Deep Q Learning para ensinar inteligência artificial a jogar Asteroids

Vítor Kei Taira Tamada

Programa:
Bacharelado em Ciência da Computação

Orientador:
Prof. Dr. Denis Deratani Mauá

São Paulo, novembro de 2018

Deep Q Learning para ensinar inteligência artificial a jogar Asteroids

Esta é a versão original da monografia elaborada pelo aluno Vítor Kei Taira Tamada

Agradecimentos

Primeiramente, gostaria de agradecer aos meus pais, Valéria e Luis Edmundo, pelo apoio ininterrupto durante toda a graduação e pelas inúmeras oportunidades que me deram para ser livre em seguir o caminho que mais me desse felicidade na vida. Eu não estaria onde estou agora sem vocês.

Um agradecimento especial à Mayte Mirio, que aguentou todas as noites que virei programando jogos e nunca parou de me apoiar. Seu amor e carinho me deram forças para enfrentar inúmeros desafios na vida e me trouxe mais felicidade do que eu jamais conseguiria imaginar.

Quero agradecer a todos meus amigos que sempre me trouxeram alegria, não importando a situação. Felipe e Sabrina, obrigado pela disposição de baixar e testar meus joguinhos consideráveis vezes, me ajudando com críticas e me dando a satisfação de ver pessoas se divertindo com meus projetos. Rodrigo, obrigado pelas eternas noites de jogatinas, as inúmeras risadas e uma amizade eviterna. Yan e Renato, sem vocês eu não seria o aspirante desenvolvedor de jogos indies que sou hoje. Obrigado por terem iniciado essa chama em meu coração, pelas longas Game Jams e pelas tardes de jogatinas. Espero que parte dessa chama continue em vossos corações e também nunca se apague.

E a todos meus outros amigos, que eu poderia listar perpetuamente, que por horas compartilharam de seu tempo comigo em atividades lúdicas, nos divertindo até altas horas, obrigado por me lembrar do verdadeiro propósito para o qual eu faço jogos.

A todos os professores que tiveram paciência e dedicação em transmitir conhecimento para mim e meus colegas, obrigado por terem aturado esse aluno desorganizado e atrapalhado por tantos anos. Espero um dia poder repassar tudo que aprendi à frente, e que minha curiosidade pelo mundo da computação e desenvolvimento de jogos que vocês iniciaram nunca se acabe.

Por último, nada disso seria possível sem o grupo extracurricular **UspGameDev** e todos seus membros, passados e presentes, que me acolheram desde o primeiro ano da faculdade e me ensinaram muito mais do que eu esperaria. Um grande agradecimento ao doutorando Wilson Kazuo Mizutani, fundador e membro ativo do grupo, que desde meu primeiro ano nunca ignorou qualquer dúvida ou questionamento que tive, seja sobre programação, desenvolvimento de jogos, *game design* ou sobre a vida acadêmica.

A todas essas pessoas e tantas outras que marcaram minha vida, tanto dentro quanto fora da faculdade, eu dedico *PsyChO: The Ball* a vocês. Obrigado por tudo.

Resumo

FONSECA, R. L. **PsyChO: The Ball**. Trabalho de Conclusão de Curso - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2017.

Desenvolvimento de jogos é uma área da computação repleta de desafios. O jogo digital PsyChO: The Ball, um *Top Down Shooter* psicodélico minimalista, foi produzido, desde seu início, utilizando apenas software livre. Feito no arcabouço *LÖVE* com a linguagem Lua, o processo de desenvolvimento passou por todas as etapas necessárias para a produção de um jogo. Ele foi inspirado no Game Design dos jogos digitais *Hotline Miami*, *Touhou* e *Hexagon* e, entre suas melhores características, temos sua jogabilidade frenética e efeitos especiais *juicy*.

Palavras-chave: desenvolvimento de jogos, game design, juiciness, top down shooter.

Sumário

Capítulo 1

Introdução

Desenvolvimento de jogos, ou *game development*, é uma área de estudos interdisciplinar onde podemos aplicar conhecimentos de Arte, Música, *Game Design* e, especialmente em jogos digitais, Ciência da Computação. Além disso, não é uma área na qual faltem desafios e problemas a serem solucionados. A produção de um jogo requer a aplicação direta de muitos conceitos e práticas ensinadas durante o curso em Ciência da Computação do Instituto de Matemática e Estatística da Universidade de São Paulo.

Em novembro de 2009 foi fundado o **UspGameDev**, o grupo extracurricular da Universidade de São Paulo dedicado a fazer a ponte entre interessados em jogos e estudantes da faculdade, em especial estudantes da computação. Aberto à toda comunidade *Uspiana*, a *UspGameDev* ou *UGD* sempre ficou de portas abertas para qualquer aluno que desejasse aprender mais sobre desenvolvimento de *games*, tendo a chance de se reunir com outras pessoas e produzir seus próprios jogos, sejam eles digitais ou analógicos.

Em 2013, no meu primeiro ano de faculdade, entrei na *UGD* e, junto de um colega, fiz meu primeiro jogo virtual: *PsyChObALL*, um *top-down shooter psicodélico*. Foi o primeiro "grande"jogo que produzi durante a faculdade e isso me inspirou a continuar estudando na área de desenvolvimento de jogos, sempre conectado à *UspGameDev*.

Foi com essa mentalidade que decidi, no meio da graduação e com início na matéria *Atividade Curricular em Cultura e Extensão*, produzir um *remake* (termo análogo a uma "refilmagem", aplicado a jogos) de *PsyChObALL* chamado de *PsyChO: The Ball*, combinando todo o conhecimento adquirido durante os anos de estudos em computação e *design* de jogos.

1.1 Motivação e Objetivos

O objetivo central desse trabalho final de formatura é aplicar todo o conhecimento aprendido durante a faculdade, focado na área de desenvolvimento de jogos (ou *"game dev"*) e utilizar esse projeto como mais uma fonte de estudos e aprendizados. Neste sentido, me interessei em percorrer todas as etapas do desenvolvimento de um jogo digital.

Primeiramente é necessária a discussão de *balanço* sobre o jogo original *PsyChObALL*, analisando seus pontos positivos e negativos. A segunda etapa seria construir uma base sólida para rodar o jogo, utilizando meus conhecimentos para criar bibliotecas e um ambiente apropriado para sua construção. A terceira etapa seria o desenvolvimento do jogo em si, utilizando como referência o jogo original. Por último, seria a etapa de testar com usuários e receber *feedback*, podendo assim voltar à etapa 3 de desenvolvimento do jogo, melhorando o que for necessário, assim repetindo o ciclo.

É esperado que depois de várias iterações, se chegue em um protótipo jogável de *PsyChO: The Ball*, para que futuramente seja possível disponibilizar o jogo em alguma plataforma de distribuição de jogos

digitais.

Capítulo 2

Fundamentos

Para se criar e treinar uma inteligência artificial, diversos arcabouçous são necessários. Por um lado, existe a parte teórica e matemática na qual a inteligência se baseia para aprender. Por outro, do lado computacional, existem as bibliotecas que auxiliam no desenvolvimento, efetuando as contas necessárias e, neste trabalho em particular, emulando o jogo que serve de ambiente para o aprendizado. Este capítulo tem o intuito de familiarizar o leitor com a teoria e as ferramentas utilizadas no treinamento da inteligência artificial deste trabalho.

2.1 *Asteroids*

Asteroids é um jogo de fliperama do gênero shooter (jogo eletrônico de tiro) lançado em novembro de 1979 pela então desenvolvedora de jogos eletrônicos Atari Inc, atualmente conhecida apenas como Atari. O jogo foi inspirado por *Spacewar!*, *Computer Space*, *Space Invaders*, e *Cosmos*, sendo este último um jogo não finalizado.

O jogador controla uma nave espacial que se encontra em um campo de asteróides e precisa atirar para destruir todos enquanto evita colisões com os mesmos. A dificuldade aumenta conforme o número de asteróides na tela aumenta. Diversas versões deste jogo foram criadas ao longo dos anos. Além das diferenças gráficas, as mudanças incluem naves espaciais inimigas que atiram contra o jogador, e tamanhos e formatos diferentes que os asteróides podem ter. *Asteroids* é considerado um dos primeiros grandes sucessos da era de ouro dos jogos de fliperama, época em que os jogos eletrônicos se estabeleceram como uma força dominante na cultura popular.

