

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Igor Batista Fernandes

**Desenvolvimento da game engine 2D Narval
utilizando OpenGL e Java**

Uberlândia, Brasil

2018

Igor Batista Fernandes

Desenvolvimento da game engine 2D Narval utilizando OpenGL e Java

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, Minas Gerais, como requisito exigido parcial à obtenção do grau de Bacharel em Ciência da Computação.

Uberlândia, Brasil, XX de XX de 2018:

Profa. Maria Adriana Vidigal de Lima
Orientador

Professor

Professor

Uberlândia, Brasil
2018

Resumo

Este trabalho propõe a criação de uma Game Engine 2D para desktop intitulada Narval e a subsequente elaboração de um protótipo de jogo. A implementação de código será realizada na linguagem Java e através da API gráfica OpenGL usando GLFW (OpenGL Frame Work). O protótipo de jogo será elaborado utilizando técnicas de algoritmos procedurais para geração de artes e ambientes, inteligência artificial para o comportamento de Non Playable Characters (NPC) e padrões de projeto aplicados a jogos.

Palavras-chave: Jogo indie, Jogos, Java, OpenGL, Game Engine, Inteligência artificial, Procedural Content Generation

Resumo

This work proposes to develop a 2D game engine for desktop entitled Narval and a subsequent game prototype elaboration. The code implementation will be done in Java through the API OpenGL using GLFW (OpenGL Frame Work). The game prototype will be implemented using techniques of procedural algorithms to generate art and environment, artificial intelligence for Non Playable Characters (NPC) behaviours and design patterns applied to games.

Keywords: Indie Game, Games, Java, OpenGL, Game Engine, Artificial Intelligence, Procedural Content Generation

NARVAL

GAME ENGINE

Listas de ilustrações

Figura 1 – Pacotes da engine Narval divididos em camadas	19
Figura 2 – Execução do game loop básico ao longo do tempo	20
Figura 3 – Execução do game loop com timestep fixo ao longo do tempo	22
Figura 4 – Execução do game loop com timestep variável ao longo do tempo	22
Figura 5 – Objeto ignorando colisão devido um timestep muito grande	23
Figura 6 – Execução do game loop com timestep semi-fixo ao longo do tempo	24
Figura 7 – Método <code>update</code> com tempo de processamento maior que o Δt	24
Figura 8 – Renderização sem interpolação	25
Figura 9 – Renderização com interpolação	26
Figura 10 – Etapas do pipeline	29
Figura 11 – Dois triângulos em coordenadas locais normalizadas que juntos formam um quadrado	31
Figura 12 – Resultado obtido com dois shaders simples	33
Figura 13 – Sistema ARGB e RGBA, ambos com canais de 8 bits (1 byte)	37
Figura 14 – Extração do canal verde em um inteiro de 3 bytes	38
Figura 15 – Wrap modes	38
Figura 16 – Ampliação de 32x usando cada tipo de filtro.	39
Figura 17 – Spritesheet com frames de 32x32 pixels	39
Figura 18 – Ordem correta de renderização: do menor y para o maior y	42
Figura 19 – Ordem correta de renderização: do menor y para o maior y conside- rando a caixa base	42
Figura 20 – Componentes do modelo de Phong	43
Figura 21 – Demonstração da luz ambiente com tonalidade alaranjada	44
Figura 22 – Ângulo θ entre raio luminoso e vetor normal da superfície iluminada . .	44
Figura 23 – Resultado da luz difusa usando a textura normal do objeto	46
Figura 24 – Luz especular do ponto de vista do observador	47
Figura 25 – Especialização das classes numa arquitetura baseada em objetos	51
Figura 26 – GameObject composto de vários componentes	52
Figura 27 – Diagrama de classes do sistema entity based	52
Figura 28 – Colisão AABB vs AABB	54
Figura 29 – Processo de transformações de coordenadas até o espaço final da tela. Fonte: (VRIES, 2015)	55
Figura 30 – Exemplos de transformação afim. Fonte: (WIKIPEDIA, 2018)	56
Figura 31 – Frustum da projeção ortográfica. Fonte: (VRIES, 2015)	57
Figura 32 – Frustum da projeção de perspectiva. Fonte: (VRIES, 2015)	58

Figura 33 – Demonstração do sistema de chunks carregando a última coluna da matriz	63
Figura 34 – Gráfico das funções utility	67
Figura 35 – Diagrama da Árvore de considerações para as funções de pontuação	68
Figura 36 – Desperdício gerado ao mapear uma base box de 32x32 pixels numa matriz de retângulos 10x10 pixels.	70
Figura 37 – Matriz booleana representada visualmente com os retângulos verdes sendo os transponíveis e os rosas intransponíveis.	70
Figura 38 – Resultado final obtido com o A*	71
Figura 39 – Ponto (0.75, 0.25) no plano. Adaptado de: (FATAHO, 2018)	72
Figura 40 – Vetores gradiente definidos em amarelo no plano. Adaptado de: (FATAHO, 2018)	73
Figura 41 – Vetor distância do ponto (0, 0)na borda até o ponto (0.75, 0.25) em vermelho. Adaptado de: (FATAHO, 2018)	73
Figura 42 – Vetores distância do ponto (0, 0) em verde e vetores gradiente em amarelo. Adaptado de: (FATAHO, 2018)	74
Figura 43 – Resultado da alteração na frequência. Adaptado de: (PATEL, 2018)	75
Figura 44 – Resultado final dos terrenos após modelagem do ruído e atribuição de significado. Adaptado de: (PATEL, 2018)	77
Figura 45 – Classes do pacote engine.ai	78
Figura 46 – Classes do pacote engine.audio	79
Figura 47 – Classes do pacote engine.engine	80
Figura 48 – Classes do pacote engine.entity.component	81
Figura 49 – Classes do pacote engine.entity	82
Figura 50 – Classes do pacote engine.graphic	82
Figura 51 – Classes do pacote engine.input	83
Figura 52 – Classes do pacote engine.logic	84
Figura 53 – Classes do pacote engine.noise	85
Figura 54 – Classes do pacote engine.renderer	86
Figura 55 – Classes do pacote engine.utilities	86
Figura 56 – Árvore de arquivos do projeto	87
Figura 57 – Fluxo de chamadas do método update a partir da Engine. Esse mesmo fluxo também vale para os métodos render e variableUpdate.	89
Figura 58 – Captura de tela 1	92
Figura 59 – Captura de tela 2	92
Figura 60 – Captura de tela 3	92
Figura 61 – Captura de tela 4	93
Figura 62 – Captura de tela 5	93

Lista de tabelas

Tabela 1 – Tabela de valores exemplo para a função de atenuação	48
Tabela 2 – Tabela de valores exemplo para as funções de pontuação	68

Listas de Listagens

3.1	Estrutura básica do Game Loop	20
3.2	Game Loop com timestep fixo	21
3.3	Game loop com timestep variável	22
3.4	Game Loop com timestep semi-fixo	23
3.5	Game Loop com timestep semi-fixo e interpolação linear	26
3.6	Inicialização da janela e contexto OpenGL	27
3.7	Inicialização do VBO e VAO	31
3.8	Cabeçalho do Vertex shader	32
3.9	Vertex shader simples	32
3.10	Fragment shader simples	33
3.11	Processo de compilação do shader	34
3.12	Processo de criação e link de um programa	34
3.13	Inicializando uma textura no OpenGL a partir de um BufferedImage	35
3.14	Função auxiliar createByteBuffer	37
3.15	Vertex shader com animações	40
3.16	Classe Animation	40
3.17	Fragment Shader com luz ambiente	43
3.18	Fragment Shader com luz ambiente e difusa	45
3.19	Fragment Shader com luz ambiente, difusa e atenuação	48
3.20	Classe renderer simples	49
3.21	Demonstração do processo de renderização	50
3.22	Colisão AABB vs AABB	53
3.23	Função Ortho do GLM	58
3.24	Função Perspective do GLM	58
3.25	Classe Camera	59
3.26	Keyboard input	61
3.27	Mouse input	61
3.28	Leitura do arquivo de áudio	64
3.29	OpenAL Buffers	65
3.30	OpenAL Listener	65
4.1	Heurística utilizada no A*	71
5.1	Interpolação dos valores <i>influência</i>	74
5.2	Implementação do ruído que usa o perlin noise com dois octaves	75
5.3	Implementação que gera e molda o ruído	76
5.4	Implementação que gera a textura final do terreno	76
7.1	Classe Main	88

7.2	Classe Game	89
A.1	Classe responsável pelo Shader	98
B.1	A* adaptado	101
C.1	Classe Entity	105
C.2	Classe EntityManager responsável por gerenciar as entidades	105
C.3	Classe ComponentSystem responsável por definir a abstração dos sistemas de componente	107
C.4	Classe SystemManager responsável por gerenciar os sistemas de componentes	107
C.5	Abstração da classe Component	108
C.6	Exemplo de especialização da classe Component na classe RenderComponent	108
D.1	Classe Resource Manager	111
E.1	Classe Engine	114
E.2	Classe GSM	117
E.3	Classe GameState	118

Lista de abreviaturas e siglas

API	Application Programming Interface
MVC	Model-view-controller
OpenGL	Open Graphics Library
GPU	Graphics Processing Unit
GLFW	Graphics Library Framework
GLSL	OpenGL Shading Language
NPC	Non Playable Character
HUD	Heads-Up Display
FPS	Frames per second
px	Pixel (Unidade de medida)
VBO	Vertex Buffer Object
VAO	Vertex Array Object
AABB	Axis Aligned Bouding Box
NDC	Normalized Device Coordinates
GLM	OpenGL Mathematics Library
IA	Inteligência Artificial
PCM	Pulse-code modulation
FOV	Field Of View

Sumário

1	INTRODUÇÃO	14
1.1	Visão Geral da Proposta	14
1.2	Objetivo	14
1.3	Justificativa e Motivação	14
1.4	Organização do Trabalho	15
2	CONCEITOS BÁSICOS	16
2.1	Computação gráfica	16
2.2	OpenGL	16
2.3	Game Engine	16
2.4	Inteligência artificial	17
2.5	Procedural Content Generation	17
3	ARQUITETURA DA GAME ENGINE NARVAL	18
3.1	<i>Game Loop</i>	19
3.2	Sistema de atualização	20
3.2.1	Timestep fixo	20
3.2.2	Timestep variável	22
3.2.3	Timestep semi-fixo	23
3.3	Sistema de renderização	24
3.3.1	Interpolação Linear	25
3.4	API gráfica OpenGL	26
3.4.1	Core-profile e Immediate mode	27
3.4.2	State Machine	27
3.4.3	Hello window	27
3.4.4	OpenGL Pipeline	29
3.4.5	Vertex Array Object e Vertex Buffer Object	30
3.4.6	Vertex e Fragment shader	32
3.4.7	Compilando o shader em um programa	33
3.4.8	Uniforms	35
3.5	Renderizador	35
3.5.1	Textura	35
3.5.2	Animações	39
3.5.3	Z-Ordering	41
3.6	Iluminação	42
3.6.1	Ambiente	43

3.6.2	Luz Difusa	44
3.6.3	Vetor Normal	46
3.6.4	Luz Especular	46
3.6.5	Atenuação	47
3.7	Classe Renderer	49
3.8	Arquitetura do Game Object	50
3.8.1	Object Based	50
3.8.2	Component Based	51
3.8.3	Entity Based	52
3.9	Sistema de colisões	53
3.9.1	Colisões do tipo AABB vs AABB	53
3.9.2	jBox2D	54
3.10	Sistema de coordenadas	54
3.10.1	Espaço local	55
3.10.2	Espaço de mundo	55
3.10.3	Espaço de visão	56
3.10.4	Espaço de recorte	56
3.10.5	Projeção	57
3.10.5.1	Projeção ortográfica	57
3.10.5.2	Projeção de perspectiva	58
3.10.6	jBox2D para espaço de mundo	59
3.11	Câmera	59
3.12	Sistema de input	60
3.12.1	GLFW Keyboard	61
3.12.2	GLFW Mouse	61
3.13	Gerência de recursos	62
3.13.1	Chunk	63
3.13.2	Sistema de Chunks	63
3.13.3	Gerenciador de Chunks	64
3.14	Aúdio	64
3.14.1	Integrando o OpenAL	64
4	INTELIGÊNCIA ARTIFICAL	66
4.1	Utility em IA	66
4.1.1	Processo de funcionamento	66
4.2	Pathfinding	69
4.2.1	Algoritmo A*	70
5	PROCEDURAL CONTENT GENERATION	72
5.1	Ruídos aleatórios	72

5.1.1	Ruído de Perlin	72
5.2	Geração de terrenos	74
6	DIAGRAMAS E PANORAMA GERAL DA ENGINE NARVAL	78
6.1	Pacote de IA	78
6.2	Pacote de Áudio	79
6.3	Pacote da Engine	79
6.4	Pacote de componentes de entidades	80
6.5	Pacote de entidades	82
6.6	Pacote gráfico	82
6.7	Pacote de input	83
6.8	Pacote lógico	83
6.9	Pacote de ruído	84
6.10	Pacote de renderizadores	85
6.11	Pacote de utilidades	86
6.12	Estrutura dos arquivos	87
7	EXPERIMENTOS E RESULTADOS	88
7.1	Inicialização e exemplo de utilização da engine	88
7.2	Demonstração	91
8	CONCLUSÃO	94
8.1	Perspectivas de Trabalhos Futuros	94
	REFERÊNCIAS	96
	APÊNDICE A – CLASSE RESPONSÁVEL PELO SHADER	98
	APÊNDICE B – IMPLEMENTAÇÃO DO ALGORITMO A* ADAPTADO	101
	APÊNDICE C – IMPLEMENTAÇÃO DO SISTEMA ENTITY BASED	105
	APÊNDICE D – CLASSE RESPONSÁVEL PELA GERÊNCIA DE RECURSOS	111
	APÊNDICE E – CLASSE NÚCLEO DO SISTEMA: ENGINE	114

1 Introdução

1.1 Visão Geral da Proposta

Estima-se que o mercado internacional de jogos movimentará aproximadamente 108.9 bilhões de dólares distribuídos entre 2.2 bilhões de jogadores em 2017. Destes, 58% representam os segmentos de PC e consoles ([MCDONALD, 2017](#)). Além de ser uma área comercial lucrativa é também interdisciplinar e aproxima diversos conceitos de Computação, Design gráfico, Música, Artes e outras esferas do conhecimento. Para a elaboração de um projeto bem construído e com alto potencial de sucesso é necessário construir uma fundação sólida do software, no caso de um jogo, uma game engine. Ao longo de todas as etapas serão definidas estratégias de implementação na linguagem java, engenharia de software e padrões de projeto aplicados para jogos. Serão desenvolvidos e abordados os seguintes módulos: núcleo da game engine, renderização, áudio, inteligência artificial e *procedural content generation*.

1.2 Objetivo

Este projeto propõe portanto a construção de uma game engine, também traduzida como motor de jogo. Para isso será utilizada a linguagem Java e a biblioteca [LWJGL \(2018\)](#) (*Lightweight Java Game Library*), que incorpora a biblioteca OpenGL, GLFW (*Graphics Library Framework*) e outras necessárias ao projeto. Desenvolvida pela Khronos Group, a API OpenGL fornece uma ampla gama de soluções para aplicações gráficas, sendo acelerada diretamente em hardware, multiplataforma e robusta. Esses fatores culminaram na sua escolha como o componente responsável pela renderização da engine Narval. Para a parte de áudio será utilizada a biblioteca [openAL \(2018\)](#) e para o módulo de física a biblioteca [jBox2D \(2018\)](#). Ao final do desenvolvimento da referida game engine será ainda construído um protótipo de jogo para demonstrar as funcionalidades da mesma.

1.3 Justificativa e Motivação

O consumo cada vez maior de jogos como uma forma de entretenimento pelo público de todas as idades traz cada vez mais oportunidades de trabalho e aprendizagem. A insuficiência de documentação técnica em língua portuguesa sobre os processos de desenvolvimento e planejamento de jogos é um grande instigador desse trabalho. Neste contexto, esse trabalho aborda todas as etapas de desenvolvimento pelas quais uma engine

passa para torna-se um software que será distribuído e utilizado por pessoas do mundo todo. Embora haja diversas opções de engines no mercado como Unity, Unreal, CryEngine e afins, a grande maioria é de código fechado, o que torna extremamente difícil senão impossível de modificá-la para as necessidades especiais de cada projeto. Além deste fato, para comercializar os produtos produzidos nelas é necessário pagar uma porcentagem da revenda obtida com o produto lançado. Portanto, para este projeto será desenvolvida uma engine própria, para ter-se uma maior flexibilidade de implementação e para ter uma licença de software próprio.

1.4 Organização do Trabalho

As bases teóricas: computação gráfica, OpenGL, arquitetura de game engines, inteligência artificial e *procedural content generation* são encontradas no Capítulo 2. No Capítulo 3 está descrita toda a arquitetura por trás de uma game engine, abordando e explicando algumas possibilidades juntamente com o desenvolvimento optado na engine Narval para cada uma das áreas. No Capítulo 4 são apresentados os conceitos de inteligência artificial utilizados neste sistema. O Capítulo 5 apresenta, explica e implementa os conceitos de geração procedural de conteúdo, bem como os processos para geração de ruído. Um panorama geral da implementação final do projeto Narvalé apresentado no Capítulo 6 na forma de diagramas. A demonstração do resultado obtido neste trabalho encontra-se no Capítulo 7. Finalmente, o Capítulo 8 expõe as conclusões obtidas do trabalho desenvolvido e apresenta perspectivas para trabalhos futuros.

2 Conceitos Básicos

2.1 Computação gráfica

Computação gráfica refere-se a tudo que envolve criação ou manipulação de imagens no computador, incluindo animações ([ECK, 2018](#)). Este ramo possui aplicações em diversas áreas, tais como: Jogos, Arquitetura, Cinema, Medicina, Artes e várias outras. Toda a computação gráfica atual é baseada em cálculos matemáticos para sintetizar pixels em uma tela 2D. Esse processo se dá através de um pipeline por onde primitivas gráficas são manipuladas para obter o resultado final mostrado na tela do usuário.

2.2 OpenGL

A API (Application Programming Interface) OpenGL fornece um conjunto de funções para manipulações gráficas ([VRIES, 2015](#)), sendo acelerada diretamente em hardware, multiplataforma e robusta. Embora comumente referida como uma API, o OpenGL é, por si só, um conjunto de especificações que determinam o resultado/saída de cada função e como devem ser executadas. Fica a cargo dos manufaturadores de placas gráficas implementarem a operação da função, respeitando as especificações do documento desenvolvido e mantido pela Khronos Group ([KHRONOS, 2017](#)).

O OpenGL foi lançado em 1992 como uma resposta direta a necessidade de se padronizar o conjunto de instruções usado em hardwares com interface gráfica. Até setembro de 2006 o padrão foi mantido pela ARB (Architecture Review Board), um conselho formado por empresas de grande renome no ramo como HP, IBM, Intel, NVIDIA, Dell e a própria fundadora, a Silicon Graphics. Em setembro de 2006 o conselho ARB tornou-se o OpenGL Working Group gerido e mantido pelo consórcio Khronos Group para Open Standard APIs([OPENGL, 2017](#)).

2.3 Game Engine

Define-se como game engine um sistema composto de outros sub sistemas cujo trabalho em coesão provêm todas as funcionalidades necessárias para o desenvolvimento de um jogo. Uma arquitetura direcionada a dados é o que diferencia uma game engine de um pedaço de software que é o jogo, mas não uma engine ([GREGORY, 2009](#)).

Portanto, game engine é um software extensível e pode ser usado como fundação para diferentes tipos e gêneros de jogos sem mudanças maiores no seu núcleo. Embora

uma game engine ideal deva ser capaz de servir como base para qualquer jogo imaginável, é fato que quanto mais de propósito geral ela se torna, menos ótima ela é para rodar um jogo em determinada plataforma.

Um sistema dessa magnitude precisa de uma organização estrutural flexível e modular para trazer uma fundação sólida ao projeto que será desenvolvido. Os principais sub sistemas da game engine são: renderização, colisão e física, inteligência artificial, áudio e gerência de recursos.

2.4 Inteligência artificial

Inteligência Artificial (IA) é um ramo da Ciência da Computação que teoriza e desenvolve sistemas capazes de realizar tarefas onde normalmente requeriam inteligência humana, como processamento de imagens, reconhecimento de fala, tomada de decisões e traduções linguísticas ([DICTIONARY, 2018](#)).

Existe uma distinção importante na IA estudada academicamente e na utilizada em jogos. A pesquisa acadêmica em IA é dividida em dois campos: *strong AI* e *weak AI*. O campo de *strong AI* preocupa-se em tentar criar sistemas que imitam o processo de pensamento humano, enquanto o campo de *weak AI* se preocupa em aplicar tecnologias baseadas em IA para solucionar problemas do mundo real. Entretanto, ambos os campos procuram resolver os problemas de maneira ótima, sem preocupar-se com limitações de tempo e hardware ([BUCKLAND, 2005](#)).

Em contrapartida, a IA aplicada em jogos preocupa-se não com a solução ótima de um problema, mas com uma solução que seja capaz de dar ao usuário uma experiência agradável respeitando as capacidades do hardware para execução em tempo real.

2.5 Procedural Content Generation

Procedural Content Generation (PCG) é a criação algorítmica de conteúdo para jogos com pouca ou nenhuma interferência do usuário ([SHAKER; TOGELIUS; NELSON, 2016](#)). Por conteúdo define-se cenários, mapas, regras de jogo, texturas, itens, música, personagens e qualquer outro elemento que componha o jogo.

Esse processo frequentemente se da pela utilização de métodos aleatórios ou pseudo-aleatórios para gerar dados que serão manipulados através de regras e transformados em uma infinidade de elementos do jogo ([WIKIDOT, 2018](#)).

3 Arquitetura da game engine Narval

Game engine (em português, motor de jogo) é o termo designado ao motor que está por trás de todo jogo. É neste conjunto de sistemas de simulação em tempo real que todo o ambiente do jogo é construído. As principais funcionalidades providas em uma engine são: um sistema de renderização 2D e/ou 3D, detecção e resolução de colisões, áudio, inteligência artificial e muitos outros. A partir desses elementos há várias ramificações em subsistemas com funcionalidades específicas para satisfazer as necessidades individuais de cada projeto. Uma engine se assemelha de muitas formas a um sistema operacional. Ela lida com aspectos de baixo nível da máquina como a *Graphic Processing Unit* (GPU) e é comumente construída utilizando-se o padrão de projeto em camadas. A Figura 1 ilustra o diagrama de pacotes da arquitetura utilizada na engine Narval.

Alguns exemplos muito populares de engines disponíveis no mercado são a Unity, Unreal Engine, Godot e Construct 2. Cada uma apresenta sua própria estrutura e especificidade. Algumas são voltadas para projetos 3D e 2D, outras são otimizadas especificamente para um ambiente 2D ou 3D. Cabe ao projetista decidir qual desses produtos irá melhor atendê-lo conforme suas necessidades. Para este projeto, a engine utilizada será de desenvolvimento próprio e seguirá o diagrama apresentado adiante. Cada pacote desse diagrama será melhor detalhado no capítulo 6.

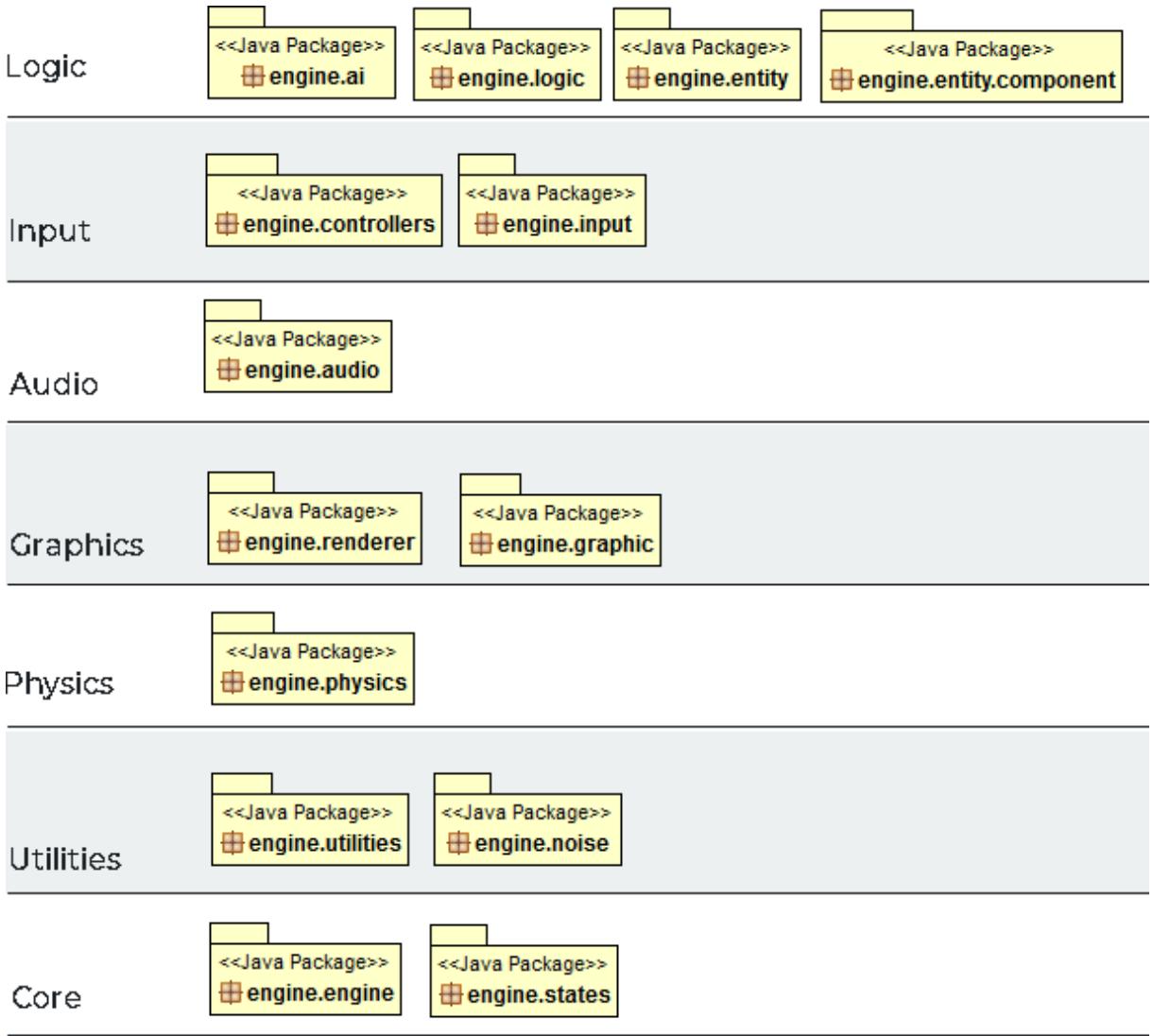


Figura 1 – Pacotes da engine Narval divididos em camadas

3.1 Game Loop

Game loop é o núcleo da arquitetura de uma engine. É neste loop que todos os subsistemas da engine são chamados e executados, como a renderização, detecção e resolução de colisões, áudio e muitos outros (GREGORY, 2009). Por se tratar de uma simulação em tempo real, onde a tela inteira deve ser atualizada em uma quantidade muito alta de vezes, é fundamental que tudo seja executado o mais rápido possível e em tempo constante para que o usuário tenha uma experiência fluída e dinâmica.

Portanto, o tempo demanda um papel chave neste sistema e deve ser cuidadosamente levado em consideração para que não haja quaisquer gargalos que deturpem a fluidez e experiência final do usuário. A estrutura mais simples de um game loop é

demonstrada na Listagem 3.1.

