Universidade Federal de Pernambuco
Centro de Informática

Detecção Automática de Erros Visuais em Jogos Love2D, Através de Visão Computacional

Lucas Cavalcanti Calabria

Recife
22 de Setembro, 2023

Universidade Federal de Pernambuco
Centro de Informática

Lucas Cavalcanti Calabria

Detecção Automática de Erros Visuais em Jogos Love2D, Através de Visão Computacional

Trabalho apresentado ao Programa de Graduação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Engenharia da Computação.

Orientador: *Prof. Dr. Breno Alexandro Ferreira de Miranda*

Recife
22 de Setembro, 2023

Ficha de identificação da obra elaborada pelo autor,
através do programa de geração automática do SIB/UFPE

Calabria, Lucas Cavalcanti.
Detecção automática de erros visuais em jogos Love2D, através de visão computacional / Lucas Cavalcanti Calabria. - Recife, 2023.
37p : il., tab.

Orientador(a): Breno Alexandro Ferreira de Miranda
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Informática, Engenharia da Computação - Bacharelado, 2023.

1. Visão Computacional. 2. Detecção de erros. 3. Automação de testes. 4. Love2D. I. Miranda, Breno Alexandro Ferreira de. (Orientação). II. Título.

000 CDD (22.ed.)

Abstract

This work presents a novel approach to automate the detection of visual anomalies in games developed using the Love2D framework. Errors of this nature pose a persistent challenge in the gaming industry, highlighting the need to simplify parts of this process. To address this issue, a two-step method was developed. Firstly, a debugging class captures relevant data from the graphical elements on the screen while also saving a screenshot. Then, using the execution data, this image is recreated and compared to the original to detect the presence of visual anomalies through three developed techniques. When applying the model to a database containing 20 different cases of visual errors, considered relevant to the industry, the effectiveness rate was 93.1%, with no false positives detected.

Keywords: detection of errors, computer vision, test automation, Love2D

Resumo

Este trabalho apresenta uma nova abordagem para automatizar a detecção de anomalias visuais em jogos desenvolvidos no *framework* Love2D. Erros dessa natureza são um desafio persistente na indústria de jogos, evidenciando a necessidade de simplificar parte desse processo. Para enfrentar esse problema, foi desenvolvido um método em duas etapas. Primeiramente, uma classe de depuração captura os dados relevantes dos elementos gráficos em tela, ao mesmo tempo em que é salva uma captura de tela. Em seguida, utilizando os dados de execução, essa imagem é recriada e comparada com a original a fim de detectar a presença de anomalias visuais, através de três técnicas desenvolvidas. Ao se aplicar o modelo em uma base de dados contendo 20 casos diferentes de erros visuais, considerados relevantes para a indústria, a taxa de eficácia foi de 93.1%, sem qualquer caso de falso positivo.

Palavras-chave: detecção de erros, visão computacional, automação de testes, Love2D

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Códigos-Fonte

CAPÍTULO 1

Introdução

A indústria de videogames tem experimentado um crescimento exponencial nas últimas décadas, tornando-se uma popular forma de entretenimento ao redor do mundo [1]. Buscando oferecer experiências imersivas e interativas, esse setor sempre esteve à frente de inovações tecnológicas desde sua concepção há mais de quarenta anos atrás. Contudo, apesar dos crescentes investimentos na área, muitos jogos são lançados com erros, comumente também chamados de *bugs*, e problemas de performance que prejudicam a experiência do usuário, as vendas dos produtos, a imagem da empresa e podem até gerar processos judiciais [2],[3],[4].

Nesse contexto, a linguagem de programação Lua ganha destaque como uma frequente escolha na indústria de desenvolvimento de jogos devido à sua simplicidade, eficiência e extensibilidade [5]. Essas características fazem com que a linguagem seja bastante adaptável a necessidades específicas. Dessa forma é uma boa opção para ser incorporada em diferentes motores gráficos e *frameworks*, fazendo com que se torne frequentemente utilizada no âmbito do desenvolvimento de jogos eletrônicos [6],[7].

Um *framework* popular para o desenvolvimento de jogos em duas dimensões (2D) em Lua é o Love2D, também conhecido como LÖVE. Segundo um levantamento de 2018, essa foi a décima *engine* mais popular, no site *Itch.io*, para o desenvolvimento de jogos de pequeno porte, também chamados jogos *indie*, geralmente desenvolvidos por uma pessoa ou um pequeno grupo de desenvolvedores [8].

Tanto para jogos nesse *framework*, quanto para outros tipos de jogos, a dificuldade de se realizar testes é um desafio inerente à indústria de desenvolvimento de jogos. Diferentemente de muitos outros tipos de software, jogos eletrônicos são sistemas altamente complexos, compostos por um alto número de elementos interativos e gráficos, mecânicas de jogo e conteúdo em constante evolução. Tudo isso torna a detecção de erros um processo complexo, uma vez que os testadores devem não apenas verificar a funcionalidade do jogo, mas também avaliar a experiência do jogador em busca de falhas visuais, problemas de desempenho e comportamentos inesperados [4],[9],[10].

Nesse cenário desafiador, a automação de parte do processo de depuração se revela como uma solução promissora para identificar anomalias visuais. A aplicação de ferramentas automatizadas de teste de detecção de erros pode acelerar o processo de desenvolvimento, reduzir custos, aprimorar a qualidade do produto final e, consequentemente, proporcionar experiências mais satisfatórias aos jogadores [4],[11],[12].

É nesse contexto que é proposto neste trabalho, um método de detecção de erros visuais em jogos Love2D. Com dados capturados durante a execução do jogo de forma automática através de um *script*, diminuindo a necessidade de interações manuais do testador. Para cada amostra capturada, além dos dados, também é salva uma captura de tela contendo todos os elementos gráficos em um determinado instante da execução do jogo. Através da utilização desses dados, um programa na linguagem *Python* utiliza essas informações para recriar a imagem capturada utilizando as mesmas figuras do jogo a ser testado. Em seguida, são utilizadas as imagens original e recriada em uma série de comparações visuais a fim de encontrar discrepâncias entre

elas, que indiquem a presença de defeitos visuais na imagem original capturada da tela do jogo. Esse mesmo processo é repetido para todas as amostras coletadas.

A abordagem proposta neste trabalho oferece uma contribuição para a indústria de desenvolvimento de jogos, pois aborda um desafio persistente e crucial: a detecção de anomalias visuais. A relevância desse método é evidenciada pela sua capacidade de automatizar o processo de depuração de jogos no que tange os erros gráficos, economizando tempo e recursos de desenvolvimento, melhorando a qualidade geral dos jogos e aumentando a satisfação dos jogadores. O fato de usar dados capturados durante a execução do jogo é outro destaque positivo do método proposto, pois torna mais simples a reprodução de defeitos. Além disso, métodos automatizados como o desenvolvido nesse artigo possibilitam um *feedback* mais rápido para os desenvolvedores, fazendo com que problemas possam ser identificados e corrigidos mais rapidamente. Também é importante ressaltar que devido a competitividade do mercado de jogos eletrônicos, lançar um jogo de alta qualidade é essencial para o sucesso. Nesse contexto, o método proposto pode ajudar as empresas a manterem-se competitivas, entregando jogos mais confiáveis e atraentes para os jogadores [13][14].

O método proposto não se limita apenas a etapas específicas do desenvolvimento. Ele pode ser aplicado em versões iniciais e finais do *software*, como também ao longo do processo de desenvolvimento como em testes de regressão, por exemplo. Dessa forma, garantindo que atualizações e modificações futuras não introduzem problemas visuais. Nesse contexto de testes de regressão, a técnica proposta apresenta robustez suficiente para que o testador tenha a liberdade de escolher entre repetir o mesmo percurso de jogabilidade em versões diferentes do jogo ou pode aplicar caminhos e estratégias diferentes. Em ambos os casos o modelo proposto pode ser utilizado, apresentando resultados satisfatórios.

Para a avaliação da eficácia do método foram geradas 20 anomalias visuais diferentes, divididas igualmente nas quatro categorias propostas pela taxonomia de Macklon et Al. [15], a fim de garantir a relevância dos casos de teste para a indústria de desenvolvimento de jogos. As categorias são: erros de estado, aparência, *layout* e renderização. Ao final da etapa de análise dos resultados, constatou-se que o modelo possui uma eficácia de 93,81% ao ser aplicada no conjunto de erros gerados. Também foram realizadas alterações no código do jogo usado como teste para salvar localmente os dados de execução. Esses resultados apontam que a eficácia do modelo torna-o uma ferramenta relevante para a indústria de jogos eletrônicos, com potencial de melhorar a qualidade dos jogos e reduzir os aspectos negativos associados a lançamentos contendo erros visuais.

A fim de permitir a reprodutibilidade do experimento, todos os códigos abertos utilizados para a detecção de *bugs* podem ser encontrados pelo *link* a seguir:

https://github.com/LucasCalabria/gaming-test

A versão do jogo utilizada, com as alterações realizadas para a captura dos dados de execução, podem ser encontradas pelo *link* a seguir:

https://github.com/LucasCalabria/flappybird

CAPÍTULO 2

Contexto

Nesta seção busca-se oferecer um contexto em relação ao desenvolvimento de jogos em Lua, utilizando o *framework* Love2D, e a realização de testes nesse tipo de software. O fato de apresentar um design minimalista e intuitivo, somado a uma comunidade ativa, tem atraído tanto desenvolvedores iniciantes quanto experientes.

2.1 Jogos em Lua (Love2D)

A linguagem Lua é utilizada em diversos projetos globalmente. Entre eles podem ser citados gigantes como *Wikipedia*, *Github* e *Celestia*. Além desses, essa linguagem desempenha um papel significativo no desenvolvimento de jogos, como pode ser evidenciado pelo caso do *Roblox*, que conta com uma base de 48 milhões de usuários ativos diariamente [6],[7].

No contexto deste trabalho, a análise aqui realizada será direcionada para jogos que são construídos utilizando o *framework* de código aberto Love2D. Essa ferramenta proporciona uma abordagem simplificada para a criação de programas computacionais em Lua, especialmente direcionados à concepção de jogos em duas dimensões e aplicações interativas para múltiplas plataformas, incluindo *Microsoft Windows, macOS, Linux, Android* e *iOS*. A API também oferece aos desenvolvedores acesso às funções locais de vídeo e áudio da máquina em que os programas estão rodando através das interfaces SDL e OpenGL [16].

Nesse *framework*, as cenas do jogo, como menus e níveis, podem ser desenhadas com elementos geométricos do módulo gráfico do Love2D. Contudo, para se obter telas de jogo mais elaboradas e com elementos mais complexos, são utilizadas imagens de tipos específicos, como PNG e JPG. Essas são geralmente armazenadas como arquivos independentes em um diretório dedicado à parte gráfica da aplicação. Através de funções oferecidas pela ferramenta, essas imagens são carregadas em memória e podem ser manipuladas, redimensionadas, recortadas ou exibidas em posições específicas da tela do jogo. Na criação de uma cena geralmente várias imagens são posicionadas e combinadas, utilizando parâmetros internos do código, como por exemplo os valores em números de *pixels* do posicionamento das figuras nos eixos vertical (y) e horizontal (x) [17] [18] [19].

A essência de jogos mais complexos em Love2D reside em uma máquina de estados, uma concepção que organiza a lógica do jogo em número finito de diferentes estados, também chamados cenários ou telas, permitindo que a transição entre elas ocorra de forma fluida. Essas mudanças de cenários geralmente ocorrem quando acontece um evento ou o usuário gera alguma entrada específica, como por exemplo pressionar uma tecla. Cada estado, em sua essência, representa um momento diferente e ações tomadas na execução do jogo. Alguns exemplos são: menu principal, níveis do jogo, tela de pausa, etc [20]. Os estados do jogo utilizado para teste do modelo proposto nesse artigo, pode ser observado na figura 2.1.

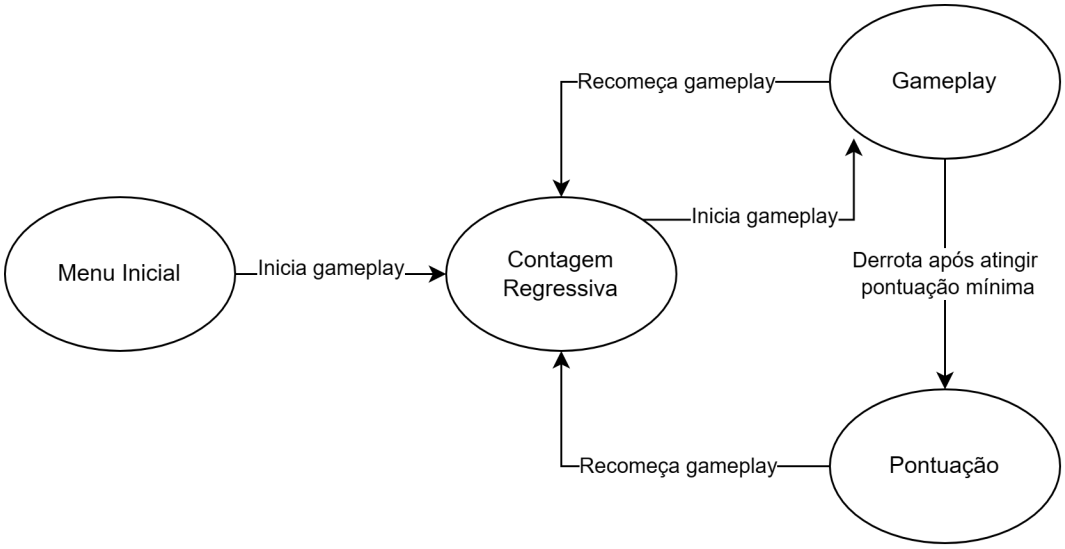


Figura 2.1: Máquina de estados do jogo de teste.

