POWER PPLAY 2.0

Framework de Alta Performance para Desenvolvimento de Jogos

Desenvolvido por:
Kauã Neves Jesus de Paula

Orientadores/Equipe Original:
Prof. Esteban Clua, Prof. Anselmo Montenegro, Gabriel Saldanha, Adônis Gasiglia, Yuri Nogueira, Sergio Herman

Versão 2.0 | 2026

Sumário

Explore em detalhes os módulos e funcionalidades do framework Power PPlay 2.0

Fundamentos

Introdução aos conceitos basilares, configuração inicial do ambiente e gestão da janela principal.

Os Pilares da Versão 2.0
Guia de Instalação e Configuração
Classe Window e Gestão de Fluxo

Core Systems

O coração do framework: processamento de entrada de dados, renderização e motor físico.

Sistemas de Entrada (InputManager)
GameImage, Animation e Sprite
O Motor de Física 4.0

Visual e Performance

Recursos avançados para profundidade, iluminação dinâmica e efeitos de feedback visual.

Sistemas de Visualização e Profundidade
Sistema de Iluminação Dinâmica
Post-Processing e Feedback (Juice)

Arquitetura e Ferramentas

Utilitários de suporte, como navegação por A*, interpolação e práticas recomendadas de código.

Interpolação (Tweens) e Navegação (A*)
Buffer Virtual e Pixel Perfect
Guia de Performance e Clean Code

Desenvolvimento e Referência

Exemplos práticos, sistemas de criação de mundos e documentação completa da API.

Exemplos de Código
Construção de Mundos por Grade (TXT)
Dicionário de Classes e Referências

Os Pilares da Versão 2.0

A PowerPPlay 2.0 foi construída sobre quatro pilares fundamentais que a distinguem completamente da versão anterior e de outras bibliotecas educacionais. Cada pilar representa uma área crítica do desenvolvimento de jogos modernos, implementada com técnicas profissionais que aproximam os estudantes da realidade do mercado.

Performance Bruta

Implementação de *Resource Caching* (gestão inteligente de memória RAM) e *ViewPort Culling* (desenho apenas do que é visível na tela), permitindo o processamento de milhares de objetos simultâneos onde a versão anterior falhava. O sistema elimina o retrabalho de carregar recursos repetidamente e otimiza o renderização para hardware moderno.

Física Cinematográfica

Substituição da colisão reativa por um motor de cinemática de sub-pixel, eliminando erros de interpenetração (corner snapping) e oferecendo uma jogabilidade fluida com *Coyote Time* e *Jump Buffering*. Técnicas profissionais que tornam a movimentação dos personagens precisas e responsivas, essencial para plataformas e jogos de ação.

Independência de Resolução

Sistema de buffer virtual que permite rodar jogos em qualquer monitor (Full HD, 4K) mantendo a proporção original e a nitidez de *Pixel Art*, sem distorções. A adaptação automática garante que o jogo se veja perfeito em qualquer dispositivo, desde notebooks até monitores ultrawide, preservando a intenção artística do desenvolvedor.

Abstração e Modularidade

Ferramentas integradas para desenvolvedores iniciantes, como o **ObjectGroup** para gestão eficiente de coleções de objetos e o **SceneManager** para troca de telas com transições suaves. Além disso, a ferramenta automática **Architect** organiza arquivos automaticamente, mantendo o projeto limpo e escalável à medida que cresce em complexidade.

Estes pilares trabalham em conjunto para criar um ambiente de desenvolvimento que não apenas ensina conceitos básicos, mas prepara o estudante para as demandas reais da indústria de jogos. A combinação de otimização de performance com abstração de alto nível permite que desenvolvedores focem na criatividade e gameplay, enquanto o framework cuida das complexidades técnicas.

Guia de Instalação e Configuração

A PowerPPlay 2.0 foi projetada para ser portátil e de fácil integração. Siga os passos abaixo para preparar seu ambiente de desenvolvimento e começar a criar seus jogos imediatamente.

Pré-requisitos

Antes de iniciar, certifique-se de possuir o ambiente Python configurado em sua máquina:

- Python 3.8 ou superior instalado (Recomendado 3.10+)
- A biblioteca Pygame-CE (Community Edition) ou Pygame padrão
- Para instalar, abra o terminal e digite: `pip install pygame`

Estrutura de Diretórios

Para que o sistema de módulos e a ferramenta Architect funcionem corretamente, organize seu projeto da seguinte forma:

```
meu_projeto/
├── PPlay/
│   ├── window.py
│   ├── sprite.py
│   ├── physics.py
│   └── ... (demais arquivos)
└── main.py
```

Primeiro Passo: Importação

Diferente da versão 1.0, na PowerPPlay 2.0 você não precisa inicializar o Pygame manualmente. A classe Window gerencia o ciclo de vida do motor. Para começar, basta importar a janela no seu arquivo main.py:

```
from PPlay.window import Window

# Inicializa a janela virtual de 800x600 pixels
janela = Window(800, 600, "Meu Primeiro Jogo 2.0")

while True:
    # O comando update() processa eventos, tempo e desenho
    janela.update()
```

Ativando a Ferramenta Architect (Opcional)

Se o seu código no main.py começar a crescer demais, você pode usar a ferramenta de organização automática. No terminal, dentro da pasta do seu projeto, utilize:

```
python PPlay/architect.py main.py NomeDaSuaClasse pasta_destino
```

Isso moverá automaticamente sua lógica para um arquivo separado e configurará os imports necessários, mantendo seu projeto limpo e profissional.

