Iago da Silva Rios

DESENVOLVIMENTO DE UM GAME VERTICAL SCROLLER SHOOT’EM UP

Monografia baseada em trabalho técnico apresentada ao Departamento de Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito para obtenção do título de Bacharel em Sistemas de Informação.

Orientador: Lucas N. Ferreira

Belo Horizonte

2025

À minha família, Aline, Jésus e Maria;
À orientação do Professor Lucas, sempre solícito
e que me proporcionou trabalhar na minha área predileta;
Aos meus amigos, Luciano Trindade, Pedro Martini, Ravel Santos, Igor Goulart, Matheus Ricardo e Felipe Maia;
Aos amigos que fiz na graduação, Ítalo Dellareti, Raquel Rosa, Maria Clara, Rafael Lima, Artur Padovesi e Túlio Henrique.

Sumário

1. Introdução

A indústria dos *jogos digitais* comerciais remonta a 1971, com a criação do jogo *Computer Space*. Desde então, passamos por fases notáveis no desenvolvimento dos jogos: a era Atari, com jogos simples programados em *assembly*; a era Super Nintendo (SNES), que trouxe mecânicas mais complexas e equipes maiores[1]; a era PlayStation, que popularizou os gráficos 3D e o uso de linguagens como C; e a era Xbox 360/PS3/Wii, que diversificou a interação e ampliou o acesso ao desenvolvimento por meio de *game engines*, como Unity e Unreal Engine (MADHAV, 2014).

No Brasil, a trajetória do desenvolvimento de jogos começou em 1983, com *Amazônia*, de Renato Degiovani. A produção nacional passou por fases distintas, desde a criação individual e a distribuição artesanal em revistas e disquetes, até a organização de uma indústria com editoras, eventos e políticas públicas de incentivo. Entre 2011 e 2017, o setor teve um crescimento expressivo, impulsionado por cursos superiores, pela ampliação do acesso à internet e pela distribuição digital. Tal período culminou em um amadurecimento institucional, marcado pela realização do primeiro Censo da Indústria Brasileira de Jogos Digitais (FORTIM, 2022).

Atualmente, os jogos são uma das maiores indústrias culturais do mundo. Em 2022, seu valor global foi estimado entre 190 e 205 bilhões de dólares. Somente os Estados Unidos, segundo maior mercado, geraram aproximadamente 90 bilhões de dólares em receitas e mais de meio milhão de empregos. No Brasil, o número de estúdios cresceu de 375 para 1.009 entre 2018 e 2022 — um aumento de 169%, segundo a Abragames. O país figura entre os dez maiores mercados globais, com faturamento de 2,3 bilhões de dólares em 2021 e forte potencial de internacionalização (MRE, 2022).

Diante da relevância crescente do setor no Brasil, este projeto tem como objetivo contribuir para o aumento da produção nacional de jogos, oferecendo meios acessíveis para desenvolvedores independentes - especialmente aqueles com perfil técnico, como programadores. Pretende-se, com a proposta, explorar processos de desenvolvimento a partir de ferramentas acessíveis para desenvolvedores *indie*, documentando, ao longo do trabalho, metodologias eficazes e desafios enfrentados.

Em particular, este projeto explora o desenvolvimento de jogos do tipo *vertical scroller shoot'em up*, analisando suas mecânicas e técnicas de implementação, além de avaliar como esse gênero pode ter papel significativo como um modelo de aprendizado para desenvolvedores iniciantes. Dessa forma, o projeto visa não apenas viabilizar a criação de um jogo, mas também fornecer material útil para futuros trabalhos na área, contribuindo para o aprimoramento e crescimento do setor.

[1] Além dos primeiros kits de desenvolvimento para jogos.

2. Referencial Teórico

O desenvolvimento de jogos digitais envolve uma ampla gama de conhecimentos, que abrangem desde fundamentos artísticos até algoritmos e arquiteturas de *software*. Para fomentar a crescente produção de jogos no Brasil, é fundamental compreender tanto os pilares do design de jogos quanto as particularidades do cenário nacional e as ferramentas disponíveis para desenvolvedores independentes.

2.1. Pilares do Game Design

Todo projeto de jogo, do mais simples ao mais complexo, se sustenta sobre os fundamentos do *game design.* Para orientar esse processo, diversos manuais e guias metodológicos foram desenvolvidos[2], oferecendo abordagens teóricas e práticas para a criação de jogos completos.

Entre os referenciais mais influentes está o trabalho de Macklin e Sharp (2016), cuja abordagem está voltada principalmente para o *design* de jogos — isto é, a concepção, estruturação de mecânicas, dinâmicas e experiências de jogo. Os autores defendem que a criação de jogos deve ser encarada como uma prática criativa e iterativa, envolvendo experimentação, *feedback* e refinamento constante.

Complementar ao *design* de jogos, Madhav (2014) oferece um manual mais orientado ao desenvolvimento técnico de jogos digitais, apresentando os fundamentos estruturais da implementação de jogos. Sua obra trata de aspectos como laços de jogo (*game loops)*, sistemas de entrada e saída e a organização interna dos *game objects*, *actors* e componentes. Além disso, o autor explora áreas cruciais como física de jogos, câmeras, inteligência artificial para atores e interfaces de usuário, fornecendo tanto os fundamentos teóricos quanto as implementações necessárias para a criação de jogos dinâmicos e responsivos.

Em ambos os casos, cabe discernir que a fronteira entre o design de um jogo e seu desenvolvimento nem sempre é clara. Essa contradição manifesta-se no trabalho de Macklin e Sharp (2016)[3], que posicionam o design como um subconjunto do desenvolvimento, mesmo que um projeto inicial seja necessário para começar a desenvolver. Tal paradoxo se resolve quando abandonamos uma visão linear do processo. Na prática, a criação de jogos é um ciclo iterativo, onde design e desenvolvimento estão em um diálogo constante.

---

[2] Para mencionar alguns, temos os livros:

- Game design workshop, de Tracy Fullerton (2014);
- Rules of play: game design fundamentals, de Katie Salen e Eric Zimmerman (2003);
- A game design vocabulary: exploring the foundational principles behind good game design, de Anna Anthropy e Naomi Clark (2014).

[3] "Game design is the practice of conceiving of and creating the way a game works, including the core actions, themes, and most importantly, the game's play experience. Game design requires an understanding of different kinds of games, how they work, and the processes game designers use to create them. Game development, on the other hand, encompasses the creation of the game, including **game design**, programming, art production, writing, sound design, level design, producing, testing, marketing, business development, and more" (MACKLIN E SHARP, 2016).

Para que este diálogo entre concepção e implementação seja produtivo, é imprescindível que ambas as frentes compartilhem um vocabulário. É essa linguagem comum que permite a uma equipe traduzir uma visão abstrata — a "experiência do jogador" que se deseja criar — em componentes concretos e tarefas de desenvolvimento.

Nesse vocabulário, o conceito de mecânicas é central. Diferente das regras — entendidas como as instruções explícitas de um jogo — as mecânicas são os processos, algoritmos e dados que operam no sistema, muitas vezes de forma oculta para o jogador. Conforme detalham Adams e Dormans (2012), as mais influentes são chamadas de mecânicas essenciais (*core mechanics*), como, por exemplo, a física de um jogo de plataforma.

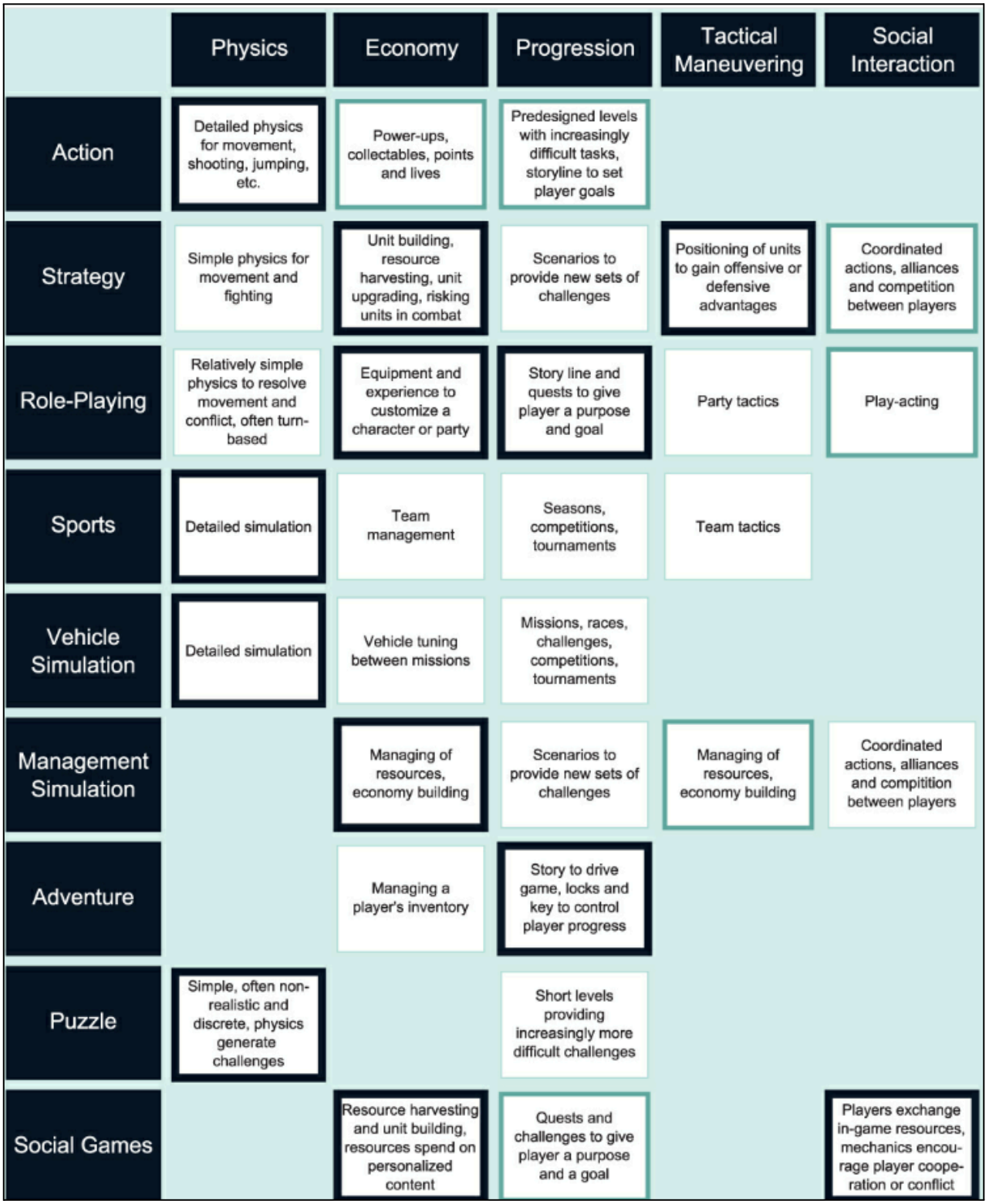
Para fins de design, elas podem ser agrupadas em cinco categorias principais: Física (movimento e colisão), Economia Interna (recursos, vida, pontos), Mecanismos de Progressão (desbloqueio de fases e conteúdo), Manobras Táticas (posicionamento estratégico) e Interação Social (alianças e trocas entre jogadores). A combinação e a ênfase dada a cada uma dessas categorias são o que, em grande parte, definem o gênero e a experiência de um jogo, conforme ilustrado nas Figuras 1 e 2b.

Essa categorização de mecânicas encontra um reflexo direto na arquitetura de software de um jogo, especialmente quando se utiliza um padrão de projeto como o Ator-Componente (Figura 2a). Nesse modelo, um ator (um objeto genérico no jogo, como a nave do jogador ou um inimigo) é composto por diversos componentes, onde cada componente é responsável por implementar uma faceta específica do seu comportamento. Assim, as mecânicas de design abstratas são materializadas através da composição de componentes de software concretos.

Por exemplo, para que um ator participe da mecânica de Física, ele precisa receber componentes específicos de movimento (*MoveComponent*), de corpo rígido (*RigidBodyComponent*) e de colisões (*CollisionComponent*). Para que o jogador possa controlá-lo, aplicando Manobras Táticas, um componente de entrada (*InputComponent*) traduz os comandos do teclado em ações que são executadas pelo componente de movimento. A própria capacidade do ator de ser visível na tela é delegada a componentes de desenho (como o *DrawSpriteComponent* ou o *AnimSpriteComponent*), todos parte da arquitetura do projeto.

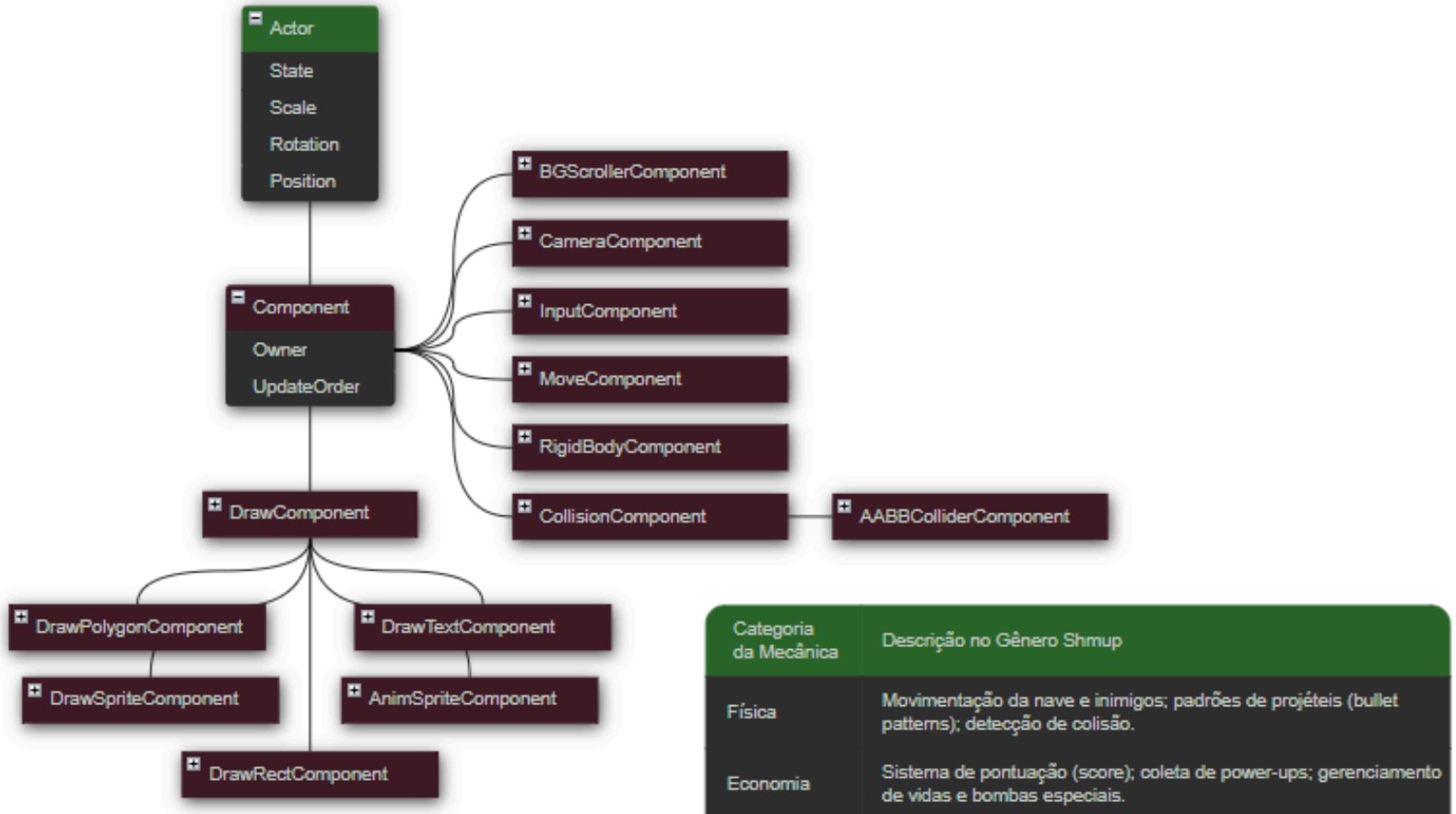
Contudo, nem toda categoria de mecânica se traduz em um único componente. Mecânicas como a Economia Interna (pontos, vidas) podem ser implementadas como variáveis dentro da classe do Ator ou em um componente de estado. Já os Mecanismos de Progressão (avançar de fase, chamar ondas de inimigos) são usualmente gerenciados por atores especializados de nível superior. A Interação Social, por sua vez, transcende a arquitetura de um único ator e seria gerenciada por sistemas externos, como serviços de placares online. Assim, a arquitetura de software de um jogo não apenas viabiliza suas mecânicas, como também materializa as prioridades do designer.

