Critérios de teste baseados em grafo de cena para aplicações de realidade virtual

Adriano Bezerra

Critérios de teste baseados em grafo de cena para aplicações de realidade virtual

Adriano Bezerra

Orientador: **Prof. Dr. Márcio Eduardo Delamaro**

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação - ICMC-USP, como parte dos requisitos para obtenção do título de Mestre em Ciências - Ciências de Computação e Matemática Computacional. *VERSÃO REVISADA*

USP – São Carlos
Março de 2012

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados fornecidos pelo(a) autor(a)

Bezerra, Adriano
B574c Critérios de teste baseados em grafo de cena para aplicações de realidade virtual / Adriano Bezerra; orientador Márcio Eduardo Delamaro. -- São Carlos, 2012.
106 p.

Dissertação (Mestrado - Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional) -- Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2012.

1. Teste de Software. 2. Critérios de Teste de Software. 3. Realidade Virtual. 4. Grafos de Cena. I. Delamaro, Márcio Eduardo, orient. II. Título.

Dedico este trabalho a toda minha família, em especial ao meu avô José Bezerra Neto que aos 79 anos terminou sua missão na Terra dando-me exemplo de luta, garra, determinação, fé e perseverança.

Agradecimentos

Primeiramente agradeço ao grandioso Deus por ter me dado forças e condições para realizar este trabalho.

Agradeço aos meus pais pela confiança, incentivo, paciência e o investimento dedicado em mais uma etapa da minha vida.

Ao meu orientador Prof. Dr. Márcio Eduardo Delamaro, pela amizade e a excelente orientação, aliados a experiência intelectual e profissional. Obrigado pelas horas de orientação e conselhos. Agradeço-lhe por te confiado em mim e aceitado me orientar nesse projeto.

À minha coorientadora Profa. Dra. Fátima de Lourdes dos Santos Nunes Marques por ter me incentivado sempre para realizar esse trabalho, e por ser um exemplo de honestidade e profissionalismo. Uma pessoa inesquecível que merece todo o meu respeito.

As minhas irmãs Lidiane Bezerra e Juliana Bezerra pela paciência, amor e amizade.

A todos meus amigos e todos os integrantes dos laboratórios de pesquisa da Universidade de São Paulo (USP), em especial à galera do Laboratório de Engenharia de Software (LabES) que me ajudaram para que esse meu sonho se concretizasse.

A todos os integrantes do Laboratório de Aplicações de Informática em Saúde (LApIS) pela ajuda, companheirismo e amizade.

Ao meu amigo Sr. Sabatine, no qual tenho um grande respeito. Um exemplo de tranquilidade, dedicação e amizade.

À FAPESP (Fundação de Amparo à Pesquisa do Estado de São Paulo), processo (2009/03803-1), e ao Instituto Nacional de Ciência e Tecnologia – Medicina Assistida por Computação Científica (INCT–MACC), processo (573710/2008-2 Edital MCT/CNPq Nº 015/2008 – Institutos Nacionais de Ciência e Tecnologia), pelo apoio financeiro

”A maior recompensa do nosso trabalho
não é o que nos pagam por ele, mas
aquilo em que ele nos transforma.”

John Ruskin

”A palavra impossível só existe
no dicionário dos perdedores”

Napoleão

Resumo

A atividade de teste de software tem recebido considerável atenção de pesquisadores e engenheiros de software que reconhecem a sua utilidade na criação de produtos de qualidade. No entanto, os testes são caros e propensos a erros, o que impõe a necessidade de sistematizar e, portanto, a definição de técnicas para aumentar a qualidade e produtividade na sua condução. Várias técnicas de teste têm sido desenvolvidas e têm sido utilizadas, cada um com características próprias em termos de eficácia, custo, fases de aplicação, etc. Sistemas de realidade virtual frequentemente utilizam uma estrutura hierárquica denominada grafo de cena para representar as características dos objetos em um ambiente virtual tridimensional. Os grafos de cena também armazenam informações sobre o relacionamento entre os objetos, permitindo respostas adequadas ao usuário quando ocorrem interações. Neste trabalho, critérios de teste baseados em grafo de cena são estudados e definidos a fim de aumentar a qualidade de aplicações de realidade virtual. Além disso, estudos de caso são apresentados, utilizando os critérios definidos aplicados a um *framework* de realidade virtual construído para gerar aplicações na área médica, além de utilizar uma aplicação de demonstração. Como forma de apoio aos critérios definidos foi desenvolvida uma ferramenta de teste capaz de verificar se os nós, que representam os objetos virtuais na cena, satisfazem seus requisitos conforme foram especificados.

Abstract

The activity of software testing has received considerable attention from researchers and software engineers who recognize its usefulness in creating quality products. However, the tests are expensive and prone to errors, which imposes the need to systematize and hence the definition of techniques to increase quality and productivity in their driving. Several testing techniques have been developed and have been used, each with its own characteristics in terms of effectiveness, cost, implementation stages, etc. Moreover, these techniques can also be adapted. In this work, test criteria based on scene graph are studied and defined in order to increase the quality of the Virtual Reality software. In addition, case studies are presented, using the criteria applied to a framework built to generate virtual reality applications in medicine, in addition to using a demo application. As a form of support to the criteria a testing tool was developed. It verifies whether the nodes that represent the virtual objects meet their requirements as they were specified.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Listagens

Lista de Abreviaturas e Siglas

GC	*Grafos de Cena*
RV	*Realidade Virtual*
AV	*Ambiente Virtual*
RA	*Realidade Aumentada*
OO	*Orientação a Objetos*
NIST	*National Institute of Standards and Technology*
TADs	*Tipos Abstratos de Dados*
MEF	*Máquina de Estados Finitos*
API	*Application Programming Interface*
LOD	*Level of Detail*
2D	*Duas Dimensões*
3D	*Três Dimensões*
INCT	*Instituto Nacional de Ciência e Tecnologia*
MACC	*Medicina Assistida por Computação Científica*
ViMeT	*Virtual Medical Training*
VETesting	*Virtual Environment Testing*

Lista de Publicações

- BEZERRA, A.; DELAMARO, M. E.; NUNES, F. L. S. Aplicações baseadas em Grafo de Cena uma abordagem estrutural para critérios de teste. In: VII Workshop de Realidade Virtual e Aumentada - WRVA, 2010, São Paulo – SP – Brasil. Anais do VII Workshop de Realidade Virtual e Aumentada - WRVA 2010. São Paulo, Editora Mackenzie, 2010.

- BEZERRA, A.; DELAMARO, M. E.; NUNES, F. L. S. Definition of test criteria based on the Scene Graph for VR applications. In: *XIII Symposium on Virtual and Augmented Reality* (SVR), 2011, Uberlândia – MG – Brasil. Anais do XIII *Symposium on Virtual and Augmented Reality* (SVR). Editora Sociedade Brasileira de Computação, 2011.

- BEZERRA, A.; DELAMARO, M. E.; NUNES, F. L. S. Critérios de teste baseados em grafo de cena para aplicações de Realidade Virtual. IX Workshop de Teses e Dissertações em Qualidade de Software (WTDQS). X Simpósio Brasileiro de Qualidade de Software (SBQS) 2011. Curitiba – PR – Brasil, Brasil.

Capítulo

1

Introdução

1.1 Contextualização

A atividade de teste de software é considerada fundamental no contexto da Engenharia de Software (Bertolino, 2007). Usualmente, 50% do tempo e do custo de desenvolvimento de um sistema são empregados em atividades de teste e correção de problemas (Naik e Tripathy, 2008). Um defeito encontrado em fase de produção pode custar à empresa cem vezes ou mais o valor de correção em relação à fase de requisitos, ou seja, quanto mais tarde encontrado um erro, mais caro é o custo para corrigi-lo (Perry, 2006). O *National Institute of Standards and Technology* (NIST) estima que, nos Estados Unidos, durante o ano 2000, o prejuízo associado à insuficiência na realização das atividades de teste de software foi de aproximadamente 59 bilhões de dólares (Blackburn et al., 2003).

Desde 1975, quando Goodenough e Gerhart (1975) publicaram o primeiro artigo para formalizar os conceitos de teste de software, houve diversos avanços na área, com a criação de técnicas e critérios que objetivam o desenvolvimento de software confiável. A realização de atividades de teste, porém, não pode garantir a ausência de erros. Na maioria dos casos é impraticável a utilização de todo o domínio possível de dados de entrada para avaliar as características funcionais e operacionais do programa que está sendo testado. Assim, de acordo com Myers (2004), é importante que se foque na criação de um subconjunto de casos de teste eficaz, ou seja, aquele que possua alta probabilidade de detectar o maior número possível de erros.

Diversas técnicas e critérios de testes foram definidas e cada uma tem particularidades em termos de custo, aplicabilidade e efetividade. Ntafos (1988), por intermédio de análise de inclusão, observou que diferentes erros podem ser encontrados com o uso de diferentes critérios de teste. A escolha dos critérios, além de influenciar a qualidade do teste, também produz efeito sobre o tempo de planejamento e execução.

Uma base teórica sólida foi construída por pesquisadores ao longo dos anos abordando o uso de diversas técnicas e critérios de teste. Alguns trabalhos exploram características específicas de domínios de aplicação e podem requerer a adaptação dessas técnicas e critérios de teste ou mesmo a criação de técnicas ou critérios de teste específicos para tais domínios. Em Spoto et al. (2000) é proposta uma técnica para identificar definição e utilização de variáveis persistentes em aplicações de banco de dados relacional e foi definido um conjunto de critérios estruturais de teste de unidade e de integração. Em Lemos et al. (2007) são definidos critérios estruturais para lidar com características específicas de programas orientados a aspecto. Delamaro et al. (2007b) discutem como usar as características da análise de bytecode Java e como estendê-la para a implementação de critérios de teste estruturais para dois domínios específicos: programas orientados a aspectos e aplicações de banco de dados. Um possível domínio de aplicação a ser explorado são os sistemas de Realidade Virtual (RV).

1.2 Motivação

Visto a importância da atividade de teste no desenvolvimento de software, tais atividades deveriam estar presentes em todos os tipos de projetos de software. Todavia, dentro do contexto das pesquisas científicas em ciência da computação, esta prática não é comum (Tichy et al., 1995). A fim de verificar como os trabalhos descritos nos artigos são validados, uma pesquisa foi realizada por Zelkowitz e Wallace (Zelkowitz e Wallace, 1998). A pesquisa relatou que metades dos trabalhos publicados no ano de 1998 tinham um nível de avaliação insuficiente. Em 2009, Wainer et al. (Wainer et al., 2009) retomaram a análise quantitativa realizada por Tichy et al. (Tichy et al., 1995) em 1993, avaliando 147 artigos publicados em 2005. Nesta análise, os pesquisadores concluíram que 33% das propostas de projetos ou modelagens no contexto pesquisado não apresentaram nenhum tipo de avaliação.

No desenvolvimento de sistemas não convencionais como os que utilizam a *Realidade Virtual* (RV) necessita-se explorar com maior intensidade suas atividades de teste e avaliação, sendo de vital importância, principalmente quando se trata de aplicações para treinamento de procedimentos ou habilidades. Uma análise exploratória realizada por Nunes et al. (Nunes et al., 2010), visou a identificar o tipo de avaliação realizado em

aplicações de RV. Dos artigos produzidos nas edições de 2008 e 2009 do *Symposium on Virtual and Augmented Reality* (SVR), mais de 55% dos artigos não contemplavam algum tipo de teste ou avaliação. Embora o estudo não tenha a pretensão de fornecer uma análise aprofundada na questão, há fortes indícios que comprovam a necessidade de testes e avaliações em tais sistemas.

A RV pode ser considerada como a junção de três ideias básicas: imersão, interação e envolvimento. Essas ideias não são exclusivas de RV, mas nela coexistem (Morie, 1994). O crescente poder computacional faz com que cenas de RV cada vez mais complexas e mais realistas possam ser geradas e visualizadas em tempo real com o uso de simples computadores. Essa complexidade também se aplica ao desenvolvedor da cena, que precisa especificar e modelar várias características.

De um modo geral, tal complexidade implica na exigência de um modelo ou uma estrutura de dados capaz de organizar os elementos na cena e disponibilizá-los ao usuário de forma rápida e eficiente. Para isso, existem os grafos de cena, que são estruturas de dados organizadas em classes, nas quais, por meio de hierarquia de objetos e atributos, podem-se facilmente especificar cenas complexas. Cada objeto ou atributo é representado por um nó, que possui informações sobre sua aparência física, dentre outras características (Woo et al., 1999).

Assim, é possível ver um grafo de cena como um modelo e uma abstração de um programa de RV e, portanto, pode-se pensar na utilização de critérios de teste que utilizem esse modelo para derivar requisitos de teste. Semelhante a critérios estruturais, é possível utilizar o Grafo de Cena (GC) para selecionar estruturas a serem exercitadas durante a atividade de teste.

1.3 Objetivos

Este trabalho contribui com as pesquisas relacionadas a definições de critérios de teste estruturais, dando continuidade aos trabalhos desenvolvidos pelo grupo do Laboratório de Engenharia de Software do ICMC/USP (LabES)[1] e do grupo de pesquisa em Realidade Virtual do Laboratório de Aplicações de Informática em Saúde (LApIS) [2] da EACH/USP.

Dentro desse contexto, o objetivo deste trabalho é definir critérios de teste de software estruturais baseados em grafos de cena a fim de serem utilizados nas atividades de testes aplicadas ao desenvolvimento de sistemas de realidade virtual.

A contribuição pretendida dar-se-á, por meio de um estudo direcionado dos grafos de cena e os modos de representação de ambientes virtuais. Além disso, pretende-se

[1] http://www.labes.icmc.usp.br/site/
[2] http://www.each.usp.br/lapis/

automatizar esses critérios por meio de uma ferramenta de teste e, por fim, realizar três estudos de caso. Dois com um *framework* de RV desenvolvido no LApIS, denominado *Virtual Medical Training* (ViMeT) e um com uma classe de demonstração da *Application Programming Interface* API Java3D.

1.4 Organização

Esta dissertação possui, além desta introdução, cinco capítulos, a saber:

Capítulo 2 - é feita uma contextualização sobre teste de software, técnicas e critérios de teste de software.

Capítulo 3 - são apresentados conceitos de Realidade Virtual e Grafos de Cena.

Capítulo 4 - é destacada a definição dos critérios de teste baseados em Grafos de Cena bem como a automatização por meio de uma ferramenta de teste.

Capítulo 5 - são apresentados os estudos de caso realizados e seus resultados.

Capítulo 6 - são apresentadas as conclusões, destacando as contribuições do trabalho e os possíveis trabalhos futuros.

Capítulo

2

Teste de software

A fase de teste é uma atividade de grande importância no desenvolvimento de software, contribuindo para a diminuição do número de defeitos e aumentando a confiança nos produtos. Neste capítulo, é apresentada uma revisão bibliográfica dos conceitos relacionados à atividade de teste de software. As principais técnicas e critérios de teste, inicialmente propostos para programas procedimentais, foram investigadas no contexto de software desenvolvido sob outras abordagens, por exemplo, a orientação a objetos (OO). Assim, faz-se necessária a apresentação dos conceitos básicos e das técnicas tradicionais para que as abordagens de teste de software possam ser adequadamente apresentadas nos capítulos posteriores deste trabalho.

Na Seção 2.1 são apresentados os principais conceitos e nomenclatura relacionados ao teste de software. Na Seção 2.2 são apresentadas técnicas e critérios de teste contidos na literatura e, por fim, na Seção 2.3, são apresentadas as considerações finais deste capítulo.

2.1 Conceitos básicos e nomenclatura

A construção de um software não é uma tarefa trivial. Pelo contrário, pode tornar-se bastante complexa, dependendo do sistema a ser criado. Assim, podem aparecer diversos problemas em sua construção que acarretam a obtenção de um produto diferente daquele que se esperava (Delamaro et al., 2007a).

O teste de software é uma das atividades mais caras e mais importantes no processo de desenvolvimento de software. Por esse motivo, e pela aceitação entre os pesquisadores e engenheiros de que o conceito de qualidade é um fator essencial no desenvolvimento de software, muito se tem investido na pesquisa na área de teste de software (Beizer., 1990; Myers, 2004; Pressman, 1997).

Diversos problemas podem emergir durante o desenvolvimento de software. É importante ressaltar que esses problemas são divididos e diferenciados entre os termos defeito, engano, erro e falha. Segundo o padrão IEEE 610.12-1990 (IEEE, 1990), defeito (*fault*) é um passo, processo ou definição de dados incorretos; engano (*mistake*) é uma ação humana que produz um resultado incorreto; erro (*error*) é a diferença entre o valor obtido e o valor esperado e falha (*failure*) consiste na produção de uma saída incorreta com relação à especificação. A execução de um programa P, com um caso de teste t pode ser responsável por "revelar um defeito" ou mostrar uma falha e não "encontrar um defeito". Essa última afirmação caracteriza a etapa de depuração.

Define-se "dado de teste" de um programa **P**, um elemento do domínio de entrada de **P**. Domínio de entrada pode ser entendido como possíveis valores que um parâmetro pode ter (Ammann e Offutt, 2008). Um par formado por um dado de teste mais o resultado esperado para a execução do programa com aquele dado de teste é denominado "caso de teste" (Delamaro et al., 2007b). Por exemplo, no programa que computa x^y, teríamos os seguintes casos de teste: $\langle(2,2),4\rangle$, $\langle(2,3),8\rangle$, $\langle(4,-1),\text{“}Erro\text{”}\rangle$. A um conjunto de casos de teste usados durante uma determinada atividade de teste costuma-se chamar "conjunto de teste" ou "conjunto de casos de teste".

O processo de desenvolvimento de software envolve diversas de atividades nas quais, apesar das técnicas, métodos e ferramentas empregadas, erros no produto de software ainda podem ocorrer. A atividade de teste consiste em uma atividade dinâmica do produto de software, ou seja, na execução do produto de software com o objetivo de verificar a presença de defeitos e aumentar a confiança de que o mesmo esteja correto, representando a última revisão da especificação, projeto e codificação (Harrold, 2000; Pressman, 1997).

Segundo (Myers, 2004), o principal objetivo do teste de software é revelar erros e defeitos no produto. Assim, um teste bem sucedido é aquele que consegue determinar casos de teste para os quais o programa que está sendo testado falhe.

A atividade de teste envolve quatro etapas: planejamento de testes, projeto de casos de teste, execução e avaliação dos resultados de teste (Pressman, 2006), que podem ser subdivididas em três fases:

- Teste de unidade: consiste em testar a menor unidade de projeto de software que pode ser um procedimento, um método ou uma classe. Nessa fase, procura-se identificar defeitos na lógica e na implementação de cada módulo do programa.

- Teste de integração: nessa fase procura-se descobrir defeitos na interface entre os módulos e na integração da estrutura do programa.

- Teste de sistemas: essa fase consiste em verificar se as funções estão de acordo com a especificação, se todos os elementos combinam-se adequadamente assim como se as características de desempenho e robustez estão sendo satisfeitas.

Além dessas três fases de teste, destaca-se um tipo de teste realizado durante a manutenção do software comumente chamado de "teste de regressão". Entende-se como teste de regressão, o teste realizado após a manutenção do sistema a fim de mostrar que as modificações efetuadas estão corretas, ou seja, que os novos requisitos implementados (se for o caso) funcionam como o esperado e que os requisitos anteriormente testados continuam válidos (Delamaro et al., 2007b).

Um ponto muito importante que deve ser considerado é o projeto e/ou a avaliação de um determinado conjunto de testes. Idealmente, um programa P deveria ser testado com um conjunto de testes T formado pelo conjunto completo do domínio de entrada D. No entanto, sabe-se que isso é impraticável, pois o domínio de entrada pode ser infinito ou possuir um número muito grande de elementos. Sendo assim, por questões de produtividade e tempo, o objetivo é utilizar casos de teste que tenham alta probabilidade de revelar um defeito com o mínimo de tempo e esforço (Delamaro et al., 2007b). Para chegar nesse objetivo, diversos critérios, métodos e técnicas têm sido propostos de forma a oferecer ao testador uma abordagem sistemática e teoricamente fundamentada na condução e na avaliação da atividade de teste (Rapps e Weyuker, 1985). Esses critérios dividem o domínio de entrada de forma que o conjunto de teste seja eficiente em revelar defeitos.

2.2 Técnicas e critérios de teste

De modo geral, os critérios de teste estão agrupados em três técnicas, que se distinguem pela origem da informação necessária para derivar os requisitos de teste. São elas: funcional (ou caixa preta) - os requisitos de teste são derivados a partir da especificação; estrutural (ou caixa branca) - casos de teste são criados a partir do código do programa; e baseada em defeitos - os requisitos de teste são derivados a partir de informações sobre os defeitos mais frequentes encontrados no desenvolvimento de software. Salienta-se que essas três técnicas são vistas como complementares e o uso em conjunto proporciona maior qualidade e confiança na atividade de teste (Maldonado, 1991).

Os critérios de teste podem ser usados tanto para geração de um conjunto de casos de teste quanto para a avaliação da adequação desse conjunto, evidenciando a suficiência do teste (Frankl e Weyuker, 2000).

Uma atividade muito citada na condução e avaliação da atividade de teste é a análise de cobertura, que consiste em determinar o percentual de elementos requeridos por um dado critério de teste que foram exercitados pelo conjunto de casos de teste utilizado (Vergilio et al., 1993). A partir dessa informação, o conjunto de casos de teste pode ser melhorado, adicionando-se novos casos de teste para exercitar os elementos ainda não cobertos. Além disso, é importante ressaltar que apenas um caso de teste é possível executar vários requisitos.

Dentre as técnicas tradicionais de teste, enfatiza-se neste trabalho a técnica estrutural, pois essa técnica permite analisar o código para que sejam gerados os grafos de cena sobre os quais serão executados os testes. A técnica estrutural tem como objetivo requerer o exercício de partes elementares da implementação. Tais estruturas, como comandos, desvios ou pontos do programa onde as variáveis são utilizadas, constituem os requisitos de teste a serem satisfeitos. A porcentagem de tais requisitos, executados – ou cobertos – pelos casos de teste, representa a cobertura desse conjunto em relação ao teste estrutural. A técnica estrutural, bem como seus critérios de teste, são descritos nas próximas seções.

Para ilustrar conceitos nas próximas subseções, são usados exemplos extraídos de (Barbosa et al., 2007). Tais exemplos são baseados no programa *identifier*. O *Identifier.c* é um programa que determina a validade de um identificador em "Silly Pascal" (uma variante de Pascal). Pode-se dizer que um identificador é válido se ele começar com uma letra e possuir apenas letras ou dígitos. Além disso, o identificador deve ter no mínimo 1 caractere e no máximo 6 caracteres de comprimento. Na Figura 2.1 é mostrado o código do programa.

2.2.1 Técnica estrutural

Na técnica de teste estrutural, como dito anteriormente, aspectos de implementação são fundamentais na escolha dos casos de teste. O teste estrutural baseia-se no conhecimento da estrutura interna da implementação. Um programa pode ser decomposto em um conjunto de blocos disjuntos de comandos. A execução do primeiro comando de um bloco acarreta a execução de todos os outros comandos desse bloco, na ordem dada. Todos os comandos de um bloco, possivelmente com exceção do primeiro, têm um único predecessor e exatamente um único sucessor, exceto possivelmente o último comando. Em geral, são utilizados grafos orientados para representar o controle lógico do programa. Esses grafos contêm um único nó de entrada, nos quais cada vértice representa um bloco indivisível de comandos e as arestas indicam um possível desvio de fluxo de controle (Myers, 2004). O teste estrutural pode ser caracterizado a partir das escolhas dos elementos do *Grafo de Fluxo de Controle* (GFC) que devem ser executados (Delamaro et al., 2007a).

```
            include <stdio.h>
            main()
/* 01 */    {
/* 01 */      char achar;
/* 01 */      int length, valid_id;
/* 01 */      length = 0;
/* 01 */      valid_id = 1;
/* 01 */      printf(''Identificador: '');
/* 01 */      achar = fgetc(stdin);
/* 01 */      valid_id = valid_s(achar);
/* 01 */      if(valid_id)
/* 02 */      {
/* 02 */        length = 1;
/* 02 */      }
/* 03 */      achar = fgetc(stdin);
/* 04 */      while(achar != '\n')
/* 05 */      {
/* 05 */        if(!(valid_id(achar)))
/* 06 */        {
/* 06 */          valid_id = 0;
/* 06 */        }
/* 07 */        length++;
/* 07 */        achar = fgetc(stdin);
/* 07 */      }
/* 08 */      if(valid_id
                 && (length >= 1) && (length < 6))
/* 09 */      {
/* 09 */        printf(''Válido\n'');
/* 09 */      }
/* 10 */      else
/* 10 */      {
/* 10 */        printf(''Inválido\n'');
/* 10 */      }
/* 11 */    }
```

```
            int valid_s(char ch)
/* 01 */    {
/* 01 */      if(((ch >= 'A') &&
                  (ch <= 'Z')) ||
                 ((ch >= 'a') &&
                  (ch <= 'z')))
/* 02 */      {
/* 02 */        return(1);
/* 02 */      }
/* 03 */      else
/* 03 */      {
/* 03 */        return(0);
/* 03 */      }
/* 04 */    }


            int valid_f(char ch)
/* 01 */    {
/* 01 */      if(((ch >= 'A') &&
                  (ch <= 'Z')) ||
                 ((ch >= 'a') &&
                  (ch <= 'z')) ||
                 ((ch >= '0') &&
                  (ch <= '9')))
/* 02 */      {
/* 02 */        return(1);
/* 02 */      }
/* 03 */      else
/* 03 */      {
/* 03 */        return(0);
/* 03 */      }
/* 04 */    }
```

Figura 2.1: Programa *identifier* (adaptado de (Barbosa et al., 2007))

O GFC associa uma aresta com cada desvio possível no programa, e um nó com cada sequência de instruções (Ammann e Offutt, 2008). Como exemplo, pode-se observar o grafo de fluxo de controle do programa *identifier*, ilustrado na Figura 2.2

Na Figura 2.1, pode-se observar as linhas de código do programa *identifier* divididas em blocos. Estes blocos são correspondentes aos nós do GFC descrito na Figura 2.2. Formalmente, um bloco básico é uma sequência de instruções do programa. Se alguma instrução do bloco for executada, todo o bloco será executado (Ammann e Offutt, 2008).

Os critérios de teste estrutural baseiam-se em diferentes tipos de conceitos e elementos de programas para determinar os requisitos de teste (Delamaro et al., 2007a), classificados em: critérios baseados na complexidade, critérios baseados em fluxo de controle e critérios baseados em fluxo de dados. Tais critérios são discutidos nas seções seguintes.