Para este trabalho, *Asteroids* foi emulado utilizando a plataforma Gym-Retro da companhia de pesquisas de inteligência artificial OpenAI. Todas as informações sobre o jogo apresentadas a seguir referem-se a tal versão. Para o aprendizado da inteligência artificial, não há naves inimigas e os asteróides podem assumir três tamanhos e três formatos distintos. Os tamanhos são grande (inicial), médio e pequeno, enquanto os formatos são consideravelmente parecidos. Há cinco comandos disponíveis: mover-se para frente, girar a nave no sentido horário, girar a nave no sentido anti-horário, atirar para frente, e entrar no hiper espaço. Mover-se para frente e girar no sentido horário ou anti-horário são as principais formas de movimento disponíveis ao jogador, e atirar serve para destruir os asteróides. Mover-se no hiper espaço consiste em fazer a nave desaparecer e, depois de alguns instantes, reaparecer em um local aleatório da tela. Há o risco de reaparecer em cima de um asteróide e, com isso, ter a nave destruída e perder uma vida. A nave possui aceleração e desaceleração - ou seja, inércia. Em outras palavras, mesmo que o jogador deixe de pressionar o botão de mover-se para frente, ele continuará em movimento por um curto período de tempo antes de parar por completo. Isso gera um grau a mais de complexidade, pois faz com que

manobras de esquiva e curvas sejam mais difíceis de serem devidamente executadas.

2.2 Gym-Retro

Gym-Retro é uma plataforma para pesquisa de aprendizado por reforços e generalização em jogos desenvolvida e mantida pela empresa de pesquisas em inteligência artificial OpenAI. O lançamento mais recente inclui, como ambientes para o desenvolvimento de IA, jogos do Sega Genesis, Sega Master System, Nintendo Entertainment System (NES), Super Nintendo Entertainment System (SNES) e Nintendo Game Boy, além de suporte preliminar para Sega Game Gear, Nintendo Game Boy Color, Nintendo Game Boy Advance e NEC TurboGrafx. Em qualquer um desses consoles, a ROM (Read Only Memory) do jogo é necessária. Apesar de não ter sido utilizada neste trabalho, a plataforma disponibiliza uma ferramente que permite criar save states (salvar um estado a partir do qual é possível continuar o jogo), encontrar locais da memória, criar cenarios para o agente resolver, gravar e passar arquivos de vídeo, dentre outras funcionalidades.

Gym-Retro baseia-se na ferramenta Gym, desenvolvida e mantida pela OpenAI, que também tem como objetivo pesquisas em aprendizado por reforço, mas não apenas para jogos. Esta ferramenta foi utilizada por ter suporte para desenvolvimento de aprendizado para o jogo *Asteroids* e ser de fácil uso. A plataforma permite a entrada de oito ações diferentes: UP, DOWN, LEFT, RIGHT, BUTTON, SELECT, RESET, null, sendo que a ação realiada por cada botão varia de acordo com o jogo. Como descrito anteriormente, *Asteroids* utiliza apenas cinco deles: UP (mover-se para frente), DOWN (mover-se no hiper espaço), RIGHT (girar no sentido horário), LEFT (mover-se no sentido anti-horário), e BUTTON (atirar). Os demais botões (SELECT e RESET) possuem funções relacioandas ao sistema e não ao jogo, e null é não realizar nenhuma ação.

2.3 TensorFlow

TensorFlow é um arcabouço de código aberto para computações numéricas de alta performance, desenvolvido e mantido pela Google. Seu núcleo de computação numérica flexível permite o uso da biblioteca em diversos campos da ciência. Oferece, em particular, grande suporte a aprendizado de máquina e deep learning. Esta ferramenta foi utilizada por oferecer uma API em Python estável, ter grande suporte, comunidade ativa, e ser de código aberto. Apesar de não ter sido utilizado, esta biblioteca também possui uma ferramenta de visualização de dados chamada TensorBoard.

2.4 Inteligência artificial (IA)

Inteligência artificial (IA) é um dos campos mais recentes de ciência e engenharia, tendo trabalhos no assunto sendo iniciados pouco depois da Segunda Guerra Mundial. Atualmente, ela é composta por diversos campos menores de estudo, podendo ser mais genérico, como aprendizado e percepção, até mais específico, como a capacidade de jogar um jogo, provar teoremas matemáticas, ou dirigir um carro em uma via movimentada. No livro Artificial Intelligence: A Morden Approach, de Stuart Jonathan Russel e Peter Norvig, oito definições são apresentadas em uma tabela de duas linhas por duas colunas. A linha de cima define processo de pensamento (*thought process*) e raciocínio (*reasoning*), enquanto a linha de baixo define comportamento (*behaviour*). Além disso, a coluna da esquerda mede o grau de fidelidade da inteligência quando comparado com performance humana, enquanto a da direita mede a racionalidade da performance - ou seja, se toma a ação "correta"dado o que o sistema sabe.

Pensando como um humano	**Pensando racionalmente**
O empolgante novo esforço de fazer computadores pensarem, serem máquinas com mentes, no sentido completo e literal da expressão (Haugeland, 1986)	*O estudo das faculdades mentais através de modelos computacionais* (Charniak & McDermott, 1985)
[A automação de] atividades que são associadas ao pensamento humano, como resolução de problemas, tomada de decisão, aprendizado, ... (Hellman, 1978)	*O estudo das computações que tornam possível a percepção, razão, e ação* (Winston, 1992)
Agindo como um humano	**Agindo racionalmente**
A arte de criar máquinas capazes de realizar funções que requerem inteligência quando feitas por pessoas (Kurzweil, 1990)	*Inteligência computacional é o estudo do design de agentes inteligentes* (Poole *et at*, 1998)
O estudo de como fazer os computadores fazerem coisas que, no momento, pessoas fazem melhor (Rich and Knight, 1991)	*IA... está relacionada a comportamento inteligente em objetos* (Nilsson, 1998)

Em linhas gerais, as definições da coluna da esquerda dizem respeito a uma inteligência artificial que se pareça com um humano, enquanto as da direita sobre uma inteligência artificial que toma ações visando estar correta e a atingir o melhor resultado possível. Este trabalho terá um foco maior na categoria "**Agindo racionalmente**", pois as ações tomadas pelo agente terão como objetivo o retorno da maior recompensa possível.

2.4.1 Aprendizagem por reforço

Aprendizagem por reforço (*reinforcement learning*) é uma técnica de aprendizado de inteligência artificial e uma das bases da utilizada neste trabalho. Para domínios mais simples, como Jogo da velha, é possível determinar qual a ação com maior recompensa esperada para cada estado - ou seja, a ação com maior probabilidade de vitória. Conforme o domínio se torna mais complexo, fazer esse mapeamento se torna inviável por conta da quantidade de estados que precisam ser armazenado, como é o caso do jogo *Asteroids*. Além disso, é comum haver situações em que não é possível determinar qual ação retornará a maior recompensa. Nesses casos, é mais viável criar e treinar um agente que aprenda a se comportar no ambiente em que está inserido do que informar se cada uma de suas ações em cada um dos estados possíveis é boa ou ruim.

Essa abordagem é conhecida como **Processo de Decisão de Markov** (*Markov Decision Process* (*MDP*)) para ambientes desconhecidos.

Processo de Decisão de Markov

Em um MDP padrão, a probabilidade de se chegar em um estado S' dado que o agente se encontra no estado S depende apenas da ação A tomada nesse estado s, o que caracteriza a **propriedade Markoviana**, existe um modelo probabilístico que caracteriza essa transição, dado por $P(S'|S, A)$; todos

os estados do ambiente e todas as ações que o agente pode tomar em cada estado são conhecidas; e a recompensa é imediatamente recebida após cada ação ser tomada.

As probabilidades de o agente tomar cada ação em um dado espaço são definidas por uma política π. A qualidade de uma política é medida por sua *utilidade esperada*, e a política ótima é denotada por π^*. Para calcular π^*, utiliza-se um algoritmo de iteração de valor (*value iteration*), que computa a utilidade esperada do estado atual: começando a partir de um estado arbitrário S tal que seu valor esperado é $V(S)$, aplica-se a equação de Bellman (*Bellman update* ou *Bellman equation*) até haver convergência de $V(S)$, que será denotado por $V^*(S)$. Esse $V^*(S)$ é usado para calcular a política ótima $\pi^*(s)$.

