Balanceamento Dinâmico de Dificuldade em Jogos Eletrônicos

um subtítulo

Henrique Cerquinho

Dissertação apresentada ao
Instituto de Matemática e Estatística
da Universidade de São Paulo
para obtenção do título de
Mestra em Ciências

Programa: Ciência da Computação
Orientador: Prof. Dr. Marco Dimas Gubitoso

São Paulo
10 de agosto de 2017

Balanceamento Dinâmico de Dificuldade em Jogos Eletrônicos

um subtítulo

Henrique Cerquinho

Esta é a versão original da dissertação
elaborada pela candidata Henrique Cerquinho,
tal como submetida à Comissão Julgadora.

Esta seção é opcional e fica numa página separada;
ela pode ser usada para uma dedicatória ou epígrafe.

Agradecimentos

Do. Or do not. There is no try.

— Mestre Yoda

Texto texto. Texto opcional.

Resumo

Henrique Cerquinho. **Balanceamento Dinâmico de Dificuldade em Jogos Eletrônicos: *um subtítulo***. Dissertação (Mestrado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2017.

O objetivo deste trabalho foi estudar e implementar o balanceamento dinâmico de dificuldade em jogos eletrônicos. Foi desenvolvido um protótipo de um jogo no estilo bullet-hell. Dada a infame e elevada dificuldade encontrada neste estilo de jogo, o foco principal do estudo foi de torná-lo acessível a todos os públicos, independente da habilidade do jogador. Foram adotadas funções matemáticas que levam em consideração diversas variáveis do jogo e definem uma métrica de dificuldade a ser usada no balanceamento de inimigos e recursos em tempo real. Foi enfim criado um ciclo de jogabilidade capaz de se adaptar às ações do jogador e dirigir o nível de dificuldade do jogo, procurando otimizar o entretenimento proposto.

Palavras-chave: Dificuldade. Balanceamento dinâmico. Lógica matemática. Jogos eletrônicos.

Abstract

Henrique Cerquinho. **Dynamic Difficulty Balancing in Video Games: *a subtitle***. Thesis (Masters). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2017.

The main goal of this thesis was to study and implement the dynamic balancing of difficulty in video games. With that purpose in mind, a bullet-hell style game prototype was developed. Given the infamous high difficulty in this game genre, the main focus of the study was to make it as accessible to all publics as possible, regardless of the players ability. Mathematical funcions were used to consider many game variables and define a difficulty metric to be later used in the balancing of enemy difficulty and resource abundancy, all in gameplay time. Thus, a full cycle of playability was developed, capable of adapting to the player's actions and steer the level of difficulty of the game, aiming to optimize the enjoyment.

Keywords: Difficulty. Dynamic balancing. Mathematical logic. Video games.

Lista de Figuras

Lista de Tabelas

Lista de Programas

Sumário

Apêndices

Anexos

Capítulo 1

Introdução

1.1 Motivação

A dificuldade em um jogo tende a ser um elemento extremamente visível e importante para o seu aproveitamento. Jogos mais desafiadores são conhecidos por serem mais frustrantes, e os menos desafiadores, mais relaxantes. Porém, esse pensamento pode levar à falsa ideia de impossibilitação do aproveitamento por parte da dificuldade elevada. No artigo de Jesper Juul (2009) é discutido o verdadeiro papel da dificuldade em jogos, de forma enxuta e didática. Quando apresentamos o conceito de vitória em um jogo, não faria sentido termos este sem o conceito de derrota. O jogador não teria a sensação de superação no momento da vitória se o conceito de derrota não estivesse constantemente presente na experiência. A derrota entra é introduzida ao jogo de forma a induzir os usuários a reajustarem sua perspectiva perante ao jogo, como visto no estudo de Juul. O jogador é então levado a sempre pensar em novas estratégias e, consequentemente, adicionando conteúdo ao jogo.

O nivel de dificuldade de um jogo pode ser visto como o responsável pelo desenvolvimento nas habilidades do jogador em experimentos como o de Aponte *et al.* (2011), porém, neste mesmo experimento, vemos ela também influenciando fortemente o aproveitamento geral do jogo, e podendo prejudicá-lo se não definida apropriadamente. Segundo Crandall e Sidak (2020), jogos eletrônicos consolidam uma das formas de entretenimento mais consumidas atualmente, totalizando uma receita de 159.3 bilhões de dólares no mundo inteiro em 2020. Portanto é natural associarmos jogos a entretenimento e, consequentemente, imaginarmos que um dado jogo é divertido, porém adequar a dificuldade de um jogo à experiência desejada prova-se um trabalho difícil.

1.2 Objetivos

Considerando a tese apresentada, é necessário manter em mente que um jogador pode se sentir "violado" pelo jogo se este sofre mudanças drásticas em sua dificuldade durante ou entre as sessões jogadas (Hunicke, 2005). Com isso, foi almeijado um sistema que possibilitasse o uso de ajustes dinâmicos de dificuldade em um jogo, ao mesmo tempo que

se adapta ao nível da habilidade do jogador. Isso é, se a habilidade do jogador se enquadra na dificuldade atual do jogo, esta tende a aumentar, propondo um desafio adequado à habilidade do jogador. Em contrapartida, se o jogador é muito prejudicado pela dificuldade atual do jogo, esta tende a diminuir, com o fim de manter o jogador em um estado mais favorável à sua capacidade atual, porém também o forçando a adaptar suas estratégias e seu estilo de jogo e não o impedindo de "avançar" na dificuldade de jogo.

Para este fim, foi desenvolvido um protótipo de jogo no estilo *Shoot 'em Up* (2.3) através da plataforma de desenvolvimento *Godot Engine* (4), junto a um sistema de balanceamento dinâmico de dificuldade adequado ao jogo.

Capítulo 2

Dificuldade e Gêneros de Jogos

2.1 Arcades e Consoles

Arcades, mais comumente conhecidos como *Fliperamas*, foram introduzidas na década de 1930, com jogos mecânicos ainda hoje famosos, como *Pinball*. As máquinas eram projetadas com uma entrada de moeda, que quando inserida permitia que fossem jogados determinado número de vezes ("rounds" ou "vidas"). Não se afastando desse sistema, os jogos eletrônicos em gabinetes foram introduzidos apenas na década de 60 às *Arcades*. Como relatado por Green (2017), jogos como *Pac-Man*, *Donkey Kong*, *Space Invaders*, *Asteroids* e diversos outros títulos dominavam as *Arcades* no mundo inteiro.

A partir desse ponto, foram apenas introduzidas novas tecnologias aos jogos de *Arcade*. *Duck Hunt* foi considerado revolucionário por usar um controle no formato de um revólver, usado para mirar diretamente na tela, simulando um tiro-ao-alvo. Alguns anos depois, jogos de corrida como *Grand Prix* levaram desenvolvedores à ideia de um controle no formato de um volante e um pedal, simulando, similarmente ao *Duck Hunt*, um cenário de pilotagem mais próximo da realidade.

Por mais atraentes que os jogos de *Arcade* estavam se tornando, sua dificuldade tendia a surpreender mais ainda o jogador. Até este momento, o papel principal da dificuldade era de se aproveitar ao máximo do sistema de moedas do jogo. Quanto mais difícil o jogo, mais moedas seriam gastas para completá-lo, e poucos desenvolvedores deixavam de ver essa vantagem.

Com a entrada dos *consoles*[1], principalmente do *Nintendo Entertainment System*, também conhecido como *NES* e *Nintendinho*, jogos começaram a ser reestruturados, deixando de ter apenas uma fase e seu objetivo de arrecadar o maior número de pontos possível, e adquirindo uma estrutura mais linear, com diversas fases e, consequentemente, tendo seu objetivo principal deslocado para terminar todas as fases, ou derrotar o último chefe. Com isso, o papel da dificuldade sofreu uma leve mudança, deixando de buscar as moedas do jogador, e passando a ser a sustentação principal do tempo de jogo. Quanto mais difícil,

[1] Aparelho dedicado a rodar um determinado grupo de jogos, com hardwares de interface (ex.: Playstation 2, Nintendo GameBoy, Xbox360).

mais tempo o jogador levaria para terminar o jogo, portanto maior aproveitamento seria tirado. Ideologias como essa moldaram a grande maioria dos jogos de *NES* como jogos extremamente difíceis de se completar.

2.2 Cenário Atual

Após uma onda de jogos notoriamente difíceis, desenvolvedores começaram a explorar a ideia de dificuldades selecionáveis em jogos, em busca de poderem atender públicos mais abrangentes com seus projetos. Estava então emergindo a mecânica de, ao começar um *novo jogo*, era apresentada a opção de dificuldade, normalmente variando entre *fácil*, *médio* e *difícil*, mas não se limitando a essas opções.

