Universidade de São Paulo
Instituto de Matemática e Estatística
Bachalerado em Ciência da Computação

Gustavo Spelzon Caparica

Bugs em jogos e suas aplicações em speedruns

São Paulo
Dezembro de 2018

Bugs em jogos e suas aplicações em speedruns

Monografia final da disciplina
MAC0499 – Trabalho de Formatura Supervisionado.

Supervisor: Prof. Dr. Alfredo Goldman
Cosupervisores: Guilherme Amantea e
Wilson Kazuo Mizutani

São Paulo
Dezembro de 2018

Resumo

Esse trabalho apresenta em detalhes a relação entre *speedrunners* e os *bugs* que eles utilizam. *Speedrunning* é a prática de terminar jogos o mais rápido possível, e para isso os jogadores utilizam falhas presentes no código do *videogame* para pular seções do jogo ou realizar manobras não esperadas pelos desenvolvedores. O texto explora características de *bugs* comumente exploradas por *speedruns* e dá exemplos de *bugs* existentes, explicando seu funcionamento. Também explora falhas que não podem ser facilmente replicadas por jogadores em tempo real, além das ferramentas e métodos utilizadas por *speedrunners* para entender o funcionamento de *glitches* do jogo. Por meio desse estudo, mostramos como a descoberta e utilização de *bugs* realizado por *speedrunners* é similar ao processo de engenharia reversa, explicando de maneira acessível os conceitos necessários.

Palavras-chave: *speedruns*, *tool-assisted speedruns*, *videogames*, *bugs*, *glitches*, emuladores.

Abstract

This thesis explains in detail the relationship between speedrunners and the bugs used in their speedruns. Speedrunning is the practice of finishing games as fast as possible, and to achieve this goal speedrunners utilize implementation flaws to skip as much as possible and perform actions unintended by the developers. The text explains the most desirable bug types and characteristics for speedruns, with examples of existing glitches and the mechanism behind them. We also explore bugs that cannot easily be reproduced by players in real time, as well as the tools and methodologies used to understand the inner workings of bugs. Through this work we show how the discovery and usage of bugs by speedrunners is similar to reverse engineering, explaining the necessary concepts in an accessible manner.

Keywords: speedruns, tool-assisted speedruns, videogames, bugs, glitches, emulators.

Sumário

Capítulo 1

Introdução

O que é um *bug*?

A maioria dos leitores já deve ser familiar com *bugs* em programas de computador. Computadores são tão ubíquos hoje em dia que mesmo aqueles que nunca tiveram contato direto com programação serão eventualmente expostos a algum tipo de *glitch* (termo que usaremos como sinônimo de *bug* ao longo do texto) em uma de nossas numerosas interações com sistemas computadorizados. Conceitualmente, *bugs* representam uma dissonância entre a especificação (formal ou não) do sistema e a implementação realizada pelos desenvolvedores. Às vezes também podem revelar problemas na própria especificação, que pode estar incorreta ou incompleta. Alternativamente, *bugs* podem ser descritos como experimentos com resultados diferentes do esperado segundo o nosso modelo conceitual do programa ou sistema.

Uma vantagem de pensar desse modo é que um experimento naturalmente nos leva a fazer uma série de perguntas diferentes: qual a diferença entre o valor esperado e o resultado? Qual sequência de ações pode ser usada para replicá-lo? Existem outras ações ou condições que também levam a resultados similarmente inesperados? Qual é o estado do sistema durante o experimento, e qual parte dele é responsável pelo desvio do resultado? As respostas para essas (e outras) perguntas são essenciais no processo de aprendizado do verdadeiro comportamento da implementação do sistema.

Outro modo, mais típico, de encarar *bugs* são como erros ou falhas de um sistema. Por exemplo, a taxonomia descrita por Avizienis *et al.* (2004) que separa os diferentes aspectos de um problema de algum sistema em *erro*, *falha* e *causa* a fim de identificar a causa raiz e como o estado inválido do programa é propagado pelo sistema até ser observado pelo usuário final. A grande maioria desses *bugs* são problemas técnicos simples, que geralmente são descobertos, entendidos e corrigidos durante o processo de desenvolvimento, seja por testes manuais ou automatizados. Outros estudos examinam *glitches* a partir de outras perspectivas. Em Bainbridge e Bainbridge (2007) o autor examina *glitches* em relação a cultura *gamer*, como um modo de manipular as regras do jogo, contrastando com a resposta natural que é eliminá-los. Lewis *et al.* (2010) é um estudo que classifica *bugs* encontrados por jogadores em uma taxonomia para ajudar testadores e desenvolvedores a encontrarem *glitches* similares.

À medida que a complexidade de um sistema aumenta, é natural que a complexidade e

sutileza dos *bugs* presentes aumente correspondentemente. Uma maneira simples de lidar com esse inconveniente é encarar o processo de descoberta e mitigação de cada *bug* como uma oportunidade para aprender sobre a implementação existente e como melhorar seu *design*. De fato, o entendimento da causa raiz do defeito e suas repercussões são de grande utilidade para evitar o surgimento de outros *bugs* similares.

Motivação

A analogia com experimentos pode parecer estranha ao dizermos que *bugs* são problemas, erros, falhas ou defeitos, pois não usaríamos esses termos para descrever fenômenos físicos (por exemplo o efeito fotoelétrico) descobertos através de experimentos que contradiziam o modelo físico usado na época. De fato, a motivação dessa monografia não é focar na parte negativa de *bugs* como problemas inesperados que podem ter consequências desastrosas no mundo real (desde programadores demitidos, vazamento de informações pessoais e até pessoas mortas, dependendo do tipo de programa e falha), e sim na parte positiva que diz respeito sobre o que podemos aprender a partir do estudo de *bugs* de *videogames* e quais podem ser as suas utilidades para *speedrunners*. A analogia acima faz mais sentido ao nos limitarmos apenas a jogos eletrônicos, pois muitos dos *bugs* que iremos descrever realmente foram encontrados a partir de experimentos de tentativa e erro realizados dentro do mundo simulado pelo jogo que, consequentemente, revelaram novos comportamentos que provavelmente não eram intencionados pelos desenvolvedores originais.

Speedrunning, a prática de terminar jogos o mais rápido possível sem o uso de dispositivos de trapaça externos[1], por sua vez é um *hobby* competitivo de uma grande comunidade de jogadores que possui uma relação peculiar com *bugs*. O site *Speedrun.com*, um dos mais famosos atualmente, por exemplo, possui mais de 14000 jogos em seu banco de dados, 170000 usuários registrados e por volta de 700000 *speedruns* submetidas ao longo de 4 anos de funcionamento[2].

A posição de *speedrunners* sobre *glitches* em jogos é, sucintamente, que apenas modificações externas no jogo ou sistema são consideradas trapaça. *Bugs* presentes no jogo devido a problemas de implementação, portanto, são técnicas que qualquer pessoa poderia realizar usando apenas um controle. Essa perspectiva levou a comunidade de *speedrunners* a pesquisar a fundo o funcionamento de *bugs*, a fim de utilizarem o conhecimento proveniente desse processo como vantagem competitiva. O estudo de *bugs* é muito similar a *engenharia reversa*, pois os jogadores não possuem acesso ao código fonte original e precisam realizar uma análise de *caixa preta* para descobrir exatamente o funcionamento do código do jogo em questão. O processo de descoberta de *bugs* se torna muito mais fácil ao analisarmos os tipos de *glitches* que aparecem recorrentemente em *speedruns* de jogos diferentes e descobrirmos suas similaridades.

[1] Descreveremos conceitos relevantes da prática em mais detalhes no capítulo 2, porém outros detalhes sobre o hobby e a comunidade podem ser encontrados em Scully-Blaker (2016) ou nos próprios sites usados pelas comunidades.

[2] Informações referentes a Outubro de 2018. Fonte: https://www.speedrun.com/statistics

Objetivo

O objetivo deste trabalho é descrever de maneira clara os principais tipos e características de *bugs* que *speedrunners* procuram em jogos. No caminho, vamos esclarecer as técnicas, métodos e algumas das ferramentas usadas para entendê-los e aplicá-los. Além disso, exibiremos detalhes técnicos sobre alguns exemplos de *bugs* famosos em diversos jogos, ilustrando a maneira que esses conceitos se materializam a partir de exemplos reais.

O restante do texto está organizado da seguinte forma. No capítulo 2, descrevemos a prática de *speedrunners* em mais detalhes e discutimos os principais tipos de *bugs* que esses jogadores buscam em jogos, dando exemplos de casos reais. No capítulo 3, focamos em *speedruns* realizadas com a ajuda de emuladores especializados (emuladores são programas que imitam um sistema, possibilitando rodar jogos feitos para tal em outras arquiteturas) e, portanto, sobre tipos de *bugs* e técnicas que não podem ser facilmente realizadas por pessoas em tempo real. No capítulo 4, concluímos mostrando alguns recursos adicionais para encontrar outras *speedruns* parecidas com as descritas ao longo do trabalho.

Contribuições

Apesar da comunidade fazer um bom trabalho documentando as informações necessárias para que novos membros consigam aprender a executar *bugs*, isso fica espalhado informalmente pela Internet. Por isso, muitos dos detalhes técnicos não estão facilmente acessíveis para uma população leiga em computação ou sem conhecimento técnico da implementação do jogo.

Nesse contexto, esse trabalho pode ser visto como uma exposição de conceitos e exemplos que conectam *bugs* e *speedruns*. Aqui, temos a intenção de tocar esses temas de modo a torná-los palatáveis para públicos distintos. Por isso, é de se imaginar que haja variações no proveito que cada leitor terá do texto.

Por exemplo, o leitor que é familiar com computação pode usar essa leitura para averiguar como algumas classes de *bugs* que o acompanham no dia a dia também aparecem em *videogames*. Por outro lado, o leitor familiar com *speedrunning* pode consultar esse material para ver como os *glitches* funcionam no seu jogo favorito. Adicionalmente, outra contribuição que segue da descrição de *bugs* existentes é despertar interesse por *speedruns* em leitores que não são familiares com esses conceitos.

Capítulo 2

Speedruns em tempo real

O objetivo desse capítulo é enumerar algumas categorias de *bugs* importantes para *speedruns* em tempo real. A primeira seção descreve os termos e conceitos relacionados ao assunto. A seguir descrevemos características desejáveis para que *bugs* possam ser utilizados em *speedruns* efetivamente. A terceira seção enumera as categorias de causa raiz que vamos considerar. Cada descrição de causa raiz é seguida por um ou mais exemplos de *bugs* relacionados. Finalmente resumimos a relação entre os *bugs* descritos no capítulo.

2.1 Introdução

Muitos jogos medem o tempo gasto para terminar suas fases ou o tempo total jogado. Às vezes isso é usado para recompensar (com um final ou evento especial) jogadores que terminam o jogo abaixo de um certo tempo ou, também, como incentivo para que as pessoas se desafiem e tentem descobrir qual é o menor tempo possível. Uma evolução natural disso é a criação de *speedrunning*, a prática de terminar um jogo (ou uma fase de um jogo) o mais rápido possível, não importando se o jogo incentiva essa prática ou não. Existe uma grande comunidade de *speedrunners online*, que surgiram por diversas razões e ficam concentradas em diversos *sites* como plataformas de vídeo ou *livestreaming* nas quais os jogadores exibem suas tentativas e recordes pessoais. Também organizam eventos beneficentes que atraem milhares de espectadores nessas plataformas, como por exemplo *GDQ*[1] e *ESA*[2]. Por sua vez, cada comunidade possui um foco distinto, algumas delas mais abrangentes e generalistas e outras focadas em jogos específicos[3].

A fim de direcionar o leitor para outras perspectivas acadêmicas sobre *speedruns*, vamos recomendar algumas publicações. Em Scully-Blaker (2016), o autor analisa em detalhes as origens das comunidades de *speedrunners* e as regras que as governam. O autor de Hilburn (2017) examina *speedrunners* como um fenômeno cultural, ou seja uma prática que transforma a experiência dos jogadores. Similarmente, em Franklin (2009) o autor analisa a narrativa

[1] *Games Done Quick*: https://gamesdonequick.com/

[2] *European Speedrunner Assembly*: https://esamarathon.com/

[3] Atualmente, https://speedrun.com/ é um dos sites de *speedruns* em geral com maior número de jogos e *runs* submetidas em seus banco de dados, mas existem outros como *SpeedRunsLive* (http://www.speedrunslive.com), no qual jogadores competem entre si em corridas ao vivo, e comunidades específicas a certos jogos, por exemplo SourceRuns (https://sourceruns.org), focada em jogos com a *engine Source*, da Valve.

social por volta tanto de *speedruns* como de *TASes*. Em Brewer (2017), o autor discute o desenvolvimento de regras contra trapaças em comunidades de *speedrunners*, comparando-as com outras comunidades de jogadores. Por sua vez, Scully-Blaker (2014) interpreta a relação entre *speedrunning* e seu impacto em jogos como espaços narrativos. O autor de Parker (2008) considera as implicações de regras adicionais impostas pelos próprios *speedrunners* e discute a relação entre jogadores e os sistemas de regras implícitos de *videogames*. Finalmente, em Lafond (2018) o autor formaliza matematicamente os aspectos de uma *speedrun* e analisa a complexidade computacional de resolver os problemas de otimização resultantes.

Terminologia básica

O termo *speedrun* origina do fato que, em inglês, *run* é uma palavra usada para se referir a uma "sessão" de jogatina, geralmente com algum objetivo. Uma *score run*, por exemplo, consiste em jogar um jogo com o objetivo de alcançar a maior pontuação possível. Em geral os *speedrunners* submetem um vídeo para comprovar que alcançaram um certo tempo, e existem comunidades *online* que possuem placares para classificar essas submissões e os melhores tempos conhecidos. Essas submissões são analisadas por membros da comunidade para detectar possíveis trapaças, e eventualmente o vídeo fica disponível em plataformas como *YouTube* ou *Twitch.tv*.

