INSTITUTO DE MATEMÁTICA E ESTATÍSTICA

UNIVERSIDADE DE SÃO PAULO

GIULIA CUNHA DE NARDI

O Pesadelo de Fluffy - Desenvolvimento de um jogo puzzle 3D

Trabalho de Conclusão de Curso apresentado para obtenção do título de bacharel em ciência da computação

São Paulo

2022

Giulia Cunha de Nardi

Graduando em Ciência da Computação

O Pesadelo de Fluffy - Desenvolvimento de um jogo puzzle 3D

Orientador:

Prof. Paulo Andre Vechiatto de Miranda

Trabalho de Conclusão de Curso apresentado para obtenção do título de bacharel em ciência de computação

São Paulo

2022

AGRADECIMENTO

Agradeço ao meu orientador por me dar um ponto de partida com o projeto, guiar nos passos que eu deveria seguir e ajudar tanto na geração dos gráficos e os consertando quando eu os quebrei.

Agradeço também meus professores que me ensinaram nessa jornada que foi a graduação.

RESUMO

A proposta deste trabalho foi o desenvolvimento de um jogo criando uma engine própria. O jogo criado (“O Pesadelo de Fluffy”) teve como ponto de partida a jogabilidade do mini-jogo Rapunzel (Catherine). Utilizando as bibliotecas SDL e OpenGL, foi desenvolvido o programa em C + + que possui todos os módulos necessários de uma engine para que o jogo funcione, desde o gerenciamento das entidades e eventos à geração dos gráficos.

O resultado foi um jogo puzzle 3D com uma mecânica e lógica similar ao jogo-base, mas com elementos inovadores como tipos diferentes de blocos e formas variáveis de se completar uma fase.

Palavras-chave: jogo, desenvolvimento, puzzle, 3D, OpenGl, SDL, C + +, engine, Fluffy

SUMÁRIO

São Paulo

2022

INTRODUÇÃO

Breve História do Desenvolvimento de Jogos Digitais

Os primeiros jogos digitais eram pouco mais que loops de eventos, tabelas de estado e as rotinas gráficas necessárias para jogos 2D como Space Invaders (Figura 1) e Raptor (Figura 2). Em 1993 Doom (Figura 3) foi lançado pela id Software e iniciou uma nova era de games.

Figura 1: Screenshot de Space Invaders (1978)

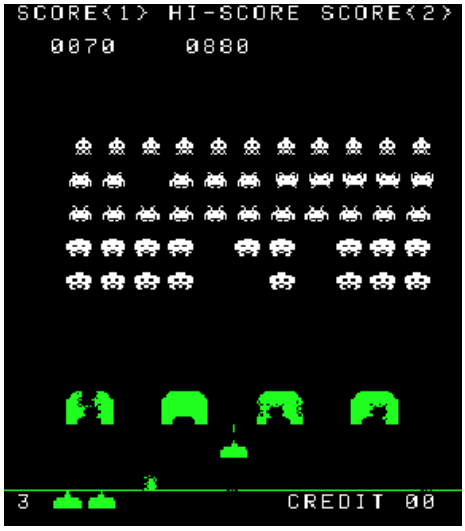


Figura 2: Screenshot de Raptor (1994)

Doom não foi o primeiro jogo a oferecer uma jogabilidade em primeira pessoa imersiva, mas sim o primeiro a ter sucesso nisso. Ele conseguiu oferecer um senso de realismo mesmo em computadores extremamente limitados como os da época (computadores com processador Intel i486 por exemplo, com apenas 8KB de RAM) (Wikimedia 2005). Esse senso de realismo se baseou no uso de mapas de texturas, animações 2.5D e outros truques de programação que proporcionam uma experiência melhor que possível por meio da força bruta. Outro ponto iniciado por Doom foi o modo de jogo multiplayer por uma rede de forma peer-to-peer (os sucessores depois trocaram para uma arquitetura cliente/servidor).

O que Doom iniciou, outros jogos, como a franquia Quake (Figura 4), continuaram e evoluíram.

Figura 3: Screenshot de Doom (1993)

Figura 4: Screenshot de Quake (1996)

Quake tem um ambiente verdadeiramente 3D com o personagem e polígonos texturizados, tendo sido necessário uma divisão otimizada nas filas de renderização entre entidades e ambiente para que tudo funcionasse com velocidade suficiente a não prejudicar a experiência do jogador. Quake também adotou a arquitetura cliente/servidor deixando ao cliente toda a parte computacionalmente cara (3D engine, por exemplo) e ao servidor apenas as tabelas de estado do jogo. Outra evolução que vale comentar foi a extensão no código para criar a primeira engine independe do jogo tanto para edições no layout quanto para mudanças nos comportamentos das simulações. (Lewis and Jacobson 2002).

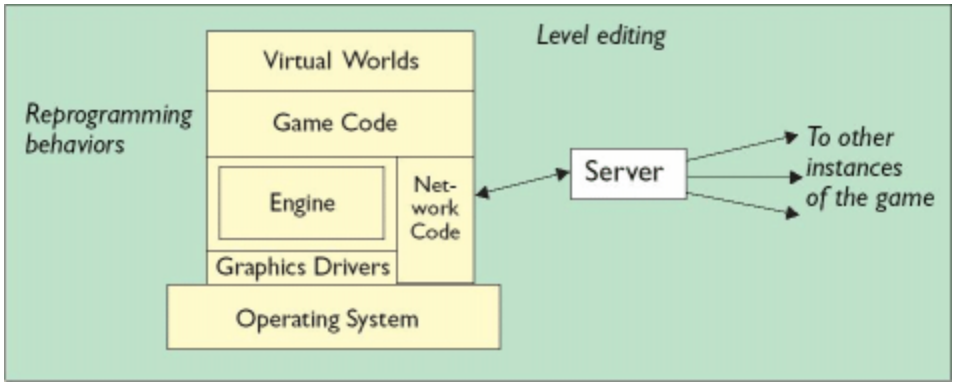
Estrutura de um Jogo

Atualmente, jogos são construídos com formato modular, nessa ideia, uma *game engine* são todos os módulos de código que não especificam diretamente o comportamento do jogo (lógica do jogo) ou ambiente do jogo. Na engine inclui o gerenciamento de input, output (renderização dos gráficos 3D/2D e sons) e gerenciamento da física/dinâmicas dentro do jogo.