Seja i a iteração atual, S o estado atual, S' o estado futuro, A a ação tomada no estado atual, $R(S, A, S')$ a recompensa pela transição do estado S para o estado S' por tomar a ação A, e γ o valor de desconto (valor entre 0 e 1 tal que determina a importância de recompensas futuras para o agente), temos que:

Equação de Bellman:

$$V^{(i)}(S) = \max_A \sum_{S'} P(S'|S, A)[R(S, A, S') + \gamma V^{(i-1)}(S')] \tag{2.1}$$

Política gulosa para função valor ótima:

$$\pi^*(s) = \operatorname*{argmax}_A \sum_{S'} P(S'|S, A)[R(S, A, S') + \gamma V^*(S')] \tag{2.2}$$

Entretanto, essas fórmulas são aplicáveis somente quando as funções de reforço R e de probabilidade de transição P são conhecidas, que não é o caso do jogo *Asteroids*. Para lidar com esse problema, foi adotado o uso de **Q-learning**.

Q-learning

Quando não se conhece as probabilidades de transição, informação necessária para se obter a função valor pela equação de Bellman, é possível estimar $V(S)$ a partir de observações feitas sobre o ambiente. Logo, o problema deixa de ser tentar encontrar P e passa a ser como extrair a política do agente de uma função valor estimada.

Seja $Q^*(S, A)$ a função Q-valor que expressa a recompensa esperada por começar no estado S, tomar a ação A e continuar de maneira ótima. $Q^*(S, A)$ é uma parte da política gulosa para função valor ótima e é dada por:

$$\begin{aligned} Q^*(S, A) &= \sum_{S'} P(S'|S, A)[R(S, A, S') + \gamma V^*(S')] \\ &= \sum_{S'} P(S'|S, A)[R(S, A, S') + \gamma \max_{A'} Q^*(S', A')] \end{aligned} \tag{2.3}$$

Logo, substituindo 2.3 em 2.2, temos que a política gulosa ótima para a função Q-valor ótima é dada por:

$$\pi^*(S) = \operatorname*{argmax}_A Q^*(S, A) \tag{2.4}$$

O próximo passo será entender como atualizar a função Q-valor.

Supondo que o agente se encontra no estado S e toma a ação A, que causa uma transição no ambiente para o estado S' e gera uma recompensa $R(S, A, S')$, como computar $Q^{(i+1)}(S, A)$ baseado em $Q^{(i)}(S, A)$ e em $R(S, A, S')$, sendo i o momento atual? Para responder a essa pergunta, duas restrições precisam ser feitas: $Q^{(i+1)}(S, A)$ deve obedecer, pelo menos de forma aproximada, a equação de Bellman, e não deve

ser muito diferente de $Q^{(i)}(S, A)$, dado que são médias de recompensas. A seguinte equação responde essa questão.

Seja α a taxa de aprendizado (valor entre 0 e 1 que determina o quão importantes informações novas são em relação ao conhecimento que o agente possui),

$$\begin{aligned} Q^{(i+1)}(S, A) &= (1 - \alpha)Q^{(i)}(S, A) + \alpha[R(S, A, S') + \gamma \max_{A'} Q^{(i)}(S', A')] \\ &= Q^{(i)}(S, A) + \alpha[R(S, A, S') + \gamma \max_{A'} Q^{(i)}(S', A') - Q^{(i)}(S, A)] \end{aligned} \tag{2.5}$$

A convergência de $Q^{(i)}(S, A)$ em $Q^*(S, A)$ é garantida mesmo que o agente aja de forma subótima contanto que o ambiente seja um MDP, a taxa de aprendizado seja manipulada corretamente, e se a exploração não ignorar alguns estados e ações por completo - ou seja, raramente. Mesmo que as condições sejam satisfeitas, a convergência provavelmente será demasiadamente lenta. Entretanto, é interessante analisar os problemas levantados pela segunda e pela terceira condição que garantem a convergência e maneiras de solucioná-los.

Se a **taxa de aprendizado** for muito alta (próxima de 1), a atualização do aprendizado se torna instável. Por outro lado, se for muito baixa (próxima de 0), a convergência se torna lenta. Uma solução possível para essa questão é utilizar valores que mudam de acordo com o estado: utilizar valores mais baixos em estados que já foram visitados muitas vezes, pois o agente já terá uma boa noção da qualidade de cada ação possível, então há pouco que aprender; e utilizar valores mais altos em estados raramente visitados, pois o agente precisa aprender melhor sobre o estado.

Uma vez que a política é gulosa, o agente sempre tomará a ação que retorne a maior recompensa imediata. Isso é bom somente se todas as recompensas possíveis para aquele estados são conhecidas. Porém, se houver ações não exploradas, o agente pode perder uma recompensa maior do que ele já conhece apenas porque não está ciente de seu valor. Essa situação caracteriza o dilema *Exploration versus Exploitation*: é melhor tomar a ação que retorna a maior recompensa ou buscar uma melhor? Da mesma forma que na taxa de aprendizado, uma forma de contornar esse problema é mudar a probabilidade de decidir explorar o ambiente (*explore*) de acordo com a situação. Conforme o mundo é descoberto, se torna cada vez mais interessante agir de forma gulosa (*exploit*) do que explorar em estados muito visitado, e vice-versa em estados pouco visitados. Esse comportamento pode ser definido por uma função de exploração (*exploration function*).

Outro problema enfrentado por Q-learning é o de generalização. A política $\pi^*(S)$ determina a melhor ação a se tomar em cada estado. Logo, utiliza-se uma tabela para armazenar todas essas escolhas. Porém, como mencionado anteriormente, isso se torna inviável para espaços de estado muito grandes. Portanto, a solução é generalizar o aprendizado de um estado para o outro: se o agente sabe se comportar em um pequeno conjunto de estados, o ideal é ele saber o que fazer em um estado desconhecido contanto que seja parecido com um já aprendido. Em outras palavras, o agente aprende propriedades (*features*) dos estados ao invés dos estados propriamente ditos, e toma decisões a partir dessas informações. Essa forma de fazer escolhar é chamada de *approximate Q-learning*.

Approximate Q-learning

Para lidar com o enorme espaço de estados que alguns ambientes possuem, o agente armazena e aprende apenas algumas propriedades, que são funções de valor real, para tomar as decisões. Tais informações são armazenadas em um vetor e cada elemento desse vetor recebe um peso que determina a respectiva importância para que escolhas sejam feitas. Ou seja, a função Q-valor é representada por uma combinação linear das propriedades.

$$Q(S, A) = \omega_1 f_1(S, A) + \omega_2 f_2(S, A) + ... + \omega_n f_n(S, A) \tag{2.6}$$

Como o $V(S')$ é o valor esperado e $Q(S, A)$ é o valor previsto, a atualização pode ser interpretada como ajustar o valor do Q-valor pela diferença desses dois valores. Além disso, como o *approximate Q-learning* avalia características, apenas os pesos precisam ser atualizados:

$$\omega_k^{(i+1)}(S, A) = \omega_k^{(i)}(S, A) + \alpha[R(S, A, S') + \gamma V(S') - Q^{(i)}(S, A)] f_k(S, A), k = 1, 2, ..., n \tag{2.7}$$

Duas grandes vantagens de representar o Q-valor como uma combinação linear são evitar *overfitting* (a IA aprender tanto com o conjunto de treinamento que não consegue tomar decisões que diferem demais dele), e ser matematicamente conveniente, ter maneiras convenientes de calcular erro e funções que generalizem as decisões.

2.4.2 *Deep learning*

Game Design é o ato de decidir como um jogo deve ser.[1]

```
1  while (true)
2  {
3    processInput();
4    update();
5    render();
6  }
```

2.5 Técnicas

Apesar de grande parte do desenvolvimento de *PsyChO: The Ball* ter sido feita apenas por uma pessoa, foi muito benéfica a utilização de sistemas de versionamento que ajudam tanto na centralização de todo o código do jogo, facilitando na transição entre ambientes de trabalho, quanto na possibilidade de guardar versões antigas e resgatar códigos passados quando necessário. Durante todo o desenvolvimento de *PsyChO: The Ball* foi utilizado um sistema de controle de versão para código e nele se encontram todas as versões de lançamento do jogo.

