UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Igor Batista Fernandes

Desenvolvimento de jogos 2D multiplataforma utilizando OpenGL e Java

Uberlândia, Brasil

2018

Igor Batista Fernandes

Desenvolvimento de jogos 2D multiplataforma 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 um jogo multiplataforma para desktop intitulado Traveller. A elicitação dos requisitos para a elaboração do jogo será documentada em um artefato denominado *Game Design Document* que permite estruturar, sistematizar e organizar o processo de construção do jogo. A implementação de código será realizada na linguagem Java e através da API gráfica OpenGL usando GLFW (OpenGL Frame Work). O jogo será elaborado utilizando-se algoritmos procedurais para geração de algumas artes e ambientes, algoritmos genéticos para o comportamento de Non Playable Characters (NPC) e padrões de projeto como o MVC.

Palavras-chave: Indie Game, Game Design Document, Jogos, Java, OpenGL

Resumo

This work proposes to develop a multiplataform game for desktop entitled Traveller. The elicitation of requisites for the game will be documented in an artifact called Game Design Document which allows to structure, systematize and organize the process of building a game. The code implementation will be done in Java through the API OpenGL and GLFW. The game will be elaboreted using procedural algorithms to generate some art and environment, genetic algorithms for NPC behaviours and design patterns such as MVC.

Keywords: Indie Game, Game Design Document, Games, Java, OpenGL

Lista de ilustrações

Lista de tabelas

Lista de Algoritmos

Lista de abreviaturas e siglas

GDD	Game Design Document
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
ASD	A Ser Definido
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

Sumário

1 Introdução

1.1 Visão Geral da Proposta

O mercado internacional de jogos movimentará aproximadamente 108.9 bilhões de dólares distribuídos entre 2.2 bilhões de jogadores. Destes, 58% representam os segmentos de PC e consoles (MCDONALD, 2017). Além de ser uma área comercial extremamente lucrativa é também interdisciplinar e aproxima 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 respeitar as etapas de pré-produção, produção, testes e pós-produção. Ao longo de todas as etapas serão definidas estratégias de implementação em linguagens orientadas a objetos, Engenharia de software voltada par jogos e metodologias ágeis. Desenvolvida pela Khronos Group a API OpenGL fornece tudo que há de mais avançado em aplicações gráficas, sendo acelerada diretamente em hardware, multiplataforma, robusta e totalmente escalável.

1.2 Objetivo

O objetivo é desenvolver um jogo seguindo a construção de um Game Design Document (GDD) e ter o produto final publicado e distribuído comercialmente de maneira digital. Ao final deste documento será apresentada a primeira fase do capítulo introdutório, jogável e com qualidade profissional.

1.3 Justificativa e Motivação

O consumo cada vez maior deste tipo de produto pela população de todas as idades traz cada vez mais oportunidades de trabalho e aprendizagem. A insuficiência de materiais 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 pelas quais um jogo passa para torna-se um software que será distribuído e jogado por pessoas do mundo todo.

1.4 Metodologia

Este trabalho se classifica como uma pesquisa aplicada, sendo o método de desenvolvimento baseado nas etapas a seguir:

Etapa 1 – Revisão sistemática em desenvolvimento de jogos

- Desenvolvimento de Jogos estilo RPG - levantamento bibliográfico, identificação e organização dos trabalhos relacionados e do estado da arte;
- Pesquisa documental sobre engenharia de software e padrões de projeto aplicada a jogos;
- Pesquisa documental sobre o uso de metodologias ágeis no desenvolvimento de jogos.

Etapa 2 – Projeto do jogo identificando os conceitos a serem aplicados

1. Pré-Projeto: análise de mercado, tendências e viabilidade; definição da ideia do seu jogo; atividades a serem realizadas; equipe.
2. Pré-Produção: elaboração do Game Design Document, elaboração do Roteiro, elaboração do Documento de Arte e Design Gráfico; criação de protótipos para testar ideias, mecânicas e conceitos;
3. Plano de Produção: criação de um cronograma e planejamento dos ciclos de trabalho da produção, considerando o desenvolvimento do jogo utilizando uma metodologia ágil (SCRUM). Desenvolvimento de uma game engine/framework para a construção do jogo. Análise de requisitos do jogo. Os requisitos serão documentados usando diagramas de casos de uso em paralelo ao uso do GDD.

Etapa 3 – Implementação do jogo em formato digital. O desenvolvimento será dividido nos seguintes ciclos:

1. Desenvolvimento e testes de unidade do módulo gráfico (cenários, animações, mecânica).
2. Desenvolvimento e testes de unidade do módulo de físicas (colisões em geral).
3. Desenvolvimento e teste de unidade do módulo de áudio.
4. Desenvolvimento de interfaces e ferramentas.
5. Desenvolvimento das artes e teste de fluxo.
6. Teste de sistema e ajustes.

Etapa 4 – Testes e avaliação do jogo.

1. Etapa alfa. Consiste de um período de testes com um grupo seleto de pessoas.
2. Etapa Beta. Etapa de testes aberta e já próxima do lançamento final.
3. Lançamento.

1.5 Cronograma

O cronograma é apresentado segundo formatos e exemplos do livro Manual de produções de jogos digitais (CHANDLER, 2012):

Passo	Tempo estimado	Início estimado	Final estimado	Tarefa	Completada? [Y/N]
Definir as mecânicas e características principais do jogo	1 - 2 semanas	ASD	ASD	Todas as principais características e essência definidas.	
Validar tecnologias	1 semana	ASD	ASD	Selecionar todas as tecnologias que serão utilizadas.	
Definir ferramentas e o pipeline	1 semana	ASD	ASD	Definir o pipeline de construção	
Artes conceituais	ASD	ASD	ASD	Gerar uma arte conceitual para os principais personagens e ambientes do jogo.	
Documentação de design	3 - 4 semanas	ASD	ASD	Documentar as principais mecânicas do jogo, incluir protótipo onde possível.	
Documentação artística	ASD	ASD	ASD	Documentar o "look and feel" artístico, lista de assets necessários.	
Documentação técnica	3 - 4 semanas	ASD	ASD	Documentar os padrões de código, design técnico e ferramentas do jogo.	
Desenvolvimento do módulo gráfico	3 - 4 semanas	ASD	ASD	Desenvolvimento do módulo gráfico: 2D, animação, normal map, height map, iluminação etc	
Desenvolvimento do módulo físico	6 - 8 semanas	ASD	ASD	Desenvolvimento do módulo físico: colisões AABB entre retângulos e outras formas geométricas	
Desenvolvimento do módulo de aúdio	2 semanas	ASD	ASD		
Desenvolvimento das interfaces e ferramentas de auxílio	6 - 8 semanas	ASD	ASD	Desenvolvimento de ferramentaas para criação de mapa, importações etc	
Desenvolvimento das artes finais	ASD	ASD	ASD	Produção dos assets finais	
Testes e ajustes	ASD	ASD	ASD	Fase de testes, corrigir bugs da versão final estável	

Figura 1 – Tabela com uma estimativa de tempo para cada passo do desenvolvimento

2 Revisão Bibliográfica

2.1 Referencial Teórico

2.1.1 OpenGL

A API (Application Programming Interface) OpenGL fornece um conjunto de funções para manipulações gráficas (VRIES, 2015), sendo acelerada diretamente em hardware, multiplataforma, robusta e totalmente escalável. 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).