2.2.2 Critérios baseados na complexidade

Os critérios baseados na complexidade utilizam informações sobre a complexidade do programa sendo testado a fim de derivar os requisitos de teste. O critério de Mccabe (1976), também conhecido como teste de caminho básico, é um dos critérios baseados em

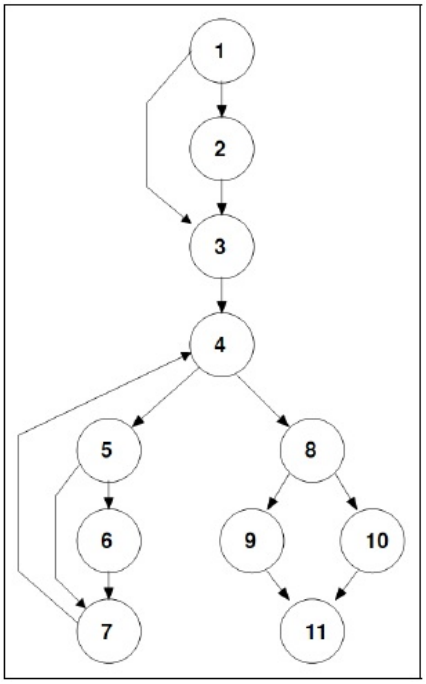

Figura 2.2: GFC do programa *identifier* (Barbosa et al., 2007).

complexidade mais difundidos. Esse critério utiliza a complexidade ciclomática do GFC a fim de derivar os requisitos de teste.

Complexidade ciclomática (V) pode ser entendida como uma métrica a fim de representar quantitativamente a complexidade lógica de um programa (Pressman, 2006).

Segundo Pressman (2006), o cálculo da complexidade ciclomática pode ser feito de três maneiras:

1. Computando o número de regiões informalmente descritas em um GFC, contando-se todas as áreas delimitadas ou não fora do grafo; ou

2. $V(G) = E$ - N + 2, tal que G é o GFC, E é o número de arestas e N é o número de nós do GFC G; ou

3. $V(G) = P$+1, tal que P é o número de nós predicativos contido no GFC G.

Em linhas gerais, a complexidade ciclomática mede quantitativamente a complexidade lógica de um programa. Utilizado no contexto do teste de caminho básico, o valor da complexidade ciclomática institui o número de caminhos linearmente independentes do conjunto básico de um programa, proporcionando um limite máximo de casos de teste que devem ser derivados de modo garantir que todas as instruções sejam executadas pelo menos uma vez (Pressman, 2006).

Caminho linearmente independente pode ser entendido como qualquer caminho do programa que introduza pelo menos uma nova condição a ser satisfeita ou um novo conjunto

de instruções de processamento. Por exemplo, um caminho linearmente independente em um GFC, deve incluir pelo menos um arco que não tenha atravessado antes que o caminho seja definido. Deste modo, cada novo caminho introduz um arco (Barbosa et al., 2007).

2.2.3 Critérios baseados em fluxo de controle

Os critérios baseados em fluxo de controle utilizam características de controle da execução do programa para derivar requisitos de teste, por exemplo, comandos ou desvios (Barbosa et al., 2007). Segundo Myers (2004) e Pressman (2006), os critérios mais conhecidos dessa técnica são:

- Todos-Nós: estabelece que cada comando do programa seja exercitado ao menos uma vez;
- Todas-Arestas: requer que cada desvio do fluxo de controle do programa seja exercitado ao menos uma vez;
- Todos-Caminhos: exige que todos os possíveis caminhos do programa sejam exercitados.

Barbosa et al. (2007) ressaltam que o mínimo esperado de uma boa atividade de teste se dá quando é atingida a cobertura do critério Todos-Nós. Contrário a isto, o programa é entregue sem a certeza de que todos os comandos presentes foram executados pelo menos uma vez. Ressalta também a impraticabilidade de se executar todos os caminhos de um programa, visto que na presença de laços o número de caminhos de um programa pode ser muito elevado ou infinito. Este aspecto motivou a introdução dos critérios baseados em fluxo de dados.

Em um grafo de cena, não se sabe se será possível cobrir todos os nós, uma vez que não há laços em sua estrutura. Sabe-se que em um ambiente muito grande haverá muitos nós no grafo e, consequentemente muitos nós a serem cobertos.

2.2.4 Critérios baseados em fluxo de dados

Os critérios baseados em fluxo de dados empregam análise de fluxo de dados para derivar os requisitos de teste. Assim, para derivação de casos de teste tais critérios baseiam-se nas associações existentes entre uma definição de variável e seus possíveis usos subsequentes. Dois exemplos de famílias de critérios de teste baseados em fluxo de dados são: a família de critérios proposta por Rapps e Weyuker (1985) e a Potenciais-Usos, proposta por (Maldonado, 1991).

A falta de eficácia para revelar defeitos simples e triviais, mesmo de programas pequenos, foi um dos motivos para a introdução dos critérios baseados em fluxo de dados a fim de estabelecer uma hierarquia entre os critérios Todas-Arestas e Todos-Caminhos, tendo em vista aumentar a severidade do teste estrutural (Delamaro et al., 2007a).

Nesse contexto Rapps e Weyuker (1985) propuseram alguns conceitos importantes, como o de "Grafo Def-Uso" (*def-use graph*), semelhante ao GFC. Como complemento, foram definidos dois conceitos importantes: o de *definição* de uma variável, que ocorre no momento em que um valor lhe é atribuído e o de *uso* de uma variável, que expressa uma referência ao valor daquela variável. Esse *uso* caracteriza-se como uso computacional (*c-uso*) ou uso predicativo (*p-uso*). No primeiro, a variável é usada em um comando de computação, no segundo, ela é usada em uma expressão de decisão.

Os caminhos a serem exercitados são definidos em função das associações *definição-uso* das variáveis do programa. A partir dessas informações é construído o grafo Def-Uso, do qual deriva-se o conceito de par *def-uso*, que se refere a uma *definição* e subsequente *uso* daquela variável (podendo ser um *c-uso* ou *p-uso*). O grafo Def-Uso do programa *identifier* está apresentado na Figura 2.3.

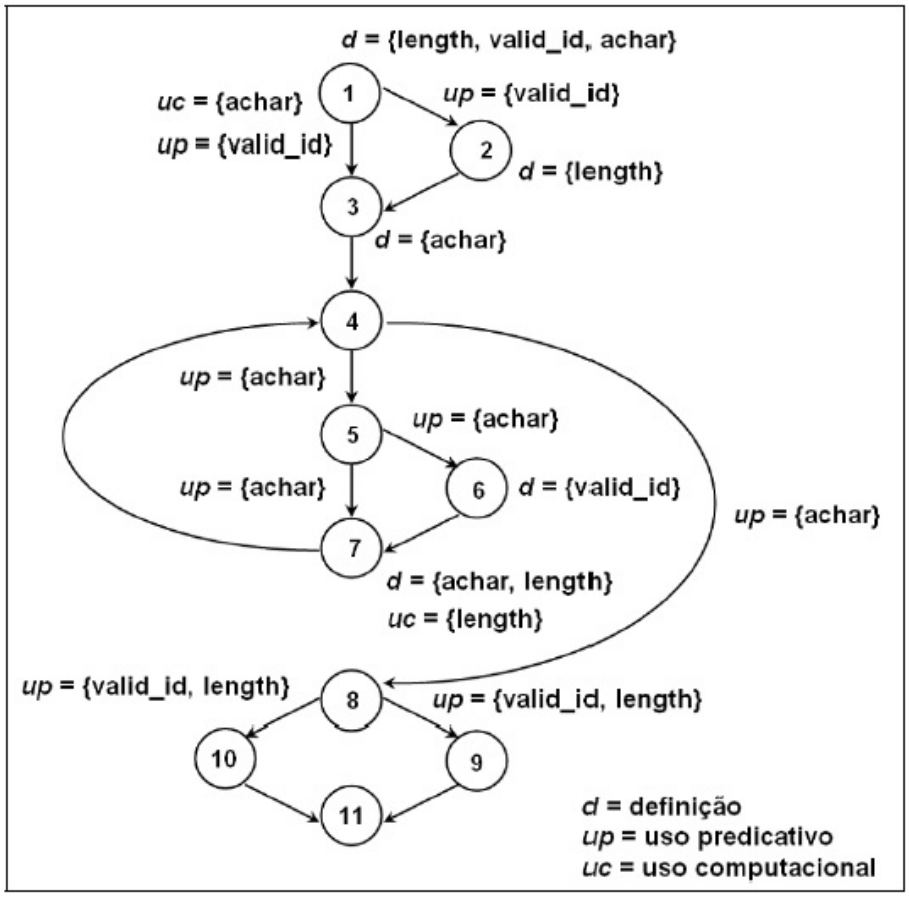

Figura 2.3: Grafo de fluxo de dados do programa identifier (Barbosa et al., 2007).

Outro conceito importante é o de *caminho livre de definição*. Um caminho (n_1, ..., n_m), $m \geq 1$, é um *caminho livre de definição* para uma variável x se não contém nenhuma definição de x nos nós n_1, ..., n_m (Rapps e Weyuker, 1982).

Como exemplos de critérios citados são:

- Todas-definições: esse critério exige que exista, para cada definição de uma variável, pelo menos um caso de teste que alcance um *uso* daquela variável, por meio de um *caminho livre de definição* (Rapps e Weyuker, 1982).

- Todos-usos: esse critério requer que, para cada *definição* de uma variável, possua pelo menos um caso de teste que alcance cada uso daquela variável, por meio de um *caminho livre de definição*. Ou seja, para cada *definição* de variável, todos os *usos* de valores atribuídos nessa *definição* devem ser executados (independentemente do tipo de *uso*).

- Todos-P-Usos e Todos-C-Usos: são variantes do critério **Todos-Usos**, incluindo os critérios *p-usos* ou todos os *c-usos*, respectivamente (Rapps e Weyuker, 1982).

- Todos-potenciais-usos: esse critério requer associações que são caracterizadas independentemente de uma referência (um *uso*) explícito de uma determinada definição. Portanto, são necessários que sejam exercitados, a partir da definição de uma determinada variável, *caminhos livres de definição*, independentemente de ocorrer uma referência explícita dessa variável no caminho (Maldonado, 1991).

Atividades distintas relacionadas às avaliações e comparações de critérios de teste podem ser úteis para a definição da relação custo/benefício entre os critérios, auxiliando na definição de estratégias de teste. Em estudos teóricos e experimentais para a comparação entre critérios, três aspectos podem ser considerados (Wong, 1993): custo, que pode ser medido de diversas formas como, por exemplo, o número de casos de teste necessários para cobrir os requisitos; efetividade, que diz respeito à capacidade dos defeitos serem revelados, baseados em um determinado critério; e dificuldade de satisfação, que refere-se à probabilidade de satisfazer um critério C_2 tendo sido satisfeito um critério C_1.

Delamaro et al. (2007a) dizem que a complexidade e a relação de inclusão, utilizadas nos estudos teóricos, geralmente refletem as propriedades básicas que devem ser consideradas no processo de definição de um critério de teste. Ou seja, o critério C deve:

- incluir o critério Todas-Arestas;

- requerer ao menos um uso de todo resultado computacional, do ponto de vista do fluxo de dados;

- requerer um conjunto de casos de teste finito.

Maldonado (1991) ainda lembra que esses fatores também influenciam na escolha entre os diversos critérios de testes existentes.

2.2.5 Outros critérios baseados em grafos[1]

Grafos dirigidos servem de fundamentação para a cobertura de muitos critérios. Dado um artefato em teste, idealiza-se a obtenção de um grafo. O exemplo mais comum é o mapeamento do código fonte para o grafo de fluxo de controle, já visto anteriormente (Figura 2.2).

Um grafo G é composto por, um conjunto de nós N, um conjunto de nós iniciais N_0, tal que $N_0 \subseteq N$, um conjunto de nós finais N_f, $N_f \subseteq N$, e um conjunto de arestas E, tal que E é um subconjunto de N x N. Para um grafo ser útil na atividade de teste, o grafo deve conter pelo menos um nó N, um N_0 e um N_f, a fim de deixar claro o caminho do teste a ser percorrido. Dentro deste contexto, também é possível considerar apenas parte de um grafo. Um *subgrafo* de um grafo é também um grafo e é definido por um subconjunto de N, juntamente com os subconjuntos correspondentes da N_0, N_f, e E. Arestas são formadas por dois nós descritos como (n_i, n_j). O nó N_i é chamado de antecessor e o nó N_j é chamado de sucessor (Bondy e Murty, 2007).

Segundo Ammann e Offutt (2008), é importante entender que grafo não é o mesmo que artefato. A mesma abstração que produz o grafo a partir do artefato também mapeia os casos de teste para o caminho no grafo. Um critério de cobertura baseado em grafo avalia um conjunto de teste de um artefato em termos de como os caminhos correspondentes aos casos de teste "cobrem" o grafo de tal artefato.

Quando aplicados critérios de teste em grafos específicos, algum caminho, por alguma razão, pode não ser executado. Por exemplo, um caminho pode exigir que um ciclo venha a ser executado zero vezes em uma situação em que o programa sempre executa o ciclo pelo menos uma vez. Esse tipo de problema é baseado na semântica do artefato de software que o grafo representa (Ammann e Offutt, 2008).

Define-se como um nó sintaticamente alcançável, aquele nó n que possui um caminho do nó inicial n_i até o nó n. Do mesmo modo, um nó n pode ser definido como sendo semanticamente alcançável, se e somente se, for possível executar pelo menos um dos caminhos determinados pelos dados de entrada.

Algoritmos baseados em grafos podem ser usados para calcular a acessibilidade sintática. Um caminho de teste representa a execução de um caso de teste, começando sempre no nó n_i, pois os casos de teste sempre começam a partir de um nó inicial. Ammann e

[1]Extraído de Ammann e Offutt (2008).

Offutt (2008) enfatizam que um único caminho de teste pode corresponder a muitos casos de teste, assim como um caminho de teste pode corresponder a zero caso de teste. Assim como no teste estrutural, os grafos estão sendo utilizados em outras abordagens de teste como: cobertura de grafo para elementos de projeto, cobertura de grafo para especificação e cobertura de grafo para caso de uso, definidos por Ammann e Offutt (2008). A seguir são levantados os pontos mais importantes de cada abordagem.

Devido ao aumento na ênfase do reuso de software e da modularidade, o teste de software baseado em elementos de projeto está se tornando cada vez mais importante. Um dos benefícios da modularidade é que os componentes de software podem ser testados de forma independente, o que normalmente é feito por programadores durante o teste de unidade. Um módulo caracteriza-se por uma coleção de unidades ou componentes relacionados (Pressman, 2006).

A cobertura de grafo para elementos de projeto começa pela criação dos grafos que são baseados em *acoplamentos* (*coupling*) entre componentes do software. O *acoplamento* mede a relação de dependência entre duas unidades, refletindo suas interconexões de modo oferecer informações resumidas sobre a concepção e a estrutura do software; falhas em uma unidade podem afetar a unidade acoplada. A maioria dos critérios de teste baseados em elementos de projeto exigem que as conexões entre os diversos componentes do programa sejam visitadas (Ammann e Offutt, 2008). A cobertura de um nó exige que cada método seja chamado pelo menos uma vez, também denominado *cobertura de método*. A cobertura de aresta exige que cada chamada seja executada pelo menos uma vez e é também chamada de *cobertura de chamadas*.

Em relação ao critério baseado em grafo de fluxo de dados para elementos de projeto, as ligações de controle entre os elementos do projeto são simples e diretas e os testes baseados nestes elementos podem não ser muito eficazes em encontrar defeitos. Por outro lado, as conexões de fluxo de dados são muito complexas e difíceis de serem analisadas. A questão principal é onde as *definições* e os *usos* ocorrem. Ao testar unidades de programa, as *definições* e os *usos* estão na mesma unidade. Durante os testes de integração, *definições* e *usos* estão em unidades diferentes.

Fundamentalmente, para o critério baseado em fluxo de dados, para elementos de projeto, alcançar a confiança das interfaces entre as unidades do programa integrado, deve ser assegurado que as variáveis definidas em unidades chamadoras são adequadamente utilizadas nas outras unidades (Ammann e Offutt, 2008). O critério requer a execução das definições de parâmetros reais por meio de chamadas para usos de parâmetros formais. Três tipos de *acoplamentos* de fluxo de dados podem ser identificados: acoplamento de parâmetro, acoplamento compartilhado e acoplamento de dispositivos. No acoplamento de parâmetro, os parâmetros são passados em chamadas. O acoplamento compartilhado

de dados ocorre quando há um acesso de duas unidades aos mesmos dados, como um objeto global. Já o acoplamento de dispositivos externos ocorre quando há duas unidades acessando um mesmo meio externo, como um arquivo.

Testadores também podem utilizar as especificações do software como fontes de grafos. Nos grafos com base em restrições de sequenciamento entre os métodos de uma classe, quando os grafos representam o comportamento do estado do software, as chamadas de grafos para as classes muitas vezes acabam sendo desconexas. Essas chamadas fazem uso de *Tipos Abstratos de Dados* (TADs), sendo que os métodos de uma classe não compartilham chamadas para todos. No entanto, a ordem das chamadas é quase sempre restringida por regras. Por exemplo, não é possível retirar um elemento na pilha até que tenha algo nela. Uma restrição de sequenciamento impõe restrições sobre a ordem em que os métodos podem ser chamados.

O outro método para o uso de grafos baseados em especificações é o modelo de comportamento do estado do software desenvolvido por meio de algum tipo de *Máquina de Estados Finitos* (MEF). Uma MEF é um grafo em que os nós representam os estados do comportamento de execução do software e as arestas representam transições entre os estados (Gill, 1962). Um estado representa uma situação reconhecível que continua a existir durante algum período de tempo. Um estado é definido por valores específicos para um conjunto de variáveis. Uma transição pode ser pensada como uma ocorrência em tempo zero e, geralmente, representa uma mudança de valores de uma ou mais variáveis.

Em MEFs, muitos dos critérios anteriores podem ser definidos diretamente. A cobertura do nó requer que cada estado da MEF seja visitado pelo menos uma vez e é chamada de *cobertura do estado*. A cobertura da aresta é aplicada ao exigir que cada transição da MEF tem que ser visitada pelo menos uma vez, denominada de *cobertura de transição*. Outro critério originalmente definido para as MEFs é o critério de cobertura chamado de *par de transições*.

Os critérios de cobertura de fluxo de dados, quando aplicados em MEFs, são mais problemáticos. Na maioria das formulações de MEFs, os nós não têm *definições* ou *usos* das variáveis. Ou seja, toda a ação está na transição. Ao contrário dos grafos com o código-fonte, arestas diferentes do mesmo nó em uma MEF não precisam ter o mesmo conjunto de *definições* e *usos*. Além disso, a semântica dos desvios implica que os efeitos de uma mudança para as variáveis envolvidas são sentidos de imediato na transição para o próximo estado (Ammann e Offutt, 2008).

Casos de uso são amplamente utilizados para explicar e expressar os requisitos do software (Pressman, 2006). Eles se destinam a descrever a sequência de ações que o software executa como resultado de entradas de usuários, ou seja, elas ajudam a expressar

o *workflow* (fluxo de trabalho) de uma aplicação. Caso de uso, quando desenvolvido no início de um projeto, pode ajudar o testador a começar a atividade de teste mais cedo.

O caso de uso pode ser considerado tecnicamente um grafo, não é um grafo muito útil para o teste de software. O melhor que um testador poderia fazer é usar uma cobertura de nó, o que equivale a "tentar cobrir cada caso de uso". No entanto, casos de uso são elaborados, ou "documentados", com uma descrição mais detalhada do texto. Uma alternativa a ser utilizada é o chamado diagrama de atividade. Ele mostra o fluxo entre as atividades que podem ser usadas para modelar uma variedade de coisas, incluindo mudanças de estado, retorno de valores, e computação. Segundo Ammann e Offutt (2008), o diagrama de atividade pode ser usado para modelar casos de uso na forma de grafos, considerando atividades como passos ao nível do utilizador.

Dois critérios que são mais aplicáveis a grafos de caso de uso são *cobertura de nós* e *cobertura de arestas*. Casos de teste são derivados da interpretação dos nós e determinados como entradas para o software. Outro critério usado para grafos de caso de uso utiliza o conceito de cenários.

Um cenário de caso de uso é caracterizado como uma *instância* de um caso de uso. O cenário deve fazer algum sentido semântico aos usuários e muitas vezes, é possível derivá-lo. Se o grafo do caso de uso é finito, então é possível obter todos os cenários possíveis. Entretanto, o conhecimento de domínio pode ser utilizado para reduzir o número de cenários interessantes. Isso caracteriza uma *cobertura de um caminho especificado* (Ammann e Offutt, 2008).

Um conjunto S para a *cobertura de um caminho especificado* é simplesmente o conjunto de todos os cenários. Se o testador escolhe todos os caminhos possíveis dos cenários, então à *cobertura de um caminho especificado* é equivalente a cobertura completa do caminho. Os cenários são escolhidos por especialistas do domínio. Portanto, não é garantido que a *cobertura de um caminho especificado* cubra as arestas ou os nós. Ou seja, é possível escolher um conjunto de cenários S, tal que não incluam todas as arestas.

2.3 Considerações finais

Neste capítulo, foram apresentados conceitos relacionados a técnicas e critérios de teste, enfatizando a técnica estrutural que utilizam grafos orientados para que possam representar o controle lógico de um programa. A fim de selecionar um conjunto de casos de teste que satisfaçam os requisitos extraídos do ambiente virtual a serem testados, critérios de teste serão definidos baseados no grafo de cena.

Nessa linha de pesquisa, um domínio que até agora pouco se explorou é o de sistemas de Realidade Virtual (RV). No próximo capítulo, é realizado um estudo sobre RV e Grafos de Cena (GC), uma forma para se descrever Ambientes Virtuais (AVs).

Capítulo

3

Realidade virtual

A Realidade Virtual pode ser entendida como uma interface avançada para aplicações computacionais, na qual o usuário pode interagir, em tempo real, a partir de um ambiente tridimensional sintético, utilizando dispositivos multissensoriais (Kirner e Siscoutto, 2008).

Aplicações de RV são aquelas nas quais um determinado Ambiente Virtual (AV) é simulado por meio do computador e que permitem que o usuário interaja com esse AV, seja por meio de uma simples tela de computador, seja por meio de dispositivos específicos que permitam a imersão do usuário (Burdea e Coiffet, 1994; Jacobson, 1991; Krueger, 1991).

Um dos precursores da tecnologia de RV foi o pesquisador Ivan E. Sutherland, que desenvolveu o primeiro sistema gráfico interativo. Este sistema fazia interpretações de desenhos como dados de entrada e realizava associações com topologias conhecidas, gerando novos desenhos (Sutherland, 1964).

O termo conhecido como Realidade Virtual surgiu em meados dos anos 70. Pesquisadores sentiram a necessidade de uma definição para diferenciar as simulações computacionais tradicionais dos mundos digitais que começavam a ser criados. Nasciam então as interfaces de terceira geração. Interações eram produzidas sobre as situações geradas, utilizando-se comandos não convencionais, diferenciando-se das interfaces dotadas apenas de reprodução multimídia, mantidas até então por interfaces bidimensionais de primeira e segunda geração (Bolt, 1980; Krueger, 1977; Lanier, 1992). O termo é bastante abrangente levando acadêmicos, desenvolvedores de software e pesquisadores a definirem RV

baseados em suas próprias experiências. De acordo com Pimentel e Teixeira (1995), a RV conduz o usuário a uma experiência interativa e imersiva fundamentada em imagens gráficas tridimensionais geradas em tempo real por um computador.

Machover e Tice (1994) afirmam que a qualidade dessa experiência de RV é essencial, pois deve estimular ao máximo, de forma criativa e produtiva, o usuário. Os sistemas de RV também precisam fornecer uma reação de forma coerente aos movimentos do participante, tornando a experiência consistente. O principal objetivo da tecnologia de RV é fazer com que o participante desfrute de uma sensação de presença no mundo virtual (Jacobson, 1994).

Para proporcionar uma sensação de presença em aplicações de RV é essencial que o ambiente simulado seja representado por meio de objetos tridimensionais (3D) e proporcione interação em tempo real. Por outro lado, esse tipo de representação em dispositivos que são inerentemente bidimensionais não é uma tarefa simples e demanda alto consumo de recursos computacionais (Silva et al., 2009).

No final de 1986, uma equipe da NASA já possuía um AV que permitia aos usuários ordenar comandos pela voz, ouvir fala sintetizada e som 3D, além de manusear objetos virtuais diretamente por meio de movimentos das mãos. Assim, verificou-se a possibilidade de comercialização de um conjunto de novas tecnologias, com o custo de aquisição e desenvolvimento cada vez mais acessível (Pimentel e Teixeira, 1995). A conscientização de que as iniciativas da NASA tornavam-se tecnologias comercializáveis deu início a inúmeras pesquisas em RV no mundo inteiro. Organizações variando de empresas de software até grandes corporações de informática começaram a desenvolver e vender produtos e serviços ligados à RV. Em 1987, a *VPL Research Inc.*[1] começou a vender capacetes e luvas digitais e em 1989 a *AutoDesk*[2] apresentava o primeiro sistema de RV baseado em um computador pessoal (PC) (Jacobson, 1994).

A construção de um AV, segundo Silva et al. (2004), aumentou em complexidade, estabelecendo novas formas de representação de objetos na cena. Dentre elas estão os chamados grafos de cena, que proporcionam camadas de abstração para organizar de forma hierárquica os objetos de um AV. Grafos de cena serão discutidos em detalhes na Seção 3.3.

Com o desenvolvimento das tecnologias de RV, diversos tipos de dispositivos de entrada e de saída foram surgindo ao longo do tempo. Para ocorrer uma interação com o ambiente virtual, o usuário pode utilizar equipamentos convencionais, como um mouse

[1] http://vpl.astro.washington.edu/sci/index.html
[2] http://usa.autodesk.com/

ou um teclado comum (Figura 3.1), ou equipamentos não convencionais, como luva de dados[3] e equipamento háptico[4] (Figura 3.2).

Figura 3.1: Equipamentos convencionais, teclado e mouse.

Figura 3.2: Equipamento Háptico (Sensable, 2012) e Luva de Dados (5DT, 2012).

Como um exemplo de interação, o usuário pode sentar-se em uma cadeira comum, utilizar um capacete de RV, também chamado de HMD[5] (*Head Mounted Display*) (Figura 3.3) ou um monitor comum para visualizar a cena virtual, ou até mesmo pode estar cercado por paredes onde as imagens são projetadas, denominada CAVE (DeFanti et al., 2009) (Figura 3.4), movendo os modelos de objetos utilizando um mouse sem fio ou um dispositivo háptico, eletromagnético ou uma luva de dados (Corrêa et al., 2009).

Para facilitar o desenvolvimento de aplicações multidispositivos, surgiram algumas bibliotecas de RV que são capazes de fornecer dados como a posição do dispositivo, ou a sua orientação, sem que a aplicação necessite saber de qual dispositivo este dado está vindo. Elas agem como uma camada de abstração. Além dos dispositivos de entrada, a aplicação pode gerar a saída visual sem saber se ela será apresentada em um monitor ou em uma parede com várias telas de projeção.