Uma boa prática na produção de um jogo, seja esse comercial ou um projeto pessoal, é a de lançamentos, ou *releases*, constantes. O propósito disto é ter, periodicamente, *releases* de versões estáveis e jogáveis do jogo, de forma que outras pessoas possam jogar e dar *feedback* frequente, assim ajudando no encaminhamento do projeto. Essa técnica é muito comum em metodologias ágeis de desenvolvimento de software e essa linha de pensamento foi o que guiou todo o desenvolvimento de *PsyChO: The Ball*.

Inicialmente foi utilizado o sistema *Kanban*, uma abordagem moderna de conceitos ágeis muito comum em empresas ou grupos de desenvolvimento de software. Kleber Bernardo, especialista em métodos ágeis, descreve bem a metodologia em um artigo no site Cultura Ágil **?**:

> *O Kanban lhe ajuda a assimilar e controlar o progresso de suas tarefas de forma visual. É, normalmente, utilizado um quadro branco com alguns pequenos papéis (Post-it) colados, esses papéis representam as suas tarefas, ao termino de cada tarefa o papel é puxado para a etapa*

[1] Tradução livre feita pelo autor

seguinte até que a mesma seja finalizada. Ao olhar para um quadro Kanban é fácil enxergar como o trabalho seu e de sua equipe fluem, permitindo não só comunicar o status, mas também dar e receber feedbacks.

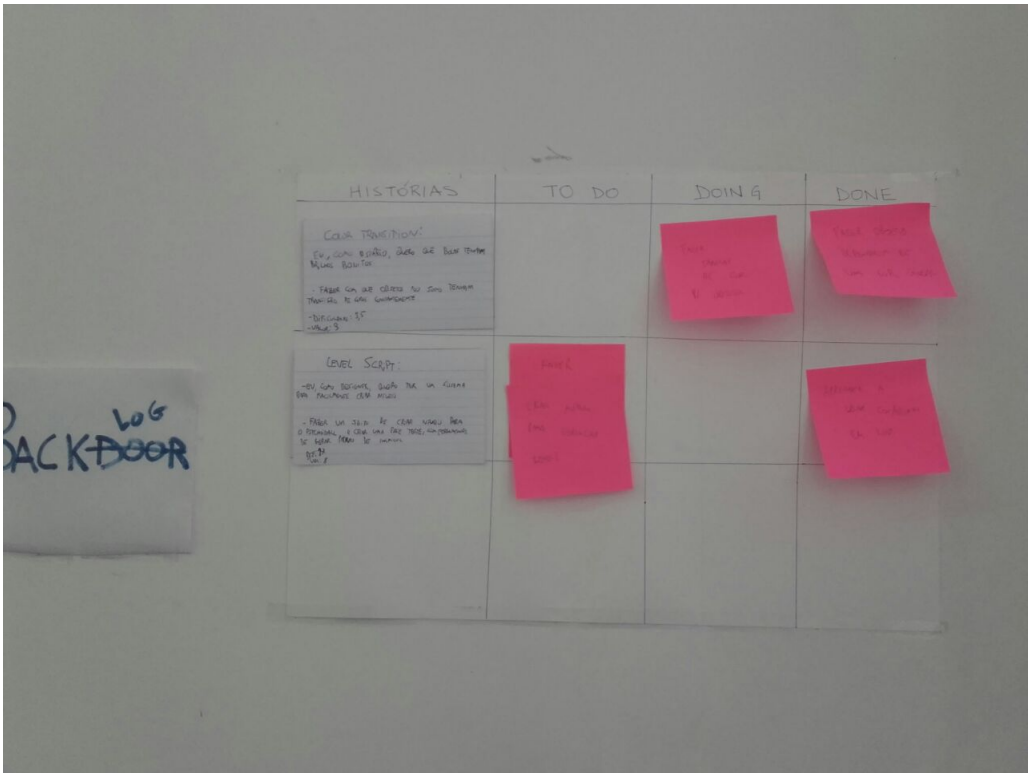

Figura 2.1: Kanban utilizado durante o desenvolvimento de *PsyChO: The Ball*

Entretanto, após alguns meses utilizando essa técnica, o tempo gasto em sua manutenção acabou se tornando mais trabalhoso do que produtivo para o projeto e seu uso foi decaindo. Desta forma o *Kanban* foi substituido por várias pequenas técnicas ágeis, como lançamentos frequentes do jogo, *sprints* (onde era determinado um prazo de uma ou duas semanas para terminar funcionalidades no jogo) e constantes re-avaliações com consulta de *feedback*.

Por último, uma técnica muito presente no desenvolvimento do jogo foi a utilização de documentação para marcar mudanças e melhorias no projeto. Os dois documento mais importantes são o *DEVLOG*[2] e *CHANGELOG*[3]. O *DEVLOG* foi um necessário para mostrar o progresso e tempo gasto no projeto, enquanto este fazia parte da disciplina *MAC0214 Atividade Curricular em Cultura e Extensão*. Nele existem várias entradas detalhando atividades feitas no jogo durante o desenvolvimento. O *CHANGELOG* por outro lado serve para marcar lançamentos do jogo, descrevendo *features* novos adicionados em cada *sprint* feito. Ambos documentos foram de grande utilidade no início do desenvolvimento de *PsyChO: The Ball*, pois exibem concretamente o fluxo de progresso na criação do jogo.

2.6 Ferramentas

Uma das escolhas feitas para o desenvolvimento de *PsyChO: The Ball* foi a utilização exclusiva de *Software Livre*. Essa decisão se deu tanto pela liberdade que essas ferramentas permitem em sua utilização, quanto pelo objetivo de incentivar mais o uso desse tipo de software, mostrando que é tão eficiente quanto

[2]https://github.com/uspgamedev/Project-Telos/blob/dev/docs/DEVLOG.md
[3]https://github.com/uspgamedev/Project-Telos/blob/dev/docs/CHANGELOG.md

o uso de software proprietário. Assim, como critério de escolha para cada área de desenvolvimento do jogo, foi determinado algum software livre que tenha uma comunidade ativa (para a solução de problemas e disponibilidade de bibliotecas), constante atualização de suas funcionalidades, versões estáveis e uma recomendação de seu uso no desenvolvimento de jogos.

O arcabouço utilizado no desenvolvimento de *PsyChO: The Ball* foi a *Love2d* ou *LÖVE*[4], uma *framework* gratuita que utiliza a linguagem *Lua*[5].

Como sistema de controle de versão foi utilizado *Git*[6] através da plataforma de uso grátis *Github*[7].

Para a produção de música foi utilizado *LMMS*[8], um software livre gratuito para manipulação e criação de sons.

Por último, para manipulação de imagens, foi utilizada a ferramenta *Gimp*[9].

Figura 2.2: Da esquerda pra direita, ícones de *LÖVE*, *Lua*, *Git*, *Github*, *LMMS* e *Gimp*

Dentre essas ferramentas, vale ressaltar um ponto muito interessante do arcabouço *LÖVE*: seu *game-loop* tem o diferencial de deslocar todas as verificações de *input* do usuário como *callbacks assíncronos*. Estes são funções que a própria *framework* vai chamar quando ocorrer algum *input* do usuário, como quando ele pressionar um botão do teclado ou mover o mouse. Através de subrotinas, o arcabouço está constantemente aguardando sinais que ativem essas funções e assim o desenvolvedor não precisa se preocupar em lidar com otimização ou código confuso na hora de esperar e tratar comandos dados pelo usuário. Isso foi uma grande vantagem que determinou a escolha dessa ferramenta na produção de *PsyChO: The Ball*.

[4] https://love2d.org/
[5] https://www.lua.org/portugues.html
[6] https://git-scm.com/
[7] https://github.com/
[8] https://lmms.io/
[9] https://www.gimp.org/

Capítulo 3

Proposta

Durante o desenvolvimento de um jogo, seja ele digital ou analógico, é de extrema importância que todos os detalhes (como mecânicas, estética ou até mesmo público alvo) estejam bem documentados em algum lugar para que todos que participam da criação do jogo estejam em sintonia. Desta forma foi criado o *Game Design Document*, ou *GDD*.

O objetivo do *GDD* é servir como um guia para artistas, desenvolvedores ou músicos em um jogo. Entretanto, ele não é um documento estático, pois um jogo nunca está completamente definido desde seu início. Características ou detalhes do jogo vão sendo acrescentados ou removidos conforme vão se mostrando benéficos ou prejudiciais para a experiência desejada do projeto final. Desta forma, fica a cargo do *Game Designer* mantê-lo sempre atualizado.