Em relação ao armazenamento de variáveis associadas a cada estado, esse *framework* normalmente recorre a uma estrutura de tabelas que funcionam como uma forma de repositório de dados [21]. Cada estado possui a sua própria tabela local que armazena informações e variáveis específicas para aquele cenário em particular. Nessas tabelas podem ser guardadas informações relevantes à lógica da execução do jogo, como por exemplo: pontuações, status de objetivos e pontos de vida do personagem. Além desse tipo de dados, também são armazenadas informações relevantes na parte gráfica da tela do jogo como posição e orientação de imagens e textos. Durante a execução do jogo, o Love2D é responsável por gerenciar as transições entre os estados e preservar as informações nas tabelas relevantes, permitindo que cada cenário seja capaz de manter seu próprio conjunto de dados sem conflitos com os outros estados.

Em suma, é uma prática comum em jogos em Love2D adotar uma abordagem de máquina de estados para orquestrar a lógica do jogo. Além disso, as variáveis associadas a cada estado são armazenadas em tabelas específicas, garantindo um fluxo suave e apropriado de informações durante a execução da aplicação [21].

2.2 Erros em jogos eletrônicos

Os jogos eletrônicos cada vez mais desempenham um papel significativo na cultura contemporânea. Essa indústria no ano de 2022 foi avaliada em US$ 202,7 bilhões, com a projeção de atingir o valor de US$ 343,6 bilhões em 2028 [1]. Contudo, da mesma forma que os investimentos aumentam, também crescem os casos de jogos que são lançados com os chamados *day-one bugs*, erros claros que poderiam ter sido corrigidos caso mais tempo fosse alocado para a etapa de teste e depuração do *software* [4]. Isso fica claro ao se levar em conta o fato de que um terço dos jogos de maior orçamento para as plataformas *PlayStation 4, Wii U* e *Xbox One*, em 2014, tiveram atualizações para correção de defeitos no mesmo dia de lançamento dos títulos [22].

As diferenças na maneira com que jogos e outros tipos de software são desenvolvidos, pode representar uma explicação para esse problema. Enquanto softwares tradicionais têm uma tarefa específica que devem realizar, a principal função de um jogo é entreter o jogador [12]. Essas diferenças nos processos de desenvolvimento foram primeiramente constatadas por Murphy-Hill et al. que conduziram um estudo empírico, entrevistando desenvolvedores de jogos. Os autores concluíram que estes profissionais acreditam que seus projetos utilizam metodologias ágeis

mais frequentemente, necessitam de equipes mais diversificadas e requerem mais habilidades comunicativas com não engenheiros, quando comparados com desenvolvedores de *software* tradicionais. Um exemplo disso, é o fato de que na criação de jogos, existe uma participação significativa de *designers* e artistas no desenvolvimento dos aspectos visuais do produto [10].

Muitos são os fatores que tornam a depuração de jogos uma atividade mais complexa do que a testagem em outros tipos de programas. Um exemplo claro é a recriação de erros. Em jogos eletrônicos, o estado do jogo que gerou um determinado erro pode ser bastante específico em relação a um tipo de interação do usuário. As propriedades desse estado podem não ser tão evidentes ao desenvolvedor que necessita reproduzir o problema a fim de encontrar a solução [12]. Dessa forma, técnicas como a proposta nesse artigo que mantém salvas as informações do estado atual da execução do jogo no momento em que um erro foi detectado, buscam mitigar o problema da reprodutibilidade.

A presença de defeitos em jogos eletrônicos pode resultar em consequências significativas que afetam tanto a experiência do usuário quanto a imagem e receita das empresas que desenvolveram o produto [23],[24],[25],[26]. Em relação a experiência dos usuários, erros podem ter impactos negativos significativos. Falhas como texturas ausentes ou mal renderizadas, modelos tridimensionais distorcidos, animações defeituosas e qualquer outro tipo de efeito visual que não corresponda ao esperado, podem comprometer a imersão e a satisfação do jogador com o produto. O conceito de imersão é frequentemente utilizado em relação aos jogos virtuais que, de acordo com Ermi and Mayra, pode ser descrito como "se tornar fisicamente ou virtualmente parte da experiência" [27]. Portanto, no que tange à imersão dos jogadores, defeitos visuais podem prejudicar bastante a profundidade da experiência dos usuários. Por conseguinte, a presença de *bugs* pode levar a avaliações e comentários negativos por parte de críticos e consumidores. Dessa maneira, deteriorando a reputação do jogo e dos desenvolvedores, assim como também diminuindo a probabilidade de recomendação a outros potenciais jogadores [23],[28].

Além dos impactos negativos diretos na experiência dos usuários, os *bugs* também podem ter implicações financeiras substanciais para os desenvolvedores de jogos. A indústria de jogos eletrônicos é altamente competitiva, com grandes lançamentos acontecendo com bastante frequência e os consumidores têm expectativas elevadas em relação à qualidade e funcionalidade dos produtos que adquirem [13]. Quando um jogo é lançado com muitos erros, os desenvolvedores podem enfrentar críticas severas e reações negativas por parte dos jogadores, o que pode acarretar em redução do número de vendas e por conseguinte, prejuízos financeiros [24],[25],[29].

Além de todas as consequências negativas citadas previamente, a necessidade de corrigir e atualizar jogos após o lançamento a fim de resolver problemas pode resultar em despesas extras para os desenvolvedores. Tempo e dinheiro que poderiam ser alocados para melhorias de funcionalidades, desenvolvimento de expansões ou novos projetos, são alocados para a correção de erros que muitas vezes poderiam ter sido identificados e corrigidos se uma etapa de testes mais robusta e rigorosa tivesse sido aplicada antes do lançamento do produto [30].

Em resumo, a mitigação de defeitos em jogos eletrônicos é de suma importância, não apenas para garantir a satisfação e qualidade da experiência dos jogadores, como também para salvaguardar a reputação e saúde financeira dos desenvolvedores. Estratégias eficazes de controle de qualidade, testes abrangentes e automação da detecção de erros são algumas das abordagens cruciais para minimizar os riscos associados aos defeitos em jogos eletrônicos, assegurando uma experiência positiva para os usuários e preservando o sucesso comercial dos desenvolvedores.

2.3 Testes em jogos eletrônicos

Nos últimos anos, o desenvolvimento de jogos eletrônicos tem experimentado um crescimento exponencial em termos de complexidade, recursos gráficos e interatividade. Esse fato, somado com o aumento do orçamento dos maiores projetos da área, levaram a uma crescente demanda por técnicas de depuração mais abrangentes e eficazes, a fim de garantir a qualidade e estabilidade dos produtos. Tradicionalmente esses testes são conduzidos de forma manual, envolvendo testadores humanos que exploram várias partes do jogo, procurando erros e problemas de jogabilidade. Contudo, essa abordagem manual apresenta desafios significativos à medida que os jogos se tornam mais intrincados em sua programação e *design* [4].

O principal objetivo de realizar testes em jogos eletrônicos é, além de verificar se o programa está funcionando da maneira esperada, também verificar que o código não está causando consequências inesperadas. Além disso, ao se desenvolver os jogos com técnicas de depuração em mente, geralmente faz com que o desenvolvedor tenha uma ideia mais clara de qual a função e o comportamento esperado de cada parte do programa, o que leva a diminuição do número de erros [9].

Apesar da clara necessidade da realização de testes durante o desenvolvimento de jogos, ainda há um longo caminho a ser percorrido nesse campo para que técnicas de testagem sejam amplamente utilizadas por desenvolvedores dessa indústria. Nesse contexto, a criação de novas abordagens se faz necessária, tendo em vista que a maioria dos atuais modelos propostos estão mais focados no desempenho de modelos de aprendizado de máquinas do que no objetivo de teste. Além disso, para os profissionais existe a necessidade de novas técnicas de teste que não interrompam o fluxo de trabalho do desenvolvedor [31].

2.4 Automatização de testes em jogos eletrônico

Diante dos desafios inerentes aos testes manuais em jogos eletrônicos, a automação dessa etapa do processo de desenvolvimento surgiu como uma solução promissora para melhorar a eficiência, precisão e abrangência dos testes em jogos. Além disso, esse tipo de técnica também busca solucionar o problema da robustez necessária para que os testes manuais apresentem resultados eficazes [11]. A automação desses testes envolve o uso de ferramentas e técnicas para que a testagem ocorra com baixa interação dos desenvolvedores e de forma sistemática e repetível. No contexto de jogos utilizando o Love2D, testes dessa natureza podem oferecer a capacidade de detectar de forma autônoma erros visuais, sonoros e funcionais, bem como problemas de desempenho, ao combinar análise de imagens e de som, com dados coletados durante a execução do jogo por um testador.

Apesar de todos os pontos positivos de se encontrar uma forma de tornar essa etapa mais automatizada, existem alguns pontos que dificultam o desenvolvimento e adoção de técnicas dessa natureza. Um dos principais obstáculos é a aleatoriedade inerente desse tipo de programa. As partes não determinísticas de um jogo como, uso de múltiplas threads, tomadas de decisões por inteligências artificiais e a própria aleatoriedade de certos componentes, como rolagens de dados por exemplo, faz com que seja mais complexo determinar qual o comportamento correto esperado e, caso algum erro seja detectado, como reproduzi-lo [4],[12].Outro ponto importante no teste de jogos é medir o aspecto da diversão proporcionada por um jogo [9]. Pela própria natureza subjetiva do tópico, a diversão é algo que testes automatizados são incapazes de mensurar [4].

A automação dos testes pode ser usada, não apenas para realizar de forma independente

os tipos de testes que já são realizados por seres humanos hoje, como também permitem a implementação de novas técnicas que não são viáveis quando se trata de testadores interagindo com o jogo. Um exemplo é o chamado teste de imersão. Com a utilização de scripts é possível deixar o jogo rodando por longos períodos de tempo de forma ininterrupta e assim encontrar erros, que normalmente não seriam detectados, como por exemplo vazamentos de memória e perdas de taxa de quadros por segundo [9].

2.5 Tipos de técnicas de automação de testes

Os testes automáticos de jogos podem ser divididos em três categorias principais, em relação com o seu escopo. No caso mais restringido, estão os testes de unidades. Nesse tipo cada módulo do código é testado de forma isolada. Utilizando técnicas de injeção de dependências e dados simulados, é possível executar partes do código do jogo de forma unitária. Essa etapa pode ser bastante custosa e demandar tempo de desenvolvimento, mas é recomendada em casos em que a complexidade do código é bastante elevada. Além disso, ao se desenvolver o jogo com esse tipo de teste em mente, tende a gerar códigos menos acoplados, o que facilita o desenvolvimento e depuração do programa [9].

Em seguida, existem os testes de integração. Esse tipo de teste é similar aos testes unitários. Contudo, diferentes módulos ou partes significativas do sistema são testados ao mesmo tempo. Em outras palavras, a abrangência desse tipo de teste é maior em relação aos de unidade, pois geralmente nenhuma parte do sistema é simulada. Em relação aos testes unitários, essa técnica é bem menos custosa devido ao menor número de restrições proveniente da política de isolamento. Desenvolver jogos com esse tipo de teste em mente permite que os programas sejam criados de forma mais rápida, porém com códigos menos organizados e desacoplados [9].

Por fim, o terceiro tipo de teste são os testes de fumaça. Nesse caso todo o sistema, ou ao menos a maior parte dele, é testado de forma simultânea. Também chamado de teste de compilação completa, esse é o caso menos custoso e que pode ser realizado em jogos que não foram desenvolvidos com técnicas de depuração em mente. Contudo, técnicas dessa natureza podem ser as mais complexas para se iniciar, pois não existem maneiras preconcebidas de que um script interaja com o jogo [9].

No caso do modelo proposto neste artigo, foi implementado um tipo de teste de fumaça por duas razões. Primeiramente, a intenção da técnica aqui proposta é realizar o teste de forma automática de toda a parte gráfica do jogo de forma simultânea. O segundo motivo é o fato de que o jogo escolhido já havia completado a sua fase de desenvolvimento, impossibilitando a implementação de testes de outra natureza. Assim como foi mencionado anteriormente, essa abordagem apresentou alta complexidade no momento inicial de adaptação do código já existente.

CAPÍTULO 3

Metodologia

Nesta seção será tratada a forma com que os erros visuais são capturados e detectados de forma automática pela técnica proposta neste artigo.