2.1.2 Engenharia de software

A engenharia de software em um jogo, para ser bem sucedida, precisa respeitar as etapas de pré-produção, produção, testes (ou Quality Assurance) e pós-produção [Manual de produção de jogos digitais – Pg. 3]. A metodologia de desenvolvimento a ser utilizada neste trabalho será o SCRUM.

2.1.3 Pré-Produção

A etapa de pré-produção é crítica e determina como será o jogo, quanto tempo levará o desenvolvimento, quantas pessoas serão necessárias e quanto irá custar tudo. Geralmente consome de 10 a 25% do tempo total do desenvolvimento (CHANDLER, 2012). É nessa etapa que se elabora o Game Design Document (GDD) contendo todo o conceito do jogo e requisitos do projeto.

2.1.4 Produção

Durante a produção será elaborado os assets e código do jogo. É nela que ocorrem a criação do conteúdo propriamente dito e o rastreamento do progresso e conclusão de tarefas(CHANDLER, 2012). Para equipes pequenas é interessante metodologias ágeis que focam na produção invés de documentação.

2.1.5 Testes

Em jogos há duas grandes fases para verificar se tudo está funcionando como o esperado: Alfa e Beta. Durante todo o processo é necessário uma equipe do Departamento de Qualidade verificando bugs e reportando-os. Entretanto, as fases Alfa e Beta são as mais importantes. A fase alfa é quando uma seleta quantidade de usuários é escolhida para testar o jogo e dar feedback à desenvolvedora. Ela é fundamental para garantir que o jogo funciona como esperado e quais aspectos precisam ser melhorados antes de ser distribuída para o público geral. A etapa seguinte, Beta, é geralmente aberta ao público e já possui boa parte dos bugs corrigidos. Ela é essencial para testar a recepção do público e fazer as correções finais antes do lançamento oficial.

2.1.6 Game Design Document

Como em todo software o jogo também possui um documento de requisitos. Entretanto, um documento de requisitos não é suficiente para detalhar todos os elementos que compõe um jogo. Essa carência de especificações como história, personagens, roteiro, câmera entre outras coisas são supridas pelo GDD. Portanto, no caso especial de um jogo é necessário ambos documentos para detalhar e especificar adequadamente o projeto. Entretanto, não há regra universal ou normas que ditem como exatamente um GDD deve ser construído ou quais conteúdos deve abranger. A estrutura de um GDD é, possivelmente, composta dos seguintes itens (ROGERS, 2016):

Objetivos de jogo – Detalha o conceito geral do jogo.

Visão geral da história – Constitui um breve resumo da história. Deve entrelaçar os diversos elementos narrativos como ambientes e personagens e conter o início, meio e fim da narrativa.

Controles do jogo – Lista de movimentos que o jogador poderá realizar como ataques, rolamento e corrida. Deve mapear cada botão do controle com a ação a ser realizada.

Exigências de tecnologia – Ferramentas que serão utilizadas ou implementadas para design de níveis, câmeras, engine, física etc.

Front end do jogo – Indica quais telas de crédito serão mostradas quando o jogo é ligado pela primeira vez incluindo:

- Distribuidor

- Logo do estúdio
- Licenciadores
- Produtores de software terceirizados
- Tela com legislação

Tela de título/início - Deve conter em detalhes o que é apresentado ao jogador nas telas iniciais, de preferência com imagens, como por exemplo: Título e como ele aparece na tela, tela de configurações (vídeo, áudio, música, subtítulos etc.), lista de detalhes dos arquivos salvos etc.
Outras telas - Todas as outras telas que não as de início. Por exemplo: Créditos, Conteúdos desbloqueáveis, easter eggs, roupas e armas alternativas etc.
Fluxo de jogo - Demonstra através de um fluxograma como todas as telas interagem entre si.
Câmeras(s) de jogo - Deve conter todos os tipos de câmeras utilizadas, sejam primeira pessoa, terceira pessoal, de rolagem etc.
Sistema de HUD - Informações apresentadas em tela para o jogador. Deve conter imagens demonstrados aspectos como: Saúde, Vidas, Dinheiro, Mini Mapa, Sistema de mira, Sistema de navegação, habilidades etc.
Personagem do jogador - Alguns conceitos visuais e textuais de quem é o personagem que o jogador irá controlar.
Métricas do jogador - Relações de tamanho do personagem do jogador com outros/elementos no mundo:

- Movimento (caminhada, corrida, movimento furtivo, mergulho,rolagem, rastejada)
- Navegação (nado, pulo, voo)
- Pendurar/Balançar
- Movimentos sensíveis ao contexto (empurrar/puxar, iterações com objetos etc.)
- Reações/danos/morte

Habilidades do jogador - Descrição de cada uma das habilidades e seus upgrades, modificadores e métricas. **Mecânicas Universais de jogo** - Descrição breve de como cada uma das mecânicas que compõe o jogo funciona. **Pontuação** - Métricas de como o jogador é recompensado. Placares de liderança e Achievements. **Economia** - Sistema monetário e o que é possível obter com a moeda. Lista custos. **Veículos** - Como funcionam os veículos. Como interagir, suas métricas e controles. **Personagens relevantes na história** - Um breve resumo de cada personagem importante para a história e o seu

papel. Mostrar imagens. **Esboço da progressão do jogo** - Mostrar em um esboço textual ou visual como seria jogar o jogo do começo ao fim. Deixe explicito como o jogador é recompensado conforme progride na história. **Regras gerais dos inimigos** - Listar:

- Tipos de comportamento (patrulheiro, caçador etc.)
- Regras de IA e métricas de detecção
- Parâmetros de nascimento
- Parâmetros de derrota etc.

Personagens não jogáveis (NPCs) - Descrição breve dos tipos de NPC e listar possíveis características como nomes, onde encontrá-los, história etc. Determinar também as interações possíveis.
MiniGames - Lista quais minigames estão presentes bem como seus controles e onde aparecem.
Cenas de corte - Listar as cenas de corte.
Músicas e efeitos especiais - Listar as músicas e efeitos juntamente com seu tom/clima e onde aparecem.
Apêndices - Lista de animações do jogador, Lista de animações dos NPCs e inimigos, Lista dos efeitos sonoros, Lista de músicas, Roteiros etc.

