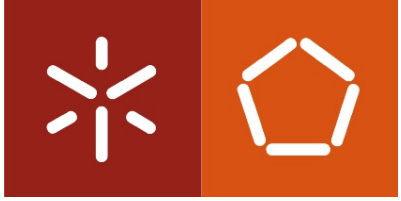
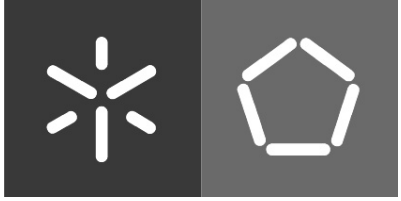


Universidade do Minho
Escola de Engenharia

Rui Filipe Veiga Rebelo Da Silva

Implementação de grafos no ambiente de programação visual Scratch

Abril de 2012



Universidade do Minho
Escola de Engenharia

Rui Filipe Veiga Rebelo Da Silva

Implementação de grafos no ambiente de programação visual Scratch

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho efectuado sob a orientação de
Prof. Doutor Fernando Mário Junqueira Martins

Abril de 2012

Declaração

Nome: Rui Filipe Veiga Rebelo Da Silva

Endereço Electrónico: ruidasilva11@gmail.com

Telefone: 913435636

Cartão de Cidadão: 13223462

Título da Tese: Implementação de grafos no ambiente de programação visual Scratch

Orientador: Professor Doutor Fernando Mário Junqueira Martins

Ano de conclusão: 2012

Designação do Mestrado: Mestrado em Engenharia Informática

Universidade do Minho, 30 de Abril de 2012

Rui Filipe Veiga Rebelo Da Silva

Agradecimentos

Agradeço ao meu orientador Prof. Dr. Fernando Mário J. Martins pela experiência, por todas as contribuições, disponibilidade, encorajamento e orientação que me deu desde o início.

Aos meus pais por toda a dedicação e motivação ao longo de todo o meu percurso académico. À minha mãe pelo apoio incondicional. Ao meu pai por toda a compreensão. Sem o seu sacrifício dificilmente teria sido possível chegar até aqui.

A toda a minha família e ao meu irmão pela amizade e companheirismo.

À Sara pela inspiração, ajuda, amizade e encorajamento em todos os momentos.

Ao colega e amigo, João Gonçalves, pelos conhecimentos partilhados e incentivo mútuo durante todo trabalho.

A todos os amigos por todos os momentos, apoio e amizade, em especial ao Pedro Soares.

A todos os colegas de curso pelo grande companheirismo e amizade ao longo de toda a minha vida académica.

A todos os demais...

Abstract

The Scratch language is a visual programming language launched in 2007 by the MIT Media Labs which has been used in e-learning environments for children. The Scratch environment was developed using the object oriented programming language Squeak. The visual programming in Scratch is based on the metaphor of the LEGO block that fits into compatible blocks to create behaviours, usually animations. The success of the open source environment led to the need of revision and improvement because of its possible application in other contexts. This work aims to create a set of extensions to the library of data structures of the Scratch visual programming language, which in its current version only allows manipulation of lists. The objective is to implement a new data structure, graphs, giving them visual representations, and creating ways to manipulate them interactively either by using the Scratch blocks or by direct manipulation using the mouse.

Implementação de grafos do ambiente de programação visual Scratch

Resumo

A linguagem Scratch é uma linguagem de programação visual lançada em 2007 pelo MIT Media Labs que tem vindo a ser usada em contextos de *e-learning* para crianças. O ambiente Scratch foi desenvolvido usando a linguagem de programação por objetos Squeak. A programação visual em Scratch baseia-se na metáfora do bloco de LEGO que se encaixa em blocos compatíveis para criar comportamentos, em geral animações. O sucesso deste ambiente *open source* conduziu à necessidade de o rever e aumentar visando a sua eventual aplicação noutros contextos. Este trabalho tem por objetivo criar um conjunto de extensões à biblioteca de estruturas de dados da linguagem visual de programação Scratch, que na sua versão atual apenas permite a manipulação de listas. Pretende-se implementar uma nova estrutura de dados, grafos, atribuindo-lhes representações visuais e criando formas de as manipular de forma interativa através de blocos Scratch ou de forma direta através de interação com o rato.

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

Neste capítulo será feita uma introdução geral aos temas abordados nesta dissertação de mestrado. Em primeiro lugar, na secção 1.1, será feita uma apresentação ao tema da programação visual contendo a sua definição, objetivos e um pouco da sua história.

De seguida, na secção 1.2, será explicada a relação importante das linguagens de programação visual com o ensino e também serão apresentados alguns conceitos fundamentais destas linguagens visuais.

Na secção 1.3 será feita uma pequena introdução ao ambiente visual de programação no qual o trabalho desta dissertação se focou, o Scratch. Serão expostos alguns dos seus conceitos essenciais e objetivos principais do seu lançamento bem como uma breve explicação da sua interface e linguagem visual de programação.

Depois disto, na secção 1.4, será feita uma pequena apresentação dos objetivos desta dissertação e também uma primeira descrição do trabalho desenvolvido para os alcançar.

Por fim será feita uma descrição dos vários capítulos deste documento na secção 1.5.

1.1 Programação visual

De acordo com [BB94]: "A programação visual refere-se a qualquer sistema que permite ao utilizador especificar um programa usando duas (ou mais) dimensões. Embora esta seja uma definição muito ampla, as convencionais linguagens textuais não são consideradas bidimensionais, pois os compiladores ou interpretadores processam-nas como longos *streams* unidimensionais".

A programação visual utiliza mais de uma dimensão para demonstrar a sua semântica e construir expressões visuais, daí o termo "visual". Uma expressão visual é um conjunto de *tokens* visuais que correspondem a cada palavra nas linguagens de programação tradicionais. Esses *tokens* visuais correspondem a objetos multidimensionais ou relações semânticas espaciais. Muitas vezes, estas expressões visuais assumem a forma de diagramas, variáveis, ícones, objetos ação/controlo e objetos gráficos.

O objetivo da programação visual, conforme descrito em [BBB+95] é fazer com que o processo de expressar e/ou compreender programas se torne mais fácil através de simplicidade, concretude, clareza e capacidade de resposta. Isto é conseguido de várias formas, nomeadamente reduzindo o número de conceitos que fazem parte da linguagem de programação, permitindo que objetos de dados sejam explorados diretamente representando conexões e relacionamentos entre si e também ao mesmo tempo apresentando *feedback* visual de todas as interações, cálculos e semântica.

De acordo com [Web99] a programação visual baseia-se em quatro estratégias principais para alcançar seus objetivos: concretude, manipulação direta, clareza e *feedback* visual imediato. Concretude significa que, usando objetos particulares a linguagem de programação visual permite ao programador especificar aspetos importantes de qualquer parte específica do programa (objetos, valores, etc.). Manipulação direta significa permitir a manipulação de partes do programa diretamente pelo programador, reduzindo as ações necessárias para alcançar essa

mudança quando comparado com o processo equivalente em linguagens textuais. Clareza refere-se à representação explícita dos aspetos importantes da semântica como as relações entre variáveis e declarações. E por último, o *feedback* visual imediato refere-se à representação automática no ecrã dos efeitos de mudanças em objetos editados do programa e nas suas partes e valores afetados usando o paradigma visual do ambiente de programação visual.

A programação visual começou desenvolver-se de duas formas. Uma delas foi o desenvolvimento de fluxogramas executáveis devido a que na altura se pensava que estes eram uma boa maneira de ensinar conceitos de programação. A outra foi o desenvolvimento de abordagens aplicáveis a programação visual como por exemplo, programas que demonstram as ações desejadas no ecrã. Estas abordagens iniciais, apesar de intuitivas, tinham alguns problemas quando estendidas para implementar programas de maior complexidade, [Nic94].

Na década de 70 e 80 as várias linguagens de programação visuais continuaram a desenvolver-se, mas tiveram alguma dificuldade em estabelecer-se devido às más capacidades gráficas dos computadores na altura. Uma das exceções foi o Prograph que permaneceu em desenvolvimento até o ano de 1990. Hoje, existem muitas linguagens de programação visual populares em uso.

1.2 Programação visual no ensino

O ensino de linguagens de programação e algoritmos a alunos principiantes é uma tarefa difícil. Isto acontece porque é necessário que estes desenvolvam muito cedo no curso um alto nível de abstração e raciocínio lógico, o que torna o processo de aprendizagem de programação uma tarefa lenta e gradual [Dij88]. Para ter sucesso o aluno deve adquirir capacidades de resolução de problemas, raciocínio lógico, codificação e depuração. Isto é muitas vezes difícil de conseguir em pleno

devido à grande quantidade de informação e conceitos apresentados nos primeiros tempos do ensino de programação.

Estas dificuldades que os alunos experimentam em assimilar, por exemplo, o processo abstrato de um algoritmo, são frequentemente a razão que leva muitos ao fracasso e desistência, por vezes muito cedo no curso. Pereira Jr. e Rapkiewickz [PJ04] [JR05] apresentam pesquisa sobre este assunto. Além disso, a metodologia de ensino de hoje ainda requer grandes volumes de leitura do texto, o que faz com que todo o processo de aprendizagem se torne por vezes monótono. Por outro lado, e devido à sua complexidade, as linguagens de programação textuais em uso hoje em dia não parecem ser ferramentas adequadas para ensinar alunos principiantes os fundamentos da programação. É aqui que a programação visual é necessária e útil.

Diversas pesquisas mostram que o uso de gráficos e animação, como são usados em ambientes de programação visual, são uma ferramenta de ensino eficazes para manter os estudantes interessados nos seus cursos [Rod02], [GU03]. Na aplicação destas ferramentas ao ensino de programação os conceitos desta são apresentadas através de objetos visuais de uma forma animada, tornando o desenvolvimento de programas num processo mais apelativo e divertido para os alunos [Rod02]. Por consequência é alcançada a captura da atenção dos alunos com atividades divertidas e interessantes ao mesmo tempo que é mantido o rigor académico necessário contribuindo isto para o melhoramento da taxa de recrutamento e retenção de alunos [HSS+10] nas fases iniciais dos cursos de ciências da computação. Este facto também é válido para o ensino de muitas outras disciplinas.

Os ambientes visuais de programação para o ensino têm vários objetivos [Pas09]:

- O uso de metáforas que sejam compreendidas com facilidade por um público amplo;
- A redução da carga cognitiva sobre os estudantes que aprendem sua primeira

linguagem de programação;

- A criação de código que é facilmente lido e entendido;
- Ser facilmente utilizado por professores;
- Suportar as matérias atuais e de fácil integração;
- Suportar os alunos na transição para linguagens de programação textuais atuais;
- Ser divertido e apelativo.

Existe uma grande variedade de linguagens de programação e ambientes visuais que foram projetados explicitamente como primeiras linguagens de programação. Existem ferramentas narrativas que suportam a programação para "contar uma história"(por exemplo, Scratch, Alice [MLC04] , Jeroo [SD03]), onde os mundos de histórias criados podem ser interativos e não interativos. Este conceito de "contar uma história"é muito atraente porque faz com que os alunos se interessem pelos programas que eles próprios constroem, e devido a essa motivação gastam mais tempo na tarefa em curso e na sua programação.
Existem também ferramentas de programação visual que suportam a construção de programas através de uma interface *drag-and-drop*. Estas ferramentas permitem ao utilizador manipular diretamente as representações gráficas dos conceitos de programação ao invés de digitar caracteres individuais (por exemplo, Scratch, JPie [Gol04] , Alice [MLC04] , Karel Universo). Desta forma, o utilizador leva sempre o programa de um estado legal para outro legal, eliminando assim os possíveis erros de sintaxe no desenvolvimento.
E existem também ferramentas de fluxogramas (por exemplo, Raptor [CWHH05], Iconic Programmer [CM05], VisualLogic, Prograph) que constroem programas

através da conexão de elementos do programa para representar a ordem de computação. Estas ferramentas permitem que os alunos visualizem a maneira como os programas funcionam e permitem o desenvolvimento dos programas de uma forma mais intuitiva. Isto é conseguindo reduzindo a complexidade sintática o que foca os alunos na resolução do problema em questão e não na sintaxe do programa.

1.3 Ambiente Scratch

O ambiente Scratch foi desenvolvido pelo MIT Media Lab e construído em Squeak. O Squeak é uma moderna implementação do Smalltalk-80 e contém uma *framework* para construção interativa de objetos gráficos animados que podem ser manipulados diretamente. Esta *framework* é o Morphic.

Após a sua criação o Scratch foi lançado em 2007 com o lema "imagina-programa-partilha"considerado adequado aos seus objetivos primários. Um dos objetivos foi fornecer um ambiente de programação atraente e simples para a iniciação dos jovens às tecnologias de informação e proporcionar, através do seu uso, o desenvolvimento nestes de capacidades na resolução de problemas por computador. Outro dos objetivos foi fornecer uma plataforma para o ensino de conteúdos de várias disciplinas de uma maneira gráfica e interativa.

O ambiente Scratch aparece intimamente ligado ao projeto OLPC (One Laptop Per Child) do MIT, liderado pelo fundador do Laboratório do MIT, Nicholas Negroponte, a quem se uniram desde o início Seymour Papert (criador do Logo e pioneiro em inteligência artificial) e Alan Kay (um dos criadores do Smalltalk, pioneiro da programação orientada a objetos e ambientes interativos baseados em janelas).

Inicialmente o Scratch foi disponibilizado para Windows e plataformas Mac, com projetos paralelos em andamento para que o IDE esteja disponível para dispositi-

vos móveis, robótica e outras plataformas (OLPC, Classmate, Magalhães, etc.). A componente de partilha do Scratch é atualmente garantida pelo servidor ScracthR, que é um portal onde as aplicações atuais em Scratch, tipicamente animações e construções gráficas, podem ser carregadas e compartilhadas. Em Portugal, desde 2009, os portais Sapo Kids e Meo Kids usam esta tecnologia.

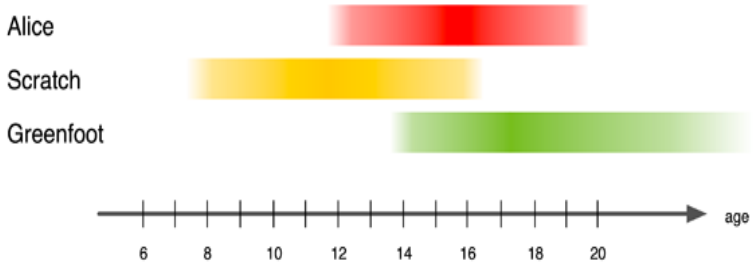


Figura 1.1: Intervalo de idades do Scratch

A programação visual em Scratch é baseada na metáfora do bloco LEGO. Existem blocos de instruções que se podem encaixar noutros blocos compatíveis criando **Scripts** que representam comportamentos de objetos animados.
Um projeto Scratch é composto por um palco, **Stage**, e por objetos visuais chamados **Sprites**. Cada um dos **Sprites** é uma imagem bidimensional que é integrada numa cena maior e que tem associados a si os seus **Scripts**, sons e trajes. O traje de cada **Sprite** representa a sua aparência no palco. Este pode ser trocado dando ao **Sprite** uma nova aparência de palco. Estes trajes podem ser importados a partir de imagens, podem ser construídos no editor de pintura embutido no ambiente ou até criados tirando uma foto com a câmara do computador. Cada **Sprite** também pode ter sons associados a ele que podem ser invocados através dos blocos Scratch. Para controlar o comportamento e animação dos **Sprites** estes recebem instruções. Estas instruções correspondem aos seus **Scripts** associados, que são construídos juntando os blocos visuais do Scratch de acordo com a regras

de semântica da linguagem visual. O Scratch oferece blocos com muitas funcionalidades distintas. A partir deles é possível controlar as movimentações do **Sprite** no palco, a sua aparência e sons e interação com outros **Sprites** e palco. Também com os blocos é possível criar variáveis e listas visuais que se podem manipular diretamente em palco.

O enorme sucesso internacional do ambiente Scratch levou a expectativas do seu uso noutros contextos mais amplos, embora baseado no mesmo paradigma. Existe a possibilidade de melhorar as suas características com o objetivo de apoiar o ensino de conceitos mais complexos de programação. Este é um dos objetivos deste trabalho como será explicado na secção seguinte.

1.4 Trabalho desenvolvido

Esta dissertação tem dois objetivos fundamentais. Um deles é o aumento dos conceitos de programação presentes na linguagem visual do Scratch e possíveis de serem utilizados no ensino e aprendizagem de programação. O segundo é o aumento das funcionalidades do Scratch de maneira a que seja possível aos seus utilizadores criar projetos mais complexos de uma maneira mais simples e intuitiva através da utilização destes novos componentes. Consequentemente, é pretendido que este poderoso ambiente de ensino e desenvolvimento de animações se torne uma ferramenta ainda mais completa e útil, tanto na aprendizagem de programação como no desenvolvimento de projetos Scratch.

Para cumprir estes objetivos foram incorporados no ambiente visual Scratch um conjunto de novos objetos visuais. Estes objetos representam conceitos de programação mais avançados do que os já existentes no ambiente, nomeadamente novas estruturas de dados visuais.

Assim o trabalho desenvolvido nesta dissertação centrou-se na criação de uma

extensão do Scratch que permitisse a visualização e manipulação de uma nova estrutura de dados no ambiente. Esta estrutura teria de ser mais complexa do que as já existentes e também permitir que a sua manipulação pudesse ser feita tanto por blocos Scratch como diretamente no palco através de uma janela de visualização de manipulação direta.

Numa primeira fase este trabalho consistiu no estudo do funcionamento do código fonte *open source* do Squeak e da sua *framework* de criação de objetos animados, o Morphic. A partir deste estudo foi possível compreender a implementação do ambiente gráfico do Scratch, como são formados e criados os seus objetos e como é feita a interação com cada objeto. Foi analisado também o mecanismo de controlo dos blocos, dos **Sprites**, das variáveis e das listas.

Depois de compreendido o funcionamento do Scratch foi implementada a nova estrutura de dados visual no ambiente, o grafo. Para a total integração de grafos na linguagem de programação visual do ambiente Scratch foram desenvolvidos vários componentes e funcionalidades necessárias:

- Foi criada uma estrutura de dados genérica em Squeak para guardar e manipular toda a informação de um grafo.

- Foi criado um objeto visual novo e interativo para representação, manipulação e navegação sobre um grafo no ambiente Scratch.

- Foram criados novos blocos Scratch específicos para manipulação de grafos no ambiente visual.

- Foi implementada uma funcionalidade de exportação do conteúdo de um grafo para um ficheiro do tipo *.dot*. Este tipo de ficheiros pode ser lido pelo programa Graphviz para construir uma imagem representativa de um grafo completo.

- Todos estes componentes novos foram completamente integrados na interface e na linguagem visual do Scratch.

- Foram atualizadas as funcionalidades de gravação e carregamento de projetos Scratch para que suportem os novos componentes da linguagem visual.

1.5 Descrição desta dissertação

Nesta dissertação, o capítulo 2 apresenta no geral a interface do ambiente Scratch bem como algumas das suas extensões já existentes e contributos de cada uma para o ambiente. Neste capítulo também são apresentados alguns dos ambientes de programação visual mais relevantes hoje em dia que são mais direcionados ao ensino de programação.
No capítulo 3 é apresentado o ambiente Scratch a fundo. É feita uma apresentação total da sua interface e das suas capacidades. São analisadas as categorias e tipos de blocos Scratch e também o processo pelo qual são construídos os **Scripts** de blocos na linguagem visual. Além disto são também explicados o modelo de objetos do Scratch baseado em **Sprites**, bem como o modelo de comunicação entre estes. Neste capítulo são também identificados e demonstrados os conceitos de programação que podem ser replicados pelos blocos Scratch.
No capítulo 4 é explicada a *framework* de interação Morphic do Squeak. São apresentados os seus principais blocos de construção os *morphs*. Estes são os objetos interativos que sustentam a construção de qualquer animação e é a partir destes que o ambiente Scratch é construído na sua totalidade.
Depois de compreendido o Morphic e as possibilidades que oferece passamos para o capítulo 5. Neste capítulo são descritas todas as extensões desenvolvidas no ambiente e como foram implementadas. São explicados todos os novos componentes que suportam o uso de grafos visuais pelos utilizadores. Estes componentes são

os novos blocos de grafo e o visualizador de palco para manipulação e navegação sobre grafos. Aqui é descrita a maneira como se encontram construídos e também como foram integrados nas classes que compõem o código Squeak do ambiente visual Scratch.

Em seguida o capítulo 6 apresenta o manual e um exemplo de uso das novas funcionalidades. É identificada a função de cada um dos blocos de grafo bem como o processo de manipulação direta do grafo através do seu visualizador de palco. Em seguida é apresentado um exemplo prático do uso destes novos componentes. Finalmente no capítulo 7 é feita uma revisão geral do que foi conseguido e também são propostas algumas funcionalidades futuras que seriam relevantes para o avanço do Scratch como ambiente de programação visual para ensino de programação.

1.6 Conclusão

Neste capítulo foi feita uma introdução aos principais temas desta dissertação. Foi feita uma introdução à programação visual, seus principais conceitos e sua utilização como ferramenta de ensino de programação. Foi feita também uma introdução ao ambiente visual Scratch que é o ambiente visual de programação no qual este trabalho se foca. Depois disto foram explicitados os objetivos que foram a base do trabalho desenvolvido. Finalmente foi feita uma descrição geral e resumida dos vários capítulos desta dissertação.

Capítulo 2

Estado da Arte

Neste capítulo será descrito o estado da arte relevante investigado no âmbito do tema desta dissertação. Será feita em primeiro lugar uma apresentação não muito extensa da interface básica e da linguagem visual atual do ambiente Scratch na secção 2.1. Ainda nesta secção será apresentado o Squeak que é a linguagem em que o Scratch foi construído e também o Smalltalk-80 que é a linguagem base de programação que o Squeak implementa.
Na secção seguinte, secção 2.2, serão enumeradas as principais extensões ou *mods* já existentes do Scratch e as novas funcionalidades que cada um oferece.
Por fim, na secção 2.3, serão referidos alguns ambientes e linguagens visuais relevantes para o ensino de programação ou com semelhanças com o Scratch .

2.1 Scratch

2.1.1 Interface e linguagem visual do Scratch

Na figura 2.1 encontra-se os principais componentes da interface do ambiente Scratch. O palco, ou **Stage**, é o lugar onde as animações são apresentadas pois é aqui que os **Sprites** se movem e interagem. Abaixo do palco localizam-se os botões

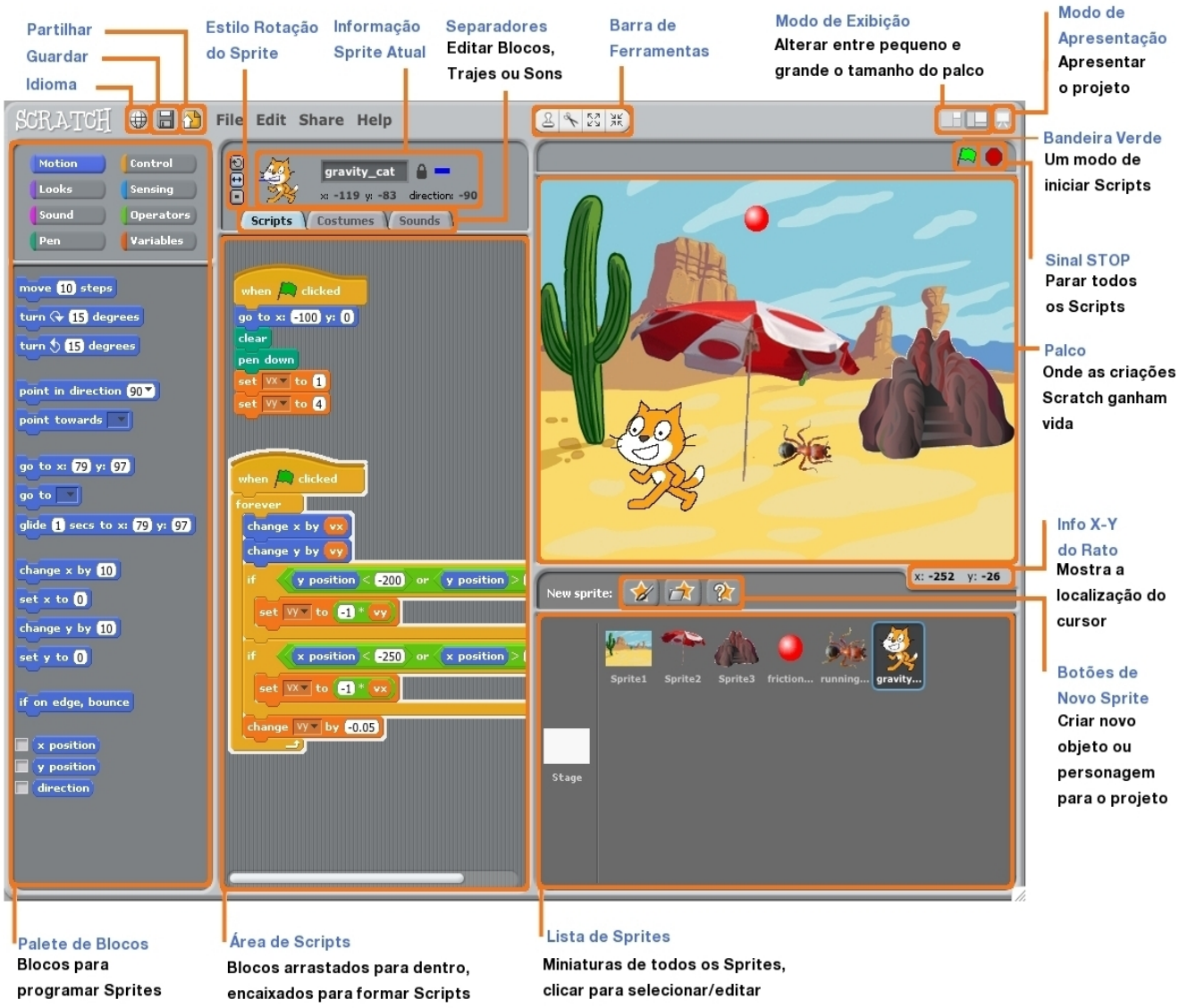


Figura 2.1: Interface do Scratch

de criação de novos **Sprites** onde é possível criar um **Sprite** novo, importando uma imagem, selecionar um **Sprite** existente ou pintar o seu próprio **Sprite**. Em seguida, por baixo destes botões localiza-se a lista de **Sprites** que exibe todos os **Sprites** existentes no projeto, os seus nomes, número de **Scripts** associados a ele e seus trajes.

Para construir **Scripts** o utilizador deve escolher blocos visuais de instruções que estão exibidos na paleta de blocos e uni-los na Área de **Scripts**. Além disso nesta área de **Scripts** o utilizador pode também ver os trajes e os sons associados ao **Sprite** ativo clicando na aba **Costumes** ou na aba **Sounds** na parte superior desta área.

A paleta de blocos oferece oito categorias de blocos de diferente funcionalidade: movimento, aparência, som, caneta, controlo, sensores, operadores e variáveis.

Blocos de movimento controlam o movimento e rotação do **Sprite**;

Blocos de aparência controlam a aparência do **Sprite**, permitem mudar o tamanho dos trajes de um **Sprite** bem como mudança de traje e outras.

Blocos de som controlam os tempos e mudanças dos sons.

Blocos de caneta controlam o desenho no palco através do controlo de uma "caneta"e suas características.

Blocos de controlo oferecem estruturas de controlo sobre os **Scripts**, como ciclos, iteração, condições e também permitem o envio e receção de eventos criados pelo utilizador.

Blocos de sensores detectam eventos ou propriedades especificas, como por exemplo: reportar se o botão do rato está premido ou se o **Sprite** tocou numa determinada cor ou a distância entre dois **Sprites**, etc.

Blocos de operador executam funções matemáticas e manipulam *strings*.

Blocos de variáveis armazenam valores e *strings* em variáveis e também suportam a criação e manipulação de listas.

De acordo com o guia de referência do Scratch [Scr09] existem três tipos principais de blocos: Blocos **Stack**, **Hats** e **Reporters**.

Blocos Stack têm ranhuras na parte superior e podem ter ou não encaixes na parte inferior, estes blocos podem ser encaixados uns nos outros. Alguns destes contêm uma área de entrada no seu interior para digitar um número ou escolher um item de um menu suspenso. Dentro desta categoria existem alguns blocos, como o bloco **if** e o **forever if**, que são **C-shaped** porque têm uma forma de 'C' e permitem a inserção de outros blocos *Stack* no seu interior.;

Blocos Hat têm o topo arredondado e são colocados sempre no topo dos **Scripts** de blocos. Estes esperam pelo acontecimento de um determinado evento e, quando ele acontece executam o **Script** de blocos que iniciam, controlando deste modo o início da execução desse **Script**;

Blocos Reporter são blocos que reportam valores a outros blocos e encaixam-se em áreas de entrada de outros blocos. Estes podem ter duas formas, podem ter pontas arredondadas reportando números ou *strings* e podem ter pontas pontiagudas reportando valores booleanos. Cada um destes tipos encaixa-se noutros blocos com áreas de entrada com estas formas especificas.

A interface e todas as possibilidades originais do ambiente Scratch estão explicadas a fundo no capítulo 3.

2.1.2 Squeak

O Squeak, como já foi referido, é a linguagem em que o ambiente Scratch foi construído. É uma implementação moderna e portável de Smalltalk-80, cuja máquina virtual é também inteiramente escrita em Smalltalk. Foi desenvolvido em Macintosh, mas foi portado para muitos outros sistemas operativos. Inclui suporte para som, cor e processamento de imagem independente da plataforma. O Squeak é conhecido por ser um Smalltalk prático no qual um qualquer programador pode examinar o código fonte de qualquer parte do sistema, incluindo primitivas de gráficos e mesmo a máquina virtual. Isto significa a possibilidade de fazer alterações imediatamente sem a necessidade de lidar com qualquer outra linguagem que não o Smalltalk.
No Squeak o original MVC (*Model View Controller*) *toolkit* de gráficos do Smalltalk-80 é substituído por um *framework* para construção gráfica interativa de objetos animados e também pelas ferramentas para suportar o ambiente Smalltalk. Este *framework* é totalmente desenvolvido em Smalltalk e tem o nome de Morphic. "Morphic é uma *framework* de interface que torna mais fácil e divertido a construção de *User Interfaces* (UI) interativas e animadas"[Mal02], é um *kit* de construção com uma interface de utilizador (UI) de manipulação direta baseado em *display trees*. Os objetos animados de construção são chamados *morphs* e podem ser construídos tanto interativamente (por manipulação direta) ou por código Smalltalk.

2.1.3 Smalltalk

Como já foi dito o Scratch foi desenvolvido em Squeak que é uma implementação de Smalltalk-80 [III03]. O Smalltalk é um dos primeiros ambientes verdadeiramente

orientados a objetos e caracteriza-se por ter um ambiente e linguagem de alto nível de programação declarativa. Assim, o Smalltalk implementa todos os cálculos como objetos que trocam mensagens entre si sendo que qualquer mensagem pode ser enviada para qualquer objeto e, em seguida, o objeto receptor determina se esta mensagem é apropriada e o que fazer para processá-la. Existem apenas três operadores integrados na linguagem: enviar uma mensagem a um objeto, atribuir um objeto a uma variável e retornar um objeto de um método.

O Smalltalk é também uma linguagem altamente extensível, pois permite a criação de objetos que podem ser reutilizados facilmente. Tudo é modificável, o IDE (*Integrated Development Environment*), estruturas de controlo e, em alguns casos, até mesmo a sintaxe, bem como o trabalho do coletor de lixo (*garbage collector*). O coletor de lixo é o sistema de gestão automática de memória que procura resgatar a memória usada pelos objetos. Além disto, a fim de tornar a linguagem mais concisa, não há tipos definidos no código, portanto, todos eles são dinâmicos.

Outra importante caraterística da linguagem Smalltalk é a utilização de uma arquitetura MVC (*Model-View-Controller*). Nesta arquitetura os componentes estão separados em três grupos, modelo de dados, interface de utilizador e lógica de controlo. O modelo de dados gere o comportamento e as informações da aplicação, envia mensagens de resposta sobre o seu estado e muda o seu estado no comando. A componente de interface é o elemento de interface com o utilizador, gere a visualização da aplicação. E por fim o controlador recebe informações e processa-as enviando mensagens de controlo para outros objetos para estes executarem ações.

2.2 Extensões do Scratch

Serão apresentadas nesta secção algumas das extensões mais importantes e relevantes já existentes do Scratch.

2.2.1 Chirp

Chirp é uma versão ligeiramente modificada do ambiente Scratch e baseada no seu código fonte. É totalmente compatível com o Scratch V.1.2.1 e com o *website* Scratch. Esta versão tem algumas características adicionais. É possível exportar/importar **Scripts** de blocos como arquivos XML, permite ao utilizador alterar blocos através do menu de contexto, fazer *scroll* na janela do IDE em situações de baixa resolução e ampliar o painel de *scripting*. Este projeto também tem um Windows *Installer* para iniciar diretamente projetos no modo de apresentação, e permite também a utilização projetos Scratch como *screensavers* do windows. Além disto, esta extensão permite também distribuir projetos Scratch como ficheiros executáveis *stand-alone*.

2.2.2 Listas e ficheiros para Scratch

Este protótipo [Mon08] foi desenvolvido a partir do Scratch 1.2.1. Este protótipo introduziu de um novo tipo de dados baseado em *arrays*, permitindo a leitura e escrita desse novo tipo de dados para ficheiro. Adicionalmente, permite que projetos que acedam a ficheiros sejam compilados em ficheiros executáveis *stand-alone* no ambiente Windows para permitir a sua partilha por diferentes pessoas. Os ficheiros que representam as listas são lidos/escritos para a mesma pasta onde o código fonte do projeto está guardado, permitindo ao utilizador compartilhar o seu projeto com os ficheiros de lista associados. Esta extensão já não se encontra em desenvolvimento. Isto acontece poque as novas atualizações do ambiente Scratch

suportam estas funcionalidades.

2.2.3 Elements

Elements [Mon09b] é uma interface gráfica para a linguagem de programação Smalltalk-80 inspirada no Scratch. Oferece uma interface *drag/drop* de tijolos que representam código, muito parecida com o Scratch. Estes tijolos podem ser acumulados e montados como peças de LEGO em construções de programação complexas. O projeto Elements pode ser uma ótima ferramenta para fins educativos (ensino de Smalltalk) e também para descobrir se e como o *design* dos blocos Scratch pode ser aplicado, ou não, a mais campos profissionais e, neste caso, a um ambiente de programação orientada a objetos.

2.2.4 BYOB - "*Build Your Own Blocks*"

BYOB [Mon09a] é uma extensão para o Scratch que atualmente se encontra na sua versão 3.1.1 (19 Maio 2011) e que permite ao utilizador construir blocos personalizados. Estes blocos personalizados podem ser baseados em blocos predefinidos do Scratch ou em outros blocos criados pelo utilizador, e são definidos para um **Sprite** específico. Além disto estão presentes importantes novos mecanismos: definição de procedimentos e funções, passagem de parâmetros, variáveis locais de procedimentos/funções, recursividade e atomicidade. Este importante projeto já lançou três versões. Abaixo segue uma descrição das funcionalidades que cada versão trouxe para a mesa. Embora a versão atual 3.1.1 encapsule as funcionalidades dos seus antecessores, é importante mostrar a evolução deste projeto, pois é um dos projetos paralelos mais importantes do Scratch.

BYOB 1.0 A primeira versão do BYOB é baseada em Scratch 1.3. e permite a definição de procedimentos (blocos de comando), funções (blocos **Reporter**)

e funções booleanas. Permite a definição de parâmetros e variáveis globais e também suporta recursividade na medida em que é possível usar o bloco que está a ser definido na sua própria definição. Oferece também controlo de atomicidade do bloco personalizado, duplicação de blocos personalizados através de *drag/drop* entre **Sprites** ou por clonagem de **Sprites** e por fim leitura/escrita de um projeto Scratch ou **Sprite** para um ficheiro.

BYOB 2.0 A segunda versão do BYOB já é baseada na versão mais recente do Scratch, o Scratch 1.4. O objetivo do seu lançamento foi melhorar a experiência de criar blocos personalizados e introduzir a noção de objetos compostos no Scratch, bem como eliminar algumas das limitações da versão anterior. É possível abrir/importar qualquer projeto Scratch ou **Sprite**, os argumentos podem ser números ou texto (e blocos **Reporter**), clicar duas vezes num bloco **Reporter** personalizado mostra o seu resultado, melhoramento da funcionalidade *drag/drop* de blocos personalizados, funções de depuração introduzidas (blocos com erros mostrados a vermelho), correção do problema de terminar um ciclo infinito atômico, o editor do bloco é redimensionável. Características sobre **Sprites** aninhados: possibilidade de criar **Sprites** que consistem em sub-**Sprites** (composição), **Sprites** podem ser aninhados infinitamente, sub-**Sprites** seguem o movimento,orientação e efeitos gráficos do **Sprite** a que pertencem, sub-**Sprites** podem optar por seguir a rotação do **Sprite** a que pertencem, ou podem optar por rodar de forma independente. Outras características: é possível compartilhar **Sprites** e **Sprites** aninhado numa rede *mesh*, o compilador incluído permite converter um projeto do Scratch/BYOB para um exe, *autoscrolling*, *scrolling* por arrastamento (*scroll* ativado quando se arrasta um elemento para as bordas de uma janela como o editor de blocos), desfazer a ultima ação. Nesta versão a interface gráfica (GUI) Elements [Mon09b] foi integrada, permitindo a

inspeção do código dos blocos Scratch.

BYOB 3.0 É baseado em Scratch 1.4 e muitos recursos novos foram introduzidos, com uma forte influência vinda do Scheme. Novos recursos: biblioteca de funções de ordem superior, procedimentos e *closures* lambda (blocos anónimos personalizados, funções de primeira classe), listas de primeira classe, listas dinâmicas anónimas, listas de listas (permitindo a possibilidade de construção de estruturas de dados, tais como árvores e outros), listas de **Scripts** e blocos, procedimentos locais e globais, projetos com estado persistente, programação baseada em imagem como o Smalltalk, blocos e **Scripts** podem ser usados como dados, verificação de tipos (*typechecking*).

2.2.5 Outros *mods*

Existem muitas extensões do Scratch disponíveis e em desenvolvimento com muitos novos blocos e muitos recursos novos criados. Os *mods* mais relevantes, disponíveis e ativos são o Streak, Panther, Bingo, Slash e The Ultimate Gallery. Referência especial para o Panther, devido à sua funcionalidade CYOB *Code Your Own Blocks*. Esta permite que os utilizadores com conhecimentos mais avançados de Squeak codifiquem os seus próprios blocos no ambiente. Os blocos personalizados são escritos numa janela que pode ser acedida a partir da parte inferior do painel de blocos da categoria de variáveis clicando em **Make a new block**.

2.3 Outros ambientes e linguagens visuais

Nesta secção serão descritos alguns dos ambientes de programação visual mais importantes para o ensino.

2.3.1 Comikit

O ComiKit [Kin05] é um toolkit de programação para crianças, desenvolvido em Squeak, e é baseado na linguagem visual de *comics*, banda desenhada. Este ambiente permite que as crianças criem os seus próprios jogos interativos e brinquedos. Em ComiKit, um programa é criado desenhando figuras de personagens em *comics* associando-lhes ações e comportamento através de tiras visuais que representam eventos. Uma personagem pode ter uma ou várias imagens, e os eventos podem ser usados para alterar a imagem de uma personagem como resposta a uma determinada ação.
A integração de sinais simbólicos no contexto das representações icónicas de objetos de domínio, é uma técnica de apresentação poderosa usada nos *comics* e que poderia também ser usada para projetar e representar programas de um modo visual e expressivo. O Comikit é usado também por professores para ensinar matemática de uma forma interativa aos alunos.

