Processing como Ferramenta para *Game Design*

Ricardo Nakamura[A] Romero Tori[A,B]

[A]Laboratório de Tecnologias Interativas
Departamento de Engenharia de Computação e Sistemas Digitais
Escola Politécnica da Universidade de São Paulo

[B]LPAI – Laboratório de Pesquisa em Ambientes Interativos
Mestrado em Design
Centro Universitário Senac

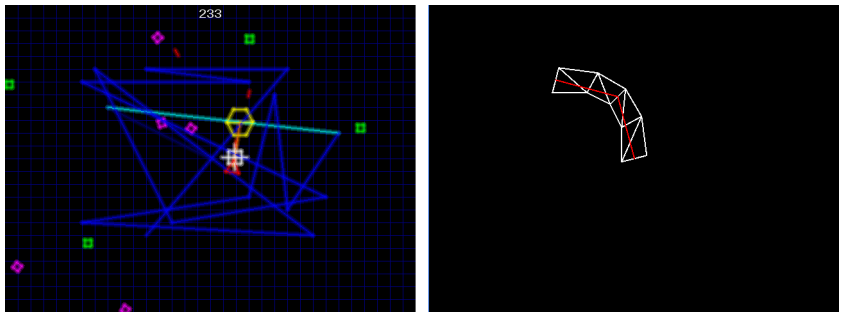

Figura 1: Exemplos de protótipos desenvolvidos no Processing. Esquerda: teste de *gameplay* de jogo controlado com movimentos do mouse; direita: simulação de animação *skinned mesh* bidimensional.

Abstract

We introduce in this tutorial the Processing development environment and discuss how it can be used to create interactive applications, and more specifically, game prototypes. Although there are other options for rapid prototyping of game design concepts, Processing has been consolidating itself as a programming tool for designers and artists, blending ease of use and extensibility.

Keywords: game design, game prototyping, game development tool.

Authors' contact:
{ricardo.nakamura,
romero.tori}@poli.usp.br

1. Introdução

O objetivo deste tutorial é apresentar o software Processing e discutir a sua utilização como alternativa para a criação de protótipos de *gameplay,* principalmente por *game designers.*

O Processing é um ambiente de desenvolvimento para aplicações multimídia, baseado na linguagem de programação Java e disponibilizado como software livre para diferentes plataformas. Originalmente, ele foi criado como ferramenta para auxiliar no ensino de fundamentos de programação para estudantes de design mas ao longo do tempo, evoluiu para o ambiente de desenvolvimento atual [Processing 2009].

A criação de protótipos é uma atividade essencial no trabalho de um *designer*. *Designers* de projeto visual fazem esboços e experimentos com *layouts* e cores; *designers* de produto constroem diferentes tipos de modelos. Em todos os casos, estes protótipos auxiliam o *designer* a visualizar, analisar e comunicar suas propostas.

Da mesma forma, protótipos são importantes na área de *game design,* para que o *designer* possa avaliar de forma ágil e simples suas idéias e conceitos de *gameplay*. No domínio dos jogos digitais, isto implica na necessidade de se escrever programas que implementem os protótipos, levando o *designer* a duas alternativas: colaborar com um programador, que será responsável por converter suas propostas em código-fonte ou desenvolver habilidades de programação. As duas abordagens apresentam vantagens e desvantagens; autores como Schuytema [2008] e Rouse III [2005] observam que a proliferação de linguagens de *scripting* em jogos digitais são outro motivador para que *game designers* adquiram conhecimentos sobre os fundamentos de lógica de programação. No entanto, é preciso que haja um compromisso entre a autonomia proporcionada pela programação de protótipos pelo *game designer* e o tempo consumido em tais atividades.

Neste contexto, o ambiente de desenvolvimento Processing surge como uma opção viável para que *designers* possam criar protótipos de jogos digitais sem a necessidade de dominar ferramentas de programação complexas. Sua origem – discutida anteriormente – e características (que serão abordadas ao longo do tutorial) o tornam atraente para o desenvolvimento rápido de protótipos e pequenos programas de teste. A Figura 1 apresenta imagens de dois protótipos desenvolvidos com o Processing. A imagem da

esquerda corresponde a um jogo em que o veículo controlado pelo jogador se move vinculado a outra estrutura maior, que tem uma trajetória definida. A imagem da direita corresponde a um teste de um sistema de animação baseada em uma malha deformável (*skinned mesh*) para gráficos 2D.

A sequência de tópicos deste tutorial busca apresentar os fundamentos da utilização do Processing antes de iniciar a discussão de como o utilizar para o desenvolvimento de protótipos de *gameplay*. Desta forma, a seção 2 apresenta o ambiente do Processing, incluindo uma explicação sobre sua interface de usuário e os principais recursos disponíveis. A distribuição padrão do Processing inclui um grande número de exemplos; por este motivo, essa seção apresenta uma seleção de alguns deles para ilustrar as capacidades desta ferramenta de software.

A seção 3 contém uma revisão (bastante resumida) de conceitos de programação e a maneira como são aplicados no ambiente do Processing. Recomenda-se que mesmo aqueles com experiência no desenvolvimento de software façam uma leitura breve desta seção para se familiarizar com os tipos de dados e objetos disponíveis no Processing.

A seção 4 apresenta a estrutura dos programas escritos em Processing, que são comumente chamados de *sketches*. Tanto os *sketches* estáticos como aqueles com comportamento dinâmico são discutidos nesta seção, acompanhados de exemplos.

A seção 5 é dedicada à discussão sobre a criação de protótipos de *gameplay* utilizando-se o Processing. Diferentes cenários são explorados, incluindo-se exemplos de implementação.

Por último, a seção 6 apresenta uma visão geral de outras bibliotecas de programação que podem ser utilizadas no ambiente do Processing, bem como informações sobre como incluir novas bibliotecas.

2. O Ambiente de desenvolvimento do Processing

Cada programa escrito no Processing é referido como um *sketch,* remetendo à idéia de esboço ou protótipo. O código-fonte é escrito em uma linguagem de programação derivada da linguagem Java, com pequenas modificações.

Ao se iniciar o ambiente de desenvolvimento, uma interface semelhante à da Figura 2 é apresentada (no caso desta figura, a interface corresponde à versão 1.0.5 do ambiente executando no sistema operacional Windows XP). Pode-se dividir esta interface de usuário em cinco áreas, organizadas verticalmente do topo até a base da janela: (A) barra de botões de atalho, (B) abas, (C) editor de texto, (D) barra de mensagens e (E) área de saída de texto.

A barra de botões de atalho permite o acesso rápido às operações mais comuns durante a edição de um programa. Da esquerda para a direita, os botões são utilizados para se executar o programa, forçar o término da execução, criar um novo programa, carregar, salvar ou exportar o programa.

As abas sobre a área de edição de textos identificam os arquivos que compõem um *sketch*. Pode-se escrever um programa todo em uma só aba ou criar novas abas (clicando-se no botão com uma seta para a direita, no canto direito da janela) para melhor organizar as informações do programa.

O editor de textos do ambiente de desenvolvimento possui alguns recursos de auxílio, como a utilização de cores distintas para representar palavras-chave da linguagem e a indentação automática. A barra de mensagens é utilizada para comunicar mensagens de erro do Processing, em geral relacionadas a problemas na compilação de um *sketch*. A área de saída de texto pode ser utilizada para a emissão de mensagens de texto geradas durante a execução do *sketch*. Quando um erro ocorre, as informações mais detalhadas sobre o erro também são mostradas nesta área.

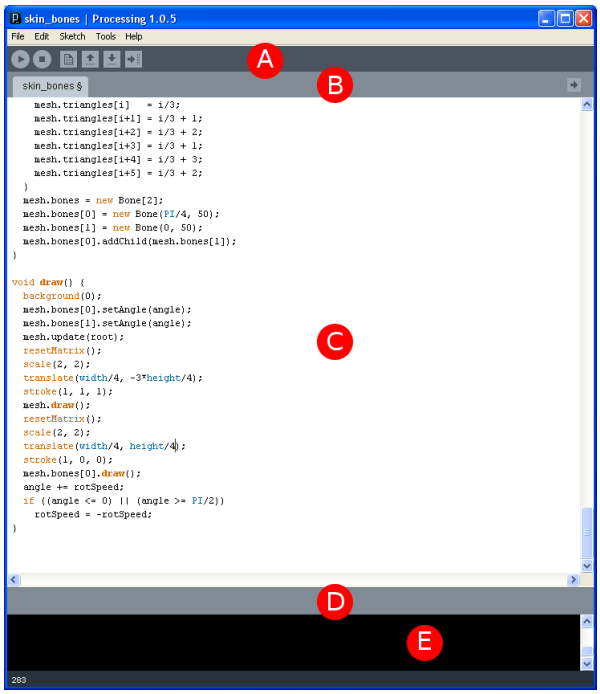

Figura 2: O ambiente de desenvolvimento do Processing. (A) botões de atalho; (B) abas; (C) editor de texto; (D) barra de mensagens; (E) saída de texto

2.1 Recursos de ajuda e referência

O ambiente de desenvolvimento do Processing também conta com alguns recursos de ajuda e referência, agrupados sob o menu *Help*. São documentos em formato HTML, que podem ser visualizados através de um navegador Web qualquer. Especificamente, os documentos contendo explicações sobre o ambiente e a referência dos comandos da

linguagem de programação são instalados localmente, de forma que não é preciso ter acesso a Internet para consultá-los. Outros documentos, como uma lista de perguntas frequentes, são recuperados remotamente.

Outro recurso de ajuda disponível no ambiente é a busca de palavras-chave digitadas no editor de texto. Basta selecionar a palavra e pressionar a combinação CTRL+SHIFT+F (ou selecionar a opção "*Find in Reference*" no menu *Help*). Caso a palavra não seja encontrada, uma mensagem será apresentada na barra de mensagens.

Por fim, o ambiente de desenvolvimento também conta com algumas outras ferramentas relacionadas à utilização de arquivos de imagens, áudio e fontes de caracteres em um *sketch*, que serão discutidas em outras seções deste tutorial.

2.2 Explorando os exemplos

Ao avaliar a utilidade de uma nova ferramenta, procura-se estabelecer quais suas capacidades e limitações. Desta forma, pode-se ter mais subsídios para se decidir se ela é adequada às necessidades que se tem no momento. Por este motivo, esta seção apresenta uma seleção de alguns dos mais de 250 exemplos que são distribuídos juntamente com o Processing.

A forma mais simples de acessar um dos exemplos é através do botão de carregar arquivos na barra de atalhos. Ao se clicar neste botão, um menu é apresentado. Os quatro itens inferiores no menu são submenus contendo listas de *sketches* de exemplo, conforme ilustrado na Figura 3.

