, desde a sua criação, por volta de 1967, até 1976, ficou confinado em alguns laboratórios, principalmente no Massachusetts Institute of Technology (MIT) e em outros centros, como o Departamento de Inteligência Artificial da Universidade de Edimburgo e o Instituto de Educação da Universidade de Londres. Isso porque os microcomputadores ainda não existiam, e o interpretador LOGO estava disponível somente para computadores de grande porte (NASCIMENTO, 2004, p.2).

Interessante notar, com base na citação de Nascimento, como a ideia de criar uma linguagem de programação alternativa para o público infantil e o intuito de facilitar a programação já estavam presentes na comunidade de pesquisadores da área da computação antes mesmo do surgimento do computador pessoal. Válido também ressaltar que, assim como a linguagem *Logo*, o *Processing* surgiu no MIT, estando, portanto, relacionado à tradição de criar linguagens alternativas que existe na instituição. Uma outra linguagem que surgiu no MIT com objetivos similares é o *Scratch*, que será analisado a seguir.

### 2.2.2 *Scratch*

O projeto *Scratch* foi desenvolvido pelo grupo de pesquisa *Lifelong Kindergarten Group* do MIT *Media Lab* e lançado em 2007, 40 anos depois do *Logo* e 6 anos depois do *Processing*. É um projeto criado sob uma licença de *software* livre e direcionado para o público infantil e adolescente, de 8 a 16 anos. Os criadores do *Scratch*, inspirados pelo trabalho de Alan Kay com o *Dynabook,* tinham o mesmo objetivo de Kay quando projetaram o *Scratch*:

> nós queríamos tornar fácil para todas as pessoas, de todas as idades, origens e interesses a programação de suas próprias histórias interativas, jogos, animações e simulações, e também o compartilhamento das suas criações com as outras pessoas (RESNICK et al., 2009, p.60, tradução nossa).

O *Scratch* engloba dois aspectos: a linguagem de programação *Scratch* e um site[26], no qual os programadores podem compartilhar seus projetos, comentar, olhar projetos de outras pessoas e aprender. Compartilhamento e *remix* estão na base do *Scratch*. O projeto não é somente uma linguagem de programação, mas um ambiente *online* para a formação de uma comunidade que troca experiências, que cria e que aprende a programar em conjunto. Resnick et al. (2009) explicam que o desenvolvimento da linguagem de programação *Scratch* está fortemente relacionado ao seu website. Segundo

[26] scratch.mit.edu

os autores, para obter sucesso, o *Scratch* precisa estar associado a uma comunidade de pessoas que apoiam o projeto, colaboram e trocam experiências.

É possível fazer um paralelo entre os aspectos de comunidade e de *software* livre no *Scratch* e as mesmas características no *Processing.* Como será analisado no Capítulo 3, o *Processing* também apresenta uma forte comunidade de programadores alternativos que compartilham códigos e aprendem em conjunto na rede. Além disso, as licenças de *software* livre e os aspectos culturais de abertura de informação, traços característicos da cultura *hacker*, também são semelhantes nessas duas linguagens. Desse modo, ambas as linguagens utilizam os mesmos três pilares: o objetivo de facilitar a programação para pessoas não-técnicas, *software* livre e cultura *online* de compartilhamento. Os últimos dois assuntos dessa tríade são debatidos na Seção 2.3, Cultura *hacker.*

A gramática do *Scratch* tem como inspiração o brinquedo Lego. Interessados pela facilidade com que as crianças faziam experimentações com os blocos do Lego, os pesquisadores do grupo *Lifelong Kindergarten* do MIT resolveram criar uma linguagem que também pudesse ser programada em blocos. Os blocos de código no *Scratch* têm formas e conectores diferentes que indicam como eles devem ser encaixados, como pode ser visto na Figura 18. Estruturas de controle como *forever* e *repeat* apresentam a forma da letra C, por exemplo, indicando que outros blocos devem ser posicionados dentro deles. Esse aspecto visual do *Scratch* tem o objetivo de facilitar o processo de experimentação com o código, como esclarecem Resnick et al. (2009, p.63, tradução nossa):

> a maioria das linguagens de programação (e cursos de ciência da computação) privilegiam o planejamento estruturado, do geral para o detalhe, em vez da experimentação aleatória. Com Scratch, nós queremos que fuçadores se sintam tão confortáveis quanto planejadores.

Normalmente, a programação é vista como uma atividade linear, em que no início da escrita do *software*, o programador já sabe qual é o seu objetivo final e, para atingi-lo, escreve o código de uma maneira estruturada. Por isso, na programação tradicional, desenvolvedores costumam construir o *software* com base em especificações claras e sem ambiguidades. No entanto, quando se trata de programadores alternativos tais como artistas-programadores ou crianças, o processo de criação do código envolve

experimentação aleatória e um constante fuçar por possibilidades sem objetivos claros. De acordo com McLean (2011), uma maneira de olhar esse processo de programação é através do conceito de programação de bricolagem:

> o bricoleur lembra um pintor que para entre pinceladas, olha para a tela e, somente depois dessa contemplação, decide o que fazer. Bricoleurs usam a maestria de associações. Para os planejadores, erros são passos equivocados; bricoleurs usam a navegação de correções no caminho. Para planejadores um programa é um instrumento para controle premeditado; bricoleurs têm objetivos porém buscam realizá-los no espírito de uma aventura colaborativa com a máquina (TURKLE; PAPERT apud MCLEAN, 2011, p.199, tradução nossa).

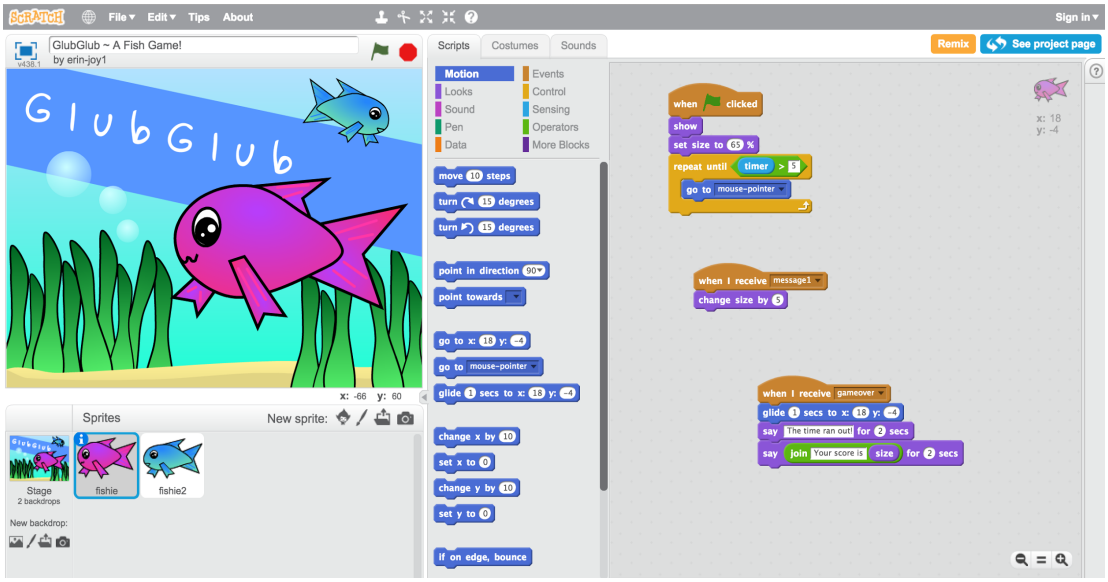



Figura 18 –Ambiente de desenvolvimento do *Scratch*. Fonte: GlubGlub (2015).

Esse mesmo processo é apontado por Banzi (2009) em relação ao *Arduino*, como será visto mais adiante. O argumento central é a diferença entre o processo de criação dos programadores alternativos de *Arduino* e de *Scratch* e o processo de desenvolvimento de *software* da computação clássica. O processo da programação alternativa é baseado na experimentação e o da computação clássica, na estruturação. Tanto para crianças quanto para artistas, a programação é uma atividade lúdica, na qual escrever o código é o meio e o resultado final o objetivo.

O *Scratch* é usado em mais de 150 países e disponível em mais de 40 idiomas. No momento de escrita da presente pesquisa[27] - são 10.134.503 projetos compartilhados, 7.123.021 usuários registrados e 51.301.052 comentários publicados no site. No Brasil, existem quase 150.000 usuários de *Scratch*, sendo o quinto país com maior número de usuários no mundo, depois dos Estados Unidos, Inglaterra, Austrália e Canadá (SCRATCH STATISTICS, 2015). Dentre as comunidades examinadas nesta pesquisa, a do *Scratch* é a que apresenta o maior número de participantes.

### 2.2.3 *Max* e *Pure Data*

*Max* é um ambiente gráfico de programação desenvolvido por Miller Puckette na década de 1980. É voltado para a composição musical e, principalmente, para performances ao vivo. O ambiente básico do *Max* (*Max*/*MSP*/*Jitter*) inclui MIDI (*Musical Instrument Digital Interface*), uma interface visual e objetos de controle de tempo. A interface de programação em *Max* utiliza caixas e linhas conectoras, que representam o fluxo e a lógica do programa como um esquema visual (Figura 19). É, portanto, uma alternativa à estrutura padrão de programação, que utiliza linhas de texto para a elaboração do programa.

Puckette (2002) explica que muitas das ideias que deram origem ao *Max* surgiram no *Experimental Music Studio* do MIT, no início dos anos 1980 – grupo que depois se tornou parte do MIT *Media Lab*. O projeto nasceu, no entanto, quando Puckette estava no centro de pesquisa musical IRCAM (*Institut de Recherche et Coordination Acoustique/Musique*), na França. Ele conta que tomou a decisão de não incluir algumas funcionalidades básicas de programação que não eram necessárias para a produção musical, exatamente por conta do objetivo de simplificar a interface e de torná-la mais fácil para músicos que não sabem programação. Dessa maneira, esclarece (1997) que o *Max* não foi projetado exatamente como uma linguagem de programação, mas sim como um linguagem visual de controle de diferentes módulos (*patches*), imitando a estrutura de um sintetizador analógico. Na década de 1990, Puckette criou uma nova linguagem

[27] 19 de julho de 2015.

(baseada no *Max*) também para performances ao vivo, mas desta vez *open source*: *Pure Data* (Figura 20).

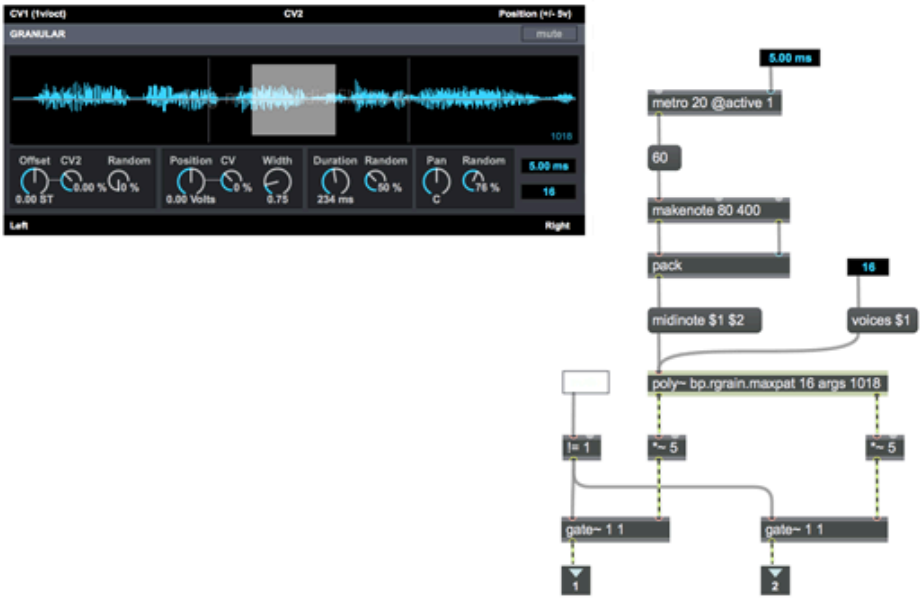



Figura 19 – Programação em *Max*. Fonte: Max (2015).

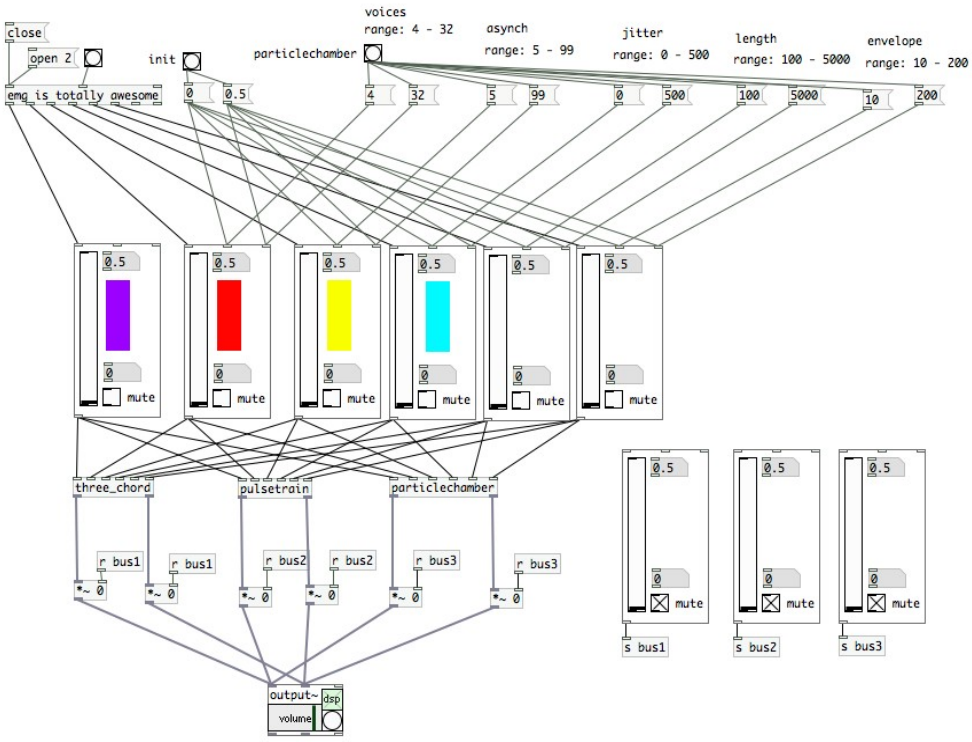



Figura 20 – Programa em *Pure Data*. Fonte: Ontonilux (2015).

*Max* e *Pure Data* são linguagens visuais de programação, que buscam facilitar a criação de programas por meio de elementos gráficos, caixas, linhas conectoras e diagramas. Ambas as linguagem se baseiam em uma gramática espacial, na qual o artista-programador consegue visualizar todo o fluxo do programa e fazer alterações em um processo contínuo de *feedback* com a máquina, como explica Schachman (2012, tradução nossa):

> comunicar no vai e volta em texto é uma experiência baseada na vez. Programador fala, computador fala, etc. Interfaces alternativas permitem um ciclo de feedback contínuo entre humano e computador. Por exemplo, cada um dos ambientes visuais baseados em patches permitem que o usuário ajuste parâmetros com sliders e vejam os resultados em tempo real.

Justamente por sua capacidade de ajuste em tempo real, *Max* e *Pure Data* são frequentemente utilizadas em performances de *live coding*, na qual o artista faz a programação ao vivo, como mostrado na Figura 21 – performance do artista Young Choi no *Museum Live Coding Show*, em Seul, em 2012[28].

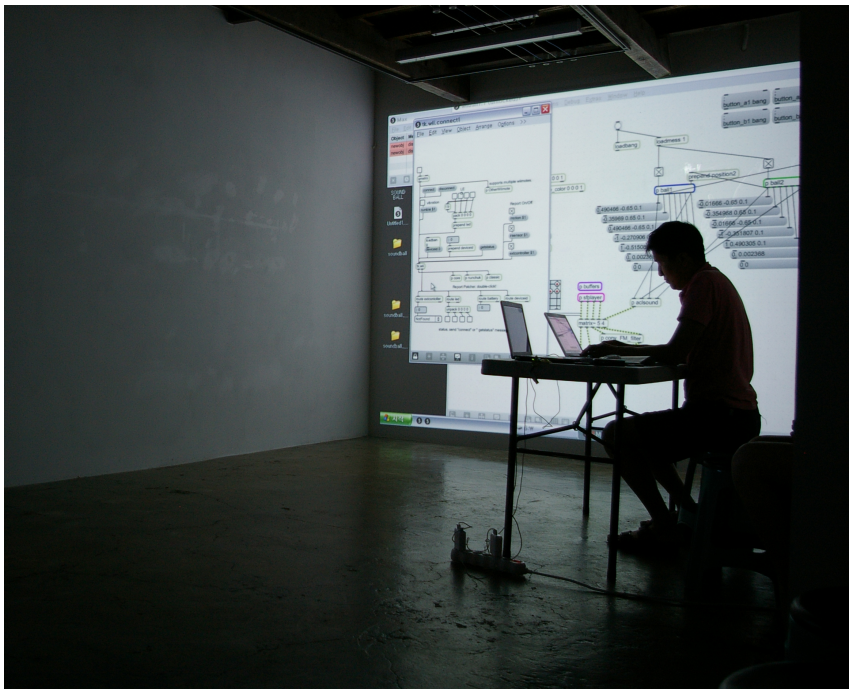


Figura 21 – Performance de *Live Coding* do artista Young Choi (2012). Fonte: Choi (2012).

[28] Vídeo disponível em: youtu.be/Ekx0BGOBkHw (Acesso em 20 set. 2015).

Apesar de não apresentarem uma relação direta com a maneira de se programar em *Processing*, as linguagens *Max* e *Pure Data* são relevantes para o contexto das linguagens de programação para as artes não só por terem sido anteriores ao *Processing*, mas também por apresentarem um ambiente de programação visual, diferente da programação clássica. Linguagens de programação visual, em que o programa é controlado pelo artista por meio de uma interface gráfica, exemplificam a diversidade de formas que a programação de computadores pode ter. A criação de novas linguagens de programação alternativas é primordial para engajar na programação pessoas com diferentes maneiras de pensar, como explicam Reas e Fry (2006, p.529, tradução nossa):

> muitas pessoas pensam que programação é somente para pessoas que são boas em matemática e em outras disciplinas técnicas. Uma das razões pelas quais a programação permanece restrita a esse tipo de personalidade é que pessoas que pensam dessa maneira normalmente criam linguagens de programação. É possível criar diferentes tipos de linguagens de programação que engajam pessoas com mentes visuais e espaciais.

### 2.2.4 *Arduino*

O *Arduino* é uma plataforma de prototipagem que foi projetada para a interação física entre o ambiente e o computador utilizando *software* e *hardware* livres e era, incialmente, destinado a artistas e designers para a criação de ambientes interativos, como instalações multimídia. Apesar de o *Arduino* não ser uma linguagem de programação e sim um microcontrolador (Figura 22), cabe nesta pesquisa a sua contextualização em relação ao *Processing*, pois, assim como o *Processing* e os outros projetos analisados anteriormente (*Logo*, *Scratch*, *Max* e *Pure Data*), apresenta o mesmo objetivo de levar a programação (e, no caso, a eletrônica) para fora dos muros da engenharia. Além disso, a sintaxe utilizada no *Arduino* foi inspirada no *Processing* e o ambiente de desenvolvimento do microcontrolador é uma implementação do ambiente de desenvolvimento do *Processing* (PROCESSING, 2015).

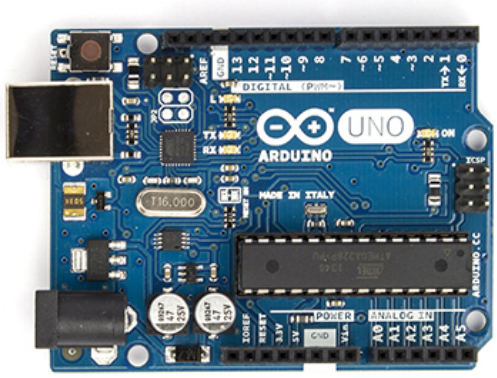



Figura 22 – Placa Arduino Uno. Fonte: Arduino (2014).

O projeto começou em 2005 na Itália, quando Massimo Banzi, professor do *Interaction Design Institute Ivrea*, procurava uma maneira de ensinar eletrônica para seus alunos do curso de design de interação. Desde o início, o *Arduino* teve como público-alvo artistas e designers. E assim como as outras linguagens vistas anteriormente, o *Arduino* foi concebido para permitir que não-engenheiros pudessem também programar microcontroladores. O processo de programação é, do mesmo modo que no *Scratch*, baseado na experimentação, como explica Banzi (2009, p.5, tradução nossa):

> a engenharia clássica confia em um processo rígido para chegar de A a B; o processo do Arduino se encanta com a possibilidade de se perder no caminho e criar uma alternativa que encontre C. [...] Esse é o processo de experimentação do qual gostamos tanto – brincar com o meio sem objetivos fixos e encontrar o inesperado.

Mais uma vez, podemos observar aqui a intenção de facilitar a programação para pessoas com formas diferentes de pensar – no caso do *Arduino*, facilitar a programação de microcontroladores. Assim como no *Scratch* e no *Processing*, no *Arduino* a experimentação sem objetivos fixos tem um papel central. Os focos são a criação, a arte, a imaginação e não a resolução lógica de problemas. Em vez de pensar sobre os *designs* e sobre as obras de arte, o *Arduino* incentiva os artistas a prototipar rapidamente suas ideias. "Nós acreditamos que é essencial brincar com a tecnologia, explorando diferentes possibilidades diretamente no *hardware* e no *software* – às vezes sem um objetivo definido" (BANZI, 2009, p.7, tradução nossa).