3 Trabalhos Correlatos

No Place for bravery: Jogo estilo Roguelike em pixel art. Atualmente em desenvolvimento pelo estúdio Glitch Factory situado no Distrito Federal.

Eitr: Jogo estilo Action RPG inspirado em Dark souls com uma temática sombria e em pixel art. Atualmente em desenvolvimento pelo estúdio Devolver Digital.

Kingdom: Jogo de plataforma feito em pixel art.

Moon Hunters: Action RPG em pixel art com mundo extenso e procedural, muito similar à proposta deste trabalho em alguns aspectos centrais.

Children of morta: Action RPG em pixel art.

4 Arquitetura de uma *Game Engine*

Game engine (em português literal, 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 em muitas características 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 2 ilustra um típico diagrama da arquitetura deste sistema e seus componentes.

Alguns exemplos muito populares de engines disponíveis no mercado são a Unity, Unreal Engine, Godot, Construct 2 entre muitas outras. Cada uma apresenta sua própria estrutura e especificidade. Algumas são voltadas para projetos 3D e 2D, outras são otimizadas especificamente para um ambiente 2D e outras 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.

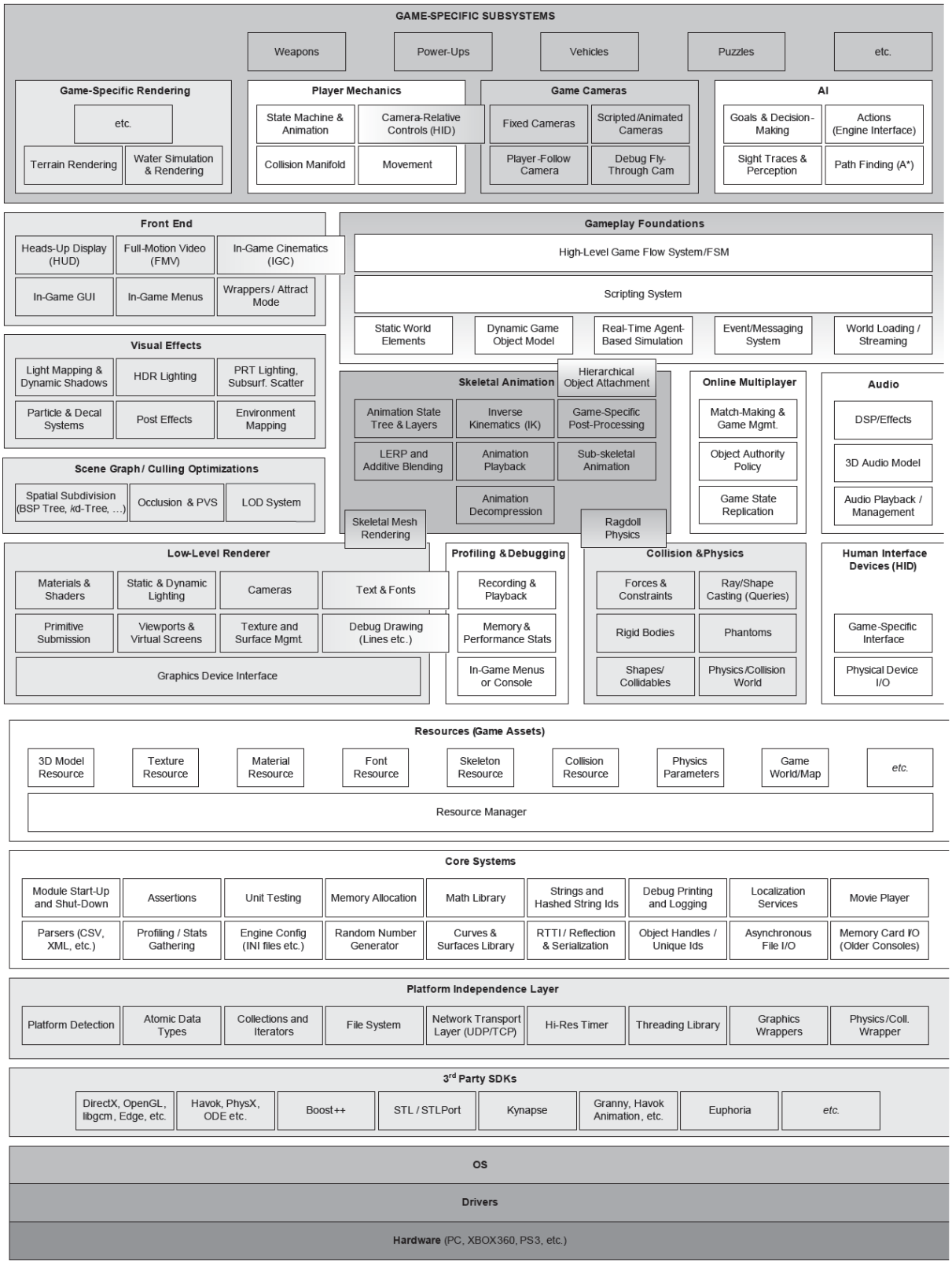

Figura 2 – Componentes de uma Game Engine. Fonte: (GREGORY, 2009)

4.1 *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 composta como se segue:

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

Algoritmo 4.1 – Estrutura básica do Game Loop

O algoritmo 4.1 sendo executado ao longo do tempo é representado pela Figura 3. 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`.

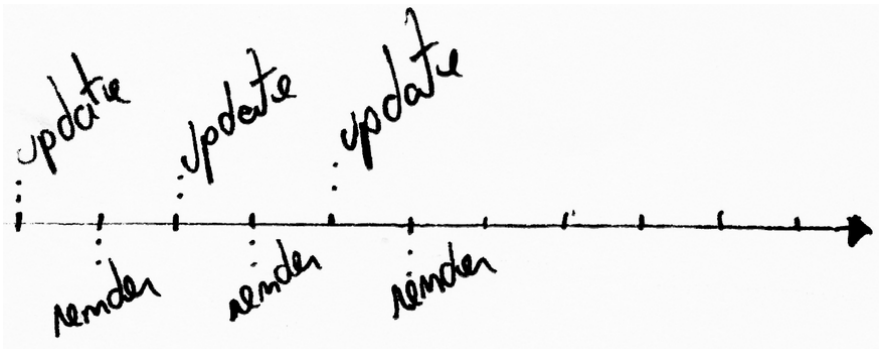

Figura 3 – Execução do game loop básico ao longo do tempo

4.2 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.