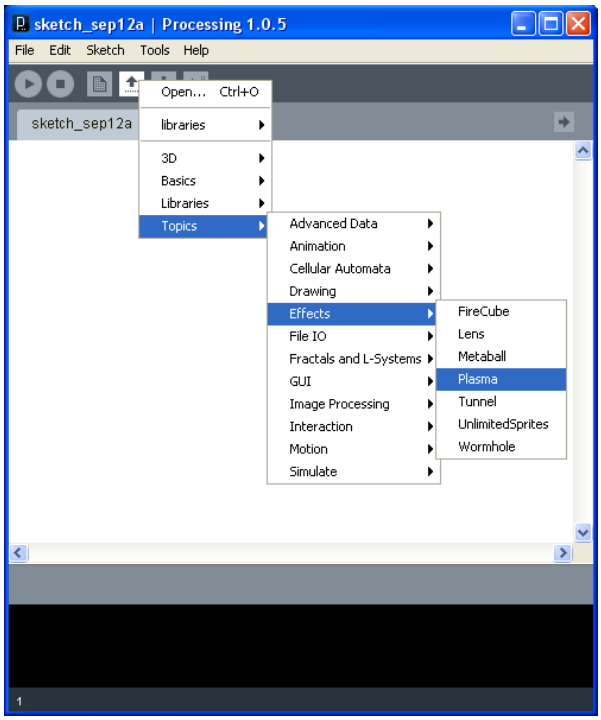

Figura 3: Carregando um exemplo através da barra de atalhos

Os exemplos do submenu 3D ilustram a utilização dos recursos de renderização 3D do Processing. É possível gerar imagens 3D através da biblioteca OpenGL ou também utilizando-se um sistema de renderização baseado em software. Embora este recurso tenha um impacto no desempenho, ele permite que o *sketch* possa ser utilizado mesmo em computadores sem placas de vídeo 3D. A Figura 4 contém imagens de dois destes exemplos: o primeiro (*3D → Form → Icosahedra*) ilustra a renderização de sólidos geométricos e o segundo (*3D → Image → Explosion*) aplica regiões de uma imagem como textura sobre um conjunto de pequenos quadriláteros que mudam de posição com o movimento do mouse.

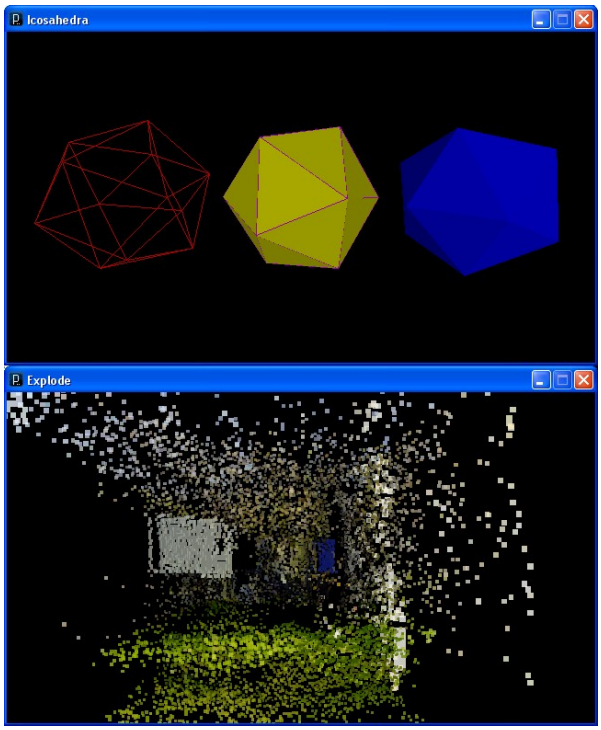

Figura 4: Exemplos 3D. Superior: Icosahedra; Inferior: Explosion.

O submenu *Basics* traz exemplos que ilustram o funcionamento de diversos comandos e recursos do Processing. Ainda assim, muitos deles geram resultados interessantes. Entre estes exemplos, destacam-se:

- *Basics → Form → BezierEllipse,* que demonstra a utilização de curvas para o desenho de formas elaboradas (duas delas mostradas na Figura 5);
- *Basics → Input → MouseSignals*, que permite a visualização de informações do mouse (movimento e clique de botões) na forma de gráficos interativos;
- *Basics → Math → Noise3D*, que ilustra a geração de uma textura procedural a partir de uma função de ruído;
- *Basics → Transform → Arm*, que permite a manipulação interativa de uma estrutura de esqueleto.

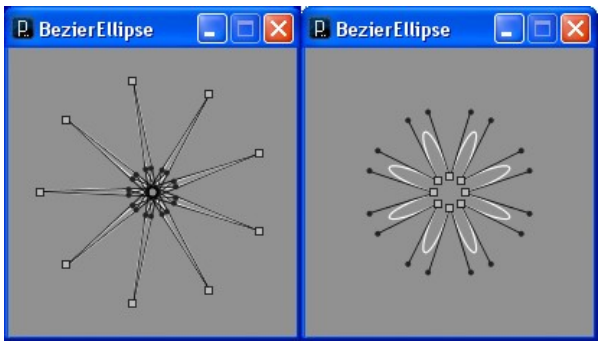

Figura 5: Formas geradas pelo sketch BezierEllipse

Os exemplos do submenu *Libraries* são utilizados para demonstrar a utilização de outras bibliotecas juntamente com o Processing. Elas adicionam recursos como reprodução e síntese de áudio, comunicação em rede, renderização de gráficos 3D através de OpenGL, captura de vídeo através de câmeras e reprodução de arquivos de vídeo. A seção 6 do tutorial é dedicada a discutir em mais detalhes o uso de bibliotecas.

O submenu *Topics* contém a lista de exemplos com temas mais variados. Em geral, são exemplos com código-fonte mais complexo e resultados mais elaborados. Os exemplos do submenu *Topics* → *Effects* ilustram alguns efeitos visuais que podem ser produzidos no Processing. A Figura 6 apresenta imagens de dois deles: *Fire Cube* e *Wormhole*, efeitos baseados na manipulação de pixels individuais das imagens.

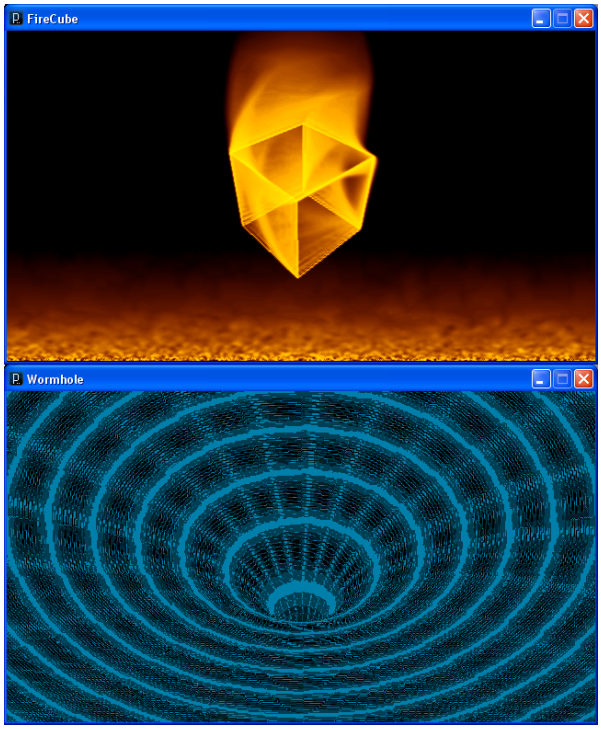

Figura 6: Exemplos de efeitos gráficos. Superior: Fire Cube; Inferior: Wormhole.

O Processing expõe uma interface simplificada para a realização deste tipo de operação, fazendo com que não seja necessário considerar detalhes de implementação como a localização da imagem na memória do computador (ou placa de vídeo) ou forma de representação das cores dos pixels.

O submenu *Topics* → *Interaction* contém exemplos de formas de interação com o usuário através do uso do mouse. Na Figura 7, a imagem da esquerda corresponde ao exemplo *Topics* → *Interaction* → *Follow3*, no qual um objeto composto de segmentos de reta segue a trajetória do mouse; a imagem da direita exibe um outro objeto segmentado, com uma extremidade fixa e outra que procura alcançar a posição do mouse (*Topics* → *Interaction* → *Reach2*).

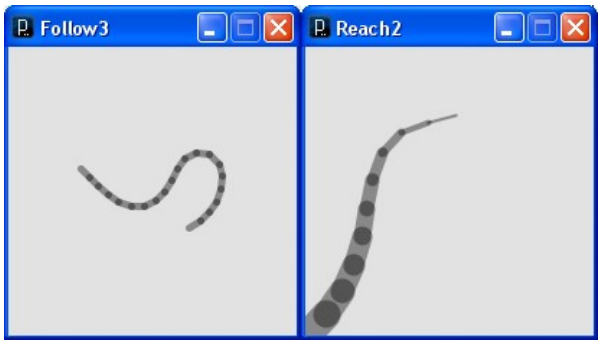

Figura 7: Exemplos de interação. Esquerda: Follow3; Direita: Reach2

Os submenus *Topics* → *Motion* e *Topics* → *Simulate* são dedicados a exemplos de movimentos, detecção e resposta a colisão e simulação física. Desta forma, são exemplos especialmente interessantes do ponto de vista do desenvolvimento de protótipos de jogos digitais. A Figura 8 apresenta imagens de dois exemplos de programação de movimentos com colisão. O exemplo da imagem superior (*Topics* → *Motion* → *BouncyBubbles*) demonstra a detecção de colisão entre múltiplos objetos. O exemplo da imagem inferior (*Topics* → *Motion* → *Reflection1*) demonstra a colisão contra uma parede com inclinação arbitrária.

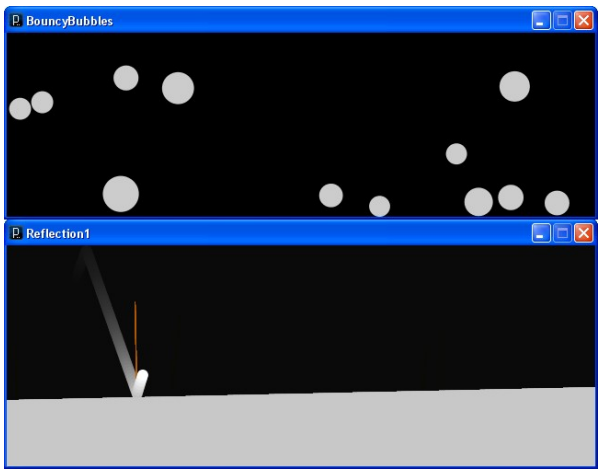

Figura 8: Exemplos de movimentação. Superior: BouncyBubbles; Inferior: Reflection1

O exemplo *Topics* → *Simulate* → *SimpleParticleSystem*, que pode ser visto na Figura 9, demonstra a implementação de um sistema de partículas em Processing. Este exemplo também ilustra de forma clara a utilização de classes e objetos para estruturar um *sketch*.

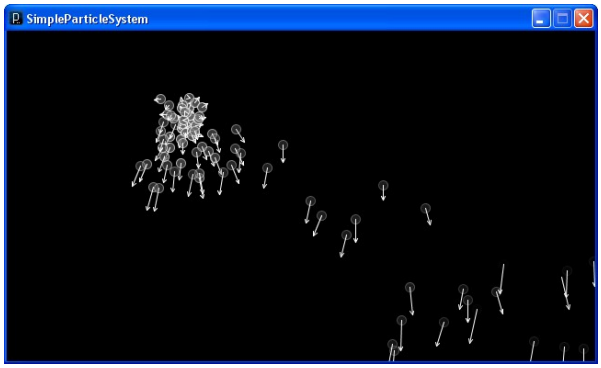

Figura 9: Exemplo que demonstra um sistema de partículas

Com isto, conclui-se a lista de exemplos de destaque, com a qual se espera ter apresentado um panorama das capacidades do Processing. No entanto, recomenda-se que algumas horas sejam dedicadas a explorar o restante dos exemplos que são distribuídos com o Processing, bem como outros *sketches* que podem ser encontrados no site dessa ferramenta e em outros sites da Internet.

3. Conceitos de programação

Esta seção faz uma revisão breve de fundamentos de programação, voltada para o Processing. Mesmo aos leitores já familiarizados com programação de computadores, recomenda-se ao menos a leitura desta seção para conhecer as particularidades da linguagem e observar os exemplos apresentados.

De forma simplificada, pode-se comparar um programa de computador a uma lista de comandos que são executados em ordem. Esses comandos, em geral, manipulam informações, que são representadas através de variáveis. Além disso, comandos especiais podem ser utilizados para alterar o fluxo sequencial de execução do programa, por meio de desvios condicionais ou repetições. O restante desta seção discute cada um desses conceitos.