3.1 Jogo de teste

Para avaliar o desempenho da técnica proposta neste artigo, foi selecionado um jogo de código livre, escrito na linguagem Lua, utilizando o *framework* Love2D. A máquina de estados do jogo pode ser vista na figura 2.1. O *software* de teste é uma versão do popular jogo *Flappy Bird*, reimaginado em Love2D. Esse jogo é um exemplo icônico do gênero conhecido como *endless runner* (corredor infinito), que ganhou popularidade por sua simplicidade e desafio envolvente [32]. Em uma interface completamente bidimensional, o jogador controla um pássaro que está em movimento constante para a direita. O objetivo é evitar os pares de obstáculos que aparecem em formato de canos verticais, posicionados em diferentes alturas da tela. A jogabilidade é bastante simples. A gravidade atua de forma constante sobre o pássaro, fazendo com que ele caia de forma gradual. Pressionar a barra de espaço no teclado faz com que o pássaro dê um leve salto para cima. Assim, cabe ao jogador cronometrar o momento dos pulos para navegar pelos espaços vazios, pois cada par de canos aumenta a pontuação do jogador. O jogo termina quando ocorre uma colisão do pássaro com o chão ou um dos obstáculos. Um exemplo da tela principal de *gameplay* jogo pode ser visualizado na figura 3.1.

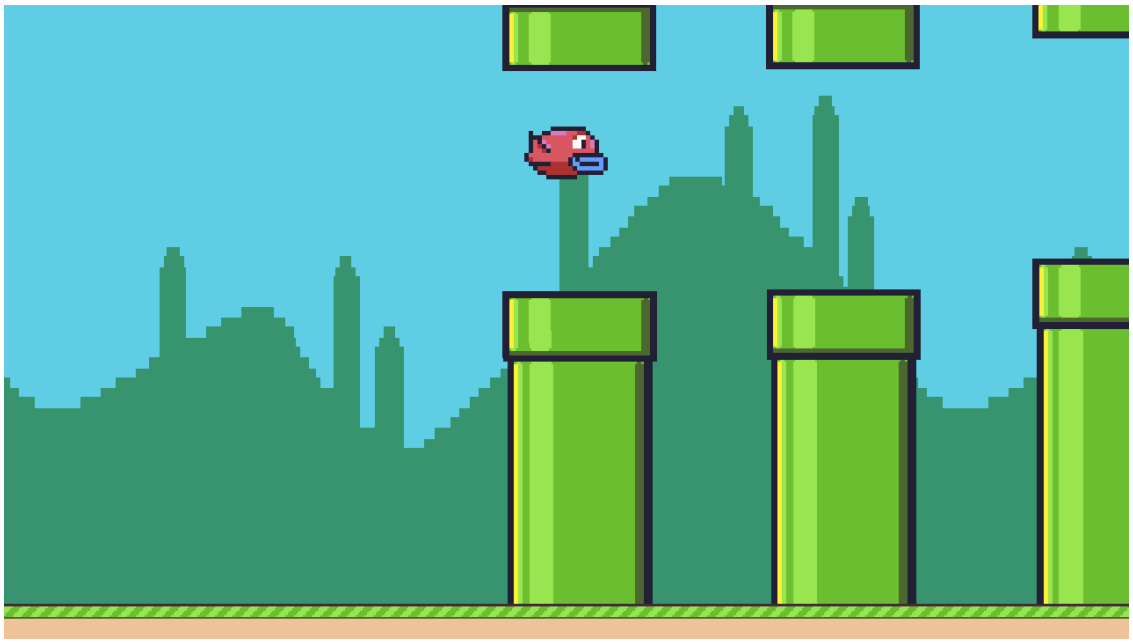


Figura 3.1: Tela principal de *gameplay* do jogo de teste.

Em relação a parte técnica do jogo, esse roda completamente em duas dimensões, com taxa de quadros máxima de 60 FPS, o jogo de teste é executado localmente utilizando aplicação de código aberto *Love*. As dimensões originais da tela do jogo são 1280px por 720px. Contudo, esses valores podem ser redimensionados pelo jogador.

Em relação a execução do jogo para o teste, não é necessária qualquer interferência dos desenvolvedores, além de definir a variável de depuração como verdadeira. Para simular as interações de um jogador, foi desenvolvido um código em *Python*, que reproduz as interações de um jogador, enquanto as amostras são salvas em um intervalo de tempo medido em segundos. Os valores do número de amostras e do intervalo podem ser estabelecidos pelo desenvolvedor. No caso do teste realizado nesse artigo, em cada teste foram coletadas 30 capturas de tela com um segundo de diferença entre elas. Do ponto de vista do testador, a única diferença perceptível é uma leve queda na taxa de quadros por segundo (FPS) no momento em que uma amostra é capturada.

3.2 Captura de dados

A primeira etapa para a detecção do jogo começa com a captura dos dados de execução em tempo real enquanto a aplicação é jogada de forma automática por um *script* em *Python*. Durante o teste, serão salvas capturas de telas de quadros do jogo, assim como todas as variáveis relevantes para a recriação da imagem capturada em etapas posteriores do processo. Todas as informações são gravadas em arquivos *Python* no formato de variáveis, que podem ser facilmente importadas por códigos usando a mesma linguagem, permitindo a reconstrução precisa das condições do jogo em momentos específicos. O número de capturas e o intervalo com que são salvas são pré-determinados pelo desenvolvedor antes do início do teste.

3.2.1 Tipos de dados

Os dados salvos podem ser divididos em duas categorias principais aqui nomeadas variáveis globais e locais. Representações de ambos os tipos de dados podem ser observadas na figura 3.2. O primeiro grupo representa aquelas que se mantêm constantes durante toda a execução da aplicação e não variam entre os momentos de capturas de dados. Exemplos desse tipo de dados são: valores da resolução de tela, número total de amostras coletadas e dicionário contendo informações relevantes sobre todos os tipos de fontes utilizadas, como seus nomes no código do jogo, caminho para o diretório em que estão salvas e tamanhos de fontes. De maneira geral, funções que utilizam variáveis globais são chamadas um número limitado de vezes, a depender do código do jogo. Já os métodos que envolvem variáveis locais são chamados com frequência, a cada iteração do código do jogo são responsáveis por exibir todos os elementos visuais em tela.

As variáveis chamadas locais são aquelas específicas para um momento da execução do jogo e servem para representar as informações visuais que o jogador deveria estar vendo em tela. Um exemplo desse tipo de dados são os prints, que são salvos em forma de dicionário em um arquivo *Python* em uma pasta individual contendo apenas as informações relacionadas a uma amostra específica. Nesse arquivo cada mensagem que aparece na tela está registrado contendo o seu texto, nome da fonte utilizada, alinhamento e posição na tela utilizando as coordenadas horizontal (x) e vertical (y), como base de referência.

Outro tipo relevante de variáveis locais são as imagens que estão na tela do jogo. Essas são divididas em dois tipos: imagens de fundo e imagens de objetos. As imagens de fundo

são aquelas que preenchem completamente uma parte da tela de jogo, no caso da aplicação usada de teste, existem duas imagens desse tipo, a que representa o chão e a que simboliza montanhas e árvores, que podem ser observadas na figura 3.1. Ambas as figuras são salvas em um dicionário contendo o caminho para o diretório onde estão salvas, assim como seu posicionamento nos eixos horizontal e vertical. Em relação aos ícones dos objetos, esses são salvos em outro dicionário no mesmo arquivo *Python* contendo as mesmas informações das figuras de fundo, com a adição de informações indicando se a imagem foi rotacionada ou não e se houve espelhamento, em relação a qualquer um dos dois eixos. Um exemplo de imagem rotacionada são os canos na parte superior da figura 3.1. A mesma imagem é utilizada para todas as instâncias e para representar os que estão invertidos, apenas é feito o espelhamento da imagem em relação ao eixo horizontal.

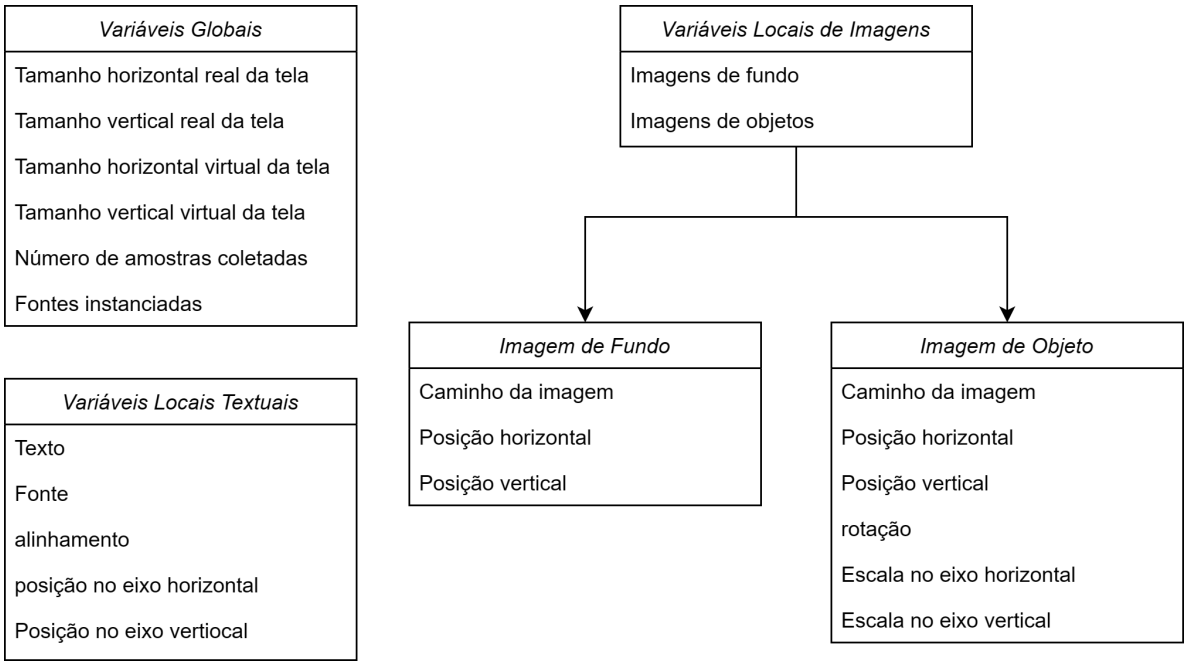


Figura 3.2: Representação dos dados de execução salvos.

3.2.2 Estratégia de captura de dados

Para que a captura dos dados de execução aconteça em tempo real, foram necessárias alterações no código do jogo. Nesse processo, foi prioridade que o número de mudanças necessárias fosse o mínimo possível, a fim de reduzir ao máximo as interferências no ciclo de desenvolvimento do software a ser testado, no caso de se aplicar a técnica proposta neste artigo durante as etapas de criação de um jogo . Dessa maneira a maior parte da lógica de execução responsável pela captura e salvamento dos dados a serem tratados foi restringida a uma única classe nomeada “Debug”. Para iniciar o processo de registro dos dados, o desenvolvedor no início do código apenas deve chamar uma função que define uma variável local da classe a fim de entrar no modo de depuração. É também nessa classe que é definido o número de capturas desejados e o intervalo de tempo entre elas.

Na criação de jogos utilizando o *framework* Love2D, são recorrentemente utilizadas as mesmas funções, como por exemplo a função *love.graphics.printf()* responsável por escrever na tela um texto passado como parâmetro. Para que as informações passadas para esse tipo de método fique registrada, foi criada uma nova função de escrita na classe de depuração que realiza

exatamente o mesmo procedimento da função nativa do *framework*, com a adição da seguinte declaração condicional: casa o jogo esteja em modo de depuração, os parâmetros devem ser salvos em variáveis locais da classe. Métodos dessa natureza são constantemente chamados pelo código do jogo a cada iteração de atualização da tela. Por esse motivo essas variáveis apenas são armazenadas na mesma iteração em que ocorre uma captura de tela. Dessa forma, garantindo que o número de amostras salvas nas tabelas locais é o mesmo de capturas da tela do jogo realizadas. Isso também garante que não haverá vazamento de memória, pois limita o número de entradas nas tabelas de dados locais. Esse novo método no caso da escrita de textos pode ser observado no código-fonte 3.1. A fim da demonstração da tecnologia proposta nesse artigo foram substituídas diversas funções do Love2D para apresentarem comportamentos semelhantes ao mencionado anteriormente.

```
function printf(text, font, x, y, limit, align)
    limit = limit or 'none'
    align = align or 'none'

    if TAKING_SAMPLE then
        table.insert(print_texts, text)
        table.insert(print_fonts, font)
        table.insert(print_xs, x)
        table.insert(print_ys, y)
        table.insert(print_aligns, align)
    end

    love.graphics.setFont(Debug:getVariableObj(font))

    return love.graphics.printf(text, x, y, limit, align)
end
```

Código-fonte 3.1: Método em Lua para escrever na tela de jogo, salvando os dados passados para a função.

As funções que foram tratadas podem ser divididas em dois grupos: funções de texto e funções de imagens. Os métodos relacionados à tratativa de textos foram o registro de novas fontes, salvos como variáveis globais e as funções de definição da fonte a ser utilizada e de escrever o texto como variáveis locais. A fim de exemplificação, a função desenvolvida para realizar o registro de novas fontes pode ser visualizada no código-fonte 3.2. Em relação às imagens, o método desenvolvido de criação de instâncias de novas imagens não registra seus parâmetros e o método de colocar as imagens na tela foi dividido em dois métodos, um salva os dados globalmente como imagens de fundo e o outro guarda como dados locais os ícones dos objetos. O método responsável por salvar na memória os dados das imagens de fundo pode ser visualizado no código-fonte 3.3.

```
function newFont(name, path, size)
    if DEBUGGING then
        table.insert(fonts_names, name)
        table.insert(fonts_paths, path)
        table.insert(fonts_sizes, size)
    end

    return love.graphics.newFont(path, size)
end
```

Código-fonte 3.2: Método em Lua para registrar nova fonte, salvando os dados passados para a função.

