UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO

LINGUAGEM DE PROGRAMAÇÃO VISUAL BASEADA EM TIPOS ABSTRATOS DE DADOS

GLAUCO KNIHS

BLUMENAU
2008

2008/1-15

GLAUCO KNIHS

LINGUAGEM DE PROGRAMAÇÃO VISUAL BASEADA EM TIPOS DE DADOS ABSTRATOS

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciências da Computação — Bacharelado.

Prof. José Roque Voltolini da Silva - Orientador

BLUMENAU
2008

2008/1-15

LINGUAGEM DE PROGRAMAÇÃO VISUAL BASEADA EM TIPOS ABSTRATOS DE DADOS

Por

GLAUCO KNIHS

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

	_____
Presidente:	Prof. José Roque Voltolini da Silva, FURB
	_____
Membro:	Prof. Joyce Martins, FURB
	_____
Membro:	Prof. Mauro Marcelo Mattos, FURB

Blumenau, 10 de julho de 2008

Dedico este trabalho a web, a qual dissemina o conhecimento.

AGRADECIMENTOS

À minha família, especialmente a minha mãe por ter me incentivado todo o tempo.

Ao meu orientador, José Roque Voltolini da Silva, por ter acreditado neste trabalho e me ajudado quando precisei.

A todos amigos e professores que me acompanharam no longo período do curso de Ciências da Computação.

Poucos professores compreendem que a finalidade do ensino da história não deve consistir em aprender de cor datas e acontecimentos ou obrigar o aluno a saber quando esta ou aquela batalha se realizou, quando nasceu um general ou quando um monarca quase sempre sem significação, pôs sobre a cabeça a coroa dos seus avós. Não, graças a Deus não é disso que se deve tratar. Aprender história quer dizer procurar e encontrar as forças que conduzem às causas das ações que vemos como acontecimentos históricos. A arte da leitura como da instrução consiste nisto: conservar o essencial, esquecer o dispensável.

Adolf Hitler

RESUMO

Este trabalho descreve o processo de implementação de Tipos Abstratos de Dados (TAD) e processos concorrentes na ferramenta *Language Tangram Draw* (LTD) implementado inicialmente por Alcântara Jr. (2003) e estendida por Theiss (2006). O objetivo deste trabalho é facilitar o ensinar de programação baseada em TAD a crianças alfabetizadas através de um ambiente visual. A ferramenta é composta de um editor de modelos e um editor de mundos, onde cada modelo, formado pelas peças do jogo Tangram, representa um TAD e em um mundo vários objetos podem ser instanciados a partir dos modelos. O LTD disponibiliza uma linguagem textual ligada à programação visual. A implementação foi feita na linguagem Java utilizando a biblioteca *Java bindings for Open Graphics Library* (JOGL).

Palavras-chave: Linguagem visual. Ensino de programação. Tipos abstratos de dados. Concorrência.

ABSTRACT

This work describes the implementation process of Abstract Data Types (ADT) and concurrent processes in the tool denominated Language Tangram Draw (LTD) implemented initially by Alcântara Jr. (2003) and extended by Theiss (2006). This work aims to facilitate the programming teaching based on the ADT for alphabetized children through a visual environment. The tool is composed by an editor of models and an editor of worlds, where each model, formed by parts of the game Tangram, is an ADT and, in a world, several objects can be instantiated from a model. The LTD provides a textual language associated to visual programming. The implementation was done in Java using the library Java bindings for Open Graphics Library (JOGL).

Key-words: Visual language. Teaching of programming. Abstract data types. Concurrent processes.

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS

API – *Application Programming Interface*

AWT – *Abstract Windows Toolkit*

IDE – *Integrated Development Environment*

JOGL – *Java bindings for OpenGL*

Lisp – *LISt Processor*

LTD – *Language Tangram Draw*

LWJGL – *LightWeight Java Game Library*

OO – Orientado a Objetos

OpenAL – *Open Audio Library*

OpenGL – *Open Graphics Library*

Prolog – *PROgramming in LOGic*

RF – Requisito Funcional

RNF – Requisito Não Funcional

TAD – Tipos Abstratos de Dados

UML – *Unified Modeling Language*

SUMÁRIO

1 INTRODUÇÃO

A utilização da informática como instrumento de aprendizagem vem aumentando rapidamente de tal forma que a educação vem passando por mudanças estruturais e funcionais (ROSEMEIRE, 1998 apud SILVA; MARTINS; ALCÂNTARA JR, 2004, p. 1176). Vários softwares[1] surgiram nesses últimos anos para auxiliar o ensino, para várias faixas etárias, em diversas áreas, como por exemplo o ensino de programação de computadores a crianças.

Para programar um computador precisa-se de uma linguagem de programação. Existem vários tipos de linguagens. Por sua vez, uma linguagem segue um determinado paradigma. Entre os paradigmas cita-se o imperativo, o funcional, o lógico, o Orientado a Objetos (OO) e o visual.

Segundo Gerber (2000 apud ALCÂNTARA JR, 2003, p. 17), as linguagens imperativas e seqüenciais são orientadas a ações, onde a computação é vista como uma seqüência de instruções que manipulam valores de variáveis (leitura e atribuição). Pode-se citar a linguagem C como uma linguagem imperativa.

Sebesta (2000, p. 541) explica que uma linguagem funcional tem o propósito de imitar as funções matemáticas em seu maior grau possível. Como exemplo, cita-se a linguagem *LISt Processor* (Lisp).

Para Sebesta (2000, p. 38), as linguagens de programação lógicas baseiam-se em regras e não possuem uma ordem de execução dos comandos ou instruções. Segundo Baranauskas (1993, p. 2-3), *PROgramming in LOGic* (Prolog) é uma linguagem de programação desenvolvida em máquina seqüencial, que mais se aproxima do modelo de computação de programação em lógica.

Uma linguagem OO, para Sebesta (2000, p. 418), dispõe de três recursos: tipos de dados abstratos, herança e um tipo particular de vinculação dinâmica, como a troca de mensagens entre objetos. Java pode ser citada como uma linguagem OO.

As linguagens de programação visuais partem do princípio de que gráficos são mais fáceis de serem entendidos do que textos. Quando se especifica um programa por meio de diagramas e outros recursos gráficos, mesmo os usuários sem muita habilidade em programação podem gerar programas pelas facilidades que os recursos gráficos oferecem (GUDWIN, 1997, p. 13).

---

[1] Software, programa, ambiente e ferramenta serão utilizados como sinônimos neste documento.

O LTD, implementado inicialmente por Alcântara Jr. (2003), é uma ferramenta que possui um ambiente de programação visual, usando os sete tipos de figuras geométricas que compõem o jogo matemático Tangram e tendo como objetivo o ensino de programação de computadores a crianças. Este ambiente foi continuado e reimplementado por Theiss (2006), acrescentando novas características.

A linguagem do LTD pode ser classificada como seqüencial. No entanto, hoje a tendência é o uso de linguagens OO, como por exemplo Java. Verifica-se que linguagens OO são uma evolução de TAD. Segundo Guezzi e Jarayeri (1982, p. 36), "[...] tipos abstratos de dados escondem detalhes de representações e direcionam o acesso aos objetos por meio de procedimentos [...]".

No intuito de ampliar e melhorar o LTD, este trabalho mostra a criação de uma linguagem baseada em TAD. O objetivo dessa nova linguagem é facilitar a construção de mundos por crianças, utilizando o Tangram para representar os objetos. Com esta linguagem será possível construir um modelo, sendo que a partir desse várias instâncias poderão ser criadas. Salienta-se ainda que cada uma das instâncias será um objeto que poderá agir de forma independente. Para modelar esta independência, unidades do tipo *thread* são usadas.

O software ainda conta com um editor gráfico (interface visual) e um editor textual. Através do editor gráfico o usuário da ferramenta pode criar figuras (usando peças do Tangram) e animações das mesmas. No editor textual, comandos poderão ser gerados automaticamente a partir do editor gráfico ou podem ser diretamente criados ou modificados (digitados).

O ambiente foi implementado na linguagem Java e para os recursos gráficos utiliza-se a *Application Programming Interface* (API) JOGL.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é criar uma linguagem de programação baseada em TAD para o ambiente LTD.

Os objetivos específicos do trabalho são:

a) disponibilizar o uso de TAD na nova linguagem;
b) manter as características do ambiente do LTD com relação à edição, compilação e execução de programas;

c) dividir o editor visual em editor de modelos (figuras) e editor de mundos, onde no mundo poderão ser instanciados objetos a partir de modelos, criados no editor de modelos;
d) disponibilizar um mecanismo para criar métodos para cada figura, junto ao editor de modelos;
e) permitir que sejam criadas várias instâncias a partir de um modelo;
f) permitir que os métodos dos modelos instanciados possam ser chamados pelo editor de mundos;
g) modelar cada objeto[2] do mundo como sendo um processo concorrente.

1.2 ESTRUTURA DO TRABALHO

O capítulo 2 apresenta uma explanação de assuntos relacionados ao trabalho, tais como: linguagens de programação, tipos abstratos de dados, processos concorrentes, a linguagem Java e a biblioteca gráfica JOGL, o Tangram, o ambiente LTD, vetores e trabalhos correlatos. No capítulo 3 é descrito o desenvolvimento da nova versão da ferramenta. Por fim, o capítulo 4 traz as conclusões do trabalho.

[2] Uma instância de um tipo de dado abstrato é chamada de objeto (SEBESTA, 2000, p. 398).

2 FUNDAMENTAÇÃO TEÓRICA

A seguir são apresentados os assuntos sobre linguagens de programação, tipos abstratos de dados, processos concorrentes, a linguagem Java e a biblioteca gráfica JOGL, o jogo temático TANGRAM, o ambiente LTD, vetores e quatro (4) trabalhos correlatos.

2.1 LINGUAGENS DE PROGRAMAÇÃO

O conjunto de instruções de um processador, que é conhecido como linguagem de máquina, é pouco amigável para o desenvolvimento de programas. Linguagens de programação surgiram para implementação de algoritmos, complexos ou não, em um computador, tornando a tarefa de programar mais fácil e menos susceptível a erros. Ao longo do tempo as linguagens de programação foram introduzindo facilidades e recursos, e atualmente, com as linguagens visuais, programar deixou de ser uma arte restrita a poucos, tornando-se mais uma ferramenta para os usuários comuns (GUDWIN, 1997, p. 1).

Gudwin (1997, p. 14-15) ainda afirma que as linguagens visuais são divididas em duas categorias:

a) linguagens de programação visuais híbridas: incluem-se as linguagens de programação que não são somente visual, mas têm parte de sua programação determinada de forma visual. Muitas dessas linguagens de programação são ferramentas de programação visual. Como exemplos, incluem-se Delphi e Visual Basic;
b) linguagens de programação visuais puras: nessa categoria incluem-se as linguagens de programação em que os programas são determinados apenas por meio visual. O paradigma mais eficiente é a elaboração de diagramas com arcos e nós, limitando seu uso a aplicações bem definidas. Duas linguagens de programação visual puras são o Khoros e o Simulink.

Com a evolução das linguagens de programação, novas técnicas foram surgindo, como a construção de TAD e a programação OO. Essas técnicas são progressivamente incorporadas as linguagens, tornando-as mais comuns entre os programadores (RANGEL, 1993).

2.2 TIPOS ABSTRATOS DE DADOS

> O nome tipos abstrato de dados vem da matemática, do estudo de estruturas algébricas compostas por domínios de valores e de operações sobre esses domínios. Um domínio de valores corresponde em programação a um tipo, e as operações sobre valores do domínio correspondem a procedimentos e funções que manipulam os valores do tipo. Usamos o adjetivo abstrato para enfatizar que devemos nos abstrair da forma exata de implementação, durante a utilização do tipo, levando em consideração apenas às propriedades especificadas dos valores do tipo e das operações. (RANGEL, 1993).

Ainda segundo Rangel (1993), as linguagens de programação baseadas em TAD utilizam dois mecanismos fundamentais, encapsulamento e ocultação de informação.

Escrever programas com milhares de linhas, o que muitas pessoas já fizeram, torna difícil de mantê-los se o programador decidir por fazê-lo em apenas uma coleção de subprogramas. Uma solução para este problema seria projetá-los em recipientes sintéticos, também chamados de módulos. Outro problema em ter programas maiores é a recompilação, já que em uma pequena alteração é necessária a compilação de todo o programa. Para resolver isso, o programador deve organizar os programas em coleções de subprogramas, junto com os dados manipulados pelo subprograma, que podem ser compilados separadamente. O encapsulamento resolve os dois problemas descritos anteriormente, já que ele é um agrupamento de subprogramas com seus dados, constituindo um sistema abstrato, organizado logicamente para execução de uma coleção de computações relacionadas (SEBESTA, 2000, p. 397-398).

Ocultar as informações, para Sebesta (2000, p. 399-400), tem as mesmas vantagens do encapsulamento, pois os dados não são visíveis pelas unidades que o utilizam, sendo que são acessados apenas pelas operações oferecidas pela definição do TAD. Desta forma as operações sobre objetos de um TAD estão contidas em apenas uma unidade, podendo ser compiladas separadamente. Além disso, há um aumento de confiabilidade, já que os dados do TAD não podem ser alterados diretamente por outro TAD, o que afetaria sua integridade.

2.3 PROCESSOS CONCORRENTES

Para Sebesta (2000, p. 471), a concorrência tem sua importância por pelo menos dois motivos. Primeiramente, muitos problemas prestam-se naturalmente à concorrência, como

programas de simulação, onde muitas vezes existem várias entidades físicas que agem de forma simultânea. O segundo está no fato que atualmente estão sendo utilizados computadores com múltiplos processadores e processadores com múltiplos núcleos, criando assim a necessidade que o software use efetivamente este recurso.

Sebesta (2000, p. 470) ainda divide a concorrência em duas categorias distintas:

a) concorrência física: acontece quando existe um processador para cada programa ou parte dele em execução;
b) concorrência lógica: acontece quando existe um processador para vários processos, sendo que cada um é executado um pouco de cada vez de forma intercalada.

Para compreender a idéia de concorrência, é importante entender os conceitos de processos e *threads*.

Os programas de computador são seqüências de instruções a serem executadas, e quando são executadas são chamados de processos. Cada processo é uma entidade ativa com seu próprio contexto, identificação, informações de controle, variáveis de ambiente e sua área de memória. Desta forma um programa pode ser executado por mais de um usuário sem que um processo interfira no outro (VAREJÃO, 2004, p. 280–281).

Threads são fluxos de execução concorrentes em um mesmo processo. Os *threads* compartilham os recursos do processo que o criou e cada *thread* pode ser uma função ou procedimento de um programa. Enquanto os processos manipulam grandes quantidades de dados e têm um custo expressivo para alterar sua execução entre outros processos, *threads* são processos leves com baixo custo para alternar sua execução com outros *threads* do mesmo processo, já que compartilham o mesmo contexto e recursos do processo (VAREJÃO, 2004, p. 282-283). Para Sebesta (2000, p. 471) cada *thread* é uma tarefa que pode estar em concorrência com outra tarefa em um programa.

Uma tarefa pode compartilhar recursos e comunicar-se com outras tarefas através de mensagens ou parâmetros. Quando uma tarefa não compartilha recursos ou não interfere na execução de outra tarefa, diz-se que é uma tarefa disjunta. No caso de interferir na execução de outra tarefa é necessário um mecanismo de sincronização, que controle a execução das tarefas. Existem dois tipos de sincronização: de cooperação e de competição. Sincronização de cooperação é quando uma tarefa produz um valor ou recurso que é usado por outras tarefas. Sincronização de competição impede que duas tarefas utilizem um recurso compartilhado ao mesmo tempo (SEBESTA, 2000, p. 471-472).

2.4 A LINGUAGEM JAVA E A BIBLIOTECA GRÁFICA JOGL

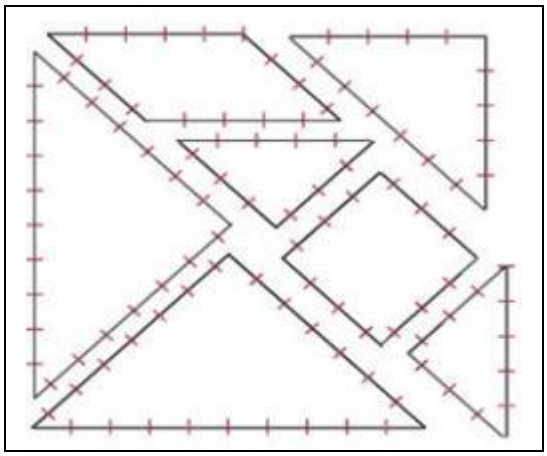
Java foi lançado oficialmente em maio de 1995 e gerou um interesse imediato pela possibilidade de desenvolver páginas interativas e dinâmicas à internet. Nos dias de hoje, além de páginas da internet, encontra-se softwares para celulares, *pagers* e para muitas outras finalidades destinadas ao consumidor final (DEITEL; DEITEL, 2002, p. 59).

A linguagem Java possui sua própria biblioteca gráfica, a Java3D, que é executada no topo das bibliotecas gráficas *Open Graphics Library* (OpenGl) e DirectX (DAVISON, 2007, p. 3). Uma aplicação de Java 3D é projetada a partir de um grafo de cena contendo objetos gráficos, luz, som, objetos de interação, entre outros, que possibilitam a criação de mundos virtuais com personagens interativos (SOWIRZAL; NADEAU; BAILEY, 1998).

Existem outros modos para desenvolver aplicações tridimensionais além do Java3D, como o JOGL e o *LightWeight Java Game Library* (LWJGL). A API JOGL surgiu para que aplicações desenvolvidas em Java possam ter suporte direto a biblioteca OpenGL através de rotinas das bibliotecas *Abstract Window Toolkit* (AWT) e *Swing*. O LWJGL, além de permitir o acesso a biblioteca OpenGL, possibilita o acesso a dispositivos como *joystics*, volantes, teclados e a *Open Audio Library* (OpenAL) (TOSTES, 2006, p. 31-35). Ainda segundo Tostes (2006, p. 33-34), JOGL oferece mecanismos para lidar com aplicações que utilizam muitas *threads* e gerencia os recursos gráficos através de várias *threads*.

2.5 TANGRAM

O Tangram, conhecido na China por volta do século VII a.C. como as “Sete Taboas da Astúcia”, é um jogo baseado em figuras. A versão mais contada sobre a sua origem é a de que o monge Tai-Jin deu uma missão ao seu discípulo Lao-Tan, que consistia em percorrer o mundo e registrar numa placa de porcelana toda a beleza que encontrasse. Muito emocionado por ter sido escolhido para essa missão, o discípulo deixou cair a placa quadrada de porcelana, que quebrou-se em sete pedaços, como as peças (tans) do Tangram (Figura 1) (LONGHI, 2004).



Fonte: Kong (2003).

Figura 1 – Peças do Tangram

As regras básicas do Tangram são as seguintes (KONG, 2003):

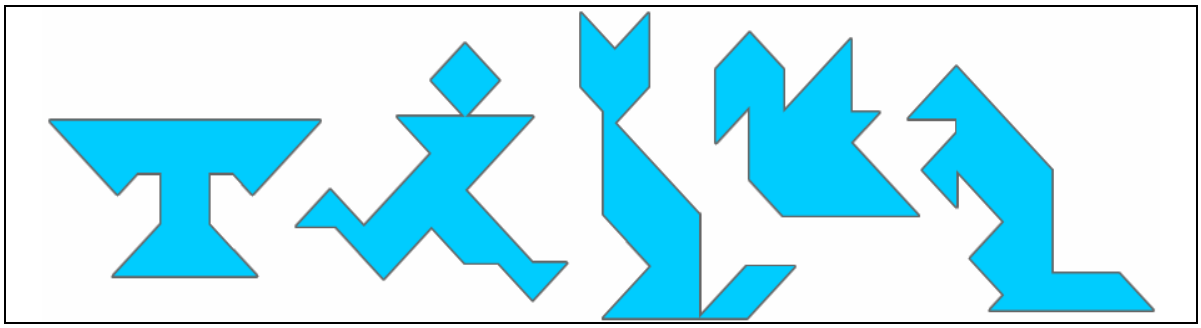
a) tem de utilizar os sete (7) tans;

b) os tans têm que estar deitados;

c) os tans têm que se tocar;

d) nenhum tan pode sobrepor-se a outro.

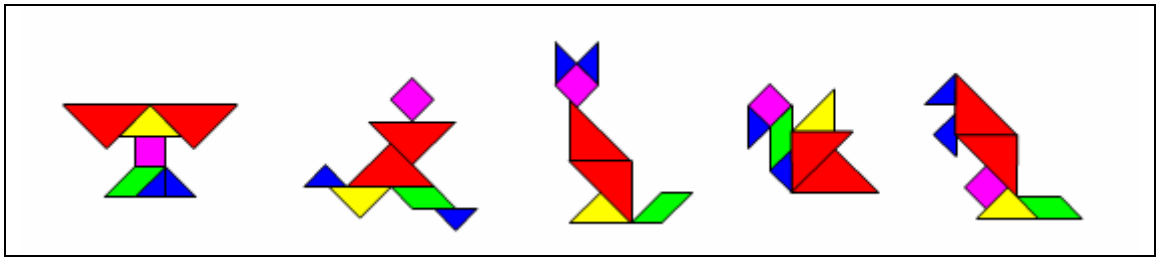
A partir das sete (7) peças do Tangram, desenhos podem ser feitos. Na figura 2, da esquerda para a direita, tem-se respectivamente uma fonte, um corredor, um gato um cisne e um canguru.



Fonte: Serafim (2008).

Figura 2 – Desenhos utilizando as sete (7) peças do Tangram

Na figura 3 são identificadas, individualmente, cada uma das peças que compõem os desenhos apresentados na figura 2.



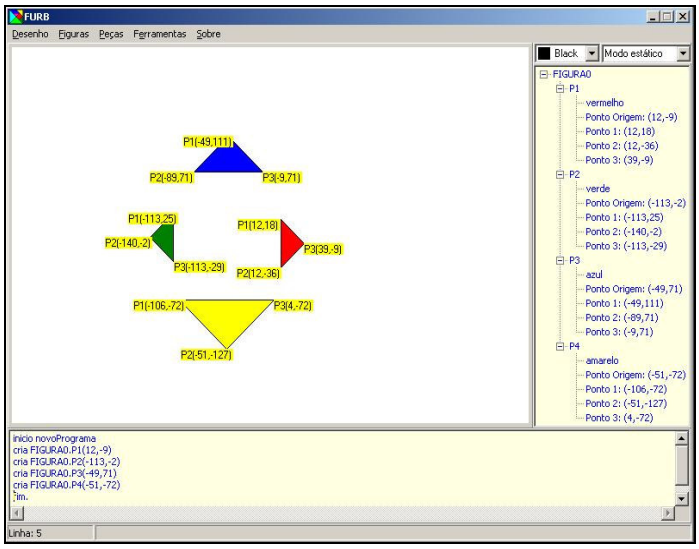
Fonte: Serafim (2008).

Figura 3 – Identificação das peças dos desenhos apresentados na figura 2

2.6 O AMBIENTE LTD

Alcântara Jr (2003) desenvolveu o LTD (Figura 4), o qual possui um ambiente de programação visual voltado ao ensino de programação para crianças alfabetizadas. O software disponibiliza uma linguagem textual de ligação com a programação visual, formando duas áreas de trabalho, um editor de figuras e um de texto. Através do editor de figuras, textos (comandos) são gerados, sendo que estes também podem ser alterados diretamente.

Theiss (2006) re-implementou o LTD, criando uma segunda versão, corrigindo funções que não funcionavam como o esperado e implementando novas funcionalidades, como a visualização da terceira dimensão, permitindo o mesmo ser mostrado em perspectiva (profundidade). Ainda funcionalidades como a seleção de figuras e peças, o editor de figuras de tamanho dinâmico e a separação dos editores gráfico e textual também foram agregadas. A API OpenGL foi utilizada para fazer a implementação.



Fonte: Theiss (2006, p. 75).

Figura 4 – Ambiente LTD

O LTD usa uma linguagem seqüencial, onde os programas são descritos. Não usa TAD e também não permite a modelagem de processos para serem executados de forma independentes (concorrentes).

2.7 VETORES

As sete (7) peças do Tangram são compostas por três (3) formas geométricas diferentes, as quais são: cinco (5) triângulos, um (1) quadrado e um (1) trapézio.

As formas geométricas são representadas por um conjunto de vértices que, junto com a indicação de como estes vértices estão conectados, podem produzir triângulos, quadriláteros, linhas, pontos, entre outras formas (LENGYEL, 2004, p. 1). Por sua vez, cada vértice é representado por um vetor de tantos valores reais quanto o número de dimensões do espaço.

Cada forma geométrica pode sofrer transformações, tais como: translação, escalonamento e rotação. Para realizar estas transformações é necessário conhecer as propriedades dos vetores e como manipular seus valores.

A seguir são apresentadas as propriedades e fórmulas matemáticas para manipulação de vetores em um espaço tridimensional.

2.7.1 PROPRIEDADES DOS VETORES

Segundo Lengyel (2004, p. 12), um vetor é representado por n valores reais. O quadro 1 mostra a representação do vetor V de n dimensões.

$$\mathbf{V} = \langle V_1, V_2, \ldots, V_n \rangle,$$

Fonte: Lengyel (2004, p. 12).
Quadro 1 – Equação de um vetor

A multiplicação de um valor escalar a por um vetor V (Quadro 2) produz um outro vetor com os mesmo elementos multiplicados pelo valor escalar.

$$a\mathbf{V} = \mathbf{V}a = \langle aV_1, aV_2, \ldots, aV_n \rangle.$$

Fonte: Lengyel (2004, p. 12).
Quadro 2 – Equação do produto de vetor por escalar

Outra propriedade é a soma, ou subtração, entre dois vetores, sendo que neste caso cada elemento de um vetor é somado, ou subtraído, com o respectivo elemento do outro vetor. A soma do vetor P com o vetor Q é definida no quadro 3.

$$\mathbf{P} + \mathbf{Q} = \langle P_1 + Q_1, P_2 + Q_2, \ldots, P_n + Q_n \rangle.$$

Fonte: Lengyel (2004, p. 12).
Quadro 3 – Equação da soma entre dois vetores

O módulo de vetor V de *n* dimensões é definida como ||V|| e sua fórmula é apresentada no quadro 4. O módulo de um vetor é conhecido também como magnitude, ou comprimento, de um vetor. Quando um vetor tem o módulo igual a um (1) é conhecido como vetor unitário.

$$\|\mathbf{V}\| = \sqrt{\sum_{i=1}^{n} V_i^2}.$$

Fonte: Lengyel (2004, p. 13).
Quadro 4 – Fórmula do módulo de um vetor

Um vetor V que possua ao menos um elemento diferente de zero (0) pode ser transformado em um vetor unitário multiplicando-o por 1/||V||. Está operação chama-se normalização.

Segundo Lengyel (2004, p. 14) o produto escalar, ou produto interno, entre dois vetores é uma das operações mais usadas na computação gráfica porque permite calcular a diferença entre a direção de dois pontos. A equação do produto escalar entre os vetores P e Q de *n* dimensões é definida no quadro 5.

$$\mathbf{P} \cdot \mathbf{Q} = \sum_{i=1}^{n} P_i Q_i.$$

Fonte: Lengyel (2004, p. 15).
Quadro 5 – Equação do produto escalar entre dois vetores

O produto vetorial entre dois vetores resulta em um novo vetor, o qual é perpendicular aos vetores de origem. Esta operação pode ser usada para calcular a normal de um triângulo ou polígono, sendo útil para determinar colisões, simular iluminação entre outras funções. A equação entre dois vetores tridimensionais P e Q é apresentada no quadro 6.

$$\mathbf{P} \times \mathbf{Q} = \langle P_y Q_z - P_z Q_y, P_z Q_x - P_x Q_z, P_x Q_y - P_y Q_x \rangle.$$

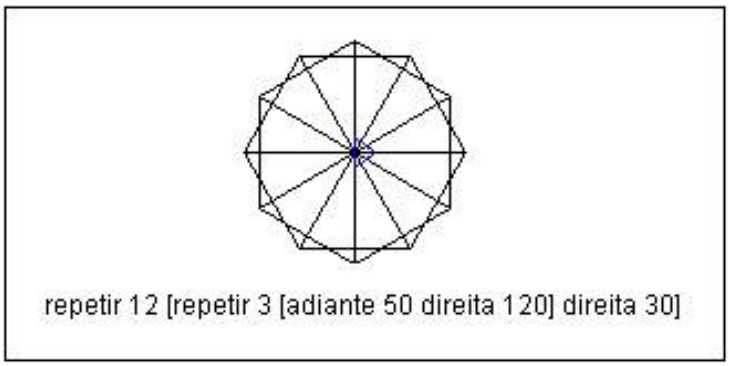
Fonte: Lengyel (2004, p. 20).
Quadro 6 – Equação do produto vetorial entre vetores tridimensionais

2.8 TRABALHOS CORRELATOS

A seguir são apresentadas duas ferramentas voltadas para o ensino de programação, o Logo e o Mundo dos Atores, juntamente com outras duas ferramentas de programação visual, o Khoros e o Simulink.

A linguagem de programação Logo (Figura 5) possui características para explorar o

processo de aprendizagem. O objetivo do Logo é manipular uma tartaruga através de comandos. A tartaruga responde aos comandos fornecidos movendo-se no ambiente gráfico. Para comandar a tartaruga são usados conceitos espaciais, os quais estão presentes de forma intuitiva nos usuários. Com a explicação e o entendimento destes comandos, condições para o desenvolvimento de conceitos espaciais, numéricos e geométricos são proporcionadas (ZACHARIAS, 2007). Outra característica importante da linguagem Logo é que ela é considerada procedural, facilitando assim a criação de novos termos ou procedimentos (VALENTE, 1991, p. 32-43).



Fonte: Zacharias (2007).

Figura 5 – Linguagem Logo

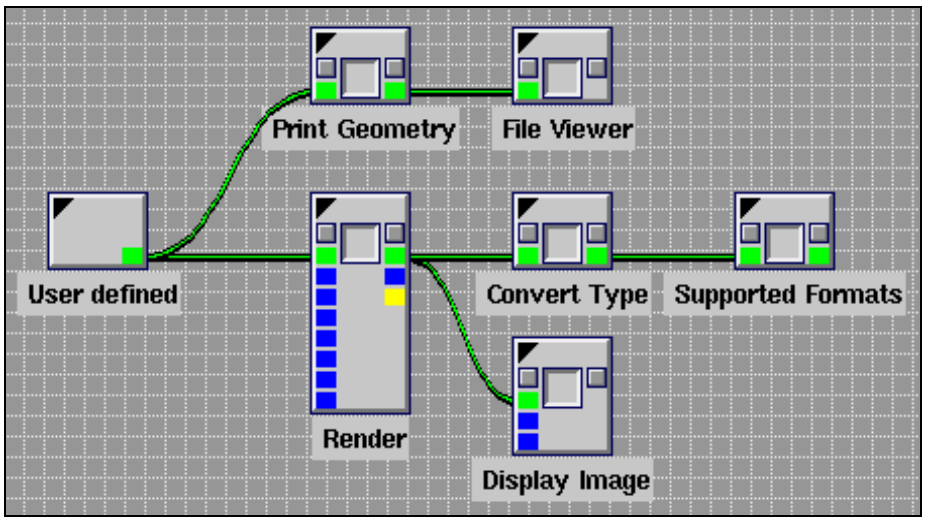
No Mundo dos Atores (Figura 6), um ambiente similar ao oferecido pelo Logo, o usuário (aprendiz) controla um ator, como por exemplo uma caneta, fazendo-a traçar vários desenhos na área gráfica (palco). O objetivo desta ferramenta é facilitar o ensino de linguagem de programação OO. Interagindo com atores, o aprendiz adquire várias noções de programação. Entre as noções pode-se citar a de algoritmos, variáveis, parâmetros, estrutura de controle, modularização e reusabilidade. O Mundo dos Atores faz uma analogia com um teatro, onde atores atuam num palco, similar aos reais. O palco ainda pode ser modificado para se adequar a uma nova situação. Outros atores, além da caneta, podem ser inseridos (MARIANI, 1998).

Fonte: Mariani (1998).