3.1 Variáveis e tipos de dados

Como mencionado anteriormente, programas de computador podem ser vistos como sequências de instruções que modificam informações. Para que o computador tenha acesso a tais informações, é preciso que elas estejam disponíveis de alguma forma na sua memória. Ao escrever um programa, utilizam-se variáveis para representar essas áreas de memória.

No Processing, cada variável tem um nome, que pode ser composto de letras, números e do caractere "_". Além disso, cada variável tem um **tipo**, que indica qual o formato de informação que esta pode armazenar. Assim como na linguagem Java, os tipos distinguem-se entre primitivos (que armazenam somente um valor) e compostos, que armazenam um conjunto de informações. Os principais tipos primitivos utilizados no Processing são:

- **int**: representa um número inteiro positivo ou negativo;
- **float**: representa um número real, utilizando-se um formato de ponto flutuante.;
- **boolean**: representa um valor lógico verdadeiro (*true*) ou falso (*false*);
- **color**: representa uma cor, armazenada normalmente no formato RGB.

Variáveis de tipos primitivos são declaradas utilizando-se o formato "<tipo> <nome>;". Pode-se também declarar diversas variáveis do mesmo tipo separando seus nomes por vírgulas. Desta forma, os seguintes exemplos são declarações de variáveis válidas:

```
int idade;
float velocidade;
boolean ligado;
color vermelho;
int x, y, z;
```

Os principais tipos compostos existentes no Processing são:

- **String**: representa uma sequência de caracteres. Este tipo é utilizado para armazenar textos em geral;
- **PFont**: representa uma fonte de caracteres, utilizada para desenhar texto na tela;
- **PImage**: representa uma imagem matricial (bitmap). Pode ser utilizado tanto para criar imagens como para carregar arquivos de imagem em diversos formatos;
- **PVector**: representa um vetor geométrico de duas ou três dimensões. Este tipo é bastante utilizado na implementação de algoritmos de computação gráfica e simulação física.

Os tipos compostos são, efetivamente, objetos definidos na linguagem Java. Variáveis de tipos compostos são declarados da mesma forma que tipos primitivos mas precisam ser inicializados antes de serem utilizados. Isto normalmente é feito através do operador **new** conforme os exemplos a seguir.

```
String t;
t = new String("Texto");
String t2 = "Outro texto";
PVector v1;
v1 = new PVector();
PVector v2 = new PVector();
PVector v3 = new PVector(1, 2, 3);
```

Deve-se notar que, exclusivamente no caso de Strings, pode-se omitir o **new** e atribuir diretamente uma sequência de texto entre aspas. No caso dos tipos compostos PImage e PFont, não se utiliza a inicialização com **new**. Ao invés disso, utiliza-se os comandos createImage, loadImage, createFont e loadFont, que serão discutidos na seção 4.

O tipo PVector é composto de três valores float, que são identificados por x, y e z e correspondem aos componentes do vetor. Para se acessar estes componentes, utiliza-se um ponto entre o nome da variável e o nome do componente, conforme os exemplos a seguir:

```
PVector v;
v = new PVector();
v.x = 0;
v.y = 12;
v.z = v.x + v.y;
```

O Processing define automaticamente uma série de variáveis, de diferentes tipos, que contêm informações sobre o ambiente do programa. Dois exemplos são as variáveis **width** e **height**, que contêm, respectivamente, a largura e altura da janela gráfica do programa. Essas variáveis aparecem em uma cor cinza-azulado na área de edição.

Pode-se também declarar variáveis do tipo *Array*, que é um tipo composto correspondente a uma sequência de valores de outro tipo. Por exemplo, para armazenar uma sequência de dez valores inteiros, ao invés de declarar dez variáveis do tipo int, pode-se fazer:

```
int[] x = new int[10];
```

Neste caso, os valores são acessados através de um índice colocado entre colchetes. Por exemplo, para se somar os três primeiros valores da sequência, pode-se escrever:

```
int soma = x[0] + x[1] + x[2];
```

É possível declarar *arrays* de tipos primitivos ou compostos. Em qualquer caso, no entanto, todos os elementos do *array* só podem armazenar o mesmo tipo de dados. *Arrays* são frequentemente utilizados em conjunto com comandos de repetição, como será discutido na seção 3.3.

Exercício: A Listagem 1 demonstra o uso de variáveis no Processing. Crie um novo *sketch* e digite este pequeno programa. Experimente modificar os valores das variáveis e observar os efeitos.

Listagem 1: Exemplo de uso de variáveis

```
int diametro = 70;
int lado = 50;
color vermelho = color(255, 0, 0);
size(200, 200);
// desenha um quadrado
rect(10, 10, lado, lado);
// e um circulo
fill(vermelho);
ellipse(100, 100, diametro, diametro);
```

Por fim, deve-se observar que é possível construir expressões matemáticas e lógicas utilizando-se variáveis. Além dos operadores aritméticos básicos, o Processing conta com uma série de funções matemáticas para operações trigonométricas, exponenciação, raiz quadrada, logaritmos e outros. Os exemplos a seguir demonstram algumas expressões válidas:

```
int x, y, z;
x = 10; z = 15;
y = 4*x + z;
float k, w;
k = pow(3, 4) + sqrt(10);
w = sin(k/12);
PVector v = new PVector();
v.x = w/k;
```

3.2 Condicionais

Embora seja possível escrever programas que seguem sempre a mesma sequência de instruções, em muitos casos é necessário que este fluxo seja modificado de acordo com uma determinada condição lógica. Por exemplo, a rotina de alguém acordando às 7 da manhã poderia ser descrita como:

```
Se hoje é sábado ou domingo:
↳ Dormir até mais tarde
Caso contrário:
↳ Tomar banho
  Se estiver frio:
  ↳ Vestir um agasalho
  Tomar café
  Dirigir até o trabalho
```

No Processing, o controle de fluxo condicional é realizado através da estrutura **if...else**. O formato geral desta estrutura é mostrada a seguir, sendo que os blocos **else** são opcionais. As reticências indicam que mais condições (else if) podem ser acrescentadas.

```
if (expressão lógica) {
  <comandos>
}
else if (expressão lógica) {
  <comandos>
}
...
else {
}
```

O fluxo lógico do **if...else** é exatamente o mesmo aplicado às frases do exemplo mostrado anteriormente. Caso a expressão lógica seja verdadeira, o conjunto de comandos correspondente será executado. Caso contrário, se houver um "**else if**" a próxima expressão lógica será avaliada. Se nenhuma das expressões for verdadeira e houver um "**else**", os comandos

correspondentes serão executados. As chaves não são obrigatórias quando se desejar executar apenas um comando em resposta a uma condição; no entanto, seu uso consiste em uma boa prática de programação pois evita erros e facilita a leitura do código-fonte por outras pessoas (e não tem nenhum efeito sobre o desempenho do programa).

Uma expressão lógica é uma expressão que pode ter como resultado somente os valores verdadeiro (true) ou falso (false). Em geral, consiste em uma comparação entre uma variável e um valor ou entre duas variáveis. Alguns exemplos de expressões lógicas são mostrados na Tabela 1:

Tabela 1: Exemplos de expressões lógicas

`(x > y)`	x é maior do que y
`(a == b)`	a é igual a b
`(z >= 2 && z <= 5)`	z é maior ou igual a dois e menor ou igual a cinco
`(k == r || k < w)`	k é igual a r ou k é menor do que w
`(x + 4 < s/3)`	x+4 é menor do que s/3

Deve-se notar que o símbolo para indicar uma **comparação de igualdade** entre dois valores é formado por dois sinais de igualdade ou "==". Adicionalmente, o símbolo "||" é utilizado para representar o operador OU lógico (a expressão resultante será verdadeira se pelo menos uma das duas expressões for verdadeira). O símbolo "&&" é utilizado para representar o operador E lógico (a expressão resultante será verdadeira somente se as duas expressões conectadas forem verdadeiras).

A Listagem 2 apresenta um exemplo de programa que pinta o fundo da janela de cores diferentes de manhã, à tarde ou à noite.

Listagem 2: Fundo que muda de cor com as horas

```
color fundo;

if (hour() < 12) {
  fundo = color(224, 248, 255);
}
else if (hour() < 19) {
  fundo = color(148, 153, 240);
}
else {
  fundo = color(48, 48, 120);
}

background(fundo);
```

Quando é preciso testar a igualdade de uma variável contra uma tabela de vários valores, existe uma alternativa ao encadeamento de vários "else if", que consiste na estrutura **switch/case**. O formato desta estrutura é mostrado a seguir:

```
switch(expressão) {
   case <valor>:
      <comandos>
      [break];
   ...
   default:
      <comandos>
}
```

A expressão utilizada no **switch** deve resultar em um valor inteiro. Ela pode ser uma variável ou uma expressão calculada com muitas variáveis. Dentro do bloco entre chaves do **switch**, podem existir um ou mais blocos **case** e, opcionalmente, um bloco **default**. O valor da expressão é testado contra os valores de cada case e, quando forem iguais, os comandos passam a ser executados. No entanto, ao contrário dos "else if" encadeados, todos os comandos até o final do switch serão executados, a menos que exista um comando **break**. O bloco **default** é utilizado para indicar comandos que devem ser executados quando nenhum dos valores testados for igual ao valor da expressão do switch. A Listagem 3 ilustra o uso da estrutura switch/case.

Exercício: Experimente modificar o valor da variável v e observar o resultado. Note que, como não há comandos break nos casos 1 e 2, quando o valor de v for igual a estes valores, a cor final da janela será azul.

Listagem 3: exemplo de uso de switch/case

```
int v = 2;
color c;

switch(v) {
  case 1:
    c = color(255, 0, 0);// vermelho
  case 2:
    c = color(0, 255, 0);// verde
  case 3:
    c = color(0, 0, 255);// azul
    break;
  case 4:
    c = color(255, 255, 255);// branco
    break;
  default:
    c = color(128, 128, 128);// cinza
}

background(c);
```

3.3 Repetição

Além da execução de código dependente de uma condição lógica, outra situação frequente no desenvolvimento de programas de computador é a necessidade de se repetir um conjunto de operações sobre vários dados diferentes, ou até que uma condição seja satisfeita. Para atender estas necessidades, existem as estruturas de repetição.

A estrutura de repetição mais simples é definida pela instrução **while**. Esta estrutura repete uma sequência de comandos enquanto uma condição lógica for verdadeira. A estrutura do while é apresentada a seguir:

```
while (condição) {
   <comandos>
}
```

A condição do while é uma expressão lógica, do mesmo tipo e forma das utilizadas em estruturas **if...else**. É importante observar que, se a expressão tiver valor falso inicialmente, os comandos contidos dentro do bloco do **while** nunca serão executados. Por outro lado, se a condição permanecer verdadeira para sempre, o bloco continuará em execução infinitamente – o que normalmente é indesejável. Desta forma, é preciso ter atenção ao definir a condição lógica do while. A Listagem 4 ilustra um exemplo de utilização da estrutura de repetição while para desenhar um conjunto de formas geométricas. O resultado é apresentado na Figura 10.