2.3.2 Etoys

O Etoys [BPT08] é um ambiente computacional orientado a objetos baseado em Squeak e é usado como veículo de ensino em todo o mundo. É um ambiente de criação, com um modelo de objetos simples e poderoso para muitos objetos diferentes. Corre em diferentes plataformas e é *open source*. Inclui gráficos 2D e 3D, imagens, texto, partículas, apresentações, páginas web, vídeos, som e MIDI, etc. Etoys fornece uma interface onde o utilizador deve arrastar e soltar código que manipula diretamente objetos visuais. Todos os objetos visuais têm uma representação gráfica. Inclui a capacidade de partilhar *desktops* com outros utilizadores Etoys em tempo real, e muitas formas de aprendizagem e jogo podem ser feito através da internet. Todos os objetos podem ser criados e utilizados em todos os

lugares. Assim, implementa objetos integrados e é baseado na idéia de entidades virtuais programáveis. O Etoys é uma grande ferramenta de programação visual e foi uma grande influência na criação do ambiente Scratch.

2.3.3 Alice

O ambiente de programação Alice [DCP05] é uma linguagem livre de programação visual orientada a objetos. Foi desenvolvida pela Universidade de Carnegie Mellon com um ambiente de desenvolvimento integrado escrito em Java e é utilizada em vários cursos de programação introdutórios.

A linguagem é altamente visual e as instruções do programa traduzem-se diretamente numa animação 3D. Alice é composto por um motor de gráficos 3D e uma grande biblioteca de animação de modelos 3D. Neste ambiente, os programas são construídos com uma interface visual *drag-and-drop*. O utilizador seleciona animações a partir de uma lista de comandos disponíveis montando-os numa sequência desejada. A linguagem de programação fornece equivalentes para todos o fluxos padrão de estruturas de controlo.

Um programa Alice é um guião de um filme para personagens no ecrã. Estas personagens têm um conjunto de comandos pré-determinados parametrizados (por exemplo, *Move* (n) e *Turn* (n)) que controlam seu comportamento. Os utilizadores podem criar novas construções de comportamentos (sub-rotinas) combinando comandos existentes. O estado das personagens (por exemplo, a sua localização) pode ser inspecionado e modificado em tempo de execução.

A programação com Alice é altamente envolvente, reduz a carga sintática para programadores iniciantes com a sua interface *drag-and-drop* e mapeia muito bem o modelo de programação orientada a objetos. Todos os personagens em Alice são objetos encapsulados, cada um tendo informações (propriedades) e ações (métodos) associadas, e, além disto, o seu comportamento é controlado por envio e

receção de mensagens.
Por fim o Alice é uma ótima ferramenta para o ensino de programação orientada a objetos, dado que todos os conceitos da teoria orientada a objetos estão presentes na própria linguagem de programação.

2.3.4 GreenFoot

O Greenfoot [KÏ10] é um ambiente de desenvolvimento integrado que visa a aprendizagem e o ensino de programação. Foi desenvolvido na Universidade de Kent e na Universidade de Deakin, com o apoio da Sun Microsystems. É baseado no ambiente de desenvolvimento BlueJ e permite o desenvolvimento fácil de aplicações gráficas bidimensional, como simulações e jogos interativos. O Greenfoot é adequado ao ensino superior. Permite a visualização explícita de importantes conceitos de programação orientada a objetos e torna os conceitos de programação concretos e visuais, combinando assim um *output* gráfico interativo com programação em Java. Este ambiente consegue tudo isto através da criação de experiências tangíveis e da visualização explicita de conceitos de programação, orientando assim os utilizadores através das interações que envolvem manifestações concretas destes conceitos num ambiente 2D. Isto permite a introdução dos conceitos fundamentais em primeiro lugar, antes mesmo da necessidade de lidar com a sintaxe do código fonte.

2.3.5 JubJub

JubJub [Pas09] é um ambiente protótipo que foi inspirado por muitas características de corrente pedagógica das linguagens visuais existentes, bem como entrevistas com especialistas em diversas áreas: educação em ciências de computador, linguagens de programação visual, membros das equipas do Scratch, Alice, RoboLab e BlueJ.

O JubJub é um ambiente visual desenvolvido para permitir a criação de conjuntos de blocos de código personalizado. O objetivo é dar suporte a um conjunto diversificado de aspetos e fornecer uma arquitetura para uma interface de programação personalizável, expansível e icónica. No JubJub a programação é desenvolvida juntando blocos de instruções que representam ações de controlo como *while*, variáveis, classes e outros. O código que é desenvolvido também aparece numa forma textual através de uma janela lateral. Este recurso permite que o aluno faça a conexão entre a junção de blocos de instruções e a programação textual real.

2.3.6 Puck

Puck [Koh07] é uma linguagem de programação visual desenvolvida na Alemanha de acordo com os desejos dos professores na Universidade de Thuringia. O Puck utiliza um sistema visual de programação para evitar erros de sintaxe. Este sistema é baseado em blocos de instruções que representam código. Pode gerar código Java, Oberon-2 ou pseudocódigo que prepara os alunos para a programação textual. Os projetos Puck podem ser executados em qualquer etapa durante a criação.
Em Puck um programa é feito combinando blocos de instruções num programa de visual. Estes blocos podem representar diferentes tipos de dados como variáveis de diferentes tipos (booleanos e inteiros) e procedimentos com parâmetros. Além disso, foi implementado um método de medição de complexidade para comparar as diferentes soluções. Isto permite que muitos conceitos e algoritmos que só poderiam ser ensinados com linguagens baseadas em texto, agora podem ser ensinados com a ajuda de Puck, sem a preocupação com sintaxe.

2.4 Conclusão

Neste capítulo foi apresentada a linguagem visual e interface do ambiente Scratch, suas funcionalidades originais e mais relevantes extensões atuais. Foram também apresentados alguns ambientes de programação visual direcionados ao ensino que são relevantes hoje em dia.

Capítulo 3

O Ambiente Scratch

Neste capítulo será explorado ao pormenor o ambiente visual Scratch e todos os seus componentes bem como a sua linguagem visual de programação.

Em primeiro lugar serão explorados os vários constituintes da interface gráfica do ambiente Scratch e suas funcionalidades na secção 3.1.

Em seguida, na secção 3.2, será apresentado o processo de programação visual implementado no Scratch através da sua linguagem visual. Nesta secção também será abordado o modelo de objetos e concorrência do Scratch e os vários tipos de blocos e de dados existentes

Por fim, serão expostos também alguns conceitos de programação importantes e possíveis de representar na sua linguagem visual, o que será feito de uma forma simples e direta na secção 3.3 através de uma tabela.

3.1 Interface do ambiente

A programação visual em Scratch é baseada na metáfora do bloco *LEGO*. Os blocos de instruções encaixam-se noutros blocos compatíveis criando agrupamentos de blocos chamados **Scripts** que representam comportamentos, geralmente animações.

Um projeto Scratch é composto por um palco ou **Stage** e por objetos animados chamados **Sprites**. Um **Sprite** é uma imagem bidimensional que é integrada numa cena maior. A aparência de um **Sprite** pode ser alterada, alterando o seu traje ou **costume**. Um traje pode ser predefinido do Scratch, importado como uma imagem ou então pode ser construído no editor de pintura existente no ambiente. No Scratch é possível criar comportamentos para os **Sprites** ou para o **Stage** através da combinação dos blocos visuais que representam instruções. Na figura 3.1 está representada a interface do ambiente Scratch e os seus componentes de maior relevância.

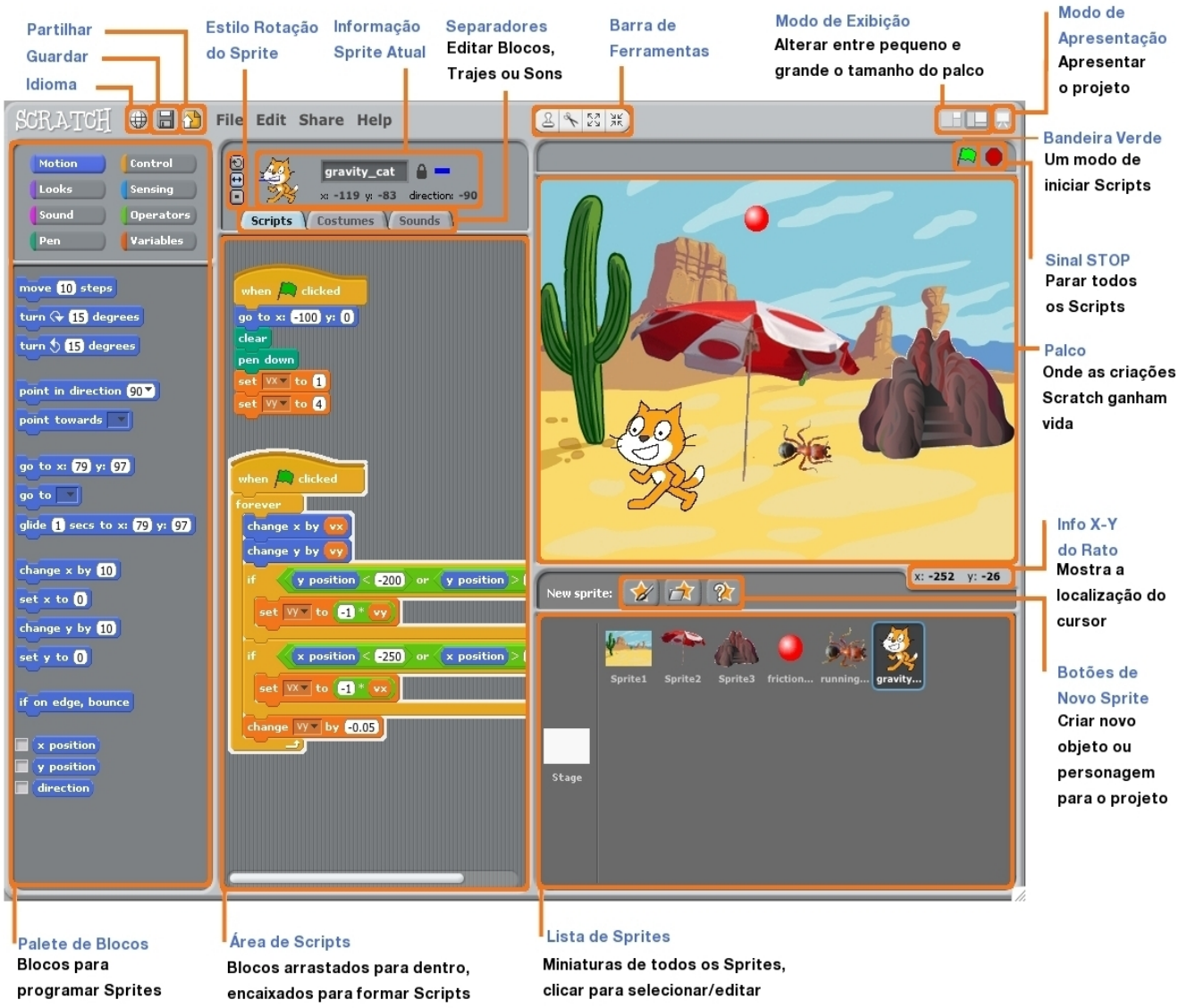


Figura 3.1: Interface do Scratch

3.1.1 Menu

O menu do Scratch, representado na figura 3.2, oferece variadas funcionalidades importantes do ambiente Scratch.

Figura 3.2: Menu

- O primeiro ícone do menu, que tem a forma de um globo, permite mudar a linguagem do ambiente para outra predefinida.
- O ícone com a forma de uma disquete permite gravar o projeto que está a ser desenvolvido.
- O ícone de cor amarela permite partilhar o projeto desenvolvido com outros utilizadores no site `http://scratch.mit.edu/`.
- O submenu **File** permite criar um projeto novo, abrir um projeto existente ou salvar o projeto atual. Além disto também permite exportar **Sprites** como ficheiros, adicionar notas ao projeto e possibilita a importação de **Sprites** de outros projetos para o projeto em curso.
- O submenu **Edit** permite recuperar o último objeto que foi apagado, é possível comprimir o tamanho de imagens e sons em projetos muito grandes podendo com isto existir alguma redução de qualidade. Permite também mostrar na categoria de blocos **Motion** de movimento, os blocos **Motor**. Por fim, oferece também uma funcionalidade que permite visualizar um programa Scratch executando um passo de cada vez, em que cada bloco é destacado enquanto está a ser executado. Também é possível controlar a velocidade de cada passo e do seu destaque. Esta funcionalidade é útil para depuração dos programas criados.

- O submenu **Share** permite partilhar um projeto no site do Scratch e também fornece um *link* direto ao site http://scratch.mit.edu/.

- O submenu **Help** fornece acesso a uma página de *links* para materiais de referência, tutorais e perguntas frequentes. No menu também se encontram disponíveis uma página com as janelas de ajuda do Scratch.

3.1.2 Barra de ferramentas

A barra de ferramentas, representada na figura 3.3, oferece várias funcionalidades de acordo com o botão clicado.

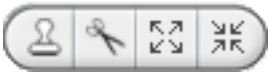


Figura 3.3: Barra de ferramentas

- O ícone carimbo, primeiro da esquerda, permite duplicar **Sprites**, trajes, cenários, sons, blocos e **Scripts**. Carregar no objeto a duplicar juntamente com a tecla *shift* impede que a função de duplicação se desative, sendo assim possível continuar a duplicar objetos sem ser necessário carregar novamente no ícone carimbo.

- O ícone tesoura, à direita do carimbo, permite apagar **Sprites**, trajes, cenários, sons, blocos e **Scripts**. Carregar no objeto a eliminar juntamente com a tecla *shift* impede que a função de apagar se desative, sendo assim possível continuar a apagar objetos sem ser necessário carregar novamente no ícone tesoura.

- O ícone com quatro setas apontadas para fora, à direita da tesoura, permite aumentar o tamanho dos **Sprites** em palco. Ao carregar no **Sprite** junta-

mente com a tecla *shift* o tamanho deste aumenta com maiores incrementos por clique.

- O ícone com quatro setas apontadas para dentro, primeiro ícone à direita, permite diminuir o tamanho dos **Sprites** em palco. Ao carregar no **Sprite** juntamente com a tecla *shift* o tamanho deste diminui com maiores incrementos por clique.

3.1.3 Palco e ferramentas

O palco ou **Stage** do ambiente Scratch, representado na figura 3.4, é um espaço de formato retangular de largura 360 unidades e comprimento 480 unidades, situando-se no lado direito da interface.

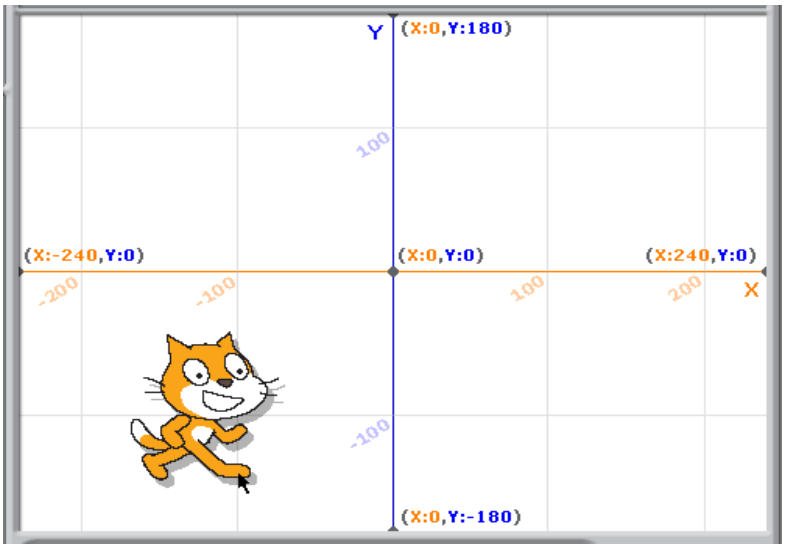


Figura 3.4: Palco do Scratch

O **Stage** é o espaço onde os **Sprites** animados são apresentados, podendo interagir entre si e ter um comportamento próprio programado previamente pelo utilizador. Deste modo o **Stage** é o local onde as animações criadas são exibidas. Como se pode constatar na figura 3.4 a posição de um **Sprite** no **Stage** é definida pelos

eixos representados (x , y). Estes cruzam-se no centro referencial do **Stage** e podem tomar os valores dos intervalos $-240 \leq x \leq +240$ e $-180 \leq y \leq +180$.
Por baixo do **Stage** à direita, é possível visualizar a cada momento as coordenadas do cursor no **Stage**. Esta área é apresentada na figura 3.5.

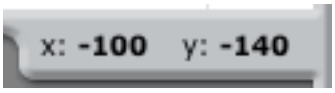


Figura 3.5: Posição do rato no **Stage**

Por cima do **Stage**, à direita, encontram-se os botões Bandeira verde e Sinal de STOP.

(a) Bandeira verde (b) Sinal de STOP

Figura 3.6: Bandeira verde e Sinal STOP

- O botão Bandeira verde, representado na figura 3.6a, permite iniciar vários **Scripts** ao mesmo tempo. Quando este botão é carregado todos os **Scripts** iniciados pelo bloco **When "green flag"clicked** começam a sua execução.
- O botão Sinal STOP, representado na figura 3.6b, permite parar todos os **Scripts** de todos os **Sprites** que estão atualmente em execução.

Por cima do **Stage** superiormente aos botões de Bandeira verde e Sinal de STOP encontram-se os botões de modo de exibição e o botão de apresentação.

Figura 3.7: Botão de apresentação

O botão de apresentação representado na figura 3.7 permite ver os projetos animados no ecrã inteiro em modo de apresentação, representado na figura 3.8. Para voltar ao modo normal do ambiente basta clicar na tecla *Esc*.

Figura 3.8: Modo de apresentação

Os botões de exibição representados na figura 3.9 permitem mudar o tamanho do **Stage**. O botão esquerdo torna o **Stage** mais pequeno aumentando assim a Área de **Scripts** e o botão direito torna o **Stage** maior e diminui o tamanho da Área de **Scripts**.

Figura 3.9: Botões de modo de exibição

3.1.4 Lista de Sprites

A Lista de **Sprites**, representada na figura 3.10, mostra miniaturas de todos os **Sprites** de um projeto, em que os nomes dos **Sprites** aparecem na parte inferior da miniatura correspondente. Além disto, a Lista de **Sprites** apresenta também uma miniatura para o **Stage** sempre visível do lado esquerdo da sua área. Para

Figura 3.10: Lista de **Sprites**

criar ou editar **Scripts**, trajes e sons de um **Sprite** basta clicar na sua miniatura. Ao selecionar um **Sprite** da lista, a Área de **Scripts** abre na aba de **Scripts** contendo esta os agrupamentos de blocos ou **Scripts** que esse **Sprite** já contém. Esta área encontra-se por baixo do painel de informação, representado na figura 3.14, que apresenta informações relativas ao **Sprite** específico selecionado. Caso se queira editar os seus trajes ou sons basta clicar nas abas de **Costumes** e **Sounds** respetivamente.

Na Lista de **Sprites** clicando na miniatura relativa ao **Stage** é possível também visualizar, criar e editar os seus **Scripts** e sons da mesma maneira que para os **Sprites** normais. A diferença é que o **Stage** em vez de ter trajes como os **Sprites** tem cenários ou **Backgrounds**, os quais também podem ser alterados e criados clicando na aba **Backgrounds**.

Botões de novo Sprite

Os **Sprites** dos projetos Scratch são criados pelo utilizador através dos botões de novo **Sprite** representados na figura 3.11. Existem três maneiras de criar um novo **Sprite** que correspondem a cada um dos três botões representados na figura 3.11.

Figura 3.11: Botões de novo **Sprite**

- O botão da esquerda permite ao utilizador pintar um novo traje para um novo **Sprite** através do editor de desenho representado na figura 3.12. O editor de desenho permite desenhar novos trajes e cenários e oferece variadas ferramentas de criação e edição de imagem.

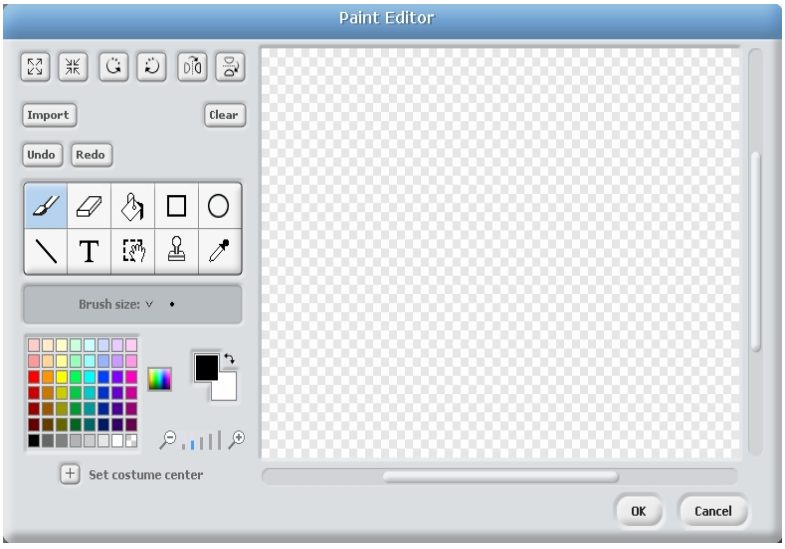


Figura 3.12: Editor de desenho

- O botão central permite ao utilizador importar um novo traje para um novo **Sprite** a partir de uma imagem ou, alternativamente, importar um **Sprite** completo já criado. As importações são feitas através de uma janela de importação como a da figura 3.13.

Figura 3.13: Janela de importação

- O botão mais à direita cria um **Sprite** com um traje surpresa importado dos trajes dos arquivos Scratch.

3.1.5 Painel de informação do Sprite atual

O painel representado na figura 3.14 fornece variadas informações sobre o **Sprite** selecionado na Lista de **Sprites** 3.10.

Figura 3.14: Painel de Informação do **Sprite** atual

- Mostra uma miniatura da imagem do **Sprite** com um traço azul sobreposto mostrando a direção para a qual este aponta;
- Mostra o seu nome numa caixa de texto, sendo possível a sua alteração;
- Mostra o valor das suas coordenadas no **Stage** e o angulo da sua direção atual;
- Mostra também, em frente ao nome do **Sprite**, um cadeado. Este cadeado se estiver aberto significa que em modo de apresentação o utilizador pode alterar a posição do **Sprite** com o rato. Por outro lado caso esteja fechado a posição do **Sprite** não pode ser alterada através do rato no modo de apresentação.
- Do lado esquerdo encontram-se os botões que definem o estilo de rotação do **Sprite**. Com o botão superior selecionado o traje roda à medida que o **Sprite** muda de direção. Com o botão do meio selecionado o traje do **Sprite** apenas se vira para a esquerda ou para a direita. E com o botão inferior selecionado o traje nunca roda, mesmo que o **Sprite** mude a sua direção.

3.1.6 Trajes

Clicando na aba **Costumes** é possível aceder à Área de trajes de um **Sprite** representada na figura 3.15. Aqui é possível ver os trajes associados ao **Sprite** atualmente selecionado na Lista de **Sprites**. Além disto o traje que está presentemente a ser usado por esse **Sprite** também é apresentado selecionado sendo possível clicar noutro traje tornando-o no novo traje atual dso **Sprite**.

Figura 3.15: Aba **Costumes**

Nesta Área de trajes e relativamente a cada traje o utilizador pode também:

- Ver e alterar o seu nome através de uma caixa de texto;
- Duplicá-lo através do botão **Copy**;
- Editá-lo através do botão **Edit**;
- Eliminá-lo através do botão de cruz.

É possível também criar novos trajes para o **Sprite** selecionado. E para este efeito o utilizador tem várias opções que são:

- Pintar um novo traje no editor de desenho 3.12 através do botão **Paint**;
- Importar uma imagem do disco com o botão **Import** através de uma janela de importação 3.13;
- Tirar uma fotografia com a webcam com o botão **Camera** criando com esta um novo traje;
- Arrastar imagens do *desktop* ou da *web* diretamente para a Área de trajes.

3.1.7 Cenários

Tal como os **Sprites** têm trajes que são a sua aparência, o **Stage** tem cenários que também podem ser criados pelo utilizador, editados e trocados. Isto é feito na Área de cenários que é representada na figura 3.16.

Figura 3.16: Aba **Backgrounds**

Para aceder à Área de cenários do **Stage** basta que na Lista de **Sprites**, 3.10, o **Stage** seja selecionado e em seguida clicar na aba **Backgrounds**. Todas as funcionalidades para a Área de trajes descritas nas secção Trajes 3.1.6 são também válidas para a Área de cenários. A diferença é que os trajes são referentes a **Sprites** e os cenários são referentes ao **Stage**.

3.1.8 Sons

Clicando na aba **Sounds** é possível aceder à Área de sons de um **Sprite** ou do **Stage** representada na figura 3.15. Aqui é possível ver os sons associados ao **Stage** ou ao **Sprite** atualmente selecionado na Lista de **Sprites**.

Figura 3.17: Aba **Sounds**

Nesta Área de sons e relativamente a cada som o utilizador pode:

- Ver e alterar o seu nome através de uma caixa de texto;
- Ouvi-lo através do botão *play* com o símbolo triangular;
- Parar a sua audição através do botão *stop* com o símbolo de um quadrado;

- Ver o seu tamanho e duração;
- Eliminá-lo através do botão de cruz.

É possível também criar novos sons para o **Stage** ou para o **Sprite** selecionado. E para este efeito o utilizador tem duas opções que são:

- Gravar um novo som carregando no botão **Record** que abre o gravador de sons representado na figura 3.18;
- Importar um som do disco com o botão **Import** através de uma janela de importação.

Figura 3.18: Gravador de Sons

3.1.9 Scripts de blocos

Clicando na aba **Scripts** é possível aceder à Área de **Scripts** de um **Sprite** ou do **Stage**, representada na figura 3.19. Aqui é possível ver e criar os **Scripts** associados ao **Stage** ou ao **Sprite** atualmente selecionado na Lista de **Sprites**. Aqui os **Scripts** são criados juntando blocos arrastados a partir da Paleta de blocos, representada na figura 3.20. Estes são encaixados uns nos outros formando **Scripts** de instruções que determinam o comportamento animado dos **Sprites** e do **Stage**.

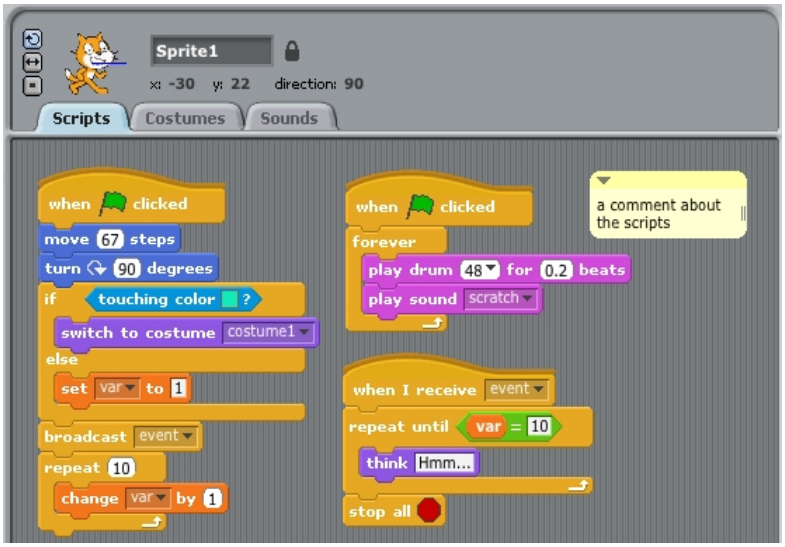


Figura 3.19: Aba **Scripts**

Para perceber o que faz um bloco específico o utilizador pode aceder à janela de ajuda clicando no botão direito do rato (em mac: *ctrl* + clique) em cima do bloco e selecionando **help**. Isto mostra uma janela de informações e um exemplo de uso desse bloco.

É possível também salvar uma foto de todas os **Scripts** presentes na área, adi-

cionar uma caixa na qual se podem escrever comentários relevantes e também organizar automaticamente os blocos da área toda através de um menu que é acedido clicando no botão direito do rato (em mac: *ctrl* + clique) numa área livre da Área de **Scripts**.

Quando um bloco é arrastado na Área de **Scripts** por cima de um bloco ou de um **Script**, uma linha branca indica onde esse bloco pode ser encaixado, esta linha é visível na figura 3.21. Nesta área é possível mover um **Script** inteiro e para isso basta pegar-lhe a partir do bloco de topo. Para pegar apenas numa parte deste **Script** basta pegar num bloco inferior ao de topo o que faz com que os blocos inferiores a este também sejam arrastados.

Para executar um **Script** o utilizador deve aplicar um duplo clique em qualquer lugar deste **Script**, com isto as instruções presentes neste **Script** são executadas uma a uma e de cima para baixo. Por fim para copiar um **Script** de comandos para outro **Sprite** deve-se arrastar esse **Script** para cima da miniatura do outro **Sprite** na Lista de **Sprites** 3.10.

3.1.10 Paleta de blocos

A Paleta de blocos, representada na figura 3.20, funciona como uma biblioteca contendo todos os blocos que podem ser utilizados. A partir deste componente é possível arrastar blocos para a Área de **Scripts**, 3.19, com o objetivo de criar **Scripts** de blocos e assim programar as animações para os **Sprites** ou para o **Stage**. Na Paleta, os blocos encontram-se divididos por categorias, tendo todos os blocos de uma categoria a sua cor própria. Para ver todos os blocos de uma determinada categoria o utilizador pode carregar em cada um dos botões de categoria que se encontram no cimo da figura 3.20. Existindo oito painéis de categorias de blocos apresentados na paleta:

- **Motion**
- **Looks**
- **Sound**
- **Pen**
- **Control**
- **Sensing**
- **Operators**
- **Variables**

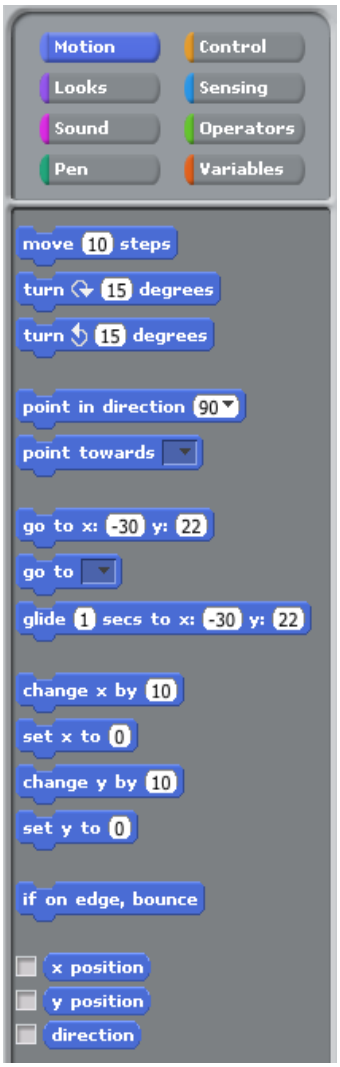


Figura 3.20: Paleta de blocos

3.2 Programar em Scratch

3.2.1 Categorias dos blocos Scratch

- **Motion**: Movimento - Estes blocos de cor azul escura são usados para controlar o movimento dos **Sprites**. Desde movimento numa direção um determinado número de passos, rotação, deslizar para um determinado ponto até saber as coordenadas do **Sprite** no **Stage** e editá-las. Como estes blocos controlam movimento dos **Sprites** não existe nenhum bloco desta categoria disponível para o **Stage**.

- **Motor**: Motores - Estes blocos têm também uma cor azul escura como os blocos de movimento, mas diferenciam-se destes devido a que a sua cor é mais escura que os restantes. Estes blocos de início não se encontram ativos no Scratch. Para serem apresentados necessitam ser ativados no menu **Edit**, e são apresentados no mesmo painel da categoria de blocos de movimento. Estes blocos são usados para programar um motor LEGO ligado ao computador e funcionam com o kit LEGO @ Education WeDo http://www.legoeducation.com.

- **Looks**: Aparência - Estes blocos de cor roxa controlam a aparência dos **Sprites** e do **Stage**. Controlam, relativamente a **Sprites**, as mudanças de traje, tamanho, balões de fala, cor e visualização relativa a outros **Sprites**. E controlam, relativamente ao **Stage**, as mudanças de cenário e cor.

- **Sound**: Som - Estes blocos de cor rosa controlam a reprodução de sons presentes na Área de sons 3.17, o volume e ritmo do som, e permitem criar musica com vários instrumentos e batidas disponíveis. Estes blocos são comuns ao **Stage** e **Sprites**.

- **Pen**: Caneta - Estes blocos de cor verde escura controlam o desenho no **Stage** através do controlo de uma "caneta", sua espessura, tom e cor. O **Sprite** desenha no **Stage** com o seu movimento através dele. O único bloco de Caneta do **Stage** é o **clear** que permite limpar completamente o **Stage** de desenhos feitos previamente.

- **Control**: Controlo - Estes blocos de cor amarela oferecem estruturas de controlo para os **Scripts**, como ciclos e condições. Permitem o envio e receção de eventos criados pelo utilizador e ainda o início de **Scripts** através de blocos específicos para essa função: os blocos **Hat**. Os blocos desta categoria são essenciais a qualquer projeto porque permitem controlar o fluxo de processamento dos diferentes **Scripts** que representam comportamentos. Estes blocos são comuns ao **Stage** e aos **Sprites**.

- **Sensing**: Sensores - Estes blocos de cor azul clara detetam eventos ou propriedades específicas, por exemplo, reportar se um determinado botão está premido ou se um determinado **Sprite** tocou numa determinada cor. Permitem implementar um temporizador, fazer uma pergunta ao utilizador no **Stage** e obter resposta, e outras. Existem quatro blocos desta categoria que estão disponíveis para os **Sprites** mas não para o **Stage**. Estes blocos são blocos que detetam contacto e distância de um **Sprite** com outros **Sprites** ou com cores específicas no palco.

- **Operators**: Operadores - Estes blocos de cor verde clara executam operações tais como operações booleanas, manipulação de *strings* e funções matemáticas como soma, raízes quadradas, arredondamento e módulo. Todos estes blocos são comuns ao **Stage** e aos **Sprites**.

- **Variables**: Variáveis - Estes blocos permitem criar variáveis que permitem guardar valores numéricos ou *strings*. Os blocos relativos a variáveis e sua

manipulação são de cor laranja claro. Todos estes blocos são comuns ao **Stage** e **Sprites**. O blocos relativos a listas são considerados uma diferente categoria de blocos das variáveis, mas são expostos no mesmo painel na Paleta de blocos. Todos estes blocos são comuns ao **Stage** e aos **Sprites**.

- **List**: Listas - Estes blocos permitem criar listas que permitem guardar valores numéricos ou *strings* em forma de lista e permitem a sua amostragem no **Stage** através de um visualizador de listas. Os blocos relativos a listas e sua manipulação são de cor laranja escuro. Estes blocos apesar de serem uma diferente categoria de blocos, são expostos no mesmo painel das variáveis na Paleta de blocos. Todos estes blocos são comuns ao **Stage** e aos **Sprites**.

3.2.2 Tipos dos blocos Scratch

- **Stack** - Os blocos **Stack**, também referidos como blocos **Command**, são os mais comuns do Scratch, realizam os comandos principais e podem ser encontrados em qualquer categoria de blocos exceto nos **Operators**. Estes blocos assemelham-se a peças de um puzzle devido a possuírem ranhuras na parte superior podendo (ou não) também ter encaixes na parte inferior. Este formato permite que outros blocos **Stack** possam ser encaixados por cima ou por baixo destes (caso possuam encaixes inferiores). Assim podem formar **Scripts** que se podem estender muito. Existem setenta e sete blocos **Stack** sendo que para um **Script** ser funcional necessita de pelo menos um bloco **Stack** na sua constituição.

Figura 3.21: **Script** de blocos **Stack**

Dentro deste tipo de blocos existem os blocos **C-shaped**, também conhecidos por blocos **Wrap** ou blocos **Control Structure**. Um exemplo deste tipo de blocos encontra-se na figura 3.22.

Figura 3.22: Bloco **C-shaped** com três blocos aninhados na sua cavidade

Estes blocos são usados para implementar ciclos e para testar condições. Podem ter um único encaixe no topo e têm, em geral, a forma de um 'C' para que seja possível encaixar outros blocos **Stack** nas suas cavidades. Existem seis blocos **C-shaped** encontrando-se todos na categoria **Control**.

Também dentro desta categoria dos blocos **Stack** existem os blocos **Cap** que estão representados na figura 3.23. Estes são os blocos responsáveis por terminar a execução de **Scripts**. Têm ranhuras na parte superior e a parte inferior é lisa sem encaixes para que não seja possível encaixar outros blocos por baixo deles.

Figura 3.23: Blocos **Cap**

Estes blocos são utilizados para parar um **Script** de blocos numa determinada altura ou parar todos os **Scripts** de um projeto. Existem dois blocos **Cap** e encontram-se na categoria **Control**.

- **Hat** - Os blocos **Hat**, também conhecidos por blocos **Trigger**, são os blocos que iniciam todos os **Scripts** e deste modo são essenciais para a criação de projetos. Têm uma forma redonda na sua parte superior e na parte inferior tem um encaixe para se unirem com outros blocos.

(a) **Hat** - bandeira verde

(b) **Hat** - tecla

(c) **Hat** - evento

(d) **Hat** - **Sprite**

Figura 3.24: Os blocos **Hat**

Sem os blocos **Hat** só seria possível iniciar **Scripts** manualmente através de um duplo clique. Assim é possível definir quando cada **Script** inicia a sua execução automática. Com estes blocos é possível implementar programação baseada em eventos. Existem quatro blocos **Hat** que se encontram todos na categoria **Control** e estão representados na figura 3.24. Os blocos definem quatro maneiras diferentes de iniciar a execução de um **Script**:

  - o bloco 3.24a inicia a execução quando a Bandeira verde 3.6a é clicada;
  - o bloco 3.24b inicia a execução quando uma determinada tecla é premida;
  - o bloco 3.24d inicia a execução quando o **Sprite** ou o **Stage** são clicados;
  - o bloco 3.24c inicia a execução quando um determinado evento é transmitido.

- **Reporter** - Os blocos **Reporter**, também conhecidos por blocos **Function**, reportam valores e são desenhados para encaixar nas área de parâmetro de outros blocos. Para visualizar o valor retornado por um bloco **Reporter** basta ao utilizador clicar na área interior deste. Existem trinta e nove blocos **Reporter** predefinidos e podem ter duas formas distintas: formato arredondado 3.25a ou formato hexagonal 3.25b com pontas pontiagudas.

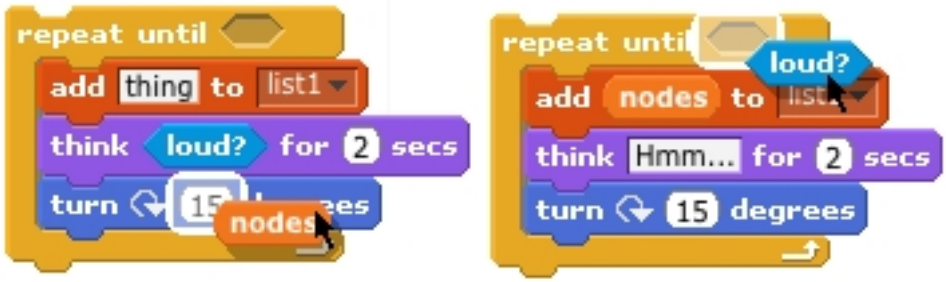
(a) Arredondado (b) Hexagonal

Figura 3.25: Formatos dos blocos **Reporter** com caixa de seleção

Os blocos de formato arredondado 3.25a reportam valores numéricos ou *strings*. Estes blocos arredondados encaixam em blocos com áreas de parâmetro arredondadas ou retangulares, como representado na figura 3.26, e são usados sempre que um **Script** precisa de um valor numérico ou de uma *string*. Usam-se para efetuar equações reportando os resultados destas e também para reportar valores de variáveis que podem ser predefinidas do sistema ou criadas pelo utilizador. Existem vinte e seis blocos **Reporter** arredondados, não contando com os correspondentes às muitas variáveis que podem ser criadas pelo utilizador que também são representadas neste formato 3.32.