No *Arduino*, o aspecto da comunidade *online* também se repete. Segundo Banzi (2009), a colaboração entre usuários é um dos princípios fundamentais do *Arduino*. Para isso, existe um fórum no site do *Arduino*[29] e uma Wiki, chamada de *Playground*[30], onde pessoas de diferentes partes do mundo podem se ajudar, compartilhar e aprender em conjunto. O fórum oficial do *Arduino* tem, no momento de escrita desta pesquisa[31], 2.319.815 *posts* e 313.262 membros (ARDUINO FORUM, 2015). O *Arduino* também apresenta os mesmos três pilares presentes no *Scratch*: o objetivo de facilitar a programação para pessoas não-técnicas, *software* livre e cultura *online* de compartilhamento. Também como já foi assinalado, no *Processing* essa características estão igualmente presentes, como será visto em mais detalhe no Capítulo 3. A seguir são explorados aspectos da cultura *hacker*, principalmente o trabalho colaborativo e a abertura da informação.

## 2.3 Cultura hacker

A importância da cultura *hacker* no contexto do *Processing* não pode ser ignorada. A cultura *hacker* começou no MIT nas décadas de 1950 e 1960 e dela surgiu na década de 1980 o movimento pelo *software* livre, que posteriormente se ramificou no movimento pelo *software open source*[32]. Assim como os outros exemplos já vistos, o *Processing* é uma linguagem de programação *open source* e está inserido no contexto da cultura *hacker*. Nesta pesquisa, define-se linguagem de programação *open source*[33] como linguagem que faz parte do contexto do movimento pelo *software* livre ou do movimento pelo código aberto e que apresenta pelo menos partes da sua estrutura publicadas sob

[29] arduino.cc
[30] arduino.cc/playground
[31] 12 de setembro de 2015.
[32] Cf. RAYMOND, 1999.
[33] Existe uma diferença importante entre *software* livre e *software open source*, ou de código-aberto. Enquanto o *software* livre tem o objetivo de preservar liberdades de reuso do código, as licenças de código-aberto (*open source*) estão mais voltadas para o processo de produção e uso comercial, procurando estimular o desenvolvimento do *software* em comunidades abertas e defendendo a possibilidade de um posterior uso proprietário. Apesar de apresentarem diferenças importantes tanto do ponto de vista legal quanto do ponto de vista político, muitas vezes os dois termos são utilizados como sinônimos, inclusive nesta dissertação.

esses tipos de licença. Desse modo, para se construir o entendimento do contexto de programação em que o *Processing* se encontra, é primordial examinar esses aspectos da cultura *hacker*, estabelecendo também a base para a análise, no Capítulo 3, da programação de arte no *Processing*.

A notícia do site *G1*, publicada em março de 2014, traz a seguinte manchete: "SP pagará *hackers* para melhorar trânsito; salário é de até R$ 5,9 mil" (GOMES, 2014). A notícia explica como a prefeitura de São Paulo pretende, através da criação de um espaço para *hacker*s, nomeado Laboratório de Mobilidade, solucionar problemas de trânsito na cidade. Assim como nessa notícia, a cultura *hacker* tem tido menções positivas na mídia, não faltando exemplos do tipo, desde as *hackatons* organizadas por prefeituras para solucionar problemas urbanos até iniciativas de transparência política nascidas na sociedade civil como o movimento Transparência *Hacker*. Parece que já passou o tempo em que *hacker*s eram constantemente retratados na mídia com uma conotação negativa, como criminosos que praticam atividades ilegais na rede – para o qual o termo mais apropriado provavelmente seria *cracker*, como explica Castells (2003).

Em sintonia com a imagem positiva da figura do *hacker* está o crescimento do número de *hackerspace*s no Brasil e no mundo. *Hackerspaces* são espaços comunitários com recursos de tecnologia, onde pessoas com interesses em experimentação e fabricação se encontram e aprendem interagindo umas com as outras. Segundo Holm e Joseph (2014), a história dos *hackerspace*s tem origem no início do movimento *hacker*, sendo um exemplo o *Homebrew Computer Club*, que funcionou de 1975 a 1986 no Vale do Silício e no qual participava um dos fundadores da Apple, Steve Wozniak.

De acordo com a Hackerspace Wiki (LIST, 2015), existem 1.175 espaços do tipo ativos no mundo e, desde 2009, surgiram, somente no Brasil, 28 *hackerspace*s (Figura 23)[34]. Importante notar que esses dados são declarados pelos organizadores dos *hackerspace*s que se cadastram voluntariamente no site e não foi realizado nenhum tipo de filtro qualitativo nos dados, tendo sido apenas excluído um *hackerspace* que não forneceu a data de fundação.

[34] Dados acessados em 26 de julho de 2015.

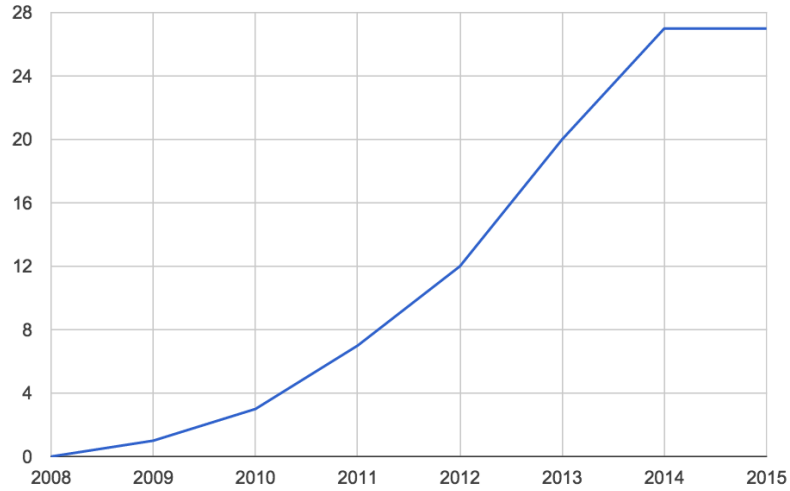



Figura 23 – Número de *hackerspace*s ativos no Brasil por ano. Gráfico criado pela autora com base nos dados disponíveis em List (2015). Lista completa de *hackerspaces* utilizada para a elaboração do gráfico disponível no Apêndice B.

Também se observa uma explosão de espaços semelhantes a *hackerspace*s, como *fab labs*, *makerlabs*, *media labs*, laboratórios de inovação e espaços de *co-working* não só no Brasil, mas no mundo, sendo a maioria deles focada na inovação (MAXIGAS, 2012). Apesar de esse tipo de ambiente ainda ser restrito, o crescimento do número de espaços no Brasil e no mundo é significativo, podendo-se concluir que há também um aumento do interesse pelo assunto e por esse tipo de atividade, além de uma popularização da cultura *hacker*. Nesta pesquisa, só se constata o fato de que existe um crescimento desse tipo de espaço, porém as razões por trás não são abordadas. Sem dúvida essa é uma questão interessante a ser investigada em pesquisas futuras.

A microeletrônica com *Arduino* é um assunto recorrente nos *hackerspace*s e o *Processing* é uma linguagem de programação comum nesses espaços, especialmente em projetos de artes visuais. No Garoa Hacker Club[35], em São Paulo, por exemplo, acontecem as Noites do *Arduino* e no SESC em São Paulo são realizadas oficinas de Computação Criativa com *Processing* e *Arduino* – que antes eram chamadas *HackLabs*[36]. Igualmente, o

[35] garoa.net.br

[36] De acordo com Maxigas (2012), os termos *hacklab* e *hackerspace* são comumente usados como sinônimos, porém existem diferenças históricas e políticas entre os dois que resultam em diferentes abordagens da tecnologia: os *hacklabs* surgiram de um contexto de ativismo midiático anticapitalista e os *hackerspace*s são centros de experimentação tecnológica voltados para a arte e para a inovação, sem necessariamente uma ideologia política. Cf. MAXIGAS, 2012.

festival *Campus Party*[37], que acontece no Brasil anualmente, oferece aos participantes espaços de experimentação tecnológica alinhados com a cultura *hacker,* como *hackatons* e oficinas.

Como já assinalado anteriormente, o *Arduino* é baseado no *Processing*, então é interessante ressaltar o papel do *Processing*, ainda que indireto, na popularização da microeletrônica e da cultura *hacker* nesses espaços que estão surgindo. A relevância do *Processing* nesse contexto é maior do que a linguagem original voltada para as artes visuais, pois o projeto teve um papel fundamental em criar uma estrutura de sintaxe e um modelo de ambiente de programação que foram replicados para o aprendizado da computação, indo além do seu objetivo inicial.

Para um *hacker,* o computador é sinônimo de diversão. Sua atividade está relacionada ao prazer de programar e de desvendar possibilidades no computador. Himanen (2001) explica que a ética de trabalho dos *hackers* é movida pela paixão. Os *hackers* não exercem o "hackerismo" por obrigação ou por ser seu trabalho, exercem porque são movidos por um entusiasmo criativo. A ética de trabalho dos *hackers* está mais relacionada a uma atividade de hobby ou de lazer do que a uma atividade econômica profissional. O sistema operacional *open source* Linux, por exemplo, não foi desenvolvido dentro de uma empresa, mas por uma rede de voluntários liderada por Linus Torvalds e coordenada através da internet no início da década de 1990. Esse espírito de diversão da atividade *hacker* pode ser relacionado aos processos de programação baseados na experimentação das linguagens de programação alternativas, como *Scratch*, *Arduino* e *Processing*, já abordados anteriormente. A criança ou o artista escrevem o código em uma atitude de brincadeira, uma atitude *hacker*, percorrendo as possibilidades do *software* e da máquina.

O *hacker* também é aquele que se apropria da tecnologia para criar algo novo, um inventor, como explicam Paixão, Menezes e Sganzerlla (2012, p.39-40, grifo dos autores) mostrando a origem do termo:

[37] brasil.campus-party.org

> o uso primário do verbo *to hack* vem associado à palavra cortar/entalhar e, como substantivo (the hacker), passou a ser aplicado ao trabalho talentoso de artesões [sic] da madeira. Com o passar dos tempos, o termo começou a ser associado ao ato de se esmerar em modificar ou inventar algo novo e é com essa acepção que ele foi apropriado pela primeira geração de hackers, na década de 50.

Quando se pensa na cultura *hacker* pelo prisma criativo da invenção de novas possibilidades tecnológicas, não é preciso percorrer um longo caminho para se pensar também na prática artística tecnológica como uma postura *hacker*. Um "artista-*hacker*" conecta sistemas, abre aparelhos, cria circuitos, solda placas, conecta sensores, reaproveita código de outros projetos, testa, quebra, conserta e assim segue em seu processo de criação artística.

Entretanto, a inventividade e a diversão com o computador não são os únicos princípios da cultura *hacker*. Segundo Levy (2010), que investigou os primeiros *hackers* que se reuniam em laboratórios do MIT, nas décadas de 1950 e 1960, antes mesmo do surgimento da internet, a ética *hacker* apresenta seis princípios que, apesar de não terem sido acordados formalmente, estavam presentes na cultura daquele primeiro grupo de *hackers*:

1. Acesso a computadores – e a qualquer coisa que possa ensinar algo sobre como o mundo funciona – deve ser ilimitado e completo. E sempre se deve dar preferência a um método prático de aprendizado.
2. Toda informação deve ser livre.
3. Não se deve confiar nas autoridades; devem-se buscar processos sem burocracia e descentralizados.
4. Meritocracia: *hackers* devem ser julgados pela sua capacidade de "hackear" e não por nível educacional, idade, etnia ou cargo.
5. É possível criar arte e beleza em um computador, sendo esse princípio relativo ao código em si e não ao que se produz com o código. *Hackers* buscavam a maneira mais simples e reduzida de se escrever um programa.
6. Os computadores podem tornar a vida melhor.

Dentre os seis princípios acima, destacam-se, principalmente, os dois primeiros, relativos à abertura da informação, para o contexto cultural do *Processing*. Na cultura *hacker*, tanto a informação que pode ser usada para o aprendizado quanto a informação que se produz devem ser abertas e devem estar disponíveis para aqueles que desejem acessá-las. Esse preceito é uma das principais chaves para se entender também o *software* livre, que surgiu dentro do contexto da cultura *hacker* na década de 1980. Quando a indústria do *software* começou a se consolidar e surgiu o conceito de *software* proprietário, Richard Stallman, um *hacker* que na época trabalhava no *Artificial Intelligence Laboratory* do MIT, criou a *Free Software Foundation* com o objetivo de manter aberta a informação sobre como o *software* funciona, ou seja, o código-fonte. No trecho abaixo do Manifesto GNU[38], documento escrito na fundação do movimento, Stallman (2003, p.546, tradução nossa) defende o seu direito e de todos os outros programadores de compartilhar o código com a comunidade:

> eu considero que, por princípio, se eu gosto de um programa eu tenho que compartilhar com outras pessoas que também gostam. Empresas que vendem software querem dividir os usuários e conquistá-los, forçando cada usuário a concordar em não compartilhar com outros. Eu me recuso a quebrar a solidariedade com outros usuários dessa maneira.

Esse direito defendido por Stallman é o direito de dividir a informação com os outros programadores, evidenciando um espírito comunitário da cultura *hacker* alimentado pelo compartilhamento de conhecimento. Porém, conhecimento aberto não é exclusivo da cultura *hacker* e do *software* livre, muito pelo contrário, é a base do funcionamento da academia e do processo científico. A ciência depende da abertura da informação, pois é compartilhando os achados e elaborando teorias coletivamente que se avança na pesquisa científica. Essa mesma abertura da informação teve um papel fundamental no desenvolvimento da internet. Como afirma Castells (2003, p.34), "a cultura da Internet é a cultura dos criadores da Internet". Muito do que se observa atualmente na rede em relação a colaboração e troca de informação *online* está

[38] O termo GNU deriva do nome do projeto liderado por Stallman para a elaboração de um sistema operacional livre em oposição ao Unix, que era proprietário: *GNU is Not Unix*, GNU não é o Unix.

relacionado à cultura inicial de abertura, a qual foi diretamente influenciada pela cultura acadêmica e pela cultura *hacker*.

A academia e o desenvolvimento científico dependem de um processo em que a comunidade de pesquisadores e cientistas fazem revisões e críticas aos trabalhos uns dos outros (a revisão por pares, ou em inglês, *peer review*), eliminando erros e garantindo o avanço da pesquisa. O modelo científico é descentralizado, pois, apesar de existirem autoridades em diversos assuntos, qualquer novo cientista está livre para desafiar teorias vigentes. E a cultura *hacker*, inclusive no que diz respeito ao *software* livre e *open source*, funciona da mesma maneira:

> tanto cientistas quanto hackers aprenderam na prática que a ausência de estruturas fortes é uma das razões que torna esse modelo tão poderoso. Os hackers e os cientistas só têm de concretizar suas paixões e trabalham em grupo com outros indivíduos com quem as compartilham. Esse espírito difere claramente do espírito encontrado em empresas e governos (HIMANEN, 2001, p. 72).

Enquanto que nos governos e nas empresas a ordem, a estrutura e o controle são aspectos organizacionais constantes, na cultura *hacker* e na ciência, a liberdade de horário e a flexibilidade do local onde se realiza o trabalho e a pesquisa são comuns. Um cientista ou um *hacker* não produz apenas em horário comercial, de segunda a sexta-feira, e não é raro se ouvirem relatos de pesquisadores e programadores que passam noites em claro em seus projetos. Essa flexibilização de estruturas organizacionais é algo que acontece atualmente em muitas empresas na área de tecnologia, bastante influenciadas pela cultura *hacker* e que permitem que seus funcionários escolham seus horários de trabalho, por exemplo.

Contudo, ausência de estruturas fortes não significa que existe uma anarquia na cultura *hacker* ou nas comunidades de *software* livre, e muito menos na ciência. Existem pessoas que guiam projetos ou pesquisas e a liderança é reconhecida pela comunidade. Assim como na academia a reputação se forma por meio do reconhecimento dos pares, no desenvolvimento de *software* aberto e na cultura *hacker* o prestígio dos programadores acontece em decorrência da qualidade das suas contribuições para a comunidade. Além disso, assim como na ciência a livre circulação de informação contribui para a ampliação do conhecimento de toda a comunidade científica, na cultura *hacker*

essa mesma abertura é essencial. *Hackers* acessam o código de outros *hackers*, questionam, trabalham para melhorar o funcionamento do programa e contribuem de volta para a comunidade. Esse processo pode ser entendido como um modelo de aprendizado:

> um aspecto bem característico do modelo de aprendizagem dos hackers reside no fato de que o conhecimento de um hacker ensina a outro. Quando um hacker estuda o código-fonte de um programa, não é raro esse hacker desenvolver ainda mais esse código e, dessa forma, outras pessoas podem aprender com o seu trabalho. Quando um hacker verifica fontes de informação na Rede, ele muitas vezes agrega informações úteis a partir de suas próprias experiências. Forma-se uma discussão contínua, crítica e crescente em relação a diversos problemas. O mérito de participar dessas discussões é simplesmente adquirir conhecimento (HIMANEN, 2001, p. 75).

Empiricamente, sabe-se que o processo de programação em uma linguagem *open source* segue o modelo descrito acima por Himanen. Quando se escreve o código-fonte de um programa, é comum serem examinados exemplos, tutoriais e explicações na internet. Os *hackerspaces* descritos anteriormente funcionam exatamente como centros de cooperação presencial. Fóruns de discussões *online* entre programadores também são formas comuns de se trocar conhecimento. Um exemplo é o *Stack Overflow*[39]: um site de perguntas e respostas para programadores profissionais ou entusiastas, que tem o objetivo de construir, com a ajuda dos usuários, uma biblioteca de respostas detalhadas para perguntas sobre programação (STACK OVERFLOW TOUR, 2015). No ano de 2014, 4.8 bilhões de páginas foram visualizadas no site (STACK EXCHANGE, 2015). Interessante notar no tutorial de como usar o *Stack Overflow* (disponível no próprio site) a recomendação de não se fazerem perguntas para as quais o programador ainda não tentou achar uma resposta sozinho. Esse tipo de comportamento não é bem visto, pois novas perguntas devem beneficiar todos e é necessário pesquisar a resposta antes de perguntar. Esse aspecto comportamental evidencia a coletividade da informação e a forma como se avança o conhecimento do grupo: fazendo novas perguntas.

[39] stackoverflow.com

Para exemplificar esse modelo de aprendizado em relação às linguagens de programação alternativas, vale observar as linguagens *Scratch* e *Arduino* mais uma vez. Um programador de *Scratch* conta com mais de 10 milhões de projetos abertos no site da comunidade podendo consultar o código-fonte de cada um deles e reutilizar o que desejar em seus programas. Ademais, os usuários podem comentar e fazer perguntas uns aos outros; e o fazem: são mais de 50 milhões de comentários (SCRATCH STATISTICS, 2015). Um programador de *Arduino* conta com o fórum e uma wiki com mais de 300 mil membros (ARDUINO FORUM, 2015), além de tutoriais disponíveis no site oficial e em vários outros lugares na internet.

De acordo com Pretto (2010), esses aspectos de troca, colaboração e aprendizado que acontecem no ambiente *online* podem também ser vistos como formas de educação. Para o pesquisador, pensar educação somente pela perspectiva da escola é limitante no mundo atual. O autor (p.315) defende a ideia de uma escola capaz de produzir cultura e conhecimento de forma aberta, que se insira no contexto das comunidades *online* e da internet, propondo, dessa maneira, uma ampliação da visão de educação, que vai além da sala de aula e que alcança a cultura *hacker*:

> pensamos ser possível, considerando a ética hacker que nos alimenta, a construção de outras educações, com base na pluralidade como parte integrante dos processos. Também para as palavras necessitamos desse plural. Portanto, em vez de educação, falamos em educações, com esse plural pleno, implicando todas e todos num rico processo de criação permanente.

A citação de Pretto é relevante, nesta pesquisa, para a reflexão sobre como aplicar o modelo de aprendizado *hacker* com artistas-programadores – ou até mesmo crianças. Isso não significa que todas as pessoas são capazes de aprender sozinhas, apenas entrando em contato com uma comunidade (apesar de algumas serem capazes). Porém, a possibilidade de desenvolver *software* em uma constante troca com outros artistas-programadores permite formas de aprendizado diferentes daquelas que só contam com a sala de aula. *Processing*, *Arduino* e *Scratch* já fazem parte da cultura *hacker*, então a possibilidade de pensar como potencializar esse aspecto em uma metodologia de ensino presencial é extremamente interessante. Abrir canais de comunicação *online*, grupos de discussão por email, fóruns de discussão e publicar todo o

material e o código em licenças abertas, como a *Creative Commons*[40] e as de *software* livre, são alguns exemplos simples de como combinar a cultura *hacker* com metodologias presencias de aprendizado.

[40] *Creative Commons* (creativecommons.org) é uma organização fundada por Lawrence Lessig, em 2001, em Mountain View na Califórnia, que publicou diversas licenças de direitos autorais que permitem aos autores de projetos, pinturas, filmes, fotografias, livros etc. publicarem suas obras e definirem quais direitos de propriedade querem preservar (como atribuição de autoria) e quais querem dar ao público (como livre reprodução).