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

Listagem 3.1 – Estrutura básica do Game Loop

A Listagem 3.1 sendo executada ao longo do tempo é representada pela Figura 2. Cada execução do método `render` significa o desenho de uma imagem na tela e a quantidade total de imagens desenhadas ao longo de um segundo é representada pela unidade de medida FPS (*Frames per second*). Cada execução do método `update` significa um passo no tempo do jogo (*timestep*). Da mesma forma que o relógio move-se em tiques de um segundo em um segundo, o tempo do jogo avança em tiques de `update` em `update`.



Figura 2 – Execução do game loop básico ao longo do tempo

3.2 Sistema de atualização

O sistema de atualização é responsável por controlar o aspecto lógico da engine. Nele ocorrem todos os cálculos relativos a movimentação dos objetos, colisões, inteligência artificial e outros. Sendo assim, é um sistema composto de outros sistemas, cada um rodando com uma taxa de atualização específica e não obrigatoriamente atrelados ao FPS.

3.2.1 Timestep fixo

Sistemas de atualização com timestep fixo são aqueles que estavam diretamente atrelados ao FPS e eram utilizados em jogos antigos (GREGORY, 2009). As unidades de medida de tempo eram diretamente atreladas ao FPS tal que, se uma máquina fosse capaz de rodar o jogo a 30 FPS e outra a 60 FPS, na segunda máquina o jogo daria impressão de estar duas vezes mais rápido ou duas vezes mais lento dependendo do valor fixado para o timestep. Isso acontecia por que os jogos eram desenvolvidos para plataformas específicas e, sabendo em qual taxa de FPS o jogo iria rodar, era fácil delimitar um timestep fixo.

Entretanto, conforme as máquinas se tornaram mais potentes e o mercado passou a oferecer mais opções de hardware, logo a indústria estava produzindo jogos para um SO com múltiplas possibilidades de hardware. Essa gama de computadores com capacidades de processamento distintas gerou o problema descrito acima. Por exemplo, seja uma máquina MA capaz de rodar o jogo a 30 FPS e uma máquina MB capaz de rodar a 60 FPS, sendo a máquina MA o alvo do projeto. Se um personagem deveria mover-se a 300 pixels por segundo, logo 10 pixels por frame ($300px/30FPS$), na máquina MB ele estaria se movendo a 600 pixels por segundo pois ao dobrar o *FPS* dobra-se a quantidade de vezes que o método `update` é chamado, e portanto, o personagem passa a mover-se a 600 pixels por segundo ($60FPS * 10pxp/frame$). Isso acontece por que antigamente o jogo era projetado para rodar numa máquina cuja capacidade de FPS seria x e a variável Δt , que representa o tempo transcorrido entre um frame e outro, seria o inverso de x , ou simplesmente o inverso da frequência, o período $1/x$. A partir disso a posição do objeto era calculada efetuando-se $pos(i) = pos(i - 1) + velPerFrame$ e como Δt é um valor fixo dado por $1/x$ (baseado num FPS de x), tem-se que em um computador mais potente, o tempo transcorrido entre um frame e outro é menor e portanto o período aumenta. Como o valor foi pré-calculado para uma máquina alvo, ele não diminui na máquina mais potente e acaba sendo somado mais vezes, resultando em uma velocidade maior que a originalmente desejada. Esse problema pode ser facilmente representado por uma série.

Em um PC capaz de rodar a 30FPS:

$$\sum_{n=1}^{30} 10px = 300px/s$$

Em um PC capaz de rodar a 60FPS:

$$\sum_{n=1}^{60} 10px = 600px/s$$

Sendo assim, a estrutura do game loop com timestep fixo é dado na Listagem 3.2 e ilustrado pela Figura 3.

```

1 public static final float dt = 1f/30f;
2
3 while(true) {
4     update(dt);
5     render();
6 }
```

Listagem 3.2 – Game Loop com timestep fixo



Figura 3 – Execução do game loop com timestep fixo ao longo do tempo

3.2.2 Timestep variável

Para que a taxa de atualização Δt seja dinâmica ao invés de fixa, ela precisa ser independente do FPS. Isso é possível medindo-se quanto tempo transcorre entre um frame e outro. Dessa forma o game loop fica definido na Listagem 3.3 e representado pela Figura 4.

```

1 private long lastFrame;
2 private long dt;
3
4 while(true) {
5     long currentFrame = System.nanoTime();
6     dt = currentFrame - lastFrame;
7     lastFrame = currentFrame;
8
9     update(dt);
10    render();
11 }
```

Listagem 3.3 – Game loop com timestep variável

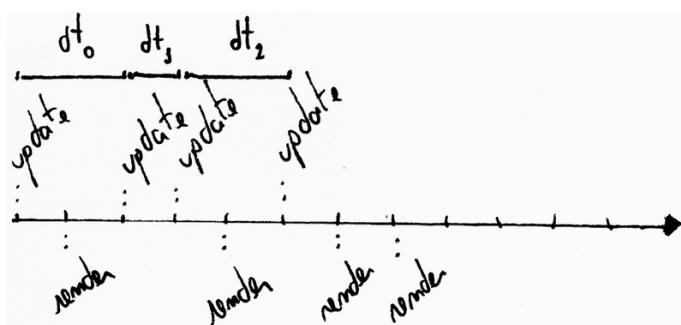


Figura 4 – Execução do game loop com timestep variável ao longo do tempo

Embora tenha-se resolvido o problema anterior de dois computadores com capacidades diferentes de processamento, o sistema ainda não é ideal. A falha está no fato de utilizar o Δt anterior ao frame atual. Se o tempo passado entre um frame e outro for muito grande, ou seja, se houver um pico de performance, será avançado um tempo muito

grande e um passo do personagem que era para ser 10pixels , passa a ser $10\text{px} + \text{atraso}$. Isso gera um efeito chamado de *stuttering* e é perceptível ao jogador, pois atrapalha a fluidez da movimentação. Isso também traz consequências na lógica do programa. Um objeto que deveria percorrer 10 pixels por timestep, ao percorrer mais em um único timestep poderia, por exemplo, estar ignorando uma colisão que iria ocorrer entre o ponto atual e o próximo. A Figura 5 ilustra esse problema.

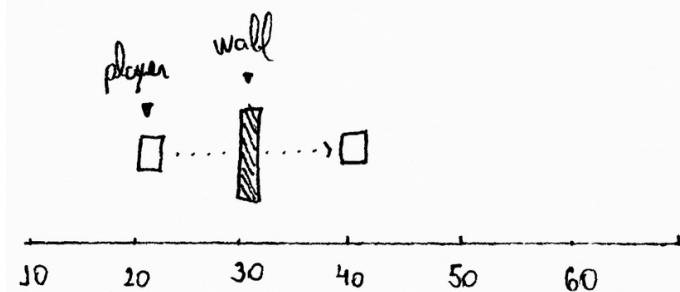


Figura 5 – Objeto ignorando colisão devido um timestep muito grande

Não só isso, mas um timestep variável traz toda uma complicação com a depuração. Ao introduzir um fator não determinístico no processo, pode ser que seja impossível reproduzir um cenário de bug para efetuar seu diagnóstico e correção.

3.2.3 Timestep semi-fixo

Um game loop com timestep semi-fixo tenta trazer o melhor dos dois mundos. Isso é possível usando um Δt fixo para cada chamada do método `update` e, quando o sistema demorar mais que o Δt fixado, faz-se a recuperação do mesmo chamando o método `update` quantas vezes necessário. Isso é fácil visualizar na demonstração feita em código pela Listagem 3.4.

```

1 private long lastFrame;
2 private long accumulator = 0;
3 private long dt;
4 public static final long ONE_SECOND_IN_NANOSECONDS = 10^9;
5 public static final long STEPS_PER_SECOND = 30;
6 public static final long FIXED_DT = ONE_SECOND_IN_NANO/STEPS_PER_SECOND;
7
8 while(true) {
9     long currentFrame = System.nanoTime();
10    dt = currentFrame - lastFrame;
11    lastFrame = currentFrame;
12    accumulator += dt;
13
14    while (accumulator >= FIXED_DT){
15        update(dt);
16        accumulator -= FIXED_DT;
17    }
18}

```

```

19     render();
20 }

```

Listagem 3.4 – Game Loop com timestep semi-fixo

É importante ter cuidado com o valor escolhido para o `FIXED_DT` (timestep). Se o timestep for menor que o tempo que se leva para processar o método `update` o sistema nunca irá recuperar seu atraso, tendo um acumulador que sempre cresce e nunca fica próximo de zerar (NYSTROM, 2014). O sistema de timestep semi-fixo é demonstrado pela Figura 6.

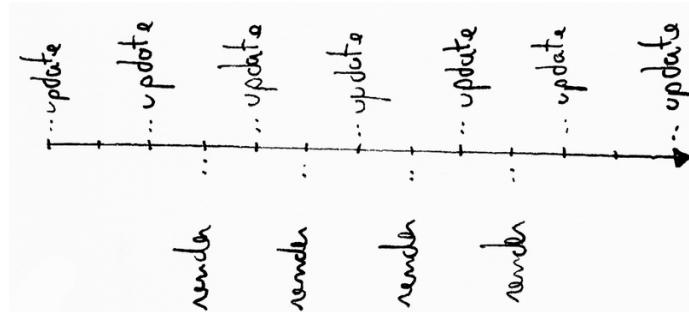


Figura 6 – Execução do game loop com timestep semi-fixo ao longo do tempo

O problema em que o processamento do método `update` é maior que o Δt é mostrado na figura seguinte. Note que o valor de dt_0 é maior que o Δt fixado e o sistema só tende a piorar ao longo do tempo, acumulando cada vez mais atraso nos valores de dt e nunca se recuperando. Essa situação é ilustrada pela Figura 7.

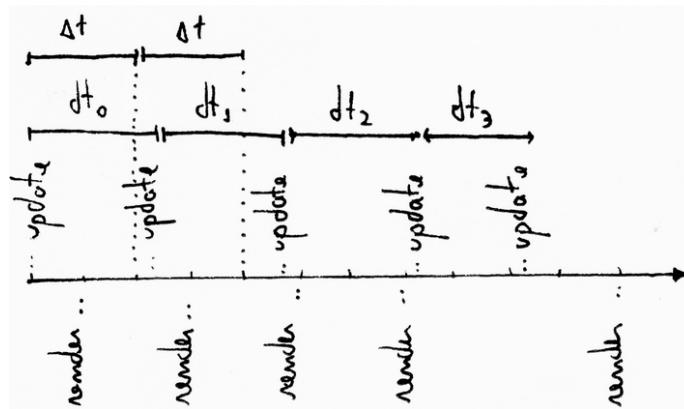


Figura 7 – Método `update` com tempo de processamento maior que o Δt

3.3 Sistema de renderização

É neste sistema que todos os objetos visíveis na tela são desenhados. Esse sistema é executado diversas vezes e em rápida sucessão ao longo de um segundo através do

método `render`, criando uma ilusão de movimento. Esse processo se inicia no processador e termina na placa gráfica acontecendo quantas vezes a máquina conseguir ou quantas vezes o usuário desejar configurar. Os valores mais comuns para fixar o FPS são atrelados à frequência do monitor que, atualmente, variam de 30 Hz até 144 Hz e para todos os efeitos Hertz é uma medida equivalente ao FPS (GREGORY, 2009).

3.3.1 Interpolação Linear

Com o timestep definido ainda é necessário mais um procedimento para que o sistema renderize objetos em movimento de forma suave. O efeito de *stuttering* pode ser causado tanto pelo aspecto lógico, através dos picos de performance, quanto pelo simples fato de que não se possui total controle sobre como o SO gerencia a aplicação. Não é possível garantir uma taxa de atualização e renderização intercalada e perfeita. Haverá momentos que depois de um único `update` o método `render` será chamado várias vezes se processado em um tempo menor que o Δt e, sem nenhum tratamento, este efeito também causa *stuttering*. Ao chamar o método `render` consecutivamente e não atualizar a posição do objeto em cada chamada, o usuário tem a impressão de um sistema engasgado com movimento não fluído. Para resolver este último problema é necessário realizar uma interpolação linear entre a posição anterior e atual do objeto, tornando seu movimento suave. Esse efeito pode ser visualizado nas figuras 8 e 9 onde um objeto move-se 10 pixels por segundo. Na primeira figura a função `render` é chamada sem nenhum tratamento e portanto renderiza o objeto no mesmo lugar até que sua posição seja atualizada no próximo `update`. Já na segunda figura, é renderizada a interpolação da posição desse objeto, tornando seu movimento muito mais suave para o usuário.

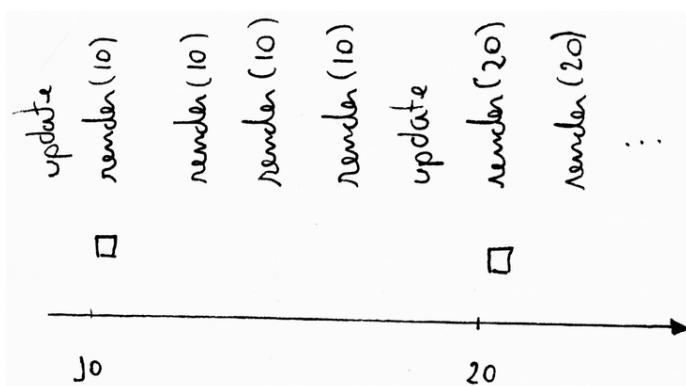


Figura 8 – Renderização sem interpolação

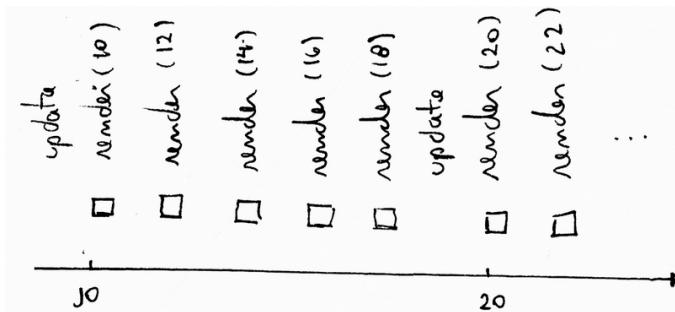


Figura 9 – Renderização sem interpolação

O código final do game loop, para um sistema com timestep semi-fixo e com interpolação linear é demonstrado pela Listagem 3.5.

```

1  long lastFrame;
2  long accumulator = 0;
3  long dt;
4  public static final long ONE_SECOND_IN_NANOSECONDS = 10^9;
5  public static final long STEPS_PER_SECOND = 30;
6  public static final long FIXED_DT = ONE_SECOND_IN_NANO/STEPS_PER_SECOND;
7
8
9  while(true) {
10    long currentFrame = System.nanoTime();
11    dt = currentFrame - lastFrame;
12    lastFrame = currentFrame;
13    accumulator += dt;
14
15    while (accumulator >= FIXED_DT){
16      update(dt);
17      accumulator -= FIXED_DT;
18    }
19
20    long interpolationFactor = accumulator / FIXED_DT;
21
22    render(interpolationFactor);
23 }
```

Listagem 3.5 – Game Loop com timestep semi-fixo e interpolação linear

Dentro da função `render` o fator de interpolação é utilizado na seguinte fórmula para obter a posição onde o objeto deve ser renderizado:

$$\text{renderPosition} = \text{currentPosition} * \text{interpolationFactor} + \text{previousPosition} * (1 - \text{interpolationFactor})$$

3.4 API gráfica OpenGL

OpenGL é a API encarregada da comunicação com a GPU. Através dela são realizadas todas as chamadas de função responsáveis por desenhar objetos na tela. Cada

Engine deve utilizar uma API dependendo da plataforma na qual o produto final será disponibilizado. Por exemplo, para um sistema mobile existe a API OpenGL ES, para Windows tem-se a OpenGL, DirectX etc. De certa forma, as APIs gráficas compõem essencialmente o sistema de renderização.

3.4.1 Core-profile e Immediate mode

Immediate mode (legado) é o modo antigo de se operar com OpenGL e foi depreciado em 2008 com o lançamento da versão 3.0 ([KHRONOS, 2018a](#)). Hoje ele é substituído pelo modo *Core-profile*. No modo antigo as funções eram mais fáceis de usar, com muitas funcionalidades já abstraídas pela API. Entretanto, por serem funções abstraídas elas forneciam uma menor flexibilidade de controle sobre como o OpenGL operava e eram também ineficientes ([VRIES, 2015](#)). Com o passar do tempo e uma demanda dos desenvolvedores por maior flexibilidade a API foi depreciada e substituída pelo modo *Core-profile*, em que tem-se muito mais controle sobre como o OpenGL opera. Entretanto, isso vem ao custo de uma maior curva de aprendizagem e complexidade de implementação.

3.4.2 State Machine

O OpenGL funciona como uma grande máquina de estados, possuindo uma enorme coleção de variáveis que definem como operar no estado vigente. Chamam-se as funções para configurar o estado atual, passar dados (coordenadas geográficas, cores e outras informações) ou alterar modos de operação. Um desses estados é o *rendering state* cujo divisão é feita em várias categorias como: *color*, *texturing*, *lighting* e assim por diante. O *rendering state* é manipulado pelas funções `glEnable(feature)` e `glDisable(feature)` que recebem como argumento um `enum` indicando qual o atributo que se deseja habilitar.

3.4.3 Hello window

A primeira etapa necessária para usar o OpenGL é criar um contexto e uma janela onde renderizar. O processo de criação da janela é específico de cada SO e faz-se necessário o uso de outra API para abstrair esse passo. A biblioteca a ser utilizada será a `GLFW`. O código dado na Listagem 3.6 demonstra esse processo.

```
1 public class Window {
2     private int width;
3     private int height;
4     private Vec2 size;
5     private String name;
6     private long id;
7
8     public Window(int width, int height, String name) {
9         this.width = width;
10        this.height = height;
11        this.name = name;
```

```

12     size = new Vec2(width, height);
13 }
14
15 public void init() {
16     if (!glfwInit())
17         System.err.println("Could not initialize GLFW.");
18
19     glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
20     glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
21     glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
22     glfwWindowHint(GLFW_RESIZABLE, GL_FALSE);
23
24     id = glfwCreateWindow(width, height, name, NULL, NULL);
25     glfwMakeContextCurrent(id);
26
27     glfwSetWindowPos(id, 2000, 60);
28     glfwShowWindow(id);
29     GL.createCapabilities();
30     glfwSwapInterval(1);
31
32     glViewport(0,0, width, height);
33     glEnable(GL_CULL_FACE);
34     glEnable(GL_BLEND);
35     glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
36 }
37 }
```

Listagem 3.6 – Inicialização da janela e contexto OpenGL

Primeiramente inicia-se o GLFW através da chamada `glfwInit()`. Em seguida usa-se a função `glfwWindowHint(hint, value)` para adicionar atributos a janela, sendo o primeiro argumento o nome do atributo (de tipo `enum`) e o segundo argumento o valor que se deseja atribuir. No exemplo definido para a função `init` acima, foram inseridos os atributos `GLFW_CONTEXT_VERSION_MAJOR` e `GLFW_CONTEXT_VERSION_MINOR` indicando qual a versão mínima para rodar a aplicação. Essa versão é representada no formato [MAJOR].[MINOR] e, portanto, neste cenário a versão é 3.3. O atributo `GLFW_OPENGL_PROFILE` altera entre o modo core profile e immediate mode (legado). Cria-se então a janela com o método `glfwCreateWindow(width, height, title, monitor, share)` e usa-se o `id` retornado para torná-lo contexto vigente com `glfwMakeContextCurrent(id)`. A função `glfwShowWindow(id)` é chamada para tornar essa janela visível ao usuário. Para utilizar o OpenGL e criar seu contexto é necessário chamar `GL.createCapabilities`. A partir deste ponto o OpenGL pode ser utilizado normalmente. A função `glViewport(0,0, width, height)` define o tamanho do viewport ou, simplesmente, o tamanho da área de renderização e é geralmente usado como tendo o mesmo tamanho da janela.

3.4.4 OpenGL Pipeline

No intuito de desenhar objetos na janela, é necessário abordar dois elementos do OpenGL: *buffer objects* e *pipeline*. Os *buffer objects* são estruturas responsáveis por transmitir e encapsular os dados a serem enviados para a GPU. Existem 9 tipos de objetos e estes são divididos em duas categorias ([KHRONOS, 2018b](#)):

1. **Regular objects:** objetos dessa categoria contêm dados.

- Buffer object
- RenderBuffer object
- Texture object
- Query object
- Sampler object

2. **Container objects:** objetos dessa categoria servem de containers para transportar os elementos da lista anterior (*regular objects*).

- Framebuffer object
- Vertex Array object
- Transform Feedback object
- Program pipeline object

Neste primeiro momento será abordado o *Vertex Buffer Object* (VBO) e *Vertex Array Object* (VAO). A fim de desenhar um objeto na tela faz-se necessário enviar a GPU os vértices que o compõem. O *pipeline* recebe como entrada uma série de vértices, os processa e transforma em pixels 2D na tela ([VRIES, 2015](#)). Cada etapa do *pipeline* recebe como entrada a saída da etapa anterior.

As etapas customizáveis do *pipeline* são processadas por pequenos programas chamados *shaders*. Esses programas são escritos em GLSL pelo desenvolvedor da aplicação e ficam alojados na GPU, sendo altamente paralelizáveis e especializados. Na Figura 10 tem-se uma representação simplificada do processo que ocorre no pipeline para desenhar um pequeno triângulo.

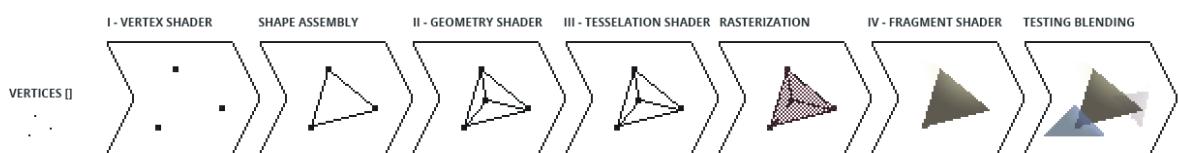


Figura 10 – Etapas do pipeline

Na etapa I o *vertex shader* é responsável, principalmente, por receber um único vértice e transformar sua coordenada cartesiana de um sistema para outro. Por exemplo, as coordenadas finais desse objeto vão mudar conforme a câmera se movimenta no espaço. É também necessário converter a posição local do objeto para a sua posição global no ambiente. Esses e outros processos serão melhor abordados no capítulo sobre Sistemas de Coordenadas.

Na etapa do *shape assembly* o OpenGL monta todos os vértices recebidos como entrada na primitiva selecionada para renderização. São algumas delas: *GL_POINTS*, *GL_LINES* e *GL_TRIANGLES*.

A etapa do *geometry shader* recebe como entrada a primitiva de saída do *shape assembly*. A partir desses vértices o *geometry shader* é capaz de gerar novos vértices e formar novas primitivas. No exemplo da Figura 10 ele recebe um triângulo e cria um novo vértice no centro, resultando em 3 sub-triângulos.

O *tesselation shader* é altamente especializado em subdividir uma primitiva em outras muito menores. Essa função é útil para, por exemplo, detalhar objetos mais próximos da tela e generalizar objetos mais distantes reduzindo sua quantidade de vértices.

A etapa de *rasterization* processa a saída do *tesselation shader* e converte todas essas informações em coordenadas 2D na tela ou, simplesmente, em pixels na tela. Esses pixels são então repassados como fragmentos para o *fragment shader*.

Na penúltima etapa do pipeline, o *fragment shader* processa todos os fragmentos para dar aos pixels sua cor final. É neste estágio que todos os efeitos visuais são aplicados como, por exemplo, a iluminação.

Por fim, no último estágio é aplicado o *blending* e outros testes. *Blending* nada mais é do que calcular o quanto a transparência de um objeto afeta outro. No exemplo da Figura 10 isso é representado pelos dois triângulos menores que afetam a cor final do triângulo maior aonde estes se sobrepõem. Neste estágio também são realizados os testes de *depth* e *stencil*.

3.4.5 Vertex Array Object e Vertex Buffer Object

Todo objeto no OpenGL é manipulado através de seu ID. Portanto, faz-se necessário gerar o ID do VAO e VBO chamando as funções `glGenVertexArrays()` e `glGenBuffers()`. Após gerados os IDs vincula-se esse *data object* como o vigente através da função `glBindBuffer(type, id)` e insere-se os dados nele através da função `glBindBuffer(type, data, draw_type)`. Para o *object container* o processo é similar. Vincula-se o VAO com `glBindVertexArray(id)` e habilita-se a localização do atributo do vértice no *vertex shader* através da função `glEnableVertexAttribArray(index)`. Por fim, configura-se como cada vértice deve ser interpretado, nesta situação como um

bloco de 4 floats, sendo 2 deles as posições x e y do objeto e os dois últimos a posição x e y da textura que compõem esse objeto. Essa configuração é feita pela função `glVertexAttribPointer(index, size, type, normalized, stride, pointer)`. O código de inicialização do VBO e VAO é dado na Listagem 3.7.

```

1  private void init() {
2      float vertices [] = {
3          //Pos //Texture
4          0,    1,    0,    1f,
5          1,    0,    1f,   0,
6          0,    0,    0,    0,
7
8          0,    1,    0,    1f,
9          1,    1,    1f,   1f,
10         1,    0,    1f,   0
11     };
12
13     int quadVAO = glGenVertexArrays();
14     int VBO = glGenBuffers();
15
16     glBindBuffer(GL_ARRAY_BUFFER, VBO);
17     glBufferData(GL_ARRAY_BUFFER, BufferUtilities.createFloatBuffer(getVertices(3)
18                     ), GL_STATIC_DRAW);
19
20     glBindVertexArray(quadVAO);
21     glEnableVertexAttribArray(0);
22     glVertexAttribPointer(0, 4, GL_FLOAT, false, Float.BYTES * 4, 0);
23
24     glBindBuffer(GL_ARRAY_BUFFER, 0);
25     glBindVertexArray(0);
}

```

Listagem 3.7 – Inicialização do VBO e VAO

A variável `vertices` contém os vértices no espaço local do objeto. Essas coordenadas são então posteriormente processadas no vertex shader e transformadas em coordenadas globais.

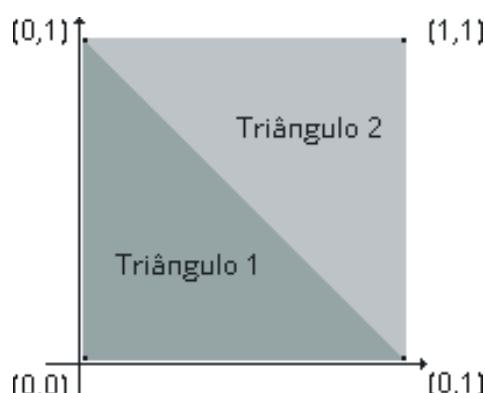


Figura 11 – Dois triângulos em coordenadas locais normalizadas que juntos formam um quadrado

3.4.6 Vertex e Fragment shader

Após a geração do objeto, configuração e posterior transferência dos dados para a GPU, pode-se iniciar o processo de manipulação desses dados no *shader*. Como ilustrado na Figura 10, o primeiro shader do pipeline é o *vertex shader*. Todo código escrito no shader é iniciado com a versão do OpenGL. Neste caso, a versão utilizada é a 3.3 e o modo, que deve ser explicitamente citado, é o modo *core profile*. O index layout, onde o dado é recebido no shader, para este exemplo é 0 e usa o formato de um vetor de 4 floats onde as coordenadas xy são a posição e as coordenadas zw são a posição da textura. O código do cabeçalho é dado na Listagem 3.8.

```
1 #version 330 core
2 layout (location = 0) in vec4 vertex;
```

Listagem 3.8 – Cabeçalho do Vertex shader

Definido o cabeçalho e index layout é necessário dizer quais serão as saídas do shader, ou simplesmente, seu retorno que servirá de entrada para a próxima etapa do pipeline. Isso é feito usando-se a palavra reservada **out** seguida do tipo e nome da variável. Ainda neste escopo são definidas as variáveis do tipo **uniform**, que funcionam como variáveis globais e podem ser acessadas em qualquer shader. Em seguida declara-se a função **main**, responsável pelo processamento destes dados. O processo completo é demonstrado no código da Listagem 3.9.

```
1 #version 330 core
2 layout (location = 0) in vec4 vertex;
3
4 out vec2 TexCoords;
5
6 uniform mat4 model;
7 uniform mat4 projection;
8 uniform mat4 camera;
9
10 uniform vec2 flip;
11
12 void main(){
13     TexCoords = vertex.zw;
14     gl_Position = projection * camera * model * vec4(vertex.xy, 0.0 , 1.0);
15 }
```

Listagem 3.9 – Vertex shader simples

Esse shader recebe um vetor de 4 floats contendo as coordenadas *x* e *y* do objeto e as coordenadas *z* e *w* da textura. As coordenadas da textura são passadas adiante para o fragment shader através da variável **out vec2 TexCoords** e as coordenadas do vértice são multiplicadas pelas matrizes de projeção, câmera e modelo para obter as coordenadas globais. O resultado dessa multiplicação é armazenado na variável global embutida no OpenGL chamada **gl_Position**.