Com isso vamos descrever um breve *GDD* do *PsyChO: The Ball* com todos os conceitos e estéticas que foram planejados até o momento.

3.1 Temática

PsyChO: The Ball é um *Top-Down Shooter* psicodélico, minimalista e frenético. Ele tem grandes influências de outros grandes jogos do mesmo gênero como ***Hotline Miami?*** ou jogos da série ***Touhou Project?***. Além disso, foi pensado para ser um jogo difícil, desafiando o jogador constantemente. É esperado que ele fracasse bastante, mas melhore em cada partida nova, conforme vá melhorando seus reflexos.

O estilo psicodélico e minimalista foi inspirado no jogo ***Hexagon?***. Quase todos elementos de *PsyChO: The Ball* são composições de círculos com cores vibrantes que mudam ao longo do tempo. O jogador, desta forma, não fica sobrecarregado de informação na tela e pode focar sua atenção em desviar de projeteis e destruir inimigos.

O jogo foi desenvolvido para ser jogado com um teclado e mouse, porém com controles minimalistas para que, no futuro, tenha suporte a *joysticks*.

3.2 Mecânicas

PsyChO: The Ball segue a fórmula básica de jogos *Top-Down Shooters*: o jogador controla um círculo colorido, chamado de *PsyChO*, que pode se mover livremente no espaço da tela. Desta forma, o jogador consegue se esquivar de golpes inimigos ou obstáculos durante o jogo.

Para interagir com os inimigos, *PsyChO* possui dois golpes: atirar (ataque ofensivo) ou usar um *ULTRABLAST* (ataque defensivo).

Atirar é o jeito normal de atacar e destruir inimigos. Segurando um botão e mirando com o mouse, o jogador escolhe onde vai atirar projéteis, que andam em linha reta, até colidir com um inimigo ou com os cantos da tela de jogo. Os projéteis são bem pequenos e velozes, de forma que uma boa precisão seja necessária para acertar um alvo. Atirar não gasta nenhum recurso e pode ser utilizado a qualquer momento.

ULTRABLAST é um golpe extremamente poderoso, porém limitado. Ao pressionar um botão especial, *PsyChO* cria um anel em sua volta que se expande rapidamente. Esse anel consegue destruir um número grande de inimigos antes de desaparecer, sendo muito útil para escapar de situações de extremo perigo. Além disso, ao utilizar o golpe, *PsyChO* fica invulnerável por um curto período de tempo, dando uma folga para o jogador se reposicionar estrategicamente.

Por último, o jogador pode segurar um botão para entrar no modo *Focus*. Neste modo, o tamanho do jogador diminui ligeiramente e sua velocidade de movimento é drasticamente reduzida. Também neste modo o jogador pode fazer movimentos mais precisos e delicados para desviar de obstáculos ou outros perigos.

O número reduzido de mecânicas (em comparação a outros jogos do mesmo gênero) foi uma escolha proposital no *design*. Com um arsenal limitado de ações para fazer, se torna um desafio tanto para o jogador (que vai ter que masterizar cada detalhe e nuância das ações disponíveis para enfrentar os obstáculos do jogo), quanto para o *game designer* (que vai ter de pensar em interações e níveis inovadores que aprofundem o uso de cada mecânica para manter o interesse do usuário).

3.3 Gameplay

PsyChO: The Ball segue o modelo clássico de estruturação encontrado em vídeo-games: níveis. O jogo será composto de 5 níveis sequenciais, cada um apresentando novos desafios e uma curva crescente de dificuldade. Ao final de cada nível, o jogador enfrenta um *chefão*, sendo este um inimigo bem mais forte e que resiste muito mais aos golpes de *PsyChO*, em comparação com os demais. Se conseguir derrotá-lo, avançará para o nível seguinte. Se sobreviver ao chefão final do último nível, o jogo termina em sucesso.

O jogador começa cada partida do jogo no primeiro nível e com 10 vidas, necessitando enfrentar todos os desafios que cada um deles fornece. Estes desafios podem ser inimigos que atiram projéteis em direção ao jogador, que se multiplicam, ou até mesmo invencíveis (de forma que o jogador precisará evitá-los até desaparecerem). Porém, *PsyChO* é extremamente frágil e qualquer objeto inimigo que o atinja é o suficiente para destruí-lo e lhe tirar uma vida. Cabe ao jogador aprender a utilizar seus golpes eficientemente e desvendar quando é mais prudente atacar ou fugir dos inimigos. Se a qualquer momento o jogador perde todas suas vidas, o jogo termina em fracasso e será preciso recomeçar a partir do último nível.

PsyChO irá, normalmente, atirar para destruir inimigos. Porém, quando estiver em apuros, se o jogador tiver reflexos rápidos poderá utilizar seu golpe especial *ULTRABLAST* para matar inimigos próximos e ficar invunerável temporariamente. *PsyChO* começa cada vida com 2 *ULTRABLASTs* e precisará destruir inimigos para ganhar mais destes.

Ao destruir inimigos, o jogador ganha pontos. Além do uso comum em jogos da utilização de pontos para recompensar o jogador e servir de métrica para seu progresso no jogo, ao acumular pontos, o jogador ganha vidas e *ULTRABLASTs* para gastar. Desta forma, o jogo influencia o jogador a jogar ofensivamente para chegar mais longe.

Por último, cada nível é dividido em seções. Isto ajuda no *level design* do jogo de forma a organizar o fluxo do *gameplay* e melhor apresentar elementos novos. De maneira geral, as primeiras seções de um nível servem para apresentar inimigos ou conceitos novos que estão relacionados entre si. A penúltima seção

junta todos os elementos novos apresentados, que deveriam se encaixar organicamente, já que seguem um tema em comum. A seção final é reservada para o *chefão* do nível, que vai testar todo conhecimento acumulado pelo jogador até então.

3.4 Recursos Audiovisuais

Os recursos audiovisuais do jogo foram pensados para complementar a jogabilidade minimalista e psicodélica presente. Todos objetos em *PsyCho: The Ball* são composições geométricas abstratas, com o intuito de facilitar a associação do jogador com o que é benéfico ou prejudical. Cada tipo diferente de inimigo segue um padrão próprio de cores. *PsyChO* permuta entre muitas cores vibrantes. Além disso o fundo do jogo fica transitando entre cores com baixa saturação para não distrair o jogador.

Para complementar a experiência do jogo, a trilha sonora original é baseada em gêneros eletrônicos e psicodélicos, que combinam muito bem com a jogabilidade frenética e as cores pulsantes. Os efeitos sonoros possuem vários efeitos de ressonância e eco, típicos em jogos com a mesma temática.

Para ficar mais emocionante, *PsyChO: The Ball* é repleto de efeitos de partículas e explosões para criar a ilusão de que todas ações do jogador tem um impacto enorme sobre o jogo.

Capítulo 4

Desenvolvimento

PsyChO: The Ball teve seu início em 2013 como *PsyChObALL*. Durante os primeiros meses, o projeto servia como um meio prático de se aprender técnicas de programação e desenvolvimento de jogos e, aos poucos, foi tomando proporções maiores até ter seu primeiro lançamento em agosto de 2013. O jogo continuou em desenvolvimento até meados de 2014, mas lentamente começou a receber menos atualizações conforme seus criadores foram focando em outros projetos.

Foi somente no segundo semestre de 2015 que ele ressurgiu como *PsyChO: The Ball*, na disciplina **MAC0214 - Atividade Curricular em Cultura e Extensão**. O objetivo do projeto era revitalizar o *PsyChObALL* original, fazendo tudo do início e utilizando todo conhecimento acumulado durante a graduação. O fim de 2016 teve como resultado um protótipo bem elaborado do jogo, com todas mecânicas originais do *PsyChObALL* implementadas e melhoradas.

O desenvolvimento do jogo passou por várias fases. Seguem neste capítulo algumas das infraestruturas mais importantes desenvolvidas para o projeto.

4.1 STEAMING

Antes de começar o desenvolvimento direto de *PsyChO: The Ball*, os primeiros meses foram direcionados em criar um template de jogos para o arcabouço *Love2D*. Esse template se tornou o **STEAMING** (*Simple TEmplAte for MakINg Games*) e todo o código do jogo foi construído em cima dele.