# 3 O Processing

## 3.1 A linguagem

O *Processing* surgiu em 2001 e foi criado por Ben Fry e Casey Reas, na época alunos de doutorado do MIT *Media Lab*, sob a orientação do Professor John Maeda, e membros do grupo de pesquisa *Aesthetics and Computation*. O objetivo de Fry e Reas era elaborar uma linguagem de programação simples, que inspirasse o processo criativo nas artes visuais e pudesse ser utilizada por artistas e designers. Como visto no Capítulo 2, o *Processing* está inserido em um contexto mais amplo de linguagens de programação alternativas que têm o objetivo de tornar a programação mais fácil e de levá-la para além dos muros da engenharia.

O *Processing* teve como inspiração direta um outro projeto que foi liderado por Maeda e lançado em 1999: a linguagem *Design By Numbers* - DBN (Figura 24). Durante o doutorado, Fry e Reas trabalharam na manutenção e no desenvolvimento do DBN e essa experiência serviu como base para o projeto do *Processing*. Enquanto trabalhava no DBN, Fry criou vários experimentos em outras linguagens de programação (*Phyton* e *Scheme*) e algumas funcionalidades visuais, que, no entanto, não cabiam nele. Interessados em combinar as capacidades mais complexas realizadas por Fry nos seus experimentos com o aspecto educacional do DBN, Fry e Reas criaram o *Processing* (FAQ, 2015).

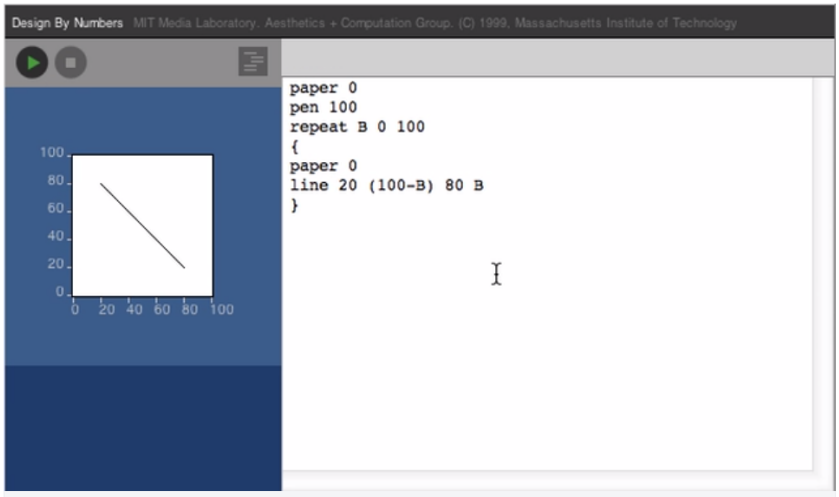



Figura 24 – Ambiente de desenvolvimento da linguagem DBN. Fonte: Design (2013).

Uma linguagem de programação pode ter como alicerce uma outra linguagem de programação. O *Processing* foi criado utilizando como base o Java e, por isso, tem muito potencial, havendo muito pouco que não seja possível fazer com ela, diferentemente do *Logo* e do DBN, que apresentam algumas limitações (SHIFFMAN, 2008). O *Processing* é uma linguagem de alto nível[41] composta de dois aspectos fundamentais: um ambiente de desenvolvimento (IDE) e a linguagem de programação em si.

A linguagem *Processing* é orientada a objeto[42] e permite a escrita de programas gráficos (denominados *sketches*) sem a complexidade e a estrutura rígida normalmente necessária no Java (GREENBERG, 2007). A seguir um exemplo de código em *Processing*:

```
void setup() {
  size(480, 120);
}

void draw() {
  if (mousePressed) {
    fill(0);
  } else {
    fill(255);
  }
  ellipse(mouseX, mouseY, 80, 80);
}
```

Como se pode notar no exemplo, a linguagem *Processing* apresenta duas seções: *void setup()* e *void draw()*. No *void setup()*, são definidos os comandos de configuração inicial do programa e, no *void draw()*, é realizada a execução dos comandos para desenhar em um *loop* (a não ser que seja especificado o contrário).

Como o *Processing* é voltado para as artes visuais, o *output* do programa é primordialmente um desenho ou representação visual – diferentemente de outras linguagens de programação que privilegiam o *output* em texto. No universo da programação existe uma tradição: o primeiro programa escrito em uma nova linguagem

[41] Uma linguagem de alto nível apresenta um grau de abstração elevado e é mais fácil e mais simples para o programador manipular.

[42] Programação orientada à objeto é um modelo de programação governado pelos relacionamentos entre os "objetos" do programa. Esse modelo de programação permite que o programador molde o funcionamento de cada objeto e que determine as regras de interação entre eles.

deve produzir o texto "*Hello, World!*" (Olá, Mundo!). No *Processing*, todavia, um programa *Hello, World!* é, normalmente, uma imagem, como a linha mostrada na Figura 25.

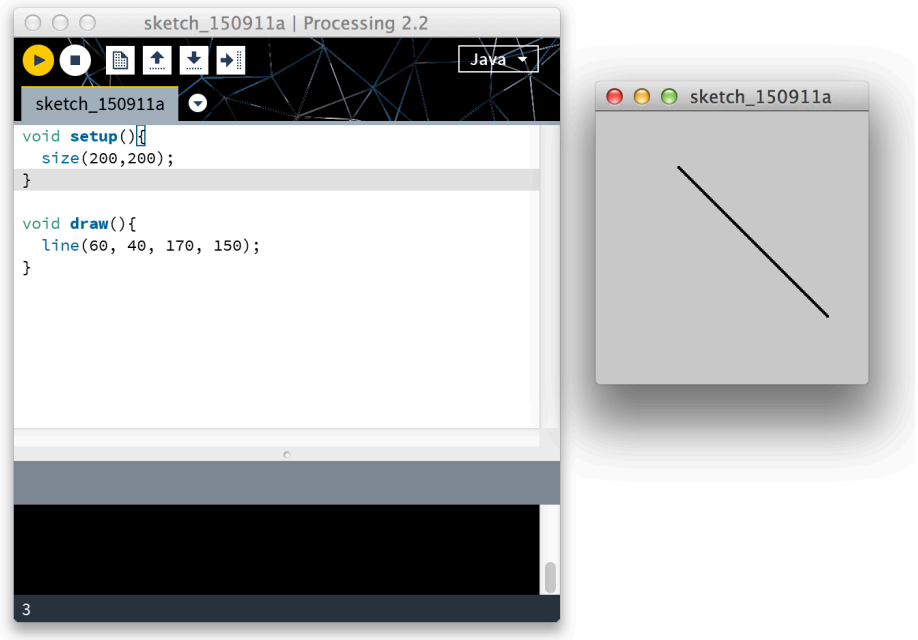



Figura 25 – *Hello World!* em *Processing*: o desenho de uma linha. Imagem produzida pela pesquisadora.

A linguagem *Processing* não representa uma ruptura radical em relação às linguagens de programação clássicas. Diferentemente da linguagens de programação visual *Max*, *Pure Data* ou *Scratch* (analisadas no Capítulo 2), o *Processing* é uma linguagem textual. Isso pode representar para algumas pessoas um nível maior de dificuldade, principalmente quando se compara o *Processing* com outras linguagens visuais de programação. No entanto, o *Processing* reposiciona a programação textual e linear de uma maneira que a torna alcançável para pessoas que se interessam por ela mas que se sentem intimidadas ou desinteressadas pelo tipo de programação que acontece nos departamentos de ciência da computação (REAS; FRY, 2006). Com o *Processing* é possível produzir imagens com comandos mais simples, enquanto que no Java, um programa para realizar a mesma tarefa requer mais linhas de código, como pode ser observado abaixo:

Código em Java para desenhar um retângulo:

```
import java.awt.Graphics;

import javax.swing.JComponent;
import javax.swing.JFrame;

class MyCanvas extends JComponent {

  public void paint(Graphics g) {
    g.drawRect (10, 10, 200, 200);
  }
}

public class DrawRect {
  public static void main(String[] a) {
    JFrame window = new JFrame(;
     window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOS
E);
    window.setBounds(30, 30, 300, 300);
    window.getContentPane().add(new MyCanvas());
    window.setVisible(true);
  }
}
```

Código em *Processing* para desenhar um retângulo:

```
void setup (){
 size (300, 300);
}

void draw (){
  rect(100,90,100,100);
 }
```

Como é possível observar nos exemplos acima, o *Processing* simplifica a escrita do código-fonte para a produção de imagens. Assim, apesar de não ser uma linguagem visual, o *Processing* é mais simples para os programadores que desejam criar efeitos visuais a partir do código ou que ainda estão aprendendo a programar.

Para o Professor Doutor Hermes Renato Hildebrand, da UNICAMP e da PUC-SP, em entrevista concedida para esta pesquisa[43], a simplicidade do *Processing* é um dos seus aspectos mais importantes, pois apesar de ser uma linguagem de programação, permite uma estruturação simples para iniciantes. O professor utiliza a linguagem para ensinar programação aos seus alunos. Antes do *Processing*, ele utilizava a linguagem *Logo*, porém

[43] Entrevista realizada em agosto de 2015.

os alunos aprendiam a lógica da programação mas não conseguiam alcançar a etapa da produção de imagens e da experimentação artística – como acontece com a utilização do *Processing*.

De acordo com Ribeiro (2013), a possibilidade de criar efeitos visuais interessantes de forma simples e quase imediata é a principal razão para se usar a linguagem *Processing* no ensino de programação para alunos de cursos artísticos:

> as estratégias de ensino/aprendizagem no ensino de linguagens de programação a alunos de cursos artísticos, devem ser adaptadas, tendo em conta as limitações decorrentes da preparação que estes alunos têm. Essas metodologias devem ainda ter em conta a sensibilidade que este tipo de público-alvo revela para estímulos de caráter especialmente sensorial [...] Esta tarefa fica simplificada pelas características apresentadas pelo ambiente de desenvolvimento e pela linguagem Processing. O ambiente de desenvolvimento é especialmente acessível; a sintaxe da linguagem de programação é particularmente inteligível; e a criação de resultados, através da implementação, é entusiasmante, pois é possível obter resultados interessantes de forma bastante simples e direta.

Interessante notar com base na citação de Ribeiro como é crucial colocar o resultado artístico do programa como foco do processo de ensino de programação para alunos das áreas de artes e design. A programação é vista por esses alunos como um meio para se criar, ou como um material artístico, e não como o objetivo final do aprendizado. Esse entendimento é decisivo para se obter êxito na metodologia de ensino aplicada às artes.

Ademais, por ser uma linguagem textual, o artista-programador-aprendiz que utiliza o *Processing* consegue, posteriormente, migrar para outras linguagens com facilidade e explorar novas possibilidades artísticas em plataformas como o *Arduino*, por exemplo. Além de ser uma ferramenta usada na educação, o *Processing* se desenvolveu de modo a se tornar uma linguagem de uso profissional e também uma maneira de programar.

O *Arduino*, analisado no Capítulo 2, por exemplo, utiliza o código-fonte do *Processing* para o ambiente de desenvolvimento, além de apresentar uma sintaxe de linguagem extremamente parecida (Figura 26) replicando a estrutura do *Processing* no mundo da eletrônica. Uma comparação entre a Figura 26 e a Figura 27 revelará algumas dessas semelhanças entre os ambientes de desenvolvimento e entre as linguagens. A segunda parte estrutural do código em *Processing* é *void draw(),* enquanto que no *Arduino* recebe o nome de *void loop().* Apesar dos nomes diferentes, em ambas as linguagens trata-se de um *loop* infinito dentro do qual se encontram os comandos de execução do programa. Assim, um programador de *Processing* consegue entender a lógica da programação em *Arduino* com facilidade, e vice-versa. O trecho abaixo, retirado do site oficial do *Processing* (PROCESSING, 2015, tradução nossa), esclarece esse ponto:

> o Processing foi inicialmente lançado com uma sintaxe baseada em Java e um léxico de primitivas gráficas inspirados no OpenGL, Postscript, Design by Numbers e outras fontes. Com a adição gradual de interfaces de programação alternativas – incluindo JavaScript, Python e Ruby –, ficou cada vez mais claro que Processing não é uma linguagem de programação única, mas sim um modo orientado à arte de aprender, ensinar e criar com código.

Também é importante assinalar que o ambiente de programação do *Processing* é capaz de compilar o código rapidamente, possibilitando que o artista-programador veja o resultado do seu código sempre que desejar. Da mesma maneira que um artista, quando está desenhando, produz esboços durante seu processo, no *Processing* o artista-programador pode visualizar o resultado do seu programa no decorrer do desenvolvimento do programa. Isso é muito importante, pois como foi visto anteriormente em relação às linguagens *Scratch* e *Arduino*, o processo de criação do artista-programador, ou do programador alternativo, se baseia na experimentação e no improviso e, por essa razão, é essencial ter um fluxo de trabalho em que o resultado estético do código seja examinado com frequência.

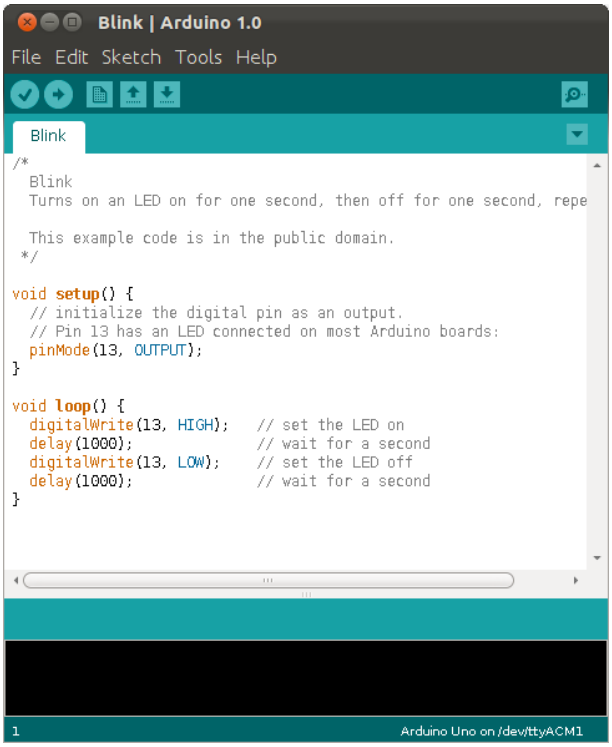



Figura 26 – Ambiente de desenvolvimento do *Arduino.* Fonte: Arduino (2014).

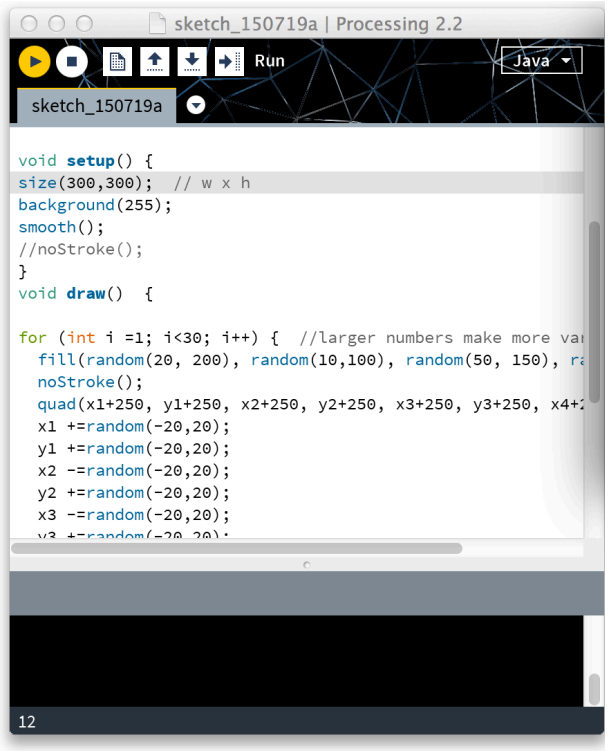



Figura 27 – Ambiente de Programação do *Processing.* Imagem capturada pela pesquisadora.

O ambiente de desenvolvimento do *Processing* pode ser obtido gratuitamente (devido à licença de *software* livre) no website oficial do projeto[44] e consiste de um editor de texto onde se escreve o código, uma área de mensagens, um painel de controle, uma janela de exibição do resultado do código em execução, abas para a organização de partes diferentes do código e uma barra de ferramentas. O editor do *Processing* (Figura 28) é semelhante a muitos outros editores de texto usados por programadores em outras linguagens. As abas (*tabs*) existentes no editor permitem que o artista-programador organize seu código em diferentes seções, como geralmente se faz na programação tradicional com diferentes arquivos. O painel de controle (*console*) possibilita o acompanhamento do código, pois é nele que se podem obter informações textuais e inclusive *outputs* numéricos do programa que está sendo executado. Existe também uma área de mensagens de erro ou de sucesso do programa (*message area*).

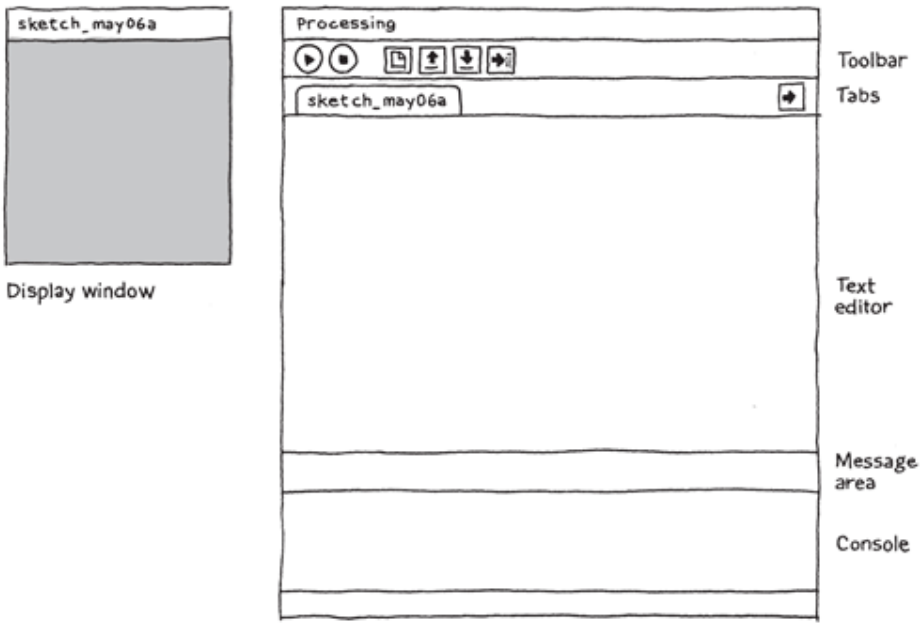



Figura 28 – Descrição das funcionalidades do ambiente de desenvolvimento do *Processing*.
Fonte: Processing (2015).

[44] processing.org

Em 2012, foi fundada a *Processing Foundation* com o mesmo propósito do projeto inicial de Fry e Reas: promover alfabetização em programação no campo das artes visuais e alfabetização visual no campo tecnológico. “Nosso principal objetivo é capacitar pessoas com todos os tipos de interesse e formação a aprender a programar” (PROCESSING FOUNDATION, 2015). A Fundação também oferece suporte e incentivo financeiro para pessoas interessadas em ampliar a linguagem, tendo sido responsável pelas extensões da linguagem em JavaScript[45] e Python[46].

### 3.2 Cultura hacker e software livre na programação de arte

Assim como outras linguagens de programação analisadas no Capítulo 2, *Scratch*, *Arduino* e *Pure Data*, o *Processing* também foi lançado sob uma licença de *software* livre e conta com uma comunidade ativa de desenvolvedores que permanentemente o ampliam e realizam manutenção do código corrigindo erros e eliminando *bugs*. Esses contribuidores compartilham programas, desenvolvem bibliotecas (pacotes de código prontos para serem usados) e criam formas de expandir as possibilidades do *software*. Todo o código do *Processing* está disponível no GitHub[47], um site de compartilhamento de código entre programadores da comunidade de *software* livre. É no GitHub que acontece grande parte da colaboração no desenvolvimento da linguagem. Como explicam Fry e Reas (2006, p.531), criadores do *Processing*, a opção pela licença de *software* livre tinha exatamente o objetivo de incentivar a cultura do *software* livre nas artes:

> o Processing se empenha em aplicar o espírito da inovação do open source ao domínio das artes. Nós lutamos para oferecer uma alternativa aos softwares comerciais disponíveis e para aumentar o conhecimento e as habilidades daqueles que participam das comunidades artísticas estimulando o interesse em iniciativas similares.

A relevância da comunidade de desenvolvimento de *software* livre pode ser observada, por exemplo, nas bibliotecas de código. As bibliotecas desenvolvidas pela comunidade do *Processing* são um dos aspectos mais importantes do projeto e têm um

[45] p5.js
[46] processing.py
[47] github.com/processing/processing/

enorme impacto na maneira como as pessoas usam a linguagem (LIBRARY, 2015). Em 2007, havia 5 bibliotecas criadas por usuários (GREENBERG, 2007). No momento da escrita da presente pesquisa[48], já existem 107 bibliotecas que foram contribuições da comunidade de desenvolvedores do *Processing* (LIBRARIES, 2015). Esse aumento do número de bibliotecas desenvolvidas pela comunidade revela a atividade e o potencial do grupo de desenvolvedores interessados em ampliar as possibilidades da linguagem e como o trabalho colaborativo *online* pode ser um fator impulsionador no crescimento de um *software*. Os tipos de bibliotecas são variados: conexão com *Arduino*, bibliotecas para se trabalhar com mapas, imagens 3D, dentre muitas outras.