Variáveis do jogo como quantidade de pontos de vida do jogador e dos inimigos oscilavam de acordo com a dificuldade escolhida. Os primeiros grandes jogos de sucesso a explorar mais a fundo a mecânica de diversos níveis de dificuldade, como *Halo*, *Call of Duty*, saga *The Elder Scrolls*, saga *Final Fantasy*, entre outros, moldaram esse padrão para uma grande parcela dos jogos futuros, porém não a perfeccionando instantaneamente. Picos na dificuldade eram comumente encontrados em níveis mais altos no jogo. As últimas dificuldades, normalmente rotuladas como *Impossível* ou termos do gênero, costumavam inflar exageradamente variáveis do jogo como pontos de vida dos inimigos, muitas vezes tornando o jogo praticamente impossível e distanciando o jogador do cenário de entretenimento que os desenvolvedores se propunham a oferecer.

Em paralelo, é possível observar certas estruturas de jogos que permitem ao jogador explorar suas mecânicas de modo a afetar a dificuldade. Por exemplo, jogos no estilo *Online Multiplayer*[2], jogadores frequentemente participam de partidas e deliberadamente evitam usar certos itens do jogo, de modo a propor um desafio maior a si mesmos, se exibir ou até mesmo estabelecer recordes. Neste último cenário, também vemos a grande presença das *Speedruns* em jogos *Singleplayer*. Estas consistem em o jogador tentar completar determinado jogo no período de tempo mais curto possível. Para isso, o jogo é estudado profundamente e suas estratégias são otimizadas, procurando pular fases, explorar pontos fracos de inimigos ou até mesmo *bugs* nos jogos. Essa cultura é popularmente conhecida por propor tarefas de dificuldade significativamente mais elevadas que o jogo "base", consequentemente demandando muito da habilidade do jogador, ao mesmo tempo que adiciona conteúdo ao jogo.

Com o decorrer das décadas de 2000 e 2010, desenvolvedores começaram a aprofundar a complexidade do balanceamento da dificuldade de seus jogos (BYTE, 1981). A introdução de *checkpoints*[3] tornou jogos extensos muito mais acessíveis (Moran, 2010), por impedir que a derrota levasse o jogador de volta para o primeiro nível do jogo, e reduzindo o quanto do jogo teria que ser rejogado. Jogos mais datados são conhecidos por praticamente não possuírem checkpoints, o que contribuia fortemente para sua dificuldade elevada. É importante, porém, manter em mente que um uso excessivo de *checkpoints* pode trivializar

[2]Em contrapartida ao *Singleplayer*, é um estilo de jogo onde diversas pessoas jogam uma partida juntas, em tempo real, através de seus respectivos aparelhos ou *consoles* e uma conexão à *internet*.

[3]Quando encontrado, garante que o jogador possa voltar a este ponto do jogo após ser derrotado. Funciona normalmente como uma "volta no tempo" para não punir tão severamente o jogador.

as conquistas do jogador, uma vez que este não será apropriadamente punido por seus erros.

Neste mesmo período, com o avanço das Inteligências Artificiais (IAs), jogos *Single-player*[4] introduziram novas dinâmicas, não apenas de dificuldade, mas de jogabilidade no geral. Um dos gêneros que mais se benficiou desse avançou foi o de jogos de luta. Originados da geração dos *arcades*, jogos de luta eram projetados no estilo *versus* para dois ou mais jogadores invariavelmente. Porém, com a entrada das IAs, foi possibilitado o modo de jogo para um único jogador, onde este enfrentaria um ou mais personagens controlados pelo próprio jogo, e, em muitos casos, com níveis de dificuldade selecionáveis. Com isso, uma grande gama de habilidades poderia ser atendida pelo jogo, ao mesmo tempo que era reduzida a necessidade da presença de outro jogador para que o jogo pudesse ser aproveitado ao máximo.

O recém-lançado *Risk of Rain 2*, desenvolvido pela *Hopoo Games*, tornou-se uma referência interessante para os estudos de dificuldade de jogos eletrônicos. Trata-se de um *Rogue-Like*[5] onde a dificuldade é aumentada com o decorrer do tempo, independente das ações do jogador, que por sua vez fica encarregado de "acompanhar" a dificuldade do jogo conforme esta avança. A progressão da dificuldade é baseada em valores como tempo de jogo, quantidade de jogadores, fase atual do jogador, entre outros valores, cuja fórmula será aprofundada em um capítulo mais adiante.

2.3 Gênero Escolhido

Com os avanços não apenas na tecnologia disponível para desenvolvimento de jogos, mas também na criatividade dos desenvolvedores, jogos eletrônicos começaram a tomar uma complexidade que tornou inevitável uma segmentação destes em gêneros. Atualmente, o gênero de um jogo não necessariamente definirá seu nível de dificuldade. Em grande parte dos gêneros de jogos podemos encontrar tanto jogos difíceis como fáceis. Porém, existe um gênero que se destaca entre os outros por englobar alguns dos jogos mais difíceis conhecidos, portanto o protótipo de jogo desenvolvido neste estudo seguiu este modelo de jogo.

O *Bullet Hell*, também conhecido como *Shoot 'em Up*, ou apenas *Shmup*, é um subgênero de jogos eletrônicos de tiro em 2D[6], onde o jogador controla um único personagem, normalmente uma espaçonave ou avião, este disparando projéteis em diversos inimigos, em uma única direção, enquanto foca paralelamente em desviar de uma quantidade demasiada de projéteis vindos de tais inimigos (Davies, 2008). Em grande parte dos títulos neste gênero, o jogador, uma vez que acertado por um tiro, perde qualquer poder que tenha acumulado ao longo da fase, e uma vida. Quando o seu número de vidas chega em 0, ele perde o jogo totalmente.

[4]Estilo de jogo projetado para ser jogado por uma única pessoa simultaneamente.

[5]Gênero de jogo onde os mapas são, normalmente, gerados aleatória ou proceduralmente. A derrota do jogador o força a recomeçar o jogo desde o primeiro mapa, porém permitindo que sejam desbloqueados elementos novos para o jogo neste processo.

[6]Em um jogo 2D, o jogador controla a posição do personagem em apenas duas dimensões. No caso dos *Shmup*s, o personagem se move apenas para cima, para baixo, esquerda e direita em um plano.

Uma das definições populares para um shmup, como discutido por Nexic (2006), engloba o conceito de não permitir que o jogador mate muitos inimigos sem precisar se preocupar com desviar de seus tiros. Grande parte da experiência de um *Shmup* se encontra na necessidade de concentração para desviar de todos os projéteis que apresentam perigo ao jogador. Isso é, trata-se de jogos onde o usuário dificilmente se sentirá superior aos inimigos, constantemente adaptando suas estratégias para superar os grandes desafios proporcionados. Nexic também menciona que, se este conforto for dado ao jogador, este estará mais propício a se frustrar quando for atingido por uma "bala perdida".

O gênero teve sua origem com o famoso *Space Invaders* em 1978 e seus sucessores *Galaxian* e *Defender*, em 1980, mas sua emersão foi com a série *Radiant Silvergun* em 1998 (Picard, 2013). Possibilitado pelo avanço na capacidade computacional tanto de computadores pessoais como de *consoles*, novas mecânicas e visuais foram introduzidos ao gênero rapidamente. Apesar de jogos no subgênero de *Shmups* terem sido e ainda serem classificados como jogos de nicho, a trajetória de seu avanço os tornou significativamente únicos e realçados dentro do gênero de jogos de tiro.

Os jogos que serviram como a maior inspiração para este estudo foram os da série *Touhou Project*, esta contendo mais de 20 jogos, desenvolvida únicamente por Jun'ya Ota, mais conhecido como *ZUN* (Loxely, 2013). Apesar de alguns jogos da série seguirem o gênero de jogos de luta, a grande maioria não apenas se enquadra no gênero de *Shmup* como também serve de referência até os dias de hoje. A enorme variedade de inimigos, personagens jogáveis, fases completas e percursos de treinamento encontrada na série foi o fator determinante para a escolha deste gênero para o estudo, uma vez que a criatividade define o limite de qualquer mecânica que venha a ser implementada. Poderes especiais dos personagens, padrões de projéteis de inimigos, padrões de movimentação de inimigos, interações dinâmicas de elementos (Rodriguez, 2003), entre outras, são características que influenciam fortemente a dificuldade do jogo, de acordo com o modo como são implementadas. Este estudo procura atuar principalmente sobre esses aspectos do jogo para atingir seu objetivo de adaptação dinâmica de dificuldade.