4.2.1 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 seria:

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

Algoritmo 4.2 – Game Loop com timestep fixo

E pode ser representado pela figura:

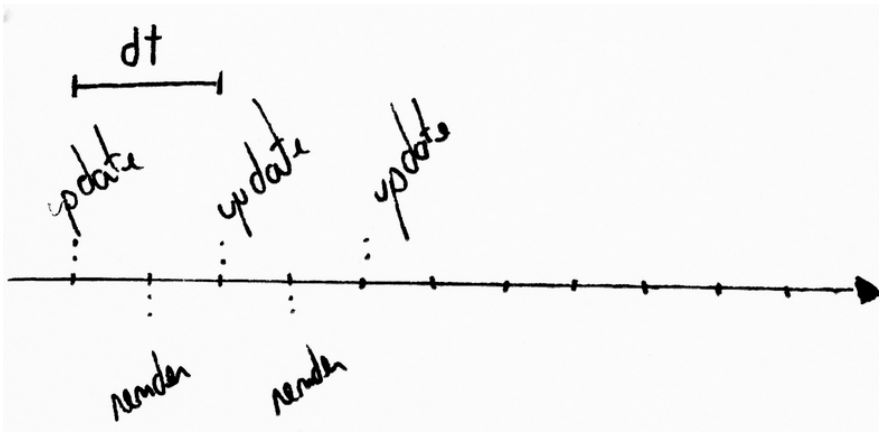

Figura 4 – Execução do game loop com timestep fixo ao longo do tempo

4.2.2 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 como se segue:

```
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 }
```

Algoritmo 4.3 – Game loop com timestep variável

Esse código é representado pela figura:

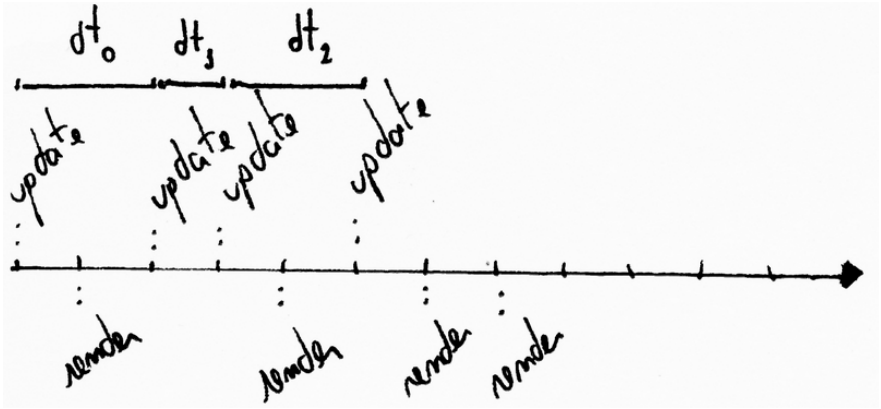

Figura 5 – Execução do game loop com timestep variável ao longo do tempo

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 $10 pixels$, passa a ser $10px + atraso$. Isso gera um efeito chamado de *stuttering* e é percetível 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.

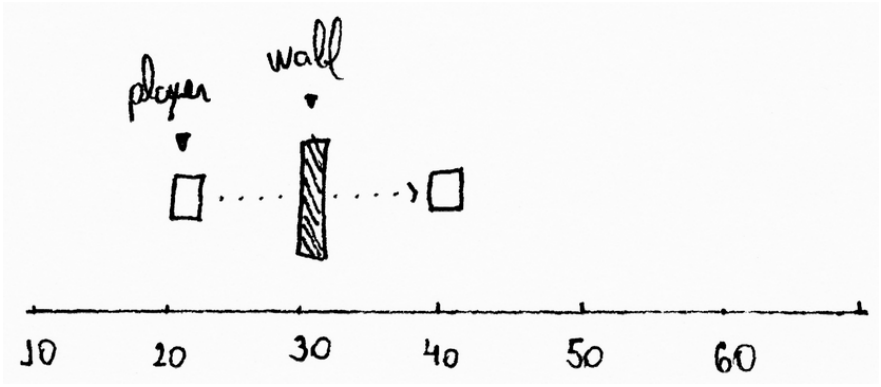

Figura 6 – Objeto ignorando colisão devido um timestep muito grande

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.

4.2.3 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 quando demonstrado em código:

```
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 }
```

Algoritmo 4.4 – 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 ilustração:

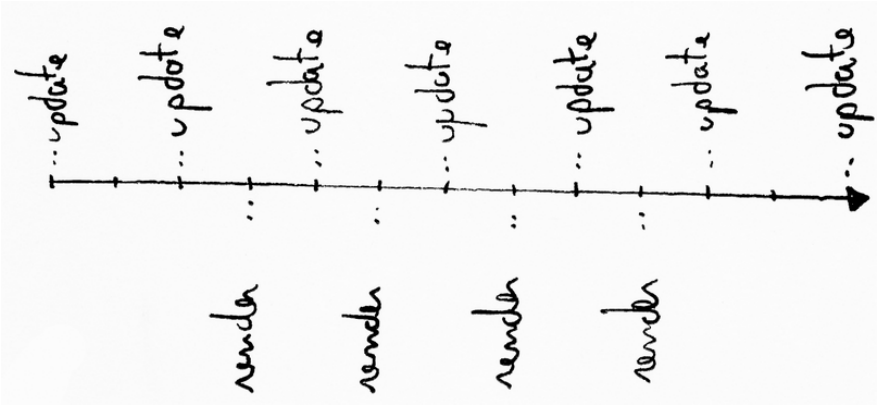

Figura 7 – Execução do game loop com timestep semi-fixo ao longo do tempo

O problema em que o processamento do método update é maior que o Δt é mos-

trado 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.

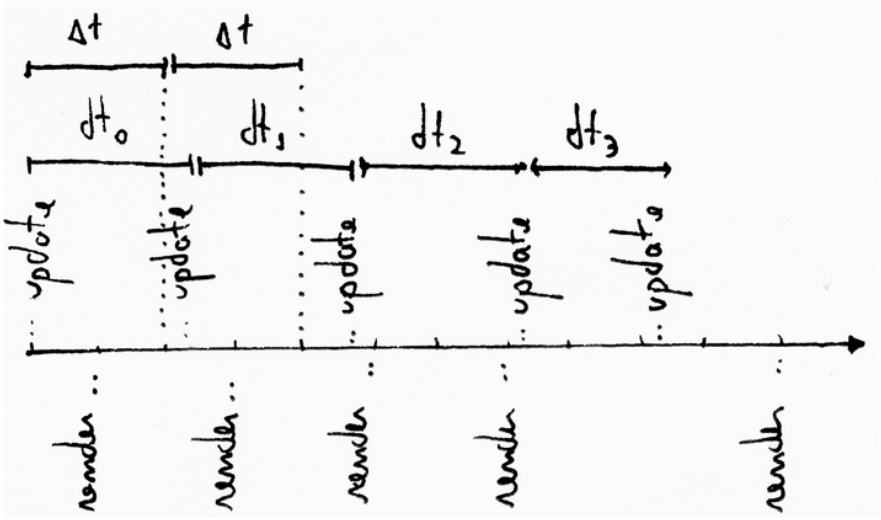

Figura 8 – Método `update` com tempo de processamento maior que o Δt

4.3 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 durante 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).