O termo *tempo real* se refere ao fato de que as *speedruns* são realizadas do começo ao fim de uma única vez e o tempo medido é o tempo real (em oposição ao tempo medido pelo jogo, caso exista) que passou. O termo usado por muitos para descrever isso é *Real-Time Attack* (*RTA*), originário da comunidade japonesa de *speedrunners*. Hoje em dia é a forma mais popular de *speedruns*, pois é um conjunto de regras que torna mais difícil trapacear. Neste capítulo, usaremos o termo *speedrun* como sinônimo de *RTA*. No capítulo 3 falaremos sobre uma modalidade de *speedrun* que não possui essas restrições.

Uma *categoria* de *speedrun* é um conjunto de regras que devem ser seguidas e um ou mais objetivos dentro do jogo que devem ser alcançados, todos definidos pela comunidade de jogadores. Como o objetivo é fomentar a competição dentro das comunidades, é necessário que as definições de cada categoria sejam claras e objetivas. Por exemplo, o tipo de categoria mais comum e simples, que consiste apenas em terminar o jogo do começo ao fim sem regras ou limitações adicionais, chama-se *any%*, onde a porcentagem é usada para significar o nível de completude do jogo. Os *speedrunners* precisam concordar em aspectos como momentos de início e término da marcação de tempo, técnicas ou *glitches* que são banidos (ou permitidos), e condições necessárias dentro do jogo que precisam ser atingidas.

Outro exemplo de categoria comum é 100% na qual é necessário completar todo o conteúdo opcional do jogo (por exemplo obter todos os itens e vencer todos os níveis). Dependendo do jogo e como os itens obtidos pelo jogador são representados pelo jogo, podem existir diversas definições diferentes mas igualmente aceitáveis de "conteúdo opcional", por exemplo no caso de eventos opcionais que acontecem uma única vez no jogo mas, além disso, não há nenhuma outra indicação por parte do jogo de que esse evento ocorreu, apesar do jogo internamente

possuir essa informação. Existem argumentos convincentes tanto para considerar esse tipo de evento quanto para desconsiderá-los, e portanto se um número suficiente de pessoas demonstrar interesse, categorias diferentes mas similares podem coexistir.

Uso de *bugs* em *speedruns*

Similarmente, se existir interesse por parte dos jogadores, *glitches* podem ser banidos de certas categorias, em geral por razões específicas a cada jogo ou *bug*, por exemplo em casos em que um *bug* por si só pode ser usado para trivializar demais a *speedrun*. Desse modo, o uso de *bugs* ou *glitches* em *speedruns* depende do que é aceitável para a categoria em questão. Em geral, se as regras da categoria não excluem o uso de *bugs*, é aceitável o seu uso em *speedruns* na maioria das comunidades. A lógica por trás dessa decisão está no fato de que, exceto os desenvolvedores originais do jogo, não é possível diferenciar com exatidão o que é um "*bug*" do jogo e o que são apenas particularidades do funcionamento do jogo, ou "truques" inteligentes, tornando a criação de qualquer lista de "*bugs* banidos" em uma discussão com argumentos baseados, fundamentalmente, em suposições e opiniões. Isso não significa que trapaça é permitida, apenas que *bugs* presentes no jogo não são considerados como tal, desde que seja possível recriá-los usando apenas sistemas e controles originais, sem modificações. Exemplos de práticas que seriam consideradas como trapaça em qualquer categoria e comunidade de *speedrun* são edição do vídeo submetido, como cortar seções ou modificar a sua velocidade e o uso de dispositivos externos ou modificações não oficiais no sistema para criar uma vantagem, como *Gameshark* ou *Action Replay*[4].

Ainda mais, podemos dizer que o uso de *bugs* em *speedruns* aumenta o número de ações possíveis que o jogador pode realizar a cada momento, aumentando a complexidade da *speedrun*. Como alguns *bugs* são difíceis de reproduzir, principalmente sob a pressão de tempo, a dificuldade de terminar uma *speedrun* que utiliza *bugs* também é maior. Para muitos *speedrunners*, essas características são positivas e uma das principais razões que tornam *speedruns* mais interessantes para os espectadores.

Desse modo, fica claro o porquê *speedrunners* possuem um grande incentivo para encontrarem novos *bugs* em jogos que ajudem a diminuir o tempo necessário para terminá-los. Por essa razão, jogos populares possuem uma grande quantidade de recursos disponíveis sobre os *bugs* conhecidos e suas aplicações em *speedruns*, a fim de ensinar novos membros da comunidade a realizarem as manobras necessárias para conseguirem um tempo competitivo com os melhores *speedrunners*. Esse conhecimento comunitário é compartilhado entre outras comunidades que também possuem interesse em detalhes de implementação de jogos, por exemplo pessoas que fazem engenharia reversa de jogos[5] e *modders*[6].

É natural se perguntar como esses *bugs* são descobertos em primeiro lugar. Aqui a

[4]Dispositivos externos que modificam a memória do sistema enquanto o jogo roda.

[5]Um exemplo é o site *The Cutting Room Floor*, https://tcrf.net/, que possui informações obtidas por meio de engenharia reversa de vários jogos.

[6]Pessoas que realizam modificações no código de um jogo para adicionar funcionalidades ou até criar novos jogos na mesma *engine*.

nossa analogia com experimentos científicos é apta, pois um grande número de *glitches* são encontrados completamente ao acaso e, após serem postados em fóruns ou similares, os membros da comunidade (não necessariamente *speedrunners*) se mobilizam para experimentar com o *bug*, com a finalidade de entendê-lo, torná-lo consistente e considerar possíveis aplicações em *speedruns*. Claro, esse não é o único método: jogadores que possuem experiência sobre os detalhes de implementação de jogos (obtidos a partir de engenharia reversa) podem teorizar novos *bugs* ou maneiras de combinar e utilizar *bugs* com um específico propósito em mente. Essencialmente, o processo de descoberta de novos *glitches* envolve bastante criatividade e uma parcela de sorte.

2.2 Características desejáveis

O termo usado para descrever a sequência de ações para completar uma *speedrun* é *rota*. Apesar de ser um *hobby* individual (pois a maioria das *speedruns* são de jogos de apenas um jogador), o processo de criação e otimização de rotas é algo realizado em conjunto, com a colaboração de muitos membros das comunidades de cada jogo. Quando um novo *bug* é descoberto e incorporado na rota de alguma *speedrun* observamos uma queda no tempo do recorde mundial da categoria correspondente, como podemos ver por exemplo na figura 2.1 que mostra o impacto da descoberta de novos *bugs* no recorde mundial do jogo *The Legend of Zelda: Ocarina of Time* no período de 2012 a 2018. Vários fatores influenciam a aplicabilidade de um *bug* em uma *speedrun*, como os pré-requisitos para reproduzir a falha e a diferença de tempo com uma nova rota que utiliza esse *bug* da maneira mais eficiente. De fato, um *bug* só é útil se a rota de uma *speedrun* pode ser modificada para incorporá-lo.

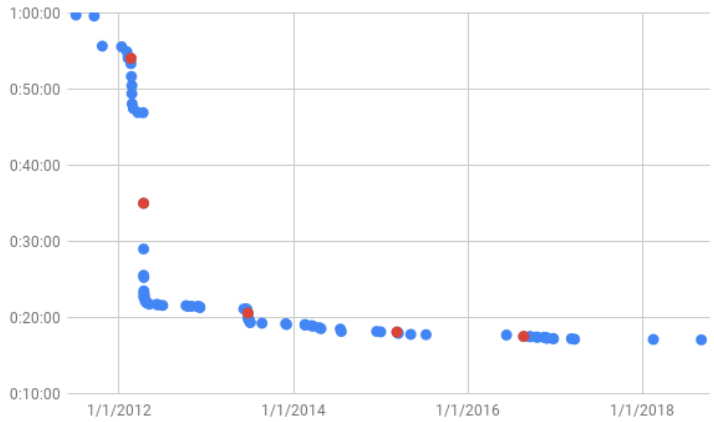

Figura 2.1: *Gráfico que mostra o recorde mundial para o jogo* The Legend of Zelda: Ocarina of Time*, na categoria* any% *nos últimos* 6 *anos. Os pontos vermelhos representam o primeiro recorde a utilizar uma nova rota que incorpora um* bug *novo. Fonte: Jbop (2014).*

Existem algumas condições para que um *bug* consiga ter um impacto grande em alguma *speedrun*. Primeiramente deve ser possível replicá-lo sem muitas dificuldades de uma maneira consistente. Por exemplo, se a execução de um *bug* depende de uma conjuntura muito específica, fica difícil incorporá-lo a uma possível rota. Mesmo assim, se isso representar uma

possibilidade de ganhar muito tempo, por exemplo pulando uma grande parte do jogo, é possível utilizar o *bug* para criar uma rota alternativa e mais arriscada. Em certos jogos, técnicas foram desenvolvidas para facilitar a execução de *bugs* precisos usando sequências de movimentos fáceis de serem realizados que permitem o jogador a obter os valores necessários de posição, ângulo e velocidade. A existência dessas técnicas claramente depende dos detalhes de implementação do jogo, mas são extremamente úteis para tornarem *bugs* consistentes o suficiente para serem realizados no maior número de situações possível.

Outra condição desejável é que não haja muitos pré-requisitos para o *bug* acontecer. Se existir um *bug* relacionado ao comportamento de um item opcional, pode ser que o tempo para obter o item seja maior do que o tempo ganho ao utilizar o *bug*. Essas são as principais condições para permitirem que um *glitch* seja incorporado em uma *speedrun*.

Por outro lado, as características desejáveis do efeito que o *bug* causa no jogo podem ser vistos como modos alternativos de minimizar o tempo gasto pela *speedrun*. Um exemplo de como isso pode acontecer é caso um *glitch* permita que o jogador pule seções do jogo, como níveis ou itens necessários para progredir. Por sua vez, isso pode ser alcançado de diversas maneiras, e uma delas é utilizar outros *bugs* para sair dos limites do nível, por exemplo abusando problemas na verificação de colisões que permitem o personagem a atravessar paredes ou barreiras sólidas.

Outros aspectos benéficos seriam técnicas para movimentar o personagem de maneira mais rápida do que o normal, ou causar mais dano do que o normal para inimigos. De certo modo, qualquer efeito que facilite completar o jogo é desejável. Portanto, os esforços dos caçadores de *glitches* se focam em encontrar esses tipos de aplicações para os *bugs* que descobrem.

2.3 Categorias de *bugs*

Vamos agora descrever em mais detalhe alguns *bugs* usados em *speedruns*, separados por causas raiz. Consideraremos causas raiz abstratas, a fim de ressaltar similaridades entre os exemplos apresentados. Primeiramente descrevemos em alto nível as possibilidades que o *bug* proporciona a *speedrunners*. Em seguida detalharemos exemplos de jogos que possuem *glitches* do tipo.

2.3.1 Tratamento incorreto ou inexistente de casos de borda

Pode-se dizer que todo *bug*, por não ser intencionalmente adicionado ao código, corresponde a alguma situação que não é considerada corretamente pela implementação. Porém nessa seção vamos falar sobre alguns casos aonde só existe uma falha devido a uma omissão de um pedaço simples de código, que faz com que existam casos de borda tratados incorretamente pelo jogo. Em geral *bugs* desse tipo representam situações que não foram consideradas ou encontradas durante o desenvolvimento. *Bugs* também podem ser combinados para criar situações excepcionais que não são tratadas corretamente.

Exemplo 1: Invulnerabilidade de inimigos acaba prematuramente (*Mega Man 1*)

Diferentemente de muitos jogos da época, *Mega Man 1* (NES) pode ser pausado de duas maneiras. O jogador pode usar o botão *START* para escolher o tipo de arma por meio de um menu, pausando o jogo enquanto um menu aparece na tela. Devido a limitações do console e a complexidade gráfica do menu o jogo não desenha *sprites* (do jogador, inimigos e projéteis) ao mostrar o menu e o jogo reutiliza a memória gráfica ocupada pelos projéteis para os elementos gráficos do menu. Provavelmente para simplificar o código, esse menu só pode ser mostrado quando não há projéteis do jogador na tela, já que o console não possui memória gráfica suficiente para mostrar projéteis de várias armas diferentes ao mesmo tempo. Outro método pelo qual o jogador pode pausar o jogo que não possui essa última limitação e pode ser usado a qualquer momento é o botão *SELECT*, que simplesmente pausa a simulação do jogo até o botão ser pressionado novamente, mas continua desenhando todos os *sprites*. Esse método provavelmente foi adicionado devido à limitação do botão *START*, para que o jogador conseguisse pausar o jogo a qualquer momento[7].

Para evitar que o jogador pudesse apenas apertar o botão de tiro o mais rápido possível e destruir os inimigos rapidamente, existe um período no qual, após tomar dano pela primeira vez, os inimigos não tomam dano (muito comum em jogos). Estranhamente, o tempo restante para o período de invulnerabilidade terminar é decrementado dentro da rotina responsável por desenhar sprites na tela, como pode ser visto na função *DrawObject* em Yliluoma (2013), uma listagem do código do jogo com comentários. Isso significa que, como ao pausar o jogo por meio do botão *SELECT* os sprites de inimigos ainda são renderizados na tela, esse período é decrementado mesmo com o jogo pausado, pois o código responsável não verifica isso. *Speedrunners* podem abusar isso por meio de armas com projéteis que não desaparecem imediatamente após sua colisão com um inimigo, para pausar o jogo após o período de invulnerabilidade do inimigo começar até ele terminar, repetindo o processo e causando dano múltiplas vezes por meio de um único projétil.

Esse descuido foi consertado nos próximos jogos da série (mesmo os que são baseados no mesmo código fonte) e está presente apenas no primeiro jogo. Nesse exemplo, o caso de borda não considerado é simplesmente o fato que o sistema responsável por controlar a invulnerabilidade com inimigos não verifica se o jogo está pausado.