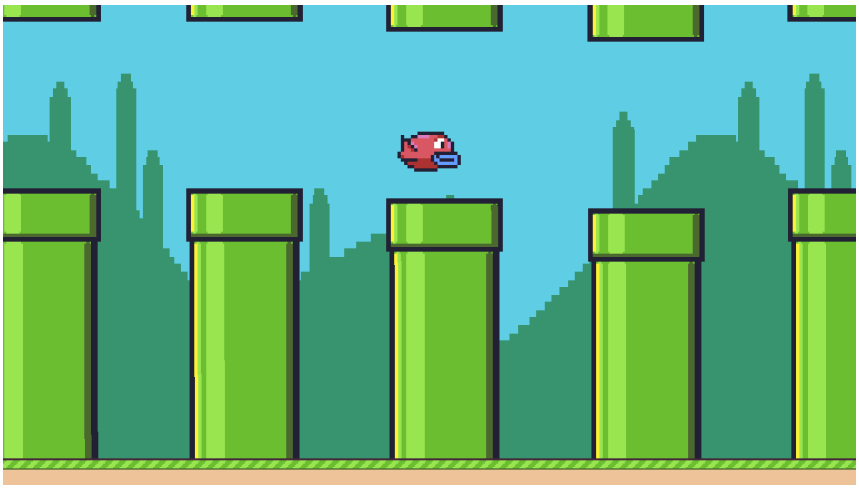
```
function Debug:write_background()
    if TAKING_SAMPLE then
        local path_file = FILE_NAME  ..
            tostring(numeroPrint) ..  "/images.py"

        local images = "background_images = ["

        for i=1,table.getn(background_paths) do
            images = images .. "\n\t{\n\t\t'path': '"
                .. background_paths[i] .. "',\n\t\t"

            images = images .. "'x': '"
                .. background_xs[i] .. "',\n\t\t"

            images = images .. "'y': '"
                . background_ys[i] .. "'\n\t},"
        end

        images = images .. "\n]\n"

        Debug:write(path_file, "w", images)

        background_paths = {}
        background_xs = {}
        background_ys = {}
    end
end
```

Código-fonte 3.3: Método em Lua para salvar um arquivo contendo os dados de execução relativos as imagens de fundo.

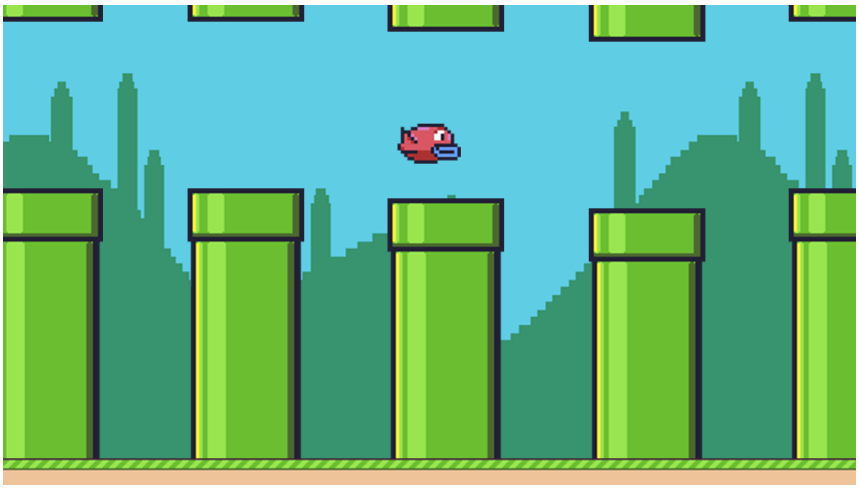
O último conjunto de funções da classe são as funções responsáveis por escrever todas as informações capturadas que até o momento estavam salvas em memória, como variáveis

locais da classe. Para que os desenvolvedores não necessitem chamar cada método de forma individual, também foi criada uma função que, ao ser chamada uma única vez, registra em arquivos no formato *Python* todas as variáveis locais e globais que estão salvas na memória. Todos os métodos dessa natureza são chamados apenas uma vez ao fim da captura de todas as amostras de teste. Dessa forma, garantindo que o número de amostras coletadas é o determinado pelo desenvolvedor antes do início dos teste.

Por fim, também foi desenvolvido um método responsável por realizar as capturas de tela. É importante ressaltar que existe um leve atraso entre a captura dos dados de execução e da imagem da tela, como pode ser observado nas figuras 3.3a e 3.3b. Esse fato deve ser levado em consideração na etapa de comparação das imagens. A quantidade de vezes que esse método é chamado durante a execução do teste do jogo é determinada pelo número de amostras a serem coletadas.



(a) Imagem capturada original.



(b) Imagem recriada.

Figura 3.3: Atraso entre a imagem capturada e a recriada.

3.3 Recriação das imagens

A fim de recriar as imagens capturadas a partir dos dados salvos durante a execução do jogo anteriormente, foi desenvolvida uma aplicação na linguagem *Python*. Com esse objetivo,

foi utilizada a biblioteca *Python Imaging Library* (PIL), que oferece suporte para a abertura, manipulação e salvamento de diversos formatos de arquivos de imagens para códigos dessa linguagem de programação [33]. Também foi usada em conjunto a biblioteca OpenCV que fornece funções para a utilização de técnicas de visão computacional em tempo real para códigos em *Python* [34].

Primeiramente são importadas as variáveis globais e para cada amostra são importadas as locais. Em seguida é criada a tela utilizando as mesmas dimensões da tela do jogo. Na etapa seguinte são coladas na telas as imagens, na ordem que foram utilizadas pelo código do jogo, pois como muitas vezes as imagens estão se sobrepondo, a ordem de colagem é fundamental para garantir a fidelidade da recriação em relação a imagem original. Para se selecionar a figura adequada, é utilizado o caminho da imagem salvo no dicionário contendo os dados de execução capturados nas etapas anteriores.

Em relação ao posicionamento das figuras na tela foi levado em consideração a proporção entre a resolução real e virtual da tela, pois essa técnica também chamada de renderização em escala, é comumente utilizada em jogos Love2D. Como essa prática consiste em o conteúdo gráfico ser criado em uma resolução menor (resolução virtual) e, em seguida, escalado para se ajustar à resolução real da tela onde o jogo está sendo executado, é necessário utilizar a proporção entre as diferentes resoluções na recriação da imagem. Nesta etapa existe uma diferença no tratamento das imagens de fundo e de objeto. As contidas no primeiro grupo são redimensionadas levando em conta a dimensão vertical da tela como limite, pois no jogo utilizado como teste, as imagens de fundo são maiores do que a tela no eixo horizontal e simulam um movimento constante no sentido da direita para a esquerda. Por fim, levando em consideração todos os pontos levantados anteriormente, é possível encontrar as novas medidas que cada imagem na tela deve possuir e, utilizando a biblioteca PIL, é possível fazer o redimensionamento adequado de cada figura. É importante ressaltar dois pontos importantes nessa etapa do processo. Primeiramente, a forma com que as imagens são redimensionadas pelo Love2D é diferente do PIL e OpenCV. Dessa forma, existe uma leve diferença nas cores entre a imagem original capturada e a recriada pelo código *Python*. O outro ponto relevante é a utilização do modo adequado de tratamento das figuras utilizadas, a fim de manter a transparência, caso haja nos ícones originais, com o objetivo de diminuir o ruído gerado na imagem recriada. A função responsável pelo tratamento, recriação e posicionamento das imagens de fundo pode ser observada no código-fonte 3.4

Após tratar todas as imagens da tela são recriados os textos que foram exibidos durante a execução do jogo. Para cada entrada no dicionário de mensagens, é selecionada, do dicionário de fontes, as informações de qual fonte foi utilizada para cada texto e utiliza-se a classe *ImageFont* da biblioteca PIL para instanciá-la no código de recriação das imagens. Em seguida é tratado o posicionamento de cada frase. Caso haja alinhamento centralizado, ela é colocada no centro da imagem em relação ao eixo horizontal, caso contrário é usado o posicionamento contido no dicionário. Em relação ao eixo vertical, o posicionamento é sempre derivado dos dados de execução. É importante ressaltar, que da mesma maneira que ocorreu no tratamento das imagens, sempre que uma coordenada é proveniente do dicionário de variáveis, é realizado o redimensionamento desses valores levando em conta a proporcionalidade entre a resolução virtual e a real da tela em que ocorreu a execução do jogo. A função responsável pelo tratamento dos textos presentes na tela do jogo no momento da captura, pode ser observada no código-fonte 3.5.

Por fim, a imagem recriada com todos os textos e figuras é salva separadamente em correspondência com qual amostra ela deve representar, a fim de tornar mais intuitiva a compreensão dos resultados dessa etapa do processo. O número de vezes que toda a etapa de recriação das

```
def draw_background_image(canvas, draw, image):
    # Redimensionando posicoes virtuais para valores reais
    x = int(float(image.get('x')) * difSize)
    y = int(float(image.get('y')) * difSize)

    image_path = path_assets + image.get('path')

    # Carrega a figura
    background = Image.open(image_path).convert("RGBA")
    background_ratio = background.width / background.height

    # Calcula as novas dimensoes da figura
    background_height = int(global_variables.WINDOW_HEIGHT - y)
    background_width = int(background_height * background_ratio)

    # Redimensiona a figura
    background_size = (background_width, background_height)
    background = background.resize(background_size)

    # Cola a figura na imagem da tela
    canvas.paste(background, (x, y))

    return canvas, draw
```

Código-fonte 3.4: Método em Python de recriação de imagem de fundo.

imagens é processada, é determinado pela variável global do número de amostras que foram capturadas durante a execução do jogo na etapa de captura de dados. A função responsável pelo controle de todo o processo de recriação das imagens, baseado nos dados de execução, pode ser visualizada no código-fonte 3.6. Como pode ser observado essa função segue todos os passos explicados anteriormente, assim como também realiza a chamada dos métodos dos códigos-fonte 3.5 e 3.4.

3.4 Comparação das imagens e métodos de detecção de erros visuais

Para facilitar a compreensão das comparações realizadas, primeiramente é necessário agrupar cada captura de tela da etapa inicial com os seus respectivos dados registrados e a imagem recriada. Dessa maneira, foi desenvolvido um simples código em *Python* que move essas imagens que originalmente são salvas no diretório de arquivos relacionados ao jogo na pasta de usuário no computador utilizado para a execução. Essa etapa foi realizada dessa maneira, pois ao se tentar incluir esse processo na classe de *Debug* do código em Lua, não foi possível garantir que todas as imagens foram transferidas corretamente para seus respectivos destinos.

No passo seguinte é feita a recriação de todas as imagens a partir dos dados de execução como foi discutido na seção 3.3. Em posse de ambas as imagens, é possível aplicar três métodos de comparação desenvolvidos em *Python* para encontrar possíveis erros visuais.

O primeiro teste realizado é focado majoritariamente nas formas das imagens. Para cada par

```
def draw_texts(draw, prints):
    for text in prints:
        font = None
        for font_input in global_variables.fonts:
            if font_input.get('name') == text.get('font'):
                path_font = path_assets + font_input.get('path')
                font = ImageFont.truetype(
                    path_font, int(float(
                        font_input.get('size')) * difSize))

        if font is None:
            exit('Fonte nao encontrada')

        if text.get('align') == 'center':
            _, _, w, h = draw.textbbox(
                (0, 0),text.get('text'), font=font)

            x = (global_variables.WINDOW_WIDTH - w) / 2
        else:
            x = text.get('x')

        draw.text((x, float(text.get('y')) * difSize),
                  text.get('text'), font=font)
```

Código-fonte 3.5: Método em Python recriação dos elementos textuais da iamgem original.

de imagem capturada e recriada, são extraídos todos os *pixels* que contém diferenças através do cálculo de diferença de fundo. Em seguida, a imagem contendo as diferenças é transformada para tons de cinza. O próximo passo é a realização da binarização para segmentar as áreas de grande diferença sobre a imagem resultante da etapa anterior. Contudo, devido ao alto grau de ruído, se fez necessário aplicar duas técnicas a fim de dirimir a influência dessas anomalias no resultado final. A primeira é a aplicação da operação de erosão para remover pequenos detalhes e em seguida é aplicado um método de abertura para eliminar as regiões menores. Em seguida é calculado a quantidade de *pixels* que restaram na imagem após todo o processo supracitado. Por fim, esse resultado é então comparado com um valor limite estabelecido empiricamente, caso seja maior é considerado que existe anomalia visual na amostra analisada. O método que realiza essa técnica pode ser lido no código-fonte 3.7.

O segundo teste compartilha todos os atributos e características com o primeiro método, com a principal diferença sendo a supressão do passo responsável por transformar a cor da imagem resultante da etapa de cálculo da diferença de fundo. Portanto, a imagem a ser tratada leva em conta as cores de cada pixel em seu tom original na análise. Devido às semelhanças, os valores de limites empíricos definidos não divergiram entres os dois primeiros métodos.