Essa estrutura modularizada é ilustrada na Figura 5 (e com mais detalhes depois na sessão Jogo Criado - Desenvolvimento).

Outra definição é que *Game Engines* são *frameworks* de software desenvolvidas primariamente para o desenvolvimento de jogos digitais e geralmente inclui bibliotecas e suporte para programas. O termo “engine” é uma terminologia similar ao termo “software engine” utilizado na indústria de tecnologia (Wikimedia 2022).

Figura 5: Estrutura Modular de uma Game Engine



Fonte: (Lewis and Jacobson 2002)

Ilustrado na figura estão os leveis:

- No topo o “Mundo Virtual” ou cenários que o jogador interage, a aparência varia assim como as regras de interações e físicas;
- Abaixo está o código do jogo em si, lida com a maior parte das mecânicas e jogabilidades do jogo. Inclui físicas básicas, animações e parâmetros do jogo (como leveis e pontos);
- A Game Engine é o coração de todo jogo, nela é incorporado toda parte complexa de código necessária para renderizar o que o jogador vê e os modelos 3D do cenário. Ela é a “caixa preta” do projeto e muitas vezes não permite qualquer modificação pelo usuário;

- A rede (network code) tem suporte para os protocolos de rede proporcionando acesso remoto e a possibilidade que jogadores interajam no mesmo ambiente simulado com um computador (ou servidor) agindo como “host”;
- Os drivers de gráficos são uma forma genérica de se referir às bibliotecas gráficas, exemplos são DirectX e OpenGL.

(Lewis and Jacobson 2002)

JOGO INSPIRAÇÃO

Catherine

Catherine é um jogo eletrônico de plataforma no estilo quebra-cabeça, terror e aventura desenvolvido pela Atlus e publicado por ela em 2011 para as plataformas PlayStation 3 e Xbox 360. No mesmo ano, foram vendidas 500 mil cópias, o que tornou o jogo um dos maiores sucessos da companhia (Wikimedia 2012). O jogo ainda tem uma base fiel de fãs o que contribuiu para que fosse lançado novamente em 2019 numa versão mais completa para Microsoft Windows e distribuído pela Steam (Steam 2019).

O jogo trouxe uma variedade de reações e polêmicas devido aos temas de sua história que envolve infidelidade e insinuações sexuais, apesar disso, manteve uma nota de 82/100 (Metacritic 2011) no lançamento original. No novo lançamento a nota é de 9/10 na Steam com quase 6 mil análises, segundo o site, a maioria classifica-se como “Muito positivas” (Steam 2019).

Quanto à jogabilidade do jogo em si, Catherine envolve duas dinâmicas principais - quando o personagem está acordado controlando suas ações, tendo que lidar com as consequências de suas escolhas que impactam o rumo e final do jogo. E quando o personagem está no sonho - resolução rápida do puzzle de empurrar e puxar blocos escalando para sobreviver mais uma noite.

Rapunzel (Mini-jogo)

Rapunzel, é o mini-jogo estilo fliperama dentro do Catherine que imita a jogabilidade do mesmo. A história de Rapunzel é baseada no conto de fadas do mesmo nome.

No jogo a ideia é a mesma do Catherine: arranjar os blocos para alcançar o objetivo no topo do cenário. Entretanto o que diferencia Rapunzel são as regras da jogabilidade (Fandom n.d.):

- não há tempo limite para completar as fases e o jogador não é recompensado (em pontos) por terminar a fase em tempos menores;
- há um número limitado de ações que o jogador pode fazer. Ações são apenas movimentos de blocos (puxar e empurrar) e não inclui movimentações do Player;
- há uma pequena variedade de tipos de blocos em comparação com o Catherine que possui enorme variedade;

Essas regras junto com o fato que as fases são menores e complexas, tornam Rapunzel um jogo mais baseado em estratégia que o próprio Catherine.

Figura 6: Captura de tela do jogo Rapunzel desenvolvido pela Atlus.

Fonte: (Team 2011)

Figura 7: Captura de tela do jogo Catherine desenvolvido pela Atlus. Ilustra como o jogador acessa o mini-jogo.

Fonte: (Fandom n.d.)

JOGO CRIADO

O Pesadelo de Fluffy - Proposta

Ao escolher a criação de um jogo criando uma engine própria como tema para esse trabalho, selecionamos um jogo com uma mecânica e ideia que queríamos recriar e aprimorar. Assim, criamos “O Pesadelo de Fluffy” baseado em Rapunzel (mini-jogo do jogo Catherine).

“O Pesadelo de Fluffy” é um jogo puzzle de escalada 3D cujo objetivo é chegar ao topo de um cenário composto por blocos. Para isto, o jogador deve puxar e/ou empurrar blocos para criar escadas. Existem, no entanto, diferentes tipos de blocos que dificultam o progresso do jogador (por exemplo, blocos fixos que não podem ser movidos e *“checkpoints”* que devem ser passados).