Exemplo 2: *Overflow* de inteiros durante o combate (*Chrono Trigger*)

No jogo *Chrono Trigger* (SNES) existe um equipamento chamado *Green Dream*, que revive um personagem uma vez por luta automaticamente. Quando esse efeito é ativado, internamente o jogo considera esse personagem como "morto" temporariamente apesar do personagem acabar de reviver, devido a detalhes de implementação, e como consequência ele não consegue usar itens em si mesmo por uma rodada, já que o código exclui personagens mortos da lista de alvos. Se ao mesmo tempo todos os outros personagens do time estiverem

[7] O manual do jogo não descreve o comportamento dos botões *START* e *SELECT*, então isso é apenas conjectura.

realmente mortos, o jogo permite que itens que normalmente só podem ser usados em aliados possam ser usados em inimigos, pois o jogo acredita que não existem alvos válidos no time do jogador, e automaticamente seleciona os inimigos ao usar um item.

Um item que normalmente não pode ser usado em inimigos, assim como outros itens de restauração do jogo, é o *Elixir*, que restaura os pontos de vida (*HP*) e magia (*MP*) de um personagem completamente. O jogo implementa a funcionalidade desse item somando aos valores atuais o valor total de vida e mágica do personagem. Como inteiros no SNES possuem 16 bits, os valores representáveis são de -32.768 a 32.767. Usando o *bug* descrito no parágrafo anterior, é possível explorar o fato de que se a soma do *HP* atual mais o *HP* máximo do inimigo for maior ou igual a 32.768 a vida do inimigo é tratada como um número negativo e o inimigo morre imediatamente. Os únicos inimigos do jogo que satisfazem essas condições são os dois últimos chefes do jogo: *Inner Lavos* com 20.000 *HP* e *Lavos Core* com 30.000 *HP*.

Esse é um exemplo de dois *bugs* que só são úteis para uma *speedrun* ao serem usados em conjunto. Se fosse possível apenas utilizar itens em inimigos em certas circunstâncias isso não seria nada útil para uma *speedrun*, pois itens que só podem ser usados em aliados são itens benéficos por exemplo para restaurar vida ou mágica. Analogamente, os programadores teoricamente não precisavam se preocupar com a possibilidade de ocorrer *overflow* nas variáveis usadas para guardar a vida das entidades do jogo, pois as habilidades que os inimigos usam em si mesmos para restaurar vida não adicionam vida suficiente para alcançar o valor máximo. Similarmente, a vida dos personagens do jogador não consegue chegar perto suficiente de 32.768 para isso afetar seus personagens. Porém a combinação de ambos permite que as lutas finais do jogo, com os inimigos teoricamente mais difíceis do jogo, sejam completamente ignoradas e terminem quase que instantaneamente. Como esses chefes possuem tanta vida, realizar esse *bug* é significantemente mais rápido do que lutar contra eles normalmente. Uma *speedrun* que utiliza esse *bug*, apesar de não ser em tempo real, pode ser encontrada em keylie (2014).

Exemplo 3: *Accelerated Back Hopping* (*Engine Source* (2009))

Bunny hopping é uma técnica popular para ganhar altas velocidades por meio de pulos e movimentação diagonal no ar que surgiu inicialmente na *engine* de *Quake* em 1998 e que é aplicável em alguns jogos baseados em seu código. Uma explicação do funcionamento desse *bug* em detalhes pode ser encontrada em Biagioli (2015).

Ao atualizar a *engine Source* por volta de 2009, os desenvolvedores consertaram *bunny hopping* de um modo simples: quando a magnitude do vetor de velocidade do jogador excede o limite do jogo, a *engine* tenta diminuir a velocidade de volta para o valor limite na próxima vez que o jogador pular. Para descobrir a direção de movimento do jogador o jogo verifica se as teclas para se movimentar para frente ou para trás estão apertadas e, caso nenhuma esteja apertada, assume que o jogador não modificou sua direção. A figura 2.2 mostra como esse método permite o jogador a acumular muito mais velocidade do que os outros métodos

existentes que os desenvolvedores tentaram consertar.

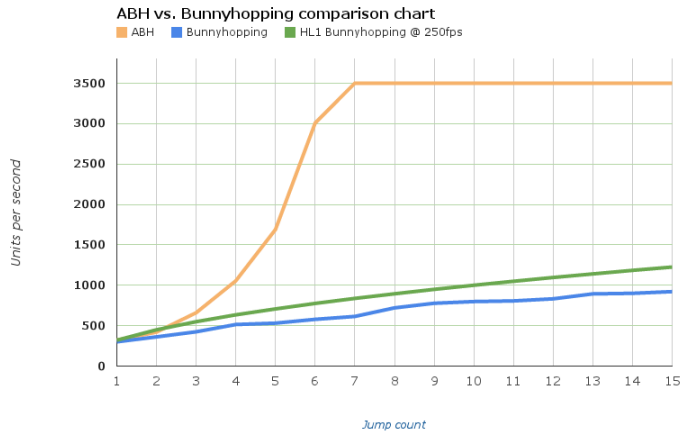

Figura 2.2: *Gráfico comparando a velocidade obtida com* ABH *e* bunny hopping*. Fonte: SourceRuns (2013).*

O *bug* é realizado da seguinte maneira: o jogador pula para frente, ultrapassando o limite de velocidade do jogo. No meio do pulo, o jogador vira de costas por meio de movimentos do mouse e, sem apertar as teclas de movimentação para frente ou para trás (W ou S), pula novamente ao encostar no chão. Quando o jogo tenta diminuir a velocidade do jogador ele ainda assume que o movimento do personagem é para frente e portanto acelera o jogador na direção contrária, mas essa é a direção do pulo original e a portanto a velocidade do jogador acumula repetidamente com cada pulo. É um *bug* relativamente simples, porém difícil de controlar pois o jogador não consegue ver para onde está se movimentando e se demorar demais para pular ao encostar no chão sua velocidade diminui drasticamente. As altas velocidades possíveis com essa técnica permitem pular vários obstáculos do jogo, pois ao colidir com superfícies inclinadas em altas velocidades o personagem é lançado verticalmente, e com a altura resultante é simples atravessar as barreiras e paredes existentes.

Exemplo 4: *Backwards Long Jump* (*Super Mario 64*)

No jogo *Super Mario 64* (N64) existem vários tipos diferentes de pulos, dentre os quais, o *long jump* se destaca por ser um dos que mais aumentam a velocidade do jogador devido ao seu uso pretendido de permitir que o jogador atravesse abismos e longas distâncias rapidamente. Como o jogo representa a velocidade por duas componentes em ponto flutuante, onde cada uma corresponde à magnitude da velocidade ao longo de eixos relativos ao Mario, o jogo atualiza as componentes separadamente no momento do pulo. Se examinarmos o código que modifica o valor da componente horizontal[8] ao realizar um *long jump*, temos o seguinte comportamento, descrito em pseudocódigo:

$velocidade\_horizontal \leftarrow 1.5 * (velocidade\_horizontal)$

[8]Valores positivos e negativos da componente horizontal correspondem ao Mario ir para frente e para trás respectivamente. A medida que a direção do jogador muda, sua velocidade ao longo desse eixo continua a mesma.

if $velocidade\_horizontal > 48$ **then**
  $velocidade\_horizontal \leftarrow 48$
end if

O valor de 48 usado pelos programadores como limitante superior provavelmente vem do fato de que a velocidade horizontal máxima do Mario no chão ao segurar o *joystick* em uma única direção oscila entre 31 e 32, resultando em um valor próximo de $48 = 32 * 1.5$ para a velocidade inicial do pulo[9]. Como consequência disso, *long jumps* encadeados não acarretam em um aumento de velocidade dentro das condições esperadas pelos programadores, e a velocidade oscila por volta de 48 [10] em circunstâncias normais.

Os programadores responsáveis por esse trecho do código esqueceram um aspecto muito importante: não existe garantia de que o valor da velocidade horizontal é positivo! Em outras palavras, quando o Mario pula para trás enquanto tem velocidade negativa, existe a possibilidade de que a magnitude da velocidade aumente sem limites de maneira exponencial! Porém isso não acontece se o jogador realizar apenas *long jumps* enquanto segura o *joystick* para trás em chão plano, pois o código acima que aumenta a velocidade só executa uma vez, no início do pulo, e enquanto o Mario está no ar a magnitude de sua velocidade cai de forma que não chega a ser menor do que -15.[11]

Em certas condições, que dependem da geometria da fase, existe a possibilidade de, imediatamente após realizar o pulo, acontecer uma colisão entre o Mario e um chão, o que permite o jogador a repetir o processo praticamente imediatamente (i.e. poucos *frames*) após o código acima aumentar sua velocidade horizontal. Isso acontece quando a próxima posição que o jogo coloca o Mario está dentro do volume de colisão de um chão, e não acontece em geral pois todos os pulos do Mario aumentam a componente vertical de sua velocidade suficientemente ao mesmo tempo que modificam o estado do Mario para que ele seja considerado "no ar". Ao colidir com a geometria de um chão, o jogo muda o estado do Mario para que ele esteja "no chão", muda a posição no eixo Y do Mario para a posição do chão e muda a componente vertical de sua velocidade para 0. Como o Mario consegue pular apenas quando seu estado é "no chão", isso é suficiente para que o jogador consiga repetir o pulo imediatamente, fazendo com que a velocidade do Mario aumente exponencialmente.

Algumas das situações onde isso é possível são em certas escadas e rampas, elevadores (ou qualquer plataforma que se mova para cima suficientemente rápido), lugares com teto baixo e bordas de certas plataformas baixas (como degraus). Os *inputs* (botões que precisam ser apertados no controle) exatos que o jogador precisa realizar dependem da situação, mas todos tem esse mesmo mecanismo básico por trás. Em um cenário ótimo, no qual o jogador realiza o maior número de pulos possível (o que significa apertar o botão de pulo 30 vezes

[9] Na verdade a velocidade no ar aumenta um pouco devido a outras partes do código que executam depois, por exemplo quando o jogo modifica a velocidade do Mario dependendo da direção do *joystick*.

[10] A velocidade do Mario diminui gradualmente após ele encostar no chão, então a velocidade inicial dele no momento de um pulo após outro pode ser maior que 48.

[11] Por essa razão, existe a possibilidade de que os programadores estavam cientes desse problema mas achavam que isso não era abusável durante o jogo.

por segundo), esse *bug* pode levar a velocidades tão altas que fica praticamente impossível controlar o personagem, principalmente se for necessário modificar a direção do movimento.

A consequência direta desse *bug* é alcançar velocidades com altas magnitudes, mas uma consequência muito útil disso vem do fato de que o código responsável por verificar a colisão do personagem com elementos do mundo (chãos, paredes, tetos e outros objetos) não foi escrito com esses valores extremos em mente. Esse código verifica, a cada *frame* de processamento do jogo, se a nova posição do jogador, calculada a partir do seu valor de velocidade atual, é válida, no sentido de que não houveram colisões com obstáculos durante a movimentação. Para ter certeza disso, o código verifica 4 posições ao longo da trajetória do movimento do Mario e, para cada uma delas, realiza um teste de colisão entre o volume que representa a colisão do Mario com os volumes de colisão de outros objetos do nível. Caso positivo uma nova posição final e velocidade do personagem é calculada e utilizada, indicando que houve uma colisão.

Portanto, é fácil ver o que ocorre quando a magnitude da velocidade do personagem é muito grande: a posição final calculada é muito distante da posição atual, de modo que o volume de colisão responsável por impedir a movimentação do personagem se encontre totalmente entre a posição inicial e a primeira posição intermediária verificada pelo código. Isso significa que a verificação é falha e desse modo o jogador pode "atravessar" paredes e quaisquer outros tipos de obstáculos sob essas circunstâncias, facilmente alcançadas ao realizar um *BLJ*, tornando possível escapar os limites do nível.

Esse *bug* é muito antigo e foi originalmente encontrado e publicado por volta de Novembro do ano 2000, em uma edição da revista *Club Nintendo* na seção de truques submetidos por leitores como um modo de burlar o número de estrelas necessário para alcançar o último chefe do jogo. Ao utilizá-lo para ultrapassar certos obstáculos que impedem o progresso do jogador, esse *bug* torna possível terminar o jogo sem coletar nenhuma estrela[12] e, por pular uma quantidade consideravelmente grande do conteúdo do jogo de uma maneira desinteressante, a maioria das categorias de *speedruns* do *Super Mario 64* banem o uso desse *glitch*.

2.3.2 Escapar dos limites do nível

Essa categoria, também chamada de *Out of Bounds* (*OoB*) diz respeito a *bugs* que permitem acessar posições incorretas, em geral fora dos limites da fase ou atravessando paredes de maneiras que não deveriam ser possíveis. Exatamente o que acontece quando o jogador ocupa coordenadas fora dos limites da fase depende do jogo, pois em geral essa situação nunca acontece durante jogatina normal. Um exemplo são erros de colisão, que permitem com que o personagem seja "empurrado" através de um obstáculo como resultado de uma colisão sob condições específicas.

[12]Como por exemplo essa *speedrun* por Akira em 6:44.2: http://nico.ms/sm26882407.

Exemplo 5: *Streaming* de dados não acompanha velocidade do jogador (*Pokémon*)

Nos jogos da quarta geração de *Pokémon* (*Diamond* e *Pearl*, Nintendo DS), os mapas do jogo são implementados por meio de seções quadradas de tamanho 32 por 32 de modo que, a cada momento, apenas 4 seções estão carregadas na memória do jogo e visíveis. A medida que o jogador se movimenta no mapa, a câmera do jogo não mostra mais do que 4 seções por vez e o jogo transparentemente carrega as informações de novos setores adjacentes antes que eles sejam visíveis ao mesmo tempo que descarrega os dados de setores que não são mais visíveis. Essa técnica é chamada de *streaming* de dados e tem como objetivo diminuir os tempos de *loading* de uma maneira transparente e assíncrona.

Cada uma das seções é divida em 4 subseções e, ao transitar de uma subseção para outra, o jogo muda quais dos 8 setores adjacentes são carregados. Ao estar no canto superior esquerdo de uma região, por exemplo, o jogo mantêm na memória a região atual e as 3 regiões adjacentes ao norte, oeste e noroeste do jogador, criando assim a ilusão de um único mapa contínuo. Ao usar a bicicleta, um item obtido durante o jogo, é possível se movimentar rápido o suficiente para que esse sistema funcione incorretamente, resultando em seções carregadas em posições incorretas (por exemplo uma seção que normalmente fica ao norte carregada na posição oeste do jogador) ou carregadas parcialmente, por exemplo sem gráficos ou *NPCs*, como podemos ver na figura 2.3. Se o jogador continuar andando em uma direção sem realizar esse *bug* novamente, o jogo carrega as próximas seções corretamente e similarmente, se o jogador abrir certas páginas do menu como a *Pokédex* o jogo recarrega as seções corretamente quando o menu é fechado.