Figura 1 - Mecânicas e Gêneros de Jogos



Fonte: Adams e Dormans, 2012

Figura 2 - Modelo atores-componentes simplificado e categorias de mecânicas para um jogo *vertical scroller shmup*



Categoria da Mecânica	Descrição no Gênero Shmup
Física	Movimentação da nave e inimigos; padrões de projéteis (bullet patterns); detecção de colisão.
Economia	Sistema de pontuação (score); coleta de power-ups; gerenciamento de vidas e bombas especiais.
Progressão	Avanço através de fases com dificuldade crescente; superação de ondas de inimigos (waves); confronto com chefes (bosses).
Manobras Táticas	Posicionamento preciso para desviar de projéteis; memorização de padrões de ataque; gerenciamento de espaço na tela (safe spots).
Interação Social	Competição por pontuação em placares online (leaderboards); modo cooperativo (se aplicável).

Fonte: elaborado pelo autor

2.2. A Indústria de Jogos no Brasil: Desafios e Oportunidades

Uma vez estabelecidos os pilares teóricos e a arquitetura técnica fundamental para a criação de um jogo, uma questão emerge naturalmente: qual o contexto em que o desenvolvedor de jogos brasileiro, munido desse conhecimento, irá atuar? Compreender as particularidades da indústria de jogos no Brasil, com seus desafios e oportunidades, é crucial para contextualizar a relevância e o potencial de projetos de desenvolvimento independentes.

Um ponto de partida para a análise da indústria brasileira é estabelecer uma referência com o mercado global, que sinaliza o perfil profissional valorizado internacionalmente. Um estudo de 2024 sobre vagas em plataformas internacionais (MORAES, 2024) revela a demanda por profissionais com senioridade intermediária (3 a 5 anos de experiência) e com forte ênfase em habilidades técnicas de programação.

Em contrapartida, a força de trabalho brasileira apresenta características distintas. Embora a área de programação seja expressiva, compreendendo 29% dos profissionais, o que distingue o cenário nacional é a predominância ainda maior de profissionais de arte e design, que representam 37%. Essa composição reforça o perfil do desenvolvedor brasileiro como um "criador cultural", no qual a engenharia de software e a expressão artística são indissociáveis, criando um ecossistema único de colaboração e desafios para o profissional de perfil técnico (HARRIS, 2023; PRIETO; NESTERIUK, 2021).

É nesse cenário que a indústria brasileira demonstra sua notável expansão, com o número de estúdios saltando de 133 em 2014 para 1.042 em 2023. Essa expansão numérica, contudo, não se traduz em grandes operações: dados da Abragames (2022) mostram que 60,4% das empresas brasileiras registraram um faturamento anual de até R$ 360 mil, categorizadas assim por sua pequena escala. Adicionalmente, a forte concentração de empresas na região Sudeste (56%) impõe um desafio de acesso a oportunidades para os profissionais de outras regiões do país.

Apesar desses desafios estruturais, a indústria brasileira se destaca por uma característica complexa e singular: uma dupla identidade que combina um forte foco no mercado doméstico com uma vocação natural para a exportação. Por um lado, o Brasil continua sendo o principal mercado-alvo para 76% das desenvolvedoras, sustentado por uma resiliente base de mais de 100 milhões de jogadores. Por outro lado, o setor demonstra um DNA exportador robusto, com metade das empresas que atuam no exterior obtendo mais de 70% de seu faturamento de fontes internacionais, consolidando a reputação do país como um polo de prestação de serviços de alta qualidade em áreas como arte e engenharia (HARRIS, 2023).

Essa forte atuação na prestação de serviços não é apenas uma oportunidade de negócio, mas muitas vezes uma necessidade para a sobrevivência dos estúdios. A pesquisa da Abragames (2023) revela que a sustentabilidade financeira é um desafio central, levando a um modelo de negócio híbrido. Dados mostram que 51% das empresas relataram prestar serviços para terceiros em 2022. Além do desenvolvimento de jogos

próprios, atividades como "Serviços de Arte" (28%) e "Gamificação" (24%) figuram entre as principais fontes de receita, indicando que muitos estúdios precisam diversificar suas operações para se manterem ativos no mercado.

Esse perfil de desenvolvedor, inserido em um modelo de negócio flexível, reflete-se diretamente na estrutura das equipes e nos regimes de contratação do setor. Afastando-se do modelo formal, o regime CLT representa apenas 26% dos vínculos. O modelo predominante é o terceirizado (freelancer/PJ), com 47% dos colaboradores, sinalizando um mercado de trabalho flexível, baseado em projetos e majoritariamente remoto (70%). Este cenário, ao mesmo tempo que amplia as oportunidades de colaboração à distância, expõe os profissionais a desafios de instabilidade e precarização (HARRIS, 2023).

É nesse contexto que um dos desafios da indústria se torna visível no nível do desenvolvedor: a cultura do *crunch* — períodos de trabalho extremo para cumprir prazos. No cenário *indie*, essa prática ocorre não apenas pela pressão externa de investidores e publishers, mas também por uma pressão interna, um fenômeno de auto exploração por parte de desenvolvedores movidos pela paixão (HARRIS, 2022).

Para combater esse ciclo, o próprio ecossistema independente fomenta uma solução estratégica: o modelo de lançamento iterativo, popularizado como Acesso Antecipado (*Early Access*). Essa abordagem troca um único lançamento de alto risco por um desenvolvimento fragmentado e contínuo, o que alivia a pressão de prazos inflexíveis e envolve a comunidade diretamente no aprimoramento do jogo.

2.3. Estratégias e Jogos de Referência no Cenário Indie

Para além da análise macro da indústria, o estudo de casos específicos de jogos de referência é fundamental para compreender a evolução das práticas, ferramentas e escopo no desenvolvimento independente. Ao contrastar títulos clássicos — muitas vezes criados sob limitações técnicas e com tecnologia própria — com sucessos modernos que se beneficiaram de *engines* comerciais, é possível mapear o que se tornou mais acessível e o que permanece como desafio para o desenvolvedor. Esta seção analisará, portanto, jogos selecionados não apenas sob a ótica de suas mecânicas, mas também considerando fatores de produção como o tamanho da equipe, o tempo de desenvolvimento e as ferramentas utilizadas, traçando um paralelo com a realidade do desenvolvedor brasileiro.

A primeira onda de sucessos *indies*, no início dos anos 2000, foi marcada por um ethos de desenvolvimento solitário e domínio técnico profundo. O caso mais emblemático é Cave Story, desenvolvido integralmente por Daisuke "Pixel" Amaya ao longo de cinco anos, em uma engine própria feita em C++ (AMAYA, 2011). Essa realidade se repete em outros clássicos: Braid, aclamado por sua mecânica de manipulação do tempo, foi programado por Jonathan Blow em uma engine própria. Da mesma forma, Super Meat Boy também dependeu de uma engine customizada, criada por um de seus dois fundadores (SWIRSKY; PAJOT, 2012). Nesses casos, a independência criativa era

quase absoluta, mas vinha acompanhada de uma total dependência da capacidade técnica de seus criadores para construir as ferramentas do zero.

A década seguinte, contudo, marcou uma mudança de paradigma com a popularização de *engines* comerciais acessíveis. Hollow Knight é o principal expoente dessa nova era. Desenvolvido por uma equipe de apenas três pessoas (denominados Team Cherry), utilizando a engine Unity, o jogo alcançou um escopo e um polimento antes restritos a grandes estúdios. O uso da Unity permitiu que o Team Cherry delegasse tarefas complexas de física e renderização, concentrando os 2,5 anos de desenvolvimento na criação do vasto mundo e da arte desenhada à mão. O financiamento via Kickstarter também representa uma evolução, ao validar o projeto diretamente com a comunidade (TEAM CHERRY, 2014).

Mesmo na era moderna, contudo, a criação de tecnologia própria ainda surge como um diferencial. O recente sucesso Animal Well, desenvolvido por uma única pessoa ao longo de sete anos, resgata o espírito de Cave Story, mas com um novo propósito. Seu desenvolvedor, Billy Basso, criou uma engine caseira não por falta de alternativas, mas para obter controle total sobre a performance e criar efeitos e mecânicas únicas, otimizadas para sua visão artística (SCULLION, 2023;). O modelo de negócio também se modernizou, com o jogo sendo apoiado e distribuído por uma publisher focada em títulos *indie* inovadores.

A análise desses casos revela uma trajetória clara: o que antes era um desafio primariamente de engenharia de software tornou-se, em grande parte, um desafio de gestão de escopo e produção de conteúdo. O acesso a ferramentas como a Unity — dominante no Brasil, com 80% de adesão (HARRIS, 2023) — democratizou o desenvolvimento e reduziu drasticamente a barreira técnica de entrada. Contudo, o custo não desapareceu; ele se deslocou para a produção de ativos de alta qualidade (arte, som, música) e para o marketing.

Conclui-se, portanto, que a evolução do desenvolvimento independente redefine a posição estratégica do profissional com perfil técnico. Com a barreira da criação de tecnologia drasticamente reduzida, seu maior valor passa da construção de *engines* para a otimização de ferramentas, criação de mecânicas inovadoras e gestão de projetos complexos.

Além disso, no contexto brasileiro, essa posição estratégica é amplificada por uma vantagem competitiva singular: a abundância de talentos em Arte e Design. Considerando que o maior gargalo do mercado *indie* contemporâneo é justamente a produção de conteúdo de alta qualidade, a forte veia artística nacional oferece um cenário ideal, no qual o desenvolvedor técnico pode potencializar talentos criativos locais para gerar projetos de impacto global.

3. Metodologia[4]

A metodologia adotada neste trabalho foi fundamentada nos guias de Macklin e Sharp (2016) e Madhav (2014), combinando princípios de *design* iterativo com técnicas estruturadas de desenvolvimento. A seleção de ferramentas, em particular, foi guiada por uma filosofia central alinhada à realidade do desenvolvedor independente brasileiro: a busca por controle técnico, alto desempenho e acessibilidade. Priorizaram-se tecnologias de baixo custo, gratuitas ou de código aberto, que não impõem barreiras financeiras e permitem a total apropriação do processo criativo e técnico.

3.1. Biblioteca de Baixo Nível vs. Game Engine

A decisão metodológica fundamental do projeto foi a escolha de utilizar uma biblioteca de baixo nível em vez de uma *game engine* comercial.

Embora muitas vezes se recorra a *game engines*, como Unity e Unreal Engine, para facilitar esse processo, o uso de bibliotecas como a Simple DirectMedia Layer (SDL) pode ser particularmente interessante. *Game engines* oferecem uma série de recursos prontos, como gráficos, física e sistemas de som, permitindo que os desenvolvedores se concentrem na criação de conteúdo e mecânicas. No entanto, elas abstraem muitos dos processos fundamentais que envolvem a implementação de jogos, como a estruturação do *game loop*, o gerenciamento de memória, a interação direta com o *hardware* e a criação de sistemas de renderização personalizados.

Nesse contexto, o uso da SDL possibilita maior domínio sobre tais aspectos essenciais. A biblioteca promove o contato direto com o gerenciamento de gráficos, áudio e entrada — componentes frequentemente simplificados por *engines* comerciais. Dessa maneira, seu uso permite compreender mais profundamente os mecanismos internos de funcionamento dos jogos digitais, especialmente nos casos em que a execução em tempo real exige precisão e desempenho, como é o caso do projeto aqui desenvolvido. A linguagem C++ foi escolhida em conjunto com a SDL por seu alto desempenho e controle granular, sendo o padrão de indústria para este tipo de aplicação.

3.2. Pipeline de Assets e Ferramentas de Suporte

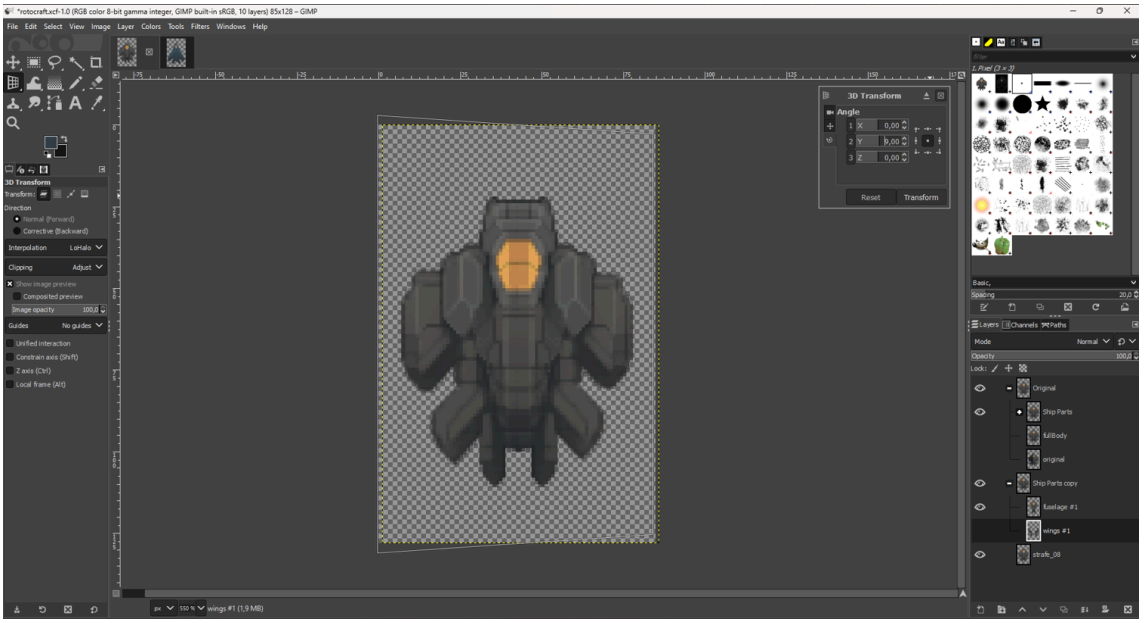
Seguindo o mesmo princípio de acessibilidade, o fluxo de trabalho para criação de ativos artísticos e a gestão do projeto foram compostos por ferramentas gratuitas ou de baixo custo, amplamente adotadas pela comunidade *indie*.

Para a criação dos ativos visuais, foi adotado um conjunto de *softwares* de código aberto: Asesprite para a produção de *pixel art* e animações; GIMP para edição de imagens e tarefas de maior complexidade; Inkscape para a prototipação visual; e Natron para composição de vídeo. A escolha se deu por serem ferramentas poderosas que não representam um ônus financeiro para o desenvolvedor. O Natron foi especificamente utilizado para criar um plano de fundo (*background)* animado para o menu principal,

[4] No final do trabalho estão disponibilizados, nos anexos, ilustrações de usos dessas ferramentas e concept-arts das mecânicas.

empregando vetorização de elementos, criação de *keyframes* e animações por meio de interpolação das características desses vetores.

