UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Uma abordagem modular a tratamento de input em jogos.

Trabalho de Graduação

Aluno: Matheus Lima Borges Prado (m98lima@gmail.com)
Orientadora: Patrícia Cabral de Azevedo Restelli Tedesco (pcart@cin.ufpe.br)
Área: Desenvolvimento de jogos

Recife
2025

MATHEUS LIMA BORGES PRADO

Uma abordagem modular a tratamento de input em jogos.

Trabalho de Conclusão de Curso apresentado ao Centro de Informática (CIn) da Universidade Federal de Pernambuco (UFPE), como requisito parcial para conclusão do Curso de Ciência da Computação.

Orientadora: Patrícia Cabral de Azevedo Restelli Tedesco

Recife

2025

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

Prado, Matheus Lima Borges.
Uma abordagem modular a tratamento de input em jogos. / Matheus Lima Borges Prado. - Recife, 2025.
29 p. : il.

Orientador(a): Patrícia Cabral de Azevedo Restelli Tedesco
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Informática, Ciências da Computação - Bacharelado, 2025.
Inclui referências.

1. Desenvolvimento de jogos. 2. Padrões de projeto. I. Tedesco, Patrícia Cabral de Azevedo Restelli. (Orientação). II. Título.

000 CDD (22.ed.)

MATHEUS LIMA BORGES PRADO

Uma abordagem modular a tratamento de input em jogos.

Trabalho de Conclusão de Curso apresentado ao Centro de Informática (CIn) da Universidade Federal de Pernambuco (UFPE), como requisito parcial para conclusão do Curso de Ciência da Computação.

Aprovado em: _16_/_12_/_2025_

BANCA EXAMINADORA

_____

Profa. Patrícia Cabral de Azevedo Restelli Tedesco (Orientadora)

Universidade Federal de Pernambuco

_____

Prof. Leopoldo Motta Teixeira

(Examinador Interno)

Universidade Federal de Pernambuco

Resumo

Desenvolvimento de jogos digitais é uma área da programação notória por sua complexidade e por atrair entusiastas que não possuem experiência com engenharia de software. Essa combinação muitas vezes resulta em frustrações, especialmente em projetos de longo prazo, quando desenvolvedores inexperientes tentam produzir jogos que estão além das suas habilidades de programação.

Com a intenção de oferecer suporte a desenvolvedores que desejam construir jogos com múltiplas formas diferentes de controle de personagem, foi desenvolvido o *input middle-manager*, uma ferramenta projetada para realizar o manuseio de múltiplos scripts de tratamento de input, oferecendo ao desenvolvedor uma arquitetura que soluciona, de maneira escalável, a gerência dos vários métodos de controle e permite que o desenvolvedor possa se concentrar em desenvolver as funcionalidades que seu jogo precisa.

Este trabalho visa não apenas apresentar essa ferramenta e discutir sua construção, mas também mensurar o potencial interesse de desenvolvedores de jogos e entusiastas na mesma. Para tal, foi realizada uma pesquisa de validação com alguns alunos e ex-alunos da UFPE entre 20 e 30 anos a fim de averiguar suas impressões sobre o *input middle-manager* e seu desejo de usá-lo.

Também foi desenvolvido um protótipo jogável para demonstrar a funcionalidade da ferramenta e servir como exemplo de implementação utilizando a mesma. O repositório contendo o código fonte do protótipo, assim como o endereço digital onde o mesmo pode ser testado encontram-se ao final deste trabalho.

Palavras-chave: Desenvolvimento de jogos; Padrões de projeto.

Sumário

1. Introdução..... 7
    1.1. O input middle-manager..... 7
    1.2. Requisitos da ferramenta..... 8
    1.3. Pesquisa de validação..... 8
2. Referencial teórico..... 9
    2.1. Padrões de projeto..... 9
    2.2. Strategy Pattern..... 9
3. Aplicações da ferramenta..... 10
4. Arquitetura de código..... 11
    4.1. Visão geral..... 11
    4.2. Personagem jogável do protótipo..... 12
    4.3. Arquitetura do input middle-manager..... 13
5. Limitações da ferramenta..... 17
6. Pesquisa de validação..... 18
    6.1. A pesquisa..... 18
    6.2. Os resultados..... 22
7. Conclusão..... 26
8. Referências..... 27
9. Links..... 28

1. Introdução

Videogames sempre foram uma área da programação notória por sua complexidade, envolvendo uma ampla variedade de competências para construir até mesmo um projeto simples, e com o rápido desenvolvimento de tecnologias, a complexidade que desenvolvedores de jogos desejam atingir tem aumentado cada vez mais. Além de precisar assumir vários papéis, como de compositor musical, artista visual, escritor de narrativas, designer de interfaces e programador, para citar alguns exemplos, desenvolvedores de jogos também precisam se certificar que todas as partes do seu jogo funcionam em harmonia entre si e entregam a experiência desejada ao jogador.

Além disso, por sua natureza multidisciplinar, desenvolvimento de jogos é uma área que frequentemente atrai aspirantes com experiências muito diversas entre si. Muitos desenvolvedores de jogos vêm de áreas como música, design, marketing ou até mesmo áreas sem nenhuma relação com jogos.

Devido aos fatores mencionados acima, desenvolvimento de jogos pode se tornar uma tarefa muito desafiadora, especialmente para iniciantes que vêm de áreas que não envolvem programação. A pouca familiaridade com engenharia de software pode deixar desenvolvedores inexperientes vulneráveis a armadilhas comuns para programadores iniciantes como utilizar arquiteturas de código muito acopladas ou escrever scripts muito extensos, potencialmente resultando em uma base de código de difícil manutenção a longo prazo.

Um caso particular, que serviu como motivação para este trabalho, é o jogo Tchia [5], que permite ao jogador se transformar em diversas criaturas e objetos existentes no jogo. Algumas das possíveis transformações presentes no jogo necessitam lógica única para descrever seu comportamento, como livre movimentação em três dimensões ou a possibilidade de andar em superfícies verticais. Tipicamente ao abordar um projeto desse tipo, desenvolvedores inexperientes se sentiriam inclinados a descrever todos os comportamentos existentes em um único script de controle de personagem, tornando sua manutenção e a adição de novas formas de controle cada vez mais difícil à medida que o projeto evolui.

Visando facilitar o desenvolvimento de um jogo com design similar ao mencionado acima, este trabalho propõe uma ferramenta que abstrai a gerência entre os diferentes métodos de tratamento de input presentes no jogo e oferece uma maneira simples para que o desenvolvedor possa realizar a troca entre eles, diminuindo sua carga de trabalho e incentivando-o a escrever código modular.

1.1. O *input middle-manager*

Para atingir tal objetivo, foi utilizada uma arquitetura de código baseada em Strategy Pattern [1], onde o input do jogador é passado adiante para ser processado por algum dos scripts de controle de personagem criados pelo desenvolvedor, dependendo das regras do jogo.

Durante este trabalho, foi desenvolvido um protótipo jogável [4] utilizando a Godot Engine com a finalidade de demonstrar a funcionalidade da ferramenta na prática. A arquitetura utilizada no personagem jogável do protótipo será discutida em mais detalhes neste trabalho.

O *input middle-manager* não gera nenhum código nem decide por conta própria

quando ocorrerá a troca do script ativo ou qual script será escolhido. O *input middle-manager* é uma ferramenta que meramente delega o tratamento do input para outros scripts e realiza a troca de qual script ficará responsável pelo tratamento do input mediante chamadas de funções oriundas de scripts externos, escritos pelo desenvolvedor que o utiliza.

