UNIVERSIDADE DE SÃO PAULO
Instituto de Ciências Matemáticas e de Computação

# Uma abordagem de teste de software para aplicações de realidade virtual utilizando testes metamórficos

**Stevão Alves de Andrade**

Tese de Doutorado do Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional (PPG-CCMC)

ICMC USP SÃO CARLOS

**Stevão Alves de Andrade**

# Uma abordagem de teste de software para aplicações de realidade virtual utilizando testes metamórficos

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

Área de Concentração: Ciências de Computação e Matemática Computacional

Orientador: Prof. Dr. Márcio Eduardo Delamaro
Coorientadora: Profa. Dra. Fátima de Lourdes dos Santos Nunes Marques

**USP – São Carlos**
**Janeiro de 2023**

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

Alves de Andrade, Stevão
A553a Uma abordagem de teste de software para aplicações de realidade virtual utilizando testes metamórficos / Stevão Alves de Andrade; orientador Márcio Eduardo Delamaro; coorientadora Fátima de Lourdes dos Santos Nunes Marques. -- São Carlos, 2023.
168 p.

Tese (Doutorado - Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional) -- Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2023.

1. Teste de software. 2. Realidade virtual. 3. Testes metamórficos. I. Delamaro, Márcio Eduardo , orient. II. de Lourdes dos Santos Nunes Marques, Fátima, coorient. III. Título.

Bibliotecários responsáveis pela estrutura de catalogação da publicação de acordo com a AACR2:
Gláucia Maria Saia Cristianini - CRB - 8/4938
Juliana de Souza Moraes - CRB - 8/6176

**Stevão Alves de Andrade**

# A software testing approach to virtual reality applications using metamorphic testing

Thesis submitted to the Instituto de Ciências Matemáticas e de Computação – ICMC-USP – in accordance with the requirements of the Computer and Mathematical Sciences Graduate Program, for the degree of Doctor in Science. *FINAL VERSION*

Concentration Area: Computer Science and Computational Mathematics

Advisor: Prof. Dr. Márcio Eduardo Delamaro
Co-advisor: Profa. Dra. Fátima de Lourdes dos Santos Nunes Marques

**USP – São Carlos**
**January 2023**

*Aos meus pais que sempre acreditaram no poder transformador da educação*
*e a todos os anônimos que, graças aos seus esforços invisíveis, possibilitaram a realização deste trabalho.*

# AGRADECIMENTOS

Agradeço primeiramente a Deus, pois sem a sua vontade, a conclusão deste trabalho jamais seria possível.

Gostaria de agradecer ao meu orientador, Dr. Márcio Delamaro e a minha coorientadora Dra. Fátima Nunes, pela oportunidade que me foi dada. Sem a orientação e postura deles, o desenvolvimento deste trabalho não teria obtido êxito. Muito obrigado pelas reuniões, pelas orientações, pelas cobranças e por contribuírem não apenas na minha formação acadêmica, mas também na minha formação pessoal. Também peço desculpas caso não tenha sido capaz de atender às expectativas.

Gostaria de agradecer aos meus pais Isabel e Raimundo, que sempre me motivaram a buscar o meu melhor e jamais mediram esforços para me ajudar. A vocês meu eterno amor e gratidão. Gostaria de agradecer meus irmãos Stênio e Stanley por sempre me servirem de espelho e motivação. Jamais terei como retribuir a contribuição de vocês na minha formação. Também gostaria de agradecer a toda a minha família em São Paulo, que me acolheu tão bem desde 2013 quando decidi trilhar o caminho da pós-graduação. Em especial à minha tia Conceição, que sempre me tratou como um filho, e aos meus primos Júnior, Thays, Romário e Camila, Robson, Henrique (*in memoriam*), Taty. Muito obrigado, vocês ajudaram a superar a saudade de estar longe de casa.

Também preciso agradecer a todos os amigos que conheci e tive o prazer de dividir esses anos em São Carlos. Não tenho a menor duvida de que, até agora, esses foram os melhores anos da minha vida. Em especial, gostaria de agradecer à Daniela pela sua companhia e por sempre ter tido a paciência de me aturar, muito obrigado de verdade! Ao Brauner e ao Mantovani por terem divido uns anos das suas vidas morando comigo. À Lina, à Lívia, ao Damasceno e à Kamila por suas amizades mais que especiais. Ao Misael por ajudar a dividir o peso de estar longe de casa. Ao Léo pela eterna parceria, mesmo em anos de pandemia, no laboratório. Ao Ítalo pelas viagens e tantos outros amigos especiais que levarei comigo para o resto da vida: Vânia, Silvana, Valeria, Jadson, FCarlos, Aline, Padilha, Mundim, Jaum, Adam, Diógenes, Faimison, Wilmax, Gisele, Ricardo, Pedro, Daniel, Laíza, Danilo, Choma, Biazoto, Henrique, Cris, Ana Cláudia, Iohan, Vinícius, Drika, Montanari, Rodolfo, Silvério, Jorge, Claudinei, Bento, Kathiani e tantos outros. Peço mil desculpas por não conseguir colocar o nome de todo mundo, mas gostaria de agradecer a todos os amigos do LabES, Biocom, UFSCar, Rep Eskoria e da pós-graduação em geral. Sem vocês essa jornada jamais teria sido a mesma coisa.

Por fim, agradeço o apoio financeiro da **Fundação de Amparo à Pesquisa do Estado de**

**São Paulo** (FAPESP), sob os processos nº 2017/19492-1 e 2019/06937-0, da **Coordenação de Aperfeiçoamento de Pessoal de Nível Superior** - Brasil (CAPES) - Código de Financiamento 001 (processo nº PROEX-8435441/D). Também agradeço o suporte financeiro do **Conselho Nacional de Desenvolvimento Científico e Tecnológico** (CNPq), sob o processo nº 308615/2018-2, ao **National Institute of Science and Technology Medicine Assisted by Scientific Computing** (INCT-MACC), sob o processo nº 157535/2017-7. Sem tal suporte financeiro, este trabalho jamais poderia ter sido realizado.

*“Toda rua tem seu curso*
*tem seu leito de água clara*
*por onde passa a memória*
*lembrando histórias de um tempo*
*que não acaba.*
*(Torquato Neto) ”*

# RESUMO

ANDRADE, S. A. **Uma abordagem de teste de software para aplicações de realidade virtual utilizando testes metamórficos**. 2023. 168 p. Tese (Doutorado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2023.

Teste de Software é uma das áreas de pesquisa existentes dentro da Engenharia de Software, sendo a principal atividade utilizada para aferir a conformidade entre requisitos de software e suas respectivas implementações. O processo de automatização da atividade de teste de software é uma tarefa fundamental que visa oferecer produtividade e efetividade à atividade de teste. A automatização da atividade de teste possibilita que a mesma possa ser conduzida sob critérios sistemáticos, o que garante reprodutibilidade, além de aumentar, significativamente, a chance de identificação de falhas no produto avaliado. Oráculos de teste desempenham uma função fundamental dentro da atividade de teste, sendo responsáveis por avaliar o comportamento das saídas produzidas por um software durante a atividade de teste. Oráculos de teste podem ser derivados a partir de especificações do software, métodos formais, assertivas, técnicas de aprendizagem de máquina, relações metamórficas, entre outros. Esta tese de doutorado propõe e avalia uma nova abordagem automatizada de teste de software para aplicações de Realidade Virtual. Diferentemente de programas convencionais, aplicações de realidade virtual sofrem de um problema denominado de "problema do oráculo de teste", que ocorre em situações nas quais as saídas do sistema em teste são dadas em formatos não convencionais como, por exemplo, imagens, objetos tridimensionais e ambientes de realidade virtual. O propósito da abordagem proposta nesta tese é utilizar-se da técnica de testes metamórficos para criar restrições de teste em aplicações de realidade virtual e utilizar aprendizado por reforço a fim de possibilitar a geração automática de dados de teste para automatizar o processo de teste de aplicações de realidade virtual. Esta tese de doutorado estende estes dois conceitos em uma abordagem de teste para o domínio de aplicações de realidade virtual. Para isso o trabalho foi desenvolvido em três diferentes etapas: *(i)* o levantamento de artefatos de software, disponibilizados em repositórios de código aberto, a fim de entender percepções e extrair modelos de práticas de qualidade de software no contexto de programas de realidade virtual; *(ii)* um *survey* para investigar a percepção de grupos de interesse, visando identificar deficiências nas práticas de qualidade de software no contexto de realidade virtual, de forma a mapear as necessidades existentes; e por fim, *(iii)* o desenvolvimento e avaliação da abordagem de teste proposta a partir dos resultados observados nas etapas anteriores.

**Palavras-chave:** Teste de Software, Realidade Virtual, Testes Metamórficos.

# ABSTRACT

ANDRADE, S. A. **A software testing approach to virtual reality applications using metamorphic testing**. 2023. 168 p. Tese (Doutorado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2023.

Software Testing is one of the existing research areas in Software Engineering, being the main activity used to verify the conformity between software requirements and their respective implementations. The process of automating the software testing activity is a fundamental task that aims to give productivity and effectiveness. The automation of the software testing activity allows it to be conducted under systematic criteria, which guarantees the activity's reproducibility, in addition to significantly increasing the chance of identifying flaws in the evaluated product. Test oracles play a fundamental role within the testing activity, being responsible for evaluating the behavior of the outputs produced by a software during the testing activity. In this context, test oracles can be derived from software specifications, formal methods, assertions, machine learning techniques, metamorphic relationships, among others. This doctoral thesis proposes and evaluates a new automated software testing approach for Virtual Reality applications. Unlike conventional programs, virtual reality applications suffer from the "test oracle problem", which occurs in situations where the outputs of the system under test are given in unconventional formats, such as machine learning models, images, three-dimensional objects and virtual reality environments. The purpose of the approach presented in this thesis is to use the metamorphic testing technique to create test constraints for virtual reality applications and to use reinforcement learning to enable the automatic generation of test data to automate the testing process of virtual reality applications. To do so, the work was conducted in three different stages: *(i)* the first stage consisted of a survey of software artifacts, available in open source repositories, in order to understand perceptions and extract models of practices of software quality in the context of virtual reality programs; in the second stage *(ii)* was a *survey* designed to investigate the perception of groups of interest, aiming to discover what is lacking about software quality practices in the context of virtual reality in order to clearly map the existing needs; and finally the third stage *(iii)* consisted in the development and evaluation of the testing approach based in the results observed in the previous steps.

**Keywords:** Software Testing, Virtual Reality, Metamorphic Testing.

# LISTA DE ILUSTRAÇÕES

# LISTA DE TABELAS

# LISTA DE ABREVIATURAS E SIGLAS

| | |
|---|---|
| ACL | *Average Clustering and Labeling* |
| AM | Aprendizado de Máquina |
| APIs | *Application Programming Interface* |
| AR | Aprendizado por reforço |
| BOOM | *Binocular Omni-Orientation Monitor* |
| CAVE | *Cave Automatic Virtual Environment* |
| CGIs | Computer-Generated Imagery |
| DNN | *Deep Neural Network* |
| GFC | Grafo de Fluxo de Controle |
| GPU | *Graphics Processing Unit* |
| GQM | *Goal, Question, Metric* |
| HMD | *Head Mounted Display* |
| LabES | *Laboratório de Engenharia de Software* |
| LOD | *Level of Detail* |
| MVC | *Model-View-Controller* |
| PAE | Programa de Aperfeiçoamento de Ensino |
| PBL | *Problem Based Learning* |
| PPO | *Proximal Policy Optimization* |
| RM | Relações Metamórficas |
| RV | Realidade Virtual |
| SVR | *21ª edição do Simpósio de Realidade Virtual e Aumentada* |
| UML | *Unified Modeling Language* |
| WYSIWYG | *What You See Is What You Get* |

# SUMÁRIO

CAPÍTULO

1

# INTRODUÇÃO

O avanço da tecnologia de Realidade Virtual (RV) tem possibilitado a capacidade de produzir ambientes digitais nos quais sentidos como a percepção visual, a audição e o tato são simulados com a ajuda de equipamentos necessários para interagir e interferir em objetos de um ambiente digital (BURDEA; COIFFET, 2017). Por meio dessa tecnologia os usuários são capazes de reproduzir sentimentos e experiências correspondentes aos de um ambiente real (ZHAO, 2009; LAVALLE, 2019). Dessa forma, aplicações de RV têm se tornado cada vez mais utilizadas no sentido de tentar compreender ou simular a natureza possibilitando a sua utilização nos mais diferentes domínios da sociedade.

O amadurecimento da área de RV possibilitou a sua expansão e utilização nos mais diversos setores, introduzindo novas metodologias para a solução de variados tipos de problemas. Wohlgenannt, Simons e Stieglitz (2020) destacam a possibilidade da utilização de aplicações de RV em diversos domínios, como prototipagem virtual para auxiliar as decisões tomadas durante a fase de design de projetos de engenharia de grande escala; simuladores e treinamento, ao destacar que usuários poderiam explorar habilidades cognitivas e motoras existentes para interagir com o mundo em uma variedade de modalidades sensoriais; telepresença e teleoperação, para simular a presença de um operador em um ambiente remoto para supervisionar o funcionamento e realizar tarefas de controle de robôs remotos, além de diversos outros domínios, como na área de entretenimento e educação (KAVANAGH *et al.*, 2017).

Tal expansão na utilização de tecnologias RV foi acompanhada pelo aumento da complexidade no desenvolvimento de soluções, uma vez que aplicações de RV fornecem uma forma avançada de interação, em tempo real, em ambientes tridimensionais entre o usuário e a aplicação (BURDEA; COIFFET, 2017). Da mesma forma, o conjunto de ferramentas para ambientes de desenvolvimento dessas aplicações passaram a ser capazes de se adaptar a múltiplas configurações de hardware e software, além de facilitar a utilização e configuração, visando abstrair ao máximo a complexidade envolvida nos equipamentos de hardware utilizados (ANTHES *et al.*,

2016).

O processo de produção de software de alta qualidade tornou-se uma incessante busca na indústria desde que a computação passou a desempenhar papel fundamental nos mais diversos tipos de domínios de aplicação. Dessa forma, o processo de desenvolvimento de software passou a envolver um conjunto de métodos, técnicas e ferramentas a serem empregadas a fim de garantir a qualidade do produto desenvolvido. Apesar disso, a atividade de desenvolvimento de software ainda está sujeita a problemas que podem comprometer a qualidade do produto gerado. Desse modo, Pressman e Maxim (2014) definem engenharia de software como uma área que aplica princípios da engenharia para construir software, com o objetivo de minimizar o seu custo de produção e aumentar a sua qualidade.

Dentre as técnicas de verificação e validação de software aplicadas durante o processo de desenvolvimento, a atividade de teste de software é uma das mais utilizadas, sendo de uma abordagem que tem por objetivo fornecer evidências sobre a confiabilidade do produto avaliado (MYERS; SANDLER; BADGETT, 2011). De acordo com Delamaro, Maldonado e Jino (2016), a atividade de teste consiste de uma análise dinâmica do produto, sendo uma atividade crucial para a identificação e eliminação de possíveis defeitos que possam ter sido introduzidos durante o processo de desenvolvimento de um software.

Naturalmente, o aumento da complexidade dos software a ser desenvolvido implica diretamente no surgimento de novos desafios relacionados à maneira como testá-los (DELAMARO; MALDONADO; JINO, 2016). Levando em conta essa motivação e a aceitação, entre pesquisadores e profissionais, de que qualidade é um fator essencial no desenvolvimento de software, muito se tem investido em pesquisas na área de teste de software explorando, entre outros, problemas em domínios complexos que ainda não foram especificamente abordados.

Especificamente no contexto de RV, destaca-se a dificuldade em se especificar qual a saída válida para uma aplicação dada a execução de um conjunto de dados de teste (DONALDSON *et al.*, 2017). Tal problema é definido na literatura como "problema do oráculo de teste", e é um desafio apresentado em aplicações que apresentam saídas complexas (BARR *et al.*, 2015). Nesse sentido, testes metamórficos têm se tornado uma alternativa viável para o desenvolvimento da atividade de teste de software, uma vez que eliminam a necessidade da presença do oráculo para a identificação de falhas na aplicação em teste (CHEN *et al.*, 2018).

## 1.1 Questão de Pesquisa, Hipótese e Objetivos

Devido à grande popularização de aplicações de RV em diversos domínios, a maior parte das pesquisas nessa área têm focado na disseminação de pesquisas aplicadas, reportando resultados relacionados a estudos de casos discutindo o sucesso na utilização de tal tecnologia (DETROZ *et al.*, 2014). De fato, conforme apresentado na revisão sistemática conduzida por Santos, Delamaro e Nunes (2013), pouco se tem explorado sobre práticas de engenharia de

software para apoiar ao desenvolvimento de aplicações nessa área.

Esta tese de doutorado sugere que a atividade de teste de software é uma das atividades de engenharia de software que pode contribuir no processo de desenvolvimento de aplicações de RV e possibilitar a entrega de um produto de melhor qualidade que atenda às demandas para as quais fora planejado. Uma vez que com o aumento na perspectiva de utilização de RV nos mais diversos contextos, aumenta-se também a demanda por mecanismos capazes de garantir que os produtos desenvolvidos estejam de acordo com as especificações para as quais foram inicialmente projetados. Esse problema, em conjunto com a contextualização apresentada, dá origem à seguinte questão de pesquisa:

**Questão de Pesquisa**

Como a utilização de testes metamórficos pode contribuir para lidar com o problema do oráculo de teste e aprimorar o processo de teste de software no contexto de RV?

Essa preocupação nasce uma vez que atividades convencionais de teste de software tendem a não ser tão efetivas quando aplicadas em domínios específicos de software, devido ao fato de que cada domínio de software apresenta particularidades e tipos de defeitos distintos (ORSO; ROTHERMEL, 2014).

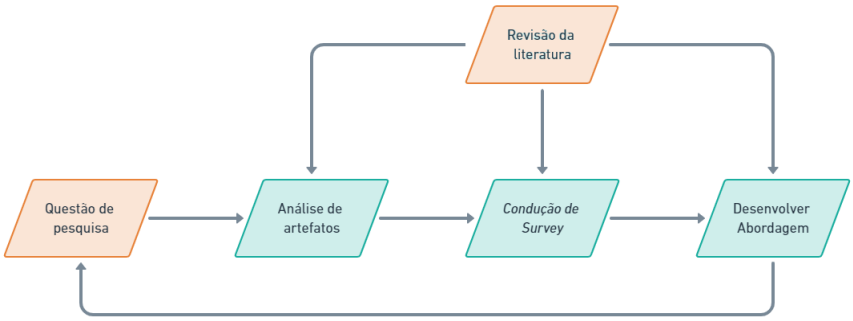
Outro ponto observado é que apesar da popularidade de aplicações de RV, nem os pesquisadores de engenharia de software nem novos desenvolvedores que migram para o desenvolvimento RV têm uma ideia clara sobre quais são as principais preocupações e desafios no desenvolvimento. Dessa forma, aplicações que utilizam funcionalidades e recursos de RV, normalmente, não incluem uma fase sistematizada de teste, fazendo com que a atividade de testes seja conduzida de forma manual e *ad-hoc* (SANTOS; DELAMARO; NUNES, 2013; RODRIGUEZ; WANG, 2021).

Nesse contexto, essa pesquisa tem como objetivo desenvolver e adaptar técnicas de teste de software a aplicações de RV. Para tanto, pretende-se explorar defeitos típicos para o domínio de RV, a fim de utilizá-los como base para propor uma abordagem de testes, utilizando testes metamórficos.

A questão de pesquisa apresentada acima preocupa-se com a definição de mecanismos capazes de propor a elaboração e a avaliação automatizadas de casos de teste de forma a levantar o seguinte problema: *como a utilização da técnica de testes metamórficos pode propor mecanismos que possibilitem a geração e execução de testes automatizados desenvolvidos para o contexto de aplicações de realidade virtual?*

Aplicações desenvolvidas para o contexto de RV tendem a apresentar uma dificuldade para uma clara distinção de um oráculo de teste. Esse problema compromete diretamente a viabilidade de aplicação de práticas de testes automatizados, por necessitarem de informações relacionadas ao oráculo de teste para possibilitar a sua aplicação. Contudo, com base em

Figura 1 – Organização das etapas desenvolvidas durante o processo de pesquisa




Fonte: Elaborada pelo autor.

possíveis soluções exploradas para superar o "problema do oráculo de teste" (BARR *et al.*, 2015), a hipótese levantada neste trabalho é que:

**Hipótese –** *É possível ter uma abordagem de testes eficiente, aplicável ao domínio de RV, por meio da utilização da técnica de testes metamórficos.*

Especula-se que essa hipótese seja praticável, uma vez que existem diversos estudos que discutem sobre a aplicação, com sucesso, de testes metamórficos em vários domínios (CHEN *et al.*, 2018).

Supõe-se que se a abordagem for utilizada de maneira adequada, ela pode garantir um mecanismo sistematizado a ser aplicado ao domínio de RV para condução da atividade de teste, bem como garantir a aplicação da atividade de geração/validação de dados de teste de forma automatizada.

Para isso, foram definidos três objetivos:

**$O_1$ –** Definir relações metamórficas que estejam alinhadas a tipos de falhas comuns identificados por *stakeholders* envolvidos no processo de desenvolvimento e utilização de aplicações de RV.

**$O_2$ –** Definir uma abordagem de geração automática de testes baseada em testes metamórficos para o contexto de aplicações de RV.

**$O_3$ –** Desenvolver um protótipo para aplicação da abordagem proposta e avaliá-lo em termos de efetividade, eficiência e utilidade.

Para atingir os objetivos elencados, o processo de construção do trabalho seguiu com o desenvolvimento de três etapas principais, conforme descrito na Figura 1. A primeira etapa do trabalho constituiu de um levantamento de artefatos de software, disponibilizados em repositórios de código aberto, a fim de entender percepções e extrair modelos de práticas de qualidade de software no contexto de programas de RV, bem como para possibilitar a identificação de *gaps* de pesquisa.

Com base nas informações compreendidas na primeira etapa, foi projetado um *survey* para investigar a percepção de grupos de interesse, visando descobrir os principais problemas relacionados às práticas de qualidade de software no contexto de RV, de forma a mapear de forma clara as necessidades existentes. Em paralelo a tais atividades, a análise da literatura permitiu verificar o que tem sido desenvolvido na área e como as abordagens descritas lidam com as limitações identificadas nos passos anteriores.

O desenvolvimento da abordagem de teste proposta foi direcionado a partir dos resultados das etapas anteriores e teve como finalidade dar apoiar às percepções apontadas sustentadas pela hipótese de pesquisa investigada.

## 1.2 Estrutura do Documento

Esta tese está organizada em oito capítulos. No primeiro capítulo, foram apresentados a contextualização, motivação, questão de pesquisa e hipótese de pesquisa do trabalho.

No Capítulo 2 é apresentada uma revisão da literatura sobre de RV, que é um dos tópicos principais que caracteriza o domínio de aplicação deste trabalho. Nele, são apresentados os principais conceitos relacionados ao desenvolvimento de software nesse domínio e os seus respectivos desafios.

O Capítulo 3 traz uma revisão bibliográfica a respeito de teste de software, apresentando os conceitos gerais da área e dando ênfase para os tópicos que são fundamentais para o entendimento deste trabalho, como fases de teste, conceitos fundamentais sobre testes metamórficos e geração automática de dados de teste.

O Capítulo 4 apresenta uma visão geral das principais características de RV que podem ter impacto nas atividades de verificação, validação e teste de software. Além disso, traz um estudo que investigou projetos de RV de código aberto para traçar um quadro sobre o perigo da falta de atividades de teste de software.

O Capítulo 5 traz os resultados de um *survey* que investiga a percepção de *stakeholders* envolvidos no contexto de desenvolvimento e utilização de aplicações de RV. O foco é observar a percepção dos mesmos quanto à necessidade de práticas de teste de software para o contexto RV, bem como apresentar os tipos de falhas que causam maior impacto sob o ponto de vista de cada um dos grupos avaliados.

O Capítulo 6 apresentará a abordagem de testes desenvolvida durante essa pesquisa para utilização de testes metamórficos em conjunto com geração automática de dados de teste para aplicações de RV.

O Capítulo 7 discorre a respeito dos resultados de uma avaliação experimental desenvolvida para avaliar a proposta apresentada no capítulo.

O Capítulo 8 revisita o problema de pesquisa que motivou o desenvolvimento do trabalho

e as questões de pesquisa investigadas ao longo do desenvolvimento do mesmo. No capítulo são sumarizadas as principais contribuições da pesquisa, suas limitações e futuros direcionamentos.

CAPÍTULO

2

# FUNDAMENTOS DE REALIDADE VIRTUAL

Este capítulo apresenta uma revisão da literatura sobre um dos temas que fundamentam as pesquisas desenvolvidas nesta tese. Ele começa descrevendo os principais conceitos relacionados a RV na seção 2.1. Na sequência, a seção 2.2 apresenta alguns dos principais dispositivos utilizados para o desenvolvimento de aplicações em RV, bem como uma lista de domínios de aplicações em que eles podem ser utilizados. A seção 2.3 descreve os domínios de aplicações de RV mais populares, a seção 2.4 apresenta uma lista das principais bibliotecas gráficas utilizadas para auxiliar o desenvolvimento de aplicações de RV, a seção 2.5 apresenta definições a respeito de uma das principal estrutura de dados utilizadas (grafo de cena) para representar objetos em uma aplicação desenvolvida para o contexto de RV, destacando suas principais características, vantagens e desvantagens, a seção 2.6 discute práticas de design e implementação de aplicações de RV e, por fim, a seção 2.7 apresenta as considerações finais deste capítulo.

## 2.1 Definições

Quando o termo RV começou a ser utilizado, causou grandes expectativas. A ideia era que tal tecnologia possibilitaria a criação de mundos imaginários, os quais seriam indistinguíveis do mundo real (BURDEA; COIFFET, 2017).

RV utiliza computadores para a criação de ambientes tridimensionais, possibilitando a navegação e interação em tempo real. O usuário pode navegar pelo ambiente, a fim de explorar possíveis características presentes na representação tridimensional, como, por exemplo, explorar a estrutura dos órgãos internos do corpo humano (LAVALLE, 2019).

A principal definição difundida na literatura sobre RV a define como a forma mais avançada de interação entre o usuário e o computador em tempo real, a partir de um ambiente tridimensional sintético, utilizando dispositivos multi-sensoriais (BURDEA; COIFFET, 2017). Por meio de dispositivos não convencionais de visualização, os objetos podem ser percebidos

como se estivessem no mesmo ambiente que o usuário.

Aplicações de RV, em sua forma mais robusta, requerem a utilização de processadores gráficos a fim de possibilitar interação em tempo real. Também podem requerer um sistema de monitoramento para cabeça e mãos, a fim de capturar e responder a movimentos do usuário. Comumente, as tecnologias utilizadas incluem dispositivos sensoriais (óculos estereoscópicos e capacetes com rastreadores de movimento), além de *joysticks* ou luvas, para possibilitar a simulação ou captura de movimentos relacionados à mão, com o propósito de garantir ao usuário a interação com objetos em uma cena virtual.

A RV pode ser classificada em diversos níveis de imersão, a depender da percepção do usuário quanto à sua presença em uma cena. É dita imersiva quando o usuário tem a sensação de estar presente dentro do mundo virtual, sensação provida por meio da utilização de dispositivos sensoriais que captam seus movimentos e comportamento. É considerada não imersiva quando o usuário é transportado de maneira parcial ao mundo virtual por meio de um monitor ou projetor (TORI; KIRNER; SISCOUTTO, 2006).

Os aspectos visuais são frequentemente os mais relacionados à criação de ambientes virtuais, devido ao fato de que a visão é a principal forma de percepção para a aquisição de informações para grande parte dos usuários. Contudo, a realidade não se limita apenas àquilo que é possível ver. Outros componentes importantes da percepção humana desempenham papel importante em modelar um ambiente virtual, como sons e percepções obtidas por meio do tato.

Na Figura 2 são apresentados exemplos de aplicações de RV imersivas (Figura 2a), utilizando óculos estereoscópicos para proporcionar a completa imersão do usuário no ambiente virtual e aplicações não imersivas (Figura 2b), utilizando uma matriz de monitores de vídeo para exibição.

O tipo de dispositivo utilizado para propiciar a sensação de imersão pode ter grande influência nas características do produto de RV, bem como no seu desenvolvimento. A seção seguinte apresenta os principais dispositivos utilizados para o desenvolvimento de sistemas de RV.

## 2.2 Dispositivos de RV

Possibilitar uma total imersão, a ilusão de estar fisicamente presente em um mundo que não existe fisicamente, tem sido uma das grandes aspirações de muitos cientistas, engenheiros, designers e artistas. Por diversos anos, vários modelos de dispositivos foram projetados, prototipados e, até mesmo, comercializados com a intenção de atingir esse objetivo.

Existem vários mecanismos para possibilitar a imersão em ambientes virtuais. Segundo Anthes *et al.* (2016), os dispositivos de *hardware* mais utilizados são:

Figura 2 – Exemplo de aplicações imersivas e não imersivas em RV

(a) Imersivo

(b) Não imersivo

Fonte: Elaborada pelo autor.

- ***Head Mounted Display (HMD)***, que é um sistema físico complexo que integra combinadores, ótica e monitores em um só dispositivo. HMD são disponibilizados em diferentes resoluções, contraste, campo de visão, etc. Podendo ser utilizados tanto para ambientes virtuais quanto para realidade aumentada;

- ***Binocular Omni-Orientation Monitor (BOOM)***, que são telas e sistemas ópticos organizados em uma caixa. Esse dispositivo é anexado a um braço mecânico que contém articulações que permitem ao usuário mover-se. São inseridos sensores nas articulações do braço, possibilitando que a posição e a orientação do BOOM sejam computadas. O usuário olha para a caixa por meio de dois furos, visualizando o mundo virtual e podendo guiar a caixa sem esforço para qualquer posição dentro do volume operacional do dispositivo;

- ***Cave Automatic Virtual Environment (CAVE)***, são ambientes totalmente imersos baseados em projeção para envolver o visualizador com quatro telas. As telas são dispostas em um cubo formado por três telas de projeção para paredes e uma tela de projeção para o piso. Os projetores e os espelhos das paredes laterais estão localizados atrás de cada parede. O projetor para o chão fica suspenso no teto da CAVE, que aponta para um espelho que reflete as imagens no chão.

Para além dos dispositivos citados acima, dispositivos hápticos, utilizados para criar uma experiência de toque ao aplicar força, vibrações ou movimentos ao usuário e dispositivos multissensoriais, que estimulam outros sentidos, que geram *feedback* tátil ou olfativo, também são incluídos na lista de de dispositivos de RV e normalmente são utilizados em conjunto com

os demais para possibilitar novas formas de interação e proporcionar uma maior sensação de imersão ao usuário (BOUZBIB *et al.*, 2021).

## 2.3 Domínios de Aplicação de RV

Aplicações em RV têm sido utilizadas com sucesso em diversos domínios de software, como:

- **medicina:** a área de saúde é um dos segmentos que mais têm-se utilizado tecnologias de RV. A utilização dessa tecnologia engloba desde simulações de cirurgias, tratamento de fobias, cirurgias robóticas, além de capacitações, que estão inclusas no domínio educacional (LI *et al.*, 2017);

- **cultura:** uma maneira muito atraente de descobrir e entender melhor os aspectos culturais de um lugar, seria ver e visitá-los em seus tempos de glória. Para isso, viajar no tempo e no espaço seria a melhor abordagem possível, e a RV oferece uma experiência similar, por meio de aplicações que são usadas para recriar visualmente elementos históricos que desapareceram com o passar do tempo (MILES *et al.*, 2016; VAINSTEIN; KUFLIK; LANIR, 2016);

  ***marketing*:** no mercado tradicional, os potenciais compradores precisam analisar diversas variáveis antes de tomar a decisão de adquirir um bem. RV pode acelerar esse processo de comercialização tradicional, além de reduzir os gastos de campanhas de *marketing* tradicionais a longo prazo, oferecendo ao consumidor uma experiência mais realista quando o mesmo faz uma busca sobre produtos de interesse (WALLIN, 2017);

- **simulações interativas:** a indústria de modelagem sempre dependeu fortemente de imagens geradas por computadores, Computer-Generated Imagery (CGIs), para mostrar conceitos de produtos a serem produzidos. O auxílio da RV possibilita exibições mais dinâmicas dos produtos, propiciando ao visualizador, efeitos e experiências mais realistas. Tais características permitem, por exemplo, modelar ambientes como simuladores de veículos (CARLOZZI *et al.*, 2013) e construções (BOSCHE; ABDEL-WAHAB; CAROZZA, 2016);

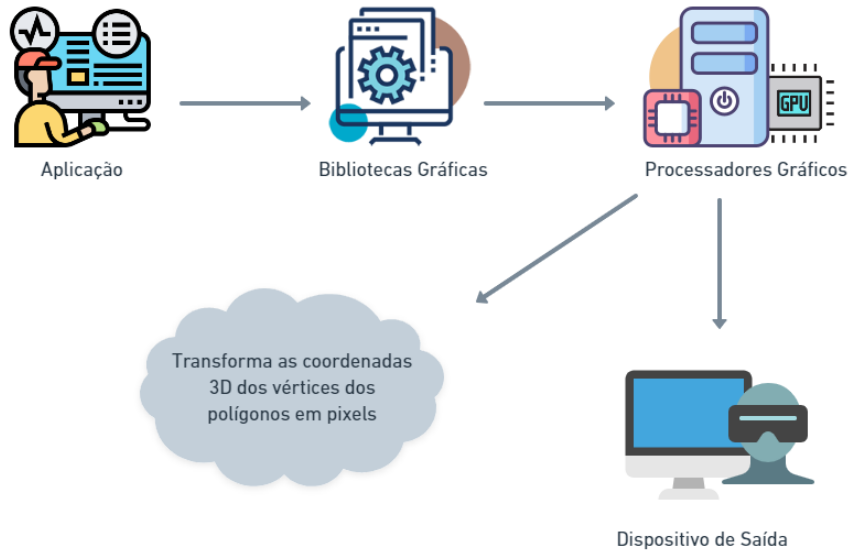
- **entretenimento e jogos:** os avanços recentes em sensores de movimento, gráficos e interação abriram o caminho para expandir as possibilidades de jogos de RV proporcionando experiências imersivas perfeitas em mundos altamente interativos. De aventuras ativas a imersões relaxantes, a RV possibilita levar aos experiencias em primeira pessoa. Além de explorar jogos sérios, como os que envolvem educação e treinamento (CRUZ-NEIRA; FERNáNDEZ; PORTALéS, 2018).

## 2.4 Bibliotecas Gráficas

O desenvolvimento de aplicações de RV consiste em representar situações do mundo real por meio do auxílio de primitivas, como esferas, cubos, superfícies, cilindros, além de modelos tridimensionais que vão ser utilizados para representar objetos com um maior grau de fidelidade.

Bibliotecas gráficas de baixo nível disponibilizam aos desenvolvedores um conjunto de primitivas geométricas, além de conjuntos de comandos que permitem a especificação de objetos geométricos em duas ou três dimensões, sendo uma interface de software para hardware gráfico. E normalmente são projetadas como interfaces independentes de hardware para possibilitar a utilização por diferentes plataformas.

Figura 3 – Etapas de renderização de um objeto 3D




Fonte: adaptado de (HUGHES *et al.*, 2020)

Segundo Walsh (2002), bibliotecas gráficas são comumente denominadas de *Application Programming Interface* (APIs) gráficas, sendo as mais utilizadas a *OpenGL* (Open Graphics Library, 2021), a *DirectX* (Microsoft Corporation, 2021) e a moderna *Vulkan* (The Khronos® Group Inc, 2021). O *OpenGL* foi a primeira biblioteca livre utilizada para desenvolvimento de aplicações gráficas, ambientes 3D, jogos, entre outros. O *DirectX* foi uma alternativa ao *OpenGL*, proposta pela *Microsoft*, para desenvolvimento de aplicações e jogos na plataforma *Windows*. Por fim, o *Vulkan* é uma das mais novas bibliotecas gráficas existentes, que foi projetada com intuito de aproveitar os recursos de hardware.

A Figura 3 resume, em uma visualização de alto nível, a operação de tais bibliotecas. Elas são utilizadas por desenvolvedores, de forma nativa, ou por meio de *frameworks* de terceiros para facilitar a construção de aplicações de RV e ficam responsáveis pela comunicação com

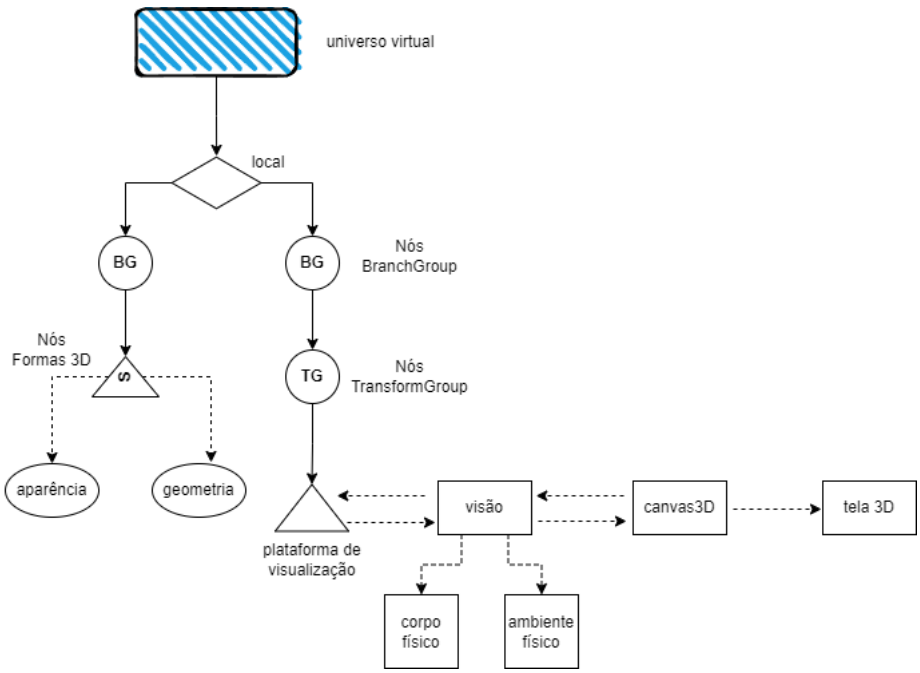
processadores e placas de vídeo para possibilitar a renderização das chamadas nos dispositivos de saída.

## 2.5 Grafo de Cena

Para facilitar o desenvolvimento de aplicações que reproduzam os requisitos necessários em um ambiente virtual surgiram diversas tecnologias e bibliotecas que são capazes de fornecer, por exemplo, informações a respeito da posição ou da orientação de um determinado dispositivo, abstraindo da aplicação a forma exata como essa informação foi processada.

Quando são observados os avanços na área de engenharia de software, é possível observar que uma das formas de melhorar a qualidade do processo de desenvolvimento do software é a aplicação de abordagens que tenham por objetivo garantir uma evolução sustentável durante o seu ciclo de vida. Para aplicações de RV, uma estrutura de dados denominada de grafo de cena foi criada com intuito de lidar com o processo de geração e manipulação de objetos em um ambiente virtual, possibilitando que os artefatos produzidos no software fossem, posteriormente, reutilizados com maior facilidade. Em geral, práticas de reuso de software que já foram adequadamente testadas contribuem para reduzir a possibilidade de defeitos serem introduzidos durante a produção do mesmo, como apresentado em pesquisas anteriores (MOHAGHEGHI *et al.*, 2004).

Figura 4 – Estrutura do grafo de cena em um ambiente virtual




Fonte: adaptado de (Sun Microsystems, Inc., 1999)

Um grafo de cena é uma estrutura de dados utilizada para representar as relações hierárquicas em um conjunto de objetos de uma cena tridimensional de RV (WALSH, 2002).

Destaca-se a sua utilização em aplicações e jogos computacionais modernos, além de programas computacionais que lidam com representações gráficas espaciais, como *Acrobat 3D*, *Ilustrator*, *Auto CAD* e *Corel Draw* (REVIEWS, 2016).

Um grafo de cena é, primariamente, composto por uma coleção de vértices, ligados por meio de arestas, constituindo uma estrutura semelhante a uma árvore. Os nós dessa árvore podem ter diversos filhos, mas, comumente, apenas um único pai. Dessa forma, possibilita que o efeito de uma operação, aplicado a um pai, seja, sumariamente, aplicado para todos os seus filhos. Tal abordagem permite processar operações de forma mais eficiente, agrupando objetos de formas relacionadas em "objetos compostos", possibilitando que uma única operação possa ser aplicada, simultaneamente, a todos os objetos-filhos que compõem aquele objeto (CHANG *et al.*, 2022).

Um ambiente virtual é constituído de uma série de aspectos do mundo real ou imaginário, de uma aplicação. De acordo com Ferreira (1999), os principais aspectos que podem ser representados na estrutura do grafo de cena são:

- **descrição geométrica:** trata-se em representar a forma do objeto a ser processado. Ferreira (1999) destaca que a descrição geométrica pode ser classificada no tempo como estática ou dinâmica. A descrição geométrica estática não varia a forma de um objeto no tempo, enquanto que na descrição dinâmica sua forma pode ser modificada;

- **aparência:** influencia na qualidade da imagem final. A aparência pode ser dada de diversas maneiras. Entre elas, pode-se destacar: ausência de aparência, cores, material, textura, brilho, sombra e reflexão. Além disso, ainda é possível haver combinações entre cada tipo aparência;

- **transformação:** qualquer objeto posicionado no mundo virtual pode sofrer uma transformação geométrica (rotação, translação ou escala). As transformações estão diretamente relacionadas à hierarquia dos objetos, pois se um nó pai for transformado, seus nós filhos herdarão sua transformação;

- **comportamento:** pode ser classificado em duas categorias: determinístico e não determinístico. O comportamento determinístico é aquele que pode ser definido em função do tempo, enquanto que o comportamento não determinístico é imprevisível, por exemplo, quando trata-se do reflexo de ações do usuário que não seguem um padrão definido;

- **câmera:** é a visão do mundo virtual. Geralmente é uma câmera de projeção perspectiva; e

- **iluminação:** várias fontes de luz podem ser adicionadas à cena, logo, nenhum tipo de impedimento referente à quantidade de luz pode ser empregado. Existem vários tipos de modelos de iluminação, dentre os quais podem ser destacados o proposto por Gouraud (1971), pois o seu cálculo é realizado nas placas gráficas atuais.

É importante destacar que cada uma dessas propriedades é inserida no grafo de cena, a fim de representar os objetos presentes no ambiente virtual. As aplicações utilizam o grafo de cena para organizar a geometria. A estrutura de um grafo de cena e seus componentes é apresentada na Figura 4.

As implementações de grafos de cena expõem um conjunto de funcionalidades que têm como objetivo facilitar o gerenciamento da geometria e estado de objetos que compõem uma cena. Com isso fornecem recursos que possibilitam uma organização espacial intuitiva, reduzem a carga geral de processamento da aplicação ao evitar processamento desnecessários, permitem renderizar objetos de forma eficiente em diferentes níveis de detalhe, maximizam o desempenho das aplicações evitando mudanças de estado redundantes e desnecessárias, além de fornecerem funcionalidades de alto nível, como efeitos de renderização (como efeitos de partículas e sombras), otimização de renderização, suporte a E/S (entrada/saída) de arquivos de modelo *3D* (MARTZ, 2007; CHANG *et al.*, 2022).

## 2.6 Design e Implementação de Aplicações de RV

Nos primórdios do desenvolvimento de aplicações de RV, Council *et al.* (1995) já apontavam para vários desafios que poderiam impactar o processo de design e implementação de programas de aplicação de RV.

Diferentemente de aplicações convencionais, arquiteturas de software para sistemas de RV tendem a possuir uma alta complexidade, devido ao fato de lidarem com diversos tipos de sistemas de entrada e saída pouco usuais, como HMDs, sistemas de rastreamento, dispositivos hápticos, mouses 3D, entre outros (CAPILLA; MARTÍNEZ, 2004; TOBLER, 2011; DUVAL, 2012).

Abaixo é apresentada uma lista, catalogada por Hempe (2016), dos principais *frameworks*, *toolkits* ou *game engines*, que utilizam o conceito de grafo de cena, e auxiliam no processo de desenvolvimento de aplicações para o domínio de RV:

- ***OpenInventor*:** é considerado um dos primeiros sistemas 3D de grafos de cena amplamente aceito. Foi desenvolvido como um sucessor do *Iris Inventor* (STRAUSS, 1993; WERNECKE, 1993). É otimizado para renderização de alto desempenho, utilizando funções de *pipeline* para os primeiros *hardwares* gráficos existentes (<http://oss.sgi.com/projetos/inventor>);

- ***OpenSG*:** sistema de grafos de cena de código aberto para aplicações gráficas em tempo real. Suas principais características são o foco no desempenho de renderização, *multi-threading* e *clustering* (VOSS*et al.*, 2002). Também incorpora recursos de *hardware* gráficos modernos programáveis (<http://www.opensg.org>);

- ***OpenSceneGraph*:** API de código aberto que é amplamente utilizada em aplicações e projetos de pesquisa (BURNS; OSFIELD, 2004). Inspirada no *Iris Inventor*, o *OpenSceneGraph* fornece um encapsulamento de estado *OpenGL* mais eficiente para melhor desempenho de renderização. Contudo, a geração de estruturas dinâmicas do grafo de cena é mais restrita em comparação com *Inventor* e com *OpenSG* (TOBLER, 2011) (<http://www.openscenegraph.org>);

- ***OGRE3D7*:** uma das *engines* de renderização, com código aberto, mais populares. É totalmente orientado a objetos, além de possuir modernas técnicas de renderização e uma separação limpa entre a hierarquia da cena e classificação de estados (<http://www.ogre3d.org>);

- ***Irrlicht Engine*:** *engine* de código aberto com uma grande comunidade de desenvolvimento. É conhecido por seu tamanho pequeno, bom desempenho, técnicas de renderização modernas e boa compatibilidade com *hardwares* gráfico novos e antigos (<http://irrlicht.sourceforge.net>);

- ***Unreal Engine*:** é uma *game engine* desenvolvido pela *Epic Games* e atualmente é usado em uma ampla gama de videogames comerciais. As versões mais antigas são de código fonte aberto e são aplicadas em sistemas de simulação (<http://www.unrealengine.com>);

- ***Unity3D*:** *game engine* que inclui vários elementos disponíveis para auxiliar o desenolvimento de aplicaões em aspectos relacionados física, sons e ferramentas para o desenvolvimento de jogos. Além do *Unreal Engine*, o *Unity3D* é considerado uma das *game engines* mais populares (NGUYEN; CORPORATION, 2008) (<http://www.unrealengine.com>); e

- ***Visualization Library*:** *middleware* de código aberto, escrito em C++, para aplicações gráficas similar à *API OpenGL 4*. Fornece um mapeamento intuitivo de funcionalidades em *framework* orientado a objetos amigável. *VL* propõe evitar limitações arquiteturais do grafo de cena, promovendo a noção de separação de dados estruturais e espaciais (<http://www.visualizationlibrary.org>).

*Game engines* profissionais têm se tornado cada vez mais uma ferramenta de escolha para o desenvolvimento de realidade virtual, devido à adequação entre suas capacidades e os requisitos necessários para a criação de uma boa aplicação de RV. As *game engines* fornecem uma série de recursos, incluindo renderização de alta qualidade, simulações de física, iluminação em tempo real, *scripts* e editores *What You See Is What You Get* (WYSIWYG).

As *game engines*, *frameworks* e *toolkits* listados acima diferem com relação às suas principais características, representação gráfica e domínio de aplicação. É possível destacar uma série de vantagens na adoção de grafos de cena como modelo para a representar as relações hierárquicas de objetos em uma cena *3D*. Silva, Raposo e Gattas (2004) destacam as principais vantagens e desvantagens para a adoção de grafos de cena:

- **desempenho:** exploram técnicas de eliminação de objetos que não contribuem para o resultado final da imagem;
- **produtividade:** além de implementarem grande parte dos requisitos necessários para se desenvolver uma aplicação, possibilitam a aplicação de paradigmas como orientação a objetos, permitindo a reutilização de projetos;
- **portabilidade:** traduções da aplicação para execução em outros ambientes de hardware podem necessitar apenas de recompilação de código fonte; e
- **escalabilidade:** consiste no poder oferecido pelos grafos de cena para aumentar a complexidade da simulação de maneira controlada.

Com relação às desvantagens destaca-se, principalmente, o *overhead* relacionado com a travessia do grafo para resgatar um determinado objeto. Tal operação pode acarretar em uma perda de desempenho em ambientes muito complexos. Dessa forma, orienta-se um cuidado no momento da construção dos mesmos para evitar que tais operações se tornem um gargalo no ambiente modelado.

## 2.7 Considerações Finais

Esse capítulo descreveu os conceitos fundamentais relacionados à RV, que é uma das áreas do conhecimento abordadas nesta tese. O Capítulo 3 apresenta os conceitos fundamentais relacionados ao teste de software, que é o segundo pilar fundamental para o entendimento deste trabalho.

CAPÍTULO

3

# FUNDAMENTOS DE TESTE DE SOFTWARE PARA REALIDADE VIRTUAL

Neste capítulo são apresentados os principais conceitos sobre teste de software relacionados a esta tese, de modo a fornecer um embasamento teórico que é de fundamental importância para o entendimento a respeito do problema de pesquisa abordado.