Figura 2.3: *A esquerda vemos uma seção de mapa que não foi carregada (também chamada de "*Void*"), e a direita uma seção carregada na posição incorreta. Fonte: Bulbapedia (2017).*

Várias técnicas diferentes foram criadas para realizar esse processo consistentemente, que envolvem andar em uma sequência específica de passos aonde cada passo tomado pelo jogador força o jogo a carregar novas seções. Esse *bug* pode ser usado para passar por áreas que normalmente bloqueiam o jogador, sobrescrevendo essas áreas com informação de uma seções adjacentes sob as quais o jogador consegue passar. Outra possibilidade é carregar uma seção na qual o jogador consegue andar, se movimentar até uma posição específica e recarregar a seção atual por meio do menu, o que permite o personagem a ocupar posições normalmente impossíveis no mapa.

Outra variação dessa técnica permite que o jogador entre em um prédio a partir de uma direção normalmente impossível, como dos lados ou por trás da porta, o que faz com que o jogo carregue o mapa do interior do prédio porém coloque a posição do jogador *OoB*, fora das paredes do mapa interno, pois a posição inicial que o jogo coloca o jogador ao entrar em prédios é definida a partir da direção usada ao entrar na porta. Ao se movimentar nessa área *OoB* (chamada de *void*), o jogo se confunde e acaba carregando informações (como conexões de mapas e outros *triggers*) de outras áreas aleatórias do jogo. Dependendo do caminho realizado pelo jogador, do mapa inicial, e do número de passos, é possível manipular isso para carregar conexões a mapas específicos. Isso é usado na *speedrun any%* desses jogos para ativar o final do jogo prematuramente, em aproximadamente um quarto do tempo que categorias sem *glitches* demoram[13].

2.3.3 Sobreposição ou interrupção de eventos

De uma maneira superficial, isso acontece quando dois ou mais eventos distintos são ativados ao mesmo tempo (ou temporalmente muito próximos), levando a algum tipo de confusão ou valor inválido no estado do jogo. Em geral, eventos em um jogo são tratados por meio de um ou mais trecho(s) de código que executam cada *frame* (o nome dado para cada quadro renderizado pelo jogo na tela) em que o evento está ativo, apesar de existirem outros tipos de implementação. A princípio, os eventos que acontecem durante um jogo (por exemplo colisões, tomar dano, atirar, carregar um novo nível, etc.) podem ser compostos em qualquer ordem de maneira independente, e o jogo é responsável por prevenir a geração de novos eventos em estados inválidos (por exemplo, o jogador não poder pular novamente enquanto seu personagem está no ar). Esse tipo de *bug* ocorre, portanto, quando essa invariante é violada e o jogo executa um evento em um momento em que não deveria ser possível executá-lo normalmente.

Problemas desse tipo podem existir mesmo se a implementação do jogo não possui uma definição explícita de "evento", e em geral ocorrem por meio de outros *bugs* que levam à realização de eventos quando normalmente não é permitido. Muitas vezes a janela de tempo na qual o jogador pode interromper um evento indevidamente dessa maneira é limitada a um ou dois *frames* do jogo, dependendo as características dos trechos de código responsáveis, o que torna mais difícil encontrar e reproduzir *bugs* desse tipo.

Dependendo do tipo da confusão, podem acontecer uma variedade diferentes de efeitos benéficos para *speedruns*. Por exemplo, um evento muito longo pode ser interrompido prematuramente com um evento mais curto. Esses *bugs* são difíceis de serem encontrados durante a fase de desenvolvimento pois muitas vezes o intervalo de tempo no qual o *bug* pode ocorrer é muito pequeno, na ordem de poucos *frames*, e nem sempre é claro a partir do código do programa quais eventos podem ser interrompidos.

[13]Para comparação, atualmente, o melhor tempo *glitchless* é 3 horas e 44 minutos por Werster, enquanto o melhor tempo *any%* é 57 minutos e 21 segundos, e esse *bug* é responsável pela maioria da diferença de tempo.

Exemplo 6: Cancelamento ou adiamento de eventos (*The Wind Waker*)

No *The Legend of Zelda: The Wind Waker* (Nintendo GameCube) várias ações como falar com *NPCs* (personagens controlados pelo computador), abrir baús e usar certos itens são implementadas por meio de eventos nos quais, enquanto estão em execução, o jogador não consegue se movimentar. Quando um desses eventos termina ou é cancelado prematuramente, o jogo faz isso por meio de uma *flag* (ou seja, uma variável booleana) que interrompe o próximo evento a ser executado, removendo a *flag* após o evento ser interrompido. Certos itens, como o *Wind Waker* (o instrumento usado pelo personagem do jogo), são implementados por meio de eventos desse tipo e ao cancelar seu uso, por exemplo apertando B, o jogo utiliza essa mesma *flag*, que é genérica para vários tipos de evento.

Esse mecanismo é explorado por meio de um truque chamado de *storage*, no qual o jogador cancela um evento (por exemplo ao guardar o *Wind Waker*, um item do jogo) ao mesmo tempo que o código tenta cancelar o evento devido a alguma outra razão externa. O método mais simples de realizar isso é subir em alguma plataforma na qual paredes empurram o jogador e o fazem cair imediatamente após alguns *frames*, usar o *Wind Waker* antes de cair e ao encostar novamente no chão, o jogo interrompe o uso do item automaticamente. Existe uma janela de 3 *frames* após encostar no chão nas quais o jogador pode cancelar o uso do item normalmente enquanto o jogo tenta cancelar o evento. Em outras palavras o jogo acaba modificando a *flag* de interrupção de eventos duas vezes e, como o evento original só é interrompido uma vez, o valor da *flag* continua verdadeiro.

A consequência disse é que o próximo evento a ser ativado será imediatamente cancelado e o jogador volta a ter certas características do seu estado "normal", em particular ele consegue se movimentar e utilizar itens. Ao ativar algum outro evento, o evento que foi interrompido volta a executar (por isso o nome do *bug*, pois o evento fica "guardado"). Esse *bug* pode ser ativado de outras maneiras (a maneira descrita acima é a que possui menos pré-requisitos) e causa vários comportamentos estranhos, pois praticamente qualquer interação dentro do jogo, como conversar com *NPCs*, obter novos itens e até abrir portas pode ser interrompida. Esse *bug* também pode ser utilizado para executar outras técnicas, por exemplo uma técnica famosa que descreveremos na próxima seção (2.3.4).

2.3.4 Combinações de *bugs* mais simples

Existem diversas estratégias que os desenvolvedores podem utilizar para garantir que as invariantes necessárias para seus algoritmos valham. O código pode verificar explicitamente se as condições relevantes são válidas, mas também seria possível omití-las, por exemplo ao realizar testes simplificados ao modificar certos valores como garantia. Em geral, essas decisões são principalmente influenciadas pelo *design* do jogo em questão.

Dependendo desses detalhes, dois ou mais *bugs*, individualmente inúteis para *speedruns*, podem ser combinados de maneiras criativas a fim de explorar as verificações de invariantes realizadas pelo código. O resultado disso pode ser tanto a criação de técnicas alternativas mais consistentes de realizar um *bug*, como a descoberta de *glitches* novos. Principalmente em

jogos mais antigos, que já foram analisados mais a fundo, esse é um dos principais mecanismos pelos quais novas técnicas são descobertas.

Exemplo 7: *Superswimming* (*The Wind Waker*)

Outro *bug* existente em *The Legend of Zelda: The Wind Waker* (Nintendo GameCube) conhecido como *superswimming*, similar ao *BLJ* (seção 2.3.1), ocorre devido ao fato de que, por alguma razão[14], ao mudar de direção enquanto o Link (o personagem da série) está nadando, o jogo aumenta a magnitude de sua velocidade em média por 3 unidades na direção contrária. Isso pode ser explorado se o jogador realizar uma mudança de direção enquanto nada a cada *frame* do jogo e assim acumular velocidade sem um limitante superior explícito (dependendo apenas do tempo até o medidor de ar do jogador acabar[15]).

Como podemos ver na figura 2.4, a velocidade do personagem aumenta linearmente, apesar de oscilar consideravelmente. Por muito tempo essa técnica não era útil para realizar em tempo real pois é humanamente impossível modificar a direção do *joystick* a cada *frame*, apesar de ser trivialmente possível em *TASes*. Um método de realizar esse *bug* em tempo real, utilizado poucas vezes em *speedruns* do jogo, é repetidamente pausar o jogo e alternar a direção cada vez. Esse método é comparavelmente ineficiente e demasiadamente demorado, mas as velocidades extremas que podem ser obtidas a partir desse *glitch* compensam o tempo gasto.

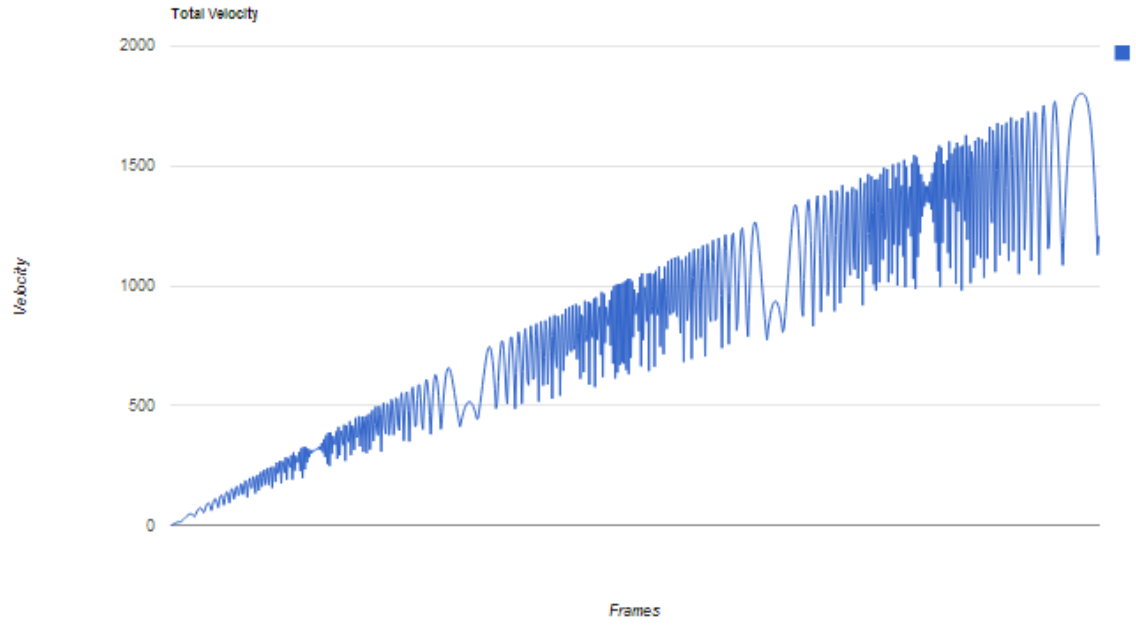

Figura 2.4: *Gráfico da oscilação da velocidade do jogador por número de* frames *do* superswim. *Fonte: ZeldaSpeedRuns (2013).*

Um aspecto interessante do item *Wind Waker* é que durante seu uso a câmera do jogo fica

[14] A teoria mais aceita pela comunidade é que isso é uma consequência de uma animação do personagem ao mudar de direção na água.

[15] É possível conservar a velocidade atual ao encostar rapidamente em terra firme para reabastecer o medidor de ar, se o jogador rapidamente voltar para dentro d'água.

com a posição travada imediatamente de frente para o Link. Uma consequência de realizar o *storage* (descrito na seção 2.3.3) é que, ao armazenar o uso do *Wind Waker* com esse *glitch*, o jogador consegue se movimentar enquanto as propriedades do item, em particular a posição fixa da câmera, estão ativas. Isso significa que a medida que o jogador se movimenta a câmera continua travada na mesma posição relativa ao personagem e, portanto, ao se movimentar com o *joystick* para cima, o personagem muda de direção e, consequentemente, a câmera imediatamente se movimenta novamente para frente do Link. Essa característica é extremamente útil pois permite a realização de *superswims* apenas segurando uma direção no *joystick* dentro d'água, pois a câmera fixa faz com que a direção do personagem oscile naturalmente. Para controlar a direção na qual o Link vai nadar pode-se pausar o jogo e verificar a sua orientação no mapa, soltando o *joystick* caso seja favorável.

Como o mundo do jogo é muito grande e há uma grande distância entre as ilhas (que os desenvolvedores esperem que você viaje por meio de barco), *superswims* são essenciais para diminuir o tempo de viagem no início de uma *speedrun*, principalmente pois o único item necessário para a realização de *storage* é o *Wind Waker*, um dos primeiros itens obtidos no jogo. Em particular, esse item pode ser obtido mais cedo do que o normal ao realizar um *superswim* pausando o jogo repetidamente, permitindo muito mais mobilidade ao início do jogo do que normalmente é possível.

Exemplo 8: Vários *bugs* e suas combinações (*Ocarina of Time*)

O jogo *The Legend of Zelda: Ocarina of Time* (Nintendo 64) possui uma grande comunidade de *speedrunners* e quantidade de *bugs*, e muitos deles essencialmente consistem da sobreposição de dois eventos para interromper um deles. Falaremos superficialmente sobre alguns desses *bugs* e suas utilidades para a economia de tempo em *speedruns*. Recomendamos que o leitor visite a Wiki do *ZeldaSpeedRuns* [16] para ver mais detalhes sobre os *bugs* nesse jogo e como realizá-los. A comunidade de *speedrunners* para esse jogo é uma das maiores e mais ativas e novos *bugs* foram encontrados nos últimos 2 anos, mesmo com o enorme número de *glitches* já conhecidos.