A cultura de colaboração e abertura de informação do *software* livre também pode ser observada na criação de arte em *Processing*, e não somente no desenvolvimento da ferramenta. Na comunidade *online OpenProcessing*, uma comunidade de artistas, designers e educadores que utilizam *Processing*, existe uma funcionalidade que permite reutilizar o código de uma outra pessoa. No mundo de desenvolvimento de *software* livre ou *open source*, *forking* é o nome que se dá a esse processo que é uma bifurcação no código, ou seja, quando um programador resolve utilizar o código original de alguém mas opta por um outro caminho. Várias pessoas podem fazer uma bifurcação ao mesmo tempo e, ao final, o código assim desenvolvido pode ser incorporado no programa original ou realmente tomar um rumo independente. No *OpenProcessing*, o processo de bifurcação é chamado de *Tweaking*, palavra inglesa que significa "ajuste".

Sinan Ascioglu, criador da comunidade *OpenProcessing*, explica em entrevista concedida para esta pesquisa[49] que optou por denominar essa funcionalidade no *OpenProcessing* de *Tweak* em vez de *Fork* porque percebeu que no *Processing* a maioria das pessoas utiliza o código-fonte de outros usuários para fazer pequenas modificações ou ajustes. Já em outros contextos, o *Fork* muitas vezes representa uma ruptura com a linha de desenvolvimento que estava sendo seguida podendo resultar em um novo *software*.

[48] 12 de julho de 2015.
[49] Entrevista realizada em maio de 2014.

Todos os *sketches* publicados no *OpenProcessing* podem ser "ajustados" ("*tweaked*"). Na Figura 29, observa-se um gráfico da atividade de bifurcação do código, ou *Tweak*, na comunidade *OpenProcessing*. O gráfico foi gerado em uma análise realizada por Ascioglu com base na atividade do site em 2012. A comunidade apresenta três perfis de usuários: estudantes, professores e usuários comuns. Os estudantes estão representados em verde, professores em azul e usuários comuns em rosa. Cada ponto no perímetro do círculo é um usuário e quanto maior o seu tamanho, maior a sua atividade, tanto na disponibilização de código quanto no aproveitamento de código de outros usuários. Constata-se que alguns usuários do site – os quatro maiores círculos azuis, o maior rosa e o maior verde – são mais influentes do que outros. Isso significa que esses usuários são grandes contribuidores de código para a comunidade – e a maior parte deles, os quatro círculos azuis, são educadores. Desse modo, verifica-se que a maior habilidade de alguns beneficia toda a comunidade através da abertura da informação e do compartilhamento de conhecimento.

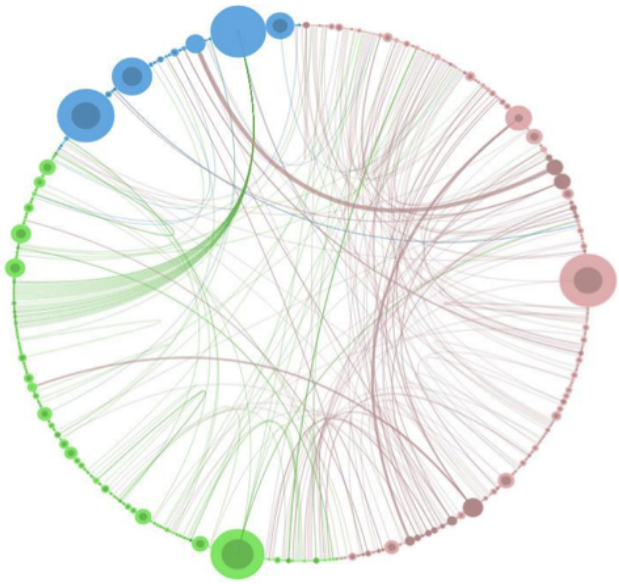


Figura 29 – Visualização da atividade *Tweak* na comunidade *OpenProcessing*. Em azul, professores; em verde, estudantes; em rosa, usuários comuns. Fonte: Ascioglu (2012).

A seguir será exposto um exemplo de *Tweak* feito pela autora desta pesquisa no *OpenProcessing*. Por apresentar um código que realiza um efeito de movimento parecido com o que a autora desejava, o *sketch AngularRectangularRev*[50] do usuário Jeff Hendrickson[51] (Figura 30) foi escolhido para ser adaptado. O *sketch* original utiliza a repetição da forma retangular com movimentos, tamanhos e cores aleatórias. O código-fonte do *sketch* AngularRectangularRev pode ser encontrado no Anexo 1.

Figura 30 – AngularRectangularRev de Jeff Hendrickson no *OpenProcessing*.
Fonte: Hendrickson (2009).

Após a escolha do *sketch*, foi selecionada a opção *Tweak*, que faz o *download* do código para o computador do usuário. Interessante assinalar que o *Tweaking* pode ser feito de forma quase automática baixando o *sketch* para o ambiente de quem irá modificá-lo. O objetivo do *sketch* da pesquisadora é criar um protótipo de uma instalação artística inspirada pelo engarrafamento em São Paulo, denominada Trânsito (Figura 31), que foi elaborada em uma das disciplinas do curso de mestrado no TIDD na PUC-SP. Com base no conceito do trânsito como sistema circulatório da cidade de São Paulo, a pesquisadora criou um *sketch* que utiliza dados da CET-SP para determinar a velocidade com que a animação acontece na obra. Além disso, a instalação também utiliza sons do trânsito para criar um ambiente imersivo. O *sketch* de Jeff Hendrickson foi escolhido por

[50] openprocessing.org/sketch/4578
[51] openprocessing.org/user/749

já ter o código pronto para a estrutura da criação dos objetos geométricos e da aleatoriedade de movimento e de cores.

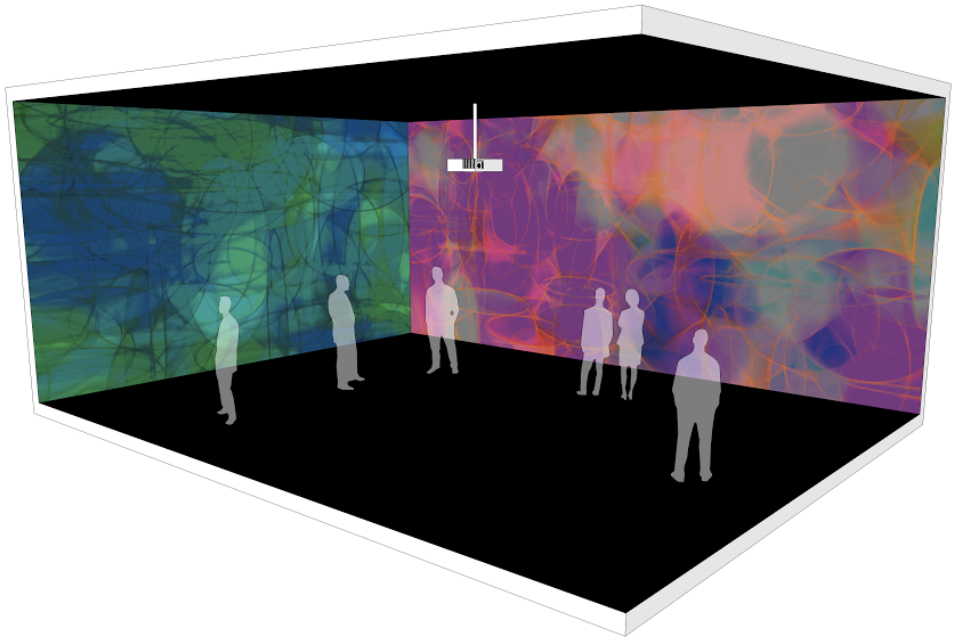


Figura 31: Projeto para a instalação Trânsito criado pela pesquisadora com base em código-fonte disponível na comunidade *OpenProcessing*.

No código da nova aplicação criada pela pesquisadora, foram feitas algumas modificações: criação de uma classe para melhor organizar o código, alteração da forma base para elipse, substituição da palheta de cores, mudança na lógica de movimentos das formas geométricas fazendo com que elas retornem ao atingir o limite da janela de exibição e que a animação permaneça sempre em *loop* (o *sketch* de Hendrickson é finito), adição de uma cadência na mudança de cores em intervalos de tempo pré-determinados, acesso ao site da CET-SP para aquisição do número de quilômetros de engarrafamento e utilização dessa informação para determinar a velocidade da animação. A Figura 32 apresenta um fluxograma do funcionamento do protótipo e o código completo encontra-se no Apêndice C.

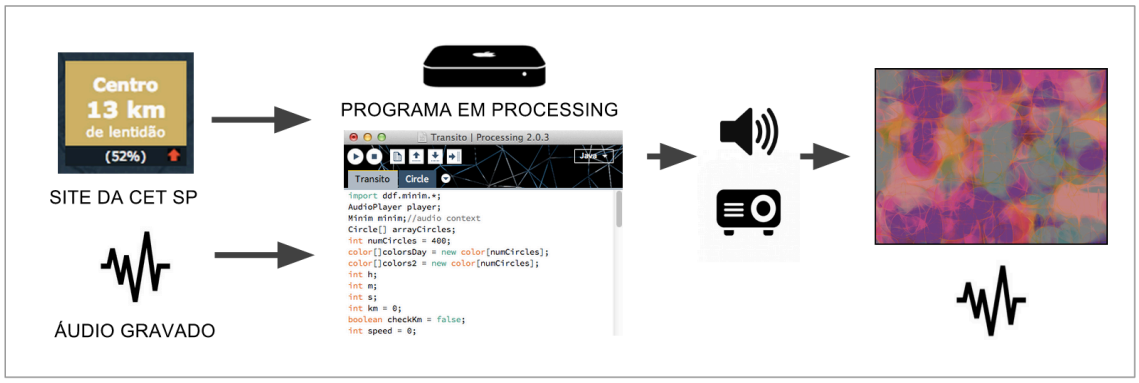



Figura 32 – Esquema de funcionamento do protótipo para a instalação Trânsito (Figura 31) com base no código-fonte do *sketch* AngularRectangularRev de Jeff Hendrickson disponível no *OpenProcessing*. Código-fonte do protótipo disponível no Apêndice C.

O processo de criação do protótipo da instalação Trânsito teve como início uma pesquisa por referências de código-fonte na comunidade *OpenProcessing*. Quando foi encontrado um *sketch* com a funcionalidade desejada (*AngularRectangularRev* de Jeff Hendrickson; código-fonte disponível no Anexo 1), a pesquisadora adaptou o código e criou em cima para chegar ao resultado desejado (Figura 31; código-fonte no Apêndice C). Dessa forma, é interessante notar, com base nessa demonstração da funcionalidade *Tweak*, como o artista-programador do *Processing* pode utilizar código de outros artista-programadores de maneira semelhante ao que acontece na cultura *hacker* e no *software* livre, em que a abertura da informação e do código-fonte é um ponto central. Essa possibilidade aponta para características culturais do desenvolvimento de *software* livre e da cultura *hacker* na produção de arte em *Processing*. Quando alguém reutiliza o código de um artista, isso é motivo de orgulho e não de preocupação com a possibilidade de plágio, da mesma maneira que na cultura *hacker* e no *software* livre a reputação é construída com base na contribuição para a comunidade.

Apesar de não ser o foco desta pesquisa, vale ressaltar que o aspecto colaborativo do *Processing* também remete a questões interessantes sobre a autoria da arte feita em código, uma vez que vários artistas compartilham código-fonte e permitem a sua reutilização. Quando um artista utiliza o código de outro artista, quem é o autor da obra? Apesar de ser prática comum se dar crédito ao programador original (como se pode observar no código fonte do protótipo da instalação Trânsito no Apêndice C), a questão da autoria e do direito-autoral da obra final são temas férteis para debate em relação ao

processo de criação do artista-programador. Sem dúvida, um assunto que pode ser explorado em investigações futuras.

Um outro aspecto da cultura do *software* livre na criação de arte em *Processing* pode ser observada na página de cada *sketch* da comunidade *OpenProcessing*, onde são apresentadas as licenças de publicação, no canto inferior direito: *Creative Commons* e GNU-GPL (Figura 33, destacado em vermelho).

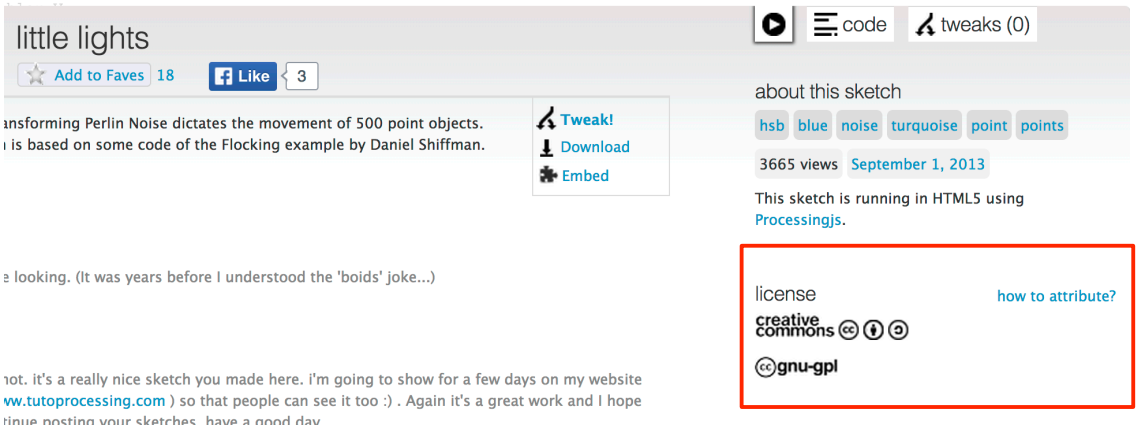



Figura 33 – Licenças abertas no *OpenProcessing*: *Creative Commons* CC BY- SA 3.0 e GNU-GPL.
Fonte: Rahm (2013).

São duas licenças: *Creative Commons* e GPL. A licença *Creative Commons* protege ou autoriza direitos relacionados aos aspectos visuais e artísticos da obra. Já a licença GNU-GPL é fornecida pela *Free Software Foundation*, organização fundada pelo criador do *software* livre, Richard Stallman. Com essa licença, é possível garantir que o programa criado livre continue sendo livre. A licença utiliza a legislação de direito autoral (*copyright*) de maneira reversa, daí ser também chamada de *Copyleft*. Essa licença é normalmente usada para projetos de engenharia de *software* como, por exemplo, o sistema operacional Linux. Porém, no *OpenProcessing*, essa mesma licença é utilizada para o código artístico. Como explica Mendes (2006), a licença GPL tem o objetivo de preservar algumas liberdades:

- Livre execução do programa.
- Liberdade para estudar e adaptar o programa.
- Livre compartilhamento.
- Liberdade de alterar o programa e redistribuí-lo.

Interessante observar como esses fundamentos, originários da cultura *hacker*, também estão presentes na criação de arte do *Processing.* Ascioglu explica que foi necessário incluir as duas licenças porque a GNU-GPL não cobre o aspecto visual do *sketch* e a *Creative Commons* não cobre o código-fonte. Como a produção em *Processing* é tanto código quanto arte visual, foi necessário implementar as duas licenças garantindo assim a abertura completa da informação.

A licença *Creative Commons* usada pelo *OpenProcessing* é a CC BY- SA 3.0, *Attribution-ShareAlike* (em português conhecida como Atribuição-CompartilhaIgual 3.0)[52], que permite a ampla distribuição e reutilização do *sketch* por meio de *remix*, transformação ou extensão para qualquer fim, inclusive comercial, sendo exigida apenas a atribuição de autoria. Já a licença GNU-GPL cobre o *software* em si, o código. A licença protege o direito ao acesso livre ao código e a sua reutilização.

O artista-programador que utiliza *Processing* não é obrigado a divulgar o código da sua obra, porém os que participam do *OpenProcessing* sempre têm que publicar seus *sketches* sob essas duas licenças. Toda arte postada no *OpenProcessing* tem o código aberto (GNU-GPL) e está disponível para ser remixada (CC BY- SA 3.0). Desse modo, nota-se que os aspectos da cultura *hacker* não ficaram restritos à comunidade de desenvolvedores do *Processing*, que mantêm a linguagem e criam extensões e bibliotecas, pois também fazem parte da maneira como os artistas-programadores de *Processing* se comportam ao criar arte. Reas e Fry (2006, p.531, tradução nossa) esclarecem que

[52] creativecommons.org/licenses/by-sa/3.0/br/

> abrir o código-fonte do Processing permite que as pessoas aprendam a partir da sua construção e através da sua extensão com seus próprios códigos. As pessoas são estimuladas a publicar o código dos seus programas escritos em Processing. Da mesma maneira que a funcionalidade "ver fonte" em navegadores estimularam a rápida expansão da Web, o acesso ao código em Processing escrito por outras pessoas possibilita que membros da comunidade aprendam uns com os outros e que as habilidades da comunidade cresçam como um todo.

Como explicam Reas e Fry na citação acima, essas formas de colaboração e troca *online* são também maneiras de se aprender e ensinar *Processing*. O aprendizado da linguagem, portanto, não acontece somente nos cursos formais das escolas de arte e design e nos *workshops* ao redor do mundo. Um outro destino importante, além do *OpenProcessing*, é o fórum oficial do *Processing* (PROCESSING FORUM, 2015), mostrado na Figura 34, com mais de 10.500 tópicos postados até julho de 2015. Nesse fórum, artistas-programadores, estudantes e desenvolvedores perguntam e respondem sobre os mais variados temas. Também é possível observar a troca *online* no site Stack Overflow (Figura 35), onde existem 2.672 perguntas relacionadas ao *Processing* no momento de escrita da presente pesquisa[53] (STACK, 2015).

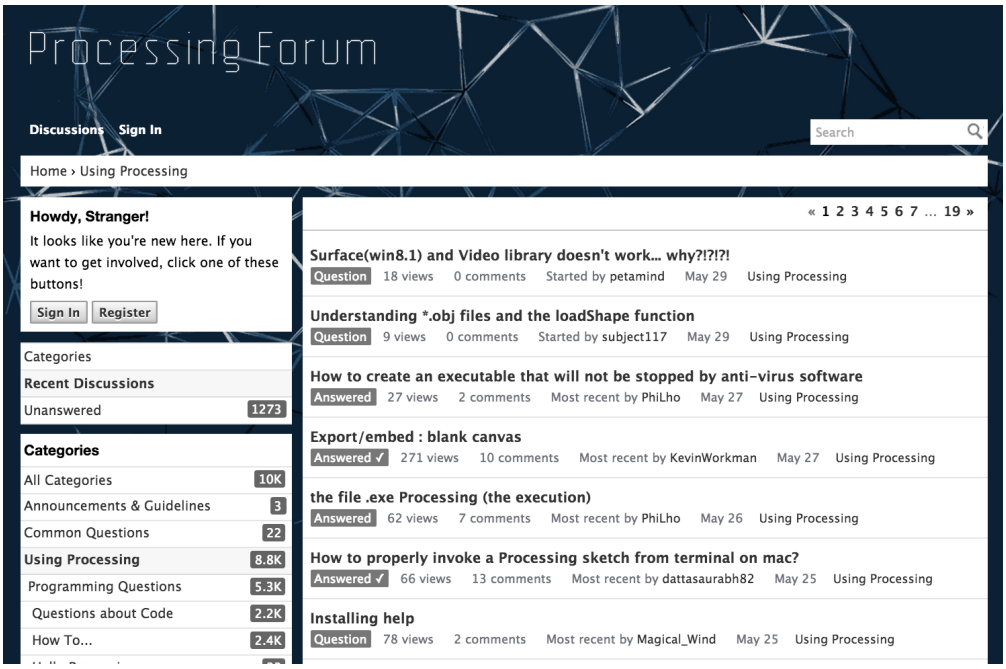



Figura 34 – Fórum oficial do *Processing*. Fonte: Processing Forum (2015).

[53] 12 de julho de 2015.

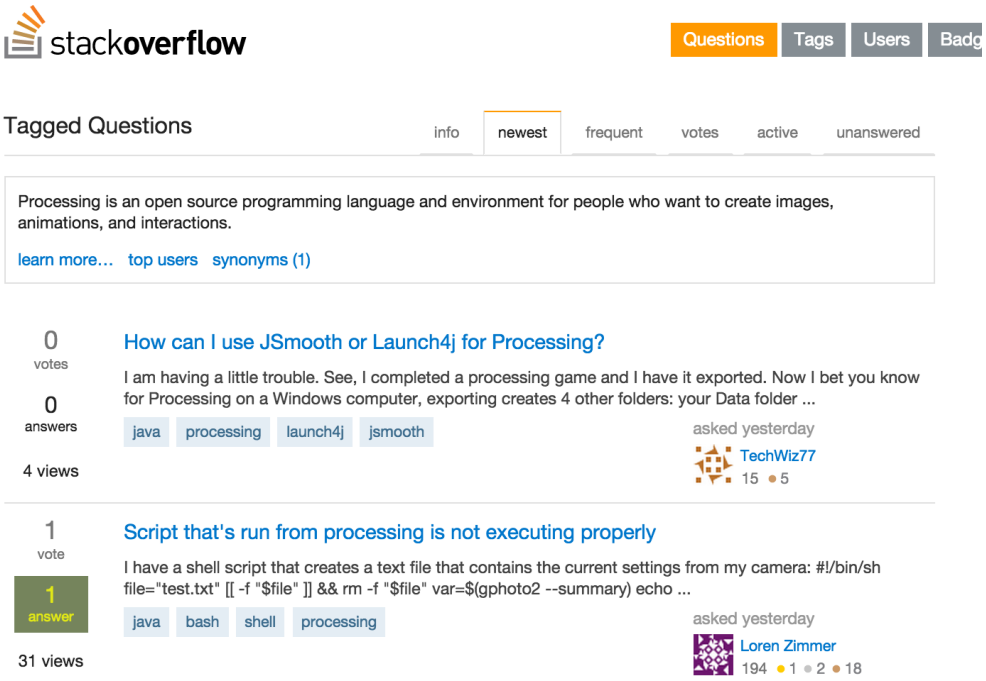



Figura 35 – *Processing* no site *Stack Overflow*. Fonte: Stack (2015).