Na seção 3.1 são apresentados conceitos básicos sobre teste de software. A seção 3.2 [1] discute os mecanismos para sistematização da atividade de teste de software, elencando as principais técnicas de teste utilizadas para esse fim. A seção 3.3 apresenta um conjunto de estudos existentes na literatura que versam a respeito de abordagens de teste voltadas para programas desenvolvidos para RV.

Com base na análise das técnicas e critérios de teste existentes, bem como no estado da arte de técnicas de teste voltadas para RV, a seção 3.4 discute um problema definido na literatura como "problema do oráculo de teste", culminando na Seção 3.5, que discute brevemente uma abordagem de teste que visa solucionar o problema discutido.

Por fim, a seção 3.6 discute abordagens para geração de dados de teste por meio de técnicas de aprendizado de máquina e a Seção 3.7 apresenta as considerações finais para este capítulo.

## 3.1 Introdução

O principal objetivo da atividade de teste de software é garantir que um produto de software esteja de acordo com a especificação para a qual o mesmo foi projetado (DELAMARO; MALDONADO; JINO, 2016). Técnicas e critérios de teste de software permitem que o desenvol-

[1] As seções destacadas constituem em parte de uma versão revisada e atualizada da dissertação de mestrado desenvolvida pelo aluno, disponível em: <https://goo.gl/J2dHUa>

vedor possa utilizar uma abordagem sistemática e teoricamente bem fundamentada para conduzir a atividade de teste (GALIN, 2018). Essas abordagens têm como principais objetivos auxiliar na garantia da qualidade dos casos de teste produzidos e identificar o maior número possível de defeitos no produto de software avaliado. Diversas técnicas são descritas na literatura para auxiliar a atividade de teste de software, variando essencialmente na origem das informações a serem utilizadas no processo de definição dos requisitos de teste.

A atividade de teste é comumente dividida em fases, em que cada etapa é responsável por garantir objetivos distintos no plano de teste do produto avaliado. A finalidade deste modelo é garantir que cada fase apresenta características únicas que servem para guiar o processo de teste de software. Delamaro, Maldonado e Jino (2016) definem as fases de teste como:

- **teste de unidade:** tem como objetivo testar as menores unidades que compõem o sistema (métodos, funções, procedimentos, classes, etc.). Espera-se encontrar com maior facilidade defeitos de programação, algoritmos incorretos ou mal implementados, estruturas de dados incorretas, limitando-se a lógica interna dentro dos limites da unidade;
- **teste de integração:** o foco principal é verificar as estruturas de comunicação entre as unidades que compõem o sistema. As técnicas de projeto de casos de teste que exploram entradas e saídas são as mais utilizadas durante a fase de integração (PRESSMAN; MAXIM, 2014); e
- **teste de sistema:** o objetivo é verificar se os requisitos satisfazem a especificação e se as funcionalidades do sistema foram implementadas corretamente, isto é, o sistema é testado como um todo procurando simular um ambiente de execução real.

Nota-se ainda que podem existir classificações alternativas na literatura (SOMMERVILLE, 2015). Tais classificações tendem a dividir as fases de teste destacadas acima em duas ou mais sub-fases.

## 3.2 Técnicas e Critérios de Teste

Lidar com o tamanho do domínio de entradas é um grande desafio tanto para pesquisadores, quanto para profissionais. Mesmo para programas simples, o número de entradas existentes pode ser considerado muito grande e em alguns casos, até infinito (DELAMARO; MALDONADO; JINO, 2016). Portanto, é necessário definir mecanismos capazes de sistematicamente particionar o domínio de entrada em subdomínios, a fim de possibilitar a redução do número de casos de teste necessários para atestar um grau de qualidade ao software que está sendo testado.

A solução para isso é a definição de critérios de teste, que consistem de um conjunto de regras para particionar o domínio de entradas em subconjuntos. Um critério de testes define elementos de um programa que devem ser exercitados durante a sua execução, guiando, dessa

forma, o testador no processo de projetar os casos de teste para o sistema. Um elemento a ser exercitado durante essa execução é definido como requisito de teste (AMMANN; OFFUTT, 2016). Um requisito de teste pode ser, por exemplo, um caminho de execução específico do software, uma funcionalidade obtida por meio da especificação, etc.

Dessa forma, as técnicas de teste de software existentes procuram estabelecer regras para definir subdomínios, com intuito de criar conjuntos de casos de teste que satisfaçam os requisitos pertencentes ao subdomínio abordado. Porém, nenhuma delas é suficiente para atestar de fato a qualidade dos testes produzidos. O cenário ideal é que as técnicas sejam utilizadas de forma complementar, considerando a fase em que são aplicadas, de modo que seja possível extrair as a vantagens de cada abordagem (JAMIL *et al.*, 2016).

As próximas subseções exemplificam brevemente as técnicas de teste funcional, estrutural e baseada em defeitos. São abordados conceitos a respeito de como as técnicas são definidas, bem como apresentados alguns dos critérios mais utilizados para cada uma das técnicas. Por fim, também são discutidas iniciativas que foram desenvolvidas para adaptar a utilização de tais técnicas de Software ao contexto de RV.

### 3.2.1 Teste funcional

A técnica de teste funcional visa observar o software como uma caixa preta, devido ao fato de não utilizar detalhes de código fonte para a derivação de requisitos. Assim, a atividade é conduzida utilizando como informação artefatos produzidos durante a fase de elicitação dos requisitos de software ou com modelos produzidos durante a etapa de modelagem para representar as especificações das suas funcionalidades. Essa técnica ignora os mecanismos internos do programa e foca apenas nas saídas geradas em resposta às entradas utilizadas e às condições de execução (FABBRI; VINCENZI; MALDONADO, 2016).

Como todos os critérios da técnica funcional baseiam-se em artefatos produzidos durante as etapas de levantamento de requisitos e modelagem do sistema, é de fundamental importância que estejam corretos e sejam equivalentes entre si, para que seja possível garantir a criação de casos de teste efetivos em detectar defeitos no produto avaliado.

### 3.2.2 Teste estrutural

Na técnica de teste estrutural, os critérios e requisitos são derivados exclusivamente a partir das características do código-fonte do software. Portanto, os casos de teste são projetados essencialmente a partir de detalhes da estrutura interna do código-fonte do programa. Essa abordagem possibilita que seja dada uma maior atenção a pontos do código-fonte considerados críticos (KOSCIANSKI; SOARES, 2007).

A partir do código-fonte é definido um conjunto de elementos de software que devem ser executados para que se atinja a cobertura mínima para um determinado critério. A maioria dos

critérios dessa técnica utiliza uma representação do programa conhecida como Grafo de Fluxo de Controle (GFC) (MALDONADO, 1991).

A técnica estrutural tem como objetivo ser aplicada a pequenas partes do código, como sub-rotinas, ou operações relacionadas a uma unidade. Dessa forma, o testador pode analisar o código antes de iniciar os testes e criar casos que possuam dados que possivelmente possam ser tratados de forma inesperada. Outra vantagem da utilização de critérios de teste baseados na técnica estrutural é a possibilidade de poder garantir que o programa tenha seus comandos executados pelo menos uma vez por pelo menos um caso de teste (BARBOSA *et al.*, 2016).

### 3.2.3 Teste baseado em defeitos

Na técnica baseada em defeitos são utilizados defeitos típicos encontrados durante o processo de implementação de um software para derivar os requisitos de teste. A técnica baseada em defeitos baseia-se em modelos de defeitos de um determinado domínio para criar hipóteses sobre possíveis defeitos que podem estar presentes no programa em teste (DEMILLO; LIPTON; SAYWARD, 1978). Permite criar conjuntos de casos de teste ou avaliar conjuntos de testes com base na sua eficácia em revelar os defeitos hipoteticamente modelados.

Eles representam defeitos típicos criados pelos desenvolvedores de software e os requisitos de teste da técnica são derivados por meio destes defeitos. No entanto, para não inviabilizar a aplicabilidade da técnica, somente defeitos considerados plausíveis devem ser inseridos no programa a ser avaliado (DELAMARO *et al.*, 2016).

## 3.3 Teste de Software Para Programas de Realidade Virtual

Conforme discutido anteriormente, a grande expansão da adoção de RV para o desenvolvimento de aplicações em diversas áreas trouxe novos desafios para as atividades de garantia de qualidade de software. Por exemplo, sistemas de softwares desenvolvidos para o contexto de RV apresentam estruturas não convencionais de software (Seção 2.5), o que pode representar novas fontes de defeitos para os programas desenvolvidos. Esses novos desafios motivaram o desenvolvimento de diferentes abordagens que têm por objetivo contribuir para o processo de garantia de qualidade de software no contexto de RV.

Uma característica evidenciada no processo de desenvolvimento de aplicações para o domínio de RV é que, em geral, a atividade de teste é, normalmente, realizada de maneira manual e, em sua maioria, é conduzida apenas após o término do desenvolvimento das mesmas (SCHLUETER *et al.*, 2017). Tais técnicas auxiliam na geração tanto de requisitos de testes funcionais, quanto para requisitos não funcionais (DELAMARO; MALDONADO; JINO, 2016).

Uma característica que difere aplicações de RV de aplicações de propósito geral é a forma como o usuário interage como a mesma. Para uma dada tarefa podem existir diversas maneiras distintas de interagir com a aplicação de forma a possibilitar a sua conclusão. Assim, os dados de entrada e saída gerados podem ser completamente diferentes ao comparar a utilização de dois usuários em uma mesma aplicação visando executar uma tarefa semelhante.

A característica descrita acima descreve uma das dificuldades de automatizar o processo de teste para aplicações de RV. Ao longo do tempo diversos trabalhos foram desenvolvidos visando lidar com o problema de teste de software para aplicações de RV, abaixo são apresentados os principais trabalhos e como eles lidam com parte desse problema a partir da proposição de diferentes abordagens.

Kuutti *et al.* (2001) descrevem o estudo de um protótipo virtual tridimensional voltado para testes de usabilidade e validação de conceitos por meio da internet. Um telefone conceitual foi construído utilizando *virtual reality modeling language (VRML)* e os participantes do estudo foram convidados a avaliar o modelo conceitual seguindo duas abordagens: *(i)* um grupo utilizou uma abordagem *ad-hoc*, enquanto outro grupo utilizou uma abordagem tradicional de testes constituída de análise de usabilidade.

Guo, Zhou e Zhu (2003) desenvolveram um sistema de teste chamado *VR-based test and simulation system (VTSS)*. Tal sistema permite que a atividade de teste seja planejada de forma interativa, otimizada e simulada, a fim de possibilitar uma verificação completa em ambientes virtuais. A abordagem utiliza técnicas de *case-based reasoning* para possibilitar uma abordagem inteligente no processo de teste ao possibilitar o gerenciamento detalhado de esquemas que compõem a aplicação em teste.

Bierbaum, Hartling e Cruz-Neira (2003) descrevem uma técnica para apoiar o teste de interfaces em aplicações de RV. A abordagem monitora uma lista de testes, previamente projetada, e tem como finalidade garantir o comportamento correto das aplicações em teste. Para isso, o estado das aplicações é monitorado e, quando a aplicação atinge um estado que necessita ser testado, o teste é executado, de maneira automática, a fim de verificar se a aplicação se comportou de maneira adequada ou não.

Florczyk e Winiecki (2005) apresentam um método paramétrico, baseado em fluxo de dados, para testes funcionais de instrumentos virtuais. A abordagem possibilita a criação de testes durante o desenvolvimento da interface de usuário. Os testes podem ser realizados em diferentes níveis de generalidade e por meio da escolha dos elementos mais importantes da interface do usuário. O método paramétrico para testes funcionais pressupõe que o usuário deve agir de maneira não convencionalmente durante utilização da aplicação, partindo do pressuposto que o usuário não costuma seguir as instruções incluídas no manual do dispositivo.

Xiao *et al.* (2005) um *framework* de aprendizagem ativa para a técnica de testes funcional. A abordagem de aprendizagem ativa utiliza casos de testes e, a partir deles, aprende um modelo

comportamental da aplicação. Esse modelo é então utilizado para que sejam derivados novos casos de teste para avaliar o comportamento da aplicação. A abordagem possibilita o desenvolvimento de testes principalmente durante a construção da interface do usuário. Esse *framework* foi concebido para aplicação de testes para jogos de videogames comerciais e mundos virtuais complexos.

Zhao, Xu e Li (2007) apresentam e analisam técnicas de teste de software aplicadas em um sistema de simulação digital de foguetes. Quatro métodos de teste e suas combinações foram aplicados durante o processo de teste para verificar as seguintes estruturas de: *(i)* especificações, *(ii)* software, *(iii)* códigos e *(iv)* estruturas internas. O processo de teste inclui a produção de artefatos como especificações , testes de unidade, testes de integração, testes de sistema e testes de performance. Ao fim do estudo, os autores apresentaram que a combinação das quatro abordagens de teste permitem garantir uma melhor qualidade no produto entregue e desempenham papel fundamental, mostrando-se muito eficiente, principalmente, para atividade de teste de software em sistemas que possuem uma grande complexidade.

Torens e Ebrecht (2010) apresentam uma abordagem para possibilitar o desenvolvimento da atividade de teste para sistemas distribuídos complexos, que lidem com interfaces de comunicações heterogêneas. Isso é feito integrando componentes individuais do sistema em um ambiente virtual que emula os sistemas constituintes. Os detalhes da interface são abstraídos e não é necessário ao testador conhecimento especial em relação a como é realizada essa comunicação, reduzindo a complexidade da atividade de testes para o cenário de aplicações distribuídas.

Kim, Son e Kim (2011) propõem um método para geração de casos de teste de maneira automática, utilizando como base um diagrama de estado da *Unified Modeling Language* (UML), que representa o comportamento do ambiente virtual testado. A abordagem tem como objetivo reduzir a diferença de tempo entre a prototipagem da aplicação e a execução da atividade de teste, possibilitando, consequentemente, a descoberta de falhas durante a fase de projeto da aplicação.

Bezerra, Delamaro e Nunes (2011) partem da premissa que é possível ver um grafo de cena (Seção 2.5) como um modelo e uma abstração de um programa e, portanto, pode-se pensar na utilização de critérios de teste que utilizem esse modelo para derivar requisitos de teste, semelhante a critérios estruturais, que utilizam um grafo de fluxo de controle para derivar os requisitos de teste. Dessa forma, o grafo de cena é utilizado para selecionar estruturas a serem exercitadas durante a atividade de teste. Foi estabelecido um conjunto de critérios de testes baseados em grafos de cena. Tais critérios inspiram-se em concepções desenvolvidas ao considerar um grafo de cena como uma abstração de uma aplicação RV, similar a uma representação de GFC. Como forma de apoio à aplicação de tais critérios, foi desenvolvida uma ferramenta para teste de aplicações de RV denominada *Virtual Environment Testing* (VET*esting*).

Chen *et al.* (2013) apresentam uma versão modificada para aplicação de teste de usabilidade em grupo. A abordagem, denominada de *Modified Group Usability Testing (MGUT)*, guia os

usuários durante a atividade de testes ao auxiliá-los em como relatar problemas de usabilidade no ambiente em teste. Em uma abordagem convencional do teste de usabilidade, apenas os usuários são convidados a fazer relatos a respeito das percepções encontradas durante o desenvolvimento das tarefas de teste propostas. Isso pode resultar na perda de alguns dados, pois os observadores responsáveis por registrar os problemas de usabilidade podem não ser capazes de observar e registrar diferentes problemas que são encontrados/enfrentados simultaneamente por diferentes usuários. Na abordagem modificada proposta no artigo, além de ter os observadores para fazer a gravação de suas observações, os usuários também são convidados a analisar brevemente os problemas de usabilidade que eles encontram durante a sessão de teste.

Correa, Nunes e Delamaro (2018) desenvolveram uma abordagem denominada de *Virtual Reality-Requirements Specification and Testing* (VR-ReST), que tem por objetivo apoiar a especificação de requisitos de aplicações de RV com base na descrição de casos de uso visando derivar requisitos de teste e auxiliar na geração de dados de teste a partir dos requisitos especificados. Para auxiliar a especificação desses requisitos, foi proposta uma linguagem específica de domínio, denominada de *Behavior Language Requirement Specification* (BeLaRS). A linguagem foi concebida para garantir que tal especificação seja mais formal, precisa, sem ambiguidade e mais próxima do nível de compreensão do usuário.

A partir da criação dos requisitos, por meio da linguagem específica de domínio BeLaRS, os mesmos são convertidos em um grafo de fluxo de requisitos, que representa cenários do software a ser testado. A partir do grafo de fluxo de requisitos, os requisitos de teste são derivados e em seguida, casos de teste são gerados utilizando um algoritmo de busca em profundidade de acordo com um dos critérios de teste utilizados.

Gil *et al.* (2020) propuseram uma estrutura de teste automatizado para aplicações de RV, desenvolvidas para *Android* utilizando a plataforma *Unity*. A estrutura utiliza *scripts* desenvolvidos em *Python* e cria uma abstração na forma de um canal de comunicação para interagir com uma instância em execução da aplicação, possibilitando acesso às propriedades aplicação como, por exemplo, a posição dos elementos dispostos na cena e interação com as interfaces do usuário. Com isso é possível a criação manual de testes de unidade para verificar o comportamento esperado dos componentes na cena de acordo com o cenário de teste a ser avaliado.

Corrêa *et al.* (2021) apresentam uma abordagem para automatizar o teste em aplicações de RV com interfaces hápticas - interfaces que permitem a comunicação bidirecional durante a interação humano-computador, capturando movimentos e fornecendo *feedback* de toque. A abordagem lida com a complexidade e as características das interfaces hápticas utilizando uma abordagem de *Record and Playback* registrando informações de casos de teste (entradas e saídas) para testar novas versões (teste de regressão), possibilitando testar novas versões de aplicações de RV com interfaces hápticas, mesmo se o dispositivo físico não estiver disponível, além de localizar módulos defeituosos da aplicação.

Nusrat *et al.* (2021) exploraram projetos de código aberto, desenvolvidos para *Unity3D*,

para entender como os desenvolvedores otimizam aplicações de RV. Eles categorizaram manualmente 11 categorias diferentes de tipos de otimizações e desenvolveram uma taxonomia práticas para otimizações de desempenho de RV. Apesar dos resultados não envolverem diretamente uma abordagem de teste de software, a taxonomia proposta pode ser utilizada para explorar abordagens de teste de software que visam identificar falhas relacionadas a problemas de otimização.

Além dos trabalhos abordados, destacam-se ainda trabalhos que estão em estágio de desenvolvimento e apresentaram propostas iniciais para validação da comunidade cientifica. Entre eles Prasetya *et al.* (2021) e Ricós (2022) compõem um grupo de pesquisa que tem explorado desenvolver aspectos de teste no contexto de aplicações de realidade estendida. As proposições iniciais do grupo têm direcionado a pesquisas para a definição de uma abordagem que utiliza agentes para auxiliar na geração de dados de teste para possibilitar a realização de testes automatizados, contudo, os trabalhos ainda carecem de validação e encontram-se em estágios preliminares de concepção.

Automatizar a atividade de testes de software é muitas vezes um processo difícil e complexo. As principais tarefas desta atividade compreendem: *(i)* organizar, *(ii)* executar e *(iii)* registrar a execução dos casos de teste, além de *(iv)* verificar o resultado da execução dos mesmos (KANER; PADMANABHAN; HOFFMAN, 2013). Para tal, é necessário um conjunto de valores que compõem os dados de entrada para a execução do programa e valores correspondentes ao resultado esperado para cada teste.

No contexto de aplicações de RV, destaca-se a dificuldade de sistematizar como o comportamento de um caso de teste. Esta dificuldade é descrita na literatura como “problema do oráculo de teste"e é evidenciada em cenários nos quais os meios tradicionais de medir a execução de um caso de teste são impraticáveis ou de pouca utilidade para julgar a correção de saídas geradas a partir dos dados do domínio de entrada.

A próxima seção apresenta a definição de oráculos de teste e alternativas a serem utilizadas para lidar com as limitações expostas pelo “problema do oráculo de teste".

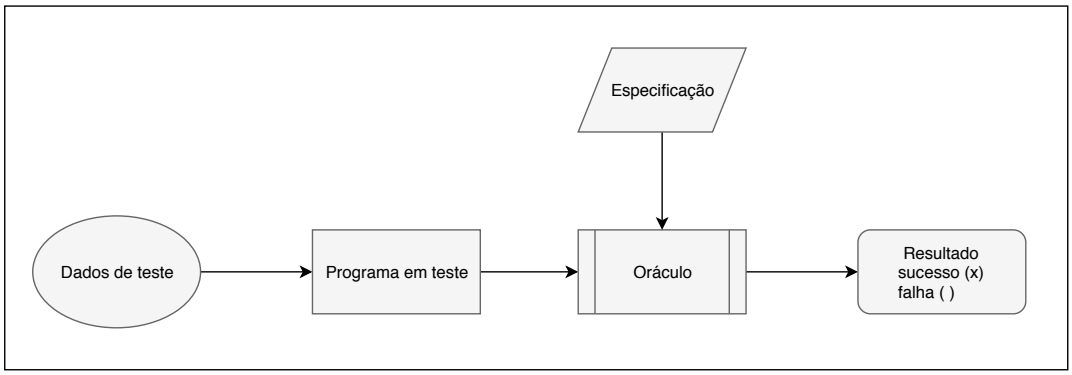
## 3.4 Oráculos de Teste

Os trabalhos descritos na seção anterior (Seção 3.3) esbarram na dificuldade de sistematizar como o comportamento de um caso de teste pode ser aferido no contexto de programas de RV. Essa dificuldade é descrita na literatura como “problema do oráculo de teste” e configura-se em casos, nos quais, meios tradicionais de aferição da execução de um caso de teste são impraticáveis, ou são pouco úteis para julgar a correção de saídas geradas a partir dos dados do domínio de entrada (RAPPS; WEYUKER, 1985; BARR *et al.*, 2015).

Os oráculos de teste lidam com uma questão primária nas atividades de teste de software

– decidir sobre a corretude do programa, com base em dados de teste predeterminados. A Figura 5 apresenta, utilizando elementos de um fluxograma, a estrutura tradicional e genérica de um oráculo de teste. Nesse caso, o oráculo de teste acessa um conjunto de dados necessários para avaliar a correção da saída do teste. Este conjunto de dados é extraído da especificação do programa em teste e deve conter informações suficientes para apoiar a decisão final do oráculo (OLIVEIRA; KANEWALA; NARDI, 2014).

Figura 5 – Estrutura genérica do oráculo de teste




Fonte: Adaptado de (OLIVEIRA; KANEWALA; NARDI, 2014).

A atividade de teste consiste em examinar o comportamento de um sistema a fim de atestar a qualidade do produto e descobrir possíveis comportamentos inesperados. Essa atividade funciona como uma espécie de mecanismo de estímulo, em que o software em teste é estimulado e, posteriormente, o seu comportamento é observado. Contudo, para grande parte desses sistemas, normalmente, o testador não possui especificações, ou artefatos formais que possam auxiliar na condução da atividade de teste (HU *et al.*, 2006). Dessa forma, o testador acaba incumbido de verificar manualmente o comportamento do sistema para todos os cenários de teste projetados.

Para possibilitar a automatização da atividade de testes é necessário, portanto, um esforço para encontrar formas de abordar o "problema do oráculos de teste" e integrar tais soluções em técnicas de teste.

Na literatura diversos autores propuseram teorias e taxonomias com intuito de criar uma classificação para os diversos tipos de oráculos (MCDONALD; HOFFMAN; STROOPER, 1998; BARESI; YOUNG, 2001; HOFFMAN, 2001; BEIZER, 2003; STAATS; WHALEN; HEIMDAHL, 2011; HARMAN *et al.*, 2013; PEZZè; ZHANG, 2014; OLIVEIRA; KANEWALA; NARDI, 2014). Contudo, devido à diversidade de sistemas e domínios de aplicação, ainda não há uma padronização a respeito e as classificações costumam categorizar oráculos com base na fonte de informações utilizada para derivá-lo, como, por exemplo: *(i)* saída gerada pelo software em teste e *(ii)* pela técnica utilizada para automatizar o oráculo.

Este projeto utiliza como base a classificação proposta por Oliveira, Kanewala e Nardi (2014), que conduziram um mapeamento sistemático sobre o tema e propuseram quatro categorias

capazes de generalizar oráculos de acordo com as informações utilizadas para a derivação dos mesmos:

- **oráculos baseados em especificações:** tecnicamente, uma especificação pode ser definida como uma formulação detalhada, que fornece uma descrição definitiva de um sistema com o objetivo de desenvolver ou validar o mesmo (CHAKRABORTY *et al.*, 2012). Quando essa formulação é usada para validação, é considerada como a informação para derivação do oráculo.

  Segundo Li e Offutt (2017), as linguagens utilizadas para descrever especificações são denominadas de linguagem de especificação e podem ser executadas por compiladores ou interpretadores, possibilitando a criação de procedimentos para a criação de oráculos automatizados. No mesmo estudo, Li e Offutt (2017) apresentam uma lista compreensiva com dez abordagens para a utilização de especificações como fonte para geração de oráculos de teste;

- **oráculos baseados em relações metamórficas:** as informações do oráculo podem não ser baseadas diretamente na especificação do sistema, mas sim em relacionamentos conhecidos entre as entradas e saídas do programa em teste (SEGURA *et al.*, 2016);

- **oráculos baseados em aprendizado de máquina:** aprendizado de máquina é um conjunto de métodos computacionais, que utilizam dados coletados a fim de possibilitar previsões a respeito de determinados eventos (MITCHELL, 1997). Existem diversas técnicas de aprendizado de máquina supervisionado aplicadas como oráculos de teste, como, por exemplo: *(i)* redes neurais artificiais (MAO *et al.*, 2006), *(ii)* máquinas de vetores suporte (WANG *et al.*, 2011) e *(iii)* redes de inferência *fuzzy* (AGARWAL *et al.*, 2012);

- **oráculos baseados em versões anteriores do software:** oráculos baseados em versões anteriores do software são mecanismos que exploram diferentes versões do software em teste para apoiar as decisões sobre a corretude da execução de um teste.

  Kazmi *et al.* (2017) apresentam uma revisão sistemática sobre teste de regressão e destacam que 4% dos estudos analisados exploram a utilização dessa abordagem para auxiliar na geração de oráculos de teste. Essas técnicas têm por objetivo tentar prever a taxa de falha ou a eficácia dos conjuntos de teste para um uso futuro. Contudo, os autores alertam que, a criação e manutenção de tais bases de informação são consideradas como dispendiosas e podem consumir recursos extras.

O *survey* conduzido por Segura *et al.* (2016) destaca os domínios de aplicação explorados para aplicação de testes metamórficos. O estudo aponta que 12% dos estudos levantados discutem a aplicação da técnica para teste de programas de computação gráfica. Os estudos abordam testes metamórficos para avaliar aplicações que lidam com simplificação de *meshs*, processamento de imagem, visibilidade de superfícies tridimensionais e softwares de imagem em geral.

Os resultados levantam indícios de que testes metamórficos podem ser uma boa alternativa para solucionar o problema do oráculo de teste para aplicações no contexto de RV. Dessa forma, a próxima seção apresenta uma breve descrição a respeito de testes metamórficos.

## 3.5 Teste Metamórfico

Testes metamórficos visam verificar se o programa em teste se comporta de acordo com um conjunto de Relações Metamórficas (RM). Uma relação metamórfica tem como objetivo especificar como uma mudança particular em um dado de entrada do programa pode alterar o seu comportamento e, consequentemente, a sua saída (CHEN; YIU, 1998).

Segundo Chen, Tse e Zhou (2003), é possível expressar uma dada relação metamórfica $R$, entre múltiplas entradas e saídas. Portanto, dada uma função $f$ e um conjunto de entradas $x_1$, $x_2$, $x_3$, ..., $x_n$, para $n > 1$ e suas respectivas saídas $f(x_1), f(x_2), f(x_3)$, ..., $f(x_n)$, é possível verificar a corretude da sua execução mesmo que a saída correta das execuções individuais seja desconhecida. Isso é possível ao verificar se a relação metamórfica não é violada durante a execução das funções.

A violação de uma relação metamórfica ocorre quando uma mudança na saída difere do que é predito pela relação metamórfica considerada. Como exemplo, uma função *seno* deve, sempre, obedecer a propriedade descrita na Equação 3.1.

$$seno(\theta) = seno(180 - \theta) \tag{3.1}$$

Tal propriedade da função *seno* pode não ser parte da sua especificação. Contudo, essa propriedade especifica um relacionamento entre um par de dados de entrada, $(\theta)$ e $(180 - \theta)$, e suas respectivas saídas $seno(\theta)$ e $seno(180 - \theta)$.

Dessa forma, para o exemplo, um dado de teste de acompanhamento pode ser criado, subtraindo *180* de um dado inicial de entrada. Isso possibilita que a corretude da implementação da função *seno* possa ser verificada com base na execução do dado de teste inicial e de novos dados de teste, observando se a saída produzida pelos pares de entrada é igual.

$$(3,177),(15,165),(45,135),(125,55),(277,-97)$$

Contudo, satisfazer uma relação metamórfica não garante que o programa em teste esteja implementado corretamente. No entanto, segundo Kanewala e Bieman (2013), uma violação da relação metamórfica indica que o programa em teste contém falhas.

### 3.5.1 Processo para aplicação de testes metamórficos

Segura *et al.* (2016) apresentam que o processo básico para a aplicação de testes metamórficos pode ser descrito com o desenvolvimento de três etapas:

#### 3.5.1.1 Identificação de relações metamórficas

Considerado um passo fundamental para o sucesso na aplicação da abordagem, o primeiro passo é a identificação de um conjunto de relações metamórficas. As relações metamórficas são derivadas utilizando um conhecimento a respeito do domínio de aplicação do software em teste.

Atualmente, testes metamórficos consideram apenas características funcionais para a derivação de relações metamórficas. As abordagens descritas na literatura para a definição das relações metamórficas envolvem *(i)* desenvolver relações metamórficas com base na especificação do software a ser testado e *(ii)* desenvolver relações metamórficas com base em requisitos esperados pelo usuário.

A primeira abordagem visa estabelecer que propriedades necessárias para a correta execução do software em teste sejam atendidas. A segunda abordagem nem sempre resulta em propriedades necessárias para que o programa funcione conforme o esperado, contudo, ela visa garantir que o software atenda a conjuntos de requisitos específicos desejados pelos usuários do mesmo.

É importante destacar que nem todas as relações metamórficas têm a mesma capacidade de detecção de falhas. Dessa forma, relações metamórficas que procuram identificar uma relação de igualdade entre dados de teste têm uma melhor prospecção para identificação de falhas, uma vez que é mais fácil de se identificar quando uma relação de igualdade, entre dados, é violada (XIE *et al.*, 2011).

#### 3.5.1.2 Criação de conjunto de dados de teste

A aplicação de testes metamórficos requer a criação de casos de teste iniciais e casos de teste de acompanhamento. Os casos de teste iniciais podem ser criados utilizando uma técnica de teste convencional, como técnica estrutural, funcional, baseada em defeitos ou teste aleatório. Em seguida, os casos de teste de acompanhamento são criados alterando os casos de teste iniciais de acordo com as modificações exigidas pelas relações metamórficas (KANEWALA; BIEMAN, 2013). Relações metamórficas que garantem que o rastro de execução dos dados de teste iniciais seja diferente dos rastros de execução dos dados de teste de acompanhamento tendem a ser mais efetivas que aquelas com rastro de execução idêntico para ambos os dados de teste.

#### 3.5.1.3 Detecção de falhas

Cada um dos pares de testes iniciais e de acompanhamento são executados para verificar se a alteração na saída está prevista pela relação metamórfica correspondente a ser avaliada.

A violação de uma relação metamórfica em tempo de execução indica uma possível falha no programa em teste. Uma vez que os testes metamórficos verificam a relação entre dados de entrada e dados de saída do programa em teste, o método possibilita atestar a conformidade do programa mesmo quando o resultado de execuções individuais do mesmo é desconhecido (CHEN *et al.*, 2018).

Se os dados de teste iniciais forem gerados automaticamente, a aplicação do teste metamórfico permite uma automatização completa da atividade de teste, ou seja, geração dos dados de entrada e verificação da saída. No exemplo *seno* apresentado no início dessa seção, o teste metamórfico pode ser utilizado em conjunto com uma técnica de testes aleatórios para gerar automaticamente dados de teste iniciais para $(\theta)$ e os respectivos casos de teste de acompanhamento para $(180 - \theta)$, até encontrar um par de dados que viole a relação metamórfica, ou até que se atinja um *threshold* de execução.

O *survey* publicado por Segura *et al.* (2016) investigou 119 artigos, publicados entre janeiro de 1998 e maio de 2016, sobre aplicação de testes metamórficos. Como resultados, os autores apontaram que o domínio mais popular para aplicação da técnica é serviços e aplicações da web (14%), seguido por computação gráfica (11%). Também foi encontrada uma variedade de aplicações para outras áreas (24%), como aplicações financeiras, programas de otimização, segurança cibernética e análise de dados, bem como softwares industriais de organizações, como *NASA* e *Adobe*. Apenas 5% dos artigos relataram resultados em programas numéricos, apesar deste ser um dos domínios mais utilizados para ilustrar o funcionamento de testes metamórficos na literatura. Os resultados também indicaram um crescente aumento na aplicação na área de bioinformática e inteligência artificial, incluindo aprendizado de máquina e veículos autônomos.

## 3.6 Geração Automática de Dados de Teste

O processo de geração automática de dados de teste é uma etapa fundamental dentro da atividade de teste de software e permite que um sistema possa ser verificado de forma intensiva. Apesar de ser uma tarefa desejável, a automatização da geração de dados de teste não é trivial. Tal dificuldade se dá, principalmente, em função da complexidade dos diferentes domínios de software existentes.

O dado de teste e sua respectiva saída esperada são os elementos que compõem um caso de teste. Assim, a geração de dados de teste consiste em estratégias para criação de dados de entrada que possam popular a aplicação e permitir a sua execução (RAMAMOORTHY; HO; CHEN, 1976). Tais dados podem assumir diferentes representações, como serem utilizados como parâmetros para a execução de uma função, definirem a ordem de execução de métodos, simularem a captação de dados de sensores, etc.

### 3.6.1 Aprendizado de máquina para geração de dados de teste

Além do volume cada vez maior de conjuntos de dados sendo constantemente produzidos e armazenados, a popularização da utilização de algoritmos de Aprendizado de Máquina (AM) se deu principalmente por dois fatores: a constante evolução do poder computacional e pela popularização de bibliotecas e ferramentas que facilitam a sua adoção (NAQA; MURPHY, 2015).

Soluções que utilizam AM abordam questões sobre como construir programas de computador que melhoram a sua performance em uma determinada tarefa por meio da experiência. Esta experiência é obtida por meio de um processo de treinamento, no qual um padrão pode ser extraído a partir de um conjunto de dados que o expressa. O processo de aprendizado seria uma busca em um espaço de possíveis soluções por aquela que melhor se adequá ao problema abordado. Para o componente de software que realiza a busca atribui-se o nome de algoritmo de aprendizado e ao produto que é obtido ao final do processo de treinamento, representando o padrão que foi aprendido a partir dos dados, é utilizada a denominação de modelo (BONACCORSO, 2017).

Para o contexto de teste de software, embora um algoritmo de AM não seja capaz de identificar todo o processo de avaliação de um software, ele pode detectar estruturas e padrões ocultos nos dados. Nesse contexto, o resultado do algoritmo é uma aproximação (ou seja, um modelo). Em um sentido amplo, algoritmos de AM processam os dados disponíveis com a finalidade de construir modelos. Os modelos resultantes incorporam padrões que permitem fazer inferências, que servem para alimentar o software em teste (MEINKE; BENNACEUR, 2018).

Essencialmente, os algoritmos de AM diferem na forma como os modelos são utilizados ou produzidos. Esses algoritmos podem ser classificados como algoritmos de aprendizado supervisionados (do inglês, *supervised learning*), não supervisionados (do inglês, *unsupervised learning*) ou por reforço (do inglês, *reinforcement learning*) (BONACCORSO, 2017).

Abaixo é descrito de forma superficial como a técnica de aprendizado por reforço é utilizada para auxiliar na tarefa da geração de dados.

#### 3.6.1.1 Aprendizado por reforço

Aprendizado por reforço (AR) é uma técnica de aprendizado de máquina que utiliza agentes para realizar uma determinada tarefa em um ambiente desconhecido por meio da coleta de recompensas. O agente é responsável por executar ações e coletar *feedback* de acordo com o estado atual do ambiente explorado. Portanto, o objetivo do agente é completar uma tarefa modelada, de tal forma que maximize a recompensa cumulativa esperada, encontrando uma sequência ótima (ou quase ótima) de ações (FRANÇOIS-LAVET *et al.*, 2018).

Uma tarefa ou objetivo é formalmente definida como um Processo de Decisão de Markov (MDP, do inglês *Markov Decision Process*) descrito em termos de um ambiente e de uma função

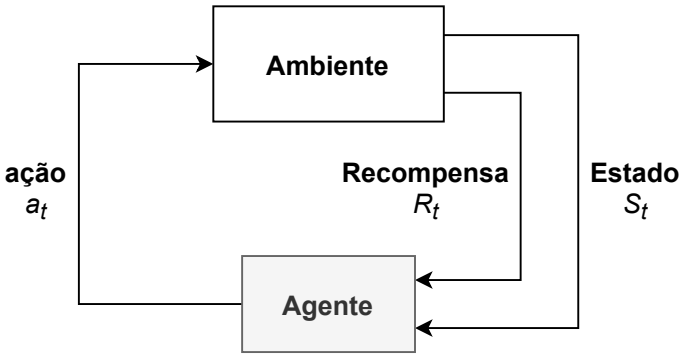
de recompensa. O ambiente determina o comportamento do mundo explorado, ou seja, o que acontece depois que alguma ação é realizada, enquanto a recompensa indica ao agente quanto ele ganhou (ou perdeu) com a execução de uma determinada ação, no estado atual do mundo (SUTTON; BARTO, 2018).

Um processo MDP pode ser descrito como uma tupla $\langle S, A, P, R, \gamma \rangle$, na qual, segundo Sutton e Barto (2018), cada elemento pode ser descrito da seguinte maneira:

- $S$: é o conjunto de estados do ambiente;
- $A$: representa o conjunto de ações possíveis;
- $P$: é uma função de transição $S \times A \rightarrow S$;
- $R$: é uma função de recompensa $S \times A \rightarrow \mathbb{R}$;
- $\gamma$: é um fator de desconto que $\in [0, 1]$.

A cada intervalo de tempo $t$, o agente observa o estado atual $S_t \in S$ e então realiza uma determinada ação $a_t \in A$. Após a ação, o agente recebe uma recompensa $r_t + 1 \sim R(s_t, a_t)$ de acordo com a tarefa para a qual foi projetado.

Figura 6 – Ciclo de aprendizado utilizando aprendizado por reforço




Fonte: Adaptado de (SUTTON; BARTO, 2018).

A Figura 6 resume o processo adotado durante a execução de um algoritmo de aprendizado por reforço. A cada intervalo de tempo, o agente altera o estado do ambiente executando uma ação. O efeito da ação é fornecido pelo ambiente como o próximo estado e recompensa. O agente usa a recompensa ou ambos os *feedbacks* para melhorar seu comportamento.

De forma geral, cada ciclo de estado-ação-recompensa é chamado de *step*. O sistema de aprendizado por reforço continua a iterar por ciclos até atingir o estado desejado ou até atingir um número máximo de *steps*. Essa série de *steps* é chamada de *episódio*. No início de cada episódio, o ambiente é definido para um estado inicial e a recompensa do agente é zerada.

O agente executa uma ação de acordo com uma política $\Pi$, que mapeia os estados do ambiente para possíveis ações que podem ser realizadas em cada estado. O objetivo do é, portanto, encontrar uma política ótima $\Pi^*$ que maximize o *retorno total esperado com desconto*. O *retorno total esperado com desconto* $G$ em um determinado momento $t$ é especificado como $G_t = \Sigma_{k=t+1}^{T} \gamma^{kt-1} R(s_k, a_k)$, onde $T$ é o número máximo de etapas.

Uma política pode ser representada de diferentes maneiras, por exemplo: como uma tabela, combinação linear ou como uma rede neural profunda. Além disso, uma política pode ser estocástica ou determinística. Uma política estocástica é uma distribuição de probabilidade sobre as ações, neste caso o objetivo seria maximizar a soma esperada de recompensas. Por outro lado, uma política determinística é uma função determinística dos estados em que apenas uma ação tem probabilidade diferente de zero (FRANÇOIS-LAVET *et al.*, 2018).

## 3.7 Considerações Finais

Este capítulo sumarizou os principais conceitos de teste de software relacionados à temática explorada nesta tese.

Um conjunto de estudos que lidam com a aplicação de técnicas de teste de software para programas no contexto de RV foram discutidos, mostrando que há um interesse na literatura sobre o tema, contudo ainda não há um consenso definitivo a respeito da aplicação de tais abordagens para auxiliar a condução da atividade de teste de software no contexto de RV. Os estudos evidenciaram principalmente uma dificuldade em lidar com oráculos de teste, que ainda são considerados como um problema de pesquisa em aberto a ser explorado.

Foram apresentados os conceitos básicos a respeito de oráculos de teste, bem como as principais taxonomias utilizadas para a classificação dos mesmos. Foram elencadas abordagens que estão sendo pesquisadas com intuito de superar o “problema do oráculo", dando destaque à técnica de teste de testes metamórficos, que tem por finalidade verificar a correção de um programa em teste por meio da execução de dados de testes que respeitem premissas definidas como relações metamórficas.

Dentro da perspectiva de utilização de testes metamórficos foi apontada utilização da técnica em diferentes domínios de aplicação, indicando a sua portabilidade de utilização independente de contexto. Especificamente, entre os domínios de aplicação explorados, vale destacar estudos de utilização de testes metamórficos no contexto de motores de busca *web* (ANDRADE *et al.*, 2019) e para aplicações que utilizam modelos de aprendizado de máquina supervisionado (SANTOS *et al.*, 2020), que foram trabalhos desenvolvidos no contexto desta tese para aprimorar os conceitos sobre o tema e verificar como a abordagem poderia ser utilizada para verificar a correção de diferentes tipos de aplicações na ausência de um oráculo.

Foram apresentados os conceitos e aspectos gerais de geração automática de dados de

teste. Dentre as abordagens mencionadas, foi dado destaque à abordagem que utiliza técnicas de aprendizado de máquina para auxiliar essa atividade. Foi dada atenção a abordagem que utiliza aprendizado por reforço, que é uma das abordagens utilizadas nos experimentos desenvolvidos neste trabalho.

O próximo capítulo aborda os conceitos apresentados nos capítulos de fundamentação teórica sobre RV e teste de software e apresenta um estudo a respeito da análise de repositórios de projetos de código aberto com relação à adoção de práticas de teste de software.

CAPÍTULO

4

# PRÁTICAS DE TESTE E PREDISPOSIÇÃO A FALHAS EM APLICAÇÕES DE REALIDADE VIRTUAL

Conforme discutido nos Capítulos 1 e 2, o rápido avanço tecnológico tem possibilitado o desenvolvimento de sistemas com novos recursos como imagens, sons, vídeos e interação diferenciada. Assim, tecnologias como a RV têm elevado as possibilidades de criações de ambientes tridimensionais com interação em tempo real.

Santos, Delamaro e Nunes (2013) apontam que há interesse na literatura sobre o assunto. No entanto, ainda não existe uma concepção sobre as práticas sistematizadas para a realização dessa atividade. Estudos têm mostrado que o maior problema permanece na dificuldade de lidar com oráculos de teste, que é considerado um problema de pesquisa em aberto.

Em geral, as atividades de teste para o domínio RV são realizadas manualmente e em sua maioria conduzidas somente após o final da fase de desenvolvimento. Tais atividades geralmente são desenvolvidas com base na verificação de requisitos de teste (funcionais e não funcionais) que devem ser garantidos antes da entrega do produto (CORREA; NUNES; DELAMARO, 2018). A escassez de estudos que avaliem o custo do desenvolvimento de novas abordagens de teste, ou que avaliem a eficácia da utilização de técnicas existentes contribui ainda mais para impactar e agravar a popularização do cenário de teste para aplicações no domínio de RV.

Neste capítulo é feita uma avaliação a respeito do quanto a falta da atividade de teste de software pode impactar negativamente o desenvolvimento de aplicações de RV. Para isso, foram analisados projetos de código aberto de forma a categorizá-los com relação à tendência de existência de falhas, que poderiam ser mitigadas pela adoção de práticas de teste de software.

Dessa forma, esse capítulo está organizado da forma: a seção 4.1 apresenta um estudo exploratório para avaliar o quanto a falta de atividades de teste podem ser prejudiciais a projetos

de código aberto; a seção 4.2 discute os resultados do experimento apresentado; a seção 4.3 aponta algumas limitações relacionadas ao estudo apresentado neste capítulo e finalmente as conclusões, direcionamentos e considerações finais são apresentadas na seção 4.4.

## 4.1 Avaliação Experimental

Com a popularização do desenvolvimento de aplicações de RV, é interessante observar, sob o ponto de vista da engenharia de software, como é conduzido o processo de desenvolvimento desse tipo de aplicações. Especialmente a respeito de práticas de teste de software, possibilitando, com isso, investigar como possíveis problemas ocorridos durante a etapa de desenvolvimento podem impactar na qualidade final do produto.

Uma das abordagens mais utilizadas para quantificar aspectos de qualidade em projetos de software é a avaliação de métricas de código-fonte (NUÑEZ-VARELA *et al.*, 2017). As métricas de código-fonte são componentes significativos para o processo de medição de software e são comumente usadas para medir o quanto um software está sujeito a falhas, e para melhorar a qualidade do próprio código-fonte (PALOMBA *et al.*, 2014).

Outro fator que pode ser explorado para avaliar a qualidade do código é a identificação de *code smells* de software, já que alguns estudos mostram que existe uma forte correlação entre *code smells* e predisposição a falhas (KHOMH *et al.*, 2012). Nesse sentido, esses são dois aspectos que são levados em consideração na avaliação realizada para investigar os aspectos de qualidade do código no contexto de teste de software.

Ghrairi *et al.* (2018) realizaram um estudo exploratório no *Github* e *Stack Overflow* para investigar quais são as linguagens e *game engines* mais populares utilizadas em projetos de RV. De acordo com os resultados, a linguagem mais popular para o desenvolvimento de RV é *C#*, e *Unity* é a *game engine* mais utilizada durante o desenvolvimento de aplicações de RV, dessa forma, as análises realizadas nesse estudo levam em consideração tais características.

### 4.1.1 Visão geral do estudo

Foram formuladas as seguintes questões de pesquisa com relação ao objetivo da análise de qualidade de projetos de RV:

- **QP$_1$** : *"Qual o estado da prática de testes em aplicações de RV de código aberto?"* Essa questão de pesquisa visa buscar entender como as práticas de teste estão sendo aplicadas em projetos de RV de código aberto.

- **QP$_2$** : *"Como está distribuída a ocorrência de code smells em projetos de RV?"* Essa questão de pesquisa visa investigar a distribuição de *code smells* com objetivo de identificar

se há um conjunto específico de *code smells* que ocorre com uma maior frequência em aplicações de RV.

- **QP$_3$** : *"É possível traçar uma relação entre métricas de código e predisposição a falhas?"* Na literatura existem vários indícios de que existe uma forte correlação entre métricas de código e predisposição a falhas (TAHIR; MACDONELL, 2012; NUÑEZ-VARELA *et al.*, 2017), indicando que se um conjunto de métricas de código atinge um limite predefinido, é muito provável que esse seja um indicativo de que o projeto também possa ter algum tipo de falhas. Essa questão de pesquisa investiga essa hipótese utilizando uma abordagem de predição de defeitos com algoritmos de aprendizado de máquina não supervisionados.

### 4.1.2 Design do experimento

O processo de seleção de projetos de código aberto consistiu em uma busca sistematizada, no repositório de projetos de código abertos *Github*, utilizando as palavras-chave "*virtual reality*"e "VR". Com o objetivo de traçar um perfil mais específico, a busca concentrou-se apenas em projetos desenvolvidos para a plataforma *Unity*, uma vez que esta tecnologia tem se destacado como a plataforma de desenvolvimento de RV mais popular (GHRAIRI *et al.*, 2018).