3. Classe Window e Gestão de Fluxo

A classe Window é a espinha dorsal da PowerPPlay 2.0. Ela não apenas cria a janela, mas gerencia o Delta Time (essencial para velocidade constante em qualquer PC), o Buffer Virtual (resolução independente) e a Fila de Eventos.

3.1. O Construtor da Janela

Ao instanciar a Window, o desenvolvedor define a "Resolução Lógica" do jogo através dos parâmetros configuráveis:

Parâmetro	Tipo	Descrição
largura_virtual	int	Largura em pixels.
altura_virtual	int	Altura em pixels.
titulo	str	Texto da janela.
pixel_art	bool	Pixels nítidos ao redimensionar.

A resolução lógica permite que seu jogo tenha o mesmo comportamento visual em monitores de diferentes tamanhos. O parâmetro `pixel_art` é vital para jogos com estética retrô, garantindo que a escala não borrife a arte.

3.2. Métodos de Fluxo e Renderização

update()

Sincroniza FPS, limpa a tela e processa inputs. Deve ser o último comando do loop.

delta_time()

Retorna o tempo do último frame. Essencial para movimentos constantes.

set_background()

Define a cor de fundo usando nomes (ex: "black") ou tuplas RGB.

3.3. Exemplo de Ciclo de Vida

```
janela = Window(800, 600, "Meu Jogo")
while True:
    janela.set_background_color("black")
    # Lógica e Desenho aqui
    janela.update()
```

4. Sistemas de Entrada (Keyboard, Mouse e InputManager)

A PowerPPlay 2.0 foi projetada para otimizar a interação do jogador, separando a detecção de hardware da intenção do jogador. Essa abordagem garante um código mais limpo, profissional e flexível, facilitando a adaptação do seu jogo a diferentes tipos de controle e preferências do usuário.

4.1. Teclado (Keyboard)

key_pressed("tecla")

Retorna `True` enquanto a tecla estiver sendo **SEGURADA**. Ideal para ações contínuas como movimentação.

key_down("tecla")

Retorna `True` apenas no instante em que a tecla é **PRESSIONADA**. Perfeito para pulos ou menus.

4.2. Mouse

get_position()

Retorna uma tupla `(x, y)` com as coordenadas do cursor corrigidas para a resolução lógica do jogo.

button_down(botao)

Detecta o clique único (1: esquerdo, 2: meio, 3: direito).

draw_debug()

Desenha as coordenadas do mouse ao lado do cursor, facilitando o desenvolvimento.

4.3. InputManager (Mapeamento de Ações)

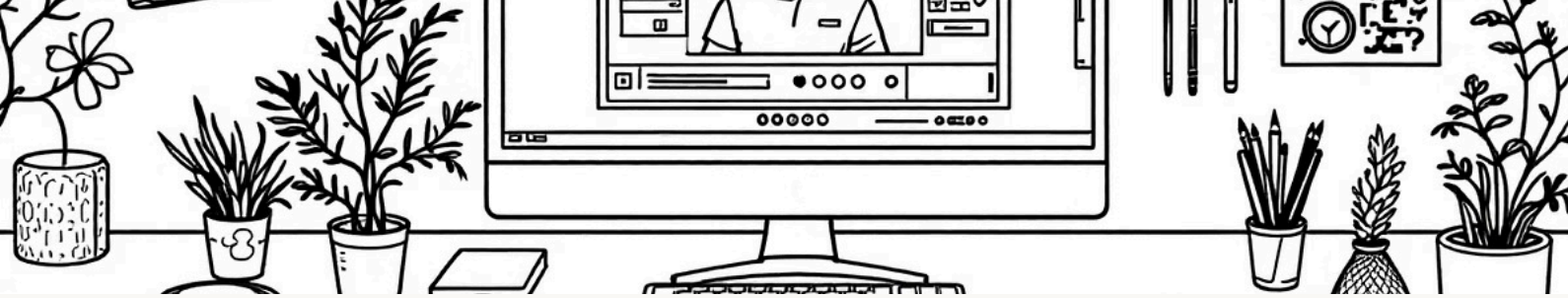
O `InputManager` promove o desacoplamento de código, permitindo criar abstrações para ações do jogador.

```
from PPlay.input_manager import InputManager

# Define que 'pulo' pode ser acionado por 3 teclas diferentes
InputManager.define_action("pulo", ["SPACE", "W", "UP"])

# No loop principal, o código fica muito mais legível:
if InputManager.action_pressed("pulo"):
    player.jump()
```

Este sistema facilita a leitura do código e permite adicionar suporte a gamepads ou outras formas de entrada no futuro, simplesmente estendendo as definições de ação sem impactar a lógica central do jogo.