[3]http://www.5dt.com/products/pdataglove5u.html
[4]http://www.sensable.com
[5]http://www.5dt.com/products/phmd.html

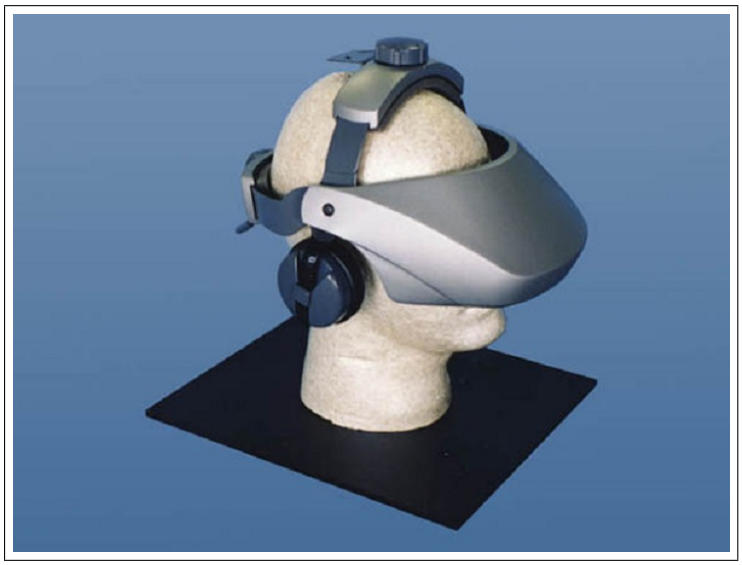

Figura 3.3: *Head Mounted Display* (5DT, 2012).

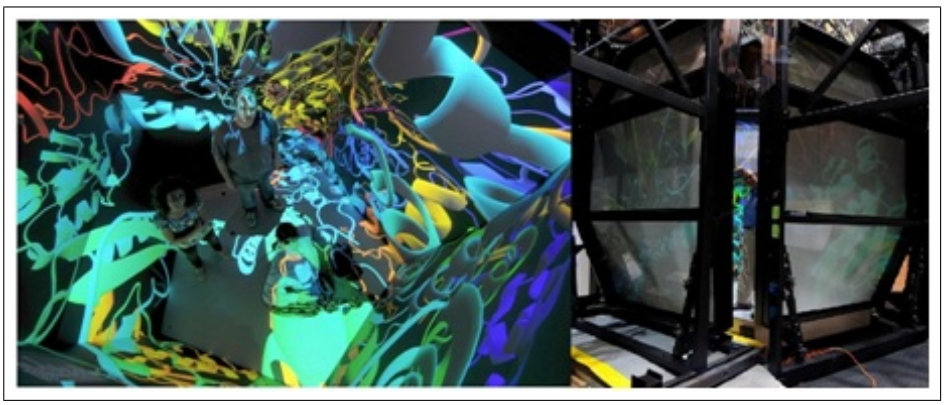

Figura 3.4: Foto de uma CAVE utilizada em DeFanti et al. (2009).

3.1 Bibliotecas gráficas

A construção de ambientes virtuais consiste em representar objetos do mundo real ou abstrato utilizando primitivas que possam reproduzi-lo no ambiente computacional. Existem várias técnicas disponíveis para essa representação, sendo que uma das mais comuns é a representação por malhas de triângulos (Birthelmer et al., 2003; Campagna et al., 1998). Campagna et al. (1998) dizem que malha de triângulos serve como uma base para a superfície primitiva adaptada para aproximar uma geometria suave de um modelo. Permite também adaptar o número preciso de primitivas para a forma geométrica, em vez de um modelo arbitrário.

Em geral as bibliotecas gráficas são responsáveis por fazer a ligação entre a aplicação e a placa gráfica, como mostra a Figura 3.5. A aplicação envia a malha, juntamente com seus atributos (como cor, material e textura) para a biblioteca gráfica e esta, por sua vez, envia os vértices para a placa gráfica que fará os cálculos necessários para transformar a

coordenada 3D dos vértices dos triângulos em pixels na tela do computador (Shreiner, 2009).

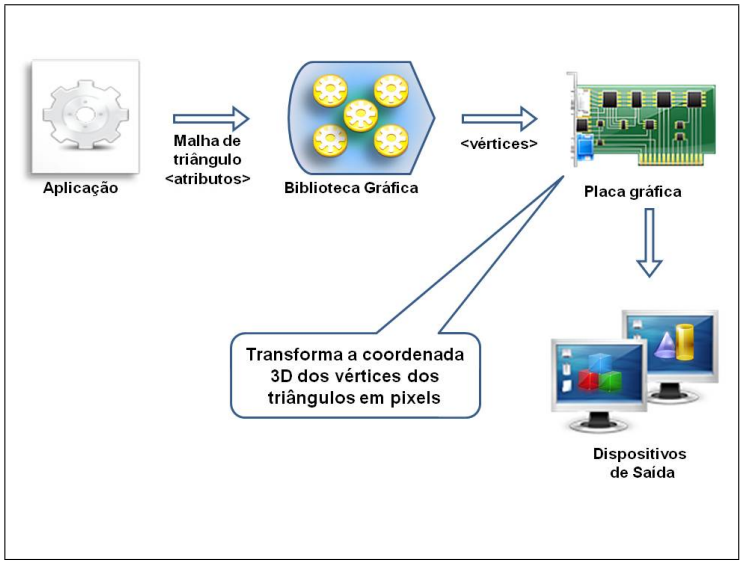

Figura 3.5: Etapas de renderização de um objeto 3D (Shreiner, 2009).

As bibliotecas gráficas, segundo Walsh (2002), também chamadas de APIs (*Application Programming Interface*) gráficas, mais comumente utilizadas são a DirectX e a OpenGL. A DirectX [6] é uma coleção de APIs desenvolvida pela empresa Microsoft e distribuída para tarefas relacionadas à programação de jogos para o sistema operacional Windows. A OpenGL [7] é uma API livre que surgiu na década de 90 utilizada na computação gráfica, disponível na maioria dos sistemas operacionais.

Com o avanço dos recursos computacionais e da tecnologia das placas gráficas, das aplicações de computação gráfica e, por conseguinte, das aplicações de RV, efeitos visuais inovadores estão sendo produzidos em tempo real. No entanto, o avanço da tecnologia das placas gráficas traz consigo um aumento de processamento e complexidade de desenvolvimento para conseguir efeitos ainda melhores. Também emprega modelos de objetos tridimensionais mais complexos, com mais vértices e maior nível de detalhe, levando o aumento dos dados a serem processados nas bibliotecas gráficas.

Segundo Akenine-Möller et al. (2008), para conseguir acompanhar este avanço na tecnologia de processamento da placa gráfica, técnicas de otimização são utilizadas. Pode-se tomar como exemplo desse tipo de mecanismo, a técnica de *Culling* (Figura 3.6). Nessa técnica, os objetos que estiverem fora do campo de visão do observador ou atrás de outro objeto, são descartados. Caso não houvesse esse algoritmo, todos os objetos teriam que passar por todo o *pipeline* de renderização até que fossem descartados no final por não estarem dentro da área visível do dispositivo de saída.

[6] Pode ser encontrada em (http://www.microsoft.com/windows/directx).
[7] Pode ser encontrada em (http://www.opengl.org/).

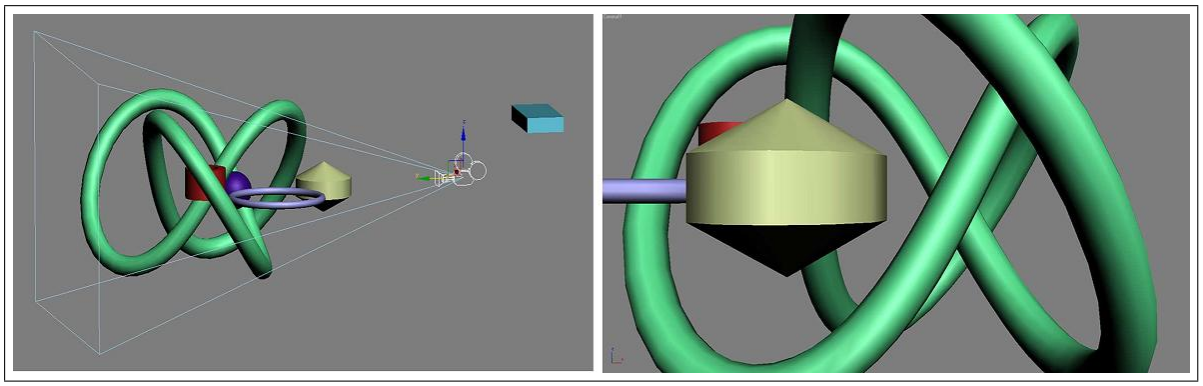

Figura 3.6: Amostra de um AV e do campo de visão do observador na câmera (Akenine-Möller et al., 2008).

Renderização ou *pipeline* de renderização é o processo pelo qual um computador cria imagens para serem exibidas em dispositivos bidimensionais a partir de modelos tridimensionais (Shreiner, 2009).

A principal função do *pipeline* é gerar, ou renderizar, uma imagem bidimensional, dados uma câmera virtual, objetos tridimensionais, fontes de luz, sombreamentos, texturas, entre outros. Na Figura 3.6 são ilustrados os objetos na cena com uma câmera delimitando o campo de visão que será renderizado, depois de aplicada a técnica de *Culling*. É ilustrada também a visão da câmera que será exibida para o observador.

A fim de facilitar o desenvolvimento de aplicações que utilizam gráficos tridimensionais, uma forma de representar os objetos em um AV pode ser utilizada: são os chamados grafos de cena, abordados na próxima seção.

3.2 Grafo de cena

Segundo Walsh (2002), os grafos de cenas resolvem problemas que geralmente surgem na montagem e no gerenciamento da cena tridimensional em aplicações de RV. Grafos de cena protegem os detalhes de renderização, permitindo que o programador se preocupe com o que renderizar e não como fazer a renderização. Como a Figura 3.7 ilustra, grafos de cena oferecem uma alternativa de alto nível para as APIs gráficas, simplificando tarefas como a de multi textura, que é um efeito de duas texturas aplicadas uma sobre a outra em um mesmo polígono, além de fornecer à API diversos algoritmos de otimização, como o *Culling*, entre outros. Por sua vez, fornecem uma camada de abstração de subsistemas gráficos responsáveis por processar e apresentar os dados da cena aos observadores por meio de dispositivos de saída (Sherman e Craig, 2003).

Antes dos grafos de cena, modelos gráficos, dados da cena, assim como seus comportamentos, eram representados proceduralmente. Consequentemente, o código que definia a cena era muitas vezes intercalado com o código que definia os procedimentos que ope-

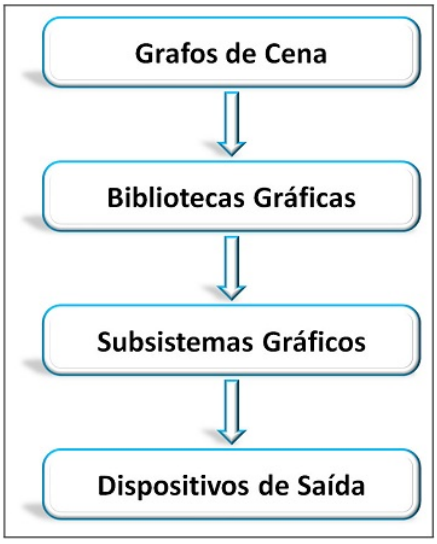

Figura 3.7: Camadas de um sistema gráfico que utiliza grafo de cena.

ravam sobre ela. Como resultado tinha-se um código complexo e inflexível que era difícil de criar, modificar e manter, problemas que os grafos de cena ajudam a resolver (Walsh, 2002).

Em se tratando de otimização, os grafos de cena, juntamente com as otimizações existentes na biblioteca, aumentam a taxa de quadros por segundo, ou seja, elevam a frequência com que as imagens sequenciais são produzidas. Segundo Tori et al. (2006), quanto maior a taxa de quadros por segundo, maior é a sensação de imersão, característica fundamental em um ambiente de RV.

Na composição de um AV, diversos aspectos como posição do objeto, forma, textura, iluminação, comportamento e visão do mundo virtual devem ser descritos (Ferreira, 1999). Esses aspectos podem ser adicionados a um grafo de cena por meio de vértices e arestas direcionadas que formam um grafo acíclico. No grafo de cena, é possível encontrar nós internos e nós folhas. Os nós folhas contêm a descrição geométrica de um objeto e os nós internos representam transformações tridimensionais como rotação, translação e escala. O nó raiz, que se conecta a todos os demais, direta ou indiretamente, é a representação do AV como um todo (Silva et al., 2004).

O conceito de grafo de cena não é totalmente novo, como pode ser visto no artigo de Strauss e Carey (1992). Diversas bibliotecas de implementação de ambientes 3D utilizam esse conceito como forma de representar os objetos, suas características e seus relacionamentos. É o caso, por exemplo, do OpenSceneGraph[8], OpenSG[9], OpenInventor[10] e Java3D[11].

[8]http://www.openscenegraph.org/projects/osg
[9]http://opensg.vrsource.org/trac
[10]http://oss.sgi.com/ projects/inventor/
[11]https://java3d.dev.java.net/

Anteriormente às bibliotecas gráficas de renderização, como OpenGL e Direct3D, os gráficos em tempo real eram processados em um gerador de imagem especial de *hardware* que continha toda a base de dados visuais com informações das modelagens. Os programadores eram limitados a fazer alterações de transformações nessas bases de dados, como a posição, a translação ou rotação de um objeto (Bar-Zeev, 2007).

Modelos de programas que utilizam grafos de cena estabelecem uma fronteira limpa entre a representação e a renderização. Assim, as cenas podem ser colocadas e mantidas independentemente das rotinas que operam sobre elas. Além de facilitar a organização, possibilitam que se crie sofisticados conteúdos utilizando ferramentas visuais sem precisar definir a forma como o trabalho é processado (Sherman e Craig, 2003).

Na Figura 3.8 pode-se observar a estrutura de um grafo de cena construído por meio da API Java3D (Sun, 2000).

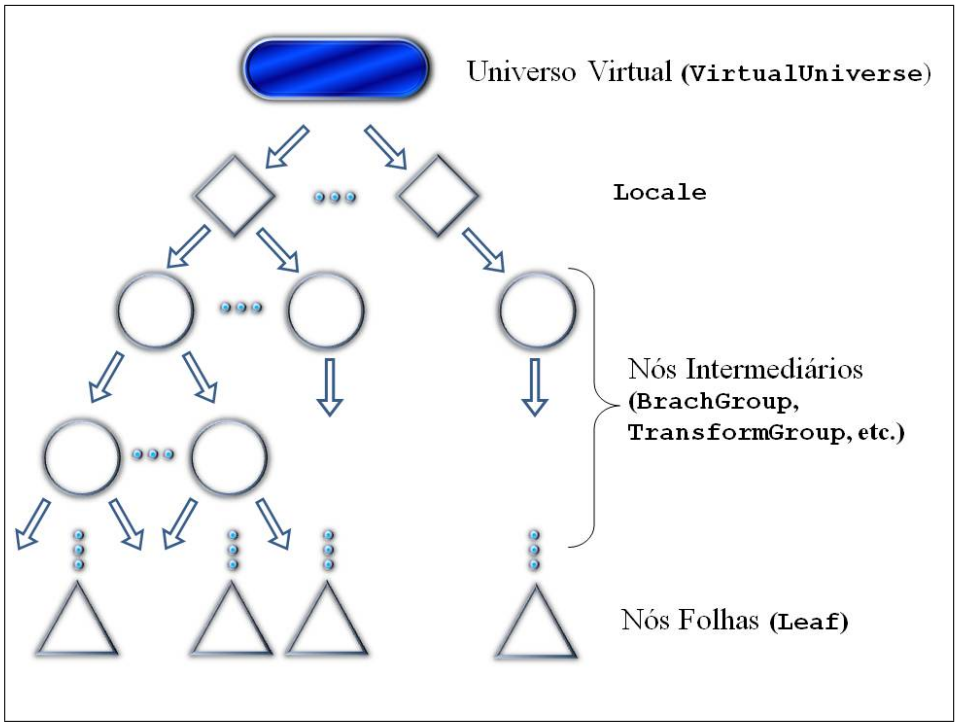

Figura 3.8: Estrutura do grafo de cena em Java3D.

Observa-se, na Figura 3.8, o grafo de cena ilustrado com símbolos padrões. Cada grafo de cena possui ao menos um objeto Universo Virtual.

Um Universo Virtual pode ser definido como um espaço tridimensional com um conjunto de objetos associados. Universos Virtuais servem como a maior unidade de representação no grafo de cena. Essa unidade pode ser muito grande, tanto em unidades de espaço físico quanto no conteúdo. Na verdade, na maioria dos casos, um único universo virtual irá atender às necessidades de todo um aplicativo. O objeto Universo Virtual tem uma lista de objetos Locale. Pensa-se em um objeto Locale como sendo um ponto de

referência utilizado para determinar a localização de objetos visuais no Universo Virtual. É tecnicamente possível para um programa ter mais de um Universo Virtual, no entanto, não há nenhuma maneira dos Universos Virtuais se comunicarem inerentemente entre eles. Além disso, um objeto Grafo de Cena não pode existir em múltiplos Universos Virtuais simultaneamente (Sun, 2000).

Todos os grafos de cena desenvolvidos com a API Java3D devem se conectar a um objeto Universo Virtual para ser exibido. O objeto Locale pode-se servir como raiz de vários subgrafos do grafo de cena. Um objeto do tipo Nó Internos é a raiz de um subgrafo, ou grafo de desvio.

Existem duas categorias diferentes de subgrafo no grafo de cena: o grafo de desvio e o grafo de conteúdo. O grafo de conteúdo especifica o conteúdo do universo virtual - geometria, aparência, comportamento, localização, som e luzes. O grafo de desvio especifica os parâmetros de visualização, como o local de visualização e direção. E por fim fazem parte do grafo de cena os objetos do tipo Nós Folhas. Outros nós que compõe a Figura 3.8 serão descritos na Seção 3.2.2.3.

Na próxima Seção são descritas as características de um grafo de cena.

3.2.1 Descrição do grafo de cena

Um grafo de cena organiza hierarquicamente os objetos presentes na cena, permitindo que os algoritmos de otimização atuem sobre ele (Kalkusch e Schmalstieg, 2006). Tradicionalmente ele é considerado uma estrutura com um alto nível para gerenciar dados de conteúdo 3D e cada vez mais está se tornando popular como um mecanismo de aplicação geral para gerenciar estes dados 3D (Walsh, 2002). De um modo geral, os AVs representam aspectos, do mundo real ou não real, de uma aplicação de computação gráfica. A seguir são descritos os aspectos mais importantes, segundo Ferreira (1999):

- Descrição geométrica: a descrição geométrica é o modo que um modelo pode ser representado. Quanto maior o nível de detalhe da descrição geométrica, melhor a qualidade da visualização e menor é a velocidade que a imagem é gerada (Figura 3.9). Ferreira (1999) lembra que a descrição geométrica pode também ser classificada no tempo como estática ou dinâmica. A descrição geométrica estática não varia a forma de um objeto no tempo, enquanto que na dinâmica sua forma pode ser variada.

- Câmera: a câmera captura o ponto de vista da localização atual, orientação e perspectiva. O sistema de visualização, em seguida, mostra esse ponto de vista para o observador (Sowizral et al., 2000). Geralmente a camera possui uma projeção perspectiva, ou seja, uma visão bidimensional de um determinado modelo tridimensional (Figura 3.10);

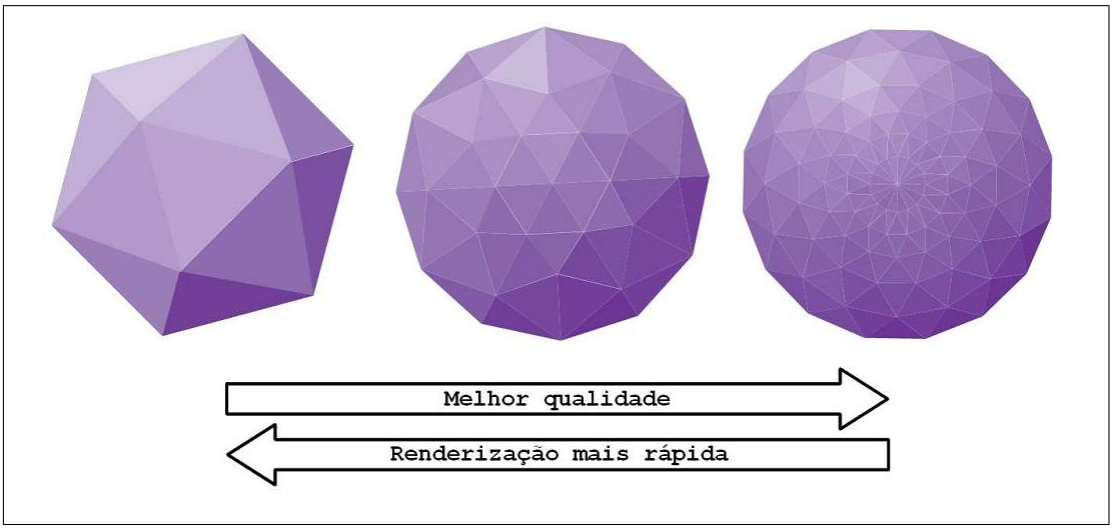

Figura 3.9: Exemplo de uma descrição geométrica.

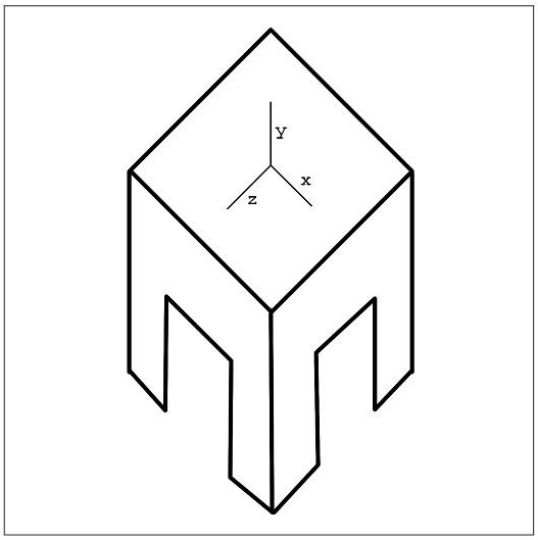

Figura 3.10: Visão em perspectiva.

- **Transformação:** qualquer modelo virtual dentro de um AV pode sofrer transformação de translação, rotação ou escala. A transformação faz com que as coordenadas locais do modelo sejam mapeadas para as coordenadas do mundo virtual. As transformações estão fortemente relacionadas à hierarquia dos objetos, pois se um nó pai for transformado, seus nós filhos relacionados também serão transformados (Bar-Zeev, 2007);

- **Aparência:** a aparência, assim como a descrição geométrica, interfere na geração e na qualidade da imagem final. Em linhas gerais, a aparência pode ser dada de diferentes maneiras. Entre elas, aquelas que geralmente estão presentes nas aplicações de RV são: cor, material, textura, transparência, sombra, reflexão e ausência

de aparência. A ausência de aparência é útil para entidades presentes no AV que não podem ser visualizadas. As outras aparências (cor, material, textura, transparência, sombra e reflexão) estão entre os diversos atributos que definem a aparência de um objeto. Pode haver também uma combinação balanceada entre as aparências (Ferreira, 1999).

- **Comportamento:** o comportamento pode ser classificado em duas categorias distintas: determinístico e não determinístico. Nos comportamentos determinísticos, seus estados são definidos em função do tempo. Este comportamento permite que se navegue para frente e para trás no tempo, sabendo exatamente qual o estado de cada entidade em cada momento. Já o comportamento não-determinístico segue um padrão definido, pois neste caso é impossível prever o estado de um objeto que apresente esse comportamento no futuro, não havendo a possibilidade de se restaurar seu estado no passado (Roehl, 1995).

- **Iluminação:** em se tratando de iluminação, não há nenhuma restrição da quantidade que pode ser empregada em um AV. Há vários tipos de modelos de iluminação. Um exemplo desse tipo de modelo é o anisotrópico que pode ser visto no trabalho de Poulin e Fournier (1990). A ansitropia utiliza cilindros para calcular a direção da iluminação.

Os aspectos relatados devem ser inseridos no grafo de cena a fim de representar o AV. Cada nó possui um conjunto de atributos que podem influenciar seus nós subsequentes. Como já mencionado anteriormente, os grafos são organizados de forma hierárquica correspondendo semântica e espacialmente com o mundo ou universo modelado (Sowizral et al., 2000).

De um modo geral, a otimização a ser feita é a ordenação do *pipeline* de processamento. É importante ressaltar que alterações de estado podem ser custosas porque podem provocar a paralisação do *pipeline* gráfico, para reconfiguração com novos parâmetros. Como visto anteriormente a hierarquia do grafo de cena ajuda na otimização e na utilização do *pipeline* (Bar-Zeev, 2007).

Um exemplo que pode causar a paralisação do *pipeline* se dá quando é utilizada iluminação na cena. O *hardware* é configurado para realizar uma série de operações especiais. Segundo Bar-Zeev (2007), quando é ligada ou desligada a iluminação, todo o processamento feito anteriormente terá que ser descartado. O *pipeline* é reiniciado, porque o custo (*overhead*) associado com essa reconfiguração pode ser muito alto para a aplicação.

3.2.2 Nós do grafo de cena

Tipos diferentes de nós em um grafo de cena armazenam algum tipo de dado ou possuem determinadas responsabilidades. Esta seção descreve alguns tipos comuns de nós encontrados em grafos de cena.

3.2.2.1 Nós internos

Como pode ser visto na Figura 3.8, nós internos ou nós de agrupamento possuem um papel de grande importância no grafo de cena. Eles se responsabilizam por estabelecer desvios que podem caracterizar sub-hierarquias dentro do grafo. Os nós de agrupamento podem ter somente um nó pai, mas diversos nós filhos (Chen e Chen, 2008).

Em grafos de cena que utilizam o paradigma de orientação a objetos, é comum estabelecer tipo de nós genéricos de grupo como a base para todos os nós subsequentes. Operações sobre este grupo incluem adicionar, remover e enumerar os nós filhos que são processados em uma ordem especificada ou em paralelo (Sowizral et al., 2000).

Na Tabela 3.1 são descritas algumas variações de nós de agrupamento utilizadas em grafos de cena.

3.2.2.2 Nós folhas

Os nós folhas, também chamados de nós de geometria, armazenam dados necessários para descrever a forma de um objeto qualquer, assim como especificar luzes e sons que estão contidos na cena virtual. Esses nós podem ser divididos em subtipos, como mostrado na Tabela 3.2.

3.2.2.3 Outros tipos de nós

Outros tipos de nós que não se enquadram nas categorias anteriores, mas que são muito importantes, são listados nesta seção.