Capítulo 3

Decisões de *Design*

3.1 Ideias Iniciais

Inicialmente, foram avaliadas diversas possibilidades de gêneros para o jogo. O Artigo de Hunicke (2005) mostra as vantagens de uma abordagem baseada em colocar o jogador em uma situação onde enfrenta diversos inimigos, em oposição a um jogo de luta, onde ele poderia enfrentar apenas um, ou jogos de quebra-cabeça, onde normalmente não existem inimigos. Para isso, foi desenvolvido um pequeno protótipo no estilo *top-down*[1], onde o personagem principal podia atirar em qualquer direção usando uma gama de armas selecináveis, em uma *arena*. Isso é, ele se encontrava em uma fase fechada, com apenas algumas portas localizadas nos cantos, de onde os inimigos surgiam em ondas. Os inimigos então se direcionavam ao personagem, causando dano caso entrassem em contato com ele. Inimigos de tipos diferentes também foram desenvolvidos, como um personagem que se aproximava do jogador, e quando dentro de uma certa distância dele, atirava projéteis em sua direção.

Conforme estes tipos diferentes ameaças eram introduzidas, a ideia para o cálculo da métrica de dificuldade apresentou-se cada vez mais como um desafio. Não apenas as ameaças, mas o nível de liberdade que o jogador possuia para se movimentar pela fase, atirar em qualquer direção e se aproveitar da arquitetura das arenas inseriu muitas variáveis a serem consideradas na parametrização da dificuldade, tanto para leitura, avaliação dos movimentos do jogador e seu desempenho, quanto para aplicação de balanceamento, alteração de variáveis dos inimigos e até mesmo do personagem principal.

Com isso, a exploração de outro gênero tornou-se um caminho mais apropriado para os objetivos deste estudo. A simplicidade dos *Shmups*, quando comparados aos jogos de *Arena* em qualquer estilo, provou facilitar consideravelmente esta interpretação de ações do jogador para formação da métrica de dificuldade. Não se limitando a isso, porém, foi possível também generalizar o modelo dos inimigos de forma que, lendo essa métrica gerada, entre outras variáveis, o nível do inimigo se adapta em tempo real, tanto quando a dificuldade é aumentada, quanto diminuída.

[1] Jogos definidos pelo ângulo de visão do jogo, vendo o personagem por cima, em oposição à visão pelos lados, usada em jogos como *Super Mario Bros.*

3.2 *Design* do Protótipo

Em um cenário de desenvolvimento de jogos, é necessário definir firmemente os conceitos fundamentais do jogo, não apenas de um ponto de vista de *design*, pensando nos objetivos do jogo, quais mecânicas serão implementadas, mas também de um ponto de vista técnico, dessa vez pensando em que ferramentas utilizar e tipos de arquitetura de jogos para usar como base. Esta seção será dedicada a essas especificações funcionais e técnicas do jogo.

3.2.1 Estrutura Geral

Para entendimento da estrutura geral do jogo, utiliza-se a metodologia de *cenas*. Uma cena pode ser interpretada de formas diferentes dentro do jogo. Uma boa maneira de se visualizá-las é pensando em um "estado de jogo". Por exemplo, o *menu* principal do jogo terá uma cena dedicada apenas a ele, assim como o local onde o jogo se passa, neste caso, no espaço, também está separado em uma cena específica. Os tipos diferentes de cenas e suas funcionalidades serão aprofundadas em um capitulo mais adiante.

O menu principal do jogo possui as opções de *Novo Jogo*, iniciando assim um jogo no modo padrão, logo após uma tela de seleção de personagens; *Tutorial*, colocando o jogador em uma fase guiada por instruções de como jogar, quais botões utilizar e explicando todas as mecânicas implementadas que o jogador precisa saber; *Demo*, levando-o a uma demonstração de que tipos de padrões de tiros os inimigos podem gerar, o qual será aprofundado em um capitulo futuro também; e por fim, *Sair*, que fecha o jogo.

3.2.2 Progressão e Ondas

Ao iniciar um *Novo Jogo*, o jogador depara-se com a tela de seleção de personagens (naves) mecionada anteriormente. Ao escolher entre uma das naves, uma tela contendo apenas a nave escolhida, o fundo temático do espaço sideral e indicadores da quantidade de vidas e de bombas do jogador é apresentada na sequência. O jogador pode se mover livremente dentro das delimitações laterais da tela e atirar sem se preocupar com sua munição. Após alguns segundos, os primeiros inimigos aparecerão pelo canto superior da tela. Estes fazem parte da primeira onda de inimigos. Os inimigos então começarão a disparar projéteis, em direção à nave principal ou não, de maneiras pré programadas. Cada inimigo permanece um tempo determinado na tela antes de ser arrastado para sua lateral mais próxima e *despawnando*[2], caso não seja derrotado pelo jogador antes. Quando todos os inimigos de uma *onda* são derrotados ou *despawnados*, é iniciada a onda seguinte de inimigos, podendo conter inimigos mais fortes, mais fracos, em maior ou menor quantidade, parâmetro a ser definido no banco de dados do jogo.

Ao chegar na última onda de inimigos, o jogador é apresentado ao chefe principal da fase. O chefe possui uma quantidade considerável a mais de pontos de vida, tornando mais difícil de matar, assim como ataques bastante distintos dos de inimigos comuns, e também em maior variedade. O chefe também pode se movimentar uniformemente pelo

[2]Em oposição ao *spawn*, que significa o *inserir* de uma entidade no estado atual do jogo, o *despawn* é quando uma entidade é liberada e deletada da cena atual do jogo.

topo da fase, de forma a forçar o deslocamento do jogador caso este esteja focando seu posicionamento em um único ponto. Podem também haver *sub-chefes* ou *mini-chefes* antes da onda final da fase, estes proporcionando uma batalha similar à com o chefe final, porém em uma dificuldade levemente inferior. Ao derrotar o chefe final, a fase atual acaba e o jogador vence o jogo após ser parabenizado.

3.2.3 Poderes Especiais

Ao entrar na fase, o jogador terá em seu arsenal dois principais meios de combate aos inimigos. O mais usado será o seu tiro primário. Como mencionado anteriormente, este não custará munição, portanto poderá ser usado a vontade. Ao derrotar um inimigo, este depositará na fase uma quantidade determinada de *drops*[3] de cristais, que quando coletados pelo jogador, somarão um número aleatório, em um intervalo específico, de pontos para o jogador. Ao juntar uma quantidade de pontos, terá o *nível* de seu tiro primário aumentado, tornando-o mais forte, com alcance maior e, assim, mais versátil e ameaçador. O jogo também apresenta uma mecânica de *auto-pickup-zone*. Se houverem *drops* na tela, o jogador pode, alternativamente a se aproximar dos drops, entrar na zona no topo da tela denominada *auto-pickup-zone*, fazendo assim com que todos os *drops* vivos se direcionem instantaneamente ao jogador, recompensando-o por entrar na zona mais perigosa da tela, onde os inimigos se encontram.

O estilo do tiro primário varia de acordo com a escolha de nave, isso é, tanto o primeiro nível do tiro quanto os demais serão diferente para cada nave implementada. Isso disponibiliza diversas opções de jogabilidade, ao mesmo tempo que não causa um impacto severo na dificuldade do jogo nem na sua complexidade, permitindo também que as mecânicas de balanceamento se mantenham estabilizadas.

O outro método de ataque disponível é a bomba. Esta, por sua vez, não será tão abundante quanto o tiro primário. Como notável pelo jogador assim que a fase é iniciada, há um contador de bombas disponíveis junto ao número de vidas. Quando uma bomba é usada, esse contador é decrementado uma unidade, e se não houverem bombas disponíveis, o jogador terá que continuar derrotando inimigos utilizando apenas seu tiro primário como ataque antes de poder usá-las novamente. Similarmente à progressão de níveis do tiro primário, ao coletar um determinado número de cristais de inimigos, o jogador ganhará uma bomba. Quando usada, a bomba causa uma explosão visível, causando dano massivo a todos os inimigos instantaneamente, e também limpando quaisquer projéteis inimigos que estiverem na tela. Com isso, a bomba se torna um recurso valioso que deve ser usado estrategicamente, possivelmente sendo guardada para o encontro com o chefe da fase.

Por último, foi implementada uma habilidade muito comumente encontrada em *Shmups*, conhecida como *Strafe*, ou *Focus*, assim chamada em *Touhou Project*. Enquanto o botão designado ao *Strafe* estiver pressionado, a nave do jogador terá sua velocidade reduzida, e um pequeno indicador da *hitbox*[4] da nave se tornará aparente. Ao soltar o botão, a

[3] Em jogos, quando um inimigo deixa para trás algum item ao morrer, este item é denominado *drop*.