Um dos *bugs* mais antigos do jogo é o *Infinite Sword Glitch* (*ISG*), que consiste em usar a espada e interromper o término do golpe com algum outro evento, por exemplo lendo uma placa ou falando com um *NPC*. Como o golpe é interrompido a espada continua com um volume de colisão ativo e qualquer inimigo que encostar nela toma dano igual ao dano do golpe interrompido. Adicionalmente o Link não pode cair de plataformas enquanto a espada está ativa, o que leva ao fato de que se *ISG* estiver ativo e o jogador usar seu escudo para bloquear alguma fonte de dano (inimigo ou explosão) enquanto estiver no ar, o jogo considera que o personagem está no "chão" no momento da colisão e, como sua espada está ativa, não deixa ele cair, o que torna possível literalmente levitar (chamado de *hovering* pelos jogadores) no ar por meio de pulos interrompidos por fontes de dano. Essas técnicas são bem antigas e extremamente úteis para *speedruns*, permitindo o acesso a várias áreas normalmente

[16] http://zelda.speedrun.com/oot/

impossíveis de serem alcançadas.

Para se movimentar rapidamente por longas distâncias existe uma técnica, chamada *superslide*, que utiliza um escudo, algo que o personagem consegue levantar e alguma fonte de dano. Ao rolar para frente usando o botão A, enquanto o jogador segura seu escudo usando os botões Z e R, existe um conjunto de *frames* no qual se Link tentar segurar um objeto ao mesmo tempo que leva dano, o escudo impede o dano e a animação do personagem fica travada em uma posição na qual o jogo não atualiza sua posição como normalmente deveria, mantendo-a constante. A fonte de dano causa uma grande velocidade negativa no Link que, devido a seu estado resultante, não é atualizada até o jogador soltar o escudo (deixando de apertar Z ou R). Um item que pode ser usado para realizar esse *bug* são bombas, que são ambas uma fonte de dano e podem ser levantadas pelo Link.

Outro *bug* bem famoso é a duplicação de garrafas, que consiste em interromper a captura de algum item no momento certo (por exemplo pausando no meio da animação de captura) e modificando a posição dos itens equipados. Quando o item é capturado o jogo utiliza o valor do último botão de item utilizado para saber qual posição do inventário corresponde ao item equipado no botão e portanto deve ser sobrescrito com a nova garrafa (que possui um *ID* de item diferente dependendo do conteúdo). Como o jogo não atualiza o valor do último botão utilizado ao modificar a posição dos itens no inventário, a posição do novo item é usada ao invés da posição da garrafa original. O resultado é que, ao equipar outro item qualquer na mesma posição que a garrafa, o item é sobrescrito por uma nova garrafa. Não é um *bug* necessariamente útil por si só pois pode sobrescrever itens que só podem ser obtidos uma única vez, mas possui variações mais úteis que possibilitam escrever o valor de uma garrafa no botão B, normalmente reservado apenas para espadas.

Os itens do jogo funcionam da seguinte maneira: ao equipar um item em algum botão, o jogo guarda qual posição do seu inventário está associado ao item e copia o *ID* do item para os itens equipados. Um *bug* descoberto recentemente (em 2017), chamado *equip swap*, se utiliza do fato que, ao mudar de alguma tela do menu de pausa para a tela do inventário, existe um intervalo de 1 *frame* no qual é possível movimentar o cursor do menu e equipar um item ao mesmo tempo. Como o menu está no meio da transição, o jogo utiliza o valor do último item apontado, mas o associa com o *slot* de um outro item no inventário, para onde o cursor se movimentou. Desse modo, o valor de item é diferente do item na posição do inventário associada, mas isso não previne o jogador de utilizar o item.

Isso torna possível equipar itens que normalmente não podem ser equipados, como usar itens de adulto enquanto criança e vice versa (normalmente existe um conjunto de itens que só pode ser utilizado quando criança e outro que só pode ser utilizado como adulto). Esse truque também permite, por exemplo, associar uma garrafa com algum conteúdo com outra posição de inventário associado a outro item, o que significa que quando o conteúdo da garrafa é modificado, a garrafa original no seu inventário fica intacta e o item associado a posição do inventário é transformado em uma garrafa vazia.

O jogo é separado em várias fases, com um chefe no final de cada uma. Após derrotado,

o jogador se teletransporta para uma cena (em inglês *cutscene*) que conta a história do jogo que não pode ser evitada normalmente. Usando uma técnica na qual o modelo do personagem está segurando uma garrafa na mão, realiza um pulo e antes de encostar no chão aperta os botões para usar a garrafa e algum outro item rapidamente em sequência, o jogo tenta tocar a animação de descarregar a garrafa ao encostar no chão (pois a garrafa já estava na mão e o botão foi apertado), porém utilizando o outro item ao invés da garrafa. Como o jogo tenta tocar uma animação em um item errado, uma animação padrão acontece, que corresponde a usar a Ocarina. Essa técnica permite utilizar a Ocarina em lugares aonde seu uso é restrito, como por exemplo na sala de chefes no final de fases.

Por sua vez, isso torna possível entrar no teletransporte do final da fase, que normalmente o transportaria para uma *cutscene*, ao mesmo tempo que o jogador realiza a técnica acima. Isso faz com que o jogador consiga cancelar a animação e ganhe controle do seu personagem enquanto a animação do transporte está executando. Se o jogador conseguir carregar uma nova sala ou levar dano suficiente para morrer no mesmo *frame* em que o jogo tenta transportar para a *cutscene*, é possível sobrescrever o valor do mapa a ser carregado e assim forçar o jogo a carregar outro mapa diferente do esperado. Existem uma variedade de detalhes que influenciam o mapa resultante e qual *cutscene* (se houver) o jogo tenta carregar, e dependendo da configuração isso pode levar a um *crash*.

Por outro lado, isso também pode ser usado para acessar o mapa do último chefe a partir do final da primeira fase, para apenas carregar uma outra área mais conveniente para a *speedrun*, ou apenas para não ser necessário assistir a *cutscene* inteira (pois os itens recebidos pelo jogador durante as *cutscenes* são na verdade entregues imediatamente no começo, mesmo se o jogo fazer parecer que o personagem só o recebe no meio, o que significa que ao morrer ou carregar uma nova área imediatamente após a *cutscene* começar o jogador já possui a recompensa da fase). Como o jogo possui *cutscenes* bem longas, a possibilidade de pular um conjunto delas é bem atraente e acaba economizando bastante tempo ao longo de uma *speedrun*.

Similarmente a esses *bugs*, muitos dos outros *glitches* presentes em *speedruns* desse jogo são resultado de uma combinação de *bugs* e técnicas mais simples que, no fundo, consistem em interromper eventos no momento correto. Isso pode parecer muito difícil de replicar em tempo real, porém existem alguns aspectos do jogo que tornam isso factível. Primeiramente, o jogo roda em apenas 20fps, o que significa que cada *frame* dura 1/20 segundos ou 50 millisegundos, o que facilita certas coisas. Existe também o fato de que muitas das ações do personagem sempre resultam em uma velocidade, mudança de ângulo ou distância percorrida constante, tornando possível a criação de guias específicos que utilizam apenas essas técnicas para rapidamente assumir uma posição necessária para a realização de algum truque preciso.

Finalmente, mas também muito importante, ao pausar o jogo, existe uma animação comprida na qual o menu aparece e some, e durante essa animação quaisquer botões apertados no *joystick* são guardados em um *buffer* e processados pela próxima *frame*. É possível também pausar o jogo durante essa animação, o que pode fazer com que a simulação não avance

nenhum (ou poucos) *frame*, o que significa que jogadores podem teoricamente pausar o jogo repetidamente e segurar quaisquer inputs necessários durante a animação do menu, e até pausar novamente após apenas um (ou às vezes dois, ou zero) *frame* novo ser processado. A combinação de todos esses fatores significa que truques que normalmente seriam impossíveis de realizar consistentemente em tempo real sejam possíveis e muito mais consistentes (mas não necessariamente facilmente, pois ainda são truques precisos).

2.4 Considerações adicionais sobre os exemplos

Conforme mencionado na seção 2.3, apresentamos causas abstratas para ressaltar similaridades entre *bugs* de jogos diferentes. Por essa razão alguns dos exemplos também poderiam se enquadrar em categorias diferentes. Um exemplo é o *BLJ* (seção 2.3.1), que tem como consequência direta possibilitar movimentação *OoB* (seção 2.3.2). Similarmente, o *overflow* de inteiros descrito na seção 2.3.1 só é possível graças a uma combinação de dois *bugs* distintos, individualmente inúteis para *speedruns*, conforme descrevemos na seção 2.3.4.

A tabela 2.1 mostra uma possível categorização dos exemplos usando as categorias apresentadas. Para preencher essa tabela, consideramos não apenas a causa raiz por trás do *bug*, mas também as suas possíveis consequências. Por exemplo, tanto *ABHs* (seção 2.3.1) quanto *superswims* (seção 2.3.4) são utilizados principalmente como uma técnica para acumular velocidade apesar de possibilitar movimentação *OoB*, pois existem outros *glitches* para atravessar paredes e outras barreiras mais práticos do que os apresentados.

Tabela 2.1: *Possível categorização dos exemplos apresentados no texto*

	1	2	3	4	5	6	7	8
Casos de borda	•	•	•	•	•	•		•
Posição *out of bounds*			•	•	•		•	•
Interrupção de eventos	•			•	•	•		•
Combinação de *bugs*		•					•	•

Referência de exemplos:

1. *Mega Man 1* (NES) (2.3.1)
2. *Chrono Trigger* (SNES) (2.3.1)
3. *Engine Source* (PC) (2.3.1)
4. *Super Mario 64* (N64) (2.3.1)
5. *Pokémon Diamond & Pearl* (NDS) (2.3.2)
6. *The Legend of Zelda: The Wind Waker* (*Storage*) (NGC) (2.3.3)
7. *The Legend of Zelda: The Wind Waker* (*Superswimming*) (NGC) (2.3.4)
8. *The Legend of Zelda: Ocarina of Time* (N64) (2.3.4)

Outro aspecto interessante a ser considerado é a relação entre esses tipos de *bugs* com programas em geral, não apenas *videogames*. Problemas comuns relacionados ao tratamento de casos de borda são *overflow* de inteiros ou erros de cálculo de tamanhos de *buffers*, mas também podem incluir *bugs* referentes a invariantes de funções que não são verificadas corretamente durante sua execução.

Por sua vez, posições *out of bounds* representam um conceito muito específico a jogos pois geralmente, os limites se referem aos limites de uma certa fase do jogo. Generalizando, esse tipo de *bug* pode surgir quando um valor utilizado pelo programa assume um valor fora de um certo intervalo válido, por exemplo ao acessar uma posição inválida de um vetor.

Bugs de interrupção ou sobreposição de eventos ocorrem geralmente em sistemas que realizam processamento assíncrono, por exemplo aplicativos que recebem informações através da Internet. Outro exemplo de uma classe de *bug* comum desse tipo são condições de corrida, nas quais um programa com múltiplas *threads* pode chegar a estados diferentes dependendo da ordem de execução de cada *thread* individual.

Em geral, vulnerabilidades que envolvem escalação de privilégios só são possíveis graças a combinação de vários *bugs* que podem ser individualmente inofensivos, mas representam um sério problema de segurança ao serem combinados. Essas técnicas são necessárias hoje em dia, em grande parte, devido a mitigações de *exploits* implementadas por sistemas operacionais modernos, por exemplo randomização de espaço de endereço e prevenção de execução de dados, que requerem informações adicionais para que um atacante consiga derrotá-las. É necessário burlar todas as defesas presentes no sistema para que o atacante consiga realmente causar execução de código arbitrário como um usuário privilegiado a partir de, por exemplo, um *bug* existente em um navegador de Internet.

Capítulo 3

Tool-assisted Speedruns

O objetivo deste capítulo é descrever técnicas e ferramentas utilizadas por *tool-assisted speedruns* (*TASes*) para manipular o comportamento de jogos. A primeira seção introduz os conceitos por trás de *TASes*. Após isso descrevemos as razões responsáveis pela dificuldade em realizar técnicas consideradas exclusivas a *TASes*. Na terceira seção descrevemos quatro técnicas que *TASes* costumam explorar, cada uma seguida de um exemplo de *TAS* que a utiliza.

3.1 Introdução

Uma *Tool-assisted speedrun* é, conforme o nome sugere, uma *speedrun* criada com a ajuda de ferramentas especializadas, i.e. emuladores com funcionalidades extras. O objetivo de um *TAS* é mostrar como seria a *speedrun* perfeita, desconsiderando o "fator humano", para fins de entretenimento. A maior comunidade de *TASers* online é representada pelo site TASvideos[1], que possuem uma extensa lista de regras para garantir a legitimidade de seus filmes e um grande número de emuladores que podem ser utilizados[2]. Existem outras publicações acadêmicas relacionadas a *TASes*, com perspectivas diferentes. Em LeMieux (2014) o autor descreve a história por trás do surgimento da comunidade de *TASes* na Internet e explica detalhes técnicos da criação e reprodução de *TASes*. Já em Newman (2013), o autor discute os aspectos legais de *TASes* no que diz respeito ao acesso e disseminação de cópias ilegais de jogos, necessária para que os emuladores de *TASers* possam ser utilizados.

Claro que não é possível garantir que um *TAS* é perfeitamente otimizado (e muitas vezes não são; afinal *TASes* são criados por pessoas, que não são perfeitas), mas a proposta é mostrar habilidades super-humanas de uma maneira interessante. O foco em entretenimento existe para motivar a criação de *TASes* surpreendentes, pois o uso de ferramentas torna certas coisas previsíveis e entediantes, como por exemplo nunca ser atingido por inimigos, já que o *TAS* não vai realizar um erro. Desse modo fica clara a importância de *bugs* para *TASers*, pois além de otimizar o tempo são muito úteis para deixar a jogatina mais interessante.