- **Fontes de Luz:** as fontes de luz podem comprometer a cena integralmente ou localmente. Um nó do tipo *fonte de luz* pode ser também considerado como um nó de grupo. Nesse contexto, ele obedece à ideia de hierarquia, ou seja, iluminam-se todos os seus nós filhos. Caso este nó não tenha filhos, todo o universo será iluminado (Sowizral et al., 2000). O nó fonte de luz tem associado a ele uma cor, um estado (ligado ou desligado), um objeto que especifica a região de influência da luz e o tipo (*ambientlight*, *directionallight*, *pointlight* ou *spotlight*).

Tabela 3.1: Variações de nós de agrupamento utilizadas em grafos de cena.

Nós do tipo internos ou de agrupamento.	
Transformações Geométricas	Este tipo de nó tem como principal característica estabelecer transformações como translação, rotação e escala para todos os seus nós filhos sucessivamente. Assim, todas as transformações que forem feitas no nó interno afetarão os nós descendentes.
Técnica *Level of Detail* (LOD)	Segundo Chen e Chen (2008), a técnica de LOD, ou seja, nível de detalhe, pode ser entendida como um mecanismo de aprimoramento de renderização a fim de agilizar uma rápida visualização e animação. Em alguns tipos de visões como a em perspectiva, quanto maior a distância de um objeto em relação ao observador, menor o seu tamanho (esse efeito é conhecido como *perspective foreshortening*) e, portanto, menor deve ser o seu nível de detalhe.
Switch	Este tipo de nó pode ser entendido como um desvio ou uma chave, semelhante ao comando switch encontrado nas linguagens de programação. Ele tem a opção de escolha entre uma lista de alternativas, representadas por nós filhos.
Sequenciamento	Este tipo de nó é caracterizado por controlar automaticamente o fluxo de uma animação. Cada nó filho representa um quadro (*frame*) de uma animação.
Grupo Ordenado	Um nó de grupo ordenado estabelecem uma ordem que os modelos devem ser renderizados, em programação orientada a objetos esta ordem pode ser indexada.

- **Câmeras:** uma câmera em um AV representa uma abstração para o ponto de vista do observador, como visto na Figura 3.6. No grafo de cena há um nó especialmente para isso. O ponto de vista inclui a posição e orientação do observador em relação à cena. É incluída também a projeção utilizada para formar o volume de visão e para a conversão para duas dimensões (2D).

- **Som:** segundo Sowizral et al. (2000), este tipo de nó pode descrever o seu conteúdo e o seu tipo de fonte sonora, como por exemplo, o som localizado, de ambiente, de fundo e o de apresentação. Além disso, os nós podem registrar a posição e a direção da fonte sonora.

3.2.2.4 Grafos de Cena na API Java 3D

Java3D é uma API desenvolvida pela *Sun Microsystems* (Sun, 2000) para renderizar gráficos interativos 3D usando a linguagem de programação Java. A API Java 3D oferece um conjunto de classes que permite o desenvolvimento de aplicações 3D em alto nível, utilizando-se de recursos como criação e manipulação de geometrias 3D, animações e,

Tabela 3.2: Subtipos de nós folhas.

Variações de nós folhas de um grafo de cena.	
Geometry	Este tipo de nó faz o registro de informações como coordenadas de vértices e valores para vetores normais, de modo a especificar o tipo de primitiva a ser usada para renderizar a geometria. Em Java3D este tipo de nó é referenciado pelo nó *Shape* (Sowizral et al., 2000).
Billboard ou *Sprite*	A técnica de *billboards* é bastante utilizada em jogos, ela simula um modelo complexo de uma forma mais simples. Um exemplo dessa técnica é a utilização de texturas em polígonos para simular árvores (Sowizral et al., 2000).
Appearance	Este tipo de nó caracteriza-se por conter definições de propriedades de materiais como cores, brilho, rugosidade, informações sobre o tipo de textura e seus dados, entre outras. Assim como o nó *Geometry*, o nó *Appearance* pode ser representado pelo nó *Shape* no Java 3D (Sowizral et al., 2000).
Shape	Caracteriza-se por representar primitivas que podem ser também um objeto tridimensional, como cubos, esferas ou pirâmides. O nó *Shape* faz referência com o nó *Geometry* e o nó *Appearance*.

ainda, interatividade com dispositivos convencionais e não convencionais. São definidas mais de 100 classes apresentadas no pacote *javax.media.j3d.*, possibilitando que o desenvolvimento de aplicações 3D seja totalmente orientado a objetos.

O GC na API Java3D é composto por objetos das classes *VirtualUniverse*, *Locale SceneGraphObject*, *Node*, *NodeComponent*, *Group*, *Leaf* , *BranchGroup*, *TransfomGroup*, *Shape3D*, *Light*, *Behavior*, *Sound*, *Geometry*, *Appearance*, *Material*, *Texture* entre outras. Um exemplo de GC na API Java3D é mostrado na Figura 3.12 e sua legenda é apresentada na Figura 3.12.

Algumas implementações orientadas a objetos, como Java3D, separam as informações das câmeras e visualização dos conteúdos do grafo de cena, como mostra a Figura 3.12 (Sun, 2000). Desta forma, em Java3D, existem dois ramos ou desvios principais: o de conteúdo e o de visualização.

Existem outras variantes sobre a definição de câmera. Na biblioteca de implementação *OpenSceneGraph* não é definido o tipo de nó para câmeras. A especificação da câmera é feita por meio de transformações matriciais. Já em *OpenSG*, é definido um tipo de nó para câmera que faz parte do grafo. Na prática, o efeito final é o mesmo porque a transformação que a câmera define deve ser a primeira a ser aplicada na fase de renderização.

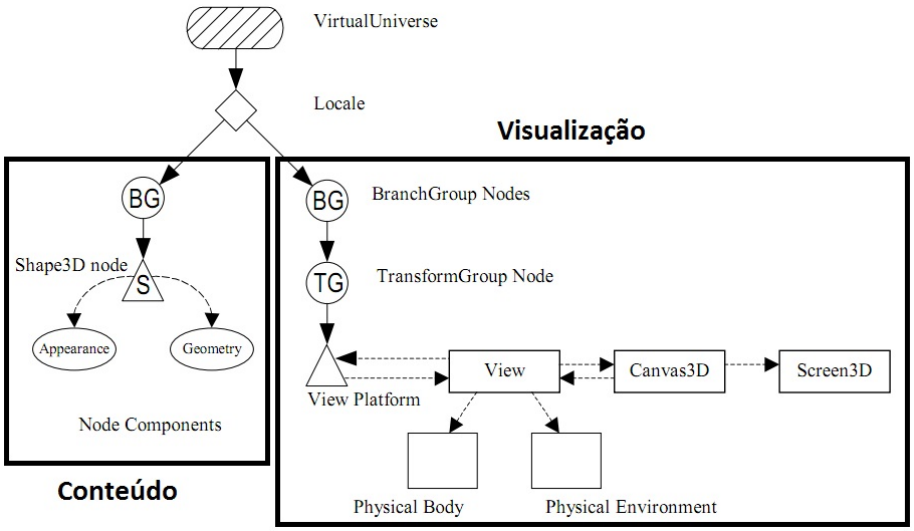

Figura 3.11: Representação de uma câmera na API Java3D (Sun, 2000).

Símbolo	Nome	Finalidade
	VirtualUniverse	Representa o universo virtual
	Locale	Define uma área geográfica com tamanho e localização no universo Virtual. Por *default* (0, 0, 0)
	Group	Organiza os dados, controla a ordem de renderização e mudar a posição, orientação e tamanho dos objetos visíveis no AV.
	Leaf	Representa os objetos que estarão na cena gráfica.
	NodeComponent	Especifica as propriedades de um determinado objeto na cena gráfica.
	Outros objetos	Representa outras classes de objetos.
Símbolo	**Finalidade**	
	Pai e filho	
	Referência	

Figura 3.12: Legenda do GC na API Java3D (Sun, 2000).

3.2.3 Utilização de grafos de cena

A utilização de um grafo de cena em uma aplicação envolve basicamente três passos:

- **Construção:** a construção consiste em criar instâncias dos nós, determinar os valores que os nós representam e determinar as conexões entre os nós seguindo a hierarquia. Logo após determina-se o ponto de entrada do grafo de cena que é o nó raiz.

- **Travessia:** a travessia do grafo é um passo importante e está diretamente relacionada com o desempenho da aplicação. A travessia do grafo de cena começa pela raiz e termina em nós folha. Após a travessia do grafo, obtem-se uma imagem renderizada. Diz ser uma travessia completa se é percorrido todos os caminhos possíveis do nó raiz até os nós folhas.

- **Otimizações:** algumas bibliotecas de grafos de cena, como a Java3D, oferecem uma operação definida como compilação do grafo de cena. A compilação do grafo de cena pode realizar duas tarefas: reordenação dos nós e otimização do grafo. A operação de reordenação dos nós tem como objetivo reagrupar os nós de modo a minimizar o número de alterações de estados do *hardware* gráfico feitas durante a travessia. Já a otimização do grafo tem como objetivo produzir um grafo de funcionalidades equivalente com um menor número de nós. A compilação do grafo pode ser executada uma única vez, no início da aplicação. Contudo, se durante a execução alguma parte do grafo é alterada, pode ser necessário realizar uma nova compilação, o que pode prejudicar o desempenho.

Grafos de cena trazem como vantagem: desempenho, por explorarem técnicas de eliminação de objetos que não contribuem para o resultado final da imagem; produtividade, pois além de implementarem grande parte dos requisitos necessários para se desenvolver uma aplicação, um grafo de cena possibilita a aplicação de paradigmas como orientação a objetos, permitindo a reutilização de projetos; portabilidade, porque traduções da aplicação para execução em outros ambientes de *hardware* podem necessitar apenas de recompilação de código fonte; escalabilidade, a estrutura oferecida pelos grafos de cena faz com que seja possível aumentar a complexidade da simulação de maneira controlada. As desvantagens estão relacionadas ao *overhead* relacionado com a travessia do grafo. Travessias mal implementadas podem acarretar na perda de desempenho.

3.3 Framework ViMeT

O trabalho de mestrado desenvolvido está inserido no contexto do *Instituto Nacional de Ciência e Tecnologia de Medicina Assistida por Computação Cientifica* (INCT–MACC), Edital Nº 15/2008 do MCT, CNPq, FNDCT, CAPES, FAPEMIG, FAPERJ e FAPESP. Um objetivo particular de tal projeto visa a desenvolver um *framework*, denominado *Virtual Medical Training* (ViMeT), que disponibiliza classes destinadas a gerar AVs, com funcionalidades para simular exames de biópsia (Oliveira e Nunes, 2010)[12].

[12]http://www.each.usp.br/lapis/

O ViMeT é um *framework* de RV orientado a objetos que utiliza tecnologia Java, juntamente com a API Java 3D para gerar aplicações voltadas para treinamento médico disponibilizando funcionalidades classificadas como importantes para simulações de treinamento médico, a citar: interface gráfica, detecção de colisão, deformação, interação com equipamentos convencionais e não convencionais, estereoscopia, importação e modelagem de objetos 3D e geração de AVs (Oliveira e Nunes, 2010).

As aplicações geradas pelo ViMeT disponibilizam objetos sintéticos que representam um órgão e um instrumentos médicos. O usuário pode manipular o instrumento médico até que a aplicação identifique que houve uma colisão entre tais objetos, realizando a deformação do local selecionado no objeto que representa o órgão humano. A Figura 3.13 e a Figura 3.14 apresentam interfaces de aplicações geradas pelo *framework* ViMeT.

Com o ViMeT pretende-se tornar possível o desenvolvimento de novas aplicações para treinamento médico com maior produtividade, possibilitando aos estudantes de Medicina treinarem um procedimento médico simulado antes de realizá-lo em pacientes reais. Em linhas gerais, almeja-se diminuir o custo do treinamento médico e oferecer subsídios para incluir uma nova cultura na rotina da educação médica. Dentro desse cenário, a contribuição deste trabalho baseia-se em testar as classes geradas por meio do *framework* ViMeT.

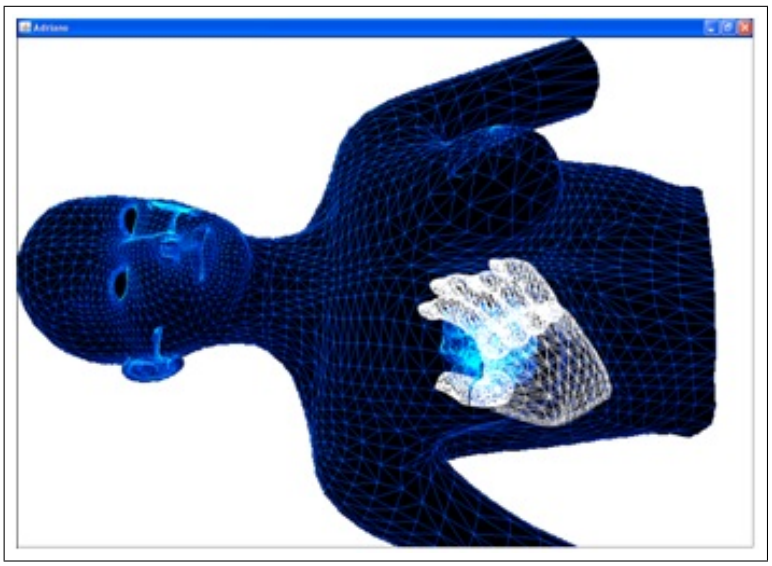

Figura 3.13: Exemplo de classe gerada pelo *framework* ViMeT.

3.4 Considerações finais

Neste capítulo foram apresentados fundamentos de RV, uma breve descrição de bibliotecas gráficas, conceitos relacionados a grafo de cena, juntamente com sua descrição, organiza-

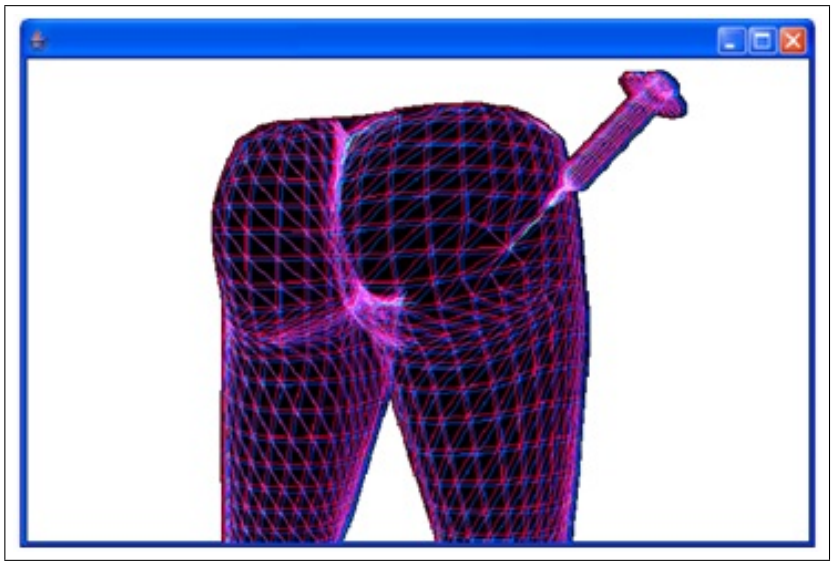

Figura 3.14: Exemplo de classe gerada pelo *framework* ViMeT.

ção, nós e utilização a fim de apoiar uma base teórica para o desenvolvimento do projeto proposto. Adicionalmente, foi apresentado o *framework* ViMeT. O próximo capítulo irá apresentar os critérios de teste baseados em grafos de cena que foram definidos, bem como a ferramenta de teste que apoia tais critérios.

Capítulo

4

Proposição de critérios de teste

Em todo processo de desenvolvimento de software é de vital importância que sejam realizadas atividades de teste. Tais atividades são realizadas seguindo técnicas e critérios de testes já definidas na literatura, como visto na Seção 2. Realizando estudos na literatura a fim de encontrar trabalhos que definem critérios de teste específicos para o desenvolvimento de sistemas de realidade virtual, não foram encontrados nenhum trabalho que define ou discute os possíveis critérios. Em alguns trabalhos, como o de (MATTIOLI, 2009), é discutido como desenvolver sistemas nesse domínio por meio de uma metodologia ou processo. Contudo, ao explicarem a fase de teste, não descrevem quais critérios de teste são necessários para que um sistema de realidade virtual esteja correto.

Em termos de análise de software, um grafo de cena pode ser comparado a um modelo que descreve o comportamento dos objetos em um AV de um sistema de RV considerando as características que possui, como mostrado no capítulo anterior.

Dentro deste contexto, o objetivo central deste trabalho consiste na proposição de critérios de teste baseados em um GC. Para isso foi estudado como os aspectos do AV, descritos no grafo de cena, podem ser utilizados para extrair requisitos a serem exercitados na atividade de teste. Assim, foram definidos cinco critérios de teste baseados no GC, e como forma de apoio a tais critérios, foi desenvolvida uma ferramenta de teste denominada *Virtual Environment Testing* (VETesting).

4.1 Critérios de teste baseados em grafo de cena

Como descrito no capítulo anterior, no GC é possível encontrar nós de agrupamento, nós internos e nós folhas. Analisando o GC de um AV, é possível derivar requisitos de teste, baseados nas características de sua construção. Transformações geométricas como uma rotação, uma translação ou uma escala, por exemplo, definidas em um nó interno são exemplos desses requisitos. Essas transformações geométricas deverão se propagar pelos nós filhos. Dessa forma, um caso de teste pode ser definido tendo como dado de entrada uma ou mais transformações geométricas a serem executadas no objeto correspondente e como saída esperada a confirmação visual de que ocorreu tal transformação.

Dentro desse contexto, foram definidos cinco critérios de teste para a derivação dos requisitos de teste baseados no GC:

- **Todos–Nós–Internos:** determina que, por meio de um ou mais casos de teste, todos os nós internos do GC que contemplem algum tipo de transformação (rotação, translação ou escala) sejam exercitados pelo menos uma vez;

- **Todos–Nós–Folhas:** exige que todos os nós folhas do GC que contemplem modificações de aparência (iluminação, sombreamento etc) sejam exercitados ao menos uma vez;

- **Todos–Caminhos–Ascendentes:** requer que, por meio de um ou mais casos de teste, todos os caminhos de um nó folha até um nó raiz, sejam exercitados;

- **Todos–Caminhos–Descendentes:** determina que, por meio de um ou mais casos de teste, todos os caminhos de um nó raiz até um nó folha, sejam exercitados;

- **Todas–Transformações:** exige que, por meio de um ou mais casos de teste, todas as transformações requeridas para cada nó interno do GC sejam exercitados pelo menos uma vez.

Vale ressaltar que os exemplos utilizados neste trabalho, bem como a implementação dos critérios de teste definidos, são baseados em Java3D. Contudo, os conceitos servem para qualquer biblioteca que utilizam o conceito de GC.

Para exemplificar, suponha-se que o GC, mostrado na Figura 4.1, contenha transformações definidas para os seus nós. No nó intermediário T_1G define-se uma transformação geométrica de rotação e no nó S_3 é definida uma modificação de cor, mostrada na Listagem 4.1. Nas linhas 5 e 6 é definida uma transformação para ser aplicada em uma instância da classe `TransformGroup` (linha 7), nas linhas 8 a 12 é definida uma cor para

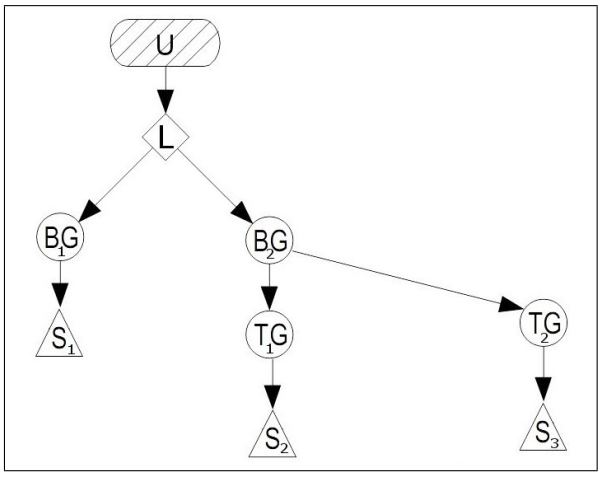

Figura 4.1: Grafo de cena representando um AV simples.

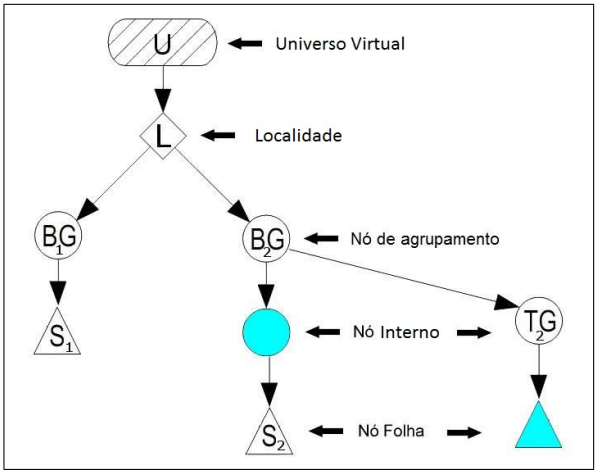

Figura 4.2: Identificação dos nós exercitados pelos critérios.

ser atribuída ao nó folha, instância da classe `Shape3D`, S_3 (linha 14) e, por fim, na linha 15 adiciona-se o nó folha S_3 ao `TransformGroup` T_1G.

A execução de um caso de teste que faça uma rotação do objeto relacionado com o nó T_1G e que modifique a cor do objeto relacionado ao nó S_3 faz com que sejam cobertos os nós correspondentes, conforme mostra a Figura 4.2. Esses critérios de teste definidos fazem com que à medida que forem adicionados e executados os casos de teste, todos os nós do GC que contemplem algum tipo de transformação ou modificação sejam cobertos. Um caso de teste que execute uma transformação geométrica e uma modificação em um determinado objeto na cena cobre os dois critérios definidos. Consequentemente, os nós que não possuem os requisitos de transformação ou modificação definidos, não serão requeridos com este tipo de critério de teste.

A cobertura dos nós internos denota que ao menos uma transformação definida no nó foi executada e, portanto, foi testada. Da mesma forma, a cobertura dos nós folhas, que contemplam modificações de aparência, refere-se a, no mínimo, uma modificação testada

e, individualmente, funcionam como esperado. Existem, porém, falhas que se manifestam somente quando sequências específicas de operações são executadas, e é justamente disso que tratam os critérios *Todos–Caminhos–Ascendentes* e *Todos–Caminhos–Descendentes*.

Em uma cena virtual é comum que seja atribuída mais de uma transformação a um objeto. Para isso é necessário mais nós internos, como é mostrado em outro GC representado na Figura 4.3. Contudo, a execução dessas transformações em determinada ordem pode revelar algum defeito. A Figura 4.3 exemplifica a execução de um caminho ascendente (C_1) *(translação, rotação e escala)* e um caminho descendente (C_2) *(escala, rotação e translação)*. Para se conseguir a cobertura de tal critério, é necessário executar os nós, por meio de um conjunto de casos de teste, na mesma ordem do caminho identificado. Quando a ordem do caso de teste estiver correta em relação à ordem que foi identificada, o caminho será coberto.

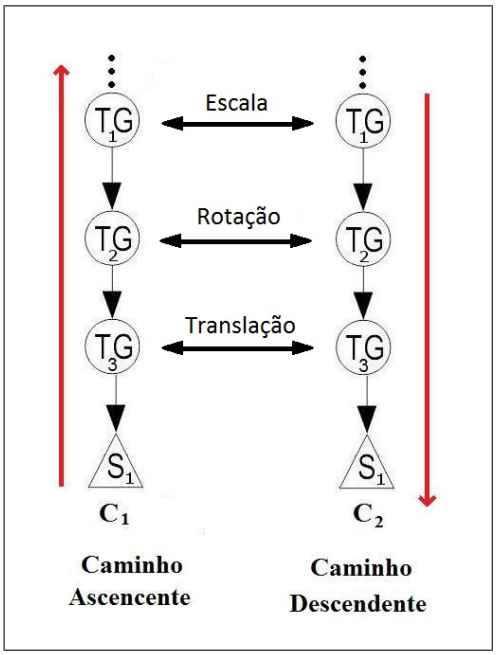

Figura 4.3: Exemplos de caminhos identificados no GC.

Na API Java3D, para se atribuir uma transformação geométrica a um nó interno, é necessário ter um objeto *Transform3D*, que é a estrutura responsável pela execução de uma transformação geométrica (Sun, 2000). Dessa forma, o critério *Todas–Transformações* exige que todos os objetos do tipo *Transform3D* atribuídos a um nó interno sejam exercitados ao menos uma vez. A Figura 4.4 mostra um nó interno passível de mais de uma transformação. De acordo com esse critério, todos os seus requisitos (transformações) têm que ser executados ao menos uma vez.

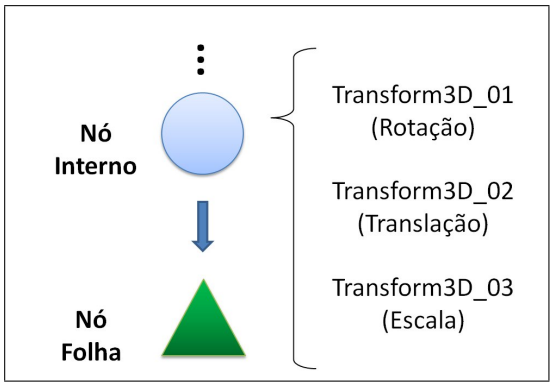

Figura 4.4: Exemplo de transformações atribuídas a um nó interno.

4.2 Virtual Environment Testing - VETesting

Como forma de apoio aos critérios de teste definidos, uma ferramenta denominada *Virtual Environment Testing* (VETesting) foi desenvolvida, utilizando a linguagem de programação Java, juntamente com a API Java3D, gerando-se uma plataforma de código aberto, que poderá ser explorada para ensino e pesquisa. Na Subseção 4.2.1 é apresentada a estrutura interna da ferramenta e na Subseção 4.2.2 são mostrados os aspectos de interação com o usuário.

4.2.1 Estrutura interna da ferramenta

A estrutura interna da ferramenta VETesting está dividida em alguns componentes (classes) principais: **VETesting, Loader, TestCase, Scan, Extractor, GraphView, Comparator, StructUniverse, StructLocale, StructGroup, StructInternal, StructLeaf**.

A execução da ferramenta pode ser dividida em: início do teste e adição de caso de teste. Ao iniciar o teste o componente **Loader** é responsável por carregar a classe selecionada para o teste, o componente **Scan** analisa o GC da classe selecionada e identifica os nós e o tipo destes nós. Ao mesmo tempo, o componente **Scan** chama o componente **Extrator** para extrair os requisitos do nós correspondentes, passando para o componente relacionado a estrutura do tipo dos nós identificados (**StructUniverse, StructLocale, StructGroup, StructInternal, StructLeaf**), e também envia para o componente **GraphView** as informações para a geração da imagem adequada do GC construído na classe em teste.