Figura 6 – Mundo dos Atores

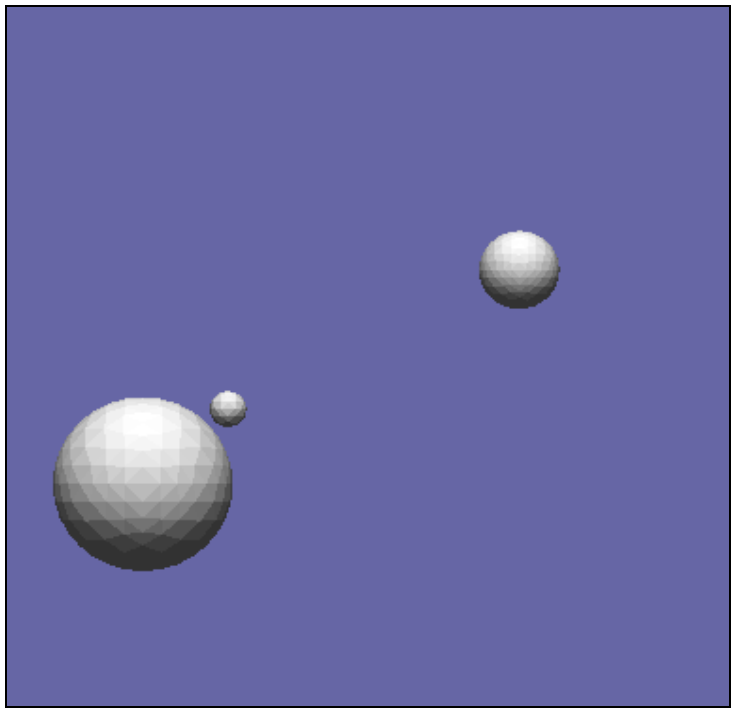
> O Khoros é uma linguagem de programação visual utilizada principalmente em computação gráfica. Na verdade o Khoros é mais do que simplesmente uma linguagem, mas todo um "ambiente" de programação gráfica, que permite ao programador efetuar o processamento de imagens e análises sofisticadas por meio da elaboração de diagramas com nós e arcos. O Khoros vem sendo utilizado com resultados muito positivos tanto para a análise como síntese de imagens gráficas, principalmente nas áreas médicas. Entretanto, por ter uma especificação aberta, pode ser utilizado em princípio em qualquer outro tipo de aplicação. (GUDWIN 1997, p. 15).

A figura 7 apresenta a área de trabalho do ambiente Khoros e a figura 8 mostra um possível resultado do projeto desenvolvido (Figura 7). Santos (1997) disponibiliza um tutorial de como programar no ambiente Khoros.



Fonte: Santos (1997).

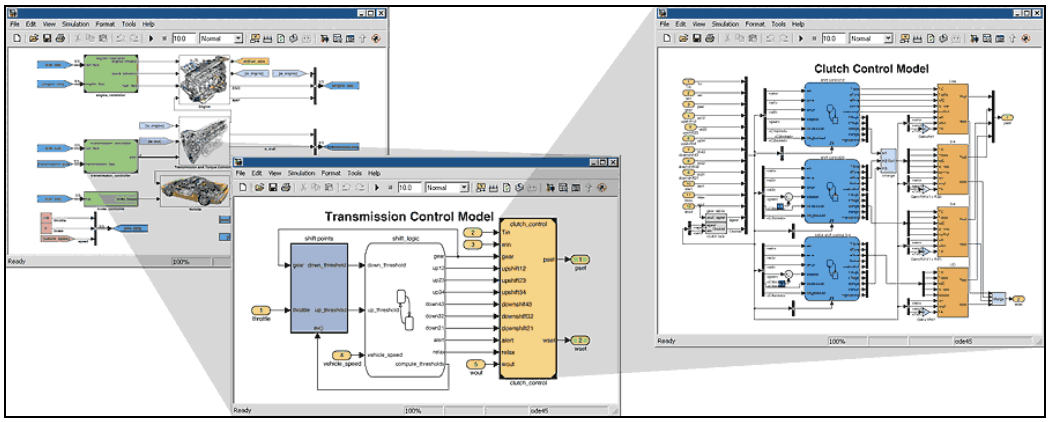
Figura 7 – Ambiente Khoros



Fonte: Santos (1997).

Figura 8 – Programa Khoros

O Simulink (Figura 9) é um ambiente de simulação para modelagem de sistemas dinâmicos e incorporados. Ele possui um ambiente gráfico interativo e um conjunto de bibliotecas que permitem projetar, simular, implementar e testar uma variedade de sistemas, incluindo comunicações, controles, processamento de sinal, processamento de vídeo e processamento de imagem (MATHWORKS, 2008).



Fonte: Mathworks (2008).

Figura 9 – Simulink

O Simulink está integrado com o MATLAB, fornecendo acesso imediato a uma ampla gama de instrumentos que permitem desenvolver algoritmos, analisar e visualizar simulações, criar processos *batch scripts*, modelar e personalizar o ambiente através de definições de sinais, parâmetros e outros dados para as simulações (MATHWORKS, 2008).

3 DESENVOLVIMENTO DO SOFTWARE

Neste capítulo são apresentados os requisitos, a especificação, a implementação e a operacionalidade da ferramenta. Ainda, resultados e discussão são relatados.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

Os requisitos para o novo LTD são:

a) disponibilizar uma linguagem baseada em TAD em um contexto de concorrência (Requisito Funcional – RF);
b) criar novos comandos na linguagem para suportar TAD e concorrência (RF);
c) manter as mesmas funcionalidades do ambiente LTD desenvolvido por Alcântara (2003) e estendido por Theiss (2006), as quais são: mover uma peça ou uma figura, rotacionar uma peça ou uma figura em um ponto selecionado, espelhar uma peça, mudar a cor das peças ou de uma figura, mudar a cor de fundo e movimentar a câmera (RF);
d) apresentar um editor visual de modelos de figuras para desenvolver os TAD (RF);
e) separar o editor visual em duas partes, sendo uma para editar os modelos e outra para editar as animações (Requisito Não Funcional – RNF);
f) ser implementado em Java, utilizando JOGL para desenvolver os recursos gráficos (RNF).

3.2 ESPECIFICAÇÃO

A especificação da nova linguagem baseada em TAD, tomando como referência o descrito em Silva, Martins e Alcântara Jr (2004), é apresentada com as alterações necessárias para a nova construção. As definições regulares, *tokens*, e a gramática da linguagem são especificados em notação especifica para o ambiente Gerador e Analisador Léxico e Sintático (GALS) (GESSER, 2003). A especificação do ambiente é feita utilizando a *Unified Modelling*

Language (UML) através dos diagramas de casos de uso, de classes e de seqüência. O *Integrated Development Environment* (IDE) Netbeans 6.0 é usado para descrever a especificação.

3.2.1 Especificação da nova linguagem LTD

Nesta seção são apresentados a definição da linguagem, as definições regulares, os *tokens* e a gramática da nova linguagem, baseada na definição de Theiss (2006), em notação GALS (GESSER, 2003).

3.2.1.1 Definição da linguagem

Para incluir TAD no ambiente do Tangram são necessárias alterações na linguagem LTD.

A representação de cada figura do Tangram, composta por sete peças, é um TAD identificado como um modelo. Cada modelo poderá ter vários métodos. Cada método, por sua vez, terá associado um bloco de comandos.

Cada modelo criado tem quatro métodos nativos, os quais são:

a) `CRIA`: contém comandos para criar e posicionar as peças quando um modelo é criado. Os comandos deste método só podem ser alterados no editor de modelos;
b) `VIVA`: executa repetidamente um método do modelo. Para finalizar a execução deste método é necessário executar o método `TERMINA`. O método `VIVA` não pode ser alterado, podendo apenas ser executado através do mundo;
c) `TERMINA`: pára a execução do método `VIVA`. Este método não pode ser alterado, podendo apenas ser executado através do mundo;
d) `APAGA`: apaga o modelo do mundo. Este método não pode ser alterado, podendo apenas ser executado através do mundo.

Cada modelo pode ser desenvolvido e salvo separadamente no editor de modelos. Depois de desenvolvido um ou mais modelos, pode-se criar um mundo, através do editor de mundos, que contenha uma ou mais instâncias de um ou mais modelos.

Os vários modelos criados em um mundo têm seus métodos executados simultaneamente, categorizando o uso de processos concorrentes. Além disso, existe a

possibilidade de executar um método de um modelo seqüencialmente, fazendo que o próximo comando do mundo seja executado apenas após o término do método executado.

3.2.1.2 Exemplo de programas na nova linguagem

No quadro 7 é apresentado um programa utilizando todos os comandos de um modelo, os quais são: cria; gira; espelha; cor; repita; move; pisca e faça.

```
01  modelo Galinha
02    metodo CRIA
03      cria p1(-62,-150,-7000) gira(270.0) espelha
04      cria p2(-709,353,-7000) espelha
05      cria p3(790,980,-7000) gira(45.0) espelha cor(vermelho)
06      cria p4(285,-414,-7000) espelha cor(marrom)
07      cria p5(-394,-395,-7000) gira(90.0) espelha cor(marrom)
08      cria p6(588,644,-7000) gira(45.0) espelha cor(azul)
09      cria p7(415,279,-7000) gira(45.0)
10    fim;
11    metodo VAI
12      repita 10 vezes
13      inicio
14        Galinha.move(30,0,0)
15        pisca(100)
16      fim
17    fim;
18    metodo VOLTA
19      repita 10 vezes
20      inicio
21        Galinha.move(-30,0,0)
22        pisca(100)
23      fim
24    fim;
25    metodo VIRA
26      Galinha.espelha
27    fim;
28    metodo MEXE_CABECA
29      p3.gira(-10) no ponto (15)
30      pisca(100)
31      p3.gira(10) no ponto (15)
32      pisca(100)
33    fim;
34    metodo VAI_E_VOLTA
35      faça VAI
36      faça VIRA
37      faça VOLTA
38      faça VIRA
39    fim;
40  fim.
```

Quadro 7 – Código de um modelo

Um TAD (modelo) começa com a palavra reservada `modelo` seguida do nome do modelo, o qual, no quadro 7, é `Galinha` e termina com a palavra `fim` seguida de um ponto.

Um método começa com a palavra reservada `metodo`, seguida do nome do método, e

termina com a palavra `fim` seguida de ponto e vírgula. O bloco de comandos do método é inserido entre as duas palavras reservadas `metodo` e `fim`.

No método `CRIA`, linha 2 do quadro 7, são inseridos os comando para criar as peças, as quais são: `p1`, `p2`, `p3`, `p4`, `p5`, `p6` e `p7`. O comando que cria uma peça começa com `cria` seguido da peça que se deseja criar, abre parênteses, os três valores dos eixos x, y e z separados por vírgula e, para terminar, fecha parênteses. Após criar uma peça é possível girá-la, espelhá-la e mudar sua cor, sendo que para isso basta incluir o respectivo comando após o comando cria, conforme apresentado nas linhas 3 a 9 do quadro 7.

O comando gira começa com a palavra `gira` seguida de abre parêntese, o ângulo de rotação e fecha parêntese. É possível escolher o ponto de rotação da peça (linha 29 do quadro 7) incluindo as palavras `no ponto` seguida de abre parêntese, o número do ponto e fecha parêntese. O número do ponto é único para cada modelo, ou seja, cada ponto de cada peça do modelo tem um número distinto. Além dos vértices das peças serem considerados como pontos, o centro da mesma também é considerado um ponto. Caso não seja incluído o ponto de rotação, o centro da peça será assumido.

Para o comando espelha basta a palavra reservada `espelha`.

O comando muda cor começa com a palavra `cor` seguida de abre parêntese, o nome da cor e fecha parêntese. Os nomes das cores existentes estão descritos no Apêndice A.

Os comandos gira, espelha e muda cor podem ser utilizados separados do comando `cria`, sendo que para isso é preciso informar em qual peça o comando será executado, ou se deseja executar o comando em todas as peças do modelo. Neste caso, deve-se usar o nome dado ao modelo (linha 14 do quadro 7). Após informar a peça ou modelo, em que será executado o comando, coloca-se um ponto seguido do comando desejado. Além destes comandos existe o comando move (linha 14 do quadro 7) que pode ser executado em uma peça ou no modelo.

O comando move começa com a palavra `move` seguida de abre parêntese, os três valores dos eixos x, y e z separados por vírgula e, para terminar, fecha parêntese. O comando move soma os valores de x, y e z aos respectivos valores da peça ou modelo e move para esta nova coordenada.

Os outros comandos possíveis para um modelo são: repita, pisca e faça. O comando repita (linha 12 do quadro 7) começa com a palavra `repita` seguida do número de repetições, as palavras `vezes` e `inicio`, o bloco de comandos e a palavra `fim`. O comando pisca (linha 15 do quadro 7) começa com a palavra `pisca` seguida de abre parêntese, o tempo em milisegundos e fecha parêntese. O comando faça (linha 35 do quadro 7) permite a execução

de um método do próprio modelo. Este comando começa com a palavra `faça` seguido do nome do método a ser executado.

No quadro 8 é apresentado um programa utilizando todos os comando para criação de um mundo.

```
01  mundo Fantastico
02    cria nomeA como Galinha(-1000,-1000,-9000)
03    cria nomeB como Galinha(1000,1000,-9000)
04    cria nomeC como OutroModelo(0,0,-9000)
05
06    faça nomeA.VAI_E_VOLTA em paralelo
07    faça nomeB.VAI_E_VOLTA
08    faça nomeC.VIVA(nomeDoMetodo)
09    repita 10 vezes inicio
10      pisca(10)
11    fim
12    faça nomeA.APAGA
13    faça nomeC.TERMINA
14  fim.
```

Quadro 8 – Código de um mundo

Um mundo começa com a palavra reservada `mundo` seguido dos seus comandos e termina com a palavra `fim` seguida de um ponto.

Os comandos das linhas 2 a 4 do quadro 8 instanciam objetos de modelos no mundo. Para isso é utilizada a palavra `cria` seguido do nome que identifica a instância, seguido por `como` e o nome do modelo, o qual foi criado no editor de modelos. Após é informado a coordenada onde o modelo será criado, para isso, informa-se abre parêntese, os valores de `x`, `y` e `z` separados por vírgula e fecha parêntese.

Para executar um método de um modelo no mundo utiliza-se a palavra reservada `faça`, seguida no nome que identifica a instância, seguida de ponto e o nome do método. Na linha 7 do quadro 8 é executado um método da instância identificado por `nomeB`. As palavras `em paralelo` (linha 6 do quadro 8) são opcionais e identificam se o comando será executado simultaneamente aos outros comandos do mundo. Na execução do comando da linha 6 do quadro 8, após a chamada do método `VAI_E_VOLTA` do objeto `nomeA`, o comando da linha 7, o qual chama o método `VAI_E_VOLTA` do objeto `nomeB`, é executado simultaneamente. O comando da linha 8 do quadro 8 só será executado após o termino do método `VAI_E_VOLTA` do objeto `nomeB`, pois na ativação desse não são utilizadas as palavras `em paralelo`.

A sintaxe para chamar os métodos `VIVA`, `APAGA` e `TERMINA` de um modelo são apresentados, respectivamente, nas linhas 8, 12 e 13 do quadro 8.

3.2.1.3 Definições regulares

Para especificar a gramática na ferramenta GALS é necessário desenvolver primeiro as definições regulares, as quais são apresentadas no quadro 9.

```
id: [a-zA-Z]+[a-zA-Z0-9_ç]*
ws:[\ \t\s\r\n]+
comentario: /[/][^\n]+
multilinha:(/\*)([^\*]|\*[^/])*(\*/)
```

Quadro 9 – Definições regulares

A palavra `id` (Quadro 9) é um identificador para uma definição regular. Após o operador dois pontos (“`:`”) é especificada a definição regular. A seqüência de caracteres `[a-zA-Z]` define que o `id` deve começar com uma letra minúscula ou maiúscula. O operador mais (“`+`“) significa que após a primeira podem ser concatenados, ao `id`, outros caracteres definidos por `[a-zA-Z0-9_ç]`. Esta definição permite, além de letras minúsculas e maiúsculas, números e os caracteres *underline* (“_“) e cê-cedilha (“`ç`”). O operador asterisco (“`*`“) significa nenhum, um ou vários.

3.2.1.4 *Tokens*

A partir das definições regulares são definidos os *tokens* (Quadro 10).

```
identificador:{id}+
 : {ws}+
numero: [\+\-]?[0-9]+[\.0-9]*
modelo = identificador: "modelo"
metodo = identificador: "metodo"
mundo = identificador: "mundo"
CRIA = identificador: "CRIA"
VIVA = identificador: "VIVA"
APAGA = identificador: "APAGA"
TERMINA = identificador: "TERMINA"
gira = identificador: "gira"
cor = identificador: "cor"
espelha = identificador: "espelha"
repita = identificador: "repita"
em = identificador: "em"
paralelo = identificador: "paralelo"
faca = identificador: "faça"
depois = identificador: "depois"
de = identificador: "de"
no = identificador: "no"
ponto = identificador: "ponto"
inicio = identificador : "inicio"
fim = identificador: "fim"
cria = identificador: "cria"
como = identificador: "como"
move = identificador: "move"
piscar = identificador: "pisca"
vezes = identificador: "vezes"
Peca1 = identificador: "p1"
Peca2 = identificador: "p2"
Peca3 = identificador: "p3"
Peca4 = identificador: "p4"
Peca5 = identificador: "p5"
Peca6 = identificador: "p6"
Peca7 = identificador: "p7"
amarelo = identificador: "amarelo"
azul = identificador: "azul"
azulMarinho = identificador : "azulMarinho"
azulPiscina = identificador: "azulPiscina"
branco = identificador: "branco"
cinza = identificador: "cinza"
marrom = identificador: "marrom"
oliva = identificador: "oliva"
prata = identificador: "prata"
preto = identificador: "preto"
rosa = identificador: "rosa"
verde = identificador: "verde"
verdePiscina = identificador: "verdePiscina"
verdeLima = identificador: "verdeLima"
vermelho = identificador: "vermelho"
violeta = identificador: "violeta"
";" : ;
"(" : \(
")" : \)
"," : ,
"." : .
:{comentario}
:{multilinha}
```

Quadro 10 – *Tokens*

A palavra `identificador` (Quadro 10), representa um *token* definido pela definição regular `id`. A palavra `modelo` tem a sua definição a partir da definição de outro *token*. A expressão que representa `modelo` é informada entre aspas duplas.

3.2.1.5 Gramática

A especificação da gramática da nova linguagem é apresentada no quadro 11.

```
<codigo> ::= <modelo> | <mundo> | <metodo_cria> #2 ;
<modelo> ::= modelo  #0 <figura> <Lista_de_metodos> fim "." #27 ;
<Lista_de_metodos> ::= <metodo_cria> <metodos_outros> ;
<metodos_outros> ::= <metodo> <metodos_outros> | î ;
<metodo_cria> ::= metodo CRIA #4 #12 <bloco> fim ";" #26 #36 ;
<metodo> ::= metodo <nome_Metodo> #12 <bloco> fim ";" #26 ;
<bloco> ::= <comando> <bloco_2> ;
<bloco_2> ::= <bloco> | î ;
<comando> ::= <comando_cria> | <comandos_em_ids> |  <comando_piscar> #22
#14 | <comando_repete> | <comando_faca> ;
<comandos_em_ids> ::= <id> "." <comando_de_id> ;
<comando_de_id> ::= <comando_move> #14 | <comando_gira> #14 |
<comando_cor> #14 | <comando_espelha> #14 ;
<comando_cria> ::= cria <peca> "(" <X> "," <Y> "," <Z>  ")" #13 #14
<cria_extra> ;
<cria_extra> ::= <extra> | î ;
<extra> ::= <comando_cor> #14 <cria_extra> | <comando_gira> #14
<cria_extra> | <comando_espelha> #14 <cria_extra> ;
<id> ::= <figura> #15 | <peca> #16 ;
<comando_move> ::= move  "(" <X>  "," <Y>  "," <Z>  ")" #17 ;
<comando_gira> ::= gira "(" <X> ")" <gira_extra> #18 ;
<gira_extra> ::= no ponto "(" <Y> ")" | î ;
<comando_cor> ::= cor "(" <cor> ")" #20 ;
<comando_espelha> ::= espelha #21 ;
<comando_piscar> ::= piscar "(" <X> ")" ;
<comando_repete> ::= repita <X> vezes #23 inicio <bloco> fim #24 ;
<comando_faca> ::= faca identificador #25 #14 ;
<figura> ::= identificador #3 ;
<nome_Metodo> ::= identificador #4 ;
<modelo_id> ::= identificador #5 ;
<nome_do_mundo> ::= identificador #6 ;
<peca> ::= Peca1 #7 |Peca2 #7 |Peca3 #7 |Peca4 #7 |Peca5 #7 |Peca6 #7
|Peca7 #7 ;
<cor> ::= amarelo #8 | azul #8 | azulMarinho #8 | azulPiscina #8 | branco
#8  | cinza #8  | marrom #8  | oliva #8 | prata #8 | preto #8 | rosa #8 |
verde #8 | verdePiscina #8 | verdeLima #8 |vermelho #8 | violeta #8 ;
<X> ::= numero #9 ;
<Y> ::= numero #10 ;
<Z> ::= numero #11 ;
<mundo> ::= mundo  #1 <nome_do_mundo> <bloco_do_mundo> fim"." ;
<bloco_do_mundo> ::= <comando_mundo> <bloco_do_mundo_2> ;
<bloco_do_mundo_2> ::= <bloco_do_mundo> | î ;
<comando_mundo> ::= <comando_cria_como> | <comando_repete_mundo> | #33
<comando_faca_mundo> #14 | <comando_piscar> #22 #14 ;
<comando_cria_como> ::= cria <modelo_id> como <figura> #28 #14 "(" <X>
"," <Y>  "," <Z>  ")" #29 #14 ;
<comando_repete_mundo> ::=  repita <X> vezes #23 inicio <bloco_do_mundo>
fim #24 ;
<comando_faca_mundo> ::=  faca <modelo_id> "." <metodo_do_id> ;
<metodo_do_id> ::= <comando_viva> #30 | TERMINA #31 | APAGA #32 |
<nome_Metodo> <em_paralelo> #34 ;
<comando_viva> ::= VIVA "(" <nome_Metodo> ")" ;
<em_paralelo> ::= em paralelo #35;
```

Quadro 11 – Gramática

Os operadores não terminais estão entre < e >. A expressão de cada não terminal é definida após os caracteres ::=. No ambiente GALS o caractere î, um terminal, representa a palavra vazia (símbolo *epsilon*).

O significado de cada ação semântica, representada pelo caractere "#" seguida de um número, é descrita no Apêndice A.

3.2.2 Diagramas de casos de uso

Nesta seção são apresentados os diagramas de casos de uso da tela inicial, do editor de modelos e do editor de mundos.

3.2.2.1 Diagrama de casos de uso da tela inicial

O diagrama da figura 10 representa os casos de uso da tela inicial (figura 11).

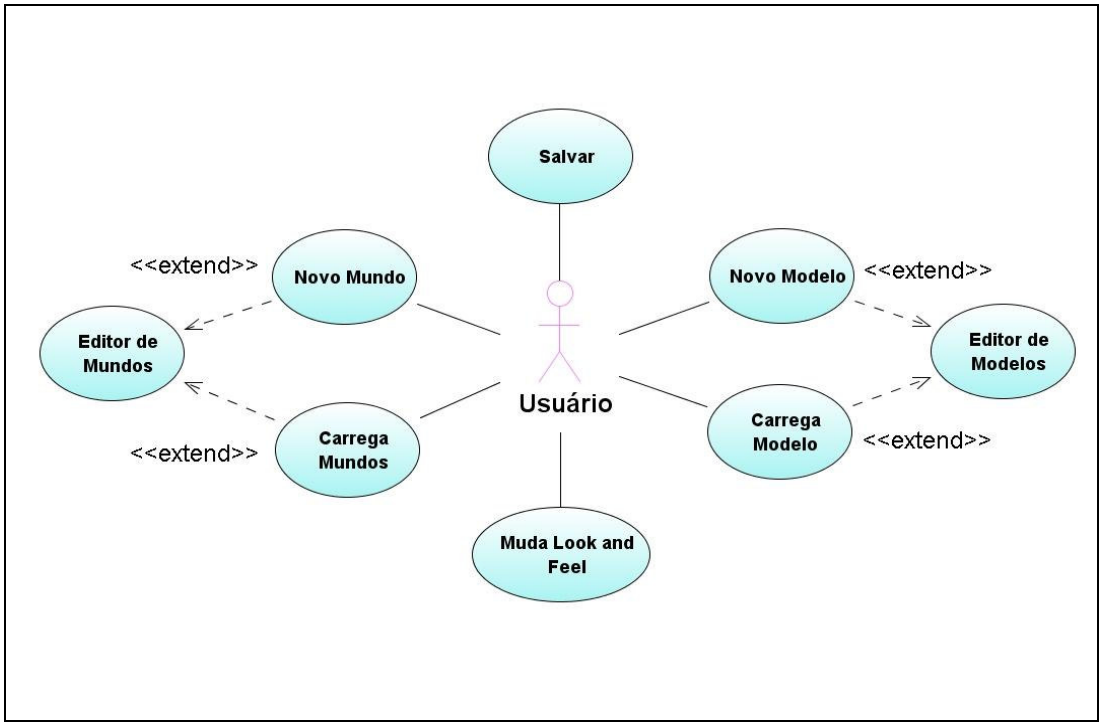


Figura 10 – Casos de uso da tela inicial

A seguir são detalhados os casos de uso apresentados na Figura 10, os quais são:

a) `Salvar`: salva o modelo ou mundo se houver aberto;

b) `Novo Modelo`: cria um novo modelo informando o nome. Se existir um modelo ou mundo aberto, pergunta se deseja salvar o que está aberto e o fecha. Depois de

criado, abre o editor de modelos;

c) `Carrega Modelo`: carrega um modelo já salvo. Se existir um modelo ou mundo aberto pergunta se deseja salvar o que está aberto e o fecha. Depois de criado, abre o editor de modelos;

d) `Novo Mundo`: cria um novo mundo informando o nome. Se existir um modelo ou mundo aberto, pergunta se deseja salvar o que está aberto e o fecha. Depois de criado, abre o editor de mundos;

e) `Carrega Mundos`: carrega um mundo já salvo. Se existir um modelo ou mundo aberto pergunta se deseja salvar o que está aberto e o fecha. Depois de criado, abre o editor de mundos;

f) `Editor de Modelos`: abre duas janelas, uma para o editor textual e outra para o editor gráfico de modelos;

g) `Editor de Mundos`: abre duas janelas, uma para o editor textual e outra para o editor gráfico de mundos;

h) `Muda` *`Look and Feel`*: muda a aparência das janelas. Estão disponíveis os visuais *Metal*, *Windows*, *Motif* e *Windows Classic*.

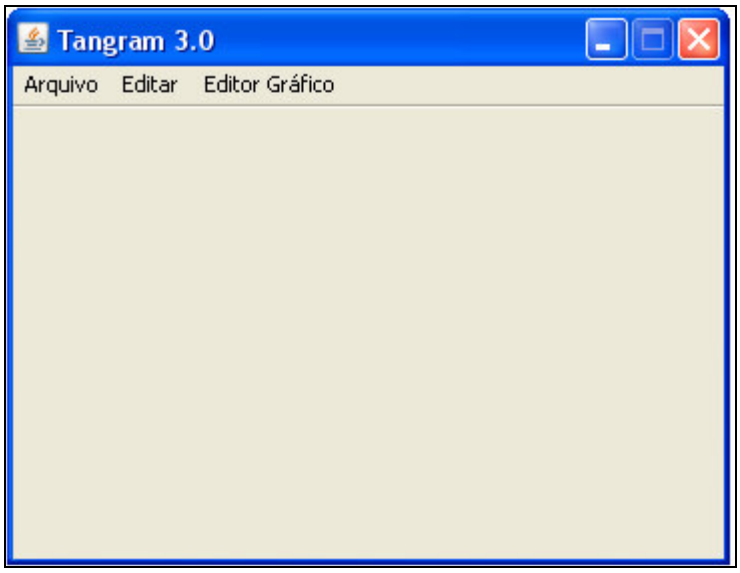


Figura 11 – Tela inicial

3.2.2.2 Diagrama de casos de uso do editor de modelos

O diagrama da Figura 12 representa os casos de uso do editor de modelos.

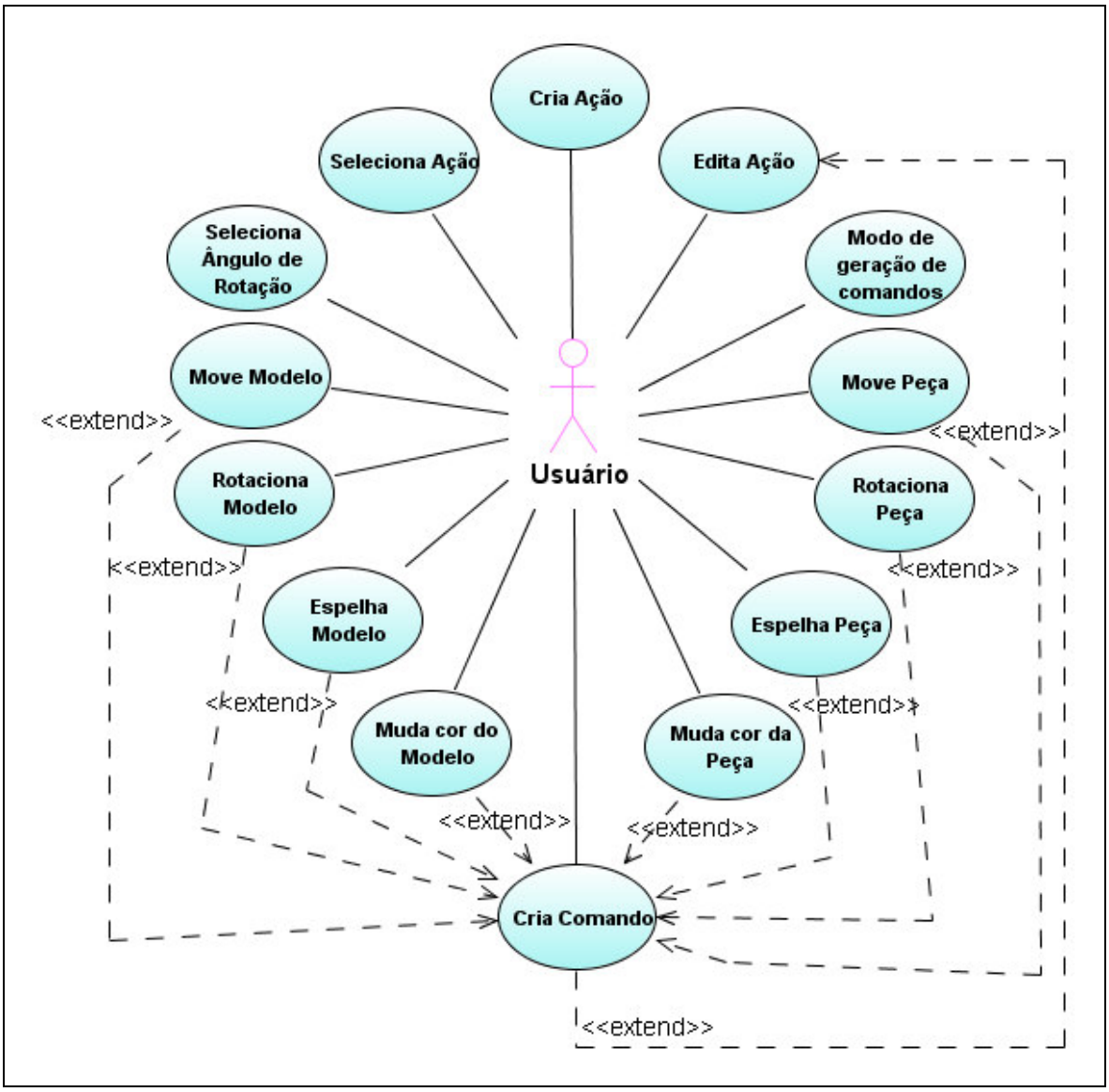


Figura 12 – Casos de uso do editor de modelos

A seguir são detalhados os casos de uso apresentados na Figura 12, os quais são:

a) `Cria Ação`: cria uma nova ação[3] para o modelo, informando o nome;

b) `Edita Ação`: insere, altera, ou exclui comandos de uma ação, através do editor de texto;

c) `Seleciona Modo de geração de comandos`: configura o modo da geração de código a partir da interação do usuário com as peças do Tangram no editor gráfico. Os modos para geração de código são: não gerar código, simples e iterativo. No modo simples uma interação cria apenas um determinado comando, e no modo iterativo é disponibilizada uma tela (Figura 13) para gerar uma iteração com comandos. Através de parâmetros informados na tela (Figura 13), comandos são

[3] Para os usuários do sistema, ação é sinônimo de método.

criados;

d) `Seleciona Ação`: permite selecionar uma das ações, que o modelo possui, para ser editada;

e) `Seleciona Ângulo de Rotação`: permite ao usuário selecionar o ângulo que a peça irá rotacionar nos casos de uso `Rotaciona Peça` e `Rotaciona Modelo`;

f) `Move Peça`: move uma peça no editor gráfico;

g) `Rotaciona Peça`: rotaciona uma peça no editor gráfico;

h) `Espelha Peça`: espelha uma peça no editor gráfico;

i) `Muda a Cor da Peça`: muda a cor de uma peça no editor gráfico para a cor selecionada;

j) `Move Modelo`: move todas as peças de uma única vez no editor gráfico;

k) `Rotaciona Modelo`: rotaciona o modelo no editor gráfico;

l) `Espelha Modelo`: espelha o modelo no editor gráfico;

m) `Muda a Cor do Modelo`: muda a cor do modelo no editor gráfico para a cor selecionada;

n) `Cria Comando`: cria um comando, em forma de texto e insere no editor de texto.