Uma outra forma de troca de ideias na comunidade do *Processing* acontece no *OpenProcessing* com a possibilidade de comentar nos *sketches*. Todos os comentários são públicos, revelando mais uma vez a cultura de abertura da informação. A Figura 36 mostra o gráfico da atividade de comentários realizados no *OpenProcessing* no ano de 2012. É interessante observar que os estudantes (em verde) comentam muito entre si. Como o número de estudantes é significativamente menor do que o número de usuários comuns (em rosa)[54], pode-se concluir que eles comentam mais do que os outros tipos de usuários. Segundo Ascioglu, mesmo em número menor, os estudantes são responsáveis por mais de 50% de toda a atividade no site, o que expõe seu caráter educativo e a presença na comunidade de alunos matriculados em cursos formais. Percebe-se também que dois usuários, o maior círculo em rosa (um usuário comum) e o maior círculo em azul (um professor), são muito ativos e funcionam como catalisadores dos debates.

[54] No dia 25 de agosto de 2015, havia 46.301 usuários comuns (rosa), 4.783 estudantes (verde) e 357 professores (azul). Dados fornecidos por Ascioglu para esta pesquisa.

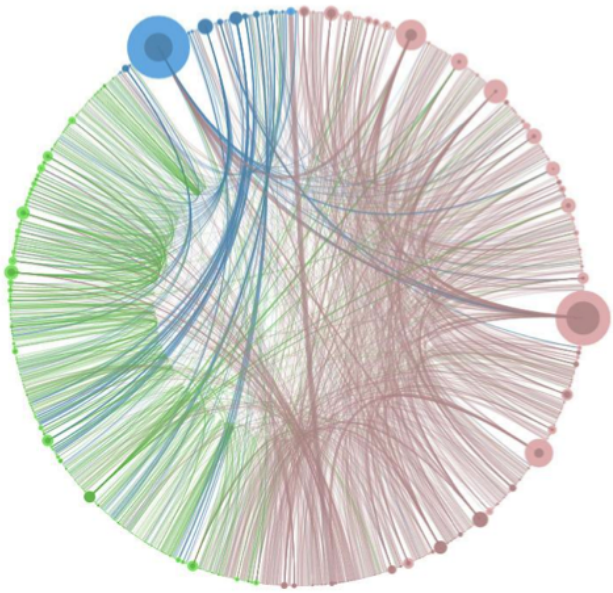


Figura 36 – Visualização de comentários na comunidade *OpenProcessing*. Em azul, professores; em verde, estudantes; em rosa, usuários comuns. Fonte: Ascioglu (2012).

Ao criar o *OpenProcessing*, Ascioglu desejava manter simples a interação entre os usuários e minimizar qualquer problema com privacidade e segurança de dados. Não sendo um programador profissional e por trabalhar sozinho, Ascioglu se empenhou em simplificar a estrutura do site ao máximo. Por esse motivo optou por manter tudo público. As únicas informações privadas no site eram e-mail e senha[55]. Ele conta que isso atraiu mais gente e, assim, a comunidade se formou.

Essa abertura dos dados, portanto, foi um fator crucial para a formação da comunidade *OpenProcessing*, pois estava alinhada com a cultura do *Processing* e em sintonia com a cultura do *software* livre e com a cultura *hacker*. O surgimento de uma nova comunidade como o *OpenProcessing* não é uma surpresa, no entanto, pois atentos para o potencial da colaboração na internet, os criadores do *Processing* tinham como meta, desde a concepção do projeto, a formação de comunidades *online*:

[55] Atualmente, é possível obter acesso a áreas restritas no site por meio da assinatura *OpenProcessing Plus*. Essa parte do site não é analisada nesta pesquisa, mas é uma estratégia que Ascioglu criou para aumentar a arrecadação monetária do site e manter seu funcionamento.

> o Processing foi projetado para utilizar-se da força das comunidades online, o que permitiu que o projeto crescesse de maneiras inesperadas. Milhares de alunos, educadores e praticantes nos cinco continentes estão usando o software (REAS; FRY, 2004, p.1, tradução nossa).

Segundo Hildebrand, a comunidade do *Processing* auxilia os alunos iniciantes na resolução de problemas, pois quando eles se deparam com um desafio, podem encontrar na comunidade caminhos para a resposta. Então, mesmo sem saber programar profundamente, o aluno consegue decupar o código e alterar as partes que necessita. Para o professor, o *Processing* "é uma linguagem que você aprende ao fazer". Interessante fazer um paralelo entre essa observação do professor e um dos seis princípios da cultura *hacker*, como apontado por Levy (2010) e exposto na seção 2.3 Cultura *hacker*: a preferência por métodos práticos de aprendizado. Dessa maneira, investir em metodologias de ensino voltadas para o aprendizado prático pode ser um caminho proveitoso para educadores.

Segundo Neto (2010, p.34), é exatamente esse aspecto colaborativo que leva artistas a trabalhar com *software* livre:

> muitas vezes os artistas optam pelo software livre especialmente pelo seu caráter colaborativo e de construção coletiva. Trabalhar com softwares livres tem a vantagem de encontrarmos inúmeras comunidades de desenvolvedores que estão dispostos a encontrar soluções para o funcionamento do software. Do mesmo modo, também encontram-se comunidades de programadores que disponibilizam algoritmos ou parte de algoritmos, o que facilita o desenvolvimento de trabalhos mais complexos.

O *OpenProcessing* apresenta áreas do site dedicadas para a interação entre professores e alunos, chamadas de salas de aula (Figura 37). Ascioglu explica que as salas de aula no *OpenProcessing* surgiram de forma espontânea. Por causa da facilidade oferecida pelo *OpenProcessing* para o compartilhamento e aprendizado em grupo, professores de diversas escolas de arte começaram a colocar o *OpenProcessing* nas suas ementas e o site acabou se tornando popular no mundo educacional. Vendo isso, Ascioglu decidiu criar a funcionalidade específica das salas de aula.

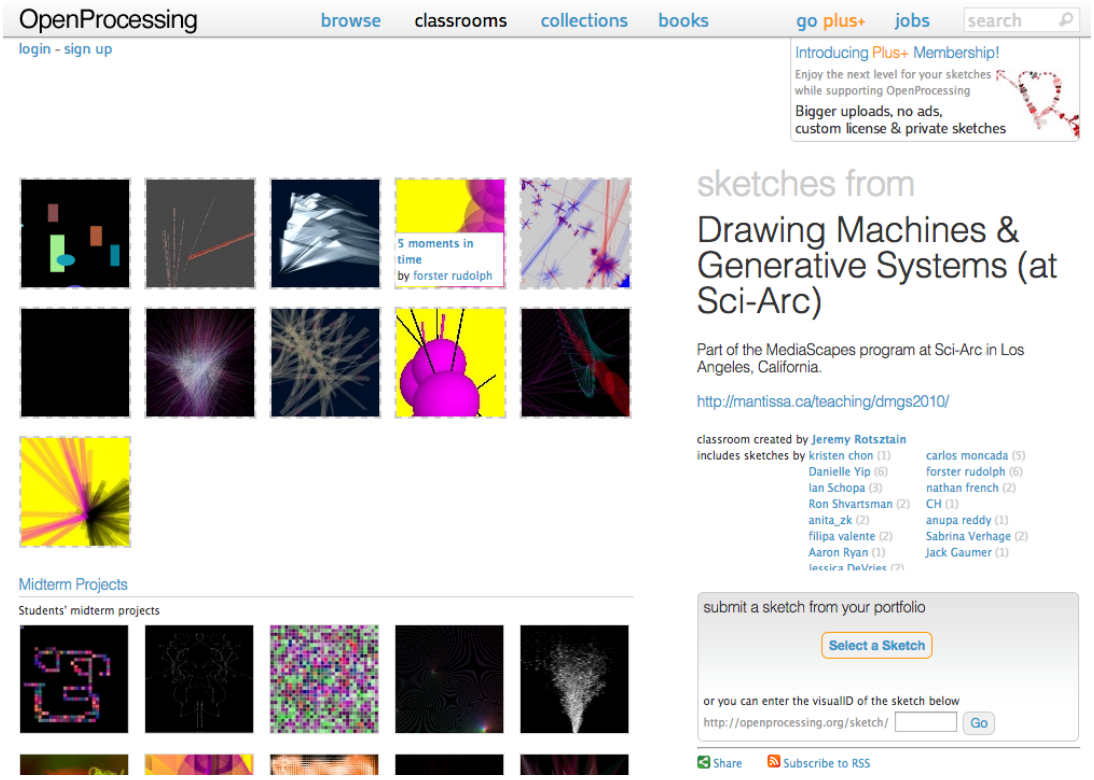



Figura 37 – Sala de Aula no *OpenProcessing*. Fonte: OpenProcessing Classrooms (2015).

Em junho de 2015, havia mais de 1.000 salas de aula no *OpenProcessing* de lugares diversos do mundo e em idiomas diferentes (OPENPROCESSING CLASSROOMS, 2015). No Brasil, há algumas salas de aula mais antigas e outras mais recentes, com presença expressiva da Universidade Tecnológica Federal do Paraná (UTFPR). Alguns destaques são:

- *Interfaces Físicas e Lógicas*[56], PUC-Rio de 2013 criada por João Bonelli, professor da PUC-Rio e supervisor acadêmico do Laboratório de Interfaces Físicas Experimentais (CNPQ, 2015);
- *Algoritmos I (S73) @ UTFPR - 2014.2 UTFPR*[57], criada pela Professora Doutora Silvia Amelia Bim, que tem algumas outras salas de aula criadas no site;

[56] openprocessing.org/classroom/247
[57] openprocessing.org/classroom/4367

- *Algoritmos I (S73) @ UTFPR - 2015.1 Sistemas de Informação, Bacharelado*[58], mesma disciplina lecionada pela Professora Doutora Silvia Amelia Bim, criada pelo Professor Doutor Luiz Ernesto Merkle, membro ativo da comunidade tendo criado três outras salas de aula na comunidade para cursos na mesma universidade;
- *Creative Computing at SESC Pompeia Brasil*[59], criada por Radamés Ajna, artista e criador do curso Hacklab no SESC Pompéia em São Paulo, oficina de ensino de *Processing* e *Arduino*.

Segundo o Ascioglu, o uso do *OpenProcessing* nas ementas de diferentes cursos foi o fator mais importante para o crescimento da comunidade – uma evidência da relevância da colaboração *online* como ferramenta de ensino e aprendizado do *Processing*. Nas salas de aula do *OpenProcessing*, notam-se algumas características da cultura *hacker* funcionando lado a lado com a educação institucional, um sinal de como esse processo de aprendizado informal baseado no compartilhamento *online* pode ser combinado com o processo formal da sala de aula no ensino de *Processing*, indicando uma possível metodologia para educadores que utilizam o *Processing* para o ensino de programação nas artes no Brasil e em outros lugares do mundo.

### 3.3 Do artista-funcionário ao artista-programador

O *Processing* é programação, mas é também uma interface que simplifica a programação. E ao facilitar a escrita de código para artistas e designers, a linguagem se torna uma ferramenta que transforma o apertador de botões em programador. Posiciona-se, assim, como uma ponte através da qual o "funcionário" caminha até chegar ao universo do "artista experimental". De acordo com Hildebrand, em entrevista para esta pesquisa, o *Processing* pode ser usado ou não como caixa-preta, pois a linguagem permite diferentes níveis de atuação, tanto do artista-programador quanto do que ele chama de artista-funcionário.

[58] openprocessing.org/classroom/4762
[59] openprocessing.org/classroom/2811

Quando se usa uma biblioteca pronta, por exemplo, é comum se empregar o código da biblioteca como caixa-preta, pois o programador muitas vezes inclui a biblioteca sem compreender como ela funciona. Para demonstrar essa questão, a seguir será examinado um exemplo de *sketch* criado pela autora em conjunto com outros participantes do encontro ZL Camp na PUC-SP, em 2014, parte do projeto ZL Vórtice[60], coordenado pelo Professor Doutor Nelson Brissac. A aplicação criada em *Processing* durante o ZL Camp (Figura 38) é um teste de possibilidades de mapeamento da Zona Leste, ou seja, um protótipo inicial.

O *sketch* utiliza a biblioteca *Unfolding Maps*[61] e alguns dados de campo para sobrepor informação ao mapa. O código-fonte completo da aplicação se encontra no Apêndice D. As marcas em azul são pontos de ônibus e o polígono destacado, a área de atuação do projeto. Para importar a biblioteca no *Processing*, tudo o que o programador precisa fazer é escrever algumas linhas de código que são ensinadas no tutorial da biblioteca disponível no site da *Unfolding Maps*. Veja a seguir:

```
import de.fhpotsdam.unfolding.mapdisplay.*;
import de.fhpotsdam.unfolding.utils.*;
import de.fhpotsdam.unfolding.marker.*;
import de.fhpotsdam.unfolding.tiles.*;
import de.fhpotsdam.unfolding.interactions.*;
import de.fhpotsdam.unfolding.ui.*;
import de.fhpotsdam.unfolding.*;
import de.fhpotsdam.unfolding.core.*;
import de.fhpotsdam.unfolding.mapdisplay.shaders.*;
import de.fhpotsdam.unfolding.data.*;
import de.fhpotsdam.unfolding.geo.*;
import de.fhpotsdam.unfolding.texture.*;
import de.fhpotsdam.unfolding.events.*;
import de.fhpotsdam.utils.*;
import de.fhpotsdam.unfolding.providers.*;
```

[60] zlvortice.wordpress.com
[61] unfoldingmaps.org

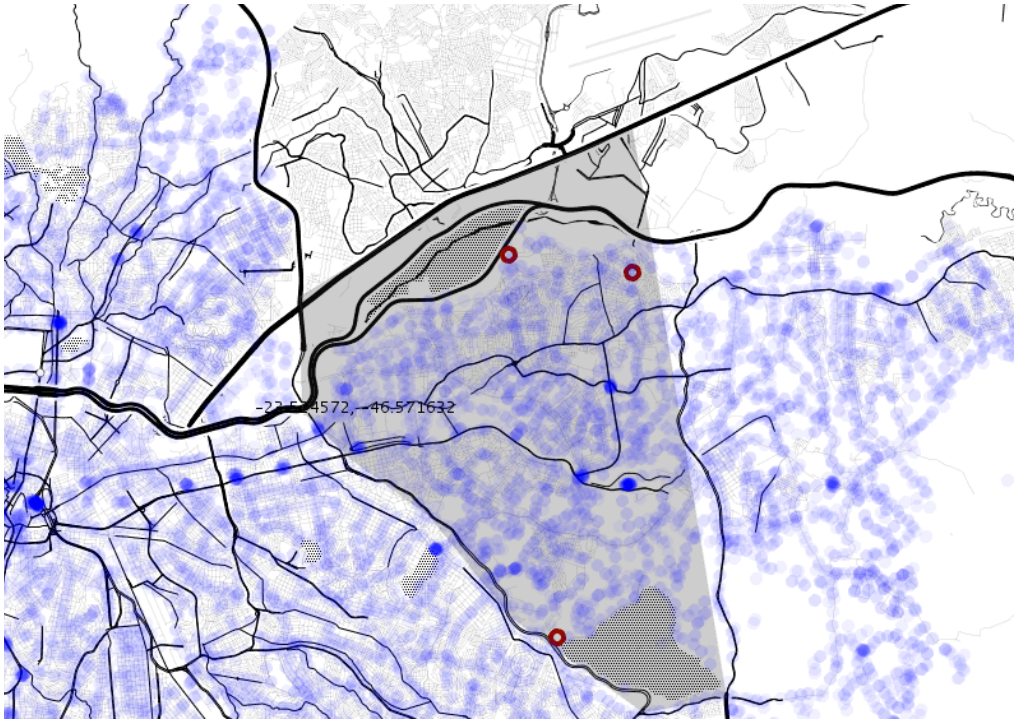



Figura 38 –Protótipo de mapeamento em *Processing* para o projeto ZL Vórtice. Fonte: ZL (2015).

Depois de importada a biblioteca, o programador pode começar a criar a sua aplicação utilizando as funções disponíveis. Apesar de ser necessário algum conhecimento de programação, não é preciso saber, por exemplo, como criar um mapa ou até mesmo como modificar a estética do mapa, pois já existem funções prontas na biblioteca para esta finalidade. Alterando apenas uma linha de código, é possível mudar completamente a aparência do mapa, por exemplo:

```
map = new UnfoldingMap(this, new StamenMapProvider.Toner());
map = new UnfoldingMap(this, new StamenMapProvider.WaterColor());
```

Cada uma das linhas de código acima cria um novo objeto "*map*", da classe *UnfoldingMap* existente na biblioteca. Não é necessário fazer nada além de incluir uma das duas opções acima no programa para criar resultados completamente diferentes, como se pode observar na Figura 39, que mostra o resultado de cada uma das duas linhas de código mostradas acima. A *StamenMapProvider.Toner* está à esquerda e a

*StamenMapProvider.WaterColor* à direita. Além dos dois exemplos aqui mencionados, há vários outros disponíveis na biblioteca.


Figura 39 – Opções de mapa na biblioteca *Unfolding Maps*. Esquerda: StamenMapProvider.Toner e Direita: StamenMapProvider.WaterColor. Fonte: Imagem capturada pela pesquisadora.

A troca do tipo de mapa com o uso da biblioteca *Unfolding Maps* no *Processing* é semelhante ao uso de filtros no Photoshop, como descrito no Capítulo 1. Do mesmo modo que um artista utilizando o Photoshop para produzir sua obra usa filtros já programados, neste exemplo do *Processing*, o artista também utiliza opções programadas por outra pessoa. Apesar de estar trabalhando com o código, o programador, neste caso, está apenas exercendo uma escolha dentre as opções possíveis, utilizando, portanto, a biblioteca *Unfolding Maps* como uma caixa-preta.

Por essa razão, é importante refletir sobre o *Processing* e sobre outras linguagens alternativas que facilitam a programação imaginando uma escala ou um espectro com diferentes níveis de controle sobre a máquina, tanto do ponto de vista do que a linguagem permite quanto do ponto de vista do conhecimento do programador. Uma das vantagens do *Processing* e das outras linguagens de programação alternativas, como *Scratch* e *Arduino*, é que os programadores dessas linguagens conseguem produzir resultados rapidamente com uma noção básica de programação. Isso é diferente de linguagens mais robustas, como, por exemplo a linguagem C, que exige um conhecimento de programação maior. Desse modo, o programador de *Processing* pode transitar entre os papéis de artista experimental ou de funcionário – em uma perspectiva flusseriana – no seu processo de criação. E ao criar o código, copiar programas e até mesmo usar

partes de código e bibliotecas como caixas-pretas, o artista-funcionário de *Processing* vai aos poucos adquirindo mais conhecimento e se tornando um artista-programador.

Nas mãos de um programador experiente, no entanto, o *Processing* pode ser usado para execuções bastante complexas. A seguir, será examinada a série *Process* de Casey Reas, com o objetivo de demonstrar as potencialidades do *Processing*. *Process* é, como o nome diz, uma série de obras criadas a partir de processos de *software* na qual o artista trabalhou de 2004 a 2010. Nessa obra, o primeiro passo do processo de criação de Reas envolve a definição de elementos, que são uma combinação de uma forma geométrica com um ou mais comportamentos. Abaixo um exemplo:

ELEMENTO 1

Forma 1: Círculo
Comportamento 1: Mover-se em linha reta
Comportamento 2: Não sair da tela
Comportamento 3: Mudar de direção ao encostar em outro Elemento
Comportamento 4: Afastar-se de um Elemento sobreposto

O Elemento 1 é, então, a combinação da Forma 1 com Comportamento 1, Comportamento 2, Comportamento 3 e Comportamento 4 (E1 = F1 + C1 + C2 + C3 + C4). Uma imagem estática do resultado, que é na verdade uma imagem em movimento, pode ser observado na Figura 40[62].

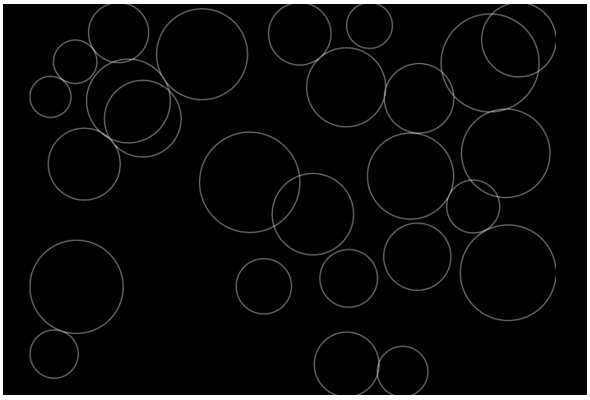


Figura 40 – Elemento 1 da obra *Process* de Casey Reas. Fonte: Process (2011).

[62] Uma explicação completa do processo de criação da série *Process* e animações disponíveis em: vimeo.com/22955812 (Acesso em 31 ago. 2015).