STEAMING possui duas características peculiares que foram vantajosas para o desenvolvimento do jogo: sua infraestrutura orientada a objetos para manipular elementos num jogo e sua estrutura para desenhar objetos na tela.

```
1 ELEMENT = Class{
2   init = function(self) --Funcao inicializadora dessa classe
3     self.tp = nil --Tipo desse elemento
4     self.subtp = nil --Subtipo que esse elemento pertence (caso exista)
5     self.id = nil --Id desse elemento (caso exista)
6   end
7 }
```

Todos os objetos no jogo foram herdados de uma classe básica chamada *Element*. Um *Element* pode possuir três atributos fundamentais: um tipo (chamado de *type* ou *tp*), um subtipo (chamado de *subtype* ou *subtp*) e uma identificação única (chamada de *id*). Com esses três atributos é possível agrupar, identificar, selecionar ou até mesmo destruir elementos no jogo com muita praticidade, ajudando imensamente seu

desenvolvimento.

```
1 --Exemplo de uma classe "inimigo" que herda de ELEMENT
2 INIMIGO = Class{
3   __includes = {ELEMENT}, --Classes que serao herdadas
4   init = function(self, x, y) --Funcao inicializadora dessa classe
5     ELEMENT.init(self)
6     self.tp = "Inimigo Simples"
7     self.subtp = "inimigos"
8
9     self.pos = {x, y}
10  end,
11  update = function(self,delta) --Funcao que atualiza a posicao
12    self.pos = {self.pos.x + 20*delta, self.pos.y + 20*delta}
13  end,
14  draw = function(self) --Funcao que renderiza o inimigo
15    local raio_do_circulo = 20
16    drawCircle(self.pos, raio_do_circulo)
17  end
18 }
```

STEAMING reduz todo o *pipeline gráfico* do jogo em camadas. Elementos do jogo desenháveis são atribuídos à alguma camada e a própria infraestrutura do template vai percorrer todas elas, em ordem, e desenhar os objetos dentro de cada uma, chamando o método correspondente da classe. Com essa organização é possível diagramar o *design gráfico* do jogo bem mais facilmente e não se preocupar com objetos sendo desenhados na ordem certa. Entretanto, objetos na mesma camada são desenhados em uma ordem imprevisível, já que são armazenados dentro de uma *tabela hash* sem ordem específica.

```
1 --Exemplo de uma organizacao de camadas de renderizacao
2 function render()
3     DrawTable(Layer.FUNDO) --Primeira camada a ser desenhada
4
5     DrawTable(Layer.INIMIGOS)
6
7     DrawTable(Layer.PROJETEIS)
8
9     DrawTable(Layer.PLAYER)
10
11    DrawTable(Layer.INTERFACE_USUARIO) --Ultima camada a ser desenhada
12 end
```

4.2 Gerenciador de Saves

Para armazenar dados entre partidas, foi desenvolvido um **gerenciador de saves**. Este é responsável por guardar as informações relevantes do jogo, como maiores pontuações, configurações de som escolhidas pelo usuário, se é a primeira vez que o jogador abriu o jogo, entre outras.

Todas as informações são guardadas em tabelas de Lua, que depois são transformadas no formato

padronizado *JSON* e salvas no disco rígido. Para recuperar essas informações, o **gerenciador de saves** faz a transformação inversa toda vez que o jogo é inicializado e atribui os valores para suas respectivas variáveis.

4.3 Script de Fases

Para organizar o fluxo de eventos em cada nível do jogo, foi implementado um gerenciador de níveis. Assim foi possível abstrair a lógica de cada fase em *scripts*.

Cada *script* descreve em que sequência inimigos ou ações se desenrolam em cada nível do jogo. Isso facilitou imensamente no *design* dos níveis, especialmente quando foi necessário balancear o jogo, pois o *script* permite editar e visualizar facilmente cada nível em poucas linhas. Vários métodos foram criados para gerar padrões específicos de inimigos, dar vidas-extras para o Psycho ou até mesmo criar chefões.

Porém, para manter um ritmo durante o nível e não fazer o gerenciador rodar todas as linhas de uma vez, foi necessário usar corotinas. Elas permitem paralelizar o código e gerar múltiplos pontos de retorno para que o jogo aguarde certas condições antes de continuar o *script* de um nível. Essas condições podem ser de tempo (como esperar 10 segundos antes de enviar a próxima sequência de inimigos) ou de aguardar até que não tenha um inimigo vivo na tela (utilizada, por exemplo, para esperar que o jogador destrua todos inimigos antes de avançar para o nível seguinte).

```
1  --Exemplo de um script de fase
2  F.single{enemy = SB,
3           x = ORIGINAL_WINDOW_WIDTH + 20,
4           y = ORIGINAL_WINDOW_HEIGHT/2,
5           dx = -1,
6           dy = 0}
7
8  LM.wait(1.5)
9
10 F.circle{enemy = {SB}, number = 4, radius = 610}
11
12 LM.wait("noenemies")
13
14 F.fromHorizontal{side = "left",
15                  mode = "center",
16                  number = 9,
17                  enemy_x_margin = 25,
18                  enemy_y_margin = 90,
19                  enemy = {SB},
20                  speed_m = 1.5}
```

Funções como *F.single* ou *F.circle* são utilizadas para criar formações de inimigos: *F.single* por exemplo cria um inimigo em uma dada posição, com uma direção e tipo específico. *F.fromHorizontal* cria uma linha de inimigos que originam da esquerda ou direita da tela. Cada função de formação de inimigos possue vários parâmetros para customizar sua organização e atributos.

```
1 --Da "yield" em uma corotina
2 function level_manager.wait(arg)
3   coroutine.yield(arg)
4 end
```

As funçoes do tipo *wait* são as responsáveis por manter a sincronização do script. Elas são um encapsulamento de uma parada da corotina atual, interrompendo o script em certa linha do código, e apenas retomando após alguma condição. Essa função vai por sua vez chamar a função *resume*, que vai lidar com o tipo de parada especificado pelo argumento *arg*.

```
1 --Versao simplificada do loop que roda o script de fases
2 function level_manager.resume()
3     local arg, status
4     --Se o nivel parou, termina a corotina
5     if not COROUTINE then return end
6
7     --Continua o script, e pega proximo argumento do yield
8     status, arg = coroutine.resume(COROUTINE)
9
10    --Retorna a corotina somente quando nao tem mais inimigos
11    if arg == "noenemies" then
12        --Checa a cada .02 segundos se possue um inimigo
13        LEVEL_TIMER:every(.02,
14            function()
15                if Util.tableEmpty(SUBTP_TABLE["enemies"]) then
16                    level_manager.resume()
17                end
18            end
19        )
20
21    --Espera "arg" segundos
22    elseif type(arg) == "number" then
23        LEVEL_TIMER:after(arg, level_manager.resume)
24
25    --Retorna a corotina somente quando nao tem mais chefoes na tela
26    elseif arg == "nobosses" then
27        --Checa a cada .02 segundos se possue um chefao
28        LEVEL_TIMER:every(.02,
29            function()
30                if Util.tableEmpty(SUBTP_TABLE["bosses"]) then
31                    level_manager.resume()
32                end
33            end
34        )
35    end
36 end
```

Todas essas funções fora implementadas com o objetivo de abstrair o máximo possível para que mesmo um não-programador consiga criar seus próprios níveis no jogo, sem precisar saber de detalhes técnicos como corotinas.

4.4 Shaders

Uma boa descrição de *PsyChO: The Ball* seria: contém muitos círculos. Logo, como este é o elemento mais característico do jogo, é desejável que os círculos sejam visualmente agradáveis.

Inicialmente os círculos eram desenhados com os métodos imbutidos no arcabouço *LÖVE*. Entretanto estes não utilizam nenhum algoritmo de suavização, como consequência as bordas dos círculos ficam serrilhadas e não combinam com a estética do jogo.

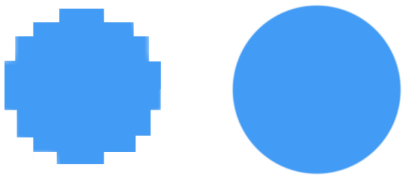

Figura 4.1: Na esquerda um círculo sem anti-aliasing. Na direita um circulo com anti-aliasing.