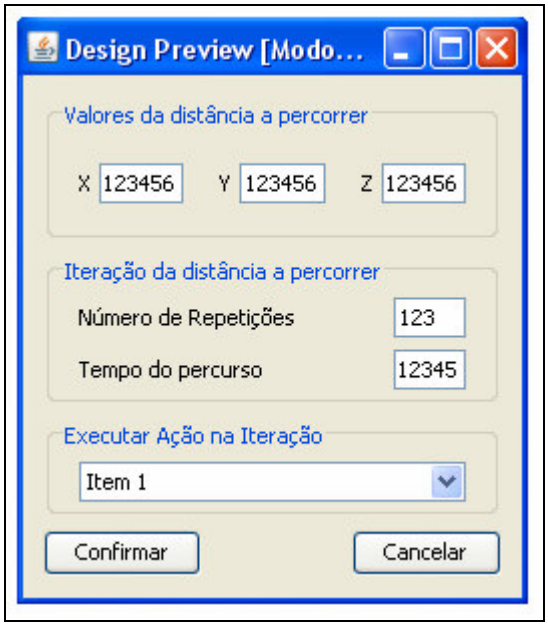


Figura 13 – Tela do modo dinâmico

A tela apresentada na figura 13 é chamada quando há uma interação no modo iterativo. O usuário pode alterar os valores que a peça ou modelo irá se deslocar, escolher em quantas repetições este deslocamento ocorrerá e informar o tempo, em milissegundos, que deve durar até a última repetição. Como exemplo, clicando sobre uma peça, arrastando-a para uma nova coordenada e soltando o botão do *mouse*, é mostrada a tela da figura 13 com os valores de `X`,

`Y`, e `Z` correspondentes ao deslocamento da peça. Se o deslocamento da peça tiver o valor `300` para x e informando o valor `10` para o `número de repetições` e o valor `500` para o `tempo do percurso`, tem-se como resultado um comando que repetirá dez (10) vezes os comandos para mover (deslocar) a peça no valor de `30` para `X` e esperando cinqüenta (50) milissegundo em cada iteração. Ainda há a possibilidade de chamar a execução de uma ação durante cada iteração. Os comandos gerados pelo exemplo descrito são apresentados no quadro 12.

```
repita 10 vezes inicio
    p1.move(30,0,0)
    pisca(50)
fim
```

Quadro 12 – Código gerado pelo modo dinâmico

3.2.2.3 Diagrama de casos de uso do editor de mundos

O diagrama da Figura 14 representa os casos de uso do editor de mundos.

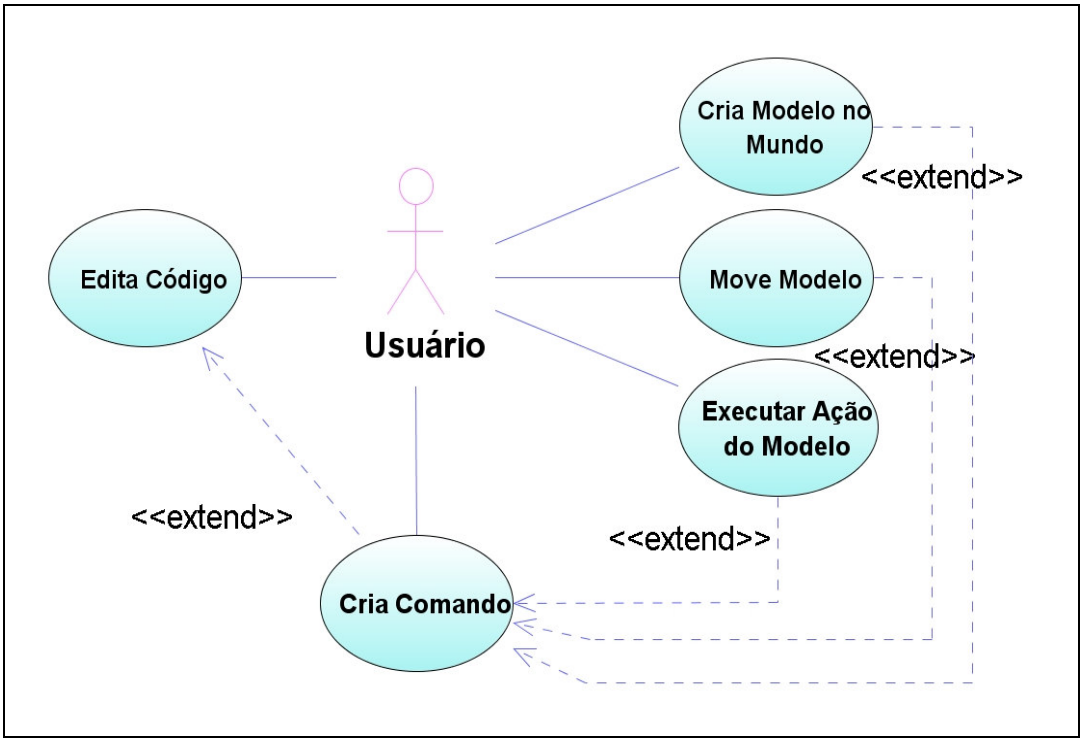


Figura 14 – Casos de uso do editor de mundos

Os casos de uso apresentados na Figura 14 são:

a) `Cria Modelo no Mundo`: insere um modelo, que já foi criado no editor de modelos, no mundo, dando um nome para ele;

b) `Move Modelo`: move um modelo pelo editor gráfico do mundo;

c) `Executar Ação do Modelo`: executa uma ação do modelo;

d) `Cria Comando`: cria um comando, em forma de texto, e insere no editor de texto do mundo;
e) `Edita Código`: insere, altera, ou exclui comandos do mundo, através do editor de texto.

3.2.3 Diagrama de classes

Nas seções seguintes são descritas as funcionalidades das classes criadas. As funcionalidades, atributos e métodos das classes desenvolvidas são apresentadas nos respectivos quadros, os quais são baseados no padrão Javadoc (SUN, 2008) para documentá-las.

3.2.3.1 Diagrama de classes do pacote `JOGL`

Nesta seção são apresentadas as seguintes classes: `Vetor3f`, `Triangulo`, `Plano`, `Linha`, `Color4f`, `Camera` e `BasicModel`. As interfaces `DrawModel`, `ColorModel` e `Câmera` também são apresentadas.

O diagrama das classes do pacote `JOGL` é apresentado na figura 15. Nos quadros 13 a 15 são apresentados respectivamente os Javadoc das interfaces `Camera`, `ColorModel` e `DrawModel`. Nos quadros 16 a 22 são apresentados respectivamente os Javadoc das classes `BasicModel`, `Camera`, `Color4f`, `Linha`, `Plano`, `Triangulo`, `Vetor3f`.

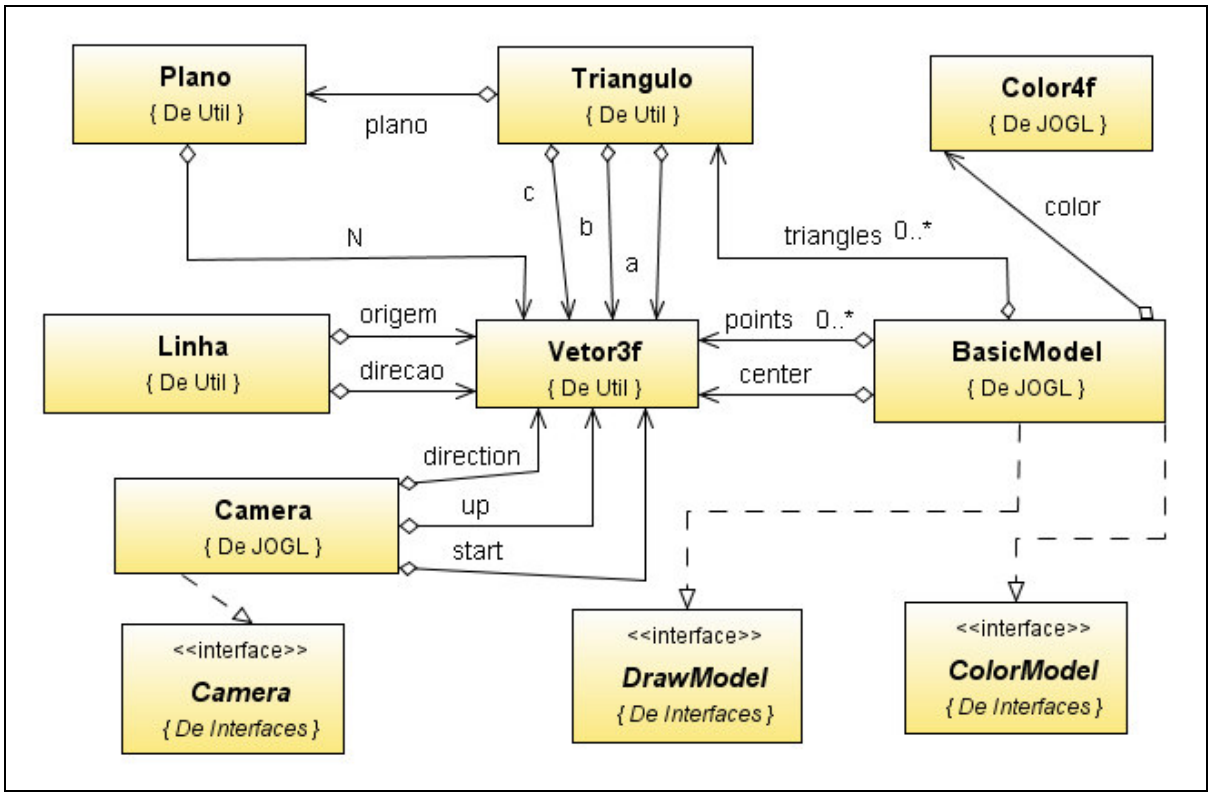


Figura 15 – Diagrama de classes do pacote `JOGL`

JOGL.Intefaces

Interface Camera

All Known Implementing Classes:

Camera

A interface Camera disponibiliza métodos para manipulação da câmera em mundo tridimensional.

Field Summary	
`static int`	**`lookAt`** Tipo da camera: lookAt.
`static int`	**`translate`** Tipo da camera: translate.

Method Summary	
`void`	**`action`**`(javax.media.opengl.GL gl)` Interage com a classe GL alterando a posição da câmera.
`void`	**`setCameraType`**`(int cameraType)` Atribui o tipo de implementação da câmera.
`void`	**`setDirection`**`(Vetor3f direction)` Atribui o ponto Direction da câmera.
`void`	**`setStart`**`(Vetor3f start)` Atribui o ponto Start da câmera.
`void`	**`setUp`**`(Vetor3f up)` Atribui o ponto Up da câmera.

Quadro 13 – Interface `JOGL.Interface.Câmera`

JOGL.Intefaces

Interface ColorModel

All Known Implementing Classes:
BasicModel, Figura, Peca

A interface ColorModel disponibiliza métodos para manipulação das cores de um modelo.
See Also:
`Color4f`

Method Summary	
`Color4f`	**`getColor`**`()` Retorna a cor do modelo.
`void`	**`setColor`**`(Color4f color)` Atribui a cor para o modelo.

Quadro 14 – Interface `JOGL.Interfaces.ColorModel`

JOGL.Intefaces

Interface DrawModel

All Known Implementing Classes:
BasicModel, Figura, Peca

A interface DrawModel disponibiliza métodos para o desenho de modelos.
See Also:
`GL`

Method Summary	
`void`	**`draw`**`(javax.media.opengl.GL gl)` Interage com a classe GL desenhado o modelo.
`boolean`	**`isVisible`**`()` Retorna se o modelo está visível ou não.
`void`	**`setVisible`**`(boolean visible)` Atribui se o modelo está visível ou não.

Quadro 15 – Interface `JOGL.Interfaces.DrawModel`

JOGL

Class BasicModel

All Implemented Interfaces:
java.lang.Cloneable, ColorModel, DrawModel
Direct Known Subclasses:
Peca

Esta classe representa uma forma geométrica em um ambiente tridimencional que possa ser desenhada pelo classe GL. Esta classe armazena vários pontos para o desenho de GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_QUADS e GL_QUAD_STRIP. Armazena também o ponto central do modelo e os triângulos formados pelos pontos.

Field Summary	
`protected Vetor3f`	**`center`** Armazena o centro do modelo.
`protected Color4f`	**`color`** Armazena a cor do modelo.
`protected int`	**`drawType`** Armazena o tipo de desenho, que pode ser entre GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_QUADS e GL_QUAD_STRIP.

protected Vetor3f[]	**points** Armazena os pontos do modelo.
protected java.util.ArrayList<Triangulo>	**triangles** Armazena os triângulos formados pelos pontos.
protected boolean	**visible** Armazena se a forma geométrica está visível.

Constructor Summary

BasicModel(Vetor3f center, Vetor3f[] points, int drawType)

Method Summary

protected java.lang.Object	**clone**() Cria uma nova instância de BasicModel clonando os atributos center, color, points e triangles.
void	**draw**(javax.media.opengl.GL gl) Interage com a classe GL desenhado o modelo.
void	**formaTriangulos**() Cria JOGL.Util.Triangulo a partir dos atributos de um modelo.
private void	**formaTriangulosDeQuadrados**() Cria JOGL.Util.Triangulo quando o drawType do modelo for GL_QUADS ou GL_QUAD_STRIP.
private void	**formaTriangulosDeTriangles**() Cria JOGL.Util.Triangulo quando o drawType do modelo for GL_TRIANGLES ou GL_TRIANGLE_STRIP.
Vetor3f	**getCenter**() Retorna o centro do modelo.
Color4f	**getColor**() Retorna a cor do modelo.
int	**getDrawType**() Retorna o valor do tipo de desenho utilizado pelo método glBegin da classe GL.
Vetor3f	**getPoint**(java.lang.String name) Retorna um ponto pelo nome do ponto.
boolean	**isVisible**() Retorna se o modelo está visível ou não.
float	**maxX**() Retorna o maior valor de x entre os pontos do modelo.
float	**maxY**() Retorna o maior valor de y entre os pontos do modelo.
float	**minX**() Retorna o menor valor de x entre os pontos do modelo.
float	**minY**() Retorna o menor valor de y entre os pontos do modelo.
protected void	**setCenter**(Vetor3f center) Atribui o ponto central da modelo.
void	**setColor**(Color4f color) Atribui a cor para o modelo.
void	**setDrawType**(int type) Define o tipo de desenho do modelo usado pelo método glBegin da classe GL.
protected void	**setPoints**(Vetor3f[] points) Atribui os pontos do modelo.
void	**setVisible**(boolean visible) atribui se o modelo está visível ou não.

Quadro 16 – Classe JOGL.BasicModel

JOGL

Class Camera

All Implemented Interfaces:

Camera

Esta classe implementa o posicionamento da câmera no espaço tridimensional. Estão disponíveis duas formas para controlar a câmera, as quais são: glTranslatef e gluLookAt.

See Also:

GL, GLU

Field Summary

(package private) int	**cameraType** Armazena o tipo de controle da câmera.
Vetor3f	**direction** Ponto para controlar a posição da câmera, utilizado como o ponto da direção quando o tipo da câmera for gluLookAt.
Vetor3f	**start** Ponto para controlar a posição da câmera, utilizado como o ponto do olho quando o tipo da câmera for gluLookAt, e como ponto para o qual a câmera deve se movimentar quando o tipo da câmera for glTranslatef.
Vetor3f	**up** Ponto para controlar a posição da câmera, , utilizado como o ponto da orientação do olho quando o tipo da câmera for gluLookAt.

Constructor Summary

Camera(Vetor3f start) Cria Camera passando o ponto para controle da câmera através do método glTranslatef.
Camera(Vetor3f start, Vetor3f direction, Vetor3f up) Cria Camera passando os três pontos necessário para o controle da câmera através do método gluLookAt.

Method Summary

void	**action**(javax.media.opengl.GL gl) Interage com a classe GL alterando a posição da câmera.
void	**setCameraType**(int cameraType) Atribui o tipo de implementação da câmera.
void	**setDirection**(Vetor3f direction) Atribui o ponto Direction da câmera.
void	**setStart**(Vetor3f start) Atribui o ponto Start da câmera.
void	**setUp**(Vetor3f up) Atribui o ponto Up da câmera.

Quadro 17 – Classe JOGL.Camera

JOGL

Class Color4f

All Implemented Interfaces:
java.lang.Cloneable

Esta classe armazena a cor de um modelo para o desenho na classe GL. A classe armazena a quantidade de vermelho (R), verde (G), azu l(A) de uma cor mais o atributo alpha através de valores reais de zero (0) a um (1).

Field Summary

(package private) float	**alpha** Valor real que representam a transparência da cor.
(package private) float	**B** Valor real que representam o azul da cor.
(package private) float	**G** Valor real que representam o verde da cor.
(package private) java.lang.String	**name** Nome da cor.
(package private) float	**R** Valor real que representam o vermelho da cor.

Constructor Summary

Color4f(float R, float G, float B, float alpha) Cria uma nova cor passando os valores de R, G, B e alpha.
Color4f(float R, float G, float B, float alpha, java.lang.String name) Cria uma nova cor passando os valores de R, G, B, alpha e o nome da cor.

Method Summary

protected java.lang.Object	**clone**() Clona o objeto desta classe.
boolean	**equals**(java.lang.Object obj) Compara se os atributos R,G,B e alpha são iguais através da função hashCode.
float	**getAlpha**() Retorna o valor de alpha.
float	**getB**() Retorna o valor de B.
float	**getG**() Retorna o valor de G.
java.lang.String	**getName**() Retorna o nome da cor.
float	**getR**() Retorna o valor de R.
int	**hashCode**() Retorna um valor inteiro representando o hash code dos atributos R, G, B e alpha.
Color4f	**negativo**() Retorna uma nova cor com o valor dos atributos R, G, B invertidos conforme a equação: novaValor = 1 - valor.
void	**setAlpha**(float alfa) Atribui o valor de alpha.
void	**setB**(float B) Atribui o valor de B.
void	**setG**(float G) Atribui o valor de G.
void	**setName**(java.lang.String name)

	Atribui o nome da cor.
void	**setR**(float R) Atribui o valor de R.
java.lang.String	**toString**() Retorna uma string com os valores dos atributos da cor.
private void	**valida**() Verifica se os valores dos atributos da cor são maiores que um (1).

Quadro 18 – Classe JOGL.Color4f

JOGL.Util

Class Linha

Usado para representar um segmento de reta ou reta. Baseado em Lengyel (2004, cap. 4.1).

See Also:
Vetor3f

Field Summary

private Vetor3f	**direcao** Outro ponto do segmento de reta ou o ponto direção de uma reta.
private Vetor3f	**origem** Um ponto do segmento de reta ou o ponto de origem de uma reta.

Constructor Summary

Linha(Vetor3f origem, Vetor3f direcao) Cria um segmento de reta, ou reta, com seus dois pontos.

Method Summary

Vetor3f	**getDirecao**() Retorna o ponto direção.
Vetor3f	**getIntersection**(Linha l) Este método calcula o ponto de intersecção entre dois segmentos de reta..
Vetor3f	**getOrigem**() Retorna o ponto origem.
void	**setDirecao**(Vetor3f direcao) Atribui um ponto ao Vetor3f direção da classe.
void	**setOrigem**(Vetor3f origem) Atribui um ponto ao Vetor3f origem da classe.
java.lang.String	**toString**() Retorna a String dos dois vetores da classe.

Quadro 19 – Classe JOGL.Util.Linha

JOGL.Util

Class Plano

Plano de um triangulo. Usado para calcular a cruzamento de uma Linha com um plano. Baseado Lengyel (2004, cap. 4.1, 4.2.1, 5.2 e 5.2.1).

See Also:

Linha, Vetor3f, Triangulo

Field Summary

private float	**D** Valor real que representa a *Distância* de um plano.
private Vetor3f	**N** Vetor3f que representa a *Vetor unitário* de um plano.

Constructor Summary

Plano(Triangulo t) Cria um plano a partir dos pontos de um triangulo (LENGYEL, 2004, pg. 144).
Plano(Vetor3f N, float D) Cria um plano recebendo a *Vetor unitário* e *Distância* de um plano.

Method Summary

Vetor3f	**getIntersection**(Linha ray) Retorna o ponto de intersecção entre uma reta e o plano (LENGYEL, 2004, pg. 107 e 143).

Quadro 20 – Classe JOGL.Util.Plano

JOGL.Util

Class Triangulo

Esta classe representa um triângulo e seus pontos são representados pela classe Vetor3f .

See Also:

Vetor3f, Plano

Field Summary

private Vetor3f	**a** Vetor3f que representam os pontos do triângulo.
private Vetor3f	**b** Vetor3f que representam os pontos do triângulo.
private Vetor3f	**c** Vetor3f que representam os pontos do triângulo.
private Plano	**plano** Plano em que está contido o triângulo.

Constructor Summary

Triangulo(Vetor3f a, Vetor3f b, Vetor3f c) Cria um Triângulo a partir dos seus pontos.

Method Summary

Vetor3f	**getA**() Retorna o ponto A do triângulo.
Vetor3f	**getB**() Retorna o ponto B do triângulo.
Vetor3f	**getBarycentricCoordinates**(Vetor3f p) Calcula as coordenadas baricêntricas.
Vetor3f	**getC**() Retorna o ponto C do triângulo.
Vetor3f	**getIntersectionPoint**(Linha ray) Detecta a intersecção de um reta com o triângulo.
Plano	**getPlano**() Retorna o plano que esta contido o triângulo.
java.lang.String	**toString**() Retorna uma string com os dados dos pontos do triângulo.

Quadro 21 – Classe JOGL.Util.Triangulo

JOGL.Util

Class Vetor3f

All Implemented Interfaces:

java.lang.Cloneable

Esta classe contém as coordenadas tridimensionais de um ponto e várias operações matemáticas baseadas em Lengyel (2004, cap. 1)

Field Summary	
(package private) java.lang.String	**name** Nome do ponto.
(package private) float	**x** Valor das coordenadas 3D de um ponto.
(package private) float	**y** Valor das coordenadas 3D de um ponto.
(package private) float	**z** Valor das coordenadas 3D de um ponto.

Constructor Summary
Vetor3f() Construtor.
Vetor3f(float x, float y, float z) Construtor.

Method Summary	
Vetor3f	**add**(Vetor3f v) Retorna o vetor resultante da soma de um vetor a esse.
Void	**addOnThis**(Vetor3f v) Adiciona o valor de um vetor a esse.
java.lang.Object	**clone**() Retorna um clone do objeto desta classe.
Vetor3f	**div**(float value) Retorna o vetor resultante da divisão de um número por este vetor.
Void	**divOnThis**(float value) Divide um número por este vetor.
Vetor3f	**dot**(float value) Retorna o vetor resultante da multiplicação de um número por este vetor.
Void	**dotOnThis**(float value) Multiplica um número a este vetor.
Boolean	**equals**(java.lang.Object obj) Compara o objeto desta classe com outro objeto.
Float	**getLength**() Calcula o comprimento de um vetor (LENGYEL, 2004, pg. 13-14).
java.lang.String	**getName**() Retorna o nome do ponto.
float	**getScalarProduct**(Vetor3f v) Calcula o produto escalar entre dois vetores (LENGYEL, 2004, pg. 15).
float	**getX**() Retorna o valor de x.
float	**getY**() Retorna o valor de y.
float	**getZ**() Retorna o valor de z.

int	**hashCode**() Faz um código *hash* para os valores x, y e z da classe.
void	**mirrorXOnThis**(float xc) Faz o espelhamento do ponto na eixo x a partir do ponto central.
Vetor3f	**normalization**() Calcula a normal deste vetor.
void	**rotateXYOnThis**(double angle, Vetor3f center) Rotaciona este ponto a partir de um ponto de origem.
void	**setName**(java.lang.String name) Atribui o nome do ponto.
Vetor3f	**sub**(Vetor3f v) Retorna o vetor resultante da subtração de um vetor a esse.
void	**subOnThis**(Vetor3f v) Subtrai o valor de um vetor a esse.
Vetor3f	**sum**(float value) Retorna o vetor resultante da soma de numero por este vetor.
void	**sumOnThis**(float value) Soma um número a este vetor.
java.lang.String	**toString**() Transforma os valores da classe em um objeto de String.
Vetor3f	**vectorProduct**(Vetor3f v) Calcula o Produto entre dois vetores (LENGYEL, 2004, pg. 20).

Quadro 22 – Classe `JOGL.Util.Vetor3f`

3.2.3.2 Diagrama de classes para `Figura`

Nesta seção são apresentadas as classes para desenhar uma figura (sete peças) do Tangram, as quais são: `Apontamento`, `Peca` e `Figura`. As interfaces `RotateModel`, `TranslateModel`, `MirrorModel` e `RayTrancing` também são apresentadas.

Na figura 16 está definido o diagrama de classe para a classe `Apontamento`, `Peca` e `Figura`. Compõem este diagrama as classes `Apontamento`, `BasicModel`, `Figura`, `Peca`, `Linha` e `Vetor3f`. As Interfaces `ColorModel`, `DrawModel`, `MirroModel`, `RayTracing`, `RotateModel` e `TranslateModel` também compõem o diagrama.

A Classe `Peca` representa uma peça (Tan) do Tangram. Esta classe implementa as interfaces `MirrorModel`, `RotateModel` e `TranlateModel` que disponibilizam métodos para espelhar, rotacionar e mover uma peça, além de herdar os métodos da classe `BasicModel`. A interface `RayTracing`, também implementada pela classe `Peca`, disponibiliza métodos para detectar a intersecção entre uma reta e uma peça, retornando um `Object` como resultado. Na implementação da classe `Peca` os métodos retornam um objeto da classe `Apontamento`.

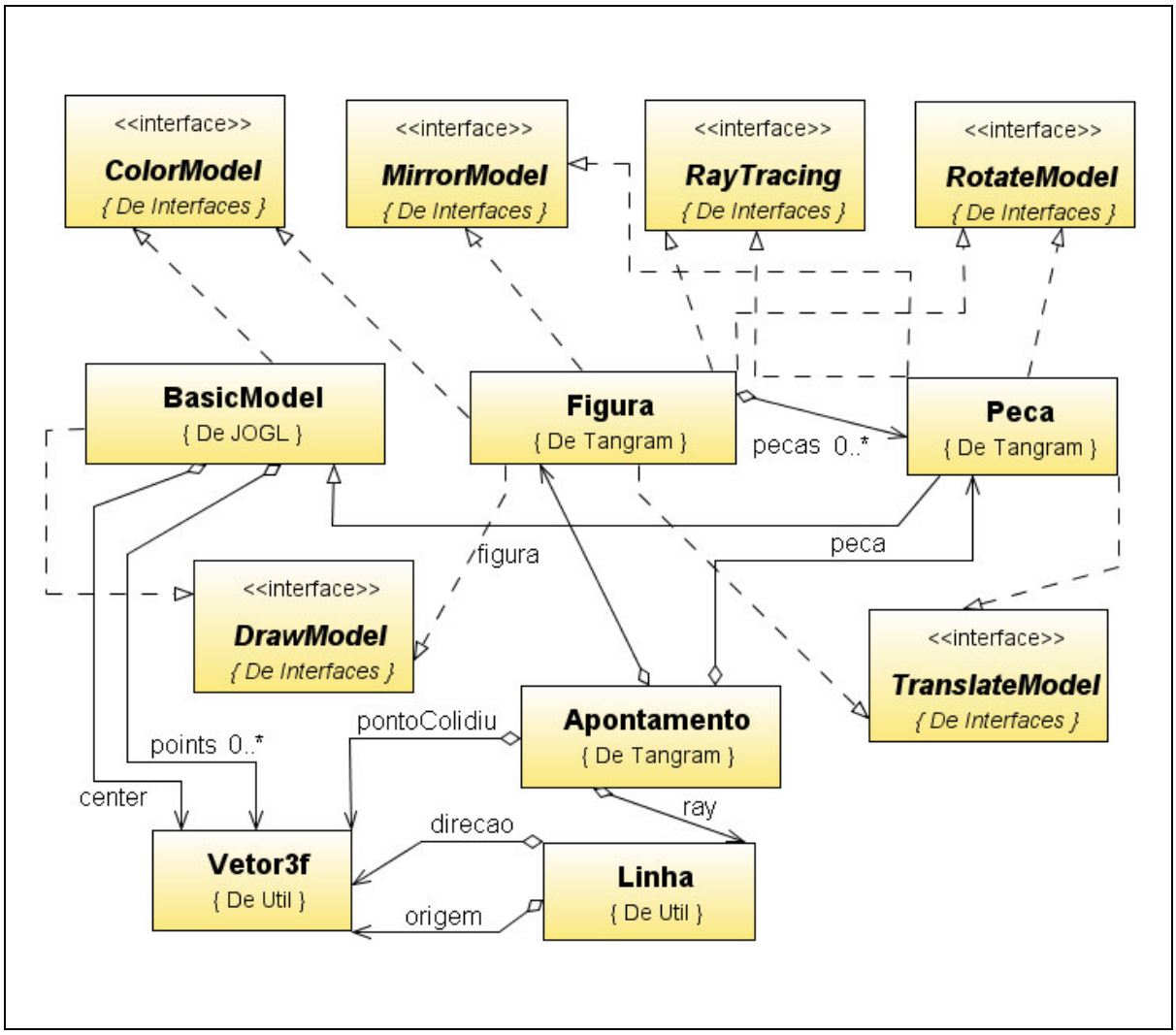


Figura 16 – Diagrama de classe para `Apontamento`, `Peca` e `Figura`

A classe `Figura`, a qual representa uma figura (sete peças) do Tangram, implementa as mesmas interfaces das classes `Peca` e `BasicModel`. Desta forma é possível mover, girar, espelhar e mudar a cor de uma peça (Tan) ou da figura (sete peças) do Tangram.

A documentação das interfaces `RotateModel`, `TranslateModel`, `MirrorModel` e `RayTrancing` são encontradas respectivamente nos quadros 23 a 26 e as classes `Apontamento`, `Peca` e `Figura` estão documentadas nos quadros 27 a 29 respectivamente.

JOGL.Intefaces

Interface RotateModel

All Known Implementing Classes:
Figura, Peca

A interface RotateModel disponibiliza métodos para rotação de modelos.

Method Summary	
void	**Rotate**(float angulo) Rotaciona o modelo em sentido anti-horário em torno de seu centro.
void	**Rotate**(float angulo, Vetor3f origem) Rotaciona o modelo em sentido anti-horário em torno de um ponto.

Quadro 23 – Interface `Tangram.RotateModel`

JOGL.Intefaces

Interface TranslateModel

All Known Implementing Classes:
Figura, Peca

A interface TranslateModel disponibiliza métodos para translação de um modelo .
See Also:
Vetor3f

Method Summary	
void	**translate**(Vetor3f v) Realiza a translação do modelo adicionando um valor a posição em que o modelo se encontra.
void	**translateTo**(Vetor3f v) Realiza a translação do modelo para um ponto específico.

Quadro 24 – Interface `Tangram.TranslateModel`

JOGL.Intefaces

Interface MirrorModel

All Known Implementing Classes:
Figura, Peca

A interface MirrorModel disponibiliza métodos para espelhar um modelo.

Method Summary	
void	**mirror**() Espelha o modelo, no eixo X, a partir do centro dele.
void	**mirror**(Vetor3f v) Espelha o modelo, no eixo X, a partir do ponto representado pelo Vetor3f v.

Quadro 25 – Interface `Tangram.MirrorModel`

JOGL.Intefaces

Interface RayTracing

All Known Implementing Classes:
Figura, Peca

A interface RayTracing disponibiliza um método para detectar a intersecção entre uma linha e um triângulo. Disponibiliza o método pontoProximo() o qual detecta o ponto mais próximo a uma linha.

See Also:
`Linha`, `FormaTriangulos`

Method Summary	
`java.lang.Object`	**`interseccao`**`(Linha ray)` Detecta a intersecção entre uma reta representada pela classe Linha e a classe que implementa esta interface.
`java.lang.Object`	**`pontoProximo`**`(Linha ray)` Detecta se um ponto da classe que implementa este método está próximo de uma reta.

Quadro 26 – Interface `Tangram.RayTrancing`

Tangram

Class Apontamento

Esta Classe guarda informações sobre a intersecção de um segmento de reta com um objeto da classe Figura.

Field Summary	
(package private) Figura	**figura** Figura em que a linha colidiu.
(package private) Peca	**peca** Peca em que a linha colidiu.
(package private) Plano	**plano** Plano em que a figura esta contida.
(package private) Vetor3f	**pontoColidiu** Vetor3f que representa o ponto em que a linha colidiu.
(package private) Linha	**ray** Linha que gerou a colisão.

Constructor Summary
Colisao(Peca pecaColidiu, Vetor3f pontoColidiu, Linha ray, Plano p) Cria nova instância de Colisao passando a peca, o ponto, a linha e o plano em que ouve a colisão.

Method Summary	
Figura	**getFigura**() Retorna a Figura em que houve a colisão.
Peca	**GetPeca**() Retorna a Peca em que houve a colisão.
Plano	**GetPlano**() Retorna o Plano em que houve a colisão.
Vetor3f	**getPontoColidiu**() Retorna o Vetor3f em que houve a colisão.
Linha	**getRay**() Retorna a Linha que gerou a colisão.
void	**setFigura**(Figura figura) Atribui a Figura em que houve a colisão.
java.lang.String	**toString**()

Quadro 27 – Classe `Tangram.Apontamento`

Tangram

Class Peca

All Implemented Interfaces:
java.lang.Cloneable, ColorModel, DrawModel, FormaTriangulos, MirrorModel, RayTracing, RotateModel, TranslateModel

---

extends BasicModel
implements TranslateModel, RotateModel, MirrorModel, RayTracing, java.lang.Cloneable
Esta classe implementa uma peça do jogo Tangram. Esta classe implementa métodos para detectar intersecção entre uma linha, espelhá-la, move-la, desenhá-la, rotacioná-la, mudar a cor e cloná-la.