5. GameImage, Animation e Sprite

Na PowerPPlay 2.0, a gestão de recursos visuais é otimizada para garantir alta performance e eficiência de memória. Ao invés de carregar o mesmo arquivo diversas vezes, a engine utiliza um padrão **Flyweight**. Isso significa que, se você criar uma centena de inimigos com a mesma imagem, o arquivo será carregado do disco apenas uma vez, economizando significativamente RAM e processamento, um diferencial crucial para jogos complexos.

5.1. GameImage e Resource Cache

`GameImage` é a classe fundamental para qualquer elemento visual no seu jogo. Ela abstrai a complexidade do carregamento e renderização de imagens.

- `draw()`: Renderiza a imagem na posição (x, y) atual.
- **Otimização:** Imagens convertidas para `convert_alpha()`, acelerando o desenho em até 5x.

5.2. Animation (Spritesheets)

A classe `Animation` simplifica o uso de spritesheets. O desenvolvedor apenas informa o número de frames e a duração total, e a engine gerencia a transição.

```
from PPlay.animation import Animation

# Carrega uma imagem com 4 frames horizontais de uma moeda girando
moeda = Animation("assets/moeda_sheet.png", 4)
moeda.set_total_duration(500) # Define 0.5 segundos para o ciclo completo
```

5.3. Sprite (O Ator)

A classe `Sprite` herda capacidades visuais e adiciona funcionalidades essenciais para interação e movimento.

move_key_x(velocidade)

Facilita movimentação horizontal com uso de Delta Time para consistência.

setup_physics(motor)

Acopla motor de física cinemática para resposta a forças e colisões.

update_physics(solidos)

Processa automaticamente gravidade, inércia e colisões.

Com `Sprite`, você tem um controle completo sobre a interação visual e física dos elementos do seu jogo, tornando o desenvolvimento mais intuitivo e poderoso.

6. O Motor de Física 4.0 (Inquebrável)

O sistema de física da PowerPPlay 2.0 foi projetado para eliminar o "Corner Snapping" (engate de quinas) e o efeito de atravessar paredes. Ele utiliza Separação de Eixos e Sub-stepping.

6.1. O Algoritmo de Movimento

Diferente da versão 1.0, o movimento não é aplicado de uma vez. A engine fatia o deslocamento em passos de 1 pixel e resolve a colisão na seguinte ordem:

Eixo X

Aplica velocidade horizontal → Checa colisão → Resolve snap lateral.

Eixo Y

Aplica gravidade e velocidade vertical → Checa colisão → Resolve snap de chão/teto.

6.2. KinematicBody: O "Feeling" de Jogo Profissional

Ao usar o componente `KinematicBody` através do método `setup_physics()`, o Sprite ganha automaticamente:

Coyote Time

Permite pular até 150ms após ter saído de uma plataforma.

Jump Buffering

Executa o pulo no instante do contato com o chão.

Inércia e Fricção

Adiciona peso ao movimento, evitando paradas bruscas.

6.3. Métodos Principais

setup_physics(motor)

Acopla o cérebro físico ao Sprite.

update_physics(solidos)

Processa toda a física do personagem no loop.

pular(forca)

Aplica impulso respeitando timers de tolerância.

7. Sistemas de Visualização e Profundidade

A PowerPPlay 2.0 introduz o conceito de Mundo Virtual, onde o cenário pode ser vastamente maior que a janela do computador.

7.1. Camera e Viewport

A classe Camera atua como um tradutor de coordenadas. Ela converte a posição global do objeto para a posição relativa na tela.

> ### follow(alvo, suavizacao)
>
> Faz a câmera seguir um Sprite com interpolação linear (Lerp) para evitar tremores.

> ### set_world_bounds(largura, altura)
>
> Define as "paredes invisíveis" da câmera, impedindo que ela mostre o vazio fora do mapa.

7.2. ParallaxSystem (Profundidade Infinita)

Permite criar fundos que se movem em velocidades diferentes, simulando 3D.

- **Aritmética Modular:** O fundo é automaticamente infinito. Ao sair por um lado, a imagem reaparece pelo outro de forma imperceptível.
- **Fatores de Velocidade:**
  - 0.0: Estático (Céu/Estrelas distantes)
  - 0.1 - 0.4: Longe (Nuvens, montanhas)
  - 0.5 - 0.8: Médio (Árvores de fundo)

7.3. Exemplo Prático

```
parallax = ParallaxSystem()
parallax.add_layer("ceu.png", 0.0)
parallax.add_layer("montanhas.png", 0.4)

while True:
    parallax.draw() # Desenha o fundo infinito
    player.draw()   # Desenha o jogo sobre o fundo
```

8. Organização para Grandes Projetos

Para evitar o código "macarrônico" (tudo em um único arquivo ou loops infinitos), a PowerPPlay 2.0 oferece abstrações de alto nível.