O componente **Test Case** é responsável por iniciar a execução de um caso de teste. O componente **Comparator** realizar uma comparação entre o nó corrente e o nó salvo no início do teste nas estruturas correspondentes. Ao comparar o nó, o componente **Comparator** envia o resultado da comparação para o componente **Scan** e o componente

Scan envia para o componente **GraphView** as informações dos nós para a geração da imagem do GC resultante da execução do caso de teste.

Na Figura 4.5 é representada a arquitetura da ferramenta. A seguir são comentadas as características dos seus componentes.

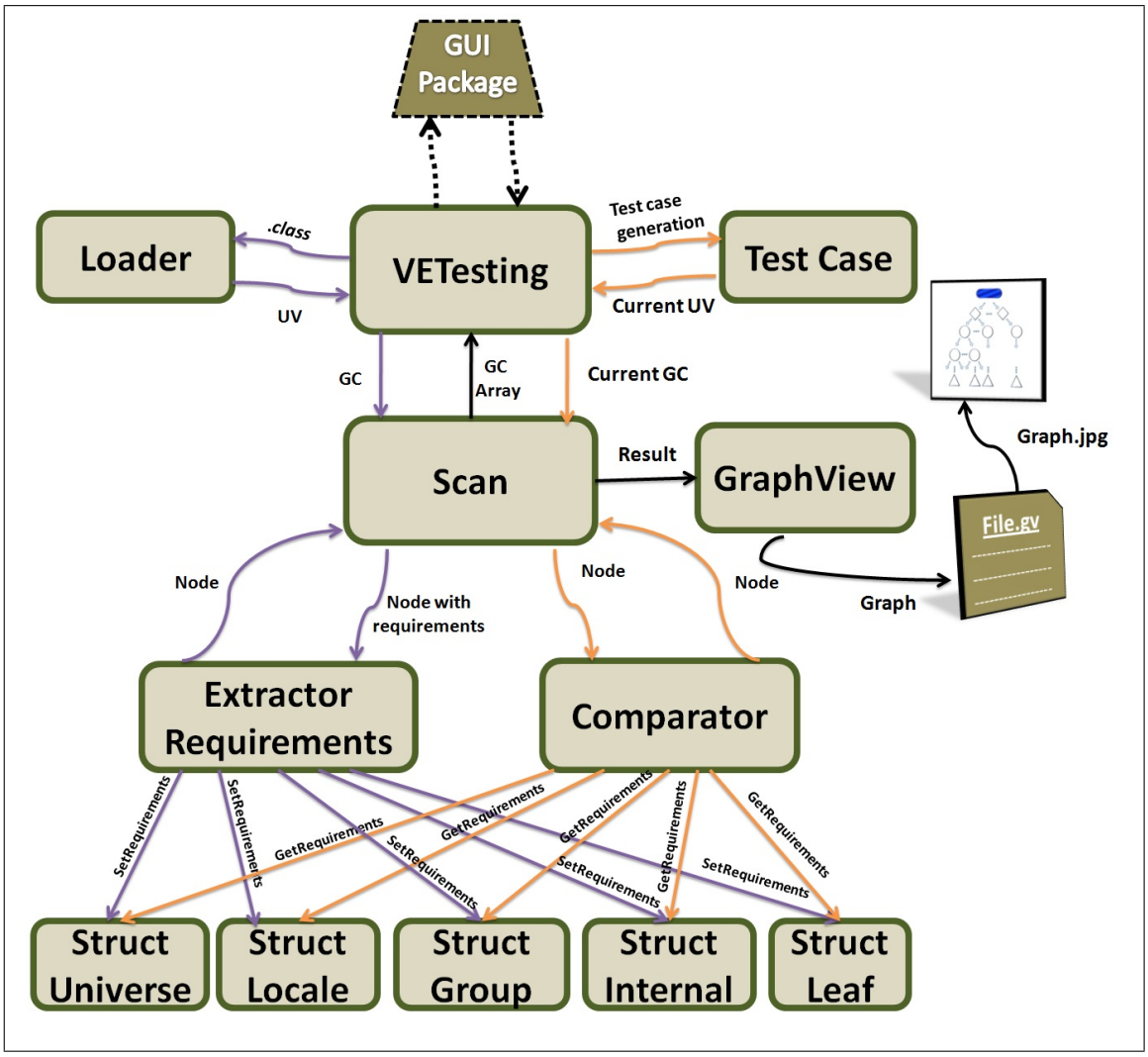

Figura 4.5: Componentes da ferramenta VETesting.

- **Loader:** esse componente é encarregado de fazer o carregamento da classe a ser testada. Tal componente recebe a classe do componente **VETesting**, carrega-a por meio de um *ClassLoader* específico e retorna o objeto que representa o universo virtual do GC da classe. Arquivos fonte (.java) contêm texto que é transformado, por meio da compilação, em um código multiplataforma (*bytecodes*) e armazenado em outro arquivo (.class). Um objeto do tipo *ClassLoader* consegue ter acesso a esse código, utilizando o pacote *java.lang*[1]. Na Listagem 4.3 é mostrado um trecho

[1]http://download.oracle.com/javase/1.4.2/docs/api/java/lang/package-summary.html

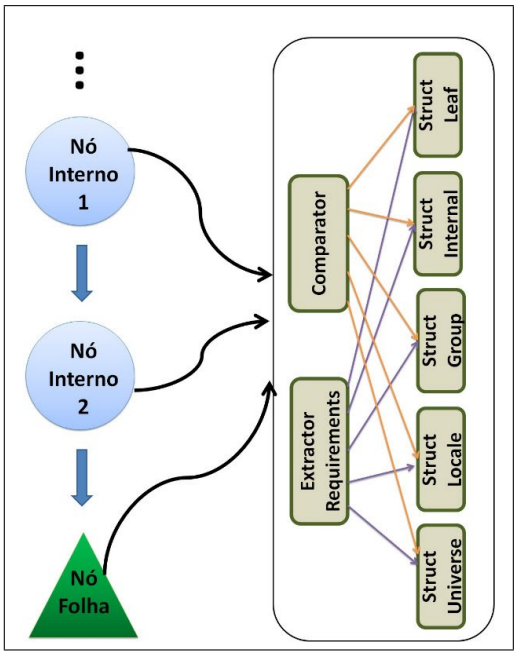

Figura 4.6: Exemplo da execução do componente **Scan**.

de código do *ClassLoader* utilizado para instanciar a classe selecionada pelo usuário (linha 6 e 7).

- **TestCase:** o componente **TestCase** é responsável por detectar as alterações realizadas no AV por meio da interação com o usuário e transmitir o GC atualizado com suas respectivas alterações para o componente **VETesting**.

- **Scan:** esse componente é encarregado de percorrer o GC recebido pelo componente **VETesting**. Se a execução da ferramenta for a de início, o nó que está sendo analisado é passado para o componente **Extractor**. Se a execução da ferramenta for a de adição de caso de teste, o nó que está sendo analisado é passado para o componente **Comparator**. Essa análise é feita para todos os nós do GC, como é mostrado na Figura 4.6. Adicionalmente, esse componente fornece as informações do nó correspondente para o componente **GraphView**.

- **Extractor:** tal componente é responsável por computar os requisitos de cada nó do GC. Assim, é verificada qual a instância que o nó correspondente representa e os seus respectivos requisitos de teste são fornecidos para o componente encarregado de armazená-los (**Structs**). A Listagem 4.2 detalha o trecho de código que armazena os requisitos do universo virtual (linha 2) e de todos os nós `Locale` do grafo (linha 6).

- **Comparator:** este componente tem a função de comparar o GC original da classe em teste com o GC alterado no caso de teste. O GC corrente, que foi alterado

pelo caso de teste, é recebido do componente **Scan** e comparado com o nó do GC inicial. Para verificar a cobertura dos requisitos o componente **Comparator** busca as informações necessárias no componente correspondente ao nó (**Structs**).

- **VETesting:** o componente **VETesting** é responsável por gerenciar a maioria dos outros componentes. Adicionalmente, esse componente se comunica com o pacote GUI (*Graphical User Interface*). As questões relacionadas ao pacote GUI serão discutidas na Subseção 4.2.2.

- **GraphView:** este componente é incumbido de criar a imagem para a exibição correspondente ao GC. Durante a varredura do componente **Scan**, informações relativas ao GC são informadas a esse componente. Ao final da varredura, cria-se um arquivo descritor de grafos, utilizando a linguagem DOT [2] e, por fim, é criado um arquivo imagem do GC.

- **Structs:** os componentes **StructUniverse, StructLocale, StructGroup, StructInternal e StructLeaf** são responsáveis por armazenarem os requisitos de cada nó correspondente. Este componente comunica-se apenas com os componentes **Comparator** e **Extractor**. Como já ressaltado, as **Structs** recebem as informações do nó do componente **Extractor** no qual podem ser requisitadas pelo componente **Comparator**.

Para que seja possível verificar a cobertura de um caso de teste, ou seja, quais foram os nós do GC exercitados por alguma ação no AV, é necessário instrumentar o GC de tal modo que se possa monitorar quais foram os nós exercitados na atividade de teste. Segundo Ammann e Offutt (2008), a instrumentação pode ser entendida como um código de programa adicional que não muda o comportamento funcional do programa. No caso do GC, instrumentar significa adicionar funcionalidade aos nós do grafo, de forma que se possa obter informação sobre quais nós foram "exercitados" durante um caso de teste.

É importante ressaltar que, para o contexto da ferramenta, os GCs não devem ter as suas estruturas alteradas durante a execução. Antes de iniciar o teste, o testador deverá ter a certeza de que todos os requisitos a serem cobertos estarão no AV, ou seja, o GC não é alterado (inclusão ou remoção de nós) durante a execução do ambiente virtual. Dessa forma, se a classe em teste possibilitar adicionar mais objetos no AV em tempo de execução, a ferramenta não é capaz de atualizar o GC. Então, a ferramenta possibilita adicionar casos de teste interagindo com os objetos que já estão no AV que está sendo testado.

[2] http://www.graphviz.org/doc/info/lang.html

4.2.2 Aspectos de interação da ferramenta

A ferramenta *VETesting* permite que sejam fornecidos casos de teste por meio da interação do usuário com o AV que está sendo testado. Na Figura 4.7 pode ser observado um diagrama que ilustra a execução da ferramenta.

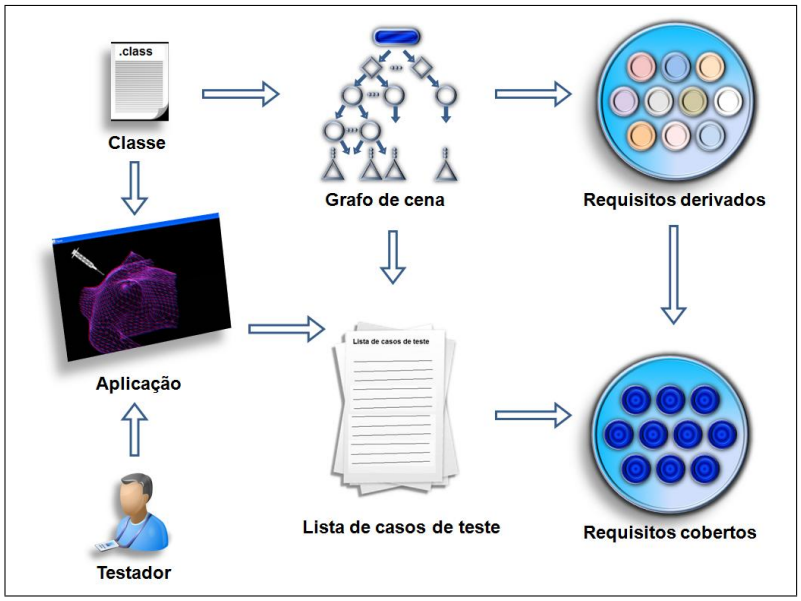

Figura 4.7: Diagrama de execução da ferramenta *VETesting*.

Inicialmente, o testador seleciona o botão *Open* na barra de menu *Test* da ferramenta ou as teclas de atalho (Ctrl+A). Uma nova janela é aberta para que seja selecionada a classe a ser carregada (Figura 4.8). A classe selecionada é instanciada e carregada. Na (Figura 4.9) pode ser visualizada a janela de uma classe em teste, já instanciada e em execução.

Após a classe ser carregada e instanciada, é necessário adicionar nas estruturas de dados internas da ferramenta todos os objetos virtuais do AV que irão participar da atividade de teste, pois a ferramenta não trata GC dinâmico, ou seja, GC que permitem incluir ou retirar nós em tempo de execução. Para iniciar a atividade de teste, o testador tem que selecionar o botão *Test Init* na barra de menu *Test* ou as tecla de atalho (Alt+I). Iniciando a atividade de teste, é mostrada uma lista com os nós do GC correspondente ao AV da classe em teste (Figura 4.10), além da imagem correspondente ao GC (Figura 4.11).

Para a criação de um novo caso de teste, deve-se selecionar o botão *Add Test Case* na barra de menu *Test* ou as tecla de atalho (Alt+C)(Figura 4.12). Nessa janela, o testador deve atribuir um nome ao caso de teste. Adicionalmente, essa janela possibilita descrever o caso de teste, ou seja, descrever quais foram as ações realizadas pelo testador durante a execução do caso de teste. Por exemplo, ao adicionar um novo caso de teste na ferramenta *VETesting*, o caso de teste é incluído na tabela *Test Case Id* na aba *Coverage*.

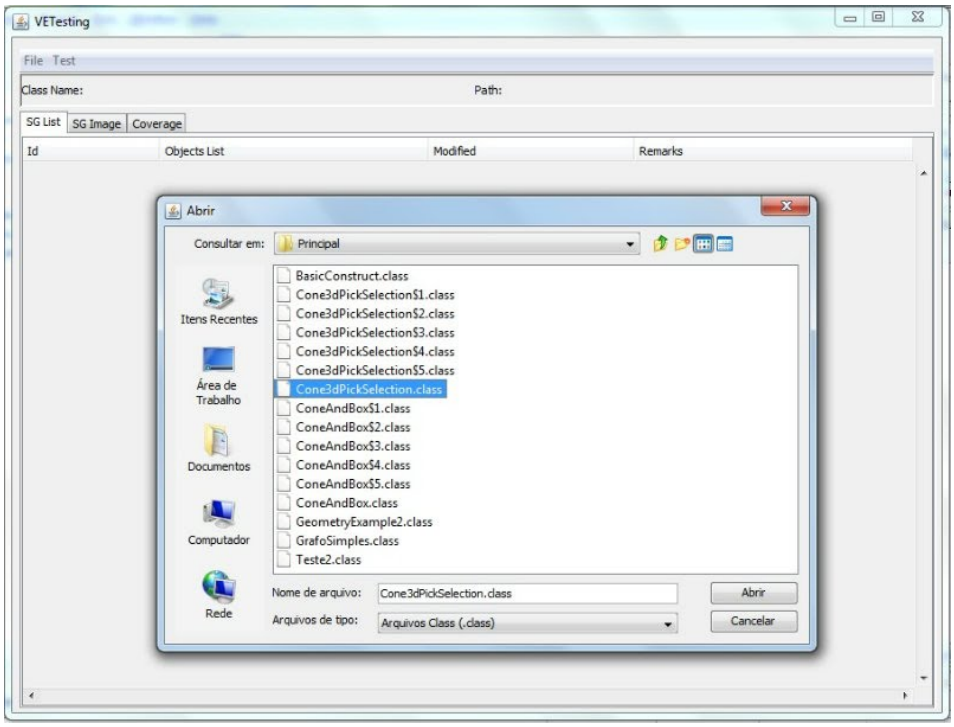

Figura 4.8: Seleção da classe a ser testada.

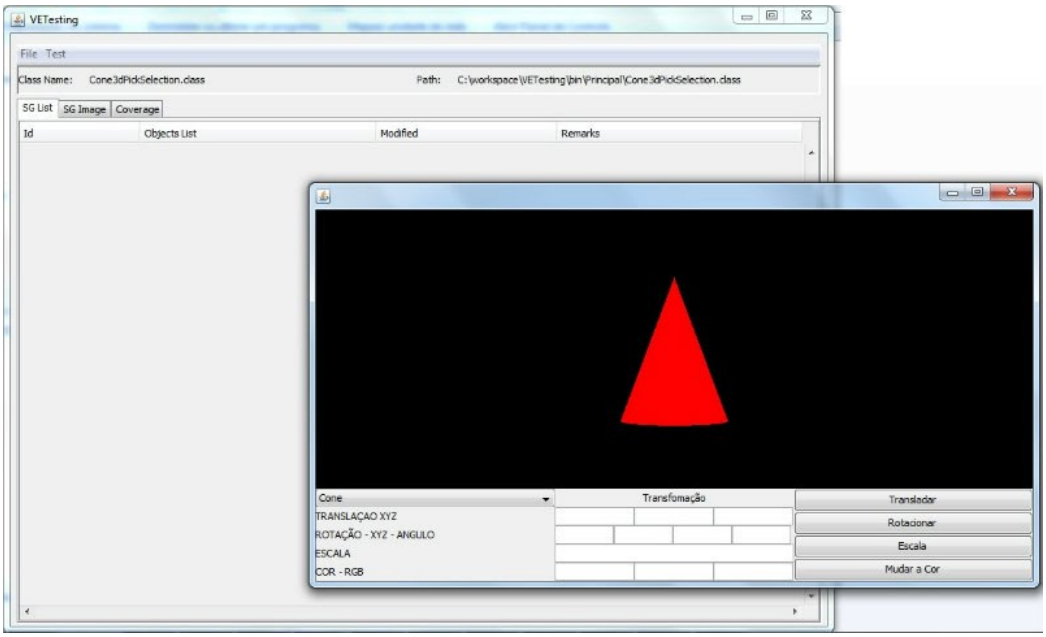

Figura 4.9: Classe carregada por meio da ferramenta *VETesting*.

Depois de adicionado um caso de teste, a ferramenta *VETesting* proporciona a visualização da cobertura do caso de teste por meio de uma lista dos nós do GC (Figura 4.13) ou por meio da imagem do GC correspondente ao critério de teste selecionado (Figura 4.14). Para visualizar a imagem do GC correspondente ao caso de teste de interesse, é necessário selecionar o caso de teste na tabela *Test Case Id*.

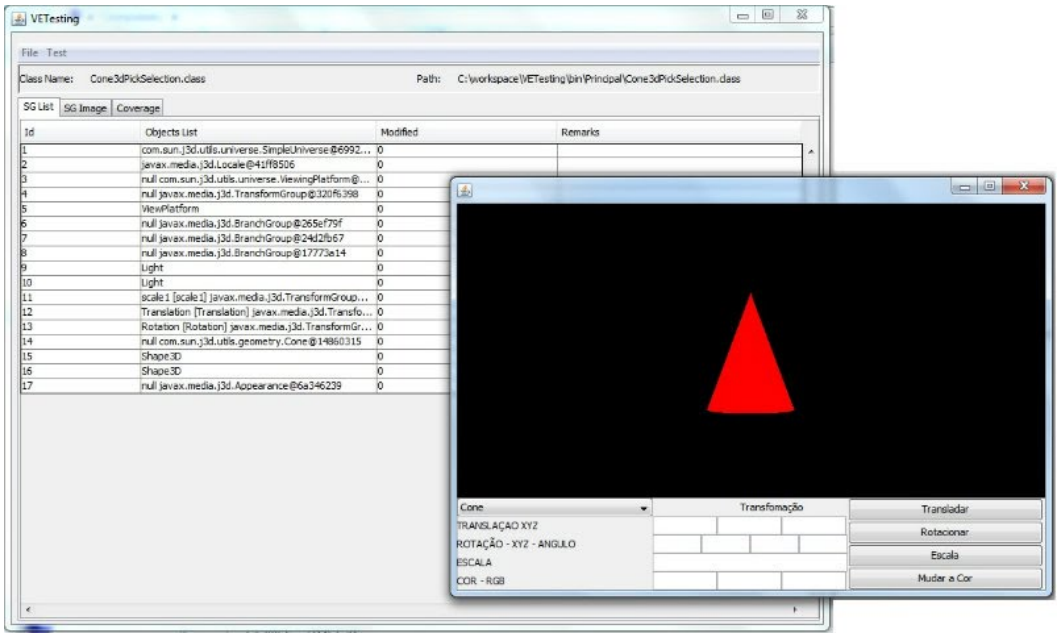

Figura 4.10: Início da atividade de teste e apresentação da lista de nós do GC em teste.

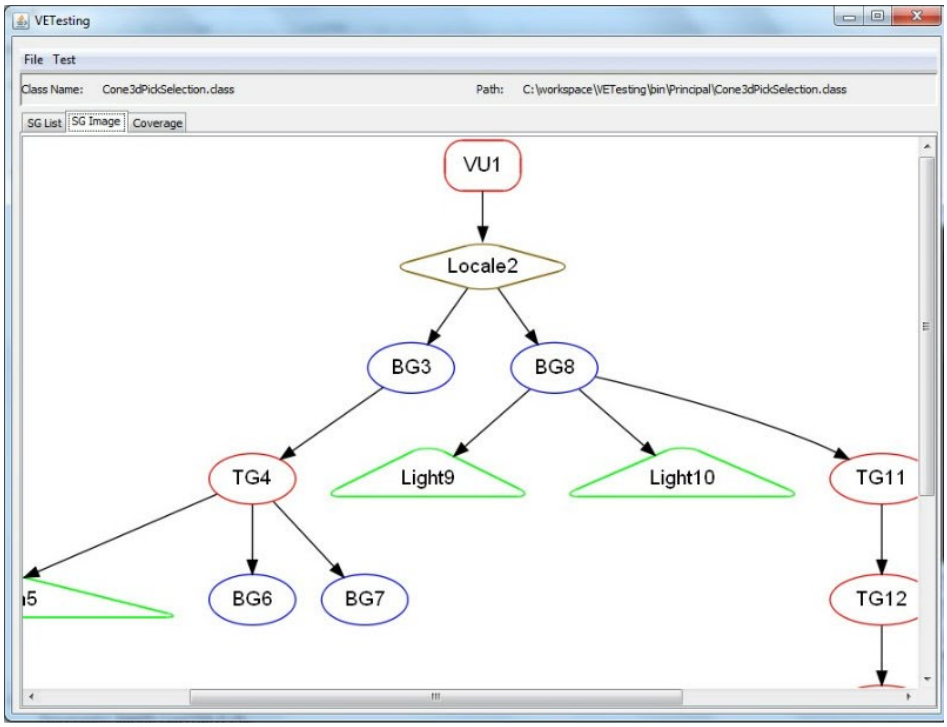

Figura 4.11: Visualização do GC inicial.

A ferramenta *VETesting* possibilita que o testador execute casos de teste diretamente no AV que foi instanciado. Dessa forma, o testador pode verificar a cobertura do caso de teste executado e pode adicionar mais casos de teste para cobrir os requisitos que ainda não foram cobertos. A ferramenta também proporciona a visualização da imagem do GC correspondente, assim como os nós que foram cobertos ou não pela execução dos casos de teste.

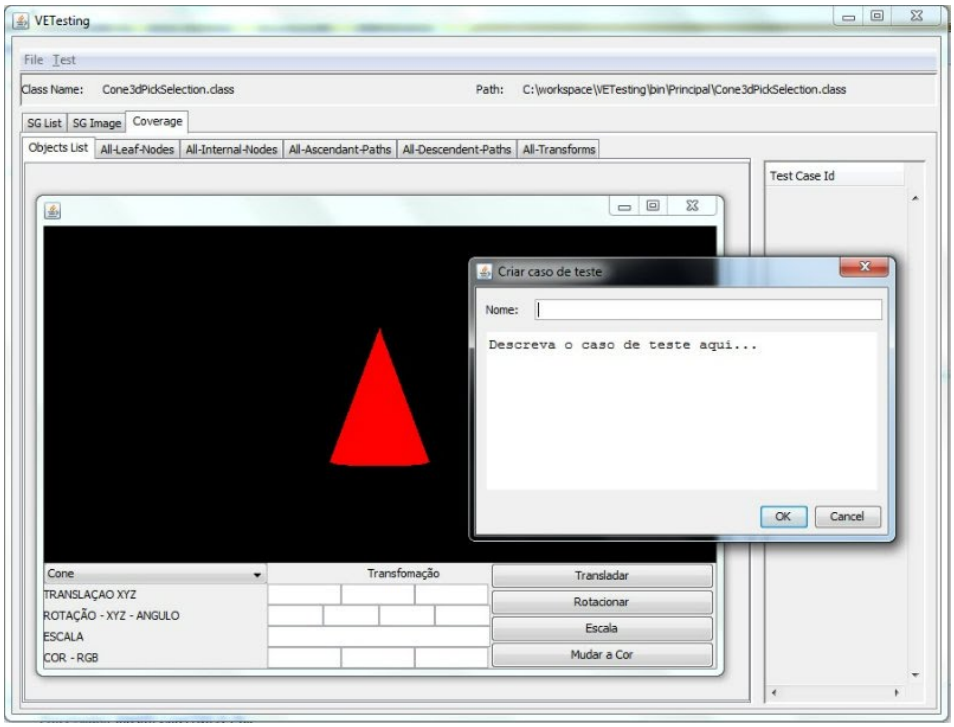

Figura 4.12: Tela de descrição do caso de teste.

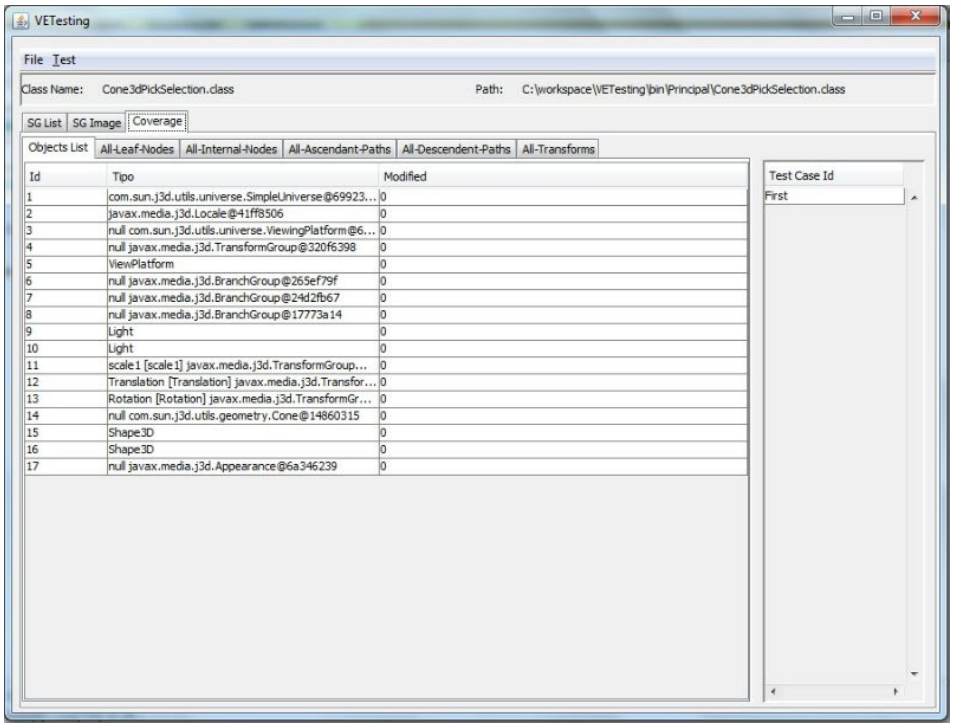

Figura 4.13: Lista de nós do GC.