A próxima etapa a ser abordada é a do fragment shader. Para a construção desse primeiro exemplo a única manipulação que será feita nos fragmentos é a renderização da textura. Com a entrada da etapa anterior (vertex shader) extrai-se da textura contida na variável `image` a fatia desejada usando as coordenadas em `TexCoords`. Esse resultado é então enviado para a última etapa usando-se a variável `out vec4 color`. O código do fragment shader é dado na Listagem 3.10. O resultado final obtido com os shaders apresentados é ilustrado pela Figura 12.

```
1 #version 330 core
2 in vec2 TexCoords;
3 out vec4 color;
4
5 uniform sampler2D image;
6
7 void main(){
8     vec4 imgTex = texture(image, TexCoords).xyzw;
9     color = imgTex;
10 }
```

Listagem 3.10 – Fragment shader simples



Figura 12 – Resultado obtido com dois shaders simples

3.4.7 Compilando o shader em um programa

Com o código dos shaders salvos em arquivo é necessário compilá-los e enviá-los a GPU antes de poderem ser efetivamente usados. Assim como qualquer objeto no

OpenGL o primeiro passo é gerar um ID para referenciá-lo. Esse processo é feito chamando `glCreateShader(type)`. Os tipos possíveis de shader são aqueles demonstrados e numerados de I a IV na figura 10. Portanto, os valores possíveis são: `GL_COMPUTE_SHADER`, `GL_VERTEX_SHADER`, `GL_TESS_CONTROL_SHADER`, `GL_TESS_EVALUATION_SHADER`, `GL_GEOMETRY_SHADER`, e `GL_FRAGMENT_SHADER` ([KHRONOS, 2018c](#)). Com o objeto criado atribui-se o código ao objeto e o mesmo é compilado a partir das funções `glShaderSource(id, code)` e `glCompileShader(id)`. Para checar se a compilação foi realizada com sucesso chama-se a função `glGetShaderi(id, GL_COMPILE_STATUS)`. Se o retorno for 0 (FALSE) não houve erros. Caso contrário, a função `glGetShaderInfoLog(id)` retorna a mensagem de erro, que pode ser posteriormente impressa na tela. A primeira etapa do processo é demonstrado na Listagem 3.11.

```

1 int vertexShader = glCreateShader(GL_VERTEX_SHADER);
2 glShaderSource(vertexShader, vertexSource);
3 glCompileShader(vertexShader);
4
5 int success = glGetShaderi(vertexShader, GL_COMPILE_STATUS);
6
7 if(success==0) {
8     String log = glGetShaderInfoLog(vertexShader);
9     System.out.println(log);
10 }
```

Listagem 3.11 – Processo de compilação do shader

Tendo os shaders necessários compilados agora é necessário criar um programa. Essa etapa segue um padrão baseado no processo de dois estágios para compile/link usada nos programas escritos em C e C++. O código fonte é primeiro servido ao compilador, produzindo um object file. Para obter o executável final é necessário encadear um ou mais object files juntos. Com o programa criado e os shaders encadeados a ele é possível utilizar todos esses shaders com apenas uma chamada do programa. O processo de link e criação do programa é demonstrado na Listagem 3.12. Os shaders uma vez encadeados podem ser removidos para liberar espaço usando a função `glDeleteShader(id)`.

```

1 int id = glCreateProgram();
2 glAttachShader(id, vertexShader);
3 glAttachShader(id, fragmentShader);
4 glLinkProgram(id);
5
6 int success = glGetProgrami(id, GL_LINK_STATUS);
7
8 if(success==0) {
9     String log = glGetProgramInfoLog(id);
10    System.out.println(log);
11 }
```

Listagem 3.12 – Processo de criação e link de um programa

3.4.8 Uniforms

Existem duas formas de se enviar dados para um shader, através dos *objects* e através das variáveis globais definidas como `uniform`. Para alocar um valor nessas variáveis é necessário primeiro usar o programa que contém este shader com a função `glUseProgram(id)` e em seguida usar uma das funções da família `glUniform...()`. O formato dessas funções é `glUniform+quantity+type`. Por exemplo, para atribuir um valor a uma variável float utiliza-se `glUniform1f (glGetUniformLocation (programID, variableName), value)`, para um vetor de 3 floats usa-se `glUniform3f(glGetUniformLocation (programID,variableName), x, y, z)`, para uma matriz 4x4 por `glUniformMatrix4fv(glGetUniformLocation (programID,variableName), false, BufferUtilities.createFloatBuffer(m))` e assim por diante. A classe final do shader é dada no Apêndice [A](#).

3.5 Renderizador

O sistema de renderização fica responsável por encapsular todas as operações gráficas da engine. Independente da API em utilização o sistema deve abstrair suas funções para que sejam de uso simples e independente. Para isso as funções devem receber parâmetros genéricos que descrevam o objeto apenas em termos de dados.

3.5.1 Textura

Com o shader compilado e encadeado em um programa e os VBO's e VAO's prontos tudo que falta para renderizar a primeira imagem é carregá-la do arquivo e transferí-la para o shader usando uma *texture unit*. Esse processo tem início, como todo objeto no OpenGL, gerando um *id* via função. Para texturas a função utilizada é `glGenTextures()`. Em Java utiliza-se a classe `BufferedImage` para ler a imagem do arquivo e depois passar os pixels em formato de Buffer para o OpenGL. A classe completa da textura é dada na Listagem 3.13.

```
1 public class Texture {
2     private int id;
3     private BufferedImage textureImage;
4     private static final int BYTES_PER_PIXEL = 4;
5     private int width, height;
6
7     public Texture(String path) {
8         id = glGenTextures();
9
10    try {
11        textureImage = ImageIO.read(getClass().getResourceAsStream(path));
12        width = textureImage.getWidth();
13        height = textureImage.getHeight();
```

```

14     } catch (IOException e) {
15         e.printStackTrace();
16     }
17
18     init();
19 }
20
21 private void init() {
22     int[] pixels = new int[textureImage.getWidth() * textureImage.getHeight()];
23     textureImage.getRGB(0, 0, textureImage.getWidth(), textureImage.getHeight(),
24                         pixels, 0, textureImage.getWidth());
25     ByteBuffer buffer = BufferUtilities.createByteBuffer(
26             new byte[textureImage.getWidth() * textureImage.getHeight()
27                     () * BYTES_PER_PIXEL]
28         ); //4 for RGBA, 3 for RGB
29
30     for(int y = 0; y < textureImage.getHeight(); y++){
31         for(int x = 0; x < textureImage.getWidth(); x++){
32             int pixel = pixels[y * textureImage.getWidth() + x];
33
34             buffer.put((byte) ((pixel >> 16) & 0xFF)); // Red component
35             buffer.put((byte) ((pixel >> 8) & 0xFF)); // Green component
36             buffer.put((byte) (pixel & 0xFF)); // Blue component
37             buffer.put((byte) ((pixel >> 24) & 0xFF)); // Alpha component
38         }
39     }
40
41     buffer.flip();
42
43     glBindTexture(GL_TEXTURE_2D, id); //Bind texture ID
44
45     //Setup wrap mode
46     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL12.GL_CLAMP_TO_EDGE)
47         ;
48     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL12.GL_CLAMP_TO_EDGE)
49         ;
50
51     //Setup texture scaling filtering
52     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
53     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
54
55     //Send texel data to OpenGL
56     glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, textureImage.getWidth(),
57                 textureImage.getHeight(), 0, GL_RGBA, GL_UNSIGNED_BYTE, buffer);
58
59     glBindTexture(GL_TEXTURE_2D, 0); //Unbind texture
60     textureImage = null; //'free' BufferedImage
61 }
62 }
```

Listagem 3.13 – Inicializando uma textura no OpenGL a partir de um BufferedImage

Primeiro cria-se um vetor de inteiros com tamanho da área total da imagem (width*height). Logo em seguida esse vetor é preenchido com os pixels do BufferedImage usando `textureImage.getRGB(startX, startY, width, height, toArray, offset,`

`scanSize`). Com todos os pixels da imagem no array `pixels` basta criar um `ByteBuffer` usando a função auxiliar `BufferUtilities.createByteBuffer()`, cuja implementação é dada na Listagem 3.14.

```

1  public class BufferUtilities {
2      public static ByteBuffer createByteBuffer(byte[] array) {
3          ByteBuffer result = ByteBuffer.allocateDirect(array.length).order(ByteOrder.
4              nativeOrder());
5          result.put(array).flip();
6          return result;
7      }

```

Listagem 3.14 – Função auxiliar `createByteBuffer`

Com o byte buffer alocado a próxima etapa é preenchê-lo com os dados contidos no array `pixels`. O formato padrão escolhido para a engine Narval foi o RGBA com canais de 1 byte cada, em que cada canal pode assumir valores entre 0 e 255 (2^8). Entretanto, a função `getRGB()` retorna um espaço de cores no formato ARGB e precisa ser convertido para RGBA antes da inserção no buffer. Os dois formatos são ilustrados pela Figura 13.

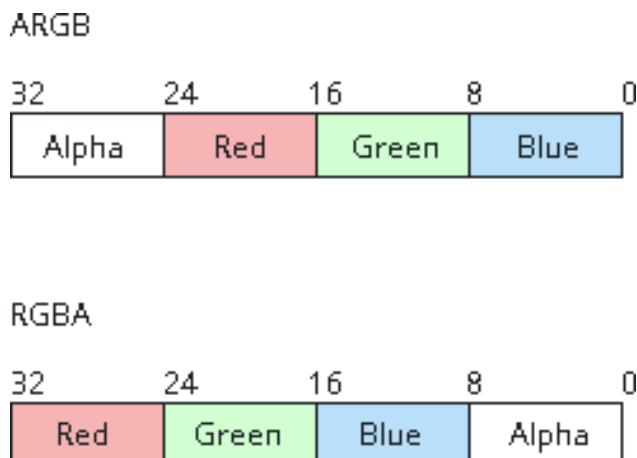


Figura 13 – Sistema ARGB e RGBA, ambos com canais de 8 bits (1 byte)

Para realizar essa conversão basta aplicar um deslocamento de 24, 16 ou 8 bits conforme o canal e depois aplicar uma máscara de $0x0000FF_{16}$ bits (e.g 11111111_2) para preservar somente o byte menos significativo. O tamanho do deslocamento depende da posição em que o canal se encontra. A extração do canal verde é exemplificada na Figura 14.

R:192	G:128	B:255
110000000	100000000	11111111
>>8		
<hr/>		
000000000	110000000	100000000
&&		
000000000	000000000	11111111
<hr/>		
G: 100000000		

Shift 8 bits to the right

Apply mask 0x0000FF

Figura 14 – Extração do canal verde em um inteiro de 3 bytes

Uma vez que todos os dados foram inseridos no buffer, usa-se `buffer.flip()`. Antes de qualquer operação na textura vincula-se ela como atual usando `glBindTexture(type, id)`. Para este exemplo será utilizado o tipo `GL_TEXTURE_2D`, indicando uma imagem normal em 2D. Em seguida configura-se o modo de *wrapping*. Por padrão, quando as coordenadas de uma textura excedem seu tamanho o openGL as repete com o modo `GL_REPEAT`. Entretanto, existem 4 modos possíveis: `GL_REPEAT`, `GL_MIRRORED_REPEAT`, `GL_CLAMP_TO_EDGE` e `GL_CLAMP_TO_BORDER` ilustrados pela Figura 15.

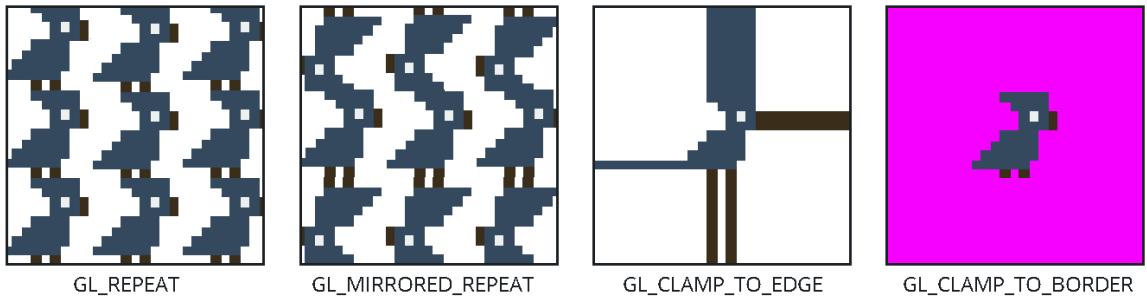


Figura 15 – Wrap modes

Esses modos são configurados através da função `glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE)`. O segundo argumento indica o eixo para o qual se deseja aplicar a configuração, sendo eles S e T, equivalentes a x e y. Em seguida é escolhida a fórmula utilizada para escalar a imagem. São dois modos possíveis: `GL_NEAREST` e `GL_LINEAR`. Cada um desses modos é ilustrado na Figura 16.

O modo `GL_NEAREST` é o modo *default* do openGL e opera escolhendo o pixel cujo centro é o mais próximo da coordenada da textura. De uma maneira simples é o modo que ao ampliar a imagem não é aplicada nenhuma suavização, passando uma impressão de imagem serrilhada/pixelizada.

O modo `GL_LINEAR` interpola um valor entre os pixels vizinhos. É o modo que ao ampliar ou diminuir a imagem causa uma impressão de bordas mais suaves.



Figura 16 – Ampliação de 32x usando cada tipo de filtro.

Esses modos podem ser aplicados distintamente para operações de aumentar ou diminuir a imagem. O atributo responsável por dizer em qual das duas situações o filtro deve ser aplicado é: `GL_TEXTURE_MAG_FILTER` e `GL_TEXTURE_MIN_FILTER`. Sendo o primeiro para operações de *magnifying* (aumento) e o segundo para operações de *minifying* (diminuir).

Por fim, com todos esses parâmetros configurados envia-se o bytebuffer contendo os dados da imagem propriamente dita para o OpenGL. Isso é feito na função `glTexImage2D(type, mipmap_level, internal_format, width, height, border, color_format, type, bytebuffer)`.

3.5.2 Animações

Uma vez que todo o sistema de shader program e texturas estão montados é possível usá-los para construir componentes mais avançados, como o de animação. Um objeto em animação não é nada mais que uma sequência de frames desenhados em um intervalo de milissegundos, criando ilusão de movimento. É comum a utilização de *spritesheets* para armazenar todas as imagens em um único arquivo e então recortá-las em pedaços (frames) em tempo de execução. O exemplo de uma *spritesheet* é dado na Figura 17.

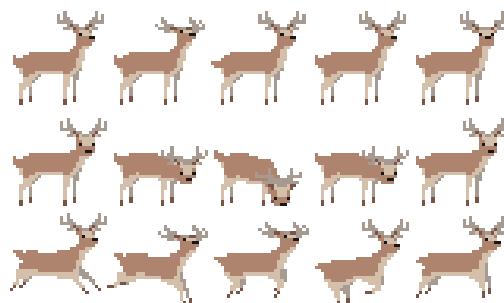


Figura 17 – Spritesheet com frames de 32x32 pixels

Para que esse recorte seja possível é necessário acrescentar alguns parâmetros ao shader para que este saiba qual pedaço deve ser recortado. Essa tarefa pode ser facilmente realizada alterando as coordenadas locais da textura enviadas através do atributo

`glVertexAttribArray` 0 definido anteriormente. A implementação do vertex shader com animações é dada na Listagem 3.15.

```
1 #version 330 core
2 layout (location = 0) in vec4 vertex;
3
4 out vec2 TexCoords;
5 out vec3 FragPos;
6
7 uniform mat4 model;
8 uniform mat4 projection;
9 uniform mat4 camera;
10 uniform vec4 spriteFrame;
11
12 void main(){
13     vec2 tCoords = vertex.zw;
14
15     tCoords *= spriteFrame.zw;
16     tCoords += spriteFrame.xy;
17
18     TexCoords = tCoords;
19     gl_Position = projection * camera * model * vec4(vertex.xy, 0.0, 1.0);
20     FragPos = vec3(model * vec4(vertex.xy, 0.0, 1.0));
21 }
```

Listagem 3.15 – Vertex shader com animações

Na variável `spriteFrame` tem-se as coordenadas `xy` de qual pedaço da textura deverá ser extraído e nas coordenadas `zw` a largura e altura desse pedaço. Essa informação é então repassada para o fragment shader onde por fim a função `texture()` irá colher esses pixels para manipulação.

Dessa forma o código da classe responsável pela manipulação lógica da animação é definido na Listagem 3.16. A classe pode ser ainda estendida para comportar frames com tempo de duração específico. Para isso basta substituir o parâmetro `long frameDuration` por um vetor com o tempo de duração correspondente a cada frame e alterar a lógica do método `update`.

```
1 public class Animation implements Serializable{
2     private Vec4 frames[];
3     private int currentFrame = 0;
4     private String texture;
5     private long frameDuration = -1;
6     private long startTime = System.nanoTime();
7     private boolean playedOnce = false;
8
9     public Animation(String texture, long frameDuration) {
10         this.texture = texture;
11         this.frameDuration = frameDuration;
12     }
13
14     public void setFrames(int quantity, Vec2 offset, Vec2 size) {
15
16         frames = new Vec4[quantity];
```

```

17     float width = ResourceManager.getSelf().getTexture(texture).getWidth();
18     float height = ResourceManager.getSelf().getTexture(texture).getHeight();
19
20     for(int i= 0; i< quantity; i++){
21         frames[i] = new Vec4(
22             ((float)i*size.x + offset.x)/width,
23             (offset.y)/height,
24             (size.x)/width,
25             (size.y)/height
26         );
27     }
28
29 }
30
31 public Vec4 getCurrentFrame() {
32     return frames[currentFrame];
33 }
34
35 public void update() {
36     if(frameDuration>0) {
37         long elapsed = (System.nanoTime() - startTime) / Engine.MILLISECOND;
38
39         if(elapsed > frameDuration) {
40             currentFrame++;
41             startTime = System.nanoTime();
42         }
43         if(currentFrame == frames.length) {
44             currentFrame = 0;
45             playedOnce = true;
46         }
47     }
48 }
49 }
```

Listagem 3.16 – Classe Animation

3.5.3 Z-Ordering

A técnica de z-ordering consiste na ordenação de objetos em sobreposição. É utilizada para organizar a ordem na qual os elementos devem ser renderizados tal que não haja elementos distantes sendo desenhados em cima de objetos mais próximos da câmera. Para ambientes 2D, basta ordenar os elementos pela posição y do menor para o maior, como ilustrado pela Figura 18.



Figura 18 – Ordem correta de renderização: do menor y para o maior y

Deve atentar-se na implementação desta técnica para ordenar apenas os objetos que estão no espaço de visão da câmera, otimizando assim a performance do algoritmo de ordenação. É ainda necessário, para dar um melhor senso de profundidade, considerar a posição dos objetos a partir da sua base que toca o solo. Isto é, considerar que sua posição y no espaço seja relativa ao ponto no qual o objeto toca o chão invés de ser simplesmente o canto superior esquerdo da imagem. Esse mesmo princípio de caixa base é utilizada para calcular a colisão dos objetos que caminham pelo ambiente. O resultado final obtido utilizando este princípio é ilustrado na Figura 19.



Figura 19 – Ordem correta de renderização: do menor y para o maior y considerando a caixa base

3.6 Iluminação

Existem muitos modelos de iluminação com equações sofisticadas que conseguem simular a luz de maneira muito próxima da realidade. Entretanto, estes modelos mais robustos não costumam servir para um sistema de tempo real pois são muito caros em

perfomance, sendo geralmente utilizados em outras indústrias como, por exemplo, a cinematográfica. O modelo de iluminação adotado e adaptado nesta engine foi o [Phong \(1975\)](#) que usa três componentes de luz em sua construção: ambient, diffuse e specular ([SCRATCHAPIXEL, 2018](#)). Os três componentes são melhor descritos visualmente pela Figura 20.



Figura 20 – Componentes do modelo de Phong

3.6.1 Ambiente

A luz ambiente representa as fontes de luz mais distantes que estão quase sempre presentes no mundo real como, por exemplo, o sol. São essas fontes que garantem a visibilidade quase constante dos objetos pois, afinal, sem luz o ser humano não é capaz de perceber o mundo visível.

Vale ressaltar duas propriedades importantes da luz: reflexão e refração. A iluminação que advém de qualquer fonte luminosa pode impactar objetos mesmo que estes não estejam sobre alcance direto. Esse efeito advém da propriedade da reflexão e é chamado de luz indireta. Algoritmos que levam esse fator em consideração são chamados de *global illumination*, mas geralmente custam caro em performance ou são muito complexos. Para esta engine será adotado um modelo simplificado que consiste em multiplicar a cor do objeto por uma variável `lightAmbientColor`, que define a cor e intensidade da luz que ilumina todo o mundo. A Listagem 3.17 expõe essa versão do shader.

```

1 #version 330 core
2 in vec2 TexCoords;
3 out vec4 color;
4
5 uniform sampler2D image;
6
7 void main(){
8     vec4 lightAmbientColor = vec4(1.1,0.7,0.25,1);
9
10    vec4 imgTex = texture(image, TexCoords).xyzw;
11
12    color = imgTex*lightAmbientColor;
13 }
```

Listagem 3.17 – Fragment Shader com luz ambiente

Dessa forma o resultado que se obtém com a cor acima é um ambiente de tonalidade um pouco alaranjada, como ilustrado pela Figura 21.



Figura 21 – Demonstração da luz ambiente com tonalidade alaranjada

3.6.2 Luz Difusa

A luz difusa simula o impacto direcional que uma fonte de luz tem sobre um objeto qualquer. Quanto mais perto o objeto está dessa fonte, mais iluminado ele será.

Para calcular a luz difusa é necessário antes ter em mãos o vetor normal do objeto que será atingido pelos raios de luz. Dessa forma, quando o raio de luz atinge o objeto calcula-se o ângulo de incidência θ entre os dois vetores usando o produto escalar. Note-se que o ângulo de reflexão é sempre igual ao ângulo de incidência. A Figura 22 ilustra esse processo.

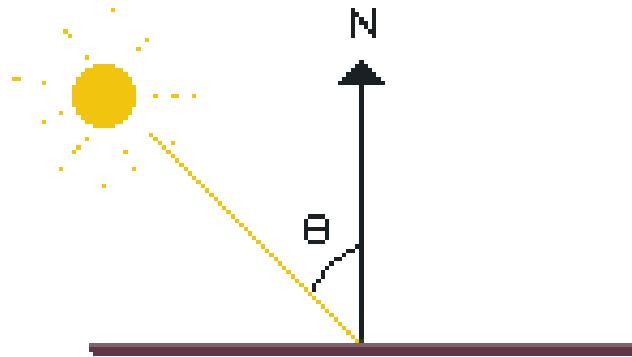


Figura 22 – Ângulo θ entre raio luminoso e vetor normal da superfície iluminada

O resultado desse produto escalar indica o impacto que a luz terá sobre a cor do fragmento sendo processado pelo shader, tendo em conta sua orientação relativa à luz. Para realizar o cálculo será necessário então um vetor com a posição da fonte luminosa e o vetor normal do fragmento sendo afetado por essa fonte. O vetor normal será obtido

através de uma *normal texture* e o vetor da fonte luminosa através de uma variável **global uniform**. O vetor normal é pré-calculado e armazenado em uma imagem enviada ao shader juntamente com as *uniforms* posição e cor da fonte luminosa. O cálculo é então feito normalizando a subtração do **lightPos** pela posição do fragmento, obtendo assim o vetor direção da luz. Com o valor normalizado realiza-se o produto escalar da direção da luz pela normal contida na textura. Multiplica-se então o resultado pela cor da luz e ao final soma-se a luz difusa a luz ambiente. Esse resultado é multiplicado pela textura para obter o resultado final. O código dado na Listagem 3.18 demonstra esse processo e o resultado obtido com ele é ilustrado pela Figura 23.

```

1 #version 330 core
2
3 in vec2 TexCoords;
4 in vec3 FragPos;
5
6 out vec4 color;
7
8 uniform sampler2D image;
9 uniform sampler2D normalTex;
10
11 uniform vec3 lightPos;
12 uniform vec3 lightColor;
13
14 void main(){
15
16     vec3 normal = normalize(texture(normalTex, TexCoords).xyz);
17
18     vec3 lightDir = normalize(lightPos - FragPos);
19     float diff = max(dot(normal, lightDir), 0.0);
20     vec3 diffuse = diff * lightColor;
21
22     vec4 lightAmbientColor = vec4(1.1,0.7,0.25,1);
23     vec4 imgTex = texture(image, TexCoords).xyzw;
24
25     color = imgTex * (lightAmbientColor + vec4(diffuse.xyz,1));
26 }
```

Listagem 3.18 – Fragment Shader com luz ambiente e difusa



Figura 23 – Resultado da luz difusa usando a textura normal do objeto

3.6.3 Vtor Normal

Um vtor normal unitrio é aquele perpendicular à superficie. A tcnica habitualmente utilizada para trabalhar com vetores normais de um objeto consiste em mapeá-los para uma textura de forma que, para cada pixel da textura original, haja um vtor normal. Como o vtor normal precisa de apenas 3 coordenadas (x,y,z) isso é facilmente feito calculando-se a normal de cada pixel da textura e convertendo esse vtor para as coordenadas de cores RGB, que sáo por fim armazenadas em uma outra imagem chamada de textura normal.

3.6.4 Luz Especular

A luz especular é o ponto brilhante que aparece em objetos lisos e reluzentes, ajudando numa melhor percepção do espaço 3D e da textura do objeto que a reflete. A luz especular é mais intensa e presente numa superficie perfeitamente lisa de, por exemplo, um espelho. O brilho gerado por essa fonte luminosa é da cor da luz e não da cor do objeto.

Assim como a luz difusa, a luz especular possui o vtor de direo da luz e o vtor normal do objeto. Entretanto, ela tambm é calculada baseando-se na direo em que a cmera est olhando. A Figura 24 ilustra esse processo.