Figura 3 - Uso do GIMP para a produção de um inimigo

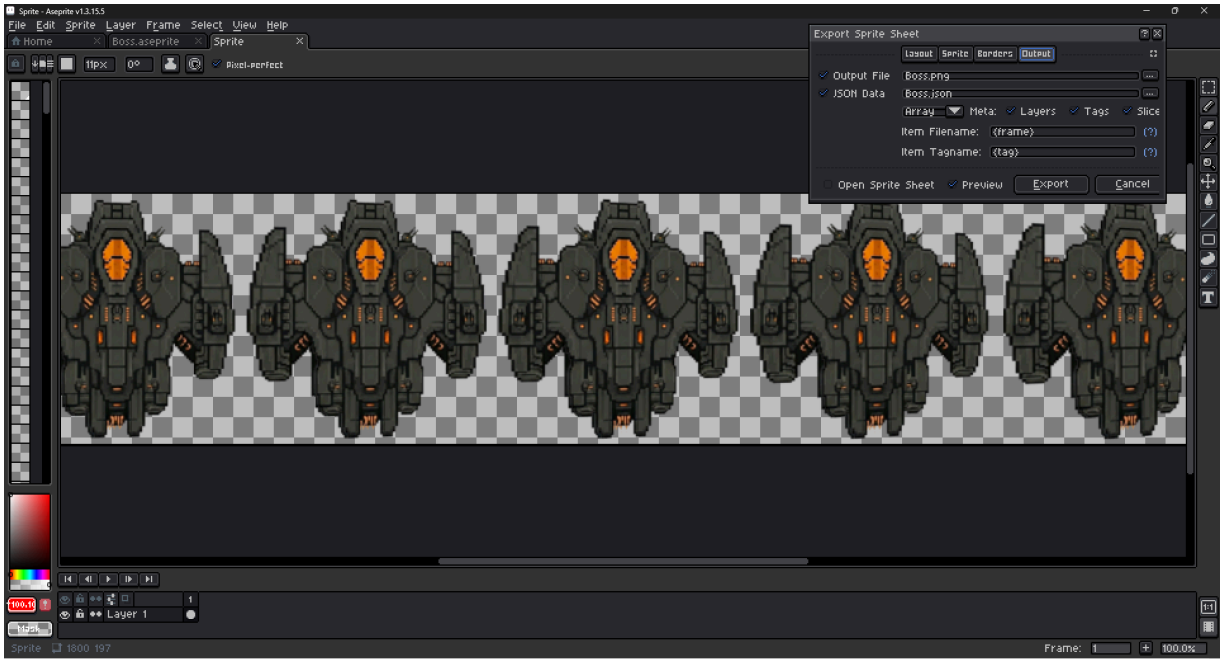


Fonte: elaborado pelo autor

O sistema de áudio do jogo foi desenvolvido utilizando FMOD, por meio de sua API gratuita, que se aproxima em funcionalidade de soluções proprietárias de alto nível da indústria. A integração do FMOD justificou-se pela ausência de um *mixer* estável e plenamente disponível para a SDL3 no momento do projeto (o *SDL3_mixer*). Além de resolver essa questão de compatibilidade, o FMOD proporcionou uma camada de abstração rica para o sistema de áudio, permitindo o acesso e o agrupamento rápido de áudios em categorias (efeitos sonoros e música de fundo - vfx e *background music*).

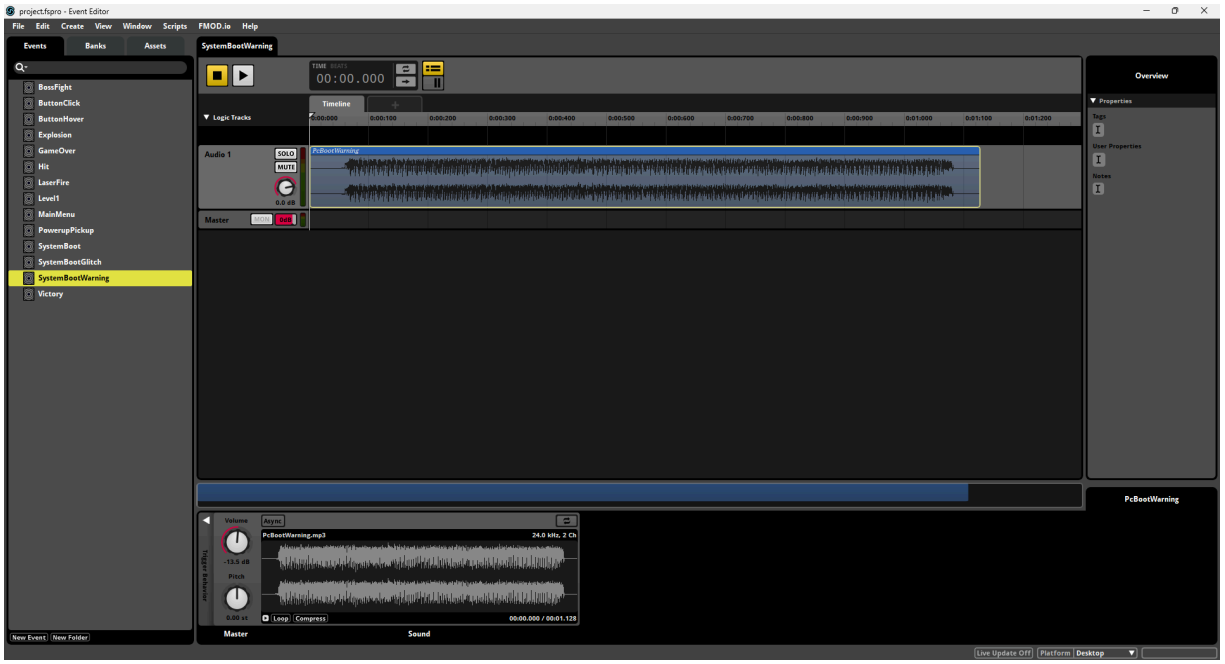
Gerenciando todo o processo, o sistema de controle de versão Git, com o repositório hospedado no GitHub, foi essencial. Além de ser o padrão da indústria, oferece planos gratuitos para projetos pessoais e pequenas equipes, eliminando custos de infraestrutura e garantindo a resiliência do projeto contra falhas de hardware.

Figura 4 - Uso do Asesprite para elaboração de uma spritesheet de movimento do boss



Fonte: elaborado pelo autor

Figura 5 - Uso do FMOD para criar um banco de áudios



Fonte: elaborado pelo autor

3.3. Mecânicas e Sistemas de Jogo

3.3.1. Mecânicas Centrais (Movimento e Combate)

A principal mecânica desenvolvida para o jogo é o sistema de movimento. Diferentemente do modelo tradicional de *shoot'em ups*, onde o movimento cessa instantaneamente, foi implementado um sistema baseado em física. Neste modelo, as teclas de direção não definem uma velocidade constante, mas aplicam um vetor de força sobre o corpo rígido do ator da nave. Como resultado, a jogabilidade ganha uma sensação de inércia, com acelerações e desacelerações suaves que, no entanto, não comprometem a responsividade exigida pelo gênero.

O sistema de combate do jogo foi implementado com suas funcionalidades essenciais. A nave do jogador possui um tiro principal fixo, e o sistema de detecção de colisão é capaz de gerenciar os impactos entre projéteis, inimigos e o jogador. Para fornecer um feedback claro sobre o estado do jogador, foi implementada uma interface de usuário (HUD) que exibe em tempo real a porcentagem de vida restante. Esta representação visual da saúde, parte da economia interna do jogo, permite que o jogador tome decisões estratégicas com base em sua vulnerabilidade.

Adicionalmente, a arquitetura do sistema de colisão foi projetada de forma granular. Cada nave é composta por três componentes de colisão distintos, representando a fuselagem e as asas. Embora o dano implementado no escopo do jogo seja genérico[5], a estrutura foi projetada de forma extensível, permitindo que mecânicas mais complexas, como dano localizado, sejam facilmente incorporadas em trabalhos futuros.

3.3.2. Sistemas de Movimento e Colisão

Inicialmente, a detecção de colisão do projeto utilizou o método simples *Axis-Aligned Bounding Box* (*AABB*). Embora rápido, este método apresentava uma limitação crítica: imprecisão na colisão de naves rotacionadas ou com caixas de colisão complexas (*hitboxes*). Para garantir a fidelidade física e a precisão exigida pelo gênero *shoot'em up*, o sistema de colisão foi implementado utilizando o *Separating Axis Theorem* (SAT), que oferece uma detecção precisa entre polígonos convexos arbitrários.

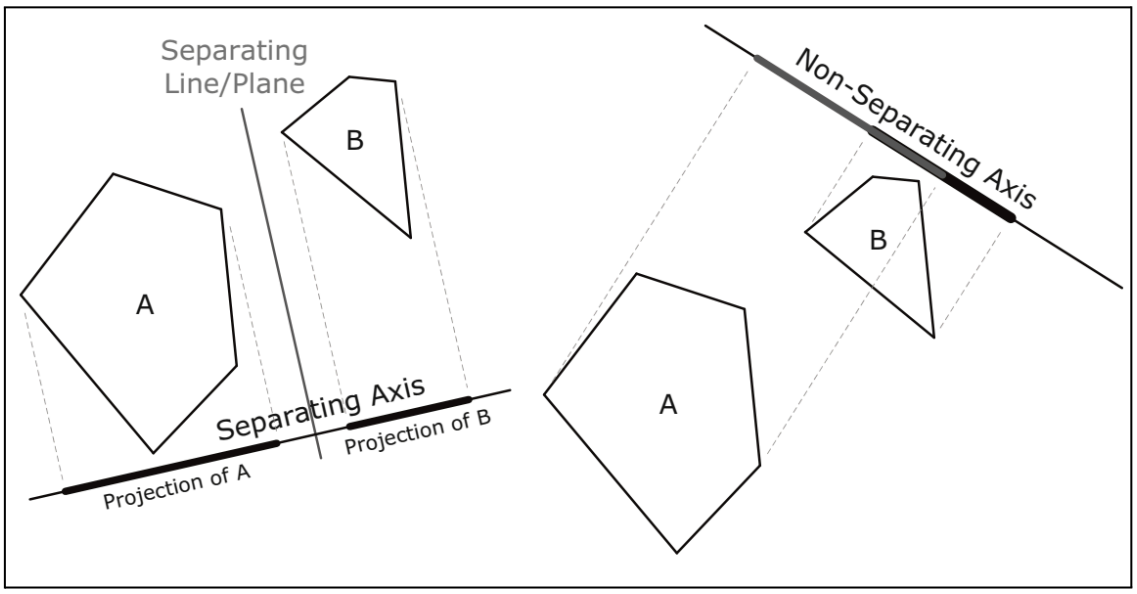
A implementação do SAT exigiu a introdução de um componente que define as formas exatas das caixas de *hitboxes* das naves e projéteis (*PolygonColliderComponent*). O cerne da detecção reside em uma função — *TestSAT* — que projeta os polígonos em todos os seus eixos normais. O princípio fundamental do SAT é que, se houver uma separação em qualquer eixo, a colisão é impossível (*separating axis*), otimizando o desempenho (Figura 6).

Para a resposta física da colisão, o sistema calcula o Vetor de Translação Mínimo (MTV)[6]. O MTV não apenas indica a colisão, mas também determina o vetor de menor

[5] Contudo, os diferentes colisores atuam na exibição localizada de efeitos de dano na nave;
[6] Para o trabalho foi implementado o SAT e o MTV presente em BITTLE, 2010. Essa implementação pode ser vista no repositório do jogo.

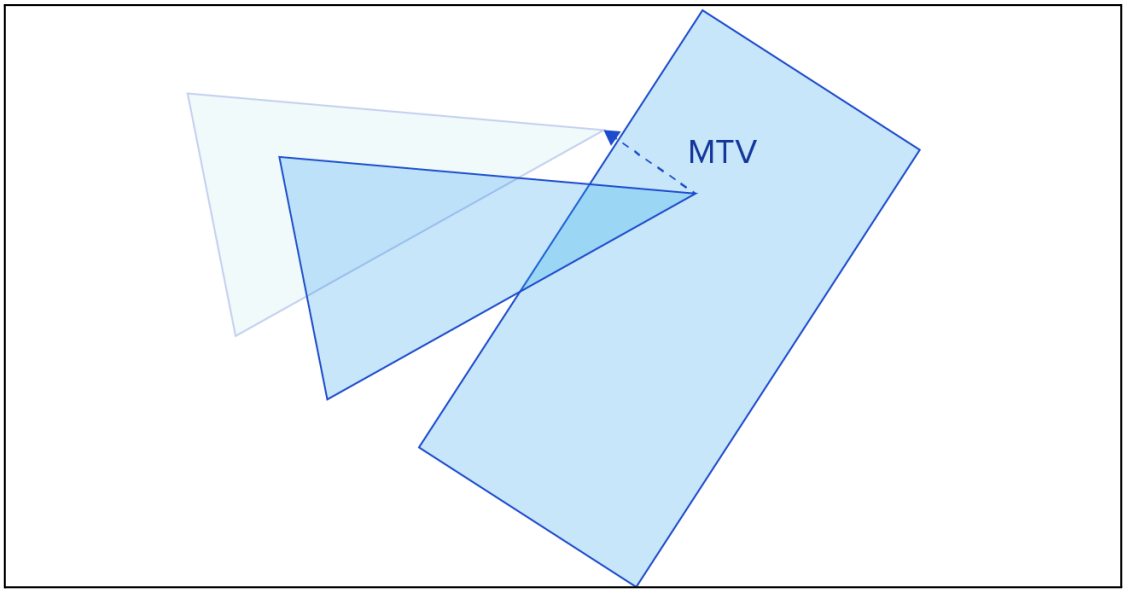
distância necessária para separar os corpos. Isso permite que o sistema de física do jogo aplique a separação de corpos e o impulso de colisão de forma realista. Essa migração não apenas elevou a precisão da jogabilidade, mas também demonstrou a capacidade da engine de integrar física de jogos de forma avançada.

Figura 6 - Detecção de interseção pelo teorema do eixo separador (SAT)[7]



Fonte: Gregory, 2018

Figura 7 - Vetor Mínimo de Translação (MTV) entre dois polígonos convexos[8]



Fonte: Souto, 2025

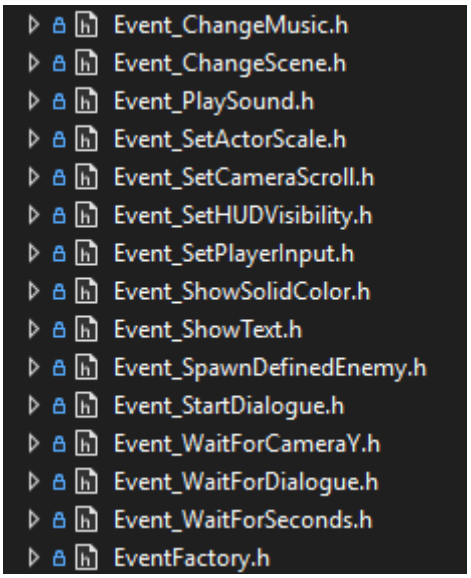
[7] Projeções em um eixo separador resultam em segmentos disjuntos; sem eixo separador, as formas se intersectam.
[8] As implementações do SAT e MTV utilizadas podem ser vistas em https://github.com/iago-r/cipherRaven, repositório do projeto.

3.4. Arquitetura Orientada a Eventos e o Sistema de Cinemáticas

Inicialmente, projetou-se uma arquitetura para a criação de cinemáticas que utilizava um *parser* de arquivos JSON que lia comandos sequenciais para exibição de texto e imagens. Embora funcional, essa abordagem de programação (*hard-coding*) se mostrou inflexível para a progressão narrativa e, principalmente, ainda possuía dependências de recompilação para qualquer ajuste na lógica da cena. Além disso, a arquitetura inicial não era adequada para sincronização de áudio ou diálogos em tempo real.

Para alcançar a extensibilidade e robustez necessárias no sistema de cinemáticas, a solução foi migrar a arquitetura para um paradigma orientado a eventos. Esta refatoração resultou na criação de um subsistema desacoplado que utiliza o padrão Observador (*Observer*), onde a lógica da cena é completamente separada da sua apresentação. Essa separação permite que os designers criem narrativas complexas sem depender de intervenção do programador.

Figura 8 - Alguns dos eventos elaborados



Fonte: elaborado pelo autor

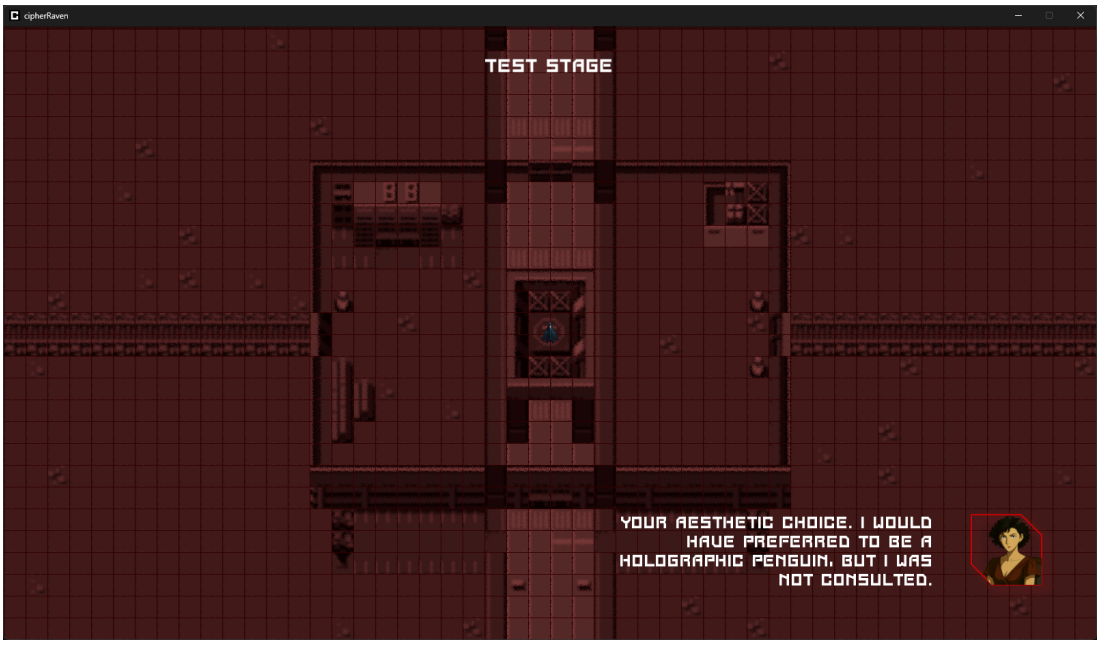
A base da arquitetura é a interface IEvent, que padroniza todos os eventos do jogo[9]. Isso permite que os sistemas se comuniquem de forma assíncrona, eliminando dependências rígidas entre os componentes. O *parser* inicial evoluiu para o diretor de cinemáticas (*CinematicDirector*), que atua como um coordenador: ele lê os *scripts* JSON e emite eventos para uma fábrica de eventos (*EventFactory*). Essa separação possibilita lógica assíncrona e *callbacks* de eventos mais complexos (como "esperar o jogador pressionar uma tecla para continuar").