1.2. Requisitos da ferramenta

Devido à familiaridade e preferência pessoal do autor, o *input middle-manager* foi construído utilizando a Godot Engine e, portanto, só pode ser utilizado diretamente na mesma. Não há, contudo, nenhuma restrição que impossibilite sua implementação em outras game engines como Unity, Unreal ou Game Maker. A reprodução da arquitetura do *input middle-manager* não deve ser uma tarefa difícil, dada sua simplicidade.

Assim como será visto na seção onde a arquitetura da ferramenta será discutida, para utilizar o *input middle-manager* é necessário que os scripts de controle de personagem escritos pelo desenvolvedor herdem a classe *input handler* e implementem seus métodos ou a ferramenta não poderá operar corretamente.

Também será necessário centralizar referências às várias partes do personagem jogável em um objeto chamado “Node Refs”, que será passado adiante para os scripts de controle.

1.3. Pesquisa de validação

Além de desenvolver o *input middle-manager* e um protótipo para demonstrar seu uso, durante este trabalho também foi realizada uma pesquisa de validação para mensurar a percepção de algumas pessoas acerca da complexidade do *input middle-manager* e seu impacto em projetos de desenvolvimento de jogos, seu interesse na ferramenta e a probabilidade de recomendá-la para outros desenvolvedores de jogos.

2. Referencial teórico

2.1. Padrões de projeto

Na engenharia de software o termo padrões de projeto se refere a soluções generalizadas e reutilizáveis para problemas que comumente emergem durante o desenvolvimento de software. O conceito de padrões de projeto foi inicialmente formalizado na área de engenharia de software pelo livro "*Design Patterns: Elements of Reusable Object-Oriented Software*" [1], escrito por Erich Gamma, John Vlissides, Ralph Johnson e Richard Helm, notoriamente conhecidos como "Gang of Four", e publicado em 1994 pela editora *Addison-Wesley*.

Como observado empiricamente por estudos acadêmicos [2] [3], padrões de projeto são capazes de prover maior flexibilidade e menor complexidade aos projetos em que são utilizadas, aumentando a qualidade do código, deixando-o mais robusto e facilitando sua manutenção a longo prazo.

2.2. Strategy Pattern

O Strategy Pattern é um padrão de projeto onde múltiplas soluções, chamadas de estratégias, independentes e substituíveis entre si, são definidas e encapsuladas sob um mesmo contexto, solucionando o mesmo problema de maneiras diferentes. Ao invés de solucionar o problema diretamente, o Strategy Pattern delega a solução a uma das estratégias disponíveis em tempo de execução, permitindo trocar dinamicamente qual solução é utilizada.

Um exemplo de aplicação concreta do Strategy Pattern seria um aplicativo de planejamento de rotas que pode levar em consideração vários métodos de locomoção, como a pé, carro, ônibus e metrô, por exemplo. As opções citadas nem sempre terão acesso às mesmas rotas e, portanto, seus percursos precisariam ser calculados de maneiras diferentes entre o mesmo ponto de partida e destino.

A utilização do Strategy Pattern deve ser considerada cuidadosamente, já que sua implementação gera uma camada extra de abstração entre o método a ser utilizado e quem o utiliza, adicionando complexidade à lógica de negócios. O Strategy Pattern também requer que novas classes sejam criadas para cada nova estratégia, o que pode inchar o projeto.

Contudo, arquiteturas que implementam o Strategy Pattern promovem maior uniformidade, escalabilidade de código devido ao funcionamento desacoplado e isolado da delegação de responsabilidade e maior adaptabilidade devido à dinamicidade dessa delegação e à especificidade de cada estratégia [1].

Por conta desses motivos, o Strategy Pattern é o padrão de projeto ideal para dar suporte a vários métodos diferentes de controle de personagem em um mesmo jogo de maneira escalável e amigável ao desenvolvedor. A utilização do Strategy Pattern permite não apenas que o desenvolvedor escreva scripts de tratamento de input de maneira desacoplada, seguindo melhores práticas de engenharia de software, mas também oferece a opção de trocar livremente qual método de controle está ativo em um determinado momento.

3. Aplicações da ferramenta

Antes de dar início à discussão sobre o protótipo, é interessante mencionar brevemente alguns possíveis casos em que a arquitetura do *input middle-manager* pode contribuir para um projeto mais escalável.

- Personagem modular: O personagem jogável possui customização modular, como um robô ou uma armadura com partes trocáveis, por exemplo. Em um jogo com este tipo de design, certos tipos de pernas podem oferecer opções diferentes de movimentação e certos tipos de braços podem oferecer diferentes tipos de ataque, porém as partes são independentes entre si, podendo ser escolhidas em qualquer combinação.
- Transformações: O personagem jogável possui várias transformações que podem alterar a maneira como o mesmo atravessa o cenário ou interage com personagens não jogáveis ou com a física do jogo. Transformações podem permitir que o jogador voe, se movimente por baixo do solo, converse com plantas e animais ou mude de tamanho. Exemplos de jogos que implementam este tipo de design são Tchia [5] e Super Mario Odyssey [6].
- Veículos: Um jogo em que o personagem jogável tem acesso a vários veículos que funcionam de maneiras diferentes como carros, barcos, aviões, helicópteros, asas-delta… Exemplos de jogos que seguem este tipo de design são os jogos da série Grand Theft Auto [7].
- Máquinas de estado: Personagens que precisam de muitos estados diferentes para descrever seu comportamento ao longo do jogo. Um exemplo seria um jogo em que o personagem deve ter vários tipos de movimento diferentes dependendo do ambiente, como lama ou neve, dunas de areia, terreno íngreme ou montanhoso, chão escorregadio, escalada ou movimentação abaixo d'água. Exemplos de jogos que podem ilustrar esse tipo de design são os jogos da série Subnautica [8]

Nos exemplos citados acima, implementar todas as formas de controle de personagem em um único script tornaria a manutenção do projeto a longo prazo (assim como a possível adição de novas formas de controle não previstas na pré-produção do jogo) mais difícil devido ao alto acoplamento do código.

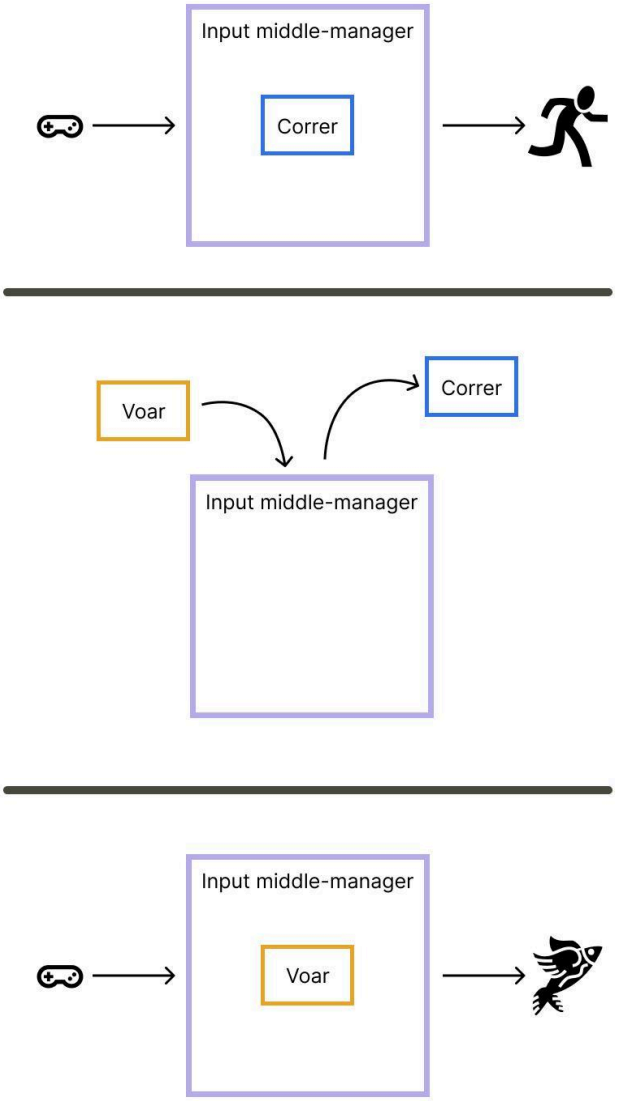
O projeto se beneficiaria de uma arquitetura de código modular, permitindo que o tratamento de input seja dividido em vários scripts de menor responsabilidade que seriam manuseados por uma classe externa.