Os blocos de formato hexagonal 3.25b, também conhecidos como blocos **Boolean** reportam para uma condição os valores booleanos **True** ou **False**. Estes blocos de pontas pontiagudas encaixam em blocos com áreas de parâmetro hexagonais ou retangulares, como representado na figura 3.26. São usados em alguns blocos **C-shaped** sempre que uma condição é necessária

para começar ou terminar a sua execução. Também são usados no bloco **wait until**<> que espera que a condição representada por um bloco **Reporter** exagonal se verifique para continuar a execução do **Script** em que está contido. Existem treze blocos **Reporter** hexagonais ou **Boolean**.



(a) (b)

Figura 3.26: Blocos **Reporter** e seu encaixe noutros blocos

Alguns blocos **Reporter** têm uma caixa de seleção localizada ao seu lado esquerdo. Esta caixa se selecionada permite mostrar o valor do bloco **Reporter** no **Stage** através de uma janela de amostragem que pode ter três formatos diferentes representados na figura 3.27.

(a) (b) (c)

Figura 3.27: Os três formatos da janela de um **Reporter**

3.2.3 Tipos de dados

A linguagem Scratch tem três tipos de dados de primeira classe que são os tipos numérico, *string* e booleano. Estes três tipos podem ser retornados pelos blocos **Reporter** que representam funções. No Scratch o formato das áreas de parâmetro dos blocos indica o tipo de dados esperado, assim como o formato de um bloco **Reporter** indica o tipo de dados retornado por esse bloco. Existem três formatos de áreas de parâmetro mas só dois formatos de blocos **Reporter**: um para valores booleanos e outro para números e *strings*.
O Scratch só autoriza a inserção de um bloco **Reporter** numa área de parâmetro de outro bloco se os dois forem do mesmo formato ou se o formato da área de parâmetro for o formato retangular. Este formato retangular permite a inserção de blocos **Reporter** dos dois tipos expostos como é percetível na figura 3.26.
No caso das variáveis Scratch, estas são representadas por blocos **Reporter** arredondados podendo conter números ou *strings*. Devido a este facto, as variáveis definidas em Scratch não são tipadas. Isto evita que o utilizador tenha de especificar o tipo de uma variável quando é criada.

3.2.4 Modelo de objetos

Os **Sprites** são os objetos do Scratch. Estes podem ter as suas próprias variáveis e comportamentos representados nos seus **Scripts**. No Scratch não existe o mecanismo de classes nem de herança, tornando-o um ambiente baseado em objetos mas não orientado a objetos. Desta forma, os blocos podem apenas atuar no **Sprite** a que estão associados, tornando impossível a invocação de **Scripts** pertencentes a um **Sprite** por outro **Sprite** de uma forma direta, como pode acontecer nas linguagens orientadas a objetos.

Cada **Sprite** tem os seus próprios **Scripts** que definem todo o seu comportamento, tornando assim a linguagem mais fácil de entender e editar pelo utilizador. Isto evita que o utilizador tenha de procurar comportamentos que foram herdados de outras classes para perceber o comportamento de um determinado **Sprite**. Além disso, o facto de não existirem classes nem herança no Scratch também torna a edição de **Scripts** local ao **Sprite** onde estão presentes, evitando as complicações na edição de métodos que são herdados por outras classes, tal como acontece nas linguagens orientadas a objetos.

Assim, quando o utilizador tenta perceber o porquê de um **Sprite** se comportar de uma determinada maneira, ele apenas tem de analisar os **Scripts** que estão definidos na Área de **Scripts** 3.19 desse **Sprite** específico, isto porque todos o comportamentos possíveis desse **Sprite** estão aí definidos. Por outro lado, o facto de os **Sprites** serem auto-suficientes torna muito fácil a sua migração entre projetos diferentes. Como não existem dependências entre **Sprites** estes podem ser copiados individualmente entre projetos sem causar muitos problemas na sua adaptação. Este facto encoraja a componente de partilha e colaboração entre utilizadores do Scratch o que é, de facto, um dos seus principais objetivos.

Todo este mecanismo torna o desenvolvimento de projetos Scratch mais simples de compreender mas por vezes também implica mais trabalho para o utilizador. Por exemplo para criar muitos **Sprites** com o mesmo comportamento é necessário duplicar o original o número de vezes necessário, e para editar o seu comportamento é necessário editar os **Scripts** de cada um individualmente ou, então, editar num deles e duplicar tudo novamente, processo que se pode tornar trabalhoso.

3.2.5 Condições

No Scratch para implementar condições utilizam-se os blocos **Hat** 3.24 e os blocos **Control** 3.28, com áreas de parâmetro hexagonais, em combinação com os blocos **Boolean** 3.25b, os quais representam os parâmetros que encaixam nessas áreas e representam as condições específicas a testar.

Os blocos **Hat** iniciam **Scripts** caso a sua condição se verifique. Estas condições podem ser por exemplo: tecla **space** ser clicada ou um determinado evento seja recebido.

Os blocos **Control** que implementam condições são os blocos **wait until**<> e os blocos **C-shaped**: **forever if**<>, **if**<>, **if**<>**else** e **repeat until**<> que estão representados na figura 3.28.

Figura 3.28: Blocos **Control** que implementam condições

Como exemplo, na figura 3.29 está representado um bloco **if**<> com um bloco **Boolean** encaixado na sua área de parâmetro e um bloco **say[ ]** na sua cavidade. Este **Script** diz-nos que se o botão do rato estiver carregado o **Sprite** diz "Hello".

Figura 3.29: Bloco **if**<> preenchido

Além disso, estes blocos também podem aparecer aninhados formando condições dentro de outras condições. Por exemplo na figura 3.30 temos o exemplo de um bloco **if**<>**else** dentro de uma cavidade de um bloco **if**<>.

Figura 3.30: Bloco **if**<> com bloco **if**<>**else** aninhado

Este **Script** diz-nos que se o botão do rato estiver carregado, uma de duas coisas pode acontecer. Se o **Sprite** ao qual este **Script** pertence estiver a tocar na borda do **Stage** o **Sprite** diz "Better Turn!"caso contrário o **Sprite** diz "Hello!".

3.2.6 Ciclos

Para implementar ciclos em Scratch utilizam-se os blocos de ciclo da categoria **Control**: **forever if**<>, **forever**, **repeat until**<> e **repeat( )**. Estes blocos estão representados na figura 3.31.

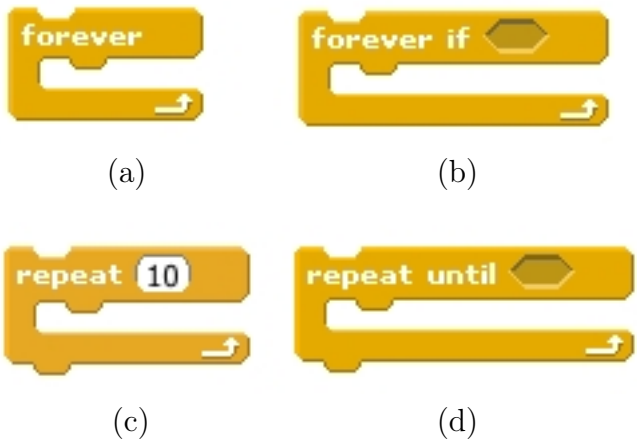


Figura 3.31: Os blocos de controlo **C-shaped** que implementam ciclos

Estes blocos permitem implementar quatro configurações para ciclos diferentes. Permitem repetir o conjunto de blocos inseridos na sua cavidade:

- um determinado número de vezes - com o bloco representado em 3.31c
- para sempre - com o bloco representado em 3.31a
- enquanto a sua condição específica é verdadeira - com o bloco representado em 3.31b
- até que a sua condição específica seja verdadeira - com o bloco representado em 3.31d

3.2.7 Variáveis

Em Scratch as variáveis 3.32 são representadas na Área de **Scripts** e na Paleta de blocos por blocos **Reporter** arredondados e são criadas e nomeadas apenas pelo utilizador. No **Stage** o seu valor pode ser apresentado através das janelas de visualização dos blocos **Reporter** 3.27.

Figura 3.32: Representação de uma variável

Estas variáveis podem ser locais ou globais. As variáveis globais podem ser lidas e editadas por qualquer **Sprite** ou pelo **Stage**. As variáveis locais apenas podem ser editadas pelo **Sprite** em que foram criadas, podem porém ser lidas também pelo **Stage** ou por outros **Sprites** através do bloco **Reporter** representado na figura 3.33, inserido na categoria **Sensing**.

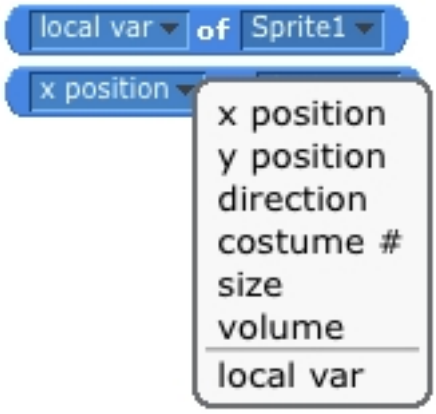


Figura 3.33: Bloco que acede a propriedades e variáveis locais de um **Sprite**

Este bloco, além de permitir o acesso ao valor de variáveis de outros **Sprites**, permite aceder a algumas das suas características como a sua posição no **Stage**, tamanho, etc. A definição desta característica global ou local das variáveis Scratch

é feita aquando da sua criação pelo utilizador através da janela 3.34, que é chamada carregando no botão **Make a Variable** do separador da categoria **Variables**.

Figura 3.34: Janela de criação de variável

Por omissão as variáveis são criadas como globais com a opção **For all sprites** selecionada como na figura 3.34, podendo o utilizador marcar a opção **For this sprite only** para criar uma variável local ao **Sprite** atual. Estas variáveis são representadas por blocos **Reporter** arredondados e por isso podem conter números ou *strings*. São usadas em projetos sempre que um valor tem ser guardado para ser usado mais tarde, como por exemplo um nome ou um valor numérico relevante. Podem também ser usadas como valores booleanos, usando apenas os valores 1 e 0 para representar **True** e **False**.

A edição de uma variável é feita através de dois blocos, sendo que um deles é bloco **set [▼] to [ ]** que é o bloco de atribuição de um valor numérico ou de *string* dado pelo utilizador. E o outro é o bloco **change [▼] by ( )** que incrementa o valor da variável por um valor numérico também dado pelo utilizador.

Na figura 3.35 está apresentado um exemplo de funcionamento de uma variável que representa um contador que toma o valor dos números de zero a dez.

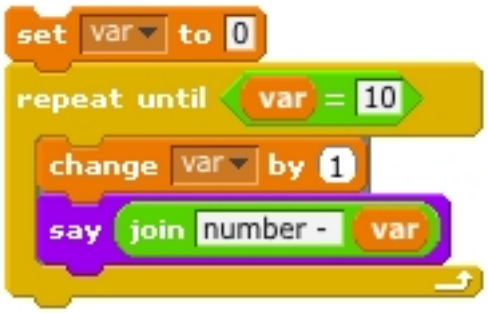


Figura 3.35: Contador

3.2.8 Listas

Em Scratch as listas são representadas na Área de **Scripts** e na Paleta de blocos por blocos **Reporter** arredondados. São criadas e nomeadas apenas pelo utilizador, tal como as variáveis. No **Stage** o seu conteúdo pode ser apresentado através do visualizador de listas representado na figura 3.36.

Figura 3.36: Visualizador de uma lista no **Stage**

Tal como as variáveis, as listas podem ser locais ou globais. As listas globais podem ser lidas e editadas por qualquer **Sprite** ou pelo **Stage**. Contudo as listas locais apenas podem ser editadas e lidas pelo **Sprite** em que foram criadas. Não existe um bloco, como para as variáveis, que permite que os **Sprites** possam ler os valores de uma lista local da qual não sejam donos.

Como nas variáveis, a definição da propriedade local ou global das listas é feita aquando da sua criação de uma maneira semelhante à descrita para as variáveis.

As listas podem ser editadas diretamente no seu visualizador no **Stage** através do rato e teclado ou indiretamente através dos blocos de listas contidos da categoria **Variables** da Paleta.

O uso das listas é muito variado, podem ser usadas para guardar valores múltiplos, inventários, etc. Existem vários blocos para editar listas que permitem inserir valores numa determinada posição, ou ler uma determinada posição, saber se existe um valor numa lista e outros.

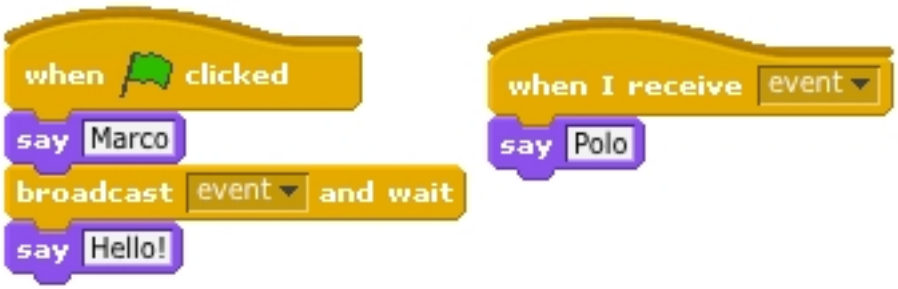
Um exemplo de uso de uma lista e de alguns blocos está representado na figura 3.37. Neste exemplo uma lista é percorrida e o **Sprite** dono ”diz” todos os seus valores um por um até a lista terminar.

Figura 3.37: Script para percorrer uma lista

3.2.9 Eventos e comunicação entre Scripts

Em Scratch os vários tipos de eventos são detetados pelos blocos **Hat** 3.24 e usam-se para dar início a **Scripts**. Os eventos detetados podem ser interações do utilizador com a animação, como por exemplo carregar na Bandeira verde, carregar numa tecla específica ou carregar num **Sprite** ou **Stage**. Além destes também existem eventos transmitidos entre **Scripts** através de blocos específicos para esse efeito.

(a) Comunicação assíncrona entre **Scripts**



(b) Comunicação síncrona entre **Scripts**

Figura 3.38: Os dois tipos de comunicação entre **Scripts**

A comunicação entre **Sprites** e sincronização entre os vários **Scripts** é feita através deste mecanismo de transmissão de eventos. Através dos dois blocos iniciados por **broadcast**, um **Script** pode transmitir um evento com um nome específico.

Quando esta transmissão acontece, qualquer **Script** de qualquer **Sprite** que seja iniciada pelo bloco **when i receive [evt▼]** começa a sua execução. De notar que o nome do evento transmitido pelo bloco **broadcast** tem que ser o mesmo do recetor do bloco **when i receive [evt▼]** para que a relação entre eles funcione.
O mecanismo de eventos implementado permite que a transmissão de apenas um evento possa ativar multiplos **Scripts**, qualquer que seja o número de **Scripts** recetores. Além disso a transmissão de eventos pode ser síncrona ou assíncrona. Estes dois tipos de comunicação entre **Scripts** estão ilustrados na figura 3.38.
Em cada uma das duas subfiguras da figura 3.38 temos dois **Scripts** pertencentes a dois **Sprites** diferentes que estão a ter um diálogo. Na figura 3.38a como é usado o bloco **broadcast [evt▼]** não há garantia que o segundo **Sprite** vai dizer "Polo"antes de o primeiro dizer "Hello!". Isto acontece porque o bloco **broadcast [evt▼]** implementa um mecanismo de eventos assíncrono.
Este bloco após transmitir o seu evento não espera que cada um dos **Scripts** que ele ativa terminem a sua execução. Ou seja, o bloco transmite o seu evento e logo de seguida a execução do **Script** onde ele está incluído continua.
Deste modo o bloco **broadcast [evt▼]** pode, por exemplo, ativar **Scripts** que executam um ciclo infinito sem bloquear a execução do **Script** onde está inserido.
Alternativamente, na figura 3.38b, como é usado o bloco **broadcast [evt▼] and wait** há garantia que o segundo **Sprite** vai dizer "Polo"antes de o primeiro dizer "Hello!". Isto acontece porque o bloco **broadcast [evt▼] and wait** implementa um mecanismo de eventos sincrono.
Este bloco transmite o seu evento e de seguida espera que todos os **Scripts** que esse evento ativa terminem a sua execução. Só depois disto o **Script** onde ele está inserido continua a sua própria execução.

3.2.10 Modelo de concorrência

No desenvolvimento de animações é normal existirem muitas situações em que os utilizadores necessitam de programar um **Sprite** com vários comportamentos que atuam ao mesmo tempo. Para este efeito é normal, em Scratch, utilizar diferentes *threads* para implementar tarefas conceptualmente diferentes.

Uma *thread* em Scratch é representada por cada **Script** de um **Sprite**. Por exemplo na figura 3.39 estão representados os dois **Scripts** pertencentes ao **Sprite** *Bouncy Ball* predefinido no Scratch. Este **Sprite** representa uma bola que salta pelo **Stage** e ressalta nas bordas. O **Script** da direita é responsável por mover a bola e fazê-la ressaltar quando bate numa borda do **Stage**. O **Script** da esquerda comanda a rotação da bola de acordo com a direção que esta tem em cada momento.

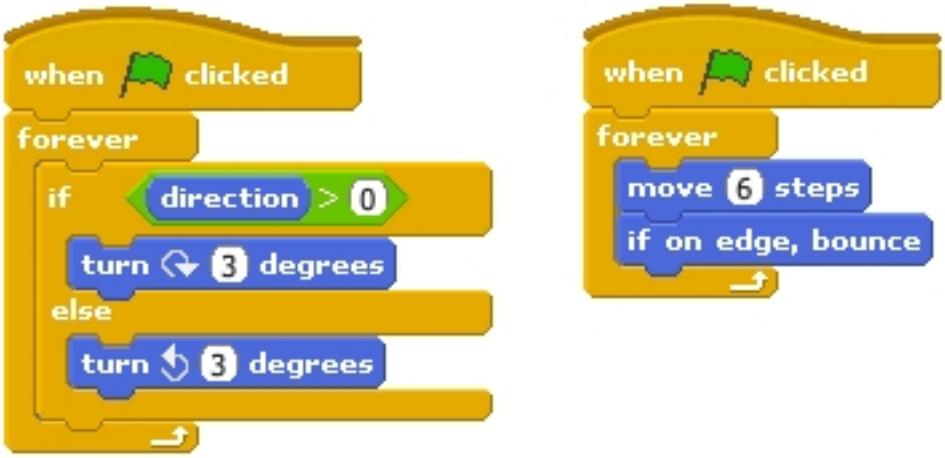


Figura 3.39: **Scripts** concorrentes do **Sprite** *Bouncy Ball*

Em Scratch podemos pensar em cada **Script** iniciado por um bloco **Hat** 3.24 como uma *thread*. Estes **Scripts** são executados concorrentemente, pois, no Scratch, apenas um bloco é executado de cada vez. Deste modo, não existe verdadeira execução paralela de **Scripts** visto que não é possível executar dois blocos exata-

mente ao mesmo tempo. O que acontece no Scratch é que a execução de **Scripts** "paralelos", como o da figura 3.40, é tão rápida que parece existir verdadeiro paralelismo. Nestas situações os **Scripts** comportam-se como *threads* concorrentes e, embora no Scratch não existam os mecanismos das linguagens textuais de controlo de concorrência (como semáforos, *locks* ou monitores), o controlo de concorrência é feito implicitamente pelo modelo de concorrência nativo do Scratch. Este modelo evita a maioria das *race conditions* fazendo com que o utilizador não tenha de se preocupar muito com estas questões durante o desenvolvimento dos seus projetos. Neste modelo de concorrência os locais onde podem ocorrer mudanças de contexto (mudança da *thread* ou **Script** que está a ser executado) estão restringidos. Particularmente, as mudanças de *thread* podem acontecer num de dois locais: num bloco onde é feita uma espera (**wait()**, **wait until**<> ou **broadcast [evt▼] and wait**) ou então no fim de uma iteração de um bloco que implementa um ciclo.
No caso do bloco **broadcast [evt▼] and wait** a mudança de contexto só acontece se existir um **Script** iniciado pelo bloco **when i receive [evt▼]** e se o nome do evento transmitido pelo primeiro for o mesmo que o nome que o segundo recebe. Ou seja, só existe mudança de contexto de o evento transmitido acionar a execução de um determinado **Script**.
No caso do bloco **wait until**<> a mudança de contexto só acontece se a condição a ser testada pelo bloco for falsa. Por outro lado, caso a condição do bloco seja verdadeira o **Script** onde este está inserido continua a sua execução sem que se dê uma mudança de *thread*.

Figura 3.40: **Scripts** concorrentes

Na figura 3.40 estão representados dois **Scripts** concorrentes pois são inicializadas pelo mesmo bloco **Hat**. Nestes casos é indeterminado qual é o **Script** que começa a execução primeiro. Assumindo que o **Script** que começa primeiro é o **Script** da esquerda, o primeiro bloco a ser executado é o **set** que atribui à variável **var** o valor 0.

De seguida é executado o bloco **wait** que provoca uma mudança de contexto e assim começa a execução do **Script** da direita.

Neste novo **Script** são executados os blocos **think** e também os dois blocos da categoria **Motion** que correspondem à primeira iteração do bloco de ciclo **repeat**. De seguida, acabada a primeira iteração do ciclo, acontece outra mudança de contexto voltando assim a execução ao primeira **Script** de blocos.

Nesta são executados os blocos **say** e seguidamente o bloco **wait until** testa se a variável **var** tem o valor 1. Como neste momento a variável **var** tem o valor 0, o teste falha e é executada nova mudança de contexto para o outro **Script**.

Neste **Script** são executados novamente os dois blocos **Motion** correspondentes agora à segunda iteração do ciclo e chegado novamente o fim da iteração do ciclo,

que neste caso corresponde também ao seu fim. O contexto muda novamente para o **Script** da esquerda.

É feito novo teste pelo bloco **wait until** que é novamente falso e provoca nova mudança de contexto.

De novo no **Script** da direita é atribuído o valor 1 à variável **var** através de um bloco **set** e como não existem mais blocos para processar neste **Script** a execução muda para o **Script** da esquerda onde é feito novo teste à variável, desta vez com resultado positivo. O **Script** então continua a sua execução terminando com o bloco **say**.

Para visualizar a ordem de execução de cada bloco basta selecionar a opção **Start Single Stepping** no submenu **Edit** no Menu 3.2. Para ver os blocos destacados lentamente seleciona-se a opção **Set Single Stepping** e de seguida **Flash blocks (slow)**.

É de assinalar o facto de que nos **Scripts** que são iniciados pelo mesmo bloco **Hat**, como os da figura 3.40, é impossível saber qual o **Script** que é executado primeiro, levando isto a que por vezes a ordem de execução das **Scripts** não seja exatamente aquela de que o utilizador está à espera. Isto acontece, por exemplo, quando vários **Scripts** iniciam a execução com o mesmo evento. Este problema é facilmente resolvido acionando apenas um **Script** com o primeiro evento lançado e criando outros eventos para acionar os **Scripts** seguintes com a ordem desejada pelo utilizador.

Outro problema, menos frequente mas importante, surge quando existe espaço partilhado de variáveis em vários **Scripts** concorrentes, ou melhor, quando vários **Scripts** utilizam e alteram as mesmas variáveis. Estas situações podem implicar por vezes não determinismo nos resultados, ou seja, é impossível prever qual o resultado da computação.

Na figura 3.41 temos um destes casos. Existe uma variável comum aos dois **Scripts**, num deles um bloco **set** atribui-lhe o valor 1 e no outro um bloco **set** distinto atribui-lhe o valor 0. Como é impossível saber qual o **Script** que começa primeiro a execução, não sabemos também qual será o resultado final da variável, implicando um resultado não determinista. Nestes casos o utilizador deve implementar controlo de concorrência explícito.

Figura 3.41: **Scripts** concorrentes com resultado final indeterminado

Aparte estes últimos casos não deterministas, é garantido que, independentemente da ordem de execução das instruções, o estado final da computação é atingido. Deste modo o utilizador pode construir cada **Script** sem ter de pensar muito nos efeitos colaterais que outros **Scripts** terão sobre o que está a ser construído.

Figura 3.42: Bloco **if**<> a implementar um lock

Neste modelo de concorrência as mudanças de *thread* nunca podem acontecer no decorrer de blocos sem esperas explicitas ou por exemplo num bloco **if**<> entre o teste de condição e o corpo de blocos inseridos na sua cavidade. Assim, este

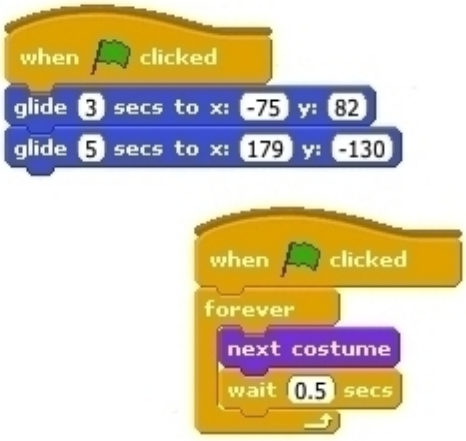
modelo implementa um conceito parecido com "regiões criticas", como é exemplo o **Script** de dois blocos da figura 3.42.

Aqui o valor da variável *lock* é garantidamente alterado depois da condição testada, sem que nenhuma outra *thread* possa ter o controlo enquanto o bloco aninhado na cavidade do **if**<> seja executado. Este mecanismo permite assim que os utilizadores mais avançados implementem os seus próprios sistemas de sincronização, ainda que tal seja poucas vezes necessário.

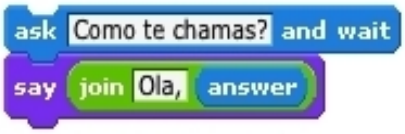
3.3 Conceitos de programação no Scratch

As tabelas seguintes são adaptadas de um trabalho de tradução/adaptação intitulado "Competências e conceitos de programação no Scratch"[dC11], sendo o documento original produzido pelo Lifelong Kindergarten Group, MIT Media Lab [KG06]. Estas tabelas mostram, de uma forma direta e simples, algumas funcionalidades básicas de programação que são possíveis de implementar em Scratch na sua maioria através dos seus blocos.

Conceito	Descrição	Exemplo
sequência	Para criar um programa Scratch é preciso pensar de forma sistemática na ordem de execução das instruções.	
iteração (ciclos)	**forever** e **repeat** podem usar-se para repetir um bloco de instruções.	
instruções condicionais	**if** e **if-else** verificam a ocorrência de uma condição.	

variáveis	A categoria **Variables** permite a criação de uma nova variável e a sua utilização num programa. As variáveis podem conter números ou sequências de carateres (texto). O Scratch permite a definição de variáveis globais e locais.	
listas	Os comandos relativos a listas permitem armazenar e aceder a conjuntos ordenados de números e texto. São estruturas de dados dinâmicas sobre as quais é possível efectuar operações de adição, enumeração, substituição e remoção de elementos.	
gestão de eventos	**when key pressed** e **when sprite clicked** são exemplos de gestão de eventos – resposta a eventos espoletados pelo utilizador ou por outra secção de um programa.	
execução "paralela"	Lançar dois blocos de comandos ao mesmo tempo cria dois fluxos independentes que são executados em "paralelo".	

<table>
<tr>
<td>coordenação e sincronização</td>
<td>broadcast e when I receive permitem coordenar a execução de múltiplos blocos de comandos. broadcast and wait possibilita a sincronização de execução.</td>
<td>Por exemplo, Sprite1 envia a mensagem victória quando a condição é verificada:
 

Este bloco de comandos do Sprite2 é accionado quando a mensagem victória é recebida:
 
</td>
</tr>
<tr>
<td>entrada de dados via teclado</td>
<td>ask and wait solicita ao utilizador que escreva. answer armazena o que foi escrito no teclado.</td>
<td> 
</td>
</tr>
<tr>
<td>números aleatórios</td>
<td>O bloco pick random escolhe números inteiros aleatoriamente dentro de uma certo intervalo.</td>
<td> 
</td>
</tr>
<tr>
<td>lógica boleana</td>
<td>and, or e not são exemplos de lógica boleana.</td>
<td> 
</td>
</tr>
<tr>
<td>interacção em tempo real</td>
<td>mouse_x, mouse_y e loudness podem ser usados dinamicamente para interação em tempo real.</td>
<td> 
</td>
</tr>
<tr>
<td>desenho de interface do utilizador</td>
<td>No Scratch é possível desenhar interfaces de utilizador interactivas – por exemplo usando sprites clicáveis para criar botões.</td>
<td> 
</td>
</tr>
</table>

3.4 Conclusão

Neste capítulo foi apresentada a interface do ambiente Scratch ao pormenor, passando por todas as áreas necessárias para uma primeira abordagem ao ambiente e sua linguagem visual. De seguida foram expostos todos os conceitos de programação que podem ser implementados pelos blocos Scratch e que são essenciais para a construção de comportamentos para **Sprites**. Foram analisadas todas as categorias e tipos de blocos existentes bem como os tipos de dados. O modelo de objetos do Scratch baseado em **Sprites** foi também analisado, bem como o seu modelo nativo de concorrência entre **Scripts** e também o seu modelo de comunicação baseado em eventos. Por fim, foi apresentada uma tabela onde se faz um paralelo entre alguns dos mais importantes conceitos de programação e o seu equivalente construído usando blocos Scratch.

Capítulo 4

Morphic

Neste capítulo será feita uma apresentação e revisão geral do importante sistema de interface com o utilizador do Squeak, o *Morphic*, sistema este que é utilizado para criar todo o ambiente Scratch. Na secção 4.1, será apresentado e explorado o Morphic e o seu principal constituinte de construção, o *morph*, e todas as suas propriedades principais. Também será exposto o funcionamento geral do Morphic, nomeadamente o seu o ciclo de atualização, como são processadas as interações com o utilizador, como lida com as atualizações de *layout* dos *morphs* e como é feita a atualização do ecrã.
Na secção 4.2 serão explicados os quatro princípios principais que são usados no desenho do sistema Morphic: concretude, manipulação direta, animação de vida e uniformidade.

4.1 Morphic e seu funcionamento

O Morphic, [Mal02], é a interface gráfica com o utilizador (GUI) do Squeak que é uma implementação do modelo MVC (*Model-View-Controler*) do Smalltalk-80. Originalmente o Morphic foi desenvolvido por Randy Smith e Jonh Maloney para

o sistema de programação Self, e posteriormente foi portado por Jonh Maloney para o Squeak Smalltalk onde substituiu o modelo original MVC. Esta interface trabalha com objetos gráficos de construção chamados *morphs*, nome este que vem da palavra grega "*morph*"que significa "forma".

Cada *morph* é representado visualmente no ambiente de maneira a criar a ilusão de um objeto real que pode ser agarrado, movido, apagado, redimensionado e largado noutro *morph*. Apesar desta ilusão, na realidade cada *morph* é um objeto plano que pode ser sobreposto noutros *morphs* em diferentes camadas. Este sistema de sobreposição em camadas tem como objetivo criar a ilusão de profundidade no ambiente, pois quando dois *morphs* se sobrepõem o que se encontra "mais à frente"cobre parte do *morph* que se encontra "mais atrás".

Para além disto, esta ilusão de profundidade e realidade dos *morphs* é também reforçada devido à sombra que surge por trás de um *morph* enquanto o utilizador lhe agarra com o rato. Na figura 4.1 estão representados três *morphs* sobrepostos em que o *morph* da frente está a ser agarrado pelo rato e projeta a sua sombra devido a esse facto.

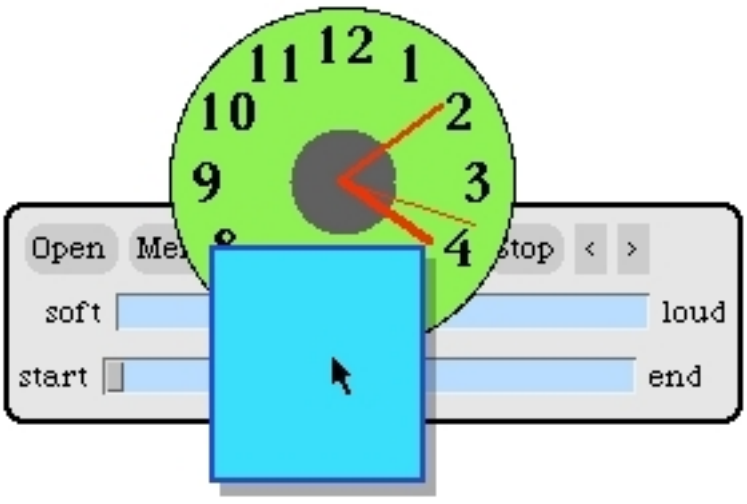


Figura 4.1: Três *morphs* sobrepostos

Um *morph* pode ser programado para executar ações em variadas situações, por exemplo em resposta a *inputs* do utilizador, entre intervalos de tempo regulares, quando é largado por cima de outro *morph* ou outro *morph* é largado por cima deste, etc.

4.1.1 *Morphs* compostos

Cada *morph* pode incorporar outros *morphs* na sua constituição que são chamados *submorphs*. Além disso, um *morph* que pertence aos *submorphs* de outro *morph* refere-se a este como o seu *owner*. Assim, a partir de *morphs* mais simples é possível criar estruturas gráficas compostas que se comportam como um objeto singular que pode ser copiado, apagado ou movido. No entanto, quando os *submorphs* que compõem um *morph* composto são separados, cada um deles representa um *morph* concreto e independente que pode ser visto e manipulado como qualquer outro. Isto está representado figura 4.2.

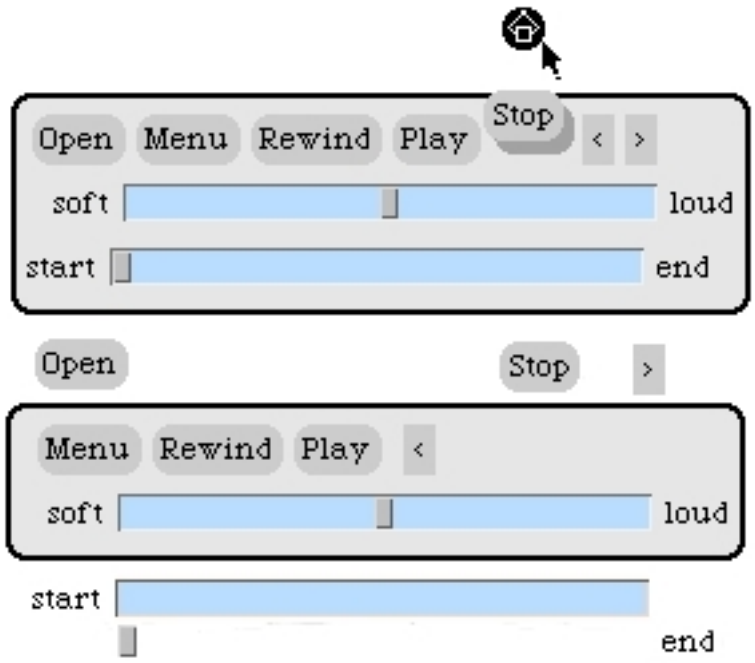


Figura 4.2: *Morph* composto e seus *submorphs*

Assim, qualquer *morph* pode ter *submorphs* próprios, pertencer aos *submorphs* de outro *morph* ou ter estas duas propriedades ao mesmo tempo.

O *morph* que é a origem de uma composição de outros *morphs*, ou seja, que não tem um *morph owner*, é designado *root*. Sendo que cada *morph* tem os seus *submorphs* e cada um destes *submorphs* pode ter os seus próprios *submorphs* então podemos concluir que a estrutura de um *morph* composto é uma árvore. Nesta estrutura, cada *morph* sabe sempre quem é o seu *owner* e os seus *submorphs*. Para descobrir o owner ou os *submorphs* de um determinado *morph* basta enviar-lhe uma mensagem `owner` ou `submorphs` respetivamente. Desta forma é possível descobrir o contexto do ambiente em que cada *morph* se insere.

Todos os *morphs* no ecrã pertencem ou à cadeia de *submorphs* do *morph* chamado `world`, que é uma instancia de `PastUpMorph`, ou aos *submorphs* do *morph* chamado `hand`, que é uma instancia de `HandMorph` e representa o cursor do rato. Quando um *morph* é agarrado pelo rato, esse *morph* é removido da cadeia de *submorphs* do *morph* `world` e adicionado aos *submorphs* do *morph* `hand`, e quando um *morph* está a ser agarrado pelo rato e é largado no mundo o processo é inverso.

Um *morph* pode também ser apagado atribuindo o valor `nil` ao seu *owner*. Isto faz com que este seja removido dos *submorphs* do seu *owner* atual.

Para descobrir qual o *morph* raiz de um determinado *morph* basta enviar-lhe a mensagem `root`. O Morphic executa uma travessia da cadeia de *owners* desse *morph* até encontrar o *morph* raiz ou o valor `nil`. Também é possível descobrir se um *morph* está numa cadeia de *morphs* que se encontra no mundo ou que se encontra a ser agarrada pelo rato. Esta deteção é feita enviando a um *morph* a mensagem `root`, para descobrir a raiz da cadeia em que se inclui, seguida da mensagem `owner`. Este processo tem como resultados os *morphs* `world` e `hand` ou o resultado `nil`. O resultado `world` significa que o *morph* composto onde o *morph* testado se insere está no mundo e não se encontra a ser agarrado pelo rato. O

resultado `hand` significa que o *morph* testado se encontra a ser agarrado pelo rato. E o resultado `nil` significa que o *morph* testado não pertence a nenhuma cadeia de *morphs*, encontra-se "apagado".

4.1.2 Ciclo geral

O sistema Morphic funciona correndo um ciclo geral que atualiza constantemente o ecrã do utilizador. Neste ciclo, num primeiro momento, o Morphic interpreta os eventos que estão a acontecer no ambiente e de seguida envia a mensagem `step` a todos os *morphs* ativos que necessitam de atualizações de *layout*. Então, cada *morph* altera o seu *layout* de acordo com a sua necessidade e finalmente o ecrã final é apresentado de novo. De uma forma simples isto é um ciclo do Morphic. Ou seja:

- processar eventos de *input*
- enviar mensagem `step` aos *morphs* ativos
- atualizar *layouts* de cada *morph*
- atualizar o ecrã

É necessário também referir que cada passo do ciclo pode não ser necessário, podendo não acontecer. Isto dá-se porque podem não existir eventos para processar ou *morphs* aos quais seja necessário enviar a mensagem `step`, podem não ser necessárias atualizações de *layout* nos diferentes *morphs* nem atualizações ao ecrã. Durante o tempo em que nenhum destes passos do ciclo necessita de execução o Morphic entra num modo *sleep* ou modo de espera.

4.1.3 Interação com o utilizador

A interação do utilizador com o Morphic é definida através de vários tipos de eventos. Existem eventos que representam o premir de teclas do teclado, `keyStroke`, e eventos que representam o estado dos botões e movimentações do rato: `mouseDown`, `mouseMove`, etc. Para os representar são usadas instancias de `MorphicEvent`. Esta classe guarda o tipo de evento, o estado do rato no momento desse evento e também o estado de três teclas especiais o *shift*, *control* e o *alt* (windows) ou *command* (Mac). Estas teclas especiais podem ser utilizadas em combinação com outros eventos para definir comportamentos alternativos. Por exemplo, executar uma certa ação apenas quando a tecla *control* é premida em combinação com o botão do rato premido.

O processamento de eventos consiste no seu envio para os *morphs* apropriados. Por exemplo, os eventos `Keystroke` que acontecem quando o utilizador carrega em teclas do teclado são enviados para o *morph* que está em foco neste momento. Esta focagem de um *morph* é definida por um clique do rato dentro da sua área, podendo apenas existir um *morph* em foco a cada momento ou nenhum. Neste último caso o evento de `Keystroke` é descartado.