8.1. ObjectGroup (Organizadores de Espaço)

Ideal para quem não domina Programação Orientada a Objetos complexa. Funciona como uma "pasta" de objetos.

add(obj)

Joga um Sprite dentro do grupo.

draw()

Desenha todos os membros do grupo de uma vez.

collided(alvo)

Retorna qual objeto do grupo o alvo atingiu. Substitui loops for manuais.

8.2. SceneManager (Troca de Telas)

Permite separar o jogo em arquivos ou classes independentes (Ex: Menu, Fase 1, Game Over).

change_scene(InstanciaCena)

Troca a tela ativa, limpando a memória da cena anterior.

Ciclo de Vida

Cada cena possui seus próprios métodos loop() e draw(), mantendo a lógica isolada.

8.3. TileMap (Construção por Grade)

Transforma arquivos de texto simples (.txt) em cenários físicos complexos.

Culling Automático: A engine detecta quais blocos do mapa estão fora da visão da câmera e não os desenha, economizando CPU e permitindo mapas com milhares de blocos.

9. Sistema de Iluminação Dinâmica (Lighting)

A PowerPPlay 2.0 permite a criação de atmosferas imersivas através do módulo LightingSystem. Este sistema utiliza renderização subtrativa para simular escuridão e fontes de luz em tempo real.

9.1. O Conceito de Névoa (Fog of War)

O sistema funciona criando uma camada de cor sólida (geralmente azul escuro ou preto) que cobre toda a tela. Fontes de luz são então "somadas" a essa camada usando o modo BLEND_RGBA_ADD, criando buracos de visibilidade. Por fim, essa camada é multiplicada pela cena original.

9.2. Componentes do Sistema

LightingSystem(opacidade)

Inicializa o sistema. A opacidade define o quão escuro será o ambiente (0 a 255).

lights.append()

Adiciona uma fonte de luz: (x, y, raio, intensidade).

9.3. Exemplo de Lanterna

```
luzes = LightingSystem(opacidade=240)

# No loop, a luz segue o jogador:
luzes.lights.append((player.x, player.y, 150, 255))
luzes.draw()
```

10. Post-Processing e Feedback (Juice)

O impacto visual é o que torna a jogabilidade satisfatória. A engine oferece ferramentas nativas para feedback cinestésico.

10.1. ScreenEffects (Shake e Flash)

Classe estática para manipular a percepção visual do jogador.

shake(intensidade, duracao)

Faz a câmera tremer. Essencial para explosões ou quedas.

flash(cor, duracao)

Preenche a tela com uma cor sólida temporária. Ideal para dano (vermelho) ou cura (verde).

10.2. Sistema de Partículas (Particles)

Permite gerar centenas de pequenos objetos visuais de vida curta sem sobrecarregar o processador.

ParticleEmitter(x, y)

Cria um ponto de origem de partículas.

explodir(quantidade)

Dispara um lote de partículas em direções aleatórias.

Atributos de Customização:

- `vida_base`: Tempo que a partícula dura.
- `spread_x / spread_y`: Força da dispersão inicial.
- `cor`: Cor base da partícula (sofre fade-out automático).

10.3. Quando usar?

Sempre que ocorrer uma interação física importante: ao pular (partículas de poeira), ao coletar um item (faíscas/brilho) ou ao colidir com um inimigo (tremor e flash).

11. Interpolação (Tweens) e Navegação (A*)

Para alunos de Ciência da Computação, a PowerPPlay 2.0 integra conceitos fundamentais de algoritmos de forma prática e aplicada.

11.1. Tween Engine (Animação Matemática)

O módulo Tween permite transições suaves entre valores sem o uso de lógica de incremento manual (x += 1).

Tween.to(objeto, atributo, valor_final, tempo, tipo)

Anima um atributo suavemente. Tipos: linear, ease_in, ease_out, bounce.

Uso Comum

Menus deslizando ou itens coletáveis pulsando suavemente.

11.2. Navigation e Pathfinding (IA)

Implementação do algoritmo A* (A-Estrela) para busca de caminho em grafos de grade (TileMaps).

encontrar_caminho(mapa, inicio, fim)

Retorna lista de coordenadas do caminho mais curto, desviando de sólidos.

Otimização

Recalcule o caminho apenas algumas vezes por segundo, não todo frame.

11.3. Raycasting (Sensores de Visão)

Lança um raio invisível entre dois pontos para detectar bloqueios de linha de visão.

Collision.raycast(origem, destino, solidos)

Retorna se a linha de visão está bloqueada. Essencial para IAs de furtividade.

12. Buffer Virtual, Fullscreen e Pixel Perfect

A PowerPPlay 2.0 resolve o problema clássico da distorção de imagem ao redimensionar janelas através de um sistema de Resolução Virtual Independente.

12.1. O Buffer Virtual (Canvas)

Quando você cria uma Window(400, 300), a engine não desenha diretamente na tela do Windows. Ela cria uma superfície interna (Canvas) de exatos 400x300 pixels.

- Todo o seu código de draw() acontece nesse buffer protegido.
- No final do frame, a engine pega esse "quadro pronto" e o estica para preencher a janela real.