Field Summary

(package private) float	**angulo** Armazena o angulo que a peça foi rotacionada.
(package private) boolean	**mirrored** Armazena se a peça foi espelhada.
(package private) boolean	**moveuPeca** Armazena se a peça foi movida.
(package private) int	**tipo** Armazena o tipo da peça, o qual é o respectivo número do nome das peças no Tangram (P1, P2, P3, P4, P5, P6 e P7).

Fields inherited from class JOGL.BasicModel

center, color, drawType, points, triangles, visible

Constructor Summary

Peca(int tipo, Vetor3f centro) Cria um peça de um modelo (Figura) do Tangram.

Method Summary

void	**AddAngulo**(float adicional) Adiciona o valor do ângulo à variável que armazena o ângulo da peça.
protected java.lang.Object	**Clone**() Super().
Colisao	**interseccao**(Linha ray) Detecta a intersecção entre uma reta representado pelo classe Linha e a esta classe.
float	**getAngulo**() Retorna o ângulo em que a peça foi rotacionada.
int	**getTipo**() Retorna o valor do atributo tipo dessa classe.
boolean	**isMirrored**() Retorna se a peça foi espelhada.
void	**mirror**() Espelha o modelo, no eixo X, a partir do centro dele.
void	**mirror**(Vetor3f v) Espelha o modelo, no eixo X, a partir do ponto representado pelo Vetor3f v.
protected void	**movePeca**(Vetor3f v) Adiciona o Vetor3f V a todos os pontos da peça.
Vetor3f	**pontoProximo**(Linha ray) Detecta se um ponto desta classe está próximo da reta recebida por parâmetro.

void	**Rotate**(float angulo) Rotaciona o modelo em sentido anti-horário em torno do seu centro.
void	**Rotate**(float angulo, Vetor3f origem) Rotaciona o modelo em sentido anti-horário em torno de um ponto.
void	**setColor**(Color4f color) Altera a cor da peça.
java.lang.String	**toString**() retorna um objeto de String do método toString() do Vetor3f que representa o centro da peça.
void	**translate**(Vetor3f v) Realiza a translação do modelo adicionando um valor a posição em que o modelo se encontra.
void	**translateTo**(Vetor3f v) Realiza a translação do modelo para um ponto especifico.

Quadro 28 – Classe Tangram.Peca

Tangram

Class Figura

java.lang.Cloneable, ColorModel, DrawModel, MirrorModel, RayTracing, RotateModel, TranslateModel

---

extends java.lang.Object
implements RayTracing, MirrorModel, TranslateModel, DrawModel, RotateModel, ColorModel, java.lang.Cloneable

Esta classe representa uma figura, composta por sete peças, do jogo Tangram. Esta classe implementa métodos para detectar intersecção entre uma linha, espelhá-la, move-la, desenhá-la, rotacioná-la, mudar a cor e cloná-la.

Field Summary

(package private) java.lang.String	**name** Armazena o nome da figura.
(package private) Peca[]	**pecas** Vetor que contém as sete peças.

Constructor Summary

Figura(java.lang.String name) Cria uma nova figura formando o desenho de um quadrado com as sete peças.
Figura(Vetor3f v1, Vetor3f v2, Vetor3f v3, Vetor3f v4, Vetor3f v5, Vetor3f v6, Vetor3f v7) Cria uma nova figura passando por parâmetro as peças.

Method Summary

void	**changeDrawType**(int type) Muda o tipo de desenho de todas as peças.
protected java.lang.Object	**clone**() Clona esse objeto.
Colisao	**interseccao**(Linha ray) Retorna um objeto de Apontamento se a Linha *ray* intersecta alguma das peças da figura.
void	**draw**(javax.media.opengl.GL gl) Desenha todas as peças que estão visíveis na tela.
Vetor3f	**getCenter**() Retorna o centro da figura pegando os valores máximos e mínimos de X e Y das peças da figura.
Color4f	**getColor**() Este método compara todas as cores das peças da figura.
java.lang.String	**getName**() Retorna o nome da figura.
Peca	**getP1**() Retorna P1.
Peca	**getP2**() Retorna P2.
Peca	**getP3**() Retorna P3.
Peca	**getP4**() Retorna P4.
Peca	**getP5**() Retorna P5.
Peca	**getP6**() Retorna P6.
Peca	**getP7**() Retorna P7.
Peca	**getPeca**(int num)

	Retorna a peça pelo número dela.
Peca[]	**getPecas**() Retorna a lista das peças.
Vetor3f	**getPoint**(java.lang.String name) Retorna o ponto, dentro todas os pontos das peças da figura, que tenha o nome passado por parâmetro.
boolean	**isVisible**() Retorna se o modelo está visível ou não.
void	**mirror**() Espelha os pontos de todas as peças da figura no eixo X, a partir do centro da figura.
void	**mirror**(Vetor3f v) Espelha os pontos de todas as peças da figura no eixo X, a partir de um ponto determinado.
Vetor3f	**pontoProximo**(Linha ray) Retorna dentre todos os pontos das peças da figura o mais próximo da Linha ray.
void	**resetPecas**() Cria um novo objeto de Peca para cada uma das peças da figura posicionadas no centro (ponto 0,0,0).
void	**Rotate**(float angulo) Rotaciona toda a Figura em torno de seu centro.
void	**Rotate**(float angulo, Vetor3f origem) Rotaciona toda a Figura em torno de uma origem especificada.
void	**setColor**(Color4f color) Muda a cor de todas as peças da Figura.
void	**setName**(java.lang.String name) Atribui o nome da figura.
void	**setP1**(Peca p1) Atribui P1.
void	**setP2**(Peca p2) Atribui P2.
void	**setP3**(Peca p3) Atribui P3.
void	**setP4**(Peca p4) Atribui P4.
void	**setP5**(Peca p5) Atribui P5.
void	**setP6**(Peca p6) Atribui P6.
void	**setP7**(Peca p7) Atribui P7.
boolean	**setPeca**(Peca p, int num) Atribui uma peça pelo número dela.
void	**setPecas**(Peca[] pecaList) Atribui a lista das peças.
void	**setVisible**(boolean visible) Atribui se o modelo está visível ou não.
void	**translate**(Vetor3f v) Adiciona o valor do Vetor3f V em todas as peças da figura.
void	**translateTo**(Vetor3f v) Subtrai do Vetor3f V os vetores correspondente ao centro da figura e adiciona o resultado em todas as peças da figura.

Quadro 29 – Classe `Tangram.Figura`

3.2.3.3 Diagrama de classes para a implementação da interface `GLEventListener`

Nesta seção é apresentada a classe `Desenha`. As interfaces `GLEventListener` e `MouseEventHandler` também serão apresentadas.

A interface `GLEventListener` da biblioteca JOGL disponibiliza os métodos `init`, `display`, `reshape` e `displayChanged`, os quais são utilizados para desenhar em um objeto da classe `GLCanvas`.

A interface `MouseEventHandler` é responsável por armazenar, em uma lista, os objetos da classe `Figura`, para realizar nestes objetos modificações originadas pela interação do usuário com a interface gráfica. Esta interface disponibiliza os métodos `action`, o qual trata os eventos do *mouse*, e `extraDraws`, o qual desenha outras informações além dos objetos de `Figura`. As classes que implementam esta interface devem guardar o modo de interação com a interface gráfica e um objeto de `Camera`.

A classe `Desenha` implementa as interfaces `GLEventListener`, `KeyListener`, `MouseEventHandler`, `MouseListener` e `MouseMotionListener`. O construtor da classe `Desenha` recebe como parâmetro um objeto de `MouseEventHandler`, o qual contém os objetos de `DrawModel` para serem desenhados na tela.

No método `display` da classe `Desenha` os eventos do *mouse* são enviados para o objeto da interface `MouseEventHandler` onde serão tratados. Em seguida os objetos de `DrawModel` são desenhados e o método `extraDraws` de `MouseEventHandler` é chamado para ser escrito, no desenho, o nome das peças.

Na figura 17 é apresentado o diagrama de dependência da classe `Desenha`. Compõem este diagrama as classes `Camera` e `Colisao`. As interfaces `DrawModel`, `GLEventListener`, `KeyListener`, `MouseEventHandler`, `MouseListener` e `MouseMotionListener` também fazem parte deste diagrama.

No quadro 30 é apresentada a documentação da classe `Desenha` e no quadro 31 a documentação da interface `MouseEventHandlres`.

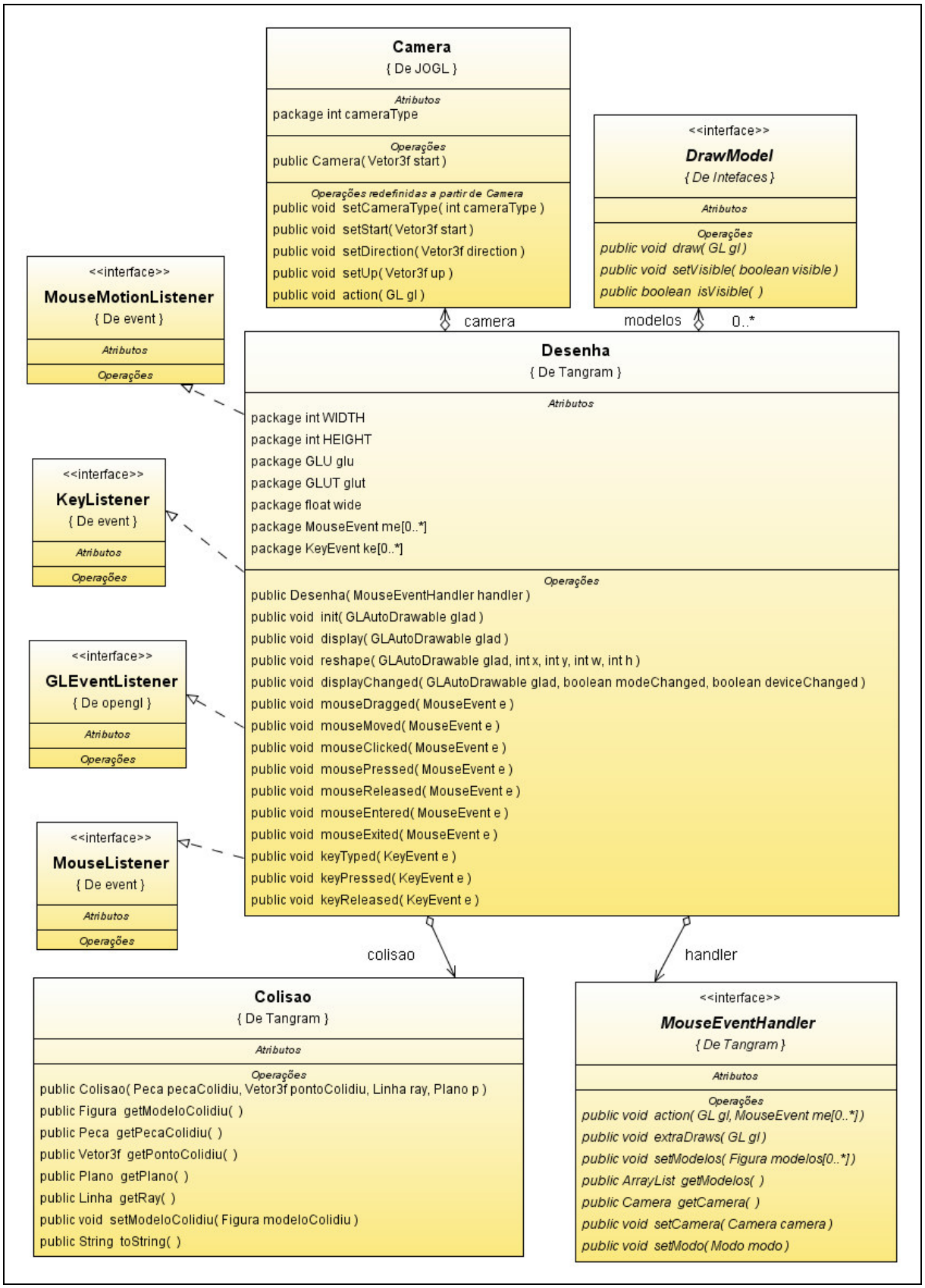


Figura 17 – Diagrama da classe `Desenha`

Tangram

Class Desenha

All Implemented Interfaces:
java.awt.event.KeyListener, java.awt.event.MouseListener, java.awt.event.MouseMotionListener, java.util.EventListener, javax.media.opengl.GLEventListener

---

Implements javax.media.opengl.GLEventListener, java.awt.event.MouseMotionListener, java.awt.event.MouseListener, java.awt.event.KeyListener
Esta classe implementa metodos para o desenho em um GLCanvas, tratamento de eventos do *mouse* e tratamento de eventos do teclado.

Field Summary

(package private) Camera	**camera** Armazena o objeto de Camera.
(package private) javax.media.opengl.glu.GLU	**glu** Armazena o GLU.
(package private) MouseEventHandler	**handler** Armazena o tratador de eventos do *mouse*.
(package private) java.util.ArrayList<java.awt.event.KeyEvent>	**ke** Armazena os eventos do teclado que ainda não foram tratados.
(package private) java.util.ArrayList<java.awt.event.MouseEvent>	**me** Armazena os eventos do *mouse* que ainda não foram tratados.
(package private) java.util.ArrayList<DrawModel>	**modelos** Armazena os modelos a serem desenhados.
(package private) float	**wide** Armazena a largura dividida pela altura da tela.

Constructor Summary

Desenha(MouseEventHandler handler) Cria nova instâcia de Desenha passando o Tratados de Eventos.

Method Summary

void	**display**(javax.media.opengl.GLAutoDrawable glad) Método de javax.media.opengl.GLEventListener.
void	**displayChanged**(javax.media.opengl.GLAutoDrawable glad, boolean modeChanged, boolean deviceChanged) Método de javax.media.opengl.GLEventListener.
void	**init**(javax.media.opengl.GLAutoDrawable glad) Método de javax.media.opengl.GLEventListener.
void	**keyPressed**(java.awt.event.KeyEvent e) Envia o evento para lista de eventos do teclado.
void	**keyReleased**(java.awt.event.KeyEvent e) Envia o evento para lista de eventos do teclado.
void	**keyTyped**(java.awt.event.KeyEvent e) Envia o evento para lista de eventos do teclado.
void	**mouseClicked**(java.awt.event.MouseEvent e) Envia o evento para lista de eventos do *mouse*.
void	**mouseDragged**(java.awt.event.MouseEvent e) Envia o evento para lista de eventos do *mouse*.
void	**mouseEntered**(java.awt.event.MouseEvent e) Envia o evento para lista de eventos do *mouse*.
void	**mouseExited**(java.awt.event.MouseEvent e) Envia o evento para lista de eventos do *mouse*.

void	**mouseMoved**(java.awt.event.MouseEvent e) Envia o evento para lista de eventos do mouse.
void	**mousePressed**(java.awt.event.MouseEvent e) Envia o evento para lista de eventos do mouse.
void	**mouseReleased**(java.awt.event.MouseEvent e) Envia o evento para lista de eventos do mouse.
Void	**reshape**(javax.media.opengl.GLAutoDrawable glad, int x, int y, int w, int h) Método de javax.media.opengl.GLEventListener.

Quadro – 30 Classe `Tangram.Desenha`

Tangram

Interface MouseEventHandler

All Known Implementing Classes:
MEHforModels, MEHforMundo, MEHforMundoExecutavel

Esta interface disponibiliza métodos para o tratamento das interações do usuário com a interface gráfica, a qual é desenhada através de um GLCanvas. A interação do usuário, utilizando o *mouse*, com a interface gráfica gera objetos da classe MouseEvent, os quais são tratados, gerando ações sobre os objetos das Classes Figura e Peca. Esta interface disponibiliza um métodos que atribui ou retorna as figuras que são desenhadas no GLCanvas, além de um método para desenhar outras informações no GLCanvas. Disponibiliza também o objeto da classe Camera.

Nested Class Summary

Static class	**MouseEventHandler.Modo** Enumeração que contém os possíveis modos de tratamento de eventos do mouse.

Method Summary

void	**action**(javax.media.opengl.GL gl, java.util.ArrayList<java.awt.event.MouseEvent> me) Trata os eventos do mouse gerados pelo GLCanvas.
void	**extraDraws**(javax.media.opengl.GL gl) Desenha informações extras na tela.
Camera	**getCamera**() Retorna a câmera.
java.util.ArrayList	**getModelos**() Retorna os modelos que devem ser desenhados na tela.
void	**setCamera**(Camera camera) Atribui a câmera.
void	**setModelos**(java.util.ArrayList<Figura> modelos) Atribui os modelos que devem ser desenhados na tela.
void	**setModo**(MouseEventHandler.Modo modo) Atribui o modo de tratamento dos eventos do *mouse*.

Quadro – 31 Interface `Tangram.MouseEventHandler`

3.2.3.4 Diagrama de classe do pacote `Comandos`

Nesta seção são apresentadas as classes: `ComandoCall`; `ComandoCallMundo`; `ComandoCor`; `ComandoCriaPeca`; `ComandoEspelha`; `ComandoEspera`; `ComandoGira`; `ComandoLaco`; `ComandoMove`; `ComandoMovePara`; `ComandoMoveParaMundo`; `ComandoViva`;

`ComandoVivaMundo`; `ModeloExecutavel` e `MundoExecutavel`. As interfaces `Comando` e `Executor` também são apresentadas.

Na figura 18 é apresentado o diagrama das classes que implementam a interface `Comando`. As classes que implementam esta interface representam um comando da linguagem do LTD e através do método `faca(Executor)`, disponibilizado pela interface `Comando`, os comandos de um programa do LTD são executados.

No quadro 32 é documentada a interface `Comando`.

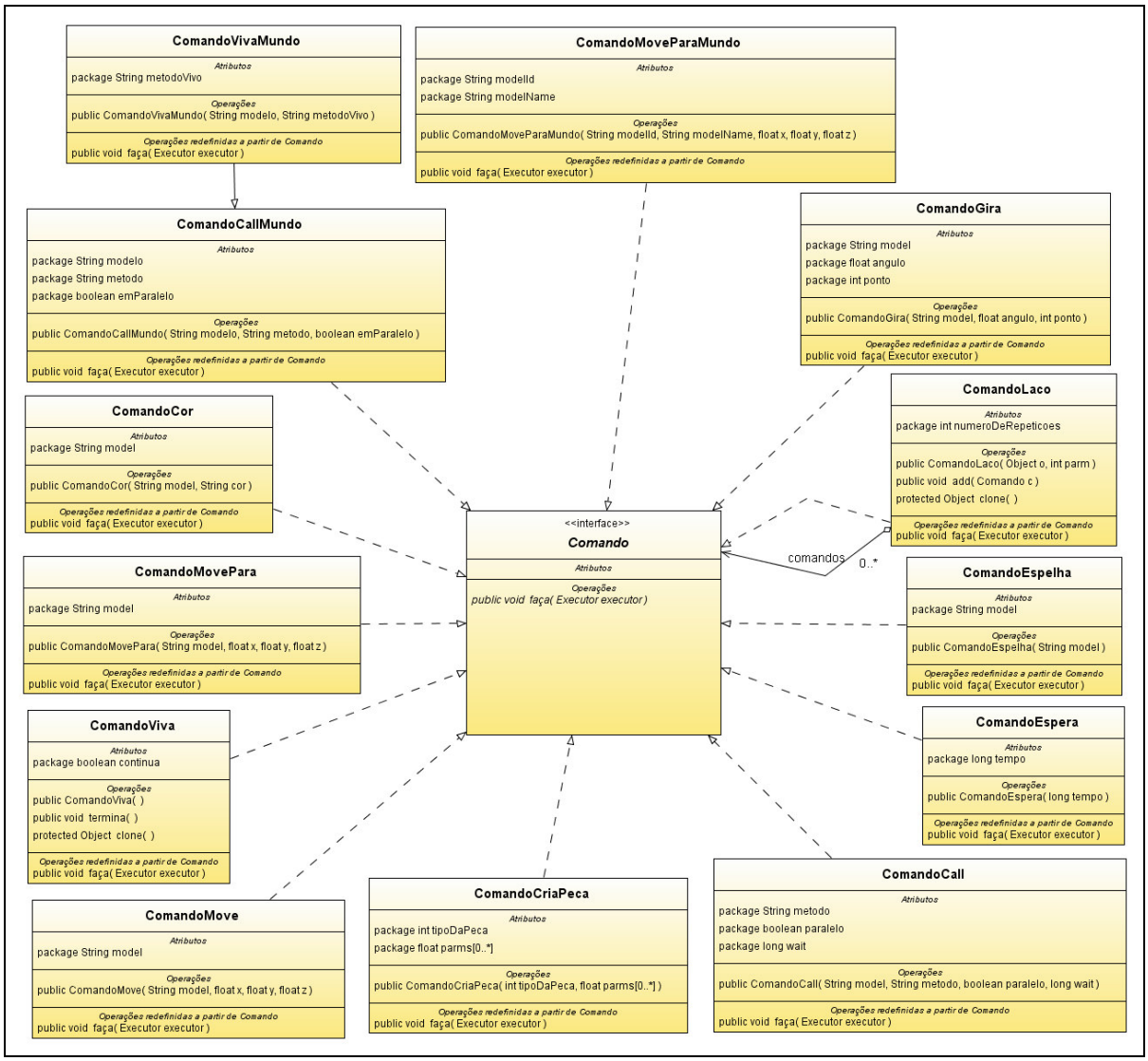


Figura 18 – Diagrama da interface `Comando`

Tangram.Comandos

Interface Comando

All Known Implementing Classes:
ComandoCall, ComandoCallMundo, ComandoCor, ComandoCriaPeca, ComandoEspelha, ComandoEspera, ComandoGira, ComandoLaco, ComandoMove, ComandoMovePara, ComandoMoveParaMundo, ComandoViva, ComandoVivaMundo

---

Esta interface disponibiliza um método para representação e execução de um comando da linguagem do LTD.

Method Summary	
`void`	**`faca`**`(Executor executor)` Executa um comando da linguagem LTD.

Quadro 32 – Interface `Tangram.Comandos.Comando`

Na figura 19 são apresentadas a classe `ModeloExecutavel` e a interface `Executor`.

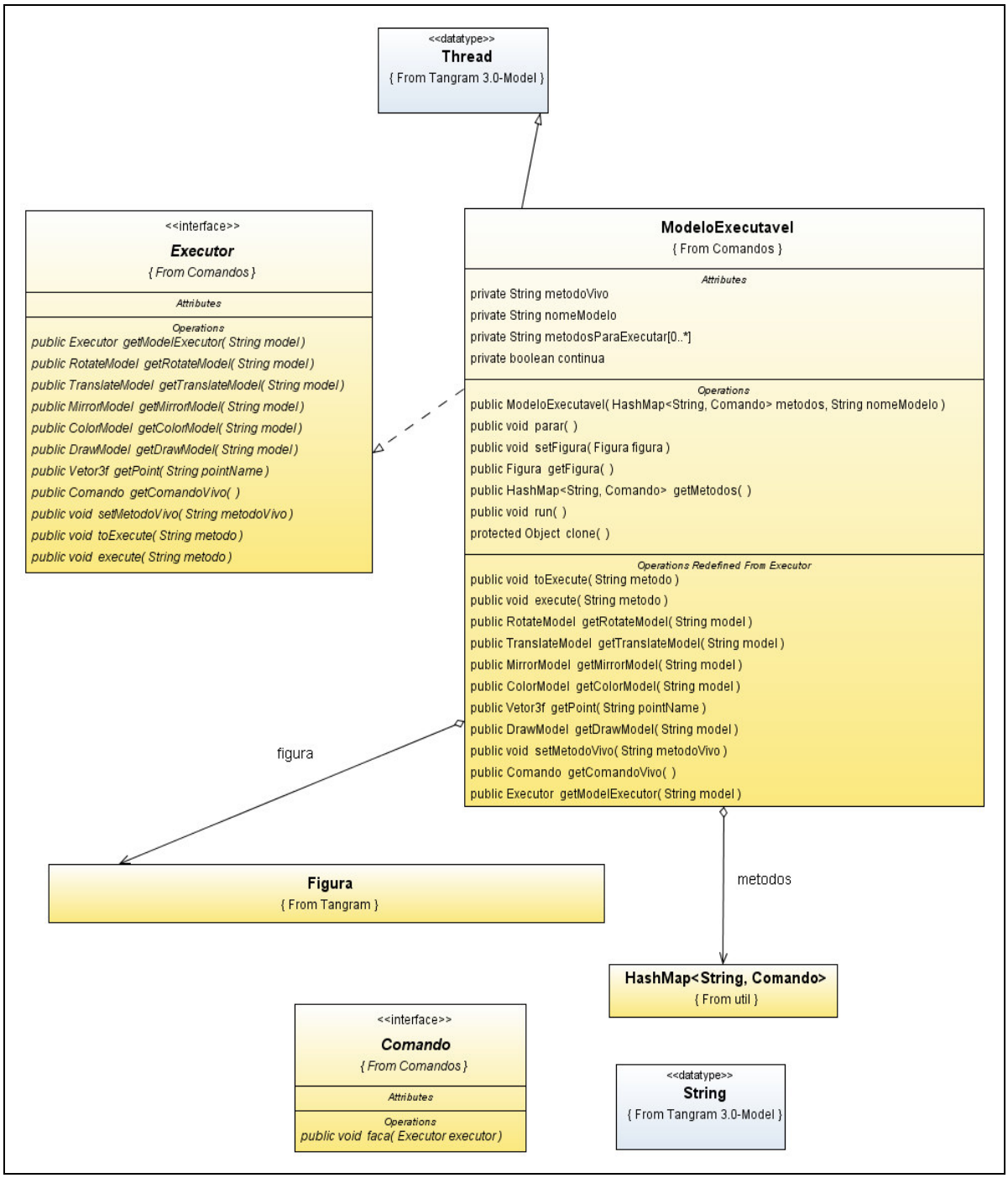


Figura 19 – Diagrama da classe `ModeloExecutavel`

A classe `ModeloExecutavel`, que é uma sub-classe de `Thread` e implementa a interface `Executor`, é criada com a compilação de um modelo. Esta classe contém um `HashMap`, formado por objetos das classes `String` e `Comando`, chamado `metodos`. Neste `HashMap` encontra-se o nome dos métodos do modelo, desenvolvido no editor de modelos, juntamente com o objeto da classe `Comando` que representa a execução do método e seus comandos da linguagem LTD. Cada objeto da classe `ModeloExecutavel` possui um objeto da classe `Figura`, o qual interage com os objetos que implementam a interface `Comando`.

A classe `MundoExecutavel`, que é uma sub-classe de `Thread` e implementa a interface `Executor`, é apresentada no diagrama da figura 20.

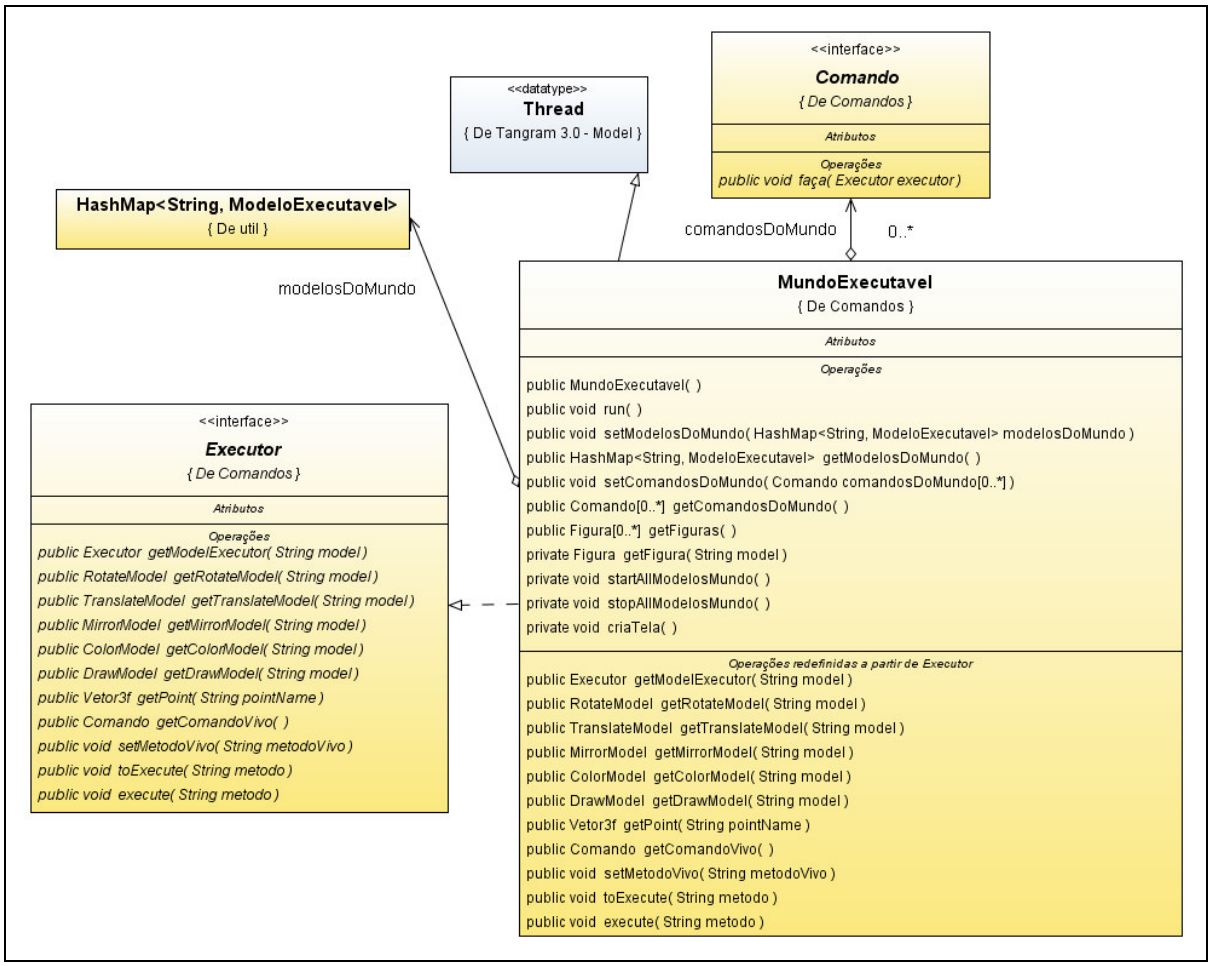


Figura 20 – Diagrama da classe `MundoExecutavel`

A classe `MundoExecutavel` contém uma lista de objetos da classe `Comando`, chamada `comandosDoMundo`, que representam os comandos do mundo desenvolvido no LTD. Os modelos de um mundo são armazenados em um `HashMap`, formado por objetos da classe `String` e `ModeloExecutavel`, chamado `modelosDoMundo`.

Um objeto da classe `MundoExecutavel` é criado a partir da compilação de um mundo e os modelos utilizado no mundo também são compilados quando necessário.

No quadro 33 é apresentada a documentação da interface `Executor` e nos quadros 34 e 35 a documentação das classes `ModeloExecutavel` e `MundoExecutavel` respectivamente.

Tangram.Comandos

Interface Executor

All Known Implementing Classes:
ModeloExecutavel, MundoExecutavel

---

public interface **Executor**
Esta interface disponibiliza os métodos necessários para execução dos comandos da linguagem LTD. As classes que implementam esta interface contém os modelos (objetos que implementam as interfaces ColorModel, DrawModel, MirrorModel, RotateModel e TranslateModel) que responderão as chamadas dos objetos da interface Comando. Esta interface disponibiliza um método para a execução de ações (métodos) de um programa da linguagem LTD.

Method Summary	
`void`	**`execute`**`(java.lang.String metodo)` Executa um método com o nome passado por parâmetro.
`ColorModel`	**`getColorModel`**`(java.lang.String model)` Retorna um objeto de ColorModel que tenha o nome passado por parâmetro.
`Comando`	**`getComandoVivo`**`()` Retorna o comando que está executando constantemente.
`DrawModel`	**`getDrawModel`**`(java.lang.String model)` Retorna um objeto de DrawModel que tenha o nome passado por parâmetro.
`MirrorModel`	**`getMirrorModel`**`(java.lang.String model)` Retorna um objeto de MirrorModel que tenha o nome passado por parâmetro.
`Executor`	**`getModelExecutor`**`(java.lang.String model)` Retorna um objeto de Executor que tenha o nome passado por parâmetro.
`Vetor3f`	**`getPoint`**`(java.lang.String pointName)` Retorna um Vetor3f que representa um ponto que tenha o nome passado por parâmetro.
`RotateModel`	**`getRotateModel`**`(java.lang.String model)` Retorna um objeto de RotateModel que tenha o nome passado por parâmetro.
`TranslateModel`	**`getTranslateModel`**`(java.lang.String model)` Retorna um objeto de TranslateModel que tenha o nome passado por parâmetro.
`void`	**`setMetodoVivo`**`(java.lang.String metodoVivo)` Seleciona o método que deve ser executado constantemente.
`void`	**`toExecute`**`(java.lang.String metodo)` Envia o nome de um método que deve ser executado paralelamente.

Quadro 33 – Interface `Tangram.Comandos.Executor`

Tangram.Comandos

Class ModeloExecutavel

All Implemented Interfaces:
java.lang.Cloneable, java.lang.Runnable, Executor

---

extends java.lang.Thread
implements Executor, java.lang.Cloneable
Esta classe implementa métodos para executar os comandos de uma modelo da linguagem LTD.