No caso dos eventos `mouseDown`, que representam o premir do rato, estes são direcionados dependendo da localização do *morph* relativamente aos outros. Isto é, o evento é enviado para o *morph* que está "mais à frente"e não é possível este evento ser recebido pelos *morphs* que estão "por trás"deste. Quando o rato é premido por cima de um *morph* o Morphic envia-lhe a mensagem `handlesMouseDown:`. Esta mensagem tem a função de testar se o *morph* está preparado para receber este evento. Caso a resposta seja positiva, o *morph* realiza o comportamento presente no método `mouseDown:` , caso contrário o *morph* é agarrado pelo rato.

No caso de o *morph* ser composto, o *submorph* que se encontra "mais à frente"vai ser o primeiro a ter a oportunidade de lidar com o evento. E, caso ele não esteja

preparado, esse evento passa para o seu *owner* que está "por trás"e assim sucessivamente.

4.1.4 Atualização do *layout* dos *morphs*

A atualização do *layout* dos *morph*s é feita que sempre que as alterações feitas a um *morph* o obrigam. As mudanças mais comuns que podem obrigar à alteração de *layout* de um *morph* são a adição ou remoção de *submorphs* a esse *morph* ou a alteração do tamanho desse *morph* ou dos seus *submorphs*.

Quando o *layout* de um *morph* necessita ser atualizado o Morphic envia-lhe a mensagem `layoutChanged`, mensagem essa que despoleta o processo em cadeia de ajuste do *layout* desse *morph* e de outros *morphs* relacionados com esse que necessitem também de ajustes.

Neste processo de atualização, em primeiro lugar o *layout* do *morph* alterado que recebe a mensagem é atualizado. Esta atualização pode ter repercussões nos seus *submorphs*, pode-se verificar por exemplo uma alteração do espaço dado a cada *submorph* o que leva a alterações dos seus *layouts*. Por outro lado o espaço necessário para acomodar o *morph* pode também ser alterado o que pode levar à necessidade atualização do seu *owner* e possivelmente também o *owner* do seu *owner* e assim sucessivamente.

4.1.5 Atualização do ecrã

Para manter a ecrã atualizada o Morphic utiliza um algoritmo de atualização gradual, com *double buffering*, eficiente, de alta qualidade e automático. Este algoritmo utiliza a técnica de *double buffering* o que quer dizer que utiliza dois *buffers* ao mesmo tempo para representar o ecrã, um que é apresentado ao utilizador e o outro para fazer atualizações. É também é eficiente porque tenta fazer

o mínimo de trabalho possível para atualizar o ecrã depois de uma alteração e é de alta qualidade porque o utilizador nunca vê o ecrã a ser repintado.
O Morphic em cada ciclo utiliza uma lista chamada *damage list* que contém informação sobre que porções do ecrã têm de ser redesenhadas. Quando um *morph* muda o seu aspeto manda a ele próprio a mensagem `changed` que adiciona o retângulo que o envolve à *damage list*. Posto isto, o Morphic redesenha todos os *morphs* que intersetam cada um dos retângulos da *damage list* no *buffer* de atualizações que se encontra separado do ecrã apresentado ao utilizador. Por fim, este novo ecrã invisível é copiado quase instantaneamente para o ecrã do utilizador.
Este sistema de atualização permite que o utilizador nunca veja as fazes intermediárias do processo de redesenho dos *morphs* no ecrã, o que torna a animação mais consistente e de melhor qualidade visual.

4.2 Princípios da construção

4.2.1 Concretude e manipulação direta

O Morphic tenta criar a ilusão de objetos concretos no ecrã baseando-se em propriedades dos objetos do mundo real. Esta propriedade, concretude, ajuda o utilizador a perceber o que acontece no ecrã tentando criar uma analogia entre os *morphs* e os objetos do mundo real. Através da experiência de uso, é percetível rapidamente que tudo no ambiente pode ser tocado e manipulado diretamente. Mesmo os *morphs* compostos podem ser separados noutros, e todos podem ser editados ou inspecionados a partir de um clique do rato dentro da área de cada um. Esta característica de manipulação direta dos *morphs* é outra das propriedades do desenho do Morphic.
Para implementar esta concretude e manipulação direta de objetos o Morphic utiliza várias técnicas. Para cria a sensação de que os *morphs* são objetos reais e

concretos a atualização do ecrã é feita através do algoritmo de *double buffering*, como já foi dito, o que permite que o utilizador nunca veja os *morphs* quando estão a ser desenhados mostrando-os sempre no seu estado completo e "real". Além disto, também quando um objeto é agarrado pelo rato é projetada uma sombra desse objeto no ecrã o que melhora a sensação de "realidade"da manipulação do *morph*.

Qualquer desenho feito no ecrã é parte de um *morph* que pode ser descoberto através de um clique do rato combinado com a tecla *command* (Mac) ou *alt* (Windows). Esta combinação de teclas possibilita o acesso ao *halo* do *morph* premido. Este *halo* está representado na figura 4.3.

Figura 4.3: Um *morph* e o seu *halo*

Através dos botões do halo é possível mudar vários aspetos do *morph* diretamente através de cliques e arrastamentos do rato. O utilizador pode mudar o tamanho, posição, rodar e até a estrutura de um *morph* composto. Esta característica reforça a ideia de manipulação direta e concretude já que os objetos reais também são manipulados através de contacto direto.

Por fim esta noção de objetos concretos que podem ser manipulados é também reforçada pelo facto de tudo o que é representado no ambiente serem *morphs* ou *morphs* compostos por outros. Ou seja, todas as peças são concretas e podem ser

montadas e desmontadas por manipulação direta.

4.2.2 Animação viva

No Morphic os *morphs* podem ser programados para terem um comportamento autónomo como no mundo real, por exemplo, como um relógio, como um telemóvel que toca etc. Assim os *morphs* podem executar comportamentos independentemente da interação com o utilizador.

O principal mecanismo que dá vida aos *morphs* é o mecanismo de *stepping*. Este mecanismo, quando inicializado num *morph* através do método `startStepping`, executa o método `step` desse *morph* a um ritmo de uma vez por segundo por omissão. A frequência com que o método `step` é executado num *morph* pode ser definida no método `stepTime`.

Este mecanismo permite ao programador definir o comportamento autónomo de cada *morph*. Para o bom funcionamento deste mecanismo é aconselhável que os métodos `step` sejam eficientes pois podem ter de ser executados com muita frequência dependendo do `stepTime` definido. Também é aconselhável escolher um `stepTime` apropriado ao objeto animado que queremos representar num *morph*.

A cada momento o Morphic sabe quais são os *morphs* que através do envio de `step`necessitam ser atualizados, e sabe também o tempo exato em que cada um tem de ser atualizado. Deste modo, a cada ciclo do Morphic, a mensagem `step` é enviada para os *morphs* aos quais a hora de atualização chegou. Quando isto acontece o Morphic atualiza a hora para a próxima atualização de cada *morph*.

4.2.3 Uniformidade

No mundo real todos os objetos obedecem às mesmas leis da física, por isso são uniformes. O Morphic tenta também integrar esta característica de uniformidade do mundo real de uma forma similar. Isto é conseguido evitando casos especiais. Todos os objetos no ecrã são do mesmo tipo: subclasses da classe `Morph` que herdam todas as suas características, métodos e propriedades.
Tudo no ecrã são *morphs*. Cada um pode ter *submorphs* ou ser um *submorph* de outro *morph*. Bem como todos os *morphs* compostos (têm *submorphs*) se comportam também como *morphs* atómicos. Assim o Morphic agrupa todos os tipos de objetos num modelo geral e uniforme com regras e características globais.

4.3 Conclusão

Neste capítulo foi feita uma apresentação do sistema Morphic do Squeak que substituiu o modelo MVC do Smalltalk-80. Foi apresentada de uma maneira simples a maneira como o Morphic atualiza a todo o momento o *layout* dos *morphs* e o ecrã do utilizador bem como as características dos objetos gráficos base do Morphic, os *morphs*. Por fim foram descritos os quatro princípios gerais da conceção do Morphic: concretude, manipulação direta, animação viva e uniformidade.

Capítulo 5

Implementação de grafos em Scratch

A introdução de grafos no ambiente Scratch começou por uma fase de análise do seu código fonte Squeak. Esta análise centrou-se em perceber como é construído o mecanismo que suporta a criação de listas e variáveis do Scratch, e, de seguida, desenvolver uma adaptação deste mecanismo para suportar a nova estrutura de dados e objetos visuais representativos de grafos no ambiente. Neste capítulo será descrita toda a implementação deste novo mecanismo no ambiente Scratch.

Primeiro, na secção 5.1, será explicado o novo dicionário de grafos que é guardado num **Sprite** ou no **Stage** e que permite que estes tenham os seus próprios grafos guardados, tal como podem ter listas e variáveis.

Na secção 5.2 explicar-se-á a nova classe que guarda toda a informação sobre um grafo e todos os métodos criados para a sua edição. Em seguida será explicada toda a construção de componentes, funcionamento e métodos essenciais do novo visualizador de grafos do palco. Esta descrição será feita na secção 5.3.

A seguir, na secção 5.4, será descrito método de criação dos novos blocos de instruções específicos para os grafos e os seus novos argumentos.

Por fim, na secção 5.5, serão apresentadas as alterações feitas para que a Paleta de blocos mostre os novos componentes relativos aos blocos grafo.

5.1 Dicionário de grafos

Qualquer variável ou lista no Scratch, é guardada no **Stage**, se for global, ou num dos **Sprites**, local. A classe que guarda a informação tanto das variáveis como das listas é a classe `ScriptableScratchMorph` 5.15. Esta é a superclasse das classes que representam o *morph* do **Stage** e os *morphs* dos **Sprites**, `ScratchStageMorph` e `ScratchSpriteMorph` respetivamente.

A informação relativa a variáveis e listas do **Stage** ou **Sprites** é guardada nas variáveis de instancia `vars` e `lists` de cada `ScriptableScratchMorph`, sendo cada uma delas um `Dictionary` inicializado em `ScriptableScratchMorph>>initialize`. No caso das variáveis, o dicionário `vars` tem como chaves o nome de cada variável e como valor associado a cada uma delas a *string* ou valor numérico que esta variável guarda. De outro modo, no caso das listas, o dicionário `lists` tem como chaves os nomes de cada lista e como valor associado a cada uma destas chaves o seu *morph* visualizador 3.36 representado pela classe `ScratchListMorph`.

Sabendo isto, para guardar os novos grafos foi adicionada uma variável de instancia `graphs` 5.1 à classe `ScriptableScratchMorph` que é também um `Dictionary`.

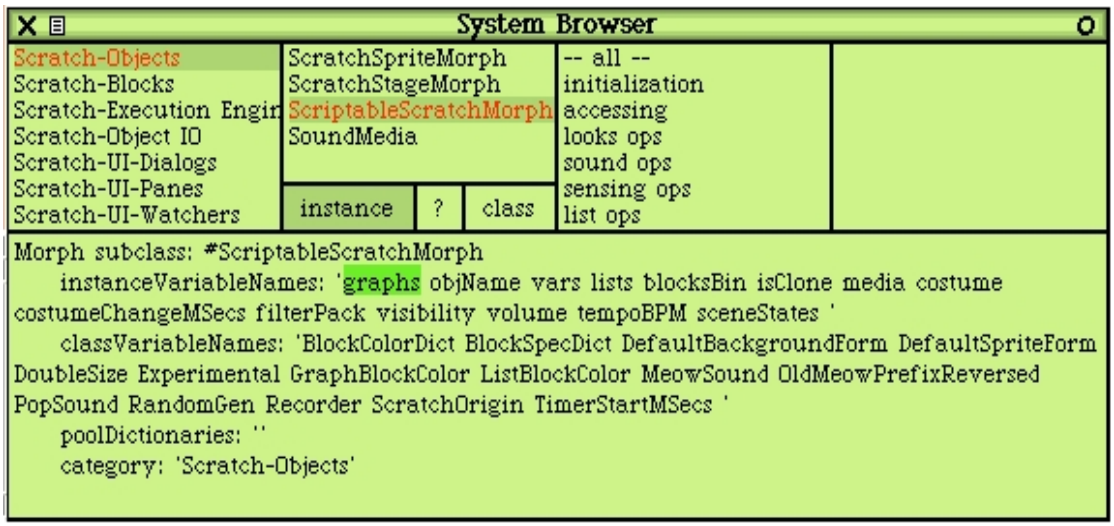


Figura 5.1: System Browser - variável de instancia graphs

Este dicionário `graphs` tem como chaves os nomes de cada grafo e tem como valor associado, a cada uma destas chaves, o seu *morph* visualizador, 5.2, representado pela classe `ScratchGraph1Morph`. O dicionário `graphs` é inicializado tal como os outros dois no método `initialize` da classe `ScriptableScratchMorph`.

```
ScriptableScratchMorph>>initialize
        super initialize.
        graphs _ Dictionary new.
        vars _ Dictionary new.
        lists _ Dictionary new.
...
```

Para o sistema suportar este novo `Dictionary graphs` foi necessário introduzir uma serie de métodos novos e fazer algumas alterações a métodos existentes. Estas alterações foram baseadas na implementação do Scratch relativamente aos `Dictionarys`: `lists` e `vars` ja existentes.
Foi também criado um novo protocolo de nome `graphs ops` na classe `ScriptableScratchMorph` que contem métodos importantes relativos a grafos, tal como existe um `list ops` e um `var ops`. Os métodos adicionados para controlar o novo `Dictionary` de grafos de cada `ScriptableScratchMorph` em muitos casos são muito semelhantes aos métodos com a mesma função nas listas. Por isso estes métodos estarão listados na secção A.2 dos anexos e não serão explicados aqui. Apenas importa dizer que neste anexo estão referenciados todos os novos métodos de manipulação e acesso do novo `Dictionary` de grafos, bem como as alterações feitas a métodos já existentes utilizados pelo ambiente Scratch. Estes métodos já existentes que foram alterados são métodos que acedem aos dicionários de variáveis e listas e neste momento também acedem ao novo dicionário de grafos.

5.2 Estrutura de dados - classe Graph

No Squeak não existe uma classe que represente toda a informação e manipule explicitamente grafos. Assim, foi criada uma classe que guarda toda a informação relativa a um grafo de uma maneira lógica e estruturada. Esta classe que representa e manipula uma estrutura de grafo chama-se `Graph` e é uma subclasse de `Dictionary`.

Em Smalltalk um `Dictionary` é uma `Collection` que representa um agrupamento ou `set` de associações entre chaves e valores. Existe uma chave única própria de cada valor guardado num Dictionary, e, através dela, é possível aceder ao seu valor associado. `Dictionary(key -> value)`

A estrutura que representa um grafo nesta classe é a seguinte:

```
Dictionary(key -> {value . Dictionary(key -> value)})
```

Esta estrutura consiste num dicionário de chaves em que cada uma está associada a um par. O primeiro elemento deste par contém um valor que pode ser qualquer objeto, enquanto que o segundo elemento do par contém um novo dicionário. No contexto de um grafo os campos desta estrutura significam o seguinte:

```
Dictionary(nodeID -> {value . Dictionary(edgeID -> weight)})
```

Cada nodo do grafo é representado por uma chave `nodeID`. Associada a esta chave está um par. Num dos campos do par, o campo `value`, guarda um *value* que é uma informação que seja necessário guardar relativa ao nodo `nodeID` e o outro campo guarda um novo dicionário `Dictionary(edgeID -> weight)`. Este dicionário representa os *edges* ou ligações do nodo `nodeID`, ou seja, representa os nodos aos quais o nodo `nodeID` tem uma ligação direta. Estes *edges* são representados

com associações entre chaves `edgeID` e valores numéricos `weight`. As chaves do dicionário de *edges* identificam os nodos do grafo aos quais o nodo `nodeID` se liga diretamente. Enquanto que os valores numéricos correspondentes a cada uma das chaves que representam o peso de cada um destes *edges*.

É de referir que o grafo representado por esta nova classe `Graph` é um grafo orientado e que os *edges* representados no `Dictionary` de *edges* de cada nodo são unidirecionais, ou seja partem do nodo identificado por `nodeID` para os nodos identificados por `edgeID` que são chaves do dicionário de *edges*.

Relativamente à estrutura descrita anteriormente que suporta a informação do grafo, pode-se dizer que esta é construída ao mesmo tempo que o grafo é também construído através dos métodos definidos na classe. Todos eles foram escritos sabendo de antemão a estrutura de informação para grafos pretendida e estão preparados para a manipular sem destruir a sua organização interna. Ou seja, a classe `Graph` é na sua base um `Dictionary` comum, e, através das inserções de informação feitas pelos métodos de inserção da classe, o conteúdo do Dictionary toma o formato padronizado da estrutura de informação explicada anteriormente que representa um grafo.

Por exemplo, para introduzir um novo nodo no grafo utiliza-se um de dois métodos: `Graph>>insertNode: nodeID value: aValue` ou em alternativa `Graph>>insertNode: nodeID value: aValue edges: aDict`.

O primeiro método de inserção indicado define-se da seguinte maneira:

```
Graph>>insertNode: nodeID value: aValue
   self at: nodeID put: {value . Dictionary new}.
```

Este primeiro método recebe como argumentos o nome do novo nodo a ser introduzido e o seu *value* correspondente. Na sua execução toma estes dois valores e cria uma nova entrada no `Dictionary` de nodos da classe `Graph`. Introduz uma nova chave `nodeID` e no valor associado a essa chave introduz um par contendo

em primeiro lugar o `value` recebido como argumento e em segundo lugar um novo `Dictionary` vazio que representa os *edges* deste novo nodo. Neste momento este dicionário de *edges* encontra-se vazio.
O segundo método de inserção indicado define-se da seguinte maneira:

```
Graph>>insertNode: nodeID value: value edges: aDict
   self at: nodeID put: {value . aDict}.
```

Este segundo método recebe como argumentos o nome do novo nodo a ser introduzido, o seu *value* correspondente e o seu dicionário de *edges*. Este método comporta-se de maneira semelhante ao primeiro, excetuando no segundo valor do par. Aqui o método em vez de criar um novo `Dictionary` vazio para os *edges* do novo nodo, introduz o `Dictionary` recebido como argumento.
Em ambos os métodos a estrutura pretendida para a informação do grafo é mantida. E isto acontece com todos os métodos definidos nesta classe, sejam do protocolo `adding`, `removing`, `accessing` ou `algorithmic`.

5.3 Visualizador de grafos

Tal como as listas do Scratch têm o seu visualizador, figura 3.36, para manipulação direta de listas no **Stage**, foi necessário criar um novo *morph* visualizador para o mesmo efeito nos grafos. Este novo componente está representado na figura 5.2. O *morph* criado para este efeito é uma subclasse de um `BorderedMorph` e tem o nome `ScratchGraph1Morph`.

Neste *morph* existe uma variável de instancia chamada `graph` que contém uma instancia da classe `Graph` explicada na secção 5.2. É aqui que este *morph* guarda toda a informação sobre o conteúdo do grafo que está a ser representado. Com cada alteração ao grafo feita pelos blocos ou diretamente no visualizador, o conteúdo de `graph` é atualizado conforme a alteração feita. Os métodos que acedem ou alteram o conteúdo desta estrutura de dados encontram-se todos no protocolo `graph ops` desta classe, `ScratchGraph1Morph`. Estes métodos são invocados tanto pelo próprio visualizador para a definição da funcionalidade e comportamento dos seus *submorphs* sobre o grafo como também por métodos pertencentes ao protocolo também chamado `graph ops` da classe `ScriptableScratchMorph`, que definem as funções dos novos blocos de grafo da Paleta de blocos.

Este visualizador permite navegar pelo grafo criado, mostrando a cada momento apenas um dos nodos do grafo, o nodo atual. Para este nodo é possível visualizar a sua lista de *values* associados, a azul, e também os índices de cada *value*. Além disto também é apresentada a lista de nodos do grafo aos quais este nodo atual se liga diretamente. Esta lista de células representa os *edges* do nodo atual a vermelho e ao lado de cada célula é também representado o peso de ligação.

As células representativas dos *values* e *edges* do nodo atual podem atuar como botões que quando carregados têm diferentes comportamentos. Esta funcionalidade pode estar ativa ou não.

Em relação aos *values* associados do nodo atual, estes podem ser de dois tipos:

textual ou evento. No caso de o *value* ser um evento o utilizador pode, através de um clique na sua área azul, lançar esse evento tal como um bloco **broadcast [evt▼]**. Caso o *value* premido pelo utilizador seja textual, o **Sprite** dono do grafo representado no visualizador transmite o texto contido no *value* através de um balão de texto.

A navegação no grafo usando o seu visualizador é feita através de cliques nas áreas da lista de *edges* de cada nodo. Quando o utilizador carrega num dos *edges* de um nodo o *layout* do *morph* visualizador é atualizado. O nodo que corresponde ao *edge* carregado pelo utilizador torna-se o nodo atual e é apresentada a sua lista de *values*, seus *edges* e respetivos pesos de cada um.

Existem dois modos de funcionamento deste *morph*: o modo de edição e o modo de execução. No modo de execução apenas é possível navegar através de um grafo já construído, lançar os eventos que são *values* de um nodo e visualizar os *values* textuais de um nodo bem como fazer com que um **Sprite** transmita, através de um balão de texto, o valor de um *value*. Não é permitido, neste modo, qualquer tipo de edição do grafo.

No modo de edição estão disponíveis todas as funcionalidades do modo de execução mais as funcionalidades de edição do grafo. Assim, aqui é também possível remover *values* ou *edges* de um nodo, remover nodos do grafo, adicionar novos nodos ao grafo e adicionar *values* e *edges* a um nodo.

5.3.1 Componentes do visualizador

Descrevem-se a seguir as variáveis de instancia que contêm cada um dos componentes e subcomponentes da classe `ScratchGraph1Morph` que define o *morph* visualizador de grafos.

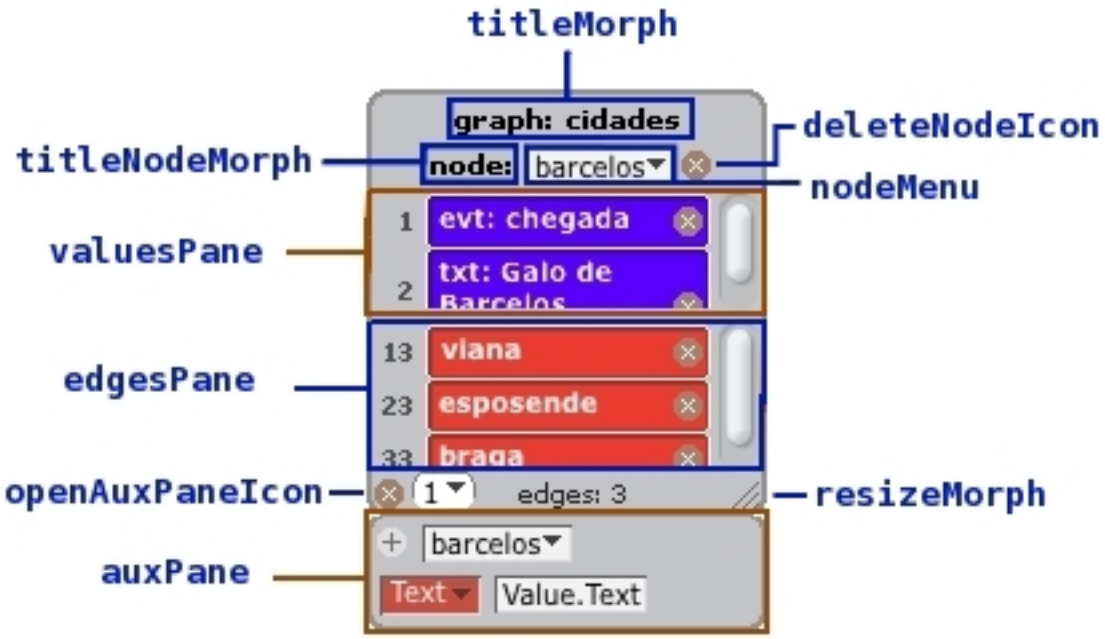


Figura 5.2: Visualizador de um grafo - modo edição

O `titleMorph`, representado na figura 5.2, contém um `StringMorph` que representa o nome do grafo que está a ser apresentado no visualizador. Localiza-se acima de todos os restantes elementos. O nome do grafo aparece sempre em frente à *string* '**graph:**' e caso este grafo seja local a um determinado **Sprite** o nome deste **Sprite** aparece também antes da *string* '**graph:**'. Como exemplo, se o grafo da figura 5.2 pertencer a um **Sprite** chamado **Sprite1** o `titleMorph` apareceria da seguinte forma: '**Sprite1 graph: cidades**'.

Figura 5.3: Visualizador de um grafo - modo execução

O `titleNodeMorph`, representado na figura 5.2, mostra uma *string* através de um `StringMorph`, e localiza-se inferiormente ao `titleMorph`. Caso o visualizador esteja no modo de execução, a *string* representada começa por '**node:**' e em seguida este *morph* mostra o nome do nodo atual, como na figura 5.3. Mas caso o visualizador esteja no modo de edição, a *string* apresentada é apenas **'node:'**, 5.2, devido a que ao seu lado direito localiza-se o `nodeMenu` que já contém o nome do nodo atual. Em ambos os estados do visualizador, execução ou edição, quando o grafo visualizado é vazio o `titleNodeMorph` mostra uma *string* com o conteúdo '**no nodes**'.

O `nodeMenu`, representado na figura 5.2, contém um `GraphExpressionArgMorph` e só se encontra ativo no modo de edição do visualizador. Através deste *morph*, o `nodeMenu` mostra a cada momento o nodo atual. Possui também um menu que pode ser acedido através de um clique no seu triângulo interior e que mostra todos os nodos do grafo num menu *drop down*. Através de um clique num dos nodos apresentados neste menu, o visualizador mostra este nodo escolhido tornando-o no nodo atual que está a ser visualizado.

O `deleteNodeIcon`, representado na figura 5.2, contém um `ImageMorph` que tem o formato de uma cruz. Este *morph* apenas está presente no modo de edição do visualizador e utiliza-se para remover o nodo atual do grafo. Quando um nodo é removido ele é apagado do `Dictionary` de grafos, em seguida é calculado o novo nodo atual com o método `ScratchGraph1Morph>>currentNode` e por fim o novo nodo atual é apresentado pelo visualizador.

O `valuesPane`, representado na figura 5.2, é o painel onde são apresentados os *values* de um nodo do grafo e os seus indices. Para isso, é utilizado um `ScrollFrameMorph2`. Este *morph* é constituído por uma *scrollbar* colocada à direita, representada por um `ScratchScrollBarMorph`, e por um *morph* genérico transparente à esquerda `Morph`. Este tem a função de reunir nos seus *submorphs* os *morphs* componentes do painel, ou seja o seu conteúdo. Neste espaço designado para o conteúdo do painel são representados os *values* do nodo atual em células azuis e também os índices de cada *value* ao seu lado esquerdo.

Cada célula azul, 5.4, da lista de *values* do nodo atual que é representada no visualizador, é uma composição de dois *morphs* diferentes e ainda um adicional que só está presente no modo de edição do visualizador.

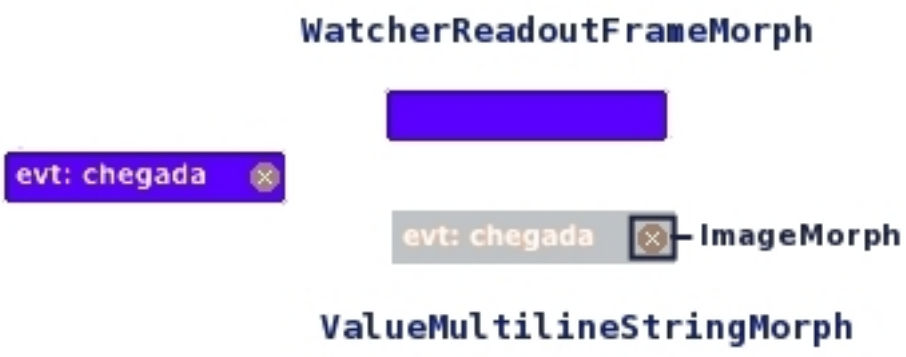


Figura 5.4: Célula de um *value*

O primeiro é um `WatcherReadoutFrameMorph` que representa a moldura azul da célula. Dentro dos *submorphs* deste último existe um `ValueMultilineStringMorph` que é uma subclasse de `MultilineStringMorph` e permite conter *strings* em várias linhas. Por fim temos incutido neste último *morph* um `ImageMorph` com a forma de uma cruz que aparece apenas quando o visualizador está no modo de edição. Esta cruz utiliza-se para remoção do *value* representado nesta célula.
Todas estas células representativas dos *values* do nodo atual, apresentadas na figura 5.5, estão guardadas numa `OrderedCollection` com o nome de `valueMorphs`.

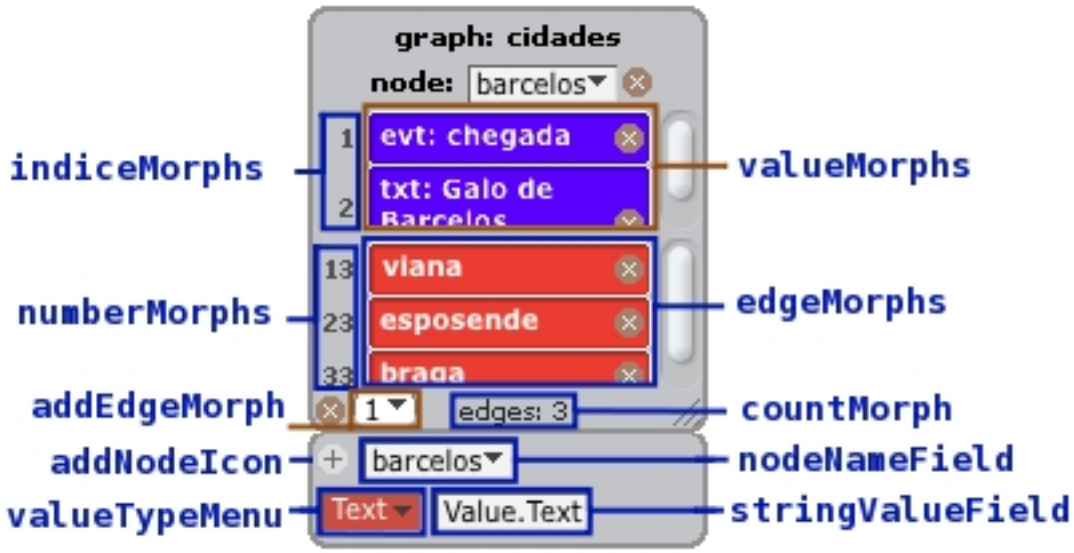


Figura 5.5: Visualizador de um grafo - modo edição

Ao lado esquerdo de cada célula azul está o seu índice. Cada um dos índices é representado por um `StringMorph` e todos estes *morphs*, representados na figura 5.5, estão guardados também numa `OrderedCollection` com o nome de `indiceMorphs`. Quando o nodo atual não tem *values* associados, ou seja quando a `OrderedCollection valueMorphs` se encontra vazia, no conteúdo do painel `valuesPane` é introduzido um `StringMorph` de nome `emptyValuesMorph`, representado na figura 5.6. Este *morph* contém a *string* '**(no values)**' que notifica ao utilizador a inexistência de *values* associados ao nodo atual.

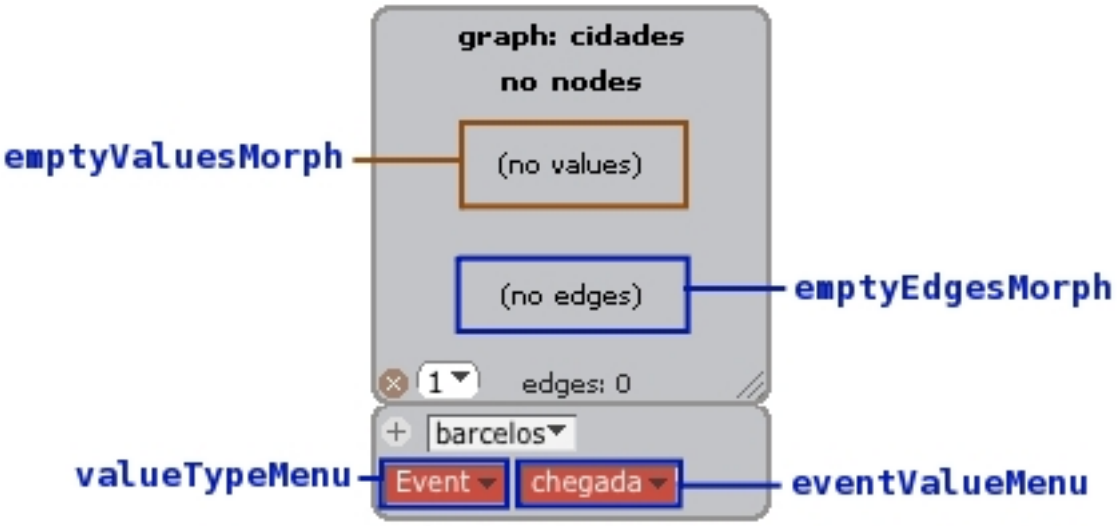


Figura 5.6: Visualizador de um grafo vazio

O `edgesPane`, representado na figura 5.2, é o painel onde são apresentados os nodos que se ligam ao nodo atual diretamente, os *edges*. Para isso é também utilizado um `ScrollFrameMorph2` como no `valuesPane`. A diferença é que no espaço designado para o conteúdo do painel são representados cada um dos *edges* do nodo atual individualmente. Cada um é apresentado numa célula vermelha estando ao lado esquerdo de cada uma o seu peso de ligação

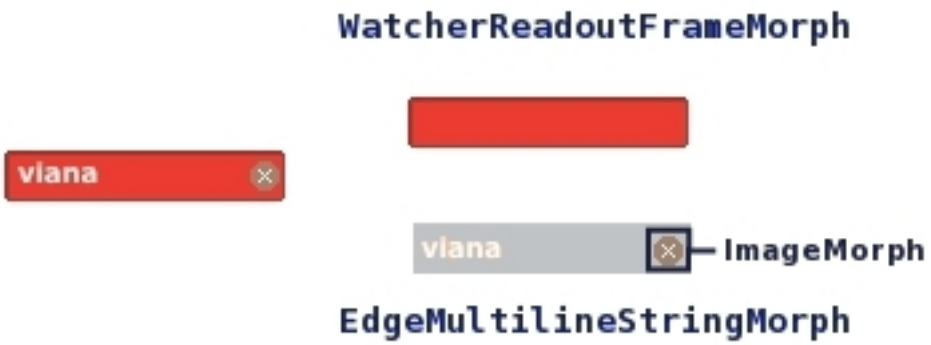


Figura 5.7: Célula de um *edge*

Cada célula vermelha, 5.7, da lista de *edges* do nodo atual que é representada no visualizador é, tal como nos *values*, uma composição de dois *morphs* diferentes e ainda um adicional que só está presente no modo de edição do visualizador. O primeiro é um `WatcherReadoutFrameMorph` que é a moldura vermelha da célula. Dentro dos *submorphs* deste último existe um `EdgeMultilineStringMorph` que também é uma subclasse de `MultilineStringMorph` e permite conter *strings* em várias linhas e tem fundo transparente. Por fim temos também incutido neste último *morph* um `ImageMorph` com a forma de uma cruz para remoção do *edge* representado na célula, este *morph* aparece apenas quando o visualizador está no modo de edição.

Todas estas células representativas dos *edges* do nodo atual, representados na figura 5.5, estão guardados numa `OrderedCollection` com o nome de `edgeMorphs`. Ao lado esquerdo de cada célula azul está o seu peso de ligação. Cada um dos pesos é representado por um `StringMorph` e todos estes *morphs*, representados na figura 5.5, estão guardados também numa `OrderedCollection` com o nome de `numberMorphs`.

Quando o nodo atual não tem *edges* aos quais se liga diretamente, ou seja quando a `OrderedCollection edgesMorphs` se encontra vazia, no conteúdo do painel `edgesPane` é introduzido um `StringMorph` de nome `emptyEdgesMorph` representado na figura 5.6. Este *morph* contém a *string* '**(no edges)**' que notifica ao utilizador a inexistência de *edges* para o nodo atual.

O `resizeMorph`, representado na figura 5.2, contém um `ScratchResizeMorph` que se situa no canto inferior direito do painel principal do *morph* visualizador. Este *morph* permite redimensionar o painel através da combinação de um clique e arrastamento do rato.

O `countMorph`, representado na figura 5.5, contém um `StringMorph` que mostra o número de *edges* que o nodo atual que está a ser apresentado pelo visualizador

tem. Por exemplo se o nodo atual tem três *edges*, a *string* apresentada será '**edges: 3**'. Este *morph* aparece inferiormente ao `edgesPane`.

O `addEdgeMorph`, representado na figura 5.5, contém também um `GraphExpressionArgMorph` e só se encontra ativo no modo de edição do visualizador. Este *morph* utiliza-se para adicionar novos *edges* ao nodo atual. Para isso, o utilizador apenas pode introduzir valores numéricos na sua área de *input*, estes serão o peso do novo `edge`. Para adicionar um novo *edge*, o utilizador tem de abrir o menu que mostra todos os nodos do grafo clicando no triângulo e de seguida carregar num dos nodos do grafo apresentados. Um novo *edge* será adicionado ao nodo atual com o peso definido na área de *input* pelo utilizador.

O `openAuxPaneIcon`, representado na figura 5.2, contém um `ImageMorph` que pode ter duas formas distintas. Este *morph* só se encontra ativo no modo de edição do visualizador e usa-se para abrir e fechar o painel auxiliar `auxPane` com cliques do rato. Quando o painel auxiliar está ativo, o ícone toma a forma de uma cruz e quando o painel auxiliar esta fechado o ícone toma a forma de um sinal mais.

O painel auxiliar `auxPane`, representado na figura 5.2, contém um `AuxPaneMorph` e só se encontra ativo no modo de edição do visualizador. Este painel auxiliar tem como funções adicionar novos *values* a nodos existentes no grafo e também criar novos nodos para o grafo. Este *morph* contém os *submorphs* com os nomes seguintes: `nodeNameField`, `valueTypeMenu`, `stringValueField`, `eventValueMenu` e `addNodeIcon`.

O `nodeNameField`, representado na figura 5.5, é uma variável do `auxPane` que contém um `GraphExpressionArgMorph`. Este menu é o local onde é definido o nodo onde ser vai adicionar o novo *value*. Neste campo é possível escrever o nome de um novo nodo ou alternativamente escolher um dos nodos já existentes no grafo a partir do seu menu. Caso o nome do nodo neste campo seja um dos nodos já existentes no grafo, o novo *value* será adicionado a esse nodo e de seguida esse

nodo é apresentado no visualizador. Caso contrário, se o nome do nodo que está neste campo anda não existir no grafo, é criado um novo nodo no grafo com um novo *value* e apresentado logo de seguida no visualizador tornando-se o nodo atual.
O `valueTypeMenu` é uma variável de `auxPane` que contém um `ChoiceArgMorph`. Este campo é um menu que define o tipo de *value* que será introduzido num nodo do grafo. Contém duas escolhas possíveis: '**Text**' e '**Event**', expostas respetivamente nas figuras 5.5 e 5.6. A alteração do tipo de *value* neste menu altera também o campo de introdução do novo *value*. Se o tipo presente neste menu for '**Text**' então o campo para introdução do novo *value* será um `stringValueField`, por outro lado caso o tipo escolhido for '**Event**' o campo de novo *value* será um `eventValueMenu`.
O `stringValueField`, representado na figura 5.5, é uma variável de `auxPane` que contém um `GraphExpressionArgMorph`. Aqui este *morph* não tem um menu como no `nodeNameField`. Este campo permite ao utilizador inserir uma *string* para um novo *value* a ser inserido.
O `eventValueMenu`, representado na figura 5.6, é uma variável de `auxPane` que contém um `EventTitlePlusMorph`. Este *morph* permite escolher eventos já existentes no grafo ou na Área de **Scripts** ou então criar novos eventos para serem posteriormente inseridos como novos *values* no grafo.
O `addNodeIcon`, representado na figura 5.5, é uma variável do `auxPane` que contém um `ImageMorph` com a forma de um sinal mais. Quando este ícone é carregado o `auxPane` adiciona o novo *value* de acordo com o preenchimento dos seus campos.