[4] Nome dado à zona de colisão de um objeto com outros em jogos. Isso vale tanto para a nave principal quanto para os inimigos. Quando um projétil entra na *hitbox* do jogador ou de um inimigo, este levará o dano no tiro.

velocidade da nave volta ao normal e o indicador da *hitobox* desaparece. Isso se prova útil não apenas para desviar de uma leva densa de balas com mais precisão, mas também para tornar evidente a verdadeira distância de perigo das balas, uma vez que, em grande parte dos *Shmups*, a *hitobox* da nave principal é consideravelmente menor do que a sua *sprite*[5], dificultando a tarefa de desviar das balas inimigas quando este indicador não está visível.

3.2.4 Visuais e Sons

Indicadores visuais e/ou sonoros provam-se convenientes ao permitir que o jogo expresse eventos que, de outra forma, não seriam notados pelo jogador. O artigo de Lovato (2015) apresenta boa parte dos conceitos básicos de *expressão* em jogos e como o uso de certas técnicas consegue separar um simples protótipo de um jogo completo. A mais importante delas, o *tweening*, ou *inbetweening*, é uma técnica de interpolação de valores utilizada constantemente em jogos. Quando se trata de animar movimentos, usar uma interpolação não linear pode ser a chave para um resultado natural, uma vez que praticamente nada no mundo real se move ou se comporta de forma totalmente linear, como citado por Lovato, 2015.

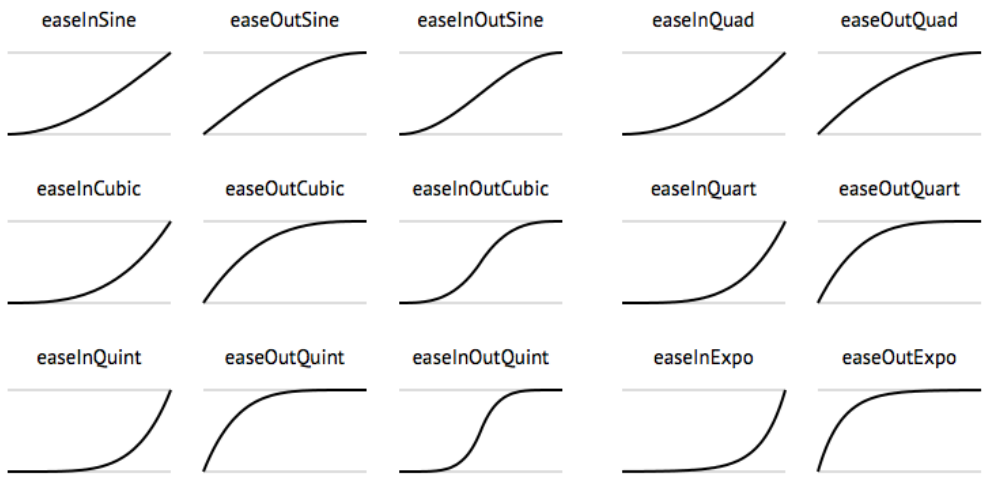


Figura 3.1: *Funções comumente usadas em interpolações de valores (Imagem retirada do artigo de Lovato, 2015).*

O uso de *tweens* se torna extremamente aparente quando utilizamos funções quadráticas, cúbicas, senoidais ou até mesmo elásticas para realizarmos a interpolação. Este efeito pode ser visto, por exemplo, no uso da bomba. Quando uma bomba é utilizada, uma circunferência (efeito da explosão) envolvendo a nave é rapidamente desenhada, tendo seu raio aumentado através de um *tween* baseado em uma função cúbica. Após algumas frações de segundos, o raio da circunferência passa a ser aumentado em uma taxa consideravelmente menor, até finalmente parar e desaparecer, passando assim a imagem de uma explosão para o jogador.

[5]Imagem do objeto desenhada na tela. No caso do personagem principal, é o desenho de sua nave.

Já não dependendo tanto dos *tweens*, alguns dos *sprites* de balas utilizados no jogos possuem um formato similar a de um losango achatado, com o intuito de clareza quanto à direção em que a bala se move. Balas circulares, por outro lado, podem deixar o entendimento de sua direção mais difícil para o jogador, e por conta disso são utilizadas mais frequentemente por chefes e sub-chefes.

Não se limitando aos visuais, a expressividade do jogo depende também de como os efeitos sonoros são utilizados. Eventos como *ganhar uma vida* e *ganhar uma bomba* são convenientemente demonstrados através de um efeito sonoro curto, distintos entre si, podendo também ser acompanhados de um pequeno indicador visual acima da nave. Neste caso, o *tweening* também se mostra conveniente para suavizar a animação do texto (algo como *Life up!* ou *Bomb get!*) para cima e tornar o evento menos engessado.

Capítulo 4

Plataforma de Desenvolvimento

4.1 Estrutura Básica

Quando se trata de desenvolvimento de jogos, o conceito de programação não pode se limitar aos tipos de programa "*batch mode*", isso é, programas que são executados em apenas duas etapas: o programador executa o programa; o programa devolve um resultado (Mendonça, 2018). Esse modelo é frequentemente usado, ainda mesmo nos dias atuais, para processamento de grandes volumes de dados, suportar reprodutibilidade, flexibilidade em termos de momento de execução, e geração de logs detalhados, entre outros cenários (FSF, 2007).

Os programas interativos, por sua vez, compõem a estrutura dos jogos digitais como conhecidos atualmente. Este modelo, por sua vez, se resume ao pequeno bloco de código a seguir.

Programa 4.1 Game Loop.

```
1   while True:
2      processInput()
3      update()
4      render()
```

Essa estrutura é conhecida como *Game Loop* (Nystrom, 2014). O jogo primeiramente processa o input recebido pelo usuário, atualiza as variáveis do jogo de acordo e, finalmente, desenha os resultados na tela. Este pequeno trecho de código é projetado para ser executado, idealmente, um total de 60 vezes por segundo, em intervalos de tempos iguais, gerando a denominada *frame*[1]. Com isso, cada *frame* do jogo será apresentada apenas durante aproximadamente 0.016 segundos até ser substituída pela próxima pelo código do *Game Loop*.

Por uma questão de simplicidade e para eliminar a necessidade de recriar essa estrutura para qualquer programa interativo, *frameworks* dedicadas para o desenvolvimento

[1]Um "quadro" do jogo. A rápida sequência de quadros gerados pelo *game loop* resulta na animação geral do jogo.

desse tipo de programa tornaram-se disponíveis, mais especificamente para o cenário de desenvolvimento de jogos, as *Game Engines*[2]. Também conhecidas como *Motor de Jogo*, essas auxiliam fortemente no desenvolvimento tanto jogos 2D quanto 3D, além de uso de gráficos, audio, bancos de dados, físicas, ou até mesmo uso de outras *frameworks* para, por exemplo, comunicação entre redes. A *engine* utilizada para desenvolver o protótipo neste estudo foi a *Godot Engine*.

4.2 Funcionalidades e Aplicações da Godot

A *Godot Engine* é uma *engine* totalmente gratuita e *open-source* sob a licença MIT, dedicada para o desenvolvimento de aplicativos 2D ou 3D, principalmente jogos eletrônicos, e tem sua manutenção feita atualmente pela comunidade de forma independente. A versão da engine utilizada neste estudo foi a 3.2.3, parte da versão 3.0 da *Godot*, lançada em 2018. Para entendimento do protótipo desenvolvido e do estudo feito, esta seção será dedicada a explicar os conceitos e funcionalidades principais da *engine*.

4.2.1 Nós

Também conhecidos como *Nodes*, quando separados, representam pequenos elementos do jogo. Quando juntos, compoem a estrutura geral do jogo, como blocos de construção. Na *Godot*, exitem diversos tipos diferentes de nós, cada um com uma função específica, e possivelmente herdando funcionalidades de outros tipos de nós. Por outro lado, os nós terão por padrão as seguintes utilidades:

- Possuem um nome;
- Possuem atributos (variáveis);
- Podem possuir ou não um trecho de código que é chamado toda *frame* do jogo, definido em um *script*;
- Podem possuir ou não métodos customizados, definidos no *script* mencionado;
- Podem herdar funcionalidades de outros nós ou scripts criados;
- Podem ser inseridos como *filhos* de outros nós, compondo uma *cena*.

Usufruindo de todas essas funções, o desenvolvimento do jogo é focado quase totalmente no uso dos nós.

4.2.2 Cenas

Como mencionado anteriormente em 3.2.1, uma cena pode representar tanto um estado específico do jogo, por exemplo uma fase, como um determinado "objeto", por exemplo, o personagem principal ou um inimigo. Cenas são sempre compostas por um ou mais nós em uma estrutura hierárquica. O editor da *Godot* possui uma seção intuitiva para criação e modificação dessa hierarquia, como demonstrado a seguir.

[2] Alguns exemplos de *engines* famosas hoje em dia: Unity; Unreal Engine; CryEngine; GameMaker Studio; RPG Maker; Godot.