O personagem principal é o Fluffy, o mascote da Atlética do IME-USP e o nome “O Pesadelo de Fluffy” faz referência ao jogo que nos baseamos, Catherine (no qual o personagem está em um pesadelo).

Jogabilidade

O jogador comanda um personagem que pode se mover nas quatro direções (para frente, para trás, para direita e esquerda) e interagir com blocos (empurrar, subir e se pendurar nas bordas).

O objetivo do jogador é passar por todos os blocos amarelos no topo da torre composta por blocos de tipos variados. Para isto, ele deve puxar e/ou empurrar blocos para criar escadas. Existem no entanto diferentes tipos de blocos que dificultam o progresso do jogador (por exemplo, blocos fixos que não podem ser movidos).

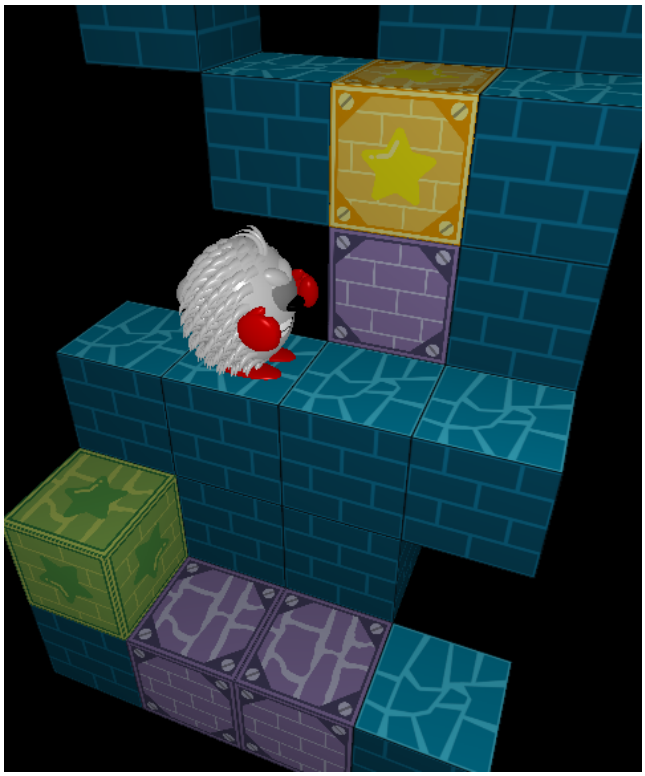


Figura 8: Captura de tela de “O Pesadelo de Fluffy”. Na imagem se vê o Player, Torre e diferentes tipos de blocos

Há dois tipos de blocos com relação a gravidade: móveis (sujeitos a gravidade) e fixos (não se movimentam, imunes a gravidade). Blocos móveis apenas se movem quando atualizados e quando isso acontece cada bloco checa se tem suporte ou se deve cair.

Para ter suporte um bloco deve ter um outro bloco diretamente abaixo ou ter alguma de sua aresta tocando outro. Em outras palavras, um Bloco na posição (x, y, z) cai se não houver nenhum bloco nas posições (x, y-1, z), (x-1, y-1, z), (x+1, y-1, z), (x, y-1, z-1), (x, y-1, z+1). Sendo y “altura”.

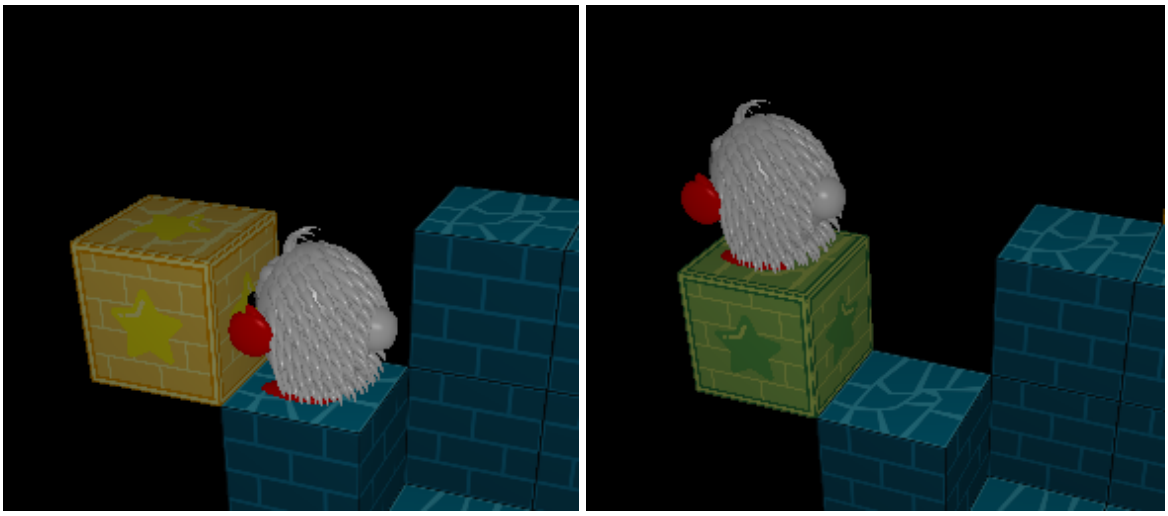
Um bloco é atualizado em dois casos: (I) quando o player puxa ou empurra o bloco e quando (II) o andar em que ele está é atualizado.