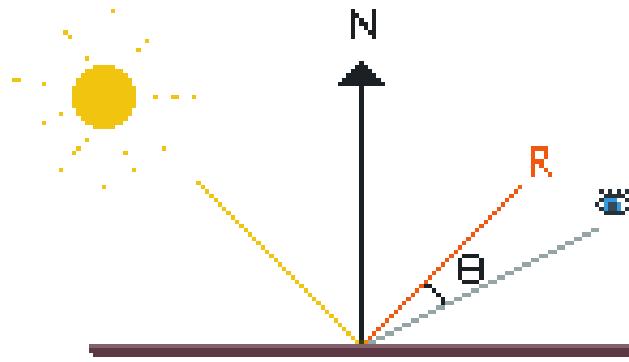


Figura 24 – Luz especular do ponto de vista do observador

3.6.5 Atenuação

A intensidade da luz diminui quanto maior a distância de um objeto, começando intensa e rapidamente desvanecendo. Aplicar uma fórmula linear para calcular esse efeito daria uma impressão errônea do comportamento da luz, tratando-a como se fosse uniforme. Portanto, faz-se necessário utilizar uma fórmula a seguir que permite uma melhor simulação deste comportamento:

$$F = \frac{1}{(K_c + K_l * d + K_q * d^2)}$$

sendo que:

- K_c é uma constante arbitrária para garantir que o denominador nunca seja 0;
- K_l é uma constante linear;
- K_q é uma constante quadrática; e
- d é a distância do fragmento até a fonte luminosa.

Existe uma tabela para auxiliar na escolha do valor dessas constantes para distâncias entre 0 e 3200. A tabela 1 pode ser facilmente estendida para distâncias maiores.

Range	Constant	Linear	Quadratic
3250	1.0	0.0014	0.000007
600	1.0	0.007	0.0002
325	1.0	0.014	0.0007
200	1.0	0.022	0.0019
160	1.0	0.027	0.0028
100	1.0	0.045	0.0075
65	1.0	0.07	0.017
50	1.0	0.09	0.032
32	1.0	0.14	0.07
20	1.0	0.22	0.20
13	1.0	0.35	0.44
7	1.0	0.7	1.8

Tabela 1 – Tabela de valores exemplo para a função de atenuação

A implementação do fator de atenuação é feita diretamente no shader multiplicando os valores da luz ambiente, difusa e especular e é dado na Listagem 3.19.

```

1 #version 330 core
2
3 in vec2 TexCoords;
4 in vec3 FragPos;
5
6 out vec4 color;
7
8 uniform sampler2D image;
9 uniform sampler2D normalTex;
10
11 uniform vec3 lightPos;
12 uniform vec3 lightColor;
13
14 void main(){
15
16     vec3 normal = normalize(texture(normalTex, TexCoords).xyz);
17
18     vec3 lightDir = normalize(lightPos - FragPos);
19     float diff = max(dot(normal, lightDir), 0.0);
20     vec3 diffuse = diff * lightColor;
21
22     float distance = length(lightPos - vec3(FragPos.xyz));
23     float constant = 1;
24     float linear = 0.007;
25     float quadratic = 0.0002;
26
27     float attenuation = 1 / (constant + linear * distance + quadratic * (distance *
28                               distance));
29
30     vec4 lightAmbientColor = vec4(1.1, 0.7, 0.25, 1);
31     vec4 imgTex = texture(image, TexCoords).xyzw;
32
33     color = vec4((imgTex.xyz + lightAmbientColor + (lightColor * attenuation*0.8)
34                  ) * attenuation, imgTex.w);

```

```
33 }
```

Listagem 3.19 – Fragment Shader com luz ambiente, difusa e atenuação

3.7 Classe Renderer

A classe renderer é responsável por conter e controlar todos métodos de desenho do sistema. Nela define-se uma função única tal que não seja necessário, a todo momento, configurar o shader e atualizar diretamente as variáveis `uniform`, `VBO` e `VAO`. Ela abstrai todos esses processos em uma simples chamada que recebe como argumento as coordenadas, textura e afins do objeto. Dessa forma, o código 3.7 será incorporado em uma classe renderer juntamente com o método render descrito a seguir. Note que é importante abstrair como parâmetros apenas o que é necessário para a renderização, evitando passar objetos específicos do jogo em criação. Dessa maneira o sistema se mantém flexível o suficiente para desenvolver jogos de qualquer tipo. A classe final utilizada para renderização é dada na Listagem 3.20.

```
1 public class TextureRenderer{
2
3     public TextureRenderer(){
4         init();
5     }
6
7     private void init(){
8         ...
9         //codigo do algoritmo 3.7 - inicializacao do VBO e VAO
10    }
11
12    public void render(Texture texture, Vec2 position, Vec2 size, float rotate,
13        Vec4 color, Vec4 spriteFrame) {
14        this.shader.use();
15        Mat4 model = new Mat4();
16
17        model = model.translate(position.x, position.y, 0);
18        model = model.translate(0.5f * size.x, 0.5f * size.y, 0); //Move a origem da
19        rotacao para o centro do objeto
20        model = model.rotate(rotate, 0, 0, 1); // Deve estar em radianos
21        model = model.translate(-0.5f * size.x, -0.5f * size.y, 0); //Retorna a
22        origem de rotacao de volta para o canto superior esquerdo
23        model = model.scale(size.x, size.y, 1);
24
25        shader.setMat4("model", model);
26        shader.setVec4("spriteColor", color);
27        shader.setVec2("flip", orientation);
28        shader.setVec4("spriteFrame", spriteFrame);
29
30        shader.setInteger("image", 0);
31
32        glActiveTexture(GL_TEXTURE0);
33        texture.bind();
```

```

32     glBindVertexArray(quadVAO);
33     glDrawArrays(GL_TRIANGLES, 0, 6*3);
34     glBindVertexArray(0);
35 }
36 }
```

Listagem 3.20 – Classe renderer simples

Dessa forma, para renderizar os objetos itera-se numa lista e para cada objeto chama-se a função `TextureRenderer.render()`, como demonstrado na Listagem 3.21.

```

1 \label{alg:dprender}
2   for(GameObject obj: listOfObjects){
3     textureRenderer.render(obj.getTexture(), obj.getPosition(), obj.getSize(),
4                           obj.size(), ...);
4 }
```

Listagem 3.21 – Demonstração do processo de renderização

Entretanto, esta implementação não é ideal pois faz-se necessário realizar n chamadas à função `render`, sendo n a quantidade de objetos na lista. Ou seja, se há um milhão de objetos na tela serão realizadas um milhão de chamadas ao método `render` e consequentemente um milhão de trocas de contexto de shader em `this.shader.user()` mais 4 transferências de informação para as variáveis uniform por chamada do método. Toda essa troca de contexto e transferência de dados entre CPU e GPU possuem um impacto enorme no desempenho. E quanto maior a lista de objetos a serem desenhados pior será a performance.

Para fins de introdução ao conceito e simplificação do processo de renderização abordou-se o modelo acima. Entretanto, o problema é solucionado implementando a técnica de *batch rendering*, que realiza uma única chamada do método `render` por tipo de shader e texture. Dessa forma, a transferência dos dados será feita em um único e enorme pacote, evitando gargalo no BUS da placa mãe e *overhead* de funções.

3.8 Arquitetura do Game Object

Os *game objects* (objetos de jogo), também conhecidos como entidades, estão no centro de tudo aquilo que dá vida ao cenário do jogo. São esses arquétipos que definem NPC's, inimigos, vida selvagem e até mesmo o próprio jogador. Existem três arquiteturas muito presentes e comuns no desenvolvimento de games: *Component Based*, *Object Oriented* e *Entity System* ([WENDERLICH, 2018](#)).

3.8.1 Object Based

Uma arquitetura baseada em objetos trabalha sobre o conceito de hierarquia de classes. Haverá uma classe `GameObject` que é então especializada para cada tipo de objeto.

Por exemplo, uma classe `Minotauro` estende uma classe `Monstro` que, por sua vez, estende a classe `GameObject`. Da mesma forma uma classe `Besouro` estende a classe `Monstro` que estende a classe `GameObject`.

O problema dessa implementação é que quanto mais especializações são criadas mais difícil e complexo se torna a manutenção dessa estrutura. Para cada nova regra implementada em uma especialização surgem exceções que começam a tornar necessário uma generalização ainda maior, resultando em uma classe `GameObject` extensa e com recursos muitas vezes usados apenas por uma única especialização.

Isso ainda pode causar relações de hierarquia que não fazem sentido lógico como, por exemplo, uma armadilha que estende a classe `Monstro` só para ser capaz de usar o sistema de dano. A Figura 25 demonstra esse problema através de um diagrama.

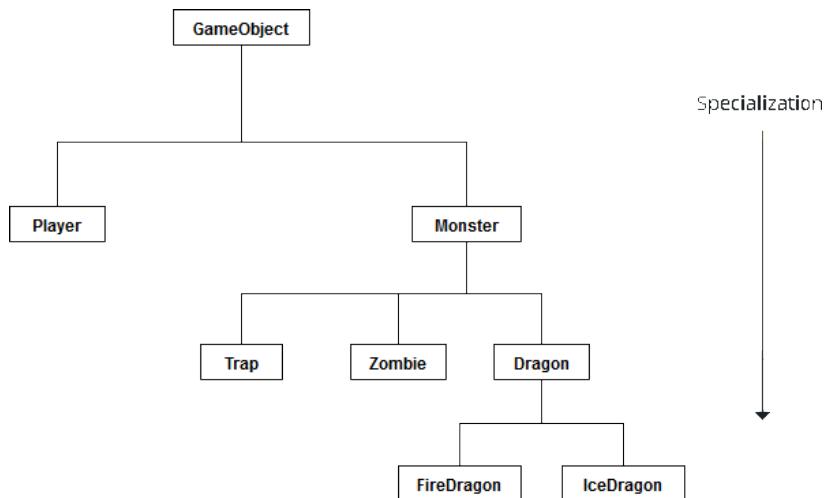


Figura 25 – Especialização das classes numa arquitetura baseada em objetos

3.8.2 Component Based

A arquitetura baseada em componentes, também conhecida pelo nome de *behavior*, consiste no uso de composição invés de herança. Ao contrário da especialização e generalização utilizados na herança o sistema de composição consiste em criar vários pequenos componentes e então “especializar” um `GameObject` atribuindo a ele os componentes que se deseja. Dessa forma, para que um `GameObject` seja um `Monstro` basta adicionar um componente de dano, vida e IA, por exemplo.

Dessa forma, quando o sistema for salvar o *game object* não será desperdiçada memória salvando dados de outros sistemas que não estão sendo utilizados por esse objeto pois, da maneira agora proposta, cada *game object* terá apenas os componentes de que necessita e fará uso.

Existem várias maneiras de se implementar essa arquitetura. Através do uso de *switches* para assinalar qual componente está ativo ou não, através do padrão de pro-

jeto com componentes passando mensagem entre outros. A maior desvantagem dessa arquitetura está na relação acoplada entre dados e o sistema que os manipula.

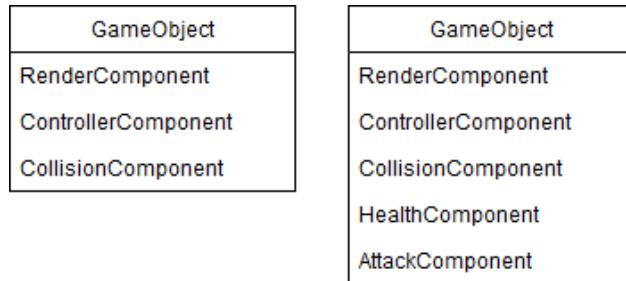


Figura 26 – GameObject composto de vários componentes

3.8.3 Entity Based

Um game object baseado em entidades trabalha de maneira similar à forma como um banco de dados trata suas informações. O sistema que manipula as informações é desacoplado dos dados em si. Dessa maneira, uma entidade é composta apenas de dados e um sistema à parte é responsável por processar essas informações e realizar suas tarefas. Assim, uma entidade do tipo Monstro precisa ter associado a ela apenas os atributos pertinentes como dano, vida e IA. O sistema então responsável verifica se a entidade possui esses atributos e, caso tenha, realiza seus cálculos senão, o sistema ignora essa entidade e avança para a próxima da lista. Esse sistema é ilustrado no diagrama da Figura 27 e sua implementação completa pode ser encontrada no Apêndice C.

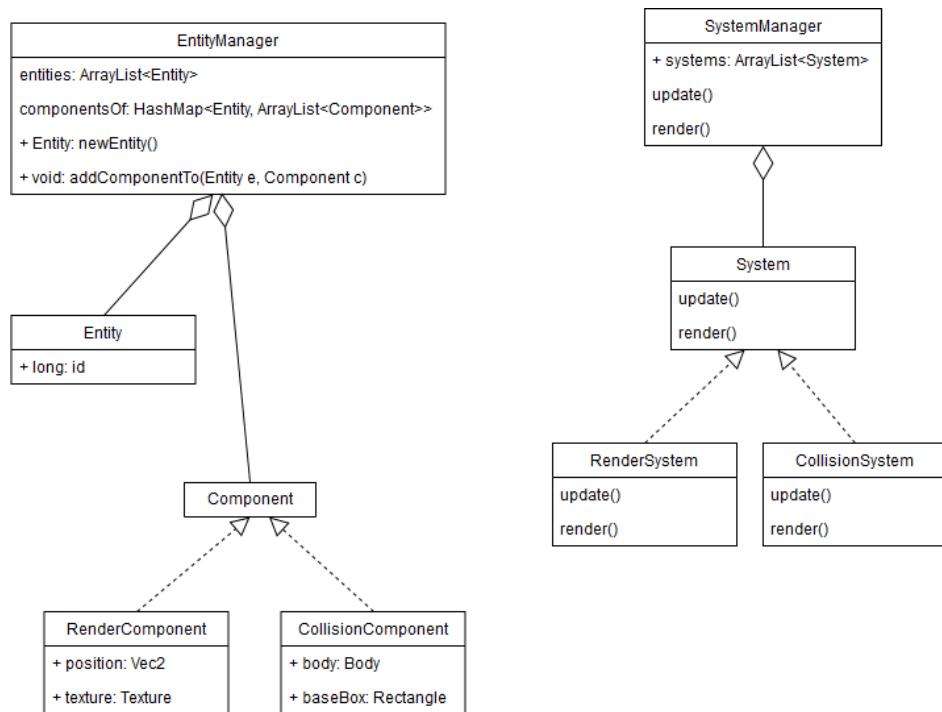


Figura 27 – Diagrama de classes do sistema entity based

3.9 Sistema de colisões

O sistema de colisões é responsável por delimitar o espaço físico que cada objeto deve ocupar no ambiente. Para formas geométricas mais básicas como retângulos, círculos e triângulos existem fórmulas simples e otimizadas para detectar essas colisões. Para polígonos mais complexos existem diversos algoritmos que usufruem de métodos avançados para detectar e resolver essas colisões.

Um problema de colisão é dividido em duas etapas: detecção e resolução. A etapa de detecção consiste em detectar quais objetos colidiram com quais outros objetos em um tempo t . Esse processo pode ocorrer de duas formas, por antecipação ou no tempo de colisão. A segunda etapa consiste em resolver essas colisões usando uma série de critérios como, por exemplo: gravidade, massa, velocidade, impulso, torque entre outros.

3.9.1 Colisões do tipo AABB vs AABB

Colisões do tipo Axis Aligned Bounding Box (AABB) são as mais simples de resolver. Um retângulo do tipo AABB possui o seu eixo x paralelo ao eixo x do plano cartesiano, e o seu eixo y paralelo ao eixo y do plano cartesiano, ou seja, sem qualquer rotação. Para detectar se dois retângulos de eixo alinhado colidiram basta checar se houve sobreposição em qualquer um dos eixos x e y . O algoritmo responsável pela checagem de intersecção é dado na Listagem 3.22.

```
1  public boolean intersects(Rectangle r) {
2      float tw = this.width;
3      float th = this.height;
4      float rw = r.width;
5      float rh = r.height;
6
7      if (rw <= 0 || rh <= 0 || tw <= 0 || th <= 0)
8          return false;
9
10     float tx = this.x;
11     float ty = this.y;
12     float rx = r.x;
13     float ry = r.y;
14     rw += rx;
15     rh += ry;
16     tw += tx;
17     th += ty;
18
19     return ((rw < rx || rw > tx) &&
20             (rh < ry || rh > ty) &&
21             (tw < tx || tw > rx) &&
22             (th < ty || th > ry));
23 }
```

Listagem 3.22 – Colisão AABB vs AABB

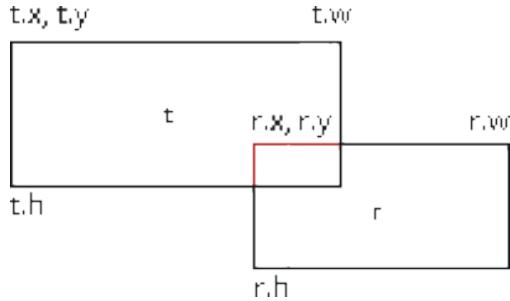


Figura 28 – Colisão AABB vs AABB

Para o exemplo da Figura 28 a intersecção ocorreu onde $rw > tx$, $rh > ty$, $tw > rx$ e $th > ry$. Ou seja, o retângulo r está abaixo e a direita do retângulo t.

3.9.2 jBox2D

A biblioteca jBox2D é um porto da biblioteca Box2D, originalmente implementada em C++. Trata-se de uma ferramente rica em recursos para manipulação de corpos rígidos 2D. Embora o sistema de medidas implementado pelo jBox2D esteja no padrão internacional (metros) isso não o torna inutilizável para um ambiente 2D com sistema de medida em pixels. Entretanto, faz-se necessário criar um fator de escala para converter entre um sistema e outro. Para o sistema de colisões funcionar ele precisa, assim como a própria engine, de um mundo e um timestep, neste caso, fixo. Esse sistema será então atualizado juntamente com o método `update()` da engine.

3.10 Sistema de coordenadas

O sistema de coordenadas comprehende todo o processo de manipulação das coordenadas de um game object para a tela final do usuário. Esse processamento tem início em espaço local e termina na sua conversão final para *Normalized Devide Coordinates* (NDC) que por fim é rasterizado pelo OpenGL na tela. Note que o OpenGL espera todas as coordenadas finais no formato NDC, que varia entre $[-1, 1]$. São cinco sistemas de coordenadas:

1. Espaço local
2. Espaço de mundo
3. Espaço de visão (ou espaço de câmera)
4. Espaço de recorte
5. Espaço da tela

Uma vez processadas elas são passadas para o *shader rasterization* e enfim convertidas para pixels 2D na tela, como ilustrado pela Figura 29.

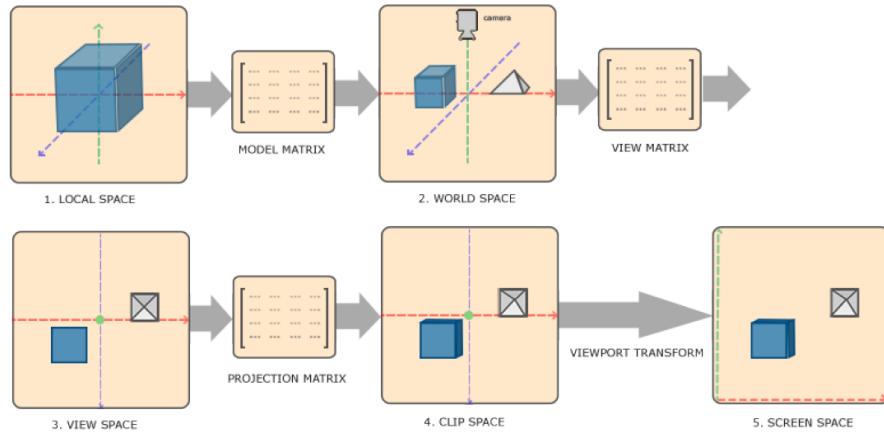


Figura 29 – Processo de transformações de coordenadas até o espaço final da tela.
Fonte: ([VRIES, 2015](#))

3.10.1 Espaço local

As coordenadas em espaço local referem-se ao próprio espaço do objeto. Representam as coordenadas do objeto relativo à sua própria origem. Geralmente são normalizadas no formato NDC com origem no ponto (0, 0).

3.10.2 Espaço de mundo

As coordenadas em espaço de mundo são obtidas através da multiplicação do espaço local pela matriz *model*. O resultado representa a posição do objeto no mundo relativo à origem do mundo. O processo de multiplicação do espaço local pela matriz *model* escala, rotaciona e translada o objeto para o espaço de mundo através do processo chamado transformação afim ([OPENCV, 2018](#)). Esses processos são ilustrados pela Figura 30.

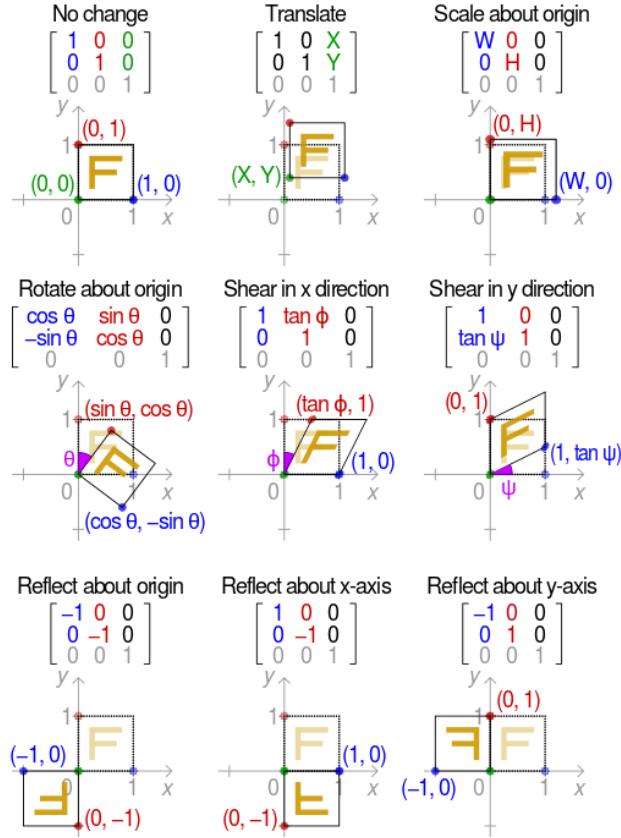


Figura 30 – Exemplos de transformação afim. Fonte: ([WIKIPEDIA](#), 2018)

Se todos os objetos fossem inseridos no mundo somente usando as coordenadas locais muito provavelmente estariam sobrepostos na origem. O processo de conversão para o espaço de mundo pode ser exemplificado por uma pilha de caixas onde se deseja inserir outra. Para empilhá-la é necessário escalá-la para o mesmo tamanho das outras, rotacioná-la para alinhar e, por fim, transladá-la para a posição na pilha.

3.10.3 Espaço de visão

O espaço de visão, também referenciado como espaço da câmera, é obtido a partir da multiplicação do espaço de mundo pela *view matrix* e representa o ponto de vista da câmera sobre o mundo.

3.10.4 Espaço de recorte

Espaço de recorte é a zona visível na tela de forma que todos os pontos fora desse espaço são descartados. Para que o usuário não tenha de especificar todas coordenadas em um intervalo NDC desde o início do processo trabalha-se com dois sistemas de projeção responsáveis por converter do espaço de visão de volta para um intervalo NDC. São os dois sistemas: projeção ortográfica e projeção de perspectiva. Note que, se um triângulo

possui apenas um parte visível e terá uma parte recortada o OpenGL divide esse triângulo em primitivas menores e então descarta aquelas que estão fora da tela.

3.10.5 Projeção

Existem duas formas de projeção, ambas baseadas no conceito de frustum, porção de um sólido contido entre duas bases paralelas que o cortam ([UNITY, 2018](#)). Os dois tipos de projeção são: ortográfica e perspectiva. A primeira é composta de um frustum retangular e a segunda de um frustum com formato similar a uma pirâmide.

3.10.5.1 Projeção ortográfica

Para especificar uma matriz de projeção ortográfica é necessário fornecer a largura, altura e comprimento do frustum. Dessa forma, todas as coordenadas que estiverem entre o plano mais próximo e o plano mais distante serão preservadas e desenhadas na tela. Todo o resto será recortado fora, como ilustrado pela Figura 31.

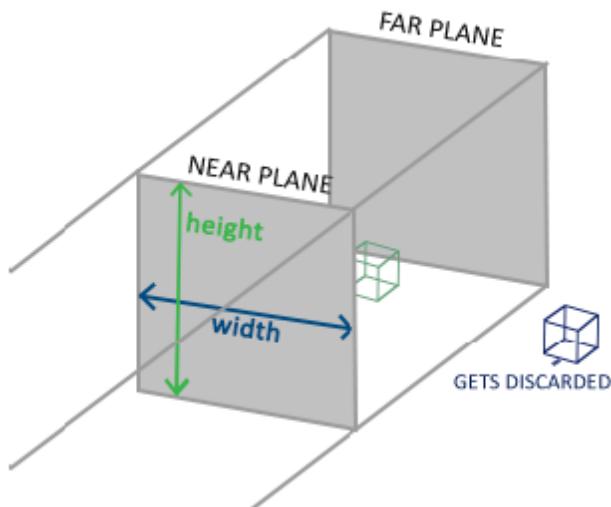


Figura 31 – Frustum da projeção ortográfica. Fonte: ([VRIES, 2015](#))

A projeção ortográfica mapeia linearmente coordenadas 3D diretamente para o plano 2D da tela, sem levar em consideração a perspectiva. Isso produz um efeito não realístico para ambientes 3D pois passa a impressão de objetos prensados na tela, sem profundidade.

Para especificar a matriz ortográfica será utilizada a função do GLM definida na Listagem 3.23. Os dois primeiros parâmetros são as coordenadas esquerda e direita do frustum, o terceiro e quarto parâmetro a parte de baixo e topo do frustum. Com esses quatro parâmetros definiu-se o tamanho do plano mais próximo e do plano mais distante. O quinto e sexto parâmetro indicam a distância entre os dois planos.

```

1 Mat4 projection = new Mat4();
2 projection = projection.ortho(float left, float right, float bottom, float top,
    float zNear, float zFar);

```

Listagem 3.23 – Função Ortho do GLM

3.10.5.2 Projeção de perspectiva

A projeção de perspectiva simula a percepção humana de profundidade. Objetos mais distantes se tornam menores e a esse efeito é dado o nome de perspectiva. Esse fenômeno acontece por que o ser humano enxerga o mundo a partir de dois olhos, com ângulos de visão distintos.

O frustum utilizado na projeção de perspectiva possui um formato similar a de uma pirâmide justamente para auxiliar na reprodução desse fenômeno. Ao invés de realizar uma transformação linear das coordenadas 3D para a tela, é feito um mapeamento que leva em consideração a distância dos objetos. O frustum pode ser visualizado na Figura 32.

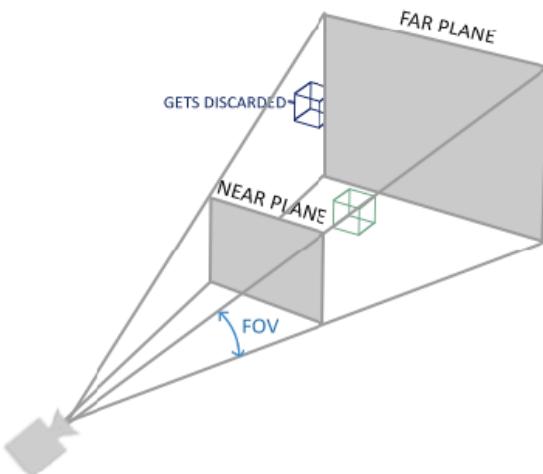


Figura 32 – Frustum da projeção de perspectiva. Fonte: ([VRIES, 2015](#))

A matriz de projeção pode ser criada utilizando a função do GLM dada na Listagem 3.24 onde o primeiro parâmetro recebe o field of view (FOV) em radianos, o segundo recebe a proporção de tela e no terceiro e quarto parâmetro recebe o plano mais próximo e distante, respectivamente.

```

1 Mat4 perspective = new Mat4();
2 perspective = perspective.perspective(toRadians(45), width/height, near, far);

```

Listagem 3.24 – Função Perspective do GLM

3.10.6 jBox2D para espaço de mundo

Para trabalhar com os dois sistemas de medida (pixels e metros) basta selecionar um fator de escalonamento. Supondo um fator de 100, basta multiplicar por 100 para converter de metros para pixels e dividir por 100 para converter de pixels para metros.