Field Summary	
`private boolean`	**`continua`** Flag para o método run().
`private Figura`	**`figura`** Armazena o objeto de Figura do modelo.
`private java.util.HashMap <java.lang.String, Comando>`	**`metodos`** Contém os comandos de cada método de um modelo.
`private java.util.ArrayList <java.lang.String>`	**`metodosParaExecutar`**

	Lista de métodos para executar paralelamente (Através do método run()).
private java.lang.String	**metodoVivo** Nome do Método que deve ser executado pelo comando viva.
private java.lang.String	**nomeModelo** Nome deste modelo.

Constructor Summary

ModeloExecutavel(java.util.HashMap<java.lang.String,Comando> metodos, java.lang.String nomeModelo) Cria nova instância da classe ModeloExecutavel passando os métodos do modelo e seu nome.

Method Summary

protected java.lang.Object	**clone**() Clona o objeto desta classe.
void	**execute**(java.lang.String metodo) Executa um método passando o nome dele por parâmetro.
ColorModel	**getColorModel**(java.lang.String model) Retorna um objeto de ColorModel que tenha o nome passado por parâmetro.
Comando	**getComandoVivo**() Retorna o comando que está executando repetidamente.
DrawModel	**getDrawModel**(java.lang.String model) Retorna um objeto de DrawModel que tenha o nome passado por parâmetro.
Figura	**getFigura**() Retorna a figura do modelo.
java.util.HashMap <java.lang.String ,Comando>	**getMetodos**() Retorna os métodos do modelo.
MirrorModel	**getMirrorModel**(java.lang.String model) Retorna um objeto de MirrorModel que tenha o nome passado por parâmetro.
Executor	**getModelExecutor**(java.lang.String model) Retorna um objeto de Executor que tenha o nome passado por parâmetro.
Vetor3f	**getPoint**(java.lang.String pointName) Retorna um Vetor3f que representa um ponto que tenha o nome passado por parâmetro.
RotateModel	**getRotateModel**(java.lang.String model) Retorna um objeto de RotateModel que tenha o nome passado por parâmetro.
TranslateModel	**getTranslateModel**(java.lang.String model) Retorna um objeto de TranslateModel que tenha o nome passado por parâmetro.
void	**parar**() Para a execução método run().
void	**run**() Este método fica em loop, executando os métodos do modelo que estão na lista de métodos para executar, até o método parar() ser executado.
void	**setFigura**(Figura figura) Atribui a figura do modelo.
void	**setMetodoVivo**(java.lang.String metodoVivo) Seleciona o método que deve ser executado constantemente.
void	**toExecute**(java.lang.String metodo) Adiciona um método do modelo a lista de métodos para executar em paralelo.

Quadro 34 – Classe `Tangram.Comandos.ModeloExecutavel`

Tangram.Comandos

Class MundoExecutavel

All Implemented Interfaces:
java.lang.Runnable, Executor

---

extends java.lang.Thread
implements Executor
Esta classe implementa métodos para executar os comandos de um mundo da linguagem do LTD.

Field Summary

private java.util.ArrayList <Comando>	**comandosDoMundo** Contém os comandos do mundo.
private EditorGrafico	**editorGrafico** Interface gráfica a qual será desenhada as figuras.
private java.util.HashMap <java.lang.String, ModeloExecutavel>	**modelosDoMundo** Contém os modelos do mundo.

Constructor Summary

MundoExecutavel() Cria nova instância da classe MundoExecutavel.

Method Summary

private void	**criaTela**() Cria a interface gráfica para desenhar os modelos.
Void	**execute**(java.lang.String metodo) Executa um método passando o nome dele por parâmetro.
ColorModel	**getColorModel**(java.lang.String model) Retorna um objeto de ColorModel que tenha o nome passado por parâmetro.
java.util.ArrayList <Comando>	**getComandosDoMundo**() Retorna os comandos do mundo.
Comando	**getComandoVivo**() Retorna o comando que esta executando repetidamente.
DrawModel	**getDrawModel**(java.lang.String model) Retorna um objeto de DrawModel que tenha o nome passado por parâmetro.
private Figura	**getFigura**(java.lang.String model) Retorna um objeto de Figura que tenha o nome passado por parâmetro.
java.util.ArrayList <Figura>	**getFiguras**() Retorna as figuras do mundo.
MirrorModel	**getMirrorModel**(java.lang.String model) Retorna um objeto de MirrorModel que tenha o nome passado por parâmetro.
Executor	**getModelExecutor**(java.lang.String model) Retorna um objeto de Executor que tenha o nome passado por parâmetro.
java.util.HashMap <java.lang.String ModeloExecutavel>	**getModelosDoMundo**() Retorna os modelos do mundo.
Vetor3f	**getPoint**(java.lang.String pointName) Retorna um objeto de Vetor3f, o qual representa um ponto, que tenha o nome passado por parâmetro.
RotateModel	**getRotateModel**(java.lang.String model) Retorna um objeto de RotateModel que tenha o nome passado por parâmetro.
TranslateModel	**getTranslateModel**(java.lang.String model) Retorna um objeto de TranslateModel que tenha o nome passado por parâmetro.

void	**run**() Este Método inicializa a execução do mundo, criando a tela, os modelos e executando os comandos.
void	**setComandosDoMundo**(java.util.ArrayList<Comando> comandosDoMundo) Atribui os comandos do mundo.
void	**setMetodoVivo**(java.lang.String metodoVivo) Seleciona o método que deve ser executado constantemente.
void	**setModelosDoMundo**(java.util.HashMap<java.lang.String,ModeloExecutavel> modelosDoMundo) Atribui os modelo do mundo.
private void	**startAllModelosMundo**() Executa o método start() em todos os objetos de ModeloExecutavel do mundo.
private void	**stopAllModelosMundo**() Executa o método parar() em todos os objetos de ModelosExecutavel do mundo.
void	**toExecute**(java.lang.String metodo) Envia o nome de um método que deve ser executado paralelamente.

Quadro 35 – Classe `Tangram.Comandos.MundoExecutavel`

3.2.4 Diagramas de seqüência

Nesta seção são apresentados os diagramas de seqüência da ferramenta.

Para desenhar as figuras do Tangram é utilizado a classe `GLCanvas`. Na criação de uma instância de `GLCanvas` é passado por parâmetro um objeto da classe `Desenha`, a qual implementa a interface `GLEventListner`. O método `display`, implementado pela classe `Desenha`, é responsável por desenhar as figuras do Tangram e enviar os eventos do *mouse* para serem tratados pelo `MouseEventHandlers`. Na figura 21 é apresentado o diagrama de seqüência do método `display`.

O objeto de `GLAutoDrawable` é enviado por parâmetro na chamado do método `display`. A mensagem `getGL`, enviada para `GLAutoDrawable`, retorna um objeto de `GL`, o qual disponibiliza o acesso as funções do OpenGl. Após obter o objeto de `GL` é limpa a tela, através da mensagem `glLoadIdentity`, para ser redesenhada as figuras.

Os eventos do *mouse* são armazenados em um objeto de `ArrayList`. Durante o redesenho das figuras os eventos são enviados, juntamente com o objeto de `GL`, para `MouseEventHandler.action`, onde os eventos são tratados. Os objetos de `DrawModel`, os quais serão desenhados, são obtidos pela chamada `MouseEventHandler.getModelos` e para cada um destes `DrawModel` é enviada a mensagem `draw(GL gl)`, onde cada objeto faz o seu desenho. Após a câmera é posicionada com a chamada `Camara.action(GL gl)`.

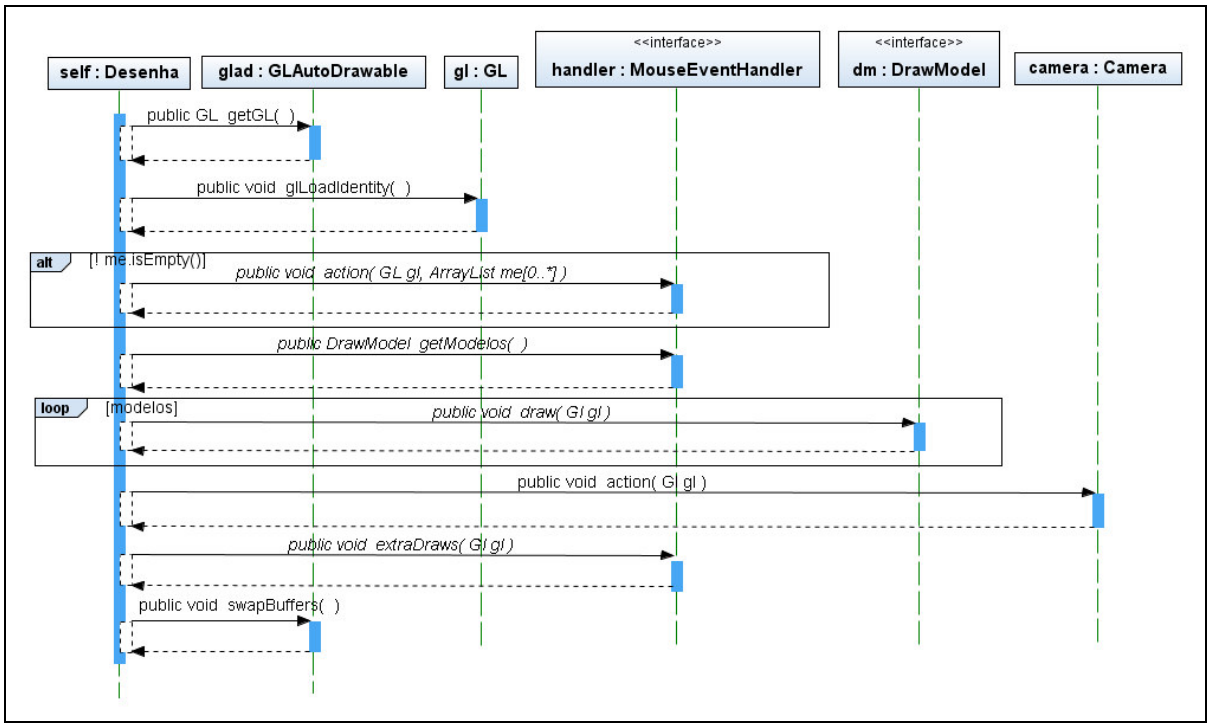


Figura 21 – Diagrama de seqüência de `Desenha.display`

O método `MouseEventHandler.extraDraws` é chamado em seguida. Este método desenha o nome das peças de uma figura do Tangram. Após todo o processo de desenho é enviada a mensagem `swapBuffers()` para `GLAutoDrawable` disponibilizar a imagem na tela.

No tratamento dos eventos do *mouse* é necessário saber se o ponteiro está posicionado sobre uma peça do Tangram. Para isso é utilizada a técnica *ray trancing* (LENGYEL, 2004, cap. 5), a qual detecta o ponto de intersecção entre um segmento de reta e um plano. O método `interserccao` da interface `RayTrancing` é implementado pela classe `Figura` e o diagrama de seqüência deste método é apresentado na figura 22.

Para cada instância de `Peca` do objeto de `Figura` é enviado a mensagem `Peca.interseccao` passando a mesma `Linha` que a `Figura` recebeu por parâmetro. Na seqüência, para cada um dos objetos de `Triangulo` da `Peca`, é enviado a mensagem `getIntersectionPoint`, que por sua vez, envia a mensagem `Plano.getIntersection`, onde é calculado o ponto de intersecção entre a linha e o plano do triângulo. Após é calculado se este ponto está dentro ou fora do `Triangulo` através da chamada do método `getBarycentricCoordinates`, passando como parâmetro o ponto de intersecção da linha com o plano.

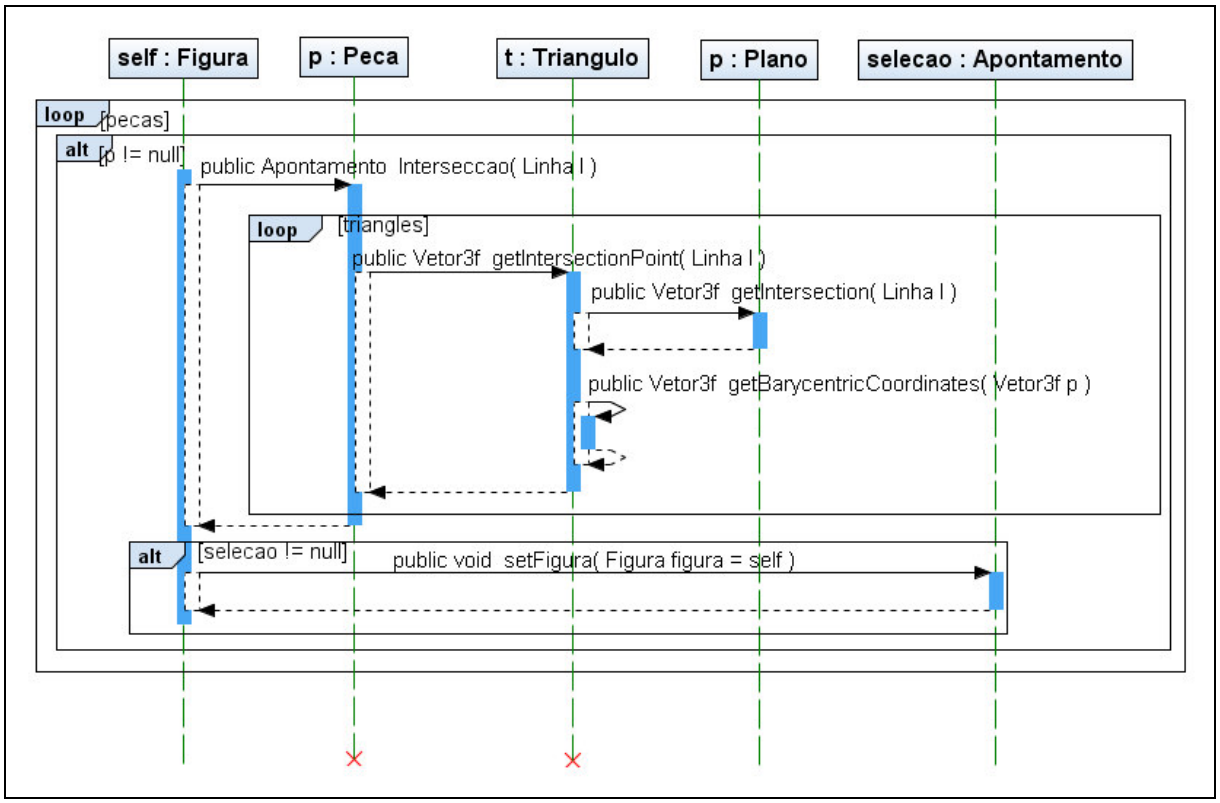


Figura 22 – Diagrama de seqüência de `Figura.interseccao`

Se o ponto encontra-se dentro do triângulo é criada uma nova instância de `Apontamento`, a qual é retornada pelo método. Voltando a execução para `Figura`, é verificado se houve a intersecção. Caso houve, a `Figura` é adicionada ao objeto de `Apontamento` através do método `setFigura` e este objeto de `Apontamento` é devolvido pelo método `interseccao`.

Os objetos de `DrawModel` devolvidos pelo método `MouseEventHandler.getModelos` são instâncias de `Figura` e o diagrama de seqüência do método `Figura.draw` é apresentado na figura 23.

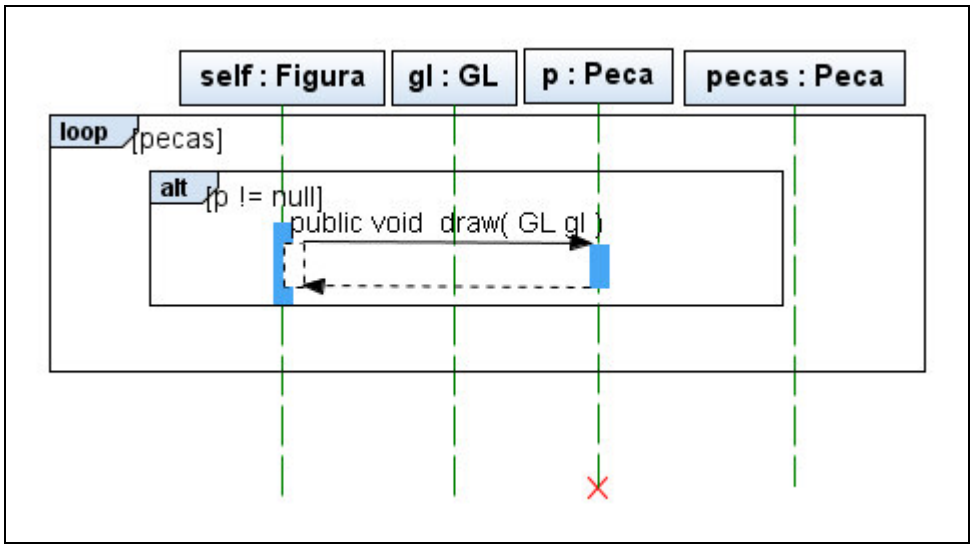


Figura 23 – Diagrama de seqüência de `Figura.draw`

Para cada `Peca` da `Figura` é enviada a mensagem `Peca.draw` (Figura 24).

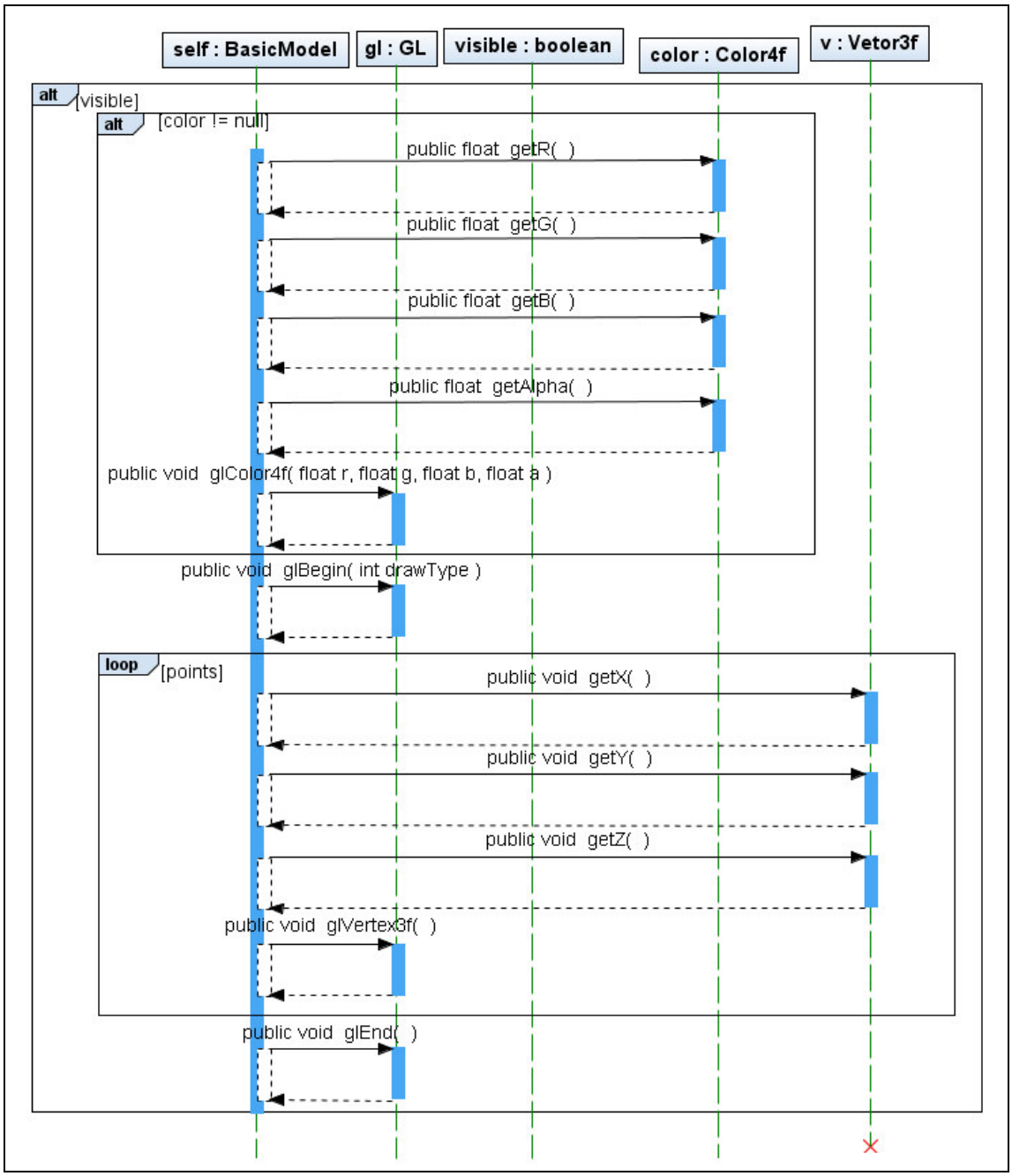


Figura 24 – Diagrama de seqüência de `BasicModel.draw`

Se a propriedade `visible` da `Peca` estiver como verdadeira, a peça é desenhada. Para desenhá-la são obtido os valores `r`, `g`, `b` e `alpha` da cor da peça e enviadas por parâmetro para o método `GL.glColor4f`. Na seqüência é enviada a mensagem `GL.glBegin` passando por parâmetro o tipo da forma geométrica que será desenhada. No caso do Tangram são apenas desenhados os tipos `GL_TRIANGLES` e `GL_QUADS`. Após, para cada ponto da peça, são obtidos os valores de `x`, `y`, `z` e desenhado um vértice através do método `GL.glVertex3f`. O desenho é finalizado enviando a mensagem `GL.glEnd()`.

Para mover uma `Figura` a um ponto específico utiliza-se o método `translateTo`

(Figura 25). Este ponto é relacionado ao centro da figura e é recebido por parâmetro. Para calcular novas posições de cada um dos pontos da figura é calculada a distância entre o ponto, atual, do centro da figura e o ponto onde a figura deverá ser movida. Para este cálculo é enviada a mensagem `sub(Vetor3f)` para o ponto recebido por parâmetro, enviando o ponto central da peça. O `Vetor3f` resultante é o valor que deve ser adicionado a cada um dos pontos da figura.

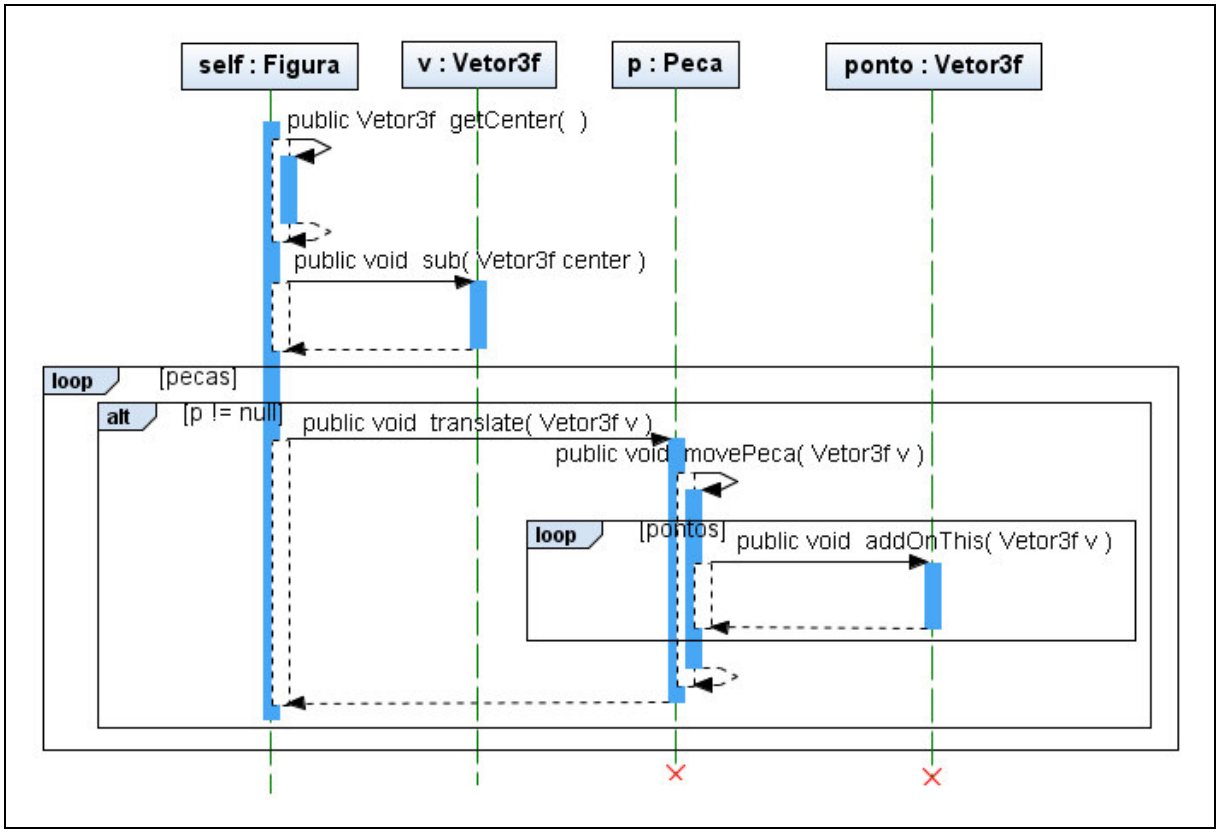


Figura 25 – Diagrama de seqüência de `Figura.TranslateTo`

O método `Peca.translate` adiciona o valor de um `Vetor3f` ao valor dos seus `Vetor3f` através do método `Vetor3f.addOnThis`.

Na figura 26 é apresentado o diagrama de seqüência do método `Figura.translate`.

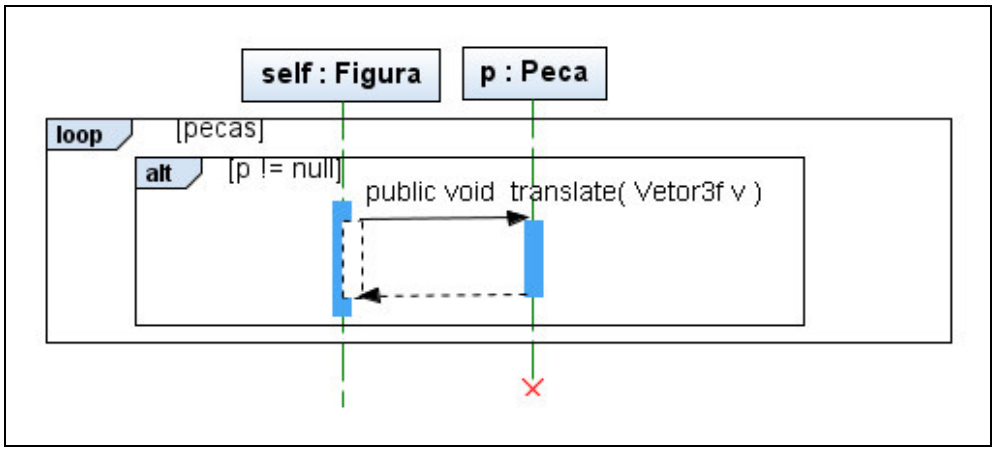


Figura 26 – Diagrama de seqüência de `Figura.Translate`

O método `Figura.translate` adiciona um valor ao valor dos pontos de uma figura.

Para rotacionar uma figura, em torno de seu centro, é utilizado o método `Figura.Rotate(float angulo)`. Neste método é obtido o centro da figura, através do método `getCenter()`, e passado como parâmetro na mensagem `Figura.Rotate(float angulo, Vetor3f center)`.

O diagrama de `Figura.Rotate(float angulo)` é apresentado na figura 27.

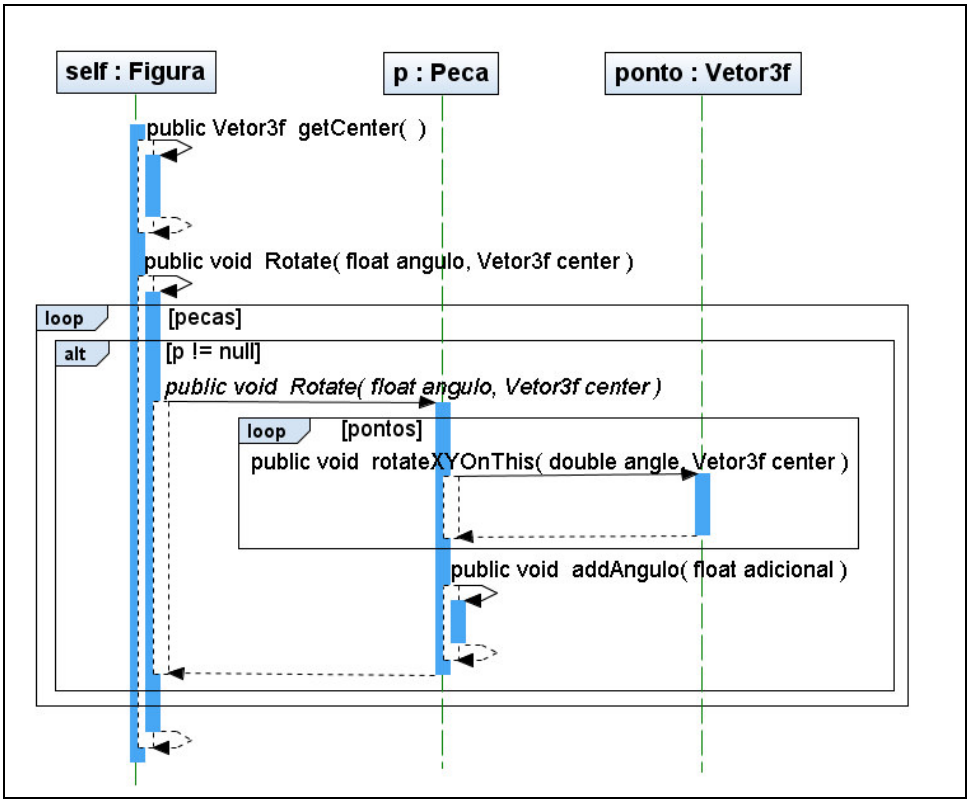


Figura 27 – Diagrama de seqüência de `Figura.rotate`

Para cada uma das peças da figura é chamado o método `Peca.Rotate` passando por parâmetro o ângulo de rotação e o ponto que representa o centro de rotação. Em seguida é chamado, para cada ponto da peça, o método `Vetor3f.rotateXYOnThis` que calcula as novas coordenadas do ponto após a rotação. A rotação acontece apenas no plano `x` e `y`, as coordenadas de `z` não se alteram na rotação. Após rotacionado os pontos da peça, é adicionado o valor do ângulo desta rotação ao valor total do ângulo que a peça foi rotacionada, através do método `addAngulo`.

Para espelhar uma figura é usado o método `Figura.Mirror()`. Este método espelha a figura em torno de seu centro. O cálculo das novas coordenadas de cada ponto é feito na chamada do método `Vetor3f.mirrorXOnThis`, passando como parâmetro o valor do eixo `x` do ponto central da figura. O diagrama de seqüência do método para espelhar uma `Figura` é apresentado na figura 28.

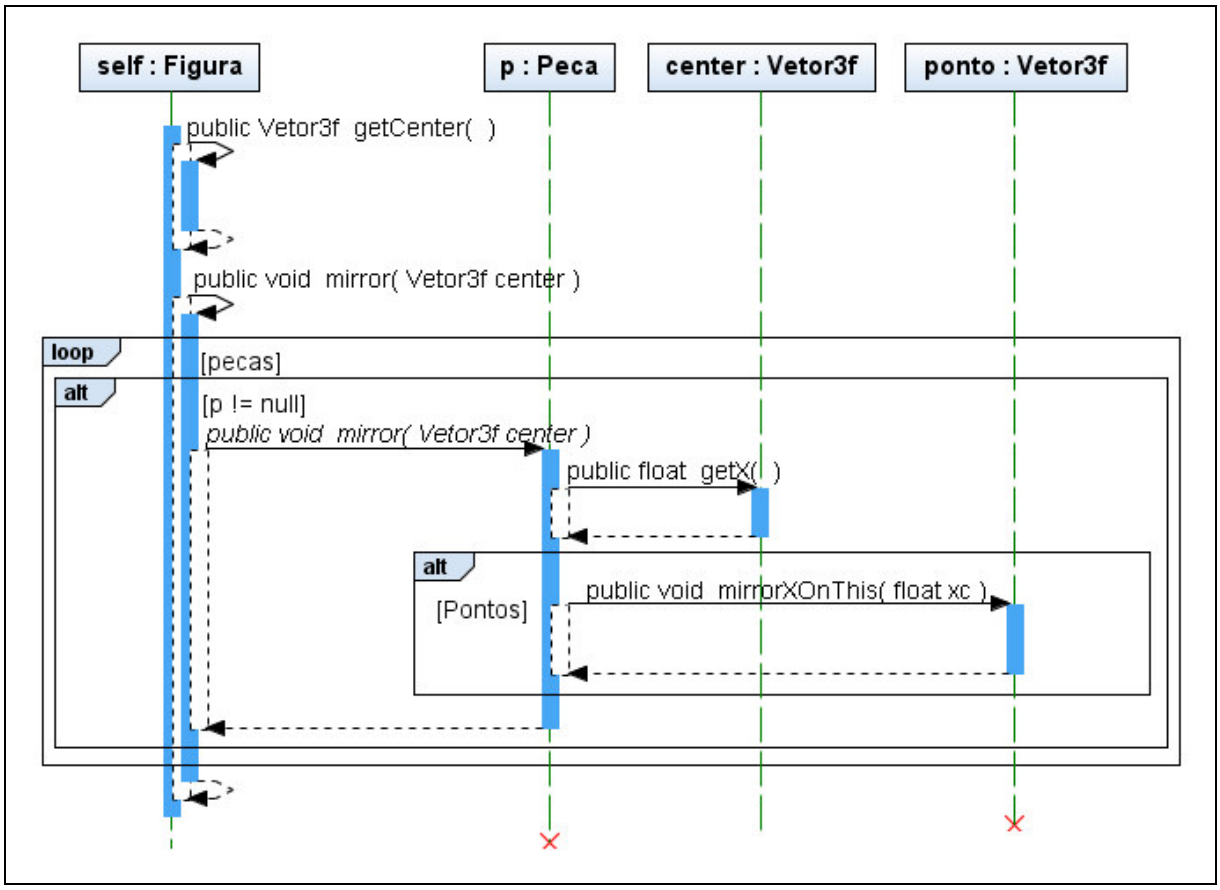


Figura 28 – Diagrama de seqüência de `Figura.mirror`

Para alterar a cor de uma figura utiliza-se o método `Figura.setColor` passando por parâmetro o objeto de `Color4f` que representa a nova cor. O diagrama de seqüência para mudar a cor de uma `Figura` é apresentado na figura 29.