O caso II apenas acontece após o caso I, isso é, se o player puxa ou empurra um bloco ele causa uma atualização no andar acima, se, devido a essa atualização, um bloco cair, o

andar acima também é atualizado. Assim, após uma ação, uma série de atualizações procede e todo o cenário fica sujeito a mudanças.

Para completar uma fase todos os blocos amarelos devem se tornar verdes. Um bloco amarelo se torna verde quando o personagem passa por ele.

Figura 9: Bloco amarelo se tornando verde



Desenvolvimento

O desenvolvimento de jogos na maioria das vezes se dá por meio de programas auxiliares comumente chamados de *engines*, esses programas facilitam todo o processo de criação e implementação, mas muitas vezes vem com o custo de performance e limitações nas mecânicas do jogo criado. Considerando isso, resolvemos abraçar o desafio de criar um jogo sem o uso dessas engines (resolvemos fazer a nossa).

O programa foi desenvolvido em C + + com o uso das bibliotecas OpenGL (renderização) e SDL (input e output).

A escolha em utilizar C + + está na enorme compatibilidade com as bibliotecas utilizadas, facilidade em trabalhar com classes (POO) e o potencial da linguagem em

performance (por ser de baixo nível é possível manipular com precisão os dados sem prejudicar o tempo de processamento).

OpenGL é uma biblioteca de renderização. Ele não mantém nenhuma informação sobre o que renderiza, tudo que ele vê é um conjunto de triângulos e estados para renderizá-los. (Khronos n.d.)

A partir da criação desses triângulos é que podemos criar figuras, usando funções que, por fórmulas geométricas, criam figuras 3D como cubos, pirâmides e cones (chamadas primitivas).

SDL é uma biblioteca de desenvolvimento criada para dar acesso de baixo nível ao áudio, mouse, teclado e gráficos do computador. Desenvolvida em C, tem compatibilidade nativa com C + + e é muito popular em jogos e emuladores. (“Simple DirectMedia Layer” n.d.)

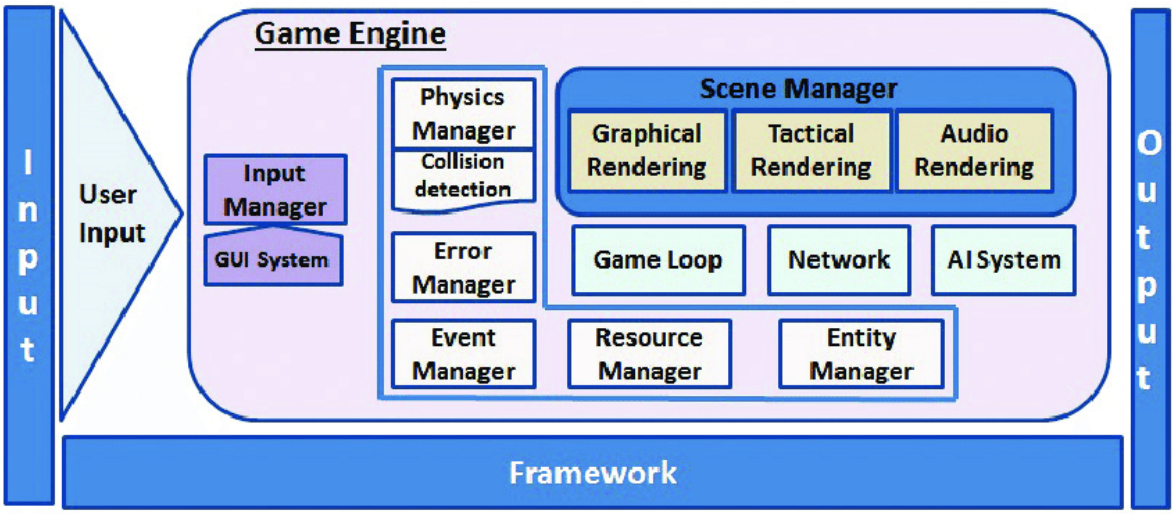
Arquitetura

O jogo criado foi dividido em 3 arquivos de código:

- game.cpp : contém o jogo em si;
- entidades.cpp : contém os objetos;
- drawFunctions.cpp : contém as rotinas de desenhos das figuras;

Essa foi uma divisão lógica para que houvesse clareza no desenvolvimento sem a criação exaustiva de vários arquivos. Apesar disso, ainda é possível ver as relações com a estrutura modular de um jogo (apresentada de forma genérica na Introdução). Para facilitar a explicação dessa relação, na figura abaixo (Figura 10) tem-se em mais detalhes os módulos.

Figura 10: Arquitetura detalhada de uma *Game Engine*



Fonte: (Zarrad 2018)

O arquivo “drawFunctions” é, em sua função, o “Graphical Rendering” . Gerenciamento da física, detecção de colisão e gerenciamento de entidades são papéis que, na implementação, a programação orientada a objetos resolve, os objetos estão presentes em “entidades”. Os papéis restantes são tratadas por funções no arquivo principal “game”.

Alguns exemplos de funções e seu papel:

- handle_key: Input Manager
- draw_screen, draw_menu: GUI System
- process_events: Event Manager
- main: Game loop

Há também as diferenças entre o desenvolvido e o modelo, como a ausência de uma Network e Sistema IA. Isso se deu devido ao projeto não ter como alcance uma jogabilidade multiplayer nem NPCs (non playable character ou personagens não controláveis).

Código

Há duas estruturas vitais dentro do programa principal, elas são “Tela” e “GameData”. “Tela” possui as informações para a visualização dos menus e “GameData” para o funcionamento da fase (contém os objetos e ListaUpdate).