4. Arquitetura de código

4.1. Visão geral

Em um projeto que utiliza o *input middle-manager*, o processamento do input do jogador não será realizado pela mesma classe que o recebe, esta repassará os dados do input para uma estratégia concreta, que pode ser trocada ao longo do jogo, e esta estratégia será responsável por controlar o personagem jogável, assim como ilustrado na **Figura 1** abaixo:

Figura 1 — Ilustração de funcionamento do *input middle-manager*.



Fonte: Matheus Lima (2025)

Essa estratégia deve herdar da classe *input handler* e sobrescrever seus métodos, que serão detalhados posteriormente nesta mesma seção, para que o *input middle-manager* consiga utilizá-la corretamente.

4.2. Personagem jogável do protótipo

Antes de discutir o código do *input middle-manager* precisamos falar sobre a estrutura utilizada no personagem jogável do protótipo, que foi construído utilizando a Godot Engine, e como seus componentes funcionam. No repositório público deste trabalho [4] há um curto vídeo onde é possível visualizar o protótipo em funcionamento.

Primeiramente, vamos definir os tipos de objetos, chamados de nodes na Godot Engine, utilizados no personagem jogável e suas respectivas funcionalidades:

- **Node**: Objeto básico que tem acesso apenas às chamadas de funções básicas da engine como atualização de frames e leitura do input do jogador e não possui nenhuma propriedade especial.
- **Node3D**: Extensão do **Node** que introduz um Transform3D, contendo posição, rotação e escala em um espaço 3D.
- **RemoteTransform3D**: Similar ao **Node3D**, porém este node copia seu Transform3D para outro node.
- **SpringArm3D**: Extensão do **Node3D** que executa colisão em linha reta ao longo do seu eixo Z e posiciona seus nodes filhos na posição mais distante disponível.
- **Camera3D**: Node utilizado para renderizar cenas 3D para a tela do jogador.
- **CharacterBody3D**: Node que representa um corpo físico feito especialmente para personagens que se movem.
- **CollisionShape3D**: Node que contém uma malha 3D que será utilizada por outros nodes, como **CharacterBody3D**, para calcular simulações de colisões físicas.

Figura 2 — Estrutura de Nodes utilizada na construção do personagem jogável do protótipo desenvolvido para este trabalho.

Fonte: Matheus Lima (2025)

Agora vamos detalhar as partes do personagem jogável, ilustradas na **Figura 2**, e como elas interagem entre si:

- Character (**CharacterBody3D**): Objeto que representa o corpo físico do personagem jogável.
- Node Refs (**Node**): Objeto que centraliza referências às partes importantes do personagem jogável (como a câmera ou o corpo físico) para facilitar seu acesso a outros objetos.
- Input Handling (**Node**): Node puramente organizacional com o intuito de isolar os nodes responsáveis pelo processamento do input.
    - Camera handler (**Node**): Node responsável por lidar com o input do jogador referente ao controle de câmera.
    - Movement Handler (**Node**): Node que utiliza o *input middle-manager* para delegar o processamento do input de movimento do jogador de acordo com qual dos personagens foi selecionado.
        - Handler capsule (**Node**): Node responsável por encapsular os objetos de processamento de input, chamados de *input handlers*, e realizar a troca de qual deles está ativo. Não é necessário utilizar este node para armazenar ou realizar a troca do *input handler* ativo, o manuseio do *input middle-manager* pode ser feito por qualquer node que tenha acesso a ele, esta foi meramente a arquitetura escolhida para satisfazer as necessidades de design específicas do protótipo.
- Meshes (**Node3D**): Este objeto meramente contém os componentes visuais dos personagens utilizados no protótipo, não é importante para o funcionamento do *input middle-manager*.
- **RemoteTransform3D**: Utilizado para copiar a posição do personagem jogável (mas não sua rotação) para a câmera com a finalidade de manter as rotações do personagem e da câmera independentes entre si.
- Player attachment (**Node3D**): Node utilizado como base para objetos que devem seguir o personagem jogável, mas ignorar sua rotação. Este node ignora o Transform3D do node pai e tem sua posição (mas não rotação) definida pelo RemoteTransform3D mencionado anteriormente.
    - Camera Base (**Node3D**): Base da câmera, utilizada para rotacionar horizontalmente.
        - **SpringArm3D**: Node utilizado para definir a distância da câmera em relação ao personagem jogável e rotacioná-la verticalmente.
            - **Camera3D**: Câmera utilizada para renderizar o jogo.
- **CollisionShape3D**: Objeto que contém a malha 3D utilizada para simular colisões do personagem jogável com outros objetos físicos.

4.3. Arquitetura do *input middle-manager*

A seguir vamos discutir a implementação da ferramenta desenvolvida para este trabalho e como utilizá-la.

A ferramenta é composta por dois scripts: o *input middle-manager*, que delega o tratamento de input para outros scripts, e a classe *input handler*, que deve ser herdada pelos scripts de tratamento de input escritos pelo desenvolvedor para que estes sejam manuseados por um node utilizando o *input middle-manager*. Ambos se encontram no repositório público deste trabalho [4].

A classe *input handler* funciona como uma interface e não foi construída para ser

utilizada diretamente, na **Figura 3** abaixo está sua estrutura, em GDScript:

Figura 3 — Código da classe *input handler* em GDScript.

```
extends Resource
class_name Input_Handler;

var node_refs: Node_Refs;

func _assign_refs(_node_refs: Node_Refs) -> void:
	node_refs = _node_refs;

func _setup() -> void:
	# Nesta função o input handler deve acessar o objeto "Node Refs" e pegar
	# referências para os nodes que serão necessários para o seu funcionamento.
	# Esta função é chamada pelo input middle-manager apenas uma vez quando o
	# input handler é inicializado.
	pass;

func _handle_input(event: InputEvent) -> void:
	# Nesta função, o input handler receberá os dados do input do jogador através
	# do objeto "input_event" e o processará. Esta função é chamada pelo input
	# middle-manager sempre que o jogador realiza um novo input.
	push_error("_handle_input function not implemented!");

func _update(delta: float) -> void:
	# Esta função será chamada pelo input middle-manager a cada frame do jogo.
	# "Delta_time" representa o tempo decorrido desde o último frame do jogo.
	pass;

func _physics_update(delta: float) -> void:
	# Esta função será chamada pelo input middle-manager a cada atualização da
	# simulação de física do jogo. "Delta_time" representa o tempo decorrido
	# desde a última atualização da simulação de física do jogo.
	pass;
```

Fonte: Matheus Lima (2025)

As funções detalhadas acima na **Figura 3**, com a exceção da função “assign_refs”, devem ser sobrescritas pelo desenvolvedor ao criar um novo script de processamento de input de acordo com a forma específica de controle de personagem desejada para aquele script.