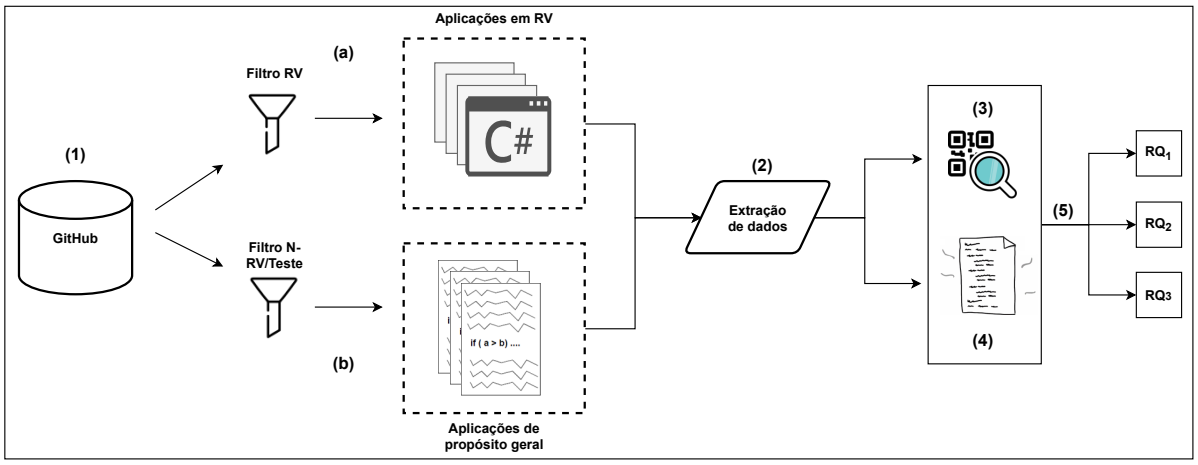
O principal objetivo do estudo foi explorar projetos de realidade virtual sob a perspectiva de projetos de código-fonte aberto. Para isso, foram catalogados e analisados um total de 151 projetos de código aberto, disponíveis no *Github*. Alguns dos projetos não puderam ser analisados devido à ausência de dependências externas ou de mecanismos que auxiliem na construção/execução dos mesmos (arquivos de configuração de projetos C# ausentes), com isso, foram analisados um total de 119 projetos.

A fim de responder os questionamentos relacionados às questões de pesquisa $QP_2$ e $QP_3$, também foram catalogados projetos de código aberto de propósito geral (identificados neste capítulo como N-RV), que possuem características semelhantes (mesma linguagem de programação C#). O objetivo foi buscar comparar as informações observadas nos projetos de RV com projetos N-RV. Nesse sentido, catalogamos um total de 177 projetos N-RV. Após uma análise de processo individual, removemos projetos duplicados e projetos que tinham dependências externas ausentes ou mecanismo de importação personalizados. No final, alcançamos um total de 107 projetos N-RV capazes de serem utilizados no experimento.

Como o objetivo do estudo foi analisar os impactos que a falta de prática de teste de software pode acarretar em projetos de RV, comparados em relação a projetos N-RV, a avaliação experimental considerou apenas dados relacionados às classes dos projetos N-RV que foram devidamente testadas.

O procedimento realizado durante a execução do experimento está descrito na Figura 7 e consistiu nas seguintes etapas:

Figura 7 – Passos adotados para a realização do estudo




Fonte: Elaborada pelo autor.

1. pesquisar na plataforma *Github* por projetos com as características desejadas;

   a) projetos de RV desenvolvidos na plataforma Unity;

   b) projetos de propósito geral que foram desenvolvidos utilizando a linguagem de programação C# e possuem práticas de teste de software em sua composição;

2. extração de dados relacionados à prática de teste em projetos de aplicações de RV;

3. extração de métricas de código e informações de *code smells* para todos os projetos;

4. uso de métricas de código para calcular a predisposição a falhas em todos os projetos;

5. sumarização dos resultados e resposta às questões de pesquisa.

### 4.1.3 Visão geral dos projetos

A Tabela 1 resume as informações sobre os artefatos utilizados nos experimentos conduzidos. Os dados relativos às principais características dos repositórios são apresentados na forma de média, classificando os repositórios em projetos pequenos (com até 80 classes), médios (projetos que contêm entre 81 e 200 classes) e projetos grandes, que contém um número de classes maior que 200.

A fim de fornecer uma visão do escopo geral dos artefatos usados, a Tabela 2 apresenta informações sobre as características gerais de todos os projetos.

Com o objetivo de tentar estimar os métricas de qualidade de código, utilizadas para fornecer uma base quantitativa e ajudar a estimar a complexidade do código dos projetos, foram calculadas métricas de projetos orientados a objetos (CHIDAMBER; KEMERER, 1994), resumidas na Tabela 3.

Tabela 1 – Características dos projetos utilizados no experimento

| Características | Pequenos ( 1 ∼ 80 classes) | | Médios (81 ∼ 200 classes) | | Grandes ( 201+ classes) | |
|---|---|---|---|---|---|---|
| | **RV** | **N-RV** | **RV** | **N-RV** | **RV** | **N-RV** |
| Nº C# Classes | 31,5 | 30,2 | 121,7 | 126,4 | 374,8 | 4320 |
| Nº *Branches* | 1,45 | 1,7 | 1,7 | 2,5 | 2,56 | 11,4 |
| Nº *Commits* | 78,4 | 45,5 | 48,8 | 245 | 66,2 | 8931 |
| Nº *Contributors* | 2,6 | 1,4 | 2,5 | 2,2 | 2,9 | 47,1 |
| Nº *Forks* | 34,7 | 1,9 | 10,9 | 1 | 76,6 | 82,4 |
| Nº *Subscribers* | 15,8 | 2 | 8,1 | 2,5 | 22,3 | 176,8 |
| Nº *Stars* | 111,6 | 4 | 23,8 | 0,7 | 187,8 | 89,4 |

Tabela 2 – Características gerais dos projetos analisados

| **Atributos** | **RV** | **N-RV** |
|---|---|---|
| Projetos | 119 | 107 |
| Número de classes testadas | 63 | 4 186 |
| Número de classes (total) | 21 508 | 21 563 |
| Linhas de código (somente C#) | 2 314 522 | 2 455 766 |

Tabela 3 – Métricas dos projetos analisados

| **Métrica** | **RV** | **Média** | **N-RV** | **Média** |
|---|---|---|---|---|
| Número de filhos | 3,81 | 0,17 | 716 | 0,17 |
| Número de atributos | 102,78 | 4,77 | 7,062 | 1,68 |
| Número de métodos | 107,51 | 4,99 | 14,374 | 3,43 |
| Número de propriedades | 24,39 | 1,13 | 67 | 0,01 |
| Número de atributos públicos | 49,79 | 2,31 | 1,368 | 0,32 |
| Profundidade da árvore de herança | 4,07 | 0,18 | 1,278 | 0,30 |
| Número de métodos públicos | 65,62 | 3,05 | 9,582 | 2,28 |
| Falta de Coesão de Métodos | 34,19 | 1,58 | 7,958 | 1,90 |
| Método ponderado por classe | 190,65 | 8,86 | 21,959 | 5,24 |

Todas as informações sobre os projetos, detalhes de implementação, dados utilizados para criação das tabelas e gráficos, bem como para execução do experimento e discussão dos resultados apresentados estão disponíveis, publicamente, no repositório do experimento (ANDRADE; NUNES; DELAMARO, 2021).

## 4.2 Resultados e Discussão

Nesta seção são abordadas as questões de pesquisa definidas para o estudo e são discutidos os resultados obtidos por meio das análises e observações dos resultados do estudo empírico.

### 4.2.1 Como aplicações de RV são testadas?

Com relação à primeira questão de pesquisa (**QP$_1$**) do estudo, que visava compreender *"Qual o estado da prática de testes em aplicações de RV de código aberto?"*, Os 119 projetos de aplicações em RV foram manualmente avaliados e observou-se que apenas 6 projetos

(*Bowlmaster - 53 testes, CameraControls - 60 testes, GraduationGame - 15 testes, MiRepository_VRPAD - 11 testes, space_concept - 11 testes, UnityBenchmarkSamples - 4 testes*) adotavam práticas de teste de software, somando um total de 154 casos de teste de unidade, para avaliar as funcionalidades dos projetos.

A *Unity* utiliza um *fork* próprio do *Mono* (The Mono - Unity fork, 2021), plataforma de código aberto baseada no *.NET Framework* que permite que os desenvolvedores criem aplicações em C#. Apesar da existência de testes de unidade nos projetos analisados, não foi possível calcular as informações sobre os critérios de teste, como cobertura de código, uma vez que *Unity*, até o momento, não fornece uma solução nativa para cálculo de cobertura de código.

Com base nas informações coletadas, observou-se que dos 119 projetos analisados, apenas 6 (5,04%) possuíam alguma prática de teste de software, e, dentre os projetos que possuíam casos de teste, não apresentaram indícios de que os testes existentes fossem capazes de garantir que as principais funcionalidades das aplicações eram devidamente cobertas pelos testes.

Outro aspecto interessante observado na análise dos 6 projetos que possuem casos de teste é o fato do projeto *Bowlmaster* fazer parte de um curso popular, com mais de 334.000 alunos inscritos [1], que visa ensinar práticas de desenvolvimento de aplicativos de RV. Isso indica que existe uma preocupação por parte dos educadores quanto à conscientização de que a atividade de teste é um fator importante no processo de desenvolvimento de aplicações de RV e, portanto. espera-se que os alunos sejam capazes de aplicar os conceitos relacionados à prática de teste de software em seus projetos futuros.

Após a análise dos projetos, com relação à $QP_1$, chegou-se à conclusão de que ainda não há consenso quanto à aplicação de práticas de teste de software para aplicações de RV. Essa percepção serviu como motivação para investigar as próximas questões de pesquisa. Os resultados observados para a $QP_1$ estão de acordo com os trabalhos recentes da literatura. Karre, Mathur e Reddy (2019) conduziram um estudo empírico com profissionais de RV para entender as práticas atuais e os desafios enfrentados na indústria. Os resultados relacionados a testes de software apontam para a ausência de ferramentas adequadas, bem como incertezas sobre como testar aplicações de RV. Como consequência, essa falta de metodologias de avaliação e ferramentas de teste tende a causar um impacto no ciclo de desenvolvimento e consequentemente no tempo gasto para lançar produtos de RV.

### 4.2.2 Distribuição de code smells

Observando a falta de prática de teste de software nos projetos de RV, decidiu-se investigar como a ausência dessa prática se reflete nos projetos. Para isso, foi feito um levantamento a respeito da incidência de *code smells* (FOWLER, 2018) dentro dos projetos investigados. Essa

[1] <https://github.com/CompleteUnityDeveloper/08-Bowlmaster-Original>

avaliação levou a investigação da segunda questão de pesquisa (**QP$_2$**) *"Como está distribuída a ocorrência de code smells em projetos de RV?"*.

Em geral, a presença de *code smells* em projetos de software indica a presença de problemas de qualidade. Por exemplo, um dos *code smells* mais conhecidos, *God Class*, é definido em classes que devido ao seu tamanho e complexidade, passam a monopolizar grande parte da inteligência de um sistema. *God Class* é, portanto, um forte indicador de possíveis problemas, pois este componente está agregando funcionalidades que deveriam ser distribuídas entre vários outros componentes, aumentando assim o risco de falhas de software (HALL *et al.*, 2014). Esses problemas afetam diretamente características como a capacidade de manutenção e contribuem para dificultar a evolução do software.

Para realizar a avaliação discutida nesta subseção, foi observada a premissa de que projetos de software que não possuem casos de teste em sua composição tendem a ter uma qualidade de código inferior em comparação com projetos que foram testados (FAROOQ; QUADRI; AHMAD, 2011). Essa premissa sustenta que, ao ignorar atividades relacionadas à qualidade de software, os desenvolvedores tendem a não observar aspectos que poderiam desencadear comportamentos inesperados na aplicação.

Para possibilitar uma análise comparativa, os resultados obtidos nos projetos de RV foram comparados com resultados obtidos em aplicações N-RV que possuíam artefatos de casos de teste em seus projetos. O estudo relacionado à **QP$_2$** explora três tipos diferentes de *code smells* nos projetos:

- ***Architecture smells***: focam na identificação de pontos de interesse para possíveis problemas estruturais que podem contribuir negativamente e dificultar atividades como depuração e refatoração, além de aumentar o custo para correção de falhas devido a características de aumento a complexidade do software, quando presente (MO *et al.*, 2015);

- ***Implementation smells***: foram introduzidos pela primeira vez por Fowler (2018) e buscam estabelecer um conceito para classificar deficiências nos princípios de design orientado a objetos. Esta classe de *code smells* cobre princípios como abstração de dados, encapsulamento, modularidade, hierarquia, etc;

- ***Design smells***: são tipos específicos de estruturas que podem indicar violação de um princípio fundamental, podendo impactar aspectos de qualidade do projeto (SURYANARAYANA; SAMARTHYAM; SHARMA, 2014).

Para calcular a distribuição dos *code smells*, foi utilizada a ferramenta *Designite* (SHARMA; MISHRA; TIWARI, 2016). Os *code smells* foram classificados de acordo com o número de ocorrências nas classes analisadas e distribuição percentual. Os dados são apresentados nas Tabelas 4, 5 e 6.

Vale ressaltar que a análise não levou em consideração classes que o código fonte era referente à implementação de casos de teste. Uma vez que o objetivo era medir os aspectos de qualidade referentes a trechos de código de classes que continham regras de negócio das aplicações. Além disso, *code smells* em códigos de teste de software requerem uma abordagem de classificação completamente diferente (TUFANO *et al.*, 2016).

#### 4.2.2.1 Resultados para Architecture smells

É possível observar na Tabela 4 que, entre os projetos de RV, há uma baixa incidência de *architecture smells*, com apenas três tipos (ACD, AFC e AGC) apresentando um percentual de ocorrência entre 0,93% e 1,70%. Observando os projetos N-RV, pode-se observar que essa categoria de *smells* teve uma incidência menor em comparação aos projetos em RV. Os *smells* AUD, AGC e AFC apresentaram as maiores taxas de ocorrência, com percentuais entre 0,26% e 0,57%.

O comportamento apresentado nos projetos de RV pode ser justificado pelo fato de que dentro da plataforma *Unity*, embora seja utilizada uma linguagem orientada a objetos (C#), o modelo de desenvolvimento é considerado uma abordagem de programação baseada em componentes, no qual cada elemento da cena têm, se necessário, um *script* que controla o seu comportamento. Essa abordagem tem como principal característica focar na separação de interesses na implementação dos componentes, fazendo com que quando há a necessidade de interação entre os componentes exista uma sobrecarga de dependências.

Tabela 4 – Descrição e distribuição de *architecture smells* para projetos RV e N-RV

| **ID** | **Smell** | **RV** | | **N-RV** | |
|---|---|---|---|---|---|
| AAI | *Ambiguous Interface* | 29 | 0,13% | 1 | 0,02% |
| ACD | *Cyclic Dependency* | 212 | 0,99% | 6 | 0,14% |
| ADS | *Dense Structure* | 3 | 0,01% | 2 | 0,05% |
| AFC | *Feature Concentration* | 366 | 1,70% | 24 | 0,57% |
| AGC | *God Component* | 201 | 0,93% | 12 | 0,29% |
| ASF | *Scattered Functionality* | 84 | 0,39% | 8 | 0,19% |
| AUD | *Unstable Dependency* | 78 | 0,36% | 11 | 0,26% |
| Desvio Padrão | | 118,34 | | 7,17 | |
| Média | | 139,0 | | 9,14 | |
| Mediana | | 84,00 | | 8,00 | |

Apesar de aplicações N-RV apresentarem taxas mais baixas de *architecture smells*, elas apresentam, principalmente, uma maior incidência do *smell* AFC. Esse *smell* ocorre quando um componente é responsável por mais de um recurso/funcionalidade arquitetural. Isso pode ser explicado devido ao modelo de programação adotado. Uma grande parte dos projetos N-RV corresponde a aplicações web, que normalmente utilizam um padrão arquitetural estilo *Model-View-Controller* (MVC). Conforme apontado por Aniche *et al.* (2018), sistemas que adotam tal modelo arquitetural podem ser impactados por tipos de práticas inadequadas que levam ao aparecimento de *smells*.

Do ponto de vista de teste de software, uma baixa incidência de *architecture smells* pode ser considerado um fator decisivo de sucesso, uma vez que a baixa dependência entre os módulos é uma característica que facilita a aplicação de testes de unidade (ANICHE; OLIVA; GEROSA, 2013). Em geral, quando é necessário comunicar-se com outras unidades de código, boas práticas de programação apontam para a utilização de objetos *stubs* ou *mock* para representar esta comunicação. Um grande benefício dessa abordagem é que, ao reduzir o acoplamento do sistema, é possível reproduzir cenários complexos de teste com maior facilidade.

Apesar de apresentar vantagens em relação ao baixo acoplamento arquitetural e portanto facilitar a adoção de testes de unidade, uma abordagem de programação orientada a componentes pode apresentar riscos para o contexto de uma fase de testes de integração. Os principais riscos estão relacionados, principalmente, ao uso excessivo de componentes produzidos por terceiros, pois, em geral, tais componentes funcionam como uma caixa preta na qual é necessário confiar em seu comportamento. Um exemplo da utilização desse modelo é a loja oficial de *assets* disponível na plataforma *Unity* [2].

Para melhor compreender os resultados apresentados, relativamente a cada uma das classes de *smells* analisadas, verificamos se existe, de fato, diferença estatística entre a presença de *smells* entre grupos de classes para aplicações de RV e aplicações N-RV. Uma vez que o experimento observou um baixo número de *smells* para cada categoria (*achitecture*, *design* e *implementation*), e como não há garantias de que os dados coletados partem de uma distribuição normal, foi aplicado o teste de Mann-Whitney (FAY; PROSCHAN, 2010) para verificar se há diferença estatística entre a presença de *smells* para cada uma das categorias avaliadas.

A hipótese nula ($H_0$) do teste de Mann-Whitney indica que *"A distribuição da variável em questão, avaliada, é idêntica (na população) para dois grupos"*, ou seja, para o contexto do estudo, indica que não há diferença na presença de *smells* entre os trechos de código avaliados em projetos RV e projetos N-RV. A hipótese alternativa ($H_1$) indica que *"As distribuições nos dois grupos não é a mesma"*, portanto, há uma diferença estatística entre a incidência de *smells* para os projetos com aplicações de RV em relação ao projetos N-RV.

Considerando o valor de alfa = 0,05, que compreende o complemento da margem de um nível de confiança de 95%, para *architecture smells*, a hipotese nula $H_0$ é rejeitada com um *p-value* = 0,00760. Esse resultado indica que, de fato, existe uma diferença estatística entre a presença de *smells* ao comparar os projetos RV e os projetos N-RV.

Utilizando uma análise descritiva, obtida por meio da análise do número de ocorrências de cada tipo de *smells* descritos na Tabela 4, é possível observar que projetos RV, que representam trechos de codigo de classes que não foram testadas tendem a apresentar um maior índice de *architecture smells* em relação aos projetos N-RV, que representam trechos de código de classes testadas.

[2] <https://assetstore.unity.com/>

#### 4.2.2.2 Resultados para Implementarion smells

Ao observar a Tabela 5 que, diferentemente dos resultados apresentados na análise de *architecture smells*, é possível identificar um alto índice de ocorrencia de *implementarion smells* nos projetos de RV. Destaca-se os *smells* ILI, ILS e IMN, que tiveram um percentual de ocorrência de 31,81%, 55,55% e 117,46%, respectivamente.

Por exemplo, embora não represente um risco direto para o código-fonte produzido, o *smell* ILI pode ser um indicador de que algo pode ser revisado/refatorado. Esse *smell* indica a existência de identificadores muito longos e pode ser um indício de que há necessidade de muito texto para distinguir/identificar variáveis e, em alguns casos, isso pode indicar que o desenvolvedor pode não estar usando a estrutura de dados mais adequada para representá-lo.

Tabela 5 – Descrição e distribuição de *implementation smells* para projetos RV e N-RV

| **ID** | **Smell** | **RV** | | **N-RV** | |
|---|---|---|---|---|---|
| ICM | *Complex Method* | 1812 | 8,42% | 9 | 0,22% |
| ICC | *Complex Conditional* | 684 | 3,18% | 14 | 0,33% |
| IDC | *Duplicate Code* | 9 | 0,04% | 1 | 0,02% |
| IECB | *Empty Catch Block* | 150 | 0,70% | 5 | 0,12% |
| ILM | *Long Method* | 583 | 2,71% | 9 | 0,22% |
| ILPL | *Long Parameter List* | 2117 | 9,84% | 13 | 0,31% |
| ILI | *Long Identifier* | 6841 | 31,81% | 12 | 0,29% |
| ILS | *Long Statement* | 11947 | 55,55% | 40 | 0,96% |
| IMN | *Magic Number* | 25264 | 117,46% | 36 | 0,86% |
| IMD | *Missing Default* | 931 | 4,33% | 17 | 0,65% |
| IVMCC | *Virtual M. C. C.*** | 35 | 0,16% | 5 | 0,12% |
| Desvio Padrão | | 7425,91 | | 11,87 | |
| Média | | 4579,36 | | 14,63 | |
| Mediana | | 931,00 | | 12,00 | |

***Virtual Method Call from Constructor*

O *smell* ILS ocorre quando existem instruções excessivamente longas. Declarações longas tendem a dificultar o gerenciamento do código e, consequentemente, são vilões se observados sob o ponto de vista da prática de teste de software. Trechos de código muito longos tendem a ser mais difíceis de testar porque geralmente se tornam muito complexos quando comparados a trechos menores que são gerenciados com mais eficiência.

Finalmente, o *smell* IMN ocorre quando um número inexplicável é usado em uma expressão. Em geral, *magic numbers* são valores únicos que possuem algum significado simbólico. As boas práticas de programação indicam que, nesses casos, tais números devem ser declarados como constantes para facilitar a leitura do código-fonte, bem como para padronizar seu uso durante todo o código.

A ocorrência do *smell* IMN, normalmente, está associada à falta de diretrizes para padronização de código, bem como à falta de práticas de refatoração de código. Normalmente os números têm um significado, portanto, recomenda-se que sejam atribuídas variáveis para tornar o

código mais legível e autoexplicativo. De forma complementar, o nome de variáveis deve refletir pelo menos o que a variável significa, não necessariamente seu valor.

Projetos N-RV novamente apresentaram menor taxa de ocorrência. Os *smells* mais frequentes foram ILS, IMN e IMD que alcançaram, respectivamente, percentagens de 0,96%, 0,85% e 0,65%.

Do ponto de vista do teste de software, optar pelo uso de constantes em vez de números mágicos pode garantir que, uma vez que o valor da constante tenha sido propriamente testado, não haja risco de que o valor da constante seja declarado erroneamente em outros pontos do código fonte no futuro.

Também foi aplicado o teste de Mann-Whitney para verificar se há diferença estatística entre a presença de *implementation smells* para os projetos RV em relação aos projetos N-RV. Adotando um intervalo de confiança de 95%, o teste apresentou o *p-value* = 0,00040, valor suficiente para rejeitar a hipótese nula do teste e reforçar as informações extraídas a partir dos dados apresentados na Tabela 5, indicando que projetos N-RV, que possuem código fonte devidamente testado, apresentam uma menor taxa de *implementation smells* em relação aos projetos de aplicações de RV, que não possuíam práticas de teste de software.

#### 4.2.2.3 Resultados para Design smells

Por fim, foram avaliados *design smells*, que buscam identificar violações dos princípios de *design*. Ao observar a Tabela 6 é possível concluir que esta classe de *smells* foi a que apresentou maior grau de incidência nos projetos de RV. Os *smells* DUA, DTA e DDE foram os que apresentaram maior porcentagem de ocorrência com 17,39%, 32,49% e 37,67%, respectivamente.

O *smell* DUA lida com a prática de abstrações desnecessárias e é identificado quando uma abstração tem mais de uma responsabilidade atribuída a ela. Este *smell* tende a ocorrer quando há uma utilização de padrões de desenvolvimento de programação procedural no contexto de linguagens de programação orientadas a objetos (SURYANARAYANA; SAMARTHYAM; SHARMA, 2014).

Do ponto de vista de aplicações de RV, que adotam a programação baseada em componentes, o aparecimento desse *smell* pode ser explicado pelo fato de que tal abordagem de desenvolvimento consiste na concentração de *scripts* de código atrelados a objetos na cena, que funcionam quase independentemente, não exigindo que tais objetos estejam familiarizados com seu funcionamento interno de objetos terceiros para uma interação dos componentes, uma vez que a mesma é feita por meio de eventos.

Abstrações desnecessárias de design aumentam a complexidade do código desnecessariamente e afetam a compreensão do design geral. Do ponto de vista de teste de software, essa prática tende a dificultar a criação de casos de teste que envolvam a interação de diferentes componentes.

Tabela 6 – Descrição e distribuição de *design smells* para projetos RV e N-RV

| ID | Smell | RV | | N-RV | |
|---|---|---|---|---|---|
| DBH | *Broken Hierarchy* | 245 | 1,14% | 8 | 0,19% |
| DBM | *Broken Modularization* | 991 | 4,61% | 46 | 1,10% |
| DCM | *Cyclically-dependent M.* | 3149 | 14,64% | 45 | 1,08% |
| DCH | *Cyclic Hierarchy* | 6 | 0,03% | 4 | 0,10% |
| DDH | *Deep Hierarchy* | 0 | 0,00% | 1 | 0,02% |
| DDE | *Deficient Encapsulation* | 8101 | 37,67% | 13 | 0,31% |
| DDA | *Duplicate Abstraction* | 2469 | 11,48% | 11 | 0,26% |
| DHM | *Hub-like Modularization* | 4 | 0,02% | 16 | 0,38% |
| DIA | *Imperative Abstraction* | 627 | 2,92% | 9 | 0,22% |
| DIM | *Insufficient Modularization* | 1171 | 5,44% | 44 | 1,05% |
| DMH | *Missing Hierarchy* | 18 | 0,08% | 2 | 0,05% |
| DMA | *Multifaceted Abstraction* | 209 | 0,97% | 2 | 0,05% |
| DMH | *Multipath Hierarchy* | 1 | 0,00% | 2 | 0,05% |
| DRH | *Rebellious Hierarchy* | 389 | 1,81% | 9 | 0,22% |
| DUE | *Unexploited Encapsulation* | 15 | 0,07% | 2 | 0,05% |
| DUH | *Unfactored Hierarchy* | 483 | 2,25% | 7 | 0,17% |
| DUA | *Unnecessary Abstraction* | 3741 | 17,39% | 17 | 0,41% |
| DTA | *Unutilized Abstraction* | 6987 | 32,49% | 23 | 0,55% |
| DWH | *Wide Hierarchy* | 64 | 0,30% | 5 | 0,12% |
| Desvio Padrão | | 2344,41 | | 14,59 | |
| Média | | 1508,94 | | 14,00 | |
| Mediana | | 389,00 | | 9,00 | |

O *smell* DTA ocorre quando uma abstração não é utilizada, não está sendo utilizada diretamente, ou porque não pode ser acessada no código-fonte. Esse *smell* está relacionado ao DUA, uma vez que abstrações desnecessárias tendem a não ser utilizadas. Outro fator de impacto para o aparecimento desse *smell* está ligado a possíveis atividades de manutenção/refatoração de código, que tendem a deixar pedaços de código que não são mais necessários.

Do ponto de vista de teste de software, a existência de um conjunto de teste que possa ser utilizada como teste de regressão tende a facilitar a localização de trechos de código-fonte que não são mais necessários, fazendo com que a ocorrência desse *smell* seja reduzida. Do ponto de vista de um testador, se houver um código que não está sendo utilizado no projeto, ele não precisa ser testado. Portanto, a identificação desses fragmentos de código pode levar a atividades de teste mais eficientes.

Por fim, o *smell* DDE, que identifica casos de encapsulamento deficiente, foi aquele que apresentou a maior taxa de ocorrência nessa classe de *smells*. Esse *smell* ocorre quando a declaração de visibilidade de um atributo de uma classe é mais permissiva do que o necessário. Por exemplo, quando os atributos de uma classe são declarados desnecessariamente como públicos.

Do ponto de vista de teste de software, a separação de interesses permite que os detalhes da implementação sejam ocultados. Se uma abstração expõe detalhes de implementação desnecessariamente, isso leva a um acoplamento indesejável no código-fonte. Isso terá um impacto

negativo na atividade de teste, porque verificar as unidades que têm um alto grau de acoplamento torna-se uma tarefa mais desafiadora devido à necessidade de *mocks* e *stubs* mais complexos. Da mesma forma, o alto grau de acoplamento faz com que alterações que são feitas em um trecho de código causem um grande reflexo em várias partes da aplicação, podendo contribuir para que testes projetados anteriormente venham a falhar, se eles não forem projetados de forma adequada.

As aplicações N-RV tiveram menor ocorrência nesta categoria de *smells*, sendo DBM e DCM os dois tipos que apresentaram maior ocorrência, com 1,10% e 1,08% respectivamente. As explicações para a taxa de ocorrência do *smell* DCM estão relacionadas a problemas de dependência cíclica, causados majoritariamente devido a utilização do modelo MVC para aplicações web, enquanto o *smell* DBM surge quando dados e/ou métodos que idealmente deveriam ter sido restritos a uma única abstração são separados e espalhados por várias abstrações durante o código.

Por fim, mais uma vez aplicamos o teste de Mann-Whitney para verificar se os dados obtidos em nossa avaliação experimental podem traçar um quadro real sobre o comportamento da classe que não possui casos de testes (projetos RV) em comparação com as classes que foram devidamente testadas (projetos N-RV). Mais uma vez o teste de Mann-Whitney apontou, com *p-value* = 0,00089, que as classes de código fonte que foram devidamente testadas (projetos N-RV) tendem a apresentar menores taxas de *code smells* quando comparadas às classes não testadas (projetos RV).

A segunda questão de pesquisa investigada neste capítulo ($QP_2$) visava entender *"Como está distribuída a ocorrência de code smells em projetos de RV?"*. Para abordá-la foram investigadas as principais classes de *code smells* para aplicações de RV e os resultados foram comparados com aplicações N-RV, que possuíam histórico de práticas de teste de software. Por meio dos resultados foi possível observar que no contexto de aplicações de RV há uma maior incidência de *code smells* dos tipos *implementation* e *design*, tal resultado se justifica uma vez que essas duas categorias de *smells* se repetem frequentemente devido às características específicas existentes no contexto do desenvolvimento de aplicações de RV.

Por meio da avaliação realizada também foi possível apresentar uma discussão sobre como as práticas de teste de software podem ser facilitadas ao evitar os *smells* que obtiveram a maior taxa de de incidência. Por fim, os resultados dos testes estatísticos realizados para comparar projetos RV e N-RV, indicam que a prática de teste de software pode contribuir para aumentar os critérios de qualidade do código fonte produzido e, por consequência, reduzir a presença de *code smells*.

De acordo com Hall *et al.* (2014), a presença de *code smells* pode ser um indicativo da presença de falhas em um código fonte de uma aplicação, além de contribuir para dificultar a manutenção e a evolução do código-fonte em projetos maiores. Isso leva à última questão de pesquisa deste estudo ($QP_3$), que visa investigar as aplicações de RV com relação a predisposição

a falhas.

### 4.2.3 Analisando os projetos com relação a predisposição a falhas

Conforme mencionado na subseção anterior, a presença de *code smells* pode indicar a ausência de atributos de qualidade no código-fonte e isso pode ser um indício da presença de falhas em um software (HALL *et al.*, 2014). Da mesma forma, conforme mencionado anteriormente, a maior taxa de ocorrência de *code smells* nos projetos pode dificultar a prática de teste de software. Para compreender os riscos de negligenciar a atividade de teste, foi realizada uma análise nos projetos de RV de código-aberto com relação à tendência dos mesmos com relação à presença de falhas.

Visto que uma das estratégias para identificar *code smells* consiste em analisar regras e/ou limites definidos a partir de métricas de código (KHOMH; PENTA; GUEHENEUC, 2009), portanto, essa etapa do estudo visou investigar a seguinte questão de pesquisa ($QP_3$): *"É possível traçar uma relação entre métricas de código e predisposição a falhas?"*. Para fazer isso, foi utilizado um conjunto de métricas de código descritas na Tabela 3, em conjunto com uma abordagem para detecção de predição de falhas, que usa as métricas de código como um indicador para sugerir se um determinado trecho de código-fonte tem tendências a possuir falhas ou não.

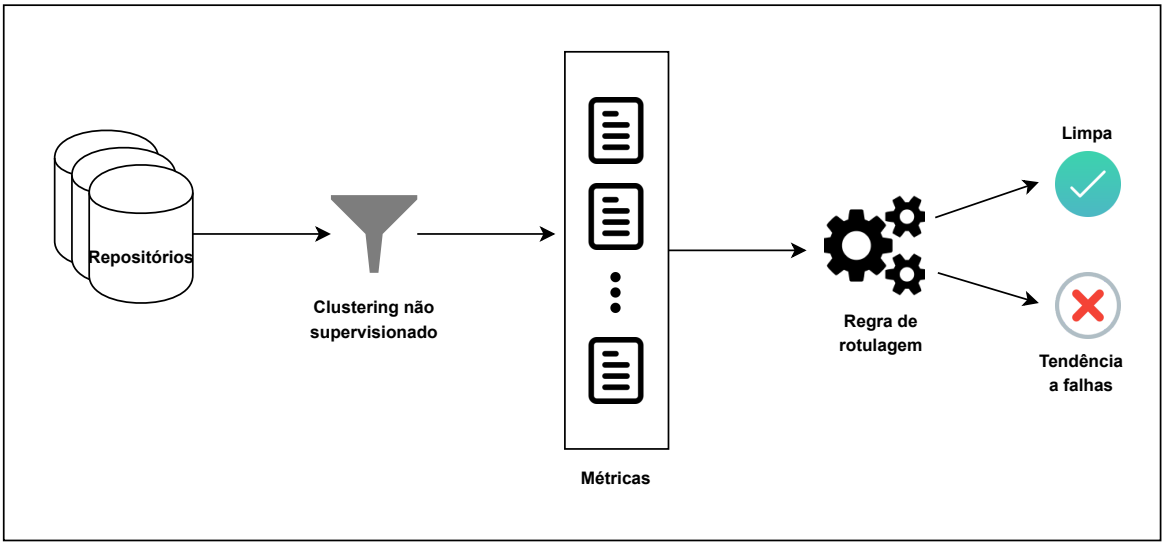
Ao explorar a relação entre as métricas de software e a tendência à existência de falhas, busca-se justificar a necessidade de atividades de teste de software. Por exemplo, um valor alto em uma determinada métrica de código pode levantar indícios, com alta probabilidade, sobre a confiabilidade de algumas partes do código fonte. Um ponto importante apontado por Garousi e Mäntylä (2016) indica que a ausência de tecnologias de teste robustas e padronizadas são fatores que impactam negativamente no processo de desenvolvimento de software e consequentemente na adoção de práticas de teste, o que contribui para o aumento da incidência de falhas evitáveis que tendem a ser identificadas somente no estágio que as aplicações passam a ser utilizadas por seus usuários finais.

A eficácia das técnicas de predição de falhas é frequentemente apresentada utilizando dados históricos do repositório de código-fonte (HERBOLD, 2017). No entanto, uma vez que um modelo é construído com essa abordagem, ainda não está claro como tal modelo se comporta quando utilizado em projetos que não correspondem às características (linguagem, plataforma, domínio) utilizadas durante o processo de treinamento do modelo (HE *et al.*, 2015).

Uma vez que há disponível publicamente um conjunto de dados ou um histórico de rastreamento de *bugs* mantido com dados de aplicações de RV, esse estudo buscou explorar uma abordagem que utiliza uma técnica de predição de falha não supervisionada (YANG; QIAN, 2016).

Foi utilizada uma abordagem denominada de *Average Clustering and Labeling* (ACL)

Figura 8 – Processo geral da abordagem ACL para predição de falhas




Fonte: Elaborada pelo autor.

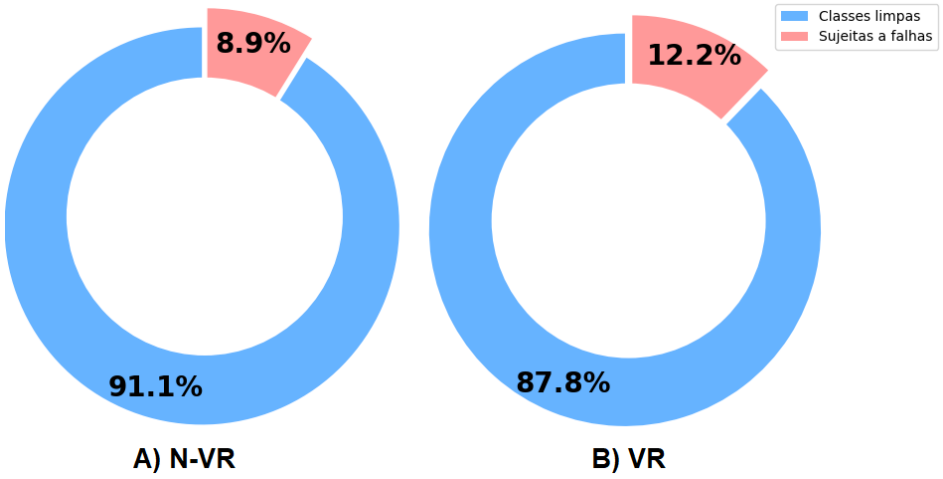
(YANG; QIAN, 2016) para buscar predizer a predisposição a falhas em conjuntos de dados não rotulados. Estudos experimentais (YANG; QIAN, 2016; YANG; YANG; QIAN, 2018) indicam que modelos construídos utilizando a abordagem ACL obtiveram um bom desempenho de predição e são comparáveis aos modelos típicos de aprendizado supervisionado em termos de métricas de avaliação como precisão e *recall*, oferecendo, portanto, uma escolha viável para investigar a predição de falhas em código-fonte quando não há dados históricos com características semelhantes às que deseja-se avaliar.

Além de ajudar a responder a questão de pesquisa ($QP_3$) apontada anteriormente, este estudo pode ajudar desenvolvedores de software a compreender as características de aplicações de RV e as implicações potenciais de negligenciar as atividades de teste de software, além de aumentar a conscientização a respeito do desenvolvimento de práticas de atividades relacionadas a verificação, validação e teste de software.

A Figura 8 descreve o processo utilizado pela abordagem para atestar se uma determinada instância de código-fonte é definida como sujeita a falhas ou não. Em linhas gerais, o processo para executar a abordagem pode ser dividido em quatro etapas principais:

- calcula o valor médio para cada uma das métricas de código utilizadas;
- constrói uma métrica denominada de matriz de violação;
- calcula métricas de violação para cada instância de código analisada; e
- define se a instância analisada é considerada como sujeita a falhas ou limpa.

Figura 9 – Distribuição de classes sujeitas a falhas de acordo com a abordagem ACL




Fonte: Elaborada pelo autor.

A primeira etapa é autoexplicativa e utiliza como dados de entrada as métricas individuais, apresentadas na Tabela 3, extraídas a partir do código-fonte de cada classe. Após essa etapa, é construída uma matriz de violações, que avalia cada métrica a partir da média dos resultados constituídos para todas as classes do projeto. A próxima etapa é verificar o número de violações para cada classe em relação às métricas avaliadas e, por fim, na última etapa, classificar se uma determinada instância (classe de código) está sujeita a falhas ou não.

Para realizar a classificação é necessário definir um ponto de corte que será utilizado como limite e, caso seja violado, identifica se a classe está sujeita a falhas ou não. O ponto de corte é calculado usando o número de métricas que são utilizadas na avaliação. Detalhes completos da abordagem e implementação podem ser encontrados em (YANG; QIAN, 2016) e no repositório que contém as informações sobre deste trabalho.

Os 119 projetos de RV foram analisados segundo a abordagem descrita. De acordo com o *threshold* de classificação adotado, das 21.508 classes contidas em todos os projetos, um total de 2.627 classes ou 12,21% (Figura 4.1b) foram classificadas como classes com alta probabilidade de apresentarem falhas, devido ao fato de extrapolarem o limite definido na abordagem para considerá-las limpas.

Da mesma forma, entre os 107 projetos N-RV, de 21.568 classes, um total de 1.921 foram rotulados como sujeitos a falhas, o que corresponde a um percentual de 8,90% (Figura 4.1a) das classes analisadas.

Ao realizar uma análise superficial, é possível obter uma visão equivocada dos resultados do estudo, ao interpretar que o percentual de classes que estão sujeitas a falhas representa um percentual pequeno dentro do universo analisado e, portanto, concluir que tempo e o investimento necessários para identificar possíveis problemas talvez não se justifiquem. No entanto, de acordo

com avaliações experimentais anteriores (WALKINSHAW; MINKU, 2018), observou-se que o princípio *Pareto*, que versa sobre a distribuição de eventos apontando que para muitos eventos, aproximadamente 80% dos efeitos são desencadeados a partir de 20% das causas, também tende a se aplicar a um contexto de falhas de software.

O trabalho conduzido por Walkinshaw e Minku (2018) apontou que cerca de 20% dos arquivos de código-fonte são responsáveis por até 80 % das falhas encontradas em um projeto. Portanto, é natural que o percentual de classes sujeitas a falhas apresenta uma tendência similar a essa. Outro ponto pertinente a ser destacada é que uma vez que a abordagem de classificação não é uma fórmula exata e sirva apenas como um mecanismo para auxiliar nos esforços de aplicação de uma abordagem de teste de software, é natural que a proporção apresentada seja inferior e não esteja completamente ajustada à proporção indicada no contexto da regra de *Pareto*.

Pode-se observar que, apesar de possuir um maior número de classes e linhas de código, os projetos N-RV apresentaram uma menor taxa de predisposição a falhas. Vale ressaltar que o algoritmo é executado apenas nas classes relacionadas ao código-fonte da aplicação, desconsiderando as classes de teste nos projetos N-RV.

Tal resultado pode ser um indicativo de que, devido à prática de testes, as classes dos projetos N-RV têm um maior grau de confiabilidade e, portanto, são menos propensas a falhas quando comparadas às classes existentes nos projetos RV, que em sua grande maioria não apresentam casos de teste.

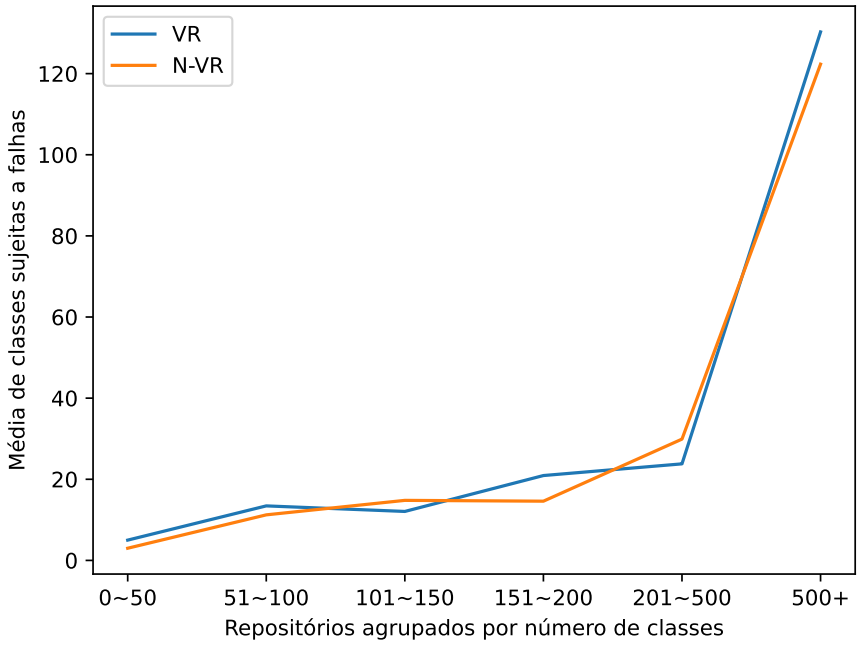
Os resultados observados no experimento e destacados na Figura 9 corroboram com a hipótese investigada, de que a falta de práticas de teste pode contribuir para uma percepção de falta de qualidade no código-fonte, e, por consequência, levar a possíveis problemas.

Outra repercussão pertinente relaciona-se com o custo de desenvolvimento de software, que tende a aumentar, uma vez que historicamente o processo de identificação e correção de falhas durante o processo de desenvolvimento de software representa mais da metade dos custos incorridos durante o ciclo de desenvolvimento (DUSTIN; GARRETT; GAUF, 2009). Dessa forma, o atraso no desenvolvimento do produto pode levar a situações como o aumento do tempo necessário para colocar um produto no mercado, podendo resultar também em perdas de oportunidade de mercado.

Como os projetos analisados no experimento possuem uma grande variedade de tamanhos, foi realizado o agrupamento dos projetos em 6 diferentes categorias, organizadas a partir do número de classes em cada projeto, a fim de observar qual a relação entre o tamanho dos projetos e a distribuição das classes sujeitas a falhas.

A Figura 10 mostra esta distribuição. É possível observar que tanto para os projetos RV, quanto para os projetos N-RV, existe uma clara relação entre o número de classes sujeitas a falhas e o tamanho dos projetos. Esta relação indica que quanto maior o número de classes nos projetos, maior a média das classes sujeitas a falhas, e, portanto, leva à conclusão que negligenciar a

Figura 10 – Distribuição das classes sujeitas a falhas de acordo com o porte dos projetos analisados




Fonte: Elaborada pelo autor.

atividade de teste em projetos maiores pode ser um fator ainda mais crítico em termos de sucesso do projeto.

Com base nos dados apresentados nas análises acima, entende-se que é possível atestar uma resposta clara para a $QP_3$, que motivou o desenvolvimento do estudo, deixando claro que, em um contexto geral, a falta de técnicas de teste de software podem impactar diretamente nos atributos de qualidade, e isso pode refletir diretamente na implicação de más práticas de desenvolvimento, que conduzem à existência de *code smells*, tornando-se consequentemente um fator que pode levar ao aumento de falhas nos projetos de software.

A $QP_3$ buscou investigar se *"É possível traçar uma relação entre métricas de código e predisposição a falhas?"*. Ao implementar a abordagem para investigar a predisposição a falhas nos projetos investigados, foi possível observar que negligenciar a atividade de teste pode aumentar a probabilidade de problemas no processo de desenvolvimento de software. De acordo com a análise realizada verificou-se que os projetos de RV, que não apresentam casos de teste, apresentaram uma maior predisposição a apresentar falhas quando comparados ao projetos N-RV, e, de forma similar, esse problema é diretamente proporcional à complexidade dos projetos, ou seja, o número de classes sujeitas a falhas tende a aumentar à medida que os projetos possuem um número maior de classes, consequentemente, uma maior complexidade.

Outro ponto observado foi que embora os projetos N-RV apresentem casos de teste em todos os projetos, eles ainda apresentam um alto índice de classes sujeitas a falhas. Isso

ressalta a importância do desenvolvimento da prática de teste de software dentro do escopo de desenvolvimento do projeto. Embora os projetos N-RV tenham casos de teste, os conjuntos de teste fornecidos não atendem aos critérios de teste básicos, como cobertura de código, de forma que os trechos de código que não são cobertos pelos testes ainda estão sujeitos a possíveis falhas.

Outro ponto específico levantado pelo estudo foi a necessidade da adoção de práticas de teste específicas para o domínio do software. Diferentes domínios de software apresentam características distintas, que devem ser investigadas de forma adequada. No contexto das aplicações de RV, a simples utilização de testes de unidade pode não ser suficiente para atestar a qualidade do produto desenvolvido, uma vez que tais aplicações fazem uso de recursos avançados como imagens, sons, vídeos e interação, apresentando novos desafios quando comparados a abordagens de teste de software em domínios convencionais.

## 4.3 Limitações e Ameaças à Validade

As principais limitações relacionadas ao desenvolvimento do estudo apresentado neste capítulo estão relacionadas ao fato de que os dados utilizados neste estudo foram coletados a partir da plataforma de hospedagem de código-fonte aberto e arquivos com controle de versão *Github*.

Embora os dados coletados tenham permitido discutir o estado da prática em relação à aplicação de técnicas de teste de software no contexto de aplicações de RV de código aberto, os projetos de código aberto ainda representam apenas uma pequena parte do que é produzido no contexto de aplicações de RV. Projetos comerciais e projetos fechados também fazem parte desse universo, não sendo possível atestar que os resultados discutidos a partir de dados extraídos de projetos de código aberto possam ser generalizados para esses outros cenários.

Além das limitações descritas acima, outro obstáculo que deve ser apontado é o fato de que apesar das suposições feitas durante o estudo estarem relacionadas ao contexto das aplicações de RV, deve-se ressaltar que todas as amostras observadas utilizam apenas uma tecnologia (*Unity*), portanto os resultados aqui indicados não podem ser generalizados para outras plataformas. Para tanto, novos estudos devem ser realizados para corroborar ou contrapor os resultados apresentados neste estudo.

A validade dos resultados obtidos em experimentos depende de fatores nas configurações do experimento (WOHLIN *et al.*, 2012). Diferentes tipos de validade podem ser priorizados dependendo do objetivo do experimento. No que se refere às ameaças à validade deste estudo, que estão relacionadas tanto à avaliação de *code smells*, quanto uma abordagem para detecção de predisposição a falhas em códigos-fonte, é possível destacar que realizar uma análise utilizando amostras extraídas de uma plataforma de projetos de código aberto é uma ameaça à validade significativa - devido ao fato de que a falta de representatividade dos projetos em servir como uma amostra real do universo de todos os tipos de projetos para o domínio de aplicações em RV.

Infelizmente a falta de representatividade é um problema que afeta toda a área de engenharia de software, uma vez que não existe uma teoria bem desenvolvida capaz de garantir que um conjunto de programas seja considerado uma amostra representativa para experimentação (JURISTO; MORENO, 2013). Para tentar mitigar esta ameaça, foi selecionado o maior número de projetos possível, variando em tamanho (pequeno, médio e grande) e fins de aplicação (entretenimento, simulação, treinamento, saúde).