Abaixo está a definição de um outro elemento, o Elemento 3:

ELEMENTO 3

F2: Linha
C1: Mover-se em linha reta
C3: Mudar de direção ao encostar em outro Elemento
C5: Entrar pelo outro lado da tela após sair

Após a definição dos Elementos, Reas cria Processos, que são regras de execução em texto para serem interpretadas por *software*. A obra *Process 4*, por exemplo, utiliza o Elemento 1, descrito anteriormente, que já tem uma forma e comportamentos definidos, e incluiu as seguintes regras para criar o Processo: desenhar uma linha entre os centros dos elementos que se tocam, colorir as menores linhas de preto e as maiores de branco com uma escala de cinza para os tamanhos intermediários. Uma imagem do resultado pode ser vista na Figura 41.

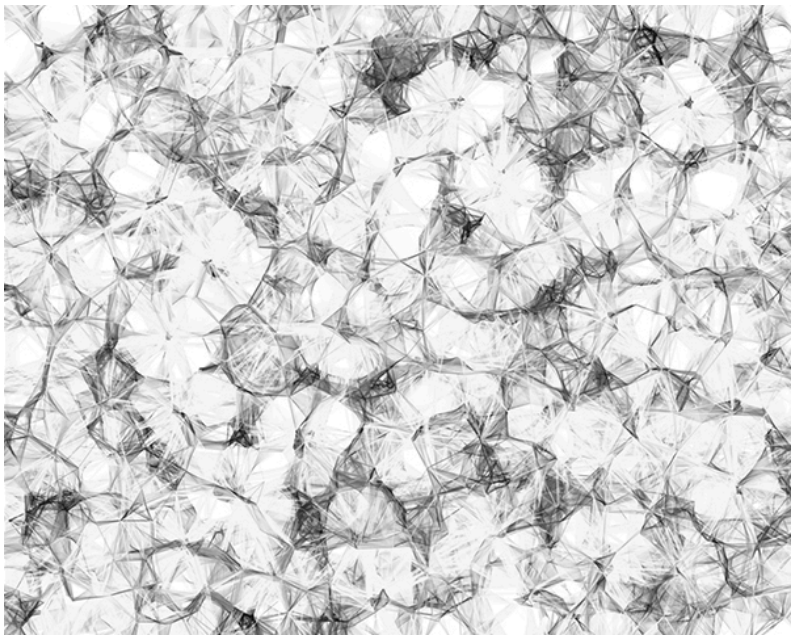


Figura 41 – *Process 4* de Casey Reas. Fonte: Reas (2005a).

E seguindo esse método o artista cria várias outras obras com novos Elementos e novos Processos. O resultado é uma série de animações e imagens criadas a partir de *software*, como mostrado na Figura 42.

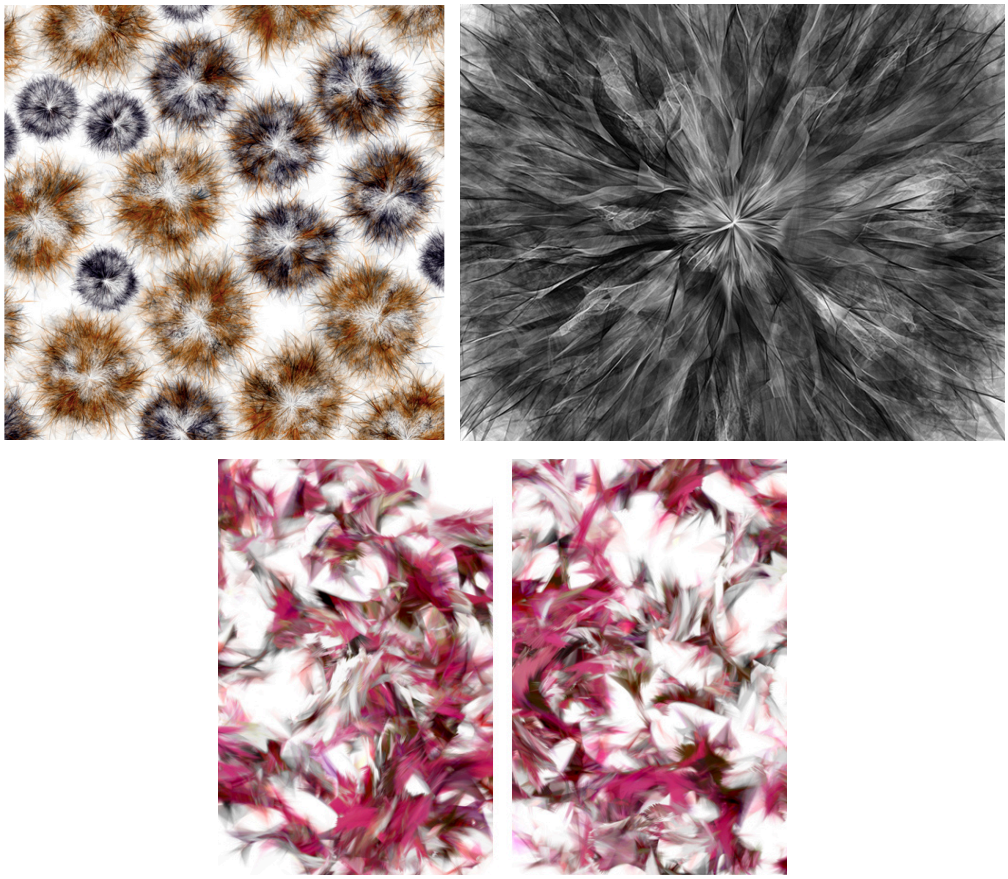


Figura 42 – Canto superior esquerdo: *Process 6* de Casey Reas (2005); canto superior direito: *Process 10* de Casey Reas (2005); embaixo: *Process 18* de Casey Reas (2010). Fontes: Reas (2005b), Reas (2005c) e Reas (2010).

Depois da elaboração do programa, o artista cria trabalhos derivados (Figura 43), como é o caso de *Process 4* (Figura 41), por exemplo, que foi incluído em instalações artísticas, performances em que o artista interpreta a música através de imagens ao vivo e também transformado em esculturas por meio de impressões em 3D.

Com base no exemplo da obra de Reas é possível perceber o potencial do *Processing* quando usado por artistas-programadores experientes. Se nas mãos de principiantes o *Processing* é uma interface e uma linguagem simplificada, nas mãos de programadores com conhecimento ele é uma ferramenta poderosa. Como explica Shiffman (2008, p.xii), “um equívoco comum é pensar que o *Processing* só serve para brincar; esse não é o caso. Pessoas (inclusive eu) estão por aí usando o *Processing* do dia 1 ao dia 365 do seu projeto”.

Figura 43 – Obras derivadas de *Process 4* de Casey Reas. Em cima: instalação em galeria; no meio: performance ao vivo com orquestra; embaixo: impressões em 3D. Fonte: Process (2011).

A diferença entre o *Processing* e as interfaces simplificadas que produzem resultados instantâneos ao se apertar um botão, como é o caso da câmera fotográfica, é que ele, assim como o *Dynabook* de Alan Kay (descrito na Seção 2.1, *Linguagens de programação alternativas*), não discrimina o programador de um usuário. E nessa junção de papéis, oferece um caminho de aquisição de conhecimento computacional e alfabetização de código.

Além disso, o *Processing* convida os artistas a entrar no mundo da computação por meio do aprendizado prático, da abertura da informação e da troca de conhecimento com outros membros da comunidade. E percebendo ou não, ao adotar esses comportamentos os artistas-programadores do *Processing* aderem à cultura *hacker*.

Por conseguinte, é possível compreender como o *Processing* não se encaixa em uma visão dicotômica da tecnologia computacional como sendo ou não uma caixa-preta. Como foi demonstrado, o artista-programador de *Processing* utiliza uma interface que simplifica a programação e pode usar partes do código como caixas-pretas. O *Processing* cria, assim, diferentes níveis de transparência e visibilidade do que está dentro da caixa-preta na medida em que o usuário adquire conhecimento sobre como funciona a máquina e avança no entendimento da computação.

# Conclusão

O objetivo desta pesquisa foi desvendar as relações entre *software*, programação e arte pela perspectiva da linguagem *Processing*. Com base em uma conceituação de *software* como prática cultural, e não somente como entidade abstrata que funciona nos bastidores do computador, realizou-se uma trajetória teórica. Em primeiro lugar, o *software* na sociedade atual e sua utilização (ou não) como caixa-preta foi analisado, levando à discussão sobre o aprendizado de programação como uma questão de alfabetização digital. Depois, em um movimento em direção às linguagens de programação, material de trabalho do artista que cria a partir do código-fonte, examinou-se o papel do artista-programador além de se terem investigado também as linguagens de programação alternativas e a cultura *hacker*. Na análise sobre o *Processing*, os conceitos destacados acima foram aplicados e discutidos com base em dados coletados, entrevistas e experimentos realizados pela pesquisadora.

Ao final, verifica-se que a primeira hipótese da pesquisa, de que o sucesso do *Processing* se deve à licença de *software* livre e à cultura *hacker,* está correta. Esses aspectos culturais fazem parte não só da comunidade de desenvolvedores do projeto, como também se apresentam na prática de criação artística com a linguagem, revelando os desdobramentos da cultura *hacker*, da abertura de informação e da colaboração *online* na produção de arte e no ensino de programação em *Processing*.

Já a segunda hipótese, de que o *Processing* surgiu em uma época em que os artistas computacionais passaram a aprender programação para criar suas obras, tornando-se nesse processo artistas-programadores, se confirma apenas parcialmente. Para a elucidação das conclusões, cabe separar essa hipótese em duas partes: o papel do artista-programador e o crescimento do interesse por programação nas artes. Em relação ao papel do artista-programador, na verdade, já no início da arte computacional, grande parte dos artistas pioneiros eram engenheiros e cientistas da computação, ou seja, artistas-programadores. Como demonstrado a partir do conceito de *téchne grega*, o papel de um artista que tem domínio desses dois universos não é um fenômeno recente. Assim

sendo, o papel do artista-programador, aquele que domina tanto a técnica quanto a estética em seu processo de criação artística, tem raízes históricas antigas.

Contudo, apesar de não se ter verificado uma mudança significativa em relação ao papel do artista-programador do ponto de vista conceitual, uma constatação é o crescimento do interesse pela computação por programadores alternativos, inclusive artistas, atrelado à expansão de linguagens alternativas que facilitam a programação. Isso fica claro com os números de utilização de algumas das linguagens abordadas e também com o crescimento do número de *hackerspaces* – ou *hackerlabs*, *media labs*, *makerspaces* e *fab labs* – no Brasil e no mundo, evidenciando uma expansão da cultura *hacker* e do interesse por computação e programação na sociedade.

Dessa maneira, apesar de não se verificar o surgimento de um perfil novo de artista, o artista-programador, pode-se concluir que existe uma ampliação da busca pelo aprendizado de programação por programadores alternativos, inclusive provenientes do mundo das artes, que muitas vezes utilizam linguagens de programação alternativas como *Processing*, *Arduino*, *Scratch*, entre outras. O *Processing* não só faz parte desse movimento, como teve um papel central no desenvolvimento de uma sintaxe de linguagem, de um ambiente de desenvolvimento e de uma filosofia de computação no universo da facilitação da programação para não-engenheiros. Um protagonista na expansão dos *hackerspaces* é o *Arduino*, projeto inspirado diretamente pelo *Processing*. E apesar de não haver evidências de que o *Processing* foi referência também do *Scratch*, sabe-se que ambos os projetos tiveram como inspiração o *Dynabook* de Alan Kay e compartilham o seu mesmo objetivo de transformar o usuário em programador, estando, portanto, de alguma maneira relacionados. As causas do crescimento do número de *hackerspaces* assim como os seus objetivos, que podem ser de ativismo político, criação artística ou de geração de inovação, não foram aqui abordados e podem, portanto, se consolidar em um intrigante objeto de pesquisa para uma futura investigação.

Um caminho relevante para educadores que utilizam a linguagem *Processing* – ou o *Scratch* ou o *Arduino* – é incorporar em suas metodologias de ensino o processo de aprendizado *hacker*, que é fragmentado, informal, voltado para a prática e baseado na abertura da informação. Observa-se que a possibilidade de incorporar a cultura *hacker* no processo de ensino de linguagens de programação alternativas tem o potencial de

ampliar as formas de aprendizado dos recém-chegados ao mundo da programação. Seguir alguns princípios como aprendizado prático, abertura da informação e criação de canais de troca de conhecimento (*online* ou não) que estimulem o processo de aprendizado informal baseado na interação entre membros da comunidade pode ser frutífero. Para ilustrar esse ponto, vale destacar um contato, no dia 28 agosto de 2015, com o Professor Doutor Jean Carlo Rossa Hauck, da Universidade Federal de Santa Catarina e membro da organização *Computação na Escola*, citada no Capítulo 2. Durante o encontro a pesquisadora compartilhou essa conclusão sobre a cultura *hacker* alcançada na pesquisa. Por email enviado à pesquisadora no dia 02 de setembro de 2015 (informação pessoal), Hauck explica uma possível aplicação dessa recomendação:

> atualmente já temos trabalhado com a perspectiva de aprendizado livre em mente, tanto que o material é sempre disponibilizado sob Creative Commons e código livre. O ponto em que percebo que podemos melhorar seria na possibilidade de fomentar uma comunidade que propicie a interação e o aprendizado na forma como você muito bem descreve. Já estive verificando as possibilidades aqui de abrir uma área no nosso site para possibilitar esse tipo de funcionalidade. Entretanto, não será tão simples como parecia, pois dependemos de algumas questões burocráticas e técnicas (necessidade de atualização da plataforma para suportar o plugin, por exemplo), sem falar na atual restrição de recursos [...] Mas, de qualquer forma, é uma ideia muito boa que pode ser aplicada sem que necessite qualquer mudança de rumo, somente questões de infra e software.

Apesar dos desafios de recursos enfrentados pela iniciativa Computação na Escola, essa experiência já é capaz de demonstrar o potencial de aplicação prática do conhecimento adquirido nesta investigação em iniciativas de ensino de programação para crianças e artistas. Ademais, os resultados da criação do fórum *online* no site Computação na Escola podem ser um objeto de estudo futuro.

No percurso da pesquisa, diversas outras questões afloraram e se mostraram material fértil. Do ponto de vista teórico, assim como diferentes linguagens de programação produzem estéticas diversificadas na arte, também cabe perguntar qual é a influência das linguagens de programação na sociedade de uma maneira mais ampla. Alguns desdobramentos da pesquisa podem surgir a partir desse questionamento, colocando como foco, principalmente, o papel político do *software*. Um outro tópico de pesquisa possível são as máquinas de desenhar utilizadas por artistas de diferentes

tempos, nesta pesquisa exemplificadas pelos perspectógrafos de Albrecht Dürer (Figura 12) do século 16 e o programa AARON de Harold Cohen (Figura 13) do século 20, podendo levar a um estudo sobre as técnicas e as tecnologias usadas em práticas artísticas de regiões e tempos diversos, inclusive embasando uma discussão sobre autoria e até que ponto a arte pode ser automatizada. E por último, também em relação à autoria, uma possível pesquisa é investigar as práticas de autoria compartilhada, por meio da reutilização de código aberto nas comunidades de arte e os efeitos provocados no entendimento do que é a criação artística e no direito autoral.

Quando criaram o *Processing*, em 2001, Casey Reas e Ben Fry afirmaram que seus objetivos eram desafiar o modelo de *software* proprietário vigente na época e aplicar o espírito da inovação *open source* na arte. Quinze anos depois, é possível afirmar que eles conseguiram. O *Processing* demonstra como o trabalho colaborativo e a abertura da informação provenientes da cultura *hacker* podem ser instrumentos transformadores.

O mundo roda em *software*. E a tendência é um crescimento cada vez maior da lógica algorítmica no nosso cotidiano. Além das habituais interfaces usadas para a comunicação e consumo de conteúdo diariamente, estamos assistindo à emergência de novas possibilidades computacionais com aprendizado de máquina, inteligência artificial, dispositivos *wearable*, *Big Data*, internet das coisas, robótica e computadores quânticos. Na medida em que o *software* se torna mais presente na sociedade, aprender a lógica da programação passa a ser cada vez mais imprescindível, pois independente das novas formas que o computador pode adquirir, ele ainda é, pelo menos por enquanto, uma máquina de rodar *software*.

## REFERÊNCIAS BIBLIOGRÁFICAS


BANZI, Massimo. **Getting started with Arduino.** Sebastopol: O'Reilly Media, 2009.

BARRIÈRE, L. Arte y algoritmos. In: VII JORNADAS DE MATEMÁTICA DISCRETA Y ALGORÍTMICA, 2010, Castro Urdiales. **Anais eletrônicos**. Disponível em <http://www.jmda2010.unican.es/p2.html>. Acesso em: 29 ago. 2015.

BENJAMIN, Walter. **A obra de arte na época de sua reprodutibilidade técnica**. Porto Alegre: Zouk, 2012.

CAETANO, Alexandre Cristina Moreira. **Interface**: processos criativos em arte computacional. 2010. 169 f. Dissertação (Mestrado em Arte). Instituto de Artes, Universidade de Brasília, 2010. Disponível em < http://repositorio.unb.br/handle/10482/7172>. Acesso em: 10 ago. 2015.

CASTELLS, Manuel. **A galáxia da internet**: reflexões sobre a internet, os negócios e a sociedade. Rio de Janeiro: Zahar, 2003.

COHEN, Harold. The further exploits of AARON, painter. **Stanford Humanities Review**, n.2, v.4, 1995. Disponível em <http://haroldcohen.com/aaron/publications/furtherexploits.pdf>. Acesso em 10 jul. 2015.

CORDEIRO, Waldemar. Arteônica. In: MASCARENHAS, Nelson; VELHO, Luiz; HESS, Lilia. **Catálogo da exposição Sibgrapi**, 1993. Disponível em <http://www.visgraf.impa.br/Gallery/waldemar/catalogo/catalogo.pdf>. Acesso em: 12 abr. 2015.

COSTA, Rachel Cecília de Oliveira. A relação entre a língua e a imagem em Vilém Flusser. **Revista Ghrebh**, 1, mai. 2008. Disponível em <http://revista.cisc.org.br/ghrebh/index.php?journal=ghrebh&page=article&op=view&path%5B%5D=16&path%5B%5D=20>. Acesso em: 11 mar. 2015.

COX, Geoff; MCLEAN, Alex. **Speaking code**: coding as aesthetic and political expression. Cambridge: MIT Press, 2013.

CRAMER, Florian; GABRIEL, Ulrike. *Software* Art. **American Book Review**, issue "Codeworks", 2001. Disponível em < http://cramer.pleintekst.nl/all/software_art_and_writing/software_art_and_writing.pdf> . Acesso em 15 mai. 2015.

CRAMER, Florian. **Words Made Flesh**: code, culture, imagination. Roterdam: Piet Zwart Institute, 2005. Disponível em <http://www.netzliteratur.net/cramer/wordsmadefleshpdf.pdf>. Acesso em 25 set. 2015.

FEURZEIG, Wallace et al. Programming-Languages as a Conceptual Framework for Teaching Mathematics. **Final Report on the First Fifteen Months of the LOGO Project,** 1969. Disponível em <http://eric.ed.gov/?id=ED038034>. Acesso em: 8 out 2014.

FLORES, Claudia. Teoria e representação geométrica na obra de Albrecht Dürer: um ensino de matemática para pintores e artesãos. **Revista Iberoamericana de Educación Matemática**, n.11, p.179-188, set. 2007. Disponível em <http://www.fisem.org/www/union/revistas/2007/11/Union_011_016.pdf>. Acesso em 17 jul. 2015.

FLUSSER, Vilém. **Filosofia da caixa preta**: ensaios para uma futura filosofia da fotografia. São Paulo: HUCITEC, 1985.

GAMA, Ruy (org.). **História da técnica e da tecnologia**: textos básicos. São Paulo: Editora da Universidade de São Paulo, 1985.

GREENBERG, Ira. ***Processing***: creative coding and computational art. Berkley: Apress, 2007.

HIMANEN, Pekka. **A ética dos hackers e o espírito da era da informação**. Rio de Janeiro: Campus, 2001.

HOLM, Van; JOSEPH, Eric. What are Makerspaces, Hackerspaces, and Fab Labs?. **Social Science Research Network (SSRN)**, 7 nov. 2014. Disponível em <http://ssrn.com/abstract=2548211>. Acesso em 26 jul. 2015.

**ICT FACTS & FIGURES: The world in 2015.** Geneva: International Telecommunication Union, 2015. Anual. Disponível em: <http://www.itu.int/en/ITU-D/Statistics/Documents/facts/ICTFactsFigures2015.pdf>. Acesso em: 26 set. 2015.

KAY, Alan; GOLDBERG, Adele. Personal Dynamic Media. In: WARDRIP-FRUIN, Noah; MONTFORT, Nick. **The New Media Reader**. Cambridge: The MIT Press, 2003.

KITTLER, Friedrich. There is No Software. **Ctheory**, [s. L.], 18 out. 1995. Disponível em: <http://www.ctheory.net/articles.aspx?id=74>. Acesso em: 26 set. 2015.

LEVY, Steven. **Hackers**: heroes of the computer revolution. Sebastopol: O'Reilly Media, 2010.

MACHADO, Arlindo. Tecnologia e arte contemporânea: como politizar o debate. **Revista de Estudios Sociales**, Bogotá , n.22, dec. 2005. Disponível em <http://www.scielo.org.co/scielo.php?script=sci_arttext&pid=S0123-885X2005000300006&lng=en&nrm=iso>. Acesso em: 12 set. 2015.

_____. Máquina e Imaginário. In: DOMINGUES, Diana. **Arte, ciência e tecnologia**: passado, presente e desafios. São Paulo: UNESP, 2009.