Para jogar um jogo eletrônico, existe, em todo o caso, algum dispositivo de entrada (por exemplo um controle) usado pelo jogador para interagir com o *videogame*. Do ponto de vista

[1] https://tasvideos.org/
[2] http://tasvideos.org/EmulatorResources.html

do jogador existe uma reação imediata ao realizar qualquer tipo de mudança no controle usado, mas do ponto de vista do código do jogo, existe na verdade um valor para cada botão do controle e o jogo repetidamente coleta todos os valores em um determinado instante de tempo, com um intervalo curto entre cada leitura para garantir responsividade. Esse processo acontece pelo menos uma vez por *frame* e portanto podemos caracterizar a interação entre o jogador e o jogo como uma sequência de conjunto de valores que correspondem ao estado do controle em um determinado instante de tempo.

Como computadores são máquinas determinísticas, é razoável pensar que teoricamente toda aleatoriedade usada pelo jogo é proveniente da interação do jogador e, possivelmente, de alguns fatores externos dependendo da plataforma, como, por exemplo, o valor do relógio interno. Poderíamos então imaginar um dispositivo posicionado entre o controle e o *videogame* que grava todos os sinais enviados pelo controle na mesma resolução usada pelo jogo e, por meio disso, pode reproduzir os mesmos em outro momento, fazendo o mesmo papel de um controle do ponto de vista do console. Se todos os fatores aleatórios forem exatamente os mesmos, então dentro do universo do *videogame* aconteceria exatamente a mesma coisa, pois nesses casos o conceito de aleatoriedade do jogo (i.e. o estado do seu *pseudo-random number generator* (PRNG)) é o mesmo e para o código não existe diferença entre as situações.

Legitimidade

O argumento de que *TASes* são legítimos é o mesmo usado para *speedruns* que usam *bugs*: nem o jogo nem o sistema são modificados, o único modo que o jogador interage com o jogo é através dos *inputs* enviados através do(s) controle(s). Para garantir essa legitimidade também é possível (e foi feito em alguns casos[3]) criar um circuito, conforme descrito no parágrafo anterior, que se conecta com o videogame por meio do conector de controle e toca a sequência de *inputs* do mesmo modo que o emulador, para verificar se um *TAS* é realmente replicável em um console original.

Não existe limitação no tipo de ferramenta utilizada para a confecção de uma sequência de *inputs* de um *TAS* (também chamado de "filme"), pois o que importa de fato são os *inputs* finais e não o método usado para chegar lá. *Savestates* são uma funcionalidade de emuladores essencial para a criação de bons *TASes*, pois possibilitam "re-gravar" partes do filme a partir de um determinado instante no tempo. Basicamente o emulador salva todo o seu estado relevante em um ponto do tempo e pode, a qualquer momento, carregar um arquivo para voltar a esse estado, descartando quaisquer *inputs* após aquele ponto. Isso permite com que quaisquer erros sejam eliminados, pois a qualquer momento é possível voltar e tentar uma seção novamente. Outra funcionalidade essencial é *frame advance* que consiste em pausar a emulação completamente e avançar um *frame* por vez. Desse modo o jogador possui controle total sobre o "tempo" dentro da simulação do jogo, possibilitando grande precisão durante a confecção do *TAS*. Um *TAS* também pode ser criado usando a ajuda de programas, que

[3] Alguns desses *TASes* podem ser encontrados em http://tasvideos.org/ConsoleVerifiedMovies.html, e um guia para a criação de um dispositivo desses para o NES usando um Arduino existe em https://www.instructables.com/id/NESBot-Arduino-Powered-Robot-beating-Super-Mario-/.

às vezes rodam dentro do próprio emulador usando uma API, que realizam, por exemplo, uma busca por força bruta para encontrar uma sequência de ações que minimiza o tempo, dadas certas restrições para que o problema seja factível. Combinadas, essas e outras técnicas permitem a criação de filmes super-humanos, com reflexos e previsões perfeitas e sem erros.

Características de técnicas exclusivas a *TASes*

Devido a legitimidade de *TASes*, poderíamos dizer que teoricamente tudo realizado por um *TAS* pode ser realizado em tempo real por alguém suficientemente habilidoso ou sortudo. De fato, à medida em que *speedruns* em tempo real ficam mais otimizadas e competitivas, o foco da comunidade muda para tentar incluir o maior número de técnicas possível, e isso também inclui técnicas usadas por *TASes*. Claro, em certos casos os *inputs* necessários podem ser literalmente humanamente impossíveis, mas esse é raramente o caso.

O principal fator que determina a viabilidade em tempo real de uma estratégia utilizada por um *TAS* é a sua consistência. Por sua vez, a consistência de uma técnica está diretamente correlacionada com a precisão necessária para sua execução. Mesmo assim, *speedrunners* utilizam técnicas extremamente precisas em casos que o ganho de tempo é muito alto e os riscos de falhar podem ser minimizados, por exemplo modificando a rota para realizar técnicas mais difíceis o mais cedo possível, possibilitando um maior número de tentativas. Dependendo de aspectos como tipo de dado utilizado para guardar informações do jogo e número de *frames* processados por segundo, a precisão necessária pode ser muito grande para que certos truques possam ser realizadas consistentemente.

Por essas razões, a dificuldade de realizar *bugs* em geral é considerada sob a perspectiva da relação entre o ganho de tempo proporcionado e a taxa de sucesso de tentativas. Para otimizações pequenas realizadas em *TASes*, o ganho de tempo é marginal e a taxa de sucesso é muito baixa, e por isso essas técnicas não são incorporadas em *RTAs*. Porém quando o ganho de tempo é muito significativo, *speedrunners* tentam incorporar essas técnicas na medida do possível, independente da taxa de sucesso.

3.2 Tipos de técnicas utilizadas frequentemente

Como um *TAS* é, essencialmente, uma generalização de *speedruns* em geral, as técnicas utilizadas são bem similares. Todas as técnicas usadas por *speedrunners* em tempo real podem (e devem) ser utilizadas por *TASes*, mas as ferramentas utilizadas permitem um nível de otimização muito maior do que qualquer ser humano é capaz. Apresentaremos alguns exemplos de técnicas comumente consideradas exclusivas a *TASes* devido a sua precisão, apesar de algumas delas serem tecnicamente possíveis em tempo real.

3.2.1 Manipulação de aleatoriedade

Jogos em geral não necessitam de número aleatórios de alta qualidade (diferentemente de operações criptográficas por exemplo), e portanto a aleatoriedade presente em consoles de *videogame* é atualizada diferentemente dependendo de fatores difíceis de serem controlados, como o *timing* (medido em *frames*) de certas ações. A aleatoriedade é implementado por

meio de um *pseudo-random number generator* (*PRNG*, às vezes abreviado apenas para *RNG*) que é constantemente atualizado pelo jogo toda vez que um valor é gerado e o jogador realiza algum *input* e, para um jogador normal, parece completamente aleatório. No caso de *TASes*, é possível analisar o código responsável por atualizar o *PRNG* e simular, a partir do valor atual, valores futuros. Similarmente, o modo pelo qual o valor resultante é mapeado em mecanismos do jogo pode ser entendido e portanto é possível manipular aspectos aleatórios. Um exemplo são jogos no qual o valor exato do dano de um ataque é aleatoriamente escolhido dentro de um certo intervalo, que significa que o jogador pode esperar (ou realizar ações que avancem o estado do gerador) até que o valor do *PRNG* leve ao dano máximo dentre as possibilidades.

Emuladores possuem funcionalidade para analisar e pesquisar o conteúdo da memória do jogo, através do qual é possível encontrar a posição de memória correspondente ao estado interno do *PRNG*. Usando *debuggers*, é possível pausar a emulação do jogo toda vez que essa posição de memória é modificada ou acessada, e assim podemos localizar os trechos de códigos que implementam o *PRNG* e descobrir como ele é atualizado. Nesse ponto, é possível testar várias possibilidades de ações e tempos de esperas diferentes, por meio de *savestates*, até encontrar uma possibilidade favorável. Dependendo da complexidade do jogo, também é possível analisar o código que utiliza o valor aleatório para entender a relação entre o estado do *PRNG* e o resultado final desejado (como a aparição de um monstro ou o valor de dano causado).

Em geral é necessária muita precisão para que a manipulação leve ao resultado esperado, e por isso é muito difícil realizar isso em tempo real. Algumas *speedruns* em tempo real, porém, utilizam sequências de *input* para manipular o estado do jogo quando isso é muito importante para a economia de tempo. Independente da factibilidade, *TASes* sempre podem fazer isso, mesmo se for apenas através de tentativa e erro por meio de *savestates*, sem nenhuma engenharia reversa, e portanto a "sorte" também não é um fator para um *TAS*.

Exemplo 1: Manipulação de itens recebidos (*Mario Kart 64*)

Um exemplo de manipulação de *RNG*, no jogo *Mario Kart 64*, realizado por meio de engenharia reversa é o trabalho de Weatherton em Weatherton (2016), no qual o autor, além de otimizar seu movimento e estratégias ao longo de um período de 7 anos, criou um programa em `Lua` que simula o código responsável por escolher um item aleatoriamente. Vamos descrever em detalhe o mecanismo usado pelo jogo para gerar números aleatórios, pois representa os conceitos que apresentamos acima. Gostaria de ressaltar que em Weatherton (2016) o autor utiliza-se de muitas outras técnicas e conhecimento profundo delas para chegar ao produto final, e a manipulação de itens, apesar de ser muito útil é apenas uma das ferramentas utilizadas.

Primeiramente, é necessário entender como a implementação do jogo modela o comportamento aleatório que gostaríamos de manipular. Esse jogo em questão utiliza uma tabela para mapear o resultado do gerador de número aleatórios com um dos 15 IDs de itens possíveis. O

jogo usa diferentes tabelas dependendo da posição do jogador na corrida e modo selecionado (como *Versus*, *Grand Prix*, *Battle*). A distribuição de valores presentes nessa tabela[4] nos informa sobre a probabilidade de obter cada item dado o lugar na corrida. No modo principal do jogo, *Grand Prix* de um jogador, a distribuição da tabela utilizada pelo jogo pode ser vista na figura 3.1.

Item	Code	1st	2nd	3rd	4th	5th	6th	7th	8th
Banana	1	30%							
Banana Bunch	2	5%	5%						
Green Shell	3	30%	5%						
Triple Green Shell	4	5%	10%	10%					
Red Shell	5	5%	15%	20%	15%	10%			
Triple Red Shell	6		20%	20%	20%	20%	20%	20%	20%
Spiny Shell	7				5%	5%	10%	10%	15%
Thunder Bolt	8		5%	5%	10%	10%	15%	20%	20%
Fake Item Box	9	10%	5%						
Super Star	A		5%	10%	15%	15%	20%	30%	30%
Boo	B	5%	5%						
Mushroom	C	10%	5%	5%	5%	5%			
Triple Mushrooms	E		15%	20%	20%	25%	25%	10%	5%
Super Mushroom	F		5%	10%	10%	10%	10%	10%	10%
Total		100%	100%	100%	100%	100%	100%	100%	100%

Figura 3.1: *Tabela de distribuição de probabilidade de itens para o modo* Grand Prix*, single-player. Fonte: Weatherton (2016).*

Também precisamos entender exatamente como o gerador de números aleatórios do jogo funciona, para que seja possível realizar simulações precisas e coerentes com a implementação do jogo. Em Abney (2016) o autor, um *speedrunner* de *Mario Kart 64*, analisou e comentou o funcionamento do código em assembly responsável pela escolha do item, detalhando o algoritmo utilizado pelo jogo para escolher uma das entradas da tabela de itens. O jogo soma quatro fatores semi-aleatórios e pega o resultado da divisão por 100 para obter o índice final. O primeiro fator é um número aleatório proveniente de um gerador de números aleatórios igual ao utilizado pelo código de *Super Mario 64*, descrito em detalhes por Buchanan (2016a). Para manipular esse valor o jogo precisa gerar números aleatórios para outros propósitos, deste modo avançando o estado do *PRNG*, mas nem sempre é fácil manipular o jogo para que isso aconteça sem gastar muito tempo.

Outro fator da soma é o último índice da tabela calculado dessa maneira pelo jogo, que pode ser manipulado da mesma maneira. O jogo também soma um contador de *frames* desde o início da corrida, e a última variável somada é um contador que incrementa todo *frame* que algum jogador estiver apertando o botão A (acelerador), B (breque) e R (pulo). Apesar de ser diretamente manipulável, essas ações modificam a velocidade do seu kart e portanto só podem ser apertados livremente em alguns estados, e portanto o jogador perde velocidade caso esse valor seja manipulado. Após colidir com um item, o jogador possui alguns segundos durante o qual o jogo simula uma roleta, que pode ser parada pelo jogador com o botão

[4]Presentes em sua totalidade nos comentários de Weatherton (2016).

Z. Como o jogo roda em 59.94*fps*, precisamos apenas de 1.66 segundos para incrementar o contador da corrida em 100 unidades, e portanto podemos escolher qualquer item que desejarmos.

Esses conceitos podem ser utilizados para modelar outros aspectos em jogos, aleatórios ou não, como critérios usados pelo código para controlar entidades como adversários. O mecanismo utilizado para isso por *TASers* é muito similar ao descrito, independente do aspecto do jogo sendo manipulado.

3.2.2 Execução precisa com pouca margem para erros

Assumindo que estamos lidando com um jogo que lê o estado do controle 60 vezes por segundo, significa que em apenas 10 segundos existem 600 momentos diferentes nos quais qualquer combinação de botões pode estar apertado. Em consoles que possuem *joysticks* analógicos existe um grande intervalo de valores possíveis para cada eixo correspondente. Se um *TAS* possui seções que envolvem movimentação, posicionamento ou *timing* precisos realizados em rápida sucessão, é extremamente difícil, se não simplesmente impossível, realizar isso em tempo real.