Outra medida tomada para tentar mitigar a ameaça descrita acima foi a análise de projetos N-RV, que serviu de subsídio para comparação com os resultados obtidos em projetos RV, garantindo uma discussão mais fundamentada e uma base de referência mínima para comparação, uma vez que, infelizmente, ainda não há um *benchmark* de projetos catalogados com aplicações de RV que atendam aos requisitos mínimos de representatividade para a serem utilizados neste trabalho.

Em relação às ameaças à validade de construção (WOHLIN *et al.*, 2012), possíveis erros podem ser apontados tanto na análise dos *code smells*, quanto na avaliação da abordagem de códigos com predisposição a falhas. Para minimizar esta ameaça, foi utilizada a ferramenta *Designite*, que é uma ferramenta comercial e já foi usada com sucesso em outros experimentos (SHARMA; MISHRA; TIWARI, 2016).

Além de utilizar uma ferramenta comercial para apoio à coleta de dados relacionados aos *code smells*, todos os dados apresentados foram avaliados tanto por meio de análises descritivas, quanto por meio de testes de hipóteses para garantir que as conclusões extraídas do estudo permitem-nos traçar uma discussão clara sobre o assunto explorado.

Em relação à abordagem para detectar trechos de códigos sujeitos a falhas, a estratégia utilizada foi previamente validada por meio de experimentos em grandes conjuntos de dados para atestar sua eficácia (YANG; QIAN, 2016; YANG; YANG; QIAN, 2018), além de destacar o fato de que o ponto principal da abordagem não era, especificamente, detectar falhas nos projetos, mas apontar classes com alta probabilidade de apresentarem falhas, servindo como um guia para direcionar os esforços de teste e discutir a necessidade de aplicação de técnicas de teste de software para evitar tais cenários.

Conforme apontado por Nam (2014), as abordagens de predição de falhas desempenham o papel de uma abordagem complementar para ajudar a identificar potenciais problemas no código-fonte, bem como um mecanismo para melhorá-lo e, consequentemente, se livrar de gargalos de produtividade e problemas futuros. Assim, os resultados apresentados nesse estudo não pretendem apontar um número preciso de falhas nos projetos de software avaliados, mas fortalecer evidências a respeito da hipótese de que projetos que adotam critérios de qualidade, como a prática de teste de software, tendem a estar menos predispostos a apresentar problemas.

Também é importante frisar que, por não haver informações precisas sobre os critérios de teste utilizados para criação dos conjuntos de testes disponíveis nos projetos N-RV, assim

como qualquer informação sobre a cobertura alcançada pelos testes projetados, é impossível garantir que os testes projetados para uma determinada classe são adequados e, por consequência, suficientes para garantir qualquer aspecto de qualidade com relação a mesma. Portanto, é natural que haja similaridade nos resultados de predisposição a falhas entre os projetos que não foram testados (projetos de RV) e os projetos que contêm casos de teste (projetos N-RV).

Por fim, com relação às ameaças à validade interna do estudo, que estão relacionadas ao nível de confiança entre os resultados esperados e os resultados obtidos (WOHLIN *et al.*, 2012), todo o estudo foi conduzido de forma a minimizar essa ameaça. Para aumentar a confiança sobre os resultados apresentados, os dados foram analisados por meio de tabelas e gráficos e testes estatísticos, além de terem sido disponibilizados em um repositório aberto para possibilitar a replicação caso seja considerada necessária (ANDRADE; NUNES; DELAMARO, 2021).

## 4.4 Considerações Finais

Este capítulo abordou os principais desafios relacionados ao uso da prática de teste de software no domínio de aplicações de RV. Algumas das questões críticas relacionadas à qualidade desses sistemas foram apontadas e também foram discutidas possíveis soluções que poderiam ser utilizadas e adaptadas para lidar com tais questões.

O capítulo abordou se há ou não uma necessidade real de testar aplicações de RV. Para tentar compreender de forma mais clara, foi realizado um estudo abrangente, norteado por 3 questões de pesquisa, cujo objetivo foi: compreender o estado da prática de teste de software no contexto de aplicações de RV de código aberto ($QP_1$), medir métricas e atributos de qualidade de código em aplicações de RV ($QP_2$), e por fim avaliar a predisposição a falhas na coleção de projetos de software RV analisados ($QP_3$).

Para responder às questões levantadas, foi catalogada uma coleção de 119 projetos de aplicações em RV, disponíveis em projetos de código aberto, e analisados manualmente, para compreender o estado da prática relativa à aplicação de técnicas de teste de software. Com relação à aplicação de técnicas de teste de software ($QP_1$), observou-se que do total de projetos, apenas 6 deles possuíam práticas de teste de software, com um conjunto mínimo de casos de teste para avaliar um pequeno escopo de funcionalidades.

Diante dos resultados apontados pela $QP_1$, decidiu-se avaliar como a negligência da prática de teste de software pode ser prejudicial a um projeto de software, investigando a distribuição de *code smells* entre os projetos analisados. Foram analisados *smells* relacionados a características como arquitetura, design e implementação. Por meio da análise foi possível concluir que há uma alta incidência de *smells* nos projetos analisados, principalmente no que se refere a *implementation smells*. Foi apresentada uma discussão sobre os *smells* mais comuns para cada uma das categorias observadas e como eles podem inibir a prática de teste de software, bem como podem ser evitados se uma atividade de teste de software for conduzida de forma

adequada.

Com base nos resultados observados na $QP_2$ decidiu-se investigar como a falta de boas práticas e a presença de *code smells* podem impactar na qualidade do código-fonte produzido. Para isso, foi utilizada uma abordagem que avalia métricas de código para apontar classes de código sujeitas a falhas ($QP_3$). O estudo apontou que cerca de 12% das classes presentes nas aplicações de RV analisadas possuem tais características, revelando um risco significativo para o sucesso dos projetos. A distribuição dessas classes também foi avaliada quando observada quanto ao porte dos projetos analisados. Os resultados apontaram que quanto maior se torna um projeto, maior é a incidência de classes propensas a falhas, o que pode ser um indicativo de que negligenciar as práticas de teste em projetos maiores torna-se ainda mais arriscado.

Os resultados relatados neste capítulo estão disponíveis publicamente em duas versões (ANDRADE; NUNES; DELAMARO, 2019; ANDRADE; NUNES; DELAMARO, 2020) e contribuem para aumentar a evidências a respeito da necessidade de abordagens de teste de software específicas para aplicações de RV. Como a fase de teste de software, também compõe uma das fases de desenvolvimento, é necessário entender o ponto de vista dos *stakeholders* envolvidos no processo, permitindo que esses grupos possam indicar pontos de interesse que podem ser investigados durante a abordagem de teste, de forma que seja possível priorizar a verificação das falhas que devem que possuam uma maior probabilidade de se manifestar de acordo com o contexto da aplicação desenvolvida.

Observando os aspectos destacados acima é possível orientar o desenvolvimento de uma abordagem de teste específica para o domínio da RV. Assim, o próximo capítulo discute tipos de falhas em aplicações de RV contribuem para uma experiência negativa sob o ponto de vista de *stakeholders*. O objetivo deste estudo é obter uma visão dos grupos de interesse (por exemplo, quais tipos de falhas são mais críticas, quais são menos relevantes, quanto cada uma afeta a qualidade do produto final, etc.), além de investigar o conhecimento dos grupos de interesse sobre tipos específicos de falhas em aplicações de RV.

CAPÍTULO

5

# COMPREENDENDO AS DIFICULDADES DE TESTE DE SOFTWARE PARA REALIDADE VIRTUAL

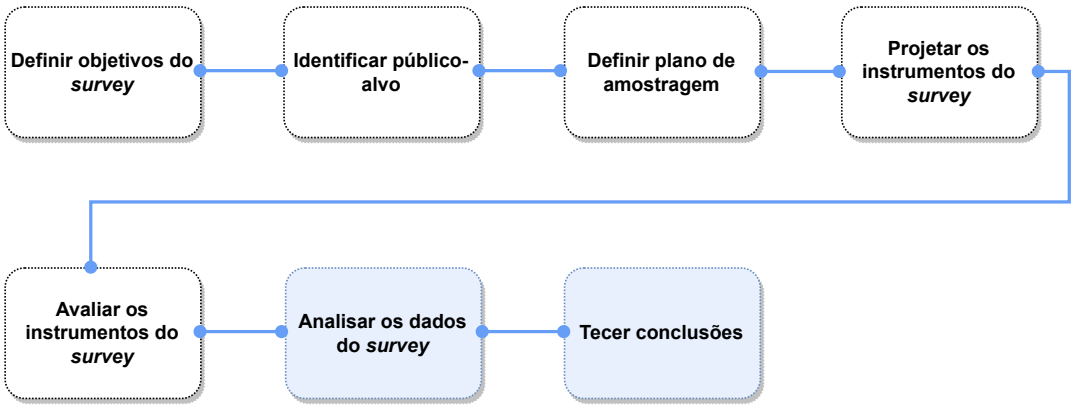
O objetivo deste capítulo é apresentar uma visão de *stakeholders* envolvidos no processo de concepção, desenvolvimento e utilização de aplicações de RV com intuito de investigar o conhecimento de tais grupos sobre tipos específicos de falhas em aplicações de RV. Como exemplo de pontos de interesse a serem investigados é possível explorar características como: quais tipos de falhas são mais críticas, quais são menos relevantes e o quanto cada uma afeta a qualidade do produto final. Para obter tais percepções, foi realizado um *survey* no qual os participantes foram convidados a responder a um conjunto de perguntas que englobam as características de interesse apontadas anteriormente.

Essa é uma prática que pode ser explorada em diferentes contextos de software e diversos domínios. Para a atividade de teste de software no domínio de RV, embora existam trabalhos na literatura que explorem a aplicação de práticas de teste de software (Seção 3.3), estudos mais atuais apontam que ainda existe uma dificuldade em preencher a lacuna na transferência de conhecimento entre academia e profissionais (ANDRADE; NUNES; DELAMARO, 2019).

Uma hipótese para essa problemática é a de que a falta de adesão esteja relacionada ao fato de os trabalhos acadêmicos tendem a explorar excessivamente o ponto de vista do pesquisador, deixando de lado os atores envolvidos no contexto prático, que serão os maiores responsáveis pela utilização do conhecimento produzido (CICO *et al.*, 2021). Assim, a fim de tentar reduzir tal lacuna no conhecimento, o ponto de vista de *stakeholders* deve ser compreendido. Esse processo pode possibilitar que o conhecimento produzido seja, portanto, direcionado ao que tais grupos consideram realmente importante, possibilitando priorizar esforços em aspectos que, idealmente, são considerados importantes.

Esse capítulo está organizado da forma: a seção 5.1 apresenta como o *survey* foi estabe-

Figura 11 – Processo adotado na condução do *survey*




Fonte: Elaborada pelo autor.

lecido, quais pontos são investigados e quais questões de pesquisa guiam o desenvolvimento do mesmo; a seção 5.2 apresenta os resultados após a compilação das respostas obtidas; a seção 5.3 discute os resultados do *survey*; a seção 5.4 aponta algumas limitações e ameaças relacionadas ao estudo apresentado neste capítulo e finalmente as conclusões, direcionamentos e considerações finais são apresentadas na seção 5.5.

## 5.1 Delineamento do Estudo

O *survey* foi planejado seguindo as diretrizes propostas por Linåker *et al.* (2015), que propõem uma abordagem para projetar pesquisas eficazes para o contexto da área de engenharia de software. Portanto, o processo utilizado durante a realização do *survey* foi dividido em sete etapas principais, representadas na Figura 11 (as caixas brancas representam as etapas seguidas para aplicar a pesquisa, e as caixas azuis identificam as fases relacionadas à extração de dados e interpretação) e discutidas nas subseções abaixo.

### 5.1.1 Objetivos do survey

O objetivo do *survey* foi formulado utilizando a abordagem *Goal, Question, Metric* (GQM) (BASILI, 1994), definindo como finalidade entender o ponto de vista de *stakeholders* com relação ao desenvolvimento de práticas de teste de software em aplicações de RV.

Além de compreender a opinião a respeito da necessidade de práticas de teste de software, o *survey* também buscou entender as principais preocupações dos *stakeholders* sobre os tipos de falhas em aplicações de RV que contribuem para uma experiência ruim do usuário. Tal avaliação foi realizada com base na análise de características que são consideradas críticas e, portanto, que poderiam ser priorizadas durante a fase de teste em aplicações de RV.

Especificamente, foram investigadas as seguintes questões de pesquisa:

- **QP$_1$**: Qual a percepção dos *stakeholders* quanto à presença de falhas ou bugs nas aplicações de RV?
- **QP$_2$**: Quais são as percepções *stakeholders* sobre quais características de falha de hardware mais contribuem para uma experiência negativa/ruim em aplicações de RV?
- **QP$_3$**: Quais são os principais tipos de falhas que os *stakeholders* têm conhecimento em aplicações de RV?

### 5.1.2 Identificação do público-alvo

No contexto de *surveys* é fundamental conhecer o público-alvo, a partir do qual são definidas a população, a base amostral e a amostra. O público-alvo descreve para quem o resultado do *survey* é de interesse. A descrição fornece subsídios para os critérios de seleção, pois ao caracterizar o público-alvo é possível identificar as características e hábitos dos mesmo (FELDERER; TRAVASSOS, 2020). Um subconjunto do público-alvo é a população. A base de amostragem é um subconjunto da população e compreende o universo acessível.

A pesquisa foi inicialmente planejada e projetada para atingir qualquer ator envolvido no desenvolvimento e uso de aplicações de RV.

### 5.1.3 Definição de plano de amostragem

Para atingir o público-alvo, o *survey* foi conduzido durante a *21ª edição do Simpósio de Realidade Virtual e Aumentada* (SVR), que é a primeira conferência sobre realidade virtual e realidade aumentada no Brasil, realizada anualmente pela Sociedade Brasileira de Computação [1].

Optou-se por realizar o *survey* durante este evento devido ao fato do SVR reunir pesquisadores, estudantes e profissionais de diversas áreas acadêmicas, industriais e comerciais interessados nos avanços e aplicações de RV, contemplando um amplo universo de pessoas que compõem o público-alvo inicialmente desejado.

Além da aplicação presencial, o *survey* também foi disponibilizado em uma versão eletrônica para atingir um público geral que reflita as percepções do ponto de vista dos usuários da tecnologia.

### 5.1.4 Definição dos instrumento do survey

Os objetivos propostos na primeira etapa do processo foram definidos, bem como também foi estabelecida a forma como seriam coletados os dados fornecidos pelos participantes. Foi estabelecido que os objetivos do *survey*, bem como a sua motivação seriam apresentados de

[1] <http://svr.sbc.org.br/>

forma oral ao público-alvo e o mesmo seria distribuído fisicamente para os participantes, sendo facultativa a sua participação.

A versão eletrônica foi dividida em quatro seções: a primeira seção contendo uma apresentação do *survey* na qual são descritos os objetivos, bem como o público-alvo e esclarecimentos sobre a importância da participação; a segunda seção teve como objetivo caracterizar o papel dos participantes e compreender suas percepções sobre o uso de tecnologias de RV; o objetivo da terceira seção era compreender os fatores críticos que podem dificultar a capacidade de utilizar e aproveitar aplicações de RV; por fim, o objetivo da quarta seção foi entender o conhecimento dos *stakeholders* sobre os vários tipos de falhas que existem nas aplicações de RV, bem como entender os tipos de falhas que os *stakeholders* julgam como mais críticas.

O *labpackage* do *survey* contendo todas as perguntas e a coleção de respostas, bem como as análises está eletronicamente disponível em (ANDRADE *et al.*, 2021).

### 5.1.5 Avaliação dos instrumento do survey

Projetar os instrumentos do *survey* é uma atividade iterativa que envolve uma avaliação por meio da realização de uma execução piloto para introduzir melhorias no instrumento de pesquisa (LINÅKER *et al.*, 2015). Portanto, para evitar perguntas ambíguas e mal formuladas, o *survey* foi inicialmente avaliado com um pequeno grupo de três pesquisadores para coletar *feedback* e entender se as perguntas propostas estavam claras e se o material fornecia informações suficientes para a sua realização.

Durante a aplicação oficial quaisquer praticantes poderiam desistir a qualquer momento e apenas informações agregadas e sumarizadas foram publicadas. Dessa forma, apenas apenas os *surveys* que tiveram todas as respostas preenchidas foram considerados para a análise final dos resultados.

### 5.1.6 Análise dos dados

As perguntas foram projetadas para se ajustar, majoritariamente, a um modelo classificado como perguntas fechadas (FINK, 2002). Tal modelo fornece uma lista fixa de opções ou categorias de respostas e pede aos participantes que selecionem uma ou mais opções como resposta.

Este modelo coleta dados quantitativos e, portanto, possibilita realizar uma análise por meio de gráficos e métodos quantitativos.

### 5.1.7 Tecer conclusões

Após a compilação dos resultados, os dados das respostas foram analisados e os resultados são apresentados na próxima seção. Para evitar viés de interpretação e excesso de confiança no

processo de interpretação dos resultados, eles foram avaliados e revisados por um grupo de 4 pesquisadores, sendo 2 deles especialistas na parte de teste de software e 2 especialistas sob o ponto de vista de aplicações de RV.

## 5.2 Resultados

O *survey* foi respondido completamente por um total de 88 *stakeholders*, que compõem diversas funções no processo de desenvolvimento e utilização de aplicações de RV e as subseções abaixo compilam os resultados obtidos.

### 5.2.1 Caracterização dos participantes

Para entender o perfil dos participantes do *survey* quanto à forma como eles se classificam no contexto de aplicações de RV, a primeira questão apresentada foi *"Qual o seu perfil no contexto do desenvolvimento/utilização de aplicações de RV"*

Tabela 7 – Perfil dos participantes

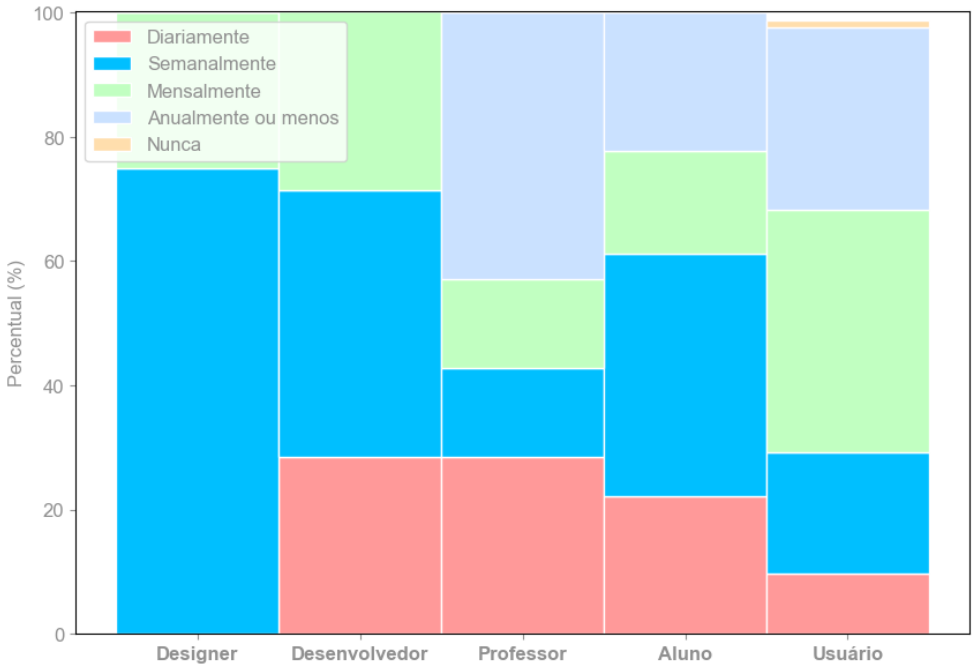
| **Perfil** | **Quantitativo** | **Percentual** |
|---|---|---|
| Usuário | 41 | 47% |
| Aluno | 18 | 20% |
| Professor | 14 | 16% |
| Designer | 8 | 9% |
| Desenvolvedor | 7 | 8% |
| **Total** | **88** | |

A Tabela 7 apresenta o resultado para o primeiro questionamento do *survey*. Com 47% das respostas, o perfil *usuário* foi o que obteve o maior número de participantes, seguido por *alunos* com 20%, *professores* com 16%, *designers* com 9% e, finalmente, *desenvolvedores* com 8%.

O fato de o *survey* ter sido realizado inicialmente durante uma conferência científica relacionada à realidade virtual (SVR) e co-localizada com a *Conference on Graphics, Patterns and Images* (SIBGRAPI) e *Simpósio Brasileiro de Jogos e Entretenimento Digital* (SBGames) permitiu obter um leque de diferentes perfis, o que é um fator positivo para o propósito deste estudo, pois tais perfis permitem observar diferentes visões sobre o tema investigado e podem ajudar a reduzir o viés das conclusões obtidas.

Após identificar o perfil dos participantes, buscou-se avaliar a frequência com que eles usam aplicações que incluem algum tipo de recursos de RV. O racional por trás dessa pergunta ajuda a entender o comportamento dos participantes, especialmente porque a função de cada participante está ligada à frequência de uso. Portanto, perguntamos a *"Com que frequência você utiliza aplicações de realidade virtual?"*.

Figura 12 – Frequência de uso de aplicações de RV de acordo com o perfil




Fonte: Elaborada pelo autor.

Os resultados apresentaram que não há padrão quanto à frequência de uso com relação ao perfil do usuário. Embora, em geral, a tendência de uso seja alta uma vez que a combinação dos resultados apresentados para uso diário e semanal representem mais de 45% dos participantes. Porém, ainda existe uma grande parcela que faz uso com pouca frequência, posicionada no quadrante "anualmente ou menos", um total de 25 % dos participantes. Compreender a frequência de uso dos *stakeholders* permite estabelecer associações interessantes, como, por exemplo, a relação entre o perfil do participante e o hábito de uso (Figura 12).

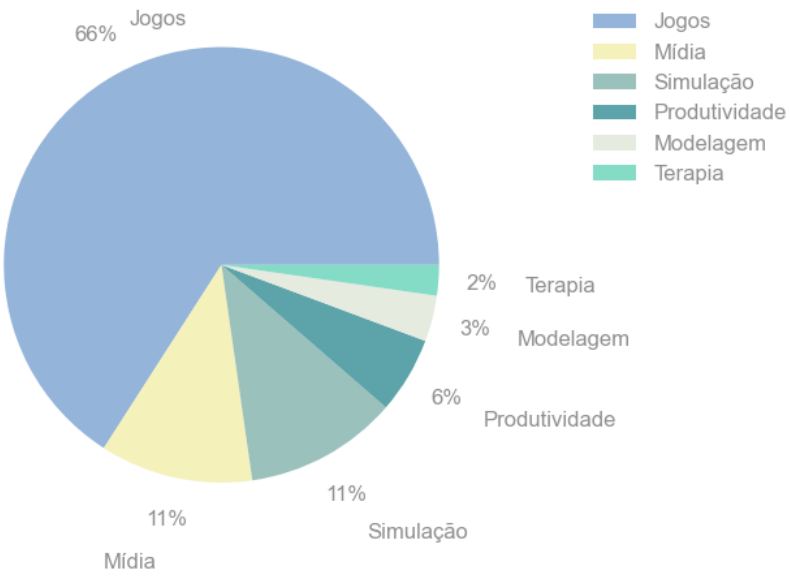
Ao verificar a associação entre o perfil dos participantes e a frequência de uso de aplicações de RV, é possível observar que há um interesse pelo uso desse tipo de tecnologia, independente do perfil analisado, uma vez que grande parte do público indicou que a frequência de uso varia principalmente entre o uso diário e mensal. Essa percepção observada está de acordo com trabalhos existentes na literatura.

Um grupo de *stakeholders* tende a ter uma série de características em comum que fazem com que os mesmos sejam impactados por um conjunto de ações em um dado contexto. Contudo, no contexto de RV, pode haver uma grande discrepância quanto aos propósitos para os quais os mesmos possam utilizar aplicações de RV. Nesse sentido, os participantes também foram questionados a respeito dos seus hábitos de uso para que fosse possível caracterizar as principais áreas de uso de aplicações de RV de acordo com o perfil do participante: *"Que tipo de aplicações de RV você mais utiliza?"*.

Ao analisar a Figura 13, é possível observar que o principal uso de aplicações de RV é

para entretenimento/jogos, sendo os jogos a resposta mais comum entre os participantes (66% das respostas).

Figura 13 – Principais áreas de uso de aplicações de RV




Fonte: Elaborada pelo autor.

Cientes de que a RV é uma tecnologia cada vez mais presente no dia a dia das pessoas, para finalizar a etapa de caracterização do papel dos participantes, questionou-se sobre a opinião dos mesmos a respeito do quanto mau funcionamento ou *bugs* poderiam contribuir para uma experiência negativa na utilização de aplicações de RV. Para isso, a opinião dos participantes foi medida por meio de uma escala *likert* (ALBAUM, 1997), visando mensurar a seguinte afirmação: *"Falhas ou bugs contribuem para uma experiência negativa usando aplicativos de RV"*.

Os resultados, apresentados na escala *likert* apresentada na Figura 14, permitem entender a importância percebida pelos participantes com relação à qualidade da experiência ao usar uma aplicação de RV. Até 70% dos participantes (69 no total) concordam ou concordam totalmente que o mau funcionamento ou *bugs* em aplicações de RV contribuem e podem gerar uma experiência negativa.

Figura 14 – Percepções dos *stakeholders* (em números absolutos) com relação a presença de *bugs* em aplicações de aplicação de RV


Fonte: Elaborada pelo autor.

Implicitamente, tais resultados indicam que os desenvolvedores de aplicações de RV devem atentar a aspectos da qualidade do produto desenvolvido, pois esse é um fator que, do ponto de vista das partes interessadas, pode ter um forte impacto na experiência final durante o uso.

Com a caracterização e a compreensão da percepção dos *stakeholders* com relação às necessidades de aspectos de qualidade para aplicações de RV, os mesmos passaram para a segunda etapa do questionário, cujo principal objetivo era verificar os pontos de vista sobre os aspectos de falha que podem impactar aplicações de RV.

### 5.2.2 Percepções sobre aspectos de software

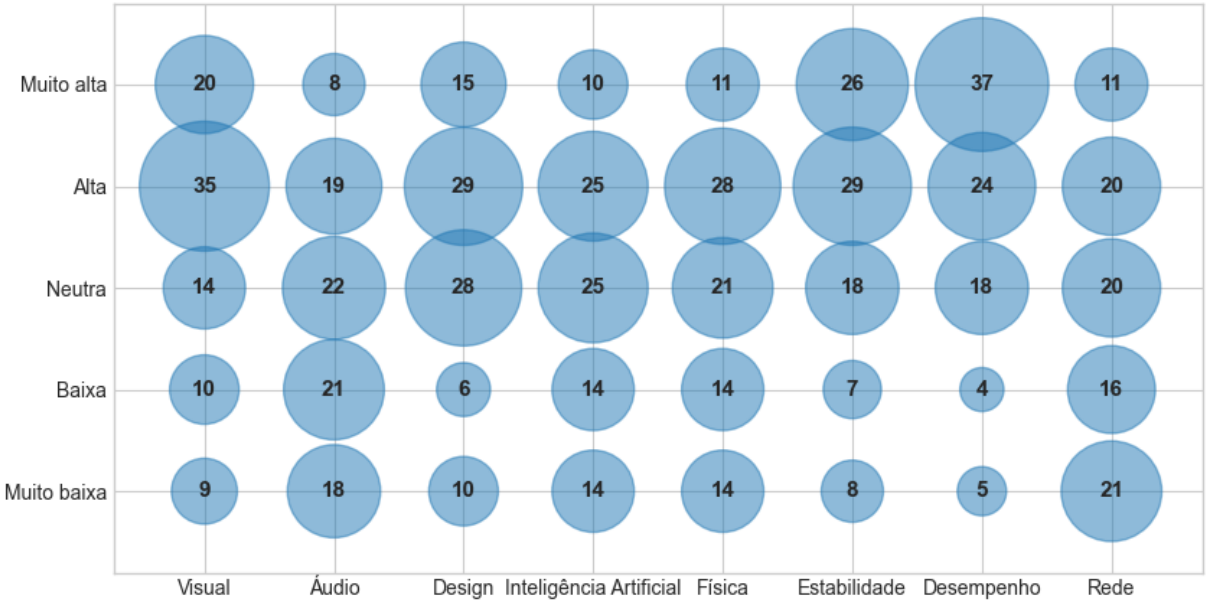
O principal objetivo dessa análise foi medir a percepção dos *stakeholders* quanto à frustração com relação a algumas características específicas que compõem os componentes de software de uma aplicação de RV. Para isso, foram divididos os principais conceitos que constituem uma aplicação de RV, com base na classificação arquitetural desse tipo específico de aplicação (CAPILLA; MARTÍNEZ, 2004; TOBLER, 2011; DUVAL, 2012) e em características observadas em modelos de desenvolvimento de jogos 3D (LEVY; NOVAK, 2009), a saber:

- aspectos visuais: estão relacionados a falhas que afetam os gráficos de aplicações de RV de geral. Podem variar de pequenas cintilações a sérios problemas de textura;
- aspectos de áudio: podem variar desde diferenças de queda de volume até grandes falhas na reprodução de áudios;
- aspectos de design: quando uma cena é mal construída, o que pode resultar em problemas como buracos no chão ou paredes invisíveis, etc;
- aspectos da inteligência artificial: principalmente relacionados às ações realizadas pelos atores na cena. Geralmente podem levar a problemas como comportamentos anômalos ou problemas de algoritmo de *pathfinding*;
- aspectos físicos: apesar de normalmente gerenciados por uma *engine*, os problemas dessa categoria podem incluir problemas com objetos quebráveis e comportamentos dinâmicos gerais;
- aspectos de estabilidade: incluem características relacionadas a problemas como travamentos não intencionais e carregamento;
- aspectos de desempenho: referem-se à velocidade com que o hardware processa o código e incluem problemas relacionados à taxa de quadros da cena, tempo de carregamento, assim como requisitos mínimos do dispositivo que executará a aplicação;

- aspectos de rede: estão especificamente relacionados à conectividade cliente-servidor e ocorrem majoritariamente em aplicações que fazem uso deste tipo de recurso.

Após esclarecer aos participantes sobre o conceito de cada uma das características descritas, eles foram solicitados a avaliar, em uma escala de gravidade substancial, o quanto as falhas relacionadas a cada um dos conceitos impactam negativamente as experiências ao usar aplicações de RV: *"Qual das opções a seguir afeta a usabilidade, incomoda ou frustra sua experiência com aplicativos de RV? "*

Figura 15 – Gráfico de bolhas que avalia aspectos de software que, em caso de falhas, podem causar incômodo/frustração na experiência de usuários em aplicações de RV




Fonte: Elaborada pelo autor.

Os resultados apresentados na Figura 15 mostram que não há consenso absoluto na opinião dos participantes. A escala representada no eixo "*y*" da figura define o grau de importância de cada um dos aspectos avaliados, no eixo "*x*" cada aspecto é listado. O ponto de intersecção compreende ao número de respostas dos participantes para cada categoria analisada.

É possível observar que, em geral, existe um grau de relevância semelhante para todos os aspectos avaliados, uma vez que as percepções *baixa* e *muito baixa* foram as menos apontadas em quase todas as categorias.

A categoria que apresentou a menor percepção de importância foi relacionada aos recursos *Rede*. Uma possível explicação para esse resultado pode ser o fato de que nem todos os tipos de aplicações de RV fazem uso de recursos de rede em suas funcionalidades. Portanto, um participante que usa esse tipo de aplicação pode julgar que recursos de rede não são fatores importantes a serem considerados.

Por outro lado, aspectos como *Design*, *Visual*, *Física* e *Desempenho* foram os que receberam maior número de votos como recursos que podem causar grande frustração ou desconforto se apresentarem problemas durante o uso de aplicações de RV. Diferentemente dos recursos de rede, esses recursos estão presentes na maioria das aplicações de RV, portanto, são considerados fundamentais para que o usuário tenha uma experiência adequada.

Em relação aos resultados apresentados para a categoria *Visual*, os resultados vão ao encontro de outros estudos que avaliam o impacto e apelo que os aspectos visuais têm nas aplicações de RV. Marchiori, Niforatos e Preto (2018) relataram em um experimento, por meio da verificação de sinais vitais de um grupo de usuários, que os aspectos visuais são fatores essenciais que contribuem para que o usuário de uma aplicação de RV se sinta realmente imerso e desfrute de uma experiência conforme o esperado. Assim, a percepção de como problemas dessa natureza se refletem na experiência do usuário é um fator chave para o sucesso de uma aplicação.

Normalmente, uma simulação *Física* é responsável por fornecer os comportamentos dinâmicos e detecção de colisão para objetos virtuais em ambientes virtuais, a fim de simular um mundo real. Assim, os aspectos dessa categoria desempenham um papel vital tanto para garantir o correto funcionamento da aplicação, como na percepção do utilizador que está relacionada com o grau de satisfação do utilizador ao interagir com a aplicação.

Por fim, distúrbios relacionados à categoria de *Desempenho* revelam problemas durante a execução de um aplicativo de RV podem levar a atrasos, problemas de tempo de resposta, queda na taxa de quadros e, por consequência, mal-estares como náusea, desorientação, dores de cabeça, suor e cansaço visual (DAVIS; NESBITT; NALIVAIKO, 2014). Tais sintomas são descritos na literatura como *cibersickness* e podem causar desconforto e, consequentemente, atrapalhar a experiência do usuário. Assim, para proporcionar uma experiência agradável em uma aplicação de RV, o desempenho, que faz parte da lista de requisitos não funcionais da aplicação, é um fator que deve ser levado em consideração.

### 5.2.3 Percepções sobre aspectos de hardware

Após coletar as opiniões dos participantes sobre características de software em aplicações de RV, foi coletada a opinião a respeito de possíveis problemas de hardware, que podem estar relacionados a dispositivos que são comumente usados para fornecer maior imersão e interação mais adequadas em um ambiente virtual.

Aplicações de RV frequentemente dependem de um hardware especializado e, da mesma forma que os aspectos do software, a interação desse tipo de dispositivos com a aplicação também está sujeita a falhas. Dentre os principais dispositivos que são utilizados em aplicações de RV, destaca-se: *head-mounted display*, periféricos hápticos, controladores de movimento, mouse e *joysticks*, além de outros tipos de periféricos para controlar rastreamento, captura de gesto ou

voz (Seção 2.2).

Com base na percepção da importância de dispositivos de hardware no contexto de aplicações de RV, o próximo ponto investigado no *survey* foi com relação à experiência de uso de tais dispositivos, especificamente, investigando o quanto as falhas relacionadas a tais dispositivos podem afetar negativamente a experiência do usuário: *Até que ponto você concorda que o seguinte IMPEDE sua capacidade de usar adequadamente e gosta de experiências de RV?*

Especificamente, foram delimitados sete aspectos distintos relacionados aos recursos de hardware que são comumente utilizados em aplicações de RV (LAVALLE, 2019):

- capacidades técnicas da aplicação **(A1)**: esse tipo de capacidade está relacionado a aspectos relativos aos componentes e sensores que compõem o hardware utilizado em aplicações de RV. Por exemplo, rastreamento baseado em câmera, giroscópios, acelerômetros e magnetômetros, que são responsáveis por funções como orientação de objetos, rastreamento de cabeça e rastreamento rotacional;

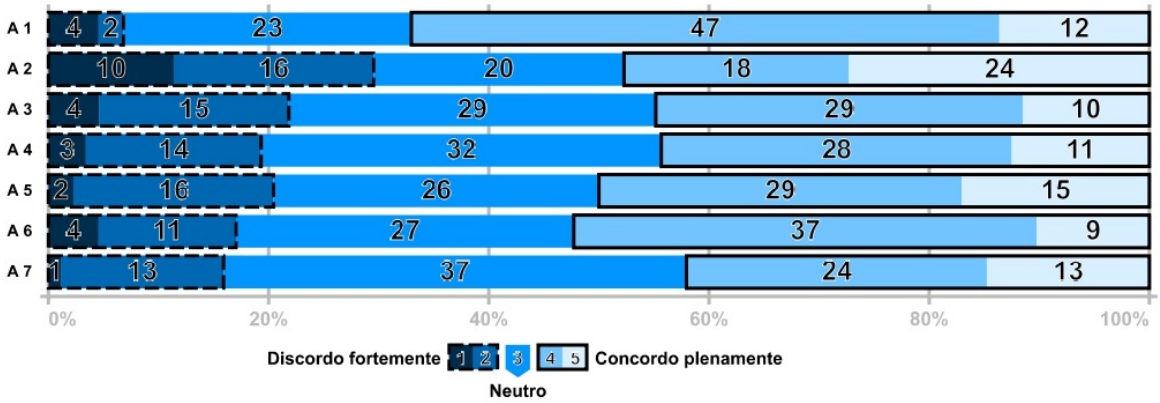
- náusea ao usar o aplicativo **(A2)**: embora não haja consenso quanto ao fator definitivo para esses sintomas, existem vários aspectos técnicos em aplicações de RV que podem induzir à sensação de enjoo, como movimento incompatível, campo de visão, paralaxe de movimento e ângulo de visão. Além disso, a quantidade de tempo exposto em um ambiente de RV pode exacerbar os sintomas;

- fidelidade do mundo virtual **(A3)**: trata de aspectos que especificam o que a "tela visual" deve mostrar ao usuário. Abrange a percepção da distância dos objetos aos olhos do usuário e também está relacionada à percepção da escala de um objeto e como percebemos o movimento;

- presença no mundo real **(A4)**: embora não seja uma característica essencialmente relacionada com uma aplicação de RV, o fato desse tipo específico de aplicação, em alguns casos, proporcionar um maior grau de imersão que exclui a percepção do usuário de aspectos relacionados ao mundo real, faz com que essa seja uma questão de preocupação sobre quem está presente quando o usuário está fazendo uso de uma aplicação completamente imersiva;

- interagir com objetos do mundo real **(A5)**: a literatura de RV aponta que ainda não é possível reconstruir um ambiente virtual totalmente idêntico ao mundo real. Embora seja possível construir ambientes fotorrealísticos, as interações, por outro lado, são complicadas, dependendo do nível de destreza necessário. Um exemplo é a replicação da percepção (como peso e física) de objetos reais. Assim, existem estudos que avaliam como é possível interagir em ambientes virtuais junto a objetos dispostos no mundo real e como isso pode impactar a experiência do usuário (YOSHIMOTO; SASAKURA, 2017);

Figura 16 – Aspectos de uso de hardware que dificultam ou frustram a experiência em aplicativos de RV




Fonte: Elaborada pelo autor.

- interagir com periféricos do mundo real **(A6)**: esse aspecto está intrinsecamente ligado à comunicação de aplicações de RV com dispositivos de hardware (teclado, mouse, controladores de movimento) utilizados para fornecer entradas e como essas entradas são interpretadas por tais aplicações (KNIERIM *et al.*, 2018);

- fornecimento de dados para o mundo virtual **(A7)**: esse é um aspecto complementar ao de interagir com periféricos do mundo real, mas lida com outros tipos de dispositivos, como rastreadores de posição, captadores de gestos ou voz (YANG *et al.*, 2019).

A escala *likert* foi, novamente, utilizada para avaliar a percepção dos participantes quanto a concordar ou discordar da afirmação relacionada aos aspectos de hardware que impedem a capacidade de usar e aproveitar adequadamente de aplicações de RV em caso de falhas e tais dispositivos, segundo os aspectos apresentados anteriormente. A Figura 16 reproduz a percepção dos participantes em relação aos sete aspectos avaliados. De forma geral, pode-se observar que existe uma certa tendência à neutralidade, direcionando a uma percepção de concordância de que os aspectos avaliados podem, de fato, impactar negativamente a experiência em aplicações de RV em casos de falhas.

Embora a maioria dos aspectos tenha apresentado tendência semelhante, é interessante observar que o aspecto **A1** (capacidades técnicas do aplicativo) apresentou a maior margem de concordância, ou seja, a maioria dos participantes concordou que o não funcionamento adequado tende a impedir o uso de aplicações de RV, impactando negativamente a experiência do usuário.

Conforme apresentado por Rebenitsch e Owen (2016), há uma série de fatores que podem impactar as sensações de *cybersickness* (**A2**), porém, ainda faltam fundamentos teóricos para unificar e entender definitivamente a causa de tais sensações.

Sob o ponto de vista de testes de software, abordagens que exploram a percepção do comportamento do usuário a partir da adoção de diferentes campos de visão podem ser uma

alternativa para investigar e diminuir parcialmente os efeitos de *cybersickness*, bem como explorar o comportamento da aplicação sob diferentes graus de liberdade nos mecanismos de navegação. Outra possibilidade é investigar a variação da velocidade quando os usuários navegam por uma cena, pois, embora não haja um consenso absoluto na literatura, essas são algumas das características que são constantemente relacionadas ao efeito de *cybersickness* (PORCINO *et al.*, 2017).

Com relação aos aspectos de fidelidade ao mundo real (**A3**), é necessário manter um equilíbrio entre características como capacidade de processamento e tempo de desenvolvimento, uma vez que experiências mais realistas, normalmente, requerem mais complexidade e, consequentemente, um tempo de produção maior. Produzir aplicações mais complexas também impacta diretamente na aplicação de técnicas de teste, pois o custo da atividade de teste estará diretamente ligado à complexidade do software a ser testado (KEOGH, 2015).

Quanto à avaliação de aspectos de presença no mundo real (**A4**), é um fator que pode ser omitido por se tratar de uma característica complementar à imersão, que é um dos pilares da RV. Porém, com a experimentação de novas possibilidades de interação, há também um aumento nos riscos de acidentes envolvendo o uso de dispositivos de RV (Oregon State University, 2020), portanto, esse é um fator que não pode ser desconsiderado. Aspectos relacionados à interação com objetos reais em conjunto com experiências de RV podem aumentar o grau de complexidade da aplicação, mas devem ser balanceados, pois podem aumentar os riscos de causar acidentes, uma vez que a aplicação deve estar preparada para identificar tais objetos reais e como reagir a eles.

Os aspectos relacionados à capacidade de interagir com objetos do mundo real (**A5**) podem descrever formas mais complexas de interação, como, por exemplo, dispositivos *e-skin* que permitem manipular objetos em mundos reais e virtuais ao integrarem sensores magnetoeletrônicos em uma pele eletrônica que rastreia os movimentos do usuário (CHEN *et al.*, 2022). Nesse sentido, a avaliação de tal capacidade passaria por validar as características especificas do dispositivo que captura os dados que são utilizados no ambiente virtual.

Por fim, a capacidade de fornecer dados em tempo real por meio de periféricos (**A6**) e como uma aplicação de RV reage a isso (**A7**) é um fator chave, pois juntamente com a imersão, a interação em tempo real é uma das principais características da RV e deve ser levada em consideração ao avaliar os aspectos relacionados à qualidade. Nesse sentido, a avaliação do tempo de resposta da aplicação e a interpretação correta das entradas fornecidas pelo usuário (RAU *et al.*, 2018) podem ser alternativas a serem exploradas para validar o correto funcionamento de tais propriedades.

Embora as técnicas de interação em tempo real e fornecimento de entradas sejam características que são amplamente utilizadas em aplicações de RV, elas ainda apresentam alguns riscos e podem ser a fonte de falhas. Algumas das limitações estão relacionadas ao tempo de resposta e problemas de *feedback* visual (ARGELAGUET; ANDUJAR, 2013). Portanto, técnicas de teste

de software podem explorar esse espaço para garantir que aplicações de RV possam entregar soluções que apresentem respostas precisas e um *feedback* visual adequado.

Os diferentes tipos de dispositivos e tecnologias usados para proporcionar a sensação de imersão em experiências de RV podem influenciar diretamente no universo de possíveis falhas que podem ocorrer em uma aplicação de RV. À medida que as aplicações atingem um grau de maturidade e desenvolvimento, capaz de criar ambientes cada vez mais realistas, também há uma tendência de aumento da complexidade. Nesse sentido, a última etapa do *survey* concentrou-se em catalogar o conhecimento dos participantes a respeito dos tipos mais comuns de falhas que podem ocorrer em aplicações de RV.

### 5.2.4 Percepções sobre falhas em aplicações de RV

O principal objetivo dessa etapa é saber dos participantes o quanto eles eram capazes de reconhecer sobre os tipos comuns de falhas que podem ocorrer em aplicações de RV. Para isso, os principais tipos de falhas de RV disponibilizados na literatura foram apresentados e os participantes foram perguntados se tinham conhecimento a respeito de algum dos tipos específicos de falhas e se, na opinião deles, tal falha era vista como crítica.

As falhas foram divididas em grupos (visual, áudio, design, inteligência artificial, física, estabilidade, desempenho e rede), utilizando a mesma classificação que foi adotada nas questões investigadas na subseção 5.2.2. Com base em tal classificação, os participantes foram questionados: *"Com relação aos tipos de falhas listados abaixo, identifique todos os tipos que você conhece e julga que são problemas críticos"*.

Como nas subseções anteriores, foi disponibilizado um documento auxiliar aos participantes com o objetivo de esclarecer e auxiliar no processo de compreensão do significado de cada uma das categorias e do tipo de característica de falha apresentada no *survey*. A Tabela 5.2.4 apresenta uma compilação dos resultados para o questionamento. As respostas foram divididas entre as classes de características avaliadas, utilizando o seguinte padrão: a coluna "Características da falha" representa o tipo de falha, a coluna "Participantes" identifica o número total de participantes que sabiam da existência dessa falha e identificou-a como crítica e, por fim, a coluna "%" apresenta o percentual de acordo com o número total de respondentes.

Tabela 8 – Tipos mais críticos de falhas segundo a visão dos participantes

| **Tipos de Falhas** | **Participantes** | **%** |
|---|---|---|
| Áudio | | |
| *Audio drops* | 31 / 88 | ≈ 35% |
| *Skiping* | 36 / 88 | ≈ 41% |
| *Distortion* | 45 / 88 | ≈ 51% |

| | | |
|---|---|---|
| *Missing sound fx* | 38 / 88 | ≈ 43% |
| *Volume too high/low* | 30 / 88 | ≈ 34% |
| Inteligência Artificial | | |
| *Does not move* | 39 / 88 | ≈ 46% |
| *Stuck (unable to move)* | 64 / 88 | ≈ 75% |
| Design | | |
| *Stuck spot* | 26 / 88 | ≈ 30% |
| *Stick Spot* | 16 / 88 | ≈ 18% |
| *Scene hole* | 37 / 88 | ≈ 42% |
| *Invisible obstacles* | 47 / 88 | ≈ 53% |
| *Missing geometry* | 43 / 88 | ≈ 49% |
| Rede | | |
| *Lag* | 66 / 88 | ≈ 77% |
| *Scoring errors* | 31 / 88 | ≈ 36% |
| *Invisible players* | 14 / 88 | ≈ 16% |
| *Cannot connect / Drop connection* | 42 / 88 | ≈ 49% |
| Desempenho | | |
| *Low frame rate* | 59 / 88 | ≈ 68% |
| *Higher loading time* | 35 / 88 | ≈ 40% |
| *High minimum requirements* | 20 / 88 | ≈ 23% |
| Física | | |
| *Object do not break* | 31 / 88 | ≈ 35% |
| *Objects floating abnormally* | 35 / 88 | ≈ 40% |
| *Problems interacting with objects* | 58 / 88 | ≈ 66% |
| *Unrealistic gravity* | 36 / 88 | ≈ 41% |
| *Impossible to pile objects* | 14 / 88 | ≈ 16% |
| Estabilidade | | |
| *Freeze* | 58 / 88 | ≈ 66% |
| *Crash* | 56 / 88 | ≈ 64% |
| *Cannot load the app* | 40 / 88 | ≈ 46% |
| *Unresponsive* | 37 / 88 | ≈ 42% |
| Visual | | |
| *Camera clipping* | 68 / 88 | ≈ 78% |

| | | |
|---|---|---|
| *Z-Fighting* | 18 / 88 | ≈ 21% |
| *Screen tearing* | 38 / 88 | ≈ 44% |
| *Missing textures* | 51 / 88 | ≈ 59% |
| *Vsible artifacts* | 28 / 88 | ≈ 32% |

Em relação às falhas relacionadas aos aspectos de áudio, é possível observar que entre 30% a 50% dos participantes apontaram características que podem ser julgadas como críticas. Falhas relacionadas a aspectos de áudio geralmente estão ligadas a problemas na criação do conteúdo, que muitas vezes só são percebidos quando o conteúdo é utilizado. Assim, os problemas, normalmente, tendem a aparecer no que as funcionalidades desenvolvidas passam a fazer uso desse tipo de recurso. Outros possíveis problemas estão relacionados à verificação da sincronia entre o som e correspondentes animações.

Dentro da categoria *áudio*, a característica de falha que obteve o maior número de votos na pesquisa foi relacionada a *distorções* no áudio. Problemas relacionados a distorções podem estar ligados ao microfone utilizado na reprodução do áudio, ao controle de volume que a aplicação faz (quando a qualidade do recurso de mídia é relativamente baixa), ou mesmo a uma interpretação errônea do problema, que pode ter origem em conexões de cabos ou botões de volume (SUTCLIFFE, 2003).