3.11 Câmera

A classe câmera dada na Listagem 3.25 é responsável por manter, calcular e enviar o valor da matriz de visão para o uniform no shader. Ela é necessária pois, a partir de uma única classe fica fácil manipular em quem está o foco da câmera, aplicar efeitos como tremulação entre outras funções. É importante frisar que ela deve atualizar a variável uniform dentro do método `variableUpdate` para que seja utilizado o valor baseado na interpolação, caso contrário ela dará impressão de objetos travando como discutido na Subseção Interpolação Linear 3.3.1.

```
1  public class Camera {
2      private float x=0,y=0,z=0;
3      private Entity focus;
4      private Mat4 camera;
5      private Mat4 transform;
6      private EntityManager em;
7
8      public Camera(EntityManager em) {
9          this.em = em;
10         camera = new Mat4();
11         transform = new Mat4();
12         camera = camera.identity();
13     }
14
15     public void setFocusOn(Entity entity) {
16         focus = entity;
17     }
18
19     public void moveDirectTo(float x, float y) {
20         this.x = x;
21         this.y = y;
22         camera = camera.identity();
23         camera.translate(x, y,0);
24     }
25
26     public void move(float x, float y) {
27         this.x += x;
28         this.y += y;
29         camera = camera.identity();
30         camera.translate(this.x, this.y,0);
31     }
32
33     public void update(float deltaTime) {}
34
35     public void variableUpdate(float alpha) {
```

```

36     RenderComponent rc = ((RenderComponent)(em.getFirstComponent(focus,
37         RenderComponent.class)));
38     Vec2 position = rc.getRenderPosition();
39     Vec2 size = rc.getSize();
40
41     if(focus != null)
42         moveDirectTo(-position.x + 1280/2 - size.x/2, -position.y +720/2 -
43             size.y/2);
44
45     transform = transform.identity();
46     transform.translate(this.x,this.y,0);
47
48     ResourceManager.getSelf().getShader("texture").use();
49     ResourceManager.getSelf().getShader("texture").setMat4("camera",
50             transform);
51
52     ResourceManager.getSelf().getShader("batchRenderer").use();
53     ResourceManager.getSelf().getShader("batchRenderer").setMat4("camera",
54             transform);
55 }
56
57     public float getX() {
58         return x*-1;
59     }
60
61     public float getY() {
62         return y*-1;
63     }
64 }
```

Listagem 3.25 – Classe Camera

3.12 Sistema de input

O sistema de input é responsável por mapear botões de uma interface física como teclado, mouse e controle de videogame em ações dentro do jogo. Para não lidar diretamente com a interface de hardware a biblioteca GLFW apresenta uma abstração que realiza essa comunicação.

O conceito para implementar o mapeamento de input consiste em definir um mapa de ações que podem então ser associadas ao input do usuário. Dessa forma, se o usuário trocar o equipamento, suponha-se, de um teclado para um controle de videogame tudo que o sistema precisa fazer é associar os novos botões às ações. Por exemplo, a ação de pular no teclado é mapeada para a tecla espaço. Quando o usuário trocar para o controle de videogame o sistema mapeia a ação pular para o botão "X" do controle, por exemplo. Dessa forma abstrai-se a ação que o game object vai realizar do hardware que está sendo

utilizado.

Entretanto, haverá casos que essa abstração não será suficiente e as ações serão mapeadas diretamente ao hardware em uso. Essa situação irá ocorrer, por exemplo, na interação com a interface onde o cursor do mouse não é diretamente mapeável para o joystick. Seria necessário implementar um pseudo cursor usando os botões analógicos do controle ou mapear o movimento de tecla em tecla ao invés de usar o conceito de ponteiro.

3.12.1 GLFW Keyboard

Para o controle do teclado é necessário primeiro definir uma classe que irá implementar o callback do GLFW. Nela será criado um vetor de booleanos que indica quais teclas estão pressionadas em determinado momento.

```
1 public class KeyboardControl extends GLFWKeyCallback implements Control {
2     private boolean keys[] = new boolean[512];
3
4     @Override
5     public void invoke(long window, int key, int scanCode, int action, int mods)
6     {
7         keys[key] = action != GLFW_RELEASE;
8     }
9
10    @Override
11    public boolean isKeyPressed(int key) {
12        return keys[key];
13    }
14
15    @Override
16    public boolean isKeyReleased(int key) {
17        return !keys[key];
18    }
}
```

Listagem 3.26 – Keyboard input

A partir da classe KeyBoardControl definida na Listagem 3.26 é possível implementar então um HashMap que mapeia as teclas para ações que serão então utilizadas pelos game objects.

3.12.2 GLFW Mouse

De maneira similar o controle do mouse contém um vetor de booleanos indicando quais teclas estão sendo pressionadas e também um Vetor de duas coordenadas indicando sua posição relativa a janela. Note-se que as coordenadas são calculadas considerando o canto superior esquerdo da janela como origem. O código completo da classe é dado na Listagem 3.27.

```
1 public class MouseControl extends GLFWCursorPosCallback implements Control{
2     private int previousState = -1;
```

```

3     private Vec2 cursorPos = new Vec2(0,0);
4
5     @Override
6     public boolean isKeyReleased(int key) {
7         int state = glfwGetMouseButton(Engine.getSelf().getWindow().getId(), key
8             );
9
10        if (state == GLFW_RELEASE)
11            return true;
12        else
13            return false;
14    }
15
16    @Override
17    public boolean isKeyPressed(int key) {
18        int state = glfwGetMouseButton(Engine.getSelf().getWindow().getId(), key
19             );
20
21        if (state == GLFW_RELEASE && previousState== GLFW_PRESS) {
22            previousState=state;
23            return true;
24        }else {
25            previousState=state;
26            return false;
27        }
28    }
29
30    @Override
31    public void invoke(long window, double xpos, double ypos) {
32        cursorPos.x = (float) xpos;
33        cursorPos.y = (float) ypos;
34    }
35
36    public Vec2 getCursorPos() {
37        return cursorPos;
38    }
39 }
```

Listagem 3.27 – Mouse input

3.13 Gerência de recursos

O sistema de gerência de recursos é responsável por ler e salvar todos os arquivos da engine. Em suma, esse sistema centraliza todas as operações de I/O num único lugar de forma que seja evitado a ocorrência de dois arquivos idênticos em memória. Isso é possível através do mapeamento dos recursos em um único lugar. A implementação completa da classe responsável pela gerência de recursos pode ser encontrada no Apêndice D.

3.13.1 Chunk

Um chunk (em português, pedaço) contém todas as entidades que intersectam sua *bounding box* e informações pertinentes do terreno como: textura, regiões de interação entre outros. Ele é serializado para leitura e escrita em disco. Sendo assim, um conjunto de n chunks sendo carregados no início do jogo representam o último estado do jogo. Esse estado contém todas as entidades, mapa e outros artefatos .

A geração da textura do mapa pode ser dinâmica, isto é, ser gerada a cada frame, ou estática, gerando-se uma única vez. A estática é utilizada para terrenos sem animação, enquanto a dinâmica para terrenos que envolvem animação como, por exemplo, as ondas que atingem a praia.

3.13.2 Sistema de Chunks

O sistema de chunks reparte toda a extensão do mapa em pequenos pedaços para uma maior facilidade de manuseio na hora de ler e salvar no disco. Um fator determinante para a escolha deste formato é a necessidade de poder gerar um mapa procedural e com carregamento dinâmico, ou seja, sem telas de carregamento. Isso é possível repartindo o mapa em pequenos pedaços que serão gerados, lidos e escritos sobre demanda.

É importante notar que este sistema é crítico e deve sermeticulosamente ajustado para as necessidades de cada projeto. Um arquivo muito grande pode gerar um atraso de leitura e escrita muito alto, enquanto um arquivo muito pequeno pode sobrecarregar o sistema com o *overhead* de muitas chamadas de função I/O num período curto de tempo. É necessário implementar este sistema utilizando técnicas de leitura e escrita assíncronas de forma que a engine não congele enquanto aguarda operações de I/O.

Para tanto, o gerenciador de chunks armazena uma matriz 3 por 3 de chunks. Sempre que a câmera intersectar um chunk da borda será carregado uma nova coluna ou linha dessa matriz contendo os novos chunks que, caso ausentes no disco são gerados e, caso presentes, lidos. Esse processo deve ocorrer de forma rápida tal que o usuário não note qualquer chunk sendo carregado. Se o usuário movimentar-se mais rápido do que o sistema é capaz de carregar os chunks ele irá perceber várias partes cinzas que subitamente serão preenchidas com textura quando sua leitura finalizar. Todo esse processo é ilustrado pela Figura 33.

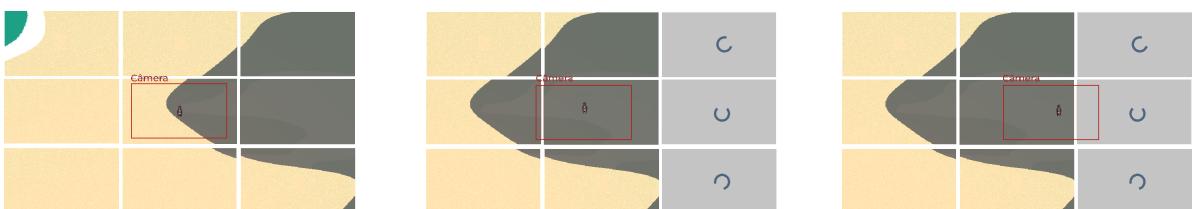


Figura 33 – Demonstração do sistema de chunks carregando a última coluna da matriz

O retângulo vermelho da Figura 33 representa a câmera. No momento em que esse retângulo intersecta qualquer um dos chunks na borda é iniciado o carregamento da linha e/ou coluna adjacentes a esse chunk. No caso da figura exemplo, quando a câmera intersecta o chunk da borda direita a primeira coluna da matriz é removida e armazenada em uma tabela Hash, enquanto a nova coluna direita da matriz é carregada. Se a câmera mover-se mais rápido do que o sistema é capaz de carregar os chunks o usuário irá presenciar partes cinzas na tela, que são as texturas ainda em carregamento.

3.13.3 Gerenciador de Chunks

O Chunk manager fica responsável por gerenciar quais chunks devem ser gerados, lidos ou salvos. Ele contém uma tabela hash com todos os chunks lidos até o momento, para evitar que tenham de ser lidos do disco caso já tenham sido em algum momento anterior. Esse gerenciador também é responsável por controlar o processo de leitura assíncrona, retornando o elemento somente quando sua leitura ou escrita foi completada.

3.14 Áudio

A implementação de áudio foi feita utilizando outra biblioteca multiplataforma da Khronos, o OpenAL, voltada especificamente para áudio. Através dela é possível utilizar diversas funções para manipulação de áudio 3D numa maneira eficiente e ágil.

3.14.1 Integrando o OpenAL

O openAL usa conceitos similares aos que são empregados no OpenGL. Cria-se e vincula-se buffers para transferir e receber dados através da API. Sempre existe um único ouvinte por *audio context*, que representa a posição de onde todas as fontes serão escutadas. Para definir uma fonte de áudio é necessário primeiro carregar o áudio do arquivo. O OpenAL por padrão usa buffers com formato Modulação por Código de Pulso(PCM) em 8 ou 16 bits, mono ou stereo.

Para ler o arquivo e prepará-lo para inserção no buffer é necessário carregá-lo usando a biblioteca auxiliar STBVorbis, que acompanha o LWJGL. Lê-se o arquivo usando `this.getClass().getResourceAsStream`, em sequência aloca-se dois buffers de inteiro numa pilha, aonde salva-se a quantidade de canais, no caso, se é mono ou stereo, e a taxa de amostragem. Por fim, com o shortBuffer usa-se a função `stb_vorbis_decode_memory` para decodificar o áudio do arquivo. O código para leitura do arquivo de áudio é dado pela Listagem 3.28.

```
1     String fileName = audioPath;
2     byte[] memory = null;
3
4     try {
```

```

5         int number0fBytes = this.getClass().getResourceAsStream("//" +
6             audioPath).available();
7         memory = new byte[number0fBytes];
8         this.getClass().getResourceAsStream("//" + audioPath).read(memory, 0,
9             number0fBytes);
10
11     } catch (IOException e) {
12         e.printStackTrace();
13     }
14
15     stackPush();
16     IntBuffer channelsBuffer = stackMallocInt(1);
17     stackPush();
18     IntBuffer sampleRateBuffer = stackMallocInt(1);
19
20     ShortBuffer rawAudioBuffer = stb_vorbis_decode_memory(BufferUtilities.
21         createByteBuffer(memory), channelsBuffer, sampleRateBuffer);
22
23     int channels = channelsBuffer.get();
24     int sampleRate = sampleRateBuffer.get();
25
26     stackPop();
27     stackPop();
28
29     int format = -1;
30     if(channels == 1) {
31         format = AL_FORMAT_MONO16;
32         System.out.println("mono");
33     } else if(channels == 2) {
34         format = AL_FORMAT_STEREO16;
35     }

```

Listagem 3.28 – Leitura do arquivo de áudio

A última etapa é gerar um openAL buffer e transferir os dados alocados em `rawAudioBuffer` no formato e taxa de amostragem escolhidos.

```

1     bufferPointer = alGenBuffers();
2     alBufferData(bufferPointer, AL_FORMAT_MONO16, rawAudioBuffer, sampleRate);

```

Listagem 3.29 – OpenAL Buffers

No gameloop ou na classe player define-se a posição do ouvinte com a função `alListener3f` dada na Listagem 3.30.

```

1 PositionComponent playerPosComp = em.getFirstComponent(player, PositionComponent
2 .class);
2 alListener3f(AL_POSITION, playerPosComp.getPosition().x, playerPosComp.
    getPosition().y,0);

```

Listagem 3.30 – OpenAL Listener

4 Inteligência artifical

O sub sistema responsável pela IA da engine é uma das partes mais importantes, pois sem ela o jogo seria um conjunto de imagens estáticas distribuídas à esmo pela tela. Em essência, IA é aquilo que da vida a todos os personagens do jogo, tornando-o interativo e dinâmico. A IA foca nas ações que um agente deve tomar, baseado nas condições atuais. A IA deve observar o ambiente, tomar decisões e então agir. Esse ciclo é descrito na literatura como observar, pensar e agir ([SIZER, 2018](#)).

A implementação da IA de um game object pode ser feita usando diversas técnicas, desde clausulas if ou else até redes neurais. Entretanto, as mais comuns na indústria de jogos não envolvem técnicas com uso de treinamento pois o objetivo aqui não é uma IA ótima, mas sim uma capaz de desafiar e entreter o jogador. Outra restrição é que, para treinar a rede neural seria necessário alimentá-la com dados de muitos jogadores, algo impraticável no estágio de produção pois o jogo nem sequer está em numa etapa jogável.

Assim como o processo de renderização não pode demorar mais do que alguns milissegundos, o mesmo se aplica para a tomada de decisões da IA. Portanto, é fundamental que esta seja a mais rápida possível para que o jogador não experiente travamentos na jogabilidade.

4.1 Utility em IA

Em economia, *utility* é uma medida da satisfação relativa de, ou desejo de, consumo de vários bens e serviços. Dada essa medida, alguém poderá expressar o acréscimo ou decréscimo desta *utility*, e assim explicar o comportamento econômico em termos de tentativas para incrementar tal *utility*.

Incorporando esse conceito em IA pode-se delimitá-lo como o desejo de uma entidade em realizar determinada ação dado certo valor do utility, calculado com base no contexto em questão. Esse cálculo é realizado através de funções que recebem como parâmetro o contexto e dele extraem os valores que quantificam o ambiente. Por fim, o cálculo é realizado e indica o utility, ou o quanto desejável, aquela ação é ([MARK; DILL, 2018](#)).

4.1.1 Processo de funcionamento

Para cada ação que o agente pode realizar é atribuída uma curva de pontuação. A função que gera essa curva recebe como parâmetros variáveis do ambiente que afetam essa ação. Por exemplo, um agente capaz de realizar três ações, atirar, abrigar-se e perseguir terá atribuída a cada uma dessas ações uma função de pontuação. Ao final, compara-se o

valor calculado (utility) das funções e o maior deles representa a ação dentre as três que será executada.

Para a função de pontuação que calcula o score da ação atirar tem-se como parâmetro a distância do inimigo. Para a função de abrigar-se tem-se a distância do inimigo e a vida do agente como parâmetros. Supondo-se que cada ação seja calculada pelas funções:

- Atirar: $return 0.5$
- Abrigar-se: $return (vida - 1)^2$
- Perseguir: $return distancia^3$

A partir dessas funções é possível calcular qual utility será o maior para cada conjunto de parâmetros. Note ainda que todos os parâmetros estão normalizados no intervalo $[0, 1]$. Para normalizar a distância é necessário escolher um valor máximo e assim todo valor que excedê-lo será fixado em 1. O gráfico para cada função é dado pela Figura 34.

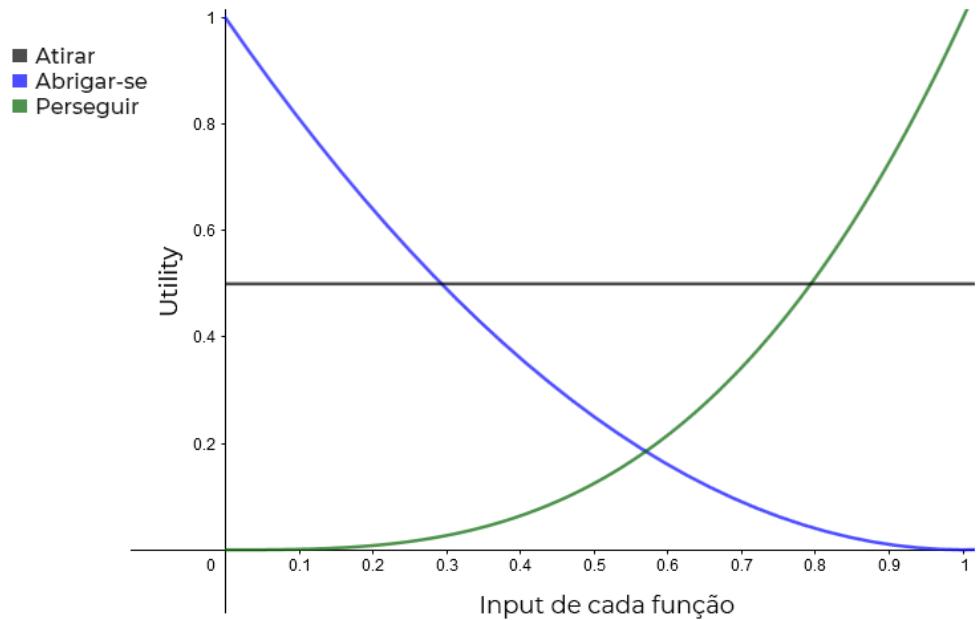


Figura 34 – Gráfico das funções utility

Sobrepondo as três funções pode-se ter uma melhor noção de qual comportamento irá se sobressair em cada situação para cada valor do parâmetro. Entretanto, vale lembrar que os valores de vida e distância são independentes e portanto as funções não necessariamente estarão no mesmo ponto do eixo X ao mesmo tempo. A tabela 2 demonstra

três situações em que, para cada conjunto de parâmetros tem-se uma ação que é decidida usando a maior utility. O asterisco indica a função cuja pontuação foi maior e será, portanto, a ação executada.

vida	distância	Atirar	Abrigar-se	Perseguir
0	0	0.5*	0	0
0.2	0	0.5	0.64*	0
0.2	0.5	0.5	0.64*	0.125
0.2	0.9	0.5	0.64	0.729*
0.3	0	0.5*	0.49	0

Tabela 2 – Tabela de valores exemplo para as funções de pontuação

Por fim, o código de implementação é composto de três classes: Action, Consideration e ConsiderationTree. Um game object com IA recebe um componente com a árvore de considerações (ComponentTree). A árvore de considerações contem as possíveis ações que esse agente pode tomar. Cada consideração implementa um método evaluate() que recebe como parâmetro o contexto e dele extrai os valores de que precisa para realizar seus cálculos e retornar a *utility*. A classe Action contém o nome daquela *utility* para que o objeto possa se guiar e executar a ação associada. Ela contém também o alvo daquela ação, caso haja algum. Esse fluxo é demonstrado no diagrama de classes da Figura 35.

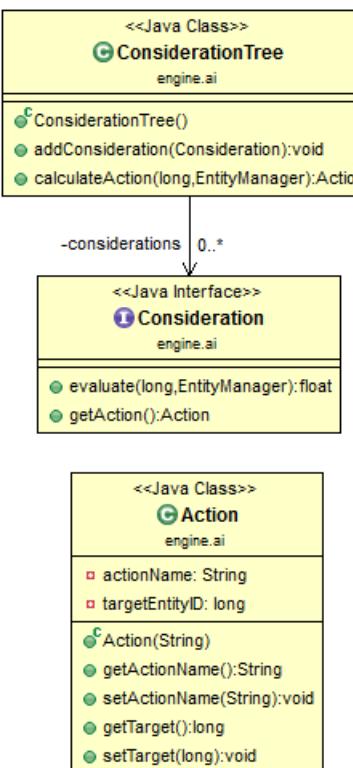


Figura 35 – Diagrama da Árvore de considerações para as funções de pontuação

4.2 Pathfinding

Existem muitos algoritmos de *pathfinding*(em português, busca pelo caminho) como, por exemplo, A*, Dijkstra's, LPA* entre outros . Alguns se adéquam muito bem para sistemas estáticos onde tem-se total conhecimento e controle do ambiente, enquanto outros são melhor adaptados para ambientes imprevisíveis e sob constante mudança.

Além dos vários algoritmos disponíveis e suas respectivas variações existem também diversas formas de representar o espaço de busca. Os algoritmos de *pathfinding* trabalham em cima do conceito de grafos. Portanto, independentemente do jogo ser em 3D ou 2D, no final deverá haver uma representação desse mundo na forma de um grafo. Para um jogo 2D baseado em grids isso é facilmente alcançável aproveitando cada célula desse grid e transformando-a em um nodo do grafo. Para cada dois nodos adjacentes onde ambos são transponíveis há uma aresta que os liga. Se o mundo é construído sem utilizar diretamente o conceito de grid ainda é possível representá-lo em um grid com células de tamanho x e, caso uma célula intersecte a *bounding box* de um objeto ela torna-se intransponível.

Existe ainda a possibilidade de utilizar uma *octree* onde cada célula desse grid possui tamanho potência de um valor estático, ao invés de um tamanho único e estático. Há ainda conceitos que usam polígonos convexos para gerar uma malha que particiona o mapa em pedaços transponíveis. De maneira geral, existem diversas técnicas e cada projeto deve optar por aquela que melhor lhe servir.

No sistema implementado na engine narval o processo de *pathfinding* em ambiente 2D consiste em primeiro mapear a seção visível do mapa e mais um offset para uma matriz. Essa seção é geralmente o retângulo que constitui a visão do personagem. Para cada entidade que intersecta essa seção será feito um mapeamento da sua caixa de colisão para a matriz. Cada posição da matriz representa um retângulo de tamanho arbitrário e fixo. Note que, a não ser que todas as entidades do jogo possuam módulo zero com o tamanho do retângulo da matriz haverá um erro a se considerar. Por exemplo, se uma entidade com retângulo de 32x32 vai ser inserida em uma matriz cuja célula é um retângulo de 10x10, na hora do mapeamento essa entidade vai ocupar 4x4 células na matriz, tendo assim, para o algoritmo de pathfinding, um tamanho de 40x40. Isso resulta em um erro de 8 pixels para o tamanho real da entidade. Esse exemplo é demonstrado na Figura 36, onde a parte vermelha representa a base box e a parte azul o desperdício. Ambas as duas partes representam o retângulo de fato contido na matriz.

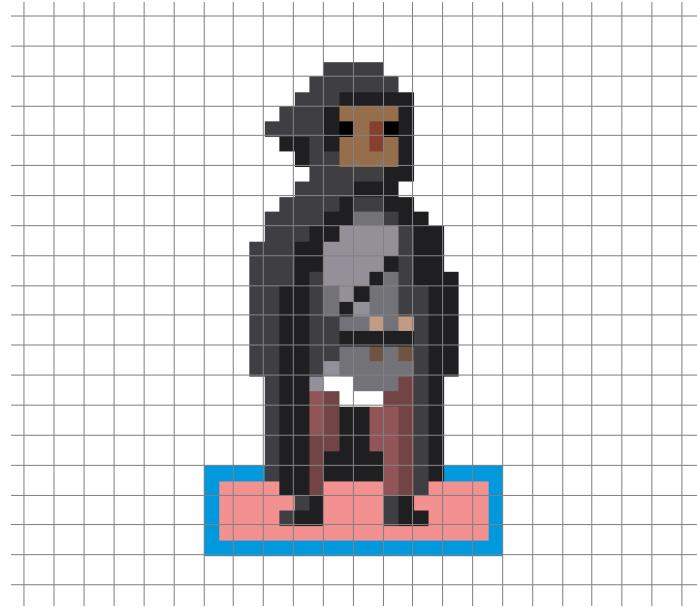


Figura 36 – Desperdício gerado ao mapear uma base box de 32x32 pixels numa matriz de retângulos 10x10 pixels.

Uma vez realizado o mapeamento tem-se uma matriz boolena representando quais pontos são transponíveis. A partir disto basta implementar um algoritmo de busca como, por exemplo, o A* para percorrer o grafo representado pela matriz e encontrar um caminho do ponto A para um ponto B. É importante levar em consideração que os objetos podem ocupar mais de um nodo devido ao seu tamanho e isso possui implicação direta na implementação do algoritmo de busca.

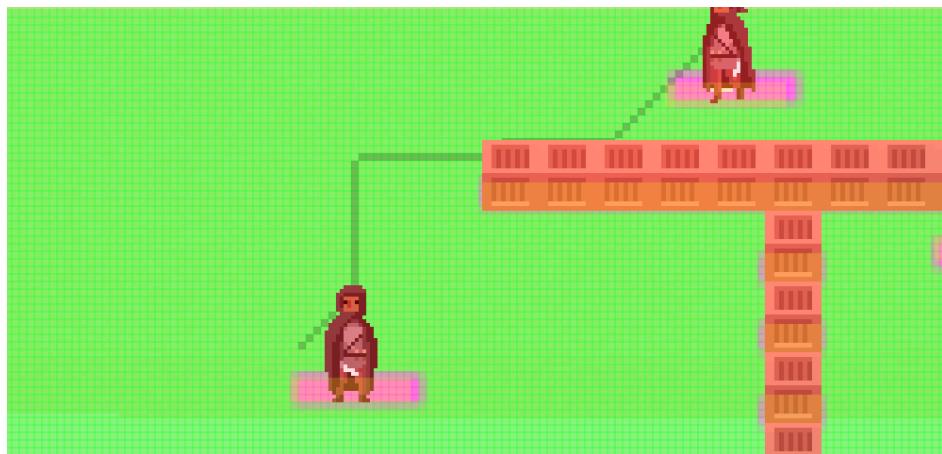


Figura 37 – Matriz boolena representada visualmente com os retângulos verdes sendo os transponíveis e os rosas intransponíveis.

4.2.1 Algoritmo A*

Para uma implementação utilizando A* é necessário considerar o ponto de início não como um único ponto, mas como um conjunto de pontos que representam a base box

da entidade. Essa aproximação permite que uma entidade cuja base box ocupa mais de um nodo possa se locomover pelo grid.