4.3 Considerações finais

Neste capítulo, foram apresentados os critérios de teste baseados em grafos de cena definidos no presente trabalho. Tais critérios requerem que os nós do grafo de cena sejam

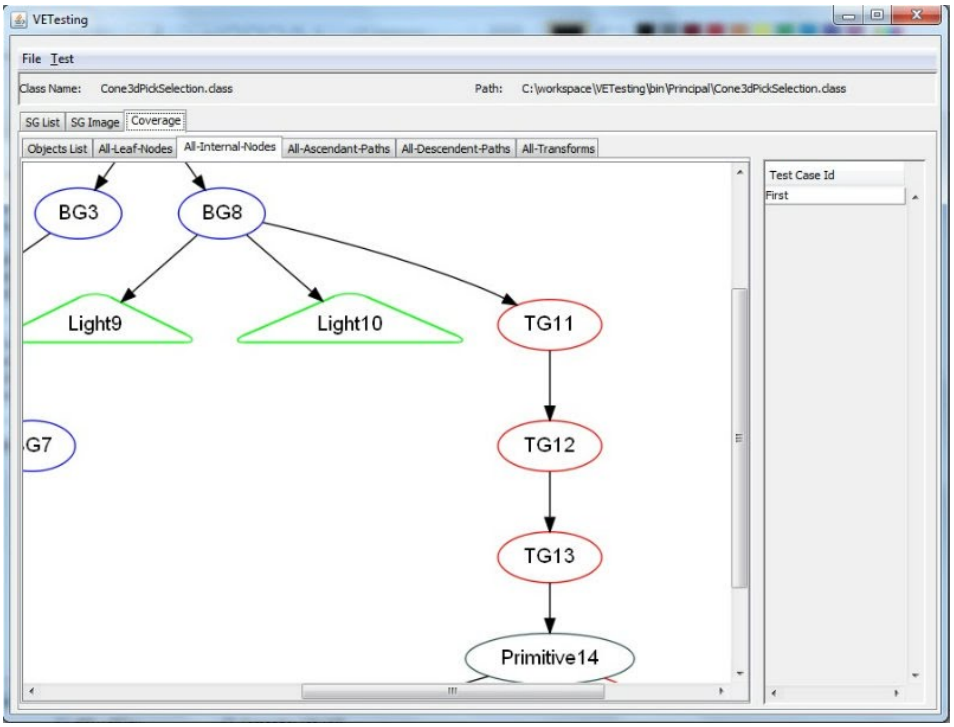

Figura 4.14: Visualização do GC correspondente ao critério.

exercitados por algum caso de teste ou conjunto de casos de teste. Adicionalmente, foi apresentada a automatização dos critérios definidos por meio do desenvolvimento de uma ferramenta de teste denominada *VETesting*. Com ela pode-se executar os testes de unidades em classes desenvolvidas com a API Java3D e verificar se estão de acordo com os seus requisitos. A ferramenta informa visualmente os nós que foram exercitados pelo caso de teste adicionado. Dessa forma o testador é capaz de identificar os nós do grafo que não foram exercitados e então projetar novos casos de teste para exercitá-los, melhorando dessa forma a qualidade do seu conjunto de teste. No próximo capítulo, são apresentados os resultados de três estudos de caso executados.

Listagem 4.1: Trecho de código em Java3D que define uma transformação a um transformGroup e uma aparência de cor a um nó folha.

```
0  public criaGrafo() {
1  ...
2   TransformGroup t1g= new TransformGroup();
3   branchGroup.addChild(t1g);
4   t1g.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
5   Transform3D rotate = new Transform3D();
6   rotate.rotX(Math.toRadians(-10.0));
7   t1g.setTransform(rotate);
8   Appearance app = new Appearance();
9        Material material = new Material(new Color3f(0.7f,0.1f,0.7f),
10       new Color3f(0.0f,0.5f,0.3f), new Color3f(0.7f,0.1f,0.7f),
11       new Color3f(1.0f,1.0f,1.0f), 60.0f);
12       app.setMaterial(material);
13  Shape3D s3 = new Shape3D();
14  s3.setAppearance (app);
15  t1g.addChild(s3);
16  ...
17  }
```

Listagem 4.2: Trecho do código que captura os requisitos do GC.

```
0  ...
1  public visiteUniverse(SimpleUniverse uv) {
2   Locale graphLocales = null;
3   list_.add(new StructUniverse(id++, 0, uv, 0, listLocale));
4   Enumeration<Locale> i = uv.getAllLocales();
5   while( i.hasMoreElements()) {
6     graphLocales = i.nextElement();
7     list_.add(new StructLocale(id++, 0, graphLocales, 0, listNode));
8     visiteLocale(graphLocales, 1);
9   }
10 }
11 ...
```

Listagem 4.3: Trecho de código de um Classloader em Java3D.

```
0  ...
1  public exClassLoader() {
2  public VirtualUniverse getUniverse(File arq){
3  try {
4   String nameClass = arq.getName();
5   String Path = arq.getParent();
6   URL[] urls=new URL[]{new URL("File:"+File.separatorChar+ Path)};
7   ClassLoader myClassLoader = GUI_Interface.class.getClassLoader();
8   Class cls = myClassLoader.loadClass(nameClass);
9   Object obj = cls.newInstance();
10  Field fieldlist[] = cls.getDeclaredFields();
11      for (int i = 0; i < fieldlist.length; i++) {
12      Field fld = fieldlist[i];
13      int mod = fld.getModifiers();
14      fld.setAccessible(true);
15      if( fld.get(obj) instanceof SimpleUniverse){
16      simpleU = (SimpleUniverse) fld.get(obj);
17 ...
18      }
19    }
20 }
```

Capítulo

5

Resultados e discussões

Como citado na Seção 1.2, um objetivo desse trabalho é a automatização dos critérios de teste definidos por meio de uma ferramenta de teste. Neste capítulo são apresentados os resultados pertinentes aos três estudos de caso realizados.

Para avaliar a adequação da ferramenta *VETesting* e dos critérios de teste definidos foram executados testes de unidade em classes geradas a partir do *framework* ViMeT e em uma classe de demonstração da API Java3D.

5.1 Estudo de caso I - ViMeT

Nesse estudo de caso, foi selecionada para teste uma aplicação gerada por meio do *framework* ViMeT. A aplicação selecionada, ilustrada na Figura 5.1, possui dois objetos virtuais que representam uma mama e uma seringa. Na Listagem 5.1 é mostrado o trecho de código da classe da aplicação selecionada para o teste, juntamente com seus principais métodos e atributos.

Na classe A, foram adicionados dois casos de teste por meio da interação dos objetos no AV. Essa interação foi realizada com equipamentos convencionais mouse e teclado. No primeiro caso de teste, foi realizada uma operação de aproximação (*zoom*) da câmera que faz parte da plataforma de visão da API Java3D (nó TG_4). No segundo caso de teste, foi realizada uma transformação de rotação no objeto virtual que representa a seringa (nó TG_{19}).

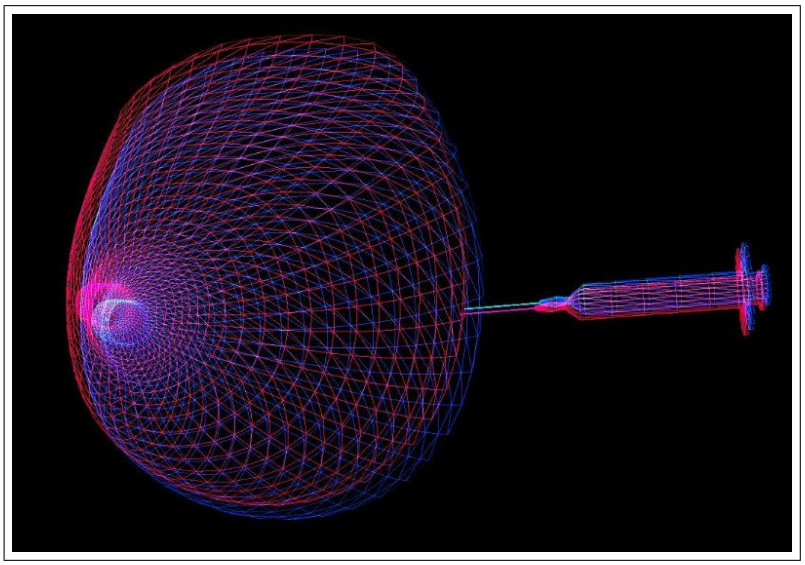

Figura 5.1: AV da classe A.

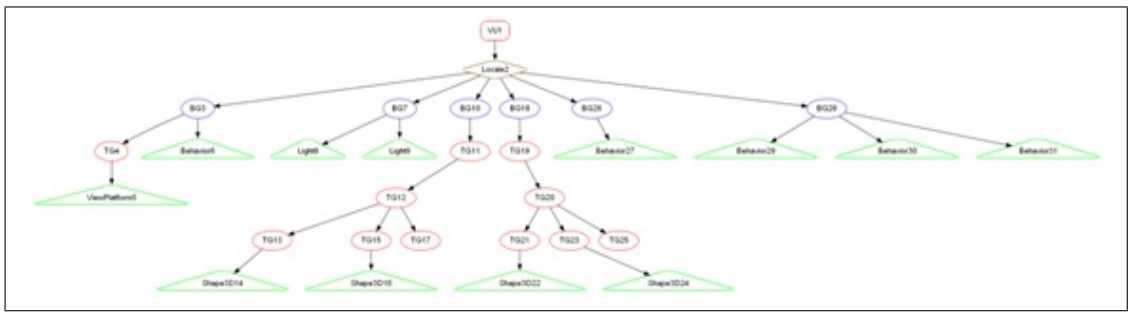

Figura 5.2: GC da classe A.

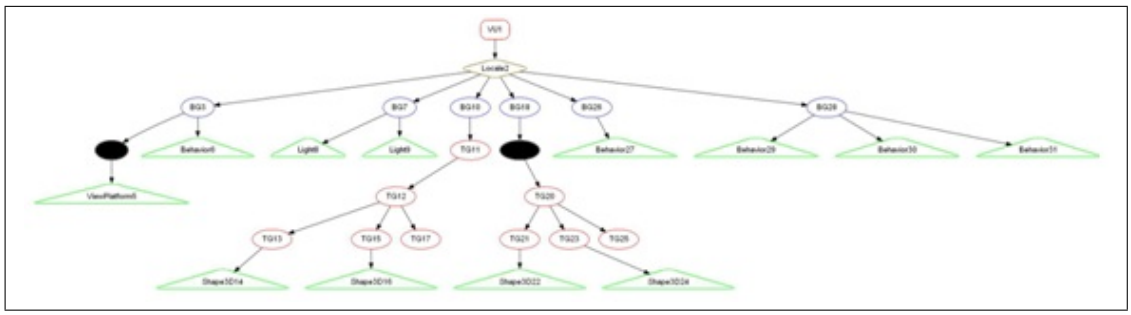

Figura 5.3: Nós cobertos pelos casos de teste executados na classe A.

Após a execução dos casos de teste na classe A, verifica-se que apenas dois nós do GC eram passíveis de serem exercitados por meio da interação do AV: um nó interno, responsável pela navegação da seringa no AV e o nó interno da plataforma de visão, também responsável pela navegação. Na Figura 5.2, pode ser visualizado o GC original da classe A e na Figura 5.3 é possível visualizar os nós que foram cobertos pelos casos de teste executados pela ferramenta, indicados na cor preta.

No GC da classe A encontram-se nós folhas $Ligh_8$ e $Ligh_9$, responsáveis pela iluminação do AV, nós folhas $Behavior_6$, $Behavior_{27}$, $Behavior_{29}$, $Behavior_{30}$ e $Behavior_{31}$,

responsáveis pelas ações no AV definidas pelo usuário e nós folhas $Shape3D_{16}$, $Shape3D_{22}$ e $Shape3D_{24}$, que são responsáveis por representar os objetos virtuais propriamente ditos. Além destes nós, encontram-se os nós internos TG_{12}, TG_{13}, TG_{15}, TG_{20}, TG_{21}, TG_{23}, que representam transformações relacionadas à estereoscopia, o nó TG_{20}, que está relacionado ao posicionamento do objeto que representa a mama virtual no AV em relação ao universo virtual e os nós TG_{17} e TG_{26}, que não são atribuídos a objetos virtuais no AV.

Muitos nós do grafo de cena que contemplavam algum tipo de transformação não foram exercitados. Nessa perspectiva, é fundamental o conhecimento sobre as limitações teóricas inerentes à atividade de teste, pois os elementos requeridos podem não ser executáveis e, em geral, determinar a não executabilidade de um dado requisito de teste envolve a participação do testador.

Nesse sentido, faz-se necessário realizar uma análise de cobertura de acordo com cada critério de teste definido:

- **Todos-Nós-Folhas:** após a execução dos casos de teste da classe A, verifica-se que nenhum nó folha foi executado. Os nós folhas $Ligh_8$ e $Ligh_9$ e os nós $ViewPlatform_5$, $Behavior_6$, $Behavior_{27}$, $Behavior_{29}$, $Behavior_{30}$, $Behavior_{31}$, $Shape3D_{14}$, $Shape-3D_{16}$, $Shape3D_{22}$ e $Shape3D_{24}$ não foram exercitados, pois a classe em teste não permite que os atributos dos respectivos nós folhas sejam alterados por meio da interação do AV. Nesse caso, esses nós são vistos como não executáveis. Os nós $Shape3D_{22}$ e $Shape3D_{24}$, por exemplo, que são responsáveis por representar o objeto seringa, não podem ter suas propriedades alteradas por meio da interação no AV. Uma alteração seria possível se a aplicação em teste permitisse trocar o objeto seringa por um outro objeto virtual e, essa troca fosse realizada pelo testador por meio da interação no AV.

- **Todos-Nós-Internos:** em relação aos nós internos, apenas dois nós que contemplavam algum tido de transformação foram exercitados por meio dos casos de teste adicionados em tempo de execução. Os nós TG_{12}, TG_{13}, TG_{15}, TG_{20}, TG_{21}, TG_{23}, o nó TG_{20} e os nós TG_{17} e TG_{26}, não foram exercitados, pois a aplicação em teste não permite que os atributos dos respectivos nós internos sejam alterados por meio da interação do AV, e também, são vistos como não executáveis. O nó TG_{20}, por exemplo, é responsável por uma transformação de escala no objeto que representa a seringa. Contudo, seu valor não pode ser alterado por meio da interação.

- **Todos-Caminhos-Ascendentes:** em relação aos caminhos ascendentes foram identificados quatro caminhos passíveis de serem executados: C_1 (TG_{13}, TG_{12} e TG_{11}); C_2 (TG_{15}, TG_{12} e TG_{11}); C_3 (TG_{21}, TG_{20} e TG_{19}); C_4 (TG_{23}, TG_{20} e TG_{19}). Após

a execução dos casos de teste, verifica-se que todos os caminhos derivados pelo critério não foram executados, pois os nós que compõem os caminhos não podem ser exercitados por meio da interação no AV. Para que um caminho seja coberto pelo critério, todos os nós que compõem um caminho devem ser exercitados, como mostra a Figura 5.4. Dessa forma, os caminhos identificados pelo critério foram definidos como não executáveis.

- **Todos-Caminhos-Descendentes:** em relação aos caminhos descendentes foram identificados quatro caminhos passíveis de serem executados: C_1 (TG_{11}, TG_{12} e TG_{13}); C_2 (TG_{11}, TG_{12} e TG_{15}); C_3 (TG_{19}, TG_{20} e TG_{21}); C_4 (TG_{19}, TG_{20} e TG_{23}). Após a execução dos casos de teste, verifica-se que todos os caminhos derivados pelo critério não foram executados, pois os nós que compõem os caminhos não foram exercitados em sua totalidade, apenas o nó TG_{19}. Para que um caminho seja coberto pelo critério, todos os nós que compõem um caminho tem que ser exercitado, como mostra a Figura 5.5. Dessa forma, os caminhos identificados pelo critério foram definidos como não executáveis.

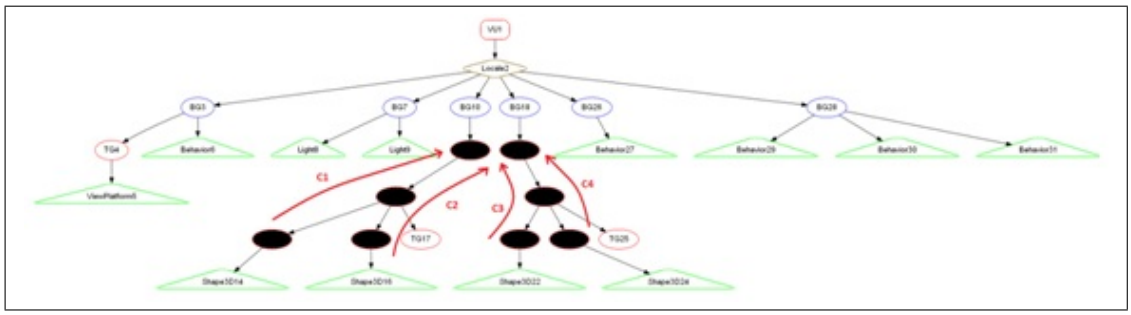

Figura 5.4: Caminhos ascendentes identificados na classe A.

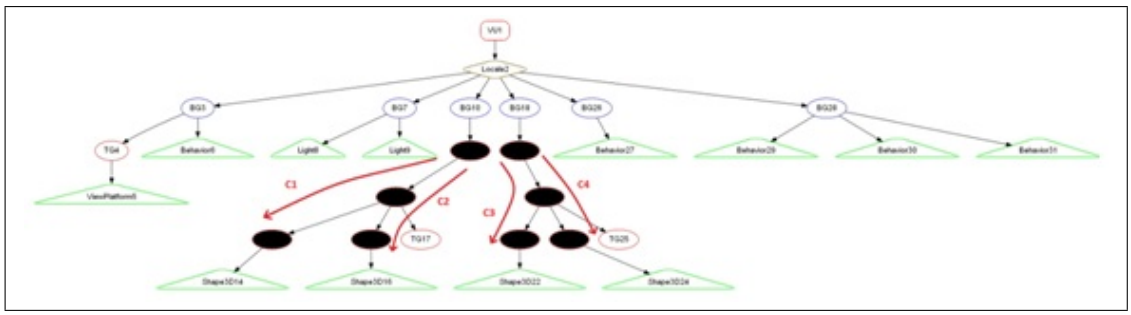

Figura 5.5: Caminhos descendentes identificados na classe A.

Após a análise de cobertura, pode-se concluir que alguns requisitos de teste como nós internos, nós folhas, caminhos ascendentes e caminhos descendentes, não podem ser exercitados por meio da interação no AV. Neste caso, é impossível executar um caso de

teste por meio da interação no AV que exercite esses requisitos. Portanto, os nós e os caminhos que compõem esses requisitos de teste são vistos como não executáveis, como é mostrado na Figura 5.6 e na Figura 5.7.

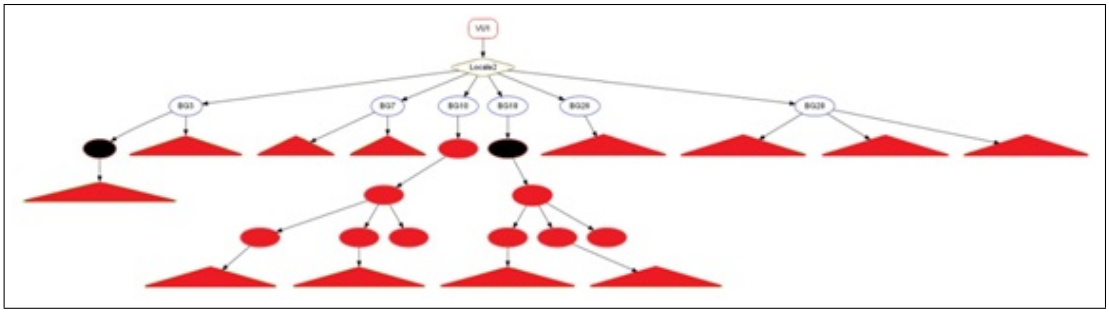

Figura 5.6: Nós não executáveis da classe A.

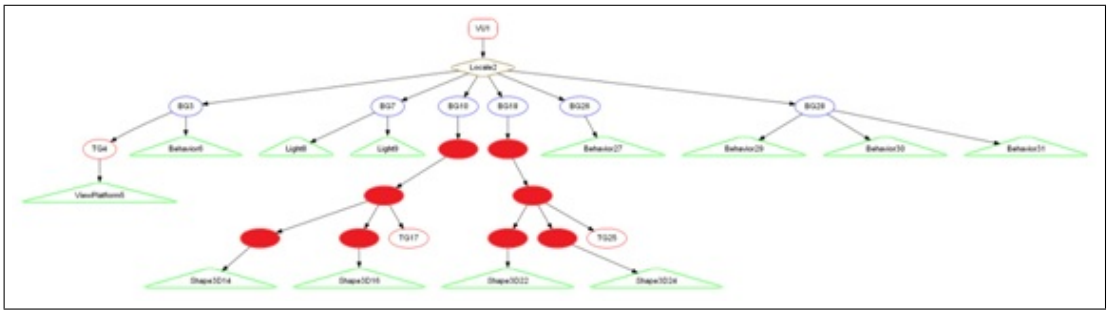

Figura 5.7: Nós que compõem os caminhos não executáveis da classe A.

Além disso, no GC encontram-se nós internos que não são atribuídos a nenhum nó folha. Dessa forma, pode-se pensar em uma estratégia de minimização dos nós de modo a elevar o desempenho do mesmo. Tal fato é relevante quando se trata de desenvolvimento de aplicações para simulação e treinamento em tempo real. Provavelmente, esses nós são voluntários para serem eliminados do GC.

A execução dos dois casos de teste selecionados pelos critérios não se garante que são suficientes para testar toda a aplicação. A execução de tais casos de teste apenas pode garantir que os requisitos dos objetos que foram exercitados por meio da interação no AV foram testados e funcionam como o esperado.

Assim, pode-se pensar também, em selecionar casos de teste que tenham outras formas de adicionar um conjunto de dados de entrada, e não só por meio da interação do testador com o AV. O conjunto de dados de entrada de um caso de teste na ferramenta *VETesting* é dado pela interação do usuário. Dessa forma, para que os nós que contemplem algum tipo de alteração do GC da aplicação em teste possam ser exercitados, a aplicação em teste deve proporcionar a alteração das propriedades dos nós por meio da interação no AV.

5.2 Estudo de caso II - ViMeT

Neste estudo de caso, também foi selecionada uma aplicação gerada por meio do *framework* ViMeT para a atividade de teste. A classe *B* (Figura 5.8) é composta por dois objetos virtuais, que representam uma mama, uma seringa e dezesseis objetos que estruturam uma mão virtual. Na Listagem 5.2 é mostrado o trecho de código resumido da classe da aplicação selecionada para o teste, juntamente com seus principais métodos e atributos.

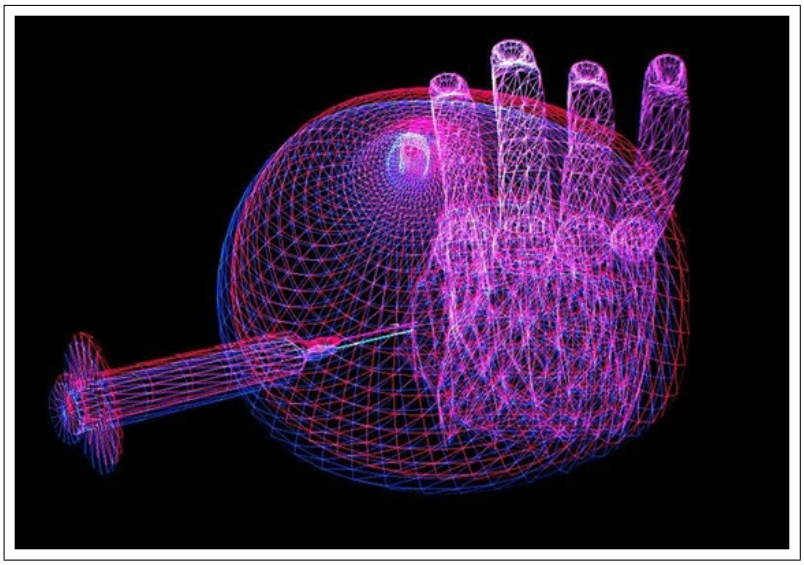

Figura 5.8: AV da classe *B*.

A classe *B* contém um número maior de nós que são passíveis de serem exercitados: um nó interno da seringa, responsável pela sua navegação no AV, um nó interno da plataforma de visão e mais cinco nós correspondentes às esferas de rotação que simulam os movimentos de abrir e fechar os dedos da mão virtual. Neste caso, foram adicionados três casos de teste por meio da interação dos objetos no AV. Essa interação também foi realizada com mouse e teclado.

No primeiro caso de teste, foi realizada uma transformação de rotação no objeto virtual que representa a seringa, no segundo caso de teste foi realizada uma operação de *zoom* na camera do AV, e por fim, foi adicionado um caso de teste que simula os movimentos dos cinco dedos da mão. Na Figura 5.9 pode ser visualizado o GC original da classe *B* e na Figura 5.10 é possível visualizar os nós cobertos pelos casos de teste executados na atividade de teste.

Após a execução dos casos de teste na classe *B*, verifica-se que dos 78 nós internos do GC analisados, apenas sete foram exercitados: um nó interno, responsável pela navegação da seringa no AV, um nó interno da plataforma de visão, também responsável pela navegação e cinco nós internos responsáveis pela navegação da mão virtual. Na Figura 5.9

pode ser visualizado o GC original da classe B e na Figura 5.10 é possível visualizar os nós que foram cobertos pelos casos de teste.

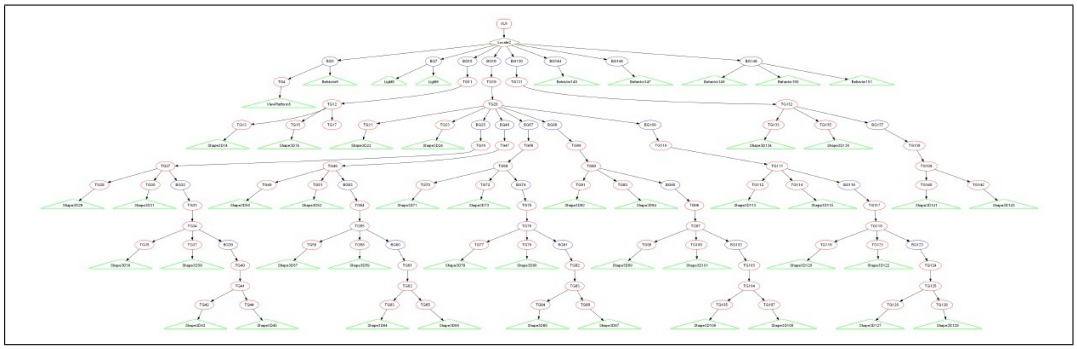

Figura 5.9: GC da classe B.