Por fim, foi implementado o componente de caixas de diálogo (*DialogueBox*) para gerenciar múltiplas linhas de diálogo e exibir texto dinamicamente. Para evitar o

[9] Como, por exemplo, eventos de jogador sendo destruído, carregamento de fase, fim de execução da cinemática etc.

transbordamento de texto e garantir a apresentação visual dos diálogos, o componente utiliza a função *TTF_RenderText_Blended_Wrapped* da biblioteca *SDL_ttf*, que processa e renderiza textos longos com quebra de linha automática.

Figura 9 - Caixa de diálogo com texto quebrado automaticamente



Fonte: elaborado pelo autor

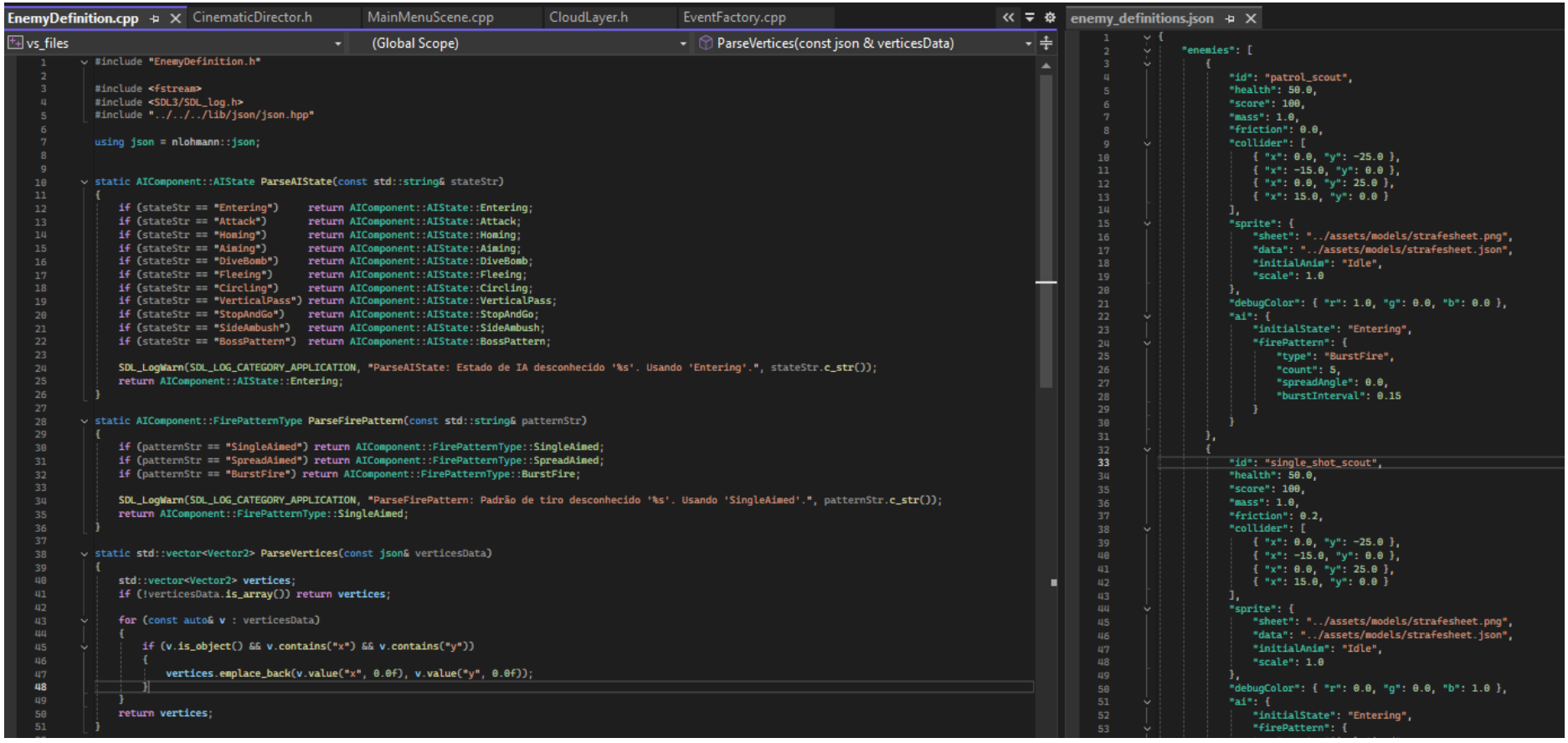
3.5. Inteligência Artificial Orientada a Dados (Data-Driven AI)

O design de inimigos e seus padrões de ataque foram desenvolvidos através da implementação de um padrão de Inteligência Artificial Orientada a Dados (*Data-Driven AI*). Esta arquitetura foi concebida para dissociar o comportamento da IA da lógica do *game object*, movendo as definições complexas (como atributos e scripts de comportamento) para arquivos de dados externos (JSON). Isso garante maior flexibilidade, pois permite que os *game designers* criem, modifiquem e balanceiem novos inimigos sem alterar o código-fonte em C++.

O sistema é sustentado por dois componentes principais. O primeiro é uma definição de inimigos (*EnemyDefinition*), que consiste em estruturas de dados lidas a partir de arquivos JSON. Estas definições carregam integralmente os parâmetros do ator inimigo — incluindo atributos essenciais (vida, massa, *sprites*), o comportamento da IA e o padrão de tiro a ser seguido, permitindo uma rápida iteração e balanceamento de dificuldade.

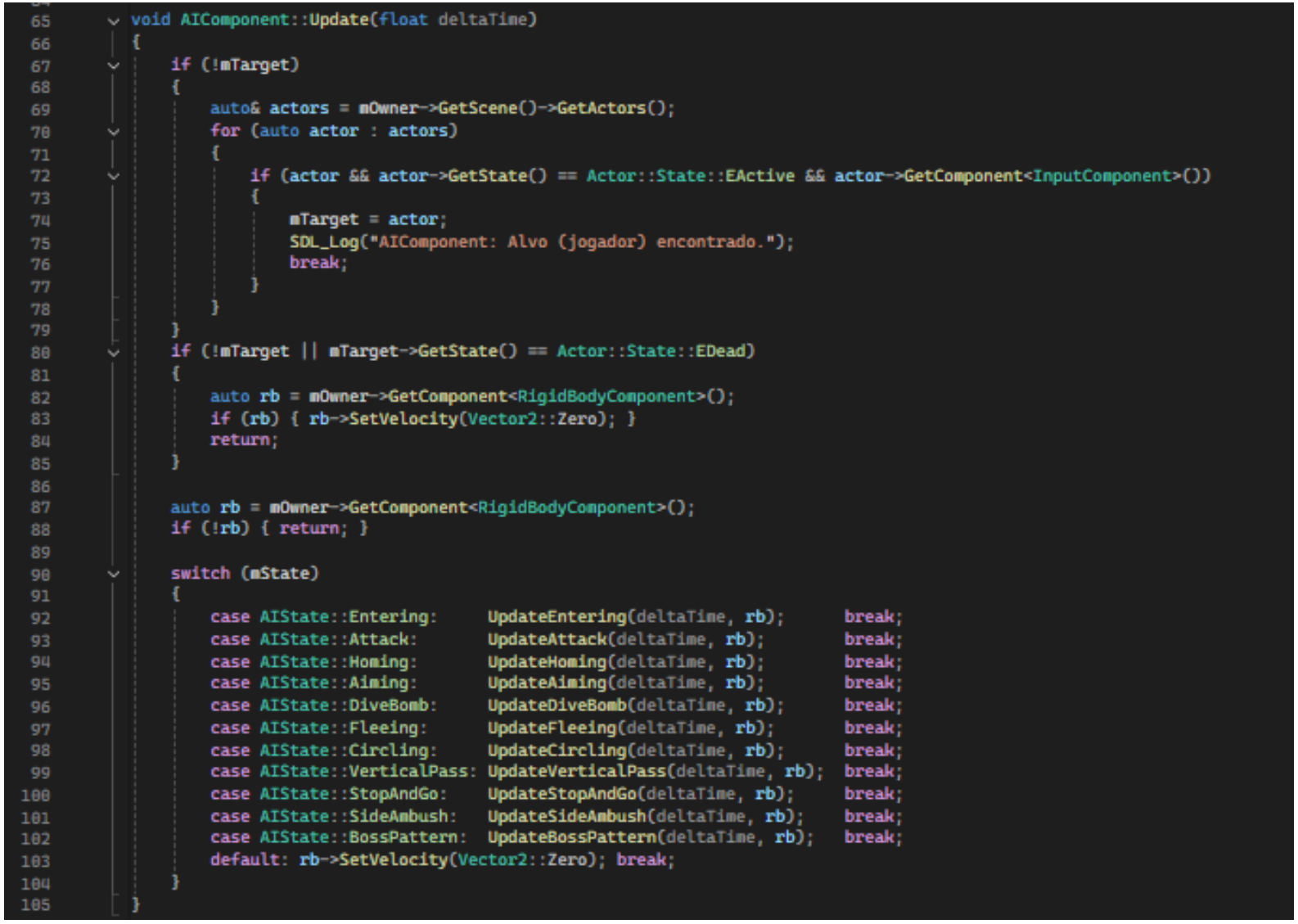
O segundo é um componente de IA (*AIComponent*), implementado como uma Máquina de Estados de Comportamento (*AIState*). Este componente gerencia a transição dinâmica da nave inimiga entre estados pré-definidos, como entrada na tela (*Entering*), perseguição ao jogador (*Homing*), movimento circular (*Circling*) e padrões de chefe complexos (*BossPattern*).

Figura 10 - Carregamento da definição dos Inimigos pelo EnemyDefinition e JSON de exemplo com a definição de um inimigo



Fonte: elaborado pelo autor

Figura 11 - Máquina de estados e componentes de IA elaborados



```
65  void AIComponent::Update(float deltaTime)
66  {
67      if (!mTarget)
68      {
69          auto& actors = mOwner->GetScene()->GetActors();
70          for (auto actor : actors)
71          {
72              if (actor && actor->GetState() == Actor::State::EActive && actor->GetComponent<InputComponent>())
73              {
74                  mTarget = actor;
75                  SDL_Log("AIComponent: Alvo (jogador) encontrado.");
76                  break;
77              }
78          }
79      }
80      if (!mTarget || mTarget->GetState() == Actor::State::EDead)
81      {
82          auto rb = mOwner->GetComponent<RigidBodyComponent>();
83          if (rb) { rb->SetVelocity(Vector2::Zero); }
84          return;
85      }
86
87      auto rb = mOwner->GetComponent<RigidBodyComponent>();
88      if (!rb) { return; }
89
90      switch (mState)
91      {
92          case AIState::Entering:     UpdateEntering(deltaTime, rb);      break;
93          case AIState::Attack:       UpdateAttack(deltaTime, rb);        break;
94          case AIState::Homing:       UpdateHoming(deltaTime, rb);        break;
95          case AIState::Aiming:       UpdateAiming(deltaTime, rb);        break;
96          case AIState::DiveBomb:     UpdateDiveBomb(deltaTime, rb);      break;
97          case AIState::Fleeing:      UpdateFleeing(deltaTime, rb);       break;
98          case AIState::Circling:     UpdateCircling(deltaTime, rb);      break;
99          case AIState::VerticalPass: UpdateVerticalPass(deltaTime, rb);  break;
100         case AIState::StopAndGo:    UpdateStopAndGo(deltaTime, rb);     break;
101         case AIState::SideAmbush:   UpdateSideAmbush(deltaTime, rb);    break;
102         case AIState::BossPattern:  UpdateBossPattern(deltaTime, rb);   break;
103         default: rb->SetVelocity(Vector2::Zero); break;
104     }
105 }
```

Fonte: elaborado pelo autor

O componente de IA não apenas controla o movimento, mas também gerencia o tiro parametrizado do inimigo. O componente coordena padrões de disparo definidos nos dados, como mira simples (*SingleAimed*), dispersão angular (*SpreadAimed*) e tiro em rajada (*BurstFire*). Para adicionar um desafio de jogabilidade sutil, o método de atualização de mira (*UpdateAiming*) assegura que a nave só dispare após virar para o ângulo exato do alvo com precisão, o que implementa uma mecânica de mira desafiadora e coerente com o gênero *shmup*.

A integração desta arquitetura com o sistema de eventos é demonstrada no processo de criação de inimigos (*spawn*). A definição de um inimigo no nível é realizada através da emissão de um evento (*SpawnDefinedEnemy*), que carrega a definição JSON do inimigo e suas coordenadas de posicionamento. Isso garante que o posicionamento dos inimigos na fase ocorra de forma assíncrona, sendo ditado pela lógica do nível (definida em dados) e não pela programação rígida do motor.

3.6. Otimização e Composição do Ambiente de Jogo

A arquitetura de *display* do motor foi concebida para oferecer um ambiente de jogo rico e dinâmico, com composição orientada a dados e otimizada para desempenho.

O componente de *tiles* de mapa (*TilemapComponent*) foi desenvolvido para possibilitar a criação de níveis complexos a partir de dados externos (como arquivos JSON). Sua função de desenho (*Draw*) aplica *culling* de tiles, técnica que renderiza apenas os elementos visíveis na área da câmera e da tela do jogador, reduzindo significativamente a carga da GPU.

Em termos de resposta visual, o componente de vida (*HealthComponent*) foi estruturado para integrar a lógica de dano e morte dos atores à geração de efeitos visuais (VFX Actor). Essa integração desacoplada assegura que a lógica de dano apenas emita um sinal (via evento) de que a vida chegou a zero, enquanto o sistema de atores se encarrega de produzir colisões, explosões e demais efeitos visuais.

Por fim, as transições entre cenas e telas do jogo — menus, cinemáticas e fases de gameplay — são organizadas pelo gerenciador de cena (*SceneManager*). Essa classe central foi projetada para assegurar a troca modular e segura entre telas (com métodos como *FadeToScene*), garantindo um fluxo robusto para carregar e descarregar recursos de forma eficiente.

3.7. Estilo Artístico, Narrativa e Progressão

Para definir a identidade visual do jogo, foi adotado um estilo artístico em pixel art. A direção de arte inspira-se na temática Cyberpunk, utilizando uma paleta de cores limitada que contrasta tons escuros com neons vibrantes para criar uma atmosfera distópica. As animações da nave, inimigos e efeitos visuais como explosões e rastros de tiros foram projetadas para serem fluidas, buscando um polimento visual que remete aos clássicos do gênero, mas com uma identidade moderna.

A experiência de jogo é ancorada por uma narrativa de ficção científica distópica, cujo enredo se centra em um piloto que busca compreender seu passado após perder a memória. A arquitetura atual do jogo estabelece a base para desenvolvimento de narrativas completas e um sistema de progressão (com a possibilidade de se expandir com níveis, chefes e desbloqueio de habilidades), por meio do sistema de eventos. O jogo no seu estado atual estabelece o tom, o estilo artístico e as mecânicas fundamentais sobre as quais essa progressão e história estão construídas e podem ser expandidas.

3.8. Uso Crítico da Inteligência Artificial e Otimização do Debugging

O uso de ferramentas de Inteligência Artificial (IA), no decorrer do trabalho, apresentou vantagens e desafios distintos que impactaram o fluxo de desenvolvimento.