Listagem 4: Desenhando formas com while

```
size(400, 200);
int x = 20;
while (x < 400) {
  ellipse(x, 100, 40, 200);
  x = x + 40;
}
```

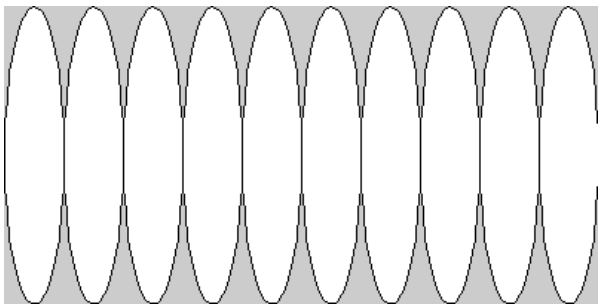

Figura 10: Resultado da execução da Listagem 4

A estrutura while também pode ser utilizada para processar os elementos de um *array*. Por exemplo, supondo que N é um *array* de números inteiros positivos, o código a seguir encontra o maior entre estes números. Observe o uso da expressão "N.length" para obter o número de elementos do *array*. Isso pode ser feito com qualquer *array*.

```
int p = 0;
int maior = -1;
while (p < N.length) {
  if (N[p] > maior) {
    maior = N[p];
  }
  p = p + 1;
}
```

Outra estrutura de repetição existente no Processing é o **for,** que tem a forma apresentada a seguir. Os colchetes indicam que os três componentes do **for** são opcionais.

```
for ([início]; [condição]; [passo]) {
   <comandos>
}
```

A estrutura **for** tem um funcionamento um pouco diferente. Primeiramente, o comando na posição **[início]** é executado. Em seguida, inicia-se um ciclo de repetições: a condição lógica é verificada e, enquanto esta for verdadeira, os comandos dentro do bloco do **for** são executados, seguidos do comando na posição **passo**. Pode-se dizer que as estruturas **while** e **for** são equivalentes, quanto à capacidade de expressão em um programa. No entanto, o formato do **for** permite expressar de maneira mais clara alguns tipos de construções. Por exemplo, o código abaixo implementa a mesma lógica de buscar o maior número inteiro positivo em um array, mostrado anteriormente utilizando-se uma estrutura **while**.

```
int maior= -1;
for(int p= 0; p < N.length; p= p + 1) {
  if (N[p] > maior) {
    maior = N[p];
  }
}
```

De fato, como os componentes do **for** são opcionais, esta estrutura pode ser utilizada para se comportar exatamente como o **while.** No entanto, este tipo de uso não é recomendado por tornar o código-fonte mais obscuro, sem nenhuma vantagem.

```
int p = 0;
int maior = -1;
for (;p < N.length;) {
  if (N[p] > maior) {
    maior = N[p];
  }
  p = p + 1;
}
```

Listagem 5: Desenhando vetores aleatórios

```
size(300, 300);
background(255);
stroke(0);

PVector[] vec;
vec = new PVector[100];

for(int i=0; i<vec.length; i++) {
  vec[i] = new PVector();
  vec[i].x = random(50, 250);
  vec[i].y = random(50, 250);
}

for(int i=0; i<vec.length; i++) {
  line(150, 150,
       vec[i].x, vec[i].y);
}
```

Por fim, a Listagem 5 ilustra o uso de estruturas de repetição juntamente com um *array* de tipo composto

(no caso, PVector). Deve-se notar que, além da inicialização do *array* (na linha 6), cada elemento do *array* deve ser inicializado, por ser um tipo composto.

3.4 Funções

Até o momento, todos os exemplos mostrados são relativamente pequenos. No entanto, para fins de organização e reutilização de código, pode-se organizar um programa em funções. Uma função pode ser definida como um bloco de código, podendo receber parâmetros, que realiza determinadas operações e, opcionalmente, pode retornar um valor como resultado. A forma da declaração de uma função é mostrada a seguir:

```
<tipo> <nome>(<parâmetros>) {
   <comandos>
   [return [valor];]
}
```

O **tipo** da função corresponde ao tipo de valor que é retornado por ela. Quando uma função não retorna nenhum resultado, utiliza-se o tipo **void**. O nome da função segue as mesmas convenções de nomes de variáveis. Os parâmetros são apresentados na forma de uma lista de declarações de variáveis, separadas por vírgulas. O comando **return**, se presente dentro do bloco da função, termina a execução da função e retorna o valor indicado. O exemplo a seguir declara uma função para o cálculo da distância entre dois pontos:

```
float distancia(float x1, float y1,
                float x2, float y2) {
  float deltaX = x1 - x2;
  float deltaY = y1 - y2;
  float dist = sqrt(deltaX*deltaX +
                    deltaY*deltaY);
  return dist;
}
```

Uma observação importante é o fato de que funções só podem ser utilizadas em *sketches* contínuos (discutidos na seção 4.2). Para se utilizar uma função, basta escrever seu nome, seguido dos parâmetros necessários. Por exemplo, o código a seguir atribui à variável d o valor retornado pela função:

```
float d = distancia(3, 2, 5, 0);
```

Variáveis declaradas dentro do bloco de uma função só são válidas naquele bloco. Assim, referências às variáveis deltaX, deltaY e dist fora do bloco da função **distancia**, acima, causariam erros. Os parâmetros da função também se comportam como variáveis declaradas dentro do bloco da função.

Quando uma variável de tipo primitivo é passada como parâmetro de uma função, seu valor é copiado e a variável original não é modificada. Para variáveis de tipos compostos, no entanto, isso não ocorre (a não ser no caso de Strings). Assim, se uma variável do tipo PVector for passada como parâmetro de uma função, por exemplo, e o valor de seus componentes forem alterados nessa função, o PVector original será modificado.

Uma outra maneira de se organizar *sketches* complexos é através da adição de abas. Pode-se, por exemplo, criar uma aba para funções gráficas, outra para funções de simulação e uma terceira para funções de áudio. Todas as funções são utilizadas, posteriormente, na aba principal do *sketch*.

3.5 Classes e Objetos

Dado que o Processing é baseado na linguagem Java, é possível também criar novas classes e objetos em um *sketch*. Estas classes também podem ser criadas em abas separadas para auxiliar na organização do código-fonte. Diferentemente de funções, classes podem ser declaradas e utilizadas também em *sketches* básicos.

Uma discussão detalhada de programação orientada a objetos vai além do escopo deste tutorial. No entanto, para efeito de referência, a forma como uma classe é declarada é apresentada a seguir:

```
class <nome> {
   <tipo> <nome>;
   ...
   <tipo> <nome>([parâmetros]) {
      <comandos>
   }
   ...
}
```

Podem ser declaradas variáveis que têm como tipo uma classe – efetivamente, funcionando como um tipo composto. Quando inicializadas, estas variáveis passam a conter objetos daquela classe. Eckel [2009] faz uma analogia de um objeto com uma "variável sofisticada" que, além de armazenar informações, também recebe "pedidos" para executar operações sobre essas informações. A Listagem 6 apresenta um exemplo simples de utilização de classes em um *sketch* e o resultado é mostrado na Figura 11.

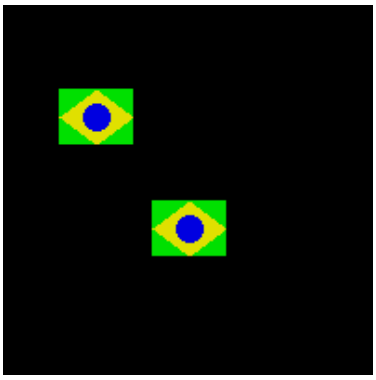

Figura 11: Resultado da execução do exemplo de uso de classes no Processing

Listagem 6: Exemplo de declaração e uso de classes

```
class Bandeira {
  int x, y;
  Bandeira(int px, int py) {
    x = px;
    y = py;
  }
  void desenhar() {
    noStroke();
    fill(0, 224, 0);
    rect(x-20, y-15, 40, 30);
    fill(224, 224, 0);
    quad(x-20, y, x, y-15,
         x+20, y, x, y+15);
    fill(0, 0, 224);
    ellipse(x, y, 15, 15);
  }
}

size(200, 200);
background(0);
Bandeira b1 = new Bandeira(50, 60);
Bandeira b2 = new Bandeira(100, 120);
b1.desenhar();
b2.desenhar();
```

Nesta revisão dos conceitos de programação, foram apresentados brevemente os principais tipos de dados utilizados no Processing, as estruturas de controle de fluxo do programa, bem como funções e classes. Para uma visão mais detalhada destes tópicos, recomenda-se a leitura de Eckel [2009], Shiffman [2008] ou outros livros de fundamentos de programação e orientação a objetos.

4. Os modelos de programação dos *sketches*

Antes de se discutir o uso do Processing para o desenvolvimento de protótipos, é preciso entender como são estruturados os programas desenvolvidos nesse ambiente. Pode-se distinguir dois tipos de *sketches* criados no Processing: básicos ou não-interativos e contínuos ou interativos. *Sketches* básicos possuem uma estrutura comparativamente mais simples e, embora sejam menos úteis do ponto de vista da criação de protótipos de *gameplay*, podem ajudar significativamente no aprendizado do Processing. *Sketches* contínuos precisam seguir uma estrutura pré-definida, mas permitem a criação de animações e outros comportamentos dinâmicos.

4.1 *Sketches* básicos

Um *sketch* básico consiste, essencialmente, em um programa formado por variáveis, comandos e estruturas de controle de fluxo, que são executados sequencialmente. Desta forma, os exemplos apresentados até o momento (nas Listagens 1 a 6) são, de fato, *sketches* básicos.

Como pode ter sido observado, a partir dos exemplos citados, *sketches* básicos não permitem interação durante a sua execução. O resultado de um *sketch* básico é, em geral, uma imagem. Também podem ser gerados arquivos e saídas de texto. A imagem gerada por um *sketch* básico pode ser gravada através do comando **save**.

A Listagem 7 apresenta um sketch que gera uma textura com componentes aleatórios e grava o resultado em um arquivo chamado "texture.png". A Figura 12 apresenta uma imagem gerada por este *sketch*.

Listagem 7: Geração e gravação de uma imagem

```
size(256, 256);
background(0);
color c1 = color(160, 160, 230);
color c2 = color(80, 80, 200);
color c3 = color(30, 30, 110);

for (int x=0; x<width; x++) {
  int h1 = (int)random(height/5,
3*height/8);
  int h2 = (int)random(5*height/8,
4*height/5);
  stroke(c1);
  line(x, 0, x, h1);
  stroke(c2);
  line(x, h1+1, x, h2);
  stroke(c3);
  line(x, h2+1, x, height);
}

save("texture.png");
```

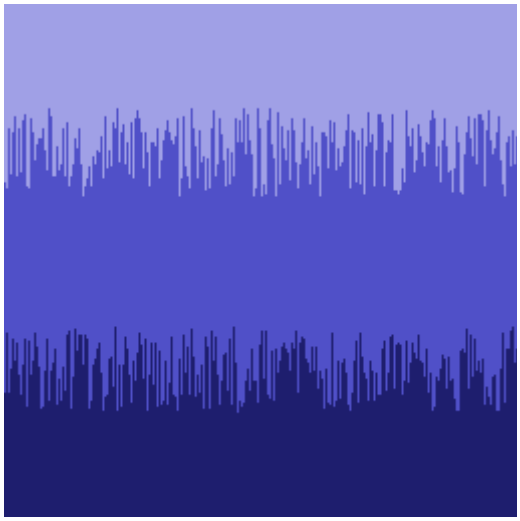

Figura 12: Imagem gerada e gravada por um sketch básico

A Listagem 8 apresenta outro exemplo de geração procedural de textura (mostrada na Figura 13). Neste caso, é gerada uma imagem em tons de cinza que poderia ser utilizada como máscara de transparência para texturas de partículas.

Em resumo, *sketches* básicos apresentam uma estrutura simples mas podem gerar resultados interessantes, ao se utilizarem dos comandos existentes no Processing. No entanto, para aplicações interativas é preciso utilizar o modo contínuo, explicado na próxima seção.