12.2. Aspect Ratio e Letterboxing

Para evitar que círculos virem elipses (o efeito "esticado"), a engine calcula o maior tamanho possível que o buffer pode ocupar mantendo a proporção original. Caso a janela tenha um formato diferente, a engine adiciona automaticamente faixas pretas (Letterboxing), garantindo a integridade visual do design original.

12.3. Pixel Perfect vs. Smooth Scaling

O parâmetro `pixel_art=True` no construtor da Window define como esse redimensionamento é feito:

Pixel Perfect (True)

Usa Nearest Neighbor. Cada pixel vira um bloco sólido. Essencial para Retrô.

Smooth (False)

Usa filtragem bilinear, suavizando bordas. Ideal para alta definição.

12.4. Comandos de Tela

F11

Alterna automaticamente entre modo Janela e Tela Cheia.

janela.mouse.get_position()

Retorna a posição do mouse já convertida para o espaço virtual.

13. Exemplos de Código: Da Base ao Avançado

13.1. Estrutura Básica (O "Hello World")

Este é o código mínimo para rodar a engine com um personagem estático.

```
from PPlay.window import Window
from PPlay.sprite import Sprite

janela = Window(800, 600, "Início Rápido")
player = Sprite("hero.png")
player.set_position(400, 300)

while True:
    janela.set_background_color("darkblue")
    player.draw()
    janela.update()
```

13.2. Personagem com Física e Câmera

Como configurar um player que pula, colide com o chão e é seguido pela visão do jogo.

```
from PPlay.window import Window
from PPlay.sprite import Sprite
from PPlay.physics import Physics
from PPlay.camera import Camera

janela = Window(800, 600, "Personagem e Câmera")
player = Sprite("hero.png")
player.set_position(janela.width / 2, janela.height / 2)

chao = Sprite("ground.png")
chao.set_position(0, janela.height - chao.height)

fisica = Physics()
fisica.add_object(player, has_gravity=True, can_collide=True)
fisica.add_object(chao, is_static=True)

camera = Camera(janela, player, max_x=1000, max_y=1000) # Exemplo de limites para a
câmera

while True:
    janela.set_background_color("darkblue")

    # Input para pular (exemplo)
    if janela.keyboard.key_pressed("SPACE") and player.is_on_ground:
        player.apply_force(0, -300) # Força para cima

    fisica.update_physics(janela.delta_time())
    camera.follow(player)

    # Desenha objetos com a câmera
    camera.begin()
    player.draw()
    chao.draw()
    camera.end()

    janela.update()
```

14. Gestão Coletiva e Inteligência

14.1. Usando ObjectGroup para Inimigos

Facilitando a vida de quem não quer gerenciar listas de objetos manualmente.

```
from PPlay.object_group import ObjectGroup
from PPlay.sprite import Sprite

# Criamos o container
grupo_inimigos = ObjectGroup()

# Adicionamos membros
for i in range(5):
    inimigo = Sprite("zombie.png")
    inimigo.set_position(100 * i, 200)
    grupo_inimigos.add(inimigo)

# No loop de jogo:
while True:
    # Desenha e atualiza todos de uma vez!
    grupo_inimigos.update()
    grupo_inimigos.draw()

    # Checa se o player tocou em QUALQUER um deles
    atingido = grupo_inimigos.collided(player)
    if atingido:
        print("Fui tocado por um inimigo!")
```

14.2. Agendando Tarefas com o Timer

Como criar eventos temporizados (ex: regeneração de vida) sem variáveis globais sujas.

```
from PPlay.timer import Timer

def recuperar_vida():
    player.vida += 5
    print("Recuperando vida...")

# Executa a função a cada 2 segundos, para sempre
Timer.every(2.0, recuperar_vida)

while True:
    Timer.update() # Processa os agendamentos
    janela.update()
```

15. Guia de Performance e Clean Code

Para manter o seu jogo rodando a 60 FPS estáveis e com um código legível, siga as diretrizes abaixo, desenvolvidas durante a arquitetura da PowerPPlay 2.0.

15.1. O Uso Sagrado do Delta Time

Nunca altere a posição de um objeto usando valores fixos (ex: x += 5). Sempre multiplique a velocidade pelo `janela.delta_time()`. Isso garante que o jogador se mova na mesma velocidade em um PC de monitor 144Hz ou em um notebook antigo.

15.2. Gestão de Memória (Flyweight)

A engine faz cache automático de imagens. No entanto, evite carregar recursos dentro do loop `while True`. Instancie seus Sprites, Sons e Grupos no início do código ou no `__init__` das suas Cenas.

15.3. Eficiência em Larga Escala

TileMap Culling

Use TileMap para cenários grandes. O Culling impede desenhar blocos fora da câmera.

ObjectGroups

Use grupos para colisões. É mais performático que loops manuais.

Raycasting

Use para IA (visão), mas evite disparar centenas de raios simultâneos.

15. Construção de Mundos por Grade (TXT)

O módulo TileMap permite criar cenários complexos sem escrever uma única linha de set_position. O mundo é desenhado em um editor de texto simples e interpretado pela engine.