O foco principal do uso da IA foi na aceleração de *debugging*. A ferramenta demonstrou uma significativa vantagem ao resolver de forma mais rápida e eficiente problemas complexos de software de baixo nível. Exemplos notáveis incluem a rápida identificação e solução de erros de ponteiros em *capture clauses* dentro de funções lambda em C++[10], tarefas que tipicamente exigiriam um tempo considerável de depuração manual. Essa aplicação validou a tecnologia como um poderoso coadjuvante na otimização da engenharia de *software* de baixo nível.

Em contraste, o uso de IA para a geração de ativos visuais apresentou desafios significativos. Plataformas de IA generativa (como Nano Banana e Sora) mostraram-se insatisfatórias para a pixel art do projeto por falharem em manter a coerência da identidade visual estabelecida. Desta forma, o uso dessas ferramentas foi limitado à criação de referências visuais para elementos chave, como modelos de personagens, templates de naves e efeitos visuais (VFX).

Essas imagens geradas pela IA exigiram extensa edição manual no GIMP para se adaptarem ao estilo do jogo e se tornarem adequadas à produção das spritesheets (finalizadas no GIMP e Aseprite). Esta dualidade demonstra que a IA é uma ferramenta poderosa para a otimização de código e lógica, mas ainda enfrenta limitações na coerência e especificidade exigidas por uma direção de arte precisa.

---

[10] Além do erro de capture clause, a IA foi crucial para solucionar bugs de difícil rastreamento, como:

- Crash do HUD (Dangling Pointer), onde a Interface de Usuário tentava acessar o componente de vida do ator do jogador milissegundos após ele ter sido destruído na memória;
- Corrupção de Listas (Invalidação de Iterador), um bug que ocorria na função central *Scene::Update*, onde a lista de atores (*mActors*) era modificada (adição ou remoção de actors mortos) enquanto estava sendo percorrida, resultando em ponteiros corrompidos e crashes.

4. Resultados

O projeto culminou na entrega de um produto mínimo viável (MVP) de um jogo *vertical scroller shoot'em up*, que validou a eficácia e a arquitetura da *game engine* desenvolvida. Este produto se materializa em um nível inicial completo, no qual é possível verificar a interação em tempo real de todos os subsistemas essenciais desenvolvidos, desde o motor de física que rege a movimentação fluida da nave até o ciclo de vida dos atores, o sistema de animação e a interface de usuário (HUD) dinâmica[11].

Mais do que um simples demonstrativo de mecânicas, essa interação coesa entre os sistemas serviu como a principal ferramenta de validação para a arquitetura de software adotada. A capacidade de gerenciar múltiplos atores dinâmicos sem apresentar falhas de gerenciamento de memória ou instabilidade comprova a robustez do padrão Ator-Componente implementado. A estrutura modular permitiu a adição de sistemas complexos, como a migração do AABB para o Separating Axis Theorem (SAT) na detecção de colisão, sem a necessidade de refatorações massivas, validando a escalabilidade e a solidez do design proposto.

Os principais avanços arquiteturais validaram a manutenibilidade do motor: a implementação do paradigma orientado a eventos e do modelo Inteligência Artificial Orientada a Dados (*Data-Driven AI*) demonstrou que usuários ou designers podem criar novos levels e inimigos e integrar narrativas complexas (via Sistema de Cinemáticas) por meio de arquivos de dados (JSON), sem a necessidade de recompilação do código-fonte em C++.

Um desafio inerente ao projeto, devido à atuação individual, foi a dificuldade em produzir um conjunto robusto de assets coesos (artes visuais, sons, efeitos e animações), o que exigiu um foco maior na infraestrutura técnica. Dada essa complexidade, o escopo e o cronograma planejados para as tarefas de assets acabaram sendo subestimados. Essa dificuldade, contudo, demonstrou na prática a importância da divisão de tarefas e da especialização de papéis (arte, design, engenharia) característica das equipes profissionais de desenvolvimento de jogos (*gamedevs*).

O projeto atingiu seu objetivo central ao entregar um motor de jogo que serve como um estudo de caso prático sobre Engenharia de Software. Além disso, o motor validou a viabilidade técnica de desenvolvimento de motores no contexto acadêmico, utilizando exclusivamente ferramentas open source e gratuitas (SDL, Aseprite, GIMP, FMOD API), oferecendo uma base funcional, robusta e escalável que pode ser utilizada como material de estudo e ponto de partida para novos projetos na comunidade acadêmica de desenvolvimento de jogos.

---

[11] O código-fonte completo do projeto, incluindo *assets* e a documentação técnica, está disponível no repositório GitHub: https://github.com/iago-r/cipherRaven.

5. Referências Bibliográficas

ADAMS, Ernest; DORMANS, Joris. Game Mechanics: Advanced Game Design. Berkeley: New Riders, 2012.

AMAYA, Daisuke. The Story of Cave Story. In: GAME DEVELOPERS CONFERENCE, 2011, San Francisco. [Palestra]. San Francisco: GDC Vault, 2011. Disponível em: <https://gdcvault.com/play/1014621/The-Story-of-CAVE>. Acesso em: 26 jun. 2025.

ANTHROPY, Anna; CLARK, Naomi. *A game design vocabulary: exploring the foundational principles behind good game design*. Boca Raton: CRC Press, 2014.

BITTLE, William. SAT Collision detection and response. Handmade Network, 2010. Disponível em: <https://hero.handmade.network/forums/code-discussion/t/7375-sat_collision_detection_and_response>. Acesso em: 26 jun. 2025.

CARDOSO, Marcos V.; GUSMÃO, Cláudio; HARRIS, Jonathan J. (Org.). Pesquisa da indústria brasileira de games 2023. São Paulo: Abragames, 2023. Disponível em: <https://www.abragames.org/uploads/5/6/8/0/56805537/2023_relat%C3%B3rio_final_v4.3.3_ptbr.pdf>. Acesso em: 27 jun. 2025.

FORTIM, Ivelise (Org.). Pesquisa da indústria brasileira de games 2022. São Paulo: Abragames, 2022. Disponível em: <https://www.abragames.org/uploads/5/6/8/0/56805537/abragames-pt.pdf>. Acesso em: 27 jun. 2025.

FULLERTON, Tracy. *Game design workshop*. 3. ed. Burlington: Morgan Kaufmann, 2014.

GREGORY, Jason. Game engine architecture. 3. ed. Boca Raton: CRC Press, Taylor & Francis, 2018.

MADHAV, Sanjay. *Game Programming Algorithms and Techniques: A Platform-Agnostic Approach*. Upper Saddle River: Pearson Education, 2014.

MACKLIN, Colleen; SHARP, John. *Games, design and play: a detailed approach to iterative game design*. Boston: Pearson Education, 2016.

MINISTÉRIO DAS RELAÇÕES EXTERIORES - MRE. Divisão de Ciência, Tecnologia e Inovação. *Estudo consolidado com informações prestadas pelos Setores de Ciência, Tecnologia e Inovação (SECTECs)*. Brasília: Ministério das Relações Exteriores, 2022. Disponível em: <https://shorturl.at/j3XJW>. Acesso em: 14 abr. 2025.

MORAES, Elyan M. F. et al. Panorama do mercado de jogos eletrônicos: um estudo de caso de vagas de emprego. In: SIMPÓSIO BRASILEIRO DE JOGOS E ENTRETENIMENTO DIGITAL, 23., 2024, Manaus. Anais [...]. Porto Alegre: Sociedade Brasileira de Computação, 2024. Disponível em: <https://sol.sbc.org.br/index.php/sbgames/article/view/32418>. Acesso em: 27 jun. 2025.

NYSTROM, Robert. Observer. In: NYSTROM, Robert. Game Programming Patterns. Monee: Genever Benning, 2014. Disponível em: <https://gameprogrammingpatterns.com/observer.html>. Acesso em: 26 jun. 2025.

PRIETO, Daniel; NESTERIUK, Sâmia. Indie Games BR: estado da arte das pesquisas sobre jogos independentes no Brasil. In: SIMPÓSIO BRASILEIRO DE JOGOS E ENTRETENIMENTO DIGITAL, 20., 2021, Evento Online. Anais [...]. Porto Alegre: Sociedade Brasileira de Computação, 2021. p. 243-252. Disponível em: <https://www.sbgames.org/proceedings2021/IndustriaFull/218217.pdf>. Acesso em: 14 abr. 2025.

SALEN, Katie; ZIMMERMAN, Eric. *Rules of play: game design fundamentals*. Cambridge: MIT Press, 2003.

SCULLION, Chris. YouTuber Dunkey's publishing company publishes first game: Animal Well will be released on May 9. Video Games Chronicle, 9 jan. 2023. Disponível em: <https://www.videogameschronicle.com/news/animal-well-the-first-game-published-by-youtuber-dunkeys-bigmode-has-a-release-date/>. Acesso em: 26 jun. 2025.

SOUTO, Nilson. Video Game Physics Tutorial Part II: Collision Detection for Solid Objects. Toptal, 2025. Disponível em: <https://www.toptal.com/game/video-game-physics-part-ii-collision-detection-for-solid-objects>. Acesso em: 01 dez. 2025.

SWIRSKY, James; PAJOT, Lisanne (Direção). Indie Game: The Movie. [Filme-documentário]. Canadá: BlinkWorks Media, 2012. 96 min. Produção independente.

TEAM CHERRY. Hollow Knight. Kickstarter, 2014. Disponível em: <https://www.kickstarter.com/projects/11662585/hollow-knight>. Acesso em: 26 jun. 2025.

6. ANEXOS

Anexo 1 - Game Design Document

DESIGN DOCUMENT

CIPHERRAVEN

GAME SUMMARY PITCH

cipherRaven é um jogo de tiro vertical no qual o jogador controla uma nave em combates intensos contra ondas de inimigos e chefes, avançando por fases cada vez mais desafiadoras.

GENERO
• *Singleplayer;* • *vertical scroller* • *shmup*

INSPIRACAO

1942

Inspirado na estética e mecânicas clássicas de 1942, este projeto adota a perspectiva aérea e o estilo de combate aéreo característico dos *shooters* da era arcade. A estrutura baseada em fases, com ondas sucessivas de inimigos, chefes ao final de cada estágio e ritmo acelerado de progressão, será reinterpretada com elementos contemporâneos.

CYBERPUNK 2077

Inspirado na ambientação distópica de Cyberpunk 2077, onde biologia e tecnologia se entrelaçam, o projeto adotará elementos visuais, interfaces e sistemas de progressão que evocam o estilo e a atmosfera do jogo. Cidades sombrias iluminadas por néons em tons de vermelho e menus de estilo fragmentado serão incorporados.

EXPERIENCIA DO JOGADOR

O jogador enfrenta desafios que exigem **reflexos rápidos, leitura de padrões de ataque inimigos** e adaptação constante. O combate é centrado no **disparo principal** e na sobrevivência, testando a capacidade do jogador de manobrar dentro das restrições físicas da nave em meio ao caos controlado do *gameplay*.

SOFTWARES PARA DESENVOLVIMENTO

- Motor de Jogo Próprio em C++ e SDL (Visual Studio), com arquitetura Actor-Componente para alto desempenho.
- Aseprite e GIMP para Pixel Art e edição de sprites.
- Natron para composição de assets e efeitos visuais.
- FMOD Studio para criação, mixagem e gestão de eventos de áudio, integrado ao motor via AudioSystem.

DESIGN DOCUMENT
CIPHERRAVEN

AUDIO E ARTE

INTERPRETACAO DO TEMA

Mantendo o tema Cyberpunk na paleta de cores, a proposta é tentar limitar as cores a tons frios e "neon". Para criar a atmosfera e estética tecnológica, serão utilizados, tons de vermelho escuro suave e rosa claro como bases, combinados a tons bege nos inimigos e verde escuro na nave do player. Esses tons serão aplicados tanto nas sprites quanto nos elementos do ambiente, garantindo uma coesão visual que remeta a um ambiente futurístico.

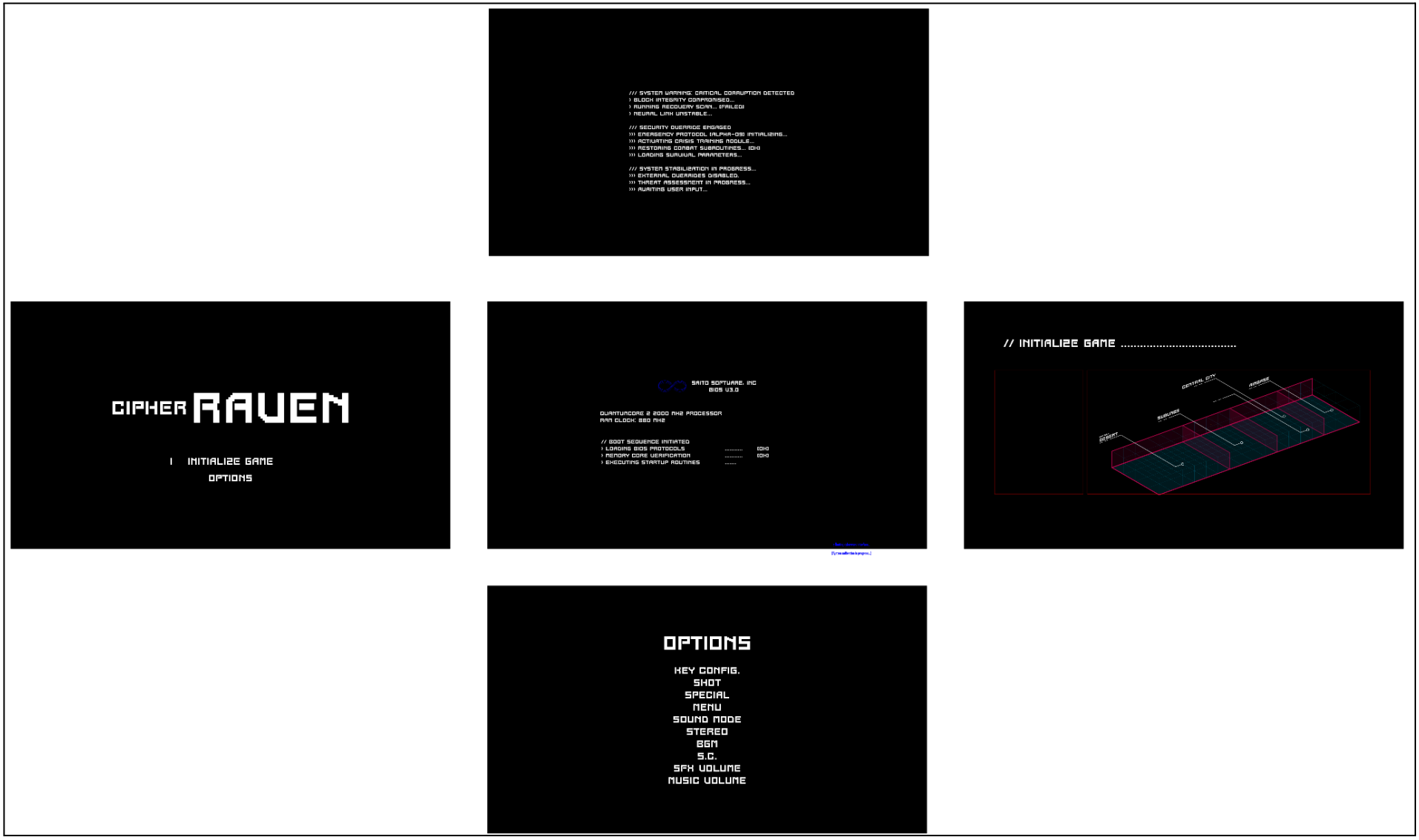
CYBERPUNK 2077

As músicas do tema geral serão criadas no estilo synthwave/eletrônico, evocando a atmosfera Cyberpunk. Os temas serão compostos inicialmente em instrumentos de cordas e adaptados para o estilo 8-bits no formato MIDI, utilizando o software FamiStudio para preservar a estética retrô e futurista. Além disso, os efeitos sonoros serão elaborados tanto artesanalmente quanto por meio de bibliotecas de áudio, com atenção para simular fielmente os sons correspondentes às interações e ambientes do jogo.