_____. **Arte e mídia.** Rio de Janeiro: Jorge Zahar, 2010.

MANDELBROT, Benoit. Fractais: uma forma de arte a bem da ciência. In: PARENTE, André. **Imagem Máquina**: a era das tecnologias do virtual**.** São Paulo: Editora 34, 1993.

MANOVICH, Lev. Estudos do Software. In: PERISSINOTO, Paula; BARRETO; Ricardo**. Teoria Digital**: dez anos do FILE - Festival Internacional de Linguagem Eletrônica. São Paulo: Imprensa Oficial do Estado de São Paulo, 2010.

_____. ***Software* Takes Command.** Nova York: Bloomsbury, 2013.

MAXIGAS. Hacklabs and hackerspaces: tracing two genealogies. **Journal Of Peer Production**: New perspectives on the implications of peer production for social change, [s. L.], v. 2, jul. 2012. Disponível em: <http://peerproduction.net/issues/issue-2/peer-reviewed-papers/hacklabs-and-hackerspaces/>. Acesso em: 15 jul. 2015.

MCLEAN, Alex Christopher. **Artist-Programmers and Programming Languages for the Arts.** 2011. 172 f. Tese (Doctor of Philosophy) [online]. Goldsmiths, University of London, 2011. Disponível em < http://research.gold.ac.uk/6611/1/COMP_thesis_McLean_2011.pdf>. Acesso em 15 set. 2015.

MENDES, Cássia Isabel Costa. ***Software* livre e inovação tecnológica**: uma análise sob a perspectiva da propriedade intelectual. 2006. 269f. Dissertação (Mestrado em Desenvolvimento Econômico). Instituto de Economia, Universidade Estadual de Campinas, 2006. Disponível em < http://www.bibliotecadigital.unicamp.br/document/?code=vtls000378144>. Acesso em 15 jul. 2015.

MOSCATI, Giorgio. Waldemar Cordeiro e o uso do computador: depoimento sobre uma experiência pioneira. In: MASCARENHAS, Nelson; VELHO, Luiz; HESS, Lilia. **Catálogo da exposição Sibgrapi**, 1993. Disponível em <http://www.visgraf.impa.br/Gallery/waldemar/catalogo/catalogo.pdf>. Acesso em: 12 abril 2015.

NAKE, Frieder. Computer art: a personal recollection. 5th Conference On Creativity & Cognition, Londres, p.54-62, abr. 2005. **Anais eletrônicos.** Disponível em: <http://design.osu.edu/carlson/history/PDFs/p54-nake.pdf>. Acesso em: 26 set. 2015.

_____. Algorithmic art. **Leonardo**, Editorial, [s. L.], v.47, n.2, p.108, 20 mar. 2014. Disponível em: <http://www.mitpressjournals.org/doi/abs/10.1162/LEON_a_00706?journalCode=leon#.Vgb8zCBViko>. Acesso em: 20 set. 2015.

NASCIMENTO, Raimundo Benedito do. Investigações em geometria via ambiente LOGO. **Ciência & Educação**, vol.10, n.1, pp. 1-21, 2004. Disponível em < http://www.scielo.br/pdf/ciedu/v10n1/01.pdf>. Acesso em 20 set. 2015.

NETO, Antonio Francisco Moreira. ***Software* [livre] na arte computacional**. 2010. 111f. Dissertação (Mestrado em Arte). Instituto de Artes, Universidade de Brasília, 2010. Disponível em < http://repositorio.unb.br/handle/10482/8573>. Acesso em 10 set. 2015.

OLIVEIRA, Eva Aparecida. A técnica, a techné e a tecnologia. **Itinerarius Reflectionis**: Revista Eletrônica da Pós-Graduação em Educação. UFG - Regional Jataí, [s. L.], n.5, jul./dez. 2008. Disponível em: <http://revistas.jatai.ufg.br/index.php/ritref/article/viewFile/20417/11905>. Acesso em: 20 set. 2015.

ORBEN, Douglas João. Imagens técnicas: origem e implicações segundo Vilém Flusser. **Comunicação & Informação**, [s. L.], v.16, n.1, p.113-126, jan. 2013. Disponível em: <http://revistas.jatai.ufg.br/index.php/ci/article/viewFile/25724/15335>. Acesso em: 26 set. 2015.

PAIXÃO, Dalton; MENEZES, Karina Moreira; SGANZERLLA, Sérgio. Aprendendo com a Ética Hacker. In: AMARAL, Sérgio Ferreira do. **Ética, hacker e a educação.** Campinas: FE/UNICAMP, 2012. Disponível em < http://www.lantec.fe.unicamp.br/lantec/publicacoes/amaral,pretto.pdf#page=41>. Acesso em 12 set. 2015.

PAPERT, Seymour. **Mindstorms**: children, computers, and powerful ideas. Nova York: Basic Books, 1980.

PAROS, Felipe Martins. O mineiro e a máquina: vida e obra de Erthos Albino de Souza. In: Simpósio Internacional de Inovação em Mídias Interativas, Goiânia, abr. 2014. **Anais eletrônicos.** Disponível em < http://siimi.medialab.ufg.br/up/777/o/13_mineiro_maquina.pdf>. Acesso em 14 set. 2015.

PRETTO, Nelson. Redes colaborativas, ética hacker e educação. **Educação em Revista**, Belo Horizonte v.26, n.3, p.305-316, dec. 2010. Disponível em <http://www.scielo.br/scielo.php?script=sci_arttext&pid=S0102-46982010000300015&lng=en&nrm=iso>. Acesso em 27 abr. 2015.

PUCKETTE, Miller. Pure Data: another integrated music environment. Second Intercollege Computer Music Concerts, Tachikawa, 1997. **Anais eletrônicos.** Disponível em < http://www.researchgate.net/profile/Miller_Puckette/publication/2596372_Pure_Data_another_integrated_computer_music_environment/links/00b7d52fd000bdc769000000.pdf>. Acesso em 20 set. 2015.

____Max at seventeen. **Computer Music Journal**, Boston, v.26, n.4, p.31-43, 2002. Disponível em < http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.76.8354&rep=rep1&type=pdf >. Acesso em 10 set. 2015.

RAYMOND, Eric S. A brief history of hackerdom. In: DIBONA, Chris; OCKMAN, Sam; STONE, Mark. **Open Sources**: voices from the open source revolution. Sebastopol: O'Reilly Media, 1999. Disponível em <http://www.oreilly.com/openbook/opensources/book/raymond.html>. Acesso em 26 jul. 2015.

REAS, Casey. Programming Media. CODE: The Language of Our Time, Ars Electronica, 2003. **Catálogo eletrônico.** Disponível em < http://90.146.8.18/en/archives/festival_archive/festival_catalogs/festival_artikel.asp?iProjectID=12322>. Acesso em 19 set. 2015.

REAS, Casey; FRY, Ben. Processing.org: programming for artists and designers. SIGGRAPH'04, 2004. **Anais eletrônicos**. Disponível em < http://yeoahn.com/saic_fall/paper/processing.pdf>. Acesso em 17 abr. 2015.

_____. Processing: programming for the media arts. **AI & Society**, [s. L.], v.20, p.526-538, 2006. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.207.7452&rep=rep1&type=pdf>. Acesso em: 20 set. 2015.

REAS, Casey; MCWILLIAMS, Chandler. **Form + code in design, art, and architecture**. Nova York: Princeton Architectural Press, 2010.

RESNICK, Mitchel et al. Scratch: Programming for All. **Communications Of The ACM,** [s.l.], v.52, n.11, p.60-67, 1 nov. 2009. Association for Computing Machinery (ACM). DOI: 10.1145/1592761.1592779. Disponível em: <http://web.media.mit.edu/~mres/papers/Scratch-CACM-final.pdf>. Acesso em: 26 set. 2015.

RIBEIRO, Madalena. As Linguagens de Programação para Artes - Metodologias de Ensino-Aprendizagem Adaptadas. **Convergências** [on-line], 2007. Disponível em <http://convergencias.esart.ipcb.pt/artigo.php?id=140>. Acesso em 24 nov. 2014. ISSN: 1646-9054.

ROBERTS, Eric S. **The Art & Science of Java**: an introduction to computer science. Boston: Pearson Education, 2008.

RUSHKOFF, Douglas. **Program or be programmed**: ten commands for a digital age. Minneapolis: 2010.

SCHACHMAN, Toby. Alternative programming interfaces for alternative programmers. ACM international symposium on New ideas, new paradigms, and reflections on programming and *software*, p.1-10, 2012. **Anais eletrônicos**. Disponível em < http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.476.6506&rep=rep1&type=pdf>. Acesso em 12 set. 2015.

SELIGMANN-SILVA, Márcio. De Flusser a Benjamin – do pós-aurático às imagens técnicas. **Flusser Studies**, v.8, mai. 2009. Disponível em <http://www.flusserstudies.net/sites/www.flusserstudies.net/files/media/attachments/seligmann-flusser-benjamin.pdf>. Acesso em: 3 set. 2015

SHIFFMAN, Daniel. **Learning Processing**: A Beginner's Guide to Programming Images, Animation, and Interaction. Burlington: Morgan Kaufmann Publishers, 2008.

STALLMAN, Richard. The GNU Manifesto. In: WARDRIP-FRUIN, Noah; MONTFORT, Nick. **The New Media Reader.** Cambridge: The MIT Press, 2003.

**Entrevistas:**

ASCIOGLU, Sinan. Entrevista realizada pela pesquisadora em 11 de maio de 2014.

HILDEBRAND, Hermes Renato. Entrevista realizada pela pesquisadora em 27 de agosto de 2015.

**Jornais e revistas:**

GOMES, Helton Simões. SP pagará hackers para melhorar trânsito; salário é de até R$ 5,9 mil: SPTrans abrirá Laboratório de Mobilidade, que reunirá desenvolvedores. Eles criarão software e aperfeiçoarão equipamentos eletrônicos. **G1**. São Paulo, p.1-1. 13 mar. 2014. Disponível em: <http://g1.globo.com/tecnologia/noticia/2014/03/sp-pagara-hackers-para-melhorar-transito-salario-e-de-ate-r-59-mil.html>. Acesso em: 20 set. 2015.

LUCKERSON, Victor. Netflix Accounts for More Than a Third of All Internet Traffic. **Time**. [s. L.], maio 2015. Disponível em: <http://time.com/3901378/netflix-internet-traffic/>. Acesso em: 26 set. 2015.

LUMINI, Milena. Iniciativa Computação na Escola leva ensino de programação às crianças: Projeto da UFSC quer estimular gosto pela tecnologia e incentivar a formação profissional na área. **Diário Catarinense.** Santa Catarina, 4 jun. 2014. Disponível em: <http://diariocatarinense.clicrbs.com.br/sc/economia/noticia/2014/06/iniciativa-computacao-na-escola-leva-ensino-de-programacao-as-criancas-4517812.html>. Acesso em: 14 set. 2015.

RICHTEL, Matt. Programação vira disciplina em escolas infantis nos EUA. **Folha de São Paulo**. Original publicado no New York Times. Mill Valley, 20 maio 2014. Tec. Disponível em: <http://www1.folha.uol.com.br/tec/2014/05/1456608-programacao-vira-disciplina-em-escolas-infantis-nos-eua.shtml>. Acesso em: 20 set. 2015.

**Sites:**

ACM SIGGRAPH Awards - Harold Cohen, Distinguished Artist Award for Lifetime Achievement. Vancouver, 2014. (22 min.), Youtube. Gravação da Palestra de Harold Cohen na conferências ACM SIGGRAPH Awards em 2014. Disponível em: < https://youtu.be/_Xbt8lzWxIQ?t=8m25s>. Acesso em: 20 set. 2015.

ALBRECHT Durer: An artist drawing a seated man (1525).jpg. Disponível em: <https://commons.wikimedia.org/wiki/File:Albrecht_Durer_-_An_artist_drawing_a_seated_man_(1525).jpg>. Acesso em: 20 set. 2015.

ARDUINO. Disponível em <http://www.arduino.cc>. Acesso em 11 out. 2014.

ARDUINO FORUM. Disponível em < http://forum.arduino.cc/>. Acesso em 12. set. 2015.

ASCIOGLU, Sinan. Visualizing OpenProcessing. 2012. Disponível em: <http://www.openprocessing.org/labs/opviz>. Acesso em: 10 abr. 2014.

CHOI, Young. 2012 Museum Night Live Coding Show. 2012. Disponível em: <http://youngchoi.org/livecoding2012/>. Acesso em: 20 set. 2015.

CNPQ - Busca Textual. Disponível em <http://buscatextual.cnpq.br/buscatextual/visualizacv.do?id=K4413122A7 >. Acesso em 31. mai. 2015.

COHEN, Harold. 40502: Pigment on paper, 46 by 57.75 inches 1/3. 2004. Disponível em: <http://aaronshome.com/aaron/gallery/FS-main-galleryS2.html>. Acesso em: 20 set. 2015.

DESIGN By Numbers: Walking through how Design By Numbers worked as a simple way to understand computational drawings. 2013. (1:27 min.), Vimeo. Vídeo postado pro John Maeda. Disponível em: <https://vimeo.com/72611093>. Acesso em: 26 set. 2015.

DICTIONARY. Disponível em <http://dictionary.reference.com/>. Acesso em 1. jun. 2015.

DRAVES, Scott**.** Electric Sheep. Disponível em: <http://scottdraves.com/sheep.html>. Acesso em: 26 set. 2015.

ELECTRIC Sheep - Generation 245 - Sheep 2706. Disponível em: <http://v2d7c.sheepserver.net/cgi/dead.cgi?id=2706>. Acesso em: 20 set. 2015.

FAQ, Processing. Disponível em: <https://github.com/processing/processing/wiki/FAQ>. Acesso em: 20 set. 2015.

FOLDER da Exposição Erthos Albino de Souza - poesia: do dáctilo ao dígito, Instituto Moreira Salles, Rio de Janeiro, 24 ago. – 3 out. 2010. Disponível em <http://cronopios.com.br/anexos/ims_erthos_folder.pdf>. Acesso em 15 abr. 2015.

GLUBGLUB ~ A Fish Game! 2015. Projeto na plataforma *Scratch*. Disponível em: <https://scratch.mit.edu/projects/69205452/?fromexplore=true#player>. Acesso em: 20 set. 2015.

GOOGLE ANALYTICS. Disponível em <http://www.google.com/analytics/>. Acesso restrito em: 25 ago. 2015.

HENDRICKSON, Jeff. AngularRectangularRev. 2009. Sketch publicado na comunidade OpenProcessing. Disponível em: <http://openprocessing.org/sketch/4578>. Acesso em: 20 set. 2015.

ISLAMIC TILING PATTERNS. 2014. Disponível em: <http://maths.myblog.arts.ac.uk/2014/12/03/islamic-tiling-patterns/>. Acesso em: 20 set. 2015.

LIBRARIES. Extend Processing beyond graphics and images into audio, video, and communication with other devices. Site do Processing (processing.org). Lista das 107 bibliotecas mencionadas na dissertação em https://goo.gl/bwNV1N. Disponível em: <https://processing.org/reference/libraries/>. Acesso em: 12 jul. 2015.

LIBRARY Overview. Repositório do *Processing* no GitHub. Disponível em: <https://github.com/processing/processing/wiki/Library-Overview>. Acesso em: 12 jul. 2015.

LIST of Hacker Spaces. 2015. Lista de hackerspaces utilizada para a criação do gráfico em https://goo.gl/zKq05l. Disponível em: <https://wiki.hackerspaces.org/List_of_Hacker_Spaces>. Acesso em: 26 jul. 2015.

LUSSAC, Olivier. TV Cello @ Nam June Paik. 2009. Disponível em: <http://www.artperformance.org/article-21865022.html>. Acesso em: 14 set. 2015.

MAX 7 is Patching Reimagined. Disponível em: <https://cycling74.com/max7/>. Acesso em: 03 out. 2015.

ONTOLINUX. Disponível em <http://www.ontolinux.com/community/PureData/pure-data-prostredi.jpg>. Acesso em 7 jun. 2015.

OPENPROCESSING CLASSROOMS. Lista das mais de 1000 salas de aula mencionadas na dissertação em https://goo.gl/QeOePU. Disponível em < http://www.openprocessing.org/classrooms/>. Acesso em 31 mai. 2015.

PROCESS Compendium (Introduction). Casey Reas. 2011. (11 min.), Vimeo. Disponível em: <https://vimeo.com/22955812>. Acesso em: 31 ago. 2015.

PROCESSING. Disponível em <http://processing.org/>. Acesso em: 15 abr. 2014.

PROCESSING BOOKS. Disponível em <https://processing.org/books/>. Acesso em 26 set. 2015.

PROCESSING FORUM. Disponível em <http://forum.processing.org/two/>. Acesso em: 14. set. 2015.

PROCESSING FOUNDATION. Disponível em: <http://foundation.processing.org/>. Acesso em: 20 set. 2015.

RAHM, Gernot. Little lights. 2013. Sketch publicado na comunidade OpenProcessing. Disponível em: <http://www.openprocessing.org/sketch/107765>. Acesso em: 20 set. 2015.

REAS, Casey. Process 4. 2005a. Disponível em: <http://reas.com/p4_s/>. Acesso em: 20 set. 2015.

_____. Process 6. 2005b. Disponível em: <http://reas.com/p4_s/>. Acesso em: 20 set. 2015.

_____. Process 10. 2005c. Disponível em: <http://reas.com/p4_s/>. Acesso em: 20 set. 2015.

_____. Process 18. 2010. Disponível em: <http://reas.com/p4_s/>. Acesso em: 20 set. 2015.

SCRATCH STATISTICS. Disponível em <https://scratch.mit.edu/statistics/>. Acesso em 19 jul. 2015.

SMALLTALK GUI. 1980. Media Library da Xerox Parc. Disponível em: <https://www.parc.com/content/news/media-library/historical_smalltalk_gui_6.14x7.22_parc.jpg>. Acesso em: 20 set. 2015.

STACK EXCHANGE. Disponível em <http://stackexchange.com/about>. Acesso em 26 jul. 2015.

STACK OVERFLOW TOUR. Disponível em <http://stackoverflow.com/>. Acesso em 26 jul. 2015.

STACK OverFlow. Perguntas postadas com a tag Processing. Disponível em: <http://stackoverflow.com/questions/tagged/processing>. Acesso em: 12 jul. 2015.

THORNTON, Mary. Golden Slope. 2007. Disponível em: <http://www.fractalartcontests.com/2007/showentry.php?entryid=286&return=winners>. Acesso em: 20 set. 2015.

VITALINO: Jarbas Jácome. 2015. Disponível em: <http://file.org.br/artist/jarbas-jacome/?lang=pt>. Acesso em: 20 set. 2015.

WERGO. Disponível em < http://www.wergo.de/shop/php/Proxy.php?purl=/rssh/Noten/Musikpaedagogik/Musikspiele/show,100655,s.html>. Acesso em 20 set. 2015.

ZL Vórtice: Intervenções Urbanas – Laboratório. 2014. Apontamentos para a Criação de um Mapa em Processing. Disponível em: <https://zlvortice.wordpress.com/Processing/>. Acesso em: 20 set. 2015.

**Bibliografia complementar:**

ARANTES, Priscila. Arte e mídia no Brasil: perspectivas da estética digital. **ARS (São Paulo)**, São Paulo, v.3, n.6, p. 52-65, 2005. Disponível em: <http://www.scielo.br/scielo.php?script=sci_arttext&pid=S1678-53202005000200004&lng=en&nrm=iso>. Acesso em: 20 set. 2015.

COTTON, Bob; OLIVER, Richard. **Understanding hypermedia 2.000**: multimedia origins, internet futures. Londres: Phaidon Press, 1997.

GREELISH, David. An Interview with Computing Pioneer Alan Kay. **Time**, [s. L.], 2 abr. 2013. Disponível em: <http://techland.time.com/2013/04/02/an-interview-with-computing-pioneer-alan-kay/>. Acesso em: 20 set. 2015.

DBN, MIT Media Lab. Disponível em <http://dbn.media.mit.edu/faq.html>. Acesso em: 25 mai. 2014.

GUTIERREZ, R. M.; ALEXANDRE, P. A. M. **Complexo eletrônico**: introdução ao *software*. BNDES. 2004. Disponível em < http://www.bndes.gov.br/SiteBNDES/export/sites/default/bndes_pt/Galerias/Arquivos/conhecimento/bnset/set2001.pdf>. Acesso em 10 jul. 2015.

PERISSINOTO, Paula. Arte e tecnologia: uma história porvir. In: PERISSINOTO, Paula; BARRETO; Ricardo. **Teoria Digital**: dez anos do FILE - Festival Internacional de Linguagem Eletrônica. São Paulo: Imprensa Oficial do Estado de São Paulo, 2010

POPPER, Frank. As imagens artísticas e a tecnociência (1967-1987). In: PARENTE, André. **Imagem Máquina**: a era das tecnologias do virtual**.** São Paulo: Editora 34, 1993.

___. **Art in the electronic age.** Nova York: Thames and Hudson, 1997.

STOCKER, Gerfried. CODE – the Language of Our Time**.** In: CODE: The Language of Our Time, Ars Electronica, 2003. **Catálogo eletrônico**. Disponível em <http://90.146.8.18/en/archives/festival_archive/festival_catalogs/festival_artikel.asp?iProjectID=12495>. Acesso em 19 abr. 2015.

TAKHTEYEV, Yuri. From Brazil to Wikipedia: The Surprising Journey of a Programming Language from Rio. **Foreign Affairs**, [s. L.], 21 abr. 2013. Disponível em: <https://www.foreignaffairs.com/articles/2013-04-21/brazil-wikipedia>. Acesso em: 10 jun. 2015.