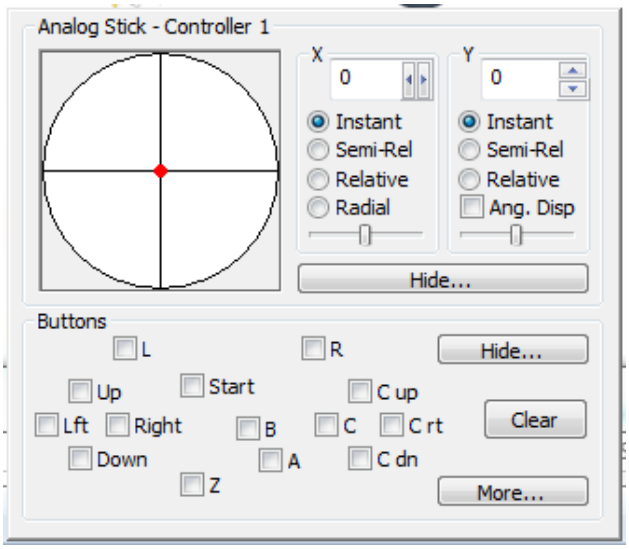

Figura 3.2: *Tela de* input *do emulador Mupen64, para o Nintendo 64. Fonte: https://mupen64plus.org/.*

No caso de *TASes*, isso claramente não é uma preocupação, e existem interfaces especializadas, como por exemplo na figura 3.2, para controlar exatamente o estado do controle em cada *frame*. Também é possível buscar por e visualizar os valores de memória correspondentes a variáveis como posição, velocidade, vida, e outros atributos de entidades do jogo, como podemos ver na figura 3.3. Essas ferramentas são essenciais para otimizar com mais granularidade aspectos locais como atravessar pequenas telas ou seções, ou conservar o máximo de velocidade e aceleração possíveis. Otimizações desse tipo não economizam muito tempo em cada instância, mas quanto mais comprido é o *TAS*, mais tempo é economizado devido a esses tipos de otimizações.

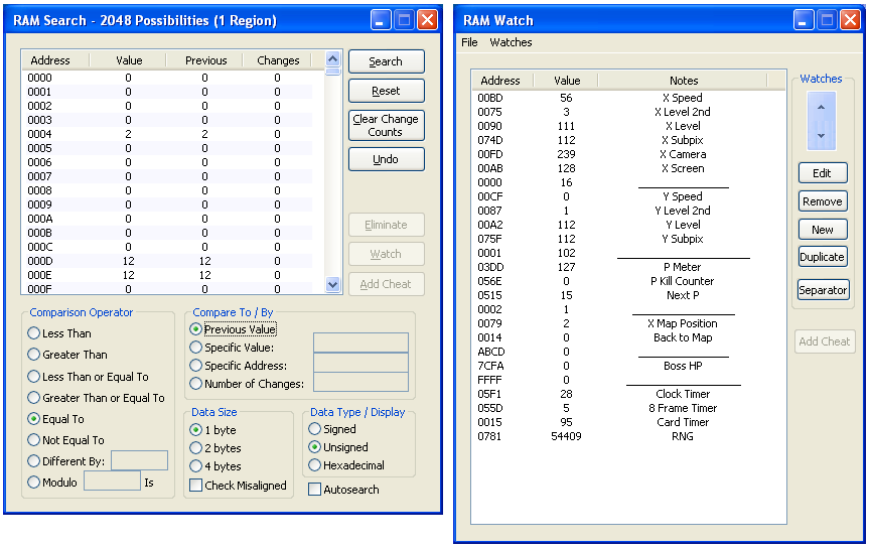

Figura 3.3: *Telas para visualizar o estado da memória do jogo do emulador BizHawk. Na esquerda temos a ferramenta para encontrar posições de memória, e à direita a ferramenta de visualização. Fonte: http://tasvideos.org/EmulatorResources/RamSearch.html e http://tasvideos.org/EmulatorResources/RamWatch.html.*

Exemplo 2: Posicionamento usando velocidades extremas (*Super Mario 64*)

Até 2016 a rota usada pelo *TAS* de *Super Mario 64* consistia em utilizar *BLJs* (seção 2.3.1) para atravessar todas as três portas que requerem estrelas para dar acesso a uma das fases do *Bowser*, aonde as duas primeiras eram necessárias para obter duas chaves, uma para o porão do castelo e outra para os andares superiores. A razão que apenas algumas das portas podiam ser ignoradas é que algumas das portas são usadas como bloqueios físicos enquanto outras são usadas como transição de mapas, por exemplo a transição do exterior para o interior do castelo. É possível atravessar essas portas e acessar a geometria atrás delas, mas não há nada atrás delas e a única forma de mudar de nível é interagindo com a porta diretamente, o que força o jogador a obter as respectivas chaves. Isso significa que, similarmente, é impossível acessar a parte superior do castelo (que contém o último nível do jogo) sem possuir a chave do porão. O método normal de acessar o porão é por meio do interior do castelo e, pela mesma razão, necessita da chave para o porão.

Para guardar informações como posição e velocidade de objetos, o código utiliza ambos números em ponto flutuante como números inteiros, dependendo da operação sendo realizada. Colisão com chãos e tetos, por exemplo, usam valores inteiros de 16-bits para comparar as posições dos objetos e da geometria do nível. Colisão com objetos, paredes e água, por sua vez, utiliza valores em ponto flutuante. Nenhum nível do jogo é grande o suficiente em unidades de espaço para qualquer um desses valores passar do valor máximo ou mínimo de 16-bits em jogatina normal. Por essa razão o código do jogo apenas arredonda o número para um inteiro e assume que é pequeno o suficiente para caber em 16-bits, ou seja, pega apenas os 16 bits menos significativos do número.

A implementação de verificação de colisões do jogo desse modo significa que, na prática,

cada posição válida dentro de um nível é geometricamente equivalente a mesma posição modificada de um múltiplo inteiro de 2^{16} em qualquer direção. Apesar disso, essa propriedade não é particularmente útil por si só pois outros objetos não existem nesses universos paralelos e algumas ações, como colidir com uma parede diretamente ou posicionar a câmera fora dos limites do nível, fazem o jogo travar devido a operações inválidas[5]. Em termos de benefícios a *speedrunners*, o maior deles é que isso permite muito mais flexibilidade no posicionamento do Mario ao usar *BLJs* pois o código do jogo considera uma posição como *out of bounds* quando não há um chão embaixo do Mario e nesse caso não atualiza a posição do jogador. Isso significa que quando o jogador estiver se movimentando rápido o suficiente para sua nova posição ser uma posição fora do nível observamos que o Mario fica parado (realizando a mesma animação de empurrar uma parede) por essa razão. A existência de universos paralelos torna possível mudar de posição dentro do mesmo com uma velocidade muito maior que o tamanho do nível.

Para entendermos exatamente como isso é feito, é necessário entender alguns detalhes de implementação dos trechos de códigos responsáveis por atualizar a posição do Mario a cada *frame* após a verificação de colisões. O código divide o passo pretendido (a posição atual mais a velocidade) em 4 e verifica se há uma colisão ou posição inválida em cada um, modificando a posição final se houve alguma colisão. Se qualquer posição intermediária cair fora dos limites do nível, todo o movimento do Mario é cancelado e ele volta para sua posição inicial, e isso significa que a velocidade necessária para se movimentar para um universo paralelo é $4 * 2^{16} = 2^{18}$, pois todas as 4 posições verificadas pelo código precisam ser múltiplos de 216. Usando valores ligeiramente maiores ou menores que 2^{18} nos permite mudar a posição do Mario relativamente dentro do nível.

Um dos maiores avanços no *TAS* desse jogo foi a utilização de universos paralelos para abrir uma porta dentro do fosso do castelo que realiza uma transição direta ao porão do castelo a partir do lado de fora (descoberto e realizado pela primeira vez por Kehne (2015)). Normalmente o jogador precisa remover a água do fosso do castelo a partir de botões dentro do porão do castelo para ter acesso a essa porta, apesar dela existir e não estar trancada, apenas estar localizada debaixo d'água. Existe um *delay* de um *frame* quando o Mario entra debaixo d'água na qual é possível interagir com objetos (nesse caso uma porta) antes do jogo modificar o estado do personagem de "andando" para "nadando" (um estado aonde não é possível abrir portas de nenhuma maneira), e o resultado dessa interação pode ser visto na figura 3.4.

A última condição necessária para a realização disso é: como acumular tanta velocidade usando a geometria presente no exterior do castelo? De fato não existe nenhum lugar nesse mapa no qual o Mario consegue acumular a velocidade necessária, pois todo lugar aonde é possível realizar um *BLJ* são ladeiras que não permitem que o jogador realize o número de pulos necessário antes de chegar no topo da ladeira e sair de posição. Se houvesse um elevador

[5]É possível evitar isso pois o jogo dá a opção de impedir a movimentação da câmera.

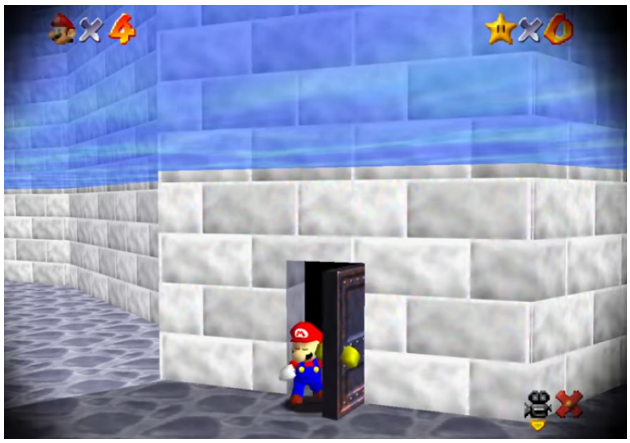

Figura 3.4: *O momento em que o Mario abre a porta do fosso do castelo, após a realização do* bug *descrito. Fonte: Kehne et al. (2016).*

encostado em uma parede, algo presente em alguns níveis por exemplos, seria possível obter a velocidade necessária, mas precisaríamos conservar essa velocidade de algum modo até o exterior do castelo. Existe uma técnica para conservar a velocidade proveniente de *BLJs* que consiste em apertar o botão *Z* no mesmo *frame* ao sair de dentro d'água devido a colisão com um chão. Isso faz com que o Mario esteja em um estado que utiliza os valores da velocidade antes de entrar na água, que não são atualizados para 0 ao entrar na água[6].

Convenientemente, existe um nível que pode ser acessado dentro do fosso do castelo, *Vanish Cap Under the Moat* (*VCUtM (DEL)*), no qual, por sorte, existem elevadores nos quais é possível acumular muita velocidade (bem mais que 2^{18}). Essa velocidade pode ser conservada pois o jogador é colocado diretamente dentro d'água imediatamente após sair do nível, e portanto é possível combinar as técnicas descritas anteriormente para finalmente obtermos a velocidade que queríamos no exterior do castelo. Após isso falta "apenas" mudar a sua posição relativa dentro do nível (por meio de universos paralelos) de modo que o Mario encoste na porta ao mesmo *frame* em que entra dentro d'água.

Esse *bug* representa o último passo no trabalho de *TASers* desse jogo para pular o máximo do jogo possível. Ao longo dos anos os requisitos para completar o jogo foram diminuídos, e hoje em dia o *TAS* realiza apenas o absolutamente necessário para ativar o final do jogo, ignorando todos os outros aspectos do jogo por meio de *BLJs* bem posicionados. Uma das únicas maneiras que isso poderia ser melhorado é se, de alguma maneira, fosse possível acessar o nível superior do castelo sem obter a chave do porão, o que é considerado impossível com o conhecimento atual do jogo.

3.2.3 Corrupção de memória e do fluxo de execução do programa

Ocorre quando o jogador consegue colocar um valor inválido em alguma variável usada pelo jogo ou quando ele modifica uma posição de memória que não deveria conseguir. A consequência desse *bug* depende do nível de controle que a falha dá sobre a posição ou

[6] Ver Buchanan (2016b) para uma explicação que entra em mais detalhes sobre os métodos de conservação de velocidade conhecidos atualmente.

valores modificados pelo jogador. A corrupção pode ser inofensiva e não afetar nenhum valor usado pelo código, mas, em casos extremos, pode ser usada para modificar qualquer valor de memória do programa para qualquer outro valor, e nesse caso isso poderia ser usado para, literalmente, fazer qualquer coisa: o jogador poderia escrever uma sequência arbitrária de código de máquina e modificar o código do jogo para executar seu código no momento apropriado.

O processo de descoberta de condições necessárias e métodos de controle referentes a esses tipos de *bugs* só é possível devido as funcionalidades de *debug* de emuladores especializados, mesmo que o *bug* resultante possa ser realizado por um jogador em tempo real, devido a necessidade de analisar a memória e código do jogo durante sua execução. Às vezes o jogo trava o console devido a algum problema que não é considerado pelo código. Esses *crashes*, no nível de código de máquina, podem acontecer pois o processador executa instruções inexistentes ou acessa posições inválidas de memória. Nesses casos o *debugger* de um emulador pode nos dizer em qual trecho de código ocorreu o problema e quais eram os valores dos registradores e memória no momento do *crash*.

Nesses casos, um processo de engenharia reversa pode elucidar o significado desses valores e posições de memória para descobrir o que é manipulável pelo jogador ou não. Encontrar uma configuração de valores que não causam um *crash* e executam um trecho código específico subverte o fluxo de execução do processador e permite a execução de qualquer tipo de código. Isso é chamado de *total control* ou *arbitrary code execution* (*ACE*). Em termos de *speedruns* o código mais útil e direto seria executar a função responsável por mostrar os créditos ou ativar o final do jogo, efetivamente pulando todo o jogo após a realização do *bug*. Muitos desses *glitches* foram replicados em consoles para comprovar que seu comportamento não depende de *bugs* de emulação e dar legitimidade ao *bug*, pois é possível que erros na implementação do emulador causem divergências desse tipo na execução do jogo.

Exemplo 3: Corrupção no tamanho de listas causa *buffer overflow* (*Pokémon*)

Os jogos da primeira geração de *Pokémon*, para o *Gameboy*, são infames por possuírem poucas garantias sobre valores de memória inválidos. Em outras palavras, os programadores não verificam se as invariantes necessárias valem de fato e por consequência o jogo pode acessar posições inválidas de memória quando índices de vetores são calculados. Um exemplo é o código relacionado a manipulação de itens do inventário do jogador. Em circunstâncias normais o jogador não consegue obter mais do que 20 itens pois o jogo verifica o número de itens ao adicionar um novo item. Porém isso não é verificado ao mostrar ou manipular o inventário pelo menu do jogo, o que significa que ao corromper o número de itens ou pokémons que o jogador possui o código livremente acessa valores fora da área de memória reservada para tal. Existe uma listagem do código de máquina do jogo comentada com nomes de variáveis e funções em https://github.com/pret/pokered que pode ser usada para verificar os comportamentos descritos.