O programa consiste de dois loops, o principal que mantém o jogo aberto e o secundário que é a jogabilidade dentro de uma fase. É o loop secundário que requer o maior processamento pois é onde o jogo realmente está e onde os objetos interagem.

No loop principal uma variável “estadoJogo” direciona o que deve estar sendo apresentado ao jogador, isso é, qual tela (menu, instruções, entre outras) (Figura 11) ou se deve iniciar o loop secundário (fase).

Figura 11: Tela Inicial do jogo, é o apresentado quando *estadoJogo = Menu.*

No loop secundário é onde a fase acontece. Isso se dá pelas interações entre os objetos que por sua vez dependem do cenário e do input do jogador. O input do jogador causa uma mudança no objeto Player e se esse objeto faz uma *ação* ele causa uma mudança no cenário (objeto Torre) (ver Figura 8).

Uma *ação* é apenas quando o jogador puxa ou empurra um bloco. Quando isso acontece, 4 objetos estão envolvidos: Player, Bloco, Torre e Desfaz.

O Player muda o estado de sua animação que significa que ao renderizá-lo, o programa chamará uma função específica para aquela animação e guardará uma variável para saber o andamento dessa animação para que a cada iteração consiga progredir.

O Bloco recebendo a ação é ejetado da Torre e posto em movimento tendo sua velocidade e estado ajustados, após ser ejetado ele entra na ListaUpdate onde terá sua posição e colisões atualizadas a cada iteração. Quando houver uma colisão que interrompa seu movimento ou o destrua (sair do alcance da Torre) o objeto sai dessa lista e, se não for destruído, volta para a Torre.

A Torre ejeta o Bloco que recebeu a ação e atualiza o Andar acima desse Bloco checando por Blocos sem suporte (ver Jogabilidade). Se algum Bloco não tiver suporte ele é ejetado entrando na ListaUpdate em estado de queda e causa uma atualização no Andar acima dele.

O Desfaz cria uma instância da Torre e do Player antes da ação, habilitando a mecânica de desfazer uma ação presente no jogo.

Nem todos os Blocos, entretanto, recebem updates. O tipo de um Bloco implica se ele pode receber ou não update (se é fixo ou móvel) e se a fase foi ou não completa (blocos amarelos).

Física - Do Discreto ao Contínuo

A primeira versão do jogo foi uma versão discreta, isto é, cada ação causava uma reação instantânea. Num espaço discreto os objetos se teleportam e não há animações.

Essa é uma forma bem simples de resolver as mecânicas e jogabilidades, mas é inviável pela simples estranheza e desconforto que é jogar um jogo assim. Um jogo precisa das animações, é necessário ver os passos intermediários (mesmo que para isso eles devam ser criados). Por isso o jogo evoluiu para um espaço contínuo.

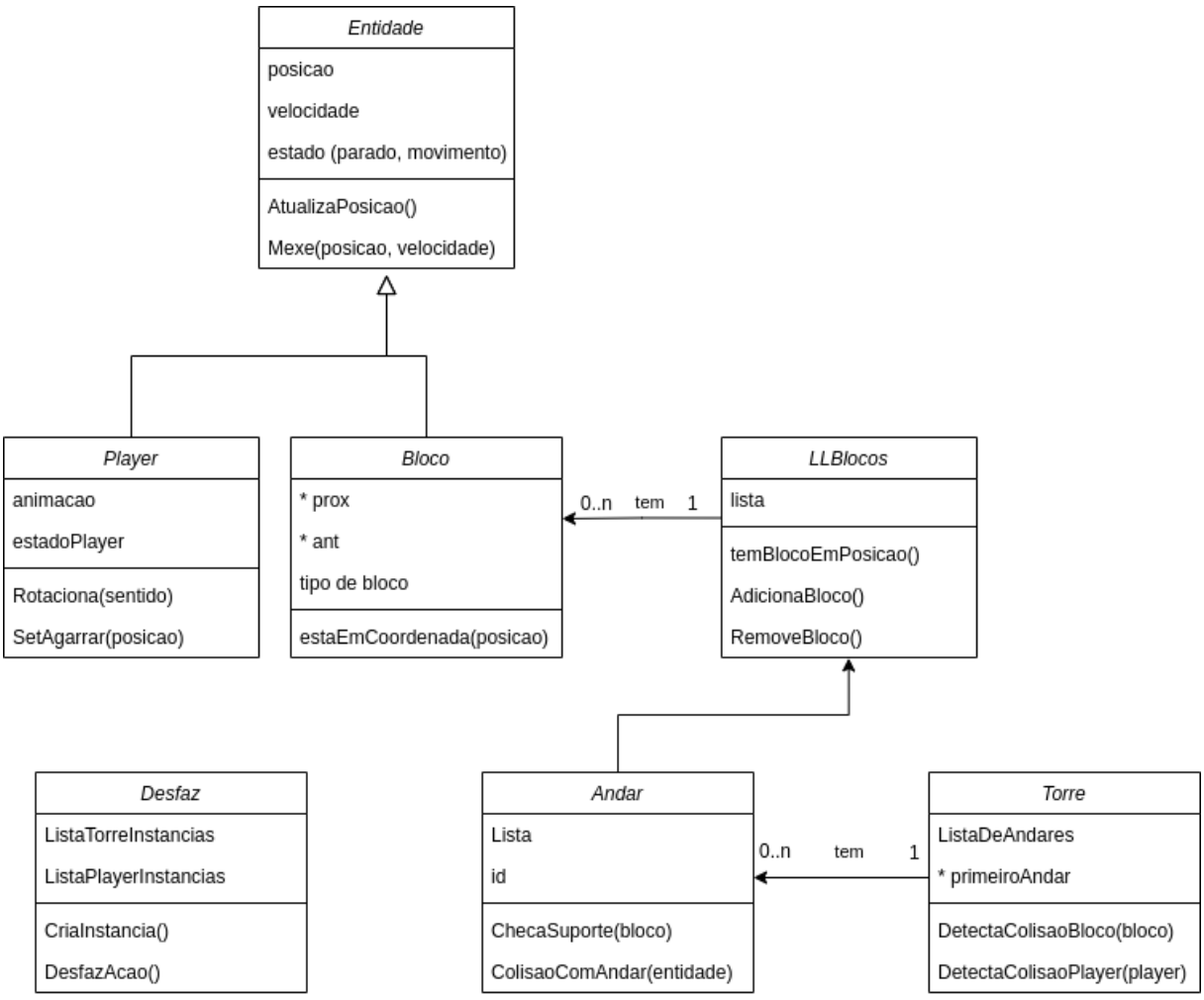
A forma de fazer um espaço contínuo é trazer realismo, os objetos devem ter posições e estarem sujeitos a interações com outros que causem que suas posições mudem ao decorrer de um tempo. Por simplicidade, não foi implementado toda uma física com conceitos como vetores força e aceleração, apenas o necessário para o jogo funcionar, que foi o conceito do vetor velocidade e posição como um ponto 3D.