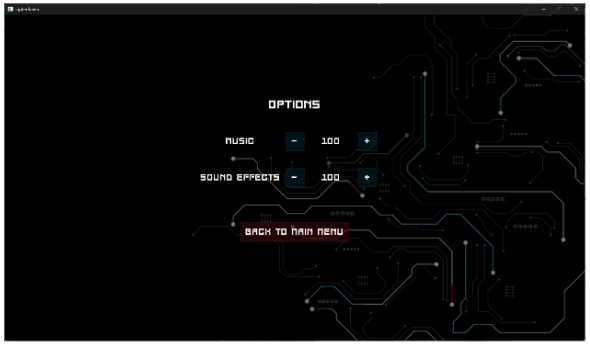
MECANICAS

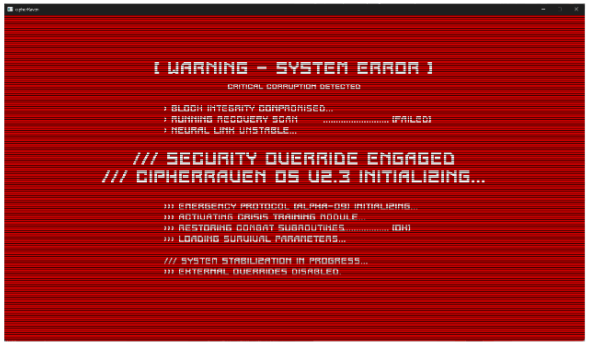
Mecânica	Descrição
Movimentação	A movimentação da nave foi implementada através de um sistema de física que simula **inércia**. Ao aplicar **força** em vez de definir a velocidade diretamente, o motor garante **acelerações e desacelerações suaves**. Este design confere à nave um **peso tático** e exige antecipação do jogador no controlo do impulso.
Colisão e Hitboxes	O sistema de colisão foi aprimorado para Polígonos Convexos, utilizando o algoritmo Separating Axis Theorem (SAT). Isso garante hitboxes perfeitamente precisas para naves e projéteis, mesmo quando rodados. A PhysWorld utiliza essa precisão para calcular o impulso de colisão e a separação de corpos, assegurando um gameplay justo e coerente com a representação visual.
Combate (Tiro Principal)	O disparo principal é controlado por um tempo de espera (cooldown) fixo para regular a cadência de tiro. O motor é responsável por instanciar os projéteis no mundo e garantir que a sua trajetória e o seu ciclo de vida sejam geridos eficientemente, interagindo apenas com as camadas de colisão dos inimigos.
Inteligência Artificial (AI)	A AI inimiga opera via um sistema Data-Driven (orientado a dados), utilizando uma Máquina de Estados de Comportamento. Esta arquitetura flexível permite transições entre padrões táticos como perseguição, movimento evasivo e lógica de Chefe. A AI suporta padrões de tiro configuráveis (dispersão, rajada) e simula um tempo de mira lenta para balancear o desafio.
Economia Interna e Pontuação	A gestão de vida, dano e recompensas é centralizada em um componente dedicado. A pontuação do jogador é persistente e integrada ao ciclo de destruição: cada inimigo derrotado comunica a sua recompensa ao sistema. Além disso, o motor garante o feedback visual (via VFX) para impactos e explosões, reforçando a economia interna.
Narrativa e Scripting	O motor inclui um Roteirista de Cinemáticas refatorado para uma arquitetura Orientada a Eventos. Esta ferramenta permite ao roteirista criar sequências complexas que controlam o fluxo do jogo (controlo de música, spawn de inimigos, etc.) e a apresentação de diálogos com efeito de typewriter e quebra de linha automática, desvinculando a narrativa da lógica de gameplay principal.

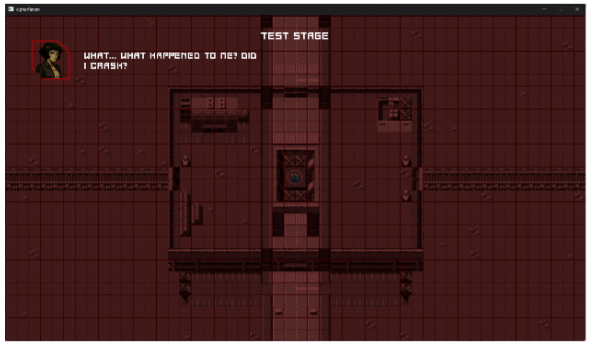
Anexo 2 - Brainstorm inicial, com algumas telas do fluxo de telas



Anexo 3 - Fluxo de telas final



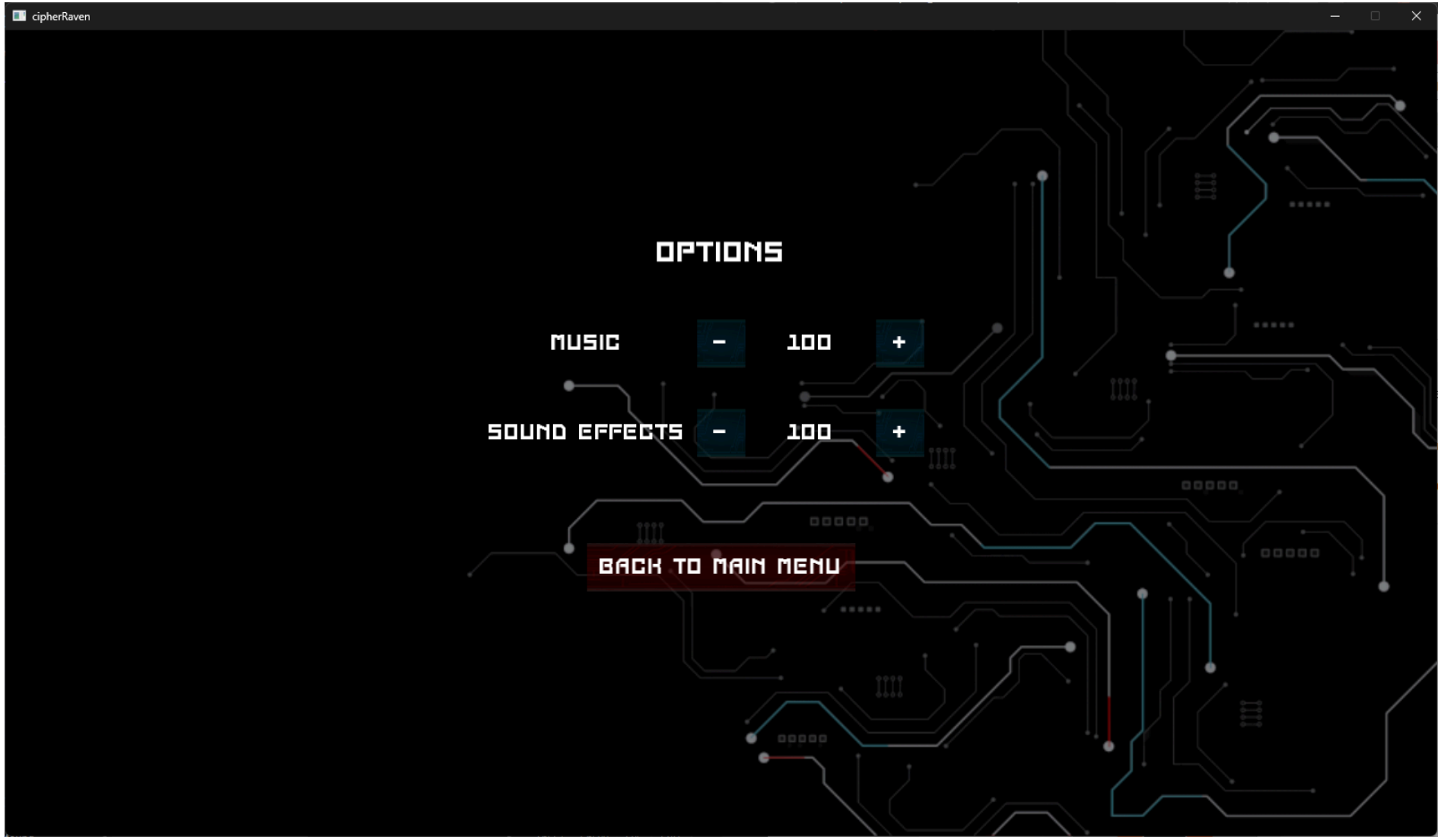




Anexo 4- Menu principal, com botões[12]

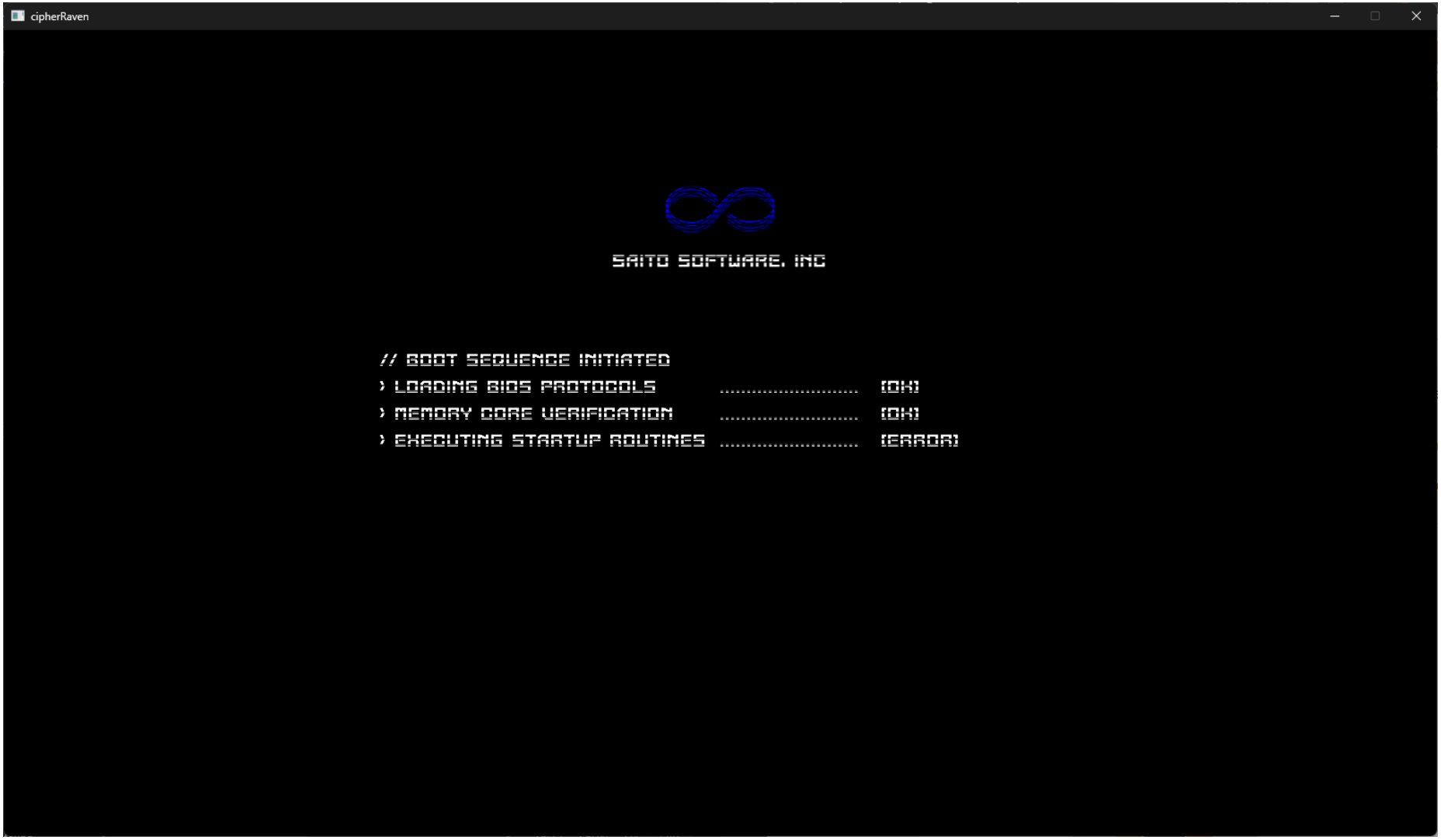
[12] O efeito de seleção e não seleção dos botões foi criado editando as curvas de cores (por meio da ferramenta *Color Curves* do Gimp) de um botão que, no modo padrão, é azul.

Anexo 4- Menu de opções, com botões para controle de volumes[13]



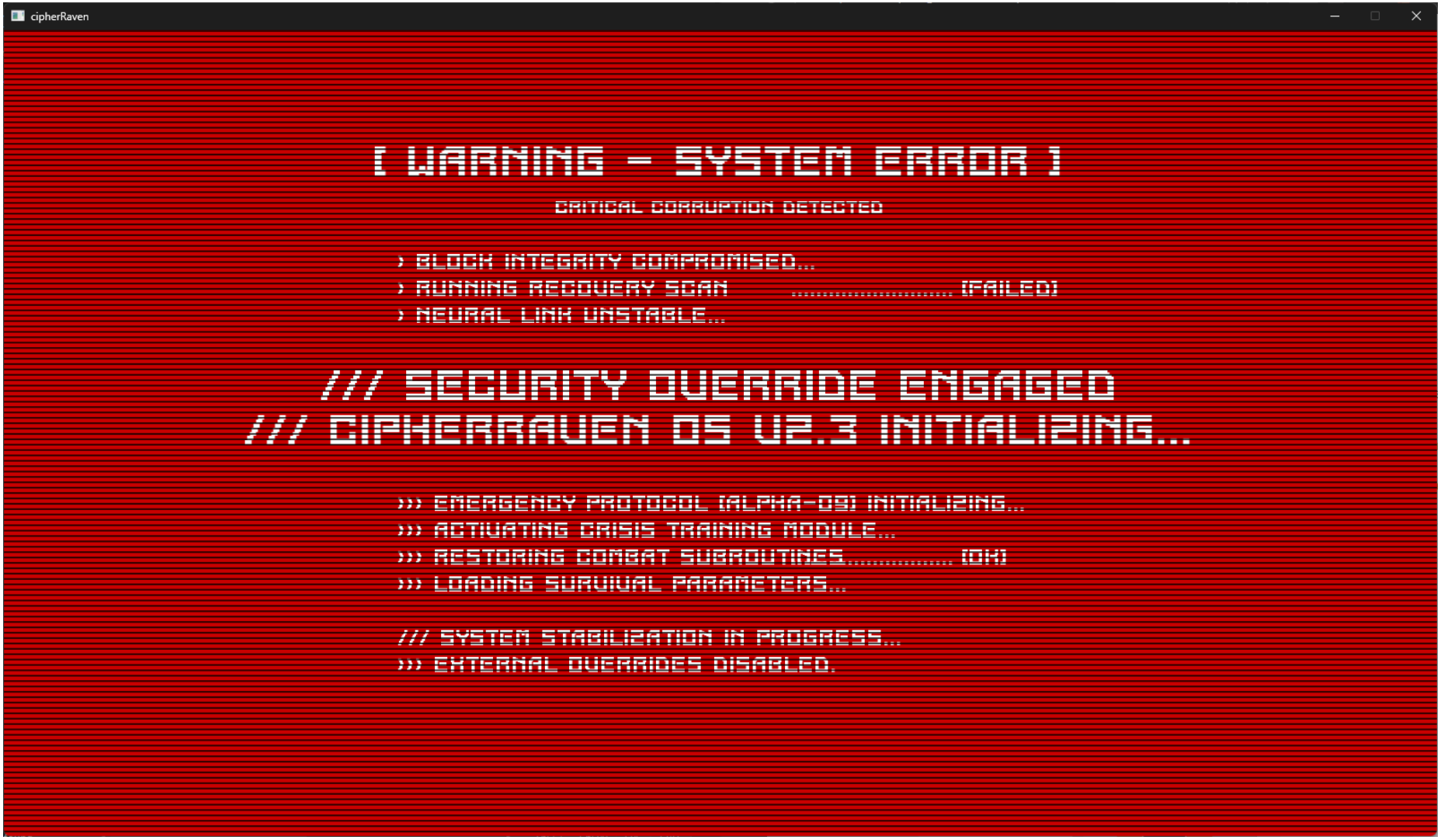
[13] O Fmod possibilita criar grupos de faixas de áudio e controles de volume. No caso do game, foram criados dois grupos: efeitos sonoros (SFX) e músicas de fundo.

Anexo 5 - Primeira cena da cinemática[14]



[14] As linhas horizontais foram criadas propositalmente com uma imagem que simulava telas de CRT. Essas linhas foram submetidas a um efeito de transparência e configuradas no código das cinemáticas para aparecerem sobre a cena. Além disso, cada linha surge gradativamente nesta cena.