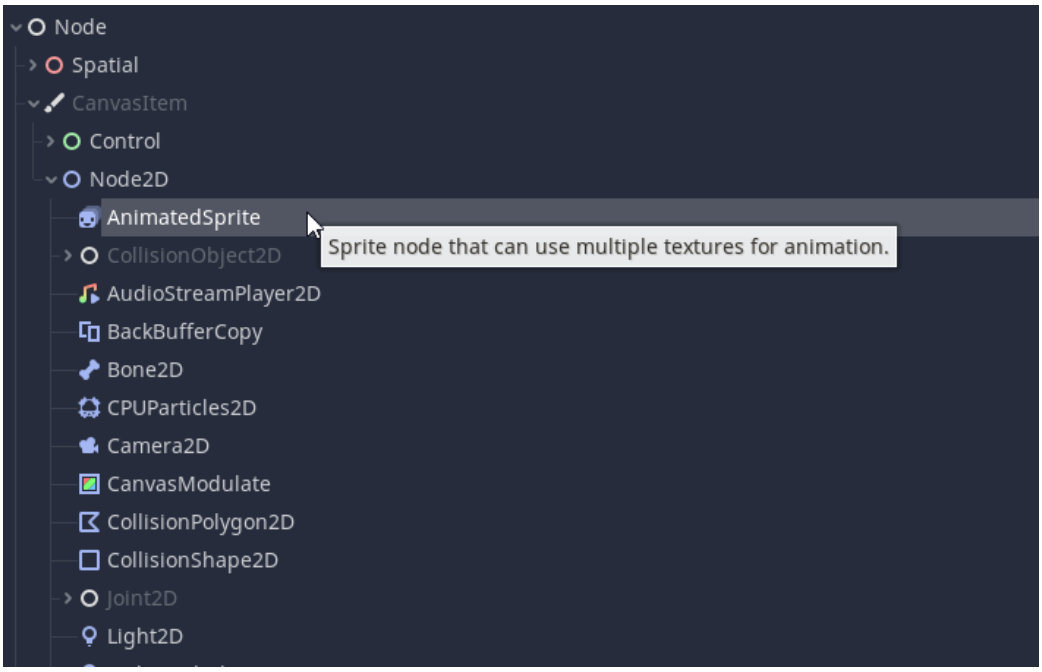


Figura 4.1: *Tela de criação de novo nó, dentro da Godot.*

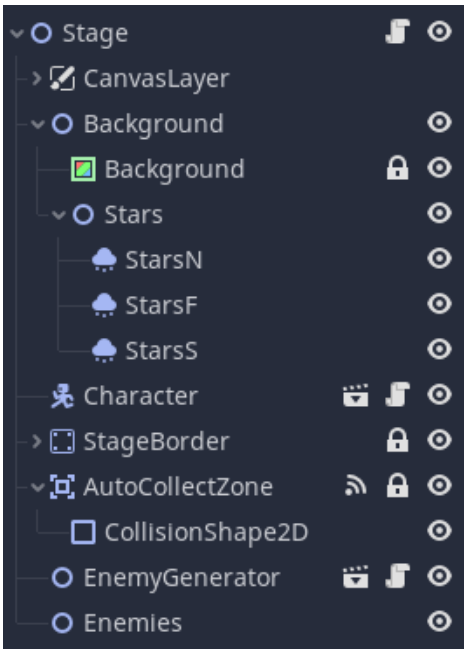


Figura 4.2: *Árvore de nós da cena/fase principal do jogo.*

O nó principal *Stage* está no topo da hierarquia da cena, portanto este controlará alguns dos atributos de seus nós "filhos", como sua posição na tela. Isso é, a posição absoluta de um nó será sempre a soma vetorial de sua própria posição e a posição de seu nó "pai".

Ao executar o jogo, significa que uma cena é executada. É possível executar a cena do personagem principal sem ela estar sendo instanciada em nenhuma outra cena. No protótipo criado, porém, sempre será executada uma cena de fase (ou *Stage*) contendo tal personagem, assim tornam-se possíveis as interações entre os ataques e movimentos do personagem com o cenário, inimigos e itens. Com isso, cenas provam-se úteis para, principalmente, organização do projeto e uso de *instanciação*.

4.2.3 Instanciação

Ao estruturar uma cena, esta pode ser tratada como uma classe e ser instanciada como um objeto para fazer parte da hierarquia de qualquer outra cena, assim como nós simples. Isso provou-se extremamente útil no desenvolvimento do protótipo em questão para, por exemplo, geração de projéteis. Com o alto número de projéteis ativos em *Shmups* no geral, basta definir um projétil simples e instanciá-lo diversas vezes na posição do personagem ou do inimigo para gerar um cenário simples de combate.

A depender do cenário em que uma cena é instanciada, porém, torna-se necessário carregá-la previamente através de um *script*, para possibilitar sua instanciação sob demanda, por exemplo, do *input*[3] do jogador. Este processo é feito frequentemente na definição de inimigos e do personagem no protótipo.

4.2.4 Sinais

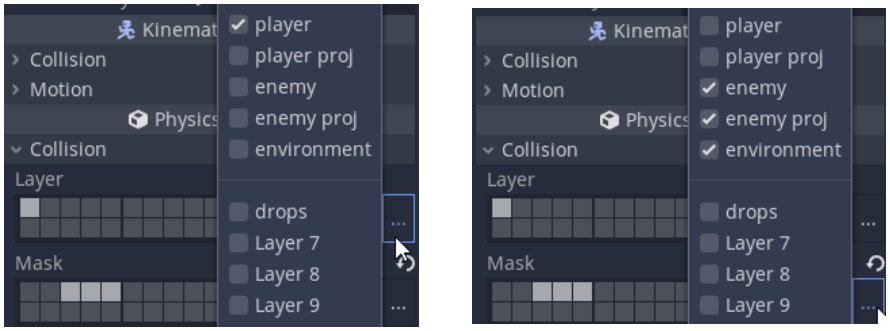
Para que nós e cenas possam detectar ocorrência de eventos fora de sua própria hierarquia, a *Godot* disponibiliza um sistema de emissão e recepção de sinais. Um sinal pode ser emitido por um nó e recebido por um ou mais outros nós ou cenas, acionando assim, um método determinado na configuração do sinal. Muitos nós possuem sinais pré definidos, porém é possível criar sinais customizados através do *script* da cena. Sinais provam-se convenientes para checagem de colisão de qualquer tipo. Por exemplo, quando um nó do tipo *Area2D* intercala seu corpo com algum outro nó *KinematicBody2D*[4], *StaticBody2D* ou *RigidBody2D*, este dispara um sinal de *Body Entered*, configurado para tratar a colisão destes dois corpos. Um exemplo mais concreto seria um tiro do personagem entrando em contato com um inimigo, e este, com isso, executando seu método de *receber dano*.

4.2.5 Grupos

Para um melhor desempenho na checagem de colisões, objetos que podem colidir com outros podem ser adicionados a grupos. Após definir a qual grupo um objeto pertence, é necessário também definir com quais grupos o objeto pode interagir. A figura a seguir mostra o grupo do personagem principal e seus grupos interagíveis.

[3] Qualquer ação, parametrizada no jogo, vinda do jogador, como um clique do mouse, ou uma tecla apertada no teclado.

[4] Nó principal do personagem, este possuindo métodos úteis de física para controle de movimento.



(a) *Grupo ao qual o personagem faz parte (Layer).* **(b)** *Grupos com os quais o personagem pode interagir (Mask).*

Figura 4.3: *Uso de grupos para colisão do personagem.*

Isso é, o personagem poderá interagir apenas com objetos pertencentes aos grupo *enemy*, *enemy proj* e *environment*, e com isso, colisões com outros grupos serão ignoradas. Este tipo de arquitetura contribui para a otimização geral do jogo ao não permitir que projéteis interajam com outros do mesmo tipo. Assim, centenas de checagens de colisão por segundo são poupadas, liberando tempo de processamento para outros cálculos.

4.2.6 Herança

Ao definir classes customizadas a partir de *scripts*, é possível fazer com que outras cenas herdem (extendam) tanto classes padrão da *Godot* quanto o *script* criado, dando acesso a métodos e atributos da classe herdada. Cenas como os diferentes níveis de projéteis do jogador herdam a classe base de "projétil". Nela, são definidos métodos de movimento básico do projétil, tratamento de colisão e atributos básicos como velocidade, direção e dano. Cálculos mais avançados e específicos do personagem são feitos em classes mais abaixo na herança.