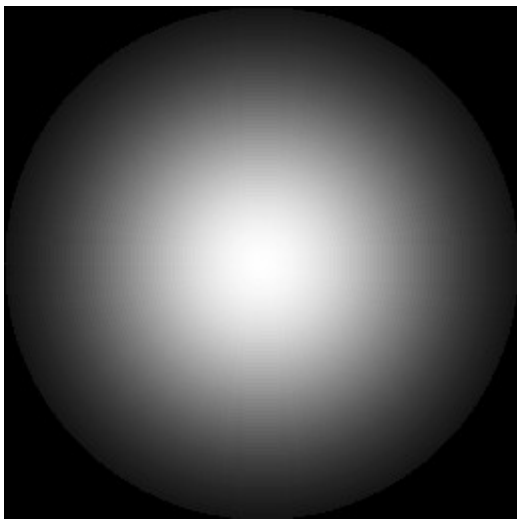

Figura 13: Imagem gerada pelo sketch da Listagem 8

Listagem 8: Gerador de textura de partículas

```
size(256, 256);
background(0);
noStroke();
int power = 4;

for (int d=256; d>0; d=d-4) {
  int v = (int)
(255.0*pow(cos(d/256.0),power));
  fill(v);
  ellipse(width/2, height/2, d, d);
}
```

4.2 *Sketches* contínuos

Sketches contínuos recebem este nome porque, ao contrário dos *sketches* básicos, são executados continuamente pelo Processing. Para operar em modo contínuo, um *sketch* precisa ter uma função definida da seguinte forma:

```
void draw() {
  <código>
}
```

A existência desta função faz com que o Processing mude a forma de interpretar o *sketch*. Ao invés de simplesmente executar todo o código uma única vez, quando o *sketch* é iniciado, a função **draw** será executada continuamente até que a janela gráfica do *sketch* seja fechada. Desta forma, se o código da função **draw** tiver comandos que variam a cada chamada, podem ser realizadas animações. Por exemplo, a Listagem 9 mostra um sketch que faz com que retângulos sejam desenhados em posições aleatórias da janela.

Listagem 9: Sketch contínuo que desenha retângulos aleatórios na tela

```
void draw() {
  background(0);
  rect(random(width), random(height),
  20, 20);
}
```

Deve-se observar que, uma vez que a função draw é definida, não é possível escrever mais código "livre" no sketch – isto é, fora de blocos de funções ou classes. Por outro lado, pode ser definida uma função **setup** que é executada automaticamente – e uma única vez – quando a execução do sketch se inicia. Assim, esta função pode ser utilizada para configurar o ambiente, inicializar variáveis etc.

A Listagem 10 contém um exemplo de sketch contínuo com as funções **setup** e **draw**. As variáveis utilizadas para mover um quadrado para a esquerda e para a direita na janela são inicializadas na função **setup** e o código de animação é executado na função **draw**.

Listagem 10: Caixa que se move horizontalmente

```
int x, vx;

void setup() {
  size(300, 150);
  x = 0; vx = 2;
}

void draw() {
  background(0);
  rect(x, 0, 50, 50);
  x = x + vx;
  if (x > (width-50) || x < 0) {
    vx = -vx;
  }
}
```

Normalmente o Processing procura executar os *sketches* contínuos a uma taxa de 60 quadros por segundo. Esta taxa pode ser configurada através do comando **frameRate** na função **setup**. Por exemplo, o comando **frameRate(10)** faz com que a função draw seja executada dez vezes por segundo. Além disso, é possível interromper ou retomar o processo de atualização do *sketch* através dos comandos **noLoop** e **loop**.

Tabela 2: Ações do usuário e funções correspondentes

Ação	Função
tecla pressionada	keyPressed
tecla liberada	keyReleased
tecla digitada (pressionada e liberada em seguida)	keyTyped
botão do mouse pressionado	mousePressed
botão do mouse liberado	mouseReleased
clique do mouse (botão pressionado e liberado em seguida)	mouseClicked
mouse arrastado	mouseMoved
mouse arrastado com botão pressionado	mouseDragged

É possível tratar a interação do usuário através do teclado e mouse definindo-se uma série de funções, que são executadas automaticamente em resposta a uma ação específica, conforme a Tabela 2.

Adicionalmente, as variáveis **mouseX**, **mouseY**, **mouseButton**, **keyPressed**, **key** e **keyCode** são definidas automaticamente pelo Processing e podem ser acessadas para se obter informações sobre os dispositivos de entrada.

Combinando-se as funções de tratamento de ações do usuário com os comandos **noLoop** (que interrompe a repetição do **draw**) e **loop** (que retoma a repetição), é possível criar *sketches* interativos que são atualizados somente em resposta a um comando do usuário. Nem sempre esta abordagem é viável, mas o resultado é uma economia no consumo de processamento. O *sketch* da Listagem 11 ilustra esta técnica.

Listagem 11: Sketch que se atualiza somente quando o usuário clica o mouse na janela

```
void setup() {
  size(300, 300);
  background(0);
  noLoop();
}

void draw() {
  background(0);
  stroke(255);
  for (int i = 0; i<width; i++) {
    line(mouseX, mouseY, i, 0);
    line(mouseX, mouseY, i, height);
  }
  noLoop();
}

void mouseClicked() {
  loop();
}
```

A Listagem 12 ilustra o controle de um objeto gráfico pelo pressionamento das setas cursoras do teclado.

Listagem 12: Controlando um objeto pelo teclado

```
int x, y;

void setup() {
  size(400, 400);
  x = width/2;
  y = height/2;
}

void draw() {
  background(0);
  fill(255);
  triangle(x-10, y+10, x+10, y+10,
           x, y-10);
}

void keyPressed() {
  if (key == CODED) {
    switch(keyCode) {
      case UP:
        y = y - 5;
        break;
      case DOWN:
        y = y + 5;
        break;
      case LEFT:
        x = x - 5;
        break;
      case RIGHT:
        x = x + 5;
    }
  }
}
```

Com isso, se encerra a explicação geral sobre a estrutura de *sketches* contínuos para a criação de programas interativos no Processing. Deve-se lembrar que, além das funções pré-definidas (setup, draw e funções de resposta a ações do usuário), podem ser criadas outras funções e classes para uso no *sketch* contínuo.

4.3 O modelo gráfico do Processing

Até este momento, vários exemplos de uso do Processing foram apresentados, empregando alguns comandos gráficos da linguagem sem maiores detalhes – esses detalhes e explicações são o assunto desta seção. De um modo geral, o Processing, tanto na geração de imagens 2D como 3D, utiliza um modelo semelhante ao do OpenGL: existe um conjunto de variáveis de estado que são configuradas para definir os atributos que serão aplicados nas primitivas gráficas (pontos, linhas, curvas, polígonos e texto) que forem desenhadas em seguida. A lista completa de atributos pode ser encontrada na referência do Processing, mas a Tabela 3 traz um resumo de alguns dos comandos mais utilizados.

Tabela 3: Atributos gráficos e comandos correspondentes

Comando	Atributo
stroke	define a cor do traçado de linhas e contornos
noStroke	desativa o traçado de linhas e contornos
fill	define a cor de preenchimento de áreas e texto
noFill	desativa o preenchimento de áreas
smooth	ativa a suavização (*anti-aliasing*) no desenho de linhas e contornos
noSmooth	desativa a suavização
strokeWeight	define a espessura de linhas e contornos

Os comandos pushStyle e popStyle podem ser utilizados para salvar e restaurar os atributos de

desenho. A utilização destes comandos é de grande valia em *sketches* que desenham muitas formas com atributos diferentes.

O sistema de coordenadas do Processing é ilustrado na Figura 14. No caso de imagens 2D, a origem do sistema de coordenadas é o canto superior esquerdo da janela. No caso de imagens 3D, é o centro da janela.

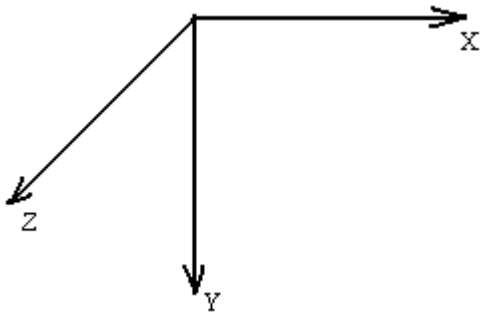

Figura 14: Sistema de coordenadas do Processing

O Processing suporta o uso de imagens carregadas de arquivos e fontes para escrita de texto, mas nos dois casos é preciso preparação. No caso de imagens, deve-se utilizar o menu *Sketch* → *Add File* para que a imagem seja copiada para um subdiretório "data" no diretório onde o *sketch* está gravado. Pode-se fazer a cópia manualmente também. Uma vez que a imagem foi copiada, ela pode ser acessada através do comando **loadImage**. Os formatos de arquivo suportados são GIF, TGA, PNG e JPG. O comando **image** pode ser usado para desenhar a imagem na tela, inclusive com mudança de escala, conforme ilustrado pelo sketch da Listagem 13.

Listagem 13: Carregando e desenhando imagens

```
PImage imagem;

void setup() {
  size(640, 400);
  // assumindo que o arquivo
  // imagem.jpg ja foi adicionado
  // ao sketch.
  imagem = loadImage("imagem.jpg");
}

void draw() {
  image(imagem, 0, 0);
  image(imagem, width/2, height/2,
width/2, height/2);
}
```

No caso de fontes para texto, é preciso utilizar o menu *Tools* → *Create Font*. Nesta ferramenta, exibida na Figura 15, primeiramente deve-se selecionar a fonte a ser utilizada e, em seguida, o tamanho do corpo. A caixa de opção "*Smooth*" gera imagens suavizadas das letras, o que é geralmente desejável. A caixa de opção "*All characters*" faz com que todos os caracteres da fonte sejam exportados, o que pode aumentar significativamente o tamanho do arquivo. Depois, basta digitar o nome da fonte e clicar no botão **OK**. Assim como no caso das imagens, um arquivo será gravado no subdiretório **data** do *sketch*. Este arquivo contém informações da fonte no formato selecionado, que podem ser utilizadas mesmo em computadores que não tenham a fonte original instalada.

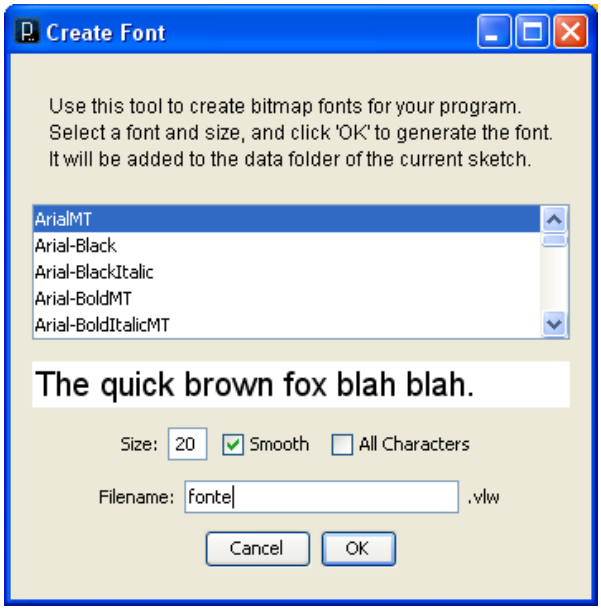

Figura 15: A ferramenta de criação de arquivos de fontes do Processing

Para se utilizar uma fonte criada com o procedimento acima no *sketch*, deve-se utilizar o comando **loadFont** para carregar o arquivo (com o nome que foi gravado anteriormente) e, em seguida, o comando **textFont** para selecionar a fonte carregada para uso com os comandos de desenho de texto. A Listagem 14 e a Figura 16 ilustram o uso de texto em um *sketch*, bem como a utilização de comandos de transformações geométricas para posicionar primitivas gráficas (inclusive texto).