O teste final, chamado de teste de distorção, diferentemente dos anteriores, leva em consideração apenas a imagem original. Esse método é responsável por encontrar variações consideradas anômalas na coloração dos *pixels* na tela. Essa etapa do processo busca encontrar pequenas variações que são imperceptíveis para os métodos anteriores devido aos processos de

```
def create_images():
    for i in range(1, global_variables.SAMPLES_NUM + 1):
        # Importando as variáveis dos módulos
        images = importlib.import_module(
            "data." + str(i) + ".images")
        prints = importlib.import_module(
            "data." + str(i) + ".prints")

        # Criando Canvas
        canvas = Image.new(mode="RGB", size=(
            global_variables.WINDOW_WIDTH,
            global_variables.WINDOW_HEIGHT))

        draw = ImageDraw.Draw(canvas)

        canvas, draw = draw_background_image(
            canvas, draw, images.background_images[0])

        canvas, draw = draw_images(
            canvas, draw, images.images)

        canvas, draw = draw_background_image(
            canvas, draw, images.background_images[1])

        draw_texts(draw, prints.prints)

        canvas.save('data/' + str(i)
                    + '/imagem_recriada-' + str(i) + '.png')
        print("Imagem gerada: " + str(i))
```

Código-fonte 3.6: Método em Python de controle do processo de recriação de todas as amostras capturadas durante a execução de teste do jogo.

eliminação de ruídos aos quais são submetidos. A primeira etapa é a divisão da imagem original em quadrados menores de mesmo tamanho definido empiricamente. Assim, a variação de cores pode ser testada em escala reduzida, permitindo maior grau de rigorosidade. Em seguida, para cada quadrado obtido na etapa anterior, é aplicada uma função que torna a imagem em graus de cinza é calculada a médio do gradiente de magnitude. Caso esse resultado seja superior ao valor estabelecido empiricamente como limite mínimo, a função retorna o resultado positivo para anomalias visuais na imagem. O método que realiza essa técnica pode ser lido no código-fonte 3.8.

A fim de reduzir o número de falsos positivos, o código responsável por aplicar os três métodos apenas considera que existem anomalias visuais caso ao menos um dos testes tenha resultado positivo em no mínimo três amostras consecutivas. Esse limite foi estabelecido empiricamente, como é explicado mais detalhadamente na seção 3.6. A necessidade desse filtro provém da busca por diminuir o número de falsos positivos, desconsiderando resultados posi-

tivos que estão presentes apenas em amostras isoladas. Existem dois principais geradores que explicam esse tipo de erro. Primeiramente, as leves variações de cores entre a imagem original e as figuras provenientes dos métodos utilizados para a criação da imagem de referência. Assim como também, o atraso que ocorre entre a captura da imagem da tela do jogo e o armazenamento dos dados de execução.

Por fim, quando for detectada a presença de anomalias visuais na amostragem como um todo, o programa indica quais amostras foram consideradas que contém erros visuais para cada teste que retornou um resultado positivo. Também são registrados os valores dos resultados de todos os métodos para cada amostra analisada, a fim de facilitar a seleção dos valores dos limites.

```
def compare_all(threshold, threshold_bug, gray=False):
    all_diffs = {}
    bug_diffs = {}

    for i in range(1, global_variables.SAMPLES_NUM + 1):
        images_path = 'data/' + str(i) + '/'
        name_image_original = \
            'imagem_original-' + str(i) + '.png'
        name_image_recreated = \
            'imagem_recriada-' + str(i) + '.png'

        test_image = cv2.imread(
            images_path + name_image_original)
        reference_image = cv2.imread(
            images_path + name_image_recreated)

        diff_image = cv2.absdiff(reference_image, test_image)
        if gray:
            gray_diff = cv2.cvtColor(
                diff_image, cv2.COLOR_BGR2GRAY)

            _, thresholded = cv2.threshold(
                gray_diff, threshold, 255, cv2.THRESH_BINARY)
        else:
            _, thresholded = cv2.threshold(
                diff_image, threshold, 255, cv2.THRESH_BINARY)

        kernel = np.ones((5, 5), np.uint8)
        eroded = cv2.erode(thresholded, kernel, iterations=1)

        opening_kernel = np.ones((10, 10), np.uint8)
        opened = cv2.morphologyEx(
            eroded, cv2.MORPH_OPEN, opening_kernel)

        num_different_pixels = np.count_nonzero(opened)
        all_diffs[i] = num_different_pixels

        if num_different_pixels > threshold_bug:
            bug_diffs[f'{i}'] = num_different_pixels

    return all_diffs, bug_diffs
```

Código-fonte 3.7: Método em Python que aplica os métodos de comparação de forma ou de cores, a depender do valor da variável *gray*.

```
def check_all_distortion(square_size=50,
                         distortion_threshold=95):
    results = []
    results_bug = {}
    for i in range(1, global_variables.SAMPLES_NUM + 1):
        images_path = 'data/' + str(i) + '/'
        name_image_original = \
            'imagem_original-' + str(i) + '.png'

        image = cv2.imread(
            images_path + name_image_original)

        squares = divide_image_into_areas(
            image, square_size)
        has_artifact = detect_distortion_in_squares(
            squares, distortion_threshold)

        results.append(has_artifact)
        if has_artifact:
            results_bug[f'{i}'] = has_artifact

    return results, results_bug
```

Código-fonte 3.8: Método em Python que aplica os métodos de análise de distorção em uma imagem.

3.5 Injeção de erros visuais

A fim de avaliar a robustez e eficácia do sistema de detecção de erros visuais proposto nesse artigo, foram injetados erros visuais de forma deliberada em cenários de teste do jogo. O objetivo principal dessa etapa do processo foi verificar se o sistema é capaz de identificar com precisão e eficiência diferentes tipos de erros visuais que podem ocorrer em aplicações Love2D. Para garantir a relevância dos erros escolhidos para teste, foi utilizada a taxonomia proposta por Macklon et al. [15].

Para cada um dos quatro tipos de erros visuais definidos na taxonomia, foram gerados 5 casos contendo as anomalias. Dessa forma, totalizando 20 diferentes erros visuais foram usados na testagem do modelo. A maior parte desses erros foram injetados diretamente no jogo, alterando a imagem da figura a ser importada pelo Love2D. Um exemplo pode ser visto na figura 3.4.

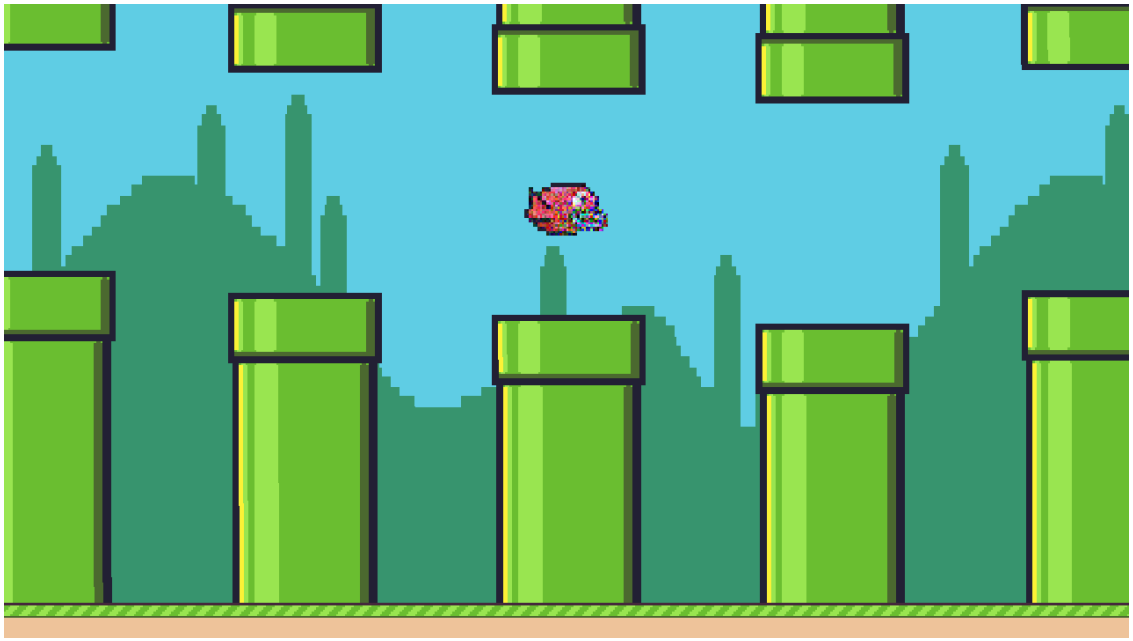


Figura 3.4: Tela do jogo contendo erro de artefatos no pássaro.

3.5.1 Erros visuais de estado

Os dois primeiros cenários de injeção de erros visuais de estado são o pássaro ou um ou mais canos invisíveis. Essas anomalias visam testar a capacidade do sistema de detectar a ausência de elementos visuais na tela. Em seguida foram gerados dois casos que visam avaliar se o sistema é capaz de identificar anomalias nos movimentos regulares do jogo. Dessa forma, foram criados os casos em que a queda do pássaro não atualiza e em que um dos canos não se movimenta para a esquerda. Por fim, foi gerado um caso para testar problemas na exibição de textos na tela. Nesse caso foi avaliado se o sistema consegue detectar a ausência de um elemento textual na tela de jogo.

3.5.2 Erros de aparência

Os casos de erro de aparência relacionados às imagens abrangem as cores de diversos elementos da tela. Em dois casos são injetados elementos na cor errada, o pássaro e os canos. Esses elementos podem ser vistos nas figuras 3.5a e 3.5c. Nos outros dois casos relacionados às figuras, o pássaro e a imagem de fundo contendo as colinas foram alteradas para tons de cinza. O pássaro em escala de cinza pode ser observado na figura 3.5a. Por fim, foi criado um caso em que um dos elementos textuais na tela foi escrito utilizando a fonte errada.

3.5.3 Erros de layout

Em relação às figuras, foram criados dois casos de erro que visam testar a capacidade do sistema de detectar problemas de posicionamento. No caso o pássaro ou um cano é renderizado em uma posição incorreta da imagem. Ainda em relação às figuras, foram testados problemas de orientação inadequada. Em um dos casos o pássaro está rotacionado em 180 graus e no outro caso, o troféu presente na tela de pontuação. Em relação aos textos, foi testado se o modelo proposto é capaz de detectar a escrita de um elemento textual na posição errada.

(a) Pássaro na cor errada. (b) Pássaro na escala de cinza. (c) Cano na cor errada.

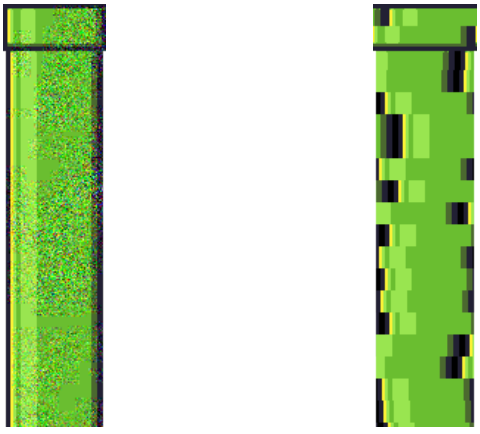
Figura 3.5: Erros de aparência.

3.5.4 Erros de Renderização

Dois cenários de erros de renderização gerados abrangem problemas de distorção, com diferentes graus de intensidade. Esses casos podem ser visualizados na figura 3.6. Além disso, foi gerado um caso que testa a capacidade do sistema de detectar a presença de artefatos em alguma área da imagem. Nesse caso, os artefatos foram introduzidos na figura dos canos, como pode ser visto na figura 3.7a. Esse mesmo elemento visual foi utilizado para gerar casos de *tearing*, em que a imagem apresenta uma aparência rasgada, como pode ser observado na figura 3.7b. Por fim, foi injetado uma anomalia em que ordem de renderização está errada. No caso, os canos foram renderizados sobrepondo a imagem que representa o chão, como pode ser visto na figura 3.8.

(a) Pássaro pouco distorcido. (b) Pássaro muito distorcido

Figura 3.6: Erros de renderização no pássaro.



(a) Cano com artefatos. (b) Cano com *tearing*.

Figura 3.7: Erros de renderização em canos.

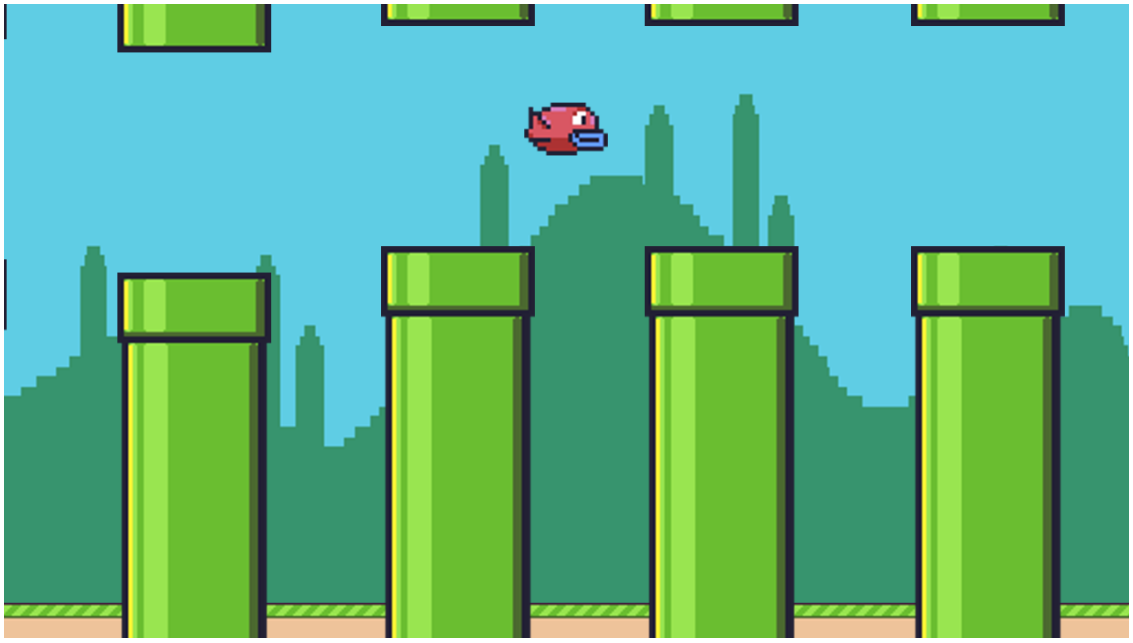


Figura 3.8: Tela do jogo contendo erro de renderização.