4.3.1 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 9 e 10 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.

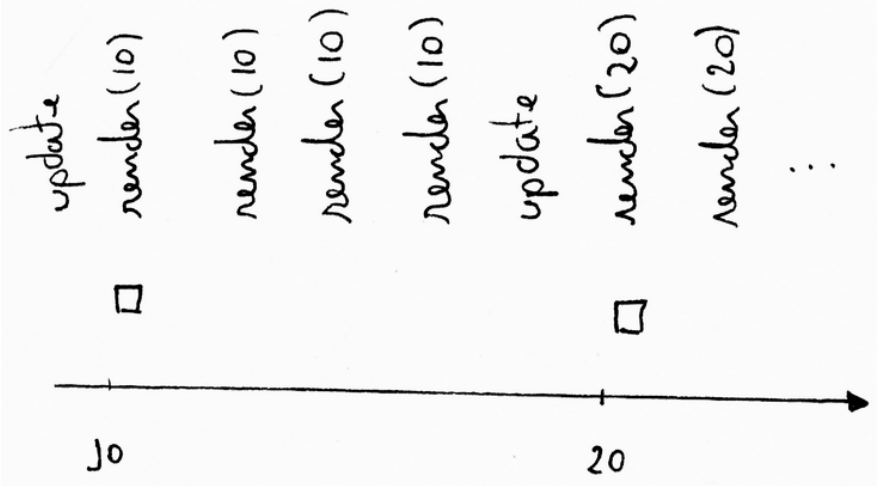

Figura 9 – Renderização sem interpolação

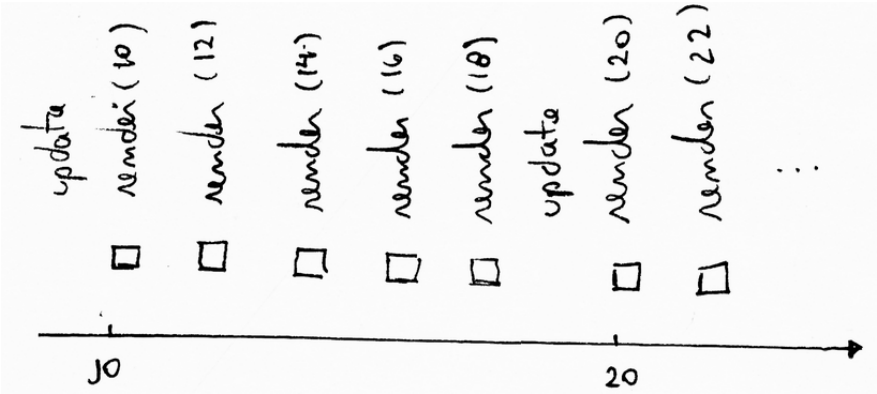

Figura 10 – Renderização sem interpolação

O código final do game loop, para um sistema com timestep semi-fixo e com interpolação linear fica como se segue:

```
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      }
```

Algoritmo 4.5 – 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:

$$renderPosition = currentPosition * interpolationFactor + previousPosition * (1 - interpolationFactor)$$

4.4 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.

4.4.1 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.

4.4.2 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.

4.4.3 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 a seguir 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); //VSYNC
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 }
```

Algoritmo 4.6 – 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.

4.4.4 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 apenas 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. A seguir tem-se uma representação simplificada do processo que ocorre no pipeline para desenhar um pequeno triângulo.

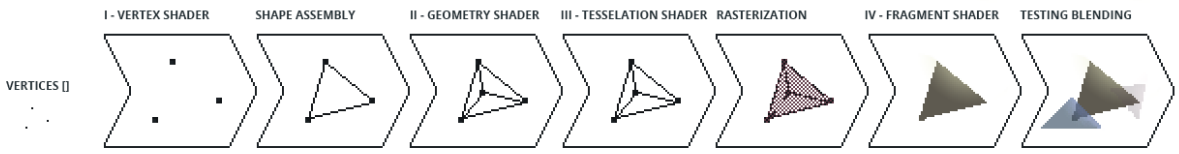

Figura 11 – Etapas do pipeline

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 11 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 11 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*.

4.4.5 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 `glBufferData(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)`.

```
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)
          ), GL_STATIC_DRAW);
18
19    glBindVertexArray(quadVAO);
20    glEnableVertexAttribArray(0);
21    glVertexAttribPointer(0, 4, GL_FLOAT, false, Float.BYTES * 4, 0);
22
23    glBindBuffer(GL_ARRAY_BUFFER, 0);
24    glBindVertexArray(0);
25 }
```

Algoritmo 4.7 – 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.

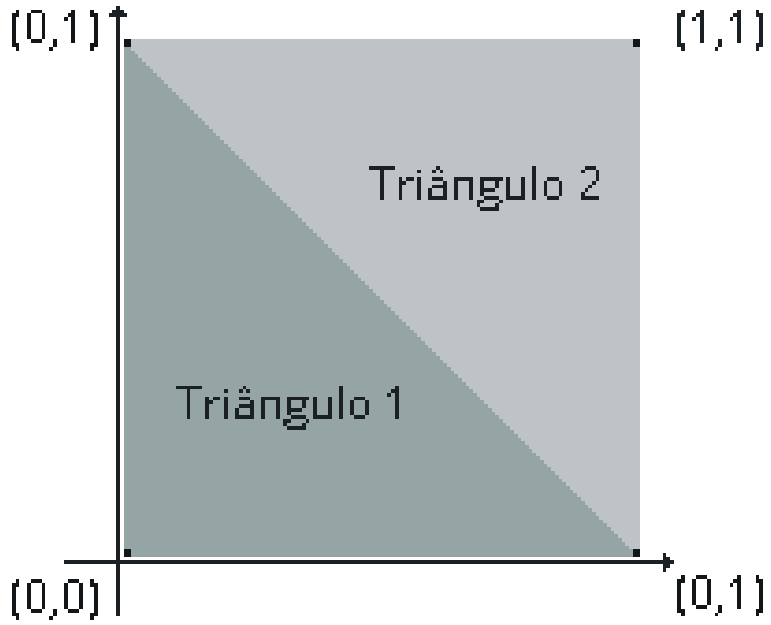

Figura 12 – Dois triângulos em coordenadas locais normalizadas que juntos formam um quadrado

4.4.6 OpenGL Shading Language (GLSL)

???

4.4.7 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 11, 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. Dessa forma a primeira parte do código fica:

```
1 #version 330 core
2 layout (location = 0) in vec4 vertex; //xy Obj coord
3                       //zw Texture coord
```

Algoritmo 4.8 – Vertex shader header

Definido o cabeçalho agora 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 pela manipulação destes dados.