Figura 16: Sketch com desenho e animação de texto

Listagem 14: Carregando e desenhando texto

```
String texto =
"ABCDEFGHIJKLMNOPQRSTUVWXYZ";
float angle, step, colorStep;

void setup() {
  size(400, 400);
  textAlign(CENTER);
  PFont font = loadFont("fonte.vlw");
  textFont(font);
  angle = 0;
  step = 2*PI/(texto.length()+1);
  colorStep = 255/texto.length();
  frameRate(10);
}

void draw() {
  background(0);
  for (int i=0; i<texto.length(); i++) {
    fill(255-colorStep*i, 0,
         colorStep*i);
    resetMatrix();
    translate(width/2, height/2);
    rotate(angle + i*step);
    translate(width/3, 0);
    text(texto.substring(i, i+1), 0, 0);
  }
  angle += 0.4*((float)mouseX/width);
}
```

Quando necessário, é possível utilizar o comando **textSize** para modificar o tamanho da fonte que será desenhada. No entanto, reduções e principalmente ampliações em relação ao tamanho que a fonte foi exportada resultam em perdas de qualidade no desenho do texto.

O desenho de formas no espaço 3D é análogo ao desenho de formas 2D. No entanto, é preciso especificar o parâmetro do renderizador 3D a ser utilizado, no comando **size**. Atualmente, as duas opções são P3D (um renderizador baseado em software) e OPENGL. No caso da opção OPENGL, é preciso adicionar a biblioteca apropriada ao *sketch*, utilizando-se o menu *Sketch → Import Library → OpenGL*. A forma mais simples de se desenhar os objetos é através dos comandos beginShape, endShape e vertex. A Listagem 15 apresenta um exemplo de visualização de uma forma procedural 3D e a Figura 17 contém uma imagem do *sketch* em execução. Os ângulos de rotação são determinados com base na posição do mouse dentro da janela.

Listagem 15: Gráficos em 3D

```
void setup() {
  size(400, 300, P3D);
  fill(255);
  noStroke();
}

void draw() {
  lights();
  background(0);
  resetMatrix();
  translate(0, 0, -20);
  rotateX(lerp(-PI/4, PI/4,
          mouseY/(float)height));
  rotateY(lerp(-PI/4, PI/4,
          mouseX/(float)width));
  fill(192, 192, 192);
  beginShape(QUAD_STRIP);
  for (float i = -5; i <= 5; i=i+0.5) {
    float z = sin(4*PI*(5-i)/10.0);
    vertex(i, -3, z);
    vertex(i,  3, z);
  }
  endShape();
}
```

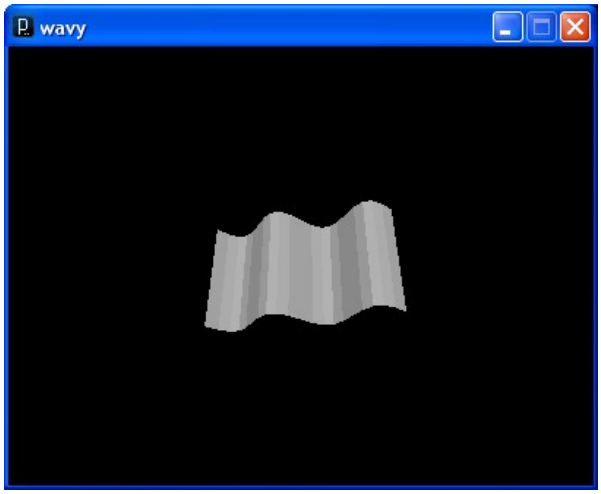

Figura 17: Desenhando gráficos 3D no Processing

Outros comandos úteis para desenhos 3D incluem **texture**, usado para selecionar uma imagem para ser aplicada como textura sobre os objetos e **normal**, usado para especificar manualmente os vetores normais das faces do objeto.

Tendo-se concluído a apresentação dos modelos de programação e de renderização do Processing, pode-se iniciar a discussão sobre o uso destes recursos para *game design*.

5. Utilizando Processing para *Game Design*

A partir do que foi exposto sobre as capacidades e modelos de programação do Processing, pode-se visualizar seu potencial como ferramenta para auxiliar em atividades de *game design*.

Provavelmente a forma mais imediata de utilização dessa ferramenta é na criação de protótipos para avaliação de propostas de *gameplay*. No entanto, propõe-se que o Processing também pode ser eficaz para a realização de testes de técnicas de interação, inclusive com dispositivos não convencionais; para a geração de conteúdo procedural (para testes ou mesmo como ferramenta de produção) e para o registro de informações. O restante desta seção aborda cada um destes usos, partindo de uma discussão sobre boas práticas de desenvolvimento de *sketches*.

5.1 Boas práticas na criação de protótipos de jogos digitais com Processing

Tendo como base a experiência na utilização do Processing para a criação de protótipos com diversas finalidades, somada ao conhecimento de boas práticas de desenvolvimento de software, são apresentadas aqui algumas recomendações para a criação de *sketches*. Estas práticas têm como objetivo simplificar o processo de desenvolvimento e facilitar a compreensão e reutilização do código-fonte.

Complexidade incremental: ao desenvolver um protótipo, recomenda-se iniciar com uma proposta simples e adicionar mais elementos progressivamente. Por exemplo, pode-se desenvolver primeiramente o controle da movimentação de uma forma geométrica pelo mouse. Quando este *sketch* estiver operacional, pode-se substituir a forma geométrica por uma imagem animada e, em seguida, acrescentar efeitos para formar o rastro da trajetória da imagem.

Mesmo no caso de se desenvolver um protótipo completo de um *game design* já definido, é recomendável seguir a mesma abordagem, dividindo-se aquele *design* em componentes mais simples que podem ser acrescentados ao *sketch* aos poucos. A motivação desta prática é a possibilidade de se limitar a complexidade do programa a ser desenvolvido, a partir da sua divisão em problemas menores.

Organização do código: embora seja possível concentrar todo o código de um *sketch* nas funções **setup** e **draw** (ou em um único bloco de programa, nos *sketches* básicos), com o aumento da complexidade isso torna o código cada vez mais difícil de entender e mais sujeito a erros introduzidos durante a programação. Por este motivo, recomenda-se a organização do programa em funções ou classes (ou ambos).

Esta divisão do programa pode ser baseada em responsabilidades ou objetivos: cada função ou classe deve ter um propósito bem definido e só realizar operações relacionadas a isto. Idealmente, estas operações também não devem ter efeitos colaterais (por exemplo, uma função que desenha um objeto na tela não deveria alterar a cor de preenchimento de outros objetos) – no entanto, isso pode ser difícil de se evitar. Esta estruturação do programa auxilia na sua compreensão e também ajuda a aplicação da abordagem de complexidade incremental.

Outra forma de se aumentar a legibilidade do código consiste na utilização de comentários. No Processing, comentários podem ter as duas formas apresentadas a seguir. Texto dentro de um comentário é ignorado e não tem impacto na execução do *sketch*, mas pode ser de grande valia para outros indivíduos compreenderem o programa.

```
// comentário de uma linha
/* comentário de múltiplas linhas...
   só termina com estes símbolos: */
```

Por fim, cabe notar que o Processing conta também com o recurso das abas para ajudar a organizar o código. Um *sketch* pode ser dividido em múltiplas abas que podem conter variáveis, funções e classes. Desta forma, pode-se criar uma aba para o código de visualização, outro para inicialização etc. A organização em abas auxilia na localização de partes específicas de código, tendo-se em vista que o ambiente do Processing não conta com recursos de navegação de código de outros ambientes de programação mais complexos.

Separação entre simulação e visualização: esta prática pode ser vista como um caso particular de organização de código, que merece atenção especial por ocorrer frequentemente em jogos digitais.

Durante a programação de entidades de um jogo – por exemplo, um personagem autônomo que se move pelo campo de jogo – é comum que haja confusão entre o código que define seu comportamento (simulação) e sua aparência (visualização). No Processing, há uma tendência maior para que isso ocorra pois todo tipo de atualização contínua ocorre a partir da função **draw**. No entanto, esta mistura leva a problemas como a mudança do comportamento do jogo dependendo da ordem em que os personagens são desenhados ou a necessidade de se modificar o código de simulação quando o tamanho da tela é modificado.

No caso mais simples, a separação entre simulação e visualização pode ser realizada com uma estrutura semelhante à da Listagem 16. De um modo geral, as variáveis que definem os atributos das entidades do jogo devem ser modificadas na função **simular** e somente lidas na função **visualizar**.

Listagem 16: Exemplo de separação de código entre simulação e visualização

```
// posicao do personagem
int x, y;
// tamanho do personagem
// (aproximado por um circulo)
int d;

void setup() {
  size(400, 400);
  x = width/2;
  y = height/2;
  d = 20;
}

void draw() {
  simular();
  visualizar();
}

void simular() {
  // modificação das variáveis x e y
  // de acordo com o comportamento
```

```
  // desejado
}

void visualizar() {
  background(0);
  fill(255);
  ellipse(x, y, d, d);
}
```

Embora, nos *sketches* mais simples, a estrutura apresentada na Listagem 16 possa parecer desnecessária, ela segue os princípios das outras práticas discutidas anteriormente e colabora para que o *sketch* possa evoluir de forma controlada.

5.2 *Sketches* para protótipos de *gameplay*

Pode-se utilizar os *sketches* do Processing para construir protótipos de partes específicas de um *game design*, como um sistema de movimentação, a interação entre certos elementos do jogo ou mesmo um efeito gráfico. A estes protótipos, será atribuída a denominação de protótipo de *gameplay*.

A vantagem de se desenvolver protótipos com um enfoque restrito está na possibilidade de se produzir *sketches* relativamente mais simples e, consequentemente, com processo de desenvolvimento mais rápido. Os detalhes de implementação dependem do conceito ou regras de jogo que se pretende testar, mas existem algumas funcionalidades que se ocorrem com frequência. *Sketches* de exemplo para estas funcionalidades são apresentados a seguir, podendo ser utilizados como referência para novos protótipos.

Utilizando-se arrays para representar múltiplas entidades: embora seja possível utilizar estruturas de dados mais sofisticadas no Processing, muitas vezes um simples array pode ser suficiente para o protótipo. A Listagem 17 mostra um sketch onde vários elementos se movem com velocidades aleatórias. As propriedades destas entidades são armazenadas em vários arrays.

Listagem 17: Utilização de arrays para múltiplas entidades

```
// arrays de posicoes
int px[], py[];
// array de velocidades
int vx[], vy[];
// numero de entidades
int num;

void setup() {
  size(400, 400);
  num = 20;
  px = new int[num]; py = new int[num];
  vx = new int[num]; vy = new int[num];
  criarEntidades();
}

void draw() {
  simular();
  visualizar();
}

void criarEntidades() {
  for(int i=0; i<num; i++) {
    px[i] = (int)random(width);
    vx[i] = (int)random(-4, 4);
    py[i] = (int)random(height);
    vy[i] = (int)random(-4, 4);
  }
}

void simular() {
  for(int i=0; i<num; i++) {
    px[i] = px[i] + vx[i];
    if (px[i]<0) {
      px[i]=0; vx[i]=-vx[i];
    }
    else if (px[i]>width) {
      px[i]=width; vx[i]=-vx[i];
    }
    py[i] = py[i] + vy[i];
    if (py[i]<0) {
      py[i]=0; vy[i]=-vy[i];
    }
    else if (py[i]>height) {
      py[i]=height; vy[i]=-vy[i];
    }
  }
}

void visualizar() {
  background(0);
  for(int i=0; i<num; i++) {
    ellipse(px[i], py[i], 10, 10);
  }
}
```

A Listagem 18 apresenta uma variação do *sketch* da Listagem 17, utilizando um array de objetos de uma classe Entidade, definida inicialmente. A vantagem desta abordagem está em concentrar, em cada objeto, todos os atributos relacionados à mesma entidade.