A função “setup” é chamada sempre que um *input handler* se torna ativo e é onde este deve acessar o objeto “node_refs” para reunir referências a todas as partes do personagem jogável que serão necessárias para o seu funcionamento.

A função “handle_input” é a mais importante para um *input handler*, essa será chamada sempre que um input do jogador for detectado e é onde o processamento de tal input deve ocorrer.

As funções “update” e “physics_update” serão chamadas a cada frame renderizado e a cada atualização do sistema de física, respectivamente. Essas funções são importantes para qualquer código que precise acontecer em tempo real, como calculações de aceleração, por exemplo.

Já a classe *input middle-manager* não precisa ser herdada ou modificada, este script

deve ser anexado diretamente a um node na cena do personagem jogável. Sua estrutura, em GDScript, está detalhada na **Figura 4** abaixo:

Figura 4 — Código do *input middle-manager* em GDScript.

```
extends Node;
class_name Input_Middle_Manager;

var input_handler: Input_Handler;

func _input(event: InputEvent) -> void:
#>| Esta função é chamada pelo pela Godot Engine sempre que o jogador realiza
#>| um novo input.
>|  if (input_handler):
>|  >|  input_handler._handle_input(event);

func _plug_new_handler(new_handler: Input_Handler) -> void:
#>| Esta função é chamada por algum script externo para trocar o input handler
#>| ativo de acordo com as regras do jogo.
>|  input_handler = new_handler;
>|  var node_refs = %"Node Refs" as Node_Refs;
#>| A sintaxe "%Node Refs" indica que a Godot Engine acessará o node
#>| "Node Refs" através de um nome único dentro da cena do personagem jogável,
#>| pode ser necessário acessar esse objeto de uma maneira diferente caso esta
#>| arquitetura esteja sendo adaptada para outra game engine.
>|  input_handler._assign_refs(node_refs);
>|  input_handler._setup();

func _process(delta: float) -> void:
#>| Esta função será chamada pela Godot Engine a cada frame do jogo.
#>| "Delta_time" representa o tempo decorrido desde o último frame do jogo.
>|  if (input_handler):
>|  >|  input_handler._update(delta);

func _physics_process(delta: float) -> void:
#>| Esta função será chamada pela Godot Engine a cada atualização da simulação
#>| de física do jogo. "Delta_time" representa o tempo decorrido desde a última
#>| atualização da simulação de física do jogo.
>|  if (input_handler):
>|  >|  input_handler._physics_update(delta);
```

Fonte: Matheus Lima (2025)

Como apresentado na **Figura 4** acima, a classe *input middle-manager* apenas repassa a responsabilidade de processar o input do jogador para o *input handler* ativo e possui uma função que pode ser chamada por outros scripts para conectar um novo *input handler* quando desejado pelo desenvolvedor.

A função “input” é chamada pela engine sempre que um novo input do jogador é detectado. Nessa função o *input middle-manager* simplesmente checa se há um *input handler* conectado e repassa os dados do input para que o mesmo possa processá-lo.

A função “plug_new_handler” é utilizada para realizar a substituição do *input handler* ativo. Esta função recebe um objeto que herda a classe *input handler* e o conecta à variável utilizada para repassar as chamadas das funções da engine. Após conectar o novo *input handler* recebido, o *input middle-manager* chama as funções “assign_refs” e “setup” para que o novo *input handler* possa se preparar para processar o input do jogador.

As funções “process” e “physics_process” são chamadas pela engine a cada frame renderizado e a cada atualização do sistema de física, respectivamente. Assim como na função “input”, o *input middle-manager* simplesmente passa a chamada de função adiante para o *input handler* ativo.

5. Limitações da ferramenta

Apesar de flexível, o *input middle-manager* ainda está em fase inicial e possui muito espaço para ser refinado e expandido além do escopo deste trabalho. Nesta seção vamos discutir algumas limitações da versão atual da ferramenta que podem servir como guias para futuras atualizações do *input middle-manager*.

O *input middle-manager* checa se o objeto conectado é um *input handler*, mas não faz distinção sobre o tipo de input processado. Devido a isto, em jogos com tipos de controles diferentes é possível conectar um *input handler* de um tipo no lugar de outro. Por exemplo, em um jogo onde há controles de movimento e de ataque, seria possível conectar um *input handler* de ataque ao *input middle-manager* responsável pelo movimento e vice-versa. Desenvolvedores que utilizarem o *input middle-manager* em seu estado atual devem exercer cautela ao realizar trocas de *input handlers*.

O *input middle-manager* não está otimizado para que múltiplos objetos acessem o mesmo *input handler*. Caso seja de interesse do desenvolvedor que múltiplos objetos dentro do jogo possuam o mesmo *input handler*, como por exemplo um jogo onde várias armas diferentes utilizem o mesmo tipo de controle de ataque, será necessário fazer várias cópias do *input handler* para cada objeto ou será necessário desenvolver uma arquitetura mais robusta utilizando um objeto global para armazenar todos os *input handlers* do jogo e referenciá-los por lá.

Devido à familiaridade e preferência pessoal do autor, este trabalho foi desenvolvido na Godot Engine utilizando a linguagem própria da engine, GDScript. O uso do *input middle-manager* pode ser um pouco limitado por ter sido implementado em uma game engine menos popular. Contudo, traduzir a lógica do *input middle-manager* para outras plataformas de desenvolvimento de jogos mais populares, como Unity ou Unreal, não deve ser uma tarefa difícil.

6. Pesquisa de validação

6.1. A pesquisa

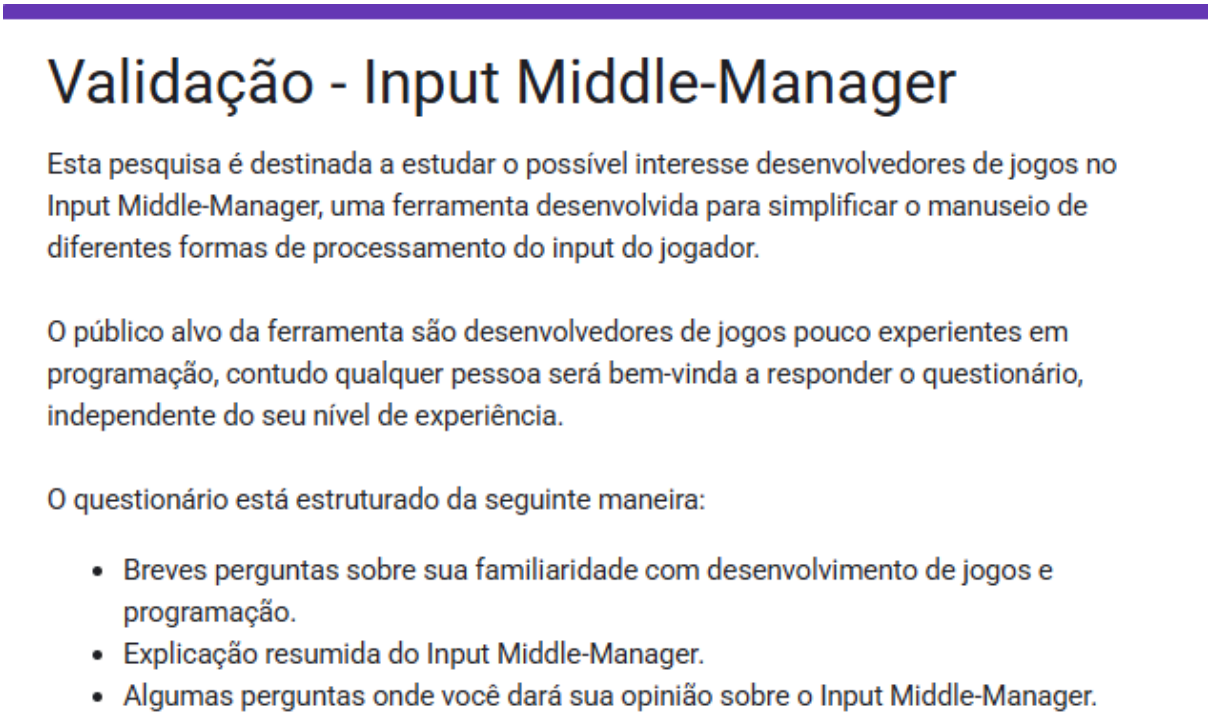
Para realizar a pesquisa sobre o potencial interesse no *input middle-manager*, foi utilizado um formulário estruturado da seguinte maneira:

1. Introdução e explicação da pesquisa.
2. Questões sobre a experiência prévia do participante com desenvolvimento de jogos e programação.
3. Breve explicação sobre o *input middle-manager* e seu funcionamento.
4. Perguntas acerca da opinião do participante sobre o *input middle-manager* e seu interesse em usá-lo.

A pesquisa foi conduzida com um pequeno grupo de 5 alunos e ex-alunos da UFPE, entre 20 e 30 anos, com históricos variados em relação a desenvolvimento de jogos e programação.

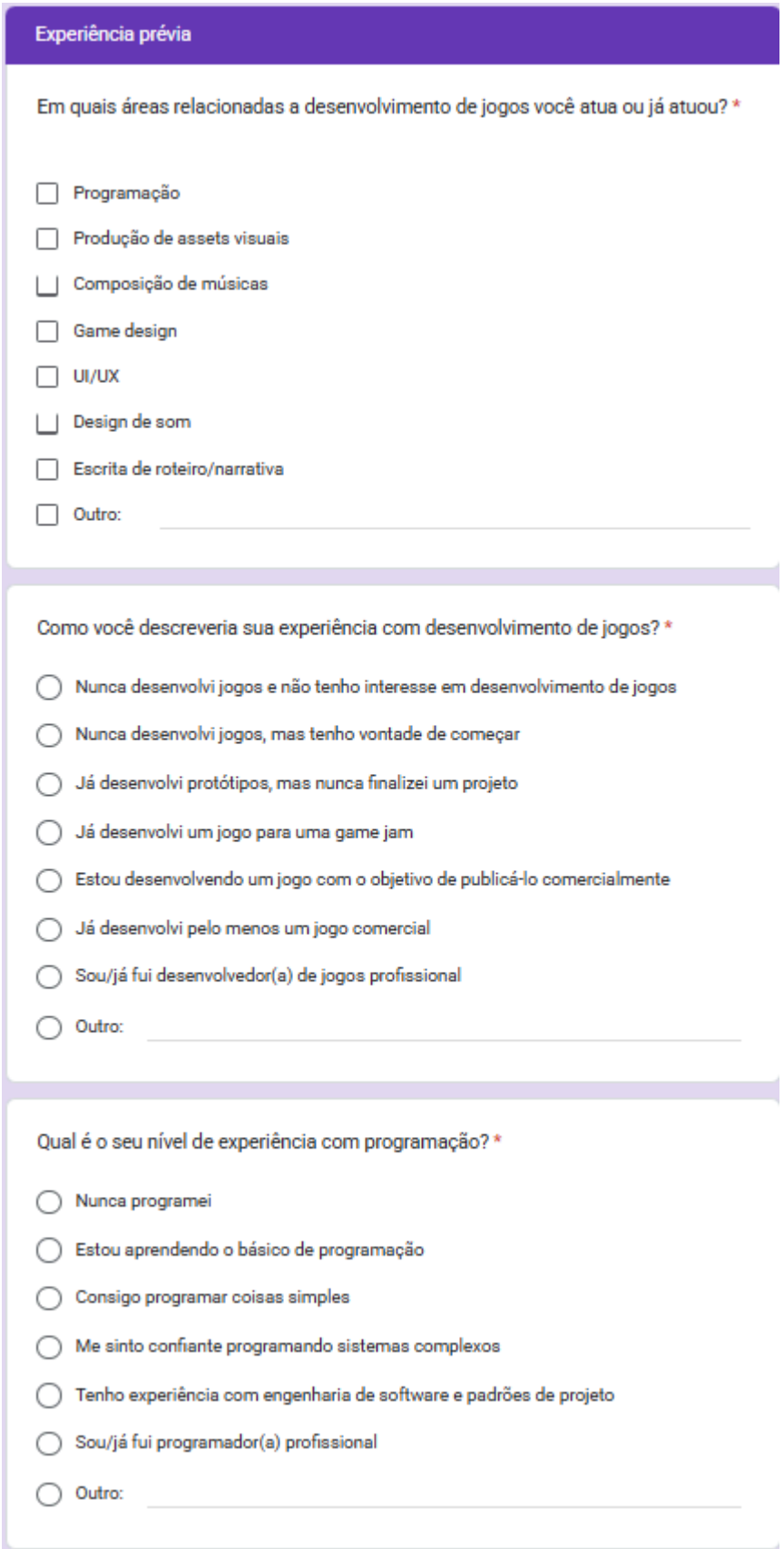
A seguir, representado na íntegra pelas **Figuras 5 a 8**, está o formulário utilizado na pesquisa de validação:

Figura 5 — Introdução do formulário.



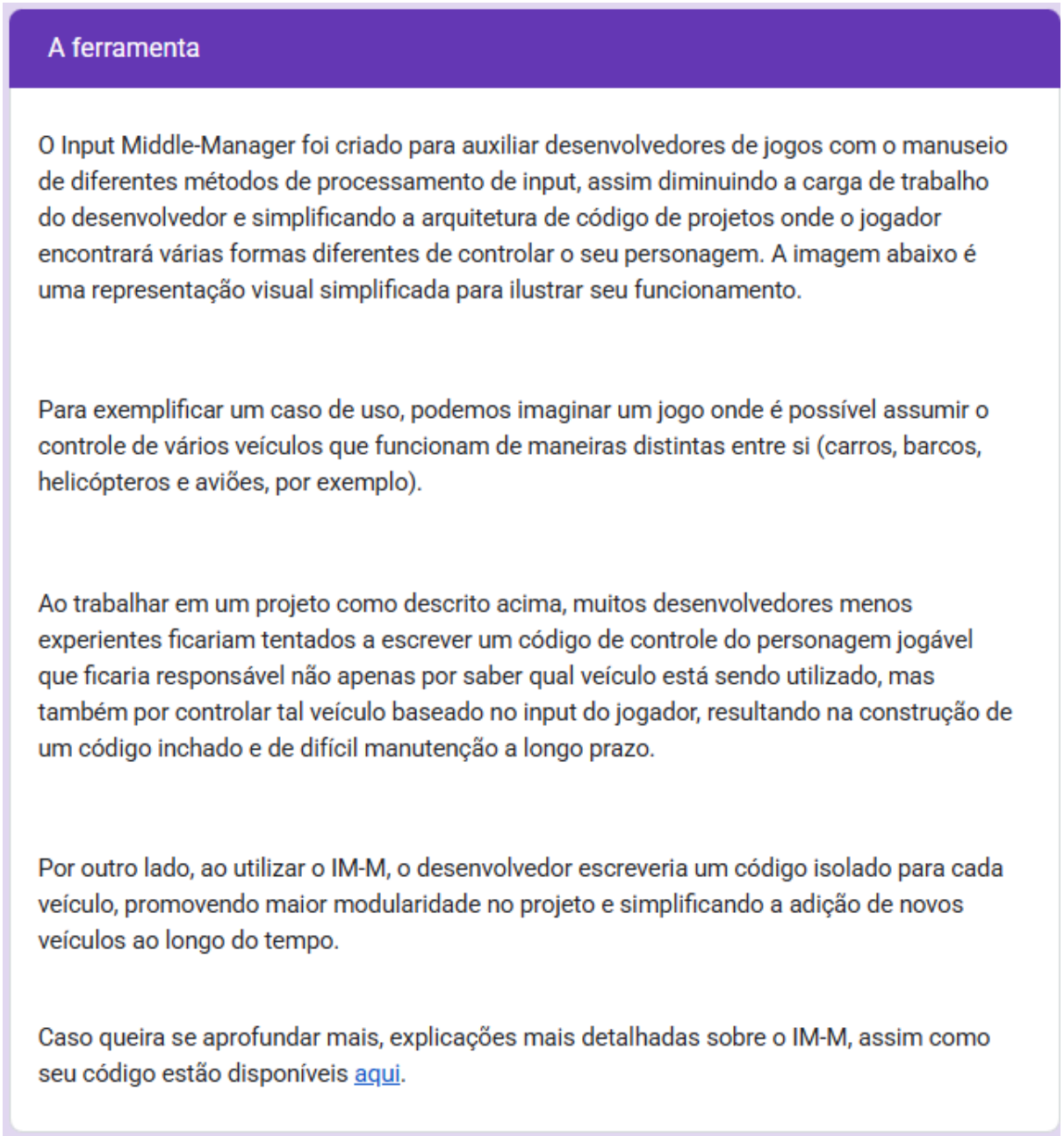
Fonte: Matheus Lima (2025)