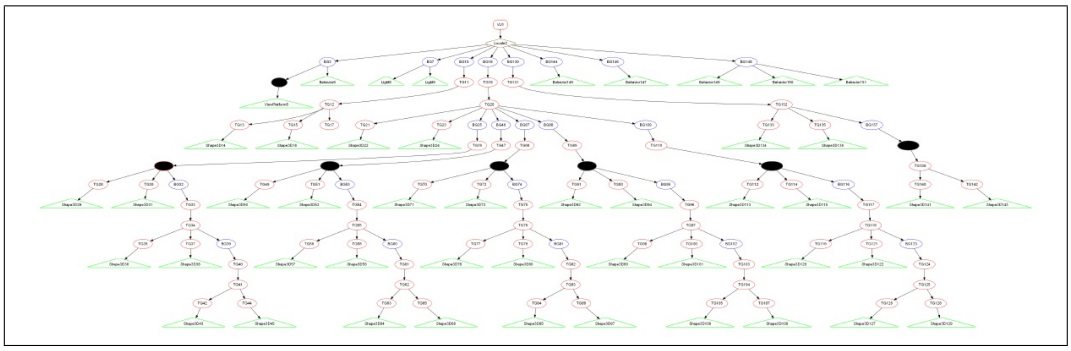

Figura 5.10: Grafo de cobertura da classe B.

No GC da classe B encontram-se os nós folhas $Ligh_8$ e $Ligh_9$, responsáveis pela iluminação do AV, os nós folhas $Behavior_6$, $Behavior_{145}$, $Behavior_{147}$, $Behavior_{149}$, $Behavior_{150}$ e $Behavior_{151}$, responsáveis pelas ações no AV definidas pelo usuário e mais 38 nós folhas do tipo $Shape3D$ que representam os objetos no AV propriamente dito.

Além destes nós, encontram-se os nós internos TG_{13}, TG_{15}, TG_{21}, TG_{23}, TG_{28}, TG_{30}, TG_{35}, TG_{37}, TG_{42}, TG_{44}, TG_{49}, TG_{51}, TG_{56},TG_{58}, TG_{63}, TG_{65}, TG_{70}, TG_{72}, TG_{77}, TG_{79}, TG_{84}, TG_{86}, TG_{91}, TG_{93}, TG_{98}, TG_{100}, TG_{105}, TG_{107}, TG_{112}, TG_{114}, TG_{119}, TG_{121}, TG_{126}, TG_{128}, TG_{133}, TG_{135}, TG_{140}, TG_{142}, que representam transformações relacionadas à estereoscopia. Já os nós TG_{11}, TG_{19}, TG_{20}, TG_{26}, TG_{27}, TG_{33}, TG_{34}, TG_{40}, TG_{41}, TG_{47}, TG_{48}, TG_{54}, TG_{55}, TG_{61}, TG_{62}, TG_{68}, TG_{69}, TG_{75}, TG_{76}, TG_{82}, TG_{83}, TG_{89}, TG_{90}, TG_{96}, TG_{97}, TG_{103}, TG_{104}, TG_{110}, TG_{111}, TG_{117}, TG_{118}, TG_{124}, TG_{125}, TG_{131}, TG_{132}, TG_{138} e TG_{139} estão relacionados ao posicionamento dos objetos virtuais que compõem o AV. O nó TG_4 é responsável pela navegação da câmera da plataforma de visão e o TG_{17} não é atribuídos a objeto virtual no AV.

Realizando uma análise de cobertura da classe B de acordo com cada critério de teste definido, têm-se os seguintes resultados:

- **Todos-Nós-Folhas:** após a execução dos casos de teste da classe B, verifica-se que nenhum nó folha foi executado. Semelhante aos nós folhas da classe A, a aplicação em teste não permite que os atributos relacionados aos nós sejam alterados por meio da interação no AV e, nesse caso, são vistos como não executáveis. Dessa forma, não há nenhum caso de teste que consiga exercitar tais nós. Ao passo que a aplicação em teste permita a alteração dos atributos do nó por meio da interação no AV, os requisitos do nó poderão ser cobertos por um caso de teste.

- **Todos-Nós-Internos:** em relação aos nós internos, sete nós que contemplavam algum tipo de transformação e eram passíveis de serem exercitados por meio da interação do AV foram exercitados por meio dos casos de teste. Na execução do primeiro caso de teste, foi realizada uma transformação de rotação no objeto virtual que representa a seringa. Destarte, este caso de teste exercitou o nó TG_{138}. No segundo caso de teste foi realizada uma operação de *zoom* na camera do AV, exercitando o nó TG_4, e por fim, foi adicionado um caso de teste que simula os movimentos dos cinco dedos da mão, exercitando os nós TG_{27}, TG_{50}, TG_{69}, TG_{90} e TG_{111}. Em relação aos outros nós internos que não foram exercitados, um nó não é relacionado a nenhum objeto no AV e os nós restantes não são passíveis de serem exercitados por meio da interação no AV. Dessa forma, todos estes nós restantes foram marcados como não executáveis. Se a aplicação em teste possibilitasse que não só o objeto relacionado às esferas de rotação realizasse transformação por meio da interação no AV, mas sim todos os objetos que estruturam os dedos, os seus respectivos nós seriam exercitados e possivelmente seriam por um caso de teste. Assim como, se a aplicação em teste possibilitasse que a mão como um todo navegasse no AV, o objeto relacionado à navegação da palma da mão seria exercitada e possivelmente seria coberta por um caso de teste que tivesse como dado de entrada tal navegação.

- **Todos-Caminhos-Ascendentes:** em relação aos caminhos ascendentes foram identificados quatro caminhos passíveis de serem executados: C_1 (TG_{13}, TG_{12} e TG_{11}); C_2 (TG_{15}, TG_{12} e TG_{11}); C_3 (TG_{21}, TG_{20} e TG_{19}); C_4 (TG_{23}, TG_{20} e TG_{19}). Após a execução dos casos de teste, verifica-se que todos os caminhos derivados pelo critério não foram executados, pois os nós que compõem os caminhos não foram exercitados. Para que um caminho seja coberto pelo critério, todos os nós que compõem um caminho tem que ser exercitado, como mostra a Figura 5.4 . Dessa forma, os caminhos identificados pelo critério foram definidos como não executáveis.

- **Todos-Caminhos-Descendentes:** em relação aos caminhos descendentes foram identificados quatro caminhos passíveis de serem executados: C_1 (TG_{11}, TG_{12} e TG_{13}); C_2 (TG_{11}, TG_{12} e TG_{15}); C_3 (TG_{19}, TG_{20} e TG_{21}); C_4 (TG_{19}, TG_{20} e TG_{23}).

Após a execução dos casos de teste, verifica-se que todos os caminhos derivados pelo critério não foram executados, pois os nós que compõem os caminhos não foram executados em sua totalidade, apenas o nó TG_{19}. Para que um caminho seja coberto por este critério, todos os nós que compõem o caminho tem que ser exercitado, como mostra a Figura 5.5 . Dessa forma, os caminhos identificados pelo critério foram definidos como não executáveis.

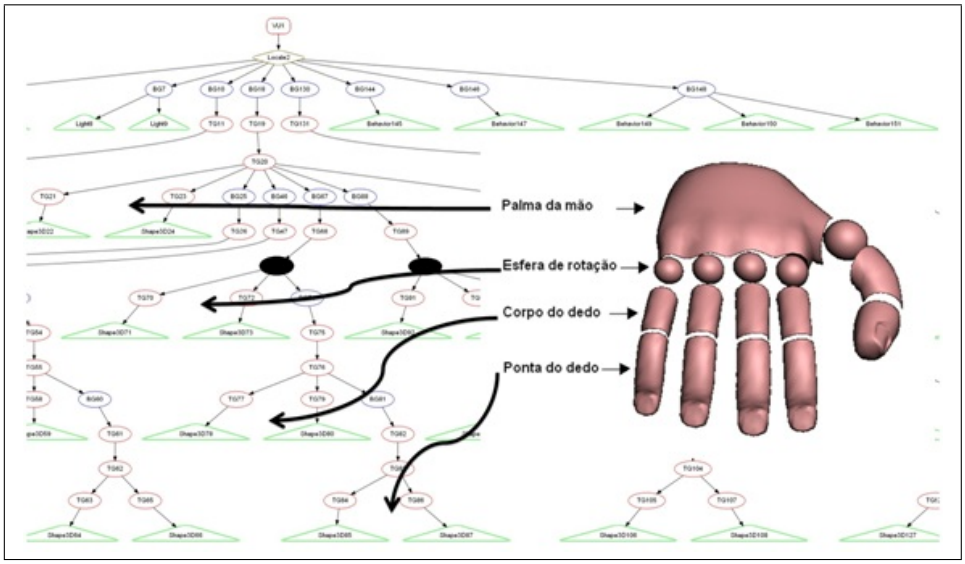

Figura 5.11: Hierarquia utilizada no *framework* ViMeT para representar a mão no AV.

Após a análise de cobertura realizada de acordo com cada critério definido, pode-se concluir que poucos requisitos de teste foram exercitados. Muitos nós marcados como não executáveis devem-se pelo fato da forma que o *framework* ViMeT representa seus objetos no AV. Na mão virtual, por exemplo, há 64 nós internos e apenas cinco foram exercitados pelos casos de teste. Um exemplo da quantidade de nós utilizados para representar um dedo de uma mão no AV pelo *framework* ViMeT é apresentado na Figura 5.11.

Na Figura 5.11, é possível visualizar dois dos cinco nós que foram exercitados pelos casos de teste que simulam os movimentos de abrir e fechar a mão. Se a aplicação em teste possibilitasse que as outras partes que estruturam a mão virtual pudessem ter suas propriedades alteradas por meio da interação no AV, seus respectivos nós também seriam exercitados pelo mesmo caso de teste.

Da mesma forma, se a aplicação em teste possibilitasse que as propriedades de iluminação do AV fossem alteradas por meio da interação no AV, poder-se-ia ter um caso de teste em que o seu dado de entrada fosse tal alteração. Sendo assim, os nós $Ligh_8$ e $Ligh_9$ seriam exercitados e possivelmente alcançariam a cobertura do critério *Todos–Nós–Folhas*.

5.3 Estudo de caso III - Demostração Java3D

Nesse estudo de caso, foi selecionado para a atividade de teste uma classe de demostração da API Java3D que pudesse, além do critério Todos-Nós-Internos, validar os critérios Todos-Nós-Folhas, Todos-Caminhos-Ascendentes e Todos-Caminhos-Descendentes. Essa classe (classe C) contém dois objetos virtuais, representados por um cone e um cubo (Figura 5.12). Tal classe permite realizar transformações geométricas (translação, rotação e escala) e transformação de aparência (mudança de cor) nos objetos por meio da interação no AV. Na Listagem 5.3 é mostrado o trecho de código resumido da classe da aplicação selecionada para o teste, juntamente com seus principais métodos e atributos.

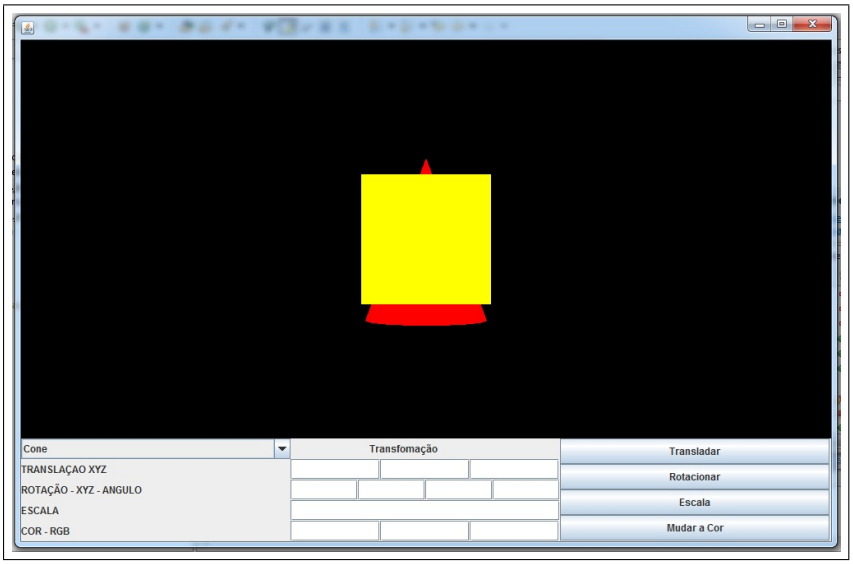

Figura 5.12: AV da classe C.

Na classe C foram adicionados quatro casos de teste. No primeiro caso de teste, foi realizada uma sequência de três transformações geométricas como rotação, translação e escala no objeto virtual que representa o cubo. No segundo caso de teste, foi realizada uma alteração de cor no objeto representado cubo. No terceiro caso de teste, foi realizada uma sequência de três transformações geométricas como rotação, translação e escala no objeto virtual que representa o cone, e por fim, no quarto caso de teste foi realizada uma mudança de cor no objeto representado pelo cone.

Na Figura 5.13 pode ser visualizado o GC original da classe C e na Figura 5.14 é possível visualizar os nós cobertos pelos casos de teste executados na atividade de teste.

No GC da classe C encontram-se os nós folhas $Ligh_9$ e $Ligh_{10}$, responsáveis pela iluminação do AV, o nó folha $ViewPlatform_5$, responsável pela plataforma de visão da API Java3D, os nós folhas $Shape3D_{15}$, $Shape3D_{16}$, $Shape3D_{22}$, $Shape3D_{23}$, $Shape3D_{24}$, $Shape3D_{25}$, $Shape3D_{26}$, $Shape3D_{27}$, representam as partes que estruturam os objetos

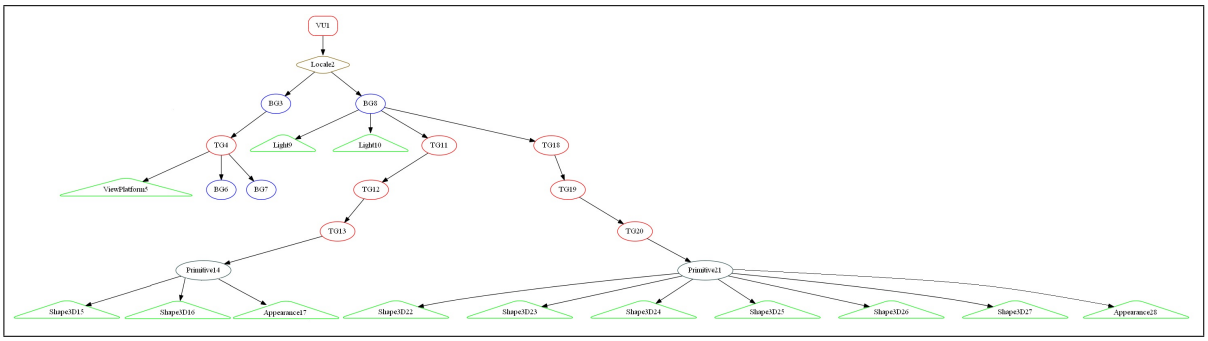

Figura 5.13: Grafo de cena original da classe C.

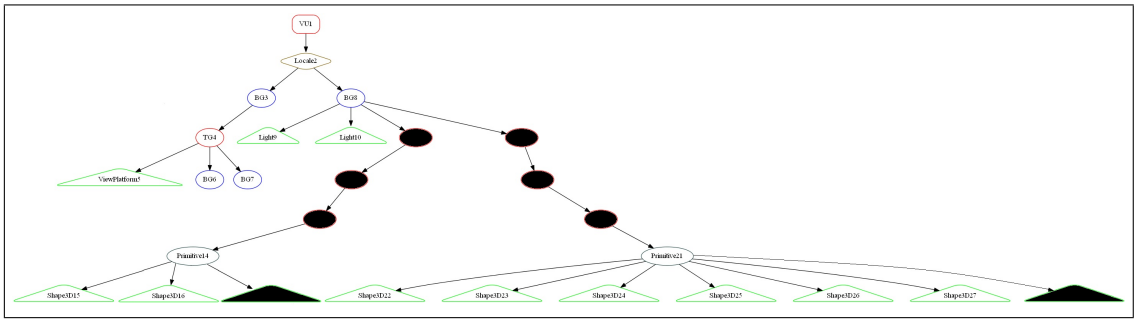

Figura 5.14: Nós cobertos pelos casos de teste executados no GC da classe C.

primitivos no AV e os nós $appearance_{17}$ e $appearance_{28}$, são responsáveis pela aparência dos objetos no AV. Além destes nós, encontram-se os nós internos, como o TG_4 no qual é responsável pela navegação da camera da plataforma de visão, os nós internos TG_{11}, TG_{12}, TG_{13}, TG_{18}, TG_{19} e TG_{20}, que são responsáveis por realizarem as transformações geométricas dos objetos virtuais que compõem o AV.

Realizando uma análise de cobertura da classe C de acordo com cada critério de teste definido, têm-se os seguintes resultados:

- **Todos-Nós-Folhas:** após a execução dos casos de teste da classe C, verifica-se que dois nós folhas foram exercitados. Para cobrir o nó $Appearence_{17}$ foi adicionado um caso de teste que realiza uma operação de modificação de cor no objeto que representa o cone. Semelhante ao cone, para cobrir o nó $Appearence_{28}$, também foi adicionado um caso de teste realizando uma operação de modificação de cor no objeto que representa o cubo. Os outros nós folhas da aplicação em teste não são alterados por meio da interação no AV, nesse caso, são vistos como não executáveis.

- **Todos-Nós-Internos:** em relação aos nós internos, seis dos sete nós que eram passíveis de serem exercitados por meio da interação do AV da aplicação. Os nós TG_{11}, TG_{12} e TG_{13} foram exercitados por meio de um caso de teste que teve como dados de entrada uma operação de rotação, translação e uma escala. Já os nós TG_{18}, TG_{19} e TG_{20} foram exercitados por meio de um caso de teste que teve como

dados de entrada uma operação de escala, translação e rotação. O nó TG_4, que está relacionado ao campo de visão da API Java3D não foi exercitado, pois os seus atributos não podem ser alterados por meio da interação no AV. Assim, foi denotado como não executável. Observa-se que nos estudos de caso anteriores, tal nó que relacionado ao campo de visão da API Java3D foi exercitado e coberto por um caso de teste. Neste caso especificamente, tal nó não foi capaz de se exercitar por meio da interação.

- **Todos-Caminhos-Ascendentes:** em relação aos caminhos ascendentes foram identificados dois caminhos passíveis de serem executados: C_1 (TG_{13}, TG_{12} e TG_{11}) e C_2 (TG_{20}, TG_{19} e TG_{18}). Para cobrir os caminhos identificados, foram executados dois casos de teste que tem como dados de entrada, sequências de operações. No primeiro foi executada uma escala, uma translação e uma rotação no objeto representado pelo cubo e no segundo foi executada uma escala, uma translação e uma rotação no objeto representado pelo cone. Na Figura 5.15 podem ser visualizados os caminhos executados pelos casos de teste.

- **Todos-Caminhos-Descendentes:** em relação aos caminhos descendentes foram identificados dois caminhos passíveis de serem executados: C_1 (TG_{11}, TG_{12} e TG_{13}) e C_2 (TG_{18}, TG_{19} e TG_{20}). Semelhante à cobertura dos critérios Todos-Caminhos-Ascendentes, foram executados dois casos de teste, tendo como dados de entrada, sequências de operações. No primeiro foi executada uma operação de rotação, uma de translação e uma de escala no objeto representado pelo cubo e no segundo foi executada uma operação de rotação, uma de translação e uma de escala no objeto representado pelo cone. Na Figura 5.16 podem ser visualizado os caminhos executados pelos casos de teste.

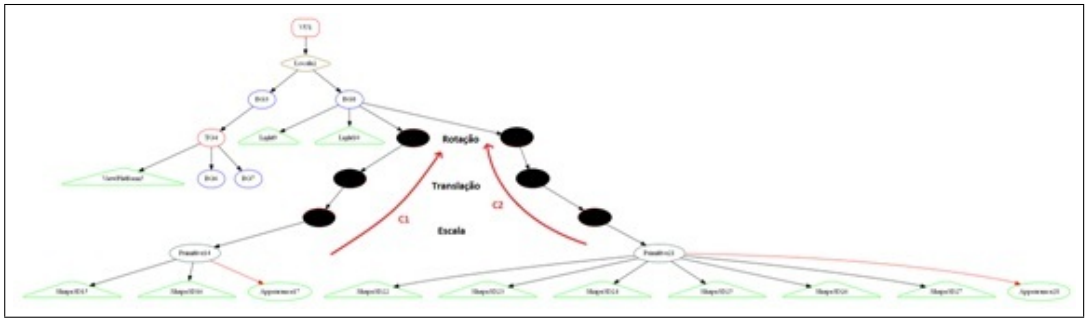

Figura 5.15: Caminhos ascendentes executados pelos casos de teste na classe *C*.

Após a análise de cobertura, pode-se concluir que todos os caminhos, tanto ascendentes, quando descendentes, foram executados por meio de um conjunto de teste. Porém, alguns requisitos de teste como nós internos, nós folhas não foram exercitados, porque

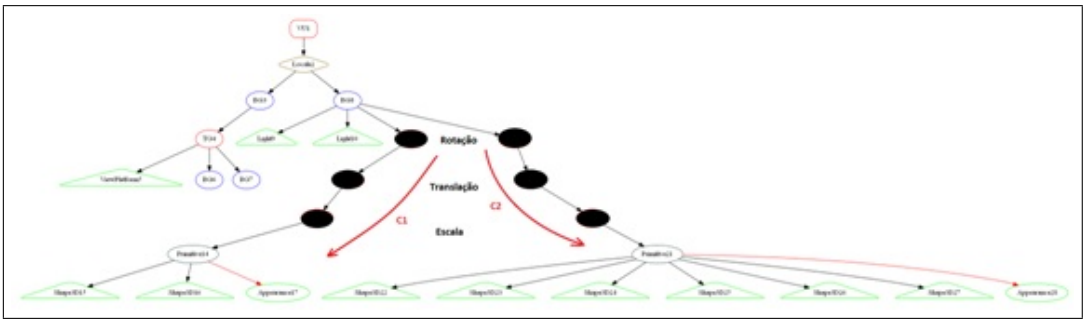

Figura 5.16: Caminhos descendentes executados pelos casos de teste na classe *C*.

a aplicação em teste não permite que tais nós sejam exercitados por meio da interação no AV. Em um exemplo simplista, o usuário poderia configurar a iluminação do AV por meio da interação do AV. Dessa forma o usuário estaria exercitando os nós folhas $Ligh_9$ e $Ligh_{10}$, que responsáveis pela iluminação do AV. Contudo, a aplicação em teste não permite realizar tal operação por meio da interação no AV. Portanto, os nós que compõem esses requisitos de teste são vistos como não executáveis, como é mostrado na Figura 5.17.

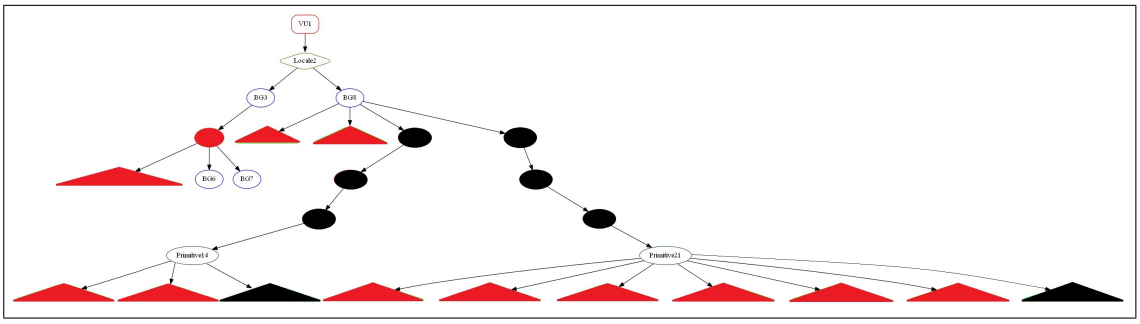

Figura 5.17: Nós não executáveis do GC da aplicação em teste.

Analisando a Figura 5.17, pode-se perceber que há dois nós (BG_6 e BG_7) que estão soltos no GC. Tais nós são do tipo *BrachGroup* e não faz sentido estarem ali, pois estes nós são utilizados apenas para a organização dos dados e neste contexto eles não estão exercendo essa função. Nesse sentido, tais nós são voluntários a serem eliminados do AV.

A execução dos quatro casos de teste selecionados pelos critérios não garantem que todos os nós que contemplem algum tipo de transformação sejam exercitados, porém, garantem que todos os requisitos correspondentes aos nós que representam os objetos virtuais contidos no AV foram exercitados e consequentemente foram testados ao menos uma vez.

5.4 Considerações finais

Este capítulo detalhou a execução de três estudos de caso. Dois estudos foram executados em classes geradas a partir do *framework* ViMeT e um estudo foi executado a partir de uma aplicação de demostração da API Java3D.

Na Tabela 5.1, é apresentada análise de cobertura em relação aos critérios dos nós as aplicações que foram testadas. Na classe A, dos onze nós internos analisados, apenas dois foram exercitados e dos 12 nós folhas analisados, nenhum foi exercitado. Na classe B, dos 78 nós internos analisados, apenas sete foram exercitados e dos 47 nós folhas analisados, nenhum foi exercitado. Já classe C, dos sete nós internos analisados, seis foram exercitados e dos 13 nós folhas analisados, apenas 2 foram exercitados.

Tabela 5.1: Análise de cobertura em relação aos critérios dos nós.

Classe	Qtd de nós folhas analisados	Nós folhas cobertos	Qtd de nós internos analisados	Nós internos cobertos
A	11	2	12	-
B	78	7	47	-
C	7	6	13	2

Na Tabela 5.2 é apresentada análise de cobertura dos critérios Todos-Caminhos-Ascendentes e Todos-Caminhos-Descendentes em relação as aplicações que foram testadas. Dessa forma, apenas a classe C executou os caminhos identificados por meio de casos de teste derivados dos critérios. Assim, tanto os dois caminhos ascendentes identificados, quanto os dois caminhos descendentes identificados, foram executados.

Tabela 5.2: Análise de cobertura em relação aos critérios dos caminhos.