Desta forma escrevemos um *shader* que desenha círculos com bordas suavizadas (ou, como é conhecido esse efeito, *anti-aliasing*). Utilizando a linguagem de *shaders* da *LÖVE* (muito semelhante à linguagem de *shaders* do *OpenGL*, *GLSL*), um *script* genérico consegue desenhar qualquer círculo com bordas suavizadas, dado o tamanho do seu diâmetro:

```
1 vec4 effect(vec4 color,Image text,vec2 text_coords,vec2 scrn_coords){
2   vec4 pixel = Texel(text, text_coords); //This is the current pixel color
3   vec2 center = vec2(0.5,0.5); //Center of the image
4   pixel.a = 1 - smoothstep(.5 - 1/%f, .5, distance(center, text_coords));
5   return pixel * color;
6 }
```

A primeira linha de nosso código define a função que criará o efeito desejado. Ela recebe como argumento a cor atual definida pela *LÖVE*, a textura da imagem a ser desenhada, as coordenadas do atual pixel na imagem (normalizada no intervalo [0,1]) e as coordenadas do pixel em relação à tela. A segunda linha vai pegar a cor atual do pixel que estamos rodando a função, enquanto a terceira linha define uma variável para representar o centro da imagem.

A linha 4 vai criar nosso efeito de anti-aliasing. Enviando o diâmetro do círculo para subsituir o *%f*, utilizamos a função *smoothstep* que vai fazer interpolação da transparência em pixels que estão na borda de nosso círculo. Essa função vai retornar 0 caso o pixel esteja dentro do raio do círculo, e 1 caso esteja fora. Entretanto, se a posição estiver no intervalo de 1 pixel desse raio (e a distância de 1 pixel é determinada pela divisão 1/diâmetro do círculo, para receber o valor normalizado do mesmo), a função vai retornar um valor correspondente entre 0 e 1. Subtraindo de 1 o resultado teremos o valor de transparência (ou *alpha*) de nosso pixel, assim chegando no resultado esperado.

Além disso, o jogo precisa de outro efeito realizado por *shaders*: borrar a tela, denominado *blur*. Esse efeito acontece quando o jogador pausa o jogo, de forma que somente a interface da tela de pausa fique nítida, enquanto todos outros elementos do jogo fiquem desfocados como se estivessem borrados.

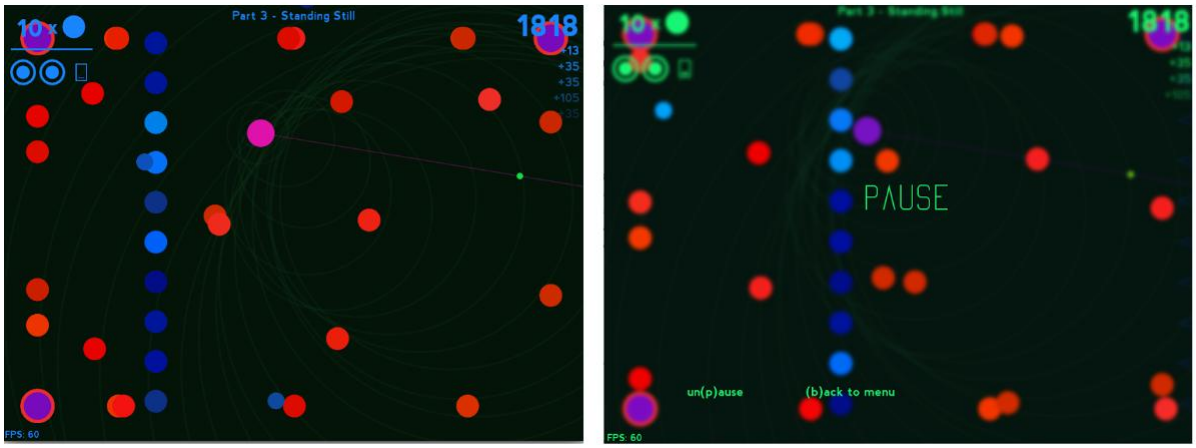

Figura 4.2: Na esquerda uma tela de jogo sem efeito de *blur*. Na direita efeito de *blur* aplicado.

Para atingir tal efeito foi criado dois *scripts* de *shader*, um que aplica um efeito de borrão horizontal e outro vertical. Assim, ao utilizar os dois em sequência, é possível chegar ao resultado esperado de uma tela desfocada, criando um efeito muito interessante enquanto o jogo estiver pausado.

```
1   extern number win_width; //1 / (comprimento da tela)
2   const float kernel[5] = float[](0.270270270, 0.1945945946, 0.1216216216,
3                                   0.0540540541, 0.0162162162);
4   vec4 effect(vec4 color, sampler2D tex, vec2 tex_coords, vec2 pos) {
5     color = texture2D(tex, tex_coords) * kernel[0];
6     for(int i = 1; i < 5; i++) {
7       color += texture2D(tex,
8             vec2(tex_coords.x + 6*i * win_width, tex_coords.y)) * kernel[i];
9       color += texture2D(tex,
10            vec2(tex_coords.x - 6*i * win_width, tex_coords.y)) * kernel[i];
11    }
12    return color;
13  }
```

Acima podemos ver o código para o shader que cria o efeito de borrão horizontal. A parte interessante consiste nas linha 2-3 e o laço das linhas 9-14. Nas linhas 2 e 3 definimos nosso *kernel*, um vetor de *floats* que vai ser aplicado em nosso pixel. O laço começando na linha 9 vai somar ao pixel atual porcentagens das cores dos pixels à esquerda e à direita do pixel, criando uma média das cores ao redor (horizontalmente apenas). Aplicando em cada pixel temos parte do efeito desejado.

O código do *shader* de desfocamento vertical é análogo ao horizontal, utilizando as cores acima e abaixo de cada pixel, e utilizando a altura da tela, em vez do comprimento para saber o tamanho vertical de um pixel:

```
1 extern number win_height; //1 / (altura da tela)
2 const float kernel[5] = float[](0.270270270, 0.1945945946, 0.1216216216,
3                                 0.0540540541, 0.0162162162);
4 vec4 effect(vec4 color, sampler2D tex, vec2 tex_coords, vec2 pos) {
```

```
5     color = texture2D(tex, tex_coords) * kernel[0];
6     for(int i = 1; i < 5; i++) {
7       color += texture2D(tex,
8             vec2(tex_coords.x, tex_coords.y + 6*i * win_height)) * kernel[i];
9       color += texture2D(tex,
10            vec2(tex_coords.x, tex_coords.y - 6*i * win_height)) * kernel[i];
11    }
12    return color;
13  }
```

Para aplicar os dois shaders na mesma imagem, utilizamos uma das ferramentas do arcabouço *LOVE2D*: *canvas*. Estes servem para definir telas intermediárias de renderização. Assim desenhamos primeiramente os elementos de interesse com o *shader* horizontal em um canvas. Logo em seguida renderizamos esse canvas na janela principal do jogo, agora aplicando o *vertical*, chegando no efeito desejado.

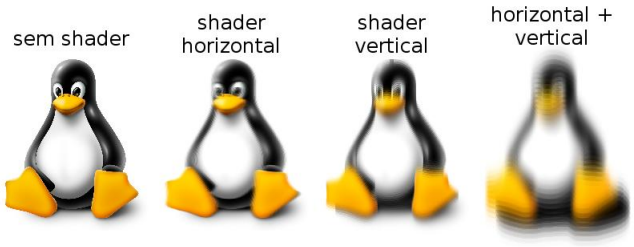

Figura 4.3: Diferença visual da utilização e combinação dos *shaders* para efeito *blur*.

4.5 Juiceness

Além desses efeitos com *shaders*, um grande foco no desenvolvimento do *PsyChO: The Ball* foi em deixar o jogo mais *juicy*. Isso é realizado com vários efeitos pequenos mas que tentam melhorar a experiência do jogador. *Juicy* vem do termo *Juiciness*, criado pelos desenvolvedores de jogos Martin Jonasson e Petri Purho, em uma palestra sobre *Game Design*?. Nela eles definem *Juiciness* como uma recompensa ou fortalecimento de uma experiência, dada alguma ação do usuário e geralmente transmitida por efeitos especiais, tais como explosões, tremer a tela, ou até mesmo por efeitos sonoros.