O algoritmo de busca A* funciona determinando, a cada iteração, qual caminho estender. Isso é feito através da equação $f(n) = g(n) + h(n)$, onde n é o próximo nodo no caminho, $g(n)$ o custo do nodo inicial até n e $h(s)$ uma função heurística que estima o custo mais baixo do caminho de n até o nodo objetivo.

A heurística utilizada considera a possibilidade de 8 direções no grid, atribuindo um maior custo para as diagonais. Como a heurística é apenas uma aproximação, define-se uma constante $\sqrt{2}$ para multiplicar a distância diagonal. Dessa forma, tem-se uma estimativa razoável sem o custo alto de realizar o cálculo completo da distância euclidiana $\sqrt{x^2 + y^2}$. Dessa forma, a função heurística $h(n)$ fica como se segue.

```

1   float a = Math.abs(start.pos.x - end.pos.x);
2   float b = Math.abs(start.pos.y - end.pos.y);
3
4   float min = Math.min(a,b);
5   float max = Math.max(a,b);
6
7   return ((M_SQRT2-1.0f)*min + max);

```

Listagem 4.1 – Heurística utilizada no A*

O código completo do algoritmo A* implementado pela engine é dado no apêndice B. O resultado obtido é também demonstrado na Figura 38.

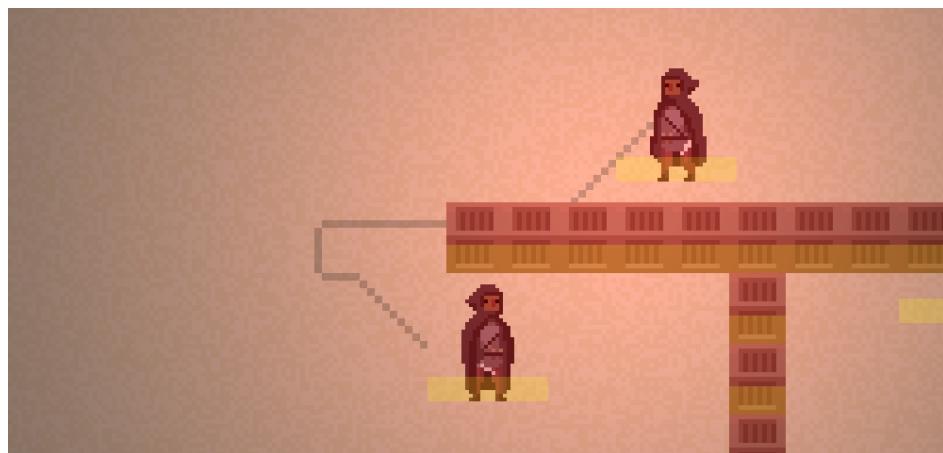


Figura 38 – Resultado final obtido com o A*

5 Procedural Content Generation

Procedural Content Generation (PCG) é o termo inglês, que significa geração procedural de conteúdo, cunhado para descrever processos de geração de dados aleatórios ou pseudo-aleatórios em jogos. Esses dados são utilizados para criar regras que definem desde terrenos até modelos 3D de maneira algorítmica ao invés de manual.

5.1 Ruídos aleatórios

O conceito mais básico na construção de elementos baseados em PCG envolve a utilização de ruídos como fonte de dados pseudo-aleatória. Existem diversos algoritmos para obtenção de ruído pseudo-aleatório como, por exemplo, perlin noise, simplex noise e gradient noise. Há ainda técnicas que se utilizam de maneiras não determinísticas e implementam outras fontes de dados.

5.1.1 Ruído de Perlin

Perlin noise(em português literal, ruído de perlin) é um tipo de ruído gradiente concebido por [Perlin \(1985\)](#) e depois melhorado por ele mesmo em 2002 ([PERLIN, 2002](#)). Esse algoritmo é comumente usado na indústria para geração de ruídos que são utilizados para criar texturas de água, fogo, terrenos, nuvens e muitos outros. O ruído pode ser representado em uma, duas, três ou até quatro dimensões.

O algoritmo recebe como entrada um ponto de n coordenadas. Esse ponto deve ser unitário, ou seja, para cada coordenada faz-se $mod\ 1$. Uma vez com as coordenadas unitárias em mãos elas são representadas em uma linha, quadrado ou cubo dependendo das dimensões do vetor. Por exemplo, para um vetor de duas dimensões com os valores $(0.75, 0.25)$ tem-se o ponto no plano ilustrado pela Figura 39.

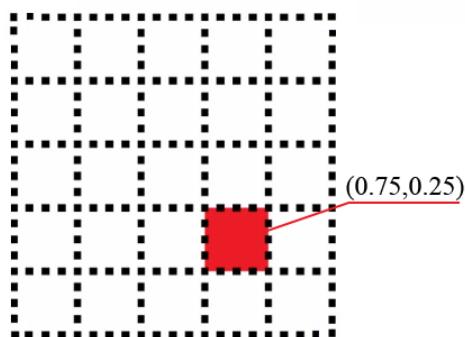


Figura 39 – Ponto $(0.75, 0.25)$ no plano. Adaptado de: ([FATAHO, 2018](#))

Para cada um dos 4 pontos unitários que definem a borda do plano (8 para o cubo em 3D), são gerados vetores gradiente de maneira pseudo aleatória. Uma vez com o vetor gradiente definido ele é fixo para a função que se está trabalhando. A Figura 40 ilustra os quatro vetores gradiente dos pontos unitários nas bordas.

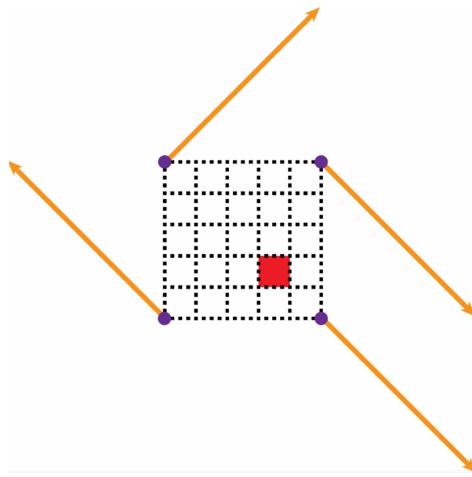


Figura 40 – Vetores gradiente definidos em amarelo no plano. Adaptado de: ([FATAHO, 2018](#))

Em posse do ponto e dos vetores gradientes é necessário agora calcular o vetor distância desse ponto até os quatro pontos na borda do plano como ilustrado pela Figura 41.

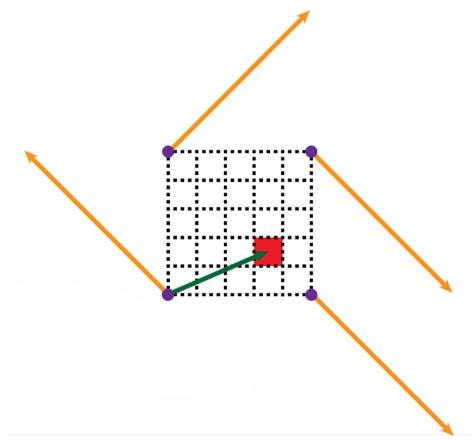


Figura 41 – Vetor distância do ponto $(0, 0)$ na borda até o ponto $(0.75, 0.25)$ em vermelho. Adaptado de: ([FATAHO, 2018](#))

Uma vez com o vetor distância, vetor gradiente e o ponto dado como parâmetro da função realiza-se o produto escalar entre ambos os vetores para obter o valor *influência*.

$$\begin{aligned}\vec{D} &= \vec{G}(-1, 1) \cdot \vec{D}(0.75, 0.25) \\ &= -1 * 0.75 + 1 * 0.25 \\ &= -0.75 + 0.25\end{aligned}$$

$= -0.5$

Esse processo é realizado para os quatro pares de vetor gradiente e vetor distância. Em sentido anti-horário, começando do ponto inferior esquerdo, os valores *influência* são $-0.5, -0.5, 0.5$ e 0 .

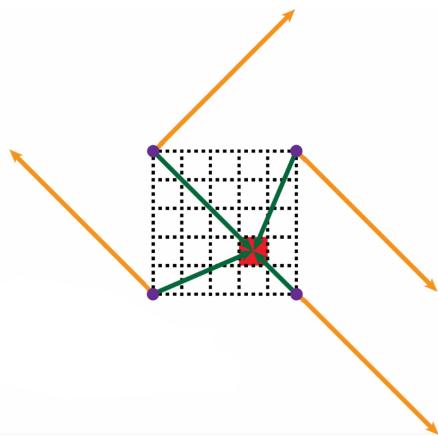


Figura 42 – Vetores distância do ponto $(0,0)$ em verde e vetores gradiente em amarelo.
Adaptado de: ([FATAHO, 2018](#))

O último passo consiste em interpolar linearmente os 4 valores *influência* $b1, b2, b3$ e $b4$ como demonstrado na Listagem 5.1. O resultado final retornado pelo algoritmo do perlin noise será dado pela variável `result`.

```
1 int b1, b2, b3, b4;
2 int x, y;
3
4 int x1 = linearInterpolation(b1, b2, x);
5 int x2 = linearInterpolation(b3, b4, y);
6
7 int result = linearInterpolation(x1, x2, y);
8
9 float linearInterpolation(float a, float b, float t){
10     return (1f - t) * a + t * b;
11 }
```

Listagem 5.1 – Interpolação dos valores *influência*

5.2 Geração de terrenos

A geração de terreno utilizada na engine consiste, num primeiro passo, em preencher um conjunto de matrizes com valores gerados por vários algoritmos de ruído na função `generateAndShapeNoise`, sendo o principal deles Perlin Noise. Cada um desses valores é então extraído da matriz e atribuído um significado baseado numa regra determinada pela função `generateTerrain`. Os ruídos são gerados utilizando a implementação `FastNoise`

(AUBURNS, 2018) incorporado na engine, onde o valor retornado pelas funções de ruído geralmente está no intervalo [-1,1].

A textura gerada para lapidar o terreno é chamada de *height map* e serve para indicar alturas no mapa. Com essa textura em mãos cada valor recebe um significado em termos de altura do terreno.

Ruídos podem ser gerados em qualquer frequência, que basicamente resultam no efeito de zoom da textura final. Esse efeito é consequência da fórmula do comprimento de onda $\lambda = \frac{\text{velocidade}}{\text{frequencia}}$. Pode-se notar que dobrar a frequência implica em diminuir o comprimento de onda pela metade do seu tamanho, conforme Figura 43.

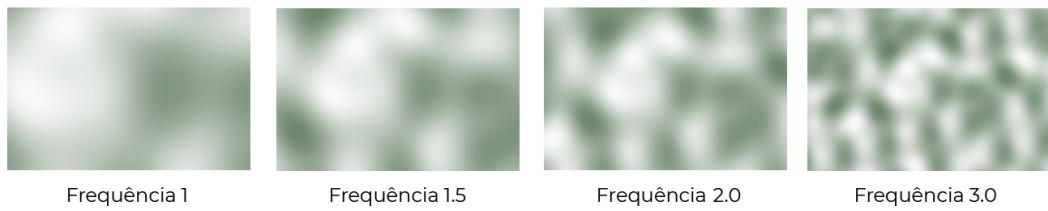


Figura 43 – Resultado da alteração na frequência. Adaptado de: (PATEL, 2018)

Outro atributo importante no processo de modelagem do ruído são os *octaves*. Eles servem para lapidar o ruído e adicionar mais detalhes na textura. Eles funcionam somando-se os ruídos gerados, mas cada um tendo uma frequência e peso diferente diferentes. Isso é demonstrado na Listagem 5.2 onde soma-se um perlin noise de frequência 0.0005 com peso 1 e outro de frequência 0.005 com peso 0.1, resultando em dois octaves.

```

1 public float noise(FastNoise fastNoise, float x, float y) {
2     float noise = 0;
3
4     fastNoise.SetFrequency(0.0005f);
5     noise += fastNoise.GetPerlin(x, y);
6
7     fastNoise.SetFrequency(0.005f);
8     noise += .1 * fastNoise.GetPerlin(x, y);
9
10    return noise;
11 }
```

Listagem 5.2 – Implementação do ruído que usa o perlin noise com dois octaves

Até então o ruído possui um aspecto fractal, com várias colinas e montanhas, mas nenhuma área de terreno plano. Para isso é necessário aplicar uma exponenciação ao ruído, achatando-o através da redução de valor. Não só é necessário um fator de exponenciação, mas também uma fórmula para gerar ruídos no formato de continentes e ilhas, onde quanto mais próximo da borda maior a probabilidade desse valor ser formatado como água. Essa fórmula consiste em primeiro calcular a distância de Manhattan d do valor (x,y) atual até o centro do mapa e multiplicar por 2.

$$d = 2 * \max(\text{abs}(x - \text{center}_x), \text{abs}(y - \text{center}_y))$$

A segunda etapa consiste em calcular a fórmula dada em seguida (PATEL, 2018) onde a é uma constante que puxa os todos os valores para cima, b puxa as bordas do mapa para baixo e c controla o quanto rápido essa queda é.

$$\text{noise} = \text{noise} + a - b * d^c$$

Dessa forma, o código completo da implementação que calcula e modela o ruído para uso é dado na Listagem 5.3.

```

1  public void generateAndShapeNoise() {
2      float d;
3      float a = 0.15f;
4      float b = 0.9f;
5      float c = 2f;
6
7      int coordX = ((this.x*chunkWidth)/NOISE_DIVISOR) ;
8      int coordY = ((this.y*chunkHeight)/NOISE_DIVISOR) ;
9
10     int constX = coordX;
11     int constY = coordY;
12
13     for(int y=0; y<textureHeight; y++) {
14         for(int x=0; x<textureWidth;x++) {
15             coordX = constX + x;
16             coordY = constY + y;
17
18             d = 2*Math.max(Math.abs((float)coordX/mapWidth - (float)(mapWidth/2)/
19                         mapWidth), Math.abs((float)coordY/mapHeight - (float)(mapHeight/2)/
20                         mapHeight));
21
22             perlinNoise[x][y] = noise(fastNoise,coordX/3f,coordY);
23             whiteNoise[x][y] = fastNoise.GetWhiteNoise(coordX, coordY);
24             fractalNoise[x][y] = fastNoise.GetPerlinFractal(coordX/4, coordY);
25
26             perlinNoise[x][y] = perlinNoise[x][y] + a - b*(float)Math.pow(d, c);
27         }
28     }
29 }
```

Listagem 5.3 – Implementação que gera e molda o ruído

A última etapa é atribuir um significado para cada um desses valores gerados. O código completo da implementação que atribui um significado a esses valores é dada a seguir. Em suma, valores abaixo de -0.255 viram água, valores abaixo de -0.1 viram areia e valores acima disso viram terra.

```

1  public void generateTerrain() {
2
3      for(int y=0; y<textureHeight; y++) {
4          for(int x=0; x<textureWidth;x++) {
5
6              if(perlinNoise[x][y]>-.1 ) {      //terra
7                  mapRGB[x][y] = Color.GRASS_GROUND;
```

```

8     if(fractalNoise[x][y]>0.2)
9         mapRGB[x][y] = Color.GRASS_GROUND_LIGHTER;
10    }
11
12    if(perlinNoise[x][y]<=-.1) { //areia
13        mapRGB[x][y] = (255<<24) | (244<<16) | (234<<8) | (187);
14        if(whiteNoise[x][y]>0)
15            mapRGB[x][y] = (255<<24) | (234<<16) | (224<<8) | (167);
16    }
17
18    if(perlinNoise[x][y]<=-.255) //preenche tudo com agua
19        mapRGB[x][y] = Color.TURKISH;
20    }
21 }
22 }
```

Listagem 5.4 – Implementação que gera a textura final do terreno

Os resultados obtidos possuem terrenos de similar aos exemplificados na Figura 44.



Figura 44 – Resultado final dos terrenos após modelagem do ruído e atribuição de significado. Adaptado de: ([PATEL, 2018](#))

6 Diagramas e panorama geral da engine Narval

Os diagramas dados a seguir descrevem toda a estrutura que compõe a fundação da engine Narval. Juntamente com o código ao longo deste trabalho os diagramas são a realização dos módulos abordados e descritos na introdução do capítulo 3.

Todas as classes até então desenvolvidas e abordadas se organizam nos pacotes descritos pela Figura 1 e são dadas formalmente a seguir. Cada pacote agrupa classes relacionadas à um tema, área ou módulo da engine.

6.1 Pacote de IA

O pacote `engine.ai` contém todas as classes relacionadas à implementação de inteligência artificial, conforme Figura 45. Essas classes implementam o A* e o conceito de utility utilizado para tomada de decisões do agente.

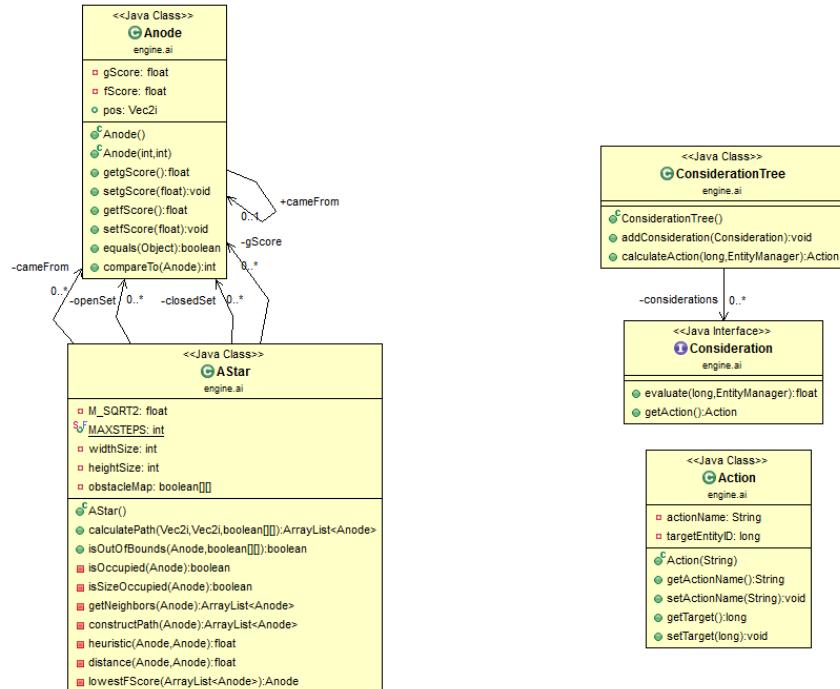


Figura 45 – Classes do pacote `engine.ai`

6.2 Pacote de Áudio

O pacote `engine.audio` contém todas as classes relacionadas à implementação de áudio. Essas classes, apresentadas na Figura 46, implementam a leitura e configuração do arquivo de áudio via OpenAL.

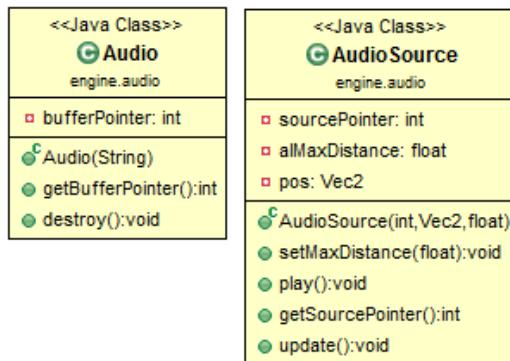


Figura 46 – Classes do pacote `engine.audio`

6.3 Pacote da Engine

O pacote `engine.engine` contém todas as classes relacionadas à implementação do núcleo da game engine. Essas classes, detalhadas na Figura 47, implementam a thread onde será rodada a primeira chamada aos métodos update e render, bem como a engine de física e também a janela OpenGL, onde deverá ser feita a renderização. É nesse pacote também que serão carregadas e definidas todas as configurações da engine.

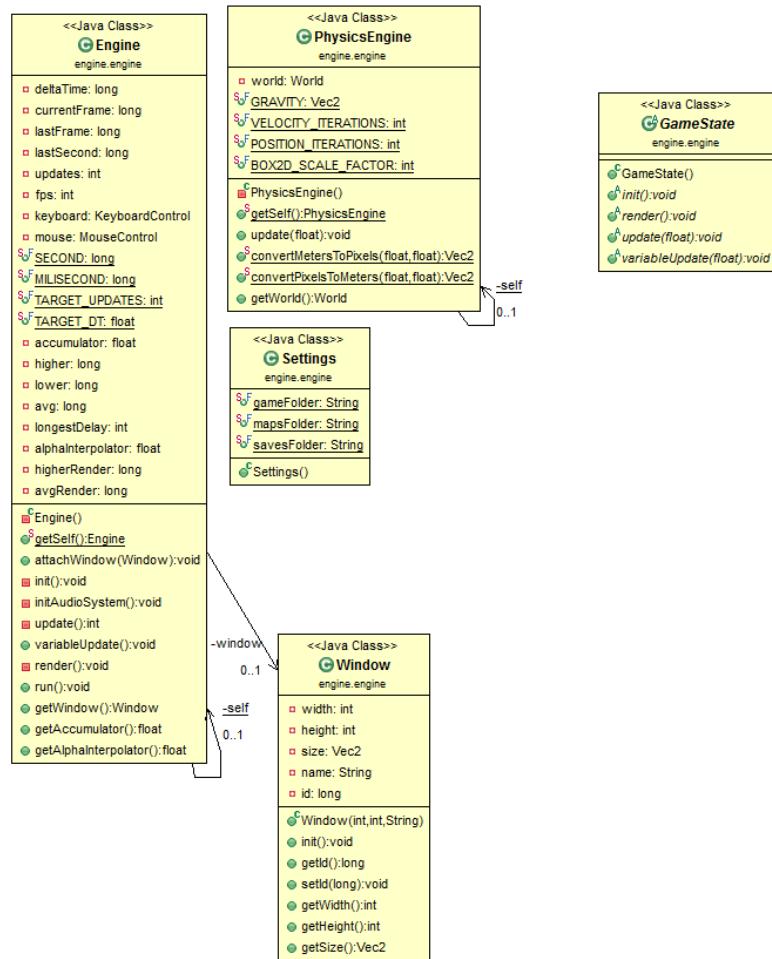


Figura 47 – Classes do pacote engine.engine

6.4 Pacote de componentes de entidades

O pacote `engine.entity.component` contém todas as classes relacionadas à implementação dos componentes das entidades e pode ser visto na Figura 48. Muito embora cada jogo deverá ter seus próprios componentes, esse pacote implementa os mais comuns que estarão presentes em, senão todos os jogos, a maioria significativa deles.

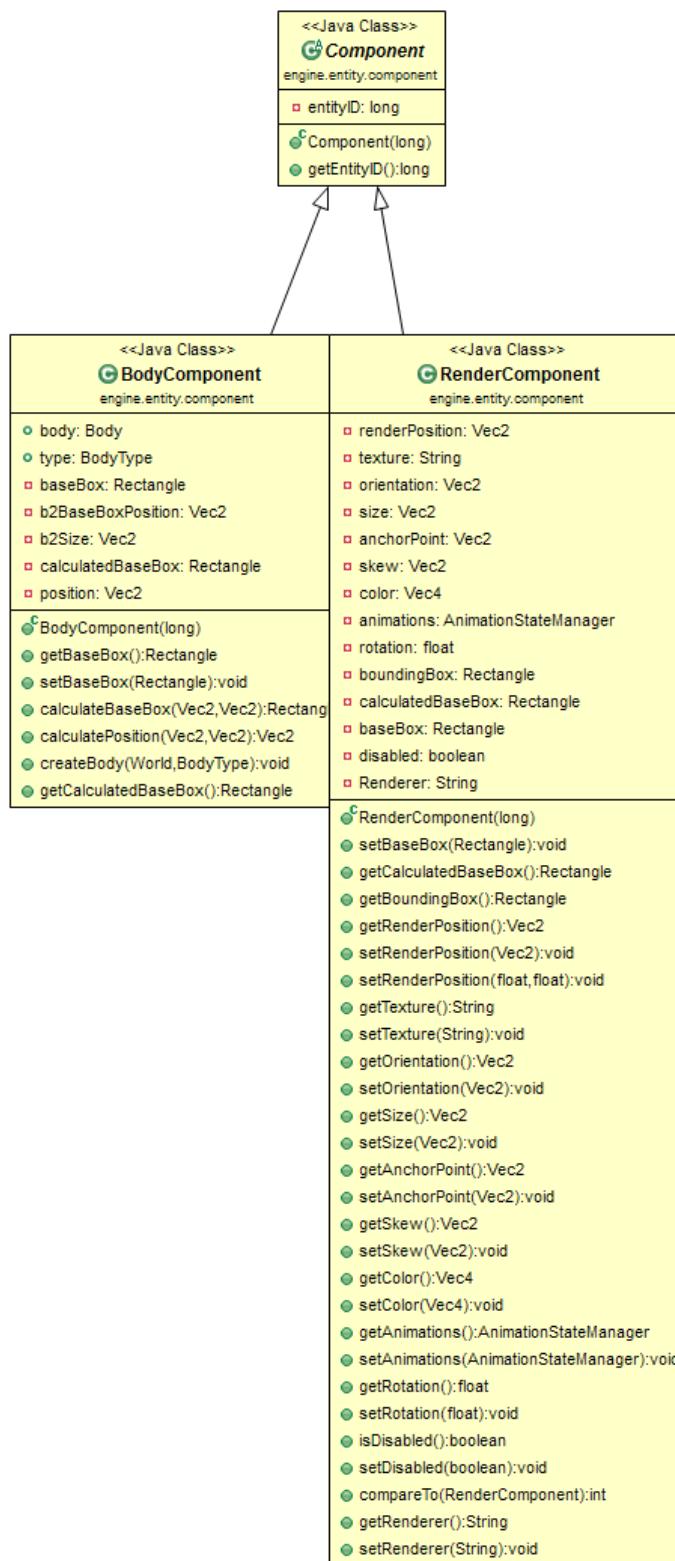


Figura 48 – Classes do pacote engine.entity.component

6.5 Pacote de entidades

O pacote `engine.entity` contém todas as classes relacionadas à implementação das entidades e os sistemas que as gerenciam. Também conta com a implementação de algumas especializações comuns em jogos. Sua implementação é ilustrada pelo diagrama da Figura 49.

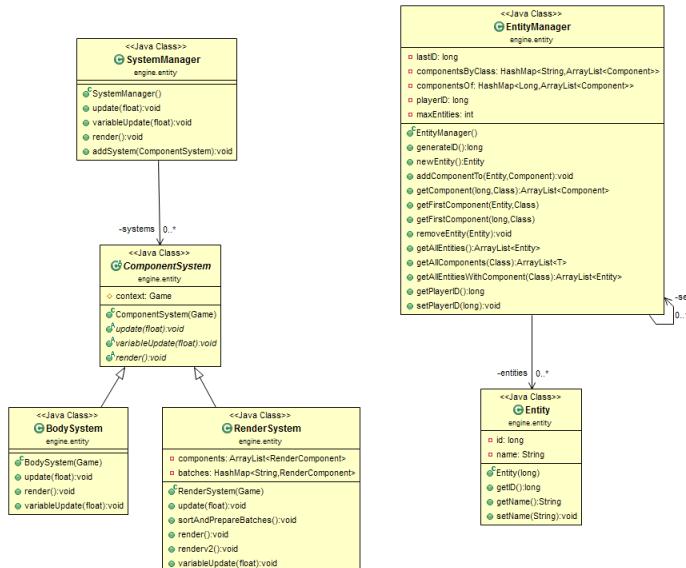


Figura 49 – Classes do pacote `engine.entity`

6.6 Pacote gráfico

O pacote `engine.graphic` contém todas as classes relacionadas à implementação dos componentes gráficos como animações, textura e os shaders utilizados pelo OpenGL. Sua implementação é ilustrada pelo diagrama da Figura 50.