Figura 6 — Perguntas sobre a experiência prévia do participante.



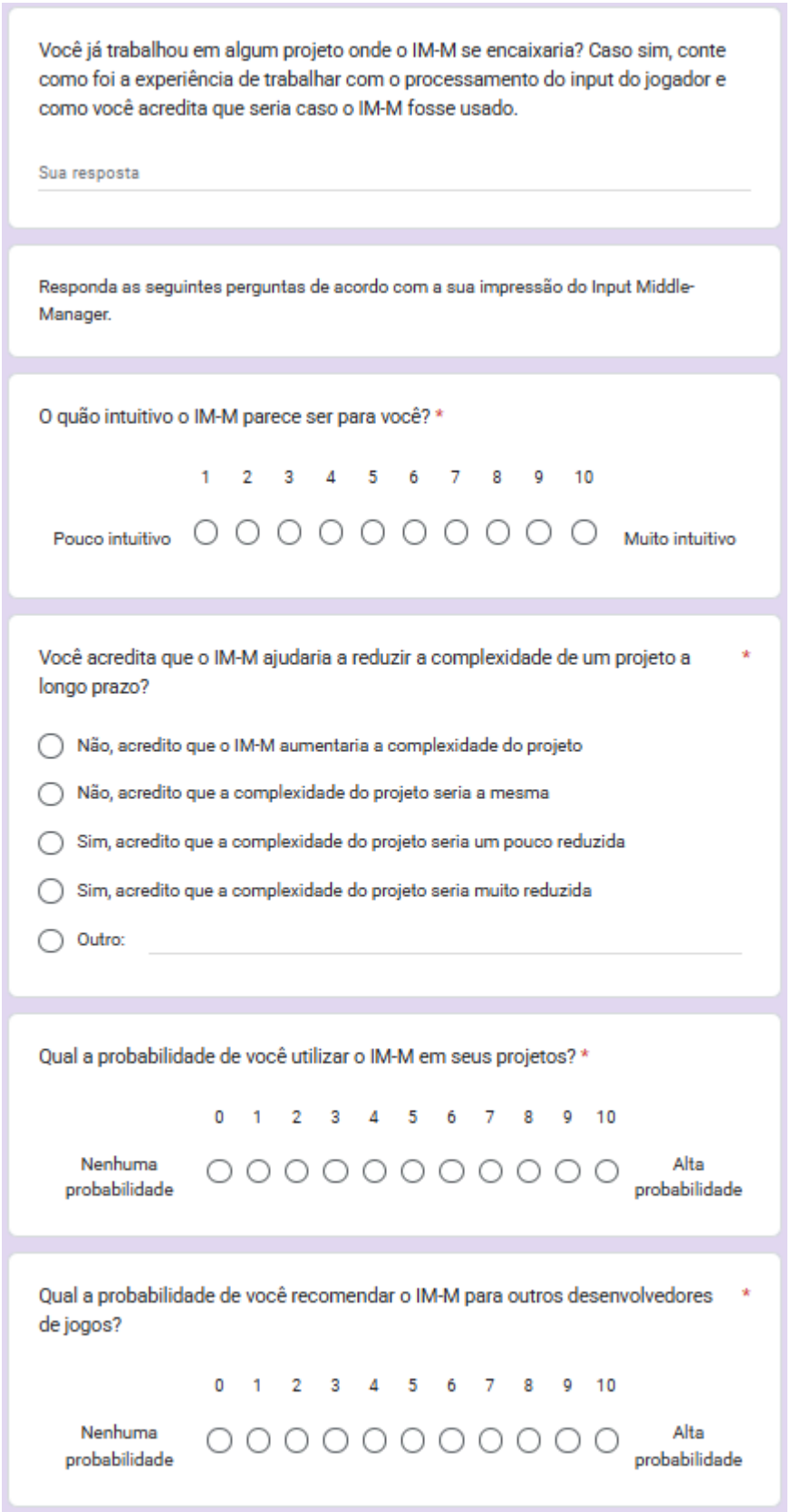
Fonte: Matheus Lima (2025)

Figura 7 — Explicação sobre o *input middle-manager*.



Fonte: Matheus Lima (2025)

Figura 8 — Questionário de opinião sobre o *input middle-manager*.

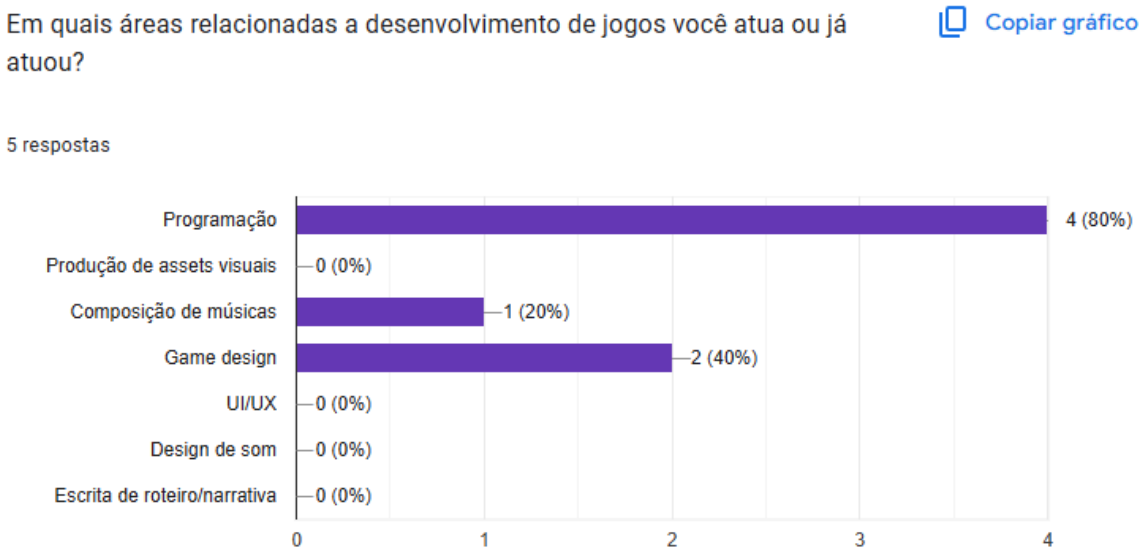


Fonte: Matheus Lima (2025)