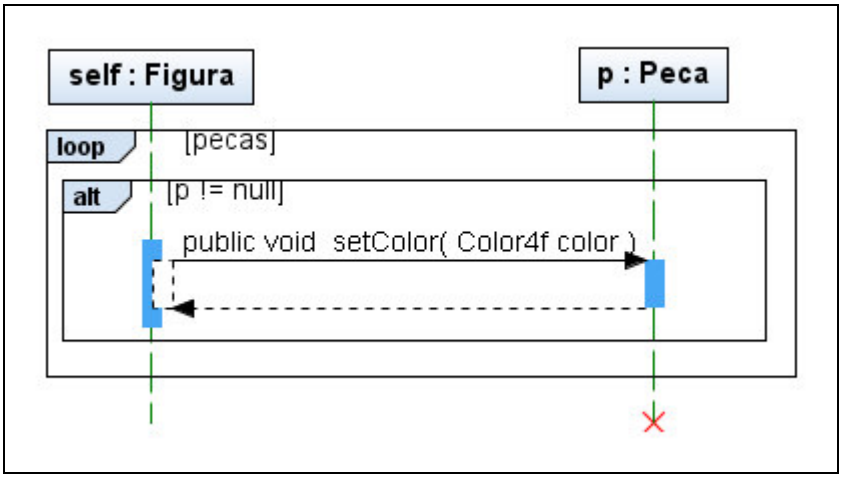


Figura 29 – Diagrama de seqüência de `Figura.setColor`

A propriedade `visible` das peças de uma figura pode ser alterada através do método `Peca.setVisible`. O diagrama de seqüência da figura 30 apresenta o método `Figura.setVisible`, o qual chama, para cada peça da figura, o método `Peca.setVisible`.

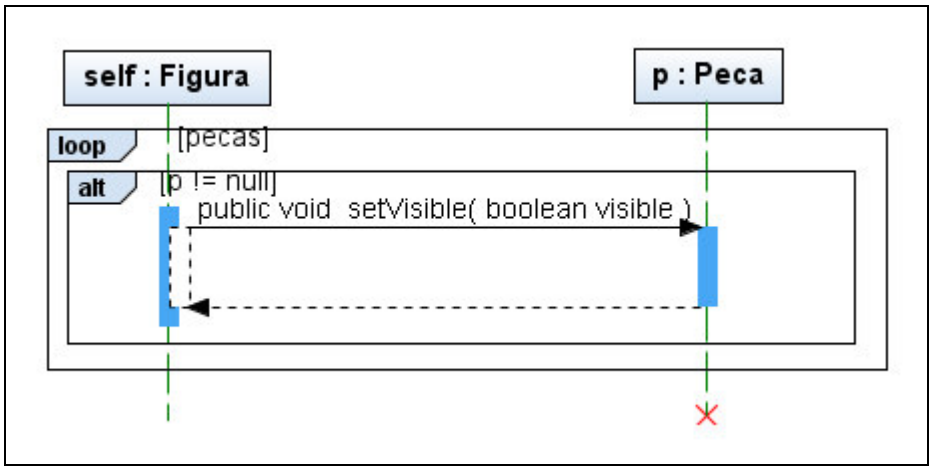


Figura 30 – Diagrama da seqüência de `Figura.setVisible`

Os métodos, das interfaces `ColorModel`, `DrawModel`, `MirrorModel`, `RotateModel` e `TranslateModel`, implementados pelas classes `Figura` e `Peca` são chamados a partir dos objetos que implementam a interface `Comando`. Estes objetos são instanciados a partir da compilação de um modelo ou mundo. Um modelo possui métodos, os quais são representados pela classe `ComandoLaco`. O `ComandoLaco` também representa o comando repita da linguagem LTD. Cada `ComandoLaco` possui uma lista de objetos de `Comando`. Quando o método `faca` do `ComandoLaco` é executado, ele chama, para cada um dos objetos de `Comando`, o método `faca`.

Na figura 31 é apresentado o diagrama de seqüência do método `ComadoLaco.faca`.

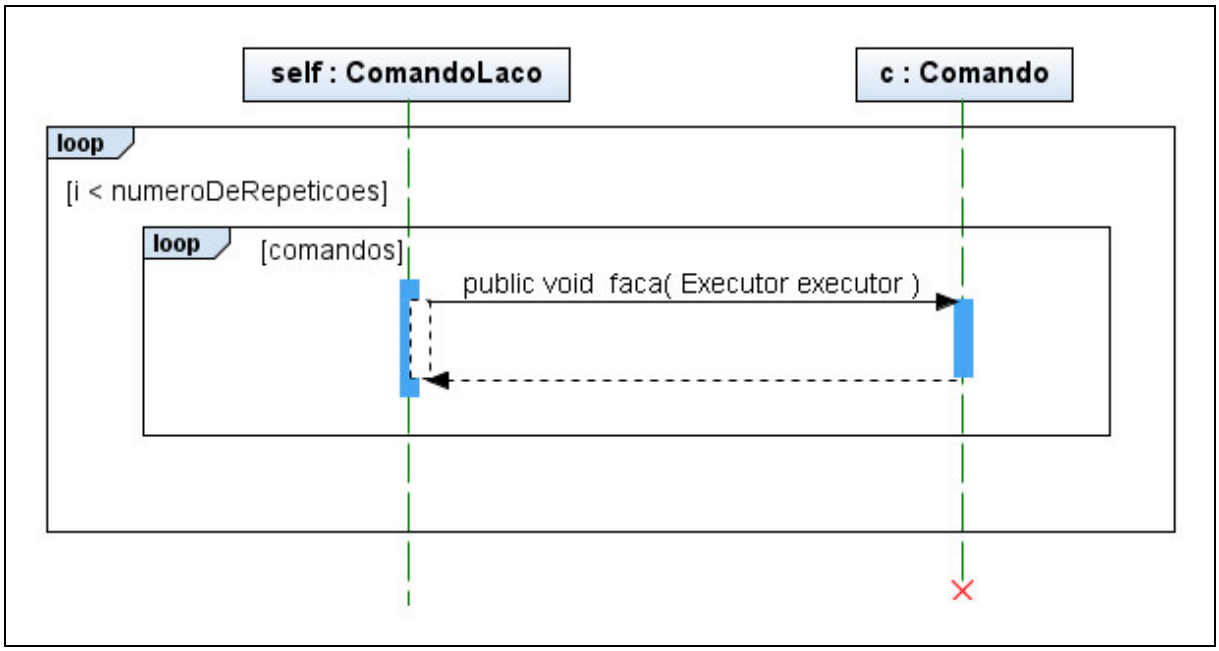


Figura 31 – Diagrama de seqüência de `ComandoLaco.faca`

O método nativo `CRIA` de um modelo possui os comandos para criar as peças de uma figura do Tangram. O comando para criação das peças é representado pela classe `ComandoCriaPeca`. O diagrama de seqüência para o `ComandoCriaPeca.faca` é apresentado

na figura 32.

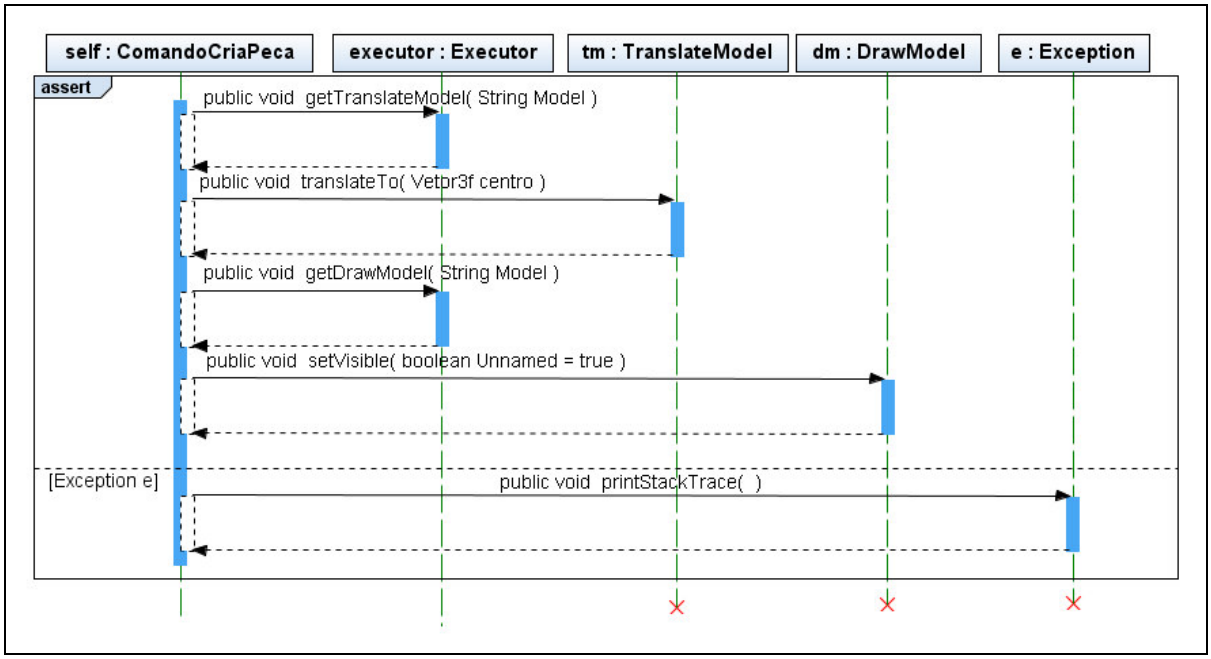


Figura 32 – Diagrama de seqüência de `ComandoCriaPeca.faca`

A classe `ComandoMove` representa o comando para mover uma `Peca` ou `Figura`. Na figura 33 é apresentado o diagrama de seqüência do método `ComandoMove.faca`.

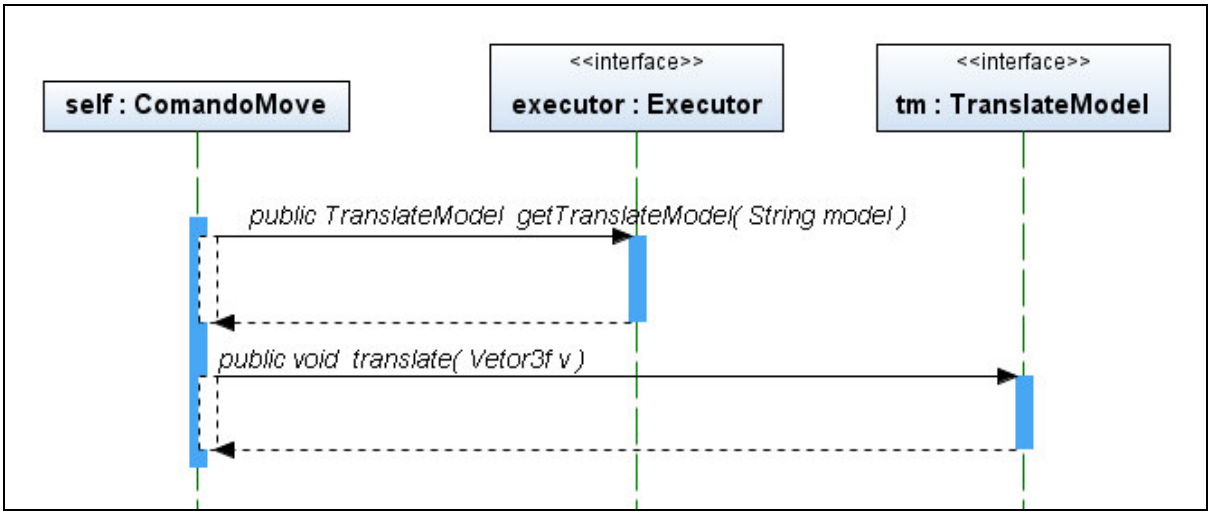


Figura 33 – Diagrama de seqüência de `ComandoMove.faca`

No método `ComandoMove.faca` é enviada a mensagem `Executor.getTranslateModel`, a qual retorna o objeto que implementa `TranslateModel`. O objeto retornado pode ser da classe `Figura` ou `Peca`, dependendo do código fonte do modelo, o qual fornece o parâmetro enviado pela mensagem. Após é enviada a mensagem `translate` ao `TranslateModel`. Desta forma a classe `ComandoMove` pode executar o método `translate` de qualquer classe que implemente a interface `TranslateModel`.

As classes `ComandoCor`, `ComandoEspelha` e `ComandoGira` têm o método `faca`

semelhante ao do `ComandoMove`, apenas chamando, respectivamente, os métodos `Executor.getColorModel`, `Executor.getMirrorModel` e `Executor.RotateModel`, no lugar de `Executor.getTranslateModel` e após chamando, respectivamente, os métodos `ColorModel.setColor`, `MirrorModel.mirror` e `RotateModel.Rotate` no lugar de `TranslateModel.translate`.

O comando `faça`, em um modelo, da linguagem LTD é representado pela classe `ComandoCall`. O método `ComandoCall.faca` chama o método `Executor.execute` passando por parâmetro o nome do método que deve ser executado. O diagrama de seqüência do método `ComandoCall.faca` é apresentado na figura 34.

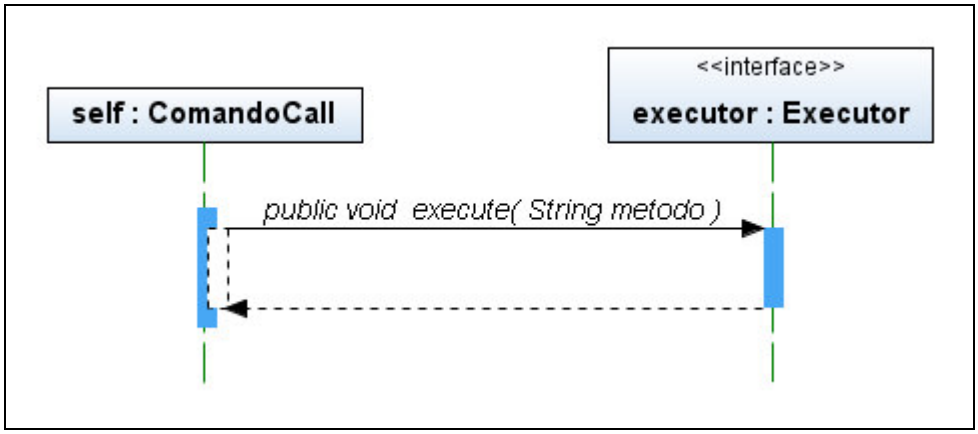


Figura 34 – Diagrama de seqüência de `ComandoCall.faca`

O comando `faça`, em um mundo, da linguagem LTD é representado pela classe `ComandoCallMundo`. A figura 35 apresenta o diagrama de seqüência o método `ComandoCallMundo.faca`.

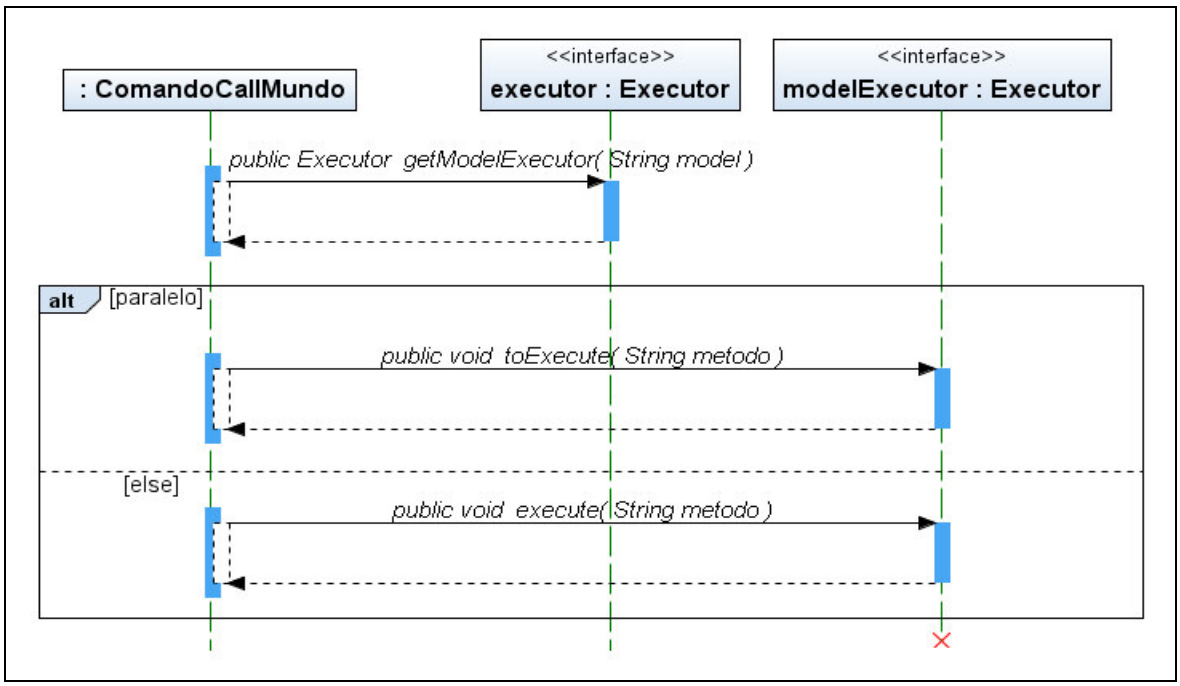


Figura 35 – Diagrama de seqüência de `ComandoCallMundo.faca`

No `ComandoCallMundo.faca` é enviada a mensagem `Executor.getModelExecutor`, o qual retorna um objeto da classe `ModeloExecutavel`. O parâmetro passado neste método é o nome dado ao modelo. Após é verificado se o comando deve ser executado em paralelo. Caso sim, chama o método `Executor.toExecute`, o qual adiciona o método em uma lista para ser executado paralelamente pelo modelo através do método `run()`.

O comando `VIVA`, em um mundo, da linguagem LTD é representado pelo `ComandoVivaMundo`. O método `ComandoVivaMundo.faca` (Figura 36) é semelhante ao `ComandoCallMundo`, chamando o método `getModelExecutor`. Após é chamado o método `setMetodoVivo`, passando por parâmetro o nome do método do modelo a ser executado. Na seqüência é chamado o método `faca` da superclasse, a qual irá executar o método `VIVA` do modelo.

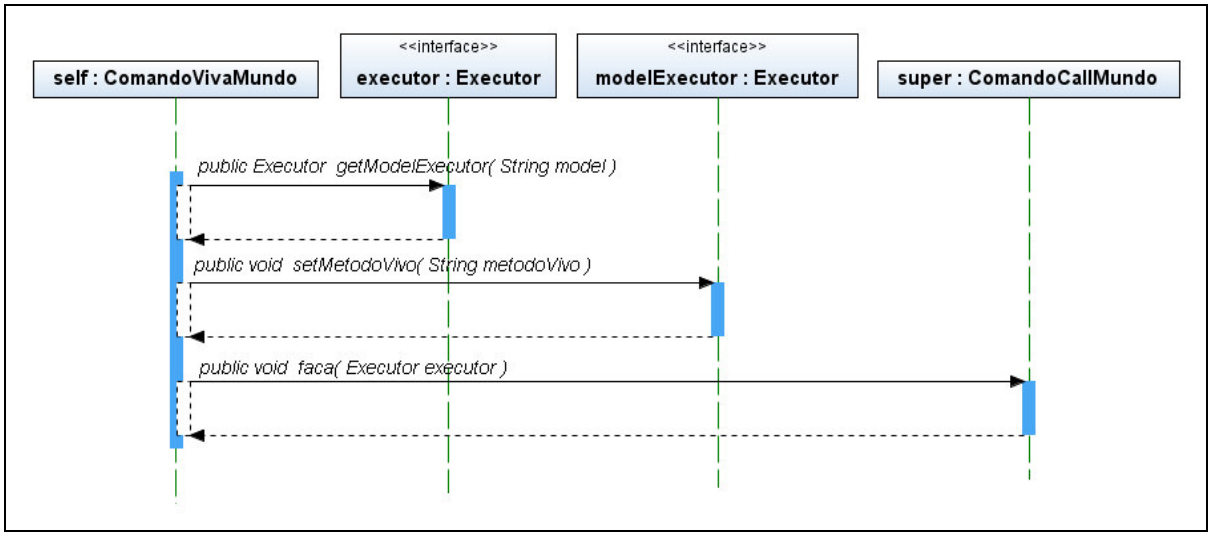


Figura 36 – Diagrama de seqüência de `ComandoVivaMundo.faca`

O método `VIVA` de um modelo é representado pela classe `ComandoViva`. Na execução do método `ComandoViva.faca` é criada uma nova `Thread` que ficará executando o método `vivo` que foi atribuído pelo objeto da classe `ComandoVivaMundo`. A execução da `Thread` criada no `ComandoViva` só é terminada após o comando `TERMINA` da linguagem LTD for enviado. O diagrama de seqüência do método `ComandoViva.faca` é apresentado na figura 37.

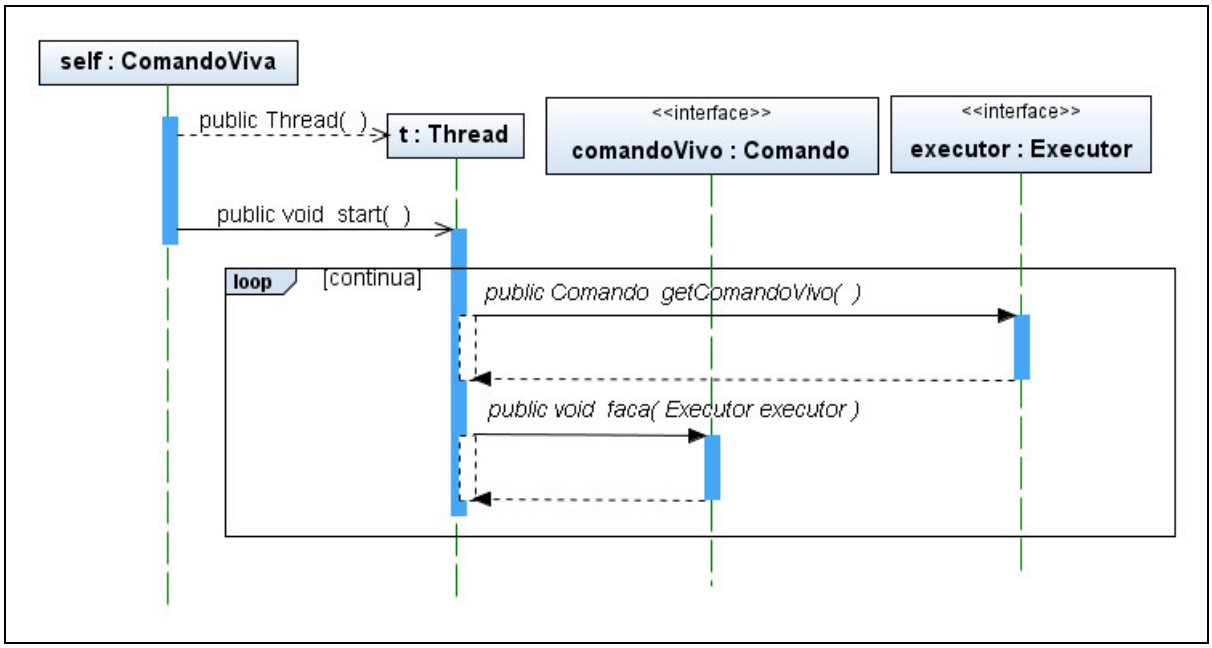


Figura 37 – Diagrama de seqüência de `ComandoViva.faca`

A classe `ModeloExecutavel` implementa a interface `Executor`, a qual é utilizada nas classes que implementam a interface `Comando`. A figura 38 apresenta o diagrama de seqüência do método `getTranslateModel`, o qual é semelhante aos métodos `getRotateModel`, `getMirrorModel`, `getColorModel` e `getDrawModel`, tendo apenas o tipo do objeto retornado diferente.

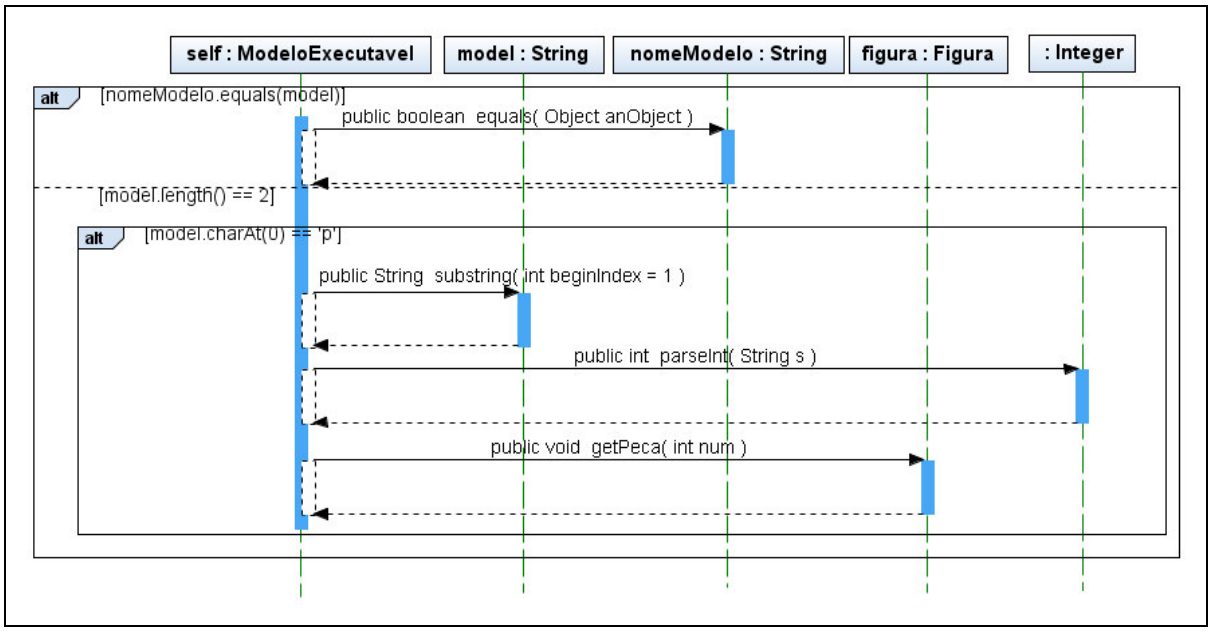


Figura 38 – Diagrama de seqüência de `ModeloExecutavel.getTranslateModel`

No método da figura 38 é comparado o nome do modelo com a `String` recebida por parâmetro. Se for igual retorna o objeto de `Figura`, caso contrário é verificado se o parâmetro é o nome de uma peça, o qual pode ser: `p1`; `p2`; `p3`; `p4`; `p5`; `p6` e `p7`. Caso seja uma peça, é retornado o objeto de `Peca` correspondente ao nome da peça.

Na Figura 39 é apresentado o diagrama de seqüência de `ModeloExecutavel.execute`.

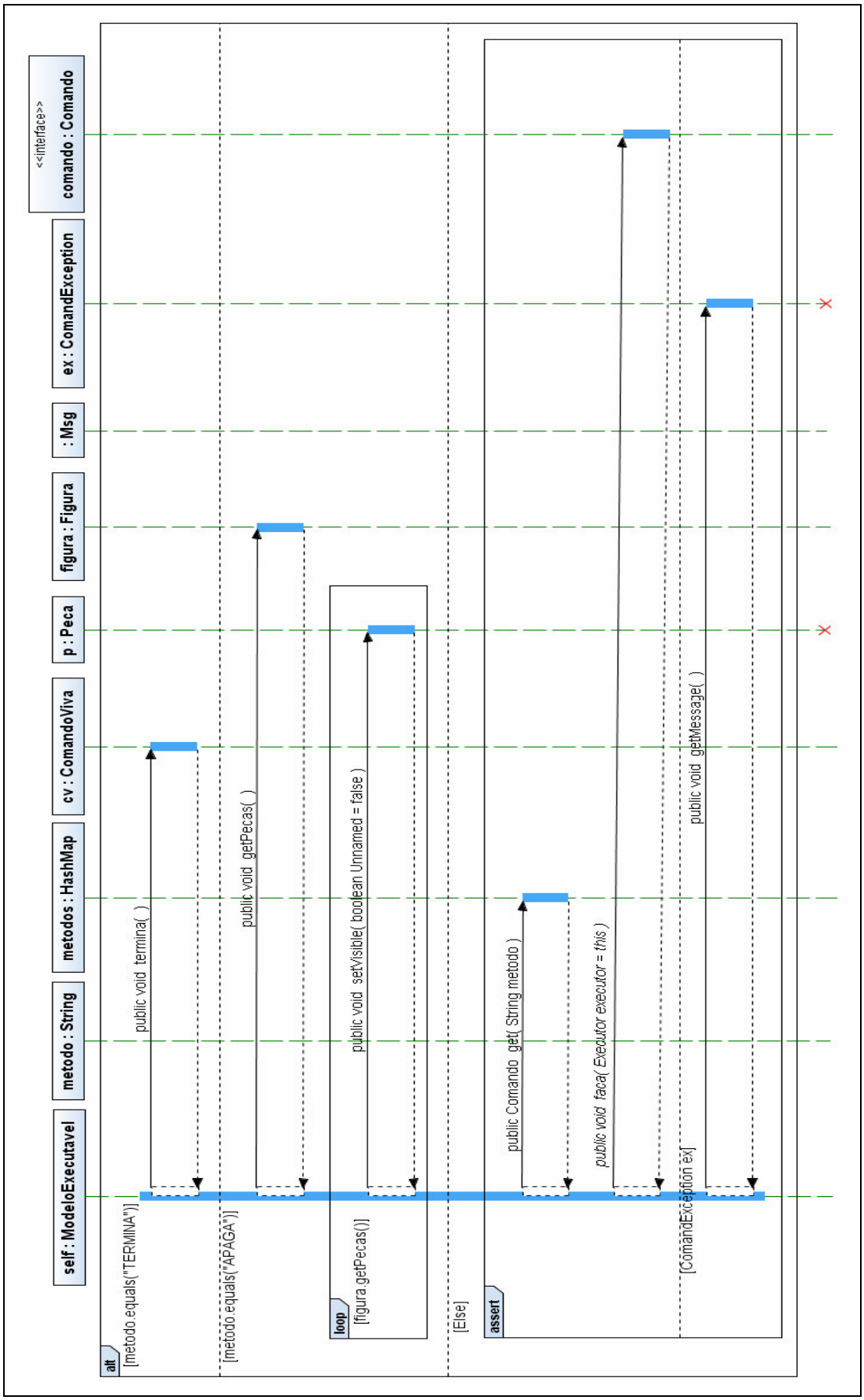


Figura 39 – Diagrama de seqüência de `ModeloExecutavel.execute`

No método `ModeloExecutável.execute` é verifica o nome do método que se deve executar. Caso seja o método `TERMINA`, nativo da linguagem LTD, é enviada a mensagem `ComandoViva.termina`. Caso seja o método `APAGA`, nativo da liguagem LTD, é chamado, para cada peça da figura, o método `Peca.setVisible` passando o valor falso como parâmetro. Caso seja outro método, é buscado da lista de métodos do modelo, representada por um `HashMap`, o `Comando` para ser executado.

Na Figura 40 é apresentado o diagrama de seqüência `ModeloExecutavel.run`. Este método obtém o nome dos métodos, do modelo, para executar paralelamente. Para cada método, do modelo, para executar paralelamente é chamado o método `execute()` passando por parâmetro o nome do método.