A inteligência artificial é uma extensa área de estudo que requer uma completa investigação por si só. No entanto, como o uso de mecânicas de inteligência artificial tem se tornado cada vez mais onipresente em todas as áreas da tecnologia, optou-se por analisar dois aspectos específicos no contexto de RV. As duas características avaliadas envolvem principalmente situações *pathfinding* (ou seja, problemas relacionados à navegação da origem ao destino em caminhos modelados em uma cena) e problemas em comportamentos na cena (quando uma dada ação do usuário não resulta na reação esperada) (LUCK; AYLETT, 2000).

Cerca de 75% dos participantes apontaram que a incapacidade de continuar um movimento (*Stuck*) é considerada uma falha grave. Na verdade, este é um fator decisivo, principalmente em aplicações de RV que possuem uma forte dependência desse aspecto para que o fluxo da cena possa acontecer de acordo com as interações do usuário (comumente presentes em jogos e em aplicações de treinamento).

As falhas de *design* estão vinculadas a aspectos como erros de projeção. Quando alguns componentes em uma cena não possuem uma semântica coerente. Tais falhas incluem cenários em que um usuário fica preso em algum espaço da cena (*stick spot*), ou quando não há a fluidez necessária para mover-se em algum espaço dentro da cena (*stuck spot*), além de problemas de

projeção de cena em geral (que incluem as categorias *scene hole*, *invisible obstacles* e *missing geometry*).

Para essa categoria específica de falhas, os resultados foram mistos. Poucos participantes indicaram baixa criticidade para elementos como *stick spot* (18%), problemas relacionados à projeção da cena receberam maior atenção, variando de 42% a 53% dos participantes. A questão dos aspectos visuais terem maior peso para essa característica de falhas reforça a percepção que foi observada na subseção 5.2.2, na qual grande parte dos participantes apontaram aspectos visuais como sendo fatores que podem impactar negativamente a experiência dos usuários.

Alguns aspectos relacionados a características de *rede* podem não estar diretamente vinculados à aplicação, uma vez que as aplicações normalmente esperam que a conectividade execute algum tipo específico de tarefa. No entanto, entender quais pontos os participantes julgam críticos pode ser um fator interessante para que aplicações possam fornecer mecanismos para se tornarem mais resilientes ao lidar com tal tipo de falhas.

A principal característica de falhas apontadas pelos participantes foi a presença de *lag* (77%). O termo *lag* está associado à perda de pacotes em uma conexão ou ao uso excessivo da largura de banda (BRISCOE *et al.*, 2014). A perda de pacotes pode ser um problema relacionado ao provedor de serviços de internet, mas o uso excessivo de largura de banda pode ser correlacionado à maneira como a aplicação se comunica com um servidor em cenários em que precisa consumir algum serviço para funcionar corretamente.

Os problemas de *desempenho* geralmente estão associados ao hardware, mas também podem ocultar outros problemas, como a falta de otimização computacional da aplicação (DELIGIANNIDIS; JACOB, 2005). Ambos os fatos impactam os pontos investigados nessa categoria.

Os tipos de falhas mais apontados pelos participantes foram problemas relacionados a *low frame rate* (68%). Ao projetar uma cena de RV, os desenvolvedores geralmente definem uma taxa de quadros mínima. Não ser capaz de atingir essa taxa de quadros em uma cena pode indicar que há muita informação na cena e que o hardware responsável por reproduzi-la não é capaz de atender a demanda, ou que existem gargalos de processamento e é necessário realizar tarefas de *profiling* para analisar e entender o desempenho das funções que compõem a aplicação para otimizar aquelas que possuem um maior custo computacional.

Com relação às características de falha relacionadas à *física*, há um grande desafio uma vez que a grande maioria das aplicações de RV, atualmente, exploram a utilização de *engines* que fornecem ao desenvolvedor grande parte dos recursos que envolvem a simulação de física, colisão e gravidade (ANDRADE, 2015). Ainda assim, deve haver uma preocupação em utilizar bibliotecas confiáveis e focar os esforços de teste sob o ponto de vista de verificar se a funcionalidade integrada a aplicação desempenha de forma adequada.

Dentre os tipos de falhas indicadas no *survey* que abrangem os aspectos da física em aplicações de RV, a que possui maior impacto segundo os participantes foi a de *ploblem inte-*

*racting with objects*, indicada por 66% dos participantes. Problemas relacionados à interação com objetos também podem estar ligados a um periférico que auxilia na captura de movimento, como luvas, ou com a comunicação entre os modelos e os objetos está sendo realizada. Uma compreensão clara de como o modelo de interação física se comporta é a chave para lidar com esses possíveis tipos de falhas (KUMAR; VERMA; PRASAD, 2012).

O conceito de falhas de *estabilidade* explorado no *survey* refere-se à previsibilidade da aplicação, ou seja, se ela é capaz de se comportar ou não de acordo com as intenções para as quais foi projetada. Portanto, para avaliar o quanto os participantes reconhecem falhas dessa categoria, foram apontados problemas como *freezing*, *crash* e *loading*. Todos os tipos de falhas apontados alcançaram alto grau de atenção dos participantes, variando entre 42% e 66%, sendo os dois tipos de falhas mais avaliadas relacionadas a *crash* e *freeze* (respectivamente 64% e 66%).

Falhas do tipo *crash* são consideradas graves, pois tendem a afetar a execução da aplicação, geralmente estão vinculadas ao acionamento de uma exceção na aplicação que não foi devidamente tratada e que acaba atingindo a camada mais superficial, causando um encerramento inesperado. Falhas do tipo *freeze* podem estar vinculadas à perda de informações de entrada, como o o aplicativo lida com as entradas feitas pelo usuário e reflete sobre seu comportamento na cena, ou por a aplicação entrar em laços condicionais infinitos que impedem o seu correto fluxo de execução (HAYNES, 2009).

Por fim, foi verificada a percepção com relação a falhas ligadas a aspectos *visuais*. Falhas visuais podem não ser necessariamente críticas para o funcionamento correto da aplicação, mas como aplicações de RV possuem um grande apelo na reprodução fiel de aspectos do mundo real, observar possíveis problemas visuais torna-se uma tarefa que pode determinar o sucesso de uma aplicação de RV (MACIEL *et al.*, 2017).

As três características mais citadas pelos participantes foram *screen tearing*, *missing texture* e *camera clipping*, com 44%, 59% e 78%, respectivamente. *screen tearing* é um tipo de falha visual que ocorre quando uma aplicação é incapaz de projetar um *frame* da cena na tela de forma adequada, geralmente é uma falha que pode estar relacionada a problemas de desempenho (este resultado mostra uma correlação na percepção dos participantes, uma vez que falhas relacionadas a desempenho também receberam grande nas respostas). *Missing texture* são problemas relacionados à maneira como a aplicação lida com a representação de texturas nos objetos que compõem uma cena, o que pode resultar no aparecimento de superfícies planas ou simplesmente espaços reservados. *Camera clipping* está relacionado à "detecção de colisão" do objeto da câmera, esse tipo de problema ocorre quando o objeto da câmera sobrepõe ou é sobreposto por algum outro objeto na cena.

## 5.3 Discussão

O *survey* foi originalmente guiado por três questões de pesquisa. A primeira questão de pesquisa ($RQ_1$) buscou entender a percepção dos participantes quanto à percepção de falhas ou bugs em aplicações de RV. Em geral, os participantes indicaram que a presença de falhas pode ser um fator impeditivo no processo de utilização de aplicações de RV, além de terem fornecido *feedbacks* a respeito do aspectos mais críticos dos diferentes tipos de falhas que uma aplicação de RV pode incorrer.

A percepção dos participantes indica a importância nas práticas de teste de software, visto que a atividade de teste é uma das abordagens mais utilizadas para fornecer evidências sobre a confiabilidade de um produto, fornecendo mecanismos que possibilitem a identificação e eliminação de possíveis falhas que possam ser introduzidas erroneamente durante o processo de desenvolvimento do processo de software.

Embora a RV ainda seja considerada uma fronteira relativamente nova, pesquisadores têm desenvolvido novas estratégias de teste para lidar com novos tipos de problemas. Uma das principais preocupações apontadas pelos participantes está relacionada às questões de desempenho de aplicações de RV. Chandra, Jamiy e Reza (2019) apontaram uma lista com diversos aspectos relacionados ao desempenho e usabilidade de aplicações de RV que devem ser considerados.

Para lidar com os aspectos visuais, uma possível estratégia é trabalhar com as especificações do usuário e explorar a experiência da equipe de testes. Tal processo é normalmente conduzido pelos requisitos de escopo do produto e isso permite que a equipe de teste entenda os cenários potenciais para o envolvimento do usuário (CORREA; NUNES; DELAMARO, 2018). No entanto, considerando que o tempo de desenvolvimento e, consequentemente, o tempo que pode ser aplicado à atividade de teste de software, é limitado, é necessário compreender os pontos críticos que podem ser usados para priorizar as atividades em uma escala de importância (DUSTIN; GARRETT; GAUF, 2009).

A segunda questão de pesquisa ($RQ_2$) focou na compreensão da percepção dos participantes com relação às características de falha em aspectos de hardware de RV. Os resultados apresentados na subseção 5.2.3 indicaram relevância para todos os aspectos avaliados, com destaque para recursos vinculados a sensores que são responsáveis por complementá-la como aplicação de RV na medição de movimento, direção no espaço, , além de dispositivos relacionados a tarefas de entrada de dados.

O uso contínuo de dispositivos de RV pode trazer consequências físicas, como dores de cabeça, enjoo e cansaço visual (DAVIS; NESBITT; NALIVAIKO, 2014). Um grande desafio é encontrar uma maneira de minimizar a fadiga e o desconforto do usuário ao usar uma aplicação. Para lidar com esse tipo de problema, um grande esforço é feito em testes de usabilidade (SUTCLIFFE; KAUR, 2000). Como os dispositivos e aplicações de RV ainda estão em fase

constante de atualização, atualmente ainda não há soluções definitivas que possam ser adotadas por desenvolvedores além de conjuntos de boas práticas.

O fato de que mesmo pequenos tipos de falhas podem ser consideradas importantes em aplicações de RV, devido ao grande apelo visual e de imersão que essas aplicações proporcionam (MOREAU, 2013), motivou a busca por entender a visão dos participantes sobre diferentes tipos de falhas que podem afetar tal tipo de aplicações ($RQ_3$). Os resultados apresentados na subseção 5.2.4 fornecem evidências para entender melhor como os participantes do estudo avaliaram os tipos específicos de falhas em relação à sua criticidade, além de discutir como as principais falhas apontadas são manifestadas no contexto de aplicações de RV.

As informações apresentadas podem auxiliar no contexto do ponto de vista de teste de software, pois esses dados podem subsidiar o processo de tomada de decisão no que diz respeito a priorizar a validação de pontos específicos de uma aplicação, permitindo estratégias de equilíbrio, com foco na equalização das necessidades do projeto (ARULDOSS; LAKSHMI; VENKATESAN, 2013), levando em consideração tanto o ponto de vista de uma equipe de qualidade de software quanto o do restante da equipe, incluindo os usuários finais.

## 5.4 Ameaças à Validade

Os resultados apresentados no decorrer deste capítulo estão sujeitos a ameaças à validade e, portanto, devem ser interpretados com cautela. Abaixo são discutidas as principais preocupações com relação à validade dos resultados do *survey* com base na classificação proposta por Wohlin *et al.* (2012).

Com relação a possível viés nas respostas por parte dos participantes, não foram identificadas evidências mínimas de viés de resposta. Uma vez que o *survey* seguiu um modelo anônimo de respostas e todos aqueles que forneceram respostas incompletas foram descartados, o viés de resposta só pode ser estimado de forma limitada.

Ameaças à validade interna estão relacionadas à diversidade e representatividade das amostras coletadas. Para evitar essa ameaça, inicialmente o *survey* foi distribuído em uma conferência científica de especialistas em RV, possibilitando obter respostas dos mais diversos perfis que compõem a área de RV e posteriormente o estudo foi complementado distribuindo-o de forma online para atingir um público de tecnologia, de tal forma a tentar garantir que os resultados que pudessem representar o maior número de perfis envolvidos no contexto deste tipo de aplicações.

Com relação a ameaças a validade de constructo, que, no contexto deste estudo, se preocupa com possíveis problemas sobre o quão bem tudo o que se pretendia medir foi realmente medido. Todos os dados apresentados no estudo correspondem à avaliação agregada de todas as respostas, sem avaliação de juízo de valor. Buscou-se apresentar os dados das questões do

*survey* sob a ótica de diferentes tipos de gráficos e discutir quais as possíveis desdobramentos dos resultados apresentados. Para garantir a confiabilidade do estudo, todo o material produzido, bem como as respostas estão publicamente disponíveis para consulta e validação (ANDRADE *et al.*, 2021).

A fim de garantir a confiabilidade do *survey* (validade da conclusão), adotou-se cuidadosamente o processo proposto por Linåker *et al.* (2015), que apresenta diretrizes para condução de *surveys* para o contexto de engenharia de software. Além disso, todos os recursos e dados disponíveis no repositório do estudo permitem a revisão dos dados por terceiros, e todos os resultados foram interpretados e discutidos pelos autores do trabalho para evitar possíveis erros de interpretação.

Por fim, para evitar ameaças à validade externa, reforça-se que os resultados desse *survey* refletem, única e exclusivamente, a interpretação dos participantes a respeito da necessidade de práticas de qualidade de software no contexto de aplicações de RV. As respostas podem não representar necessariamente uma imagem precisa da realidade e, portanto, podem apresentar algum grau de subjetividade. Aspectos como a experiência, bem como o grau de contato com tecnologias de RV podem ter influenciado as respostas. Para evitar essa ameaça, buscou-se entrevistar o maior número possível de interessados, além de evitar análises de correlação de dados para evitar um viés interpretativo do ponto de vista dos pesquisadores.

## 5.5 Considerações Finais

Este capítulo relatou os resultados de um *survey* que teve como objetivo compreender a percepção de um grupo de *stakeholders*, inseridos no contexto de RV, quanto às necessidades de práticas de qualidade de software no domínio da RV. Foram levantadas as principais preocupações relacionadas ao perfil dos participantes do *survey*, levando em consideração aspectos de software e de hardware deste tipo de aplicação, a fim de entender a percepção dos participantes a respeito daquilo que deveria ser priorizado.

Os resultados relatados nesse capítulo estão disponíveis publicamente (ANDRADE *et al.*, 2020; ANDRADE *et al.*, 2021). Apesar do *survey* não apresentar dados suficientes para que os resultados apresentados pudessem ser amplamente generalizados, os resultados deste capítulo possibilitaram indicar hipóteses de percepções de diferentes pontos de vista: sob a perspetiva de uma equipe de garantia da qualidade e sob uma ótica de área de pesquisa.

Ao olhar os resultados sob a perspetiva de uma equipe de garantia da qualidade, responsável pelo desenvolvimento de aplicações de RV, os resultados permitem perceber como equilibrar as atividades para atingir tanto as expectativas de desenvolvimento do projeto, quanto às expectativas dos atores envolvidos no uso final de tais aplicações.

Sob o ponto de vista de uma área de pesquisa o estudo permitiu aos entusiastas explorar

a percepção dos participantes do estudo de forma a direcionar novas técnicas e abordagens para aspectos indicados como importantes, possibilitando pesquisas futuras com menos viés do ponto de vista do pesquisador e mais alinhados às necessidades do utilizador.

CAPÍTULO

6

# UMA ABORDAGEM DE TESTE DE SOFTWARE PARA APLICAÇÕES DE REALIDADE VIRTUAL

Técnicas para explorar a geração automática de dados de teste (EDVARDSSON, 1999; MCMINN, 2004; GALLER; AICHERNIG, 2014; DURELLI *et al.*, 2019) e para lidar com o problema do oráculo de teste (HARMAN *et al.*, 2013; BARR *et al.*, 2015; JAHANGIROVA, 2017) têm sido amplamente investigadas na literatura. Tais estudos têm indicado que, quanto mais complexo o domínio do software, maior a dificuldade para gerar dados de entrada e também para verificar o comportamento do mesmo.

Conforme discutido no Capítulo 3, existe um crescente interesse quanto à proposição de abordagens de teste de software para o contexto de RV. Contudo, considerando a inexistência de soluções ótimas, cada trabalho busca explorar diferentes tipos de abordagens para lidar com os desafios da tarefa. Uma questão importante a ser explorada nesse contexto é como possibilitar a avaliação de dados de testes gerados automaticamente em conjunto com estratégias que permitam a verificação do comportamento da aplicação de forma automatizada.

Nesse cenário, conforme apresentado nos estudos discutidos nos Capítulos 4 e 5, os desafios enfrentados no processo de adoção de práticas para automatização de técnicas de teste para RV ainda permanecem como um obstáculo para popularização de abordagens de teste para o domínio. Em contrapartida, a popularização de abordagens que utilizam aprendizado de máquina para solucionar problemas no contexto de engenharia de software tem crescido ano a ano (MEINKE; BENNACEUR, 2018) e a popularização de novas soluções como algoritmos e bibliotecas que auxiliam na resolução de diferentes tipos de problemas tem permitido que entusiastas explorem, sob uma nova perspectiva, desafios que ainda não foram completamente resolvidos no contexto de engenharia de software.

De maneira semelhante às abordagens mencionadas na seção 3.3, a nova abordagem

apresentada neste capítulo pode ser vista como uma técnica de teste para aplicações de RV. Contudo, a abordagem aqui descrita sugere que a atividade de teste de software para esse domínio pode ser beneficiada com a utilização de testes metamórficos e a utilização de agentes para geração automática de dados de teste.

Para avaliar essa ideia, foi projetado um ambiente de execução gerenciado adicionando suporte para a utilização de testes metamórficos no contexto de RV e foi adotada a plataforma *Unity* para o desenvolvimento da abordagem, uma vez que é a mais utilizada em pesquisas relacionadas sobre o tema (GHRAIRI *et al.*, 2018).

As novidades dessa abordagem não são, propriamente, as tecnologias e/ou ferramentas utilizadas que, de certa forma, possuem ampla base de pesquisa acadêmia e utilização na indústria, mas sim sua integração em um ambiente de execução em conjunto.

Este capítulo está organizado da forma: a seção 6.1 é a seção principal do capítulo, e nela são descritos os detalhes e a estrutura da prova de conceito apresentada, como a mesma operacionaliza a parte de interação da abordagem dentro de um ambiente virtual, como são estruturadas as relações metamórficas que avaliam a conformidade das propriedades investigadas nas aplicações de RV, descreve a formulação do algoritmo de AM utilizado para auxiliar no processo de geração de dados para os agentes, além de apresentar como é realizado o processo de identificação de falhas e o registro dos dados de teste responsáveis por levar a aplicação a um estado inconsistente; a seção 6.2 descreve de forma resumida a estrutura técnica da prova de conceito, destacando os principais módulos existentes na abordagem, a finalidade de cada um deles, além de tecnologias utilizadas e, por fim, as considerações finais são apresentadas na seção 6.3.

## 6.1 Prova de Conceito: Utilizando Testes Metamórficos e Aprendizado por Reforço para Testar Aplicações de RV

De uma forma geral, a abordagem consiste na utilização de testes metamórficos para monitorar e verificar a consistência de propriedades da aplicação em teste por meio da utilização de agentes inteligentes. Os agentes podem originalmente fazer parte da aplicação RV (como personagens ativos em jogos) ou podem ser incluídos exclusivamente com o propósito de se testar um conjunto específico de propriedades na aplicação.

Nos capítulos anteriores foi observado que existe uma série de características existentes em grande parte das aplicações de RV que costumeiramente podem apresentar defeitos e, por consequência, podem causar impacto negativo na experiência do usuário desse tipo de aplicação. Características como: aspectos de design, visual da cena, ajuste da câmera, controle da física e desempenho, que são recursos presentes na maioria de tais aplicações e, portanto, são

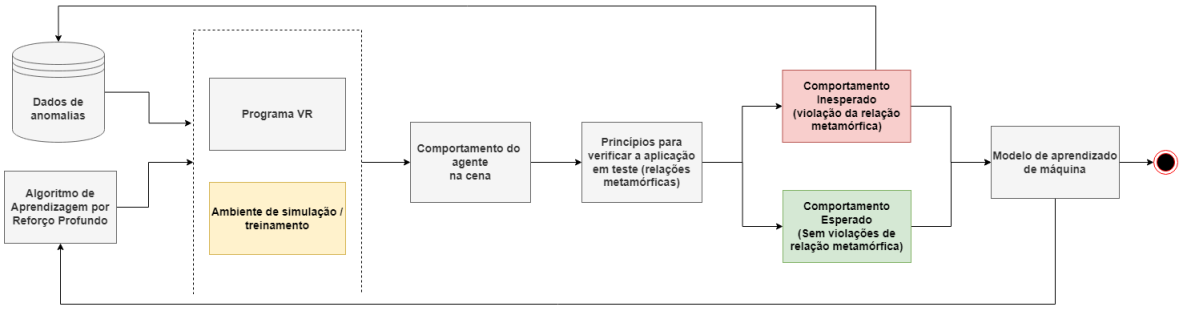
considerados fundamentais para que o usuário tenha uma experiência adequada ao utilizar esse tipo de aplicação (MURCIA-LÓPEZ *et al.*, 2020).

A hipótese abordada neste capítulo é, portanto, a utilização de agentes inteligentes para possibilitar a verificação de tais características sem que seja necessário ter um conhecimento aprofundado a respeito dos requisitos que compõem a aplicação em teste, uma vez que tais propriedades tendem a fazer parte do escopo de grande parte das aplicações desenvolvidas para esse domínio.

Considere uma aplicação RV para a qual deseja-se verificar o comportamento da propriedade de colisão de uma série de objetos dispostos na cena (*targets*) em relação a um outro dado objeto (agente). Sob o ponto de vista de teste de software o principal desafio para essa operação seria fazer com que fossem gerados dados de teste capazes de fazer com que o agente se movesse em direção a cada um dos *targets* especificados, de tal forma que fosse possível verificar o comportamento esperado do agente ao colidir com tais *targets*.

Em um cenário ideal, a depender do número de *targets* a serem verificados, essa é uma tarefa que pode ser bastante demorada se realizada manualmente, ou bastante desafiadora se o testador ficar a cargo de criar um *script* de teste específico para essa interação. Outro ponto a ser destacado se dá com relação à forma em que a tarefa é cumprida, uma vez que, conforme mencionado no Capítulo 2, devido à natureza das aplicações de RV, que possuem fortes características de interação em tempo real, uma mesma operação pode ser realizada de diferentes formas por um mesmo utilizador e, portanto, é um desafio estabelecer como definir um conjunto de dados capaz de representar o comportamento e, de forma similar, atestar se o comportamento da aplicação aconteceu de acordo com o esperado.

Figura 17 – Arcabouço operacional das etapas realizadas durante a abordagem




Fonte: Elaborada pelo autor.

A Figura 17 fornece uma visão geral dos passos envolvidos na abordagem proposta neste capítulo. De uma forma geral, um algoritmo de AR é responsável por gerar dados de entrada para um agente que será treinado para realizar um conjunto de tarefas que consiste em verificar determinadas propriedades de uma aplicação de RV. A cada ciclo de treinamento do agente (*episódio*), a correção da propriedade testada é verificada por meio de relações metamórficas, que quando violadas, os dados gerados naquele ciclo de treinamento são gravados para posterior

verificação. O processo ocorre de forma iterativa até que seja encerrado manualmente, ou atinja o número limite pré-estabelecido de *steps*.

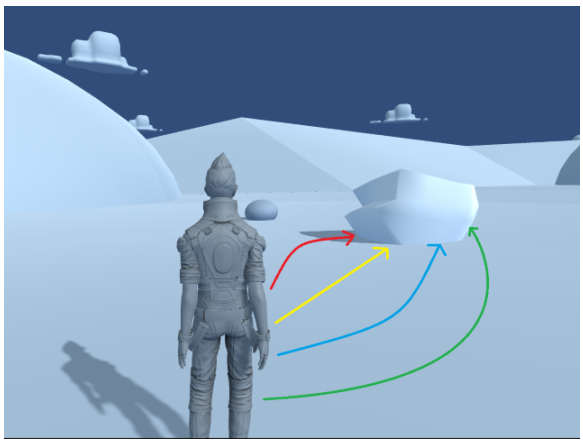
Nas subseções a seguir são descritos com mais detalhes os diferentes componentes da abordagem.

### 6.1.1 Interação com os ambientes virtuais

Aplicações de RV possuem características muito distintas quando comparadas aos demais tipos de aplicações, entre elas, uma das características que mais dificulta a atividade de automatização de testes é a característica de interação com o ambiente virtual. Para que o usuário possa completar uma dada tarefa em um ambiente virtual, as entradas do usuário reagem em função do *feedback* apresentado pela cena (interação em tempo real), dificultando portanto uma capacidade de reprodução fiel, uma vez que uma simples entrada diferente pode alterar completamente o comportamento de algum componente na cena e, consequentemente, guiar o usuário para o próximo conjunto de entradas até completar a sua tarefa. Dessa forma, para aplicações de RV há um número infinito de possíveis dados de entrada que podem ser gerados a fim de cumprir uma dada tarefa.

Portanto, mesmo para tarefas simples, como mover um objeto em direção a outro (representado na Figura 18), não existe uma solução correta única e há um número infinito de dados de entrada possíveis que podem ser gerados a fim de cumprir a tarefa.

Figura 18 – Exemplo simples de possíveis soluções para uma tarefa de deslocamento em um ambiente virtual.



Fonte: Elaborada pelo autor.

De acordo com LaViola *et al.* (2017) as técnicas de interação em ambientes virtuais são categorizadas em: navegação, seleção e manipulação, controle de sistema e entrada simbólica. Como a abordagem descrita neste capítulo depende do conceito de agentes para interagir com a cena no aplicativo de RV, optou-se por utilizar os conceitos relacionados às regras de navegação.

A navegação é considerada a forma mais comum de interação e consiste na movimentação do usuário ou participante da cena. Normalmente as técnicas de interação de navegação são classificadas em duas subcategorias: (i) ***travel***, em que o conceito está relacionado ao movimento livre sob o ponto de vista do usuário dentro da cena; e (ii) ***wayfinding***, que pode ser entendido como um processo de utilização da habilidade espacial do usuário e da percepção humana em um determinado ambiente, com o objetivo de encontrar uma dada localização (LAVIOLA *et al.*, 2017).

Ao utilizar conceitos de agentes inteligentes é possível absorver um pouco das duas definições, visando propor um mecanismo capaz de guiar os procedimentos que o agente deve realizar dentro da cena, mas sem restringi-lo da capacidade de explorar o ambiente de forma livre. Dessa maneira, é possível garantir que o agente terá a liberdade necessária para explorar aspectos gerais que compõem a cena em teste sem qualquer tipo de restrições, mas, em contrapartida, são oferecidas indicações que auxiliem o agente a cumprir a tarefa necessária para verificar a correção da propriedade que deseja-se verificar na aplicação.

Para isso, a abordagem utiliza algoritmos de *pathfinding* para ajudar a traçar rotas entre o agente e o seu objetivo de teste, de acordo com a propriedade que deseja-se verificar. Para cada objetivo de teste, utiliza-se um algoritmo A* (LESTER, 2005) para traçar uma rota com caminho ótimo para guiar o agente em direção à atividade que o mesmo deve realizar, a fim de verificar a propriedade em teste.

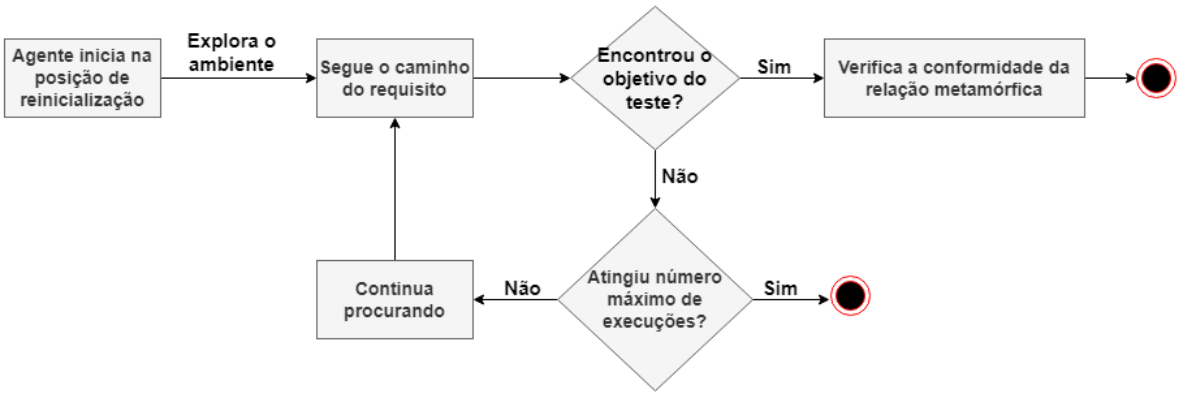
Optando pela abordagem de utilizar agentes inteligentes, pretende-se eliminar a necessidade de existir um conhecimento prévio a respeito dos requisitos funcionais da aplicação (MALAN; BREDEMEYER *et al.*, 2001), que são considerados fatores limitantes para a generalização de uma abordagem de teste, uma vez que, para o contexto de aplicações de RV, é estritamente necessário entender o comportamento do mundo virtual em teste para que se possa verificar aspectos funcionais e características de usabilidade e as suas propriedades (CHANDRA; JAMIY; REZA, 2019).

Além disso, com essa abordagem é possível explorar soluções que representam, em parte, o comportamento do usuário, uma vez que os usuários possuem tendências e padrões de uso que dificilmente são replicados em cenários convencionais de teste. Portanto, o objetivo final é incorporar a abordagem ao processo de teste com o propósito de testar a aplicação de maneira similar a como o usuário interage com ela, buscando, dessa forma, localizar e remover possíveis falhas que apareceriam apenas quando a aplicação já estivesse nas mãos do usuário (FOTROUSI, 2020).

### 6.1.2 Relações metamórficas para aplicações de RV

A abordagem de teste é baseada na definição do conjunto de propriedades (relações metamórficas) que devem permanecer invioláveis durante a execução de uma tarefa modelada

Figura 19 – Comportamento do agente no ciclo de um *episódio* regido por uma relação metamórfica




Fonte: Elaborada pelo autor.

para o agente. Em caso de violação da propriedade, diz-se que a aplicação gerou uma falha e o estado da aplicação, bem como o conjunto de dados gerado até ao momento da violação, é guardado para que seja possível verificar detalhadamente a causa.

Caso o agente seja capaz de cumprir a sua tarefa, em termos de AR ele terá cumprido um *episódio* e os dados gerados para a conclusão de tal tarefa serão considerados como um conjunto de dados de teste válidos para aquele cenário.

A Figura 19 exemplifica os passos do processo. O agente é responsável por explorar o ambiente com base em um caminho de requisitos (gerado por um algoritmo de *pathfinding*) que representa uma possível abordagem de interação do agente com o ambiente, até que encontre o objetivo do teste (a propriedade que deve ser verificada). A relação metamórfica se encarregará de identificar se o comportamento desempenhado pelo agente viola a propriedade verificada durante a execução do processo.

Como definir um conjunto adequado de relações metamórficas ainda é um problema aberto (CHEN *et al.*, 2018). Foram projetadas relações metamórficas baseadas a partir do entendimento de grupos de interesse sobre alguns dos problemas mais comuns em aplicações de RV (Capítulo 5), de forma que não estão vinculadas a nenhuma estratégia particular.

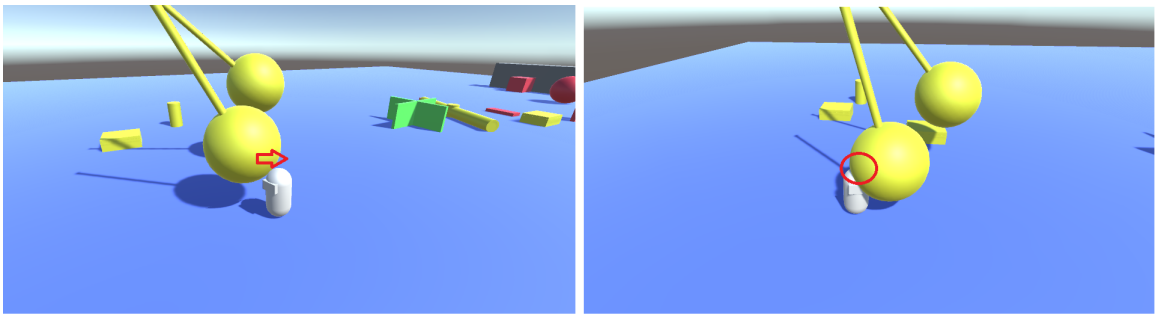
#### *6.1.2.1 Colisões*

A primeira propriedade observada é a detecção de colisão entre objetos, uma vez que essa propriedade está relacionada com a física de componentes de uma cena e por meio dela são disparados gatilhos para a realização de funcionalidades específicas de uma aplicação de RV (CAPELLMAN; SALIN, 2020). Portanto, a dispêndio da correta implementação dos requisitos funcionais de uma aplicação, caso haja problemas na detecção de colisões esse problema pode propagar para a funcionalidade, ocasionando uma falha na aplicação.

Em linhas gerais, para possibilitar a verificação do correto funcionamento da colisão de

objetos em uma cena, é necessário criar uma relação metamórfica capaz de verificar os eventos responsáveis por monitorar a existência de colisões em uma aplicação de RV. Para esse propósito, os objetos a serem monitorados devem ser individualmente identificados (por meio de uma *tag* de identificação) e o objetivo de teste deve ser definido em função de uma tarefa a ser desempenhada pelo agente inteligente (por exemplo, tentar colidir com o objeto alvo).

Figura 20 – Exemplo de problema de detecção de colisão em uma cena




Fonte: Elaborada pelo autor.

A Figura 20 representa, de forma simplificada, dois cenários: na Figura 20a, à esquerda, ao colidir com o pêndulo amarelo, um agente, representado por um cilindro branco, é arremessado para a direita, conforme o comportamento esperado para a física de colisão entre os dois objetos na cena; na Figura 20b, à direita, há a representação de uma falha nesse processo, devido a ausência de um componente responsável por possibilitar a representação do pêndulo como um objeto colidível. Nesse sentido, ao entrar em contato com o cilindro o pêndulo passa a sobrepô-lo, contrariando o comportamento esperado de forçar o deslocamento do mesmo para a esquerda.

Problemas na detecção de colisões podem ocorrer por diferentes motivos: *(i)* pela ausência da propriedade de colisão em um objeto, *(ii)* quando há uma grande complexidade na malha que representa o objeto, o que acarreta em um alto custo computacional para detecção da colisão, *(iii)* em cenários nos quais são utilizadas primitivas geométricas que não se adéquam corretamente ao formato do objeto, ou ainda *(iv)* pela complexidade em calcular a colisão quando são utilizadas múltiplas primitivas geométricas para detectar a colisão em um único objeto (WELLER, 2013).

Para avaliar essa propriedade, foram aproveitadas características próprias de aplicações de RV, que seguem uma representação de organização hierárquica no formato de um grafo de cena (Seção 2.5). Utilizando essa característica é possível, em tempo de execução, criar um objeto *foo* como *placeholder* e incluí-lo como um filho na hierarquia do objeto que é definido como objeto em teste ( *target* - aquele do qual a propriedade deseja-se verificar) e passa-se a monitorar o comportamento desse objeto *foo* para entender se o comportamento do objeto *target* está funcionando de acordo com o esperado.

Uma vez que espera-se verificar o comportamento da colisão do objeto pai, uma colisão

com o objeto filho jamais deverá acontecer, a menos que haja algum problema de colisão com o objeto pai, portanto, com base no *template* proposto por Segura *et al.* (2017), a relação metamórfica para verificar essa propriedade é definida como:

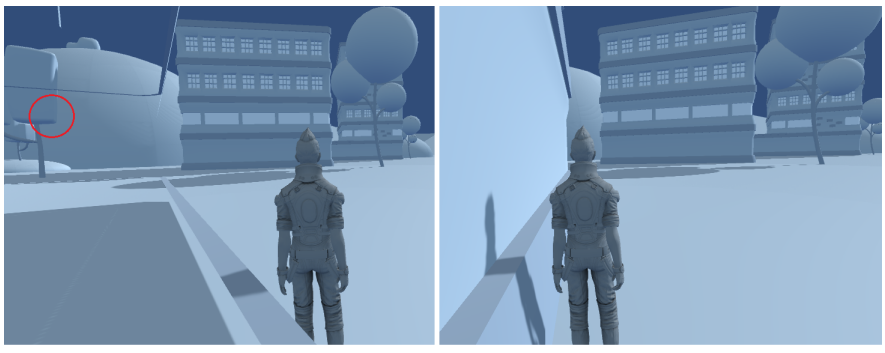
- ***se*** a propriedade de colisão de um objeto *foo* não é detectável por esse objeto ser filho de um objeto pai que também possui uma propriedade de colisão.
- ***então*** o resultado da inclusão de novos dados de teste em um *follow-up* jamais fará com que essa colisão seja detectada, uma vez que a colisão com o objeto pai (*target*) sempre deverá ser precedida à colisão com o objeto filho (*foo*).

Uma vez que é utilizada uma abordagem de AR profundo para gerar os dados que serão consumidos pelo agente para cumprir a tarefa de testar uma dada propriedade, cada nova interação do agente com o ambiente é considerada como um novo conjunto de dados de teste (*follow-up*) em relação ao estado inicial da cena (*source*).

#### 6.1.2.2 Câmera

A câmera é um componente fundamental em uma cena de uma aplicação RV e é responsável por representar o comportamento visual do usuário da aplicação. A câmera tem como objetivo principal tentar preservar a linha de visão original do usuário em relação aos objetos que compõem a cena. Problemas de oclusão da câmera podem ocorrer, por exemplo, quando a mesma está posicionada muito próxima a um objeto e a mesma não sabe como lidar com a representação desse objeto, gerando um problema no qual o ângulo de visão interno do objeto obstrui o campo de visão do usuário da aplicação.

Figura 21 – Exemplo de problema de oclusão da câmera em uma cena




Fonte: Elaborada pelo autor.

A Figura 21 apresenta um cenário representando o problema descrito. Ao lado esquerdo, Figura 21a, é possível observar que ao sobrepor o posicionamento da câmera em relação a um objeto, a câmera não sabe como lidar com a situação e acaba exibindo ao usuário partes internas

do objeto (vazio), bem como objetos que deveriam estar oclusos do campo de visão da câmera (destacado em vermelho) por conta do objeto que os sobrepõem no campo de visão. A imagem à direita, Figura 21b mostra a correta visualização quando o objeto da câmera não sobrepõe a posição de um objeto na cena.

Dessa forma, colisões de câmeras têm o propósito de evitar que a câmera "corte"os objetos e mostre detalhes na cena que não deveriam ser originalmente visíveis. No entanto, colisões de câmera podem parecer atrapalhar a usabilidade de uma aplicação em certos aspectos. Colidir com todos os objetos de uma cena poderia implicar em má experiência do usuário com a aplicação. Um grupo de objetos em uma área pequena pode fazer com que a câmera se mova ao tentar se ajustar às superfícies variáveis dos objetos (CHRISTIE; OLIVIER; NORMAND, 2008).

Nesse sentido, várias aplicações solucionam esses problemas eliminando totalmente colisões de câmeras com objetos, esmaecendo os objetos à medida que esses entram em contato com a câmera, ou reajustando automaticamente o posicionamento da câmera para evitar a sobreposição entre os objetos na cena. As colisões de câmeras seriam então reservadas apenas para áreas de solo e as paredes do ambiente (OSKAM *et al.*, 2009).

A fim de identificar esse tipo de problema, foi utilizada a mesma abordagem baseada em *template* para descrever a relação metamórfica para esta propriedade:

– ***se*** a correta visualização gerada por um objeto do tipo câmera em uma cena RV depende do objeto não sobrepor outros objetos na cena à medida que esse se desloca.

– ***então*** o resultado da inclusão de novos dados (*follow-up*) jamais fará com que o posicionamento do objeto da câmera sobreponha a posição de algum outro objeto. Uma vez que o objeto câmera deve ajustar o seu posicionamento de acordo com outros objetos existentes na cena.

A ideia por trás dessa relação metamórfica é explorar os limites existentes em objetos relativamente grandes, como chão, ou paredes, e avaliar se o comportamento do objeto “câmera” está correto quanto à sua organização para não sobrepor elementos na cena e apresentar problemas como o especificado na imagem descrita anteriormente.

### 6.1.3 Formulação do algoritmo de aprendizado por reforço

A geração de dados de teste para agentes pode ser representada como um problema de pesquisa no qual há um espaço de entrada $S$, que contém um número infinito de combinações de valores de entrada e o objetivo é gerar um conjunto de entradas válidas $s' \subset S$, que é capaz de desencadear a relação metamórfica que está sendo avaliada.

Na abordagem proposta, pretende-se resolver esse problema permitindo que o agente explore efetivamente diferentes combinações de dados de entrada no ambiente, a fim de tentar maximizar a recompensa obtida na cena.

O agente realiza um processo contínuo em que são criados novos dados de teste por meio da exploração e percepção do agente sobre o ambiente. O ambiente responde aos estímulos realizados pelo agente, produzindo o efeito da interação em tempo real (característica essencial desse tipo específico de aplicações). As recompensas recebidas pelo agente são utilizadas como fonte de conhecimento para a escolha de novas interações (geração de novos dados de teste).

Durante o processo de geração de dados, é criado um filtro entre o processo de geração de dados do algoritmo de AR e a execução dos dados na aplicação pelo agente. Dessa forma é possível manter um controle sobre os dados gerados e salvá-los nos casos em que as relações metamórficas são violadas, possibilitando a reexecução do cenário de teste e facilitando um processo de depuração.

A seguir, são definidos os componentes do processo de AR inseridos na abordagem, incluindo a representação de estado ($S$), domínio de ações ($A$), função de recompensa ($R$) e política do agente ($\Pi$).

#### 6.1.3.1 Representação dos estados

O domínio de possíveis estados $S$ é um conjunto de todos os estados para os quais um agente pode fazer a transição. Em outras palavras, um estado $s' \in S$ representa uma combinação de valores de entrada que o agente pode obter ao desempenhar um comportamento na cena.

Por exemplo, para testar a propriedade de colisão de um objeto, é traçada uma rota ideal por meio de um algoritmo de *pathfinding* e o agente tem como objetivo final chegar ao objeto alvo com base no apoio, em forma de recompensas, oferecido por essa rota. As ações tomadas pelo agente são realizadas com base no domínio de ações $A$.

#### 6.1.3.2 Domínio de ações

Essa propriedade pode variar de acordo com o objetivo do teste. Por exemplo, se a aplicação já possui um componente que pode ser utilizado como agente, ou se será utilizado um componente genérico para tal função. Caso a aplicação possua tal componente, o espaço de ações deverá ser representado com base nas funcionalidades desse componente e precisa ser reimplementado para definir o seu comportamento adequado e portanto pode variar.

De forma geral, o agente deve observar o estado atual da cena ($s'$) e executar uma ação. Com base na ação $a'$, a cena será modificada para o estado $s'$ para produzir o próximo estado $s'+1$ (ou seja, uma nova percepção da cena com base na entrada valores do agente).

Inicialmente temos uma lista de cinco operações possíveis a serem aplicadas ao agente ao longo dos eixos $x$ e $z$ simultaneamente (*não fazer nada*, *mover para frente*, *mover para trás*,

*virar à esquerda*, *virar à direita*), o espaço de ação é definido como contínuo. Em outras palavras, os valores passados para o modelo são valores reais, que podem oscilar no intervalo [-1, 1].

#### 6.1.3.3 Observações

Na abordagem proposta o agente possui sensores de raio para percepção, 6 por direção totalizando 12 raios em torno do agente e tais raios são capazes de perceber o objeto tratado como objetivo de teste (*target*) e objetos criados para auxiliar no processo de direcionamento de rota (que são utilizados como parte do modelo de recompensas).

Além dos raios para percepção do ambiente, o agente também observa a posição do objeto objetivo de teste (*target*) em relação à sua posição atual, a velocidade com a qual o agente está se movendo e a direção que o agente está tomando. A ideia por trás de observar tais valores é garantir que, para cada ação, o agente seja capaz de ter um efeito claro e facilmente interpretável com relação à sua distância e direção em relação ao alvo (objetivo do teste).

Portanto, a cada atualização da política, o agente observa 12 valores referentes aos raios, 3 valores referentes à posição do alvo (*x, y, z*), 3 valores referentes à velocidade de deslocamento do agente em relação aos seus eixos (*x, y, z*) e 3 valores em relação à direção que o agente está tomando (*x, y, z*), totalizando 21 observações a cada ação tomada e atualização da política do agente.

#### 6.1.3.4 Política

A abordagem utiliza uma política denominada de *Proximal Policy Optimization* (PPO) (SCHULMAN *et al.*, 2017), que é uma classe de algoritmos de otimização e é implementada no *PyTorch* (PASZKE *et al.*, 2019) (ferramenta utilizada na abordagem). O algoritmo proposto pela abordagem PPO lida com problemas de desempenho, em aprendizado por reforço, nos cenários em que um agente repentinamente começa a ter um mau desempenho, levando a geração de ações ruins que são então usadas para treinar ainda mais a política, comprometendo a qualidade final do modelo gerado.

Além disso, o PPO utiliza redes neurais para aproximar a função ideal que mapeia as observações de um agente para a melhor ação que um agente pode realizar em um determinado estado. A principal contribuição dessa abordagem é garantir que uma nova atualização da política não a altere muito em relação à política anterior. Isso leva a uma menor variação no treinamento ao custo de algum viés, mas garante um treinamento mais suave e também garante que o agente não siga para um caminho irrecuperável de tomar ações “sem sentido”, garantindo que, se o agente aprender com uma boa função de recompensa, ele tenha um boa probabilidade de realizar sua tarefa com sucesso (HSU; MENDLER-DÜNNER; HARDT, 2020).

#### 6.1.3.5 Função de recompensa

O agente aprende a resolver uma tarefa por meio da exploração do ambiente, o *feedback* que o agente recebe é em forma de uma recompensa que é o valor utilizado para atualizar a política e definir as melhores ações a serem tomadas pelo agente. Dessa forma, o objetivo do agente é sempre tentar maximizar a recompensa recebida garantindo a escolha das melhores ações para atingir o objetivo definido. Portanto, uma função de recompensa deve ser definida para guiar o agente para boas soluções para um determinado problema.

O objetivo é encontrar dados de entrada capazes de fazer com que o agente atinja o objetivo de teste e, portanto, verifique a propriedade da aplicação em relação a uma dada relação metamórfica. A lógica é usar o conceito de *pathfinding* para criar objetos *foo* na cena que servem como uma regra de navegação que o agente deve seguir. À medida que o agente entra em contato com tais objetos, ele deve receber recompensas parciais ($r_p$), de modo que ao final do caminho o agente seja capaz de verificar a relação metamórfica e alcançar a recompensa ao completá-la ($r_e$). Portanto, é possível definir a função de recompensa como:

- $r_p = \Sigma_n^{1/n}$, onde $n$ é igual ao número de objetos criados para projetar o caminho para o objetivo do teste e a soma se refere ao número de objetos “coletados"pelo agente;
- $r_e = x \in [0,1]$, dependendo se o agente foi capaz de completar efetivamente a tarefa *episódio*;
- $r = r_p + r_e$

Uma recompensa positiva sinaliza ao agente que ele executou uma boa ação no estado fornecido. Portanto, o agente aumentaria a probabilidade de selecionar aquela ação no futuro para aquele estado. Ao receber a recompensa pelas ações realizadas no ambiente e atualizar a política com base nas observações, o agente melhora no cumprimento da tarefa atribuída à medida que os *episódios* passam durante a fase de treinamento do modelo.

### 6.1.4 Encontrando falhas

O agente é treinado com o auxílio de um *framework* para desenvolvimento de agentes (*ML-Agents* (JULIANI *et al.*, 2018)). O treinamento ocorre na forma de *episódios*, nos quais durante cada *episódio*, o agente visa verificar uma determinada relação metamórfica em um número máximo de etapas. A cada etapa, o agente toma uma decisão com base no conjunto de ações definidas no espaço de ação e, em seguida, a recompensa é calculada com base na percepção observada no ambiente. As observações são coletadas e utilizadas para atualizar a política por meio de uma *Deep Neural Network* (DNN) (FRANÇOIS-LAVET *et al.*, 2018). Após o agente completar o número estipulado de passos ou completar com sucesso o *episódio*, um novo *episódio* é iniciado até que o número definido de *episódios* seja alcançado.

As relações metamórficas são verificadas constantemente a cada *frame* e caso violadas, o conjunto de dados gerados para atingir tal estágio é salvo para possibilitar uma avaliação posterior.

### 6.1.5 Registrando dados de teste

Para possibilitar salvar os dados gerados sempre que uma relação metamórfica for violada, foi criado um filtro para garantir a possibilidade de monitorar os dados que são gerados antes que eles sejam executados na aplicação em teste. De forma simples, esse filtro funciona como um *singleton* (SARCAR, 2020) e todos os dados de ação de um agente passam por tal filtro antes da sua execução.

O filtro segue um padrão *publish-subscribe* (EUGSTER *et al.*, 2003) e registra um evento implementado pelas relações metamórficas. Sempre que uma relação metamórfica é violada, esse evento é disparado e, portanto, o filtro é notificado e sabe que os dados de entrada gerados até aquele momento devem ser salvos para análises posteriores. A cada *episódio* os dados de entrada são gravados ou descartados e uma nova entrada será gerada em sequência até que o número de *episódios* seja alcançado.