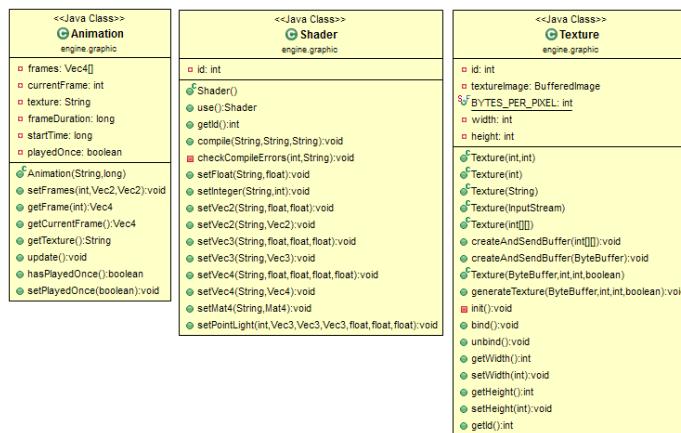


Figura 50 – Classes do pacote `engine.graphic`

6.7 Pacote de input

O pacote `engine.input` contém todas as classes relacionadas ao tratamento de input. Sua implementação é ilustrada pelo diagrama da Figura 51.

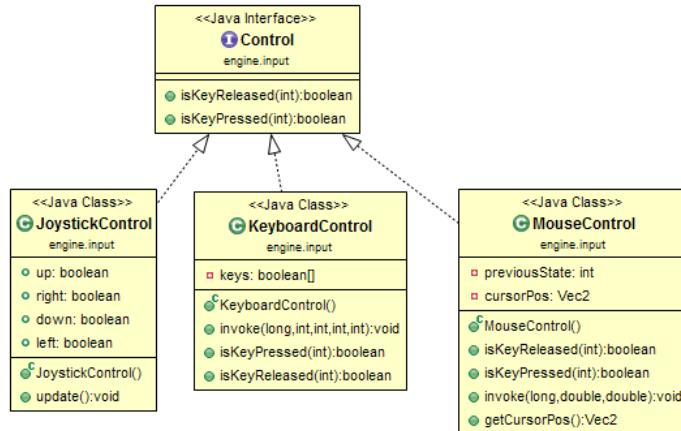


Figura 51 – Classes do pacote `engine.input`

6.8 Pacote lógico

O pacote `engine.logic` contém classes relacionadas ao aspecto lógico geral do sistema. Ele contém o sistema de chunks, timer, camera e o sistema de gerenciamento de animações. Sua implementação é ilustrada pelo diagrama da Figura 52.

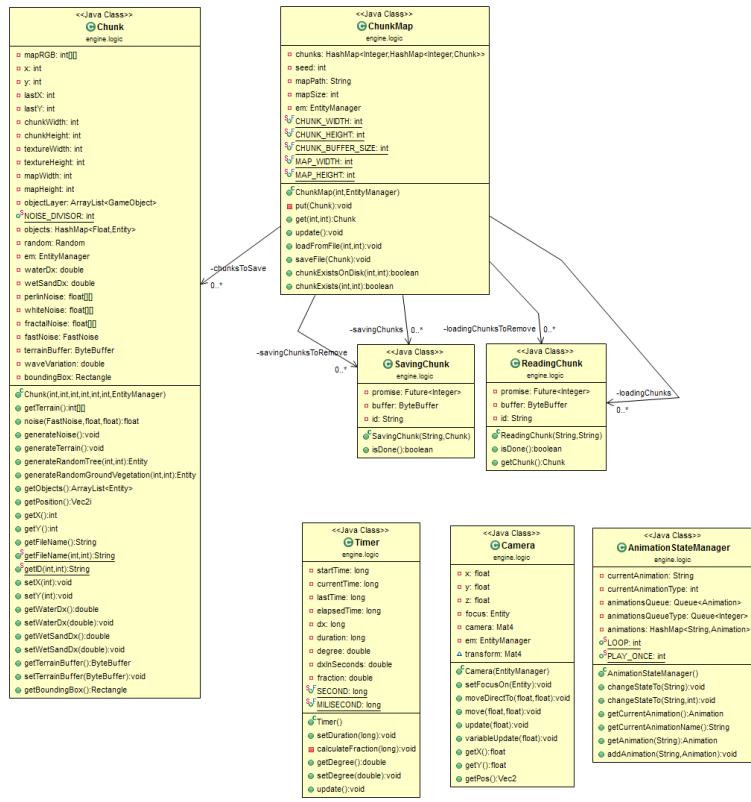


Figura 52 – Classes do pacote engine.logic

6.9 Pacote de ruído

O pacote `engine.noise` contém a classe responsável por gerar o ruído utilizado na construção dos mapas. Sendo ela uma implementação de licença MIT feita por Auburns([AUBURNS, 2018](#)). Sua implementação é ilustrada pelo diagrama da Figura 53.

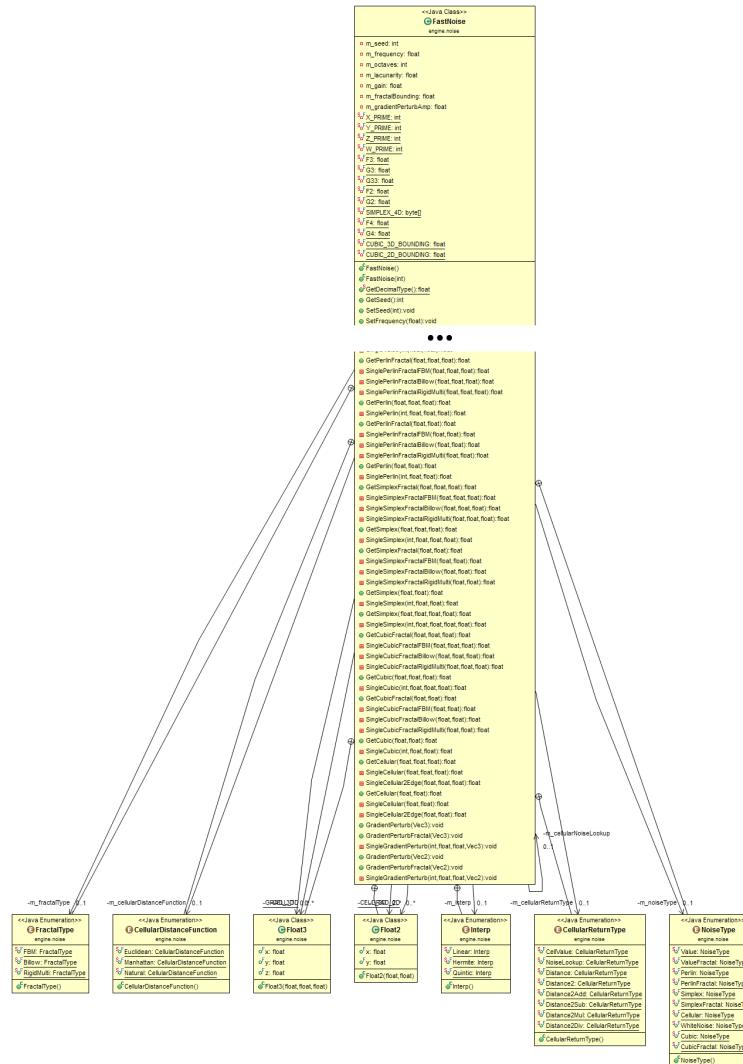


Figura 53 – Classes do pacote engine.noise

6.10 Pacote de renderizadores

O pacote `engine.renderer` contém as classes responsáveis pelos tipos de renderização. Sua implementação é ilustrada pelo diagrama da Figura 54.

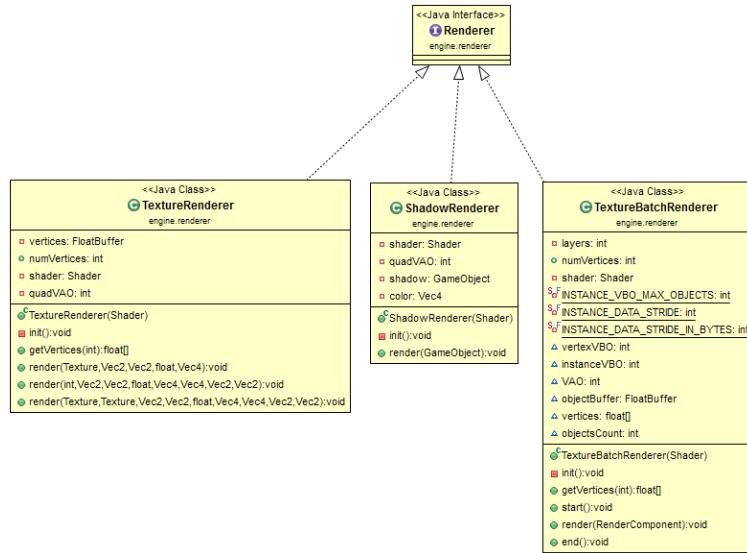


Figura 54 – Classes do pacote engine.renderer

6.11 Pacote de utilidades

O pacote `engine.utilities` contém uma série de classes consideradas utilitárias como por exemplo, criação e conversão de buffers, leitura de arquivos, conversão de cores, funções matemáticas e muitas outras. Sua implementação é ilustrada pelo diagrama da Figura 55.

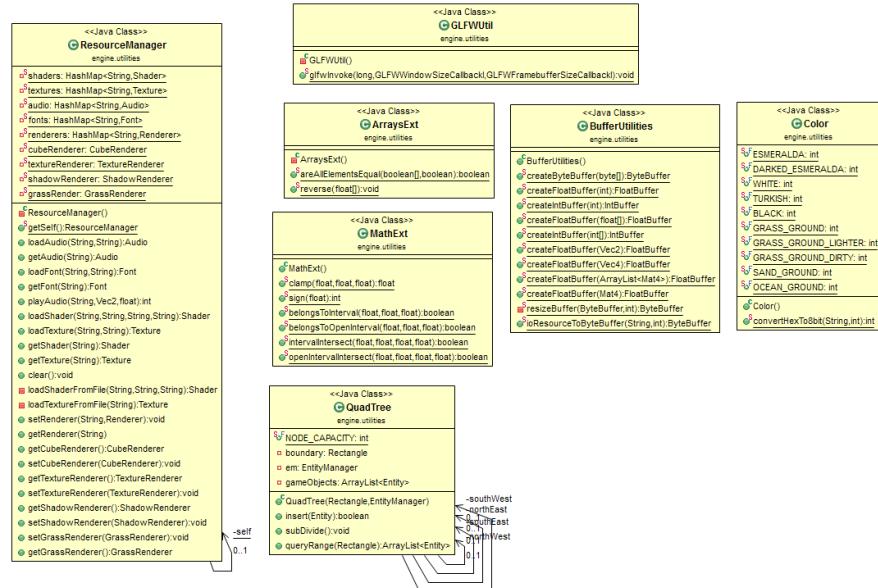


Figura 55 – Classes do pacote engine.utilities

6.12 Estrutura dos arquivos

A estrutura geral de arquivos e pacotes do projeto da Engine Narval pode ser vista na Figura 56. Cada arquivo representa uma classe Java e cada pasta é denotada por uma barra e considerada como um pacote do projeto.

```
1  Engine
2    /ai
3      Action
4        AStar
5        Anode
6        Consideration
7    /audio
8      Audio
9        AudioSource
10   /controllers
11     Controller
12     PlayerController
13   /engine
14     Engine
15     GameState
16     PhysicsEngine
17     Settings
18     Window
19   /entity
20     /component
21       Component
22       PositionComponent
23       RenderC0mponent
24     SystemManager
25     PositionSystem
26     RenderSystem
27   /geometry
28     Rectangle
29     Segment
30   /graphic
31     Shader
32     Animation
```

```
1  Texture
2  /input
3    Control
4      JoystickControl
5      KeyboardControl
6      MouseControl
7  /logic
8    Camera
9    Chunk
10   ChunkMap
11   ReadingChunk
12   SavingChunk
13   Timer
14  /noise
15  FastNoise
16  /renderer
17    AnimationsManager
18    Renderer
19    TextureRenderer
20    BatchTextureRenderer
21  /ui
22    Font
23    Glyph
24  /utilities
25    ResourceManager
26    ArraysExt
27    BufferUtilities
28    Color
29    GLFWUtil
30    MathExt
31    QuadTree
32
```

Figura 56 – Árvore de arquivos do projeto

7 Experimentos e resultados

Os resultados obtidos neste trabalho são demonstrados e sumarizados neste capítulo. Será apresentado a demonstração do protótipo final bem como alguns exemplos de código para mostrar o fluxo de funcionamento da engine Narval.

7.1 Inicialização e exemplo de utilização da engine

A engine Narval tem início criando uma janela, que incorpora o contexto OpenGL e define algumas de suas variáveis ambiente, e uma Thread onde rodar o sistema, como demonstrado na Listagem 7.1.

```
1 public class Main {
2     public static void main(String args[]) {
3         Window w;
4         w = new Window(1280,720,"Engine Early Alpha");
5
6         Engine.getSelf().attachWindow(w);
7
8         Thread tr = new Thread(Engine.getSelf());
9         tr.start();
10    }
11 }
```

Listagem 7.1 – Classe Main

Na classe `Engine`, um singleton, é onde todo o sistema acontece. Nessa classe é feita as chamadas dos métodos `update` e `render` abordados na Seção 3.1. Ela fica responsável por inicializar o contexto do OpenAL, dos *callbacks* dos sistemas de input do mouse e teclado e por criar um gerenciador de estados chamado GSM (Game State Manager). Nela também são definidos algumas variáveis estáticas utilizadas por subsistemas e outras privadas para algumas métricas. Seu código completo pode ser encontrado no Apêndice E.

A partir desse ponto todo desenvolvimento do jogo é centralizado no `GameState` configurado como atual no `GSM`. Para essa demonstração será criado uma pequena entidade com textura para ser renderizada na tela. Para isso cria-se uma classe que estenda `GameState` onde nela será implementado os métodos `init`, `render`, `update` e `variableUpdate` que são invocados em cascata a partir da `Engine`. Esse fluxo fica claro na Figura 57.



Figura 57 – Fluxo de chamadas do método `update` a partir da `Engine`. Esse mesmo fluxo também vale para os métodos `render` e `variableUpdate`.

Na classe `Game` que estende `GameState` define-se a seguinte implementação exemplo para instanciar e renderizar uma entidade simples com um componente de textura. Todo esse processo é mostrado na Listagem 7.2.

```

1
2  public class Game extends GameState{
3      private EntityManager em;
4      private SystemManager sm;
5      private Camera camera;
6
7      public void init() {
8          screenView = new Rectangle(0,0,Engine.getSelf().getWindow().getWidth(),
9              Engine.getSelf().getWindow().getHeight());
10         chunkMap = new ChunkMap(seed,em);
11         em = new EntityManager();
12         sm = new SystemManager();
13
14         //=====
15         //Loads all shaders
16         //=====
17         ResourceManager.getSelf().loadShader("texture",
18             "shaders/texture.vert",
19             "shaders/texture.frag",
20             null);
21
22         //=====
23         //Loads all textures
24         //=====
25         ResourceManager.getSelf().loadTexture("rogue",
26             "sprites/rogue.png");
27
28         //=====
29         //Loads all Audio
30         //=====
31         ResourceManager.getSelf().loadAudio("ocean_waves","audio/ocean_waves.ogg");
32
33         //=====
34         //Set all Uniforms
35         //=====
```

```

36     Mat4 projection = new Mat4();
37     projection = projection.ortho(0, Engine.getSelf().getWindow().getWidth(),
38                                 Engine.getSelf().getWindow().getHeight(), 0, -1f, 1f);
39
40     ResourceManager.getSelf().getShader("texture").use();
41     ResourceManager.getSelf().getShader("texture").setMat4("projection",
42                                         projection);
43     ResourceManager.getSelf().getShader("texture").setVec3("ambientColor", new
44                                         Vec3(0,0,0));
45
46 //=====
47 //Start renderers
48 //=====
49 TextureRenderer t = new TextureRenderer(ResourceManager.getSelf().getShader(
50                                         "texture"));
51 ResourceManager.getSelf().setRenderer("textureRenderer", t);
52
53 //=====
54 //Creates player
55 //=====
56 player = em.newEntity();
57 em.setPlayerID(player.getID());
58 player.setName("player");
59
60 AnimationStateManager asm = new AnimationStateManager();
61
62 Animation a = new Animation("rogue", 150);
63 a.setFrames(10, new Vec2(0,0), new Vec2(32,32));
64 asm.addAnimation("idle_1", a);
65
66 asm.changeStateTo("idle_1");
67
68 Vec2 size= new Vec2(128,128);
69 Rectangle baseBoxProportions = new Rectangle(0f,0.8f,1.0f,0.2f);
70
71 RenderComponent rc = new RenderComponent(player.getID());
72 rc.setSize(size);
73 rc.setColor(new Vec4(1,1,1,1));
74 rc.setAnimations(asm);
75 rc.setBaseBox(baseBoxProportions);
76 rc.setRenderer("textureRenderer");
77 rc.setRenderPosition(startPoint);
78 em.addComponentTo(player, rc);
79
80 //=====
81 //Camera
82 //=====
83 camera = new Camera(em);
84 camera.setFocusOn(player);
85 camera.move(-startPoint.x, -startPoint.y);
86
87 //=====
88 // Entity Systems
89 //=====
90
91 RenderSystem rs = new RenderSystem(this);

```

```

89     sm.addSystem(rs);
90 }
91
92 @Override
93 public void render() {
94     glClearColor(1,1,1,1);
95     glClear(GL_COLOR_BUFFER_BIT);
96
97     sm.render();
98 }
99
100 @Override
101 public void variableUpdate(float alpha) {
102     sm.variableUpdate(alpha);
103     camera.variableUpdate(alpha);
104 }
105
106 @Override
107 public void update(float deltaTime) {
108     PositionComponent ppc= em.getFirstComponent(player, PositionComponent.class)
109         ;
110     alListener3f(AL_POSITION, ppc.getPosition().x, ppc.getPosition().y,0);
111     sm.update(deltaTime);
112     camera.update(deltaTime);
113 }

```

Listagem 7.2 – Classe Game

7.2 Demonstração

A demonstração final construída utilizando a game engine evidência suas principais funcionalidades em um mini jogo de exploração com ambiente procedural 2D.

Nele pode-se caminhar através do cenário explorando o território gerado em tempo real e vislumbrar praias e florestas. Durante essa jornada pode-se notar o efeito de iluminação local, global e sua atenuação juntamente com o passar do dia. Nota-se ainda o efeito de vento aplicado aos objetos que compõem a vegetação e entidades de AI simples que vagam pelo ambiente. Nas Figuras 58, 59, 60, 61 e 62 são dadas algumas capturas de tela da demonstração final construída com a engine.



Figura 58 – Captura de tela 1



Figura 59 – Captura de tela 2



Figura 60 – Captura de tela 3



Figura 61 – Captura de tela 4



Figura 62 – Captura de tela 5

8 Conclusão

Com a primeira versão da game engine finalizada é necessário realizar algumas considerações finais. Um sistema dessa magnitude deve ser constantemente atualizado com novas tecnologias para manter-se relevante em um cenário competitivo como a indústria de jogos. A pesquisa permitiu produzir um extenso material em língua portuguesa acerca do tema, documentando e exemplificando cada etapa, desde a demonstração das amplas possibilidades para cada situação até sua implementação.

Embora distante de uma versão final, o projeto estando nessa etapa possui uma fundação sólida que lhe permite ser ampliado e melhorado para versões futuras. A engine não somente está com uma base bem construída, mas também está apta a ser utilizada na construção de jogos menos complexos.

Por ser um sistema de grande porte ele não somente embarca muitas funcionalidades, mas também muitas áreas do conhecimento. Como consequência direta disso, muitas delas ainda devem sofrer ajustes ao longo do tempo para atingirem um patamar de alta qualidade. Isso só será obtido através de tempo e, muito possivelmente, pela atuação direta de profissionais qualificados da área.

Em suma, a construção de uma game engine é a junção do esforço de diversas áreas para criar um software capaz de trazer novos mundos e entretenimento ao grande público através da arte digital.

8.1 Perspectivas de Trabalhos Futuros

Para versões futuras da engine Narval espera-se uma portabilidade para novas APIs gráficas como DirectX e Vulkan. Também espera-se o desenvolvimento de um editor de níveis que permita uma interface mais abrangente e acessível ao usuário comum. Pretende-se elaborar e criar um suporte para renderização 3D, bem como a expansão e utilização de técnicas mais avançadas de renderização para efeitos sofisticados.

Muito embora essa versão esteja finalizada, espera-se um dia lançá-la no mercado sob a forma de um produto muito bem polido. Para isso tem-se ainda muitas etapas que devem ser superadas para atingir este objetivo final. Algumas delas envolvem não somente aspectos da renderização, mas também aspectos mais gerais como uma manipulação mais eficiente de arquivos, criação de ferramentas para scripting, resolução de pathfinding usando multithreading e muitos outros.

Há muitos componentes que podem ser abstraídos e otimizados. A principal alteração que será feita nos trabalhos futuros é uma iteração mais inteligente pelos objetos

de uma forma geral, seja utilizando *quadtrees* ou *octrees* ao invés de listas encadeadas. Entretanto, para que isso seja possível é necessário desacoplar ainda mais alguns objetos para que não seja necessário realizar checagens em outros componentes que não necessariamente podem ser ordenados pelo mesmo tipo de dado comum à ambos.

Referências

AUBURNS. *Fast Noise*. 2018. Disponível em: <https://github.com/Auburns/FastNoise_Java>. Citado 2 vezes nas páginas 75 e 84.

BUCKLAND, M. *Programming Game AI by Example*. [S.l.]: Wordware Publishin, 2005. Citado na página 17.

DICTIONARY, O. 2018. Disponível em: <https://en.oxforddictionaries.com/definition/artificial_intelligence>. Citado na página 17.

ECK, D. J. *Introduction to Computer Graphics*. 2018. Disponível em: <<http://math.hws.edu/eck/cs424/downloads/graphicsbook-linked.pdf>>. Citado na página 16.

FATAHO. *Perlin Noise Explained*. 2018. Disponível em: <<https://www.youtube.com/watch?v=MJ3bvCkHJtE>>. Citado 4 vezes nas páginas 6, 72, 73 e 74.

GREGORY, J. *Game Engine Architecture*. [S.l.]: Taylor and Francis Group, 2009. ISBN 9781439865262. Citado 4 vezes nas páginas 16, 19, 20 e 25.

JBOX2D. *jBox2D*. 2018. Disponível em: <<http://www.jbox2d.org/>>. Citado na página 14.

KHRONOS. *OpenGL Specifications*. 2017. Disponível em: <<https://www.khronos.org/registry/OpenGL/specs/gl/>>. Citado na página 16.

KHRONOS. *History of OpenGL*. 2018. Disponível em: <https://www.khronos.org/opengl/wiki/History_of_OpenGL#OpenGL_3.0_.282008.29>. Citado na página 27.

KHRONOS. *OpenGL Object*. 2018. Disponível em: <https://www.khronos.org/opengl/wiki/OpenGL_Object>. Citado na página 29.

KHRONOS. *OpenGL shader compilation*. 2018. Disponível em: <https://www.khronos.org/opengl/wiki/Shader_Compilation>. Citado na página 34.

LWJGL. *LWJGL*. 2018. Disponível em: <<https://www.lwjgl.org/>>. Citado na página 14.

MARK, D.; DILL, K. *Improving AI Decision Modeling Through Utility Theory*. 2018. Disponível em: <<http://www.intrinsicalgorithm.com/media/2010GDC-DaveMark-KevinDill-UtilityTheory.pdf>>. Citado na página 66.

MCDONALD, E. *The Global Games Market Will Reach \$108.9 Billion in 2017 With Mobile Taking 42%*. 2017. Disponível em: <<https://newzoo.com/insights/articles/the-global-games-market-will-reach-108-9-billion-in-2017-with-mobile-taking-42/>>. Citado na página 14.

NYSTROM, R. *Game Programming Patterns*. [S.l.]: Genever Benning, 2014. ISBN 0990582906. Citado na página 24.

OPENAL. *openAL*. 2018. Disponível em: <<https://www.openal.org/>>. Citado na página 14.

OPENCV. 2018. Disponível em: <https://docs.opencv.org/3.1.0/d4/d61/tutorial_warp_affine.html>. Citado na página 55.

OPENGL. *About OpenGL*. 2017. Disponível em: <<https://www.opengl.org/about/>>. Citado na página 16.

PATEL, A. *Making maps with noise functions*. 2018. Disponível em: <<https://www.redblobgames.com/maps/terrain-from-noise/>>. Citado 4 vezes nas páginas 6, 75, 76 e 77.

PERLIN, K. An image synthesizer. 1985. Citado na página 72.

PERLIN, K. Improving noise. *ACM Trans. Graph.*, ACM, New York, NY, USA, v. 21, n. 3, p. 681–682, jul. 2002. ISSN 0730-0301. Disponível em: <<http://doi.acm.org/10.1145/566654.566636>>. Citado na página 72.

PHONG, B. T. Illumination for computer generated pictures. *Communications of the ACM*, ACM, v. 18, n. 6, p. 311–317, 1975. Citado na página 43.

SCRATCHAPIXEL. *The Phong Model, Introduction to the Concepts of Shader, Reflection Models and BRDF*. 2018. Disponível em: <<https://www.scratchapixel.com/lessons/3d-basic-rendering/phong-shader-BRDF>>. Citado na página 43.

SHAKER, N.; TOGELIUS, J.; NELSON, M. J. *Procedural Content Generation in Games*. [S.l.]: Springer, 2016. Citado na página 17.

SIZER, B. *The Total Beginner's Guide to Game AI*. 2018. Disponível em: <<https://www.gamedev.net/articles/programming/artificial-intelligence/the-total-beginners-guide-to-game-ai-r4942>>. Citado na página 66.

UNITY. 2018. Disponível em: <<https://docs.unity3d.com/Manual/UnderstandingFrustum.html>>. Citado na página 57.

VRIES, J. de. *Learn OpenGL*. 2. ed. [s.n.], 2015. Disponível em: <<https://learnopengl.com/book/offline%20learnopengl.pdf>>. Citado 7 vezes nas páginas 5, 16, 27, 29, 55, 57 e 58.

WENDERLICH, R. *Introduction to Component Based Architecture in Games*. 2018. Disponível em: <<https://www.raywenderlich.com/2806-introduction-to-component-based-architecture-in-games>>. Citado na página 50.

WIKIDOT, P. 2018. Disponível em: <<http://pcg.wikidot.com/>>. Citado na página 17.

WIKIPEDIA. 2018. Disponível em: <https://en.wikipedia.org/wiki/Affine_transformation#Affine_transformation_in_plane_geometry>. Citado 2 vezes nas páginas 5 e 56.