Essas mudanças causaram uma reescrita nos objetos e a inclusão de estados nas entidades (se um objeto está parado ou em movimento) já que para a jogabilidade, apesar de ser contínuo, os objetos não estão em real liberdade e não podem ser interrompidos no meio de uma animação ou quando estão entre “coordenadas inteiras”.

Objetos e Estrutura de Dados

Na figura abaixo está o esquema de classes (Figura 12).

Figura 12: Esquema de classes dos objetos do arquivo Entidades



Blocos

É o objeto principal do cenário e, de certa forma, o único já que a Torre e Andar, apesar de serem objetos e terem funções próprias, são apenas conjuntos de Blocos. Possui tipos que modificam suas interações com o Player e cenário.

Tipos de Blocos:

- Móvel: sujeito a gravidade e pode ser empurrado/puxado pelo Player;
- Fixo: imune a gravidade e não pode ser empurrado/puxado pelo Player;
- FinalFixo: o mesmo do Fixo, mas quando o Player está sobre ele, ele se transforma em FinalFixoCompleto;

- FinalFixoCompleto: o mesmo do Fixo, mas para que uma fase seja completada todos os blocos de tipo “Final” devem se tornar “Completos”;
- FinalMovel: o mesmo do Móvel, mas quando o Player está sobre ele, ele se transforma em FinalMovelCompleto;
- FinalMovelCompleto: o mesmo do Móvel, mas para que uma fase seja completada todos os blocos de tipo “Final” devem se tornar “Completos”;

Torre e Andar

O cenário visto pelo jogador é a Torre. A Torre é composta por Andares, cada Andar tem uma lista ligada de Blocos que é implementada pelo objeto LLBlocos.

Inicialmente a representação do cenário foi por meio de uma matriz 3D, entretanto, percebendo como o cenário era composto principalmente por espaços vazios e a medida que ele crescia muito mais espaço era reservado do que necessário fizemos a mudança para que apenas os espaços ocupados fossem guardados e isso se deu pela escolha em utilizar uma lista ligada para cada andar.

Player

O objeto de mais complexidade junto da Torre. Possui três sets de estados, um referente ao comum das entidades (se está em movimento, parado ou caindo), estados próprios do Player (morto, pendurado, normal ou tentando pendurar) e estados de animação (normal, pendurado, pendurado para esquerda, pendurado para direita, empurrando, puxando, andando).

Também é o único objeto que rotaciona e para isso tem funções variáveis para que essa informação seja mantida. Vale mencionar que é possível rotacionar a câmera e criar a

ilusão do cenário rotacionando, mas como a orientação do cenário em relação às coordenadas não muda, não é considerado uma rotação verdadeira.

Câmera

Não foi implementado como um objeto, mas poderia. Foi feito como parte da estrutura do “GameData” (ver Código), guarda apenas 4 variáveis (dois float referentes aos parâmetros de zoom e inclinação, dois “int” referentes às coordenadas centralizadas na tela).

Geração de Gráficos

Todos os gráficos do jogo são criados a partir de figuras primitivas e texturas projetadas nos planos dessas figuras. O objeto mais complexo criado é o player, mas até ele é um conjunto de primitivas (Figura 13).

Essa forma de gerar objetos gráficos é uma forma mais simples que mantém todo o processamento dentro do próprio código sem uso de arquivos exteriores. Um pró desse método é que toda compilação e geração da estrutura é confinada ao código, o que significa uma compilação e renderização mais simplificada. Um contra é que a complexidade e detalhes que se pode alcançar é limitada.