4.2.7 Singletons

Com o uso de *Singletons* é possível ter acesso globalmente a métodos e variáveis definidas no nessa classe, por qualquer outro *script*. Similarmente ao uso de sinais, o uso de *Singletons* permite uma rápida e simples comunicação entre cenas distintas. O código a seguir, usado no protótipo em muitas cenas, auxilia, por meio de *setters* e *getters*, o acesso às cenas do personagem, fase principal, e *stats*, uma cena utilizada durante o desenvolvimento, dedicada a expor de variáveis na tela, como número de vidas, bombas, número de balas ativas e outras variáveis utilizadas no cálculo da dificuldade.

Programa 4.2 *Main Nodes Getter.*

```
1   extends Node
2
3   var character
4   var stage
5   var stats
6
7   func get_character():
8      return character
9
10  func set_character(node):
11     character = node
12
13  func get_stage():
14     return stage
15
16  func set_stage(node):
17     stage = node
18
19  func get_stats():
20     return stats
21
22  func set_stats(node):
23     stats = node
```

4.2.8 Banco de Dados

Não sendo obrigatório no desenvolvimento de jogos, porém, o uso de um banco de dados facilita na definição e na variação de diferentes objetos. No protótipo desenvolvido, esses objetos foram, principalmente, as fases e os inimigos. Como a *Godot* não inicializa projetos por padrão com bancos de dados, foi utilizado um outro *Singleton* a fim de criar um gerenciador simples de banco de dados para uso interno no jogo, com foco na recuperação de dados.

Programa 4.3 *Database Manager* utilizado no protótipo.

```
1   extends Node
2
3   func get_vars(filename):
4      var params = {}
5      var file = File.new()
6      file.open("res://db/"+filename, file.READ)
7      var text = file.get_as_text()
8      params = parse_json(text)
9      file.close()
10     return params
11
12  func get_bullet_gen(name):
13     return get_vars('generators/'+name+".json")
14
15  func get_stage(name):
16     return get_vars('stages/'+name+".json")
```

Para definição de fases e ondas (3.2.2), assim como padrões de ataques de inimigos, foram utilizados arquivos *JSON*, os quais são recuperados utilizando as funções do singleton em questão e, em seguida, utilizados na instanciação de cenas sob demanda.

4.2.9 Linguagem de Programação

GDScript, a principal linguagem utilizada na *Godot*, é uma linguagem de programação de tipagem dinâmica de alto nível, muito semelhante a *Python* e *Lua*. A linguagem é otimizada para integrar com a *engine*, além de ser simples e elegante, como as linguagens nas quais é inspirada. O uso da linguagem junto à *Godot* possui diversas vantagens, entre elas:

- É compilada rapidamente pela própria *engine*;
- Possui diversas classes úteis para criação de jogos, como vetores 2D e 3D, matrizes e diversas funções de álgebra linear, classes de *Object* (a qual inclui nós) e identificadores de caminho de nós e *Resources*[5], além de classes mais básicas, como listas e dicionários;
- Conversa dinamicamente com a interface da *engine*, reconhecendo nós da cena atual, *Singletons*, *Resources*, entre outros elementos da interface;
- Suporte a trechos de código em C e C++ para otimização;
- Suporte para uso de múltiplas *threads*.

Além disso, o editor possui diversas ferramentas dedicadas à depuração do código, permitindo o uso de *breakpoints*, inspecionar o estado atual da execução, valores de variáveis, chamadas de funções e tempo de execução em cada uma delas, essa última sendo crucial para a etapa de otimização do jogo mencionada em 4.2.5.

[5] Arquivos do jogo, variando desde arquivos de *scripts* até cenas, *sprites* e vídeos.

Capítulo 5

Balanceamento Dinâmico

5.1 Conceito de Interpretação

Com o objetivo de medir a dificuldade de um jogo e parametrizá-la em uma ou mais variáveis utilizáveis pelo desenvolvedor, é proposta a divisão da experiência geral do jogo em desafios menores e separados, para realização, assim, da medição da dificuldade de cada um destes desafios, assim como realizado por Aponte *et al.* (2011). Aponte propõe que a interpretação de *um desafio* como *o objetivo de obter um determinado resultado* permite a extração de um conjunto de desafios que compoem a experiência de jogo apresentada ao jogador.

Com isso, ao medir a dificuldade de tais desafios que compoem o jogo, torna-se possível medir a dificuldade geral do jogo em questão. Estes desafios menores podem derivar tanto de elementos gerados dinamicamente, quanto estaticamente, como por exemplo fases e inimigos descritos em um banco de dados, método usado neste estudo.

5.2 Habilidades e Desafios Medidos

Independente da origem de um desafio em um determinado estado do jogo, o sistema implementado sempre interpretará as ações do jogador em intervalos de tempo em termos de *sucesso* ou *falha*. Essa separação torna-se necessária para o uso de uma heurística para aproximação da dificuldade dos desafios. Tendo essas duas opções em mente, a dificuldade de um desafio é estimada pela probabilidade do jogador *falhar* em concluir tal desafio com sucesso. Além disso, é importante pensar também na variação contínua de dificuldade por parte das habilidades do jogador. Conforme este se adapta aos desafios, sua habilidade cresce, e portanto a sua probabilidade de falhar os desafios diminui. Com isso, os desafios e suas recompensas devem crescer de acordo. Estes conceitos e suas abordagens de implementação serão aprofundados ainda neste capítulo.

Em favor da simplicidade, os desafios são ainda separados em categorias menores, denominadas *tarefas simples* e *tarefas compostas* por Aponte *et al.* (2011) e Juul (2009). *Tarefas simples* representam as ações mais simples possíveis do jogo, como movimentação, ataques, desvios, bloqueios, entre outras, por isso tendem a ser repetidas diversas vezes em

um curto período de tempo. *Tarefas compostas* tendem a consumir mais tempo, diversas ações, e serem relativamente únicas, como a tarefa de derrotar um inimigo qualquer, derrotar um chefe, sobreviver uma onda de inimigos, ou até mesmo completar o jogo.

Como as tarefas simples tendem a ser apresentadas diversas vezes para o jogador, estas não proporcionarão uma recompensa tão considerável quanto a de tarefas compostas, uma vez que a habilidade do jogador não mudará de forma drástica o suficiente para tal. Por outro lado, tarefas compostas apresentarão uma recompensa maior, assim como também estarão sucetíveis a variações de recompensa maiores entre suas conclusões.

Para cada tarefa, é necessário definir uma ação de conclusão e uma condição de recompensa. Por exemplo, o jogador terá constantemente a tarefa de *não ser atingido por balas inimigas*. Isso é, para cada bala inimiga que o jogador não for atingido, ele receberá uma recompensa, neste caso, pequena, pelo fato deste frequentemente enfrentar múltiplas balas simultaneamente. Caso falhe em concluir a tarefa (seja atingido), o jogador é punido (perde uma vida) e a dificuldade geral do jogo é diminuída de acordo.

5.3 Aplicação no Protótipo

5.3.1 Leitura

Neste estudo, o cálculo da dificuldade se baseou na seguinte divisão de ações/tarefas do jogador:

- Pontos de dano;
- Pontos de *graze*.

Graze, terminologia utilizada na saga *Touhou* (Loxely, 2013), denomina a ação de passar os mais próximo possível de uma bala inimiga com o a *hitbox* do personagem, em um *Shmup*, literalmente "raspando" próximo à bala.

Os pontos de dano são calculados considerando o tempo total desde que o jogador entrou em contato com uma bala inimiga e quantas balas inimigas estão ativas na tela no momento, da seguinte maneira:

$$no\_hit\_points = \frac{no\_hit\_time^{1.15} * n\_bullets}{500}$$

onde no_hit_time representa o tempo, em segundos, desde a última colisão do jogador com uma bala inimiga, e $n\_bullets$, o número de balas inimigas na tela no momento. Este valor é apenas considerado quando existem inimigos ativos no momento.

Os pontos de *graze*, por sua vez, são calculados da seguinte maneira:

$$graze\_points = grazed\_bullets * (overall\_difficulty + 1)^{0.2}$$

onde $grazed\_bullets$ é o número de balas "raspadas" desde o último cálculo de dificuldade. Após o cálculo, $grazed\_bullets$ é zerado. $overall\_difficulty$ é a dificuldade total

atual, calculada da maneira a seguir:

$$overall\_difficulty = overall\_difficulty + \frac{no\_hit\_points + graze\_points}{2}$$

tendo $overall\_difficulty$ inicializada com 0. O método a seguir, realizador do cálculo, é definido no *script* da fase e chamado pelo personagem apenas a cada meio segundo, a fim de evitar inflação de valores e otimizar o tempo de processamento.