APÊNDICE A – Classe responsável pelo shader

```
1 public class Shader {
2     private int id;
3
4     public Shader use() {
5         glUseProgram(this.id);
6         return this;
7     }
8     public int getId() {
9         return id;
10    }
11
12    public void compile(String vertexSource, String fragmentSource, String
13        geometrySource) {
14        int vertexShader, geometryShader = 0, fragmentShader;
15
16        vertexShader = glCreateShader(GL_VERTEX_SHADER);
17        glShaderSource(vertexShader, vertexSource);
18        glCompileShader(vertexShader);
19        checkCompileErrors(vertexShader, "VERTEX");
20
21        fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
22        glShaderSource(fragmentShader, fragmentSource);
23        glCompileShader(fragmentShader);
24        checkCompileErrors(fragmentShader, "FRAGMENT");
25
26        if(geometrySource!=null) {
27            geometryShader = glCreateShader(GL_GEOMETRY_SHADER);
28            glShaderSource(geometryShader, geometrySource);
29            glCompileShader(geometryShader);
30            checkCompileErrors(fragmentShader, "GEOMETRY");
31        }
32
33        id = glCreateProgram();
34        glAttachShader(id, vertexShader);
35        if(geometrySource!=null)
36            glAttachShader(id, geometryShader);
37        glAttachShader(id, fragmentShader);
38        glLinkProgram(id);
39        checkCompileErrors(id, "PROGRAM");
40
41        glDeleteShader(vertexShader);
42        glDeleteShader(fragmentShader);
43        if(geometrySource!=null)
44            glDeleteShader(geometryShader);
45    }
46
47    private void checkCompileErrors(int object, String type) {
48        int success;
```

```

48     String log;
49
50     if(type != "PROGRAM") {
51         success = glGetShaderi(object, GL_COMPILE_STATUS);
52
53         if(success==0) {
54             log = glGetShaderInfoLog(object);
55             System.out.println(log);
56         }
57
58     }else {
59         success = glGetProgrami(object, GL_LINK_STATUS);
60
61         if(success==0) {
62             log = glGetProgramInfoLog(object);
63             System.out.println(log);
64         }
65
66     }
67 }
68
69 public void setFloat(String name, float value) {
70     glUniform1f(glGetUniformLocation(id, name), value);
71 }
72
73 public void setInteger(String name, int value) {
74     glUniform1i(glGetUniformLocation(id, name), value);
75 }
76
77 public void setVec2(String name, float x, float y) {
78     glUniform2f(glGetUniformLocation(id, name), x, y);
79 }
80
81 public void setVec2(String name, Vec2 pos) {
82     glUniform2f(glGetUniformLocation(id, name), pos.x, pos.y);
83 }
84
85 public void setVec3(String name, float x, float y, float z) {
86     glUniform3f(glGetUniformLocation(id, name), x, y, z);
87 }
88
89 public void setVec3(String name, Vec3 pos) {
90     glUniform3f(glGetUniformLocation(id, name), pos.x, pos.y, pos.z);
91 }
92
93 public void setVec4(String name, float x, float y, float z, float w) {
94     glUniform4f(glGetUniformLocation(id, name), x, y, z, w);
95 }
96
97 public void setVec4(String name, Vec4 pos) {
98     glUniform4f(glGetUniformLocation(id, name), pos.x, pos.y, pos.z, pos.w);
99 }
100
101 public void setMat4(String name, Mat4 m) {
102     glUniformMatrix4fv(glGetUniformLocation(id, name), false, BufferUtilities.
103                         createFloatBuffer(m));

```

104 }

Listagem A.1 – Classe responsável pelo Shader

APÊNDICE B – Implementação do Algoritmo A* Adaptado

```
1 public class AStar {
2     private ArrayList<Anode> openSet = new ArrayList<>();
3     private ArrayList<Anode> closedSet = new ArrayList<>();
4     private float M_SQRT2 = (float) Math.sqrt(2.0);
5     public static final int MAXSTEPS = 900;
6     private int widthSize = (128/8) +1, heightSize = (20/8)+1;
7     private boolean obstacleMap[][];
8
9     public ArrayList<Anode> calculatePath(Vec2i start, Vec2i end, boolean
10        obstacleMap[][]){
11         Anode startNode = new Anode();
12         Anode endNode = new Anode();
13         Anode current;
14         this.obstacleMap = obstacleMap;
15
16         startNode.pos    = start;
17         endNode.pos    = end;
18
19         startNode.setgScore(0);
20         startNode.setfScore( heuristic(startNode, endNode) );
21
22         openSet.add(startNode);
23
24         int k=0;
25         int index = 0;
26         while(!openSet.isEmpty()) {
27             if (k++ > MAXSTEPS)
28                 return null;
29
30             Collections.sort(openSet);
31
32             current = openSet.get(0);
33
34             if(current.pos.compareTo(endNode.pos) == 0)
35                 return constructPath(current);
36
37             openSet.remove(current);
38             closedSet.add(current);
39
40             for(Anode neighbor: getNeighbors(current)) {
41                 if(isSizeOccupied(neighbor))
42                     closedSet.add(neighbor);
43
44                 if(closedSet.contains(neighbor))
45                     continue;
46
47                 if(!openSet.contains(neighbor))
48                     openSet.add(neighbor);
```

```

48
49     float gScore = current.getgScore() + distance(current,neighbor);
50     if(gScore >= neighbor.getgScore())
51         continue;
52
53     neighbor.cameFrom = current;
54     neighbor.setgScore( gScore );
55     neighbor.setfScore( neighbor.getgScore() + heuristic(neighbor, endNode)
56                         );
57 }
58
59     return null;
60 }
61
62 public boolean isOutOfBounds(Anode node, boolean obstacleMap[][]){
63     if(node.pos.x>obstacleMap.length-1 || node.pos.y>obstacleMap[0].length-1 ||
64         node.pos.x<0 || node.pos.y<0)
65         return true;
66     return false;
67 }
68
69 private boolean isOccupied(Anode a) {
70     return (!isOutOfBounds(a, obstacleMap) && obstacleMap[a.pos.x][a.pos.y]);
71 }
72
73 private boolean isSizeOccupied(Anode u) {
74     Anode tempState = new Anode(u.pos.x , u.pos.y);
75
76     for(int y=0; y<heightSize; y++) {
77         for(int x=0; x<widthSize; x++) {
78             tempState.pos.x = u.pos.x + x;
79             tempState.pos.y = u.pos.y + y;
80
81             if(isOccupied(tempState))
82                 return true;
83         }
84     }
85
86     return false;
87 }
88
89 private ArrayList<Anode> getNeighbors(Anode current){
90     ArrayList<Anode> neighbors = new ArrayList<>();
91     Anode temp;
92
93     temp = new Anode(current.pos.x + 1, current.pos.y);
94     neighbors.add(temp);
95
96     temp = new Anode(current.pos.x + 1, current.pos.y + 1);
97     neighbors.add(temp);
98
99     temp = new Anode(current.pos.x , current.pos.y + 1);
100    neighbors.add(temp);
101
102    temp = new Anode(current.pos.x - 1, current.pos.y + 1);
103    neighbors.add(temp);

```

```

103
104     temp = new Anode(current.pos.x - 1, current.pos.y);
105     neighbors.add(temp);
106
107     temp = new Anode(current.pos.x - 1, current.pos.y - 1);
108     neighbors.add(temp);
109
110     temp = new Anode(current.pos.x , current.pos.y - 1);
111     neighbors.add(temp);
112
113     temp = new Anode(current.pos.x + 1, current.pos.y - 1);
114     neighbors.add(temp);
115
116     return neighbors;
117 }
118
119 private ArrayList<Anode> constructPath(Anode current){
120     Anode temp = current;
121     ArrayList<Anode> path = new ArrayList<>();
122
123     while(temp.cameFrom!=null) {
124         path.add(temp.cameFrom);
125         temp = temp.cameFrom;
126     }
127
128     return path;
129 }
130
131 private float heuristic(Anode start, Anode end) {
132     float temp;
133     float min = Math.abs(start.pos.x - end.pos.x);
134     float max = Math.abs(start.pos.y - end.pos.y);
135
136     if (min > max){
137         temp = min;
138         min = max;
139         max = temp;
140     }
141
142     return ((M_SQRT2-1.0f)*min + max);
143 }
144
145 private float distance(Anode a, Anode b){
146     float x = a.pos.x-b.pos.x;
147     float y = a.pos.y-b.pos.y;
148     return (float) Math.sqrt(x*x + y*y);
149 }
150
151 private Anode lowestFScore(ArrayList<Anode> list) {
152     Anode current = list.get(0);
153
154     for(Anode a: list) {
155         if(current.getfScore()>a.getfScore())
156             current = a;
157     }
158
159     return current;

```

```
160      }
161  }
```

Listagem B.1 – A* adaptado

APÊNDICE C – Implementação do sistema Entity Based

```
1 public class Entity {  
2     private long id;  
3     private String name;  
4  
5     public Entity(long id) {  
6         this.id = id;  
7     }  
8  
9     public long getID() {  
10        return id;  
11    }  
12  
13    public String getName() {  
14        return name;  
15    }  
16  
17    public void setName(String name) {  
18        this.name = name;  
19    }  
20}
```

Listagem C.1 – Classe Entity

```
1  
2 public class EntityManager {  
3     private long lastID = 0;  
4     private ArrayList<Entity> entities = new ArrayList<>();  
5     private HashMap<String, ArrayList<Component>> componentsByClass = new HashMap  
      <>();  
6     private HashMap<Long, ArrayList<Component>> componentsOf = new HashMap<>();  
7     private long playerID;  
8     private EntityManager self;  
9     private int maxEntities = 20000;  
10  
11  
12     public long generateID() {  
13         return lastID++;  
14     }  
15  
16     public Entity newEntity() {  
17         long id = generateID();  
18         Entity e = new Entity(id);  
19         if(id>maxEntities)  
20             entities.set((int) (id%maxEntities),e);  
21         else  
22             entities.add(e);  
23         return e;  
24     }  
25 }
```

```

26     public void addComponentTo(Entity e, Component c) {
27
28         if(componentsOf.get(e.getID())==null)
29             componentsOf.put(e.getID(), new ArrayList<>());
30
31         if(componentsByClass.get(c.getClass().getName())==null)
32             componentsByClass.put(c.getClass().getName(), new ArrayList<>());
33
34         componentsOf.get(e.getID()).add(c);
35         componentsByClass.get(c.getClass().getName()).add(c);
36     }
37
38     public ArrayList<Component> getComponent(long entityID, Class c) {
39         ArrayList<Component> comps= new ArrayList<>();
40
41         for(Component cp: componentsOf.get(entityID))
42             if(c.isInstance(cp))
43                 comps.add(cp);
44
45         return comps;
46     }
47
48     public <T extends Component> T getFirstComponent(Entity e, Class c) {
49         for(Component cp: componentsOf.get(e.getID()))
50             if(c.isInstance(cp))
51                 return (T) cp;
52
53         return null;
54     }
55
56     public <T extends Component> T getFirstComponent(long id, Class c) {
57         for(Component cp: componentsOf.get(id))
58             if(c.isInstance(cp))
59                 return (T) cp;
60
61         return null;
62     }
63
64     public void removeEntity(Entity e) {
65         componentsOf.remove(e.getID());
66     }
67
68     public ArrayList<Entity> getAllEntities() {
69         return entities;
70     }
71
72     public <T extends Component> ArrayList<T> getAllComponents(Class c){
73         return (ArrayList<T>) componentsByClass.get(c.getName());
74     }
75
76     public ArrayList<Entity> getAllEntitiesWithComponent(Class c) {
77         ArrayList<Entity> ents = new ArrayList<>();
78
79         for(Entity e: entities)
80             for(Component cp: componentsOf.get(e.getID()))
81                 if(c.isInstance(cp)) {
82                     ents.add(e);

```

```

83         continue;
84     }
85
86     return ents;
87 }
88
89     public long getPlayerID() {
90         return playerID;
91     }
92
93     public void setPlayerID(long playerID) {
94         this.playerID = playerID;
95     }
96 }
97 }
```

Listagem C.2 – Classe EntityManager responsável por gerenciar as entidades

```

1  public abstract class ComponentSystem {
2      protected Game context;
3
4      public ComponentSystem(Game context) {
5          this.context = context;
6      }
7
8      public abstract void update(float dt);
9      public abstract void variableUpdate(float alpha);
10     public abstract void render();
11 }
```

Listagem C.3 – Classe ComponentSystem responsável por definir a abstração dos sistemas de componente

```

1  public class SystemManager {
2
3      private ArrayList<ComponentSystem> systems = new ArrayList<>();
4
5
6      public void update(float dt) {
7          for(ComponentSystem cs: systems)
8              cs.update(dt);
9      }
10
11     public void variableUpdate(float alpha) {
12         for(ComponentSystem cs: systems)
13             cs.variableUpdate(alpha);
14     }
15
16     public void render() {
17         for(ComponentSystem cs: systems)
18             cs.render();
19     }
20
21     public void addSystem(ComponentSystem cs) {
22         systems.add(cs);
23     }
24 }
```

```
24 }
```

Listagem C.4 – Classe SystemManager responsável por gerenciar os sistemas de componentes

```
1 public abstract class Component{
2     private long entityID;
3
4     public Component(long entityID) {
5         this.entityID = entityID;
6     }
7
8     public long getEntityID() {
9         return entityID;
10    }
11 }
```

Listagem C.5 – Abstração da classe Component

```
1 package engine.entity.component;
2
3 import engine.geometry.Rectangle;
4 import engine.logic.AnimationStateManager;
5 import engine.logic.GameObject;
6 import glm.vec._2.Vec2;
7 import glm.vec._4.Vec4;
8
9 public class RenderComponent extends Component implements Comparable<
10    RenderComponent>{
11     private Vec2 renderPosition = new Vec2(0,0);
12     private String texture;
13     private Vec2 orientation = new Vec2(0,0);
14     private Vec2 size      = new Vec2(0,0);
15     private Vec2 anchorPoint = new Vec2(0,0);
16     private Vec2 skew      = new Vec2(0,0);
17     private AnimationStateManager animations;
18     private float rotation;
19     private Rectangle boundingBox = new Rectangle(0,0,0,0);
20     private Rectangle calculatedBaseBox = new Rectangle(0,0,0,0);
21     private Rectangle baseBox = new Rectangle(0,0,0,0);
22     private boolean disabled = false;
23     private String Renderer = "";
24
25     public RenderComponent(long entityID) {
26         super(entityID);
27     }
28
29     public void setBaseBox(Rectangle baseBox) {
30         this.baseBox = baseBox;
31     }
32
33     public Rectangle getCalculatedBaseBox() {
34
35         calculatedBaseBox.x = renderPosition.x + size.x * baseBox.x;
36         calculatedBaseBox.y = renderPosition.y + size.y * baseBox.y;
37     }
38 }
```

```

38     calculatedBaseBox.width = size.x * baseBox.width;
39     calculatedBaseBox.height = size.y * baseBox.height;
40
41     return calculatedBaseBox;
42 }
43
44 public Rectangle getBoundingBox() {
45     boundingBox.x = renderPosition.x;
46     boundingBox.y = renderPosition.y;
47     boundingBox.width = size.x;
48     boundingBox.height = size.y;
49
50     return boundingBox;
51 }
52 public Vec2 getRenderPosition() {
53     return renderPosition;
54 }
55 public void setRenderPosition(Vec2 renderPosition) {
56     this.renderPosition = renderPosition;
57 }
58 public void setRenderPosition(float x, float y) {
59     this.renderPosition.x = x;
60     this.renderPosition.y = y;
61 }
62 public String getTexture() {
63     return texture;
64 }
65 public void setTexture(String texture) {
66     this.texture = texture;
67 }
68 public Vec2 getOrientation() {
69     return orientation;
70 }
71 public void setOrientation(Vec2 orientation) {
72     this.orientation = orientation;
73 }
74 public Vec2 getSize() {
75     return size;
76 }
77 public void setSize(Vec2 size) {
78     this.size = size;
79 }
80 public Vec2 getAnchorPoint() {
81     return anchorPoint;
82 }
83 public void setAnchorPoint(Vec2 anchorPoint) {
84     this.anchorPoint = anchorPoint;
85 }
86 public Vec2 getSkew() {
87     return skew;
88 }
89 public void setSkew(Vec2 skew) {
90     this.skew = skew;
91 }
92 public Vec4 getColor() {
93     return color;
94 }
```

```

95  public void setColor(Vec4 color) {
96      this.color = color;
97  }
98  public AnimationStateManager getAnimations() {
99      return animations;
100 }
101 public void setAnimations(AnimationStateManager animations) {
102     this.animations = animations;
103 }
104 public float getRotation() {
105     return rotation;
106 }
107 public void setRotation(float rotation) {
108     this.rotation = rotation;
109 }
110 public boolean isDisabled() {
111     return disabled;
112 }
113 public void setDisabled(boolean disabled) {
114     this.disabled = disabled;
115 }
116
117 @Override
118 public int compareTo(RenderComponent rc) {
119     Rectangle r = getCalculatedBaseBox();
120
121     if(r.y > rc.getCalculatedBaseBox().y)
122         return 1;
123     if(r.y == rc.getCalculatedBaseBox().y)
124         return 0;
125     if(r.y < rc.getCalculatedBaseBox().y)
126         return -1;
127
128     return 0;
129 }
130
131 public String getRenderer() {
132     return Renderer;
133 }
134
135 public void setRenderer(String renderer) {
136     Renderer = renderer;
137 }
138 }
```

Listagem C.6 – Exemplo de especialização da classe Component na classe RenderComponent

APÊNDICE D – Classe responsável pela gerência de recursos

```
1  public final class ResourceManager {
2      private static ResourceManager self;
3      private static HashMap<String, Shader> shaders = new HashMap<String, Shader>();
4      private static HashMap<String, Texture> textures = new HashMap<String, Texture>();
5      private static HashMap<String, Audio> audio = new HashMap<String, Audio>();
6      private static HashMap<String, Font> fonts = new HashMap<String, Font>();
7      private static HashMap<String, Renderer> renderers = new HashMap<String, Renderer>();
8
9
10     private ResourceManager() {}
11
12     public static ResourceManager getSelf() {
13         if(self == null)
14             self = new ResourceManager();
15         return self;
16     }
17
18     public Audio loadAudio(String name, String audioPath) {
19         if(audio.containsKey(name))
20             return audio.get(name);
21         return audio.put(name, new Audio(audioPath));
22     }
23
24     public Audio getAudio(String name) {
25         return audio.get(name);
26     }
27
28     public Font loadFont(String name, String fontPath) {
29         if(fonts.containsKey(name))
30             return fonts.get(name);
31         return fonts.put(name, new Font());
32     }
33
34     public Font getFont(String name) {
35         return fonts.get(name);
36     }
37
38     public int playAudio(String name, Vec2 pos, float maxDistance) {
39         AudioSource a = new AudioSource(audio.get(name).getBufferPointer(), pos,
40                                         maxDistance);
41         a.play();
42
43         return a.getSourcePointer();
44     }
45 }
```

```

45     public Shader loadShader(String name, String vertexShaderPath, String
46         fragmentShaderFilPath, String geometryShaderPath) {
47         if(shaders.containsKey(name))
48             return shaders.get(name);
49         return shaders.put(name, loadShaderFromFile(vertexShaderPath,
50             fragmentShaderFilPath, geometryShaderPath));
51     }
52
53     public Texture loadTexture(String name, String imgPath) {
54         if(textures.containsKey(name))
55             return textures.get(name);
56         return textures.put(name, loadTextureFromFile(imgPath));
57     }
58
59     public Shader getShader(String name) {
60         return shaders.get(name);
61     }
62
63     public Texture getTexture(String name) {
64         return textures.get(name);
65     }
66
67     private Shader loadShaderFromFile(String vertexShaderPath, String
68         fragmentShaderPath, String geometryShaderPath) {
69         String vertexCode = null, fragmentCode = null, geometryCode = null;
70
71         try {
72             //Vertex Code
73             StringBuilder stringBuilder = new StringBuilder();
74             BufferedReader bufferedReader;
75
76             bufferedReader = new BufferedReader(new InputStreamReader(
77                 this.getClass().getResourceAsStream("//" + vertexShaderPath)
78             ));
79             String line;
80
81             while((line = bufferedReader.readLine())!=null)
82                 stringBuilder.append(line).append("\n");
83
84             bufferedReader.close();
85             vertexCode = stringBuilder.toString();
86
87             //Fragment Code
88             stringBuilder = new StringBuilder();
89
90             bufferedReader = new BufferedReader(new InputStreamReader(
91                 this.getClass().getResourceAsStream("//" + fragmentShaderPath
92             ))
93             );
94
95             while((line = bufferedReader.readLine())!=null)
96                 stringBuilder.append(line).append("\n");
97
98             bufferedReader.close();
99             fragmentCode = stringBuilder.toString();

```

```

98
99         //Geometry Code
100        if(geometryShaderPath!=null){
101            stringBuilder = new StringBuilder();
102            bufferedReader = new BufferedReader(new InputStreamReader(
103                this.getClass().getResourceAsStream("//" +
104                    geometryShaderPath)
105            ));
106            while((line = bufferedReader.readLine())!=null)
107                stringBuilder.append(line).append("\n");
108
109            bufferedReader.close();
110            geometryCode = stringBuilder.toString();
111        }
112    } catch (IOException e) {
113        e.printStackTrace();
114    }
115
116    Shader shader = new Shader();
117    shader.compile(vertexCode, fragmentCode, (geometryCode==null)? null :
118        geometryCode);
119    return shader;
120}
121
122 private Texture loadTextureFromFile(String imgPath) {
123     Texture t = new Texture("//"+imgPath);
124     return t;
125 }
126
127 public void setRenderer(String name, Renderer r) {
128     renderers.put(name, r);
129 }
130
131 public <T extends Renderer> T getRenderer(String name) {
132     return (T) renderers.get(name);
133 }

```

Listagem D.1 – Classe Resource Manager

APÊNDICE E – Classe núcleo do sistema: Engine

```
1  public class Engine implements Runnable{
2      private Window window;
3      private long deltaTime, currentFrame, lastFrame, lastSecond;
4      private int updates = 0, fps = 0;
5      private KeyboardControl keyboard;
6      private MouseControl mouse;
7      public static final long SECOND = 1000000000L; //10^9
8      public static final long MILISECOND = 1000000L; //10^6
9      public static final int TARGET_UPDATES = 60;
10     public static final float TARGET_DT = 1f/(float)TARGET_UPDATES;
11     private float accumulator = 0;
12     private static Engine self;
13     private long higher = 0;
14     private long lower = Long.MAX_VALUE;
15     private long avg = 0;
16     private int longestDelay = 0;
17     private float alphaInterpolator = 1;
18
19
20     private long higherRender = 0;
21     private long avgRender = 0;
22
23     private Engine() {}
24
25     public static Engine getSelf() {
26         if(self==null)
27             self = new Engine();
28
29         return self;
30     }
31
32     public void attachWindow(Window w) {
33         if(window==null)
34             window = w;
35     }
36
37     private void init() {
38         window.init();
39         initAudioSystem();
40
41         GSM.getSelf().changeStateTo(GSM.GAME_STATE);
42
43         keyboard = new KeyboardControl();
44         mouse = new MouseControl();
45
46         glfwSetKeyCallback(window.getId(), keyboard);
47         glfwSetCursorPosCallback(window.getId(), mouse);
48     }
```

```

49     GSM.getSelf().setKeyboard(keyboard);
50     GSM.getSelf().setMouse(mouse);
51 }
52
53 private void initAudioSystem() {
54     String defaultDeviceName = alcGetString(0, ALC_DEFAULT_DEVICE_SPECIFIER);
55     long device = alcOpenDevice(defaultDeviceName);
56
57     int[] attributes = {0};
58     long context = alcCreateContext(device, attributes);
59     alcMakeContextCurrent(context);
60
61     ALCCapabilities alcCapabilities = ALC.createCapabilities(device);
62 }
63
64 private int update() {
65     int count = 0;
66     currentFrame = System.nanoTime();
67     deltaTime = currentFrame - lastFrame;
68     lastFrame = currentFrame;
69
70     float deltaTimeMiliSeconds = (float)deltaTime/(float)SECOND;
71
72     if(deltaTimeMiliSeconds > TARGET_DT*(TARGET_UPDATES*0.1f))
73         deltaTimeMiliSeconds = TARGET_DT*(TARGET_UPDATES*0.1f);
74
75     accumulator += deltaTimeMiliSeconds;
76
77     while (accumulator > TARGET_DT) {
78         count++;
79         accumulator -= TARGET_DT;
80     }
81
82     alphaInterpolator = accumulator / TARGET_DT;
83
84     for(int i=0; i<count;i++) {
85         long previous = System.nanoTime();
86
87         glfwPollEvents();
88         GSM.getSelf().update(TARGET_DT);
89         PhysicsEngine.getSelf().update(TARGET_DT);
90
91         long elapsed = System.nanoTime() - previous;
92         if(elapsed>higher)
93             higher = elapsed;
94         else if(elapsed<lower)
95             lower = elapsed;
96
97         avg +=elapsed;
98     }
99
100    if(count>longestDelay)
101        longestDelay = count;
102    return count;
103 }
104
105 public void variableUpdate() {

```

```

106     GSM.getSelf().variableUpdate(alphaInterpolator);
107 }
108
109 private void render() {
110     long previous = System.nanoTime();
111
112     GSM.getSelf().render();
113     glfwSwapBuffers(window.getId());
114
115     long elapsed = System.nanoTime() - previous;
116     if(elapsed>higherRender)
117         higherRender = elapsed;
118     avgRender +=elapsed;
119 }
120
121 @Override
122 public void run() {
123     init();
124     lastSecond = System.nanoTime();
125
126     while(!glfwWindowShouldClose(window.getId())) {
127         updates += update();
128
129         variableUpdate();
130         render();
131         fps++;
132
133         if((currentFrame-lastSecond)>SECOND) {
134
135             System.out.printf(
136                 "\n=====\n"
137                 "\nUPS :\t"+updates+
138                 "\nFPS :\t"+fps+'\n'+
139                 "\nUPS AVG :\t"+(float)(avg/uploads)/MILISECOND+"ms "+
140                 "\nUPS pike :\t"+(float)higher/MILISECOND+"ms "+
141                 "\nUPS pike calls :\t"+longestDelay+'\n'+
142                 "\nFPS AVG :\t"+(float)(avgRender/fps)/MILISECOND+"ms "+
143                 "\nFPS pike :\t"+(float)higherRender/MILISECOND
144
145         );
146
147         lastSecond = System.nanoTime();
148         updates = 0;
149         fps = 0;
150         lower = Long.MAX_VALUE;
151         higher = 0;
152         avg = 0;
153         longestDelay = 0;
154
155         avgRender = 0;
156         higherRender = 0;
157     }
158 }
159 }
160 }
161
162 public Window getWindow() {

```

```

163     return window;
164 }
165
166 public float getAccumulator() {
167     return accumulator;
168 }
169
170 public float getAlphaInterpolator() {
171     return alphaInterpolator;
172 }
173 }
```

Listagem E.1 – Classe Engine

```

1  public final class GSM {
2      private static GSM self;
3      private GameState actualState;
4      private KeyboardControl keyboard;
5      private MouseControl mouse;
6      public static final int GAME_STATE = 0;
7      public static final int MENU_STATE = 1;
8      public static int CURRENT_STATE = GAME_STATE;
9
10
11     private GSM() {
12     }
13
14     public static GSM getSelf() {
15         if(self==null)
16             self = new GSM();
17         return self;
18     }
19
20     public void changeStateTo(int state) {
21         switch (state) {
22             case GAME_STATE:
23                 CURRENT_STATE = state;
24                 actualState = new Game();
25                 actualState.init();
26                 break;
27             default:
28                 break;
29         }
30     }
31
32     public void render() {
33         actualState.render();
34     }
35
36     public void update(float deltaTime) {
37         actualState.update(deltaTime);
38     }
39     public void variableUpdate(float deltaTime) {
40         actualState.variableUpdate(deltaTime);
41     }
42
43     public void setKeyboard(KeyboardControl keyboard) {
```

```
44     this.keyboard = keyboard;
45 }
46
47 public void setMouse(MouseControl mouse) {
48     this.mouse = mouse;
49 }
50 public MouseControl getMouse() {
51     return mouse;
52 }
53 public KeyboardControl getKeyboard() {
54     return keyboard;
55 }
56 }
```

Listagem E.2 – Classe GSM

```
1 public abstract class GameState {
2     public abstract void init();
3     public abstract void render();
4     public abstract void update(float deltaTime);
5     public abstract void variableUpdate(float deltaTime);
6 }
```

Listagem E.3 – Classe GameState