Anexo 6 - Segunda cena da cinemática[15]



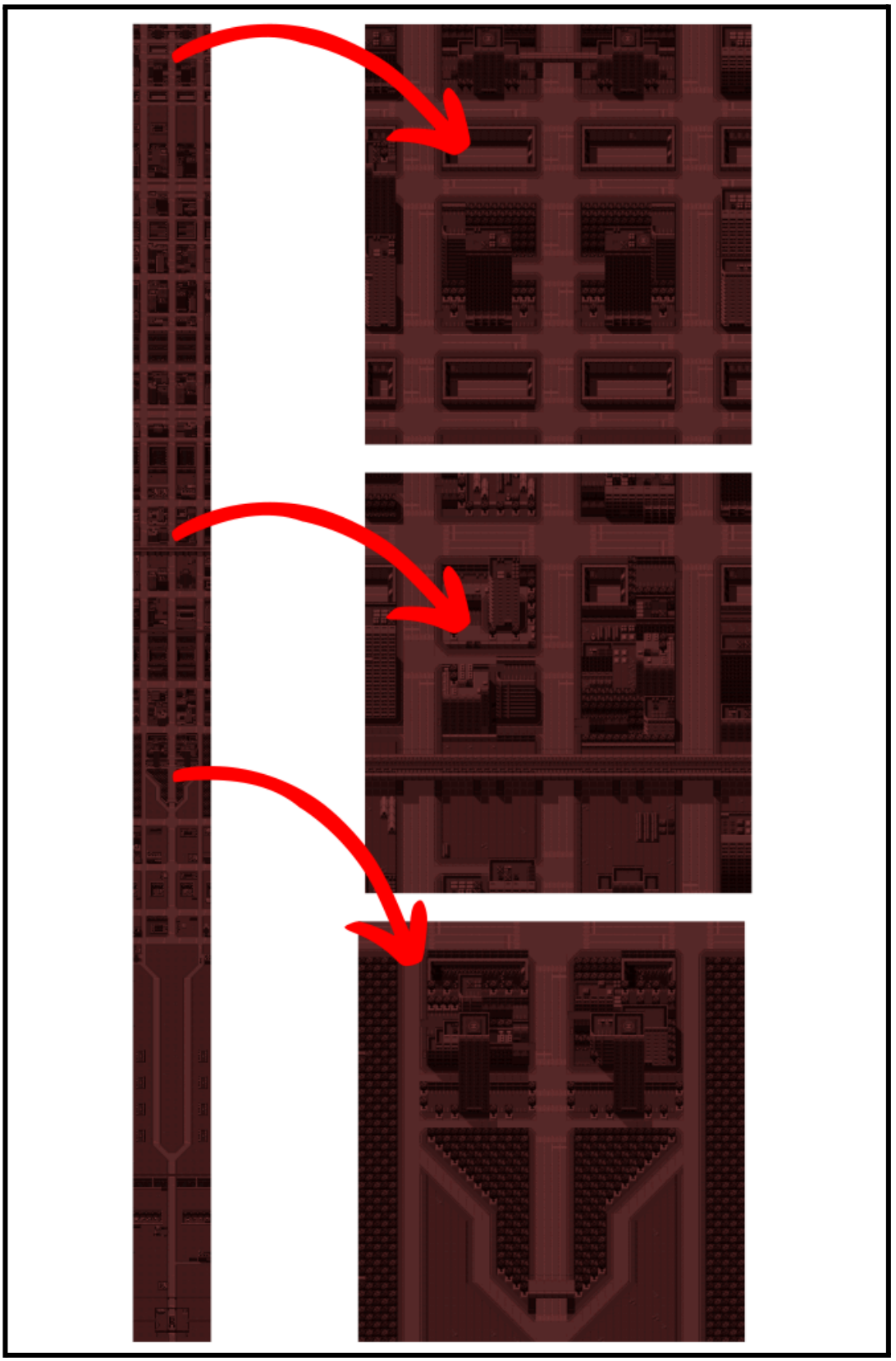
[15] A história do jogo, com seu enredo baseado em *cyberpunk*, assume que uma piloto acorda sem memória e seus sistemas de pilotagem, configurados por implantes, falharam. Isso daria início a uma tela de treinamento, na qual a piloto poderia recuperar suas habilidades.

Anexo 7 - Fase inicial do jogo, com caixa de diálogo para construção da narrativa

Anexo 7 - Uma cena de batalha com um *boss*[16]

[16] Repare que o background é o mesmo do anexo anterior: isso se deve a inserção do boss no início do mapa para fins de *debug*, por meio do posicionamento via arquivo JSON

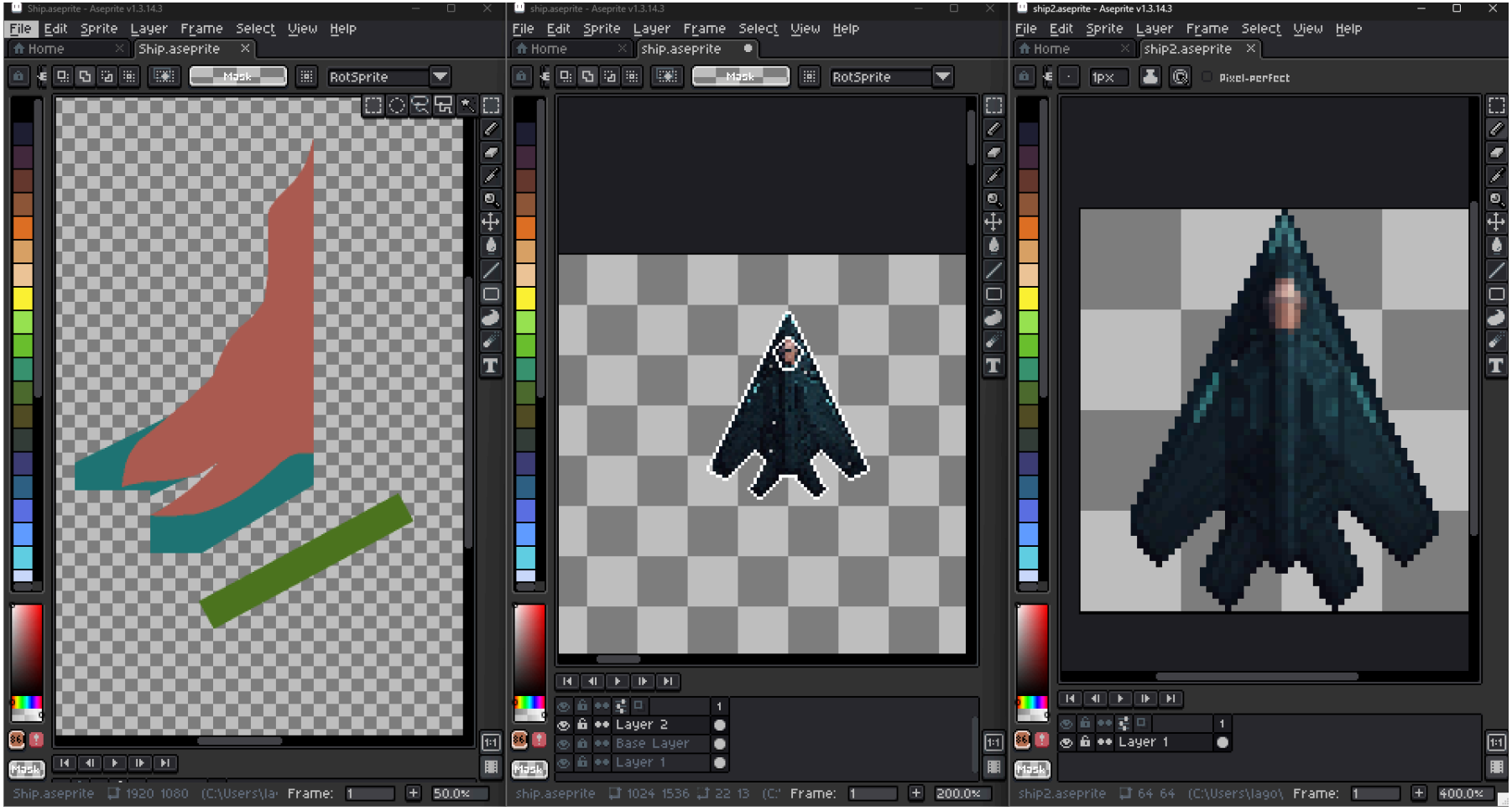
Anexo 8 - Level design, com algumas trechos de destaque do mapa



[17]

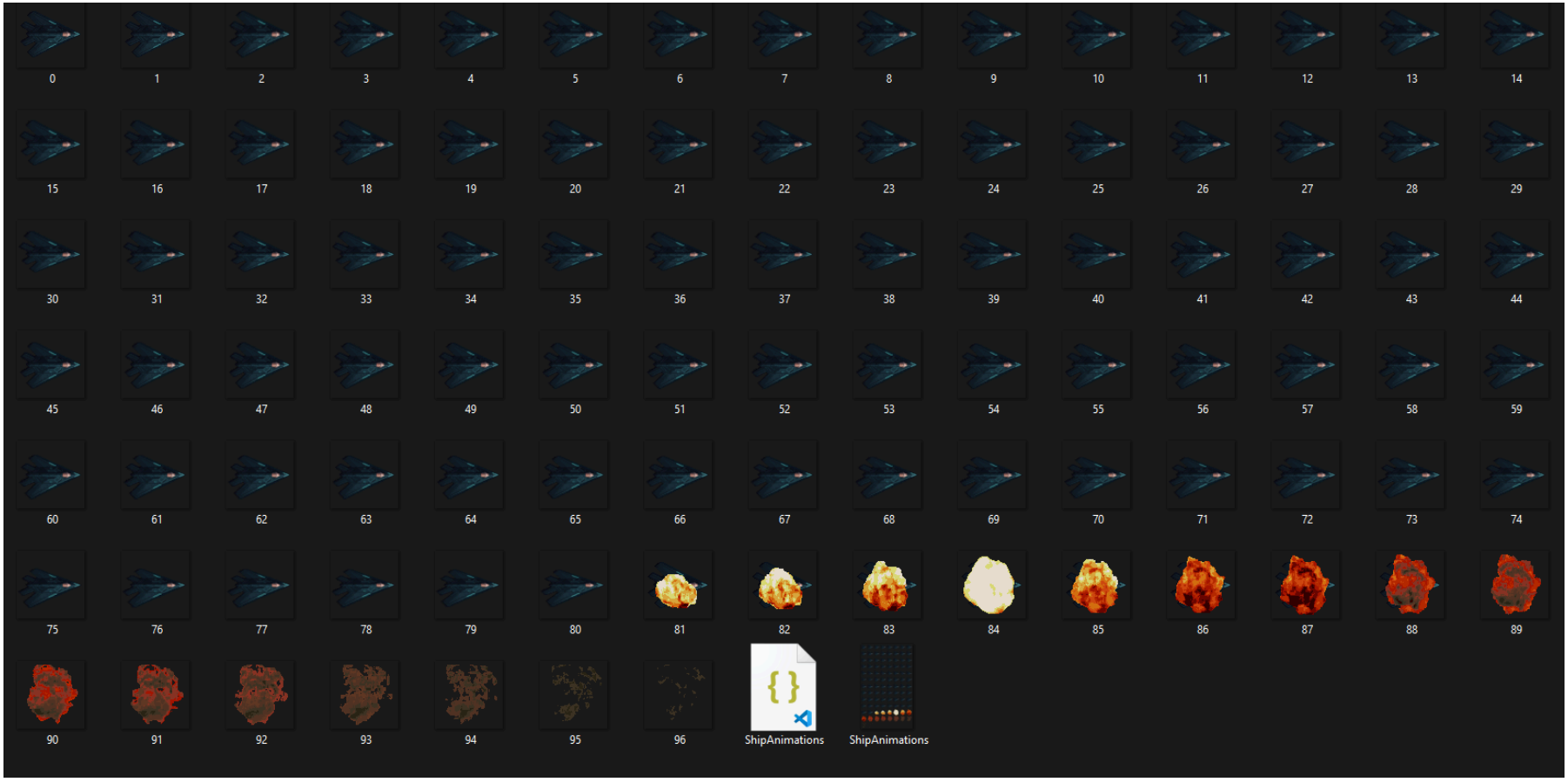
[17] O mapa foi projetado em sua extensão para uma gameplay de cerca de três minutos, tempo padrão para mapas de jogos do gênero. Para isso, o mapa possui dimensões de 1600 pixels por 27000 pixels.O posicionamento dos inimigos foi feito iterativamente: posicionava-se o inimigo, via arquivo json e executava-se o jogo, buscando verificar como se tornava a experiência.

Anexo 9 - Primeiros esboços da nave[18]



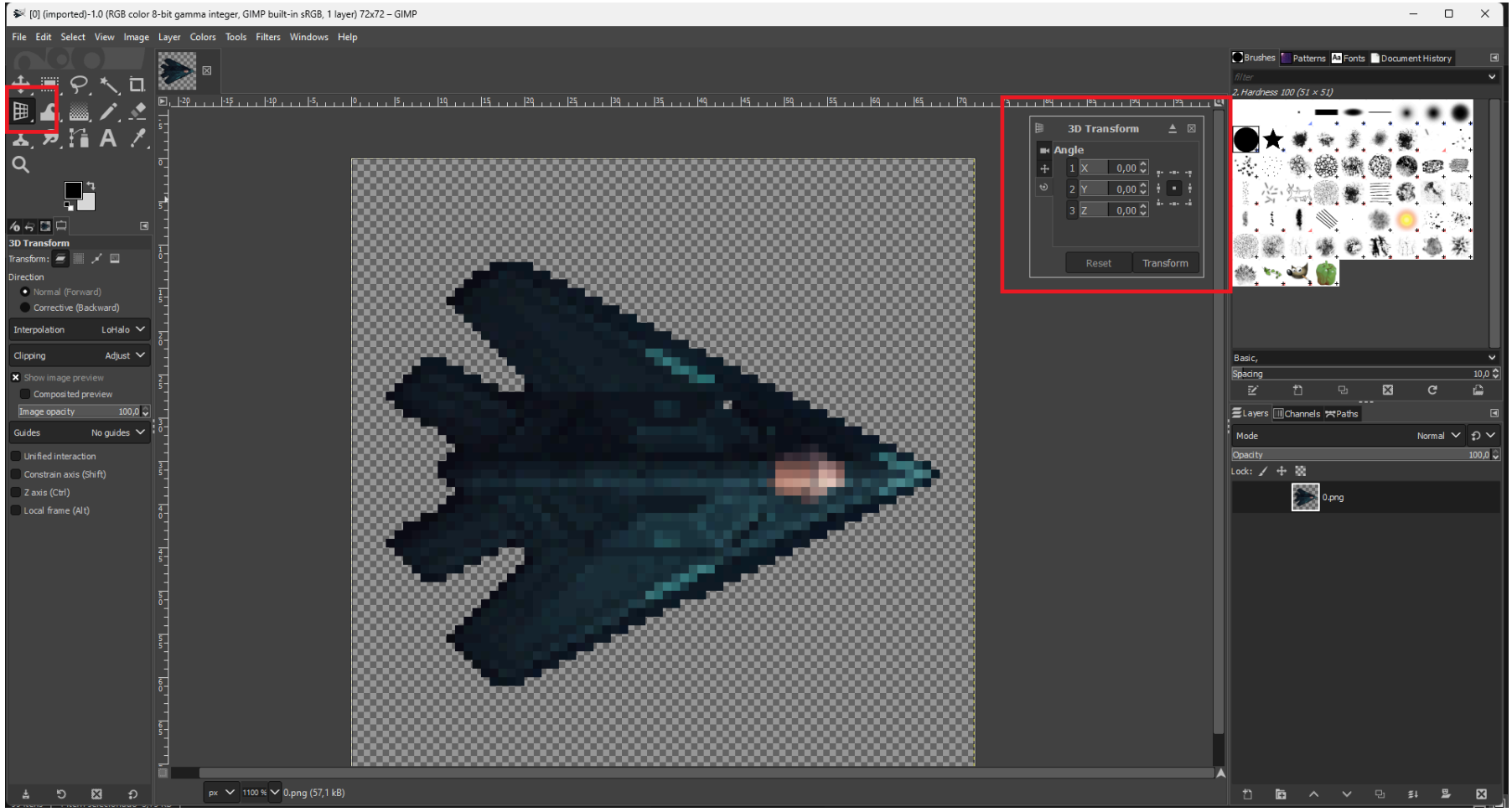
[18] Inicialmente, foi feito de forma espelhada, buscando apenas uma forma geral. Em seguida, trabalhou-se em um esboço de baixa resolução, inspirado nos aviões F-117 (aeronaves com tecnologia *stealth* de camuflagem). Todos os desenhos foram feitos no Aseprite, um software gratuito de edição de imagens pixel art.