VENTURELLI, Suzete. *Software* ARt/códIgO ARte. 9° Encontro Internacional de Arte e Tecnologia (#9ART): sistemas complexos artificiais, naturais e mistos. **Anais eletrônicos.** Brasília: Instituto de Artes da Universidade de Brasília, 2010. Disponível em <http://www.fav.ufg.br/9art/nono_art.pdf>. Acesso em 25 nov. 2014.

WARDRIP-FRUIN, Noah. Estudos do *Software*. In: PERISSINOTO, Paula; BARRETO; Ricardo**.** **Teoria Digital**: dez anos do FILE - Festival Internacional de Linguagem Eletrônica. São Paulo: Imprensa Oficial do Estado de São Paulo, 2010.

WENGER, Etienne. Communities of practice and social learning systems. **Organization,** [s. L.], v. 7, n. 2, 2000. Disponível em: <http://wenger-trayner.com/wp-content/uploads/2012/01/09-10-27-CoPs-and-systems-v2.01.pdf>. Acesso em: 20 set. 2015.

**APÊNDICE A – Código-fonte do *sketch* em *Processing* que produziu a imagem utilizada na capa da dissertação desenvolvido pela autora.**

```
/*
Código-fonte usado para a criação da capa da dissertação
Imagem gerada a partir do código-fonte do Processing disponível no
GitHub: https://github.com/processing/processing/
Stekch usa o código-fonte em core > src > processing > core.
Acesso no dia 20/set/2015.
Patricia Oakim - setembro/2015

Links úteis para a elaboração deste sketch:
http://forum.processing.org/two/discussion/4311/how-to-turn-
characters-of-a-text-into-images
http://forum.processing.org/two/discussion/1631/#Comment_4810
http://stackoverflow.com/questions/11726023/split-string-into-
individual-words-java
http://stackoverflow.com/questions/5554734/what-causes-a-java-
lang-arrayindexoutofboundsexception-and-how-do-i-prevent-it
*/



import processing.pdf.*;

void setup() {

  size(2480, 5060, PDF, "cover33.pdf");
  background(255);
```

```
  //load txt file with Processing source code and breaks every
line into an array of strings
  String s[] = loadStrings("Processing_SRC_Test2.txt");

  //for loop to break every line of Processing source code into
separate strings with words
  for (int i=0; i < s.length; i++) {
    String[] arr = s[i].split(" ");

      //for loop to check strings in arr array and draw on the
screen

      for (int j=0; j < arr.length; j++) {
        if (arr[j].contains("int")) {
          textSize(j*20);
          fill(255, 128, 0, random(230, 255));
          text("int", random(displayWidth),
random(displayHeight));
        }
        if (arr[j].contains(";")) {
          textSize(j*40);
          fill(0, random(103, 203), 0, random(150, 255));
          text(";", random(displayWidth), random(displayHeight));
        }
        if (arr[j].contains("while")) {
          textSize(j*100);
          fill(random(80), random(150), random(70, 200), random(3,
3));
          text("while", random(displayWidth),
random(displayHeight));
        }
        if (arr[j].contains("+=")) {
          textSize(j*100);
          fill(random(80), random(150), random(70, 200),
random(50, 200));
          text("+=", random(displayWidth), random(displayHeight));
        }
        if (arr[j].contains("()")) {
          textSize(j*15);
          fill(random(80), random(150), random(70, 200),
random(200, 220));
          text("()", random(displayWidth), random(displayHeight));
        }
        if (arr[j].contains("for")) {
          textSize(j*15);
          fill(random(40), random(5), random(10, 90), random(10,
200));
          text("for", random(displayWidth),
random(displayHeight));
        }
        if (arr[j].contains("x++")) {
          textSize(j*30);
          fill(255, 128, 0, random(100, 200));
          text("x++", random(displayWidth),
random(displayHeight));
        }
```

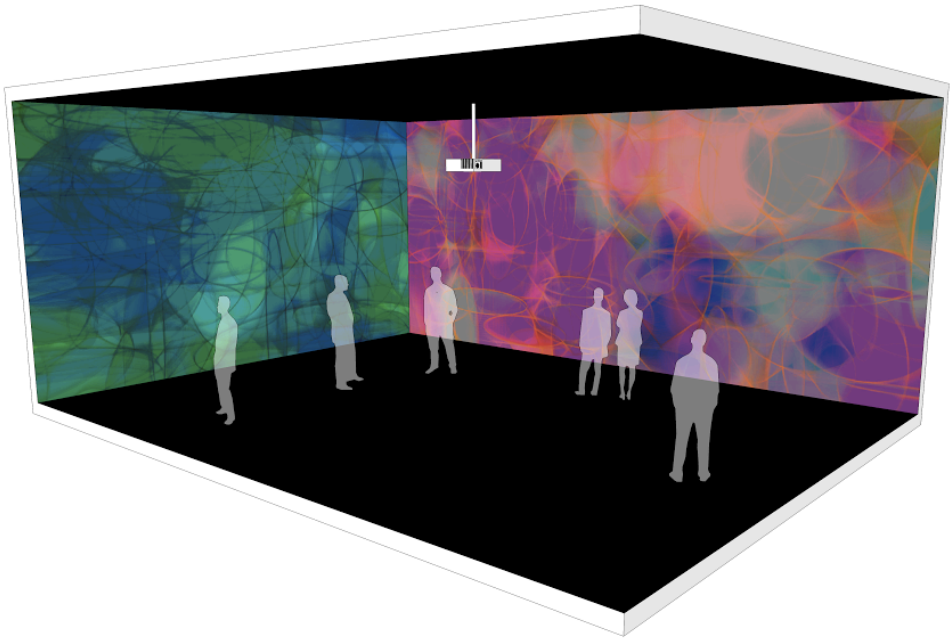
```
        if (arr[j].contains("y++")) {
          textSize(j*30);
          fill(random(80), random(150), random(70, 200), random(3,
200));
          text("y++", random(displayWidth),
random(displayHeight));
        }
        if (arr[j].contains("{")) {
          textSize(j*100);
          fill(random(80), random(150), random(70, 200), random(3,
3));
          text("{", random(displayWidth), random(displayHeight));
        }
        if (arr[j].contains("}")) {
          textSize(j*100);
          fill(random(255), random(150), random(70, 200),
random(50, 200));
          text("}", random(displayWidth), random(displayHeight));
        }
      }
    }
  }

  println("Finished.");
  exit();
}
```

**APÊNDICE B – Lista de *hackerspace*s ativos no Brasil**

Fonte: List (2015).

| Nome | Fundação |
|---|---|
| Weekend Tinkers | 10/31/2009 |
| Mirako | 1/17/2010 |
| Intellecta | 1/3/2011 |
| Hackerspace Jaragua | 1/5/2012 |
| CG Hackspace | 10/9/2011 |
| Garoa Hacker Clube | 8/28/2010 |
| Hackerspace Natal | 6/14/2013 |
| LabHacker Câmara dos Deputados | 1/1/2014 |
| Tarrafa Hacker Clube | 5/25/2013 |
| HackerSpace Maringá | 10/5/2013 |
| Laboratório Hacker de Campinas | 10/12/2011 |
| Evoino Lab | 2/5/2012 |
| Tesla Hacker Lab | 1/12/2012 |
| Club Hacker JP | 9/14/2012 |
| Concas Hackerspace | 7/10/2013 |
| SJC Hacker Clube | 8/13/2011 |
| Area31 | 4/29/2013 |
| Dumont Hackerspace | 5/7/2014 |
| Fabrica de Ideias | 4/12/2014 |
| XAP-Hacks | 8/9/2013 |
| Triangulo Hackerspace | 2/17/2013 |
| Hackerspace Londrina | 8/19/2014 |
| Garagemhacker | 11/14/2012 |
| Teresinahc | 5/25/2014 |
| Matehackers | não disponível |
| KackLab Sorocaba | 8/7/2014 |
| Raul Hacker Club | 3/29/2013 |
| Laboratório Hacker | 11/1/2014 |

**APÊNDICE C – Código-fonte do protótipo da Instalação Trânsito criado pela autora.**



```
/* OpenProcessing Tweak of
*@*http://www.OpenProcessing.org/sketch/4578*@* */
// Patricia Oakim, 2013
// Originals: Angular Rectangular from Jeff Hendrickson, Sept 09 -
-- based on orig code from Ira Greenberg.

//Importa a biblioteca de áudio
import ddf.minim.*;
AudioPlayer player;
Minim minim;

//Cria um vetor de objetos da Classe Circle
Circle[] arrayCircles;
int numCircles = 400;

// Cria um vetor para colorir os objetos Circles
color[]colorsDay = new color[numCircles];

//Inicia as variáveis globais para hora, minuto e segundo
int h;
int m;
int s;

//Inicia a variável global para quilometragem de trânsito
int km = 0;

//Inicia a variável global para traduzir quilometragem
//de trânsito em velocidade de movimento das elipses
int speed = 0;
```

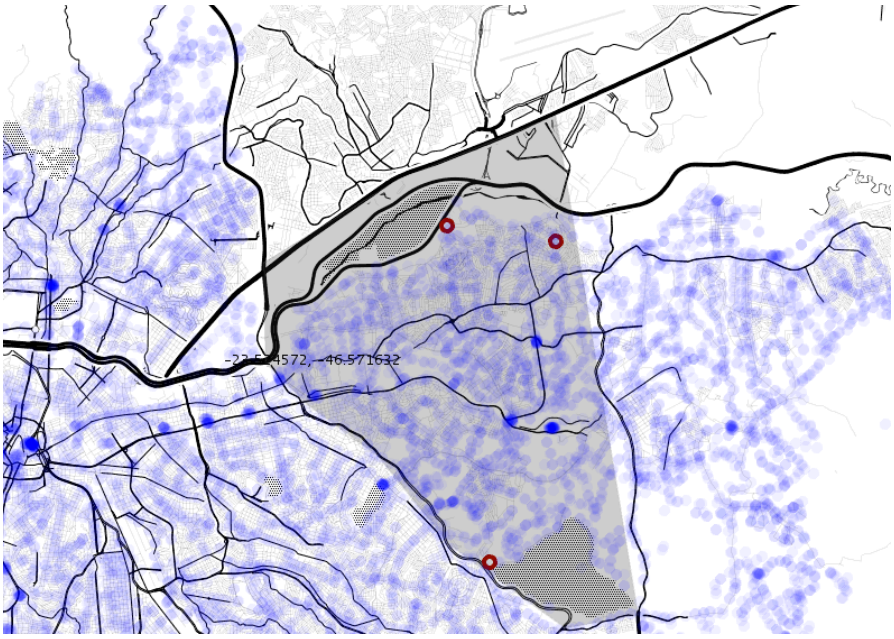
```
//Inicia duas variáveis semáforo para controlar
//chamados ao site da CET SP
boolean checkKm = false;
boolean switcher = false;

void setup() {
  size(1000, 800);
  smooth();
  frameRate(18); //muda o framerate, default é 60

//Cria cada um dos objetos da classe Circle
arrayCircles = new Circle[numCircles];


for (int i=0; i < arrayCircles.length; i++) {
   arrayCircles[i] = new Circle(random(width), random(height),
random(-1, 1), random(-1, 1), random(width/4), random(height/3));
   colorsDay[i] = color(random(255), random(150), random(70, 200),
random(3, 3));
}

//Inicia o player e coloca o arquivo de áudio em loop
minim = new Minim(this);
player = minim.loadFile("audio_traffic.mp3", 2048);
player.loop();
}

//Cria método para checar o congestionamento no site da CET
//e determinar a velocidade das elipses
void traffic() {
  if (m % 2 == 0 && !checkKm) {
    if (switcher) {
      return;
    }
    else {
      switcher = true;
    }

//Acessa o site da CET SP e checa 'km' da região Centro
 String[]data=loadStrings("http://www.cetsp.com.br/cet.aspx");
 println("accessed the site!");
 String linhaCorreta = "";
 String texto = join(data, "\n");
 String[] m = match(texto, "<div class=\"info centro\">(.*?)(\\d+)
km");
 String[] m2 = match(m[0], "(\\d+)");
 km = Integer.parseInt(m2[0]);
 println(km+" Km of traffic");
 checkKm = true;

if ( km >= 0 && km <= 4 && speed != 4) {
      speed = 4;
      println("Speed changed to "+speed);
    }
    else if ( km >= 5 && km <= 8 && speed !=3) {
      speed = 3;
```

```
      println("Speed changed to "+speed);
    }
    else if ( km >=9) {
      speed = 1;
      println("Speed changed to "+speed);
    }
    else {
      print("Speed still the same.");
    }
  }

  else if (m % 2 != 0) {
    checkKm = false;
  }
  switcher = false;
}

void draw() {

  //Checa no relógio do computador a hora, minuto e segundo
  h = hour();
  m = minute();
  s = second();

  //Chama o método traffic em processamento paralelo
  thread("traffic");

//Desenha as ellipses, passando a velocidade (speed)
  //e alterando as cores do stroke
  for (int i=0; i < arrayCircles.length; i++) {
    if ( m % 2 == 0 && (s == 11 || s == 12 || s == 13 || s == 14
|| s == 15)) {
      arrayCircles[i].xDirection = random(-speed, speed);
      arrayCircles[i].yDirection = random(-speed, speed);
    }
    if (m==7 || m==12 || m==19 || m==27 || m==33 || m==39 | m==43
|| m==53) {
      stroke(218, 165, 32, 25);
    }
    else if (m==2 || m==10 || m==23 || m==49 || m==57) {
      stroke(255, 20, 147, 25);
    }
    else if (m== 5 || m==16 || m==29 || m==40 || m==50) {
      stroke(255, 0, 0, 25);
    }
    else {
      stroke(255, 0, 0, 25);
    }
    fill(colorsDay[i]);
    arrayCircles[i].displayCircle();
  }
}

//Para o áudio quando botão stop é clicado
void stop()
{
```

```
  player.close();
  minim.stop();
  super.stop();
}

//Cria class Circle com parâmetros de tamanho e movimento
class Circle {
  float x;
  float y;
  float xDirection;
  float yDirection;
  float circleWidth;
  float circleHeight;

Circle(float _x, float _y, float _xDirection, float _yDirection,
float _circleWidth, float _circleHeight){
  x = _x;
  y = _y;
  xDirection = _xDirection;
  yDirection = _yDirection;
  circleWidth = _circleWidth;
  circleHeight = _circleHeight;
}

void moveCircle(){
  x += xDirection;
  y += yDirection;

  if (y >= height || y <= 0){
    yDirection = yDirection * -1;
  }

  if (x >= width || x<= 0){
    xDirection = xDirection *-1;
  }
}

void displayCircle(){
  ellipse(x, y, circleWidth, circleHeight);
  moveCircle();
}
}
```

**APÊNDICE D – Código-fonte da aplicação de mapeamento realizado no encontro ZL Camp, em abril de 2014.**

Disponível também em < https://zlvortice.wordpress.com/camp*Processing*2/>. Acesso em 29 ago. 2015.



```
// BIBLIOTECA UNFOLDING MAPS
import de.fhpotsdam.unfolding.mapdisplay.*;
import de.fhpotsdam.unfolding.utils.*;
import de.fhpotsdam.unfolding.marker.*;
import de.fhpotsdam.unfolding.tiles.*;
import de.fhpotsdam.unfolding.interactions.*;
import de.fhpotsdam.unfolding.ui.*;
import de.fhpotsdam.unfolding.*;
import de.fhpotsdam.unfolding.core.*;
import de.fhpotsdam.unfolding.mapdisplay.shaders.*;
import de.fhpotsdam.unfolding.data.*;
import de.fhpotsdam.unfolding.geo.*;
import de.fhpotsdam.unfolding.texture.*;
import de.fhpotsdam.unfolding.events.*;
import de.fhpotsdam.utils.*;
import de.fhpotsdam.unfolding.providers.*;

UnfoldingMap map;
String[] dots;

void setup() {
size(800, 600, P2D);
//  map = new UnfoldingMap(this, new
OpenStreetMap.OpenStreetMapProvider());
//  map = new UnfoldingMap(this, new
OpenStreetMap.OSMGrayProvider());
```

```
//  map = new UnfoldingMap(this, new
StamenMapProvider.WaterColor()); //WATERCOLOR
map = new UnfoldingMap(this, new StamenMapProvider.TonerLite());
//  map = new UnfoldingMap(this, new StamenMapProvider.Toner());
//  map = new UnfoldingMap(this, new
StamenMapProvider.TonerBackground());

//Load CSV file
dots = loadStrings("bus_stops.csv");

//Set initial location of the map
Location zlVortice = new Location(-23.506, -46.504);
map.zoomAndPanTo(new Location(zlVortice), 12);
MapUtils.createDefaultEventDispatcher(this, map);

//Draw the polygon
drawVortice();

//Process CSV data
processData();
}


void draw() {

map.draw();
//Display coordinates on mouse position
Location here = map.getLocation(mouseX, mouseY);
fill(0);
text(here.getLat() + ", " + here.getLon(), mouseX, mouseY);
}

void drawVortice() {
Location Vortice1 =  new Location (-23.594751, -46.444786);
Location Vortice2 =  new Location (-23.598448, -46.467102);
Location Vortice3 =  new Location (-23.575089, -46.502636);
Location Vortice4 =  new Location (-23.55007, -46.524605);
Location Vortice5 =  new Location (-23.535902, -46.545883);
Location Vortice6 = new Location (-23.5239, -46.5589);
Location Vortice7 = new Location (-23.4959, -46.5594);
Location Vortice8 = new Location (-23.472622, -46.518417);
Location Vortice9 = new Location (-23.4546, -46.4700);
SimplePolygonMarker vorticeMarker = new SimplePolygonMarker();
vorticeMarker.addLocations
(new Location(Vortice1),
new Location(Vortice2),
new Location(Vortice3),
new Location(Vortice4),
new Location(Vortice5),
new Location(Vortice6),
new Location(Vortice7),
new Location(Vortice8),
new Location(Vortice9));
vorticeMarker.setColor(color(55, 55, 55, 60));
vorticeMarker.setStrokeColor(color(150, 150, 150, 50));
map.addMarkers(vorticeMarker);
```

```
// CANTEIRO 1 = AVENIDA ARICANDUVA
Location CT1Location = new Location(-23.58, -46.49);
SimplePointMarker CT1Marker = new SimplePointMarker(CT1Location);
map.addMarkers(CT1Marker);
CT1Marker.setColor(color(255, 0, 0, 0));
CT1Marker.setStrokeColor(color(150, 0, 0));
CT1Marker.setStrokeWeight(4);

// CANTEIRO 2 = AVENIDA JARDIM PANTANAL
Location CT2Location = new Location(-23.49, -46.47);
SimplePointMarker CT2Marker = new SimplePointMarker(CT2Location);
map.addMarkers(CT2Marker);
CT2Marker.setColor(color(255, 0, 0, 0));
CT2Marker.setStrokeColor(color(150, 0, 0));
CT2Marker.setStrokeWeight(4);


// CANTEIRO 3 = AVENIDA USP LESTE
Location CT3Location = new Location(-23.4856, -46.5030);
SimplePointMarker CT3Marker = new SimplePointMarker(CT3Location);
map.addMarkers(CT3Marker);
CT3Marker.setColor(color(255, 0, 0, 0));
CT3Marker.setStrokeColor(color(150, 0, 0));
CT3Marker.setStrokeWeight(4);
}

void processData() {
for (int i=1; i<dots.length; i++) {
String[] thisRow = split(dots[i], “,”);
Location thisLocation = new Location(float(thisRow[0]),
float(thisRow[1]));
SimplePointMarker here = new SimplePointMarker(thisLocation);
color c = color(0, 0, 255, 15);
here.setColor(c);
here.setStrokeWeight(0);
map.addMarkers(here);
}
}
```

**ANEXO 1 – Código-fonte do sketch AngularRectangularRev do usuário Jeff Hendrickson.**
Fonte: Hendrickson (2009).

```
//Angular Rectangular  ---  based on orig code from Ira Greenberg
//Jeff Hendrickson  Sept 09

/*  In Ira's original code, a number of colored squares started at
left and painted their
way at various angles to the right, leaving behind solid color.

I'm really fascinated with these techniques of painting and have
massaged the code into this,
on my way to a mashup of this and the Painting Worms sketch from
earlier.

This runs a bit faster than previous versions and stops at 15
seconds

*/

int shapes = 400;
float[]speedX = new float[shapes];
float[]speedY = new float[shapes];
float[]x = new float[shapes];
float[]y = new float[shapes];
float[]w = new float[shapes];
float[]h = new float[shapes];
color[]colors = new color[shapes];

void setup(){
  size(600, 320);
  frameRate(30);
  smooth();
  // fill arrays will random values
   for (int i=0; i<shapes; i++){
     x[i]=random(width);
     y[i]=random(height);
```

```
     w[i]=random(width/6);    //play   with   these   to   get   many
different size effects
     h[i]=random(height/3);
     colors[i]=color(random(200),     random(200),     random(255),
random(10,40));
     speedX[i] = random(-4, 10);  //angle and speed horizontal
     speedY[i] = random(-4, 10);  //positive int makes them all go
same way  speed vertical

   }
}
/*   if you want to do a screen cap uncomment this block
void mousePressed() {
  save("paintStripes800.tif");
}  */
 void draw(){
  for (int i=0; i<shapes; i++){
    fill(colors[i]);
    stroke(random(200),          random(240),          random(130),
random(40,80));  //this is what leaves behind the trail
    rect(x[i], y[i], w[i], h[i]);
    x[i]+=cos(speedX[i]);
    y[i]+=sin(speedY[i]);
  }
  if (millis() >= 15000) {
  noLoop();
}
}
```