15.1. Anatomia do Arquivo de Mapa (.txt)

O arquivo de texto deve ser visualizado como uma grade (matriz). Cada caractere representa um bloco de tamanho fixo (definido no código).

- **Caracteres Livres:** Use símbolos como . ou espaços para áreas onde o jogador pode caminhar livremente (fundo).
- **Caracteres de Colisão:** Use símbolos como #, X ou @ para representar paredes e chão.

15.2. O Dicionário de Mapeamento (The Mapping)

Para que a engine entenda o seu arquivo, você deve fornecer um dicionário Python que liga cada caractere a uma imagem e define quem é sólido.

```
meu_mapeamento = {
    '#': 'parede_pedra.png',
    'X': 'plataforma_madeira.png',
    '.': 'fundo_caverna.png',
    'solido': ['#', 'X']
}
```

15.3. Implementação no Código

O processo de carga é feito em duas etapas: inicialização e carregamento físico.

Inicialização

Define o tamanho de cada tile em pixels: mapa = TileMap(tamanho_tile=32)

Carga

Lê o arquivo e aplica o mapeamento: mapa.carregar_mapa("fase1.txt", meu_mapeamento)

15.4. Vantagens do Sistema

Culling Automático

Só desenha blocos dentro da Camera. Permite mapas de 10.000+ blocos sem perda de FPS.

Iteração Rápida

Mude o visual alterando apenas o dicionário de mapeamento, sem mexer na lógica.

Física Integrada

Use mapa.get_solidos() para obter a lista exata de objetos para update_physics().

16. Tabela de Consulta Rápida

Classe / Módulo	Comando Principal	Descrição
Window	update()	Atualiza tela, tempo e eventos.
Window	delta_time()	Tempo entre frames (segundos).
Keyboard	key_down(tecla)	Detecta o instante do clique (único).
Keyboard	key_pressed(tecla)	Detecta tecla segurada (contínuo).
Sprite	setup_physics(motor)	Acopla o motor cinemático.
Sprite	update_physics(solidos)	Processa movimento, gravidade e colisão.
Camera	follow(alvo, suav)	Segue um objeto de forma suave.
Camera	transform_x(x_mundo)	Converte posição do mundo para tela.
Physics	Physics(grav, terminal)	Configura o motor de gravidade.
ObjectGroup	add(obj) / draw()	Gerencia coleções de objetos em massa.
ObjectGroup	collided(alvo)	Retorna o objeto colidido do grupo.
Timer	after(seg, func)	Agenda tarefa única.
Timer	every(seg, func)	Agenda tarefa repetitiva.
Tween	to(obj, attr, fim, t)	Interpola valores (Juice).
Lighting	lights.append(...)	Adiciona fonte de luz dinâmica.
Effects	shake(pwr, dur)	Faz a tela tremer.

17. Dicionário de Classes e Métodos

17.1. NÚCLEO (Window)

- **__init__(largura, altura, titulo, resizavel=True, pixel_art=True):** Inicializa o motor e o canvas virtual.
- **update():** Finaliza o frame, projeta o buffer virtual e processa eventos.
- **delta_time():** Retorna o tempo do frame em segundos (float).
- **get_fps():** Retorna a taxa de quadros atual.
- **set_background_color(cor):** Define a cor de limpeza (String ou RGB).
- **draw_text(texto, x, y, tamanho, cor, fonte):** Renderiza texto estático no buffer.
- **screen_to_virtual_coords(x, y):** Converte coordenadas reais para o espaço do jogo.
- **close():** Encerra o processo de forma limpa.

17.2. ENTRADA (Keyboard, Mouse, InputManager)

Keyboard:

- **key_pressed(tecla):** True se a tecla está segurada.
- **key_down(tecla):** True apenas no instante do clique.
- **draw_debug(x, y):** Mostra teclas detectadas na tela.

Mouse:

- **get_position():** Retorna (x, y) virtuais.
- **button_pressed(botao):** True se botão (1, 2 ou 3) está segurado.
- **button_down(botao):** True no instante do clique.
- **is_over_object(obj):** True se o cursor está sobre o Sprite/GameObject.

InputManager:

- **define_action(nome, lista_teclas):** Mapeia um nome para várias teclas.
- **is_active(nome):** Verifica se a ação está sendo executada.
- **action_pressed(nome):** Verifica disparo único da ação.

17.4–17.8. Física, Mundo, Inteligência e Arquitetura

17.4. FÍSICA E COLISÃO (Physics, Collision, RigidBody)

- **Physics(__init__(gravidade=800, atrito=0.9, limite_velocidade_terminal=1000, tipo_integracao="verlet")):** Cria o motor físico. Use gravidade para um mundo plataforma ou 0 para top-down. Configurações de atrito e velocidade terminal para ajuste fino.
- **update_physics(objetos_animados, objetos_estaticos):** Aplica física e resolve colisões. Importante: **somente os Sprites** que chamaram `setup_physics()` devem ser passados como `objetos_animados`.
- **set_global_gravity(g):** Altera a força da gravidade globalmente.