5.3.2 Definição dos componentes e sua interação

GraphExpressionArgMorph

O *morph* `GraphExpressionArgMorph` é o *morph* base de quatro componentes do visualizador: `nodeMenu`, `addEdgeMorph`, `nodeNameField` e `stringValueField`. Destes componentes o `nodeMenu` e `addEdgeMorph` são *submoprhs* diretos do visualizador e o `nodeNameField` e `stringValueField` pertencem ao painel auxiliar `auxPane` que é *submorph* do visualizador. Estes quatro componentes do visualizador 5.8, têm todos funcionalidades e comportamentos distintos que estão todos definidos na classe `GraphExpressionArgMorph`.

Figura 5.8: Os quatro GraphExpressionArgMorphs

Um `GraphExpressionArgMorph` é uma subclasse de `ExpressionArgMorph`, este último é um *morph* que é utilizado normalmente nos blocos Scratch para introdução de valores numéricos ou *strings* e permite também edição desses valores. Por exemplo na figura 3.40 todas as caixas com fundo branco tendo números ou *strings* são `ExpressionArgMorphs`. Este *morph* pode ser definido como sendo nu-

mérico ou de *string*. Esta definição é feita aquando da criação de uma instancia da classe `ExpressionArgMorph` através dos métodos `numExpression: aNumber` e `stringExpression: aString` nos quais os argumentos significam o valor inicial que vai ser apresentado no `ExpressionArgMorph`. Caso se use o método `numExpression: aNumber`, o *morph* criado é arredondado significando que apenas se podem introduzir caracteres que representam valores numéricos. Estes caracteres são os algarismos de 0 a 9 e os sinais '.','-' e ',' sendo os restantes caracteres barrados na tentativa da sua introdução. Caso se use o método `stringExpression: aString` o *morph* criado é retangular significando que se podem introduzir quaisquer caracteres. Esta seleção dos valores introduzidos é feita no *submorph* `StringFieldMorph` que é o componente de `ExpressionArgMorph` onde são guardados e escritos os caracteres.
No visualizador de grafos, os componentes que são `GraphExpressionArgMorphs` têm diferentes regras de permissão de caracteres:

- o `addEdgeMorph` apenas permite a entrada de valores numéricos que representam os pesos de um novo *edge*;
- o `nodeNameField` apenas permite a entrada das letras do abecedário, espaços e algarismos para a definição do nome de nodos;
- o `stringValueField` permite a entrada de quaisquer caracteres para um *value* textual de um nodo;
- e o `nodeMenu` não permite a edição do valor apresentado no seu interior.

Para definir estas diferentes permissões de introdução de caracteres foi criada uma subclasse de `StringMorph` chamada `GraphStringField` que é o componente do `GraphExpressionArgMorph` que recebe e mostra os caracteres. Através de uma *string* que é passada como argumento ao método `permittedChars: aString`,

aquando da inicialização de um `GraphExpressionArgMorph`, o `GraphStringField` testa se os valores que são escritos no teclado pertencem a esta *string* de valores permitidos. Caso não pertençam são barrados.

No caso do `nodeMenu` não é permita edição nem introdução de novos caracteres. Esta característica é definida também na criação do `GraphExpressionArgMorph` através do método `isEditable: aBoolean` que define a permição de edição do seu `GraphStringField`. Este método é herdado da sua superclasse `StringFieldMorph`.

Um `GraphExpressionArgMorph` permite também mostrar um menu de opões para seleção. A construção dos métodos necessários para este menu foi baseada no *morph* `ExpressionArgMorphWithMenu`. Este *morph* é uma subclasse de `ExpressionArgMorph` que adiciona a possibilidade de ser apresentado e acedido um menu de opções através do clique num *submorph* com forma de triângulo no seu interior. Este *morph* e o menu correspondente estão representados na figura 5.9. O `ExpressionArgMorphWithMenu` corresponde ao campo com fundo branco, e é também apresentado o seu menu.

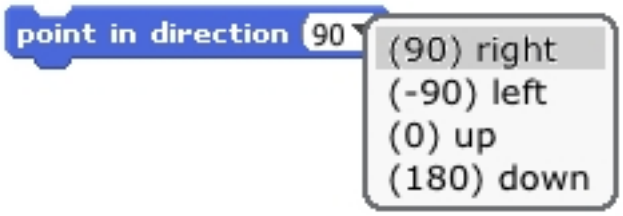


Figura 5.9: ExpressionArgMorphWithMenu

Para criar um `GraphExpressionArgMorph` com um menu, na sua inicialização define-se através do método `GraphExpressionArgMorph>>menuSelector: aSymbolOrNil` que é o método que vai ser executado para devolver as opções do menu. Nos três componentes do visualizador que são `GraphExpressionArgMorphs` e contêm um menu (`nodeMenu`, `addEdgeMorph` e `nodeNameField`) o `menuSelector` é comum: o método `ScratchGraph1Morph>>listNodesMenu`. Encontrando-se no

protocolo `graph ops` do visualizador. Este método constrói um `CustomMenu` que contém os nomes de todos os nodos do grafo.

Para definir o `menuSelector` de um novo `GraphExpressionArgMorph` é invocado o método `menuSelector` da seguinte forma: `menuSelector: #listNodesMenu`. Quando isto acontece é adicionado ao *morph* `GraphExpressionArgMorph` o `menuMorph` que corresponde ao triângulo de menu, e à variável `getMenuSelector` é atribuído o `selector #listNodesMenu` correspondente ao método `listNodesMenu`.

Para cada um dos menus destes três componentes existe uma funcionalidades diferente. Estas diferentes funcionalidades estão todas definidas no método `GraphExpressionArgMorph>>mouseDown: evt`. Em primeiro lugar quando o `menuMorph` é carregado, é apresentado o menu de todos os nodos do grafo que é construído pelo método `listNodesMenu`. Depois disto o utilizador escolhe uma opção do menu e de seguida podem acontecer três funções diferentes de acordo com a maneira como o `GraphExpressionArgMorph` foi definido na sua criação. Para a escolha da funcionalidade a usar o método `mouseDown: evt` baseia-se em dois parâmetros. Um deles é o parâmetro `isNumber` e o outro o `menuFunction`.

O parâmetro `isNumber` tem o valor `True` se o método usado para definir o valor inicial do `GraphExpressionArgMorph` foi o `numExpression: aNumber`. E tem o valor `false` se o método usado foi o `stringExpression: aString`. E o parâmetro `menuFunction` é definido pelo método `menuFuntion: aNumber` e por omissão tem o valor 2.

Caso o parâmetro `menuFunction` for 1, o valor que é escolhido no menu de nodos torna-se o valor que se encontra no `GraphStringFiled` da caixa. Este é o comportamento que pertence ao componente `nodeNameField` do painel auxiliar `auxPane`.

Caso o parâmetro `menuFunction` não seja 1, o comportamento do menu vai ser ditado pelo parâmetro `isNumber`.

Se o parâmetro `isNumber` tiver o valor `True`, que acontece quando o `GraphExpres-`

`sionArgMorph` tem um formato arredondado, o valor escolhido no menu de nodos é adicionado ao grafo como um novo `edge` do nodo atual. Este novo *edge* é adicionado ao grafo tendo o peso que estiver introduzido no `GraphStringField` do *morph* e a sua adição é feita invocando o método `ScratchGraph1Morph>>addEdgeTo: nodeName to: nodeName2 weight: weight`. O comportamento descrito é o do componente `addEdgeMorph`.

No outro caso, quando o parâmetro `isNumber` tem o valor `False`, que acontece quando o `GraphExpressionArgMorph` tem um formato retangular, o valor escolhido no menu de nodos torna-se no nodo atual e é apresentado no visualizador através do método `ScratchGraph1Morph>>showNode: nodeName`. Este é o comportamento do componente `nodeMenu`.

EdgeMultilineStringMorph e ValueMultilineStringMorph

Os *morphs* `EdgeMultilineStringMorph` e `ValueMultilineStringMorph` são subclasses de `MultiLineStringMorph`. Estes dois *morphs* são utilizados na composição das células de visualização de *edges* e *values* do nodo atual. Permitem mostrar *strings* e é nelas que estão definidos os comportamentos que são executados quando a funcionalidade de clique se encontra ativada e um utilizador clica na superfície de uma célula. O primeiro *morph* é usado para mostrar o nome de um *edge* do nodo atual e permite navegar até esse nodo através de um clique. O segundo *morph* é usado para ver o nome e tipo de um dos *values* do nodo atual. Caso esse *value* seja um evento Scratch através de um clique é possível lançar esse evento, e também, caso esse *value* seja textual, através de um clique o **Sprite** dono do grafo ao qual o *value* pertence transmite em palco o texto contido nesse *value* através de um balão de texto. Estes dois *morphs* já se encontram representados nas figuras 5.7 e 5.4.

Em ambos os *morphs*, com o visualizador de grafos em modo de edição, é possível

apagar o *value* ou *edge* que estes mostram com o `deleteMorph` neles incorporado. A superclasse destes *morphs*, `MultiLineStringMorph`, permite escrever texto no seu interior em várias linhas quando é focado com um clique do rato. Estas funcionalidades estão ativas no `ListMultilineStringMorph` que é usada no visualizador das listas do Scratch, 3.36, para introdução direta de conteúdos nas suas células. Este `ListMultilineStringMorph` é também subclasse de `MultiLineStringMorph`.

Nos *morphs* `EdgeMultilineStringMorph` e `ValueMultilineStringMorph` manteve-se a funcionalidade de apresentação de texto em várias linhas mas as funcionalidade de escrita com o teclado e focagem foram retiradas. O método que envia o sinal de focagem a ambos os *morphs*, `keyboardFocusChange: aBoolean` encontra-se alterado. Neste momento este método, em vez de ter a funcionalidade de focagem da célula, é responsável por apagar ou adicionar o `deleteMorph` à célula consoante o modo em que o visualizador se encontra, edição ou execução. Esta alteração permite que quando o visualizador de grafos está no modo de execução, o componente `deleteMorph` seja apagado dos *submorphs* e seja impossível remover um *edge* ou um *value* representado do nodo atual. Caso o visualizador esteja no modo de edição o `deleteMorph` é novamente adicionado e a funcionalidade de remoção reativada.

Em ambos os *morphs* o método que controla o comportamento de clique das células é o método `mouseDown: evt` de cada um. Estes dois métodos começam por verificar se a funcionalidade de clique se encontra ativa no visualizador de grafos correspondente da célula premida pelo utilizador. No caso do `EdgeMultilineStringMorph` a variável do visualizador testada é `edgesActive` e no caso do `ValueMultilineStringMorph` a variável do visualizador testada é `valuesActive`.

Quando a variável `edgesActive` contém o valor `false` a funcionalidade de clique das células vermelhas, correspondentes aos *edges*, encontra-se inativa. Aqui

quando um utilizador pressiona uma célula de *edge* o método `mouseDown: evt` correspondente ao `EdgeMultilineStringMorph` dessa célula reconhece que a funcionalidade de clique se encontra inativa e interrompe a sua execução através do retorno da mensagem `self`. Por outro lado quando a variável `edgesActive` tem o valor `true` o método prossegue a sua execução normal executando a funcionalidade de clique da célula do *edge* pressionada. Esta lógica, descrita anteriormente para o método `mouseDown: evt` do *morph* `EdgeMultilineStringMorph` utilizando a variável `edgesActive` do visualizador, é idêntica à utilizada para o método `mouseDown: evt` do *morph* `ValueMultilineStringMorph` utilizando a variável `valuesActive` do visualizador de grafo.

O processo dos métodos `mouseDown: evt` de ambos os *morphs* que é executado quando a funcionalidade de clique se encontra ativa encontra-se explicado a seguir. No caso do `EdgeMultilineStringMorph` no método `mouseDown: evt` é utilizado o método `ScratchGraph1Morph>>eIndexOfCell: aCell` para saber em que posição se encontra na `OrderedCollection edgeMorphs` a célula que foi premida. O próximo passo é recolher os conteúdos de `edgeMorphs`, com o método `ScratchGraph1Morph>>edgeContents`, e aceder à posição da célula clicada. Isto permite saber o nome do *edge* que se encontra na célula.

Depois de recolhido o nome do *edge* o próximo passo é invocar o método que determina a ação que será efetuada. Este método é escolhido com base no local do `EdgeMultilineStringMorph` que foi premido e depende também do modo de visualização atual do visualizador de grafos.

Assim, caso a variável `editMode` tenha o valor `false`, representando que o visualizador de grafos está no modo de execução, o nome do *edge* premido torna-se o nodo atual e é apresentado no visualizador através do método `ScratchGraph1Morph>>showNode: nodeName`. Caso contrário o visualizador está em modo de edição e aqui o local onde a célula é clicada vai ditar o seu funcionamento. Se o utilizador clicar

no `deleteMorph` o *edge* que está na célula premida é apagado do dicionário de *edges* do nodo atual no grafo através do método `ScratchGraph1Morph>>deleteEdgeOf: nodeName to: nodeName2`. Se o utilizador clicar noutra parte qualquer da célula o nodo que corresponde ao *edge* apresentado nessa célula torna-se o nodo atual do visualizador, novamente através do método `showNode: nodeName`.

No caso do `ValueMultilineStringMorph` o método `mouseDown: evt` utiliza o método `ScratchGraph1Morph>>vIndexOfCell: aCell` para saber em que posição se encontra na `OrderedCollection valueMorphs` a célula que foi carregada.

O próximo passo é invocar o método que determina a ação que será feita quando a célula é premida. Este método é escolhido com base no local do `ValueMultilineStringMorph` que foi carregado e depende também do modo de apresentação do visualizador de grafos.

Assim caso a variável `editMode` tenha o valor `true`, o visualizador encontra-se em modo de edição e, o local onde a célula é carregada dita o método que será invocado que vai definir o seu comportamento. Se o utilizador clicar no `deleteMorph`, o *value* que está na célula é apagado da `OrderedCollection` de *values* do nodo atual no grafo através do método `ScratchGraph1Morph>>deleteValueOfNode: nodeName pos: aNumber`. Se o utilizador clicar noutra parte qualquer da célula, é invocado o método `ScratchGraph1Morph>>runValueOfNode: nodeName position: aNumber` que determina o comportamento que é iniciado.

No caso em que a variável `editMode` tem o valor `false` o visualizador encontra-se em modo de visualização e aqui não é possível eliminar *values* devido a que o `deleteMorph` encontra-se inatico. Assim quando quando o utilizador clica em qualquer parte da célula é também aqui invocado o método `ScratchGraph1Morph>>runValueOfNode: nodeName position: aNumber` que determina o comportamento que se inicia.

Em ambos os modos do visualizador, edição e execução, no fim deste método é

invocado o método `ScratchGraph1Morph>>showCurrentNode` para atualizar o visualizador de grafos.

Em relação ao método `ScratchGraph1Morph>>runValueOfNode: nodeName position: aNumber` referido em cima, este determina, conforme o tipo de *value* que nela está presente, o comportamento que é iniciado quando uma célula é premida. Este método acede ao *value* que se encontra na `OrderedCollection` de *values* do nodo atual no grafo na posição calculada anteriormente. Caso este *value* seja um evento, este evento é transmitido através do método que também é utilizado pelo bloco **broadcast [evt▼]** do Scratch para lançar um evento, `ScriptableScratchMorph>>broadcast: name withArgument: arg`.

De outro modo, caso o tipo do *value* encontrado seja textual, o método testa se o dono do grafo é um **Sprite** ou o **Stage**. Caso o dono do grafo seja um **Sprite**, o conteúdo textual do *value* carregado é transmitido por esse **Sprite** através de um balão de texto. Este balão é invocado através do método que também é utilizado pelo bloco **say[ ]** do Scratch que faz um **Sprite** "dizer" um determinado texto através de um balão. Este método é o `ScratchSpriteMorph>>say: aValue`. Caso o dono do grafo seja o **Stage**, é invocado também o método `ScratchSpriteMorph>>say: aValue` para todos os **Sprites** que estão em palco, fazendo com que todos eles transmitam, num balão de texto, o valor textual do *value* que se encontra representado na célula premida pelo utilizador.

EventTitlePlusMorph

O *morph* `EventTitlePlusMorph` é uma subclasse de `EventTitleMorph` que é o `morph` usado nos blocos Scratch, por exemplo no bloco **broadcast [evt▼]**, 3.38, para criar novos eventos e mostrar todos os eventos presentes em blocos da aba de Scripts através de um menu de escolhas. O `EventTitlePlusMorph`, 5.10, tem a mesma funcionalidade mas no menu de eventos disponíveis aparecem também os

eventos que estão inseridos como *values* do grafo.

Figura 5.10: EventTitlePlusMorph

Este *morph* aparece no visualizador e também num dos blocos de grafo. Dependendo do local onde se encontra, tem maneiras distintas para descobrir os eventos presentes no grafo. Quando o menu é carregado é invocado o método `EventTitlePlusMorph>>presentMenu`. Em primeiro lugar, este método junta num `Array` chamado `eventNames`, todos os eventos que estão no `workPane`, que é a designação dada ao **Stage** no código. Aqui estão guardados todos os eventos ativos nos blocos que estão na aba de **Scripts**. Em seguida estes eventos são adicionados a um `CustomMenu` e é adicionada também uma linha a esse menu. Esta linha serve para separar, no menu, os eventos que estão em blocos dos eventos que estão nos *values* do grafo. Depois disto é preciso determinar os eventos do grafo e para isso é necessário aceder-lhe.

No caso de este *morph* se encontrar no painel auxiliar do visualizador, o `AuxPaneMorph`, o processo de aceder ao grafo é simples. Basta aceder à sua hierarquia de

owners até se encontrar o `ScratchGraph1Morph` e aceder ao seu grafo através do método `ScratchGraph1Morph>>graph`.
No caso de este *morph* se encontrar no bloco de grafos **add/change value**, é utilizado o método `EventTitlePlusMorph>>getGraph` para aceder ao grafo. Este método acede à sua hierarquia de *owners* até encontrar o `ScratchFrameMorph`, que é o *morph* que representa o ambiente total do Scratch. A partir do `ScratchFrameMorph` acede ao `workPane`, que representa o palco. Em seguida percorre todos os *submorphs* do palco que sejam `ScriptableScratchMorphs` e testa, para cada um, se existe algum visualizador de grafo com o nome do primeiro argumento do bloco para o **Sprite** ou **Stage** selecionado na lista de **Sprites**. Isto é feito através do método `ScriptableScratchMorphs>>graphNamed: aString ifNone: aBlock`. Caso exista um visualizador com este nome, este é atribuído à variável `graph1Morph`. Voltando ao método `presentMenu`, já com o grafo encontrado, são adicionados a uma `OrderedCollection` chamada `graphEvents` todos os nomes de eventos que pertencem aos *values* dos nodos do grafo. E logo de seguida estes são adicionados ao `CustomMenu` já com os eventos anteriores. Por fim é adicionada uma linha ao menu e também um valor especial '**new..**' que permite criar novos eventos.

AuxPaneMorph

O *morph* `AuxPaneMorph` é uma subclasse de *BorderedMorph* que representa o painel auxiliar e tem a função de adicionar novos nodos e *values* ao grafo. Contém as mesmas características visuais do seu *owner* o `ScratchGraph1Morph`. Este *morph* contém cinco *submorphs* na sua composição que são dois `GraphExpressionArgMorphs` um chamado `nodeNameField` e outro `stringValueField`, um `ChoiceArgMorph` chamado `valueTypeMenu`, um `EventTitlePlusMorph` chamado `eventValueMenu` e um `ImageMorph` chamado `addNodeIcon`.

Estes *submorphs* são todos inicializados e sua posição no painel definida no método `AuxPaneMorph>>initialize`. Aqui só são adicionados quatro destes *submorphs* ao painel, deixando de fora o `eventValueMenu`. Isto acontece porque a todo o momento o painel auxiliar contém ou o `eventValueMenu`, para *values* de evento, ou o `stringValueField`, para *values* de textuais.

A mudança do tipo de *value* a introduzir é feita através no `valueTypeMenu` que é um `ChoiceArgMorph` com as duas opções possíveis para tipos de um *value*: '**Text**' e '**Event**'.

Durante a execução, o `AuxPaneMorph` utiliza o mecanismo de *stepping* do Morphic para saber, a cada momento, se o tipo de *value* foi mudado no `valueTypeMenu`. Isto é feito no método `AuxPaneMorph>>step` onde o valor atual da escolha do `valueTypeMenu` é comparado com o valor de uma variável de nome `lastValueType` que guarda o último tipo de *value* que provocou uma alteração de *layout* no painel auxiliar.

Caso o tipo de *value* tenha sido mudado, é invocado o método `AuxPaneMorph>>updateAuxPane`. Este método tem a função de atualizar o *layout* do painel auxiliar quando necessário. Assim, caso o `lastValueType` tenha o valor '**Event**' e o valor da escolha atual do `valueTypeMenu` for '**Text**', o método `updateAuxPane` remove o *morph* correspondente ao `eventValueMenu` e introduz na mesma posição o *morph* correspondente ao `stringValueField` para introdução de texto. Este estado do `AuxPaneMorph` é representado na figura 5.11.

Figura 5.11: AuxPaneMorph com *value* textual

Caso o `lastValueType` tenha o valor '**Text**' e o valor da escolha atual do `valueTypeMenu` for '**Event**', o mesmo método remove o *morph* correspondente ao `stringVal-`

`ueField` e introduz na mesma posição o *morph* correspondente ao `eventValueMenu` para escolha ou criação de um *value* de evento. Este estado do `AuxPaneMorph` encontra-se representado na figura 5.12.

Figura 5.12: AuxPaneMorph com *value* de evento

No fim do método a variável `lastValueType` é atualizada para o valor da escolha do `ChoiceArgMorph` de nome `valueTypeMenu`. Depois do preenchimento dos campos com as informações necessárias o *value* pode ser adicionado ao grafo através de um clique no `addNodeIcon` que é um `ImageMorph` com a forma de um sinal mais. Esta funcionalidade é definida no método `AuxPaneMorph>>mouseDown: evt`. Aqui é testado se o evento de clique do rato está dentro da área do `addNodeIcon`. Caso isto aconteça, o método testa qual dos dois *morphs* de entrada de *values* está ativo, se o `stringValueField` ou o `eventValueMenu`. Caso o *morph* presente seja o `stringValueField`, a *string* do seu conteúdo é introduzida numa variável de nome `value`. Caso o *morph* presente seja o `eventValueMenu`, é criado um novo `ScratchEvent` com o nome do evento escolhido e é introduzido na variável `|value|` O método que se utiliza para atribuir o nome ao novo `ScratchEvent` é `ScratchEvent>>name: aString argument: anObjectOrNil`.

Em seguida é testado se o nome do nodo que está no `nodeNameField` já existe no grafo ou não. Caso o nome do nodo já exista no grafo, o novo *value*, já guardado na variável `value`, é introduzido no grafo na `OrderedCollection` de *values* do nodo com o nome que se encontra no `nodeNameField`. Esta introdução de um novo *value* na `OrderedCollection` de um nodo já existente é feita através do método `ScratchGraph1Morph>>addValueTo:nodeName value:aValue`.

Caso o nome do nodo no `nodeNameField` ainda não exista no grafo, é introdu-

zido um novo nodo já com o novo *value* definido pelo utilizador introduzido na sua `OrderedCollection` de *values*. Isto é feito invocando o do método `ScratchGraph1Morph>>addNode:nodeName value:aValue`.
Por fim, o nodo com o nome que se encontra no `nodeNameField` é apresentado no visualizador como nodo atual através do método `ScratchGraph1Morph>>showNode: nodeName` seja ele um nodo novo ou um que já existia previamente no grafo.

Interação com os restantes componentes

O funcionamento dos restantes *submorphs* do visualizador `ScratchGraph1Morph` é definido no método `ScratchGraph1Morph>>mouseDown: evt`. Estes são o `deleteNodeIcon`, o `openAuxPaneIcon` e o `resizeMorph`.
Em primeiro lugar, este método verifica se o visualizador se encontra em modo de edição, e só em caso afirmativo os comportamentos do `openAuxPaneIcon` e do `deleteNodeIcon` estão ativos quando é clicada a sua área. Desta maneira quando é detetado um clique na área do `deleteNodeIcon` e o grafo não é vazio, o nodo atual do visualizador é apagado do grafo. Esta remoção é feita invocando o método `ScratchGraph1Morph>>deleteNodea: nodename` e passando lhe como parâmetro o nodo atual que é calculado com o método `currentNode`.
Quando é detetado um clique na área do `openAuxPaneIcon`, é testado se o painel auxiliar `auxPane` está ativo. Isto é feito enviando a mensagem `owner` ao `auxPane`. Esta mensagem retorna o *owner* de um *morph*. Caso a resposta seja `nil` o `auxPane` não pertence aos *submorphs* de nenhum *morph* logo encontra-se fechado. Neste caso como o painel auxiliar se encontra fechado o método `mouseDown: evt` adiciona-o e posiciona-o no visualizador trocando também o seu formato para um sinal de cruz. De outro modo, caso o `auxPane` se encontre aberto o processo é inverso. O `auxPane` é apagado dos *submorphs* do visualizador e o formato do

`openAuxPaneIcon` é mudado para um sinal de mais.
Depois disto, neste método estão definidas as funcionalidade comuns tanto ao modo de edição como de execução do visualizador. Em seguida, é testado se o evento recebido pelo rato é um clique com o botão direito. Caso isto aconteça é aberto o menu específico para esta situação através do método `ScratchGraph1Morph>>rightButtonMenu` que permite esconder o visualizador de grafos do palco e também exportar o grafo atual para um ficheiro Graphviz.
Caso o evento seja um clique normal do rato no `resizeMorph` no canto inferior esquerdo, a funcionalidade do redimensionamento do visualizador é ativada, podendo o utilizador através do movimento do rato aumentar ou diminuir o tamanho do visualizador. Este controlo de tamanho é feito enquanto o rato esta carregado e se move, sendo controlada pelo método `ScratchGraph1Morph>>mouseMove: evt`. Por fim, se uma área do visualizador vazia for carregada, é invocado o método do *morph* que representa o rato, `HandMorph>>grabMorph: aMorph`, que permite agarrar no visualizador e movimentá-lo.

5.3.3 Histórico e nodo atual

Ao navegar no visualizador através do grafo, este guarda numa `OrderedCollection` com o nome `navigationHistory` todo o histórico de navegação. Quando um novo nodo vai ser apresentado em seguida como nodo atual do visualizador o seu nome é adicionado à primeira posição da `OrderedCollection` do histórico neste método. Isto é executado pelo método `ScratchGraph1Morph>>showNode: nodeName`. Esta inserção no histórico só acontece se o novo nodo a ser adicionado for distinto do nodo atual que se encontra na primeira posição do histórico. Também quando um nodo é removido do grafo, através do método `ScratchGraph1Graph>>deleteNode: nodeName`, todas ocorrências do seu nome são eliminadas do histórico.
A lista do histórico de navegação é muito utilizada pelo método `ScratchGraph1Gr-`

`aph>>currentNode`. Este método retorna sempre que necessário o nodo que está a ser apresentado no visualizador e também é usado para devolver o nodo que será apresentado seguidamente à remoção do nodo atual.
Este método, para calcular o nodo atual, em primeiro lugar verifica se existem elementos no histórico de navegação. Caso existam devolve o seu primeiro elemento. Este é o procedimento que acontece para a maioria dos casos de uso deste método. Os restantes procedimentos do método são para as situações em que o nodo atual é removido e já não existem elementos no histórico de navegação para calcular o próximo nodo a ser apresentado. Quando isto acontece o próximo passo é devolver o *edge* com menos peso do nodo atual que foi apagado. Por último, caso não existam elementos no histórico e o nodo que foi removido não tenha *edges* nenhuns, o método `currentNode` devolve como nodo atual um nodo qualquer dos nodos restantes do grafo.

5.3.4 Operações sobre o grafo

No protocolo `graph ops` estão reunidas todos os métodos que acedem ou alteram a informação do grafo guardado na variável `graph` do visualizador. Os métodos deste protocolo invocam métodos definidos na classe `Graph` para manipular diretamente o grafo e são utilizados tanto pelo visualizador como pelos blocos. São utilizados pelo visualizador para quando a manipulação é direta no palco por parte do utilizador e são utilizados pelos blocos quando a manipulação é feita com base nos blocos.
Na estrutura genérica dos grafos, definida na classe `Graph`, cada nodo está associado apenas a um *value* que pode ser um objeto qualquer. No grafo definido no visualizador este objeto é sempre uma `OrderedCollection`. Isto permite que os nodos do grafo do visualizador possam ter mais do que um *value*, porque são guardados na `OrderedCollection` que se encontra sempre no *value* de cada nodo da

classe `Graph`. Devido a este facto os métodos desta categoria tratam o *value* de um nodo da classe `Graph` sempre como uma `OrderedCollection` e é aí que guardam e acedem a cada *value* de um nodo. Mesmo na introdução de um novo nodo, é introduzida de imediato uma `OrderedCollection` vazia no *value* associado ao seu nome na classe `Graph`.

Muitos dos métodos aqui definidos têm implementado um sistema de reconhecimento e alerta de erros de acesso ao grafo. Estes erros só podem acontecer através dos blocos do grafo. Por exemplo, o bloco de grafos que apaga um *edge* de um nodo corre o método `ScratchGraph1Morph>>deleteEdge: nodeName to: nodeName2`, recebendo este método como argumento as *strings* colocadas pelo utilizador no bloco. O método verifica se existe algum nodo no grafo com o nome da *string* do primeiro argumento do método, e se não existir invoca o método `error`. Caso exista o nodo no grafo, em seguida o método testa se existe algum *edge* do nodo no primeiro parâmetro com o nome do segundo argumento do método. Se não existir invoca também o método `error`.

Este método `error` altera a variável `lastActivityError` para o valor `true`. Quando isto acontece o método `updateBorder` altera a borda do visualizador para a cor vermelha por um momento 5.13, sinalizando um erro ao utilizador.

Figura 5.13: Notificação de erro

5.3.5 Operações com as células de *values* e *edges*

Nos protocolos `values ops` e `edges ops` do `ScriptableScratchMorph` estão agrupados os métodos que estão associados ao acesso e criação das células dos `valueMorphs` e dos `edgeMorphs` que guardam os *morphs* das células dos *values* e os *edges* do nodo atual no visualizador. Existem seis métodos em cada um dos protocolos sendo que cada um deles num protocolo tem um equivalente com a mesma função no outro. Os métodos que existem no protocolo `edge ops` são os seguintes: `eClear`, `eContents`, `eCreateCell: aString`, `eIndexOfCell: aCell`, `eInsertLine: aString at: aNumber` e `eLineCount`.

- `eClear` tem a função de limpar todas as células da `OrderedCollection edgeMorphs`;
- `eContents` devolve uma `OrderedCollection` com o conteúdo de todas as células dos `edgeMorphs`;
- `eCreateCell: aString` cria e devolve uma nova célula combinando os *morphs* representados na figura 5.7, e dá-lhe o conteúdo igual à *string* recebida como parâmetro
- `eIndexOfCell: aCell` devolve o índice na `OrderedCollection edgeMorphs` da célula recebida como parâmetro
- `eInsertLine:aString at:aNumber` cria uma nova célula com a *string* recebida como parâmetro invocando o método `eCreateCell: aString` e de seguida insere-a na `OrderedCollection edgeMorphs` de acordo com a posição que é recebida como parâmetro;
- `eLineCount` devolve o número de células que estão neste momento na `OrderedCollection edgeMorphs`.

Os métodos do protocolo `value ops` têm as mesmas funções que os anteriormente descritos mas são relativos à `OrderedCollection valueMorphs` e criam células de acordo com as da figura 5.4. Além disso os métodos deste protocolo `value ops` têm os mesmos nomes que os do protocolo `edge ops` diferindo apenas na primeira letra do método que é um "e"nos `edge ops` e um "v"nos `value ops`.

5.3.6 Inicialização e *layout* do visualizador

A inicialização do visualizador de grafo é feita no método `ScratchGraph1Morph>> initialize`. Em primeiro lugar as características visuais do painel são definidas tais como a cor e a largura da borda. As características visuais em vigor neste painel e do seu painel auxiliar são as mesmas das do visualizador de listas do Scratch. Este facto faz com que este novo visualizador pareça familiar para utilizadores que já conhecem o ambiente Scratch e assim promove uma melhor integração na interface.
Em seguida são inicializadas todas as variáveis de instancia do Scratch que não serão *morphs*. São inicializadas então as variáveis: `graph`, `formerNode`, `fileName`, `editMode`, `navigationHistory`, `edgeMorphs`, `valueMorphs`, `indiceMorphs`, `numberMorphs` e `lastActivityError`.

- Na variável `graph` é inicializado o grafo que vai conter toda informação numa instancia da classe `Graph`.
- A variável `valuesActive` é um valor booleano que define se as células que representam os *values* do nodo atual têm a funcionalidade de botão ativa ou não.
- A variável `edgesActive` é um valor booleano que define se as células que representam os *edges* do nodo atual têm a funcionalidade de botão ativa ou não.

- A variável `formerNode` inicializada a `nil` tem a função de guardar o nome do último nodo que foi apresentado no visualizador.

- A variável `editMode` é um valor booleano que representa o modo em que o visualizador se encontra, se em modo de edição ou de execução. O modo por deito do visualizador é o modo de edição e por isso esta variável é inicializada com o valor `true`.

- Na variável `navigationHistory` é inicializada uma `OrderedCollection` que contém ao longo da execução do visualizador o histórico de navegação.

- Na variável `edgeMorphs` é inicializada um `OrderedCollection` que contém, ao logo da execução do visualizador, os `WatcherReadoutFrameMorphs` que representam as células que mostram os diferentes *edges* do nodo atual que está a ser visualizado.

- Na variável `valueMorphs` é inicializada um `OrderedCollection` que contém, ao logo da execução do visualizador, os `WatcherReadoutFrameMorphs` que representam as células que mostram os diferentes *values* do nodo atual que está a ser visualizado.

- Na variável `indiceMorphs` é inicializado um `Array` que contém, ao logo da execução do visualizador, os `StringMorphs` que representam os índices de cada *value* do nodo atual.

- Na variável `numberMorphs` é inicializado um `Array` que contém, ao logo da execução do visualizador, os `StringMorphs` que representam os pesos de ligação do nodo atual a cada um dos seus *edges*.

- A variável `lastActivityError` é um valor booleano que representa um erro de alguma operação no visualizador. Quando esta variável tem o valor `true` a

borda do visualizador torna-se vermelha por alguns momentos. É inicializada com o valor `false`.

Depois de inicializadas as variáveis é invocado o método `ScratchGraph1Morph >>addTitleAndControls`. Este método inicializa todos os componentes do visualizador excetuando o `valuesPane` e o `edgesPane` e seus componentes. Os componentes inicializados neste método, excetuando o `auxPane`, são todos adicionados ao visualizador introduzindo cada um nos seus *submorphs*. Isto é feito para cada um dos componentes utilizando o método `Morph>>addMorph: aMorph` que introduz o *morph* recebido como parâmetro nos *submorphs* de outro. O `auxPane` não é adicionado neste momento para que inicialmente esteja fechado.
Em seguida no método `initialize` são invocados os métodos `ScratchGraph1Morph>>addEdgesPane` e `ScratchGraph1Morph>>addValuesPane` que inicializam e adicionam aos *submorphs* do visualizador os dois `ScrollFrameMorph2`, `edgesPane` e `valuesPane`. Na inicialização destes *morphs* é introduzido no conteúdo de cada um deles um `Morph` genérico e transparente que irá conter os conteúdos de cada um.
No `valuesPane` os conteúdos serão as células que representam cada *value* do nodo atual e também os `StringMorphs` que representam os índices de cada *value*, Todas estas células que representam os *values* do nodo atual estão armazenados na `OrderedCollection valueMorphs` e os `StringMorphs` que representa os índices desses *values* estão armazenados no `Array indiceMorphs`.
No caso do `edgesPane` os conteúdos serão as células que representam cada *edge* do nodo atual e também os `StringMorphs` que representam os pesos de ligação do nodo atual a cada *edge*. Todas estas células que representam os *edges* do nodo atual estão armazenados na `OrderedCollection edgeMorphs` e os `StringMorphs` que representa os pesos desses *edges* estão armazenados no `Array numberMorphs`.
Seguidamente no método `initialize` o visualizador é dimensionado para o ta-

manho inicial de 150x150 e os seus componentes posicionados através do método `Scratch- Graph1Morph>>extent: aPoint`. Este método é um método sobreposto da implementação original `Morph>>extent: aPoint` que faz o redimensionamento de um *morph*. O método sobreposto invoca o método original de `extent: aPoint` para dimensionar o visualizador e invoca de seguida o método `ScratchGraph1Morph>> fixLayout` que posiciona os *submorphs* do visualizador nos seus locais apropriados.

Em primeiro lugar o método `fixLayout` invoca o método `StringMorph>>fitContents` no `titleMorph`. Este método redimensiona o `StringMorph` para acomodar o texto que lhe é associado. O `titleMorph` é posicionado no topo centro do visualizador. De seguida é invocado novamente o método `fitContents` mas no *morph* `titleNodeMorph`. Caso o grafo a representar seja vazio, ou o visualizador esteja em modo de execução o `titleNodeMorph` é representado no centro mesmo por baixo do `titleMorph`. Caso o grafo não seja vazio e o visualizador esteja em modo de edição o `titleNodeMorph` é posicionado de maneira a que a sua combinação com o `nodeMenu` e `deleteNodeIcon` esteja centrada. Em seguida o `nodeMenu` é posicionado à direita do `titleNodeMorph` e o `deleteNodeIcon` à direita deste último. Abaixo destes *morphs* é posicionado o `valuesPane` e logo a seguir o `edgesPane`. Depois o `openAuxPaneIcon` é colocado no canto inferior esquerdo e o `resizeMorph` no canto direito. Depois disto o `addEdgeMorph` é colocado ao lado do `openAuxPaneIcon`. De seguida é invocado novamente o método `fitContents` mas para o `countMorph` e depois é definida a sua posição central no fundo do painel e também o seu conteúdo.

De seguida são invocados os métodos `ScratchGraph1Morph>>updateTitleNode`, que atualiza o conteúdo do `titleNodeMorph`, `ScratchGraph1Morph>>updateValues` que atualiza os conteúdos do `valuesPane` e `ScratchGraph1Morph>>updateEdges` que atualiza os conteúdos do `edgesPane`.

Voltando ao método `initialize` o último método invocado é o `ScratchGraph1Morph>>showCurrentNode` que mostra o nodo atual no visualizador. Este método também será explicado mais a frente.

5.3.7 Métodos de *stepping*

Existem cinco métodos pertencentes ao *morph* `ScratchGraph1Morph` que, por necessidade, são executados periodicamente utilizando o mecanismo de *stepping* do Morphic. Para isso estes são invocados dentro do método `Scratchgraph1-Morph>>step`. Estes cinco métodos métodos são: `updateTitle`, `updateBorder`, `updateCountMorph`, `updateViewMode` e o `updateTitleNode` já referido anteriormente aparecendo no método `fixLayout`.

O método `updateBorder` testa a cada invocação qual o valor da variável de erro do visualizador, `lastActivityError`. Caso o seu valor seja `false` a cor da borda do painel mantém-se no estado normal, mas caso o seu valor tenha sido mudado para `true` a cor da borda do visualizador passa à cor vermelha por um momento, sinalizando assim um erro. A mudança de cor da borda do painel é feita através da invocação do método `BorderedMorph>>borderColor: colorOrSymboOrNil`. De seguida é atribuída à variável o valor `false` novamente.

O método `updateCountMorph` a cada invocação cria a *string* correspondente ao `countMorph`, representado em 5.2, e tem também a função de o manter centrado na janela do visualizador. Por exemplo quando acontece um redimensionamento da janela do visualizador pelo utilizador, este método garante que o `countMorph` se mantém sempre centrado calculando a todo o momento onde é a sua posição correta.