Ao salvar os dados do jogador para a memória persistente o código copia os dados em três

etapas. Por sorte a primeira etapa é a responsável por copiar a *checksum* utilizada pelo jogo para verificar se o *save* é válido ou não. Isso significa que ao reiniciar o console (desligando e ligando) imediatamente após escrever essa *checksum* o jogo não copiou os respectivos valores referentes ao resto dos dados, mas o código não detecta essa corrupção. Ao deletar o arquivo salvo por meio de uma combinação de botões na tela de título do jogo, a área de memória correspondente é completamente sobrescrita com valores 0xFF, os mesmos valores que vem de fábrica. Isso significa que, depois de reiniciar o console, as regiões de memória após a primeira seção é composta inteiramente de valores 255, muito maiores do que normalmente possível. Em particular o número de pokémons que o jogador possui no momento é uma das variáveis afetadas, e isso pode ser usado para modificar valores de memória ao trocar entradas no menu de posição. Um dos valores que pode ser modificado dessa maneira é o número de itens do inventário do jogador, que por sua vez permite modificar mais valores de memória com maior granularidade pois cada item é representado por um par de bytes.

O resultado disso é que o modo mais eficiente de completar esses jogos sem nenhuma restrição é imediatamente corromper a memória do jogo e ativar o fim do jogo. Um *TAS* que demonstra isso é MrWint (2014), no qual a duração total do filme é essencialmente o tempo que demora para terminar a introdução do jogo.

Em particular, essa técnica não é muito precisa e pode ser facilmente realizada em tempo real com uma consistência aceitável. A precisão dela envolve apenas em desligar o jogo em um momento específico enquanto o código está gravando as informações, uma janela de 1 *frame* ou aproximadamente 16 milisegundos. De fato, outros *bugs* similares que permitem executar código arbitrário dentro de outros jogos também podem ser executados em tempo real. Apesar disso, essa classe de *bugs* não poderia existir sem as ferramentas disponibilizadas por emuladores especializados pois é necessário analisar a memória do jogo durante sua execução para descobrir como a sequência de ações realizadas afeta as variáveis usadas pelo código e como corromper esses valores de uma maneira útil. Por essa razão incluímos esse *bug* nessa seção. Como literalmente o jogo inteiro é pulado, esse *bug* é banido de todas as categorias populares de *speedruns* em tempo real pois trivializa demais o jogo, apesar de ser tecnicamente o método mais rápido de terminar o jogo.

3.2.4 Busca por força bruta nos estados do jogo

Como não existem limitações no tipo de ferramenta utilizada para criar um *TAS*, os autores possuem a liberdade de criarem ou modificarem qualquer tipo de programa para chegar nos *inputs* desejados. Uma estratégia simples mas eficaz é realizar uma busca exaustiva sob os possíveis movimentos a fim de encontrar uma sequência mínima de passos para alcançar algum objetivo no jogo, por exemplo terminar uma fase ou realizar uma manobra específica. Essa técnica não funciona em geral pois mesmo jogos muito simples possuem um espaço de possíveis estados gigantesco, e portanto essa técnica só é viável caso o escopo da busca possa ser limitado de alguma maneira.

Um exemplo simples dessa técnica é o *TAS* de *Mario Kart 64*, descrito anteriormente na

seção 3.2.1, que utilizou um programa em `Lua` para buscar *inputs* para manipular o *PRNG* do jogo sem gastar muito tempo. Para que a busca termine rapidamente, a profundidade de busca do programa é limitada para poucos *frames* no futuro, e mesmo assim o processo chega a ser demorado.

Exemplo 4: *Inputs* gerados inteiramente por um programa (*Lunar Pool*)

Um caso extremo dessa técnica pode ser visto em dois *TASes* (Yliluoma (2010a) e Yliluoma (2010b)) do jogo de sinuca *Lunar Pool* (NES). O autor modificou um emulador[7] para verificar todas as possibilidades de ângulo e força para todas as tacadas, utilizando diversas heurísticas para minimizar o número de jogadas subsequentes a serem testadas.

A estratégia do programa é verificar, para todo ângulo e valor de velocidade da bola resultante, o tempo total que a tacada demora e quantas bolas são encaçapadas. O programa verifica todas as possibilidades e guarda sempre a que minimiza o número de *frames* gastos. Para realizar isso, o código chama funções internas do emulador para definir o *input* e avançar o estado do jogo um *frame* de cada vez.

O jogo também possui uma opção que permite configurar a quantidade de atrito simulado pelo jogo, e até desabilitá-lo completamente[8]. Ao jogar sem atrito, é relativamente fácil fazer com que o jogo entre em um *loop* infinito, pois todas as colisões entre as bolas e obstáculos são perfeitamente elásticas. Por essa razão, é muito provável que os desenvolvedores do jogo não esperam que o jogador termine todas as 60 fases do jogo nesse modo, pois caso o jogo entre em *loop* infinito a sua única opção é reiniciar o console, tornando a experiência de jogar nesse modo muito frustrante. Porém, para o autor de Yliluoma (2010b), isso é uma vantagem significativa, exatamente porque a grande maioria de tacadas possíveis nunca terminam, e portanto é possível reduzir drasticamente o espaço de busca do programa. Desse modo, o autor também consegue terminar o jogo em uma modalidade considerada impossível em tempo real.

Como em 2010, quando esse *TAS* foi feito, os emuladores usados pela comunidade não possuíam uma interface programável, foi necessário modificar o código do próprio emulador para tornar isso possível. Hoje em dia, a maioria dos emuladores usados para a criação de *TASes* possuem uma interface em `Lua`, que poderia ser utilizada para esse mesmo propósito, já que ela dá acesso às funções para controlar o emulador diretamente.

[7] As modificações podem ser encontradas na seguinte página: http://tasvideos.org/Bisqwit/Source/Bots/LunarBall.html.

[8] O jogo permite escolher um valor de atrito entre 0 e 255.

Capítulo 4

Observações finais

Speedruns demonstram como *bugs* presentes em jogos podem ser utilizados para modificar a maneira de se jogar *videogames*. Usando conceitos de engenharia reversa e análise de software, os jogadores desvendam o funcionamento do jogo e utilizam isso da maneira mais eficiente possível. Similarmente, *TASes* são uma generalização de *speedruns* na qual jogos são considerados independentemente do fator humano, criando uma forma de arte fora do comum. Apesar de existir muita sorte também, pois alguns jogos simplesmente possuem menos *bugs* que outros, existem muitos conceitos que podem ser aplicados na busca de *glitches* e suas aplicações. Isso também permite que *speedruns* transformem o jogo em algo muito diferente do esperado normalmente, o que ajuda a comunidade a crescer cada vez mais.

Também mostramos como a utilização de ferramentas especiais ajuda a analisar e manipular o comportamento do jogo em *TASes*. Tudo isso realizado usando apenas *input* válido ao jogo, tornando possível a criação de filmes visual e tecnicamente impressionantes. Existem diversos outros *bugs* surpreendentes utilizados em *speedruns* além dos mencionados nesse texto. Uma fonte de *TASes* interessantes é a seção de filmes com "estrelas" no site do TASvideos[1], que é o ranking dado a *TASes* que a comunidade achou especialmente interessantes. Muitos deles subvertem as expectativas dos espectadores por meio de *bugs* e manipulações impressionantes, muitas vezes acompanhadas de comentários provenientes dos autores explicando o funcionamento por trás das técnicas utilizadas.

Por fim *speedrunning* representa uma perspectiva diferente de encarar jogos de *videogame*. Tanto como um fenômeno cultural, com uma comunidade online que cresce cada vez mais, como uma prática que demonstra a união de conhecimento profundo da implementação de jogos e de muita prática e habilidade pessoal. Isso resulta em um estilo de jogatina completamente único.

[1] http://tasvideos.org/Movies-Stars.html

Referências Bibliográficas

Abney (2016) Beck Abney. Código assembly do Mario Kart 64 com comentários, 2016. URL https://github.com/abitalive/MarioKart64/blob/master/Notes/item_rng.txt. Citado na pág. 29

Avizienis *et al.* **(2004)** A. Avizienis, J. . Laprie, B. Randell e C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33. ISSN 1545-5971. doi: 10.1109/TDSC.2004.2. Citado na pág. 1

Bainbridge e Bainbridge (2007) Wilma Alice Bainbridge e William Sims Bainbridge. Creative uses of software errors: Glitches and cheats. *Social Science Computer Review*, 25(1): 61–77. doi: 10.1177/0894439306289510. URL https://doi.org/10.1177/0894439306289510. Citado na pág. 1

Biagioli (2015) Adrian Biagioli. Bunnyhopping from the programmer's perspective, 2015. URL http://flafla2.github.io/2015/02/14/bunnyhop.html. Citado na pág. 11

Brewer (2017) Christopher G Brewer. Born to run: A grounded theory study of cheating in the online speedrunning community. Citado na pág. 6

Buchanan (2016a) Scott Buchanan. RNG, 2016a. URL https://youtu.be/MiuLeTE2MeQ. Citado na pág. 29

Buchanan (2016b) Scott Buchanan. The extent of speed conservation, 2016b. URL https://www.youtube.com/watch?v=AFTpdXZsr_s. Citado na pág. 33

Bulbapedia (2017) Bulbapedia. Tweaking, 2017. URL https://bulbapedia.bulbagarden.net/wiki/Tweaking. Citado na pág. 15

Franklin (2009) Seb Franklin. we need radical gameplay, not just radical graphics: Towards a contemporary minor practice in computer gaming. *symplokē*, 17(1-2):163–180. Citado na pág. 5

Hilburn (2017) Kaitlin Elizabeth Hilburn. *Transformative gameplay practices: speedrunning through Hyrule*. Tese de Doutorado. URL http://hdl.handle.net/2152/62782. Citado na pág. 5

Jbop (2014) Jbop. Ocarina of Time speedrun world record history, 2014. URL https://docs.google.com/spreadsheets/d/1DgefbvS3X4geDxfwdE-K6CnGvNcDwZSmiZz_u2KGDcI/htmlview?pli=1&sle=true. Citado na pág. 8

Kehne (2015) Tyler Kehne. Explanation of moat door glitch, 2015. URL https://pastebin.com/uH5GhHbp. Citado na pág. 32

Kehne *et al.* (2016) Tyler Kehne, MKDasher, sonicpacker, Snark, SilentSlayers, Gaehne D, Eru, ToT, Plush e sm64expert. N64 Super Mario 64 (jpn) "1 key", 2016. URL http://tasvideos.org/5237S.html. Citado na pág. 33

keylie (2014) keylie. SNES Chrono Trigger, 2014. URL http://tasvideos.org/4280S.html. Citado na pág. 11

Lafond (2018) Manuel Lafond. The complexity of speedrunning video games. Em Hiro Ito, Stefano Leonardi, Linda Pagli e Giuseppe Prencipe, editors, *9th International Conference on Fun with Algorithms (FUN 2018)*, volume 100 of *Leibniz International Proceedings in Informatics (LIPIcs)*, páginas 27:1–27:19, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-067-5. doi: 10.4230/LIPIcs.FUN.2018.27. URL http://drops.dagstuhl.de/opus/volltexte/2018/8818. Citado na pág. 6

LeMieux (2014) Patrick LeMieux. From nes-4021 to mosmb3. wmv: Speedrunning the serial interface. *Eludamos. Journal for Computer Game Culture*, 8(1):7–31. Citado na pág. 25

Lewis *et al.* (2010) Chris Lewis, Jim Whitehead e Noah Wardrip-Fruin. What went wrong: A taxonomy of video game bugs. Em *Proceedings of the Fifth International Conference on the Foundations of Digital Games*, FDG '10, páginas 108–115, New York, NY, USA. ACM. ISBN 978-1-60558-937-4. doi: 10.1145/1822348.1822363. URL http://doi.acm.org/10.1145/1822348.1822363. Citado na pág. 1

MrWint (2014) MrWint. GB Pokémon: Red version (USA/Europe) "save glitch", 2014. URL http://tasvideos.org/2687M.html. Citado na pág. 35

Newman (2013) James Newman. Illegal deposit: Game preservation and/as software piracy. *Convergence*, 19(1):45–61. Citado na pág. 25

Parker (2008) Felan Parker. The significance of jeep tag: On player-imposed rules in video games. *Loading...*, 2(3). Citado na pág. 6

Scully-Blaker (2016) Rainforest Scully-Blaker. Re-curating the accident: Speedrunning as community and practice. 2016. URL https://spectrum.library.concordia.ca/982159/. Citado na pág. 2, 5

Scully-Blaker (2014) Rainforest Scully-Blaker. A practiced practice: Speedrunning through space with de certeau and virilio. *Game Studies*, 14(1). Citado na pág. 6

SourceRuns (2013) SourceRuns. ABH vs. bunnyhopping comparison chart, 2013. URL https://wiki.sourceruns.org/wiki/File:Abh-vs-bhop_comparsion.png. Citado na pág. 12

Weatherton (2016) Drew Weatherton. N64 Mario Kart 64, 2016. URL http://tasvideos.org/5243S.html. Citado na pág. 28, 29

Yliluoma (2013) Joel Yliluoma. Rockman / Mega Man source code, 2013. URL https://bisqwit.iki.fi/jutut/megamansource/. Citado na pág. 10

Yliluoma (2010a) Joel Yliluoma. NES Lunar Pool, 2010a. URL http://tasvideos.org/2676S.html. Citado na pág. 36

Yliluoma (2010b) Joel Yliluoma. NES Lunar Pool "no friction", 2010b. URL http://tasvideos.org/2609S.html. Citado na pág. 36

ZeldaSpeedRuns (2013) ZeldaSpeedRuns. Superswimming, 2013. URL http://zelda.speedruns.com/twwhd/tech/superswimming. Citado na pág. 18