Collision:

- **collided(obj1, obj2):** Retorna `True` se os objetos se colidem. Retorna `False` se não. Ideal para colisões pontuais como um tiro acertando um inimigo.
- **collided_with_list(obj, lista_objetos):** Retorna o **primeiro** objeto da lista com o qual `obj` colidiu, ou `None`. Perfeito para checar se um jogador está tocando o chão ou múltiplos inimigos.

RigidBody: (Avançado) Usado para objetos com física mais complexa (como esferas e caixas interagindo). Normalmente, `Sprite.setup_physics()` é suficiente.

17.5. MUNDO E ORGANIZAÇÃO (TileMap, Camera, ObjectGroup)

- **TileMap(__init__(tamanho_tile)):** Cria um mapa baseado em tiles. `tamanho_tile` é o tamanho em pixels de cada célula.
- **carregar_mapa(caminho_arquivo_txt, dicionario_mapeamento):** Carrega o mapa de um arquivo `.txt` e aplica o mapeamento. O dicionário define quais caracteres correspondem a quais imagens e quais são sólidos.
- **get_solidos():** Retorna uma lista de Sprites que são considerados "sólidos" (para colisão).
- **render():** Desenha apenas os tiles visíveis pela câmera (culling automático).

Camera:

- **follow(alvo_sprite, suavidade=0.1):** Faz a câmera seguir um Sprite. A suavidade controla a rapidez da câmera para alcançar o alvo (0.1 é um bom ponto de partida).
- **set_limit(xmin, ymin, xmax, ymax):** Define os limites do mundo que a câmera não pode exceder.
- **transform_x(x_mundo) / transform_y(y_mundo):** Converte coordenadas do mundo (absolutas) para coordenadas da tela (relativas à câmera).

ObjectGroup:

- **add(obj):** Adiciona um Sprite ou GameObject ao grupo.
- **draw():** Desenha todos os objetos do grupo.
- **update():** Chama o método `update()` de todos os objetos do grupo.
- **collided(alvo_sprite):** Retorna o **primeiro** objeto do grupo que colidiu com `alvo_sprite`, ou `None`.
- **remove(obj):** Remove um objeto do grupo.

17.6. INTELIGÊNCIA E TEMPO (Timer, Tween, RayCasting)

- **Timer:** Utilitário para agendar funções.
- **after(segundos, funcao_callback, *args, **kwargs):** Executa `funcao_callback` uma única vez após `segundos`.
- **every(segundos, funcao_callback, *args, **kwargs):** Executa `funcao_callback` repetidamente a cada `segundos`.
- **cancel(funcao_callback):** Cancela um timer agendado.

Tween: Para animações suaves de valores.

- **to(objeto, atributo, valor_final, duracao, easing="linear", delay=0, on_complete=None):** Interpola o `atributo` do `objeto` para `valor_final` em `duracao` segundos. `easing` (ex: "out_quad", "in_out_sine") controla a curva de aceleração. `on_complete` é uma função chamada ao finalizar.

RayCasting: Para detecção de linha de visão.

- **cast_ray(ponto_origem_x, ponto_origem_y, angulo_graus, distancia_maxima, objetos_para_colisao):** Lança um raio. Retorna o primeiro objeto colidido e a distância, ou `None`.

17.7. ÁUDIO E EFEITOS (Sound, Effects, Lighting)

- **Sound(__init__(caminho_arquivo_audio, volume=1.0, loop=False)):** Carrega e prepara um arquivo de áudio.
- **play():** Toca o som.
- **stop():** Para o som.
- **set_volume(volume):** Ajusta o volume (0.0 a 1.0).

Effects: Efeitos visuais.

- **shake(intensidade=5, duracao=0.2, fade=True):** Faz a tela tremer. `intensidade` (em pixels), `duracao` (em segundos). `fade=True` reduz o tremor com o tempo.
- **fade_out(duracao=1, cor=(0,0,0), on_complete=None):** Escurece a tela gradualmente para uma `cor`.
- **fade_in(duracao=1, cor=(0,0,0), on_complete=None):** Clarea a tela gradualmente de uma `cor`.

Lighting: Iluminação dinâmica.

- **lights.append(Light(x, y, raio, r, g, b, intensidade)):** Adiciona uma fonte de luz. `raio` define o alcance, `r,g,b` a cor, `intensidade` a força.
- **render():** Processa e desenha todas as luzes.

17.8. ARQUITETURA (GameObject, Scene, GameManager)

- **GameObject:** Classe base para todos os objetos do jogo (jogadores, inimigos, itens). Possui `x, y, width, height, image` e métodos como `update(), draw()`.
- **Scene:** Uma tela ou estado específico do jogo (menu principal, fase 1, tela de game over).
- **__init__(window):** Inicializa a cena com a janela do jogo.
- **update(dt):** Lógica de atualização da cena.
- **draw():** Desenho da cena.
- **transition_to(nova_cena_classe, *args, **kwargs):** Muda para outra cena.

GameManager: Gerencia as cenas do jogo.

- **__init__(window, cena_inicial_classe):** Inicializa o gerenciador com a janela e a primeira cena a ser carregada.
- **run():** Loop principal do jogo, que chama `update()` e `draw()` da cena atual.