6.2. Os resultados

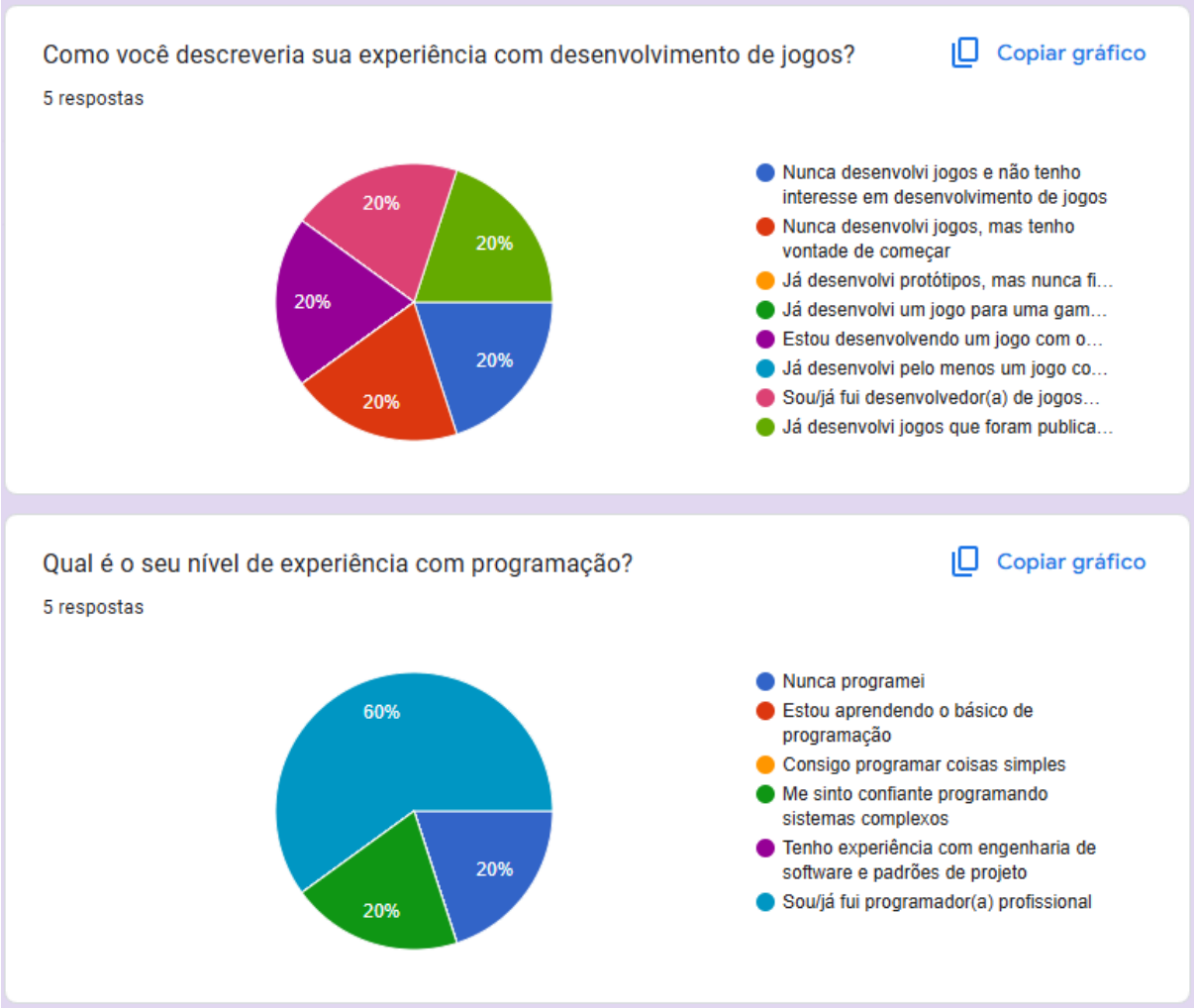
Abaixo, nas **Figuras 9 a 13**, encontra-se um resumo das respostas recebidas na pesquisa de validação. As respostas à pesquisa encontram-se, na íntegra, no repositório público deste trabalho [4].

Figura 9 — Respostas sobre áreas de atuação relacionadas a desenvolvimento de jogos.



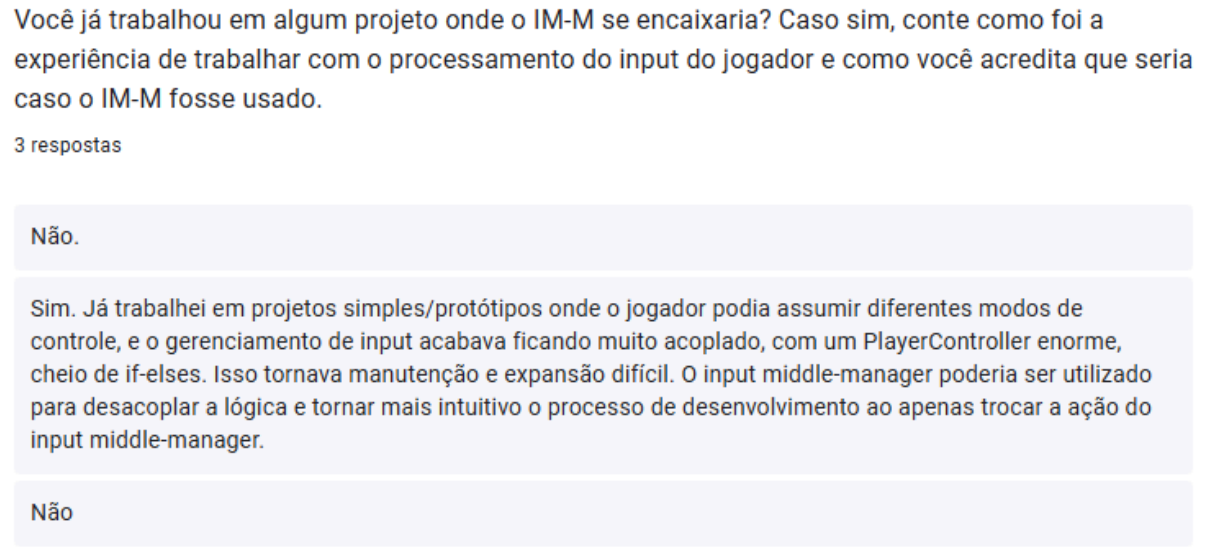
Fonte: Pesquisa de validação do *input middle-manager* (2025)

Figura 10 — Respostas sobre experiência prévia com desenvolvimento de jogos e programação.