```
1  #version 330 core
2  layout (location = 0) in vec4 vertex; //xy position
3                       //zw Tex coord
4
5  out vec2 TexCoords;
6
7  uniform mat4 model;
8  uniform mat4 projection;
9  uniform mat4 camera;
10
11 uniform vec2 flip;
12
13 void main(){
14   TexCoords = vertex.zw;
15   gl_Position = projection * camera * model *  vec4(vertex.xy, 0.0 , 1.0);
16 }
```

Algoritmo 4.9 – 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 `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 do 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 (linha 5). Esse resultado é então enviado para a última etapa usando-se a variável `out vec4 color`.

```
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 }
```

Algoritmo 4.10 – Fragment shader simples

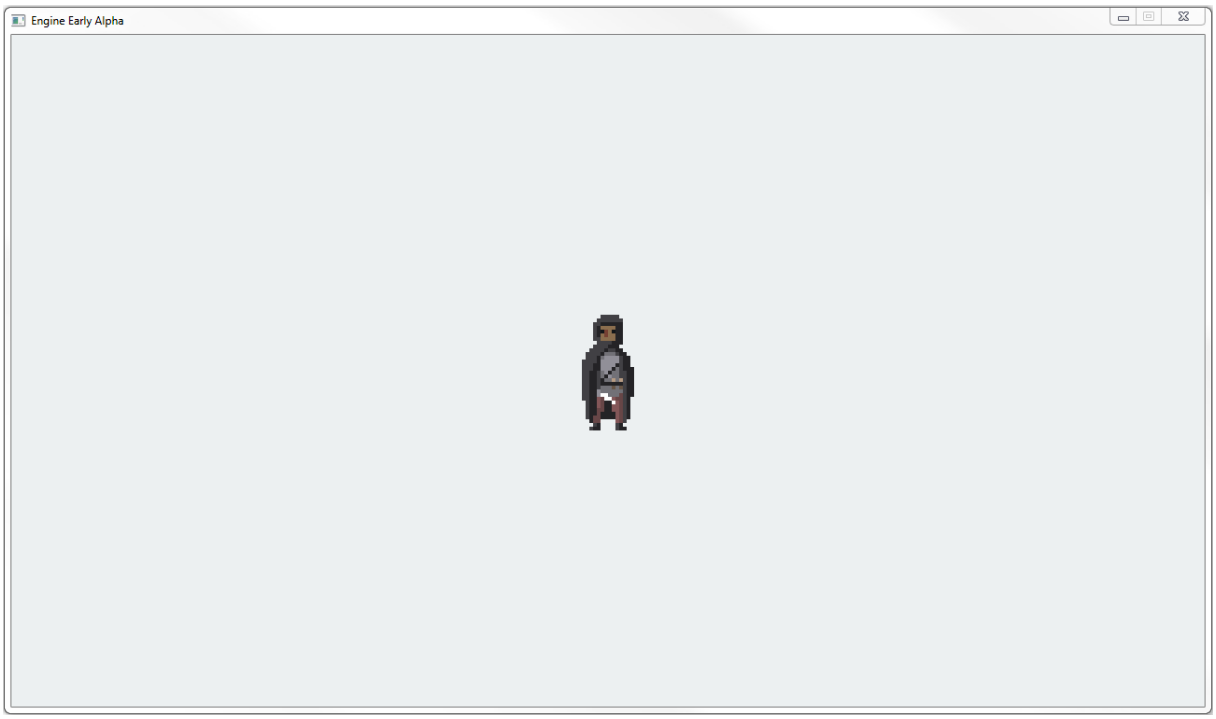

Figura 13 – Resultado obtido com dois shaders simples

4.4.8 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 11. 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 fun-

ção `glGetShaderInfoLog(id)` retorna a mensagem de erro, que pode ser posteriormente impressa na tela.

```
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.err.println(log);
10  }
```

Algoritmo 4.11 – 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.

```
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.err.println(log);
11  }
```

Algoritmo 4.12 – Processo de criação e link de um programa

Os shaders uma vez encadeados podem ser removidos para liberar espaço usado a função `glDeleteShader(id)`.

4.4.9 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 váriavel 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 4 por 4 `glUniformMatrix4fv(glGetUnif`

`(programID,variableName), false, BufferUtilities.createFloatBuffer(m))` e assim por diante. Dessa forma a classe final do Shader fica:

```
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   /**
13    * Compile all vertex, fragment and geometry source code into a linked program
          .
14    * Geometry is optional.
15    *
16    * @param vertexSource
17    * @param fragmentSource
18    * @param geometrySource
19    */
20   public void compile(String vertexSource, String fragmentSource, String
         geometrySource) {
21     int vertexShader, geometryShader = 0, fragmentShader;
22
23     vertexShader = glCreateShader(GL_VERTEX_SHADER);
24     glShaderSource(vertexShader, vertexSource);
25     glCompileShader(vertexShader);
26     checkCompileErrors(vertexShader, "VERTEX");
27
28     fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
29     glShaderSource(fragmentShader, fragmentSource);
30     glCompileShader(fragmentShader);
31     checkCompileErrors(fragmentShader, "FRAGMENT");
32
33     if(geometrySource!=null) {
34       geometryShader = glCreateShader(GL_GEOMETRY_SHADER);
35       glShaderSource(geometryShader, geometrySource);
36       glCompileShader(geometryShader);
37       checkCompileErrors(fragmentShader, "GEOMETRY");
38     }
39
40     id = glCreateProgram();
41     glAttachShader(id, vertexShader);
42     if(geometrySource!=null)
43       glAttachShader(id, geometryShader);
44     glAttachShader(id, fragmentShader);
45     glLinkProgram(id);
46     checkCompileErrors(id, "PROGRAM");
47
48     glDeleteShader(vertexShader);
49     glDeleteShader(fragmentShader);
50     if(geometrySource!=null)
51       glDeleteShader(geometryShader);
52   }
```

```
53
54    private void checkCompileErrors(int object, String type) {
55      int success;
56      String log;
57
58      if(type!="PROGRAM") {
59        success = glGetShaderi(object, GL_COMPILE_STATUS);
60
61        if(success==0) {
62          log = glGetShaderInfoLog(object);
63          System.err.println(log);
64        }
65
66      }else {
67        success = glGetProgrami(object, GL_LINK_STATUS);
68
69        if(success==0) {
70          log = glGetProgramInfoLog(object);
71          System.err.println(log);
72        }
73
74      }
75    }
76
77    public void setFloat(String name, float value) {
78      glUniform1f(glGetUniformLocation(id,name), value);
79    }
80
81    public void setInteger(String name, int value) {
82      glUniform1i(glGetUniformLocation(id,name), value);
83    }
84
85    public void setVec2(String name, float x, float y) {
86      glUniform2f(glGetUniformLocation(id,name), x,y);
87    }
88
89    public void setVec2(String name, Vec2 pos) {
90      glUniform2f(glGetUniformLocation(id,name), pos.x, pos.y);
91    }
92
93    public void setVec3(String name, float x, float y, float z) {
94      glUniform3f(glGetUniformLocation(id,name), x,y,z);
95    }
96
97    public void setVec3(String name, Vec3 pos) {
98      glUniform3f(glGetUniformLocation(id,name), pos.x, pos.y, pos.z);
99    }
100
101   public void setVec4(String name, float x, float y, float z, float w) {
102     glUniform4f(glGetUniformLocation(id,name), x, y, z, w);
103   }
104
105   public void setVec4(String name, Vec4 pos) {
106     glUniform4f(glGetUniformLocation(id,name), pos.x, pos.y, pos.z, pos.w);
107   }
108
109   public void setMat4(String name, Mat4 m) {
```