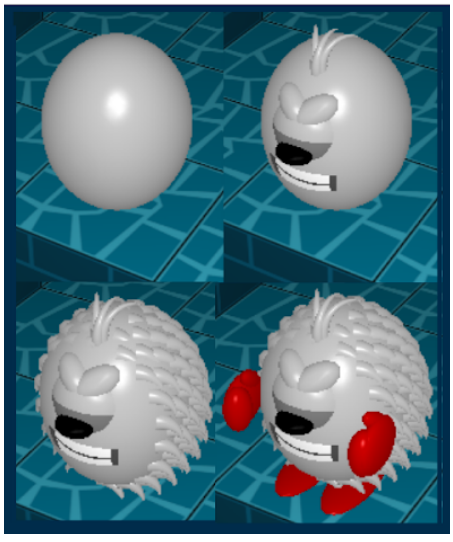


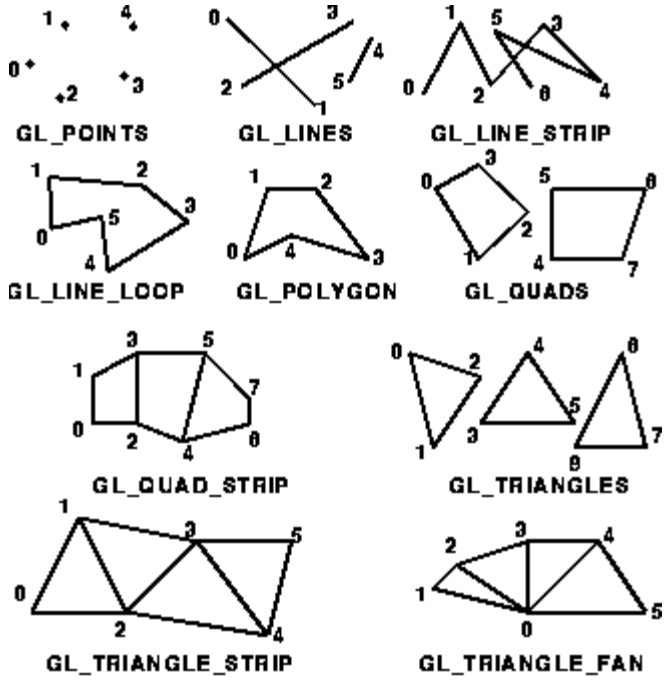
Figura 13: Ilustração das diferentes etapas de formação do Player.

OpenGL e Primitivas

Há dois conceitos de primitivas nesse contexto:

1. Funções primitivas são funções cujas rotinas criam os pontos que formam formas primitivas (como cubos, cones, pirâmides etc);
2. Primitivas dentro do OpenGL que indicam como a aplicação deve conectar os pontos no buffer. Na Figura estão as primitivas e as formas que geram.

Figura 14: Primitivas básicas presentes no OpenGL



Fonte: (“OpenGL Primitives and Colours” 2001)

Neste trabalho, o termo “primitivas” se refere às funções primitivas. Foram desenvolvidas primitivas que criam cubos, cones e cilindros foram desenvolvidos.

3D e 2D

Há momentos do jogo que a visualização é 2D (menus) o que trouxe a necessidade de implementação que suportasse isso.

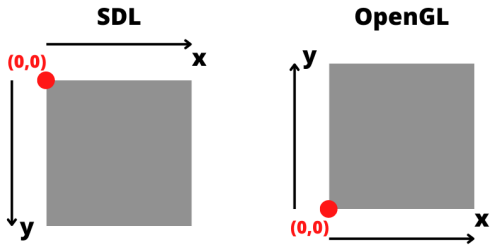
Inicialmente foi implementado uma função que utilizava a Biblioteca SDL para criar uma sobreposição de texturas de forma simples, entretanto, apesar de funcionar em um dos computadores testados, não funcionou no outro devido a uma questão de compatibilidade com o versões do OpenGL. Para contornar esse problema resolvemos usar exclusivamente o OpenGL para os gráficos, o que é um desafio em si só já que o OpenGL trabalha com o espaço 3D.

A solução foi criar e mostrar os menus num espaço 3D, mas com o ponto de vista que cria a ilusão de um espaço 2D, isso é, colocar o observador ortogonal à figura e zerar o coeficiente de distorção que a distância causaria. Quando um objeto está na frente do outro, ele é colocado numa posição mais próxima do observador para dar a impressão desejada.

Essa solução evidenciou um problema inesperado: compatibilidade com as coordenadas das figuras. Até esse momento as texturas que usamos eram simétricas horizontalmente (texturas de tijolos por exemplo), por isso o problema só apareceu quando os menus (que contém textos) foram renderizados invertidos horizontalmente. Após investigar, entendemos que o problema foi que o eixo de coordenadas que o SDL usa ao criar as texturas coloca o eixo (0,0) no canto superior esquerdo enquanto o OpenGL as lê com o eixo (0,0) no canto inferior esquerdo (Figura 15). O que resulta numa figura invertida horizontalmente.

Como o problema é da compatibilidade das bibliotecas e preparar uma forma de tratar pareceu além do escopo do projeto, resolvemos utilizar a solução mais simples de salvar as imagens invertidas para que, ao renderizar, elas aparecem corretamente.

Figura 15: Coordenadas SDL e OpenGL



Complementos

Os menus e as texturas dos blocos foram geradas por nós, muitas delas com o uso do site Canva.

A música utilizada no jogo é Subspace - by Waveeyy disponível no site Looperman e é de uso gratuito (Waveeyy 2022).

Inovação

Incluímos uma jogabilidade diferente para finalizar uma fase. No jogo-base, a fase terminava quando o jogador subia no bloco amarelo localizado no topo da fase. No nosso jogo há vários blocos amarelos que se tornam verdes quando o jogador sobe neles, para uma fase ser completa todos os blocos amarelos devem se tornar verdes.

A mudança traz a possibilidade de vários percursos que o jogador pode fazer para completar uma fase (mudando a ordem de tornar os blocos verdes).

Resultados

O jogo funciona, há espaço para melhorias como suavizar as animações e movimentos, mas criamos “a partir do nada” um jogo funcional com uma arquitetura complexa que trabalha com uma física básica, colisões, renderizações e processamentos.