Programa 5.1 Função principal de cáulculo da dificuldade.

```
1   func update_diff(no_hit_time, grazed_bullets):
2      var core_action_points = 0
3
4      var no_hit_points = 0
5      var graze_points = 0
6
7      # Points gained from not getting hit, considering time and amount of
8      # bullets on screen
9      no_hit_points = pow(no_hit_time, 1.15) * n_bullets / 500
10     graze_points = grazed_bullets * pow(overall_difficulty + 1, 0.2)
11
12     core_action_points = no_hit_points + graze_points
13
14     var accumulated_diff = core_action_points / 2
15     overall_difficulty += accumulated_diff
16
17     stats.update_diff(accumulated_diff, overall_difficulty)
18
19     for enemy in $Enemies.get_children():
20        for generator in enemy.get_node("Generators").get_children():
21           generator.update_diff(overall_difficulty)
```

Nas últimas 3 linhas do código, é feita a atualização de parâmetros dentro dos inimigos, sendo assim, a aplicação da dificuldade no ambiente do jogo.

5.3.2 Alteração de Inimigos

Os cálculos realizados acima são então interpretados pela classe de *gerador de balas* dos inimigos. O *gerador de balas* (ou apenas *gerador*) genérico é uma cena dedicada à criação de projéteis por parte dos inimigos. Esta foi desenvolvida para receber uma gama de atributos definidores do comportamento dos ataques do inimigo. Cada inimigo pode ter um número de geradores maior ou igual a zero. Alguns dos atributos do gerador seguem:

- $proj\_type$: Tipo de projétil a ser disparado;
- $life$: Tempo de vida de cada projétil gerado;
- $bullets\_per\_array$: Número de projéteis por *vetor* de projéteis;

- $individual\_array\_spread$: Ângulo formado por cada um dos vetores;
- $total\_bullet\_arrays$: Número de projéteis em cada vetor;
- $base\_spin\_speed$: Velocidade de rotação do gerador;
- $spin\_variation$: Variação da velocidade de rotação (aceleração de rotação);
- $aim\_at\_character$: *Flag* definindo se os vetores se direcionarão automáticamente à posição do personagem, ignorando a rotação do gerador;
- $bullet\_speed$: Velocidade dos projéteis;
- $fire\_rate$: Frequência na qual os projéteis são disparados;

Por exemplo, um *vetor* com 3 projéteis e 90° de abertura resultará em 3 balas com 45° de inclinação entre si, saindo do ponto de origem do gerador. A imagem a seguir mostra dois inimigos no topo da tela, ambos com um único vetor com abertura de 25°, porém o inimigo da direita com 3 projéteis por vetor, resultando em um projétil a mais no centro da abertura. Uma classe de exemplo e testes de geradores pode ser encontrada no apêndice A.

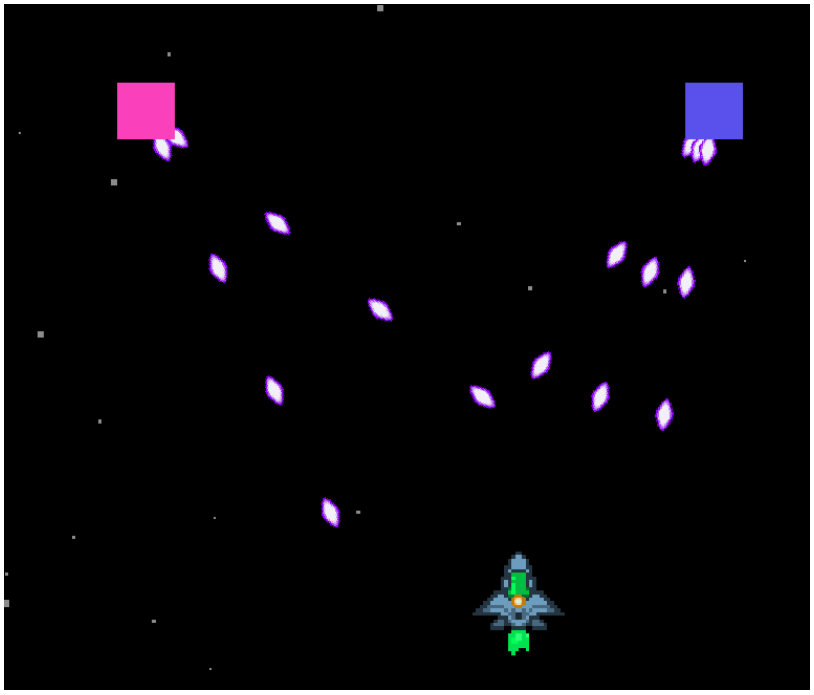


Figura 5.1: *Exemplo de geradores com projéteis (em roxo) focados no personagem.*

Estes dados são carregados sempre do banco de dados (4.2.8) para inicialização dos inimigos. Estes valores, porém, não serão sempre os mesmos para os inimigos. Conforme a variável $overall\_difficulty$ é alterada (5.3.1), modificadores para estes valores são calculados dentro dos inimigos também, processo executado nas 3 últimas linhas do programa 5.1. Nisso, as seguintes variáveis são calculadas.

$mod\_bullet\_speed = overall\_difficulty^{0.05} - 1$
mod_spin_speed =???

$$mod\_fire\_rate = log_{10} overall\_difficulty$$

$$mod\_bullets\_per\_array = \begin{cases} 0, & \text{se } overall\_difficulty < 50 \\ 1, & \text{caso contrário} \end{cases}$$

e, com isso, os valores do gerador são alterados da seguinte maneira, quando considerados na instanciação de projéteis.

$$bullet\_speed = mod\_bullet\_speed + bullet\_speed$$
$$spin\_speed = mod\_spin\_speed * spin\_speed$$
$$fire\_rate = mod\_fire\_rate * fire\_rate$$
$$bullets\_per\_array = mod\_bullets\_per\_array + bullets\_per\_array$$

aaaaaa

Capítulo 6

Resultados

6.1 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Apêndice A

Generator Demo

A cena *Generator Demo* consiste em uma fase com apenas um inimigo imortal, o personagem e um *menu* na lateral esquerda da tela. O menu permite o controle dos geradores do inimigo, permitindo adicionar geradores, selecionar um gerador criado e alterar qualquer parâmetro deste para testes. As alterações são refletidas no gerador selecionado em tempo real, facilitando a visualização de resultados de diferentes combinações de valores e auxiliando na escrita de fases e inimigos mais diversos.

Figura A.1: *Cena de demonstração de gerador de projéteis.*

Apêndice B

Código-Fonte e Pseudocódigo

Com a *package* listings, programas podem ser inseridos diretamente no arquivo, como feito no caso do Programa **??**, ou importados de um arquivo externo com o comando \lstinputlisting, como no caso do Programa B.1.

Programa B.1 Máximo divisor comum (arquivo importado).

```
1  FUNCTION euclid(a, b)  ▷ The g.c.d. of a and b
2      r ← a mod b
3      while r ≠ 0  ▷ We have the answer if r is 0
4          a ← b
5          b ← r
6          r ← a mod b
7      end
8      return b  ▷ The g.c.d. is b
9  end
```

Trechos de código curtos (menores que uma página) podem ou não ser incluídos como *floats*; trechos longos necessariamente incluem quebras de página e, portanto, não podem ser *floats*. Com *floats*, a legenda e as linhas separadoras são colocadas pelo comando \begin{program}; sem eles, utilize o ambiente programruledcaption (atenção para a colocação do comando \label{}, dentro da legenda), como no Programa B.2[1]:

Programa B.2 Máximo divisor comum (em português).

```
1  FUNCAO euclides(a, b)  ▷ O máximo divisor comum de a e b
2      r ← a mod b
3  enquanto r ≠ 0  ▷ Atingimos a resposta se r é zero
4          a ← b
5          b ← r
```

cont ⟶

[1] listings oferece alguns recursos próprios para a definição de *floats* e legendas, mas neste modelo não os utilizamos.

```
    ⟶ cont
6             r ← a mod b
7         fim
8     devolva b   ▷ O máximo divisor comum é b
9     fim
```

Além do suporte às várias linguagens incluídas em listings, este modelo traz uma extensão para permitir o uso de pseudocódigo, útil para a descrição de algoritmos em alto nível. Ela oferece diversos recursos:

- Comentários seguem o padrão de C++ (`//` e `/* ... */`), mas o delimitador é impresso como "▷".
- ":=", "<>", "<=", ">=" e "!=" são substituídos pelo símbolo matemático adequado.
- É possível acrescentar palavras-chave além de "if", "and" etc. com a opção "morekeywords={pchave1,pchave2}" (para um trecho de código específico) ou com o comando \lstset{morekeywords={pchave1,pchave2}} (como comando de configuração geral).
- É possível usar pequenos trechos de código, como nomes de variáveis, dentro de um parágrafo normal com \lstinline{blah}.
- "$...$" ativa o modo matemático em qualquer lugar.
- Outros comandos LaTeX funcionam apenas em comentários; fora, a linguagem simula alguns pré-definidos (\textit{}, \texttt{} etc.).
- O comando \label também funciona em comentários; a referência correspondente (\ref) indica o número da linha de código. Se quiser usá-lo numa linha sem comentários, use `///` \label{blah}; "`///`" funciona como `//`, permitindo a inserção de comandos LaTeX, mas não imprime o delimitador (▷).
- Para suspender a formatação automática, use \noparse{blah}.
- Para forçar a formatação de um texto como função, identificador, palavra-chave ou comentário, use \func{blah}, \id{blah}, \kw{blah} ou \comment{blah}.
- Palavras-chave dentro de comentários não são formatadas automaticamente; se necessário, use \func{}, \id{} etc. ou comandos LaTeX padrão.
- As palavras "Program", "Procedure" e "Function" têm formatação especial e fazem a palavra seguinte ser formatada como função. Funções em outros lugares *não* são detectadas automaticamente; use \func{}, a opção "functions={func1,func2}" ou o comando "\lstset{functions={func1,func2}}" para que elas sejam detectadas.
- Além de funções, palavras-chave, strings, comentários e identificadores, há "specialidentifiers". Você pode usá-los com \specialid{blah}, com a opção "specialidentifiers={id1,id2}" ou com o comando "\lstset{specialidentifiers={id1,id2}}".