Os dados são agregados em uma estrutura de dados no formato de uma pilha. Essa estrutura permite que posteriormente os dados possam ser reexecutados em ordem, para processo de depuração, em um mecanismo de *rewinding*, possibilitando explorar a reprodução dos dados gravados com maior facilidade na cena.

## 6.2 Estrutura da Prova de Conceito

A estrutura da abordagem descrita na seção anterior foi implementada no formato de um protótipo para prova de conceito. O protótipo foi concebido utilizando a linguagem de programação *C#* e faz uso de tecnologias auxiliares, como o *PyTorch* (PASZKE *et al.*, 2019), que é uma biblioteca que implementa uma vasta gama de algoritmos de aprendizado de máquina e o *ML-Agents* (JULIANI *et al.*, 2018), que é um *framework* para a plataforma *Unity* (UNITY, 2021) que apoia a criação de agentes e possibilita o treinamento e a execução em conjunto com redes neurais.

A Figura 22 apresenta uma representação dos módulos implementados na solução. O módulo *strategy* é responsável por implementar as relações metamórficas descritas na subseção 6.1.2 e lida com o processo de observação do ambiente em execução para identificar o exato momento em que a propriedade verificada é violada.

Os módulos *logic* e *handler* são responsáveis por lidar com a propriedade de possíveis objetos *foo* incluídos intencionalmente na cena para auxiliar o processo de teste.

O módulo *controls* implementa a abordagem responsável por monitorar o fluxo de dados

de entrada que estão sendo executados na aplicação a partir da geração que utiliza aprendizado profundo.

O módulo *pathfinding* implementa um algoritmo clássico de busca de caminho (*A**) que é utilizado como mecanismo para traçar rotas para guiar os agentes utilizados em direção ao objetivo de testes investigado.

O módulo *rewind* cria uma estrutura de pilha para armazenar o estado da aplicação a partir do início de um *episódio* para o contexto de teste utilizando aprendizado por reforço. Dessa forma, a partir dos dados registrados é possível reproduzir o comportamento do agente, de forma a entender quais ações realizadas levaram ao comportamento investigado.

O módulo *commons* possui uma série de rotinas que são exploradas por parte dos demais módulos.

Por fim, o módulo *agents* possui uma interface base para a construção dos agentes e implementa toda a lógica de funções de recompensa, coleta de observações e controle de ações.

A solução ainda conta com a utilização de ferramentas externas, como a biblioteca *Ml agents*, que apoia a criação dos agentes, além de uma *API* em *Python* que faz a comunicação da biblioteca *Ml agents* e o *framework Pytorch*, que lida com o processo de treinamento dos agentes utilizando aprendizado por reforço profundo.

Para o correto funcionamento, atualmente, o protótipo é vendorizado dentro do projeto alvo, criado utilizando a plataforma *Unity*.

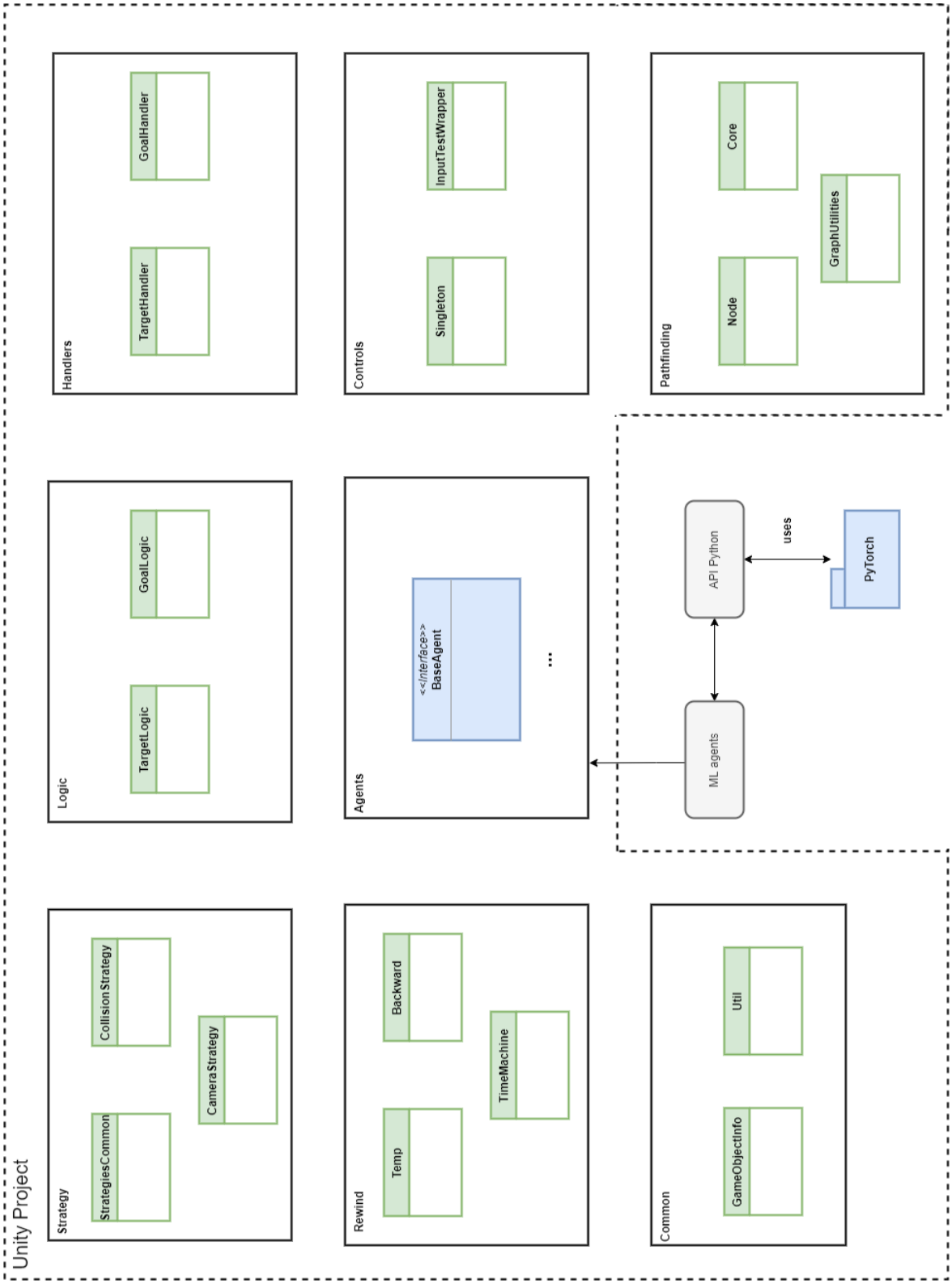
Assim como os dados apresentados nos capítulos anteriores, os artefatos produzidos para a prova de conceito encontram-se publicamente disponíveis em (ANDRADE; NUNES; DELAMARO, 2022).

## 6.3 Considerações Finais

Este capítulo apresentou uma abordagem para aplicação de testes metamórficos em aplicações de RV, utilizando uma proposta de agentes e aprendizado por reforço profundo para a geração de dados de teste. A abordagem se baseia no comportamento de agentes para verificar propriedades desejáveis em aplicações de RV. Caso o agente venha a violar o conjunto de relações metamórficas propostas, o conjunto de dados gerados para aquele cenário é salvo, possibilitando uma reexecução que facilita o processo de *debugging*.

A atividade de teste em aplicações com domínios de saídas complexas é considerada pela comunidade de teste como um problema atual, que tem sido investigado em diferentes domínios. Para o contexto de RV, esse problema se intensifica na figura de características como interação em tempo real e dificuldade de mensurar um oráculo de testes. Tais problemas fazem com que cenários de automatização de testes, ou mesmo a replicação de execuções controladas se tornem um grande desafio.

Figura 22 – Organização dos módulos do protótipo desenvolvido




Fonte: Elaborada pelo autor.

Nesse sentido, uma abordagem automática para verificar o comportamento de propriedades das aplicações, bem como possibilitar a geração de dados de teste, pode fornecer subsídios para redução de custos do teste para aplicações de RV. Além disso, observar o estado dos testes e garantir a reprodutibilidade dos dados de teste promovem um esforço no sentido de uma abordagem que pode ser completamente automatizada. Espera-se, desse modo, que a utilização da abordagem descrita neste capítulo possa explorar aspectos associados ao teste, de forma a auxiliar na detecção de falhas no contexto de aplicações de RV e contribuir para uma melhor experiência para aplicações desenvolvidas neste domínio específico.

No próximo capítulo é apresentado um estudo experimental para avaliar a abordagem proposta.

CAPÍTULO

7

# AVALIAÇÃO EXPERIMENTAL DA ABORDAGEM DE TESTE PARA APLICAÇÕES DE REALIDADE VIRTUAL

No Capítulo 6 foi apresentada a abordagem proposta para automatizar a geração de dados e a detecção de falhas em propriedades de aplicações de RV utilizando aprendizado por reforço e testes metamórfico.

Para validar a abordagem proposta, neste capítulo é apresentada uma avaliação experimental que tem como finalidade medir a capacidade da abordagem em identificar falhas com base nas restrições propostas pelas relações metamórficas apresentadas na abordagem de testes metamórficos.

De forma geral, este estudo avalia a capacidade da abordagem em gerar dados de testes com base em modelos de aprendizado por reforço para automatizar a atividade de teste em aplicações de RV desenvolvidas utilizando a plataforma *Unity*.

As demais seções deste capítulo estão organizadas da seguinte forma: na seção 7.1 são detalhados o planejamento e os objetivos da avaliação experimental, além de serem apresentadas as questões de pesquisa que derivam as hipóteses investigadas na avaliação experimental. Na seção 7.2 é descrito o procedimento utilizado para a condução do experimento, na seção 7.3 são apresentados os resultados observados e as discussões e desdobramentos relacionados a eles. Na seção 7.4 são discutidas as ameaças a validade do experimento, na seção 7.5 são apresentadas possibilidades para estender a abordagem proposta e, por fim, na seção 7.6 são apresentadas as considerações finais deste capítulo.

## 7.1 Planejamento do Experimento

Este experimento tem como objetivo analisar como a abordagem de testes para aplicações de RV apresentada no Capítulo 6 se comporta no que diz respeito à capacidade de revelar falhas de software para esse domínio em específico. Foi definido um ambiente de execução controlado para a execução da abordagem descrita nas seções 6.1 e 6.2.

Esta seção descreve a configuração dos experimentos realizados; especificamente, busca-se investigar a seguinte questão de pesquisa:

- **QP** : É possível auxiliar a prática de teste de software para aplicações de RV utilizando uma abordagem que combina testes metamórficos e aprendizado por reforço profundo?

  Tal questão é expandida em sub-questões de pesquisa, que visam investigar a eficácia da abordagem de testes em revelar falhas para cada uma das propriedades exploradas na proposta apresentada no capítulo anterior:

  - **$QP_1$** : Relações metamórficas baseadas na abordagem de teste proposta são capazes de identificar corretamente falhas de colisões em aplicações de RV?
  - **$QP_2$** : Relações metamórficas baseadas na abordagem de teste proposta são capazes de identificar corretamente falhas de oclusão de visão por câmeras em aplicações de RV?

Para analisar a abordagem proposta, é realizada uma análise utilizando quatro aplicações de RV, retiradas a partir de um repositório que faz a curadoria de projetos *open-source* para esse domínio, dentre as quais existem aplicações com diferentes configurações de cenas, partindo desde cenas compostas por um pequeno número de objetos, até cenas mais complexas, que visam representar ambientes mais complexos.

Para avaliar as questões de pesquisa os resultados são apresentados em termos de métricas de avaliação para medir a eficácia dos modelos treinados e posteriormente os modelos treinados são aplicados a versões das aplicações que contém defeitos sintéticos (manualmente projetados), com intuito de investigar a capacidade dos mesmos em revelar falhas geradas nas aplicações.

### *7.1.1 Definição dos objetivos*

Foi utilizado o modelo GQM. O modelo para aplicação do GQM apresenta o experimento dividido em cinco partes: objeto de estudo, propósito, perspectiva, foco qualitativo e contexto.

- **objetos de estudo:** os objetos de estudo são a abordagem de teste, apresentada no Capítulo 6 e, especificamente, as relações metamórficas incluídas dentro da abordagem;

- **propósito:** o propósito do experimento é avaliar a aplicabilidade de uma abordagem automatizada para geração de dados de teste e detecção de falhas em propriedades de aplicações de RV.;
- **perspectiva:** este experimento é executado do ponto de vista de um pesquisador;
- **foco qualitativo:** o efeito primário em investigação é a capacidade da abordagem em revelar os tipos de falhas investigadas;
- **contexto:** este experimento foi conduzido utilizando uma prova de conceito da abordagem descrita no Capítulo 6, e avaliado em uma máquina Intel Core I7 com 16GB de memória RAM executando o sistema operacional *Windows 10*. Como a implementação é uma prova de conceito acadêmica, as aplicações utilizadas como objetos experimentais deste experimento não possuem uma significância industrial. Ainda assim, buscou-se utilizar diferentes aplicações, visando um melhor entendimento dos resultados, contudo, ressalta-se que os resultados apresentados se destinam, exclusivamente, à discussão de ambientes acadêmicos.

Assim, o experimento realizado pode ser resumido utilizando o seguinte modelo proposto por Wohlin *et al.* (2012):

**Analisar** uma abordagem de testes para aplicações de RV

**com o propósito de** avaliar uma abordagem de testes para aplicações de RV

**no que diz respeito a** capacidade de detecção de falhas

**do ponto de vista de** um pesquisador

**no contexto de** um conjunto de aplicações de RV, escritas na linguagem *C#*, com a plataforma *Unity*, extraídas de um repositório de curadoria de aplicações de RV *open-source*.

### 7.1.2 Ambiente

O experimento foi conduzido em uma única máquina, a qual a configuração pode ser visualizada na Tabela 9.

### 7.1.3 Formulação das hipóteses

As seguintes hipóteses foram definidas a partir das questões de pesquisa apresentadas anteriormente:

Para a questão de pesquisa $\mathbf{QP_1}$ foram definidas as seguintes hipóteses:

Tabela 9 – Características do computador utilizado no experimento

| **Tipo de computador** | *Desktop* |
|---|---|
| **Sistema Operacional** | *Windows* 10 |
| **Processador** | *Core i7-9700* |
| **Memoria RAM** | 16GB |
| **Armazenamento** | 1TB |
| **Placa de Vídeo** | GeForce RTX 2060 |
| **Memoria de Video Dedicada** | 6GB |
| **Unity** | LTS build 2020.3.28f1 |

**Hipótese Nula, $H_0$:** Relações metamórficas baseadas na abordagem de testes proposta não são capazes de revelar falhas de colisão em aplicações de RV a partir dos modelos treinados utilizando aprendizado por reforço.

$$H_0\textbf{:}\ \mathbb{P} = 0,5$$

**Hipótese Alternativa, $H_1$:** Relações metamórficas baseadas na abordagem de testes proposta são capazes de revelar falhas de colisão em aplicações de RV a partir dos modelos treinados utilizando aprendizado por reforço.

$$H_1\textbf{:}\ \mathbb{P} > 0,5$$

O mesmo procedimento foi realizado para a questão de pesquisa $\mathbf{QP_2}$. Assim, foram definidas as seguintes hipóteses:

**Hipótese Nula, $H_0$:** Relações metamórficas baseadas na abordagem de testes proposta não são capazes de revelar falhas em câmeras de aplicações de RV a partir dos modelos treinados utilizando aprendizado por reforço.

$$H_0\textbf{:}\ \mathbb{P} = 0,5$$

**Hipótese Alternativa, $H_1$:** Relações metamórficas baseadas na abordagem de testes proposta são capazes de revelar falhas em câmeras de aplicações de RV a partir dos modelos treinados utilizando aprendizado por reforço.

$$H_1\textbf{:}\ \mathbb{P} > 0,5$$

O ponto principal da investigação das hipóteses é entender o comportamento dos resultados do experimento, de tal forma a justificar por meio de testes estatísticos se os resultados observados foram obtidos ao acaso, ou se podem ser conclusivos a respeito da avaliação da eficácia da abordagem investigada.

Ao definir a hipótese investigada em termos de uma probabilidade, é possível avaliar o comportamento da abordagem com relação à sua capacidade em revelar falhas. De tal forma que a hipótese nula investiga o comportamento próximo de uma aleatoriedade, ou seja, acertar com uma probabilidade de 50% ($\mathbb{P} = 0.5$). Tal cenário revelaria uma abordagem com um comportamento puramente aleatório.

Da mesma forma, a hipótese alternativa parte de uma premissa de que a abordagem não tem um comportamento aleatório, portanto, o teste estatístico deve investigar sobre a probabilidade de ter um desempenho superior ao aleatório, em outras palavras, $\mathbb{P} > 0,5$.

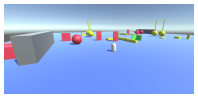
Para isso foi aplicado um teste estatístico binomial, que é recomendado para identificar evidências estatísticas em dados com resultados binários (casos de sucesso/falha) (JOHNSON, 2009). Para esse estudo, foi utilizado o teste de cauda direita (*right tail test*) binomial com nível de significância igual a $\alpha$ = 0,05, estabelecendo um grau de confiança de 95%.

### 7.1.4 Seleção dos objetos experimentais

Os objetos experimentais utilizados neste experimento consistem em uma lista aplicações de RV, escritas na linguagem *C#*, utilizando a plataforma *Unity engine*, conforme descritas na Figura 23.

As aplicações foram extraídas a partir de um repositório que realiza curadoria de aplicações *open-source*[1] e foram escolhidas a fim de explorar a eficácia da abordagem de teste em cenários com diferentes disposições de aplicações com relação à complexidade do número de objetos dispostos em cena.

Figura 23 – Descrição das aplicações utilizadas no experimento

| **Aplicação** |  |  |  |  |
|---|---|---|---|---|
| **Nome** | collision course | simulation environment | roboND rover | destruction derby |
| **# objetos em cena** | 34 | 107 | 60 | 245 |
| ***Steps*** | 15000 | 45000 | 35000 | 75000 |

Fonte: Elaborada pelo autor.

### 7.1.5 Variáveis observadas e métricas coletadas

Para todas as execuções realizadas no experimento foi controlada uma variável independente, que define a propriedade que está sendo avaliada, e a relação metamórfica observada.

[1] <https://unitylist.com/>

A variável dependente coletada pelo tratamento é o comportamento da relação metamórfica, que indicará o desempenho do modelo, gerado a partir da etapa de treinamento, na identificação correta de possíveis falhas, modeladas sinteticamente por meio da inserção de defeitos nas aplicações utilizadas no experimento.

Para o cenário descrito foram coletadas métricas de *acurácia*, *precisão*, *sensibilidade*, *especificidade* e *f1 score* (BARATLOO *et al.*, 2015) a fim de medir o desempenho da abordagem a respeito da capacidade de detecção dos tipos de falhas investigadas.

Dessa forma, para esse estudo, as métricas avaliadas são interpretadas como a seguir:

- Acurácia: é dada pela razão de registros que foram classificados corretamente sobre o número total de registros. Por exemplo, se existirem 10 objetos que contém defeitos e 10 objetos que não contém defeitos, em um cenário no qual foram identificados corretamente 8 objetos que contém defeitos e 7 objetos que não contém defeitos, então a *acurácia* para esse cenário será calculada como $8+7/20=0,75$ e, portanto, será de 75%.

$$Acurácia = \frac{TP+TN}{TP+TN+FP+FN} \tag{7.1}$$

- Precisão: utilizando o exemplo de objetos com defeitos e sem defeitos, a *precisão* mediria o número de "objetos com defeitos" identificados corretamente, dividido pelos "objetos com defeitos" identificados corretamente e pelos "objetos sem defeitos" identificados incorretamente como se possuíssem defeitos.

$$Precisão = \frac{TP}{TP+FP} \tag{7.2}$$

- Sensibilidade: mede a capacidade de identificar corretamente os Verdadeiros Positivos (objetos que contém defeitos serem identificados corretamente).

$$Sensibilidade = Recall = \frac{TP}{TP+FN} \tag{7.3}$$

- Especificidade: a capacidade de identificar corretamente os Verdadeiros Negativos (objetos que **não** contém defeitos serem identificados corretamente).

$$Especificidade = \frac{TN}{FP+TN} \tag{7.4}$$

- F1 score: o objetivo da F1 score é combinar as métricas de precisão e sensibilidade (recall) em uma única métrica, portanto, a pontuação F1 é definida como a média harmônica de precisão e recall.

$$F1 = \frac{2*Precisão*Recall}{Precisão+Recall} = \frac{2*TP}{2*TP+FP+FN} \tag{7.5}$$

Abreviações: TP, Verdadeiro Positivo (do inglês, *True Positive*); FP, Falso Positivo (do inglês, *False Positive*); FN, Falso Negativo (do inglês, *False Negative*); TN, Verdadeiro Negativo (do inglês, *True Negative*).

## 7.2 Condução do Experimento

A condução do experimento consistiu na etapa de treinamento dos agentes utilizando aprendizado por reforço. Foi estabelecido um total de **4 milhões** de *steps* (cada *step* corresponde a um "intervalo de decisão" definido após um número de atualizações fixas que ocorrem entre o momento que o agente que coleta uma observação / altera sua ação, **dez para esse experimento**) para a etapa de treinamento dos agentes. A eficácia do modelo gerado foi medida com base no valor da *função de recompensa* média por *episódio*.

Cada episódio conduzido durante a etapa de treinamento é concluído, se e somente se, a tarefa para a qual o modelo está sendo treinado for concluída com sucesso, ou caso o número máximo de *steps* para o episódio foi atingido. Cada aplicação possui um ajuste adequado do número de *steps* utilizado, o valor adotado para cada aplicação foi definido empiricamente e os valores também são representados na Tabela inclusa na Figura 23.

Os modelos gerados foram então avaliados em versões das aplicações que continham defeitos sintéticos, gerados manualmente por meio de alterações em propriedades/código-fonte (a característica dos defeitos simulados é descrita na seção de resultados para cada uma das propriedades avaliadas). Com base nas versões com defeitos, a capacidade dos modelos em revelar falhas foi avaliada em **5 iterações**, cada uma com intervalo de execução de **2 horas** para cada aplicação e os resultados apresentados nas seções abaixo correspondem à média da detecção de falhas para todas as iterações.

Optou-se por avaliar os modelos em múltiplas iterações para suavizar um possível viés de execução dos modelos gerados, uma vez que o comportamento do modelo gerado pelo algoritmo utilizado não é determinístico (estocástico) podendo, portanto, variar em diferentes iterações para a mesma tarefa (SCHULMAN *et al.*, 2017).

## 7.3 Resultados e Discussões

Esta seção descreve o processo para coleta dos dados obtidos na execução do experimento e a análise dos resultados obtidos após a execução. A etapa de análise está dividida em duas partes *(i)* análise descritiva e *(ii)* testes de hipóteses.

### 7.3.1 Análise descritiva

As análises realizadas neste estudo têm como principal propósito avaliar, experimentalmente, as principais contribuições da abordagem descrita no capítulo anterior e, assim, evidenciar como tal abordagem pode contribuir para avançar o estado da arte em automatização de testes para aplicações de RV. Dessa forma, as análises foram subdivididas em grupos, os quais visam responder às questões de pesquisa caracterizadas anteriormente.

#### 7.3.1.1 Falhas de colisões

Falhas de colisões estão associadas a objetos que possuem *Colliders*, que envolvem os objetos virtuais aos quais são acoplados e são modelados na forma de primitivas geométricas, ou por meio de malhas ajustadas à forma do objeto. Utiliza-se os pontos de coordenadas dos objetos, para verificar as interseções entre os objetos para então identificar uma colisão.

Para falhas de colisão, foram exploradas características de componentes *colliders*, que são responsáveis por detectar colisões em objetos, e são implementados no modulo de física da *Unity*[2].

Os defeitos inseridos consistem: (i) na exclusão desse componente em parte dos objetos que devem ser verificados e (ii) redimensionamento da primitiva geométrica do *collider* de forma que a mesma não represente com fidelidade a estrutura do objeto ao qual o componente está ajustado. Para esse experimento, optou-se por utilizar uma distribuição de 33% de defeitos para os objetos em cena avaliados nas aplicações. A Tabela 10 abaixo relaciona o número total de defeitos para cada uma das aplicações verificadas no estudo.

Tabela 10 – Total de falhas de colisão inseridas nas aplicações avaliados no experimento

| **Aplicação** | **# objetos em cena** | **Falhas simuladas** |
|---|---|---|
| obstacle course | 34 | 11 |
| simulation environment | 107 | 35 |
| roboND rover | 60 | 20 |
| destruction derby | 245 | 80 |

Os resultados do experimento abordam a eficácia das relações metamórficas, baseadas nas propriedades avaliadas, em revelar falhas. Assim, o desempenho das relações metamórficas foi analisado por meio do conjunto de métricas descritas na subseção anterior.

As Tabelas 11, 12, 13 e 14 apresentam os resultados para as métricas investigadas para cada uma das aplicações apresentadas na Tabela 10. Os valores calculados para as métricas *acurácia*, *precisão*, *sensibilidade*, *especificidade* e *F1* são apresentados conforme os resultados para cada uma das cinco iterações, e a discussão dos resultados é feita com base na média dos resultados para cada métrica.

[2] <https://docs.unity3d.com/ScriptReference/Collider.html>

Tabela 11 – Resultados retornados pela RM por meio das métricas de avaliação - **obstacle course**

| **Métrica** | **1ª exec.** | **2ª exec.** | **3ª exec.** | **4ª exec.** | **5ª exec.** | **Média** |
|---|---|---|---|---|---|---|
| Acurácia | 0,924 | 0,931 | 0,947 | 0,947 | 0,918 | 0,934 |
| F1 | 0,883 | 0,896 | 0,921 | 0,917 | 0,879 | 0,899 |
| Precisão | 0,887 | 0,876 | 0,915 | 0,923 | 0,858 | 0,892 |
| Sensibilidade | 0,890 | 0,922 | 0,935 | 0,916 | 0,909 | 0,914 |
| Especificidade | 0,941 | 0,935 | 0,953 | 0,963 | 0,922 | 0,943 |
| Taxa de não-conclusão | 0,034 | 0,040 | 0,027 | 0,032 | 0,048 | 0,036 |

Para a aplicação **obstacle course**, que constitui uma cena com a representação de diversos objetos dinâmicos, utilizados para a criação de percursos, como pontes, cilindros giratórios, pêndulos giratórios, esferas gigantes, entre outros, por meio dos resultados observados na Tabela 11, é possível observar que, em média, obteve-se uma taxa de *acurácia* de 93,4%, mostrando que a abordagem foi capaz de atingir um alto índice de revelação de falhas e ainda entender corretamente cenários nos quais os objetos avaliados não continham defeitos.

Tais resultados podem ser confirmados ao observar as métricas de *sensibilidade* (91,4%) e *especificidade* (94,3%), que medem, respectivamente, a capacidade de encontrar falhas e de entender que um dado objeto observado não contém defeitos.

Com relação à *precisão* , os resultados ficam em torno de 89,9%. Ao observar a característica dos objetos verificados na aplicação **obstacle course**, notou-se que a abordagem teve dificuldade de identificar o comportamento correto, ou falhas, especificamente em objetos que tinham o seu posicionamento alterado em função da dinâmica da execução da cena, por exemplo, esferas que se locomoviam de lugar e pêndulos de movimento constante.

Tabela 12 – Resultados retornados pela RM por meio das métricas de avaliação - **simulation environment**

| **Métrica** | **1ª exec.** | **2ª exec.** | **3ª exec.** | **4ª exec.** | **5ª exec.** | **Média** |
|---|---|---|---|---|---|---|
| Acurácia | 0,776 | 0,772 | 0,743 | 0,763 | 0,764 | 0,764 |
| F1 | 0,684 | 0,692 | 0,649 | 0,678 | 0,676 | 0,676 |
| Precisão | 0,639 | 0,624 | 0,592 | 0,612 | 0,619 | 0,618 |
| Sensibilidade | 0,740 | 0,777 | 0,726 | 0,763 | 0,749 | 0,751 |
| Especificidade | 0,793 | 0,769 | 0,751 | 0,763 | 0,772 | 0,770 |
| Taxa de não-conclusão | 0,171 | 0,178 | 0,192 | 0,182 | 0,177 | 0,180 |

Outro ponto importante a destacar é que, uma vez que o modelo treinado para geração dos dados de teste possui um comportamento estocástico (não determinístico), em alguns cenários de teste, pode ocorrer de os dados de teste gerados não serem suficientes para verificar a relação metamórfica exercitada. Esse cenário é descrito por meio da métrica de *taxa de não-conclusão*, que, para a aplicação **obstacle course**, se manteve em 3,6%. Nesse sentido, o cálculo das métricas descritas anteriormente também levam em consideração os resultados dessa métrica para o cálculo do resultado.

A Tabela 12 apresenta os resultados para a aplicação **simulation environment**. Essa

aplicação representa uma cena de um ambiente de médio porte, composto por construções, aspectos de relevo, pedras, árvores, entre outros objetos. A taxa de *acurácia* se manteve em 76,4%. As métricas de *sensibilidade* e *especificidade* obtiveram, respectivamente, 75,1% e 77%.

Essa aplicação apresentou uma maior *taxa de não-conclusão* comparada à aplicação anterior, 18%. Observando os resultados individualmente, observou-se que os casos de não-conclusão estavam atrelados a objetos muito pequenos. Esse comportamento pode ser explicado pelo fato de que interagir com objetos pequenos é uma tarefa mais complicada, uma vez que necessitam de um conjunto de dados mais específico para atingir o objetivo de teste.

Tabela 13 – Resultados retornados pela RM por meio das métricas de avaliação - **roboND rover**

| **Métrica** | **1ª exec.** | **2ª exec.** | **3ª exec.** | **4ª exec.** | **5ª exec.** | **Média** |
|---|---|---|---|---|---|---|
| Acurácia | 0,847 | 0,875 | 0,885 | 0,853 | 0,854 | 0,863 |
| F1 | 0,781 | 0,817 | 0,834 | 0,785 | 0,794 | 0,802 |
| Precisão | 0,753 | 0,806 | 0,810 | 0,777 | 0,758 | 0,781 |
| Sensibilidade | 0,817 | 0,833 | 0,863 | 0,804 | 0,838 | 0,831 |
| Especificidade | 0,863 | 0,896 | 0,896 | 0,877 | 0,863 | 0,879 |
| Taxa de não-conclusão | 0,125 | 0,100 | 0,094 | 0,125 | 0,113 | 0,111 |

A aplicação **roboND rover** corresponde a um ambiente de simulação de um *rover* espacial, projetado para explorar as rochas e o solo de um planeta. A cena é composta, basicamente, pelo relevo do ambiente e por rochas que devem ser coletadas pelo *rover*.

Os resultados para essa aplicação são descritos na Tabela 13. Para a métrica de *acurácia*, a abordagem obteve uma média de 86,3%, enquanto para as avaliações individuais sobre a capacidade de identificação correta de objetos que contém falhas (*sensibilidade*), e objetos que estão funcionando corretamente (*especificidade*), obtiveram respectivamente 83,1% e 87,9%.

A *precisão*, que mede a proporção de positivos que são identificados corretamente, ficou em 78,1%, e a *taxa de não-conclusão* para essa aplicação ficou em média em 11,1%. Com relação à *taxa de não-conclusão*, similarmente à aplicação anterior, a maior incidência de não-conclusão foi observada em objetos com pequenas dimensões.

Por fim, a abordagem foi verificada em uma cena de maior porte, **destruction derby**, que compreende à simulação de uma cidade, composta de diferentes tipos de construções e diferentes tipos de veículos, todos organizados de forma estática pela cena.

Os resultados podem ser observados na Tabela 14. A métrica de *acurácia* obteve média de 71,6%, enquanto a métrica de *precisão* obteve média de 55,0%. As métricas de *sensibilidade* e *especificidade* obtiveram, respectivamente, 72,8% e 71,0%.

A *taxa de não-conclusão* obteve a maior alta para essa aplicação, comparada com as demais, atingindo um percentual de 27,4%. Não foi possível observar uma característica específica dos objetos que apresentaram tal comportamento. Os cenários de não-conclusão foram distribuídos entre os diferentes tipos de objetos que compõem a cena. Entretanto, uma

Tabela 14 – Resultados retornados pela RM por meio das métricas de avaliação - **destruction derby**

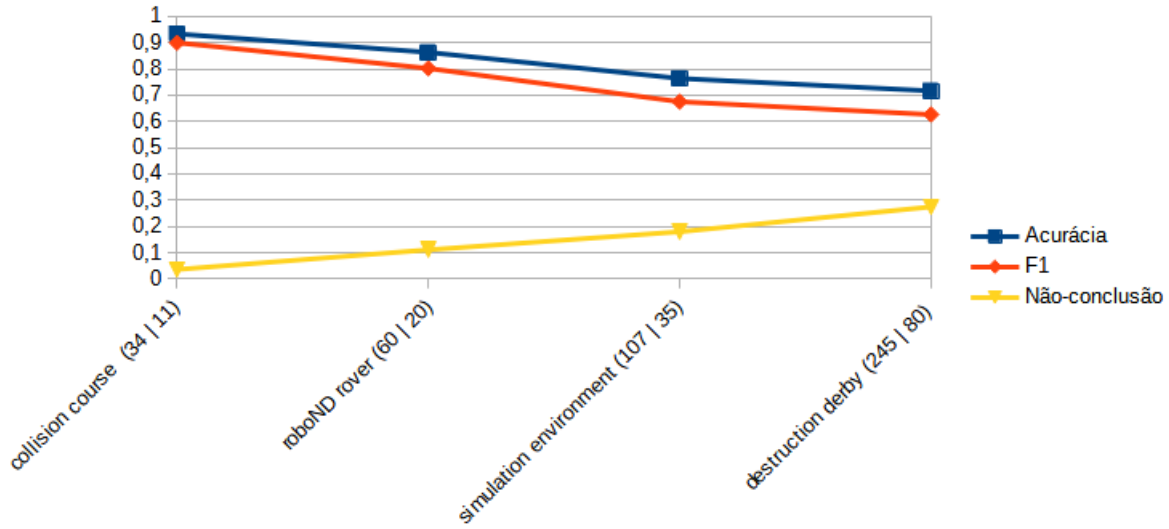
| **Métrica** | **1ª exec.** | **2ª exec.** | **3ª exec.** | **4ª exec.** | **5ª exec.** | **Média** |
|---|---|---|---|---|---|---|
| Acurácia | 0,717 | 0,703 | 0,713 | 0,718 | 0,729 | 0,716 |
| F1 | 0,627 | 0,610 | 0,622 | 0,624 | 0,646 | 0,626 |
| Precisão | 0,550 | 0,534 | 0,547 | 0,553 | 0,564 | 0,550 |
| Sensibilidade | 0,729 | 0,712 | 0,723 | 0,718 | 0,757 | 0,728 |
| Especificidade | 0,712 | 0,698 | 0,708 | 0,718 | 0,715 | 0,710 |
| Taxa de não-conclusão | 0,269 | 0,286 | 0,281 | 0,275 | 0,258 | 0,274 |

característica observada é que o agente na cena dessa aplicação tende a ficar preso (*stuck spot*, Capítulo 5) ao se chocar com objetos que possam estar na sua trajetória de movimentação, o que contribui para que o mesmo não consiga concluir a tarefa observada.

Com base nos resultados, observou-se que que à medida que a abordagem foi executada com aplicações mais complexas – aplicações com maior número de objetos e cenas com maiores dimensões, as métricas avaliadas apresentaram uma queda nos resultados. Tais indícios indicam que os modelos gerados possuíram uma maior dificuldade de generalizar a tarefa de teste em cenas mais complexas.

A Figura 24 mostra o comportamento das métricas para as aplicações avaliadas em função da complexidade das cenas, ordenando as aplicações daquela com o menor número de objetos em cena para a com o maior número. Entre parênteses, na identificação da aplicação, o número total de objetos em cena, seguido pelo número de objetos com falhas.

Figura 24 – Comportamento das métricas de avaliação para falhas de colisão em função do tamanho das aplicações




Fonte: Elaborada pelo autor.

Pelo gráfico de linhas da figura é possível observar que à medida que a abordagem foi executada com aplicações mais complexas, a *taxa de não-conclusão* aumentou, e por consequên-

cia, a taxa de *acurácia* decresceu. Para melhor compreender esse comportamento para o contexto avaliado, foi analisado o comportamento de uma métrica extra: *F1*.

A métrica *F1* é interessante quando pretende-se analisar o equilíbrio entre as métricas de *precisão* e *sensibilidade*. Enquanto a métrica de *precisão* pode ter sua avaliação amplamente relacionada à presença de um grande número de Verdadeiros Negativos (objetos que não contém defeitos e não geram falhas entendidos corretamente) que, na maioria das circunstâncias de uma abordagem de teste, tendem a não agregar tanto valor, enquanto Falsos Negativos (objetos que contém defeitos e geram falhas não detectados) e Falsos Positivos (objetos que não contém defeitos, detectados como se produzissem falhas) geralmente tendem a possuir um custo de verificação. Nesse sentido, a métrica *F1* tende a ser uma boa medida a ser utilizada em cenários nos quais é necessário buscar um equilíbrio entre *precisão* e *sensibilidade*.

Portanto, ao analisar o resultado da métrica *F1*, é possível afirmar que a queda na taxa de *acurácia* da abordagem está majoritariamente ligada ao aumento da *taxa de não-conclusão*, e não efetivamente pelo fato da abordagem falhar em revelar objetos com defeitos, ou por revelar falhas em objetos com comportamento correto.

#### *7.3.1.2 Falhas em câmeras*

Para que o usuário possa atuar em uma cena de RV, ele deve ser capaz de perceber o ambiente da cena, de tal forma que seja capaz de ver/entender o que está acontecendo, para que possa reagir de acordo. Dessa forma, o mundo virtual é comunicado ao usuário por meio da "lente" de uma câmera virtual localizada em uma determinada posição e orientada de uma determinada maneira dentro do ambiente virtual.

A avaliação realizada nesta seção trata, especificamente, de câmeras que se comportam em uma perspectiva de terceira pessoa, ou seja, quando o comportamento da câmera é separado do foco do observador principal (HAIGH-HUTCHINSON, 2009), sendo esse comportamento também chamado de visão aérea, pois, quando representado na cena, diferente de ambientes completamente imersivos, a cena é vista pela perspectiva de uma posição externa.

A complexidade da maioria dos sistemas de câmeras, e, por consequência, a principal origem de falhas nesses componentes se manifesta na forma como a câmera interage com o ambiente. Por exemplo, na maneira como reage em situações nas quais a câmera colide com uma geometria de nível ou quando objetos ficam entrepostos entre a câmera e a figura do observador, causando problemas de oclusão.

Outra perspectiva se dá com relação à noção de movimentação "relativa à câmera", que pode apresentar diferentes interpretações. Por exemplo, enquanto em ambientes completamente imersivos não existe distinção entre a "esquerda do observador" e a "esquerda da câmera". Uma visão em terceira pessoa coloca a câmera fora do observador e, portanto, a "esquerda da câmera" pode ser apontada em uma direção diferente da "esquerda do observador". Por exemplo, elas

estão literalmente em direções opostas se a câmera estiver olhando para a frente do observador.

Tabela 15 – Total de objetos observados nas aplicações avaliadas no experimento para avaliar falhas em câmeras

| **Aplicação** | **# objetos em cena** | **Objetos observados** |
|---|---|---|
| obstacle course | 34 | 8 |
| simulation environment | 107 | 28 |
| roboND rover | 60 | 16 |
| destruction derby | 245 | 73 |

Assim como na avaliação sobre falhas de colisões, as falhas investigadas em câmeras foram sinteticamente produzidas por meio da inserção manual de defeitos nos *scripts* que comandam o comportamento dos objetos de câmera nas cenas investigadas. Nesse sentido, diferentemente da seção anterior, os objetos observados na atividade de testes não contém defeitos, mas são utilizados como objetos de interesse para tentar desencadear falhas por meio dos defeitos introduzidos nos *scripts* que controlam os objetos de câmera.

A Tabela 15 mostra a distribuição desses objetos nas cenas avaliadas. Vale ainda observar que o número de objetos difere do número total de objetos na cena, uma vez que nem todos os objetos na cena são capazes de acionar o comportamento falho do objeto câmera em virtude de suas diferentes características como dimensão. Portanto, somente os objetos com tal potencial foram considerados como "objetos observáveis".

Os defeitos inseridos nos *scripts* que controlam os objetos de câmera consistem em alterar o comportamento da câmera em acompanhar o usuário na cena, para isso, foi inserida uma pequena alteração no trecho de código responsável pelo movimento de suavização, que reduz continuamente a distância entre a câmera e o usuário da cena utilizando Interpolação Linear [3], além de outro defeito visando alterar a dinâmica responsável por aproximar a câmera ao usuário da cena quando detecta algum objeto capaz de bloquear a percepção de visão da câmera.

Assim como na subseção anterior, os resultados do experimento abordam a eficácia da relação metamórfica, baseada na propriedade avaliada (câmeras em cenas RV), em revelar falhas. Assim, o desempenho da relação metamórfica foi analisado por meio do conjunto de métricas descritas na etapa de planejamento do experimento.

As Tabelas 16, 17, 18 e 19 apresentam os resultados para as métricas investigadas para cada uma das aplicações apresentadas na Tabela 15. Os valores calculados para as métricas *acurácia*, *precisão*, *sensibilidade* e *especificidade* são apresentados conforme os resultados para cada uma das cinco iterações, e a discussão dos resultados é feita com base na média dos resultados para cada métrica.

Para a aplicação **obstacle course**, por meio dos resultados observados na Tabela 16, é

[3] <https://docs.unity3d.com/ScriptReference/Vector3.Lerp.html>

Tabela 16 – Resultados retornados pela RM (câmera) por meio das métricas de avaliação - **obstacle course**

| **Métrica** | **1ª exec.** | **2ª exec.** | **3ª exec.** | **4ª exec.** | **5ª exec.** | **Média** |
|---|---|---|---|---|---|---|
| Acurácia | 0,886 | 0,931 | 0,929 | 0,915 | 0,913 | 0,915 |
| F1 | 0,882 | 0,924 | 0,924 | 0,909 | 0,898 | 0,907 |
| Precisão | 0,918 | 0,960 | 0,960 | 0,937 | 0,971 | 0,949 |
| Sensibilidade | 0,868 | 0,904 | 0,904 | 0,904 | 0,859 | 0,888 |
| Especificidade | 0,904 | 0,959 | 0,954 | 0,927 | 0,968 | 0,942 |
| Taxa de não-conclusão | 0,084 | 0,05 | 0,045 | 0,063 | 0,054 | 0,059 |

possível observar que, em média, a abordagem obteve uma taxa de *acurácia* de 91.5%, o que indica que a abordagem é capaz de atingir um alto índice de revelação de falhas e ainda entender corretamente cenários nos quais os objetos avaliados não desencadeavam falhas.

Tais resultados podem ser confirmados ao observar as métricas de *sensibilidade* (88,8%) e *especificidade* (94,2%), que medem, respectivamente, a capacidade de encontrar falhas e de entender que um dado objeto observado não desencadeia a falha simulada.

Com relação à *precisão* , os resultados ficam em torno de 94,9%. Ao observar a característica dos objetos observados na aplicação **obstacle course**, notou-se que a abordagem teve uma leve dificuldade de identificar o comportamento falho especificamente em cenários que, para desencadear a falha inserida no comportamento da câmera, era necessária a realização de uma movimentação muito específica da câmera em virtude das dimensões do objeto observado.

Assim como para as falhas de colisões, também foi medida a *taxa de não-conclusão*, que, para a aplicação **obstacle course**, se manteve em 5,9%. Nesse sentido, o cálculo das métricas descritas anteriormente também levam em consideração os resultados dessa métrica para o cálculo final do resultado.

Tabela 17 – Resultados retornados pela RM (câmera) por meio das métricas de avaliação - **simulation environment**

| **Métrica** | **1ª exec.** | **2ª exec.** | **3ª exec.** | **4ª exec.** | **5ª exec.** | **Média** |
|---|---|---|---|---|---|---|
| Acurácia | 0,760 | 0,753 | 0,721 | 0,728 | 0,735 | 0,740 |
| F1 | 0,617 | 0,620 | 0,591 | 0,584 | 0,590 | 0,600 |
| Precisão | 0,526 | 0,511 | 0,481 | 0,486 | 0,488 | 0,498 |
| Sensibilidade | 0,766 | 0,808 | 0,789 | 0,755 | 0,766 | 0,777 |
| Especificidade | 0,758 | 0,735 | 0,699 | 0,719 | 0,725 | 0,727 |
| Taxa de não-conclusão | 0,190 | 0,199 | 0,216 | 0,211 | 0,214 | 0,206 |

A Tabela 17 apresenta os resultados para a aplicação **simulation environment**. Essa aplicação representa uma cena de um ambiente de médio porte, composto por construções, aspectos de relevo, pedras, árvores, entre outros objetos. Diferentemente da avaliação de falhas de colisão, na qual todos os objetos da cena foram considerados na etapa de testes, para a avaliação de falhas na câmera foram considerados apenas os objetos relacionados a construções

e relevo, que são os objetos da cena capazes de desencadear as falhas investigadas. A taxa de *acurácia* se manteve em 74%. As métricas de *sensibilidade* e *especificidade* obtiveram, respectivamente, 77,7% e 72,7%.

Essa aplicação apresentou uma maior *taxa de não-conclusão* comparado à aplicação anterior, 20,6%. Observando os resultados individualmente, observou-se que os casos de não-conclusão estavam atrelados a objetos muito pequenos. Esse comportamento pode ser explicado uma vez que objetos pequenos necessitam de uma *precisão* maior para interação, necessitando de um conjunto de dados mais específico para possibilitar a sua interação.

Os resultados para essa aplicação apresentam uma queda acentuada se comparados aos resultados da aplicação anterior, principalmente quando observada a métrica *precisão*. Investigando o modelo gerado, observou-se que, diferentemente da aplicação **obstacle course** que possui uma movimentação de câmera livre, existe uma maior dificuldade em cumprir a atividade projetada pois a aplicação **simulation environment** possui um modelo de câmera fixa, que dificulta explorar o comportamento da câmera em função dos objetos de interesse, fazendo com que, em muitos casos, o agente não seja capaz de cumprir o objetivo de teste projetado.

Os resultados para a aplicação **roboND rover** são apresentados na Tabela 18. Para a métrica de *acurácia*, a abordagem obteve uma média de 75,8%, enquanto para as avaliações individuais sobre a capacidade de identificação correta de objetos que capazes de desencadear falhas (*sensibilidade*), e objetos que funcionanm corretamente (*especificidade*), obtiveram respectivamente 77,7% e 74,9%.

Tabela 18 – Resultados retornados pela RM (câmera) por meio das métricas de avaliação - **roboND rover**

| **Métrica** | **1ª exec.** | **2ª exec.** | **3ª exec.** | **4ª exec.** | **5ª exec.** | **Média** |
|---|---|---|---|---|---|---|
| Acurácia | 0,768 | 0,762 | 0,744 | 0,75 | 0,766 | 0,758 |
| F1 | 0,668 | 0,673 | 0,654 | 0,658 | 0,681 | 0,667 |
| Precisão | 0,611 | 0,599 | 0,583 | 0,587 | 0,608 | 0,598 |
| Sensibilidade | 0,764 | 0,786 | 0,768 | 0,773 | 0,795 | 0,777 |
| Especificidade | 0,769 | 0,751 | 0,733 | 0,739 | 0,753 | 0,749 |
| Taxa de não-conclusão | 0,202 | 0,198 | 0,218 | 0,204 | 0,194 | 0,203 |

A *precisão*, que mede a proporção de positivos que são identificados corretamente, ficou em 59,8%, e a *taxa de não-conclusão* para essa aplicação ficou em média em 20,3%. Com relação à *taxa de não-conclusão*, similar à aplicação **obstacle course**, a aplicação **roboND rover** também possui uma característica de movimentação livre de câmera, permitindo explorar de forma mais fácil o comportamento dos objetos investigados em relação à câmera. Contudo, observando os resultados individualmente, observou-se que dois objetos em específico contribuíram para o resultado da *taxa de não-conclusão* por necessitarem de um movimento muito específico da câmera em virtude do seu posicionamento em relação à construção da cena.

Por fim, a abordagem foi verificada em uma cena de maior porte, **destruction derby**. Os resultados podem ser observados na Tabela 19. A métrica de *acurácia* obteve média de

64,5%, enquanto a métrica de *precisão* obteve média de 50,4%. As métricas de *sensibilidade* e *especificidade* obtiveram, respectivamente, 65,7% e 63,9%.