Listagem 18: Utilização de arrays para múltiplas entidades

```
class Entidade {
  int px, py, vx, vy;

  void simular() {
    px = px + vx;
    if (px<0 || px>width) {
      px=constrain(px,0,width);
      vx=-vx;
    }
    py = py + vy;
    if (py<0 || py>height) {
      py=constrain(py,0,height);
      vy=-vy;
    }
  }
  void visualizar() {
    ellipse(px, py, 10, 10);
  }
}

// arrays de entidades
Entidade e[];
// numero de entidades
int num;
```

```
void setup() {
  size(400, 400);
  num = 20;
  e = new Entidade[num];
  criarEntidades();
}

void draw() {
  simular();
  visualizar();
}

void criarEntidades() {
  for(int i=0; i<num; i++) {
    e[i] = new Entidade();
    e[i].px = (int)random(width);
    e[i].vx = (int)random(-4, 4);
    e[i].py = (int)random(height);
    e[i].vy = (int)random(-4, 4);
  }
}

void simular() {
  for(int i=0; i<num; i++) {
    e[i].simular();
  }
}

void visualizar() {
  background(0);
  for(int i=0; i<num; i++) {
    e[i].visualizar();
  }
}
```

Nos dois sketches anteriores, o comportamento das entidades consiste apenas em mudar de trajetória ao colidir com os limites da tela. No entanto, esta mesma estrutura pode ser utilizada para implementar comportamentos de personagens ou efeitos gráficos de partículas, entre outros.

Detecção e resposta a colisão: testes de colisão são frequentemente necessários em jogos. No caso de protótipos (e mesmo em muitos jogos), pode-se adotar uma representação simplificada das entidades para os testes de colisão. Formas muito utilizadas são círculos e retângulos, pois permitem a realização de cálculos menos complexos. A Listagem 19 apresenta um *sketch* que realiza o teste da colisão entre dois círculos.

Neste caso, o teste corresponde a verificar se a distância entre os centros dos círculos é menor do que a soma dos seus raios, como ilustrado na Figura 18. Van den Heuvel e Jackson [2002] apresentam uma discussão mais detalhada sobre este algoritmo.

Uma imagem do sketch em execução é apresentada na Figura 19. O teste de colisão é realizado na função **colisaoCirculos**, que pode ser reaproveitada em outros *sketches*. Este exemplo também demonstra um efeito de rastro de trajetória, obtido pela sobreposição de uma cor de fundo semitransparente.

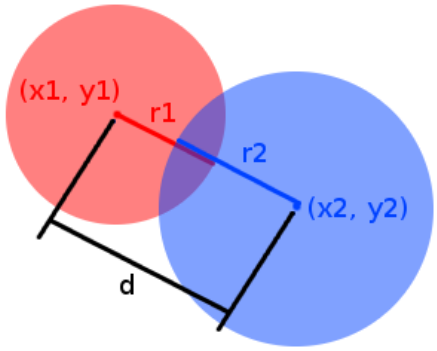

Figura 18: Teste de colisão entre círculos

Listagem 19: Detecção de colisão entre círculos

```
// coordenadas do circulo
// que segue o mouse
int px, py;
// diametro do circulo
int diam;
// coordenadas de outra entidade
int ox, oy;
// diametro da outra entidade
int diam2;
// indica se ha colisao
boolean colisao;

void setup() {
  size(400, 300);
  px = 0; py = 0;
  ox = width/2; oy = height/2;
  diam = diam2 = 80;
  colisao = false;
}

void draw() {
  simular();
  visualizar();
}

void simular() {
  px = mouseX;
  py = mouseY;
  if (colisaoCirculos(px, py, diam/2,
ox, oy, diam2/2)) {
    colisao = true;
  }
  else {
    colisao = false;
  }
}

void visualizar() {
  noStroke();
  fill(0, 0, 0, 25);
  rect(0, 0, width, height);
  fill(255, 255, 255);
  ellipse(ox, oy, diam2, diam2);
  if (colisao) {
    fill(255, 0, 0);
  }
  else {
    fill(0, 0, 255);
  }
```

```
  ellipse(px, py, diam, diam);
}

boolean colisaoCirculos(int x1, int y1,
  int r1, int x2, int y2, int r2) {
  if (dist(x1, y1, x2, y2) < (r1+r2)) {
    return true;
  }
  else {
    return false;
  }
}
```

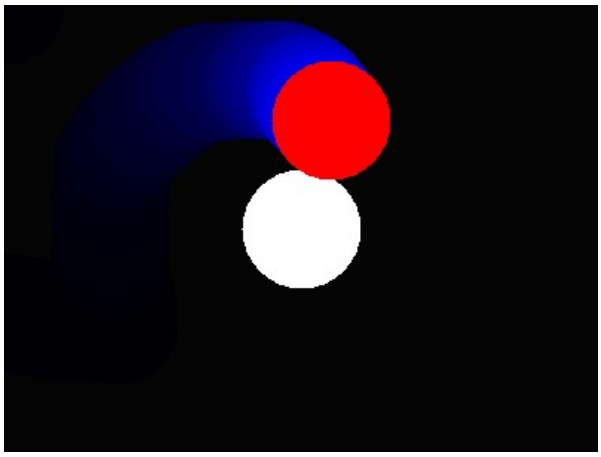

Figura 19: Exemplo de colisão entre círculos

A Listagem 20 contém o código para uma função de teste de colisão entre dois retângulos. Neste caso, o teste de colisão consiste em verificar, separadamente, se eles colidem tanto no eixo horizontal como no eixo vertical. Para esta função, considera-se que os parâmetros (x1, y1) e (x2, y2) são as coordenadas dos centros dos retângulos, como ilustrado na Figura 20. Os algoritmos de detecção de colisão apresentados nestes exemplos também podem ser expandidos para o espaço 3D. Neste caso, consideram-se esferas e paralelepípedos mas os critérios de colisão permanecem essencialmente os mesmos.

Listagem 20: Detecção de colisão entre retângulos

```
boolean colisaoRetangulos(int x1,
  int y1, int l1, int h1,
  int x2, int y2, int l2, int h2) {
  boolean horizontal, vertical;
  if ((x1-l1/2) > (x2-l2/2)) {
    horizontal = ((x1-l1/2) <
                  (x2+l2/2));
  }
  else {
    horizontal = ((x2-l2/2) <
                  (x1+l1/2));
  }
  if ((y1-h1/2) > (y2-h2/2)) {
    vertical = ((y1-h1/2) < (y2+h2/2));
  }
  else {
    vertical = ((y2-h2/2) < (y1+h1/2));
  }
  return (horizontal && vertical);
}
```

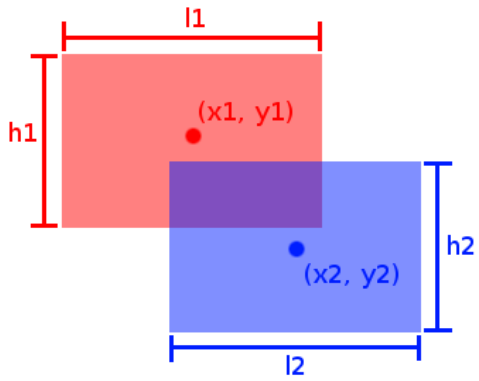

Figura 20: Detecção de colisão entre retângulos

Controle de personagem com teclado: a Listagem 21 apresenta um *sketch* para controle de um personagem através das setas cursoras do teclado. A Figura 21 apresenta o resultado da execução do *sketch*. Deve-se notar que, seguindo a estrutura de simulação e visualização, as funções **keyPressed** e **keyReleased** somente alteram variáveis de velocidade do personagem. A nova posição é calculada na função **simular**. Uma implementação mais estrita poderia, ainda, armazenar em uma variável a direção de movimento do personagem nas funções **keyPressed** e **keyReleased**, calculando tanto a velocidade como a posição na função **simular**.

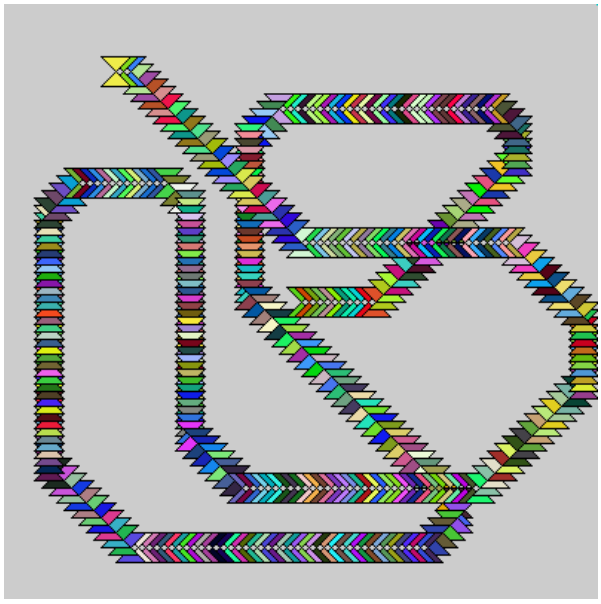

Figura 21: Sketch de controle de personagem pelo teclado

Listagem 21: Controle através do teclado

```
int x, y, vx, vy;

void setup() {
  size(400, 400);
  x = width/2;
  y = height/2;
  vx = 0;
  vy = 0;
}
```

```
void draw() {
  simular();
  visualizar();
}

void simular() {
  x = constrain((x + vx), 10, width-10);
  y = constrain((y + vy), 10, height-
10);
}

void visualizar() {
  fill(random(255), random(255),
random(255));
  triangle(x-10, y+10, x+10, y+10,
           x, y);
  triangle(x-10, y-10, x+10, y-10,
           x, y);
}

void keyPressed() {
  if (key == CODED) {
    switch(keyCode) {
      case UP:
        vy = - 5;
        break;
      case DOWN:
        vy = + 5;
        break;
      case LEFT:
        vx = - 5;
        break;
      case RIGHT:
        vx = + 5;
    }
  }
}

void keyReleased() {
  if (key == CODED) {
    switch(keyCode) {
      case UP:
        vy = 0;
        break;
      case DOWN:
        vy = 0;
        break;
      case LEFT:
        vx = 0;
        break;
      case RIGHT:
        vx = 0;
    }
  }
}
```

A movimentação através do mouse pode ser realizada através da análise das variáveis automáticas mouseX e mouseY no código de simulação. O exemplo de detecção de colisão (Listagem 19) ilustra uma implementação simples.

Espera-se que os exemplos apresentados nesta seção tenham ilustrado o uso do Processing para a criação de protótipos de *gameplay* de forma simples e concisa.

5.3 Protótipos de *game design* completo

Além de protótipos de *gameplay*, o Processing também pode ser utilizado para a prototipação rápida de jogos digitais completos, seguindo-se as práticas discutidas na seção 5.1. A imagem esquerda da Figura 1, bem como a Figura 22 ilustram o protótipo de um jogo de ação controlado com o mouse, em que o jogador deve evitar o contato com os inimigos enquanto a nave à qual ele está conectado segue uma trajetória fixa.

Figura 22: Tela de título de um protótipo de jogo desenvolvido com o Processing

A Figura 23 ilustra outro protótipo de jogo feito com o Processing. Neste caso, trata-se de um jogo no qual o jogador move seu personagem com o teclado e comanda os disparos de sua arma com o mouse, tendo como objetivo sobreviver o maior tempo possível enquanto elimina inimigos que se aproximam. O jogador pode utilizar diferentes tipos de armamentos, inclusive um "lança-chamas".