O método `updateTitle` a cada invocação acede ao nome do seu *owner* e ao nome da variável do `ScratchGraph1Morph`, de seguida constrói uma *string* com o conteúdo no formato do titulo e testa se esta *string* é igual ao conteúdo atual do

`titleMorph`. Caso seja diferente o conteúdo do `titleMorph` é alterado e o *layout* do visualizador é atualizado com uma invocação de `fixLayout`. Este método é necessário para atualizar o titulo quando o nome do **Sprite** ao qual este grafo pertence é alterado.

O método `updateTitleNode` a cada invocação verifica se existem nodos no grafo. Se não existirem o grafo está vazio, é criada uma *string* '**no nodes**' e o `nodeMenu` e `deleteNodeIcon` são apagados do visualizador. Caso existam nodos no grafo é criada uma *string* '**node:**' e caso o visualizador estiver no modo de execução a esta *string* é adiciono o nome do nodo atual. De outro modo se o visualizador estiver no modo de edição e se o `nodeMenu` não estiver ativo é invocado o método `ScratchGraph1Morph>>addNodeMenu` que adiciona o `deleteNodeIcon` e o `nodeMenu` ao visualizador. Esta situação acontece quando o grafo está vazio e é adicionado um novo nodo ao grafo.

Depois disto este método testa se a *string* criada é diferente do conteúdo atual do `titleNodeMorph` e em caso afirmativo a *string* de conteúdo deste *morph* é alterada. Por fim é invocado o método `fixLayout` para atualizar o *layout* do visualizador. Estes são os métodos que são invocados periodicamente pelo método `step`.

O método `updateViewMode` a cada invocação altera, se necessário, o modo de apresentação do visualizador do grafo consoante o modo de visualização em que se encontra o ambiente Scratch: modo de apresentação de projeto 3.8, ou modo de contrução 2.1. Esta atualização do modo de apresentação do visualizador de grafo é necessária devido a que quando o Scratch se encontra em modo de apresentação de projetos, cada visualizador de grafo presente no palco tem de se encontrar em modo de execução. Também quando o modo do ambiente Scratch é alterado de modo de apresentação para modo de construção cada visualizador de grafos é alterado para modo de edição.

Os dois modos de visualização do Scratch correspondem a dois *morphs* dife-

rentes que incorporam todos os *submorphs* visíveis nesse modo. O modo de construção do Scratch, 2.1, é representado pelo *morph* `ScratchFrameMorph` e o modo de apresentação de projetos do Scratch, 3.8, é representado pelo *morph* `OffscreenWorldMorph`.

Em cada ciclo de `stepping` o método `updateViewMode` verifica qual é o modo de visualização em que o Scratch se encontra enviando a mensagem `root` ao visualizador de grafos. Desta maneira é possível saber qual é o *morph* raiz da cadeia de *owners* do visualizador. Este *morph* raiz será `ScratchFrameMorph` ou `OffscreenWorldMorph` consoante o modo de visualização do ambiente Scratch nessa altura.

Caso o *morph* raiz seja `OffscreenWorldMorph` o ambiente encontra-se em modo de apresentação. Aqui, caso o visualizador de grafos se encontre em modo de edição, o método `updateViewMode` altera o visualizador para modo de execução.

Caso o *morph* raiz seja `ScratchFrameMorph` o ambiente encontra-se em modo de construção. Aqui este método verifica se na sua ultima invocação o ambiente se encontrava em modo de apresentação. Esta verificação é feita analisando a variável `lastRoot` que guarda a cada invocação do método o *morph* raiz do ambiente. Caso a variável `lastRoot` contenha o valor `OffscreenWorldMorph` isto significa que na ultima iteração do método `updateViewMode` o ambiente Scratch se encontrava em modo de apresentação. Assim o ambiente Scratch passou de apresentação para construção e por conseguinte o método `updateViewMode` altera o modo do visualizador de grafos de execução para edição.

No fim do método a variável `lastRoot` é atualizada com o nome do *morph* raiz atual que representa o modo em que se encontra o Scratch neste momento.

5.3.8 Apresentação do nodo atual

O método `ScratchGraph1Morph>>showCurrentNode` é responsável por atualizar o visualizador fazendo as alterações necessárias para a apresentação do nodo atual. Em primeiro lugar este método invoca `ScratchGraph1Morph>>changeNode` que verifica se o modo de visualização foi alterado e em caso afirmativo faz as alterações necessárias ao visualizador.

De seguida, o método `showCurrentNode` calcula, através do método `currentNode`, o nodo atual que vai ser apresentado. O próximo passo é adicionar o nodo atual, que será apresentado, ao histórico `navigationHistory`, isto acontece apenas se o grafo não for vazio e se o histórico estiver vazio. O nodo só é adicionado aqui nas situações em que é inserido num grafo vazio. Nos restantes casos os novos nodos são introduzidos no histórico no método `showNode: nodeName`.

De seguida, as células que representam os *values* e *edges* do nodo atual são eliminadas através da invocação dos métodos `ScratchGraph1Morph>>vClear` e `ScratchGraph1Morph>>eClear` que inicializam as `OrderedCollections` onde estas células são salvas, `valueMorphs` e `edgeMorphs`. Depois disto, caso o nodo atual calculado por `currentNode` tiver o valor `nil`, significando que o grafo está vazio, é invocado o método `ScratchGraph1Morph>>autoExportGraph` e a execução do método pára aqui.

A partir deste momento é adquirido que o grafo não é vazio e por isso o conteúdo do `nodeMenu` é mudado para o valor do nodo atual calculado em `currentNode`. Em seguida, é criada uma nova `SortedCollection` com os valores de todos os *edges* que estão no grafo no dicionário de `edges` do nodo atual. Para isto é convertido o `Dictionary` de *edges* numa `SortedCollection` ordenada por pesos através do método `ScratchGraph1Morph>>orderDictToSortedCollction: aDict`. Esta nova `SortedCollection` de *edges* é percorrida, e para cada um dos *edges* é inserido na `OrderedCollection edgeMorphs` um *morph* que representa a célula que

vai mostrar esse *edge* no `edgePane` do visualizador.

Estas células são inseridas no painel dos *edges* pela ordem estipulada na `SortedCollection`, e cada uma é inserida pelo método `eInsertLine: aString at: aNumber`. Este método antes da inserção da célula na `OrderedCollection edgeMorphs` cria o *morph* que a representa através do método `eCreateCell: aString`.

Depois da inserção das células dos *edges*, é invocado o método `ScratchGraph1Morph >>updateEdges` que tem a função de as posicionar no local apropriado do `edgesPane` ou então de inserir aqui um o `emptyEdgesMorph` no caso não existirem células a colocar. Este método também invoca o método `ScratchGraph1Morph>> updateNumbers: rightX` que posiciona no `edgesPane` os pesos de ligação do nodo atual a cada *edge*. Depois da inserção destes componentes no `edgesPane`, a *scrollBar* vertical do `ScrollFrameMorph2` correspondente ao `edgesPane` é iniciada ao valor 0, isto só acontece caso o nodo que está a ser construído no visualizador seja diferente do anteriormente apresentado.

Depois de inserir os *edges* do nodo atual, são inseridos os seus *values*. Para isso, é recolhida a `OrderedCollection` de *values* do nodo atual no grafo. Esta `OrderedCollection` de *values* é percorrida, e para cada *value* é feito um teste de tipo. Caso o *value* for um evento, a *string* correspondente ao seu nome é adicionada a uma variável e precedida de '**evt:**'. E caso o *value* seja do tipo `String` ou `UTF8` (os dois tipos de *strings*) então a *string* deste *value* é adicionada a uma variável e precedida de '**txt:**'. Assim no `valuesPane` é possível distinguir as células de *values* que correspondem a eventos ou a *strings*.

Depois de fazer esta transformação o ciclo continua e para cada *value* é inserido na `OrderedCollection valueMorphs` um *morph* que representa a célula desse *value*. Esta célula vai mostrar cada *string* representativa de um *value* no `valuePane` do visualizador.

Esta inserção é feita também pela ordem estipulada na `OrderedCollection` atra-

vés do método `ScratchGraph1Morph>>vInsertLine: aString at: aNumber`. Este método, antes da inserção da célula na `OrderedCollection valueMorphs`, cria o *morph* que representa essa nova célula através do método `ScratchGraph1Morph>>vCreateCell: aString`.
Depois da inserção das células dos *values*, é invocado o método `ScratchGraph1Morph>>updateValues` que tem a função de as posicionar no `valuesPane` ou então de inserir aqui um o `emptyValuesMorph` no caso de não existirem células a colocar. Este método também invoca o método `ScratchGraph1Morph>>updateIndices: rightX` que posiciona no `valuesPane` os índices que correspondem a cada *value*.
Depois da inserção destes componentes no `valuesPane`, a *scrollBar* vertical do `ScrollFrameMorph2` que lhe corresponde é iniciada ao valor 0, isto só acontece também caso o nodo que está a ser construído no visualizador seja diferente do nodo anteriormente apresentado.
Depois de inseridos todos os componentes relativos ao nodo atual calculado, este encontra-se completamente representado no visualizador.
Por fim, a variável `formerNode` toma o valor do nodo atual, e é invocado o método `fixLayout` e `ScratchGraph1Morph>>autoExportGraph`.

5.3.9 Exportação de grafos para ficheiros *.dot*

É possível também exportar um grafo para um ficheiro *.dot* que pode ser lido pelo programa Graphviz. A partir deste ficheiro, o Graphviz cria uma imagem do grafo num formato global. Esta funcionalidade pode ser acedida através de um bloco ou do menu do visualizador. Este pode ser aberto em Windows com um clique com o botão direito do rato numa área livre do visualizador e com ALT+Clique em MAC. Ao clicar nesta opção, é invocado método `ScratchGraph1Morph>>exportGraph`. Este método, em primeiro lugar abre uma janela que pergunta ao utilizador o nome que este pretende dar ao ficheiro *dot*. Em seguida é criado o ficheiro com o nome

apropriado com o método de criação `StandardFileStream>>newScratchFileNamed: fileName`.

Depois disto o grafo do visualizador é todo percorrido. E à medida que isto acontece a informação presente no grafo é introduzida no ficheiro *.dot* em forma de *strings* sempre atendendo às regras de formatação deste tipo de ficheiros. Também é de assinalar que o nodo atual que está a ser apresentado pelo visualizador é introduzido no ficheiro de maneira a que seja exposto com uma cor azul clara pelo Graphviz, diferenciando-o assim dos restantes.

Para introduzir *strings* no ficheiro utiliza-se o método `StandardFileStream>>nextPutAll: aString`. Por fim o ficheiro é fechado, com o método `StandardFileStream>>close`, contendo neste momento a informação codificada em *.dot* do grafo.

Este método também atribui à variável de instancia do visualizador `fileName` o nome do ficheiro *.dot* onde o grafo é guardado. Isto é feito para que a cada vez que é apresentado um nodo no visualizador com o método `showCurrentNode` o ficheiro .dot seja atualizado. Esta atualização é feita com a invocação do método `autoExportGraph` já referido acima. A diferença deste método para com o `exportGraph` é que este não questiona o nome do ficheiro ao utilizador. Em vez disto ele cria um ficheiro *.dot* por cima do ficheiro com o nome que está guardado na variável `fileName`, isto invocando o método `StandardFileStream>>oldScratchFilesNamed: fileName`. Assim é possível ter sempre o ficheiro *.dot* atualizado.

Na figura 5.14 está representado o grafo de nome 'cidades' que foi exportado para um ficheiro *.dot* e aberto pelo programa Graphviz. O visualizador representado na figura 5.2 representa o nodo atual deste grafo.

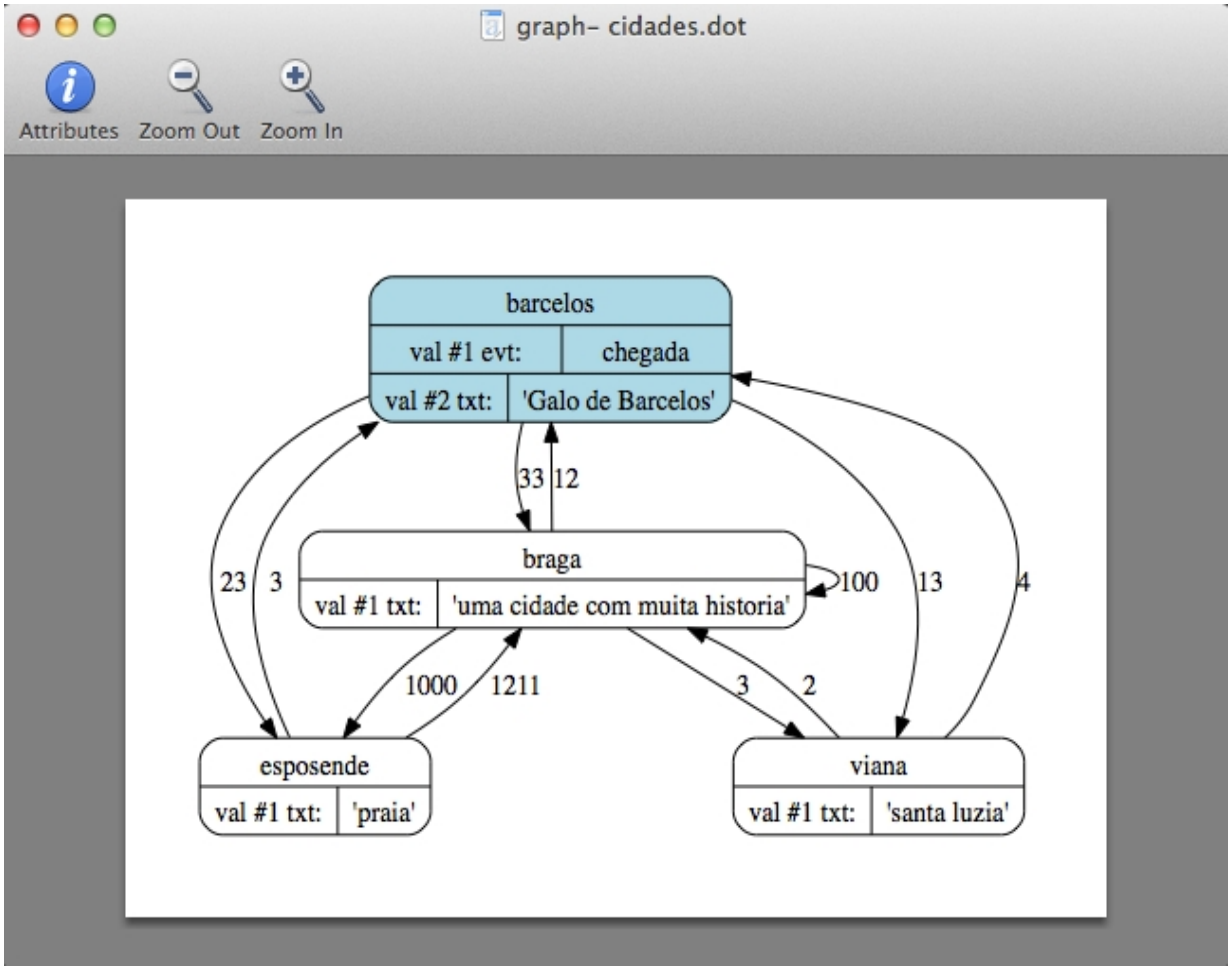


Figura 5.14: Grafo representado no Graphviz

5.4 Blocos de grafo

5.4.1 Especificação de um bloco

Cada bloco Scratch é construído a partir de uma especificação contida num *array* e de um método associado a esse bloco que dita o seu comportamento. Estas especificações podem estar presentes numa de três classes: `ScriptableScratchMorph`, `ScratchStageMorph` e `ScratchSpriteMorph`. A classe `ScratchStageMorph` é a classe que define o **Stage** do Scratch enquanto que a classe `ScratchSpriteMorph`

representa um **Sprite**.

Estas duas classes são ambas subclasses de `ScriptableScratchMorph`, 5.15.

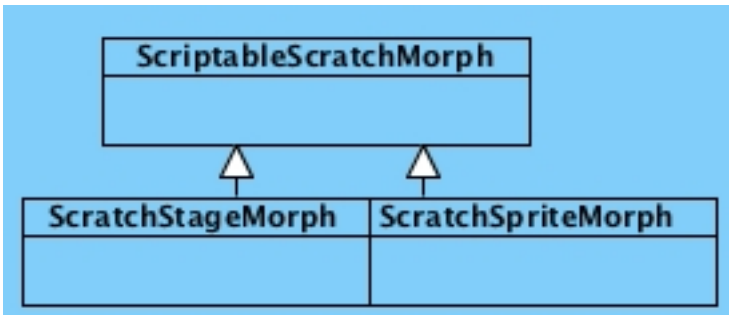


Figura 5.15: Classe ScriptableScratchMorph e suas subclasses

Em cada uma destas três classes existe um método de classe, `blockSpecs`, onde estão todas as especificações de blocos para essa classe. Assim, a classe onde o bloco está definido vai ditar a sua disponibilidade para ser usado pelo *Stage*, pelos *Sprites* ou por ambos. Caso a especificação de um bloco esteja definida na classe `ScratchSpriteMorph` esse bloco apenas estará disponível para ser utilizado pelos *Sprites*, do mesmo modo caso a especificação de um bloco esteja definida na classe `ScratchStageMorph` esse bloco apenas estará disponível para ser utilizado pelo *Stage*, e para que um bloco seja disponível para *Sprites* e *Stage* a sua especificação tem de estar definida em `ScriptableScratchMorph`.

No Scratch as categorias de blocos que estão definidas nas diferentes classes são as seguintes:

- `ScriptableScratchMorph` - **Control**, **Operators**, **Sound**, **Motor**, **Variables** e **List**;
- `ScratchStageMorph` - **Sensing**, **Looks** e **Pen** relativos ao **Stage**;
- `ScratchSpriteMorph` - **Motion**, **Sensing**, **Looks** e **Pen** relativos aos **Sprites**.

Assim com o objetivo de tornar os novos blocos da nova categoria **Graph** disponíveis tanto para o **Stage** como para os **Sprites** as suas especificações foram introduzidas na classe `ScriptableScratchMorph`, no método de classe `blockSpecs`. O método `blockSpecs` contém as especificações de blocos no formato de *arrays* sendo cada grupo de especificações de blocos de uma determinada categoria precedido de uma *string* com o nome dessa categoria.
Cada especificação de um bloco tem o seguinte formato:

```
(<block spec string> <block type> <selector> [optional initial argument values])
```

O campo `<block spec string>` contém uma especificação textual que representa o corpo do bloco e a partir desta *string* é criado o `CommandBlockMorph` que vai representar o bloco. Os argumentos de *input* dos blocos são também definidos nesta *string* e podem ser de vários tipos. Estes argumentos são definidos na especificação textual do bloco usando o símbolo `%` seguido de uma letra que define qual o tipo de argumento. Quando o novo bloco `CommandBlockMorph` é criado a partir desta especificação é utilizado o método `CommandBlockMorph>>uncoloredArgMorphsFor: specString` onde estão as correspondências entre cada letra e o tipo de argumento que estará no bloco.

Figura 5.16: Bloco letter( ) of [ ]

Dando como exemplo o bloco **letter( ) of [ ]** representado na figura 5.16, a sua especificação completa define-se da forma apresentada na figura 5.17.

```
('letter %n of %s'                    r    letter:of: 1 'world')
```

Figura 5.17: Especificação do bloco letter( ) of [ ]

Para este bloco a sua especificação textual é `'letter %n of %s'`. Os dois argumentos da especificação estão definidos no método `uncoloredArgMorphsFor: specString` como dois `ExpressionArgMorphs`: `%n` para entrada valores numéricos e `%s` para entrada de *strings*. Estes dois *morphs* são introduzidos no bloco criado como seus *submorphs*.

```
CommandBlockMorph>>uncoloredArgMorphFor: t1
...
$s = t2 ifTrue:[^ExpressionArgMorph new stringExpression:''].
...
$n = t2 ifTrue:[^ExpressionArgMorph new numExpression:'10'].
```

Existem para os blocos grafo mais tipos de argumentos novos. Estes novos tipos de argumento serão explicados mais a frente.

O campo `<block type>` da especificação de um bloco contém o tipo de formato do bloco que vai ser criado e é representado por um carácter. Por exemplo, a *flag* do bloco anterior na figura 5.16 é `r` porque este é um bloco **Reporter** arredondado. Os tipo de *flags* existentes mostram-se na figura 5.18.

```
Explanation of flags:
    -   no flags
    b   boolean reporter
    c   c-shaped block containing a sequence of commands (always special form)
    r   reporter
    s   special form command with its own evaluation rule
    t   timed command, like wait or glide
    E   message event hat
    K   key event hat
    M   mouse-click event hat
    S   start event hat
    W   when «condition» hat (obsolete)"
```

Figura 5.18: *Flags* dos tipos de blocos

Em relação aos novos blocos de grafo, estes são apenas de três tipos: **Stack**, **Reporter** arredondado e **Reporter** booleano ou hexagonal. Ou seja os blocos de grafo utilizam apenas as *flags*: `-`,`r` e `b`.

O campo `<selector>` da especificação de um bloco especifica o nome do método que será executado por este bloco. Define assim o seu comportamento. Por exemplo no bloco anterior da figura 5.16 o `selector` associado a este é `letter:of:`. Este método devolve a letra da *string* do segundo argumento que está na posição que é definida no primeiro argumento do método. E este é o comportamento do bloco da figura. Nos grafo todos os métodos que definem comportamentos dos novos blocos estão definidos num novo protocolo de `ScriptableScratchMorph` chamado `graph ops`.

Por último, o campo `[optional initial argument values]` da especificação de um bloco define os argumentos por omissão de um bloco. Este campo pode estar presente ou não. No caso do bloco anterior este campo existe e tem os valores `1 'world'`, que são os valores por omissão deste bloco e são vistos na figura 5.16.

Outro local onde os argumentos por omissão podem estar definidos é no método ScriptableScratchMorph»defaultArgsFor: blockSpec. Este método é utilizado quando o `ScriptableScratchMorph` cria um novo bloco a partir de uma especificação e tem a função de definir quais os argumentos por omissão desse bloco. Devolve um *array* com os argumentos por omissão do bloco consoante do método `selector` associado a esse bloco. Por exemplo para o bloco da figura 5.16 os argumentos por omissão estão definidos da seguinte forma:

```
ScriptableScratchMorph>>defaultArgsFor: blockSpec
...
#letter:of: = sel ifTrue: [
        defaultArgs _ Array with: 1 with: 'world' localized].
```

5.4.2 Especificações e valores de *input* de grafos

Os novos blocos de grafo estão definido no método `blockSpecs` da classe `ScriptableScratchMorph` numa categoria com o nome `graph`. As suas especificações estão colocadas depois dos blocos da categoria `list` das listas sendo a ultima categoria de blocos deste método.

```
'list'
    ('add %s to %L'                 -   append:toList: 'thing')
    -
    ('delete %y of %L'              -   deleteLine:ofList: 1)
    ('insert %s at %i of %L'        -   insert:at:ofList: 'thing' 1)
    ('replace item %i of %L with %s'    -   setLine:ofList:to: 1 'list' 'thing')
    -
    ('item %i of %L'                r   getLine:ofList: 1)
    ('length of %L'                 r   lineCountOfList:)
    ('%L contains %s'               b   list:contains: 'list' 'thing')
'graph'
    ('Gr:%Z show n:%z'              -   showGraph:node:)
    ('Gr:%Z change mode'            -   changeMode:)
    ('Gr:%Z export to .dot file'    -   exportToDot:)
    ('Gr: %Z currentNode'           r   currentNodeOf:)
    -
    ('Gr:%Z add/clear n:%z'         -   addNodeToGraph:node: )
    ('Gr:%Z add/clear n:%z type %V value %X'
                                    -   addNodeToGraph:node:type:value: )
    ('Gr:%Z delete n:%z'            -   deleteFromGraph:node:)
    ('Gr: %Z nodes in list%L'       -   nodesOfGraph:list: '' '')
    ('Gr:%Z change name n:%z to n:%A'
```

Figura 5.19: Categoria de especificações de blocos grafo

Nos blocos de grafo existem novos `ArgMorphs` de *input* nos blocos. Alguns deles são novos *morphs* criados especificamente para permitir algumas restrições na entrada de valores e também algumas funcionalidades novas nos blocos necessárias. Existem oito tipos de *morphs* de *input* que correspondem a oito caracteres diferentes nas especificações textuais: `%Z`, `%z`, `%A`, `%j`, `%J`, `%q` `%V` e `%w`. Estes novos `ArgMorphs` de *input* dos blocos estão inicializados no método `uncoloredArgMorphsFor: specString`.

O `ArgMorph`, no bloco na figura 5.20, corresponde à letra `Z` capitalizada e é um

`ChoiceArgMorph` que tem o método `ScriptableScratchMorph>>graphVarMenu` como seu `selector` de opções. Este *morph* é usado em todos os blocos de grafo e corresponde a um menu de escolha dos grafo existentes no Scratch. Este argumento determina sempre qual o grafo alvo de um bloco e é neste grafo que o bloco vai executar a sua função.

Figura 5.20: ChoiceArgMorph destacado num bloco

O `ArgMorph`, no bloco da figura 5.21, corresponde à letra `z` não capitalizada e é um `NodeArgMorphWithMenu` sendo a sua função a apresentação num menu *dropdown* dos nodo do grafo escolhido no primeiro argumento do bloco. O utilizador pode escolher um nodo através deste menu e também pode introduzir um nodo por escrita no teclado. Este *morph* é novo no Scratch e é uma sub-

Figura 5.21: NodeArgMorphWithMenu destacado num bloco

classe do *morph* `GraphExpressionArgMorph` explicado na secção 5.3.2. No método `uncoloredArgMorphsFor: specString` quando o `NodeArgMorphWithMenu` é inicializado para o código correspondente à letra `z` é utilizado o método `permitedChars: aString` da sua superclasse para definir a lista a caracteres que são possíveis introduzir neste campo.

Estes caracteres são todas as letras do abecedário capitalizadas e não capitalizadas

e também números e espaços.

Quando o seu menu é carregado este *morph* acede à primeira posição do seu *owner* à primeira posição que corresponde sempre ao nome do grafo alvo do bloco. A partir deste nome procura o visualizador do grafo que lhe correspondente através de um método chamado `NodeArgMorphWithMenu>>getGraph`. Em seguida invoca para este visualizador específico o método já conhecido `ScratchGraph1Morph>>listNodesMenu` que lhe permite criar o menu de todos os nodos do grafo alvo do bloco onde se insere.

Através do mecanismo de *stepping* do Morphic este *morph* atualiza constantemente o seu conteúdo quando na Paleta de blocos (não na aba de **Scripts**). O conteúdo deste *morph*, quando se encontra na Paleta de blocos, é sempre atualizado para o nome do nodo atual do visualizador que corresponde ao grafo alvo do bloco em que este se incute.

O `ArgMorph` correspondente à letra `A` capitalizada é também um `NodeArgMorphWithMenu` mas com o menu de nodos desligado. Isto é feito com a opção `menuOn: false` na sua inicialização. Este *morph* é utilizado apenas num bloco e corresponde ao seu terceiro argumento. O bloco é o da figura 5.22.

Este bloco muda o nome do nodo que está no segundo argumento do bloco para

Figura 5.22: NodeArgMorphWithMenu destacado no bloco

a *string* que está no terceiro argumento. Como o nome do nodo não pode ser trocado por um nome de um nodo já existente o terceiro `GraphMorph` não necessita do menu de nodos ativo.

O `ArgMorph` correspondente à letra `j` não capitalizada é um `EdgeArgMorphWithMenu` que é um subclasse de `NodeArgMorphWithMenu`. Este `ArgMorph` é usado

para introdução de nomes de *edges*. Aparece nos quatro blocos da figura 5.23 e é o último argumento deles.

Figura 5.23: EdgeArgMorphWithMenus destacados nos blocos

O menu que este *morph* apresenta é diferente do da sua superclasse. Este *morph* acede também ao grafo correspondente ao nome que está no primeiro argumento do bloco em que se insere, e invoca o método `ScratchGraph1Morph>>listEdgesMenuOfNode: nodeName` dando-lhe como argumento o conteúdo do segundo argumento do bloco. Este método cria então um menu de opções com todos os *edges* já existentes do nodo que corresponde ao segundo argumento do bloco.
Utiliza também a funcionalidade de *stepping* do Morphic para atualizar o seu conteúdo automaticamente mostrando sempre, quando na Paleta de blocos, um dos *edges* do nodo que se encontra no segundo argumento do bloco neste momento.
O `ArgMorph` correspondente à letra `J` capitalizada corresponde a um `ChoiceArgMorph` com o método `ScriptableScratchMorph>>nodesOrWeightMenu` como seu `selector` de opções. O menu apresentado por este *morph* contém as opções **'Nodes'** e **'Weight'** sendo este utilizado apenas no bloco da figura 5.24.

Figura 5.24: ChoiceArgMorph destacado no bloco

Este bloco é do tipo **Reporter** arredondado, calcula o caminho mais curto entre dois nodos e devolve ou uma *string* com os nodos desse caminho ou o peso desse caminho. O tipo de valor devolvido é definido pelo último argumento do bloco escolhido pelo utilizador no bloco.

O `ArgMorph` correspondente à letra `q` não capitalizada corresponde a um `ChoiceArgMorph` com o método `ScriptableScratchMorph>>typeValueMenu` como seu `selector` de opções. O menu apresentado por este *morph* contém as opções '**All**' e '**Event**' e '**Text**' sendo este utilizado apenas no bloco da figura 5.25 correspondendo ao seu terceiro argumento.

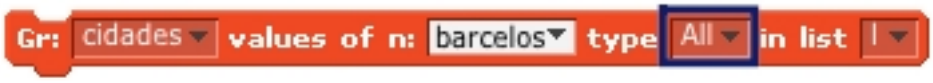


Figura 5.25: ChoiceArgMorph destacado no bloco

Este bloco introduz os *values* de um nodo numa lista Scratch. O terceiro argumento do bloco define que tipo de *values* são introduzidos se apenas eventos, apenas *strings* ou todos.

O `ArgMorph` correspondente à letra `V` capitalizada corresponde a um `ValueTypeMorph` que é uma subclasse de `ChoiceArgMorph` e o seu `selector` é o método `ScriptableScratchMorph>>textOrEventMenu`. Este método cria um menu com as opções '**Text**' e '**Event**'. É utilizado em três blocos. Nos dois blocos da figura 5.26, este `ArgMorph` é utilizado para definir que tipo de *value* vai ser testado nestes blocos **Reporter** booleanos.

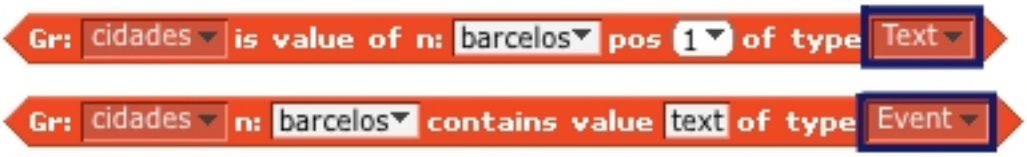


Figura 5.26: ValueTypeMorphs destacados nos blocos

Na figura 5.27, estão representadas os dois estados do outro bloco que utiliza um `ValueTypeMorph`. Este é utilizado para definir o *value* que vai ser introduzido no grafo. Consoante a escolha que é feita neste argumento, o `ArgMorph` que representa o argumento seguinte do bloco é atualizado. Se a escolha que estiver ativa for '**Text**', o `ArgMorph` que está ativo é um `ExpressionArgMorph` para introdução de *strings*. E caso a escolha for mudada para '**Event**', o `ExpressionArgMorph` do quarto argumento do bloco é alterado para um `EventTitlePlusMorph` para escolha de um evento para adicionar.

Figura 5.27: ValueTypeMorphs destacados nos blocos

Esta classe, `EventTitlePlusMorph`, de visualização de eventos também é utilizada no visualizador de grafo para criação de eventos, e o seu funcionamento está explicado na secção 5.3.2. O `ValueTypeMorph` utiliza o mecanismo de *stepping* para verificar se a sua escolha foi mudada. Quando isto acontece altera o `ArgMorph` que está no quarto argumento do bloco em que está inserido para o *morph* apropriado.

`ExpressionArgMorph` caso a escolha seja '**Text**' e `EventTitlePlusMorph` se a escolha for '**Event**'. Esta alteração de `ArgMorphs` na constituição do bloco *owner* do `ValueTypeMorph` é invocada através do método `CommandBlockMorph>>replaceArgMorph: oldMorph by: newMorph`.

Por último o `ArgMorph` correspondente à letra `w` não capitalizada corresponde a um `ListIndexValueMorph` que é uma subclasse de `NodeArgMorphWithMenu`. Este `ArgMorph` é utilizado em sete blocos, permitindo a escolha de um índice de um menu e também a entrada de valores numéricos com algarismos de 0 a 9. É usado para especificar, nos blocos, posições de *values* e *edges* nas listas de um nodo.

Nos quatro blocos da figura 5.28, o menu que este `ArgMorph` apresenta mostra os índices dos *values* que existem no nodo dado como argumento do bloco, e mostra os tipo desses *values*.

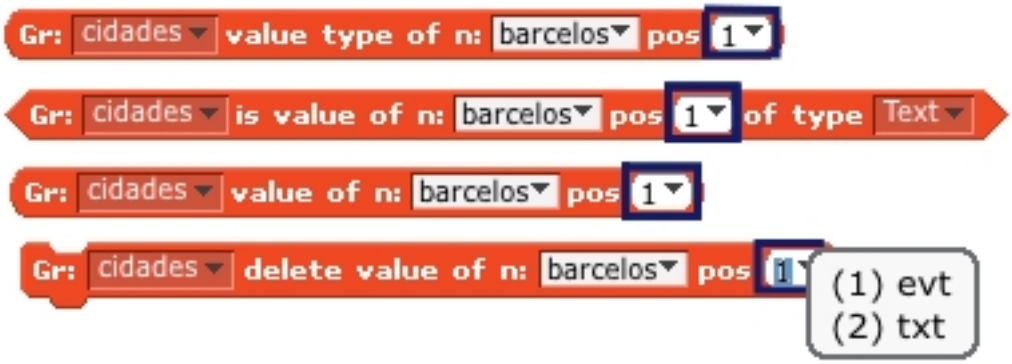


Figura 5.28: ListIndexValueMorphs destacados nos blocos

No bloco da figura 5.29 o menu apresentado contém os índices e tipos dos *values* já existentes no nodo dado como parâmetro e também adiciona uma escolha '**new**' ao menu. Esta escolha coloca no conteúdo do `ArgMorph` o índice logo a seguir ao último índice ocupado. Se na posição do índice escolhido no bloco da `OrderedCollection` de *values* já existir um *value*, este será substituindo no nodo do grafo pelo *value* definido neste bloco. Caso o índice escolhido for correspondente à escolha '**new**', o *value* introduzido será um novo *value* que é colocado no

fim da `OrderedCollection` de *values* do nodo. Por omissão, o valor deste bloco é o da escolha '**new**'.

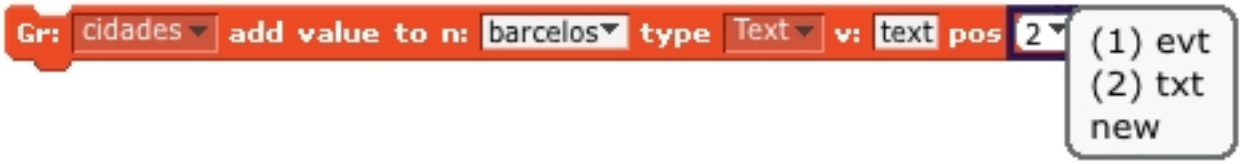


Figura 5.29: ListIndexValueMorph destacado no bloco

No bloco da figura 5.30, o menu apresentado contém os índices dos *values* do nodo dado como parâmetro. Os *values* que são eventos são apresentados no menu através do seu nome precedido de '**e.**', enquanto que para os *values* textuais é apresentado apenas o seu tipo, '**txt**'. A função deste bloco depende do tipo de *value* ao qual se aplica. Caso o *value* que está na `OrderedCollection` de *values* no índice definido pelo utilizador for um evento, este bloco transmite esse evento. Caso o *value* seja textual o bloco faz com que o **Sprite** dono do grafo alvo transmita o valor textual desse *value* através de um balão de texto. De outro modo, quando quando o grafo alvo do bloco não tem um **Sprite** dono específico, todos os **Sprites** em palco transmitem através de balões o texto contido no *value*.

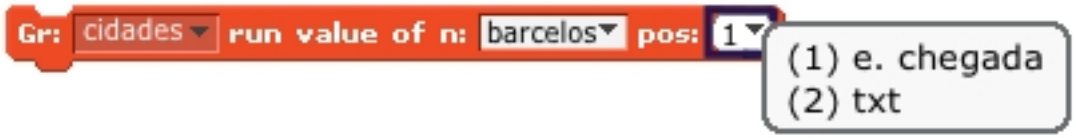


Figura 5.30: ListIndexValueMorph destacado no bloco

Por último no bloco da figura 5.31, o menu do `ListIndexValueMorph` apresenta os índices dos *edges* do nodo dado como argumento. No menu apresenta também o nome de cada um desses *edges*.

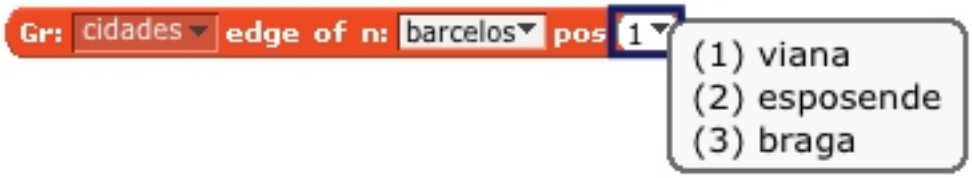


Figura 5.31: ListIndexValueMorph destacado no bloco

Todas estas apresentação diferentes do menu do `ListIndexValueMorph` são definidas no método `ListIndexValueMorph>>mouseDown: evt` reconhecendo qual é o método `selector` do bloco *owner*. A partir da especificação do método `selector` do bloco onde o `ListIndexValueMorph` está inserido, este reconhece em que bloco se encontra e tem um comportamento diferente dependendo para cada um.

5.4.3 Argumentos por omissão

Os métodos `selector` das novas especificações de grafo encontram-se todos referenciados no método `ScriptableScratchMorph>>defaultArgsFor: blockSpec`. Como já referido, este método é usado quando um bloco está a ser construído devolvendo os seus argumentos por omissão.

Por exemplo, o método `selector` do bloco representado na figura 5.25 tem como `selector` na sua especificação `#valuesOfGraph:node:type:list:`. Este `selector` é reconhecido no método `defaultArgsFor: blockSpec` e o `array` de argumentos por omissão do bloco é preenchido da seguinte forma no método:

```
ScriptableScratchMorph>>defaultArgsFor: blockSpec
...
#valuesOfGraph:node:type:list: = t4
      ifTrue: [t2 size >= 4
           ifTrue:
                    [t2 at: 1 put: self defaultGraphName.
                    t2 at: 2 put: 'node'.
                    t2 at: 3 put: 'All'.
                    t2 at: 4 put: self defaultListName]].
...
```

No primeiro argumento é invocado o método `defaultGraphName` para calcular o grafo por omissão do `ScriptableScratchMorph` que está ativo. No segundo argumento é colocada a *string* `'node'` e no terceiro a *string* `All`. Por fim no último argumento é invocado o método `defaultListName` para calcular a lista por omissão do `ScriptableScratchMorph` que está ativo.

5.4.4 Reporters sem janelas de palco

Nos blocos **Reporter** mais simples que têm apenas um argumento o Scratch por omissão cria-os na Paleta de blocos com uma caixa de seleção ao seu lado esquerdo como na figura 3.25. Esta caixa quando selecionada mostra do *Stage* o valor dessa variável. Este comportamento não é necessário aos blocos **Reporter** simples de grafos que são os da figura 5.32.