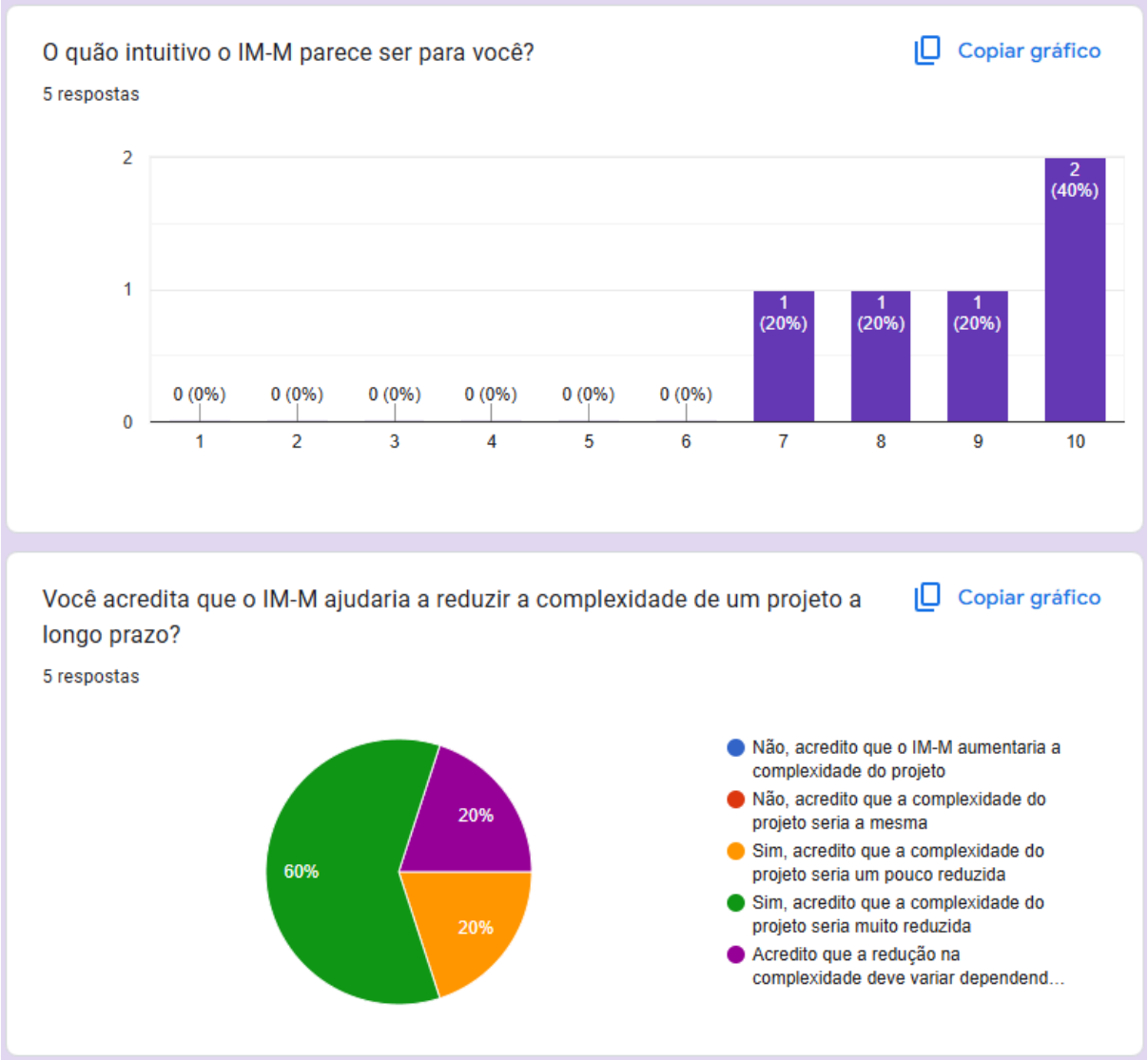
Fonte: Pesquisa de validação do *input middle-manager* (2025)

Figura 11 — Respostas sobre experiência prévia trabalhando com jogos onde o *input middle-manager* se encaixaria.



Fonte: Pesquisa de validação do *input middle-manager* (2025)

Figura 12 — Respostas sobre a complexidade e efetividade percebidas pelo participante acerca do *input middle-manager*.



Fonte: Pesquisa de validação do *input middle-manager* (2025)

Figura 13 — Respostas sobre probabilidade do participante utilizar ou recomendar o *input middle-manager* no futuro.

Fonte: Pesquisa de validação do *input middle-manager* (2025)

Como podemos observar nas **Figuras 9 a 13** acima, os participantes da pesquisa de validação perceberam o *input middle-manager* como intuitivo com base na descrição provida e acreditam que seu uso pode impactar positivamente projetos em que o mesmo seja utilizado.

Além disso, a maioria dos participantes indicou ter interesse em utilizar o *input middle-manager* ou indicar seu uso para outros desenvolvedores de jogos.

7. Conclusão

Nascido do desejo de oferecer suporte a desenvolvedores que desejam fazer jogos com diversos tipos diferentes de controle de personagem, este trabalho buscou construir uma ferramenta que realizasse a gerência entre vários métodos de tratamento de input para que o desenvolvedor não precisasse se preocupar com como integrar as múltiplas formas de controle e pudesse focar em como seu jogo deve funcionar.

Para atingir tal objetivo, foi utilizado um padrão de projeto conhecido como Strategy Pattern, onde a responsabilidade de realizar uma tarefa, neste caso o tratamento do input do jogador, é delegada para classes externas, chamadas de estratégias.

Ao aplicar o Strategy Pattern ao contexto de tratamento de input em jogos, surgiu o *input middle-manager*, que oferece flexibilidade para que o desenvolvedor possa integrar múltiplas formas de controle em um mesmo jogo de maneira escalável, permitindo que até mesmo desenvolvedores menos experientes possam trabalhar em projetos mais complexos com menos frustrações.

Durante este trabalho também foi desenvolvido um protótipo para demonstrar o funcionamento do *input middle-manager* e servir como exemplo para desenvolvedores que desejem utilizá-lo em seus projetos.

Além disso, foi realizada uma pesquisa de validação com um pequeno grupo onde foi constatado que o *input middle-manager* parece intuitivo e que existe um potencial interesse em utilizá-lo.

Em trabalhos futuros, estudos de caso podem ser realizados com grupos de desenvolvedores com níveis de experiência semelhantes com o intuito de comparar a qualidade do código entre seus projetos onde um grupo utilizaria *input middle-manager* e o outro não. Outro possível estudo seria a refatoração do código de um jogo existente, integrando o *input middle-manager* e analisando a qualidade do código antes e depois da refatoração.

8. Referências

[1] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

[2] Vihanga Piyawardhana , Tharaka Madhuwantha , Lahiru Chandika , et al. An empirical study of the impact of software design patterns on code quality. *TechRxiv*. June 22, 2023. DOI: 10.36227/techrxiv.23540073.v1

[3] Hsu, K.-H.; Szu-Tu, H.-C.; Tsai, C.-H. Assessing System Quality Changes during Software Evolution: The Impact of Design Patterns Explored via Dependency Analysis. *Electronics* **2024**, *13*, 1444. https://doi.org/10.3390/electronics13081444

9. Links

[4] Links do *input middle-manager*:
Repositório público: https://github.com/m98lima/Input-Middle-Manager
Demo jogável: https://m98lima.itch.io/input-middle-manager-demo

[5] “Tchia” disponível em: https://store.steampowered.com/app/1496590/Tchia/.

[6] “Super Mario Odyssey” disponível em:
https://www.nintendo.com/pt-br/store/products/super-mario-odyssey-switch/

[7] “Grand Theft Auto V” disponível em:
https://store.steampowered.com/app/3240220/Grand_Theft_Auto_V_Enhanced/

[8] “Subnautica” disponível em: https://store.steampowered.com/app/264710/Subnautica/