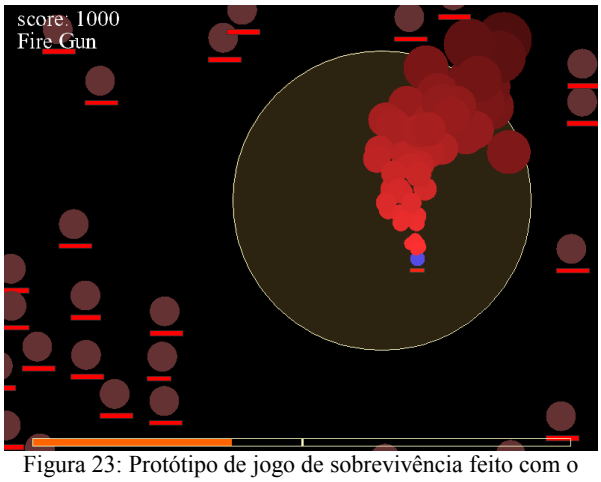

Figura 23: Protótipo de jogo de sobrevivência feito com o Processing

As listagens destes protótipos são muito extensas para serem incluídas neste tutorial, mas podem ser obtidas no site do Laboratório de Tecnologias Interativas, em http://www.interlab.pcs.poli.usp.br.

O Processing é capaz de exportar *sketches* na forma de aplicações e applets para Web. Desta forma, os protótipos podem ser distribuídos e compartilhados em sua forma compilada, caso seja necessário.

5.4 Testes de interação com outros dispositivos

Outro uso para o Processing é o teste de técnicas de interação ou dispositivos de interação diferentes. Embora o Processing só tenha suporte a teclado e mouse em seus sketches, uma maneira bastante simples de se testar técnicas de interação com dispositivos diferentes consiste em se utilizar o software Glove Programmable Input Emulator, ou GlovePIE [Kenner 2009]. Este sistema permite converter sinais de dispositivos de entrada como teclado, mouse, joysticks, luvas de realidade virtual e até mesmo o Wii Remote em sinais de outros dispositivos. Desta forma é possível, por exemplo, converter comandos do Wii Remote para movimentos do mouse e com isso, testar um protótipo de jogo em Processing com aquele dispositivo.

A interface do GlovePIE é apresentada na Figura 24. Esse software utiliza uma linguagem de script para indicar como a conversão de sinais entre dispositivos deve ser feita. A documentação disponibilizada por Kenner [2009] contém os detalhes da linguagem e o software é disponibilizado com um grande conjunto de exemplos. O código a seguir ilustra a configuração para fazer com que o controle direcional de um *gamepad* seja convertido em sinais das teclas cursoras do teclado:

```
Key.Up = Joystick1.pov1up
Key.Right = Joystick1.pov1right
Key.Down = Joystick1.pov1down
Key.Left = Joystick1.pov1left
```

Como um teste simples, pode-se ativar o GlovePIE utilizando a configuração apresentada e controlar o sketch apresentado na Listagem 21 através de um *gamepad*.

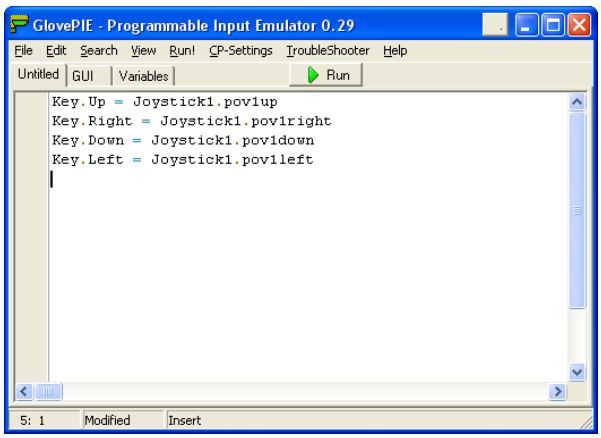

Figura 24: A interface gráfica do GlovePIE

Pela combinação das técnicas discutidas anteriormente com configurações do GlovePIE, pode-se desenvolver de forma rápida protótipos de interação.

5.5 Geração de conteúdo procedural e registro de dados

Na seção 4.1, foi apresentada a capacidade do Processing de gravar imagens. Também é possível manipular arquivos de texto e, com estas duas capacidades, pode-se desenvolver *sketches* para testar a geração procedural de recursos para jogos – ou mesmo criar ferramentas para uso em outros projetos. O *sketch* da Listagem 22 apresenta um exemplo de criação e escrita de dados para um arquivo de texto. Este arquivo é armazenado no mesmo diretório do *sketch*.

Listagem 22: Escrita de dados em arquivo

```
// O arquivo é criado com createWriter
PrintWriter arquivo;
arquivo = createWriter("dados.txt");

// Os dados são escritos com println
arquivo.println("Teste");

// Ao final, é preciso usar os comandos
// flush e close para gravar as
informações
arquivo.flush();
arquivo.close();
```

Pode-se utilizar este tipo de recurso para registrar as ações do jogador. A Listagem 23 apresenta uma classe para registrar mensagens de texto com data e hora, bem como um exemplo de utilização.

Listagem 23: Classe para registro de eventos em arquivo

```
class DataLog {
  PrintWriter arquivo;
  void iniciarRegistro() {
    arquivo =
createWriter("registro.txt");
  }
  void terminarRegistro() {
    arquivo.flush();
    arquivo.close();
  }
  void gravarMensagem(String msg) {
    arquivo.println(year() + "-" +
month() +
      "-" + day() + "#" + hour() + ":" +
      minute() + ":" + second() + "#" +
      msg);
  }
}
// Exemplo de uso
DataLog d = new DataLog();
d.iniciarRegistro();
d.gravarMensagem("Jogador iniciou o
jogo");
d.gravarMensagem("Jogador escolheu
personagem");
d.gravarMensagem("Jogador encerrou o
jogo");
d.terminarRegistro();
```

Esta técnica de registro de eventos e mensagens pode ser utilizada em conjunto com protótipos de teste de interação para coletar dados detalhados sobre a utilização do protótipo.

Com isso, conclui-se a apresentação de diferentes propostas e formas de utilização do Processing em atividades de *game design,* principalmente na construção de protótipos, tanto de funcionalidades específicas ou de jogos completos.

6. Outras bibliotecas do Processing

Ao longo deste tutorial, o foco foi na interação e geração de imagens. No entanto, o Processing é distribuído com outras bibliotecas que permitem gravar imagens em formato PDF, reproduzir arquivos de som, gerar sons sintetizados, reproduzir vídeos e acessar câmeras de vídeo, entre outros. Além disso, pode utilizar bibliotecas externas para expandir suas funcionalidades. Nesta seção, serão apresentadas duas destas bibliotecas.

6.1 Minim Audio

A biblioteca Minim é distribuída juntamente com o Processing. Para utilizá-la em um sketch, basta selecionar o comando de menu *Sketch → Import Library → Minim Audio*. Esta biblioteca permite a reprodução de arquivos de áudio em diversos formatos (inclusive MP3) e síntese de efeitos sonoros. Ela está organizada na forma de várias classes, de acordo com a funcionalidade desejada.

A classe **AudioPlayer** é utilizada para a reprodução de arquivos de som longos, como trilhas de música. Neste caso, a biblioteca automaticamente realiza o *streaming* da música, evitando uma ocupação exagerada da memória. A classe **AudioSample** é utilizada para efeitos sonoros de curta duração. Neste caso, o arquivo de áudio inteiro é carregado na memória, para execução mais rápida. As classes **AudioSignal** e **AudioEffect** podem ser utilizadas para síntese e manipulação de áudio. O Processing inclui uma série de exemplos de utilização da biblioteca Minim; recomenda-se o seu estudo para conhecer os detalhes de uso.

6.2 JMyron

A biblioteca JMyron é uma versão em Java da biblioteca de captura de imagens Myron [2009]. Seu objetivo é a captura e processamento de imagens a partir de câmeras. Ela só está disponível para a plataforma Windows.

Como a biblioteca JMyron não é distribuída com o Processing, é preciso fazer sua instalação, que consiste em duas etapas. Primeiro, deve-se descompactar o seu arquivo no diretório de bibliotecas do Processing. Normalmente, este diretório é "**\Usuários\<nome do usuário> \Documentos\ Processing\libraries**". O segundo passo consiste em copiar os arquivos **DSVL.dll** e **myron_ezcam.dll** para o diretório onde o Processing foi instalado.

Uma vez instalada, a biblioteca pode ser adicionada a um sketch através do comando de menu *Sketch → Import Library → JMyron*. Esta biblioteca define uma classe principal chamada JMyron; um único objeto desta classe deve ser criado para que se tenha acesso a uma *webcam* conectada ao computador. Recomenda-se o estudo dos exemplos distribuídos com a biblioteca para conhecer mais detalhes sobre a sua utilização.

O processo de adição de outras bibliotecas externas é essencialmente o mesmo descrito para a instalação da biblioteca JMyron. Desta forma, é possível acrescentar outras bibliotecas Java para expandir as capacidades do Processing.

7. Conclusão

Este tutorial teve como objetivo apresentar a ferramenta Processing e discutir o seu uso na prototipação de jogos digitais. Desta forma, foram descritos a linguagem, os modelos de programação de *sketches* e o modelo gráfico do Processing. Em seguida, foram apresentadas diferentes maneiras de se utilizar seus recursos em atividades de *game design*. Por fim, foi apresentada a forma de se expandir as funcionalidades da ferramenta através de novas bibliotecas. Espera-se que o material deste tutorial possa servir como referência, tanto para game designers como designers em geral interessados no uso do Processing.

Os autores acreditam no potencial de utilização de Processing em design e desenvolvimento de jogos. Por este motivo, através do Laboratório de Tecnologias Interativas da USP, estão desenvolvendo um kit de ferramentas para simplificar o desenvolvimento de jogos em Processing.

Seguindo o ponto de vista de que conhecimentos de programação tendem a ser cada vez mais relevantes para atividades de design, os autores também estão produzindo uma versão expandida deste tutorial para utilização como material didático no ensino de fundamentos de computação para estudantes de design.

As novidades sobre as ferramentas e material didático sobre Processing e desenvolvimento de jogos produzidos pelo Interlab poderão ser acompanhadas em seu site: http://www.interlab.pcs.poli.usp.br.

Agradecimentos

Os autores gostariam de agradecer Daniel Makoto Tokunaga pela colaboração com exemplos do Processing para este tutorial.

Bibliografia Recomendada

REAS, C. E FRY, B., 2007. Processing: A Programmer's Handbook for Visual Designers and Artists. Cambridge: MIT Press.

SHIFFMAN, D., 2008. Learning Processing: A Beginner's Guide to Programming Images, Animation, and Interaction. San Francisco: Morgan Kaufmann.

WATT, A., 1999. 3D Computer Graphics. Reading: Addison-Wesley.

Referências

ECKEL, B. 2009. Thinking in Java, 3rd Edition. Disponível em: http://www.mindview.net/Books/TIJ/ [Acesso em 12 Setembro 2009].

MYRON, 2009. Disponível em: http://webcamxtra.sourceforge.net/index.shtml [Acesso em 12 Setembro 2009].

KENNER, C., 2009. GlovePIE: Glove Programmable Input Emulator. Disponível em: http://carl.kenner.googlepages.com/glovepie [Acesso em 12 Setembro 2009].

PROCESSING 1.0, 2009. Disponível em: http://www.processing.org [Acesso em 12 Setembro 2009].

ROUSE III, R., 2005. Game Design: theory and practice. Plano: Wordware.

SCHUYTEMA, P., 2008. Design de Games: uma abordagem prática. São Paulo: Cengage Learning.

VAN DEN HEUVEL, J. E JACKSON, M., 2002. Pool Hall Lessons: Fast, Accurate Collision Detection between Circles or Spheres. Disponível em: http://www.gamasutra.com/features/20020118/vandenhuevel_01.htm [Acesso em 12 Setembro 2009].