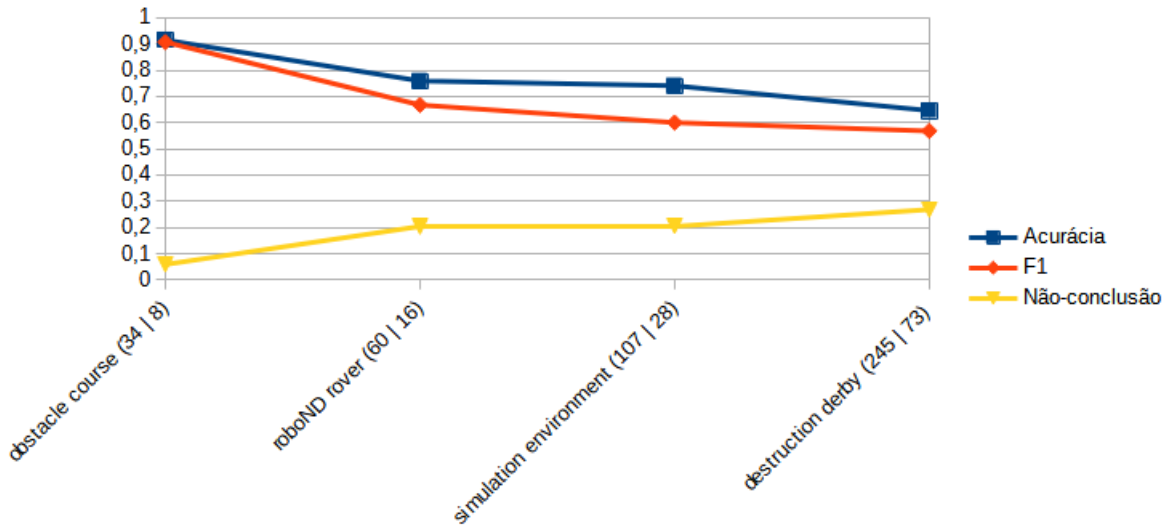
Tabela 19 – Resultados retornados pela RM (câmera) por meio das métricas de avaliação - **destruction derby**

| **Métrica** | **1ª exec.** | **2ª exec.** | **3ª exec.** | **4ª exec.** | **5ª exec.** | **Média** |
|---|---|---|---|---|---|---|
| Acurácia | 0,650 | 0,631 | 0,646 | 0,655 | 0,644 | 0,645 |
| F1 | 0,563 | 0,556 | 0,572 | 0,582 | 0,569 | 0,568 |
| Precisão | 0,510 | 0,487 | 0,506 | 0,515 | 0,500 | 0,504 |
| Sensibilidade | 0,632 | 0,652 | 0,666 | 0,673 | 0,662 | 0,657 |
| Especificidade | 0,659 | 0,619 | 0,635 | 0,646 | 0,634 | 0,639 |
| Taxa de não-conclusão | 0,263 | 0,273 | 0,268 | 0,267 | 0,272 | 0,268 |

Assim como para falhas de colisão, a *taxa de não-conclusão* obteve a maior alta para essa aplicação, comparada com as demais, atingindo um percentual de 26,8%. De forma similar, em muitos cenários de não-conclusão o agente tende a ficar preso ao se chocar com objetos que possam estar na sua trajetória de movimentação. Outro ponto identificado se dá pelo fato dessa aplicação possuir características de câmera fixa, similares à aplicação *simulation environment*. Apesar de terem sido selecionados apenas objetos com potencial para verificar o comportamento da câmera, em algumas situações o agente não é capaz de se posicionar adequadamente de forma a desencadear o comportamento falho existente no componente da câmera.

Assim como para as falhas de colisão,observou-se que à medida que a abordagem foi executada com aplicações mais complexas – aplicações com maior número de objetos e cenas com maiores dimensões, as métricas avaliadas apresentaram uma queda nos resultados.

Figura 25 – Comportamento das métricas de avaliação para falhas em objetos de câmera em função do tamanho das aplicações




Fonte: Elaborada pelo autor.

A Figura 25 mostra o comportamento das métricas para as aplicações avaliadas em

função da complexidade das cenas, ordenando as aplicações daquela com o menor número de objetos em cena para a com o maior número. Entre parênteses, na identificação da aplicação, o número total de objetos em cena, seguido pelo número de objetos observados na avaliação de falhas em objetos de câmera.

Pelo gráfico da Figura 25 é possível observar que à medida que a abordagem foi executada com aplicações mais complexas, a *taxa de não-conclusão* aumentou, e por consequência, a taxa de *acurácia* reduziu. Para melhor compreender esse comportamento também foi analisado o resultado da métrica *F1*.

Assim como para colisões, também foi analisado o resultado da métrica *F1* visando entender o comportamento das métricas de *precisão* e *sensibilidade* com relação ao tamanho das aplicações investigadas. Pelo gráfico em conjunto com os resultados apresentados nas Tabelas 16, 17, 18 e 19, é possível afirmar que a queda na taxa de *acurácia* da abordagem está majoritariamente ligada ao aumento da *taxa de não-conclusão*, e não efetivamente pôr a abordagem falhar em revelar objetos que desencadeiam falhas, ou por revelar falhas em objetos com comportamento correto.

Nesse sentido, a observação acima permite pontuar que quanto mais desiguais são os valores de *precisão* e *sensibilidade*, menor a média harmônica entre elas (*F1*), indicando para os cenários analisados, que a métrica foi bastante afetada pela métrica *precisão*, que, por sua vez, é afetada pela *taxa de não-conclusão*. Esse resultado direciona um caminho interessante para o aprimoramento da abordagem, ao indicar a necessidade da redução da *taxa de não-conclusão*.

### 7.3.2 Testes de hipóteses

Para o contexto das aplicações de RV analisadas no experimento, como os cenários de teste avaliados sempre possuem um conjunto de dados distinto que visam concluir o objetivo de teste, gerado por meio de um algoritmo de aprendizado por reforço não estocástico, não é possível afirmar que um cenário de teste é perfeitamente igual a nenhum outro. Dessa forma, não é possível considerar que os processos de execução adotados como repetições sejam idênticos para o experimento.

Nesse sentido, é importante reforçar que as condições de um modelo de probabilidade são abstrações que ajudam a simplificar realisticamente o complexo mecanismo que governa os resultados de um experimento (JOHNSON, 2009). Portanto, os resultados apresentados nos testes estatísticos devem ser vistos como uma aproximação do mundo real, de tal forma que o mérito dos mesmos reside no sucesso com que os dados apresentados na seção anterior são capazes de explicar as variações casuais dos resultados.

Para os dois cenários analisados, os testes estatísticos realizados devem investigar o comportamento da abordagem no que diz respeito à capacidade de identificar corretamente o estado do objeto em teste, ou seja, se a abordagem é capaz de identificar corretamente os casos

de Verdadeiros Positivos (casos que os objetos revelavam falhas) e Verdadeiros Negativos (casos nos quais os objetos não revelavam nenhum tipo de falha). Para tal, é utilizado como base o oráculo originalmente produzido para cada um dos objetos em teste para cada uma das aplicações avaliadas.

Tabela 20 – Descrição do número de objetos, iterações, execuções e total de observações para cada aplicação avaliada no experimento

| **Aplicação** | **# Objetos ($o$)** | **Iterações ($i$)** | **# Execuções por iteração ($e$)** | **# Observações por objeto ($t$)** |
|---|---|---|---|---|
| **Colisão** | | | | |
| obstacle course | 34 | 5 | 14 | 70 |
| simulation environment | 107 | 5 | 10 | 50 |
| roboND rover | 60 | 5 | 12 | 60 |
| destruction derby | 245 | 5 | 8 | 40 |
| **Câmera** | | | | |
| obstacle course | 8 | 5 | 55 | 275 |
| simulation environment | 28 | 5 | 38 | 190 |
| roboND rover | 16 | 5 | 45 | 225 |
| destruction derby | 73 | 5 | 22 | 110 |

A Tabela 20 apresenta o número total de observações por objeto para cada uma das aplicações avaliadas no experimento. Por exemplo, para a aplicação **obstacle course**, o comportamento da abordagem é avaliado para cada um dos objetos ($o = 34$) individualmente. Para cada teste estatístico realizado, são levadas em consideração todas as execuções realizadas no experimento, ou seja, o número total de observações por objeto utilizado para o teste estatístico será definido pela multiplicação do número de iterações do experimento ($i = 5$) pelo número de execuções ($e = 14$) obtido durante o intervalo de tempo de execução, 2 horas, de cada iteração, totalizando o número de observações para cada objeto ($t = 70$).

Para complementar a análise descritiva apresentada na seção anterior foi realizada uma análise estatística para avaliar a precisão da abordagem de teste. Para tal, conforme descrito na etapa de planejamento do experimento (seção 7.1), foi adotado o teste estatístico binomial (JOHNSON, 2009) para investigar a evidência estatística dos resultados em relação ao número de verdadeiros (VP/VN) nas duas questões de pesquisa investigadas. As Tabelas 21 e 22 apresentam, respectivamente, o número de verdadeiros para cada uma das aplicações investigadas, a probabilidade adotada para o teste estatístico em relação à hipótese nula para as questões de pesquisa investigadas e o alcance do teste para todos os objetos em cada uma das aplicações avaliadas.

A Tabela 21 apresenta os resultados dos testes estatísticos com relação à questão de pesquisa um ($QP_1$), que tinha por objetivo investigar o comportamento da abordagem com relação à capacidade de revelação de falhas de colisão em aplicações de RV por meio da aplicação de testes metamórficos. Os resultados indicam que para a taxa de probabilidade adotada ($\mathbb{P} = 0.5$),

Tabela 21 – Teste binomial a partir das métricas coletadas no estudo - Colisão

| Aplicação | Métrica | Oráculo | Prob. ($\mathbb{P}$) | $H_0$? | Alcance |
|---|---|---|---|---|---|
| obstacle course | VP | 23 | 0.5 | Rej. | 23/23 |
| | VN | 11 | | | 11/11 |
| simulation environment | VP | 72 | | Rej. | 72/72 |
| | VN | 35 | | | 34/35 |
| roboND rover | VP | 40 | | Rej. | 40/40 |
| | VN | 20 | | | 20/20 |
| destruction derby | VP | 165 | | Rej. | 151/165 |
| | VN | 80 | | | 78/80 |

a hipótese nula foi rejeitada em todos os objetos das aplicações **obstacle course**, foi rejeitada em 106 dos 107 objetos investigados para a aplicação **simulation environment**, foi rejeitada em todos os cenários de teste explorados na aplicação **roboND rover** e foi rejeitada em 229 cenários dos 245 para a aplicação **destruction derby**. Em outras palavras, os resultados apontam que a abordagem demonstrou uma sólida consistência em identificar corretamente o comportamento dos cenários de teste analisados, indicando que pode ser efetivamente explorada em ambientes de teste para avaliar falhas de colisão em aplicações de RV.

Quanto aos casos específicos que a abordagem não foi capaz de rejeitar a hipótese nula, destaca-se que tal resultado não atesta sobre a falta de eficácia da abordagem, mas pelo simples fato de não haver amostragem suficiente nos dados para sustentar a premissa adotada, sendo necessário um maior número de observações para tal.

Tabela 22 – Teste binomial a partir das métricas coletadas no estudo - Câmera

| Aplicação | Métrica | Oráculo | Prob. ($\mathbb{P}$) | $H_0$? | Alcance |
|---|---|---|---|---|---|
| obstacle course | VP | 4 | 0.5 | Rej. | 4/4 |
| | VN | 4 | | | 4/4 |
| simulation environment | VP | 21 | | Rej. | 21/21 |
| | VN | 7 | | | 7/7 |
| roboND rover | VP | 11 | | Rej. | 11/11 |
| | VN | 5 | | | 5/5 |
| destruction derby | VP | 47 | | Rej. | 42/47 |
| | VN | 26 | | | 23/26 |

A Tabela 22 apresenta os resultados dos testes estatísticos para a questão de pesquisa dois ($QP_2$) , que investigava o comportamento da abordagem quanto à capacidade de revelação de falhas em objetos de câmera em aplicações de RV. Os resultados apresentados indicam que, para o grau de probabilidade adotado ($\mathbb{P} = 0.5$), a hipótese nula foi rejeitada em todos os cenários para a aplicação **obstacle course**, em todos os cenários para a aplicação **simulation environment**, em todos os cenários para a aplicação **roboND rover** e para 65 dos 73 cenários da aplicação **simulation environment**.

É possível observar que apesar da abordagem para detecção de falhas em objetos de

câmera ter apresentado métricas ligeiramente inferiores se comparadas aos resultados das métricas para falhas de colisão, ainda assim pouquíssimos cenários de teste não foram capazes de rejeitar a hipótese investigada. Esse resultado pode ser explicado devido ao fato de que a abordagem de câmera foi investigada com um número menor de objetos em comparação com a abordagem de colisão. Esse detalhe possibilitou que fosse extraído um número maior de observações para cada cenário de teste para a abordagem de detecção de falhas em câmera (ver Tabela 20), permitindo que os testes estatísticos fossem mais precisos com relação à eficácia da abordagem.

Os resultados individuais dos testes estatísticos apresentados para todos os objetos das quatro aplicações investigadas podem ser consultados no *labpackage* do experimento (ANDRADE; NUNES; DELAMARO, 2022), que também contém todas as informações com relação aos resultados brutos, planilhas e *scripts* auxiliares utilizados para a consolidação dos resultados apresentados neste capítulo.

## 7.4 Ameaças à Validade

Um ponto importante sobre um estudo experimental diz respeito à validade dos resultados obtidos, que pode afetar a capacidade de generalização dos resultados apresentados. Dessa forma, os resultados apresentados ao longo deste capítulo estão sujeitos a ameaças à validade e, portanto, devem ser interpretados com cautela. Abaixo são discutidas as principais preocupações com relação à validade dos resultados com base na recomendação proposta por Wohlin *et al.* (2012).

Com relação às **ameaças à validade interna**, todas as variáveis nos experimentos foram controladas visando minimizar tais ameaças. Nesse sentido, para aumentar a confiança nos resultados, foram definidas métricas para melhor entender o comportamento dos resultados do experimento; os dados foram analisados não apenas por meio de tabelas e gráficos, mas também com auxílio de testes estatísticos para garantir que os resultados obtidos tenham sido interpretados de maneira correta e para assegurar que as relações apresentadas por meio da análise descritiva possuam significância e não tenham sido obtida ao acaso.

No contexto de **ameaças à validade externa**, ao realizar experimentos em engenharia de software um grande risco apresentado é a falta de representatividade das aplicações utilizadas como objetos experimentais utilizados nos processos de validação. Infelizmente, esse é um problema que sempre afeta pesquisas relacionadas à área de engenharia de software devido ao fato de não existir teoria suficientemente embasada que possa garantir que um determinado conjunto de aplicações possa constituir uma amostra representativa para experimentação.

A fim de mitigar tal risco, buscou-se utilizar um conjunto de programas com características distintas com diferentes complexidades de cena, visando explorar diferentes aspectos de aplicações de RV. Além de selecionar aplicações a partir de um repositório público, visando evitar algum tipo de viés no processo de escolha. Outro ponto a ser destacado foi a preocupação

de descrever a abordagem avaliada sem ater-se a aspectos relacionados a uma tecnologia em específico (Capítulo 6), tal medida garante a interessados a possibilidade de explorá-la com outro tipo de ferramental tecnológico, além de disponibilizar todo o material produzido de forma pública (ANDRADE; NUNES; DELAMARO, 2022).

Para **ameaças à validade de constructo** buscou-se evitar mecanismos que pudessem interferir de maneira nos resultados observados. Nesse sentido, foram evidenciados todos os resultados obtidos na condução do experimento, inclusive resultados que poderiam ser interpretados como insatisfatórios. Tal atitude visa garantir que estudos futuros que explorem o tema levem em consideração todas as observações apontadas nos resultados apresentados.

A fim de garantir a confiabilidade dos resultados apresentados e evitar **ameaças à validade da conclusão**, adotou-se cuidadosamente o processo de experimentação proposto por Wohlin *et al.* (2012), que apresenta diretrizes para condução de experimentação para o contexto de engenharia de software com base no método definido por Basili (1994). Além disso, todos os recursos e dados disponíveis no repositório do estudo permitem a revisão dos dados por terceiros.

## 7.5 Estendendo a abordagem

Esta seção discute brevemente os desafios para expandir as relações metamórficas avaliadas e o custo para o desenvolvimento de relações metamórficas para outras propriedades.

Quanto à factibilidade de estender uma das relações metamórficas apresentadas para lidar com algum requisito específico de uma aplicação, o custo é relativamente baixo, uma vez que o principal componente da abordagem que precisará ser adicionado/alterado será um dos componentes que estão dentro do pacote *Strategy* (Seção 6.2).

Por exemplo, no caso de objetos que possuem formas complexas, uma abordagem costumeiramente utilizada é utilizar um conjunto de *colliders* com formas primitivas para representar a forma do objeto do objeto complexo, evitando o uso de *colliders* na forma de malhas a fim de evitar sobrecarga de desempenho (Unity Technologies, 2022). Contudo, um possível cenário de falha pode acontecer quando o objeto que colide com dois ou mais *colliders* do objeto de forma complexa em um curto intervalo de tempo.

No cenário descrito, o objeto colisor tem a sua velocidade ajustada instantaneamente de volta à sua velocidade original antes do impacto, resultando em uma mudança maciça na aceleração e, portanto, em uma aplicação massiva de força, que em seguida entra em contato com outro *collider* parte do objeto complexo, fazendo com que o processo se repita, resultando em um enorme "efeito de salto", causando um efeito físico não-realista. É possível verificar automaticamente a ocorrência desse tipo de situação ao projetar uma relação metafórica que verifica a corretude do comportamento dos *colliders* do objeto complexo com base na velocidade do objeto colisor após ele colidir com o objeto de forma complexa.

Similarmente, no contexto de câmeras é possível especializar a verificação do comportamento do objeto de câmeras em cenários específicos. Uma estratégia muito comum utilizada para evitar *cybersickness* é tentar reduzir a movimentação da câmera enquanto outros objetos estão se movimentando. Uma estratégia comumente utilizada em jogos é *platform snapping*, na qual proíbe-se o movimento vertical da câmera durante um salto até que o *jumper* aterrisse em uma plataforma. Isso reduz o movimento da câmera enquanto o usuário tem que realizar saltos sequenciais e faz com que o usuário mantenha a percepção em um único ponto na cena.

A principal propriedade que deve ser mantida no *platform snapping* (KEREN, 2015) é a posição $Y$ da câmera enquanto o *jumper* não aterrissar e se mover no chão. Uma possível relação metamórfica para verificar esse comportamento pode ser definida em função da posição $Y$ do objeto da câmera e sempre que um cenário de "jumping"ocorrer, a mesma ficará responsável por monitorar o comportamento da posição do objeto câmera.

Também é possível explorar as demais características descritas na seção 6.2, por exemplo, no contexto de falhas relacionadas a desempenho, uma possível estratégia para identificar gargalos de processamento é verificar a correta utilização da técnica *Level of Detail* (LOD). Tal técnica consiste em reduzir o número de operações que uma *Graphics Processing Unit* (GPU) requer para renderizar objetos e malhas distantes ou fora do campo de visão da câmera (LUEBKE *et al.*, 2003).

Dessa forma, é possível propor uma relação metamórfica que verifique se o nível de LOD apropriado para um determinado objeto está sendo representado corretamente com base no campo de visão da câmera ou com base na distância da câmera para o objeto de interesse.

## 7.6 Considerações Finais

Neste capítulo foram apresentados o design experimental do estudo e os resultados obtidos por meio da avaliação experimental da abordagem de teste para aplicações de RV descrita no Capítulo 6. O capítulo abordou desde a fase de concepção e planejamento do experimento, formulação de questões de pesquisa e hipóteses e posterior condução e apresentação dos dados na forma de análise descritiva e testes estatísticos.

De acordo com os resultados observados pela avaliação experimental conduzida, é possível concluir que é viável a adoção de uma abordagem automatizada de teste de software para aplicações de RV utilizando testes metamórficos e aprendizado por reforço para a geração de dados de teste. Os estudos evidenciaram que a utilização da abordagem pode ser uma alternativa eficaz para revelar falhas relacionadas a colisão e objetos de câmera em aplicações de RV.

Os resultados dos experimentos indicaram que a abordagem avaliada apresentou altas taxas para as métricas para os cenários avaliados, com uma acentuada queda quando verificada em aplicações mais complexas. Observou-se que a queda das métricas está diretamente ligada às

*taxa de não-conclusão* dos cenários de teste investigados, fator que indica espaço para possíveis melhorias futuras. Além da análise das métricas, buscou-se realizar testes estatísticos com objetivo de dar um maior grau de confiança sobre os resultados obtidos a fim de tornar possível a generalização dos resultados.

Como trabalhos futuros, pretende-se aperfeiçoar a abordagem para reduzir a taxa de não-conclusão observada nos experimentos. Para isso, planeja-se explorar os resultados apresentados por Alonso *et al.* (2021). Além disso, pretende-se investigar a incorporação de novas relações metamórficas de forma a atender as observações discutidas durante o Capítulo 5. Por fim, submeter a abordagem à avaliação de um conjunto extra de aplicações de RV com objetivo de avaliar e corroborar com os resultados apresentados neste capítulo.

CAPÍTULO

# 8

# CONCLUSÕES E TRABALHOS FUTUROS

Aplicações de RV têm ganhado cada vez mais popularidade no mercado de software e têm sido utilizadas cada vez mais em diferentes domínios, desde entretenimento, passando pela área da saúde, prototipação, simulação, até perspectivas cada vez mais realistas de da criação de universos virtuais paralelos que visam replicar a realidade por meio de dispositivos digitais. No entanto, à medida que a aplicação dessas novas tecnologias se popularizam, novos desafios, por exemplo, como testar esse tipo de aplicação, emergem no campo da engenharia de software.

Esta tese de doutorado aprimora o estado da arte da área de teste de software para aplicações de RV, introduzindo contribuições teóricas e experimentais para abordar a aplicação de testes metamórficos e a utilização de agentes guiados por aprendizado por reforço para auxiliar na atividade de automatização de teste de software para aplicações de RV. Assim, apresenta e avalia de forma experimental soluções para o problema de pesquisa delimitado abaixo:

**Questão de Pesquisa**

Como a utilização de testes metamórficos pode contribuir para lidar com o problema do oráculo de teste e aprimorar o processo de teste de software no contexto de RV?

Com base nos resultados obtidos durante o desenvolvimento deste trabalho as próximas seções visam discutir a respeito das conclusões obtidas e estão organizadas da seguinte forma: na seção 8.1 são discutidas vantagens e desvantagens sobre a utilização da abordagem de testes proposta para o domínio de RV, além de destacar as principais publicações produzidas durante a condução deste trabalho. A seção 8.3 discute o limite das contribuições apresentadas, bem como os desafios e limitações encontrados durante o desenvolvimento dos trabalhos apresentados nesta tese. Por fim, a seção 8.4 discute brevemente a metodologia adotada no processo de gestão, curadoria e preservação dos dados produzidos durante o desenvolvimento deste trabalho e a seção 8.5 apresenta uma discussão a respeito dos possíveis desdobramentos desta pesquisa,

destacando oportunidades para a realização de trabalhos futuros a respeito dos diferentes tópicos investigado ao longo deste trabalho.

## 8.1 Contribuições desta Tese de Doutorado

A contribuição geral desta tese de doutorado é uma investigação extensiva do problema de teste de software para aplicações de RV e sobre a definição de mecanismos capazes de propor uma elaboração e avaliação automatizada de casos de teste para este domínio. O problema investigado, conforme apresentado no Capítulo 1, foi originalmente apresentado e discutido na comunidade científica por meio da sua apresentação em formato de artigo em *workshop* de teses e dissertações (ANDRADE; NUNES; DELAMARO, 2019) e, posteriormente, por meio de publicação em revista científica (ANDRADE; NUNES; DELAMARO, 2019).

Posteriormente, para melhor condução do trabalho, o problema foi subdividido e abordado sob três ramos principais: análise de artefatos de software, condução de *survey* para entender o perfil das pessoas que fazem uso desse tipo de aplicações e desenvolvimento e avaliação de uma abordagem de teste. As contribuições de cada tópico são brevemente descritas nas subseções abaixo.

### 8.1.1 Análise de artefatos de software

Essa etapa do trabalho foi apresentada nesta tese durante o Capítulo 4 e investigou os principais desafios relacionados à atividade de teste de software em aplicações de RV. Buscou elencar algumas das principais questões relacionadas à dificuldade de se testar esse tipo de aplicação, além de discutir possíveis soluções que poderiam ser usadas/adaptadas para lidar com tais questões.

Também foi discutido se existe, de fato, uma real necessidade de testar aplicações de RV. Para melhor compreensão, foi realizado um estudo experimental amplo, orientado por três questões de pesquisa, cujo os objetivos eram: *(i)* compreender o estado da prática de teste de software no contexto de programas de RV, *(ii)* mensurar métricas e atributos de qualidade em software de RV e, *(iii)* avaliar a predisposição a falhas de aplicações de RV.

Para responder às questões de pesquisa foi catalogada uma coleção de 119 projetos de RV, além de levar em consideração os dados de mais 107 projetos que não compõem o domínio de RV. Essa adição foi feita a fim de possibilitar uma comparação entre os projetos, no que diz respeito às questões de pesquisa levantadas anteriormente, uma vez que projetos de RV não apresentam práticas de teste de software e todos os demais projetos catalogados apresentam práticas consolidadas de teste.

Os resultados apontaram para uma forte evidência de maior presença de débito técnico em forma de *code smells* nos projetos de RV. A comparação com aplicações de propósito geral

permitiu evidenciar que projetos de software com práticas de teste tendem a possuir menor incidência de débito técnico. Da mesma forma, a análise quanto à predisposição a falhas também apontou para uma maior incidência nos projetos de software que não possuíam práticas de teste.

Os resultados foram apresentados à comunidade científica durante o *21st Symposium on Virtual and Augmented Reality* (ANDRADE; NUNES; DELAMARO, 2019). O artigo foi eleito entre os 8 melhores artigos da conferência, e, posteriormente, foi estendido e disponibilizados publicamente, em formato *pre-print*, em literatura cinza, na plataforma *ArXiv* (ANDRADE; NUNES; DELAMARO, 2020).

### 8.1.2 Pesquisa junto com grupos de interesse

Essa etapa do trabalho teve como objetivo entender o ponto de vista dos grupos de interesse envolvidos no processo de desenvolvimento e utilização de aplicações de RV e foi desenvolvida em parceria com o prof. Álvaro Joffre Uribe Quevedo da *Ontario Tech University*, ressaltando os laços de colaboração e internacionalização do trabalho, e foi discutida nesta tese durante o Capítulo 5.

As informações foram coletadas por meio de um *survey*, no qual os participantes foram convidados a responder perguntas específicas sobre perfil de utilização de aplicações de RV, hábitos de utilização, sobre quais tipos de falhas em aplicações em RV contribuem para uma experiência negativa, sobre quais tipos de falhas em dispositivos de hardware para RV contribuem para uma experiência negativa, e, por fim, sobre quais tipos de falhas específicas para o contexto de RV eles reconheciam.

O principal objetivo desse estudo foi compreender as diferentes visões dos grupos de interesse (por exemplo, quais tipos de falhas são mais críticos, quais são menos relevantes, quanto cada uma afeta na qualidade do produto final, etc.), além de investigar o conhecimento dos grupos de interesse com relação a tipos específicos de falhas existentes em aplicações de RV. De tal forma que tais informações pudessem ser utilizadas para auxiliar no processo de desenvolvimento de abordagens de testes capazes de abranger os aspectos levantados.

Os principais resultados dessa etapa do trabalho permitiram compreender as necessidades das práticas de teste de software no domínio de RV sob do ponto de vista de grupos de interesse (*stakeholders*), possibilitando caracterizar as principais preocupações, de acordo com os perfis envolvidos nos aspectos de desenvolvimento e utilização de aplicaçõesde RV, além de entender o que tais grupos julgam o que deve ser priorizado. Os resultados foram apresentados a partir da visão de 88 respondentes do *survey*.

Os resultados dessa etapa do trabalho permitiram tirar conclusões a partir de diferentes pontos de vista, como de uma equipe de garantia da qualidade, mas também sob o ponto de vista de um pesquisador. Os resultados produzidos foram apresentados à comunidade científica na forma de um artigo durante o *22st Symposium on Virtual and Augmented Reality* (ANDRADE *et*

*al.*, 2020).

### 8.1.3 Desenvolvimento e avaliação de abordagem de teste para aplicações de RV

Com base nos resultados apresentados na pesquisa junto a grupos de interesse, foram definidas características específicas a serem investigadas no contexto de testes de aplicações de RV, como, por exemplo, aspectos visuais, física e câmera. Uma vez que, segundo o estudo, falhas nesse tipo de recursos tendem a causar grande frustração ou desconforto ao usar aplicações de RV, uma vez que tais recursos estão presentes na maioria das aplicações de RV e, portanto, são considerados fundamentais para que o usuário tenha uma experiência adequada.

Foi proposta uma abordagem de testes automatizada, apresentada durante o Capítulo 6 e avaliada no Capítulo 7, que utiliza testes metamórficos, para lidar com o problema do oráculo de teste em aplicações de RV, em conjunto com agentes e aprendizado por reforço, para gerar dados de teste e possibilitar a avaliação extensiva das características analisadas.

A abordagem foi validada por meio de uma avaliação experimental que investigou a capacidade de detecção de falhas modeladas em diferentes aplicações de RV. Os resultados dessa etapa foram compilados em um artigo que atualmente encontra-se em fase de avaliação pela revista *IET Software Journal* [1].

## 8.2 Outras publicações

Além dos trabalhos publicados para divulgação dos resultados ao longo do desenvolvimento deste trabalho, e do manuscrito de revista submetido com os resultados finais da tese, o autor desta tese de doutorado também participou como autor/co-autor de uma série de capítulos de livros, artigos de conferência e artigos para revistas científicas desenvolvidos em colaboração com pesquisadores de outras universidades brasileiras e estrangeiras, além de parcerias com colegas de pesquisa do *Laboratório de Engenharia de Software* (LabES) que fez parte. Os artigos estão listados abaixo, em ordem cronológica, com uma breve descrição de seus, respectivos, conteúdos:

1. **Automatização de teste de software com ênfase em teste de unidade** (COSTA; ANDRADE; DELAMARO, 2016) : capítulo de livro que descreve uma introdução aos conceitos básicos de teste de software, principais técnicas de teste (estrutural, funcional e baseada em defeitos), seus principais critérios e ferramentas que auxiliam na sua aplicação;

[1] <https://ietresearch.onlinelibrary.wiley.com/journal/17518814>

2. **Ferramentas de Teste de Mutação** (OLIVEIRA *et al.*, 2018) : capítulo de livro que aborda especificamente o critério de teste de mutação e apresenta as principais características e funcionamento de ferramentas *open source* para aplicação desse critério;

3. **On Applying Metamorphic Testing: An Empirical Study On Academic Search Engines** (ANDRADE *et al.*, 2019) : artigo complementar ao material apresentado na tese que explora a aplicação de testes metamórficos em diferentes domínios. Nesse trabalho foi explorada a utilização de testes metamórficos para identificar falhas de software em motores de indexação de trabalhos científicos;

4. **Software Testing Education: dreams and challenges when bringing academia and industry closer together** (ANDRADE; NEVES; DELAMARO, 2019) : artigo desenvolvido com base em experiência de atuação do aluno enquanto assistente de professor no Programa de Aperfeiçoamento de Ensino (PAE) na Universidade de São Paulo. O artigo apresenta o relato da inclusão de uma abordagem baseada em problemas (do inglês, *Problem Based Learning* (PBL)) em um curso de teste de software, ministrado em parceria com a Universidade Federal de Juiz de Fora, focando na utilização de projetos *open source* como fonte de material de ensino;

5. **Analyzing the effectiveness of One-Op Mutation against the minimal set of mutants** (ANDRADE *et al.*, 2019) : esse artigo foi fruto de uma avaliação experimental que nasceu com base nas percepções do aluno sobre teste de mutação, tema de pesquisa investigado durante o mestrado. O artigo apresenta os resultados de uma avaliação experimental que investiga a eficácia de utilizar casos de testes adequados a um único operador de mutação para matar mutantes pertencentes ao conjunto minimal (conjunto mínimo de mutantes não equivalentes, necessários para gerar um conjunto de testes adequado a todos os mutantes);

6. **Integration of Software Testing to Programming Assignments: An Experimental Study** (AVELLAR *et al.*, 2019) : artigo desenvolvido com base em experiência de atuação do aluno enquanto assistente de professor no PAE na Universidade de São Paulo. O artigo avalia o desempenho de alunos em tarefas de ensino de programação quando há a inclusão de práticas de teste de software à atividade;

7. **Metodologias ativas – Aprendizagem baseada em projetos: Relatos de uma experiência de aplicação em cursos de computação do ICMC/USP** : o artigo relata exclusivamente para a comunidade acadêmica da Universidade de São Paulo os resultados da aplicação da abordagem PBL no contexto de cursos ministrados dentro da instituição;

8. **An Experimental Study on Applying Metamorphic Testing in Machine Learning Applications** (SANTOS *et al.*, 2020) : artigo complementar ao material apresentado na tese que explora a utilização de testes metamórficos em diferentes domínios. Nesse trabalho foi explorada a eficácia de testes metamórficos para identificar falhas em modelos supervisionados de aprendizado de máquina;

9. **Parallel Execution of Programs as a Support for Mutation Testing: A Replication Study** (DELAMARO *et al.*, 2021) : esse artigo discute os resultados dos experimentos desenvolvidos durante o trabalho de mestrado do aluno, em conjunto com experimentos extras, a respeito da viabilidade da aplicação do teste de mutação em máquinas paralelas, visando lidar com o problema do custo computacional de aplicação da técnica;

10. **Testing Techniques of Non-Functional Requirements in Mobile Apps: A Systematic Mapping Study** (COSTA *et al.*, 2022) : artigo do grupo de pesquisa, desenvolvido em parceria com professores da *University of Naples Federico II* e da *Universidade Federal de Itajubá*, no qual é realizado um mapeamento sistemático para fazer um levantamento a respeito de técnicas e ferramentas de teste para requisitos não-funcionais voltados para aplicações móveis;

11. **Uma avaliação de técnicas e critérios de teste de software para a linguagem de programação Python** (BRITO; ANDRADE; DELAMARO, 2022) : esse artigo é fruto dos resultados da experiência de orientação de uma aluno em um trabalho de iniciação científica. O artigo relata a experiência da avaliação de técnicas e critérios de teste para a linguagem de programação *Python* e faz uma avaliação das ferramentas disponíveis para essa linguagem quanto à aderência aos conceitos de teste de software.

## 8.3 Limitações e Suposições Adotadas Durante a Pesquisa

Nesta seção, são descritas as premissas e limitações das contribuições desta tese de doutorado. É importante notar que as limitações específicas de cada capítulo desta tese de doutorado já foram relatadas em suas respectivas seções de ameaças à validade. Assim, esta seção concentra-se em apontar limitações e suposições de forma mais ampla, de tal forma que possam servir para apoiar o desenvolvimento de trabalhos futuros.

No Capítulo 4, foi apresentada uma avaliação experimental sobre análise de práticas de teste, avaliação de *code smells* e predisposição a falhas. Neste estudo, as seguintes suposições foram necessárias:

1. **parâmetros de comparação**: para possibilitar uma comparação entre os resultados observados em projetos de RV catalogados para o estudo, também foram utilizados projetos de propósito geral que utilizavam a mesma linguagem de programação. Entretanto, conforme é amplamente discutido na literatura de experimentação na área de engenharia de software, não há um consenso a respeito de como medir a representatividade a partir de um conjuntos de programas (SJØBERG *et al.*, 2005), da mesma forma que traçar conclusões com base em comparações de resultados produzidos a partir de programas escritos para domínios

distintos, pode apresentar um viés de confirmação por parte do pesquisador (JURISTO; VEGAS, 2009);

2. **algoritmo para avaliar predisposição a falhas**: o algoritmo utilizado no experimento para identificar trechos de código que possuíam predisposição a falhas (YANG; QIAN, 2016) foi atualizado em trabalho seguinte (YANG; YANG; QIAN, 2018), aprimorando os resultados iniciais, o que pode levantar questionamentos a respeito de falsos-positivos/falsos-negativos nos resultados apresentados no experimento e, nesse sentido, comprometer parte das conclusões tiradas a partir dos resultados do experimento.

No Capítulo 5, no qual foi apresentado um *survey* com grupos de interesse, a respeito das conclusões, as seguintes suposições foram exigidas:

1. **perfil dos participantes**: por ter sido inicialmente conduzido durante uma conferência científica, o perfil de participantes ficou desbalanceado, apresentando principalmente os perfis de *professor* e *pesquisador*. Posteriormente, o *survey* foi disponibilizado para público geral em canal digital e o perfil *usuário* passou a ser perfil predominante nas respostas. Dessa forma, as conclusões traçadas a partir do *survey* podem ter sido impactadas por um viés de um maior número de participantes de um perfil em específico;

2. **falta de generalização**: complementarmente, pelo baixo número de participantes, não foi possível generalizar os resultados com base em cada perfil identificado no estudo, o que limita a obtenção de diferentes visões com base nos dados.

Por fim, no Capítulo 7 foi apresentada uma avaliação experimental a respeito da abordagem apresentada no Capítulo 6. A respeito dos resultados apresentados, as seguintes suposições foram exigidas:

1. **correta implementação da abordagem**: apesar de tratar-se de prova de conceito, para que os resultados tenham validade, assume-se que a implementação da abordagem descrita está correta, ou próxima do correto, de forma que os resultados apresentados refletem de fato aquilo que foi proposto no capítulo anterior;

2. **modelagem de falhas**: uma vez que ainda não há para o contexto de aplicações de RV um conjunto de programas representativo com falhas específicas estudadas nesse trabalho, a avaliação da capacidade de detecção de falhas se suporta na criação de falhas produzidas sinteticamente pelo autor para explorar as características desejadas.

3. **generalização dos resultados**: apesar de haver o cuidado de utilizar aplicações com diferentes níveis de complexidade quanto ao número de objetos e características na cena, o universo de aplicações exploradas permanece baixo, o que compromete a generalização dos resultados apresentados.

## 8.4 Plano de Gestão de Dados

Para além das limitações e suposições adotadas nesse trabalho, este trabalho esteve inserido no contexto de engenharia de software e os artefatos produzidos durante o desenvolvimento do mesmo envolvem a aplicação de práticas de engenharia de software experimental.

Em particular, os requisitos técnicos e as características necessárias para o desenvolvimento do trabalho, e otimização dos resultados são compostos de coleções de artefatos que foram construídos em cada uma das suas etapas distintas e foram divulgados por meio de publicações científicas. De maneira geral os dados produzidos foram compostos de modelos, resultados de experimentações, tabelas e ilustrações para exemplificar simulações e resultados.

O processo de coleta dos dados e produção de artefatos variou de acordo com o objetivo de cada etapa do trabalho, mas, em geral, no contexto de avaliação experimental em engenharia de software, os artefatos utilizados como objetos de estudo foram extraídos a partir de repositórios de código-fonte *open-source* públicos como o *Github* – que é a maior comunidade de desenvolvedores para descobrir, compartilhar e construir softwares de código aberto.

### 8.4.1 Metodologias e padrões utilizados

A metodologia utilizada durante o desenvolvimento do trabalho foi baseada no processo de pesquisa definido por Booth, Colomb e Williams (2009) e é delimitada por meio da definição de questões de pesquisa, que são convertidas em hipótese para posteriormente serem investigadas por meio de estudos experimentais.

O processo de validação das hipóteses foi formulado com base nas diretrizes e *guidelines* propostas por Wohlin *et al.* (2012), que define um processo sistemático a ser seguido durante a produção de experimentos controlados.

### 8.4.2 Condições de compartilhamento dos dados

O autor desta tese de doutorado acredita que a ideia de conhecimento deve estar ao alcance de todos aqueles que desejam apreciá-lo. Portanto, visa proporcionar a divulgação dos resultados científicos em veículos de "acesso aberto", para que qualquer cientista ou cidadão tenha fácil acesso aos resultados apresentados.

Nesse sentido, todos os artefatos produzidos no contexto desse trabalho foram disponibilizados livremente de forma pública no repositório pessoal do autor desta tese na plataforma *Github* [2], assim como todas as publicações geradas a partir dos resultados do trabalho estão disponibilizadas em formato *preprint* no perfil do autor na plataforma *Researchgate* [3] e no

[2] <https://github.com/stevaoaa/>
[3] <https://www.researchgate.net/profile/Stevao_Andrade>

repositório de produção acadêmica/científica da *Universidade de São Paulo* [4].

### 8.4.3 Processo de curadoria e preservação

Além das iniciativas descritas na subseção anterior, conforme apresentado nos capítulos desta tese, todos os dados oriundos dos resultados produzidos também foram disponibilizados por meio de plataforma aberta para disponibilização de dados *Mendeley*[5], que se responsabilizará pela sua segurança e disponibilidade:

- referência para os dados apresentados no Capítulo 4: (ANDRADE; NUNES; DELAMARO, 2021);
- referência para os dados apresentados no Capítulo 5: (ANDRADE *et al.*, 2021);
- referência para os dados apresentados nos Capítulos 6 e 7: (ANDRADE; NUNES; DELAMARO, 2022);

Dessa forma, todos os resultados deste trabalho foram disponibilizados em espaços pessoais do autor da tese, mas também foram replicados em tal plataforma como forma de garantia de preservação e livre acesso por qualquer interessado.

## 8.5 Trabalhos Futuros

Algumas das possíveis extensões e trabalhos futuros para as contribuições de pesquisa desta tese de doutorado incluem: aprimoramento de estudos experimentais para direcionamento de hipóteses de pesquisa no processo de automatização de propriedades específicas de aplicações de RV, aprimoramento na coleta de informações de grupos de interesse sob as percepções observadas, extensão da abordagem de teste para considerar as percepções observadas.

### 8.5.1 Estudos experimentais

Na linha do material apresentado dentro do Capítulo 4, com base no desenvolvimento de novos trabalhos é possível explorar a criação de uma taxonomia de falhas exclusiva para o contexto dos programas de RV.

Por meio de tal taxonomia seria possível catalogar más práticas de programação que contribuem para o surgimento de defeitos, além de estimular o desenvolvimento de técnicas e critérios de teste de software que explorem tipos específicos de falhas, contribuindo, dessa forma, para disseminar a prática de teste de software de forma a mitigar possíveis problemas e avançar em projetos de software para atender aos requisitos de qualidade.

[4] <https://repositorio.usp.br/result.php?filter[]=author.person.name:%22Andrade,%20Stev%C3%A3o%20Alves%20de%22>
[5] <https://data.mendeley.com/>

### 8.5.2 Coleta de novos dados

Conforme apontado nas ameaças a validade apresentadas no Capítulo 5 e nas limitações discutidas acima, aumentar o número de respondedores e ampliar o universo de características exploradas em um *survey* pode permitir uma melhor compreensão e entendimento a respeito das dificuldades enfrentadas por grupos de interesse que lidam com o processo de desenvolvimento e utilização de aplicações de RV.

Atuar nesse sentido permite que a visão dos perfis envolvidos possa ser individualmente discutida em futuros experimentos controlados e estudos de caso.

### 8.5.3 Aprimoramento e expansão da abordagem de teste

Apesar dos resultados encorajadores da abordagem de automatização de testes para aplicações de RV, persiste uma série de desafios que pode contribuir para avanço nessa área de estudos.

Uma possível extensão do trabalho é explorar a utilização da abordagem em outras propriedades de aplicações de RV, destacando-se a proposição de novas relações metamórficas, que sejam capazes de aliviar o problema do oráculo de testes e possibilitar a verificação automatizada com base na geração de dados de teste.

Sob o ponto de vista de melhorias e aprimoramentos nos modelos de aprendizado por reforço, utilizados para a geração dos dados de teste, destaca-se a possibilidade de incorporar novas informações que possam auxiliar no processo de coleta de observações para o agente e, por consequência, na formulação da política do mesmo. Nesse sentido a abordagem pode ser melhorada no que diz respeito à sua capacidade de detecção de falhas.

Sob o ponto de vista técnico, um ponto crucial para facilitar a utilização da abordagem seria a disponibilização da mesma encapsulada no formato de *plugin* para a plataforma *Unity*. Tal iniciativa removeria a barreira da necessidade de ter um conhecimento dos artefatos e facilitaria a sua utilização por terceiros, por meio de uma simples parametrização de configurações específicas.

Por fim, também há espaço para explorar os resultados apresentados com base na realização de experimentos adicionais para verificar o comportamento da abordagem proposta com uma base maior de aplicações, a fim de estabelecer uma noção mais clara a respeito da generalização dos resultados desta tese e por consequência, ter um melhor entendimento de aspectos quanto à escalabilidade da proposta, custo de aplicação, entre outros.

# REFERÊNCIAS

AGARWAL, D.; TAMIR, D. E.; LAST, M.; KANDEL, A. A comparative study of artificial neural networks and info-fuzzy networks as automated oracles in software testing. **IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans**, v. 42, n. 5, p. 1183–1193, Sept 2012. ISSN 1083-4427. Citado na página 50.

ALBAUM, G. The likert scale revisited. **Market Research Society. Journal.**, SAGE Publications Sage UK: London, England, v. 39, n. 2, p. 1–21, 1997. Citado na página 87.

ALONSO, E.; PETER, M.; GOUMARD, D.; ROMOFF, J. Deep reinforcement learning for navigation in AAA video games. In: ZHOU, Z. (Ed.). **Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021**. ijcai.org, 2021. p. 2133–2139. Disponível em: <https://doi.org/10.24963/ijcai.2021/294>. Citado na página 141.

AMMANN, P.; OFFUTT, J. **Introduction to software testing**. [S.l.]: Cambridge University Press, 2016. Citado na página 43.

ANDRADE, A. Game engines: A survey. **EAI Endorsed Trans. Serious Games**, v. 2, n. 6, p. e8, 2015. Citado na página 97.

ANDRADE, S.; NUNES, F. L.; DELAMARO, M. E. **Dataset - On exploiting Reinforcment Learning To Test Virtual Reality Software**. 2022. <doi.org/10.17632/w8bc85x9b4.1>. Acesso: 2022-01-01. Citado nas páginas 116, 138, 139 e 151.

ANDRADE, S. A.; BRITO, C.; JÚNIOR, M.; MARCIEL, A. C.; ABDALLA, G.; DELAMARO, M. E. Analyzing the effectiveness of one-op mutation against the minimal set of mutants. In: **Proceedings of the IV Brazilian Symposium on Systematic and Automated Software Testing**. [S.l.: s.n.], 2019. p. 22–31. Citado na página 147.

ANDRADE, S. A.; NUNES, F. L.; DELAMARO, M. E. A fault-based testing approach to vr applications. In: SBC. **Anais Estendidos do XXI Simpósio de Realidade Virtual e Aumentada**. [S.l.], 2019. p. 5–6. Citado na página 144.

____. **Dataset - Towards the Systematic Testing of Virtual Reality Programs**. 2021. <https://dx.doi.org/10.17632/4myfs585s9.1>. Acesso: 2021-08-26. Citado nas páginas 63, 79 e 151.

ANDRADE, S. A.; NUNES, F. L. S.; DELAMARO, M. E. Towards the systematic testing of virtual reality programs. In: **2019 21st Symposium on Virtual and Augmented Reality (SVR)**. Rio de Janeiro, Brazil: IEEE, 2019. p. 196–205. Citado nas páginas 80, 81 e 145.

ANDRADE, S. A.; QUEVEDO, A. J. U.; NUNES, F. L.; DELAMARO, M. E. Understanding vr software testing needs from stakeholders' points of view. In: IEEE. **2020 22nd Symposium on Virtual and Augmented Reality (SVR)**. [S.l.], 2020. p. 57–66. Citado nas páginas 101 e 146.

ANDRADE, S. A.; QUEVEDO, A. J. U.; NUNES, F. L. S.; DELAMARO, M. E. **Dataset - Understanding VR Software Testing Needs from Stakeholders' Points of View**. 2021. <10.17632/x733f9gy6z.1>. Acesso: 2021-10-01. Citado nas páginas 84, 101 e 151.

ANDRADE, S. A. de; NEVES, V. de O.; DELAMARO, M. E. Software testing education: dreams and challenges when bringing academia and industry closer together. In: **Proceedings of the XXXIII Brazilian Symposium on Software Engineering**. [S.l.: s.n.], 2019. p. 47–56. Citado na página 147.

ANDRADE, S. A. de; NUNES, F. L. S.; DELAMARO, M. E. Towards the systematic testing of virtual reality programs (extended version). **CoRR**, abs/2009.08930, 2020. Disponível em: <https://arxiv.org/abs/2009.08930>. Citado nas páginas 80 e 145.

ANDRADE, S. A. de; SANTOS, Í.; JUNIOR, C. B.; JÚNIOR, M.; SOUZA, S. R. de; DELAMARO, M. E. On applying metamorphic testing: an empirical study on academic search engines. In: IEEE. **2019 IEEE/ACM 4th International Workshop on Metamorphic Testing (MET)**. [S.l.], 2019. p. 9–16. Citado nas páginas 56 e 147.

ANDRADE, S. Alves de; NUNES, F. L. S.; DELAMARO, M. E. A fault-based testing approach for vr applications. **Comunicações em Informática**, v. 3, n. 2, p. 1–4, dez. 2019. Disponível em: <https://periodicos.ufpb.br/index.php/cei/article/view/49451>. Citado na página 144.

ANICHE, M.; BAVOTA, G.; TREUDE, C.; GEROSA, M. A.; DEURSEN, A. van. Code smells for model-view-controller architectures. **Empirical Software Engineering**, Springer, v. 23, n. 4, p. 2121–2157, 2018. Citado na página 66.