Anexo 10 - Conjunto de imagens elaboradas via Gimp, para compor a sprite sheet[19]



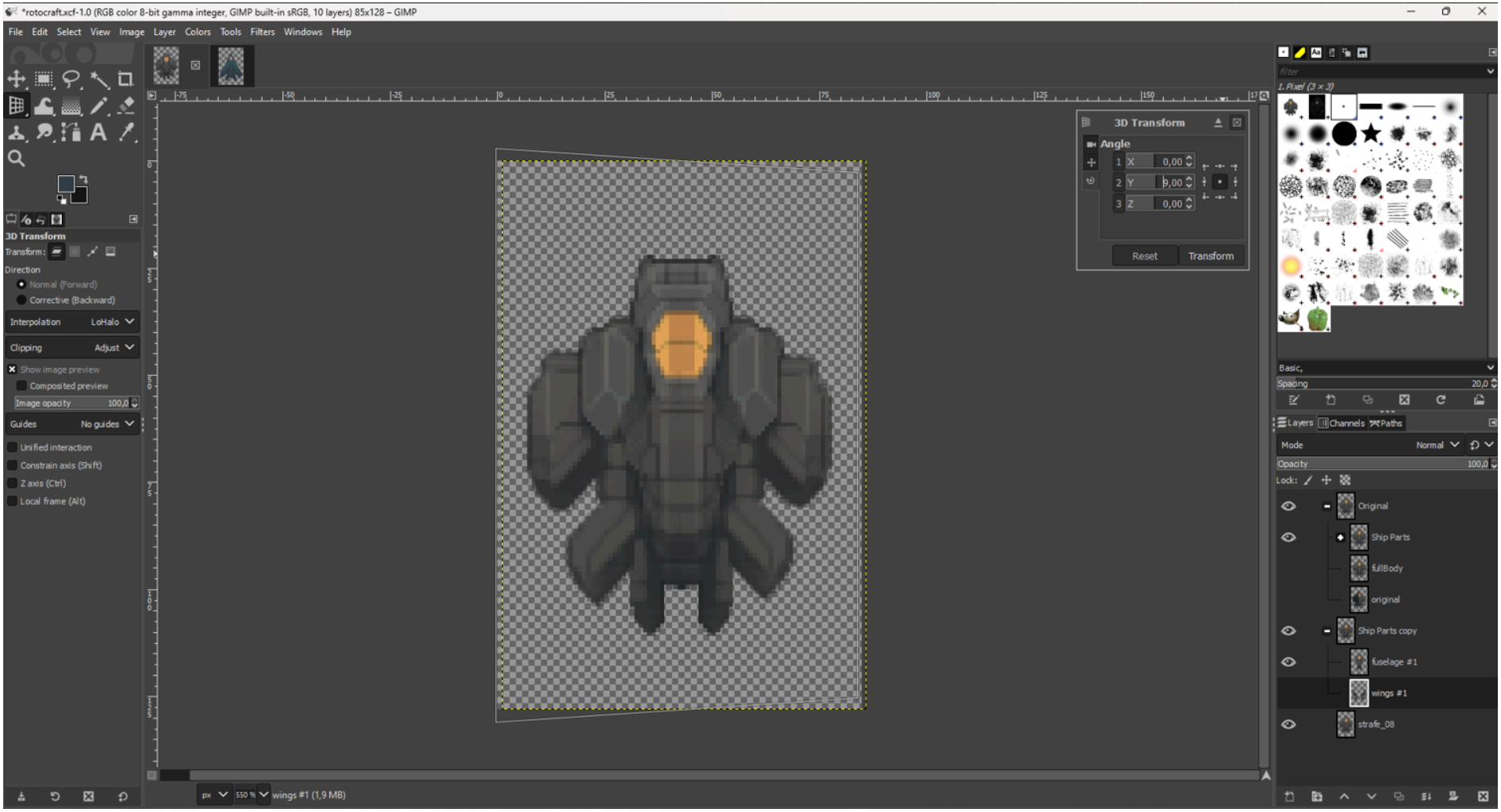
[19] O Gimp possui uma ferramenta chamada *3D transform tool* (ver anexo a seguir) que permite rotacionar imagens em três eixos (X, Y e Z). Para criar a animação de movimentação da nave para a esquerda e para a direita, foram criadas manualmente 40 imagens, rotacionando incrementalmente um grau no eixo X em cada uma (considerando que a nave é desenhada apontando para a direita). As imagens acima também incluem uma animação de explosão, gerada no Aseprite.

Anexo 11 - Elaboração do movimento de rotação da nave[20]

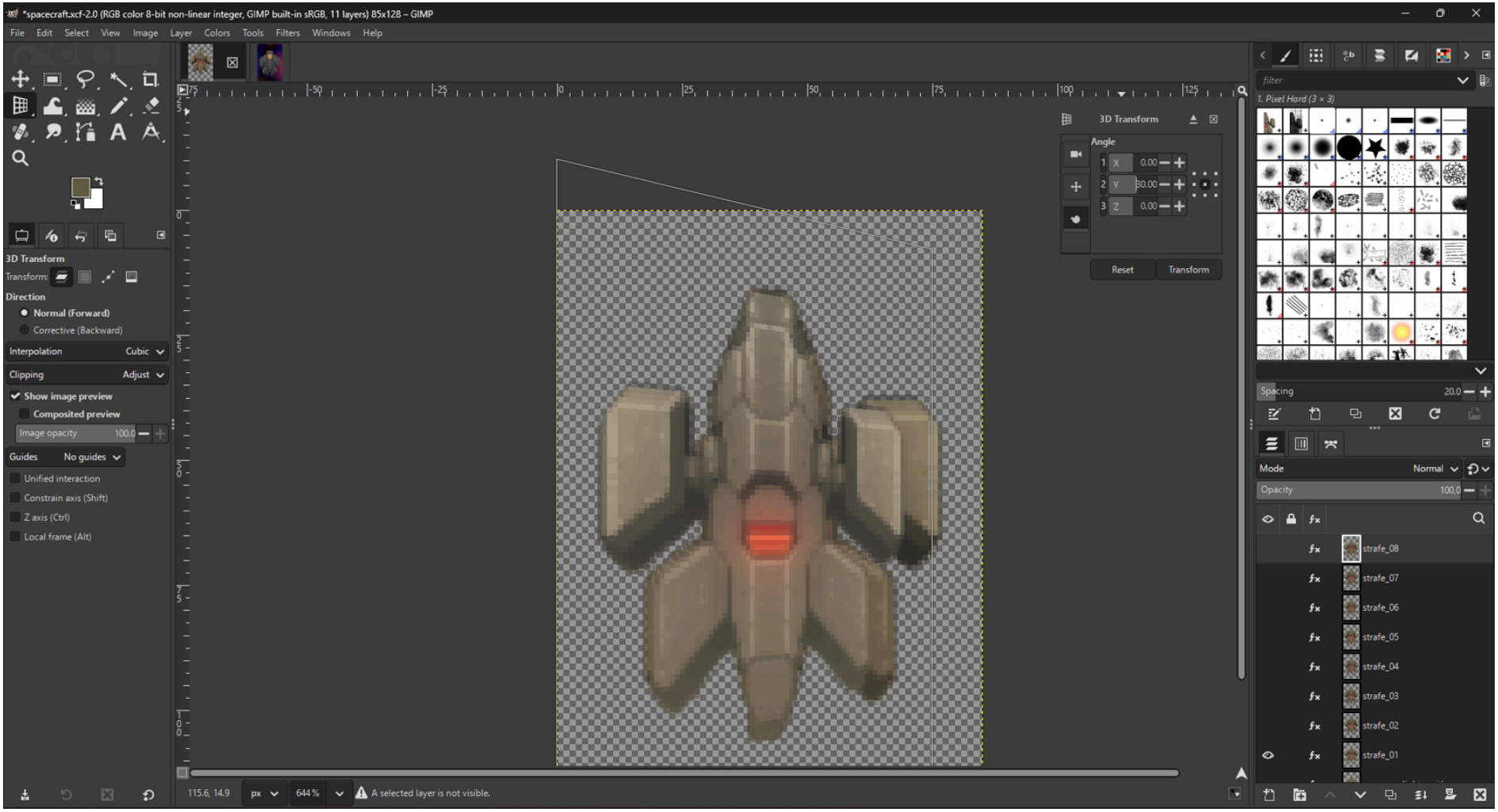


[20] Na imagem acima, as ferramentas de *3D transform tool*, marcadas em vermelho, foram usadas para rotacionar a nave e criar um efeito de movimento.

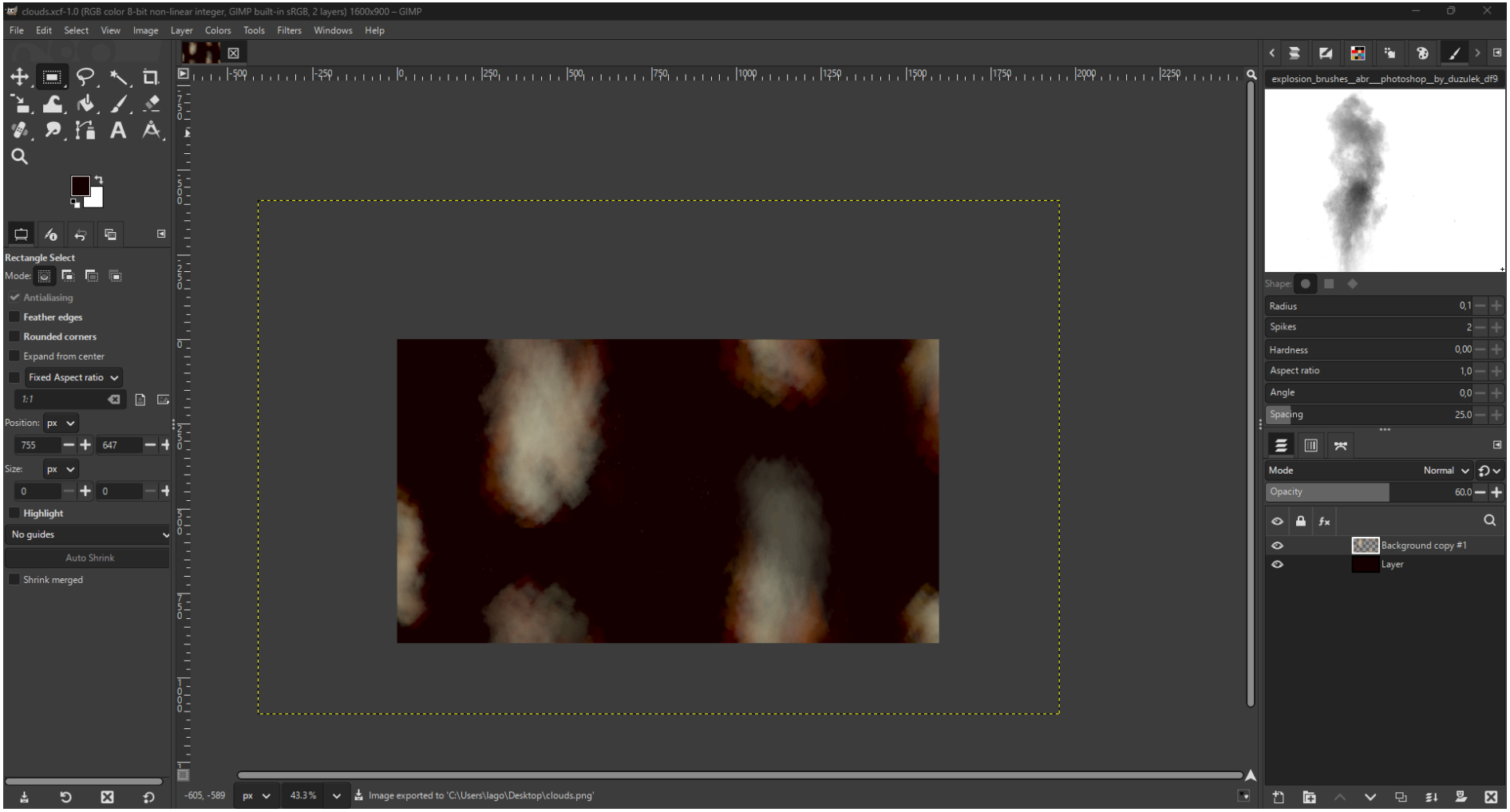
Anexo 12 - Exemplo de inimigo elaborado no GIMP



Anexo 13 - Exemplo de inimigo elaborado no GIMP, durante a produção da spritesheet de strafe

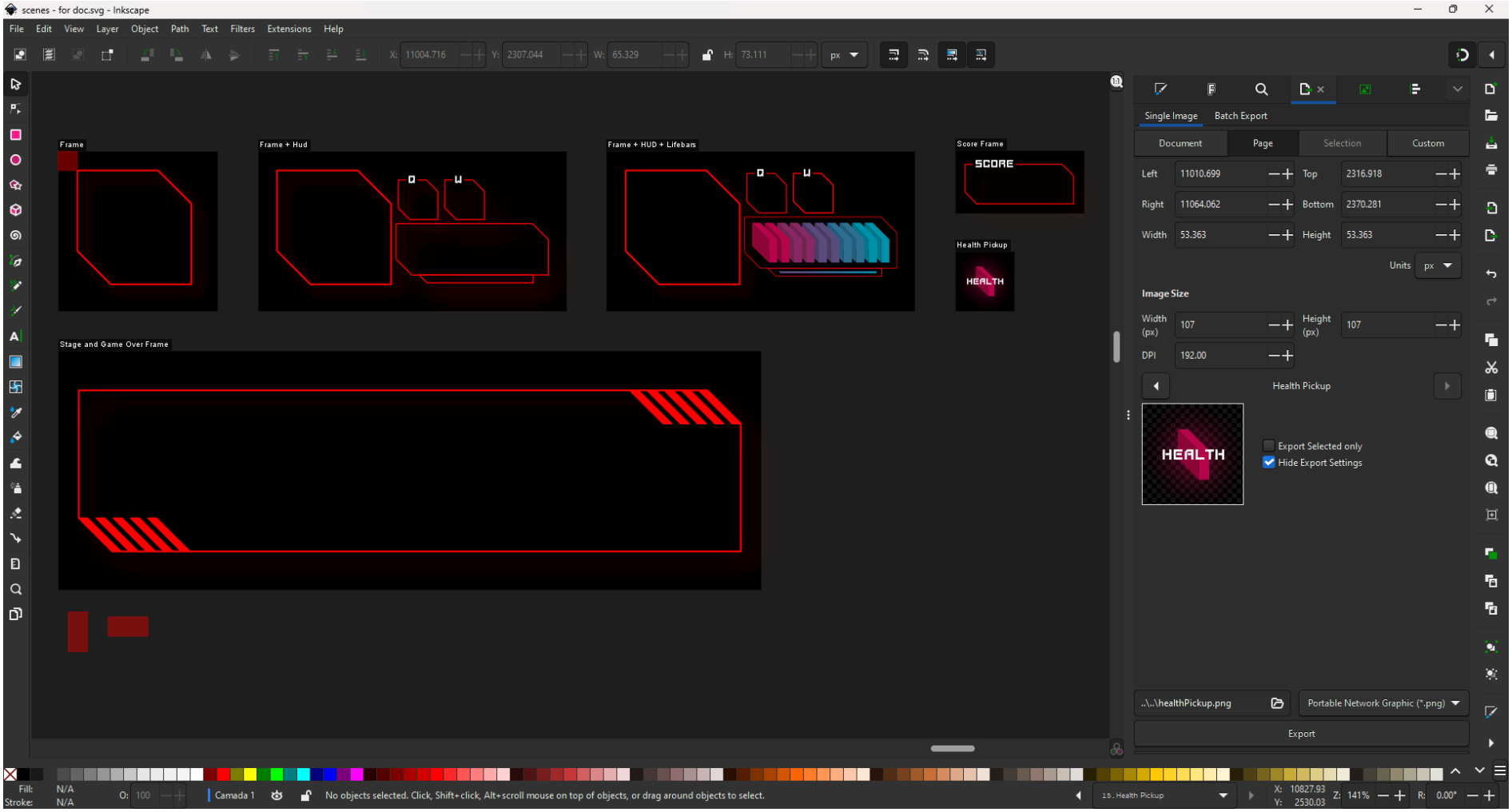


Anexo 14 - Textura de fumaça elaborada no GIMP[21]



[21] A imagem foi exportada sem o *background* preto. Para sua produção foi utilizado um *brush* de fumaça.

Anexo 15 - Frames usados no jogo elaborados no Inkscape[22]



[22] No caso dos frames foi feito um desenho vetorial direto, combinados a um filtro de glow, fornecido pela ferramenta, para o brilho no fundo dos frames.