Classe	Qtd de caminhos (Asc)	Cobertura (Caminhos Asc.)	Qtd de caminhos (Desc)	Cobertura (Caminhos Desc)
A	-	-	-	-
B	-	-	-	-
C	2	2	2	2

Como resultado, a Tabela 5.3 relata uma nova análise de cobertura das classes testadas excluindo os nós não executáveis. Dos dois nós internos da classe A, os dois foram cobertos e não houveram nós folhas executáveis. Na classe B, dos sete nós internos, os sete foram cobertos e, também, não houveram nós folhas executáveis. Já na classe C, dos seis nós internos analisados, os seis foram cobertos e dos dois nós folhas analisados, os dois foram cobertos.

Na atividade de teste de unidade, o implementador é obrigado a executar casos de teste que cubram cada critério de teste definido e analise manualmente determinado requisitos que porventura é caracterizado como não executável. Dessa forma, é fundamental o

Tabela 5.3: Análise de cobertura excluindo os nós não executáveis.

Classe	Qtd de nós folhas analisados	Nós folhas cobertos	Qtd de nós internos analisados	Nós internos cobertos
A	2	2	-	-
B	7	7	-	-
C	6	6	2	2

conhecimento sobre as limitações teóricas inerentes à atividade de teste, pois os elementos requeridos podem não ser executáveis e, em geral, determinar a não executabilidade de um dado requisito de teste envolve a participação do testador.

Nos estudos de caso, pode-se notar um número elevado de requisitos não executáveis, principalmente, em relação aos critérios relacionados a caminhos do GC. Além disso, com poucos casos de teste simples, foi possível cobrir os demais elementos requeridos. Não se dispõe de uma forma objetiva de se avaliar a qualidade desses conjuntos de teste pela falta de outros critérios e ferramentas que avaliem conjuntos de teste nesse domínio. Porém, intuitivamente, parece que tais conjuntos poderiam ser melhorados. Ou seja, os critérios propostos são úteis ao identificarem casos de testes necessários para o teste da aplicação, contudo devem ser complementados com outras formas que permitam a criação conjuntos mais completos de teste.

Outros estudos precisam ainda ser conduzidos para que tais conjuntos de teste e, possivelmente, os próprios critérios de teste e a ferramenta de apoio sejam mais bem compreendidos e melhorados.

No próximo capítulo, são apresentadas as conclusões desse trabalho.

Listagem 5.1: Trecho do código da classe em teste com métodos e atributos.

```
0  ...
1  public ClassA(Canvas3D c) {
2  super(c, true);
3              //Instanciaçao dos atributos
4              //- Instanciaçao dos objetos
5  objetos = new Object3D[2];
6  objetos[0] = new DeformableObject("objectsDatabase/mama_ac.obj",
7                                Object3D.OCTREE+Object3D.DEFORMATION+
8                                Object3D.STEREOSCOPY, ObjectFile.RESIZE);
9  objetos[1] = new RigidObject("objectsDatabase/seringa_pronta.obj",
10                               Object3D.STEREOSCOPY, ObjectFile.RESIZE);
11             //Adiçao dos objetos no universo
12 this.add(objetos[0]);
13 this.add(objetos[1]);
14                 // - Instanciaçao da deformaçao
15 Parameters p = new Parameters();
16 def = ((DeformableObject)objetos[0]).getDeformation();
17 p.setForce(3.0f, 0.0f, 0.0f);
18 p.setMass(300.0f);
19 p.setDamping(0.7f);
20 p.setConstSpring(0.3f);
21 def.setParameters(p);
22                 // - Instanciaçao da detecçao de colisão
23 cd = ((DeformableObject)objetos[0]).getCollisionDetector();
24 Octree)cd).setPrecision(0.0010);
25 cd.setCollisionListener(def);
26 BranchGroup bgTemp = new BranchGroup();
27 bgTemp.addChild(cd);
28 super.localeNode.addBranchGraph(bgTemp);
29                 // - Estereoscopia
30 super.setEyeOffset(0.017f);
31             // - Dispositivo
32 Mouse m = new Mouse(objetos[1].getMotionTransform(),
33                                         super.localeNode);
34 objetos[0].setScale(new Vector3d (1.0f,1.0f,1.0f), 1);
35 [0].setTranslation(new Vector3d (0.0f,0.0f,0.0f));
36 [0].setRotation(new AxisAngle4d (0.0f,0.0f,0.0f,0.0f));
37 objetos[1].setScale(new Vector3d (0.5f,0.5f,0.5f), 1);
38 objetos[1].setTranslation(new Vector3d (0.6f,0.5f,0.9f));
39 objetos[1].setRotation(new AxisAngle4d (0.0f,0.0f,1.0f,0.8f));
40 }
41 ...
```

Listagem 5.2: Trecho do código da classe em teste com métodos e atributos.

```
0  ...
1  public Teste5(Canvas3D c) {
2   super(c, Boolean.TRUE);
3   // Instanciaçao dos atributos
4   // - Instanciaçao dos objetos
5   this.objetos = new Object3D[19];
6   this.objetos[0] = new DeformableObject("objectsDatabase
7   mama_ac.obj", Object3D.OCTREE + Object3D.DEFORMATION
8             + Object3D.STEREOSCOPY, ObjectFile.RESIZE);
9   this.objetos[1] = new RigidObject("objectsDatabase/
10        seringa_pronta.obj", Object3D.STEREOSCOPY,
11             ObjectFile.RESIZE);
12  this.objetos[2] = new RigidObject("objectsDatabase/
13        conjuntoMao/PalmaDaMao.obj", Object3D.STEREOSCOPY,
14                        ObjectFile.RESIZE);
15  ...
16  // Adiçao dos objetos no universo
17  super.add(this.objetos[0]);
18  super.add2(this.objetos[1]);
19  super.add2(this.objetos[2]);
20  ...
21  // Seringa
22  this.objetos[18].getPositionTransform().
23              addChild(this.objetos[1].getBranchGroup());
24  ...
25  // - Instanciaçao da deformaçao
26  Parameters parameters = new Parameters();
27  this.def = ((DeformableObject) this.objetos[0]).getDeformation();
28  parameters.setForce(3.0f, 0.0f, 0.0f);
29  parameters.setMass(300.0f);
30  parameters.setDamping(0.7f);
31  parameters.setConstSpring(0.3f);
32  this.def.setParameters(parameters);
33  // - Instanciaçao da detecçao de colisão
34  this.cd = ((DeformableObject) this.objetos[0]).
35                getCollisionDetector();
36  ((Octree) this.cd).setPrecision(0.0010);
37  this.cd.setCollisionListener(this.def);
38  BranchGroup bgTemp = new BranchGroup();
39  bgTemp.addChild(this.cd);
40  super.getLocaleNode().addBranchGraph(bgTemp);
41  // - Estereoscopia
42  super.setEyeOffset(0.017f);
43  List<Object3D> transpickers = new ArrayList<Object3D>();
44  K = new Keyboard(super.getLocaleNode(),
45        new KeyboardHandBehavior(transpickers, new Hand()));
46  new Mouse(this.objetos[1].getMotionTransform(),
47            super.getLocaleNode());
48  ...
49  }
50  ...
```

Listagem 5.3: Trecho do código da classe em teste com métodos e atributos.

```
0  ...
1  private BranchGroup createSceneGraph() {
2        BranchGroup root = new BranchGroup();
3        root.addChild(createLight(0.0f, 0.0f, -1.0f));
4        root.addChild(createLight(0.3f, -0.3f, -0.3f));
5        cone = new Cone( 0.15f, 0.4f, Cone.ENABLE_APPEARANCE_MODIFY|
6     Cone.GENERATE_NORMALS, createAppearance(1.0f,0.0f,0.0f) );
7        cone.setAppearance(0,createAppearance(1.0f,0.0f,0.0f));
8        box = new Box( 0.15f,0.15f, 0.15f, Box.ENABLE_APPEARANCE_MODIFY|
9     Box.GENERATE_NORMALS, aparencia(1.0f,1.0f,0.0f) );
10 ...
11 //Botão para executar uma transformação de translação
12 JButton botaoTrans = new JButton ("Transladar");
13 botaoTrans.addActionListener(new ActionListener() {
14       public void actionPerformed(ActionEvent event)
15       { try{
16                 x = Float.parseFloat(transladaX.getText());
17                 y = Float.parseFloat(transladaY.getText());
18                 z = Float.parseFloat(transladaZ.getText());
19                 if (combo.getSelectedIndex() == 0)
20                 {
21                  myTrans3D.setTranslation(new Vector3d(x,y,z));
22                  translation1.setTransform(myTrans3D);
23                 }else if (combo.getSelectedIndex() == 1)
24                {
25                    myTrans3D2.setTranslation(new Vector3d(x,y,z));
26                        rotation1.setTransform(myTrans3D2);
27                        }else if (combo.getSelectedIndex() == 2)
28                         {
29                          myTrans3D2.setTranslation(new Vector3d(x,y,z));
30                          scale1.setTransform(myTrans3D2);
31                         }}
32               catch(Exception ex){}
33        }});
34       BotaoExecucao.add(botaoTrans);
35    ...
36    //Botão para executar uma transformação de cor
37       JButton botaoCor = new JButton("Mudar a Cor");
38       botaoCor.addActionListener(new ActionListener() {
39       public void actionPerformed(ActionEvent event)
40       { try{
41                 r = Float.parseFloat(corR.getText());
42                 g = Float.parseFloat(corG.getText());
43                 b = Float.parseFloat(corB.getText());
44                 if(combo.getSelectedIndex() == 0)
45                 {
46                 cone.setAppearance(createAppearance(r, g, b));
47                 }else if(combo.getSelectedIndex() == 1)
48                        {
49                        box.setAppearance(aparencia(r,g,b));
50                        }}
51                        catch (Exception e) {}
52                        }});
53       BotaoExecucao.add(botaoCor);
54    ...
```

Capítulo

6

Conclusões

6.1 Contribuições

A condução deste trabalho visou contribuir com as pesquisas relacionadas a definições de critérios de teste estruturais, dando continuidade aos trabalhos desenvolvidos pelo grupo do Laboratório de Engenharia de Software do ICMC/USP (LabES)[1] e do grupo de pesquisa em Realidade Virtual do Laboratório de Aplicações de Informática em Saúde (LApIS) [2] da EACH/USP.

A contribuição desse trabalho foi dada por meio de um estudo sobre as principais técnicas e critérios de testes existentes, destacando as técnicas de teste estruturais no contexto de Realidade Virtual. Adicionalmente, foi realizado um estudo sobre métodos de representação e análise de Ambientes Virtuais e Grafos de Cena. Foram apresentados conceitos relacionados a Realidade Virtual e o uso de Grafos de Cena no desenvolvimento de aplicações que fazem uso dessa tecnologia, bem como sobre técnicas de teste no domínio específico de RV. Além disso, foram investigados e estabelecidos critérios de teste baseados no grafo de cena e como forma de apoio, tais critérios foram automatizados por meio de uma ferramenta de teste denominada *VETesting* e, por fim, foi realizado um estudo de caso com um *framework* de RV desenvolvido no LApIS, denominado *Virtual Medical Training* ViMeT e uma classe de demonstração da API Java3D.

[1] http://www.labes.icmc.usp.br/site/
[2] http://www.each.usp.br/lapis/

6.1.1 Definição de critérios de teste

Uma das contribuições principais desse trabalho foi a definição de critérios de teste baseados em grafos de cena. Assim, foram definidos cinco critérios de teste que derivam requisitos de teste baseados nas características de um GC: **Todos–Nós–Internos**, **Todos–Nós–Folhas**, **Todas–Transformações**, **Todos–Caminhos–Ascendentes** e **Todos–Caminhos–Descendentes**.

Com a cobertura desses critérios pode-se garantir que as transformações ou uma sequência de transformações definidas pelos nós do grafo foram testadas pelo menos uma vez e, individualmente, funcionam como esperado.

Os critérios propostos são pertinentes ao identificarem casos de testes necessários para o teste da aplicação, contudo devem ser complementados com outras formas que permitam a criação de conjuntos mais completos de teste.

6.1.2 Ferramenta de teste

Como forma de apoio aos critérios de teste que foram definidos, uma ferramenta denominada *Virtual Environment Testing* (VETesting) foi desenvolvida, utilizando a linguagem de programação Java, juntamente com a API Java3D, gerando-se uma plataforma de código aberto, que poderá ser explorada para ensino e pesquisa.

Esse trabalho está inserido no contexto do grupo do Instituto Nacional de Ciência e Tecnologia – Medicina Assistida por Computação Científica (INCT–MACC) no qual se refere ao desenvolvimento de sistemas de RV com aplicação à área de saúde, e nele, tradicionalmente, é utilizada a API Java3D como biblioteca gráfica, o que implicou, também, na utilização da API Java3D neste trabalho.

Por meio da ferramenta é possível verificar se os nós, que representam objetos virtuais na cena, satisfazem seus requisitos conforme foram especificados. A ferramenta é capaz de apontar se, durante a realização de um teste, não forem executadas transformações definidas por meio de especificações para determinados objetos, representados por nós. Nesse contexto, a ferramenta apoia, não somente a construção de grafos de cena, mas também sua correção. Além disso, a ferramenta que está sendo desenvolvida proporciona certa facilidade para compreender o GC correspondente ao AV que será testado. Geralmente, esse trabalho é feito de forma manual.

6.1.3 Avaliação empírica

Para avaliar os critérios de teste definidos e a ferramenta de apoio desenvolvida, foram realizados estudos de caso utilizando classes disponibilizadas pelo *framework* ViMeT e classe de demostração da API Java3D.

Nos estudos de caso, pode-se notar um número elevado de requisitos não executáveis. Ainda não há uma forma objetiva de se avaliar a qualidade desses conjuntos de teste pela falta de outros critérios e ferramentas que avaliem conjuntos de teste nesse domínio.

Com esses estudos, foi possível verificar que o GC pode ser utilizado para derivar requisitos de teste. No entanto, outros estudos precisam ainda ser conduzidos para que os critérios de teste definidos sejam melhor compreendidos e melhorados ou complementados com outras formas que permitam a criação de conjuntos mais completos de teste.

6.2 Trabalhos futuros

A partir dos resultados obtidos, prevê-se a continuidade deste trabalho por meio das seguintes ações:

- incluir na ferramenta *VETesting* a possibilidade de se testar GC de outras bibliotecas gráficas de modo a não se restringir a biblioteca gráfica Java3D;
- desenvolver ou incluir na ferramenta *VETesting* a execução de casos de testes que exercitem requisitos que não são passíveis de serem executados por meio da interação no AV. Dessa forma, todos os nós do GC que foram selecionados pelos critérios poderão ser exercitados sem a interação do usuário diretamente no AV em teste;
- desenvolver um mecanismo que analise GC dinâmicos, quando um objeto é adicionado ao AV depois que o seu AV já está instanciado;
- executar mais estudos de caso para poder avaliar de uma forma mais precisa os critérios de testes desenvolvidos de modo a melhorá-los ou complementá-los;
- estudar e estabelecer critérios de teste específicos inerentes ao modo de interação utilizada em sistemas de realidade virtual. Segundo (MATTIOLI, 2009), a eficiência e a usabilidade de um sistema de realidade virtual ficarão comprometidas, caso este sistema possua anomalias ou falhas de interação.
- estabelecer formas de analisar a qualidade dos conjuntos de teste gerados.
- em função dos resultados já obtidos, artigos serão submetidos a periódicos internacionais.

Referências

5DT Fifth Dimension Technologies Products. 2012. Disponível em `http://www.5dt.com/products.html`

Akenine-Möller, T.; Haines, E.; Hoffman, N. *Real-time rendering*. 3rd ed. Natick, MA, USA: A. K. Peters, Ltd, 1045 p., 2008.

Ammann, P.; Offutt, J. *Introduction to software testing*. 1 ed. Cambridge University Press, 344 p., 2008.

Bar-Zeev, A. Scene graphs – past, present and future. Http://www.realityprime.com/articles/scenegraphs-past-present-and-future, 2007.

Barbosa, E. F.; Chaim, M. L.; Vicenzi, A. M. R.; Delamaro, M. E.; Jino, M.; Maldonado, J. C. Teste estrutural. In: *Introdução ao Teste de Software*, cáp. 4, Rio de Janeiro, Brasil: Campus, p. 47–76, 2007.

Beizer., B. *Software testing techniques*. 2nd ed. Van Nostrand Reinhold Company, New York, 1990.

Bertolino, A. Software Testing Research: Achievements, challenges, dreams. In: *FOSE'07: 2007 Future of Software Engineering*, Washington, DC, USA: IEEE Computer Society, 2007, p. 85–103.

Birthelmer, H.; Soetebier, I.; Sahm, J. Efficient representation of triangle meshes for simultaneous modification and rendering. In: *International Conference on Computational Science*, 2003, p. 925–934.

Blackburn, M.; Busser, R.; Nauman, A.; Stensvad, B. *Defect identification with model-based test automation*. Society of Automotive Engineers Inc (SAE), 37-44 p., 2003.

Bolt, R. A. "put-that-there": Voice and gesture at the graphics interface. In: *SIGGRAPH '80: Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, New York, NY, USA: ACM, 1980, p. 262–270.

Bondy, A.; Murty, U. *Graph Theory*, v. 244 de *Graduate Texts in Mathematics*. Springer London, 2007.

Burdea, G.; Coiffet, P. *Virtual reality technology*. New York, NY, 1994.

Campagna, S.; Kobbelt, L.; Seidel, H. P. Directed edges: A scalable representation for triangle meshes. *J. Graph. Tools*, v. 3, n. 4, p. 1–11, 1998.

Chen, J. X.; Chen, C. *Foundations of 3D Graphics Programming: Using JOGL and Java3D*. Springer Publishing Company, Incorporated, 2008.

Corrêa, C. G.; Nunes, F. L. S.; Bezerra, A.; Carvalho, J. P. Evaluation of vr medical training applications under the focus of professionals of the health area. In: *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, New York, NY, USA: ACM, 2009, p. 821–825.

DeFanti, T. A.; Dawe, G.; Sandin, D. J.; Schulze, J. P.; Otto, P.; Girado, J.; Kuester, F.; Smarr, L.; Rao, R. The starcave, a third-generation cave and virtual reality optiportal. *Future Gener. Comput. Syst.*, v. 25, n. 2, p. 169–178, 2009.

Delamaro, M. E.; Maldonado, J. C.; Jino, M. *Introdução ao teste de software*. Editora Campus, 2007a.

Delamaro, M. E.; Nardi, P.; Lemos, O. A. L.; Masiero, P. C.; Spoto, E. S.; Maldonado, J. C.; Vincenzi, A. M. R. Static analysis of java bytecode for domain-specific software testing. *In: XXI Simpósio Brasileiro de Engenharia de Software, João Pessoa, PB*, p. 325–341, 2007b.

Ferreira, A. G. *Uma arquitetura para a visualização distribuída de ambientes virtuais*. Dissertação de mestrado, PUC/RJ, 1999.

Frankl, P. G.; Weyuker, E. J. Testing software to detect and reduce risk. *Journal of Systems and Software*, v. 53, n. 3, p. 275–286, 2000.

Gill, A. *Introduction to the theory of finite-state machines*. McGraw Hill, 216 p., 1962.

Goodenough, J. B.; Gerhart, S. L. Toward a theory of test data selection. In: *Proceedings of the international conference on Reliable software*, New York, NY, USA: ACM, 1975, p. 493–510.

Harrold, M. J. Testing: a roadmap. In: *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, New York, NY, USA: ACM, 2000, p. 61–72 (*ICSE '00*,).
Disponível em `http://doi.acm.org/10.1145/336512.336532`

IEEE, . IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, p. 1, 1990.

Jacobson, L. *Virtual Reality: A status report.* Aug, 26-33 p., 1991.

Jacobson, L. *Garage virtual reality.* Pap/dis ed. Sams Publishing, 1994.

Kalkusch, M.; Schmalstieg, D. Extending the scene graph with a dataflow visualization system. In: *VRST '06: Proceedings of the ACM symposium on Virtual reality software and technology*, New York, NY, USA: ACM, 2006, p. 252–260.

Kirner, C.; Siscoutto, R. A. Fundamentos de realidade virtual e aumentada. In: *Realidade Virtual e Aumentada: Uma Abordagem Tecnológica*, cáp. 1, João Pessoal - PB: Sociedade Brasileira de Computação – SBC, p. 1–20, 2008.

Krueger, M. W. Responsive Environments. In: *AFIPS'77: Proceedings of National Computer Conference*, New York, NY, USA: ACM, 1977, p. 423–433.

Krueger, M. W. *Artificial Reality II.* 2nd ed. Addison-Wesley Professional, 1991.

Lanier, J. Virtual reality: the promise of the future. *Interactive Learning International*, v. 8, n. 4, p. 275–279, 1992.

Lemos, O. A. L.; Vincenzi, A. R.; Maldonado, J. C.; Masiero, P. C. Control and data how structural testing criteria for aspect-oriented programs. *The Journal of Systems and Software*, v. 80(6), p. 862–882, 2007.

Machover, C.; Tice, S. Virtual reality. *Computer Graphics and Applications, IEEE*, v. 14, n. 1, p. 15–16, 1994.

Maldonado, J. C. *Critérios potencias usos: Uma contribuição ao teste estrutural de software.* Tese de doutoramento, DCA/FEE/UNICAMP, Campinas, SP, 1991.

MATTIOLI, F. E. R.; LAMOUNIER JÚNIOR, E. A. C. A. A. N. M. M. Uma proposta para o desenvolvimento Ágil de ambientes virtuais. In: *VI Workshop de Realidade Virtual e Aumentada (WRVA), 2009*, 2009.

Mccabe, T. J. A. Complexity measure. In: *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, IEEE Computer Society Press, 1976, p. 407.

Morie, J. F. *Inspiring the future: Merging mass communication, art, entertainment and virtual environments, computer graphics*, v. 28. May, 135-138 p., 1994.

Myers, G. J. *The art of software testing.* 2 ed. John Wiley & Sons, 234 p., 2004.

Naik, S.; Tripathy, P. *Software testing and quality assurance : Theory and practice.* 6 ed. John Wiley and Sons, 616 p., 2008.

Ntafos, S. C. A comparison of some structural testing strategies. *IEEE Transactions on Software Engineering*, v. 14, n. 6, 1988.

Nunes, F. L. S.; Corrêa, C. G.; Picchi, F. L.; e Melo, C. R. M.; Tori, R.; Nakamura, R.; Barbosa, J. H. A. A importância da avaliação na engenharia de requisitos em sistemas de realidade virtual e aumentada: um estudo de caso. In: de Computação, S. S. B., ed. *XII SVR: Proceedings of the Symposium on Virtual and Augmented Reality*, 2010.

Oliveira, A. C. M. T. G.; Nunes, F. L. S. Building a open source framework for virtual medical training. *Journal of Digital Imaging*, v. 23, n. 6, p. 706–720, 2010.

Perry, W. *Effective methods for software testing.* 3th ed. John Wiley and Sons, 2006.

Pimentel, K.; Teixeira, K. *Virtual reality: through the new looking glass.* 2nd ed. New Yourk, USA: Windcrest/McGraw-Hill, 1995.

Poulin, P.; Fournier, A. A model for anisotropic reflection. In: *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, New York, NY, USA: ACM, 1990, p. 273–282.

Pressman, R. S. *Software engineering - a practitioners approach.* 4 ed. McGraw-Hill, 1997.

Pressman, R. S. *Engenharia de software.* 6 ed. Makron Books, 2006.

Rapps, S.; Weyuker, E. J. Data flow analysis techniques for test data selection. In: *ICSE '82: Proceedings of the 6th international conference on Software engineering*, Los Alamitos, CA, USA: IEEE Computer Society Press, 1982, p. 272–278.

Rapps, S.; Weyuker, E. J. Selecting software test data using data flow information. In: *EEE Transactions on Software Engineering*, 1985, p. 367–375.

Roehl, B. Some thoughts on behavior in VR systems. 1995.
Disponível em `http://ece.uwaterloo.ca/~broehl/behav.html`

Sensable Haptic Device. 2012.
Disponível em `http://www.sensable.com/industries-application-development.htm`

Sherman, W. R.; Craig, A. B. Understanding virtual reality: Interface, application, and desing. In: *Editora Morgan Kaufmann*, 2003.

Shreiner, D. *Opengl(r) programming guide: The official guide to learning opengl.* 7 ed. Addison-Wesley Professional, 2009.

Silva, R. J. M.; Raposo, A. B.; Gattas, M. Grafo de cena e realidade virtual, monografia em Ciência da Computação - Rio de Janeiro - RJ: PUC-Rio, 2004.

Silva, W. A.; Júnior, E. L.; Cardoso, A.; Ribeiro, M. W. S. *Livro do pré-simpósio, xi symposium on virtual and augmented reality*, cáp. Aplicações de Realidade Aumentada na criação de Interfaces Distribuídas Sociedade Brasileira de Computação - SBC, p. 108–127, 2009.

Sowizral, H.; Rushforth, K.; Deering, M. *The java 3rd API specification with cdrom.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc, 2000.

Spoto, E. S.; Jino, M.; Maldonado, J. C. Structural software testing: An approach to relational database applications. In: *In XIV Brazilian Symposium on Software Engineering, João Pessoa, PA, Brazil. (In Portuguese)*, 2000.

Strauss, P. S.; Carey, R. An object-oriented 3rd graphics toolkit. *Computer Graphics*, v. 26, n. 2, 1992.

Sun Java 3D API tutorial. 2000.
Disponível em `http://java.sun.com/developer/onlineTraining/java3d/`

Sutherland, I. E. Sketch pad a man-machine graphical communication system. In: *DAC '64: Proceedings of the SHARE design automation workshop*, New York, NY, USA: ACM, 1964, p. 6.329–6.346.

Tichy, W. F.; Lukowicz, P.; Prechelt, L.; Heinz, E. A. Experimental evaluation in computer science: a quantitative study. *J. Syst. Softw.*, v. 28, n. 1, p. 9–18, 1995.

Tori, R.; Kirner, C.; Siscouto, R. *Fundamentos e tecnologia de realidade virtual e aumentada.* VIII Symposium on Virtual Reality - Belém, 388 p., 2006.

Vergilio, S. R.; Maldonado, J. C.; Jino, M. Uma estratégia para a geração de dados de teste. In: *VII Simpósio Brasileiro de Engenharia de Software*, Rio de Janeiro, 1993, p. 307–319.

Wainer, J.; Novoa Barsottini, C. G.; Lacerda, D.; Magalhaes de Marco, L. R. Empirical evaluation in computer science research published by acm. *Inf. Softw. Technol.*, v. 51, n. 6, p. 1081–1085, 2009.

Walsh, A. E. Understanding scene graphs. Dr Dobb's Journal, 2002. Disponível em `http://www.ddj.com/184405094`

Wong, W. E. *On mutation and data flow.* Tese de doutoramento, West Lafayette, IN, USA, 1993.

Woo, M.; Neider, J.; Davis, T.; Shreiner, D. *OpenGL, Programming Guide.* 3th ed. 1999.

Zelkowitz, M. V.; Wallace, D. R. Experimental models for validating technology. *Computer*, v. 31, n. 5, p. 23–31, 1998.