ANICHE, M. F.; OLIVA, G. A.; GEROSA, M. A. What do the asserts in a unit test tell us about code quality? a study on open source and industrial projects. In: IEEE. **2013 17th European Conference on Software Maintenance and Reengineering**. [S.l.], 2013. p. 111–120. Citado na página 67.

ANTHES, C.; GARCÍA-HERNÁNDEZ, R. J.; WIEDEMANN, M.; KRANZLMÜLLER, D. State of the art of virtual reality technology. In: IEEE. **2016 IEEE aerospace conference**. [S.l.], 2016. p. 1–19. Citado nas páginas 26 e 32.

ARGELAGUET, F.; ANDUJAR, C. A survey of 3d object selection techniques for virtual environments. **Computers & Graphics**, v. 37, n. 3, p. 121 – 136, 2013. ISSN 0097-8493. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0097849312001793>. Citado na página 93.

ARULDOSS, M.; LAKSHMI, T. M.; VENKATESAN, V. P. A survey on multi criteria decision making methods and its applications. **American Journal of Information Systems**, Citeseer, v. 1, n. 1, p. 31–43, 2013. Citado na página 100.

AVELLAR, G. M.; SILVA, R. F. da; SCATALON, L. P.; ANDRADE, S. A.; DELAMARO, M. E.; BARBOSA, E. F. Integration of software testing to programming assignments: An experimental study. In: IEEE. **2019 IEEE Frontiers in Education Conference (FIE)**. [S.l.], 2019. p. 1–9. Citado na página 147.

BARATLOO, A.; HOSSEINI, M.; NEGIDA, A.; ASHAL, G. E. Part 1: simple definition and calculation of accuracy, sensitivity and specificity. ARCHIVES OF ACADEMIC EMERGENCY MEDICINE (EMERGENCY), 2015. Citado na página 124.

BARBOSA, E. F.; CHAIM, M. L.; VINCENZI, A. M. R.; DELAMARO, M. E.; JINO, M.; MALDONADO, J. C. **Introdução ao Teste de Software – Capítulo 4 - Teste Estrutural**. 1. ed. Rio de Janeiro: Campus, 2016. 59–87 p. Citado na página 44.

BARESI, L.; YOUNG, M. **Test Oracles**. Eugene, Oregon, U.S.A., 2001. <http://www.cs.uoregon.edu/~michal/pubs/oracles.html>. Citado na página 49.

BARR, E. T.; HARMAN, M.; MCMINN, P.; SHAHBAZ, M.; YOO, S. The oracle problem in software testing: A survey. **IEEE Transactions on Software Engineering**, v. 41, n. 5, p. 507–525, May 2015. ISSN 0098-5589. Citado nas páginas 26, 28, 48 e 103.

BASILI, V. R. Goal question metric paradigm. **Encyclopedia of Software Engineering**, John Weily and Sons, p. 528–532, 1994. Citado nas páginas 82 e 139.

BEIZER, B. **Software Testing Techniques**. [S.l.]: Dreamtech, 2003. Citado na página 49.

BEZERRA, A.; DELAMARO, M.; NUNES, F. Definition of test criteria based on the scene graph for vr applications. In: **Virtual Reality (SVR), 2011 XIII Symposium on**. [S.l.: s.n.], 2011. p. 56–65. Citado na página 46.

BIERBAUM, A.; HARTLING, P.; CRUZ-NEIRA, C. Automated testing of virtual reality application interfaces. In: **Proceedings of the Workshop on Virtual Environments 2003**. New York, NY, USA: ACM, 2003. (EGVE '03), p. 107–114. Citado na página 45.

BONACCORSO, G. **Machine learning algorithms**. [S.l.]: Packt Publishing Ltd, 2017. Citado na página 54.

BOOTH, W.; COLOMB, G.; WILLIAMS, J. **The Craft of Research, Third Edition**. [S.l.]: University of Chicago Press, 2009. (Chicago Guides to Writing, Editing, and Publishing). Citado na página 150.

BOSCHE, F.; ABDEL-WAHAB, M. S.; CAROZZA, L. Towards a mixed reality system for construction trade training. **Journal of Computing in Civil Engineering**, v. 30, n. 2, p. 4015016, 2016. Citado na página 34.

BOUZBIB, E.; BAILLY, G.; HALIYO, S.; FREY, P. "can i touch this?": Survey of virtual reality interactions via haptic solutions: Revue de littérature des interactions en réalité virtuelle par le biais de solutions haptiques. In: **32e Conférence Francophone Sur l'Interaction Homme-Machine**. New York, NY, USA: Association for Computing Machinery, 2021. (IHM '21). ISBN 9781450383622. Citado na página 34.

BRISCOE, B.; BRUNSTROM, A.; PETLUND, A.; HAYES, D.; ROS, D.; TSANG, J.; GJESSING, S.; FAIRHURST, G.; GRIWODZ, C.; WELZL, M. Reducing internet latency: A survey of techniques and their merits. **IEEE Communications Surveys & Tutorials**, IEEE, v. 18, n. 3, p. 2149–2196, 2014. Citado na página 97.

BRITO, R. O.; ANDRADE, S. A.; DELAMARO, M. E. Uma avaliaç ao de técnicas e critérios de teste de software para a linguagem de programaç ao python. **Revista Eletrônica de Iniciação Científica em Computação**, v. 20, n. 1, 2022. Citado na página 148.

BURDEA, G.; COIFFET, P. **Virtual Reality Technology**. [S.l.]: Wiley, 2017. (IEEE Press). ISBN 9781119485728. Citado nas páginas 25 e 31.

BURNS, D.; OSFIELD, R. Tutorial: Open scene graph a: introduction tutorial: Open scene graph b: examples and applications. In: **IEEE Virtual Reality 2004**. [S.l.: s.n.], 2004. p. 265–265. ISSN 1087-8270. Citado na página 39.

CAPELLMAN, J.; SALIN, L. Collision detection. In: **MonoGame Mastery**. [S.l.]: Springer, 2020. p. 191–235. Citado na página 108.

CAPILLA, R.; MARTÍNEZ, M. Software architectures for designing virtual reality applications. In: SPRINGER. **European Workshop on Software Architecture**. [S.l.], 2004. p. 135–147. Citado nas páginas 38 e 88.

CARLOZZI, N. E.; GADE, V.; RIZZO, A.; TULSKY, D. S. Using virtual reality driving simulators in persons with spinal cord injury: Three screen display versus head mounted display. **Disability and Rehabilitation: Assistive Technology**, v. 8, n. 2, p. 176–180, 2013. Citado na página 34.

CHAKRABORTY, A.; BAOWALY, M. K.; AREFIN, A.; BAHAR, A. N. The role of requirement engineering in software development life cycle. **Journal of emerging trends in computing and information sciences**, v. 3, n. 5, p. 723–729, 2012. Citado na página 50.

CHANDRA, A. N. R.; JAMIY, F. E.; REZA, H. A review on usability and performance evaluation in virtual reality systems. In: IEEE. **2019 International Conference on Computational Science and Computational Intelligence (CSCI)**. [S.l.], 2019. p. 1107–1114. Citado nas páginas 99 e 107.

CHANG, X.; REN, P.; XU, P.; LI, Z.; CHEN, X.; HAUPTMANN, A. G. A comprehensive survey of scene graphs: Generation and application. **IEEE Transactions on Pattern Analysis and Machine Intelligence**, Institute of Electrical and Electronics Engineers (IEEE), p. 1–1, 2022. Disponível em: <https://doi.org/10.1109%2Ftpami.2021.3137605>. Citado nas páginas 37 e 38.

CHEN, C. J.; LAU, S. Y.; CHUAH, K. M.; TEH, C. S. Group usability testing of virtual reality-based learning environments: A modified approach. **Procedia - Social and Behavioral Sciences**, v. 97, p. 691 – 699, 2013. ISSN 1877-0428. The 9th International Conference on Cognitive Science. Citado na página 46.

CHEN, T.; TSE, T.; ZHOU, Z. Q. Fault-based testing without the need of oracles. **Information and Software Technology**, v. 45, n. 1, p. 1 – 9, 2003. ISSN 0950-5849. Citado na página 51.

CHEN, T. Y.; KUO, F.-C.; LIU, H.; POON, P.-L.; TOWEY, D.; TSE, T. H.; ZHOU, Z. Q. Metamorphic testing: A review of challenges and opportunities. **ACM Comput. Surv.**, ACM, New York, NY, USA, v. 51, n. 1, p. 4:1–4:27, jan. 2018. ISSN 0360-0300. Citado nas páginas 26, 28, 53 e 108.

CHEN, T. Y.; YIU, S. M. **Metamorphic testing: a new approach for generating next test cases**. [S.l.], 1998. Disponível em: <https://www.cse.ust.hk/~scc/publ/CS98-01-metamorphictesting.pdf>. Acessado em: 01/03/2018. Citado na página 51.

CHEN, Y.; GAO, Z.; ZHANG, F.; WEN, Z.; SUN, X. Recent progress in self-powered multifunctional e-skin for advanced applications. In: WILEY ONLINE LIBRARY. **Exploration**. [S.l.], 2022. v. 2, n. 1, p. 20210112. Citado na página 93.

CHIDAMBER, S. R.; KEMERER, C. F. A metrics suite for object oriented design. **IEEE Transactions on software engineering**, IEEE, v. 20, n. 6, p. 476–493, 1994. Citado na página 62.

CHRISTIE, M.; OLIVIER, P.; NORMAND, J.-M. Camera control in computer graphics. In: WILEY ONLINE LIBRARY. **Computer Graphics Forum**. [S.l.], 2008. v. 27, n. 8, p. 2197–2218. Citado na página 111.

CICO, O.; JACCHERI, L.; NGUYEN-DUC, A.; ZHANG, H. Exploring the intersection between software industry and software engineering education-a systematic mapping of software engineering trends. **Journal of Systems and Software**, Elsevier, v. 172, p. 110736, 2021. Citado na página 81.

CORREA, A. C. S.; NUNES, F. L. S.; DELAMARO, M. E. An automated functional testing approach for virtual reality applications. **Software Testing, Verification and Reliability**, Wiley Online Library, v. 28, n. 8, p. e1690, 2018. Citado nas páginas 47, 59 e 99.

CORRÊA, C. G.; DELAMARO, M. E.; CHAIM, M. L.; NUNES, F. L. Software testing automation of vr-based systems with haptic interfaces. **The Computer Journal**, Oxford University Press, v. 64, n. 5, p. 826–841, 2021. Citado na página 47.

COSTA, M. J.; AMALFITANO, D.; GARCÉS, L.; FASOLINO, A. R.; ANDRADE, S. A.; DELAMARO, M. Dynamic testing techniques of non-functional requirements in mobile apps: A systematic mapping study. **ACM Comput. Surv.**, Association for Computing Machinery, New York, NY, USA, dec 2022. ISSN 0360-0300. Just Accepted. Disponível em: <https://doi.org/10.1145/3507903>. Citado na página 148.

COSTA, M. J.; ANDRADE, S.; DELAMARO, M. Automatização de teste de software com ênfase em teste de unidade. 2016. Citado na página 146.

COUNCIL, N. R. *et al.* **Virtual reality: scientific and technological challenges**. [S.l.]: National Academies Press, 1995. Citado na página 38.

CRUZ-NEIRA, C.; FERNáNDEZ, M.; PORTALéS, C. Virtual reality and games. **Multimodal Technologies and Interaction**, v. 2, n. 1, 2018. ISSN 2414-4088. Disponível em: <https://www.mdpi.com/2414-4088/2/1/8>. Citado na página 34.

DAVIS, S.; NESBITT, K.; NALIVAIKO, E. A systematic review of cybersickness. In: **Proceedings of the 2014 Conference on Interactive Entertainment**. [S.l.: s.n.], 2014. p. 1–9. Citado nas páginas 90 e 99.

DELAMARO, M. E.; ANDRADE, S. A.; SOUZA, S. R. de; SOUZA, P. S. de. Parallel execution of programs as a support for mutation testing: A replication study. **International Journal of Software Engineering and Knowledge Engineering**, World Scientific, v. 31, n. 03, p. 337–380, 2021. Citado na página 148.

DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. **Introdução ao Teste de Software – Capítulo 1 – Conceitos Básicos**. 2. ed. Rio de Janeiro: Campus, 2016. 1–7 p. Citado nas páginas 26, 41, 42 e 44.

DELAMARO, M. E.; OLIVEIRA, R. A. P.; BARBOSA, E. F.; MALDONADO, J. C. **Introdução ao Teste de Software – Capítulo 5 – Teste de Mutação**. 2. ed. Rio de Janeiro: Campus, 2016. 93–136 p. Citado na página 44.

DELIGIANNIDIS, L.; JACOB, R. J. Improving performance of virtual reality applications through parallel processing. **The Journal of Supercomputing**, Springer, v. 33, n. 3, p. 155–173, 2005. Citado na página 97.

DEMILLO, R. A.; LIPTON, R. J.; SAYWARD, F. G. Hints on test data selection: Help for the practicing programmer. **Computer**, v. 11, n. 4, p. 34–41, April 1978. Citado na página 44.

DETROZ, J. P.; JASINSKI, M. G.; BOSSE, R.; BERLIM, T. L.; HOUNSELL, M. da S. Virtual reality evolution in brazil: A survey over the papers in the"symposium on virtual and augmented reality". In: IEEE. **2014 XVI Symposium on Virtual and Augmented Reality**. [S.l.], 2014. p. 210–219. Citado na página 26.

DONALDSON, A. F.; EVRARD, H.; LASCU, A.; THOMSON, P. Automated testing of graphics shader compilers. Association for Computing Machinery, New York, NY, USA, v. 1, n. OOPSLA, oct 2017. Disponível em: <https://doi.org/10.1145/3133917>. Citado na página 26.

DURELLI, V. H.; DURELLI, R. S.; BORGES, S. S.; ENDO, A. T.; ELER, M. M.; DIAS, D. R.; GUIMARÃES, M. P. Machine learning applied to software testing: A systematic mapping study. **IEEE Transactions on Reliability**, IEEE, v. 68, n. 3, p. 1189–1212, 2019. Citado na página 103.

DUSTIN, E.; GARRETT, T.; GAUF, B. **Implementing automated software testing: How to save time and lower costs while raising quality**. [S.l.]: Pearson Education, 2009. Citado nas páginas 75 e 99.

DUVAL, T. **Models for design, implementation and deployment of 3D Collaborative Virtual Environments**. Tese (Doutorado) — Université Rennes 1, 2012. Citado nas páginas 38 e 88.

EDVARDSSON, J. A survey on automatic test data generation. In: CITESEER. **Proceedings of the 2nd Conference on Computer Science and Engineering**. [S.l.], 1999. p. 21–28. Citado na página 103.

EUGSTER, P. T.; FELBER, P. A.; GUERRAOUI, R.; KERMARREC, A.-M. The many faces of publish/subscribe. **ACM computing surveys (CSUR)**, ACM New York, NY, USA, v. 35, n. 2, p. 114–131, 2003. Citado na página 115.

FABBRI, S. C. P. F.; VINCENZI, A. M. R.; MALDONADO, J. C. **Introdução ao Teste de Software – Capítulo 2 – Teste Funcional**. 2. ed. Rio de Janeiro: Campus, 2016. 9–38 p. Citado na página 43.

FAROOQ, S. U.; QUADRI, S.; AHMAD, N. Software measurements and metrics: Role in effective software testing. **International Journal of Engineering Science and Technology**, Engg Journals Publications, v. 3, n. 1, p. 671–680, 2011. Citado na página 65.

FAY, M. P.; PROSCHAN, M. A. Wilcoxon-mann-whitney or t-test? on assumptions for hypothesis tests and multiple interpretations of decision rules. **Statistics surveys**, NIH Public Access, v. 4, p. 1, 2010. Citado na página 67.

FELDERER, M.; TRAVASSOS, G. H. **Contemporary Empirical Methods in Software Engineering**. [S.l.]: Springer, 2020. Citado na página 83.

FERREIRA, A. G. **Uma arquitetura para visualização distribuída de ambientes virtuais**. Tese (Dissertação de Mestrado) — Pontifícia Universidade Católica do Rio de Janeiro (PUC/RJ), Rio de Janeiro, RJ, Brasil, Dezembro 1999. Citado na página 37.

FINK, A. **How to ask survey questions**. [S.l.]: Sage, 2002. v. 1. Citado na página 84.

FLORCZYK, M.; WINIECKI, W. The parametric method for functional testing of virtual instruments. In: **2005 IEEE Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications**. [S.l.: s.n.], 2005. p. 310–315. Citado na página 45.

FOTROUSI, F. **Combining user feedback and monitoring data to support evidence-based software evolution**. Tese (Doutorado) — Blekinge Tekniska Högskola, 2020. Citado na página 107.

FOWLER, M. **Refactoring: improving the design of existing code**. [S.l.]: Addison-Wesley Professional, 2018. 464 p. Citado nas páginas 64 e 65.

FRANÇOIS-LAVET, V.; HENDERSON, P.; ISLAM, R.; BELLEMARE, M. G.; PINEAU, J. An introduction to deep reinforcement learning. **arXiv preprint arXiv:1811.12560**, v. 1, p. 1–140, 2018. Citado nas páginas 54, 56 e 114.

GALIN, D. **Software Quality: Concepts and Practice**. [S.l.]: Wiley, 2018. Citado na página 42.

GALLER, S. J.; AICHERNIG, B. K. Survey on test data generation tools. **International Journal on Software Tools for Technology Transfer**, Springer, v. 16, n. 6, p. 727–751, 2014. Citado na página 103.

GAROUSI, V.; MÄNTYLÄ, M. V. When and what to automate in software testing? a multi-vocal literature review. **Information and Software Technology**, Elsevier, v. 76, p. 92–117, 2016. Citado na página 72.

GHRAIRI, N.; KPODJEDO, S.; BARRAK, A.; PETRILLO, F.; KHOMH, F. The state of practice on virtual reality (vr) applications: An exploratory study on github and stack overflow. In: IEEE. **2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)**. [S.l.], 2018. p. 356–366. Citado nas páginas 60, 61 e 104.

GIL, A.; FIGUEIRA, T.; RIBEIRO, E.; COSTA, A.; QUIROGA, P. Automated test of vr applications. In: SPRINGER. **International Conference on Human-Computer Interaction**. [S.l.], 2020. p. 145–149. Citado na página 47.

GOURAUD, H. Continuous Shading of Curved Surfaces. **IEEE Transactions on Computers**, IEEE, Washington, DC, USA, v. 20, n. 6, p. 623–629, 1971. Citado na página 37.

GUO, T.-T.; ZHOU, X.-J.; ZHU, G.-X. Application of cbr in vr-based test and simulation system. In: **Proceedings of the 2003 International Conference on Machine Learning and Cybernetics (IEEE Cat. No.03EX693)**. [S.l.: s.n.], 2003. v. 4, p. 2337–2340 Vol.4. Citado na página 45.

HAIGH-HUTCHINSON, M. **Real time cameras: A guide for game designers and developers**. [S.l.]: CRC Press, 2009. Citado na página 130.

HALL, T.; ZHANG, M.; BOWES, D.; SUN, Y. Some code smells have a significant but small effect on faults. **ACM Transactions on Software Engineering and Methodology (TOSEM)**, ACM New York, NY, USA, v. 23, n. 4, p. 1–39, 2014. Citado nas páginas 65, 71 e 72.

HARMAN, M.; MCMINN, P.; SHAHBAZ, M.; YOO, S. **A Comprehensive Survey of Trends in Oracles for Software Testing**. 2013. Disponível em: <http://mcminn.io/publications/tr3.pdf>. Citado nas páginas 49 e 103.

HAYNES, D. The search for software robustness. In: CITESEER. **Excerpt from Pacific NW Software Quality Conference**. [S.l.], 2009. Citado na página 98.

HE, P.; LI, B.; LIU, X.; CHEN, J.; MA, Y. An empirical study on software defect prediction with a simplified metric set. **Information and Software Technology**, Elsevier, v. 59, p. 170–190, 2015. Citado na página 72.

HEMPE, N. **Bridging the Gap between Rendering and Simulation Frameworks: Concepts, Approaches and Applications for Modern Multi-Domain VR Simulation Systems**. [S.l.]: Springer Fachmedien Wiesbaden, 2016. Citado na página 38.

HERBOLD, S. A systematic mapping study on cross-project defect prediction. **arXiv preprint arXiv:1705.06429**, 2017. Citado na página 72.

HOFFMAN, D. **Using Oracles in Test Automation**. 2001. Disponível em: <https://goo.gl/3bgRvq>. Citado na página 49.

HSU, C. C.-Y.; MENDLER-DÜNNER, C.; HARDT, M. Revisiting design choices in proximal policy optimization. **arXiv preprint arXiv:2009.10897**, 2020. Citado na página 113.

HU, P.; ZHANG, Z.; CHAN, W. K.; TSE, T. H. An empirical comparison between direct and indirect test result checking approaches. In: **Proceedings of the 3rd International Workshop on Software Quality Assurance**. New York, NY, USA: ACM, 2006. (SOQUA '06), p. 6–13. Citado na página 49.

HUGHES, J. F.; DAM, A. V.; MCGUIRE, M.; SKLAR, D. F.; FOLEY, J. D.; FEINER, S. K.; AKELEY, K. **Computer Graphics: Principles and Practice**. 2020. Citado na página 35.

JAHANGIROVA, G. Oracle problem in software testing. In: **Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis**. [S.l.: s.n.], 2017. p. 444–447. Citado na página 103.

JAMIL, M. A.; ARIF, M.; ABUBAKAR, N. S. A.; AHMAD, A. Software testing techniques: A literature review. In: IEEE. **2016 6th international conference on information and communication technology for the Muslim world (ICT4M)**. [S.l.], 2016. p. 177–182. Citado na página 43.

JOHNSON, R. **Statistics: Principles and Methods, 6th Edition**. [S.l.]: Wiley Global Education, 2009. ISBN 9781118049426. Citado nas páginas 123, 135 e 136.

JULIANI, A.; BERGES, V.; VCKAY, E.; GAO, Y.; HENRY, H.; MATTAR, M.; LANGE, D. Unity: A general platform for intelligent agents. **CoRR**, abs/1809.02627, p. 1–28, 2018. Disponível em: <http://arxiv.org/abs/1809.02627>. Citado nas páginas 114 e 115.

JURISTO, N.; MORENO, A. M. **Basics of software engineering experimentation**. [S.l.]: Springer Science & Business Media, 2013. Citado na página 78.

JURISTO, N.; VEGAS, S. Using differences among replications of software engineering experiments to gain knowledge. In: IEEE. **2009 3Rd international symposium on empirical software engineering and measurement**. [S.l.], 2009. p. 356–366. Citado na página 149.

KANER, C.; PADMANABHAN, S.; HOFFMAN, D. **The Domain Testing Workbook**. [S.l.]: Context-Driven Press, 2013. Citado na página 48.

KANEWALA, U.; BIEMAN, J. M. Techniques for testing scientific programs without an oracle. In: **2013 5th International Workshop on Software Engineering for Computational Science and Engineering (SE-CSE)**. [S.l.: s.n.], 2013. p. 48–57. Citado nas páginas 51 e 52.

KARRE, S. A.; MATHUR, N.; REDDY, Y. R. Is virtual reality product development different? an empirical study on vr product development practices. In: **Proceedings of the 12th Innovations on Software Engineering Conference (formerly known as India Software Engineering Conference)**. [S.l.: s.n.], 2019. p. 1–11. Citado na página 64.

KAVANAGH, S.; LUXTON-REILLY, A.; WUENSCHE, B.; PLIMMER, B. A systematic review of virtual reality in education. **Themes in Science and Technology Education**, Themes in Science and Technology Education, v. 10, n. 2, p. 85–119, 2017. Citado na página 25.

KAZMI, R.; JAWAWI, D. N. A.; MOHAMAD, R.; GHANI, I. Effective regression test case selection: A systematic literature review. **ACM Comput. Surv.**, ACM, New York, NY, USA, v. 50, n. 2, p. 29:1–29:32, maio 2017. ISSN 0360-0300. Citado na página 50.

KEOGH, B. Between triple-a, indie, casual, and diy. **The Routledge companion to the cultural industries**, p. 152–162, 2015. Citado na página 93.

KEREN, I. **Scroll Back: The Theory and Practice of Cameras in Side-Scrollers**. 2015. Disponível em: <https://gdcvault.com/play/1022243/Scroll-Back-The-Theory-and>. Acessado em: 10/10/2022. Citado na página 140.

KHOMH, F.; PENTA, M. D.; GUEHENEUC, Y.-G. An exploratory study of the impact of code smells on software change-proneness. In: IEEE. **2009 16th Working Conference on Reverse Engineering**. [S.l.], 2009. p. 75–84. Citado na página 72.

KHOMH, F.; PENTA, M. D.; GUÉHÉNEUC, Y.-G.; ANTONIOL, G. An exploratory study of the impact of antipatterns on class change-and fault-proneness. **Empirical Software Engineering**, Springer, v. 17, n. 3, p. 243–275, 2012. Citado na página 60.

KIM, W. Y.; SON, H. S.; KIM, R. Y. C. A study on test case generation based on state diagram in modeling and simulation environment. In: KIM, T.-h.; ADELI, H.; ROBLES, R. J.; BALITANAS, M. (Ed.). **Advanced Communication and Networking**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. p. 298–305. Citado na página 46.

KNIERIM, P.; SCHWIND, V.; FEIT, A. M.; NIEUWENHUIZEN, F.; HENZE, N. Physical keyboards in virtual reality: Analysis of typing performance and effects of avatar hands. In: **Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems**. [S.l.: s.n.], 2018. p. 1–9. Citado na página 92.

KOSCIANSKI, A.; SOARES, M. dos S. **Qualidade de software: aprenda as metodologias e técnicas mais modernas para o desenvolvimento de software**. [S.l.]: Novatec Editora, 2007. Citado na página 43.

KUMAR, P.; VERMA, J.; PRASAD, S. Hand data glove: a wearable real-time device for human-computer interaction. **International Journal of Advanced Science and Technology**, v. 43, 2012. Citado na página 98.

KUUTTI, K.; BATTARBEE, K.; SADE, S.; MATTELMAKI, T.; KEINONEN, T.; TEIRIKKO, T.; TORNBERG, A. M. Virtual prototypes in usability testing. In: **Proceedings of the 34th Annual Hawaii International Conference on System Sciences**. [S.l.: s.n.], 2001. p. 7 pp.–. Citado na página 45.

LAVALLE, S. M. **Virtual Reality**. [S.l.]: Cambridge University Press, 2019. Citado nas páginas 25, 31 e 91.

LAVIOLA, J. J.; KRUIJFF, E.; MCMAHAN, R. P.; BOWMAN, D.; POUPYREV, I. P. **3D user interfaces: theory and practice**. Boston, USA: Addison-Wesley Professional, 2017. Citado nas páginas 106 e 107.

LESTER, P. A* pathfinding for beginners. **online]. GameDev WebSite. http://www. gamedev. net/reference/articles/article2003. asp (Acesso em 08/02/2009)**, 2005. Citado na página 107.

LEVY, L.; NOVAK, J. **Game development essentials: Game QA & testing**. [S.l.]: Cengage Learning, 2009. Citado na página 88.

LI, L.; YU, F.; SHI, D.; SHI, J.; TIAN, Z.; YANG, J.; WANG, X.; JIANG, Q. Application of virtual reality technology in clinical medicine. **American journal of translational research**, e-Century Publishing Corporation, v. 9, n. 9, p. 3867, 2017. Citado na página 34.

LI, N.; OFFUTT, J. Test oracle strategies for model-based testing. **IEEE Transactions on Software Engineering**, v. 43, n. 4, p. 372–395, April 2017. ISSN 0098-5589. Citado na página 50.

LINÅKER, J.; SULAMAN, S. M.; MELLO, R. Maiani de; HÖST, M. Guidelines for conducting surveys in software engineering. [Publisher information missing], 2015. Citado nas páginas 82, 84 e 101.

LUCK, M.; AYLETT, R. Applying artificial intelligence to virtual reality: Intelligent virtual environments. **Applied artificial intelligence**, Taylor & Francis, v. 14, n. 1, p. 3–32, 2000. Citado na página 96.

LUEBKE, D.; REDDY, M.; COHEN, J. D.; VARSHNEY, A.; WATSON, B.; HUEBNER, R. **Level of detail for 3D graphics**. [S.l.]: Morgan Kaufmann, 2003. Citado na página 140.

MACIEL, F.; LOURENÇO, A.; CARVALHO, P.; MELO, P. Visual and interactive concerns for vr applications: A case study. In: SPRINGER. **International Conference of Design, User Experience, and Usability**. [S.l.], 2017. p. 510–523. Citado na página 98.

MALAN, R.; BREDEMEYER, D. *et al.* Functional requirements and use cases. **Bredemeyer Consulting**, Citeseer, 2001. Citado na página 107.

MALDONADO, J. C. **Critérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software**. Tese (Doutorado) — DCA/FEEC/UNICAMP, Campinas, SP, jul. 1991. Citado na página 44.

MAO, Y.; BOQIN, F.; LI, Z.; YAO, L. Neural networks based automated test oracle for software testing. In: KING, I.; WANG, J.; CHAN, L.-W.; WANG, D. (Ed.). **Neural Information Processing**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006. p. 498–507. Citado na página 50.

MARCHIORI, E.; NIFORATOS, E.; PRETO, L. Analysis of users' heart rate data and self-reported perceptions to understand effective virtual reality characteristics. **Information Technology & Tourism**, Springer, v. 18, n. 1-4, p. 133–155, 2018. Citado na página 90.

MARTZ, P. **OpenSceneGraph Quick Start Guide: A Quick Introduction to the Cross-platform Open Source Scene Graph API**. [S.l.]: Skew Matrix Software, 2007. (The OpenSceneGraph programming series). Citado na página 38.

MCDONALD, J.; HOFFMAN, D.; STROOPER, P. Programmatic testing of the standard template library containers. In: **Proceedings 13th IEEE International Conference on Automated Software Engineering (Cat. No.98EX239)**. [S.l.: s.n.], 1998. p. 147–156. Citado na página 49.

MCMINN, P. Search-based software test data generation: a survey. **Software testing, Verification and reliability**, Wiley Online Library, v. 14, n. 2, p. 105–156, 2004. Citado na página 103.

MEINKE, K.; BENNACEUR, A. Machine learning for software engineering: models, methods, and applications. In: IEEE. **2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)**. [S.l.], 2018. p. 548–549. Citado nas páginas 54 e 103.

Microsoft Corporation. **Microsoft - DirectX graphics and gaming**. 2021. Disponível em: <https://docs.microsoft.com/en-us/windows/win32/directx>. Acessado em: 30/10/2021. Citado na página 35.

MILES, H. C.; WILSON, A. T.; LABROSSE, F.; TIDDEMAN, B.; ROBERTS, J. C. A community-built virtual heritage collection. In: **Transactions on Computational Science XXVI - Volume 9550**. New York, NY, USA: Springer-Verlag New York, Inc., 2016. p. 91–110. Citado na página 34.

MITCHELL, T. **Machine Learning**. [S.l.]: McGraw-Hill, 1997. (McGraw-Hill International Editions). Citado na página 50.

MO, R.; CAI, Y.; KAZMAN, R.; XIAO, L. Hotspot patterns: The formal definition and automatic detection of architecture smells. In: IEEE. **2015 12th Working IEEE/IFIP Conference on Software Architecture**. [S.l.], 2015. p. 51–60. Citado na página 65.

MOHAGHEGHI, P.; CONRADI, R.; KILLI, O. M.; SCHWARZ, H. An empirical study of software reuse vs. defect-density and stability. In: **Proceedings. 26th International Conference on Software Engineering**. [S.l.: s.n.], 2004. p. 282–291. ISSN 0270-5257. Citado na página 36.

MOREAU, G. Visual immersion issues in virtual reality: a survey. In: IEEE. **2013 26th Conference on Graphics, Patterns and Images Tutorials**. [S.l.], 2013. p. 6–14. Citado na página 100.

MURCIA-LÓPEZ, M.; COLLINGWOODE-WILLIAMS, T.; STEPTOE, W.; SCHWARTZ, R.; LOVING, T. J.; SLATER, M. Evaluating virtual reality experiences through participant choices. In: IEEE. **2020 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)**. [S.l.], 2020. p. 747–755. Citado na página 105.

MYERS, G.; SANDLER, C.; BADGETT, T. **The Art of Software Testing**. [S.l.]: Wiley, 2011. (ITPro collection). Citado na página 26.

NAM, J. Survey on software defect prediction. **Department of Compter Science and Engineerning, The Hong Kong University of Science and Technology, Tech. Rep**, 2014. Citado na página 78.

NAQA, I. E.; MURPHY, M. J. What is machine learning? In: **machine learning in radiation oncology**. [S.l.]: Springer, 2015. p. 3–11. Citado na página 54.

NGUYEN, H.; CORPORATION, N. **GPU Gems 3**. [S.l.]: Addison-Wesley, 2008. (Lab Companion Series, v. 3). Citado na página 39.

NUÑEZ-VARELA, A. S.; PÉREZ-GONZALEZ, H. G.; MARTÍNEZ-PEREZ, F. E.; SOUBERVIELLE-MONTALVO, C. Source code metrics: A systematic mapping study. **Journal of Systems and Software**, Elsevier, v. 128, p. 164–197, 2017. Citado nas páginas 60 e 61.

NUSRAT, F.; HASSAN, F.; ZHONG, H.; WANG, X. How developers optimize virtual reality applications: A study of optimization commits in open source unity projects. In: **2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)**. [S.l.: s.n.], 2021. p. 473–485. Citado na página 47.

OLIVEIRA, R. A.; KANEWALA, U.; NARDI, P. A. Chapter three - automated test oracles: State of the art, taxonomies, and trends. In: MEMON, A. (Ed.). **Advances in Computers**. [S.l.]: Elsevier, 2014, (Advances in Computers, v. 95). p. 113 – 199. Citado na página 49.

OLIVEIRA, R. A. P.; MONTEIRO, F. C.; ANDRADE, S. A.; MACIEL, A. C.; FERRARI, F. C.; DELAMARO, M. E. Ferramentas de teste de mutação. 2018. Citado na página 147.

Open Graphics Library. **OpenGL - The Industry's Foundation for High Performance Graphics**. 2021. Disponível em: <https://www.opengl.org/>. Acessado em: 30/10/2021. Citado na página 35.

Oregon State University. **Virtual reality, real injuries: How to reduce physical risk in VR**. 2020. Disponível em: <www.sciencedaily.com/releases/2020/01/200108092448.htm>. Citado na página 93.

ORSO, A.; ROTHERMEL, G. Software testing: a research travelogue (2000–2014). In: **Future of Software Engineering Proceedings**. [S.l.: s.n.], 2014. p. 117–132. Citado na página 27.

OSKAM, T.; SUMNER, R. W.; THUEREY, N.; GROSS, M. Visibility transition planning for dynamic camera control. In: **Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation**. [S.l.: s.n.], 2009. p. 55–65. Citado na página 111.

PALOMBA, F.; BAVOTA, G.; PENTA, M. D.; OLIVETO, R.; LUCIA, A. D. Do they really smell bad? a study on developers' perception of bad code smells. In: IEEE. **2014 IEEE International Conference on Software Maintenance and Evolution**. [S.l.], 2014. p. 101–110. Citado na página 60.

PASZKE, A.; GROSS, S.; MASSA, F.; LERER, A.; BRADBURY, J.; CHANAN, G.; KILLEEN, T.; LIN, Z.; GIMELSHEIN, N.; ANTIGA, L. *et al.* Pytorch: An imperative style, high-performance deep learning library. **Advances in neural information processing systems**, v. 32, p. 8026–8037, 2019. Citado nas páginas 113 e 115.

PEZZè, M.; ZHANG, C. Chapter one - automated test oracles: A survey. In: MEMON, A. (Ed.). [S.l.]: Elsevier, 2014, (Advances in Computers, v. 95). p. 1 – 48. Citado na página 49.

PORCINO, T. M.; CLUA, E.; TREVISAN, D.; VASCONCELOS, C. N.; VALENTE, L. Minimizing cyber sickness in head mounted display systems: design guidelines and applications. In: IEEE. **2017 IEEE 5th international conference on serious games and applications for health (SeGAH)**. [S.l.], 2017. p. 1–6. Citado na página 93.

PRASETYA, I. S. W. B.; SHIRZADEHHAJIMAHMOOD, S.; ANSARI, S. G.; FERNANDES, P. M.; PRADA, R. An agent-based architecture for ai-enhanced automated testing for XR systems, a short paper. **CoRR**, abs/2104.06132, 2021. Disponível em: <https://arxiv.org/abs/2104.06132>. Citado na página 48.

PRESSMAN, R.; MAXIM, D. B. R. **Software Engineering: A Practitioner's Approach**. [S.l.]: McGraw-Hill Education, 2014. ISBN 9780078022128. Citado nas páginas 26 e 42.

RAMAMOORTHY, C. V.; HO, S.-B.; CHEN, W. On the automated generation of program test data. **IEEE Transactions on software engineering**, IEEE, n. 4, p. 293–300, 1976. Citado na página 53.

RAPPS, S.; WEYUKER, E. J. Selecting software test data using data flow information. **IEEE transactions on software engineering**, IEEE, SE-11, n. 4, p. 367–375, 1985. Citado na página 48.

RAU, P.-L. P.; ZHENG, J.; GUO, Z.; LI, J. Speed reading on virtual reality and augmented reality. **Computers & Education**, Elsevier, v. 125, p. 240–245, 2018. Citado na página 93.

REBENITSCH, L.; OWEN, C. Review on cybersickness in applications and visual displays. **Virtual Reality**, Springer, v. 20, n. 2, p. 101–125, 2016. Citado na página 92.

REVIEWS, C. **Computer Science, Overview**. [S.l.]: Cram101, 2016. Citado na página 37.

RICÓS, F. P. Scriptless testing for extended reality systems. In: GUIZZARDI, R.; RALYTÉ, J.; FRANCH, X. (Ed.). **Research Challenges in Information Science**. Cham: Springer International Publishing, 2022. p. 786–794. ISBN 978-3-031-05760-1. Citado na página 48.

RODRIGUEZ, I.; WANG, X. Topic trends in issue tracking system of extended reality frameworks. In: **2021 28th Asia-Pacific Software Engineering Conference (APSEC)**. [S.l.: s.n.], 2021. p. 572–573. Citado na página 27.

SANTOS, A. C. C.; DELAMARO, M. E.; NUNES, F. L. S. The relationship between requirements engineering and virtual reality systems: A systematic literature review. In: **2013 XV Symposium on Virtual and Augmented Reality**. [S.l.: s.n.], 2013. p. 53–62. Citado nas páginas 26, 27 e 59.

SANTOS, S. H.; SILVEIRA, B. N. C. da; ANDRADE, S. A.; DELAMARO, M.; SOUZA, S. R. An experimental study on applying metamorphic testing in machine learning applications. In: **Proceedings of the 5th Brazilian Symposium on Systematic and Automated Software Testing**. [S.l.: s.n.], 2020. p. 98–106. Citado nas páginas 56 e 147.

SARCAR, V. Singleton pattern. In: ____. **Design Patterns in C#: A Hands-on Guide with Real-world Examples**. Berkeley, CA: Apress, 2020. p. 3–26. ISBN 978-1-4842-6062-3. Disponível em: <https://doi.org/10.1007/978-1-4842-6062-3_1>. Citado na página 115.

SCHLUETER, J.; BAIOTTO, H.; HOOVER, M.; KALIVARAPU, V.; EVANS, G.; WINER, E. Best practices for cross-platform virtual reality development. In: INTERNATIONAL SOCIETY FOR OPTICS AND PHOTONICS. **Degraded Environments: Sensing, Processing, and Display 2017**. [S.l.], 2017. v. 10197, p. 1019709. Citado na página 44.

SCHULMAN, J.; WOLSKI, F.; DHARIWAL, P.; RADFORD, A.; KLIMOV, O. Proximal policy optimization algorithms. **CoRR**, abs/1707.06347, 2017. Disponível em: <http://arxiv.org/abs/1707.06347>. Citado nas páginas 113 e 125.

SEGURA, S.; DURÁN, A.; TROYA, J.; CORTÉS, A. R. A template-based approach to describing metamorphic relations. In: IEEE. **Proceedings of the 2$^{nd}$ International Workshop on Metamorphic Testing (MET)**. Buenos Aires, Argentina: IEEE, 2017. p. 3–9. Citado na página 110.

SEGURA, S.; FRASER, G.; SANCHEZ, A. B.; RUIZ-CORTéS, A. A survey on metamorphic testing. **IEEE Transactions on Software Engineering**, v. 42, n. 9, p. 805–824, Sept 2016. ISSN 0098-5589. Citado nas páginas 50, 52 e 53.

SHARMA, T.; MISHRA, P.; TIWARI, R. Designite: A software design quality assessment tool. In: **Proceedings of the 1st International Workshop on Bringing Architectural Design Thinking into Developers' Daily Activities**. [S.l.: s.n.], 2016. p. 1–4. Citado nas páginas 65 e 78.

SILVA, R. J. M.; RAPOSO, A. B.; GATTAS, M. **Grafo de cena e realidade virtual**. Tese (Monografia) — Pontifícia Universidade Católica do Rio de Janeiro (PUC/RJ), Rio De Janeiro, RJ, Brasil, Abril 2004. Citado na página 39.

SJØBERG, D. I.; HANNAY, J. E.; HANSEN, O.; KAMPENES, V. B.; KARAHASANOVIC, A.; LIBORG, N.-K.; REKDAL, A. C. A survey of controlled experiments in software engineering. **IEEE transactions on software engineering**, IEEE, v. 31, n. 9, p. 733–753, 2005. Citado na página 148.

SOMMERVILLE, I. **Software Engineering**. [S.l.]: ADDISON WESLEY Publishing Company Incorporated, 2015. (Always learning). Citado na página 42.

STAATS, M.; WHALEN, M. W.; HEIMDAHL, M. P. E. Programs, tests, and oracles: the foundations of testing revisited. In: **2011 33rd International Conference on Software Engineering (ICSE)**. [S.l.: s.n.], 2011. p. 391–400. ISSN 0270-5257. Citado na página 49.

STRAUSS, P. S. Iris inventor, a 3d graphics toolkit. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 28, n. 10, p. 192–200, out. 1993. ISSN 0362-1340. Citado na página 38.

Sun Microsystems, Inc. **Java 3D API Specification**. 1999. Disponível em: <https://goo.gl/imdxoa>. Acessado em: 06/02/2021. Citado na página 36.

SURYANARAYANA, G.; SAMARTHYAM, G.; SHARMA, T. **Refactoring for software design smells: managing technical debt**. [S.l.]: Morgan Kaufmann, 2014. Citado nas páginas 65 e 69.

SUTCLIFFE, A. **Multimedia and virtual reality: designing multisensory user interfaces**. [S.l.]: Psychology Press, 2003. Citado na página 96.

SUTCLIFFE, A. G.; KAUR, K. D. Evaluating the usability of virtual reality user interfaces. **Behaviour & Information Technology**, Taylor & Francis, v. 19, n. 6, p. 415–426, 2000. Citado na página 99.

SUTTON, R. S.; BARTO, A. G. **Reinforcement learning: An introduction**. [S.l.]: MIT press, 2018. Citado na página 55.

TAHIR, A.; MACDONELL, S. G. A systematic mapping study on dynamic metrics and software quality. In: IEEE. **2012 28th IEEE International Conference on Software Maintenance (ICSM)**. [S.l.], 2012. p. 326–335. Citado na página 61.

The Khronos® Group Inc. **Vulkan cross-platform 3D Graphics**. 2021. Disponível em: <https://www.vulkan.org/>. Acessado em: 30/10/2021. Citado na página 35.

The Mono - Unity fork. **The Mono - Unity fork**. 2021. Disponível em: <https://github.com/Unity-Technologies/mono>. Citado na página 64.

TOBLER, R. F. Separating semantics from rendering: a scene graph based architecture for graphics applications. **The Visual Computer**, Springer, v. 27, n. 6, p. 687–695, 2011. Citado nas páginas 38, 39 e 88.

TORENS, C.; EBRECHT, L. Remotetest: A framework for testing distributed systems. In: **2010 Fifth International Conference on Software Engineering Advances**. [S.l.: s.n.], 2010. p. 441–446. Citado na página 46.

TORI, R.; KIRNER, C.; SISCOUTTO, R. **Fundamentos e Tecnologia de Realidade Virtual e Aumentada**. Belém/PA: VIII Symposium on Virtual Reality, 2006. 412 p. Citado na página 32.

TUFANO, M.; PALOMBA, F.; BAVOTA, G.; PENTA, M. D.; OLIVETO, R.; LUCIA, A. D.; POSHYVANYK, D. An empirical investigation into the nature of test smells. In: **Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering**. [S.l.: s.n.], 2016. p. 4–15. Citado na página 66.

UNITY. **Unity - a platform for creating and operating interactive, real-time 3D (RT3D) content**. 2021. Available from: <https://unity.com/>. Citado na página 115.

Unity Technologies. **Unity Manual: Mesh Collider component reference**. 2022. Disponível em: <https://docs.unity3d.com/Manual/class-MeshCollider.html>. Acessado em: 10/10/2022. Citado na página 139.

VAINSTEIN, N.; KUFLIK, T.; LANIR, J. Towards using mobile, head-worn displays in cultural heritage: User requirements and a research agenda. In: **Proceedings of the 21st International Conference on Intelligent User Interfaces**. [S.l.: s.n.], 2016. p. 327–331. Citado na página 34.

VOSS, G.; BEHR, J.; REINERS, D.; ROTH, M. A multi-thread safe foundation for scene graphs and its extension to clusters. In: **Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization**. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2002. (EGPGV '02), p. 33–37. Citado na página 38.

WALKINSHAW, N.; MINKU, L. Are 20% of files responsible for 80% of defects? In: **Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement**. [S.l.: s.n.], 2018. p. 1–10. Citado na página 75.

WALLIN, R.-L. The use of virtual reality to increase efficiency and profitability in different industries and functions: how head-mounted display can bring value to sectors with its competitive attributes. 2017. Citado na página 34.

WALSH, A. E. Understanding scene graphs. **Dr Dobb's**, v. 01, p. 1–38, 2002. Citado nas páginas 35 e 36.

WANG, F.; WU, J.-H.; HUANG, C.-H.; CHANG, K.-H. Evolving a test oracle in black-box testing. In: **Proceedings of the 14th International Conference on Fundamental Approaches to Software Engineering: Part of the Joint European Conferences on Theory and Practice**

**of Software**. Berlin, Heidelberg: Springer-Verlag, 2011. (FASE'11/ETAPS'11), p. 310–325. Citado na página 50.

WELLER, R. A brief overview of collision detection. **New Geometric Data Structures for Collision Detection and Haptics**, Springer, p. 9–46, 2013. Citado na página 109.

WERNECKE, J. **The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor, Release 2**. 1st. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1993. Citado na página 38.

WOHLGENANNT, I.; SIMONS, A.; STIEGLITZ, S. Virtual reality. **Business & Information Systems Engineering**, Springer, v. 62, n. 5, p. 455–461, 2020. Citado na página 25.

WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLÉN, A. **Experimentation in software engineering**. [S.l.]: Springer Science & Business Media, 2012. Citado nas páginas 77, 78, 79, 100, 121, 138, 139 e 150.

XIAO, G.; SOUTHEY, F.; HOLTE, R. C.; WILKINSON, D. Software testing by active learning for commercial games. In: **Proceedings of the 20th National Conference on Artificial Intelligence - Volume 2**. [S.l.]: AAAI Press, 2005. (AAAI'05), p. 898–903. Citado na página 45.

XIE, X.; HO, J. W. K.; MURPHY, C.; KAISER, G.; XU, B.; CHEN, T. Y. Testing and validating machine learning classifiers by metamorphic testing. **J. Syst. Softw.**, Elsevier Science Inc., New York, NY, USA, v. 84, n. 4, p. 544–558, abr. 2011. ISSN 0164-1212. Citado na página 52.

YANG, J.; QIAN, H. Defect prediction on unlabeled datasets by using unsupervised clustering. In: IEEE. **2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)**. [S.l.], 2016. p. 465–472. Citado nas páginas 72, 73, 74, 78 e 149.

YANG, L.; HUANG, J.; FENG, T.; HONG-AN, W.; GUO-ZHONG, D. Gesture interaction in virtual reality. **Virtual Reality & Intelligent Hardware**, Elsevier, v. 1, n. 1, p. 84–112, 2019. Citado na página 92.

YANG, Y.; YANG, J.; QIAN, H. Defect prediction by using cluster ensembles. In: IEEE. **2018 Tenth International Conference on Advanced Computational Intelligence (ICACI)**. [S.l.], 2018. p. 631–636. Citado nas páginas 73, 78 e 149.

YOSHIMOTO, R.; SASAKURA, M. Using real objects for interaction in virtual reality. In: IEEE. **2017 21st International Conference Information Visualisation (IV)**. [S.l.], 2017. p. 440–443. Citado na página 91.

ZHAO, Q. A survey on virtual reality. **Science in China Series F: Information Sciences**, Springer, v. 52, n. 3, p. 348–400, 2009. Citado na página 25.

ZHAO, X. Y.; XU, L. M.; LI, H. Software testing applications based on a virtual reality system. **Journal of Electronic Science and Technology**, University of Electronic Science and Technology of China, China, v. 5, p. 120–124, 2007. Citado na página 46.


ICMC
USP
SÃO CARLOS