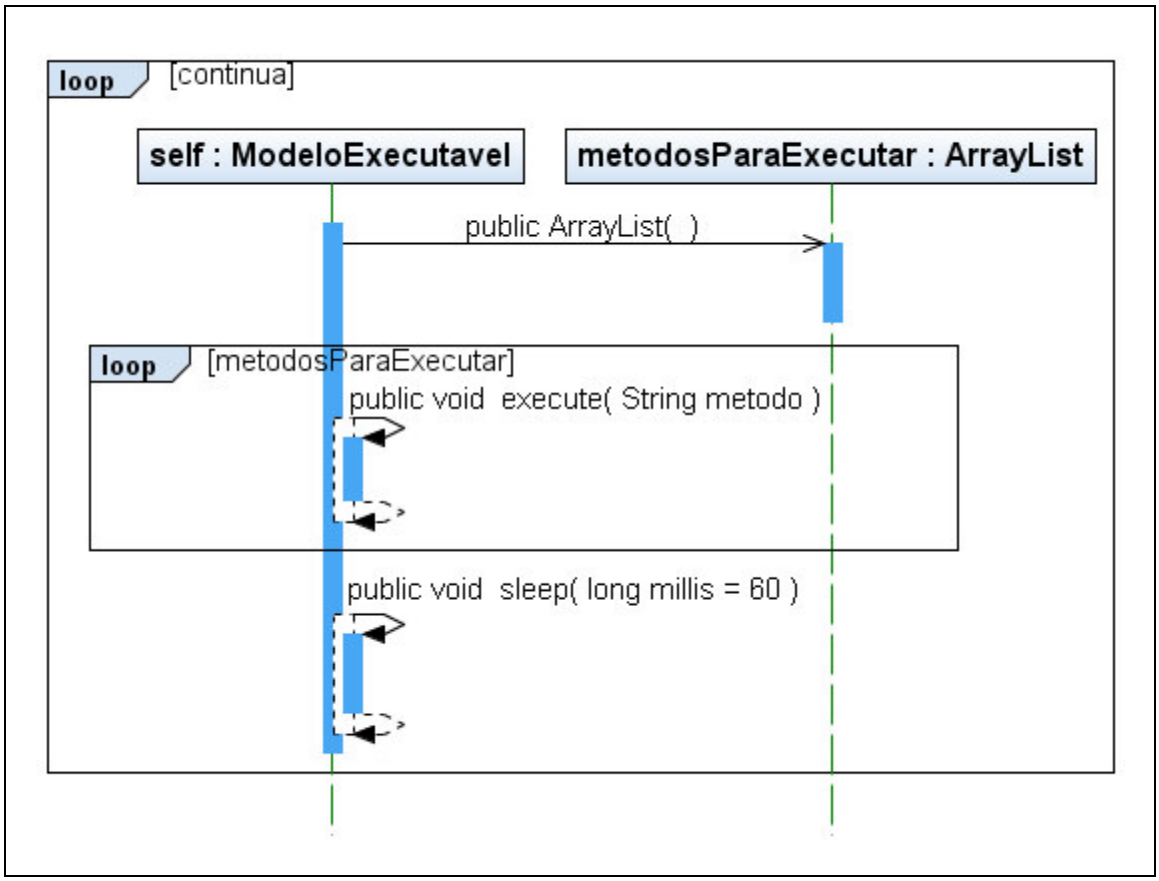


Figura 40 – Diagrama de seqüência de `ModeloExecutavel.run`

Na figura 41 é apresentado o diagrama de seqüência do método `MundoExecutavel.run`. Este método executa os comandos de um mundo do LTD, que por sua vez executa os métodos dos modelos usados no mundo.

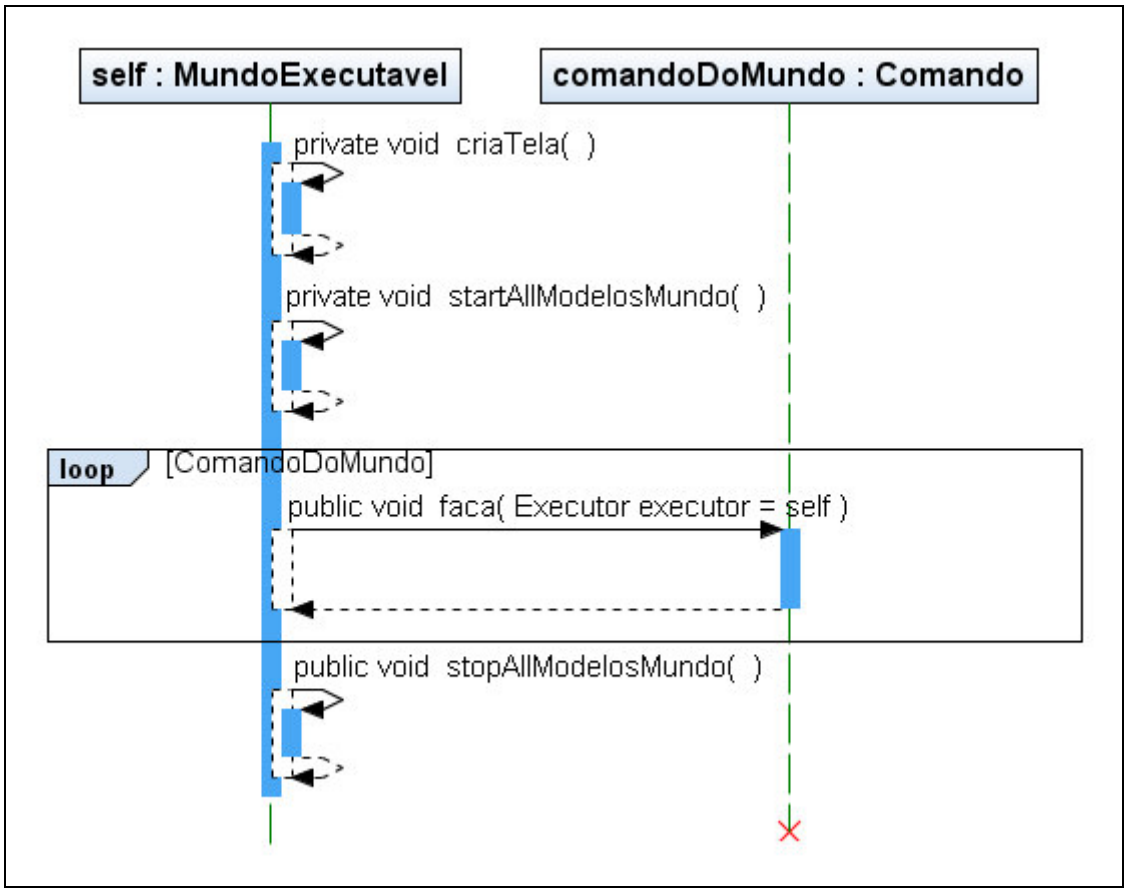


Figura 41 – Diagrama de seqüência de `MundoExecutavel.run`

3.3 IMPLEMENTAÇÃO

A seguir são apresentadas as ferramentas e técnicas utilizadas na implementação e a operacionalidade da ferramenta.

3.3.1 Ferramentas e técnicas utilizadas

Para implementação da ferramenta LTD foi utilizado a IDE Netbeans 6.0 com a linguagem Java. Também foi utilizada a ferramenta GALS, a qual é *freeware* e gera analisadores léxicos e sintáticos a partir das definições regulares, dos *tokens* e da gramática.

Para a linguagem Java, a ferramenta GALS gera as classes `AnalysisError`, `Constants`, `LexicalError`, `Lexico`, `ParserConstants`, `ScannerConstants`, `SemanticError`, `Semantico`, `Sintatico`, `SyntaticError` e `Token`. Entre as classes geradas

pelo GALS, a classe `Semantico` é a única que precisa ser implementada. Nesta classe são implementadas as ações semânticas definidas no Apêndice A.

No quadro 36 é apresentado um trecho de código utilizado para compilar um programa através das classes geradas pelo GALS.

```
01    Lexico lexico = new Lexico(code);
02    Semantico semantico = new Semantico();
03    Sintatico sintatico = new Sintatico();
04    sintatico.parse(lexico, semantico);
```

Quadro 36 – Compilando um programa com as classes geradas pelo GALS

Na linha 1 do quadro 36 é criada uma nova instância de `Lexico` passando por parâmetro o código fonte do programa que se deseja compilar. Através método `parse` da classe `Sintatico`, linha 4 do quadro 36, o código fonte é compilado. O resultado da compilação depende da implementação da classe `Semantico`. Neste projeto a classe `Semantico` cria um objeto de `MundoExecutavel`, quando o programa é um mundo, ou um objeto de `ModeloExecutavel`, quando o programa é um modelo. Também são gerados objetos das classes que implementam a interface `Comando`, onde cada um desses objetos representa um comando da linguagem LTD.

3.3.1.1 Criando uma janela para visualizar um `GLCanvas`

No quadro 37 é apresentado um código do método `MundoExecutavel.criaTela()`, o qual cria uma janela para visualização dos desenhos das peças do Tangram através de um `GLCanvas`.

A classe `MEHforMundoExecutavel`, linha 214 do quadro 37, implementa a interface `MouseEventHandler`. Esta classe não executa nenhuma ação nos métodos `action` e `extraDraws`, apenas armazena os objetos de `Figura` para serem desenhados.

A classe `Animator`, linha 229 do quadro 37, executa constantemente o método `display` da interface `GLAutoDrawable`, a qual é implementada pela classe `GLCanvas`. O método `Animator.setRunAsFastAsPossible`, linha 230, faz com que o redesenho da tela seja o realizado o maior número de vezes que é possível.

```
201 private void criaTela(){
202     // cria um GLCapabilities para definir que trabalha com duplo
203     // buffer e definir o número de bits por pixel
204     GLCapabilities caps = new GLCapabilities();
205     caps.setDoubleBuffered(true);
206     caps.setRedBits(8);
207     caps.setGreenBits(8);
208     caps.setBlueBits(8);
209     caps.setAlphaBits(8);
210 
211     // cria um cavas para o editor Grafico
212     GLCanvas canvas = new GLCanvas(caps);
213 
214     // Cria um objeto da classe que implementa MouseEventHandler
215     MEHforMundoExecutavel handler =
216             new MEHforMundoExecutavel(getFiguras(),
217                     new Camera(new Vetor3f()));
218 
219     // Adiciona o GLEventListener, ouvinte dos eventos do GLCanvas
220     canvas.addGLEventListener(new Desenha(handler));
221 
222     // cria a janela para visualizar o GLCanvas
223     JFrame janela = new JFrame();
224     canvas.setSize(640, 480);
225     janela.add(canvas);
226     janela.pack();
227 
228     // Cria um Animator que chama o método display() constantemente.
229     Animator anime = new Animator(canvas);
230     anime.setRunAsFastAsPossible(true);
231     anime.start();
232 
233     // mostra janela
234     janela.setVisible(true);
235     janela.requestFocus();
236 }
```

Quadro 37 – Código do método `MundoExecutavel.criaTela`

3.3.1.2 Desenhando as peças do Tangram

A classe `Peca` não sobrescreve o método `draw` da classe `BasicModel`, portanto o desenho das peças do Tangram é realizado no método `BasicModel.draw`, o qual o código é apresentado no quadro 38.

Na linha 135 do quadro 38 é alterada a cor para o desenho das peças através do método `glColor4f`. Os valores passados por parâmetro são obtidos do objeto de `Color4f` referenciado por `color`.

Na linha 141 do quadro 38 é iniciado o desenho da peça através do método `glBegin` e na linha 143 cada vértice da peça é definido através do método `glVertex3f`. Os valores dos vetores de uma peça são armazenados em uma lista de objetos da classe `Vetor3f`. O método

`glEnd`, linha 145 do quadro 38, termina o desenho da peça.

```
130 public void draw(GL gl) {
131     if(visible){
132         // Se a cor não for nula
133         if (color != null) {
134             // seta a cor da peça no GL
135             gl.glColor4f(color.getR(),
136                     color.getG(),
137                     color.getB(),
138                     color.getAlpha());
139         }
140         // Inicia o desenha da peça e seta os vértices da peça no GL
141         gl.glBegin(drawType);
142         for (Vetor3f v : points) {
143             gl.glVertex3f(v.getX(), v.getY(), v.getZ());
144         }
145         gl.glEnd();
146     }
147 }
```

Quadro 38 – Código do método `BasicModel.draw`

3.3.1.3 A classe `Vetor3f`

A classe `Vetor3f` é um vetor que armazena três (3) valores reais para representar um vértice de uma peça. Esta classe contém métodos, que realizam determinados cálculos sobre os seus valores, desenvolvidos a partir das propriedades dos vetores. `Vetor3f` foi construído baseado em Lengyel (2004, p. 31).

O código do método `Vetor3f.add`, que implementa a operação de soma entre dois (2) vetores, é apresenta no quadro 39.

```
143     public Vetor3f add(Vetor3f v){
144         return new Vetor3f(x + v.getX(),
145                            y + v.getY(),
146                            z + v.getZ());
147     }
```

Quadro 39 – Código do método `Vetor3f.add`

O método `add` retorna uma nova instância de `Vetor3f`. Para realizar a operação sobre o próprio vetor, ou seja, alterar os valores do vetor, se utiliza o método `addOnThis` (Quadro 40).

```
155     public void addOnThis(Vetor3f v){
156         x += v.getX();
157         y += v.getY();
158         z += v.getZ();
159     }
```

Quadro 40 – Código do método `Vetor3f.addOnThis`

A expressão `OnThis` dos métodos de `Vetor3f` representam que o resultado das operações realizadas são armazenadas sobre o próprio vetor.

No quadro 41 é apresentado o código do método `getLength`, utilizado para calcular o módulo de um vetor.

```
271     public float getLength(){
272         return (float) Math.sqrt( x*x + y*y + z*z );
273     }
```

Quadro 41 – Código do método `Vetor3f.getLength`

No quadro 42 é apresentado o código do método `setScalarProduct`, utilizado para calcular o produto escalar entre dois (2) vetores.

```
284     public float getScalarProduct(Vetor3f v){
285         return (float) x * v.getX() + y * v.getY() + z * v.getZ();
286     }
```

Quadro 42 – Código do método `Vetor3f.getScalarProduct`

No quadro 43 é apresentado o código do método `Vetor3f.vectorProduct`, utilizado para calcular o produto entre dois (2) vetores.

```
321     public Vetor3f vectorProduct(Vetor3f v){
322         return new Vetor3f( y * v.getZ() - z * v.getY(),
323                             z * v.getX() - x * v.getZ(),
324                             x * v.getY() - y * v.getX());
325     }
```

Quadro 43 – Código do método `Vetor3f.vetorProduct`

No quadro 44 é apresentado o código do método `Vetor3f.normalization`, utilizado para calcular o vetor unitário.

```
332     public Vetor3f normalization() throws Exception{
333
334         float length = getLength();
335
336         if(length == 0)
337             throw new Exception("Division by Zero");
338
339         return new Vetor3f( x / length,
340                             y / length,
341                             z / length);
342     }
```

Quadro 44 – Código do método `Vetor3f.normalization`

3.3.1.4 Criando `Triangulo` e `Plano` a partir da `Peca`

Um objeto de `Triangulo` tem três (3) objetos de `Vetor3f` e um (1) objeto de `Plano`. As peças do Tangram são formadas por triângulos, que contém três (3) vértices, e quadriláteros, que contêm quatro (4) vértices. Portanto, os objetos de `Peca` podem ter três (3) ou quatro (4) objetos de `Vetor3f`. Os objetos de `Peca` que possuem três (3) vértices formam um (1) `Triangulo`, enquanto que os objetos de `Peca` que possuem quatro (4) vértices formam

dois (2) triângulos, um formado pelos três (3) primeiros vértices, e outro formado pelo segundo, terceiro e quarto vértice.

No quadro 45 é apresentado o código do construtor da classe `Triangulo`.

```
40      public Triangulo(Vetor3f a, Vetor3f b, Vetor3f c) {
41          this.a = a;
42          this.b = b;
43          this.c = c;
44          this.plano = new Plano(this);
45      }
```

Quadro 45 – Código do construtor da classe `Triangulo`

O construtor de `Triangulo` recebe os três objetos de `Vetor3f` e cria o objeto de `Plano`, que representa o plano em que o triângulo está contido.

No quadro 46 é apresentado o código do construtor da classe `Plano`.

```
55      public Plano(Triangulo t) {
56          try{
57              N = t.getB().sub(t.getA()).
58                      vectorProduct(
59                      t.getC().sub(t.getA()));
60              D = - N.getScalarProduct(t.getA());
61          } catch(Exception e) {
62              System.out.println("Não Existe Plano!");
63          }
64      }
```

Quadro 46 – Código do construtor da classe `Plano`

No quadro 47 é apresentada a fórmula que define um plano, onde `A`, `B` e `C` são os valores das coordenadas de x, y e z de um vetor unitário (LENGYEL, 2004 p. 105) representado por `N` na classe `Plano`. Este vetor unitário é um ponto de uma reta perpendicular ao plano. O valor `D` é a distância entre `N` e o plano.

$$Ax + By + Cz + D = 0,$$

Fonte: Lengyel (2004 p. 105).

Quadro 47 – Fórmula do plano

Um plano pode ser calculado através de um triângulo. Sendo `P0`, `P1` e `P2` vértices de um triângulo, o vetor unitário `N` é calculado pela fórmula do quadro 48.

$$\mathbf{N} = (\mathbf{P}_1 - \mathbf{P}_0) \times (\mathbf{P}_2 - \mathbf{P}_0)$$

Fonte: Lengyel (2004 p. 144).

Quadro 48 – Fórmula do vetor unitário de um plano

Nas linhas 57 a 59 do quadro 46 é calculado o valor de `N` para o objeto da classe `Plano`. Segundo Lengyel (2004, p. 144), o valor de `D` corresponde ao negativo do produto escalar entre `N` e um ponto do plano. O valor de `D` é calculado na linha 60 do quadro 46.

3.3.1.5 Intersecção de uma reta com uma peça do Tangram

Para saber se há intersecção entre uma reta e uma peça do Tangram, é utilizado, para cada triângulo da peça, o método `getIntersectionPoint` da classe `Triangulo` (Quadro 49).

O ponto da intersecção é calculado a partir do plano do triângulo. Na linha 92 do quadro 49 é chamado o método `getIntersection` da classe `Plano` (Quadro 50), o qual retorna o ponto que intersecta o plano. Caso a reta não cruze com o plano, o `Vetor3f P` será nulo e não haverá intersecção. Após é necessário saber se o ponto que intersecta o plano, `Vetor3f P`, está dentro do triângulo. Para isso são calculadas as coordenadas baricêntricas (linha 97 do quadro 49). A fundamentação para o cálculo das coordenadas baricêntricas é a mesma apresentada em Theiss (2006, p. 57-60).

```
 90      public Vetor3f getIntersectionPoint(Linha ray){
 91          // P é ponto onde o ray cruza o plano do triangulo
 92          Vetor3f P = plano.getIntersection(ray);
 93          if(P == null)
 94              return null;
 95
 96          // calcula as coordenadas baricêntricas
 97          Vetor3f baryCoord = getBarycentricCoordinates(P);
 98
 99          // se todos os valores forem positivos o ponto colidiu
100          return ((baryCoord.getX() >= 0 &&
101                   baryCoord.getY() >= 0 &&
102                   baryCoord.getZ() >= 0)
103                  ? P : null);
104      }
```

Quadro 49 – Código do método `Triangulo.getIntersectionPoint`

```
 79      public Vetor3f getIntersection(Linha ray){
 80          // se o produto escalar entre a vetor unitario do plano e
 81          // o ponto direçao de uma linha for zero
 82          // não há intersecção
 83          if(N.getScalarProduct(ray.getDirecao()) == 0 )
 84              return null;
 85
 86          // aplicando (4.16)
 87          float t = ( ( - ray.getOrigem().getScalarProduct(N)) - D ) /
 88                          ray.getDirecao().getScalarProduct(N);
 89
 90          // aplicando propriedade: t deve ser maior ou igual a zero
 91          // pg. 102
 92          if( t>=0 )
 93              // aplicando (5.51)
 94              return ray.getOrigem().add(ray.getDirecao().dot(t));
 95
 96          return null;
 97      }
```

Quadro 50 – Código do método `Plano.getIntersection`

Segundo Lengyel (2004, p. 107), se o produto escalar entre o vetor unitário de um

plano e a direção de uma linha for igual a zero (0) não ocorrerá à intersecção. Esta condição é verificada na linha 83 do quadro 50.

A fórmula que define o ponto de intersecção entre uma reta e um plano é apresentada no quadro 51.

$$\mathbf{P}(t) = \mathbf{S} + t\mathbf{V},$$

Fonte: Lengyel (2004, p. 104).
Quadro 51 – Fórmula do ponto de intersecção entre uma reta e um plano

O valor `S` do quadro 51 representa o ponto de início da reta e `V` o ponto que representa a direção. O valor `t` é calculado a partir da formula do quadro 52.

$$t = \frac{-(\mathbf{N} \cdot \mathbf{S} + D)}{\mathbf{N} \cdot \mathbf{V}}.$$

Fonte: Lengyel (2004, p. 107).
Quadro 52 – Cálculo do valor `t`

Se o valor `t` for maior ou igual a zero (0) então basta aplicar a fórmula do quadro 52 para obter o ponto de intersecção.

O valor `t` é calculado na linha 87 e 88 do quadro 50 e o ponto de intersecção é calculado na linha 94 do quadro 50.

3.3.2 Operacionalidade da ferramenta

Nesta seção é apresentada a operacionalidade da ferramenta.

Ao iniciar a ferramenta é visualizada a tela inicial com os menus arquivo, editar e editor gráfico.

O menu arquivo (Figura 42) dispõe dos seguintes itens:

a) `Novo Modelo`: abre o editor de modelos;

b) `Novo Mundo`: abre o editor de mundos;

c) `Carregar`: carrega um modelo ou mundo;

d) `Salvar`: salva o modelo;

e) `Salvar Como`: salva o modelo com outro nome;

f) `Look & Feel`: muda a aparência das telas;

g) `Sair`: fecha a ferramenta.

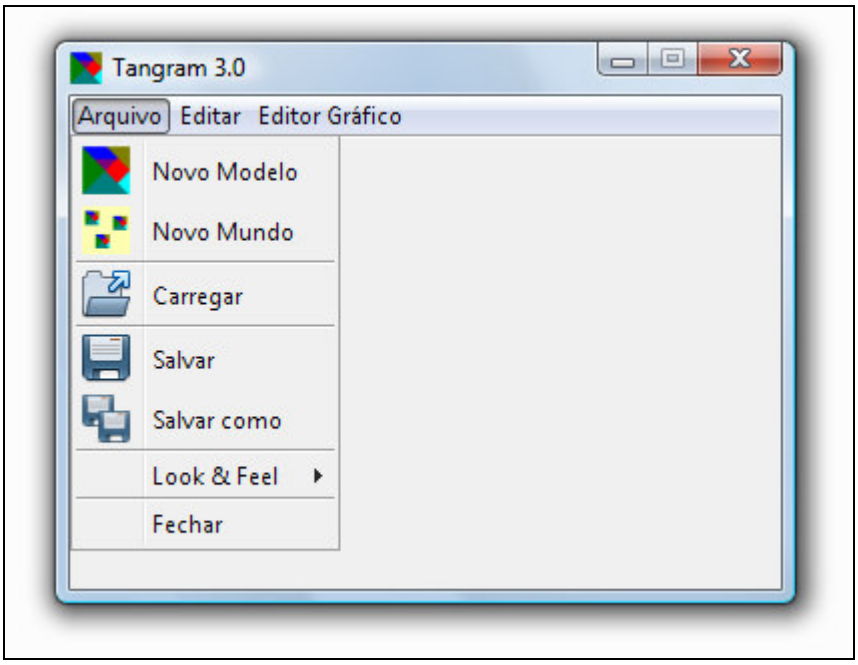


Figura 42 – Tela inicial - menu arquivo

O menu editar (Figura 43) disponibiliza funcionalidades para edição do texto de um programa. Este menu dispõe de um sub-menu `Inserir Comando` e dos itens: `Recortar`; `Copiar` e `Colar`. O sub-menu `Inserir Comando` dispõe de itens para inserir comandos da linguagem LTD no editor de texto.

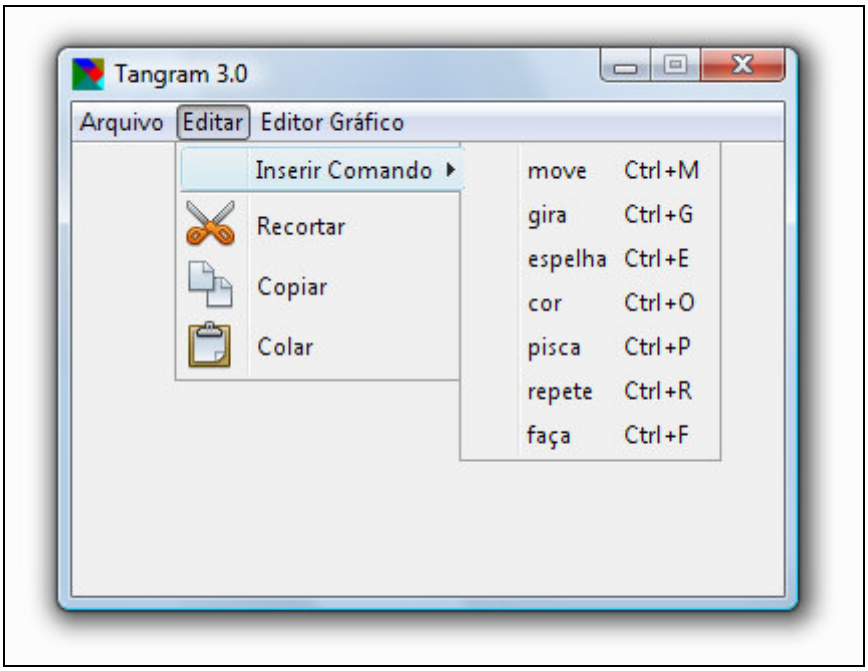


Figura 43 – Tela inicial - menu editar

O menu editor gráfico (Figura 44) disponibiliza itens para alterar o modo de interação com o editor gráfico, os quais são: `Usar Peças`; `Usar Modelo`; `Espelhar`; `Muda Cor` e `Pan`.

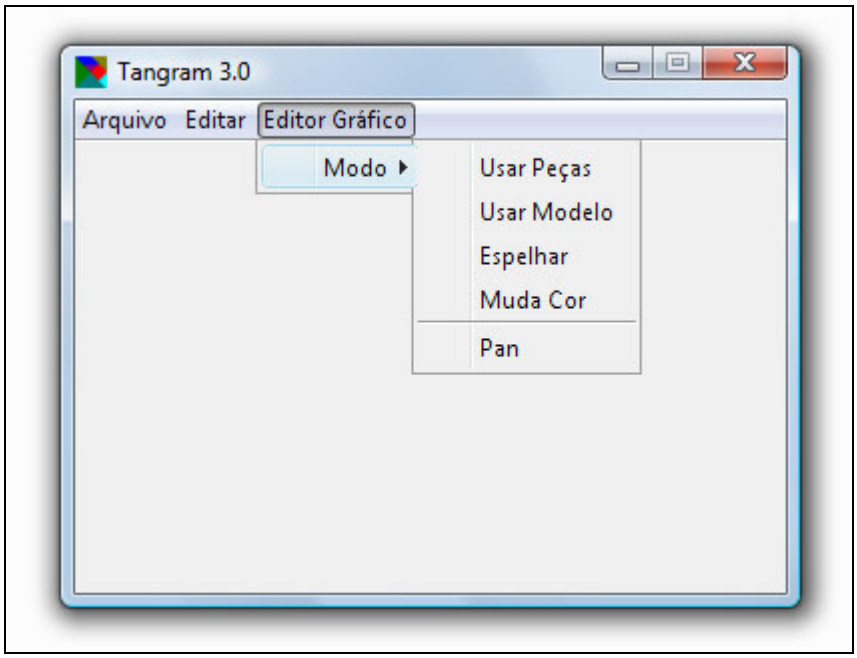


Figura 44 – Tela inicial - menu editor gráfico

3.3.2.1 Editor de modelos

Ao abrir um novo modelo, ou carregar um, é inicializado o editor de modelos. Este editor disponibiliza duas telas, o editor gráfico (Figura 45) e o editor textual de modelos (Figura 46).

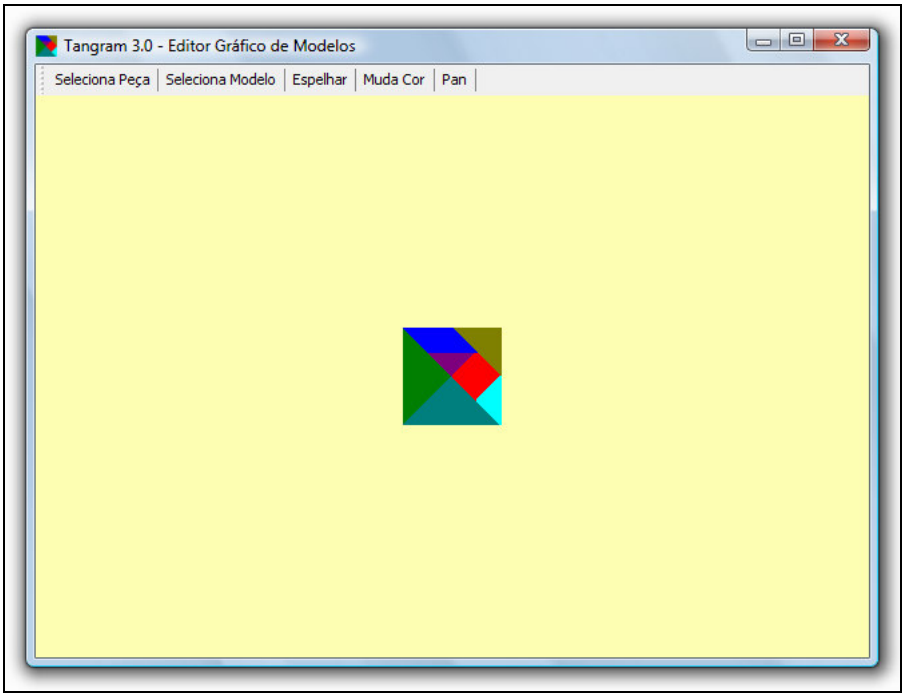


Figura 45 – Tela do editor gráfico

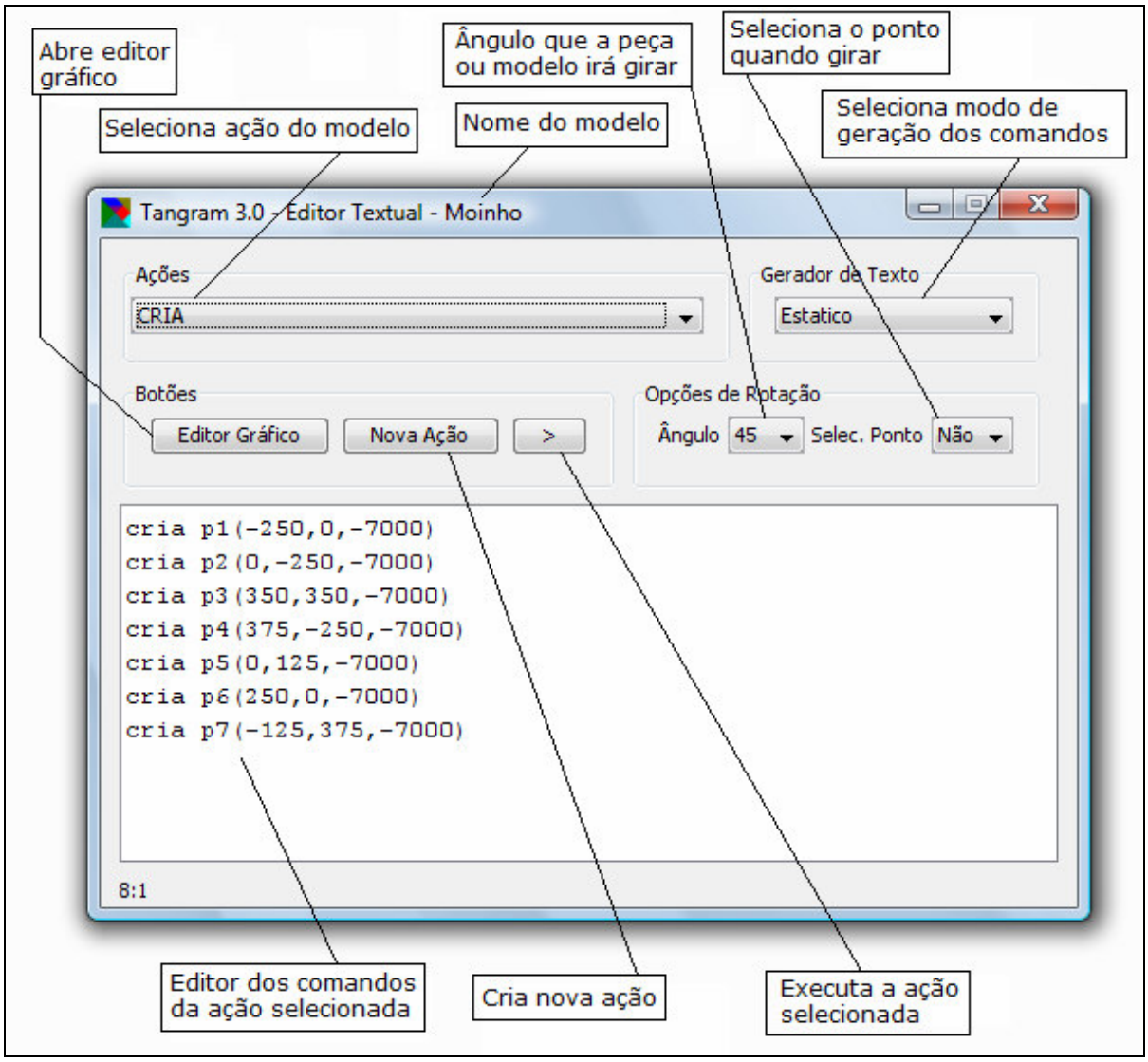


Figura 46 – Tela do editor textual de modelos

No editor textual de modelos é possível selecionar a ação que se deseja editar, criar uma nova ação, executar uma ação, selecionar o modo de geração do código através das interações com o editor gráfico, selecionar o ângulo de rotação para cada interação com o editor gráfico, selecionar se deseja escolher o ponto de rotação, abrir o editor gráfico e editar os comandos das ações do modelo.