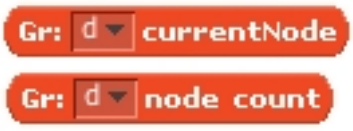


Figura 5.32: Blocos Reporter sem janelas de palco

Para que estas caixas não apareçam na Paleta de blocos os `selectors` destes dois blocos estão adicionados no método `CommandBlockMorph>>canBecomeWatsher`. Este método identifica, a partir dos `selectores`, quais são os blocos **Reporter** simples nos quais esta funcionalidade vai ser anulada.

5.4.5 Métodos associados a blocos

Como já foi dito, todos os `selectors` dos blocos de grafo correspondem a métodos que vão ser executados quando o bloco é ativado. Estes métodos estão todos definido no protocolo `graph ops` da classe `ScriptableScratchMorph`.
Qualquer destes métodos em primeiro lugar verifica se o nome do grafo indicado, sempre no primeiro argumento do método, corresponde a um grafo existente. Em caso afirmativo, o visualizador desse grafo é devolvido pelo método. Este acesso ao visualizador é feito invocando o método `ScriptableScratchMorph>>graphNamed: graphName ifNone: aBlock` que funciona acedendo ao `Dictionary` de grafos de nome `graphs` do **Sprite** ou **Stage** selecionado e devolve o visualizador do grafo que se encontra associado à chave do primeiro parâmetro deste método, correspondendo esta ao nome do grafo. Caso não exista um grafo com este nome no dicionário de grafos então é executado por omissão o bloco de instruções do segundo argumento deste método.
Nos blocos em que são utilizadas listas do Scratch para inserir valores provenientes do grafo, o acesso ao visualizador de listas é feito através de um método semelhante ao descrito em cima para mesmo efeito nos grafos. Este método é o `ScriptableScratchMorph>>listNamed: listName ifNone: aBlock`.
Depois de acedido o visualizador do grafo, os métodos associados aos blocos invocam um método deste visualizador acedido consoante a funcionalidade pretendida para este bloco. Todos os métodos do visualizador que processam acessos ou alterações à estrutura do grafo em si estão no protocolo `graph ops` da classe que

define o *morph* visualizador de grafos, `ScratchGraph1Morph`. Estes métodos são os métodos invocados pela grande maioria dos métodos `selectors` associados aos blocos.

O método do visualizador que é invocado para fazer a exportação de um grafo para um ficheiro Graphviz encontra-se no protocolo `import/export`.

Os cinco métodos restantes não acedem nem alteram o grafo em si. Um deles é o método associado ao bloco **Reporter**, que devolve o nodo atual do grafo. Este bloco tem como método associado o método `currentNode` presente no protocolo `acessing`. Outros dois métodos são os que controlam a ativação e desativação das células de *values* e *edges* do visualizador e são: `edgesActive: aBoolear` e `valuessActive: aBoolear`, também eles presentes no protocolo `accessing`.

Os restantes dois métodos estão no protocolo `private` e processam alterações estruturais no visualizador. Um é o `changeMode` que altera o modo de apresentação do visualizador, e o outro é o método `showNode: nodeName` que permite ver um nodo específico como nodo atual do visualizador.

Em relação aos métodos que estão no protocolo `graph ops` do visualizador, estes por sua vez utilizam métodos da classe `ScratchGraph1Morph`, onde se encontram, para alterações estruturais e de estado do visualizador. Também utilizam métodos da classe `Graph` para alterar e aceder ao conteúdo do seu grafo que se encontra guardado na variável de instancia de nome `graph` do visualizador de grafos.

Na figura 5.33 temos um exemplo dos métodos e classes essenciais utilizados para executar a função do bloco que adiciona um novo nodo ao grafo.

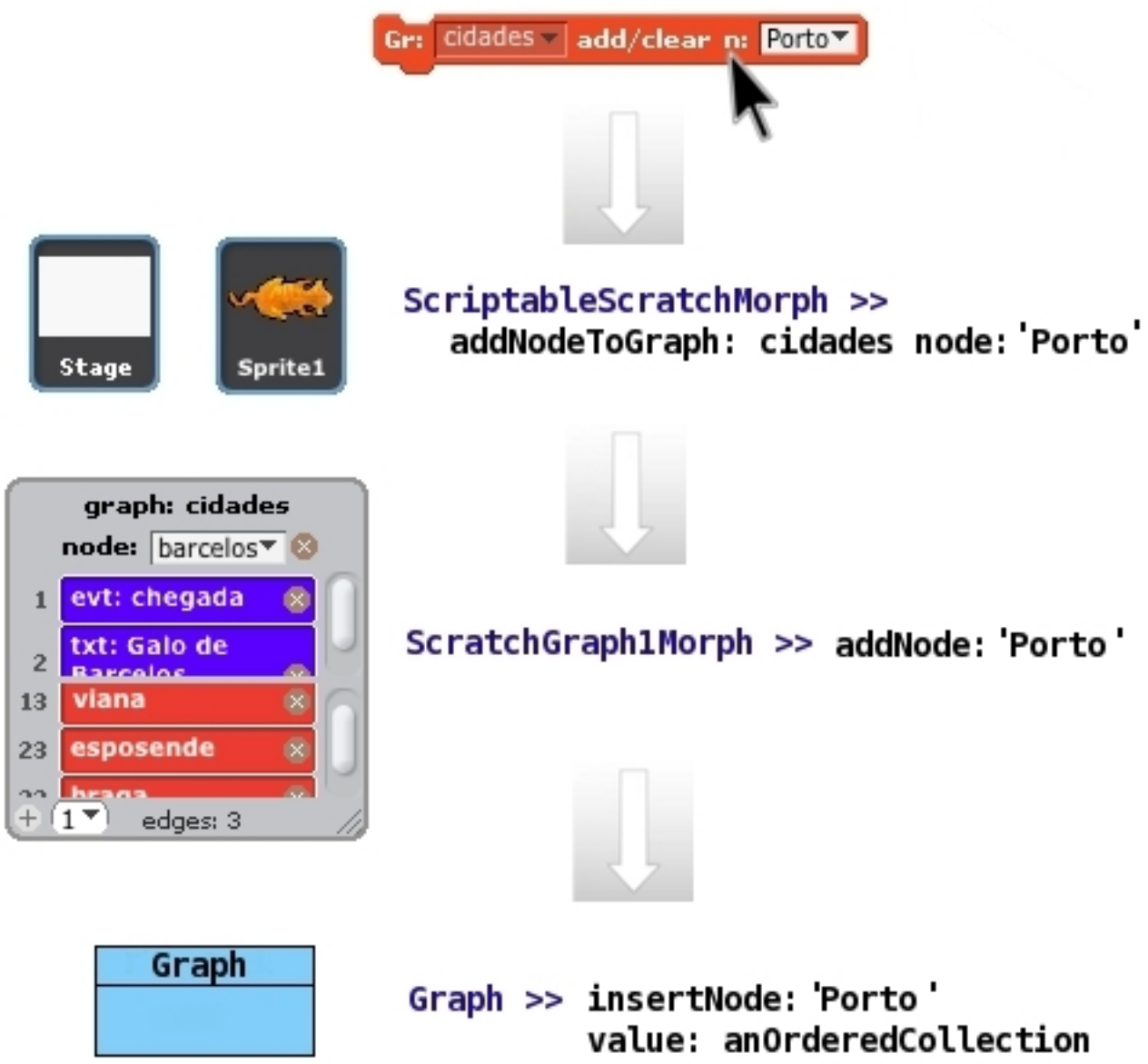


Figura 5.33: Adição de um nodo ao grafo

Quando o bloco é executado, o método que corresponde ao seu `selector` é executado com os argumentos introduzidos no bloco. Este método é executado pelo `ScriptableScratchMorph` selecionado na Lista de **Sprites**, que poder ser um **Sprite** ou o **Stage**.
O método `addNodeToGraph: graphName node: nodeName` corresponde ao método `selector` deste bloco. Em primeiro lugar este acede no `Dictionary` de nome `graphs` ao valor associado à chave 'cidades', dada como argumento no bloco. O valor a associado a esta chave é o visualizador associado ao grafo de nome 'cidades' e é atribuído a uma variável da seguinte forma:
`graphMorph _ self graphNamed: 'cidades' ifNone: ^0`.
Depois disto é invocado o método `addNode: nodeName` no visualizador da seguinte forma: `graph1Morph addNode: 'Porto'`.
Este método por sua vez cria uma `OrderedCollection` vazia que vai ser introduzida no valor do novo nodo do grafo representando uma lista vazia de *values* desse nodo. De seguida é invocado o método da classe `Graph` que por fim introduz o novo nodo no grafo guardado na variável de instancia `graph` do visualizador. Este método é invocado da seguinte forma:
`self graph insertNode: 'Porto' value: anOrderedCollection`.

5.5 Paleta de blocos com blocos grafos

5.5.1 Blocos Reporter para variáveis grafo

Os novos blocos **Reporter** de grafos são representados por uma nova classe chamada `GraphContentsBlockMorph` que é muito semelhante à classe com o mesmo efeito para as listas do Scratch `ListContentsBlockMorph`. As diferenças são nos métodos utilizados na definição dos métodos desta classe. Estes métodos que são aqui utilizados são equivalentes nas suas funções aos das listas mas são relativos

aos grafos. O código dos métodos diferentes da classe `GraphContentsBlockMorph` em comparação com a classe `ListContentsBlockMorph` estão no anexo A.1.1.
O novo `selector` destes blocos é o `ScriptableScratchMorph>>contentsOfGraph: graphName`. Este define o que acontece com o bloco **Reporter** representante da variável grafo na Paleta quando executado. Devolve todos os nodos do grafo que representa numa *string*. O seu código está em anexo em A.1.2.
Este `selector` do novo bloco encontra-se também introduzido no método `ScriptableScratchMorph>>blockFromTuple: tuple receiver: scriptOwner` evitando assim que os blocos deste tipo apareçam como obsoletos quando são carregados de um projeto para a aba **Scripts**. A alteração ao código de `blockFromTuple: tuple receiver: scriptOwner` encontra-se no anexo A.1.3.
O método `BlockMorph>>newScriptOwner: newOwner` também se encontra alterado. Este método é usado quando acontece uma copia de **Scripts** entre **Sprites** ou **Stage**. Reconhece se no **Script** copiado existem **Reporters** de grafo, `GraphContentsBlockMorphs`, ou se existem blocos com menus de escolha de grafo. Para os nomes do grafo correspondentes à escolha do menu do bloco ou ao nome do **Reporter** este método invoca o método `ScriptableScratchMorph>>ensureGraphExists: graphName` que verifica se no **Sprite** ou **Stage**, para onde este **Script** vai ser copiado, já existe o grafo correspondente ao nodo no seu `Dictionary` de grafos `graphs`. E caso ainda não exista cria este novo grafo neste `Dictionary`. A alteração ao método `newScriptOwner: newOwner` encontra-se em anexo em A.1.4 e o novo método `ensureGraphExists: graphName` encontra-se também em anexo em A.1.5

5.5.2 Adição dos blocos Reporter dos grafos

O novo método `addGraphReportersTo: page x: x y: y`, que adiciona os **Reporters** das variáveis grafo, é muito semelhante ao método que tem a mesma fun-

ção para as variáveis de lista, o método `addListReportersTo: page x: x y: y`. Na primeira parte do método são adicionadas as variáveis globais caso um **Sprite** esteja selecionado na Lista de **Sprites** 3.10, ou seja, se o **Stage** não estiver selecionado. Cada um dos grafos vai ser representado por um bloco **Reporter** que será um novo `GraphContentsBlockMorph`. Cada **Reporter** de grafo é adicionado à frente da sua caixa de seleção. Esta caixa, quando selecionada permite ver o visualizador do grafo no **Stage**.

De seguida, é adicionada uma linha para separar os grafos globais dos locais. Esta linha só aparece se um **Sprite** estiver selecionado na Lista de **Sprites** e não o **Stage**, já que o **Stage** só guarda grafos globais e por isso não é necessário separar visualmente os grafos globais dos locais.

Por fim são adicionadas os grafos locais ao **Sprite** selecionado ou os grafos globais caso o **Stage** esteja selecionado na Lista de **Sprites**. Isto é feito através de código muito semelhante ao aplicado na primeira parte este método apenas diferindo na especificação do *morph* onde o método obtem os grafos. As diferenças entre o código deste método e o código do método para o mesmo efeito nas listas encontram-se em anexo em A.1.6.

5.5.3 Adição de blocos de grafos e botões

O método que adiciona os blocos e botões relativos às listas na categoria **Variables** da Paleta de blocos é o método `ScriptableScratchMorph>>addGenericListBlocksTo: page y: startY`. Para introduzir os blocos e botões relativos aos grafos existe agora um método, na mesma classe, semelhante a este último chamado `addGenericGraphBlocksTo: page y: startY`. Estes dois métodos e os seus constituintes têm praticamente o mesmo código, sendo que as alterações que foram necessárias centraram-se em substituir os fragmentos de código que se referem a listas pelos seus equivalentes para grafos.

O método `addGenericGraphBlocksTo: page y: startY` começa por atribuir à variável `addButton` um botão com etiqueta '**Make a graph**' com o método associado `ScriptableScratchMorph>>addGraph`. De seguida, faz um teste que verifica se o novo grafo está a ser criado para o **Stage**. Em caso afirmativo o método associado ao botão de criação é alterado para `ScriptableScratchMorph>>addGlobalGraph`.
Como já foi dito, no Scratch qualquer variável pode ser local ou global. Quando um **Sprite** está selecionado na Lista de **Sprites** 3.10 e o utilizador carrega no botão de criação de uma variável grafo o método `addGraph` é invocado. Este método abre uma janela de criação que pergunta o nome do novo grafo e permite escolher se esta é local a esse **Sprite** ou global. Por outro lado se o **Stage** estiver selecionado na Lista de **Sprites** o método invocado é o `addGlobalGraph` que cria sempre uma variável global podendo esta ser usada por qualquer **Sprite** ou **Stage**. Em qualquer dos casos, quando uma variável global é criada esta é sempre inserida no dicionário de variáveis do **Stage** tanto em `addGraph` como em `addGlobalGraph`.
Em seguida é atribuído à variável `deleteButton` um botão com etiqueta '**Delete a graph**' com o método associado `ScriptableScratchMorph>>deleteGraph`. Este método mostra ao utilizador a lista das variáveis de grafo e pergunta qual é escolhida para remoção. Depois disto o método adiciona à Paleta de Blocos o *morph* respetivo ao botão de criação `addButon`.
A partir deste momento o método apenas adiciona mais elementos se existirem grafos já criados. Para isso é utilizado o método `ScriptableScratchMorph>>graphVarNames` que retorna o número de grafos já existentes no dicionário `graphs` de um `ScriptableScratchMorph`. O **Stage** é representado por um `ScratchStageMorph` e os **Sprites** são representados por `ScratchSpriteMorphs`, ambas estas classes são subclasses de `ScriptableScratchMorph`. Assim, com este método, é testado se existem grafos tanto no **Stage** (globais) como no **Sprite** selecionado (locais). Caso não existam grafos o método não necessita adicionar mais nenhum elemento

e por isso pára a execução retornando `self`, mas caso existam grafos já criados o método procede a sua introdução. Depois disto o método adiciona à Paleta de blocos o *morph* respetivo ao botão de remoção `deleteButon`.

Em seguida é invocado o método `ScriptableScratchMorph>>addGraphReportersTo: page x: x y: y` que tem como função adicionar os blocos **Reporter** correspondentes aos grafos existentes tanto globais como locais.

Por último, são adicionados os blocos criados para os grafos que estão nas especificações de blocos em `ScriptableScratchMorph>>blockSpecs`. Através do método `ScriptableScratchMorph>>blocksFor: aCategory` é criada uma lista dos blocos de grafos e também de símbolos que significam espaços a dar entre os blocos. Com esta informação o método `addGenericGraphBlocksTo: page y: startY` cria os *morphs* dos blocos e adiciona-os à Paleta de blocos assim como os espaços entre eles.

É importante realçar também que neste método as variáveis `x` e `y` representam as coordenadas onde os novos elementos são introduzidos e estas são atualizadas de acordo com o desenho pretendido. Quando o método vai adicionar um novo elemento calcula o local no eixo dos y onde o vai inserir usando a coordenada `y` que guarda o local onde o último elemento foi introduzido. A variável `x` na maior parte do método mantêm-se com o valor 13 a não ser dentro do método que adiciona os grafos existentes como blocos **Reporter** - `addGraphReportersTo: page x: x y: y`. As diferenças entre o código deste método e o código do método para o mesmo efeito nas listas encontram-se em anexo em A.1.7.

5.5.4 Construção de toda a Paleta de blocos

O método responsável por construir na Paleta de blocos 3.20 os blocos e botões de criação relativos a toda a categoria **Variables** é o método `ScriptableScratchMorph>>variablesPage`. Este método atribui à variável `page` uma nova instancia do

morph `ScratchPalleteMorph`, que é o *morph* que representa a paleta de blocos.

```
ScriptableScratchMorph>>variablesPage
...
page _ ScratchBlockPaletteMorph new
                    color: (Color r: 0.8 g: 0.8 b: 1.0);
                    borderWidth: 0.
...
```

Em seguida o método adiciona os *submorphs* que correspondem aos botões e blocos desta categoria de acordo com o estado atual das variáveis e listas do Scratch. Em primeiro lugar são adicionados o botão '**Make a variable**' e em seguida, caso existam variáveis já criadas, são adicionados o botão '**Delete a variable**', os blocos **Reporter** correspondentes às variáveis já criadas e por fim os blocos de manipulação de variáveis.
Posteriormente aparece o método que trata da adição dos blocos e botões de criação e remoção de listas, o método `ScriptableScratchMorph>>addGenericList-BlocksTo: page y: startY`. Este último é invocado posteriormente ao desenho das variáveis e aparece desta forma no código original do método `VariablesPage`:

```
ScriptableScratchMorph>>variablesPage
...
self addGenericListBlocksTo: page y: y.
...
```

A variável `y`, recebida como parâmetro pelo método `addGenericListBlocksTo: page y: startY`, representa uma coordenada. Esta coordenada representa, no eixo dos y, o último local onde um elemento novo foi adicionado à paleta de blocos. Deste modo, é a partir de `y` neste método é calculado o local onde deve adicionar novos elementos à Paleta de blocos.

Para adicionar os elementos relativos aos grafos criou-se o método `ScriptableScratchMorph>>addGenericGraphBlocksTo: page y: startY`. Este método é em tudo semelhante ao método `addGenericListBlocksTo: page y: startY` com a diferença que este adiciona os elementos relativos às novas variáveis grafo. Para que este método saiba onde vai inserir os novos elementos necessita de obter a variável `y` atualizada com o valor da coordenada y do último local onde foi inserido um bloco de listas. Para isto modificou-se o método `addGenericListBlocksTo: page y: startY` para que retornasse o valor da coordenada y onde inseriu o último bloco de lista. Assim foi necessário alterar o código deste método em dois locais. Um deles é o local depois da inserção do botão '**Make a list**'. Neste local, é feito um teste para saber se existem variáveis de lista ativas. Caso não existam listas, o método não precisa de inserir mais nenhum elemento e retorna `self` originalmente. Este valor de retorno foi alterado para a variável `y`. Alterando de:

```
ScriptableScratchMorph>>addGenericListBlocksTo: page y: startY
...
hasLists ifFalse: [^ self].
...
```

para:

```
ScriptableScratchMorph>>addGenericListBlocksTo: page y: startY
...
hasLists ifFalse: [^ y].
...
```

O outro local onde foi necessário alterar foi no fim do método. A alteração consiste em adicionar o retorno da variável `y` depois de todo o código do método.
Adicionando no fim do método:

```
ScriptableScratchMorph>>addGenericListBlocksTo: page y: startY
...
^ y
```

Depois de feitas estas alterações foi necessário ainda alterar o método `variablesPage` atribuindo à variável `y` o valor de retorno do método `addGenericListBlocksTo: page y: startY`. E finalmente adicionar o método `addGenericGraphBlocksTo: page y: startY` que adiciona os elementos relativos a grafos à Paleta de blocos no local correto. Alterando de:

```
ScriptableScratchMorph>>variablesPage
...
self addGenericListBlocksTo: page y: y.
...
```

para:

```
ScriptableScratchMorph>>variablesPage
...
y _ self addGenericListBlocksTo: page y: y.
self addGenericGraphBlocksTo: page y: y.
...
```

5.6 Conclusão

Neste capítulo foram descritos os novos componentes do Scratch que foram criados neste trabalho e também toda a sua implementação e integração no sistema. Foi descrito como o Scratch guarda os grafos em cada **Sprite** ou **Stage**, e também a classe a classe `Graph` que define a estrutura de informação de um grafo e permite a manipulação da sua informação através dos seus métodos. Depois foram explicados todos os componentes do visualizador de grafos no palco e seu funcionamento, e, de seguida, foram abordados os novos blocos para grafo abordando as suas características específicas e novos componentes de blocos. Finalmente foi exposto o processo de integração, na Paleta de blocos na categoria **Variables**, dos novos blocos e botões de criação respetivos dos grafos.

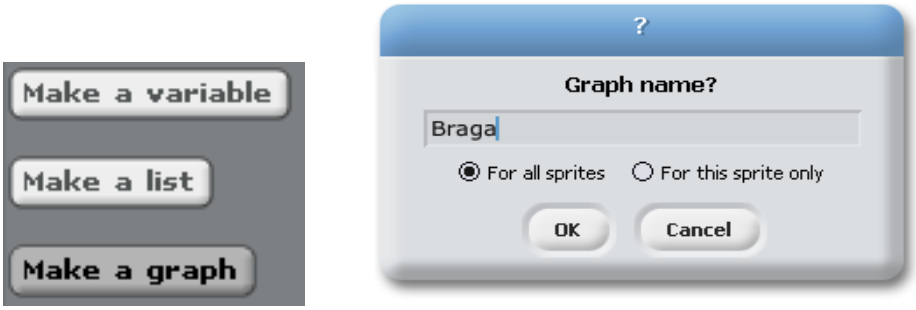
Capítulo 6

Manual e exemplo de aplicação

Neste capítulo na secção 6.1 serão apresentadas as várias funcionalidades adicionadas ao ambiente Scratch e o seu funcionamento do ponto de vista do utilizador. Na secção 6.2 será apresentado um possível exemplo prático da aplicação dos grafos no desenvolvimento de projetos em Scratch.

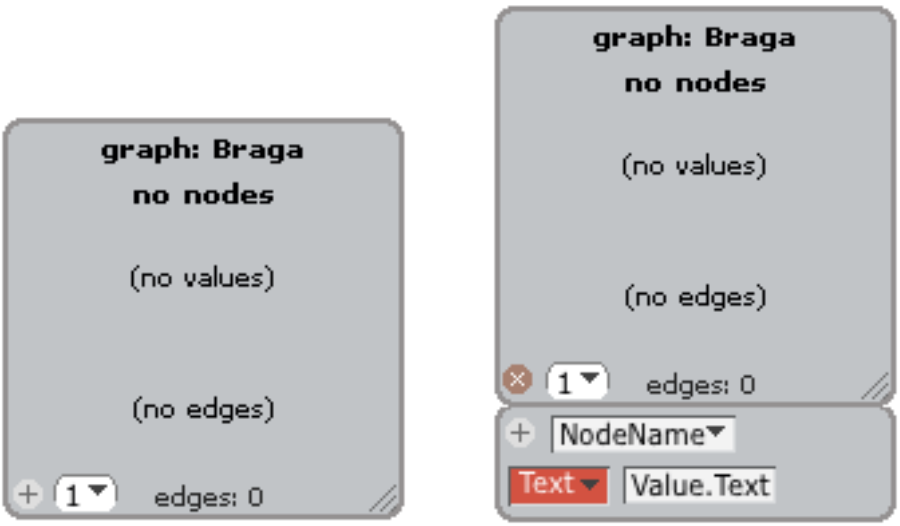
6.1 Manual

6.1.1 Visualizador de grafo



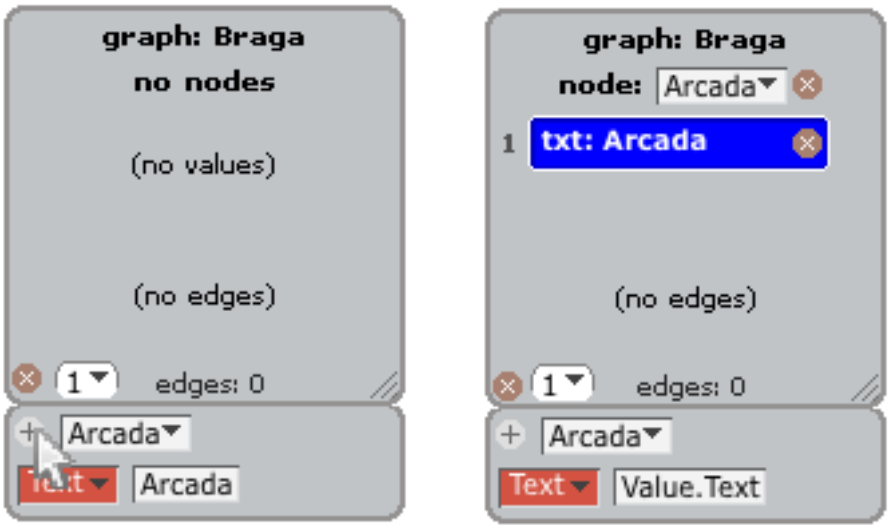
(a) (b)

Figura 6.1: Criar um novo grafo de nome '**Braga**'



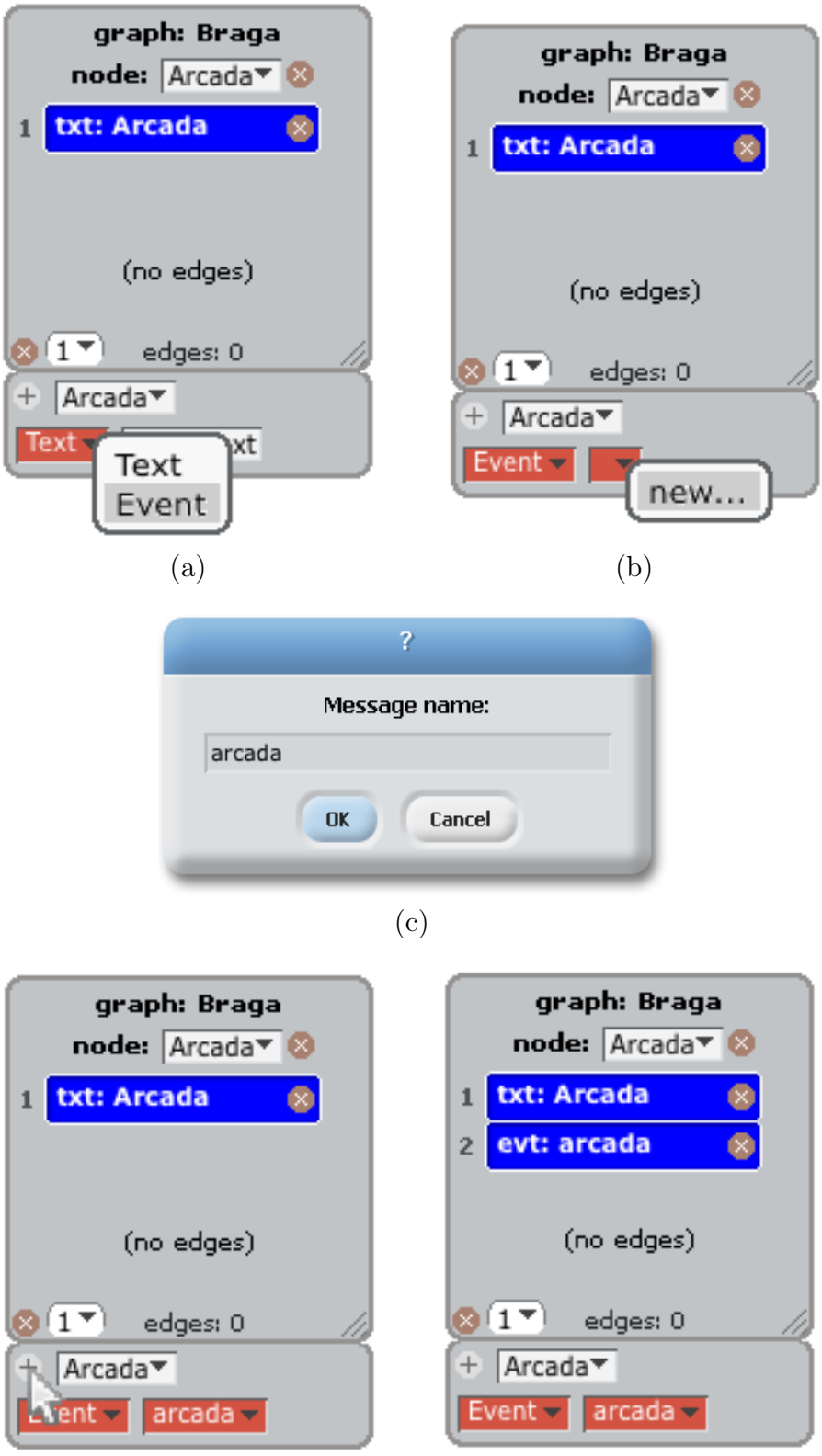
(a) (b)

Figura 6.2: Visualizador do grafo vazio em modo de edição



(a) (b)

Figura 6.3: Adicionar um novo nodo com um *value* textual



(a) (b)

(c)

(d) (e)

Figura 6.4: Adicionar um *value* evento ao nodo '**Arcada**'

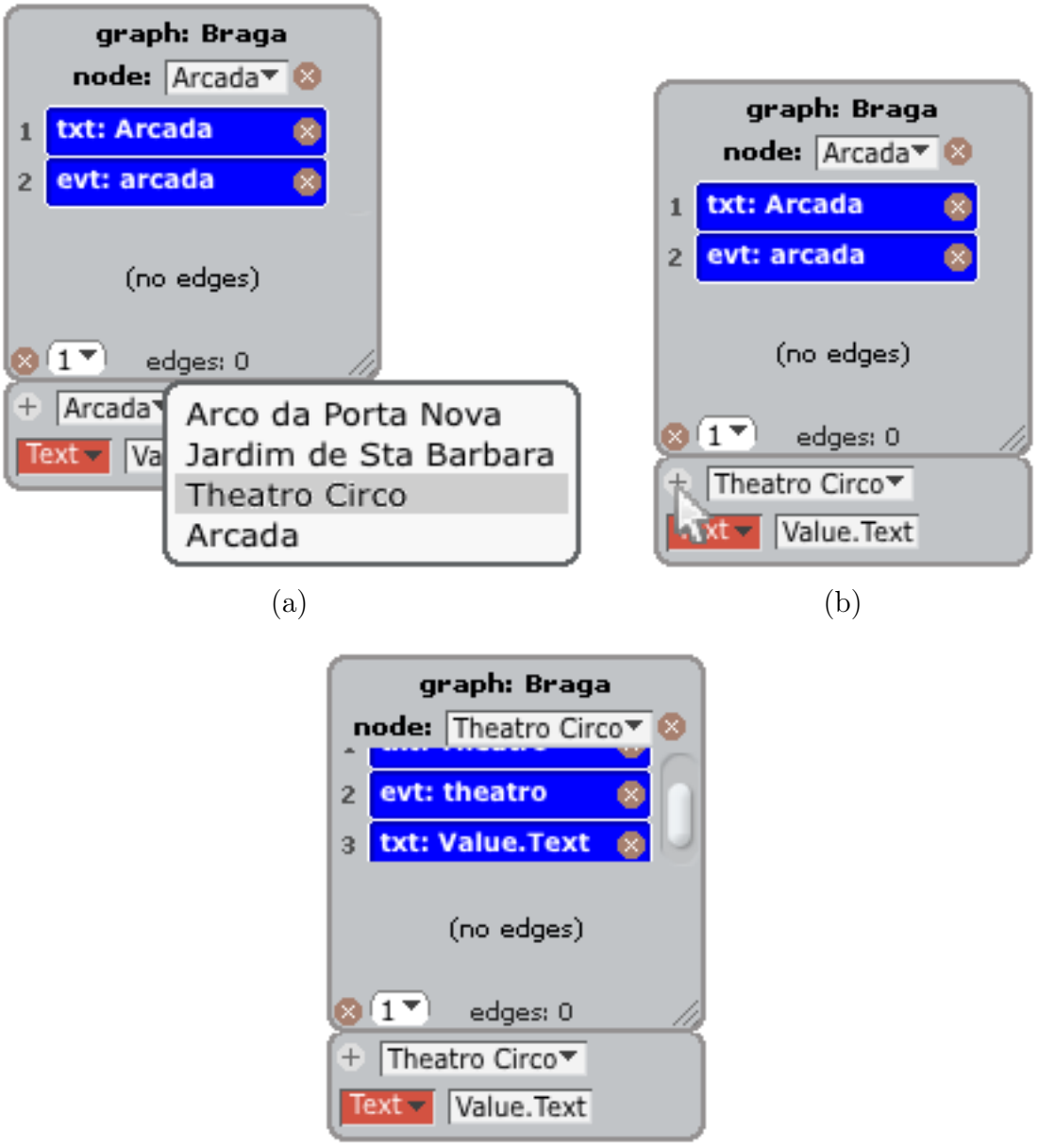
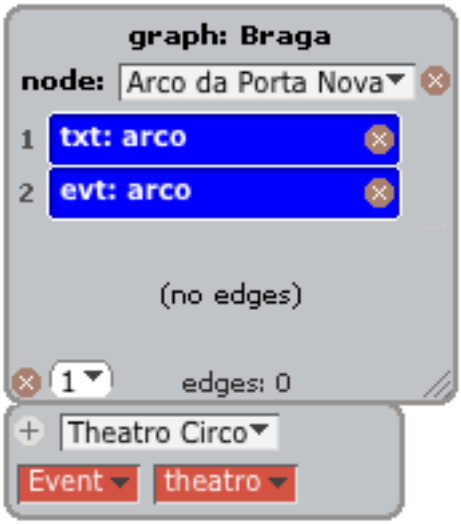


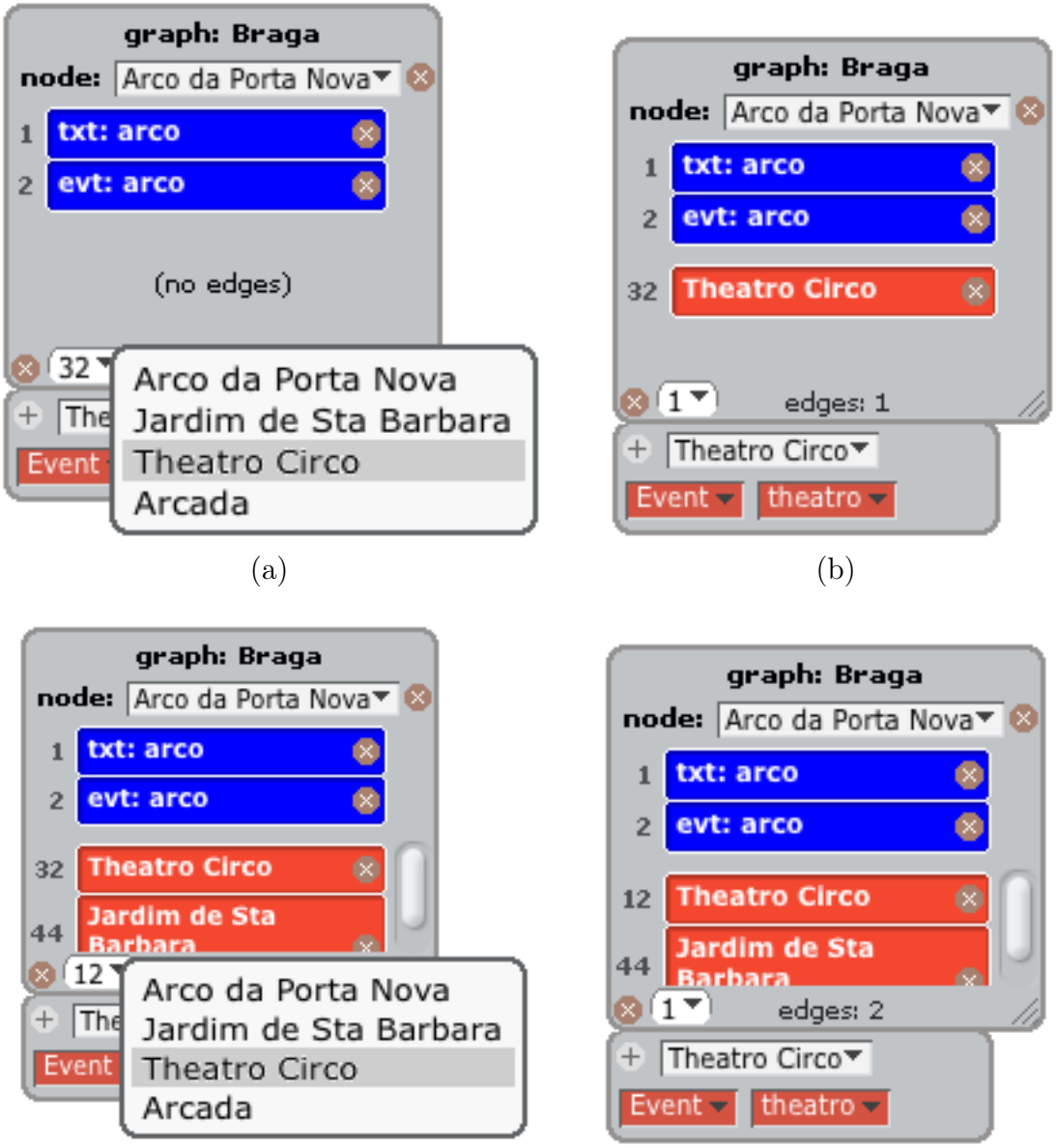
Figura 6.5: Adicionar um novo *value* textual ao nodo '**Theatro Circo**'

(a)



(b)

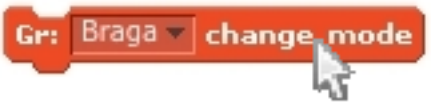
Figura 6.6: Apresentar o nodo '**Arco da Porta Nova**'



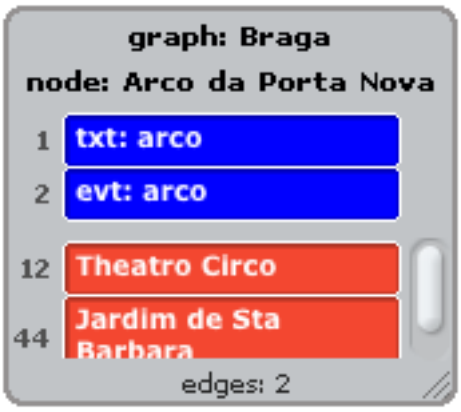
(a) (b)

(c) (d)

Figura 6.7: Adicionar dois *edges* ao nodo '**Arco da Porta Nova**'



(a) Bloco de alteração do modo de apresentação do visualizador



(b) Visualizador em modo de execução

Figura 6.8: Mudança do modo de apresentação do visualizador

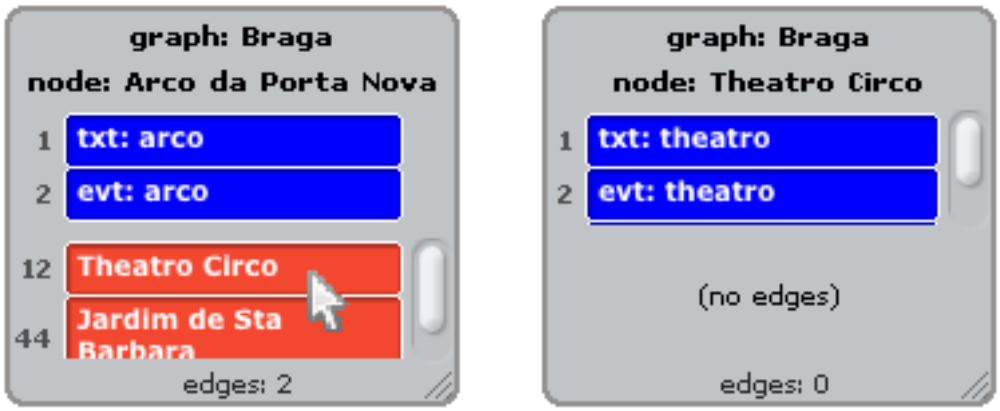
Figura 6.9: Comportamento de clique nas células de *values* textuais