Anexo A

Perguntas Frequentes sobre o Modelo[2]

- **Não consigo decorar tantos comandos!**
  Use a colinha que é distribuída juntamente com este modelo (gitlab.com/ccsl-usp/modelo-latex/raw/master/pre-compilados/colinha.pdf?inline=false).
- **Por que tantos arquivos?**
  O preâmbulo LaTeX deste modelo é muito longo; as partes que normalmente não precisam ser modificadas foram colocadas no diretório `extras`, juntamente com alguns arquivos acessórios. Já os arquivos de conteúdo (capítulos, anexos etc.) foram divididos de maneira que seja fácil para você atualizar o modelo (copiando os novos arquivos ou com um sistema de controle de versões) sem que alterações no conteúdo de exemplo (este texto que você está lendo) causem conflitos com o seu próprio texto.
- **As figuras e tabelas são colocadas em lugares ruins.**
  Veja a discussão a respeito na Seção **??**.
- **Estou tendo problemas com caracteres acentuados.**
  Veja a discussão a respeito na Seção **??**.
- **Existe algo específico para citações de páginas web?**
  Biblatex define o tipo "online", que deve ser usado para materiais com título, autor etc., como uma postagem ou comentário em um blog, um gráfico ou mesmo uma mensagem de email para uma lista de discussão. Bibtex, por padrão, não tem um tipo específico para isso; com ele, normalmente usa-se o campo "howpublished" para especificar que se trata de um recurso *online*. Se o que você está citando não é algo determinado com título, autor etc. mas sim um sítio (como uma empresa ou um produto), pode ser mais adequado colocar a referência apenas como nota de rodapé e não na lista de referências; nesses casos, algumas pessoas acrescentam uma segunda lista de referências especificamente para recursos *online* (biblatex permite criar múltiplas bibliografias). Já artigos disponíveis *online* mas que fazem parte de uma publicação de formato

[2] Esta seção não é de fato um anexo, mas sim um apêndice; ela foi definida desta forma apenas para servir como exemplo de anexo.

tradicional (mesmo que apenas *online*), como os anais de um congresso, devem ser citados por seu tipo verdadeiro e apenas incluir o campo "url" (não é nem necessário usar o comando \url{}), aceito por todos os tipos de documento do bibtex/biblatex.

- **Aparece uma folha em branco entre os capítulos.**
  Essa característica foi colocada propositalmente, dado que todo capítulo deve (ou deveria) começar em uma página de numeração ímpar (lado direito do documento). Se quiser mudar esse comportamento, acrescente "openany" como opção da classe, i.e., \documentclass[openany,...]{book}.

- **É possível resumir o nome das seções/capítulos que aparece no topo das páginas e no sumário?**
  Sim, usando a sintaxe \section[mini-titulo]{titulo enorme}. Isso é especialmente útil nas legendas (*captions*) das figuras e tabelas, que muitas vezes são demasiadamente longas para a lista de figuras/tabelas.

- **Existe algum programa para gerenciar referências em formato bibtex?**
  Sim, há vários. Uma opção bem comum é o JabRef; outra é usar Zotero ou Mendeley e exportar os dados deles no formato .bib.

- **Posso usar pacotes LaTeX adicionais aos sugeridos?**
  Com certeza! Você pode modificar os arquivos o quanto desejar, o modelo serve só como uma ajuda inicial para o seu trabalho.

- **Como faço para usar o Makefile (comando make) no Windows?**
  Lembre-se que a ferramenta recomendada para compilação do documento é o latexmk, então você não precisa do make. Mas, se quiser usá-lo, você pode instalar o MSYS2 (www.msys2.org) ou o Windows Subsystem for Linux (procure as versões de Linux disponíveis na Microsoft Store). Se você pretende usar algum dos editores sugeridos, é possível deixar a compilação a cargo deles, também dispensando o make.

- **Como eu faço para...**
  Leia os comentários dos arquivos "tese.tex" e outros que compõem este modelo, além do tutorial (Capítulo **??**) e dos exemplos do Capítulo **??**; é provável que haja uma dica neles ou, pelo menos, a indicação da *package* relacionada ao que você precisa.

Referências

[Aponte *et al.* 2011] Maria-Virginia Aponte, Guillaume Levieux e Stéphane Natkin. "Difficulty in video games : an experimental validation of a formal definition". Em: (2011), pg. 19 (citado nas pgs. 1, 21).

[BYTE 1981] BYTE. *"Zork, The Great Underground Empire.* 1ª ed. Perspectiva, 1981, pgs. 262–264 (citado na pg. 4).

[Crandall e Sidak 2020] Robert W. Crandall e J. Gregory Sidak. "Video games: serious business for america's economy". Em: (2020), pg. 48 (citado na pg. 1).

[Davies 2008] Jonti Davies. "The shooting never stops". Em: (2008) (citado na pg. 5).

[Green 2017] Katie Green. "A short history on arcade gaming". Em: (2017) (citado na pg. 3).

[Hunicke 2005] Robin Hunicke. "The case for dynamic difficulty adjustment in games". Em: (2005), pg. 6 (citado nas pgs. 1, 7).

[Juul 2009] Jesper Juul. "Fear of failing? the many meanings of difficulty in video games". Em: (2009), pg. 13. URL: https://www.jesperjuul.net/text/fearoffailing/ (citado nas pgs. 1, 21).

[FSF 2007] Archna Khubchandani. *Interactive Mode vs. Batch Mode in R Programming.* 2007. URL: https://study.com/academy/lesson/interactive-mode-vs-batch-mode-in-r-programming.html (acesso em 25/11/2020) (citado na pg. 13).

[Lovato 2015] Nathan Lovato. "Squeezing more juice out of your game design!" Em: (2015). URL: https://gameanalytics.com/blog/squeezing-more-juice-out-of-your-game-design/ (citado na pg. 10).

[Loxely 2013] Morgan Loxely. "'touhou project' offers tough levels, hours of fun". Em: (2013). URL: https://www.eghsguardian.com/2010/entertainment/touhou-project-offers-tough-levels-hours-of-fun/ (citado nas pgs. 6, 22).

[Mendonça 2018] Victor Domiciano Mendonça. "Inteligência artificial em um jogo procedural do gênero tower defense". Em: (2018), pg. 57. URL: https://bcc.ime.usp.br/tccs/2018/vidm/monografia.pdf (citado na pg. 13).

[Moran 2010] Chuk Moran. "Playing with game time: auto-saves and undoing despite the 'magic circle'". Em: (2010). URL: http://sixteen.fibreculturejournal.org/playing-with-game-time-auto-saves-and-undoing-despite-the-magic-circle/ (citado na pg. 4).

[Nexic 2006] Nexic. *Do's and don'ts of a good shmup*. 2006. URL: https://shmups.system11.org/viewtopic.php?t=7874 (citado na pg. 6).

[Nystrom 2014] Robert Nystrom. *Game Programming Patterns*. 1ª ed. Genever Benning, 2014, pgs. 123–139 (citado na pg. 13).

[Picard 2013] Martin Picard. "The foundation of geemu: a brief history of early japanese video games". Em: (2013). URL: http://www.gamestudies.org/1302/articles/picard (citado na pg. 6).

[Rodriguez 2003] Tyrone Rodriguez. "Ikaruga". Em: (2003), pg. 2. URL: https://web.archive.org/web/20160502211507/http://www.ign.com/articles/2003/04/09/ikaruga-3 (citado na pg. 6).

Índice Remissivo