Avaliações e Testes

O jogo foi testado em dois computadores distintos e os problemas de compatibilidade encontrados quanto a versões das bibliotecas foram tratados.

Além disso, testamos as mecânicas da física e updates a partir de situações criadas num mapa teste de acordo com a necessidade de cada recurso sendo testado.

Futuras melhorias

Uma ideia que tivemos mas não tivemos tempo de implementar era a ideia do jogador controla dois players, um na frente, outro atrás da torre, e trabalhar para que ambos subissem e se encontrassem no topo, tendo que lidar com o uso dos mesmos blocos e com a possibilidade das ações de um afetar o outro.

RESULTADO E DISCUSSÃO

Criar um jogo sem engine é um desafio, o desenvolvedor é forçado a lidar com mais que apenas o design do jogo e suas mecânicas e tem de resolver problemas básicos e bugs das implementações.

Matérias da graduação

Todas as matérias da graduação contribuíram para a formação e preparação necessária para realizar esse projeto, destacando algumas:

- (MAT0112) Vetores e Geometria - primitivas, entendimento da matemática e geometria de projeções e transformações;
- (MAC0420) Introdução a Computação Gráfica - conceitos como viewpoint, inclusão de texturas, renderização, projeções e matrizes de transformação;
- (MAC0210) Laboratório de Métodos Numéricos - criação de figuras e aproximações;
- (MAC0323) Algoritmos e Estruturas de Dados II - lista ligada e análises para escolha;
- (MAC0216 e MAC0218) Técnicas de Programação I e II - POO, github, padrões de design (organização de código, readme, licença);

REFERÊNCIAS

Fandom (**n.d.**) Fandom. *https://catherine.fandom.com/wiki/Rapunzel*. Accessed: 21-Dec.-2022. Retrieved 21-Dec.-2022, from https://catherine.fandom.com/wiki/Rapunzel.

Khronos (**n.d.**) Khronos. *https://www.khronos.org/opengl/wiki/Getting_Started#Writing_an_OpenGL_Application*. Accessed: 21-Dec.-2022. Retrieved 21-Dec.-2022, from https://www.khronos.org/opengl/wiki/Getting_Started#Writing_an_OpenGL_Application.

Lewis and Jacobson (**2002**) Michael Lewis and Jeffrey Jacobson. Introduction. *Communications of the ACM*. 45, 1 (Jan.-2002), 27–31. doi: 10.1145/502269.502288.

Metacritic (**2011**) Metacritic. *https://www.metacritic.com/game/xbox-360/catherine*. Accessed: 21-Dec.-2022. Retrieved 21-Dec.-2022, from https://www.metacritic.com/game/xbox-360/catherine.

“OpenGL Primitives and Colours” (**2001**) *http://www.dgp.toronto.edu/~ah/csc418/fall_2001/tut/ogl_draw.html*. Accessed: 22-Dec.-2022. Retrieved 22-Dec.-2022, from http://www.dgp.toronto.edu/~ah/csc418/fall_2001/tut/ogl_draw.html.

Pacete (**2022**) Luiz Gustavo Pacete. 2022 promissor: mercado de games ultrapassará US $200 bi até 2023. *Forbes Brasil*. Retrieved 18-Dec.-2022, from https://forbes.com.br/forbes-tech/2022/01/com-2022-decisivo-mercado-de-games-ultrapassara-us-200-bi-ate-2023/.

“Simple DirectMedia Layer” (**n.d.**) *https://www.libsdl.org/*. Accessed: 21-Dec.-2022. Retrieved 21-Dec.-2022, from https://www.libsdl.org/.

Steam (**2019**) Steam. *https://store.steampowered.com/app/893180/Catherine_Classic/?l=portuguese*. Accessed: 21-Dec.-2022. Retrieved 21-Dec.-2022, from https://store.steampowered.com/app/893180/Catherine_Classic/?l=portuguese.

Team (**2011**) Editorial Team. Catherine Rapunzel Stages Guide. *SegmentNext*. Retrieved 22-Dec.-2022, from https://segmentnext.com/catherine-rapunzel-stages-guide/.

Waveeyy (**2022**) Waveeyy. *https://www.looperman.com/loops/detail/314853/subspace-club-type-sample-free-115 bpm-disco-pad-loop*. Accessed: 21-Dec.-2022. Retrieved 21-Dec.-2022, from https://www.looperman.com/loops/detail/314853/subspace-club-type-sample-free-115 bpm-disco-pad-loop.

Wikimedia (**2022**) Wikimedia. *https://en.wikipedia.org/wiki/Game_engine*. Accessed: 22-Dec.-2022. Retrieved 22-Dec.-2022, from https://en.wikipedia.org/wiki/Game_engine.

Wikimedia (**2005**) Contribuidores dos projetos da Wikimedia. Intel 80486. *Fundação Wikimedia, Inc.* Retrieved 22-Dec.-2022, from https://pt.wikipedia.org/wiki/Intel_80486.

Wikimedia (**2012**) Contribuidores dos projetos da Wikimedia. Catherine (jogo eletrônico). *Fundação Wikimedia, Inc.* Retrieved 21-Dec.-2022, from https://pt.wikipedia.org/wiki/Catherine_(jogo_eletr%C3%B4nico)#cite_note-6.

Zarrad (**2018**) Anis Zarrad. Game Engine Solutions. *Simulation and Gaming*. InTech. Retrieved 21-Dec.-2022, from http://dx.doi.org/10.5772/intechopen.71429.