Além disso, é considerado um efeito *Juicy*, a utilização de interpolações e transições em vez de mudanças brutas de valores. Quase todas as transições do jogo são feitas desta forma. Um bom exemplo disso é a movimentação do personagem principal, *Psycho*. Em vez dele ter uma velocidade constante quando se move, o Psycho tem uma leve aceleração quando começa seu movimento e uma leve desaceleração quando o jogador solta o comando de andar. Isso cria uma ilusão de movimento muito mais realista e agradável para o jogador em contraponto a uma velocidade que se inicia e acaba instantaneamente.

Outro efeito implementado no jogo, para aumentar a *Juiciness*, é a explosão de partículas toda vez que um inimigo morre. Vários pequenos círculos de tamanho e velocidade variados explodem assim que um inimigo é destruído, dando mais emoção ao jogo e satisfação ao jogador.

Como último exemplo de efeito *juicy*, *PsyChO: The Ball* utiliza um clássico nos jogos de hoje em dia: tremer a tela quando o jogador morre. Este efeito, apesar de bem simples de se implementar (já que só é preciso transladar todos elementos na tela alguns pixels para a frente e para trás), gera um grande impacto e imersão ao jogador.

4.6 Problemas e Desafios

Durante o desenvolvimento de *PsyChO: The Ball* surgiram vários problemas e desafios. Para consertar *bugs* ou otimizar o código foi necessário aprender a fundo a linguagem Lua, explorar bibliotecas disponíveis e até mesmo relembrar algoritmos aprendidos no curso.

Um dos desafios mais interessantes que surgiu foi a transição de cor dos objetos. Para manter o tema psicodélico, quase todos objetos ficam transitando entre valores de uma tabela de cores própria. Essa transição, porém, nunca parecia natural e orgânica já que apenas eram interpolados os valores *RGB* de um objeto para os valores alvo, resultando em cores intermediárias *amarronzadas*. Para resolver este problema, foi utilizada a representação de cores *HSL*, que representa uma cor através de sua matiz, saturação e luminosidade (traduzidos do inglês *hue*, *saturation* e *lightness*).

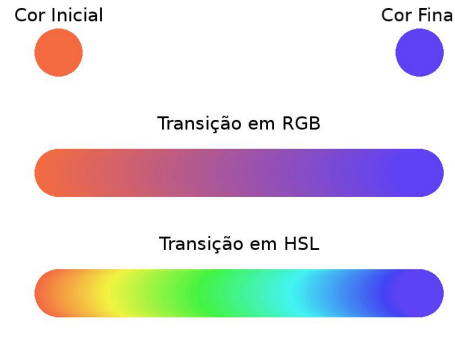

Figura 4.4: Diferença visual de transições usando representação RGB e HSL

Com essa mudança, o jogo ficou com transições entre cores muito mais naturais e visualmente belas, se encaixando melhor na temática psicodélica.

A utilização de HSL foi bem direta no programa. Em todos lugares que se tratava de cor, foi utilizado diretamente valores HSL. Somente na hora de desenhar objetos na tela que se faz a transição do valor HSL para RGB (representação utilizada pela função de definir cores no arcabouço *LÖVE*).

A função de converter valores HSL (com todos valores normalizados entre 0 e 255) para RGB é bem simples, baseada na própria geração de cores de RGB para HSL:

```
1  --Recebe valores de 'h', 's', 'l' (todos no intervalo [0,255]) e retorna
2  --valores correspondentes de 'r', 'g' e 'b' (tambem no intervalo [0,255]).
3  function convertHSLtoRGB(h, s, l)
4          if s<=0 then return l,l,l end
5
6          h, s, l = h/256*6, s/255, l/255
7
8          local c = (1-math.abs(2*l-1))*s
9          local x = (1-math.abs(h%2-1))*c
10         local m,r,g,b = (l-.5*c), 0,0,0
11
12         if h < 1     then
13                 r,g,b = c,x,0
14         elseif h < 2 then
15                 r,g,b = x,c,0
16         elseif h < 3 then
17                 r,g,b = 0,c,x
```

```
18          elseif h < 4 then
19                  r,g,b = 0,x,c
20          elseif h < 5 then
21                  r,g,b = x,0,c
22          else
23                  r,g,b = c,0,x
24          end
25
26          return (r+m)*255,(g+m)*255,(b+m)*255
27  end
```

Por último, como nossa função recebe valores nos intervalos [0,255] e normalmente a matiz, saturação e luminosidade são dadas em graus, porcentagem e porcentagem respectivamente, foi criado uma função que faz essa transformação:

```
1  --Converte os valores HSL em [graus, porcentagem, porcentagem)
2  --para o intervalo [0-255]
3  function HSLtoStandardValues(h,s,l)
4    local sh = h*255/360
5    local ss = s*255/100
6    local sl = l*255/100
7    return sh, ss, sl
8  end
```

Capítulo 5

Conclusões - Parte Subjetiva

A escolha do tema foi relativamente fácil, pois ao longo da graduação fui cada vez me envolvendo mais na área de desenvolvimento de jogos. *PsyChO: The Ball* foi provavelmente o projeto no qual mais dediquei tempo e paixão durante o curso, tanto pelo objetivo de servir como trabalho final de formatura, quanto por ser uma adaptação do meu primeiro jogo feito no início da graduação. Nele, pude ver claramente o progresso que tive como programador, *game designer* e até mesmo artista.

Todos os desafios e frustrações que foram superados durante esse processo levaram a um resultado muito gratificante. Ter a chance de apresentar esse jogo para amigos e membros da Universidade de São Paulo durante os eventos expositivos da UspGameDev faz valer a pena todo o esforço pesquisando em fóruns e artigos como *debuggar* um problema ou como implementar um algoritmo.

É com grande orgulho que levo esse projeto comigo ao fim da graduação, servindo de portofólio do que eu consigo realizar como um desenvolvedor de jogos.

5.1 Graduação e o Trabalho de Formatura

As matérias que me ensinaram técnicas e conceitos de programação: **Introdução à Ciência da Computação**; **Algoritmos e Estruturas de Dados I**; **Técnicas de Programação I**; **Algoritmos e Estruturas de Dados II** e **Conceitos Fundamentais de Linguagens de Programação**, foram todas essenciais para formar a base de conhecimento que tive em programação.

As matérias **Análise de Algoritmos** e **Introdução à Lógica e Verificação de Programas** me deram uma maior formalidade em análisar conceitos lógicos e algoritmos, me fazendo entender melhor otimização e estruturação de código. Foram de imensa ajuda para a produção de *PsyChO: The Ball*.

As matérias **Programação Concorrente** e **Sistemas Operacionais** me deram o conhecimento de paralelismo e corotinas, o qual apliquei diretamente no sistema de leitura de *scripts* pra níveis no **PsyChO: The Ball**.

As matérias: **Atividade Curricular em Cultura e Extensão**; **Design e Programação de Games**; **Laboratório de Programação II**; **Laboratório de Programação Extrema** e **Computação Móvel**, ofereceram atividades nas quais tive de fazer, na prática, um jogo (seja esse digital ou analógico). Em cada uma aprendi com os sucessos (e com os vários erros), podendo assim desenvolver e aprimorar técnicas de desenvolvimento de jogos. Quero fazer um agradecimento especial aos professores que me forneceram essas oportunidades de aplicar conhecimentos computacionais, em programação de jogos, algo que me incentivou a aprender e descobrir coisas novas.

Por último, o grupo extracurricular **UspGameDev** me ensinou muito mais do que eu poderia imaginar sobre desenvolvimento de jogos, *game design*, *game programming patterns* e me guiou durante todos

projetos lúdicos extra-acadêmicos.

5.2 Trabalhos Futuros

Meu objetivo é continuar trabalhando em *PsyChO: The Ball* até chegar no estado desejado: 5 níveis, cada um com mecânicas e inimigos únicos e interessantes. É desejado lançar o jogo assim que possível em alguma plataforma digital de distribuição de jogos como a *Steam*[1], *Humble Bundle*[2] ou *Itch.io*[3].

Além disso, quero continuar meus estudos nas área de Game Design e desenvolvimento de jogos, buscando sempre aprender novas técnicas, descobrir novas ferramentas e superar os diversos desafios que compõe essa área.

[1] store.steampowered.com
[2] https://www.humblebundle.com/
[3] https://itch.io/