3.6 Seleção empírica dos limites

A metodologia adotada para a seleção dos limites foi estruturada de maneira sistemática. Inicialmente, diversos cenários de teste foram preparados, abrangendo tanto situações sem erros visuais conhecidos quanto aquelas que possuíam anomalias dessa natureza. Cada cenário de teste foi submetido ao sistema de detecção apresentado na seção 3.4. Antes de conduzir os testes empíricos, foi necessário estabelecer um conjunto de dados de referência confiável. Para isso, as imagens das amostras dos cenários de teste sem erros visuais foram manualmente avaliadas e consideradas corretas. Dessa maneira, servindo como padrões de comparação.

A etapa seguinte consistiu na execução dos cenários de teste desprovidos de erros visuais pelo sistema de detecção de anomalias visuais. Durante esse processo, os limites de comparação foram gradativamente reduzidos a partir de valores arbitrários iniciais, sendo a incidência de falsos positivos, constantemente monitorada.

Com os valores obtidos na etapa anterior, os valores dos limites a serem utilizados para o jogo em questão foram estabelecidos através da busca de um ponto de equilíbrio entre duas considerações cruciais. Primeiramente a minimização de falsos positivos. O objetivo dessa máxima foi identificar os limiares mínimos que resultaram em uma quantidade desprezível de falsos positivos. Dessa maneira, assegurando que o sistema de detecção não interpretasse variações normais causadas pela aleatoriedade inerente ao jogo ou ruídos gerados pela técnica de recriação de imagem, como erros visuais. O segundo objetivo foi maximizar a sensibilidade à detecção de erros. Esse ponto focou em evitar que os limites escolhidos fossem desnecessariamente baixos, ao ponto que comprometesse a detecção eficaz de erros visuais genuínos. Em suma, o equilíbrio buscado procurou garantir a capacidade de reconhecer apenas anomalias visuais legítimas.

Os experimentos culminaram em uma série de valores de limites que representam um compromisso entre a minimização de falsos positivos e a maximização da sensibilidade à detecção. A escolha desses valores derivou de análises quantitativas e qualitativas dos resultados dos testes empíricos. Os números exatos de cada um desses limites podem ser visualizados na Tabela

3.1. O método desenvolvido nesse artigo apresenta uma grande quantidade de ruído que é introduzido devido ao atraso entre a captura da imagem da tela e o armazenamento dos dados de execução. Dessa maneira, se fez necessário que os valores dos limites fossem bastante específicos para o jogo em questão selecionado para o teste.

Tabela 3.1: Valores empíricos dos limites

Limite	**Valor**
Limite da binarização de imagem	75
Limite do número de pixels diferentes	750
Limite do gradiente médio de distorção	100
Área dos quadrados de distorção	80

O impacto do ruído na recriação das capturas de tela pode ser observado na figura 3.9 que representa a diferença absoluta pixel a pixel entre a imagem capturada sem erros e a imagem recriada. A fim de ilustrar os valores de limite, o erro de artefatos no pássaro da figura 3.10a não é identificado quando o valor do limite médio de distorção é aumentado para 110. A diferença absoluta desse caso pode ser observado na figura 3.10b.

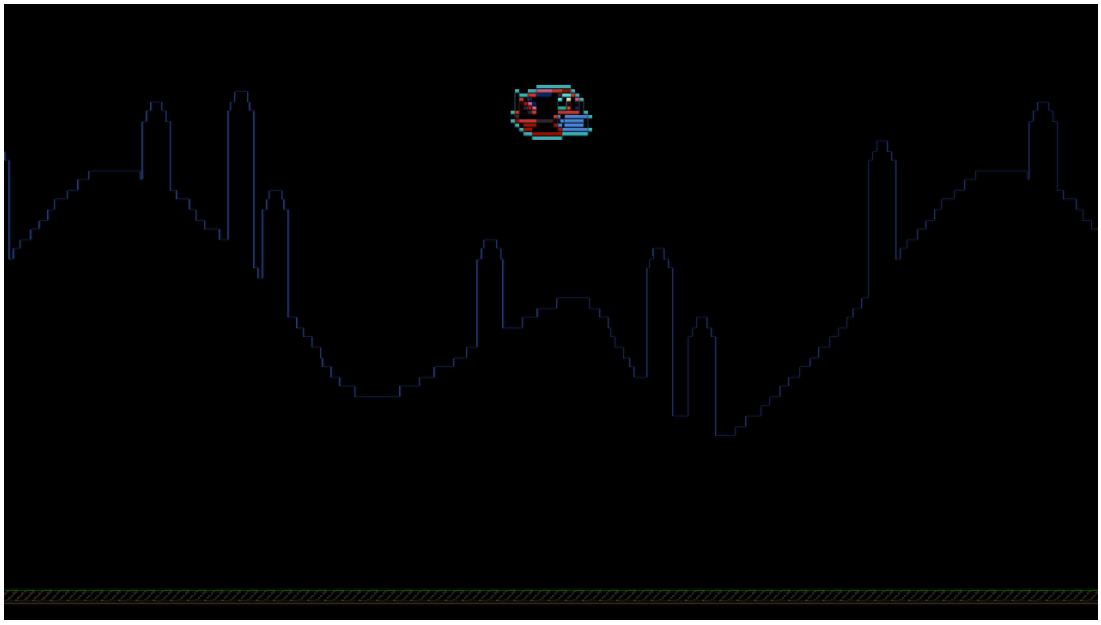
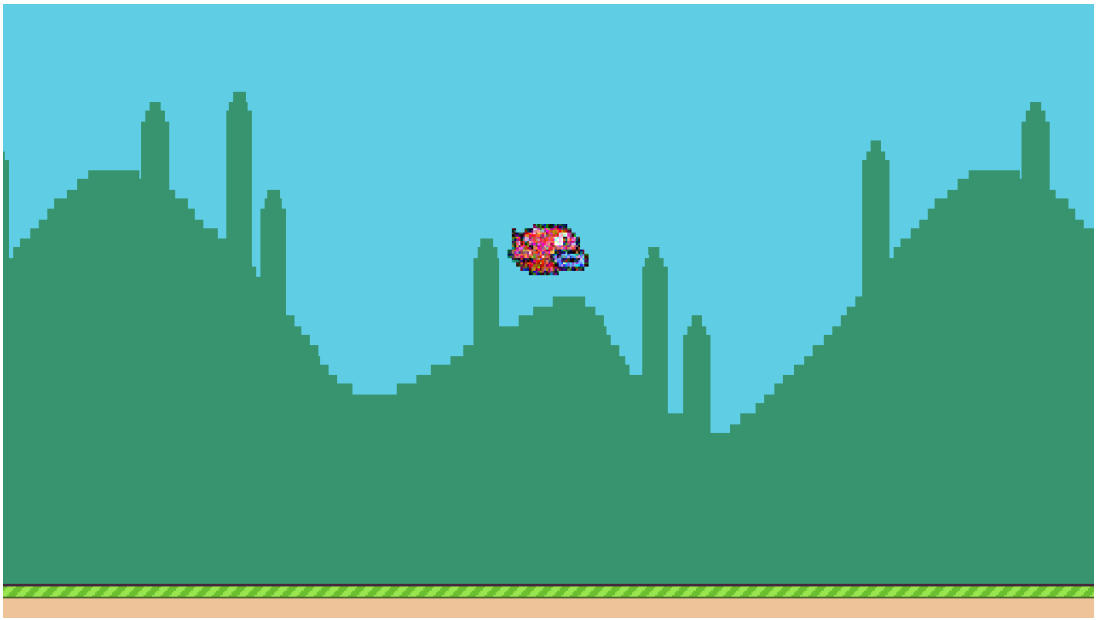
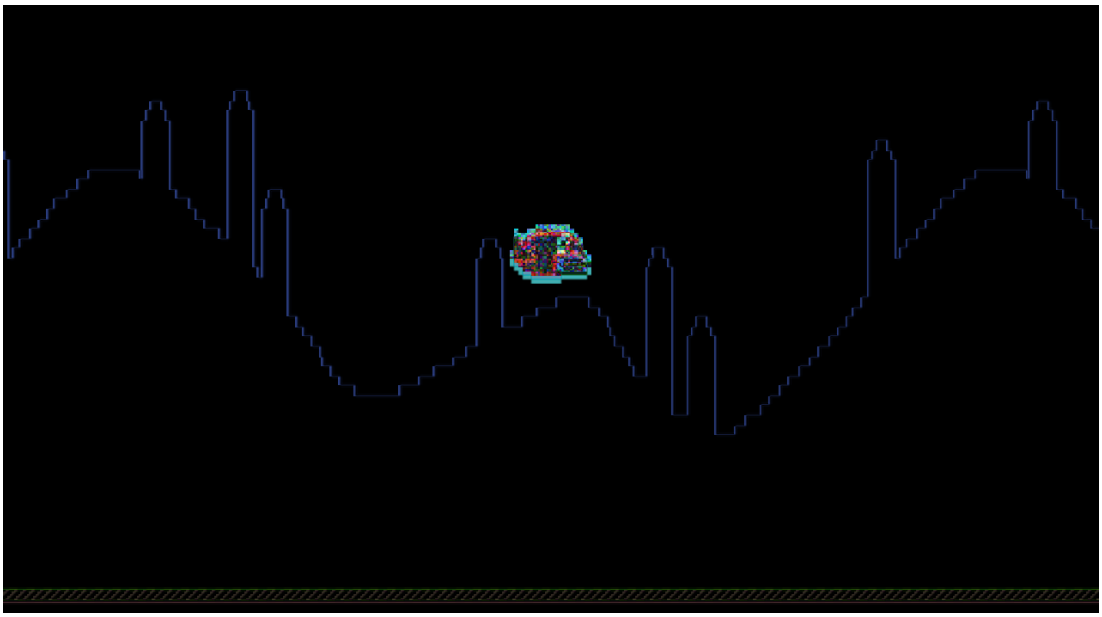


Figura 3.9: Diferença absoluta sem erros.



(a) Tela de jogo contendo artefatos no pássaro.



(b) Diferença absoluta de erros de artefatos no pássaro.

Figura 3.10: Erro de artefatos no pássaro.

CAPÍTULO 4

Resultados

Nesta seção são apresentados os resultados da avaliação do método proposto de detecção de anomalias visuais. É descrita em detalhe a metodologia de teste e os resultados obtidos são apresentados e analisados.

4.1 Metodologia de teste

Para garantir a confiabilidade dos resultados, todos os testes foram conduzidos em um ambiente de execução estritamente controlado, empregando uma configuração de hardware uniforme. Durante todo o processo foram mantidas as mesmas condições de execução a fim de evitar qualquer tipo de interferência externa nos resultados obtidos. Também permaneceram inalteradas as dimensões originais da tela de execução do jogo, provenientes do código-fonte em Lua.

No que diz respeito à injeção dos erros visuais previamente gerados, foram adotadas duas abordagens distintas. Para os casos que envolvem figuras adulteradas, as imagens corretas carregadas pelo *framework* foram substituídas por suas versões alteradas. Quanto aos outros tipos de anomalias visuais, elas foram introduzidas na execução do jogo por meio de modificações pontuais no código-fonte em Lua. Essas escolhas metodológicas asseguraram que os testes fossem executados de forma rigorosa e controlada, permitindo uma avaliação precisa da eficácia do método proposto.

4.2 Execução dos testes

Antes da injeção de erros , foram conduzidas 1- iterações de teste em uma versão do jogo que cuidadosamente depurada para eliminar qualquer tipo de anomalia visual. A integridade dessa versão foi garantida através de uma análise minuciosa de todas as capturas de tela geradas durante as execuções. Em seguida, foi realizada a injeção individual de cada um dos 20 casos de anomalias visuais geradas previamente. Cada versão do jogo contendo um dos casos de erro foi analisada em um total de 10 iterações.

As execuções de teste em duas condições distintas, com e sem anomalias visuais, desempenharam um papel importante como grupo de controle. Elas permitiram não apenas analisar a capacidade do método de encontrar erros visuais, mas também testar a incidência de resultados falsos positivos. Dessa forma, foi possível garantir a precisão e confiabilidade da técnica proposta.

4.3 Resultados obtidos

Após a conclusão de todas as iterações dos casos de teste, tanto aqueles com anomalias visuais quanto aqueles sem, os resultados completos foram compilados e agrupados. Eles podem ser observados na Tabela 4.1, que fornece uma visão abrangente dos dados coletados.

Tabela 4.1: Número de repetições, de 10, em que cada técnica e o modelo retornaram o resultado correto.