17.3. Geometria e Visuais (GameObject, GameImage, Animation, Sprite)

GameObject:

- **__init__(x, y, imagem=None, largura=None, altura=None):** Inicializa a posição e, opcionalmente, associa uma imagem ou define dimensões.
- **set_position(x, y):** Define a posição x, y do objeto no mundo.
- **get_position():** Retorna a tupla (x, y) da posição atual.
- **set_scale(escala_x, escala_y=None):** Define a escala horizontal e vertical. Se escala_y for None, usa escala_x.
- **get_scale():** Retorna a tupla (escala_x, escala_y) da escala atual.
- **set_rotation(angulo):** Define o ângulo de rotação em graus.
- **get_rotation():** Retorna o ângulo de rotação atual.
- **set_transparency(alfa):** Define o nível de transparência (0 a 255).
- **get_transparency():** Retorna o nível de transparência atual.
- **draw(camera=None):** Desenha o objeto na tela. Opcionalmente, pode ser ajustado pela câmera.
- **collides_with(outro_obj):** Verifica colisão com outro GameObject usando AABB.
- **is_out_of_screen():** Verifica se o objeto está fora da tela visível da câmera.

GameImage:

- **__init__(caminho_imagem):** Carrega uma imagem do disco. É o mesmo que Sprite, mas sem física.
- **set_origin(origem_x, origem_y):** Define o ponto de origem (pivô) da imagem para rotação e escala.
- **get_width():** Retorna a largura original da imagem.
- **get_height():** Retorna a altura original da imagem.
- **get_size():** Retorna a tupla (largura, altura) da imagem.
- **set_clip(x, y, largura, altura):** Define uma área de corte na imagem (para spritesheets).
- **clear_clip():** Remove qualquer área de corte definida.

Animation:

- **__init__(caminho_imagem, num_frames_x, num_frames_y, intervalo_frames, loop=True):** Cria uma animação a partir de uma spritesheet.
- **update():** Avança um frame na animação, respeitando o intervalo.
- **restart():** Reinicia a animação do primeiro frame.
- **is_playing():** Retorna True se a animação não terminou (e não é loop).
- **set_speed(fator_velocidade):** Ajusta a velocidade da animação.
- **set_current_frame(indice):** Define o frame atual da animação manualmente.

Sprite:

- **__init__(caminho_imagem, x=0, y=0, largura=None, altura=None):** Construtor de Sprite. Pode ser uma imagem ou um GameImage.
- **setup_physics(motor_fisica, tipo='retangular', densidade=1, atrito=0.2, restituicao=0):** Configura o corpo físico do Sprite.
- **update_physics(solidos_estaticos=[]):** Processa a física para o Sprite (gravidade, colisões, movimento).
- **apply_force(forca_x, forca_y):** Aplica uma força ao Sprite.
- **apply_velocity(vel_x, vel_y):** Define a velocidade do Sprite.
- **set_solid(is_solid):** Define se o Sprite é sólido (participa de colisões).
- **is_solid():** Retorna True se o Sprite é sólido.

18. Referências Bibliográficas

Para o desenvolvimento da PowerPPlay 2.0, foram consultadas as seguintes obras e documentações técnicas, seguindo os padrões de engenharia de software e computação gráfica:

- **PYGAME COMMUNITY.** Pygame-CE (Community Edition) Documentation. Disponível em: https://pyga.me/docs/. Acesso em: 2025.
- **NYSTROM, Robert.** Game Programming Patterns. Genever Benning, 2014. (Referência para as implementações de Flyweight, Observer e State Machine presentes na engine).
- **MILLINGTON, Ian.** Game Physics Engine Development: How to Build a Robust Commercial-Grade Physics Engine. CRC Press, 2010. (Base teórica para o motor de sub-stepping e resolução de eixos).
- **AMALFI, J.** A Pathfinding Algorithm in Grid-Based Environments. Academic Technical Report.
- **UFF - INSTITUTO DE COMPUTAÇÃO.** Documentação Original PPlay 1.0. Niterói, RJ.

19. Considerações Finais

A PowerPPlay 2.0 é uma carta de amor ao desenvolvimento de jogos no Instituto de Computação da UFF, espero que ele seja utilizado por muitos alunos.

O código aqui documentado é aberto, expansível e robusto, servindo como base sólida para que futuros alunos possam focar na criatividade e no design, sem serem limitados por barreiras técnicas de hardware ou imprecisões físicas.

Créditos de Desenvolvimento

Arquiteto e Desenvolvedor da Versão 2.0:
Kauã Neves Jesus de Paula
Universidade Federal Fluminense

Legado PPlay 1.0:
Prof. Esteban Clua, Prof. Anselmo Montenegro, Gabriel Saldanha, Adônis Gasiglia, Yuri Nogueira, Sergio Herman.

"A programação de jogos é a forma mais complexa e bela de arte interativa. Que a PowerPPlay 2.0 seja o seu pincel."

Niterói, Rio de Janeiro
Março de 2026