Para mover ou girar uma peça do Tangram seleciona-se o modo de interação `Seleciona Peça` no editor gráfico. Ao clicar e segurar o botão esquerdo do *mouse* sobre a peça ela é selecionada, sendo que quando o *mouse* é movimentado, a peça movimenta-se junto. Neste mesmo modo de interação, ao clicar com o botão direito do *mouse* sobre uma peça ela é rotacionada.

Na figura 47 é mostrada a interação, com o editor gráfico, para mover uma peça do Tangram. Enquanto a peça está selecionada são desenhadas as linhas da mesma na sua posição anterior à seleção (Figura 47). Também é desenhada uma linha entre o centro atual e

o centro anterior da peça.

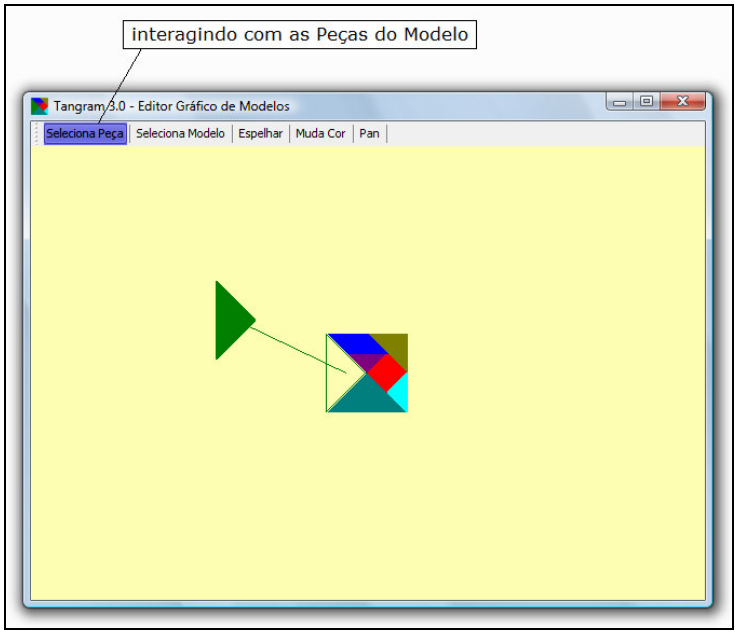


Figura 47 – Movendo uma peça

Para escolher um ponto de rotação para a peça selecionada, clica-se com o botão direito sobre a mesma, onde são visualizados os pontos (Figura 48) e então clica-se sobre o ponto desejado para realizar a operação.

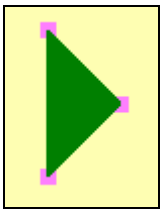


Figura 48 – Seleção do ponto de rotação

Para mover ou rotacionar a figura, formada pelas sete (7) peças, seleciona-se o modo de interação `Seleciona Modelo`. Os mesmos procedimentos do modo `Seleciona Peça` aplicam-se neste modo.

Para espelhar uma peça, ou a figura inteira, seleciona-se o modo de interação `Espelhar`. Ao clicar sobre uma peça com o botão esquerdo do *mouse* a peça é espelhada, caso clique com o botão direito a figura inteira é espelhada em torno de seu ponto central.

Para mudar a cor de uma peça seleciona-se o modo de interação `Muda Cor`. Ao clicar sobre uma peça com o botão esquerdo do *mouse*, uma janela é aberta, onde o usuário pode escolher a cor para a qual deseja mudar. Para mudar a cor de uma figura inteira utiliza-se o botão direito do *mouse*. Quando o clique não é sobre uma peça, a cor de fundo é alterada.

O modo de interação `Pan` proporciona o deslocamento da câmera. Para isso, basta clicar e arrastar o cursor do *mouse*.

O código da ação `CRIA` é gerado sempre que há uma interação no editor gráfico. O

`modo de geração de comandos` é desabilitado e fica selecionada a opção `Estático`. Na figura 49 é mostrado o código da ação cria e o respectivo desenho da figura.

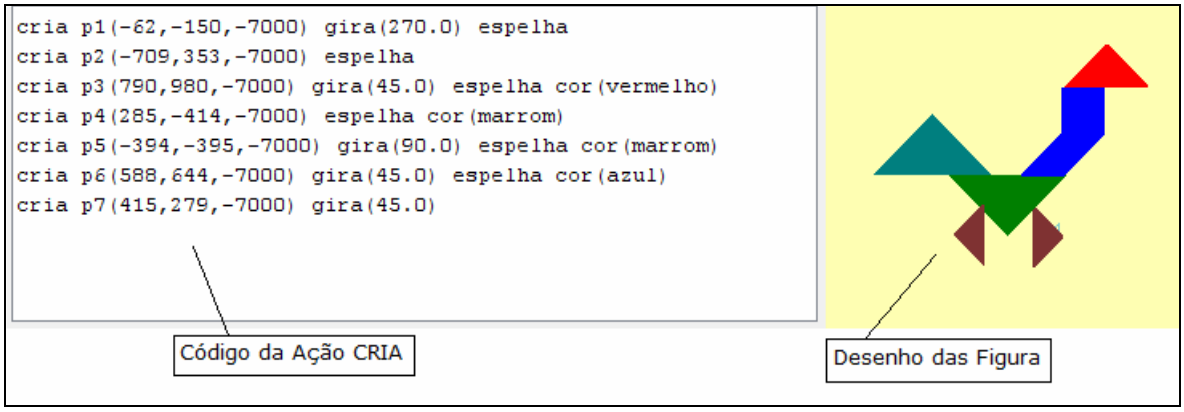


Figura 49 – Código da ação `CRIA` e o respectivo desenho da figura

Após criar uma nova ação, usando o botão `Nova Ação` do editor textual de modelos, ela pode ser selecionada (Figura 50).

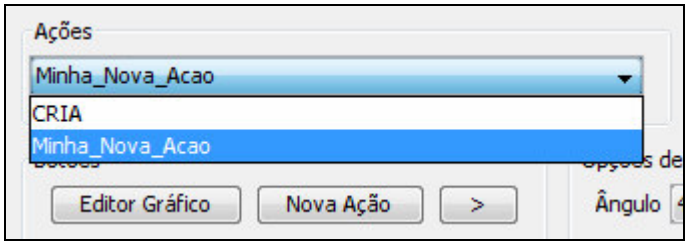


Figura 50 – Selecionando uma ação

Quando a ação selecionada não for `CRIA`, o modo de geração de comando estará habilitado.

Na figura 51 são realizados duas interações com o editor gráfico.

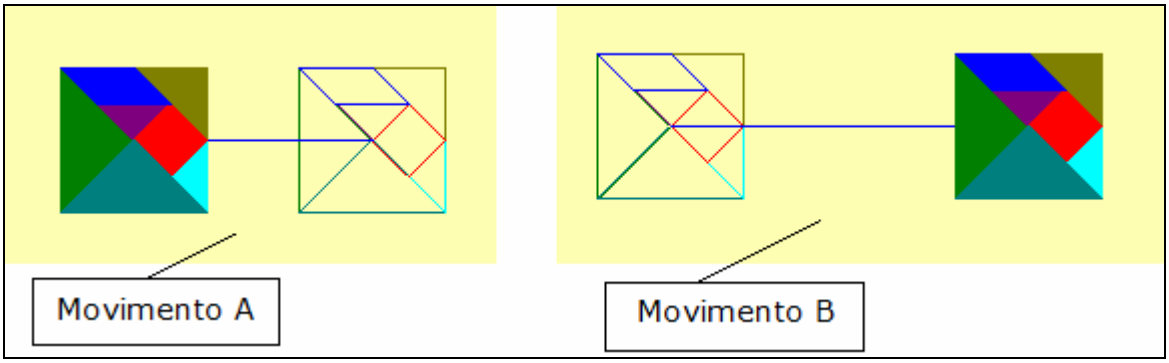


Figura 51 – Duas interações com o editor gráfico

Ao optar pelo `modo de geração de comando` como `Estático` no `Movimento A`, realizado na figura 51, é gerado o código `meuModelo.move(-1639, 0, 0)`. Alterando o `modo de geração de comandos` para `Dinâmico`, ao realizar o `Movimento B` é aberta a tela da figura 52.

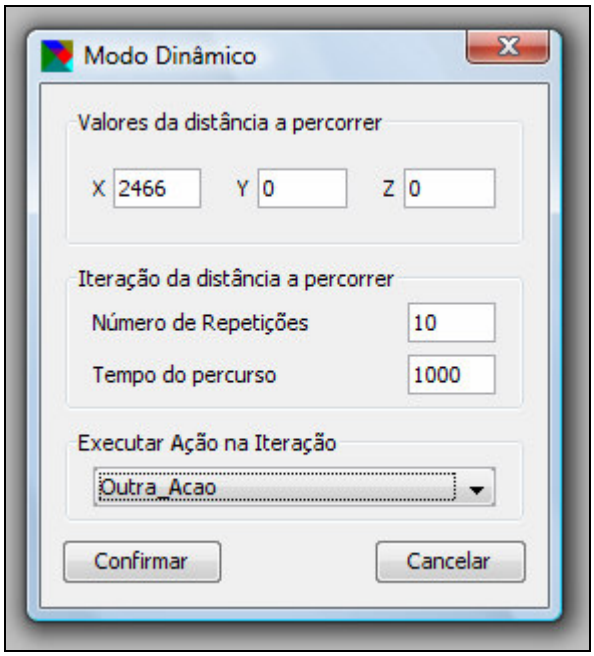


Figura 52 – Tela do modo `Dinâmico` para o `Movimento B` da figura 51

A partir dos parâmetros definidos na tela da figura 52 é gerado o código do quadro 53.

```
repita 10 vezes inicio
    meuModelo.move(247, 0, 0)
    faça Outra_Acao
    pisca(100)
fim
```

Quadro 53 – Comandos gerados pelo `Movimento B` da figura 51

3.3.2.2 Editor de Mundos

Ao criar um novo mundo, ou carregar um, é aberto o editor de mundos. Este editor disponibiliza a tela do editor textual do mundo (Figura 53), além do editor gráfico.

No editor textual do mundo é possível visualizar as ações de um modelo criado no mundo. Para isso basta selecionar o modelo desejado no editor gráfico.

O botão `faça`, figura 53, cria um comando `faça` para a ação do modelo selecionado e o botão `>` executa os comandos do mundo em uma nova janela.

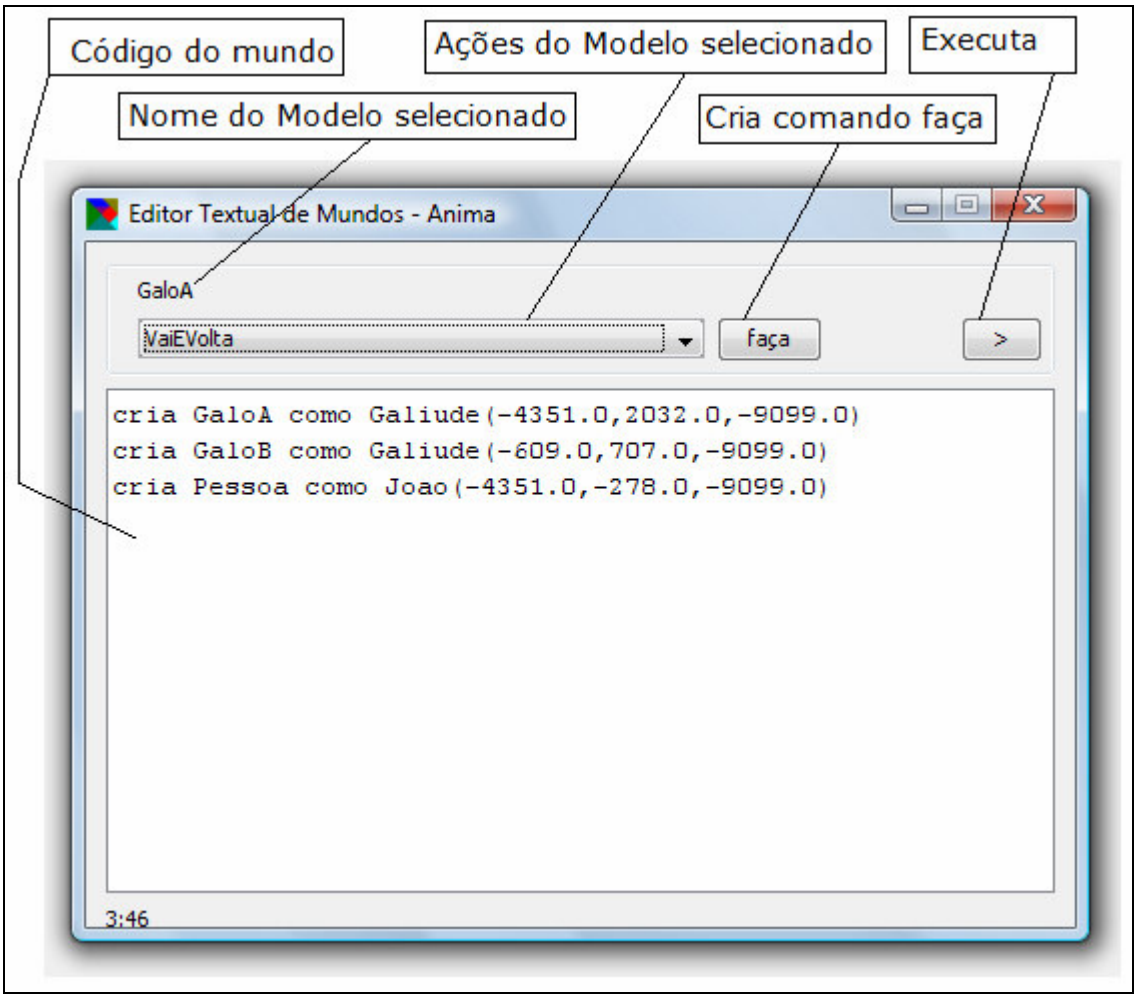


Figura 53 – Tela do editor textual do mundo

Para adicionar um modelo ao mundo, seleciona-se o modo de interação `Inserir Modelo` (Figura 54). A cada clique com o botão esquerdo do *mouse* na área de desenho é aberta uma tela para escolher um modelo já criado.

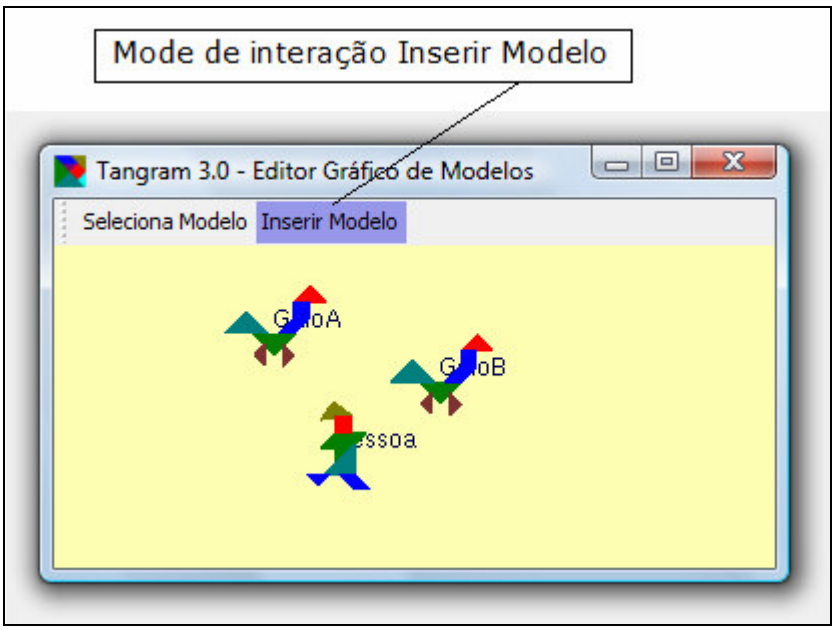


Figura 54 – Modo de interação `Inserir Modelo`

3.4 RESULTADOS E DISCUSSÃO

Nesta terceira versão do LTD é possível criar TAD, representado por modelos. Cada modelo é formado por uma figura (sete peças) do Tangram e possui métodos. Cada método possui comandos que interagem com as peças da figura. Também é possível criar mundos, onde são instanciados objetos a partir dos modelos. Em um mundo, os vários objetos instanciados podem ter seus métodos executados simultaneamente, categorizando o uso de processos concorrentes.

Esta ferramenta e os ambientes Khoros e Simulink são voltados à programação visual, no entanto o LTD difere-se porque é voltado para o ensino de programação a crianças alfabetizadas.

O LTD difere-se do Logo na forma de desenhar. Enquanto que no Logo são utilizados comandos para realizar desenhos, no LTD os comandos são utilizados para interagir com as peças do Tangram, as quais formam os desenhos.

Esta versão do LTD diferencia-se do Mundo dos Atores, pois, mesmo baseado em TAD, o LTD não utiliza variáveis. Ainda, igualmente ao Logo, o Mundo dos Atores utiliza comandos para realizar desenhos.

Nas versões anteriores não era possível selecionar, mover e visualizar o movimento de uma figura inteira, o que é possível nesta versão (Figura 51). Apenas visualizava-se o movimento de uma peça, perguntando depois se deseja mover a peça ou a figura (Figura 55).

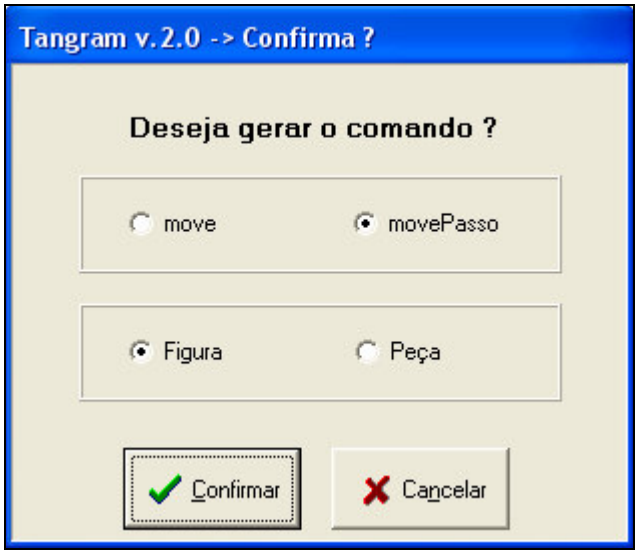


Figura 55 – Tangram 2.0, seleciona se deseja mover figura ou peça

Mesmo selecionando que deseja mover a figura, as outras peças não eram movimentadas (Figura 56). Ainda, a posição anterior da peça movimentada não era

visualizada como mostrado na figura 47.

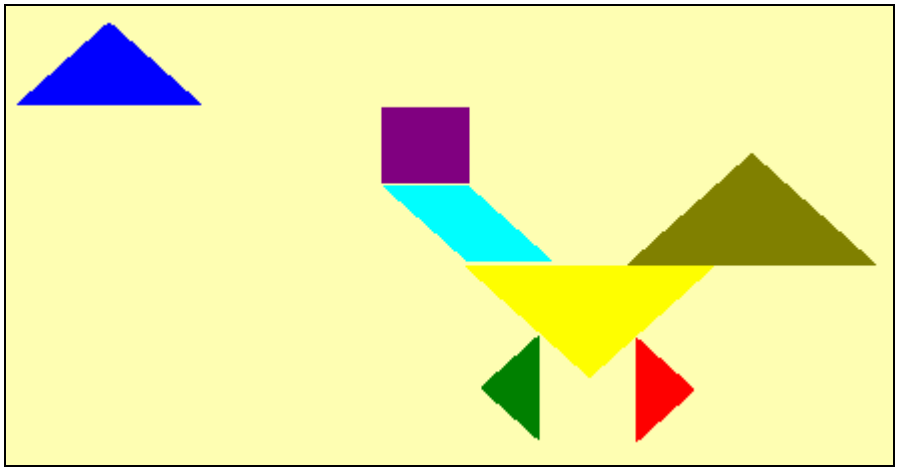


Figura 56 – Movida uma peça e selecionado para mover a figura

Para criar várias figuras iguais, na versão anterior, era necessário desenhar uma por uma. Na nova versão é desenvolvido um modelo e a partir deste, figuras são criadas, em um mundo, quantas vezes forem desejadas.

Nas versões anteriores do LTD não existia uma função para gerar vários comandos a partir de uma única interação com a interface gráfica como apresentado na figura 52 e quadro 54. Isso eleva o conceito de programação visual da nova versão.

4 CONCLUSÕES

Os objetivos de criar uma linguagem baseada em TAD para o LTD e incluir conceitos de processos concorrentes foram alcançados. Foram criados novos comandos para dar suporte a TAD e concorrência. As mesmas funcionalidades da versão anterior do LTD foram mantidas, as quais são mover, rotacionar, espelhar e mudar a cor de uma peça ou figura, além de mudar a cor de fundo e mover a câmera. Nesta implementação foi incluído em editor de modelos, os quais representam os TAD, e um editor de mundos. No editor de modelos são criados desenhos a partir das peças do Tangram, possibilitando a criação de métodos, onde os comandos são encapsulados. No editor de mundo cada modelo pode ser instanciado várias vezes e os métodos das instâncias podem ser executados simultaneamente, o que caracteriza concorrência.

Novas funcionalidades foram incluídas, como a criação de vários comandos com uma interação ao editor gráfico, o *pan* e a escolha do ponto de rotação a partir de um clique no ponto de uma peça.

O problema da versão 2.0 do LTD, onde a tela pisca freqüentemente durante uma operação de movimentação das peças, foi resolvido utilizando a função `setDoubleBuffered(true)` da classe `GLCapabilities` e trocando a função `glFlush()` por `glSwapBuffer()` no método que desenha o *canvas*. O problema de distorsão das peças quando redimensionada a janela do editor gráfico também foi resolvido.

A ferramenta foi redesenvolvida na linguagem Java, utilizando a biblioteca JOGL para os recursos gráficos. O ambiente de desenvolvimento utilizado foi o Netbeans 6.0 da empresa Sun Microsystems, com auxílio da ferramenta GALS para geração dos analisadores léxico e sintático, os quais são utilizados para interpretar os comandos da linguagem LTD. As ferramentas utilizadas para o desenvolvimento mostraram-se adequadas.

Uma atualização da ajuda (*help*) é necessário, visto que a ferramenta possui novas telas, funções e comandos da linguagem LTD, o que torna obsoleto o manual da segunda versão do LTD.

A interface gráfica onde é mostrada uma árvore com todas as informações do desenho não foi implementada.

Outra limitação existente é a falta de um mecanismo para o encaixe automático das peças quando se movimenta uma para perto da outra.

Ainda, verifica-se que quando a posição da câmera é retirada da origem, ocorre um

erro no cálculo da desprojeção do ponto do cursor na tela para o ponto no mundo tridimensional, o que faz com que as peças não possam ser selecionadas.

Observa-se também que o ambiente ainda não está apto para servir como ferramenta para crianças, visto que é necessário realizar um número maior de testes para verificar a confiabilidade da ferramenta. Complementações são necessárias para uma melhor interação dos usuários com a ferramenta. O tempo empregado nesta versão foi voltado para aspectos de definição do ambiente e aspectos técnicos para criação do mesmo.

4.1 EXTENSÕES

Como extensões para esta ferramenta, sugere-se:

a) criar uma função para o auto-encaixe quando uma peça é movida para perto da outra;
b) tratar colisões, não deixando uma peça ocupar o mesmo espaço que outra;
c) disponibilizar o LTD na *web*, via *browser*;
d) atualizar a ajuda (*help*) da nova versão;
e) acrescentar informações do desenho em uma janela especifica, como na segunda versão.

REFERÊNCIAS BIBLIOGRÁFICAS

ALCÂNTARA JR, O. **Protótipo de uma linguagem de programação de computadores orientada por formas geométricas, voltada ao ensino de programação**. 2003. 58 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

BARANAUSKAS, M. C. C. Procedimento, função, objeto ou lógica? Linguagens de programação vistas pelos seus paradigmas. In: VALENTE, J. A. (Org.). **Computadores e conhecimento:** repensando a educação. Campinas: Unicamp, 1993. Cap. 3. Disponível em: <http://www.nied.unicamp.br/publicacoes/separatas/Sep3.pdf >. Acesso em: 20 out. 2007.

DAVISON, A. **Pro Java 6 3D game development**: Java 3D, JOGL, JInput, and JOAL APIs. New York, NY: Apress, 2007.

DEITEL, H. M.; DEITEL, P. J. **Java:** como programar. 4. ed. Tradução Carlos Arthur Lang Lisboa. Porto Alegre: Bookman, 2002.

GESSER. C. E. **GALS**: gerador de analisador léxico e sintático. [S.l.], 2003. Disponível em: <http://gals.sourceforge.net/>. Acesso em: 11 jan. 2008.

GUDWIN, R. R. **Linguagens de programação**. [Campinas], 1997. Notas de aula. Disponível em: <ftp://ftp.dca.fee.unicamp.br/pub/docs/ea877/lingpro.ps.gz>. Acesso em: 20 out. 2007.

GUEZZI, C.; JARAYERI, M. **Conceitos de linguagens de programação**. Tradução Paulo A. S. Veloso. Rio de Janeiro: Campus. 1982.

KONG, A. M. **Pititi**. [S.l.], 2003. Disponível em: <http://www.pititi.com/jogos/tangram/tangram.htm>. Acesso em: 29 mar. 2006.

LENGYEL, E. **Mathematics for 3D game programming and computer graphics.** 2nd ed. Hingham: Charles River Media, 2004.

LONGHI, D. **China on-line:** conectando você com a cultura chinesa. Caxias do Sul, 2004. Disponível em: <http://www.chinaonline.com.br/artes_gerais/tangram/default.asp>. Acesso em: 30 mar. 2006.

MARIANI, A. C. **O mundo dos atores**: uma perspectiva de introdução à programação orientada a objetos. Florianópolis, 1998. Disponível em: <http://www.inf.ufsc.br/poo/atores/sbie98/sbie98-atores.html>. Acesso em: 20 out. 2007.

MATHWORKS. **Simulink**: introduction and key features. [S.l.], 2008. Disponível em: <http://www.mathworks.com/products/simulink/description1.html>. Acesso em: 20 maio 2008.

RANGEL, J. L. **Linguagens de programação**. [Rio de Janeiro], 1993. Notas de aula. Disponível em: < http://www-di.inf.puc-rio.br/~rangel/lp.html >. Acesso em: 20 out. 2007.

SANTOS, R. **Khoros programming tutorial**. [S.l.], 1997. Disponível em: <http://www.cab.u-szeged.hu/local/doc/khoros/Tutorial/index.html>. Acesso em: 30 abr. 2008.

SEBESTA, R. W. **Conceitos de linguagens de programação.** 4. ed. Tradução José Carlos Barbosa dos Santos. Porto Alegre: Bookman, 2000.

SERAFIM, T. **Racha cuca**. [S.l.], 2008. Disponível em: <http://rachacuca.com.br/tangram/>. Acesso em: 13 maio 2008.

SILVA, J. R. V.; MARTINS, J.; ALCÂNTARA JR, O. Linguagem orientada por formas geométricas, voltada ao ensino de programação. In: CONGRESSO IBEROAMERICANO DE INFORMÁTICA EDUCATIVA, 7., 2004, Monterrey, México. **Anais**... Monterrey, 2004. p. 1176-1186.

SOWIRZAL, H.; NADEAU, D.; BAILEY, M. **Introduction to programming with Java 3D.** [S.l.], 1998. Disponível em: <http://www.sdsc.edu/~nadeau/Courses/SDSCjava3d/>. Acesso em: 20 out. 2007.

SUN. **Javadoc tool**. [S.l.], 2008. Disponível em: < http://java.sun.com/j2se/javadoc/>. Acesso em: 10 jun. 2008.

THEISS, F. J. **Linguagem visual orientada por formas geométricas, voltada ao ensino de programação**. 2006. 82 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

TOSTES, R. B. O. **Estudo do ambiente Java no contexto de desenvolvimento de jogos**. 2006. 109 f. Projeto Orientado de Conclusão de Curso (Curso de Sistemas de Informação) – Centro de Ciências Exatas e Tecnológicas, Universidade Estadual de Montes Claros, Montes Claros.

VALENTE, J. A. **Liberando a mente**: computadores na educação especial. Campinas: Gráfica Central da Unicamp, 1991.

VAREJÃO, F. **Linguagens de programação**: Java, C, C++ e outras: conceitos e técnicas. Rio de Janeiro: Campus, 2004.

ZACHARIAS, V. L. C. **A linguagem Logo.** São Paulo, 2007. Disponível em: <http://www.centrorefeducacional.pro.br/linlogo.html>. Acesso em: 20 out. 2007.

APÊNDICE A – Significado das ações Semânticas

Os significados das ações semânticas apresentadas no quadro 9 são explicados no quadro 54.

Ação	Significado
#0	Inicia a compilação de um modelo.
#1	Inicia a compilação de um mundo.
#2	Retira o comando laço da pilha de comandos laço e cria um modelo com apenas o método cria.
#3	Guarda o nome do modelo na variável `nomeDoModelo`. Se o nome antigo não for nulo e o novo nome for diferente dispara uma exceção.
#4	Guarda nome do método na variável `nomeDoMetodo`.
#5	Guarda nome dado ao modelo no mundo na variável `idDoModelo`.
#6	Guarda nome do mundo na variável `nomeDoMundo`.
#7	Guarda peça na variável `idDaPeca`.
#8	Guarda a cor na variável `nomeDaCor`.
#9	Guarda o valor na variável `x`.
#10	Guarda o valor na variável `y`.
#11	Guarda o valor na variável `z`.
#12	Cria comando `laço` e o coloca no topo da pilha.
#13	As seguintes restrições devem ser atendidas: o método em que o comando irá ser inserido deve ser o método `CRIA` e a peça não deve estar criada. Se atender as restrições, então cria-se o comando `cria` (para a peça), marcando a peça como criada. Marca a variável `comandoEmModelo` igual a falso.
#14	Adiciona comando criado ao laço do topo da pilha de comandos laço e zera variáveis `x`, `y` e `z`.
#15	Marca a variável `comandoEmModelo` igual a verdadeiro.
#16	Marca a variável `comandoEmModelo` igual a falso e verifica se a peça foi criada.
#17	Cria comando `move`.
#18	Cria comando `gira`.
#20	Cria comando `cor`.
#21	Cria comando `espelha`.

#22	Cria comando `pisca`.
#23	Cria comando `laço` e coloca-o no topo da pilha de comandos laço.
#24	Retira o comando `laço` do topo da pilha de comandos laço e adiciona-o no novo topo da pilha.
#25	Cria o comando `faça` e coloca o nome do método na lista de métodos usados.
#26	Retira o comando `laço` do topo da pilha e coloca-o na lista de métodos do modelo.
#27	Verifica se os identificadores da lista de métodos usados foram todos definidos como um método do modelo. Se algum método foi usado sem ser definido, retorna um erro. Cria o método `VIVA` para o modelo e o finaliza criando um objeto da classe `ModeloExecutavel`.
#28	Verifica se o `id` dado ao modelo já foi usado. Verifica na lista de modelos compilados se o modelo já foi compilado. Se não foi, compila-o e o coloca na lista de modelos compilados. Cria novo `ModeloExecutavel` com uma nova `Figura` para o `id` dado ao modelo e adiciona na lista de `id` do mundo. Cria o comando `faça-no-mundo`, o qual chama o método `CRIA` do `id` dado ao modelo.
#29	Cria o comando `move-para`, o qual move o modelo para o ponto inicial.
#30	Verifica se o `id` dado ao modelo foi criado. Caso foi criado, verifica se o modelo não está executando o comando `VIVA` e não foi executado o comando `APAGA`. Caso atenda as verificações, então cria o comando `VIVA`.
#31	Verifica se o `id` dado ao modelo foi criado. Caso foi criado, verifica se o modelo está executando o comando `VIVA`. Caso não esteja executando, cria comando `faça-no-mundo` chamando o método `TERMINA` do `id` dado ao modelo.
#32	Verifica se o `id` dado ao modelo foi criado. Cria o comando `APAGA`.
#33	Limpa as variáveis `x`, `y`, `z`, `idDoModelo` e `emParalelo`.
#34	Verifica se o `id` dado ao modelo foi criado. Cria-se o comando `faça-no-mundo` para o modelo do `id`, chamando o método da variável `nomeDoMetodo`.
#35	Marca variável `emParalelo` como verdadeiro.

Quadro 54 – Significado das ações semânticas