Tipo	Descrição	Forma	Cor	Distorção	Modelo
Estado	Pássaro invisível	10	10	0	10
	Cano invisível	10	10	0	10
	Queda do pássaro não atualizando	10	10	0	10
	Canos não se movendo	10	10	0	10
	texto não aparecendo	10	10	0	10
Aparência	Pássaro na cor errada	0	10	0	10
	Canos na cor errada	0	10	0	10
	Pássaro em escala de cinza	0	8	0	8
	Imagem de fundo em escala de cinza	0	10	0	10
	Texto na fonte errada	10	10	0	10
Layout	Pássaro na posição errada	10	10	0	10
	Cano na posição errada	10	10	0	10
	Pássaro na rotação errada	0	0	0	0
	Troféu na rotação errada	0	10	0	10
	Texto na posição errada	9	9	0	9
Renderização	Pássaro muito distorcido	0	0	10	10
	Pássaro pouco distorcido	0	0	10	10
	Ordem errada de renderização	10	10	0	10
	Cano contém artefatos	0	0	10	10
	Cano contém tearing	0	0	10	10
Sem erros visuais		10	10	10	10
Acurácia		**51.90%**	**74.76%**	**23.81%**	**93.81%**

4.4 Análise dos resultados

A métrica central adotada para a análise da eficácia do modelo apresentado foi a taxa de detecção de erros. Este indicador foi considerado o mais apropriado para demonstrar a habilidade do método em identificar problemas visuais com precisão. Além disso, foram avaliados os números de resultados falsos positivos e negativos, o que proporciona indicadores sobre áreas que podem ser beneficiadas por aprimoramentos futuros. Essa análise dos resultados desempenha um papel crucial na avaliação da eficiência e confiabilidade do sistema.

Como pode ser visto na Tabela 4.1 os resultados apresentaram alto grau de variação a depender do tipo de *bug* injetado. De maneira geral, o teste de forma apresentou desempenho de 100% para erros de estado, mas não foi muito consistente para os outros grupos de anomalias visuais. O teste de cor superou os resultados do teste de forma em todas as categorias testadas, com os mesmos 100% de eficácia em erros de estado. Além disso, essa técnica também apresentou bons resultados para testes de aparência e de *layout*. Contudo ambos os testes não

tiveram bons resultados ao tentar detectar erros de renderização. Para erros desse tipo, o teste de distorção obteve o melhor desempenho, com eficácia de 80%. Contudo, esse teste não foi capaz de encontrar erros visuais de qualquer outra natureza. Todos os três tipos de técnicas não geraram qualquer tipo de resultado falso positivo.

Ao contrário das taxas de eficácia de cada técnica de testagem isoladamente, ao se analisar o modelo como um todo, o resultado foi bastante satisfatório, com uma taxa de eficiência de 93,81%. Entre os casos em que o modelo falhou em acusar a presença de anomalias visuais, o mais significativo é quando o pássaro está na rotação errada. Nesse caso, a técnica proposta não foi capaz de determinar a presença de erros em nenhum dos casos. A razão mais provável para isso se deve à natureza da figura do pássaro, que possui um forma, em certa medida, uniforme, com pouca variação de cores. Dessa forma, ao se rotacionar a imagem em 180 graus, a diferença gerada pelo erro não foi suficiente para que o modelo detecta-se o defeito.

Em relação aos resultados errôneos quando o pássaro está em escala de cinza, a mais provável explicação é que devido a forma do pássaro, a mudança não gerava por si só uma diferença significativa para ultrapassar os limites que filtram falsos positivos. Dessa maneira, os resultados positivos se fizeram presente quando havia mais elementos em tela, aumentando o nível de ruído entre a imagem capturada e recriada.

No que tange o resultado falso negativo do texto na posição errada, existe uma correlação direta entre a capacidade de detecção de defeitos do modelo com a diferença entre a posição em que está o texto e onde ele deveria estar localizado. No caso do jogo usado para teste e com as configurações utilizadas da Tabela 3.1, percebeu-se que o modelo foi capaz de acusar a posição errada de elementos textuais que estavam a pelo menos 15 *pixels* de distância da sua posição esperada. Também é importante destacar que o tamanho da fonte e do texto tem interferência direta na eficiência do modelo em capturar erros dessa natureza.

Portanto, ao se analisar os resultados obtidos, fazendo uma comparação entre a eficácia de cada técnica de comparação isoladamente, percebe-se que para obter a melhor eficiência possível é necessário empregar os diferentes modelos de teste em conjunto. Contudo o teste de cor apresentou resultado semelhante ou melhor que o teste de forma em todos os casos testados. Dessa maneira, é possível inferir que esse torna o teste de forma obsoleto, ao menos para o jogo testado.

4.5 Ameaças à validade dos resultados

Uma das principais ameaças para a validade dos resultados obtidos neste trabalho concerne aos erros visuais que foram selecionados para serem injetados. Como variações de alguns desses erros foram utilizados no desenvolvimento, pode haver uma tendência do método em encontrá-los mais facilmente. Contudo, buscou-se cobrir todos os tipos de defeitos da taxonomia proposta por Macklon et al. [15]. Dessa forma, o modelo foi testado com diversos tipos de erros relevantes para a indústria de jogos eletrônicos.

No que diz respeito a natureza da injeção dos erros, apesar dessa não ser a maneira usual que essas anomalias surgem e precisam ser detectadas no processo usual de depuração, a forma com que os erros foram injetados fez com que os resultados finais fossem os mesmos. A força geradora dos defeitos visuais podem ser diferentes, mas ao se observar a tela do jogo de forma isolada, a anomalia presente é a mesma.

Para a recriação das imagens de comparação, são usados dados capturados durante a execução do jogo. Portanto o programa assume que esses valores estão sempre corretos e não é capaz de detectar anomalias quando os erros encontram-se nesses dados. Contudo, defeitos

dessa natureza se encaixam melhor na categoria de erros de lógica da programação interna do jogo, do que defeitos visuais. Em outras palavras, erros desse tipo estão fora do escopo proposto por esse trabalho.

Por fim, o modelo apresentado foi testado em apenas um único jogo de teste. Os limites da Tabela 3.1, possuem alta probabilidade de não serem os mesmos para garantir a eficiência do modelo em outros jogos. Esses valores irão variar de acordo com o nível de aleatoriedade, número de elementos visuais e a velocidade de movimentação dos agentes do jogo a ser testado. Dessa forma, aplicar os mesmos limites para outro jogo não irá gerar os mesmos resultados. Assim, se faz necessário ajustar manualmente essas variáveis. Apesar de o jogo escolhido como referência para os testes possuir um nível de aleatoriedade e de elementos visuais considerados compatíveis com outros jogos no mesmo *framework*, futuros estudos deveriam avaliar mais profundamente como diferentes estilos e gêneros de jogos eletrônicos afetam os resultados aqui apresentados.

CAPÍTULO 5

Trabalhos Relacionados

Nesta seção serão discutidos artigos relacionados à esfera de testes em jogos eletrônicos. Serão abordados tópicos que englobam técnicas de depuração que se encontram em amplo uso na atualidade e o atual estado do processo de automação dessa etapa vital do processo de desenvolvimento de programas computacionais voltados ao entretenimento.

5.1 Técnicas de testes em jogos eletrônicos

Guéhéneuc et al. [4] realizam no ano de 2021, um levantamento a respeito do atual estado de testes durante o processo de desenvolvimento de jogos eletrônicos. Os autores ressaltam a importância dessa etapa em um cenário, que o aumento do investimento na área de entretenimento eletrônico muitas vezes não corresponde a qualidade do produtor entregue aos consumidores. Para exemplificar essa situação, Guéhéneuc et al. relatam que projetos de jogos eletrônicos são notórios por possuírem *day-one bugs*, independentemente do orçamento alocado e o tamanho da equipe de desenvolvedores.

Os autores realizaram um questionário e fizeram a análise de literatura para identificar e analisar quais os processos de teste são mais comumente empregados nesta indústria e como eles podem vir a ser automatizados. Com os resultados obtidos, os autores constataram que os desenvolvedores quase que exclusivamente dependem da testagem manual da jogabilidade e do conhecimento intrínseco dos testadores no âmbito relacionado a jogos eletrônicos.

Em suma, os autores concluíram que a principal razão das técnicas atuais de teste serem insuficientes é a falta de automação. Esse, na visão dos autores, aparenta ser o próximo passo a ser tomado pela indústria para aprimorar a checagem da qualidade dos jogos, ao mesmo tempo que não se acarreta em um aumento considerável dos custos. No entanto, os autores também ressaltam que técnicas desse tipo podem não ser generalistas o suficiente.

A conclusão de Guéhéneuc et al. é corroborada por Dastani et al. [11] que acreditam que as dificuldades de se realizar testes em jogos eletrônicos são originadas da natureza intrínseca desse tipo de software, que são caracterizados por ser formados por cenários não determinísticos e por possuírem um elevado grau de interações do usuário com a aplicação. Além disso, os elementos visuais de um jogo eletrônico, assim como também sua lógica interna de funcionamento podem ser alterados regularmente durante as diferentes fases de seu desenvolvimento. Segundo os autores, essas características fazem com que as técnicas atuais de testagem sejam bastante custosas devido à robustez necessária para que os testes sejam realizados com sucesso. Portanto, para Dastani et al., a automação desse processo é uma abordagem que pode atender as necessidades dos desenvolvedores.

Ao se levar em consideração os pontos levantados por Dastani et al. [11] e resultados obtidos por Guéhéneuc et al. [4], assim como as conclusões relatadas por esses autores, o presente trabalho tem como objetivo suprir em parte a necessidade apontada pelos autores dos dois artigos, no que tange a automatização do processo de depuração em jogos eletrônicos.

5.2 Automação de testes para jogos eletrônicos

Petrillo e Politowski [31], realizaram um levantamento da conjuntura de testes automáticos para a depuração de códigos em jogos virtuais, no ano de 2022. De acordo com os autores, conforme a complexidade e o escopo do desenvolvimento de jogos aumenta, o processo de depuração continua como uma atividade essencial para garantir a qualidade do produto. Apesar disso, o aspecto manual em que são feitos parte considerável do processo de testagem, abre espaço para se encontrar soluções menos custosas para realizar essa etapa vital do processo de desenvolvimento.

Durante o processo de realização de uma análise exploratória da literatura pertinente à temática em questão, os autores identificaram um aumento substancial na quantidade de estudos de pesquisa direcionados ao campo dos testes automatizados aplicados a jogos eletrônicos. Adicionalmente, uma investigação foi conduzida com indivíduos atuantes no âmbito do desenvolvimento de jogos, e os resultados extraídos desta pesquisa empírica indicaram de maneira notória que esse grupo de profissionais ainda mantém um estado de ceticismo e relutância notável em relação à adoção de agentes autônomos para fins de depuração em seus projetos.

Assim, os autores concluem que existe uma necessidade premente de desenvolver técnicas que enfrentam esse desafio com um mínimo de intrusão, evitando perturbações significativas no processo de desenvolvimento desse tipo de software. Nesse contexto, durante a elaboração da proposta apresentada neste estudo, um ponto de destaque foi a dedicação em minimizar substancialmente o número de modificações necessárias nas funcionalidades comuns às quais os desenvolvedores estão acostumados a empregar. Para alcançar esse objetivo, a abordagem foi concentrada em manter a maioria das operações sendo realizadas internamente pelo código, sem exigir intervenção constante do usuário.

5.3 Detecção automática de erros visuais em jogos eletrônicos

Finlay et al. [35] buscou encontrar uma forma de realizar testes automatizados para encontrar bugs visuais em jogos em HTML5 <canvas>, que rodam em navegadores de internet. Segundo os autores, a técnica mais convencional para jogos que utilizam essa tecnologia, chamada teste de *snapshot*, envolve a comparação de imagens de referência ou oráculo previamente salvas, com diversas capturadas realizadas durante a execução real do jogo durante determinado intervalo de tempo. Nesse tipo de teste são escolhidas, para encontrar divergências entre as imagens, uma ou mais métricas de análise de similaridade visual. Contudo, criar e manter essas imagens de referência é uma tarefa onerosa, que exige mais tempo dos desenvolvedores do que um processo mais flexível e automatizado de testes.

A fim de contornar o problema supracitado, foi proposta uma técnica que divide os elementos visuais de uma captura de tela do jogo em um conjunto de imagens dos objetos presentes, em que cada uma é então comparada com sua imagem de referência correspondente. Dessa forma, ao invés de o desenvolvedor precisar fazer a captura diversos momentos diferentes de todas as telas possíveis de um jogo, basta possuir figuras de referência de todos os objetos. Isso torna o processo bem mais simples, pois esse tipo de imagem tem uma tendência menor de ser alterada durante o processo de desenvolvimento da aplicação, em relação a tela do jogo em sua totalidade.

Para realizar a avaliação da técnica proposta, Finlay et al. injetaram 24 erros visuais sintéticos em uma versão do jogo escolhido como jogo de teste. Para poder medir os resultados dos testes foram levadas em consideração quatro métricas de análise de similaridade visual:

porcentagem de sobreposição, erro médio quadrático, similaridade estrutural e similaridade de incorporação. Essa técnica apresentou resultados bastante positivos na busca de *bugs* visuais em jogos HTML5 <canvas>, com taxa de acurácia de 100%, enquanto a técnica mais tradicional de *snapshot* apresentou acurácia de 44,6

Portanto, tendo em vista os resultados expressivos encontrados no estudo para HTML5 <canvas>, o presente estudo busca realizar algo semelhante para jogos desenvolvidos na linguagem Lua, utilizando o *framework* Love2D. Contudo, ao invés de tratar os objetos como entidades individuais e separadas, esse artigo tem como objetivo realizar a análise da tela do jogo em sua totalidade. Essa abordagem foi escolhida, para que outros elementos visuais, como textos escritos na tela, estejam dentro do escopo da busca por erros. Além disso, a técnica aqui proposta também se faz capaz de encontrar discrepâncias em imagens qualquer imagem presente na tela do jogo, como imagens de fundo.