```
110     glUniformMatrix4fv(glGetUniformLocation(id,name), false, BufferUtilities.
            createFloatBuffer(m));
111   }
112 }
```

Algoritmo 4.13 – Shader class

4.5 Renderer

4.5.1 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.

```
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(),
           pixels, 0, textureImage.getWidth());
24         ByteBuffer buffer = BufferUtilities.createByteBuffer(
25                       new byte[textureImage.getWidth() * textureImage.getHeight
                              () * BYTES_PER_PIXEL]
26                     ); //4 for RGBA, 3 for RGB
27
28         for(int y = 0; y < textureImage.getHeight(); y++){
29             for(int x = 0; x < textureImage.getWidth(); x++){
30                 int pixel = pixels[y * textureImage.getWidth() + x];
31
32                 buffer.put((byte) ((pixel >> 16) & 0xFF));        // Red
                       component
```

```
33                  buffer.put((byte) ((pixel >> 8) & 0xFF));          // Green
                        component
34                  buffer.put((byte) (pixel & 0xFF));                 // Blue
                        component
35                  buffer.put((byte) ((pixel >> 24) & 0xFF));         // Alpha
                        component. Only for RGBA
36              }
37          }
38
39          buffer.flip();
40
41          glBindTexture(GL_TEXTURE_2D, id); //Bind texture ID
42
43          //Setup wrap mode
44          glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL12.GL_CLAMP_TO_EDGE)
                ;
45          glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL12.GL_CLAMP_TO_EDGE)
                ;
46
47          //Setup texture scaling filtering
48          glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST); //
                GL_LINEAR for smooth
49          glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
50
51          //Send texel data to OpenGL
52          glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, textureImage.getWidth(),
                textureImage.getHeight(), 0, GL_RGBA, GL_UNSIGNED_BYTE, buffer);
53
54          glBindTexture(GL_TEXTURE_2D, 0); //Unbind texture
55          textureImage = null; //'free' BufferedImage
56    }
57  }
```

Algoritmo 4.14 – 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()`.

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

Algoritmo 4.15 – Função auxiliar createByteBuffer

Com o byte buffer alocado a próxima etapa é preenchê-lo com os dados contidos no array `pixels`. O formato escolhido para essa engine 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.

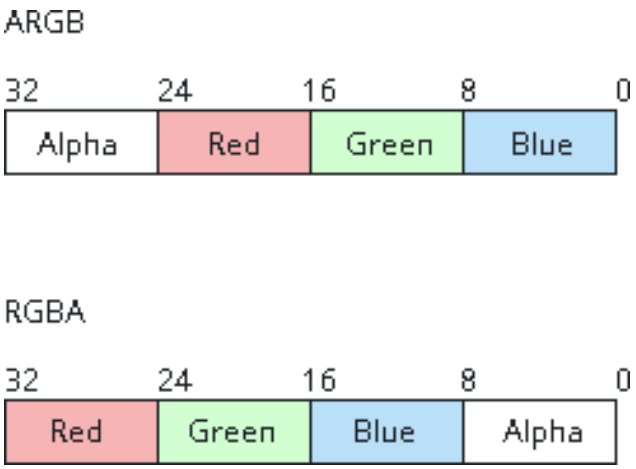

Figura 14 – Sistema ARGB e RGBA, ambos com canais de 8 bits (1 byte)

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 a seguir.

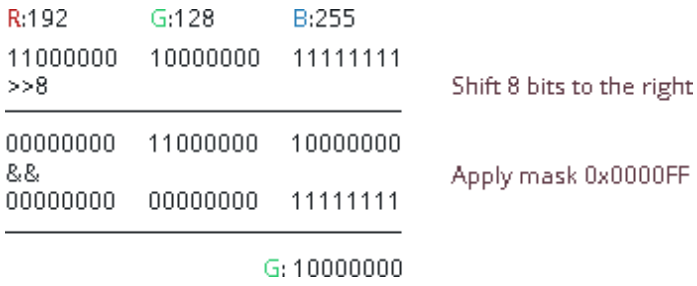

Figura 15 – Extração do canal verde em um inteiro de 3 bytes

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`.

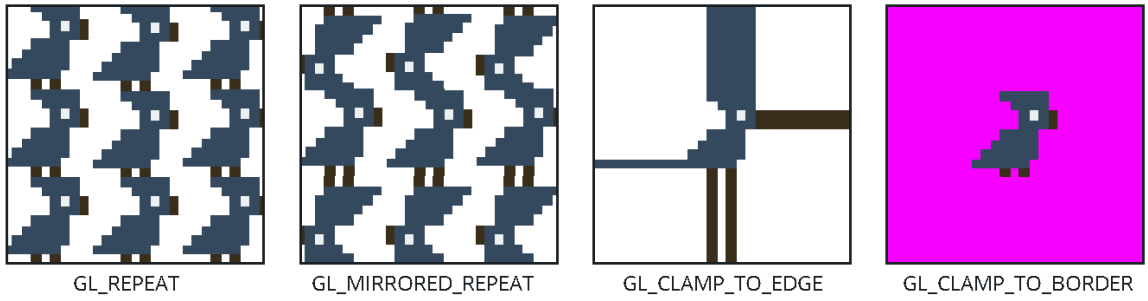

Figura 16 – Wrap modes

Esses modos são configurados através da função `glTexParameteri(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 escalonar a imagem. São dois modos possívei: `GL_NEAREST` e `GL_LINEAR`.

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. A Figura 17 demonstra esse processo.

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 17 – 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, bytebuffe`

4.5.2 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.

Figura 18 – 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 `glAttribXXXX` 0 definido anteriormente.

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

```
23 }
```

Algoritmo 4.16 – Vertex shader com animações

Na variável `spriteFrame` tem-se as coordenadas de qual pedaço da textura deverá ser extraído. 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 fica como se segue. 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 correspondete 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.MILISECOND;
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  }
```

Algoritmo 4.17 – Classe Animation

4.5.3 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 simplesmente ordenar os elementos pela posição y do menor para o maior.