(a) **Sprite** com um **Script** de deteção do evento '**arco**'

(b) Transmissão do evento '**arco**' através de um clique

Figura 6.10: Comportamento de clique nas células de *values* de evento

Figura 6.11: Bloco que ativa/desativa a funcionalidade de clique das células de *values*



(a) (b)

Figura 6.12: Navegação sobre o grafo com clique na célula de *edge* do nodo

Figura 6.13: Bloco que ativa/desativa a funcionalidade de clique das células de *edges*

(a) Exportação através do submenu do visualizador

(b) Exportação através do bloco **export**

Figura 6.14: Exportação do grafo para ficheiro *.dot*

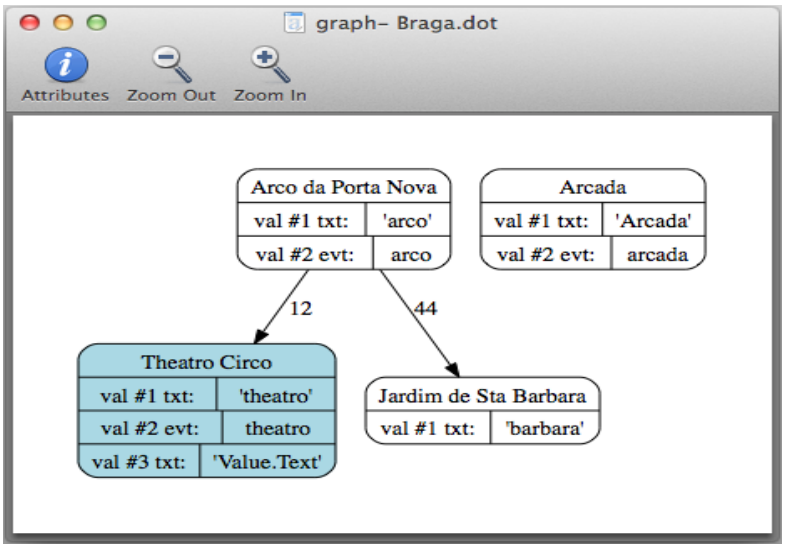


Figura 6.15: Ficheiro *.dot* gerado aberto com o programa Graphviz

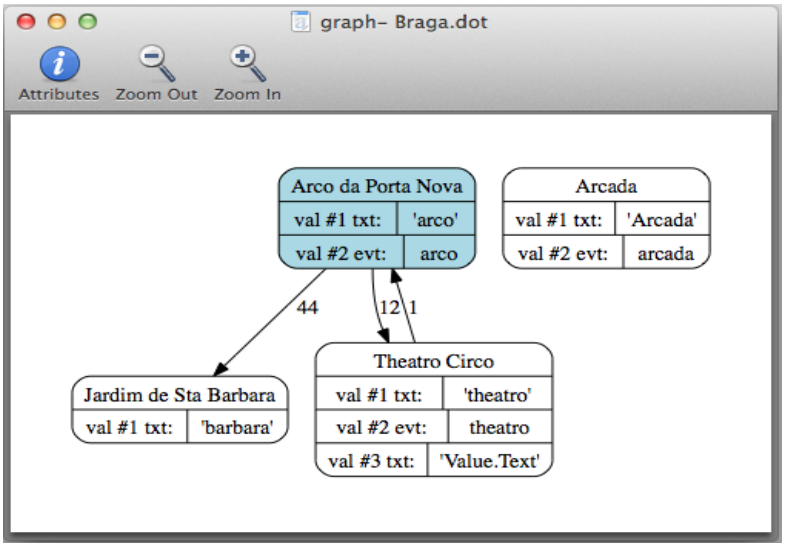


Figura 6.16: Alterações automáticas ao ficheiro *.dot* resultantes da adição de um *edge* do nodo '**Theatro Circo**' para '**Arco da Porta Nova**' e da alteração do nodo atual do visualizador para '**Arco da Porta Nova**'

6.1.2 Blocos de grafo

Blocos de funcionalidades do visualizador

Figura 6.17: Alterar o modo de apresentação do visualisador

Figura 6.18: Ativar/desativar a funcionalidade de clique nas células de *values* do visualizador

Figura 6.19: Ativar/desativar a funcionalidade de clique nas células de *edges* do visualizador

Figura 6.20: Exportar a informação de um grafo para um ficheiro *.dot*

Blocos de nodos

Figura 6.21: Definir um nodo como nodo atual do visualizador que está a ser apreentado

Figura 6.22: Adicionar um novo nodo ao grafo. Caso já exista apaga os seus *nodes* e *edges*

Figura 6.23: Apagar um nodo do grafo

Figura 6.24: Inserir os nomes de todos os nodos do grafo numa lista

Figura 6.25: Alterar o nome de um nodo existente no grafo

Figura 6.26: indicar o nodo atual do grafo

Figura 6.27: indicar quantos nodos existem no grafo

Figura 6.28: '**True**' se o nodo existir no grafo

Blocos de *values*

Figura 6.29: Adicionar um *value* um nodo do grafo

Figura 6.30: Apagar um *value* de um nodo do grafo

Figura 6.31: Apagar todos os *values* de um nodo do grafo

Figura 6.32: Inserir todos os *values* de um nodo do grafo numa lista

Figura 6.33: Transmitir um evento caso o *value* do nodo na posição escolhida seja do tipo evento. Caso seja do tipo textual, o **Sprite** dono "diz" o texto do *value*

Figura 6.34: indicar quantos *values* tem um nodo do grafo

Figura 6.35: Indicar o *value* de um nodo de uma posição da sua lista de *values*

Figura 6.36: Indicar o tipo de *value* de um nodo de uma posição da sua lista de *values*

Figura 6.37: '**True**' se o tipo de *value* de um nodo de uma posição da sua lista de *values* é do tipo indicado

Figura 6.38: '**True**' se no nodo existe o *value* indicado

Blocos de *edges*

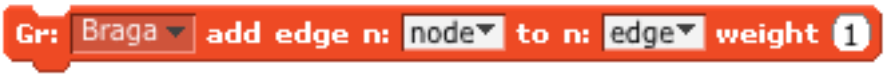


Figura 6.39: Adicionar um *edge* a um nodo do grafo

Figura 6.40: Apagar um edge de um nodo do grafo

Figura 6.41: Apagar todos os *edges* de um nodo do grafo

Figura 6.42: Inserir todos os *edges* de um nodo do grafo numa lista

Figura 6.43: Indicar quantos *edges* existem num nodo do grafo

Figura 6.44: Indicar o peso de ligação de um nodo a um *edge* da sua lista de *edges*

Figura 6.45: Indicar o *edge* de um nodo de uma posição da sua lista de *edges*

Figura 6.46: Indicar o índice de um *edge* da lista de *edges* de um nodo

Figura 6.47: '**True**' se o nodo contém o *edge* indicado

Blocos de caminhos

Figura 6.48: Determinar os nodos correspondentes ao caminho mais curto entre dois nodos do grafo e inseri-los numa lista

Figura 6.49: Indica o peso total ou os nodos do caminho mais curto entre dois nodos do grafo

.

Figura 6.50: Indica o peso do caminho que se encontra numa lista

Figura 6.51: '**True**' se a lista contém um caminho válido no grafo

Figura 6.52: '**True**' se existe caminho no grafo entre os dois nodos escolhidos

Blocos a implementar

Figura 6.53: Exemplo de blocos de grafos que podem ser implementados

Figura 6.54: Exemplo de blocos específicos de árvores que podem ser implementados

6.2 Exemplo de aplicação

Como exemplo de aplicação dos novos grafos Scratch, construiu-se uma animação interativa que representa um mapa do centro histórico de Braga. Este mapa contém vários pontos turísticos importantes assinalados no mapa e um ator que representa um turista que navega pelo vários pontos assinalados, 6.55.

Figura 6.55: Estado inicial da animação

Quando o utilizador pressiona um dos ponto do mapa o **Sprite** '**Turista**' calcula o caminho mais curto, passando pelos nodos do grafo, entre o ponto onde se encontra e o ponto carregado. De seguida percorre cada ponto desse caminho desenhando uma linha azul entre os vários pontos e transmitindo ao utilizador, através de um balão de texto, o nome de cada ponto turístico por onde passa, 6.56.

Figura 6.56: **Sprite** '**Turista**' percorrendo um caminho

Todos os pontos por onde o **Sprite** '**Turista**' vai passando mudam o seu aspeto e cor para que no final o caminho percorrido esteja bem visível. Ao longo do caminho também vão sendo somadas à variável '**distância**' todas as distâncias entre os pontos percorridos, 6.57.

Figura 6.57: Caminho completo percorrido pelo **Sprite** '**Turista**'

Quando o **Sprite** '**Turista**' chega ao último ponto do caminho é apresentada uma imagem, representativa desse ponto turístico, envolvendo todo o **Stage**, 6.58.

Figura 6.58: Slide do último ponto do caminho percorrido em 6.57, Sé de Braga

Esta animação contém vários **Sprites** que estão listados na figura 6.60. Contém um **Sprite** '**Turista**' que percorre o caminho entre os pontos e é representado pelo traje do gato tradiciona do Scratch.

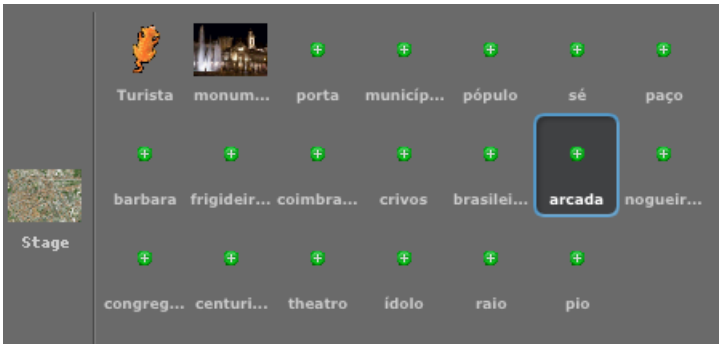
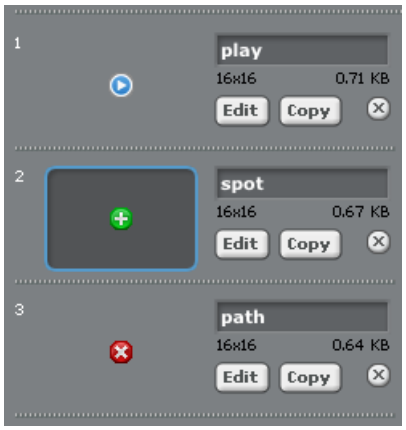


Figura 6.59: **Sprites** da animação

Contém também um **Sprite** '**monumento**' que apresenta o slide de cada ponto turístico do mapa. Os slides dos diferentes pontos estão guardas nos vários trajes deste **Sprite**, 6.60b. Os restantes **Sprites** da animação são os pontos históricos representados no mapa. Cada um deles tem três trajes diferentes, 6.60a: o azul para quando é carregado pelo utilizador com o rato, o vermelho para quando esse **Sprite** faz parte de um caminho que está a ser percorrido e o verde por omissão.



(a) Trajes dos **Sprites** que representam pontos históricos no mapa

(b) Três dos trajes do **Sprite** '**monumento**'

Figura 6.60: Trajes dos **Sprites** da animação

Para organizar a informação do mapa é utilizado um grafo. Neste grafo cada um dos nodos corresponde a um dos **Sprites** que representam pontos turísticos e contém dois *values* diferentes. O *value* número um contém um texto que corresponde ao nome do **Sprite** ponto correspondente a esse nodo. Enquanto que o *value* número dois contém um evento que é detetado apenas pelo seu *S*prite ponto correspondente. Cada nodo contém também, na sua lista de *e*dges, todos os nomes dos nodos aos quais ele se liga diretamente e as respetivas distâncias. Na figura

6.61 apresentam-se dois destes nodos do grafo.

(a) Nodo '**A Brasileira**' (b) Nodo '**Igreja dos Congregados**'

Figura 6.61: Dois nodos do grafo '**Braga**'

Para detetar quando um utilizador carrega num dos pontos turístico, cada um dos **Sprites** que os representam contém um **Script** que deteta este evento. Quando um ponto é precionado, é atribuído à variável '**destino**' o nome do nodo no grafo que corresponde ao **Sprite** ponto que foi carregado. Um exemplo deste **Script** é apresentado na figura 6.62.

Figura 6.62: **Script** de deteção de clique do **Sprite** '**sé**'

O **Sprite** '**Turista**' contém um ciclo com uma condição no seu interior que verifica a todo o momento se a variável '**destino**' é diferente do nodo atual do grafo, figura 6.63. Assim quando a variável '**destino**' é alterada, devido a um clique do utilizador num dos **Sprites** ponto, os blocos dentro dessa condição são processados.

Figura 6.63: Ciclo e condição do **Sprite** '**Turista**', sem blocos na condição

Na figura 6.66 encontra-se representado o **Script** completo do **Sprite** '**Turista**', que define o seu ciclo geral, e também os blocos dentro da sua condição. O primeiro bloco transmite o evento '**verde**' que é recebido por todos os **Sprites** ponto. Assim que detetam este evento, os **Sprites** ponto processam o **Script** 6.64 o qual altera o traje de todos para o traje de nome '**spot**' exceto um. O traje do **Sprite** ponto que foi premido pelo utilizador é alterado para o traje de nome '**play**'.

Figura 6.64: **Script** do **Sprite** '**congregados**' que altera o seu traje

Os dois blocos seguintes na condição da figura 6.66 são blocos de inicialização de variáveis. Em primeiro lugar à variável '**origem**' é atribuído o nome do nodo atual do grafo, que é o nodo correspondente ao ponto onde o **Sprite** '**Turista**' se encontra no momento. Em segundo lugar a variável distância é inicializada a zero, esta variável é utilizada para mostrar a distância do caminho que vai ser percorrido. De seguida é utilizado o bloco **shortest path** para calcular os nodos do caminho

mais curto entre o nodo onde o **Sprite** '**Turista**' se encontra, referenciado pela variável '**origem**', e o nodo que corresponde ao **Sprite** que foi pressionado pelo utilizador, referenciado pela variável '**destino**'. Estes nomes de nodos do caminho são introduzidos numa lista chamada '**caminho**' representada na figura 6.65.

Figura 6.65: Lista com o caminho entre os nodos '**Restaurante Centurium**' e '**Se de Braga**'

Por último é feita uma transmissão síncrona do evento '**percorrer**' que levará à inicialização do **Script** do **Sprite** '**Turista**' que o faz percorrer os **Sprites** correspondentes aos nodos do caminho calculado.

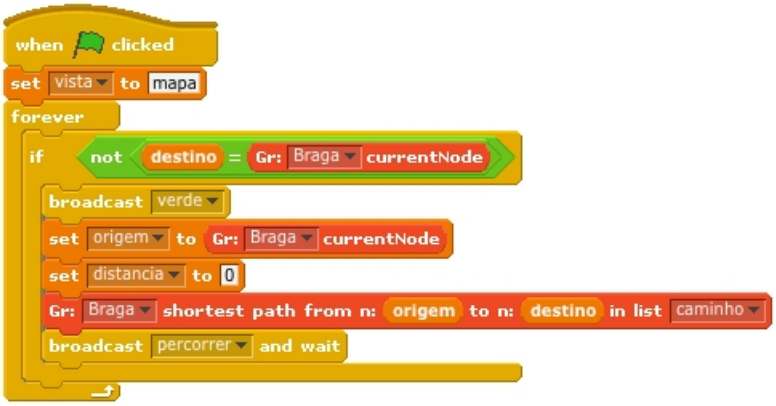


Figura 6.66: Ciclo do **Sprite** '**Turista**' e sua condição interior com blocos

O **Script** do **Sprite** '**Turista**' que deteta o evento '**percorrer**', representado parcialmente na figura 6.67, começa por inicializar uma variável '**i**' a um. Esta variável

é usada como índice de posição da lista '**caminho**' quando esta é percorrida. Em seguida o bloco **clear** apaga todos os traços da caneta feitos no teclado resultantes do caminho anteriormente desenhado. Depois disto é utilizado o bloco **repeat** que contém no seu interior blocos que definem o comportamento que o **Sprite** '**Turista**' executa para se movimentar do ponto onde se encontra para o ponto correspondente ao nodo seguinte da lista '**caminho**'. Estes blocos, interiores ao bloco **repeat**, são executados tantas vezes quantas a quantidade de nodos que existe na lista '**caminho**'.

Figura 6.67: **Script** do **Sprite** '**Turista**' acionado pelo evento '**percorrer**'

Dentro do bloco **repeat** o primeiro bloco a ser executado lança o evento que se encontra na segunda posição dos *values* do nodo onde o **Sprite** '**Turista**' se encontra neste momento. Este evento lançado é apenas reconhecido pelo **Sprite** ponto correspondente a esse nodo específico.

Figura 6.68: Bloco que transmite o evento na segunda posição do nodo atual

Quando este evento é transmitido, o **Sprite** que o recebe muda o seu traje para o traje vermelho através do **Script** da figura 6.69. Assim o ponto do mapa de onde o **Sprite** '**Turista**' vai sair muda o seu traje para o correspondente aos pontos que fazem parte de um caminho.

Figura 6.69: **Script** do **Sprite** '**congregados**' que altera o seu traje

Em seguida utiliza-se o bloco **pen down**, figura 6.70, para iniciar a caneta que desenha o caminho a azul.

Figura 6.70: Iniciar a caneta

Depois disto é atribuído à variável '**dist_parcial**' o peso de ligação entre o nodo onde o **Sprite** '**Turista**' se encontra e o nodo correspondente ao nome que se encontra na posição atual da lista caminho, o nodo seguinte. De seguida o valor desta distância entre dois pontos do caminho é multiplicado por mil e adicionado à variável '**distância**'. Esta mostra a distância do caminho à medida que este é percorrido pelo **Sprite** '**Turista**'. A distância parcial é multiplicada por mil para obter o valor de medida em metros. Os dois blocos responsáveis por este comportamento está representados na figura 6.71.

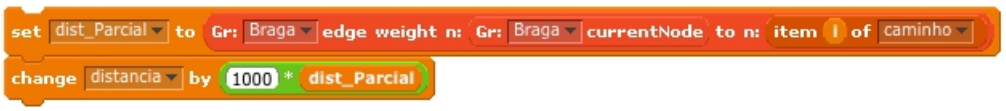


Figura 6.71: Adicionar a distância entre dois nodos do grafo à distância total

De seguida o **Sprite** '**Turista**' aponta para a posição do próximo **Sprite** ponto do caminho com os blocos da figura 6.72. O nome deste **Sprite** encontra-se no primeiro *value* do nodo cujo nome se encontra referenciado na posição atual da lista '**caminho**'.

Figura 6.72: Bloco que aponta o **Sprite** '**Turista**' para o próximo ponto

De seguida às variáveis '**x**' e '**y**' são atribuídas as coordenadas do **Sprite** ponto seguinte referenciado na lista '**caminho**'. Estas variáveis são utilizadas no bloco seguinte que faz o **Sprite** '**Turista**' mover-se do ponto onde se encontra para o ponto seguinte do caminho. Na figura 6.73 estão representados os blocos que definem este comportamento.

Figura 6.73: Blocos que movimentam o **Sprite** '**Turista**' para o próximo ponto

Depois do caminho percorrido o **Sprite** '**Turista**' transmite ao utilizador, através de um balão, o nome do nodo onde se encontra, tal como na figura 6.56. E em seguida o nodo atual do grafo é atualizado para o nodo referenciado na posição atual da lista caminho. Este comportamento é executado pelos blocos da figura 6.74.

Figura 6.74: **Sprite** '**Turista**' diz o nome do ponto atual e o nodo atual do grafo é atualizado

Ainda dentro do bloco **repeat** o último bloco processado, representado na figura 6.75, incrementa o índice '**i**' para ser utilizado na próxima iteração do ciclo.

Figura 6.75: Bloco que incrementa o índice '**i**'

Já com o caminho percorrido pelo **Sprite** '**Turista**', fora do ciclo **repeat** são chamados dois blocos que se apresentam na figura 6.76. O primeiro traz o **Sprite** '**Turista**' para trás do **Sprite** '**monumento**' que será mostrado a seguir. E o segundo lança o evento '**mostrar**' que sinaliza ao **Sprite** '**monumento**' a apresentação da foto relativa ao ponto onde o **Sprite** '**Turista**' se encontra.

Figura 6.76: Blocos de fim de **Script**

O **Script** completo do **Sprite** '**Turista**', que o faz percorrer um caminho completo, está representado na figura 6.77.

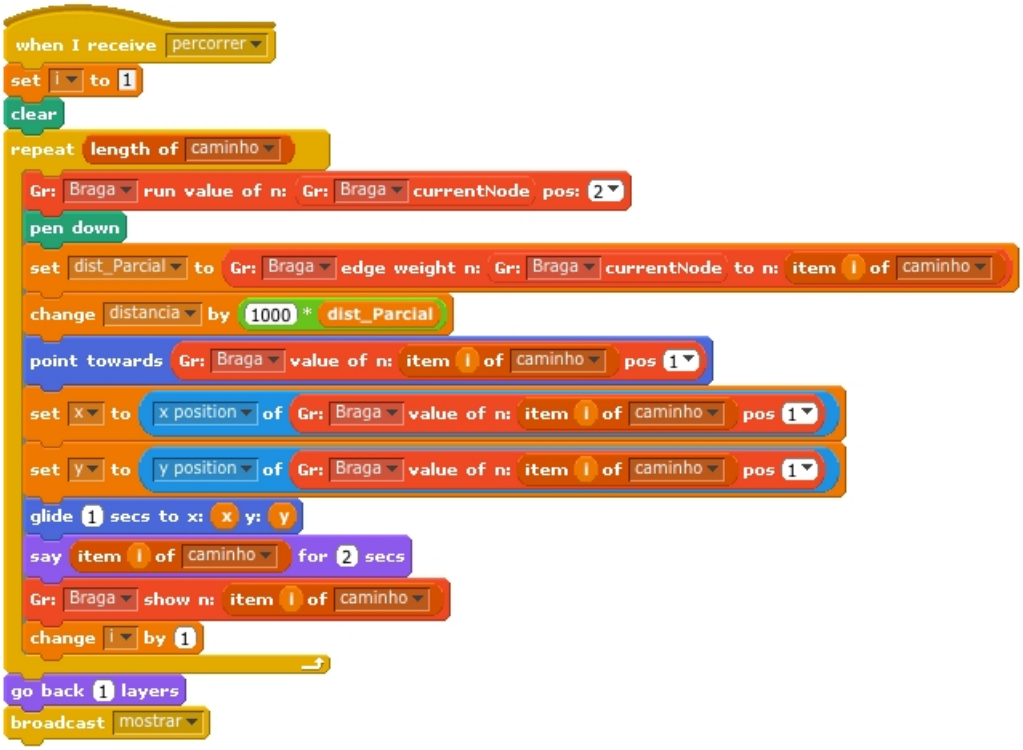


Figura 6.77: **Script** '**percorrer**' do **Sprite** '**Turista**'

No final da execução deste **Script** o **Sprite** '**Turista**' percorreu todos os pontos correspondentes aos nodos do caminho mais curto calculado desenhando um caminho entre eles. Este resultado final está representado na figura 6.78.

Figura 6.78: Resultado final do **Script** '**percorrer**' do **Sprite** '**Turista**' na animação

Assim que o evento '**mostrar**' é lançado pelo **Sprite** '**Turista**' é detetado pelo **Sprite** '**monumento**' no **Script** da figura 6.79. Este esconde as janelas de variáveis no **Stage** e muda o traje do **Sprite** '**monumento**' para o traje com o nome do primeiro *value* do nodo atual do grafo, que é o nodo final do caminho percorrido.

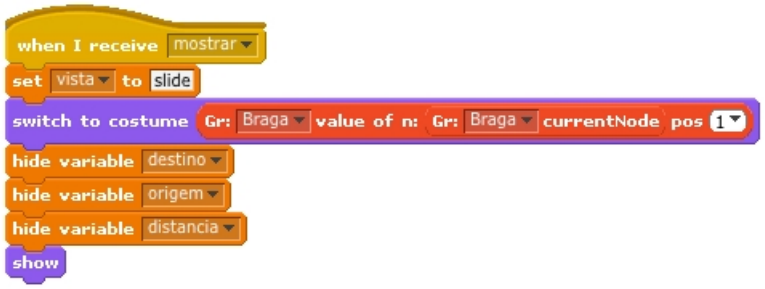


Figura 6.79: **Script** do **Sprite** '**monumento**' que mostra o slide correspondente ao ponto que se associa ao nodo atual do grafo

O traje que é apresentado é o slide correspondente do ponto turístico onde o **Sprite** '**Turista**' se encontra. De seguida, este slide é mostrado envolvendo todo o **Stage**, como é apresentado na figura 6.80.

Figura 6.80: Slide apresentado para o ultimo ponto do caminho, '**sé**'

6.3 Conclusão

Neste capítulo foi apresentado o funcionamento do novo visualizador de grafos e foram listados todos os novos blocos de grafo bem como as suas funcionalidades. De seguida foi apresentado um exemplo de utilização de grafos em Scratch implementando um grafo representativo de um mapa de pontos turísticos importantes do centro histórico de Braga.

Capítulo 7

Conclusão e trabalho futuro

Nesta dissertação foi construído um conjunto de novas funcionalidades para o ambiente visual de programação Scratch. Estas novas funcionalidades resultam da introdução no ambiente de uma nova categoria de estruturas Scratch, os grafos. Este novo componente foi implementado na categoria **Variables** que contém as listas e variáveis. Também foi implementado o visualizador de grafos que aparece no *Stage*. Este visualizador possibilita a manipulação e navegação direta sobre o grafo através de cliques por parte do utilizador, tornando a construção de grafos mais fácil, intuitiva e rápida. Para além disto os grafos implementados conseguem também guardar e lançar eventos através de um clique nas células de *values* de um nodo.

Foram também implementados blocos específicos para utilização de grafos. Estes permitem criar, manipular e aceder a todos os componentes do grafo e também implementam algoritmos específicos de grafos como testar a existência de caminho entre dois nodos do grafo e o descobrir o caminho mais curto entre dois nodos através do algoritmo Dijkstra.

Para trabalho futuro seria interessante ciar um modo de visualização dentro do Scratch que permitisse visualizar um grafo em modo geral no **Stage** e não apenas

um nodo de cada vez como está implementado neste trabalho. Outra possibilidade de melhoramento seria a possibilidade de criar grafos não pesados bem como a criação de mais blocos de grafos que ficaram por implementar como por exemplo: criar uma ligação bidirecional entre dois nodos apenas com um bloco, verificar se um grafo é fechado ou se tem ciclos bem como a implementação de outros algoritmos complexos específicos de grafos como algoritmos de resolução dos problemas de construção da arvore de extensão mínima de um grafo e o caixeiro viajante.
Também seria interessante no contexto das novas estruturas de dados, utilizar a estrutura de dados grafo como o seu subtipo, as árvores. Com esse objetivo em vista seria interessante criar blocos específicos para árvores utilizando as regras que apropriadas a este tipo de estuturas dados.
Ainda outro possível avanço para este projeto seria a sua integração com um trabalho que está a ser desenvolvido por João Gonçalves [Gon12]. Este trabalho visa criar extensões de controlo para o Scratch em particular o desenvolvimento de blocos personalizados, entre outros avanços. A combinação do controlo e disparo de eventos dos grafos deste trabalho com os blocos personalizados deste outro projeto melhoraria muito as possibilidades deste ambiente de programação visual.

Bibliografia

[BB94] Margaret Burnett and Marla Baker. A classification system for visual programming languages. *Journal of Visual Languages and Computing*, pages 287–300, 1994.

[BBB+95] Margaret Burnett, Marla Baker, Carisa Bohus, Paul Carlson, and Peiter van Zee. Scaling up visual programming languages. *Computer*, 1995.

[BPT08] Christos Bouras, Vassilis Poulopoulos, and Vassilis Tsogkas. Etoys as a collaborative environment through olpc's activity sharing. 2008. http://www.squeakland.org/.

[CM05] Stephen Chen and Stephen Morris. Iconic programming for flowcharts, java, turing, etc. In *In Proceedings of the 10th Annual SIGCSE Conference on innovation and Technology in Computer Science Education*, pages 104–107. ACM Press, 2005.

[CWHH05] Martin Carlisle, Terry Wilson, Jeffrey Humphries, and Steven Hadfield. Raptor: a visual programming environment for teaching algorithmic problem solving. In *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, pages 176–180. ACM Press, 2005.

[dC11] Fausto de Carvalho. Competências e conceitos de programação no scratch. Technical report, PT Inovação, Aveiro, 2011.

[DCP05] Wanda Dann, Stephen Cooper, and Randy Pausch. *Learning to Program with Alice*. Prentice Hall, 2005. http://www.alice.org/.

[Dij88] Edsger Dijkstra. On the cruelty of really teaching computer science. 1988.

[Dut04] Sajal Dutta. App inventor blocks, for android, 2004. http://www.appinventorblocks.com/.

[Gol04] Kenneth Goldman. A concepts-first curriculum for introductory computer science. In *In Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, pages 432–436. ACM Press, 2004.

[Gon12] João Cunha Gonçalves. Análise e desenvolvimento de extensões de controlo para o ambiente scratch (tese em desenvolvimento). Master's thesis, Universidade do Minho, 2012.

[GU03] Mark Guzdial and Usselman. Media computation course for non-majors. In *ITiCSE Proceedings*, pages 104–108, New York, 2003. ACM Press.

[HSS+10] Cristiane Hernandez, Luciano Silva, Rafael Segura, Juliano Schimiguel, Manuel Ledón, Luis Bezerra, and Ismar Silveira. Teaching programming principles through a game engine. *CLEI ELECTRONIC JOURNAL*, 13(2), aug 2010.

[III03] Harry H.Porter III. Smalltalk: A white paper overview. http:

//web.cecs.pdx.edu/~harry/musings/SmalltalkOverview.html, mar 2003.

[JR05] José Pereira Júnior and Clevi Rapkiewicz. Um ambiente virtual para apoio a uma metodologia para ensino de algoritmos e programação. In *Revista Novas Tecnologias na Educação*, pages 1–7. Revista Novas Tecnologias na Educação, 2005.

[KÏ10] Michael Kölling. The greenfoot programming environment. *ACM Transactions on Computing Education (TOCE)*, 10(4), nov 2010. http://www.greenfoot.org/door/.

[KG06] MIT Media Lab Kindergarten Group. Programming concepts and skils supported in scratch, 2006.

[Kin05] Mikael Kindborg. Comikit - programming with comic strips, european smalltalk user group 13th international conference, brussels, belgium. In *ESUG*, Linköping University SE-58183 Linköping, Sweden, aug 2005. http://www.comikit.se/.

[Koh07] Lutz Kohl. Puck - a visual programming system for schools, 7th baltic sea conference on computing education research. In *Conferences in Research and Practice in Information Technology, Vol. 88. Raymond Lister and Simon*, pages 191–194, Koli National Park, Finland, 2007. http://crpit.com/abstracts/CRPITV88Kohl.html/.

[Mal02] Jonh Maloney. *Squeak: Open Personal Computing and Multimedia*, chapter 2: An Introduction to Morphic: The Squeak User Interface Framework, pages 39–68. Prentice Hall, 2002. http://www.squeak.org/.

[MJ07] Andrés Moreno and Mike S. Joy. Jeliot 3 in a demanding educational setting. In *In Proceedings of the Fourth Program Visualization Workshop*, pages 51–59, University of Joensuu, Finland and University of Warwick, UK, 2007. http://cs.joensuu.fi/jeliot/.

[MLC04] Barbara Moskal, Deborah Lurie, and Stephen Cooper. Evaluating the effectiveness of a new instructional approach. In *In Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, pages 75–79. ACM Press, 2004.

[Mon08] Jens Monig. Lists and files for scratch a prototype, apr 2008. http://www.chirp.scratchr.org/blog/?p=16.

[Mon09a] Jens Monig. Build your own blocks in scratch a prototype, oct 2009. http://byob.berkeley.edu/.

[Mon09b] Jens Monig. Syntax-elements for smalltalk: A scratch-like gui for smalltalk-80, feb 2009. http://www.chirp.scratchr.org/blog/?p=24.

[Nic94] Jeffrey Nickerson. *Visual Programming*. PhD thesis, New York University, New York University, 1994.

[Pas09] Erik Pasternak. Visual programming pedagogies and integrating current visual programming language features. Master's thesis, Robotics Institute, Carnegie Mellon University, 2009. http://code.google.com/p/jubjub/.

[PJ04] C. E. Pereira Jr., J. C. R. Rapkiewicz. O processo de ensino-aprendizagem de fundamentos de programação: Uma visão crítica da pesquisa no brasil. In *Anais do WEI 2004 - Workshop de Educação*

em Computação, Congresso da Sociedade Brasileira de Computação. Rio das Ostras, 2004.

[Rod02] Susan Rodger. Introducing computer science through animation and virtual world. In *In Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education*, pages 75–79. ACM Press, 2002.

[Scr09] Scratch reference guide, 2009. http://info.scratch.mit.edu.

[SD03] Dean Sanders and Brian Dorn. Classroom experience with jeroo. *Journal of Computing Sciences in Colleges*, 18(4), apr 2003.

[Web99] Jonh Webster, editor. *Wiley Encyclopedia of Electrical and Electronics Engineering*. Wiley-Interscience, Cambridge, MA, 1999.

Apêndice A

Anexos

A.1 Paleta de Blocos

A.1.1 Novos Reporters para variáveis grafo - GraphContentsBlockMorph

```
GraphContentsBlockMorph>>asBlockTuple
"Answer a tuple (Array) describing this block and its arguments."
          ^ Array with: #contentsOfGraph: with: commandSpec
...

GraphContentsBlockMorph>>evaluateWithArgs: ignored
          ^ receiver contentsOfGraph: commandSpec
```

```
GraphContentsBlockMorph>>toggleWatcher
"Toggle between hiding and showing a
graph watcher for this block."
| graphMorph stage |
graphMorph _ receiver graphs at: commandSpec
          ifAbsent: [^ self].
graphMorph owner ifNil: [
          (stage _ receiver ownerThatIsA: ScratchStageMorph)
                 ifNil: [^ self].
          stage addMorph: graphMorph.
          (stage bounds containsPoint: graphMorph topLeft)
                 ifFalse: [
                 graphMorph position: stage topLeft + 10].
           graphMorph
           fixLayoutForNewLanguage;
           startStepping]
ifNotNil: [
         graphMorph delete].
```

A.1.2 Método selector para blocos Reporter de variáveis grafo

```
ScriptableScratchMorph>> contentsOfGraph: graphName
          | graph1Morph aStream |
          graph1Morph _ self graphNamed: graphName ifNone: [^ ''].
          aStream _ WriteStream on: (String new: 10000).
          graph1Morph graph keys do: [
                 :nodeName | aStream nextPutAll: nodeName,' '].
```

```
            ^ aStream contents.
```

A.1.3 GraphContentsBlockMorph reconhecido para que não se torne 'obsolete'

```
ScriptableScratchMorph>>
            blockFromTuple: tuple receiver: scriptOwner
"Answer a new block for the given tuple."

#contentsOfGraph: = k ifTrue: [
            ^ GraphContentsBlockMorph new
            color: ScriptableScratchMorph graphBlockColor;
            receiver: scriptOwner;
            commandSpec: tuple second;
            selector: #contentsOfGraph:].
...
```

A.1.4 Reconhecer quando um Script troca de dono

```
BlockMorph>>newScriptOwner: newOwner
...
(t3 isKindOf: CommandBlockMorph)
ifTrue: [t3 argMorphs do: [:t4 | ((t4 isKindOf: ChoiceArgMorph)
            and: [t4 getOptionsSelector = #graphVarMenu])
            ifTrue: [t4 choice ifNotNil: [
                  t1 ensureGraphExists: t4 choice]]]].
(t3 isKindOf: GraphContentsBlockMorph)
          ifTrue: [t1 ensureGraphExists: t3 commandSpec].
```

...

A.1.5 Inserir grafo no dicionário se não for visivel

```
ensureGraphExists: graphName
"If a graph with the given name is not visible
to this object, make one."

| stage | "graphs"
(graphs includesKey: graphName) ifTrue: [^ self].
(stage _ self ownerThatIsA: ScratchStageMorph) ifNotNil: [
            (stage graphVarNames includes: graphName)
ifTrue: [^ self]].

"graph not found; create it"
graphs at: graphName put: (ScratchGraph1Morph
           new graphName: graphName target: self).
```

A.1.6 Adicionar blocos Reporter dos grafos

```
ScriptableScratchMorph>>addGraphReportersTo: page x: x y: y
stage _ self ownerThatIsA: ScratchStageMorph.
           (stage notNil and: [stage ~= self]) ifTrue: [
           stage graphVarNames do: [:graphVarName |
           b _ GraphContentsBlockMorph new
           color: ScriptableScratchMorph graphBlockColor;
           receiver: stage blockReceiver;
           commandSpec: graphVarName;
```

```
            selector: #contentsOfGraph:.
...
(self graphVarNames size > 0) ifTrue: [
          line _ Morph new.
...
self graphVarNames do: [:graphVarName |
          b _ GraphContentsBlockMorph new
          color: ScriptableScratchMorph graphBlockColor;
          receiver: self blockReceiver;
          commandSpec: graphVarName;
          selector: #contentsOfGraph:.
...
```

A.1.7 Adicionar blocos de grafos e botões

```
ScriptableScratchMorph>>
          addGenericGraphBlocksTo: page y: startY

          | addButton deleteButton x y hasGraphs stage |
          addButton _ ScratchFrameMorph buttonLabel:
                    'Make a graph' localized selector: #addGraph.
          (self isKindOf: ScratchStageMorph)
          ifTrue: [addButton actionSelector: #addGlobalGraph].
                    deleteButton _ ScratchFrameMorph buttonLabel:
                         'Delete a graph' localized selector: #deleteGraph.
...
hasGraphs _ self graphVarNames size > 0.
          (stage _ self ownerThatIsA: ScratchStageMorph)
```

```
                    ifNotNil: [stage graphVarNames size > 0
                              ifTrue: [hasGraphs _ true]].
                    hasGraphs ifFalse: [^ self].
...
y _ (self addGraphReportersTo: page x: x y: y) + 10.
...
(self blocksFor: 'graph') do: [
...
page submorphs last color:
             ScriptableScratchMorph graphBlockColor
...
]
```

A.2 Métodos e alterações relativos ao dicionário de grafos

Os métodos criados neste protocolo para manipulação do `Dictionary` graphs da classe `ScriptabeScratchMorph` são:

- `addGlobalGraph`
- `addGraph`
- `contentsOfGraph: graphName`
- `createGraphNamed: graphName`
- `defaultGraphName`

- `deleteGraph`
- `deleteGraph: graphName`
- `graphNamed: aString ifNone: aBlock`
- `graphVarMenu`
- `graphVarNames`
- `graphs`

Os métodos de classe alterados relativos a nova cor, classe `ScriptabeScratchMorph`, são:

- `blockColorFor: aCategory`
- `initialize`
- `graphblockcolor`

Os métodos alterados protocolo variables, classe `ScriptabeScratchMorph`:

- `allvarnames`
- `ensureGraphExists: graphName`

Os métodos alterados protocolo private, classe `ScriptabeScratchMorph`:

- `copyGraphsFor: aSprite`
- `copyRecordingIn: dict c`
- `vars: varsDict lists: listsDict blocksBin: aBlocksBin graphs: graphDict`

Protocolo object i/o, classe `ScriptabeScratchMorph`:

- `storeFieldsOn: anObjStream`

- `initFieldsFrom: anObjStream version: classVersion`

Métodos alterados ou adicionados na classe `ScratchFrameMorph`:

- `deleteWatchersForSprite: c`
- `graphWatchers`
- `updatepanes c`
- `watcherShowingFor: sprite selectorAndArg: selectorAndArg c`