CAPÍTULO 6

Conclusão

Esse trabalho pretendeu propor uma nova técnica de automatização da detecção de anomalias visuais para jogos desenvolvidos no *framework* Love2D. Erros visuais são um problema recorrente na indústria de desenvolvimento de jogos eletrônicos atualmente [22],[26]. Ao se levar em consideração o fato da natureza intrinsecamente complexa da etapa de depuração desse tipo de software, torna clara a necessidade de se abordar esse problema por uma ótica que diminua o trabalho manual do testador [4],[31],[35].

Com esse intuito, foi desenvolvido um método em duas etapas. Na primeira parte, através de uma classe de depuração, são capturados todos os dados relevantes dos elementos gráficos presentes na tela do jogo. Ao mesmo tempo, também é salva uma captura de tela onde ocorrerá o processo de detecção de defeitos visuais. Após esse passo, os dados de execução são utilizados para recriar a imagem original capturada através do conjunto de figuras e fontes usadas pelo jogo em questão. Com as imagens original e recriada, o método realiza uma série de testes de similaridade visual e de variação de coloração de *pixels* para encontrar anomalias visuais. Caso algum desses testes encontre um valor de variação acima do limite estabelecido empiricamente para o jogo testado, o programa acusa a presença do erro e indica em quais amostras ele foi detectado.

Para a validação do modelo foram injetados 20 erros visuais considerados relevantes para a indústria seguindo o modelo taxonômico proposto por Macklon et Al. [15]. Após a execução de todos os casos selecionados para a validação do modelo, a taxa de eficiência para a detecção de anomalias visuais foi de 93.81%. A taxa de acerto quando não há defeitos foi de 100%. Esses resultados apontam que a eficácia do modelo torna-o uma ferramenta relevante para a indústria de jogos eletrônicos, com potencial de melhorar a qualidade dos jogos e reduzir os aspectos negativos associados a lançamentos contendo erros visuais.

A relevância do método proposto é evidenciada pela sua capacidade de automatizar o processo de depuração de jogos no que tange os erros gráficos, economizando tempo e recursos de desenvolvimento, melhorando a qualidade geral dos jogos e aumentando a satisfação dos jogadores. Além disso, outro ponto positivo é a versatilidade do método que mantém sua eficiência em diferentes etapas do desenvolvimento. Dessa maneira, permitindo ser aplicado no inicio, em testes de regressão ou até mesmo após a conclusão do desenvolvimento do projeto.

Para permitir a reprodução do trabalho realizado foi disponibilizado como código aberto, o jogo com todas as alterações feitas. Para aplicar o método proposto são necessárias algumas alterações no código do jogo a ser testado. Primeiramente deve-se criar criar e importar a mesma classe de depuração usada no jogo de teste desse trabalho. Em seguida se faz necessário trocar as funções nativas do Love2D por suas versões correspondentes da nova classe. Por fim, basta iniciar o modo de depuração no inicio da execução do programa e chamar o método de escrever os dados de execução ao final da iteração. Caso desejar, o testador pode chamar funções para alterar o número de amostras coletadas e o intervalo entre elas.

Em relação a utilização do método de recriação e comparação das imagens, dois pontos se fazem relevantes. Primeiramente é necessário atentar para a ordem de renderização das figuras.

Apesar da técnica ser generalista, pode ser necessários alterar esse aspecto do código em jogos de outros gêneros. O segundo ponto são os limites testados empiricamente que podem ser otimizados para cada jogo específico a ser testado.

Os resultados obtidos mostraram a validade de se utilizar dados de execução em conjunto com técnicas de visão computacional para a detecção automática de defeitos visuais em jogos no *framework* Love2D. Apesar disso, recomenda-se que futuros trabalhos avaliem mais profundamente como diferentes estilos e gêneros de jogos eletrônicos afetam os resultados obtidos neste trabalho e os limites definidos empiricamente.

Bibliografia

[1] T. I. M. A. Research e C. Group. «Gaming Market Report.» Acesso em 1 de setembro de 2023. (2022), URL: https://www.imarcgroup.com/gaming-market (ver pp. 1, 4).

[2] I. Khan. «Bethesda Facing Possible Class-Action Lawsuit Over Fallout 76.» Acesso em 1 de setembro de 2023. (2018), URL: https://www.gameinformer.com/2018/11/27/bethesda-facing-possible-class-action-lawsuit-over-fallout-76 (ver p. 1).

[3] B. Silva. «Assassin's Creed Unity | Por conta de bugs, Ubisoft cancela passe de temporada do game.» Acesso em 1 de setembro de 2023. (2014), URL: https://www.theenemy.com.br/assassins-creed-5/assassins-creed-unity-por-conta-de-bugs-ubisoft-cancela-passe-de-temporada-do-game (ver p. 1).

[4] Y.-G. G. Cristiano Politowski Fabio Petrillo, «A Survey of Video Game Testing,» em *2021 IEEE-ACM International Conference on Automation of Software Test (AST)*, 2021, pp. 90–99 (ver pp. 1, 4, 6, 30, 33).

[5] W. C. F. Roberto Ierusalimschy Luiz Henrique de Figueiredo. «Lua – an extensible extension language.» Acesso em 1 de setembro de 2023. (2015), URL: https://www.lua.org/spe.html (ver p. 1).

[6] PUC-Rio. «Lua Showcase.» Acesso em 1 de setembro de 2023. (2023), URL: https://www.lua.org/showcase.html (ver pp. 1, 3).

[7] M. Molina. «Roblox tem 48 milhões de jogadores ativos por dia.» Acesso em 1 de setembro de 2023. (2021), URL: https://www.techtudo.com.br/noticias/2021/09/roblox-tem-48-milhoes-de-jogadores-ativos-por-dia.ghtml (ver pp. 1, 3).

[8] R. Beschizza. «The most popular engines for indie games.» Acesso em 1 de setembro de 2023. (2018), URL: https://boingboing.net/2018/07/17/the-most-popular-engines-for-i.html (ver p. 1).

[9] A. Davis. «Testing for Game Development.» Acesso em 1 de setembro de 2023. (2016), URL: https://www.gamedeveloper.com/programming/testing-for-game-development (ver pp. 1, 6, 7).

[10] N. N. Emerson Murphy-Hill Thomas Zimmermann, «Cowboys, ankle sprains, and keepers of quality: How is video game development different from software development?» Em *In Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 1–11 (ver pp. 1, 5).

[11] S. S. et al., «Using an Agent-Based Approach for Robust Automated Testing of Computer Games,» em *Proceedings of the 12th International Workshop on Automating TEST Case Design, Selection, and Evaluation*, Association for Computing Machinery, 2021, pp. 1–8 (ver pp. 1, 6, 30).

[12] I. A. Andrew Truelove Eduardo Santana de Almeida, «We'll Fix It in Post: What Do Bug Fixes in Video Game Update Notes Tell Us?» Em *2021 IEEE-ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 736–747 (ver pp. 1, 4–6).

[13] Metacritic. «Game Releases by User Score - 2023.» Acesso em 1 de setembro de 2023. (2023), URL: https://www.metacritic.com/browse/games/score/metascore/year/all/filtered (ver pp. 2, 5).

[14] X. Y. Scott McMaster, «Developing a Feedback-Driven Automated Testing Tool for Web Applications,» em *2012 12th International Conference on Quality Software*, 2012, pp. 210–213 (ver p. 2).

[15] F. M. et al., «A Taxonomy of HTML5 Canvas Bugs,» vol. abs-2201.07351, 2022 (ver pp. 2, 20, 28, 33).

[16] S. D. L. Forums. «SDL 2.0.0 RELEASED!» Acesso em 1 de setembro de 2023. (2013), URL: http://forums.libsdl.org/viewtopic.php?t=9375 (ver p. 3).

[17] B. Wirtz. «Lua Showcase.» Acesso em 2 de setembro de 2023. (2023), URL: https://www.gamedesigning.org/engines/love2d/ (ver p. 3).

[18] B. Carnes. «Create Games with LÖVE 2D and Lua.» Acesso em 2 de setembro de 2023. (2022), URL: https://www.freecodecamp.org/news/create-games-with-love-2d-and-lua/ (ver p. 3).

[19] Love2D. «love.graphics documentation.» Acesso em 2 de setembro de 2023. (2021), URL: https://love2d.org/wiki/love.graphics/ (ver p. 3).

[20] R. Nystrom. «Game Programming Patterns - State.» Acesso em 1 de setembro de 2023. (2014), URL: https://gameprogrammingpatterns.com/state.html (ver p. 3).

[21] Love2D. «Love2D Local.» Acesso em 1 de setembro de 2023. (2021), URL: https://love2d.org/wiki/local (ver p. 4).

[22] E. Narcisse. «More Than a Third of 2014's Big-Budget Games Got Day-One Patches.» Acesso em 1 de setembro de 2023. (2014), URL: https://kotaku.com/more-than-a-third-of-2014-s-big-budget-games-got-day-on-1686398805 (ver pp. 4, 33).

[23] E. Makuch. «Battlefield 4's Rocky Launch "Absolutely"Damaged Player Trust, Producer Says.» Acesso em 1 de setembro de 2023. (2014), URL: https://www.gamespot.com/articles/battlefield-4s-rocky-launch-absolutely-damaged-pla/1100-6422805/ (ver p. 5).

[24] A. Reiner. «Cyberpunk 2077 Pulled From PlayStation Store, Refunds Offered.» Acesso em 1 de setembro de 2023. (2020), URL: https://www.gameinformer.com/2020/12/17/cyberpunk-2077-pulled-from-playstation-store-refunds-offered (ver p. 5).

[25] J. Middler. «CD Projekt's stock has fallen over 75 percent since Cyberpunk 2077's release.» Acesso em 1 de setembro de 2023. (2022), URL: https://www.videogameschronicle.com/news/cd-projekts-stock-has-fallen-over-75-percent-since-cyberpunk-2077s-release/ (ver p. 5).

[26] T. D. Mahapatra. «Star Wars Jedi: Survivor bug a nightmare, players struggle with potentially game-breaking glitch.» Acesso em 1 de setembro de 2023. (2023), URL: https://www.hindustantimes.com/technology/star-wars-jedi-survivor-bug-a-nightmare-players-struggle-with-potentially-game-breaking-glitch-101682788389878.html (ver pp. 5, 33).

[27] L. Ermi e F. Mäyrä, «Fundamental Components of the Gameplay Experience: Analysing Immersion.,» 2005, p. 15 (ver p. 5).

[28] B. C. Ryan Dinsdale. «Star Wars Jedi: Survivor é detonado na Steam por problemas de desempenho.» Acesso em 1 de setembro de 2023. (2023), URL: https://br.ign.com/star-wars-jedi-survivor/108395/news/star-wars-jedi-survivor-e-detonado-na-steam-por-problemas-de-desempenho (ver p. 5).

[29] A. Pruchnicka. «CD Projekt shares tank on bad Cyberpunk console reviews, refund reports.» Acesso em 1 de setembro de 2023. (2020), URL: https://www.reuters.com/technology/cd-projekt-shares-tank-bad-cyberpunk-console-reviews-refund-reports-2020-12-14/ (ver p. 5).

[30] R. Leston. «Cyberpunk 2077's Multiplayer Mode Was Cancelled Following Its Turbulent Launch.» Acesso em 1 de setembro de 2023. (2020), URL: https://sea.ign.com/cd-projekt-red-project-untitled/193084/news/cyberpunk-2077s-multiplayer-mode-was-cancelled-following-its-turbulent-launch (ver p. 5).

[31] F. P. Cristiano Politowski Yann-Gaël Guéhéneuc, «Towards Automated Video Game Testing: Still a Long Way to Go,» em *Proceedings of the 6th International ICSE Workshop on Games and Software Engineering: Engineering Fun, Inspiration, and Motivation*, Association for Computing Machinery, 2022, pp. 37–43 (ver pp. 6, 31, 33).

[32] J. Yarow. «That Goofy, Addictive Flappy Bird Game Is Doing $50,000 In Sales Per Day.» Acesso em 1 de setembro de 2023. (2014), URL: https://www.businessinsider.com/flappy-bird-is-doing-50000-in-revenue-per-day-2014-2 (ver p. 8).

[33] K. C. P. Library. «Python Imaging Library (PIL).» Acesso em 1 de setembro de 2023. (2020), URL: https://web.archive.org/web/20201121102218/http://www.pythonware.com/products/pil/ (ver p. 14).

[34] K. P. et al., «Realtime Computer Vision with OpenCV: Mobile Computer-Vision Technology Will Soon Become as Ubiquitous as Touch Interfaces.,» 4, vol. 10, Association for Computing Machinery, 2012, pp. 40–56 (ver p. 14).

[35] F. M. et al., «Automatically Detecting Visual Bugs in HTML5 Canvas Games,» Association for Computing Machinery, 2023 (ver pp. 31, 33).