Figura 19 – Ordem correta de renderização: do menor y para o maior

Um ponto a se tomar cuidado na implementação desta técnica é ordenar apenas os objetos que estão no espaço de visão da câmera, otimizando assim a performance do algoritmo de ordenação.

4.5.4 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 de Phong

(SCRATCHAPIXEL, 2018) que usa três componentes de luz em sua construção: ambient, diffuse e specular.

Figura 20 – Componentes do modelo de Phong

4.5.4.1 Ambiente

A luz ambiente representa as fontes de luz mais distantes que estão quase sempre presentes no mundo 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.

```
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(0.1,0,0,1);
9
10   vec4 imgTex =  texture(image, TexCoords).xyzw;
11
12   color = imgTex*lightAmbientColor;
13 }
```

Algoritmo 4.18 – Fragment Shader com luz ambiente

Dessa forma o resultado que se obtêm com a cor acima é um ambiente com uma tonalidade um pouco avermelhada:

//TODO: add sunset image

4.5.4.2 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 brilhante 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 entre os dois vetores usando o produto escalar. //TODO: inserir imagem do produto escalar

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 e 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`.

//TODO: referência vetor normal ou footnote (whatever) //TODO: adicionar o código do shader com luz difusa

4.5.4.3 Vetor Normal

Um vetor normal unitário é aquele perpendicular à superfície do vértice. A técnica comumente utilizada para trabalhar com os vetores normais de um objeto consiste em mapeá-los em uma textura de forma que, para cada pixel da textura original, haja um vetor normal. Como o vetor normal precisa de apenas 3 coordenadas (x,y,z) isso é facilmente convertido para as coordenadas de cores RGB e, por fim, são armazenadas em uma imagem/textura.

//TODO: demonstrar uma normal texture

//TODO: demonstrar um vetor normal

4.5.4.4 Luz Especular

A luz especular é o ponto brilhante que aparece em objetos reluzentes e ajuda a dar uma melhor percepção do espaço 3D.

//TODO: imagem do olho e o reflexo especular em uma superfície

4.5.5 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, texture e afins do objeto. Dessa forma, o código 4.7 será incorporado em uma classe renderer juntamente com o método render descrito a seguir:

```
1  public class TextureRenderer(){
2
3    public TextureRenderer(){
4      init();
5    }
6
7    private void init(){
8      //code from algorithm 4.7
9    }
10
11   public void render(Texture texture, Vec2 position, Vec2 size, float rotate,
         Vec4 color, Vec4 spriteFrame) {
12     this.shader.use();
13     Mat4 model = new Mat4();
14
15     model = model.translate(position.x, position.y, 0);
16     model = model.translate(0.5f * size.x, 0.5f *size.y, 0); //Move the origin
           of rotation to object's center
17     model = model.rotate(rotate, 0, 0, 1); // Must be in radians
18     model = model.translate(-0.5f * size.x, -0.5f *size.y, 0); //Move the origin
           of rotation back to it's top left
19     model = model.scale(size.x, size.y, 1);
20
21     shader.setMat4("model", model);
22     shader.setVec4("spriteColor", color);
23     shader.setVec2("flip", orientation);
24     shader.setVec4("spriteFrame", spriteFrame);
25
26     shader.setInteger("image", 0);
27
28     glActiveTexture(GL_TEXTURE0);
29     texture.bind();
30
31     glBindVertexArray(quadVAO);
32     glDrawArrays(GL_TRIANGLES, 0, 6*3);
33     glBindVertexArray(0);
34   }
35 }
```

Algoritmo 4.19 – Classe renderer simples

Dessa forma, para renderizar os objetos itera-se numa lista e para cada objeto chama-se TextureRenderer.render().

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

Algoritmo 4.20 – Demonstração do render

Entretanto, esta implementação é ingênua pois é 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 abstração do processo de renderização abordou-se o modelo acima. Entretanto, o problema será tratado em uma sessão mais adiante que implementará o método de batch rendering, que resolve esta situação realizando apenas uma única chamada do método render por tipo de shader e realizando a transferência desses dados em um único e enorme pacote de dados, evitando gargalo no BUS da placa mãe.

//TODO: atualizar seção e subseção para Arquitetura do GO

4.5.6 Arquitetura do Game Object

Os game objects, 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).

4.5.6.1 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 das vezes usados apenas por uma certa 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.

//TODO: colocar uma imagem UML demonstrando essa especialização.

4.5.6.2 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.

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 projeto com componentes usando passagem de mensagem. A maior desvantagem dessa arquitetura está na relação estrita entre os dados e o sistema que os manipula, tornando o processamento de um sistema que requer múltiplos dados não diretamente relacionados complicado.

//TODO: falar da leitura simplificada de arquivo tanto no CB quanto EB //-TODO: diagrama uml e cód. exemplo dos switches

4.5.7 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.

//TODO: img e cód.

4.6 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 principais: 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 etc.

4.6.1 Colisões do tipo AABB vs AABB

Colisões do tipo Axis Aligned Bounding Box (AABB) são as mais simples de resolver. Para detectar se dois retângulos de eixo alinhado colidiram basta checar se houve sobreposição em qualquer um dos eixos x e y. //TODO: explicar melhor eixo alinhado //TODO: explicar melhor vs

```
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    }
```

Algoritmo 4.21 – Colisão AABB vs AABB

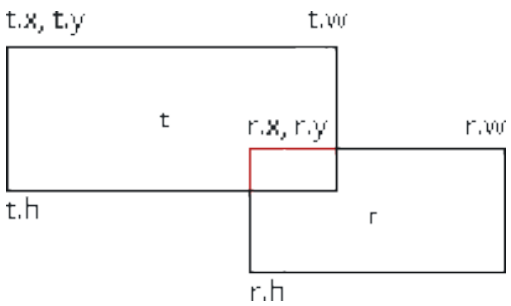

Figura 21 – Colisão AABB vs AABB

Para o exemplo da figura acima 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.

4.6.2 jBox2D

A biblioteca jBox2D é um porte 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. Esse tópico será discutido adiante no capítulo Sistemas de Coordenadas.

Para o sistema de colisões funcionar ele precisa, assim como a própria engine, de um mundo e um timestep, neste caso, fixo. Define-se abaixo o exemplo: //TODO: código exemplo

4.7 Sistema de coordenadas

O sistema de coordenadas abrange todo o processo de manipulação das coordenadas dos game objects até a sua conversão NDC e por fim rasterização para a tela. Note que o OpenGL espera todas as coordenadas finais no formato NDC (Normalized Devide Coordinates), que varia entre $[-1, 1]$. Cada etapa manipula e converte de um sistema para o outro. São cinco sistemas de coordenadas no total:

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

Uma vez processadas elas são passadas para o *shader rasterization* e enfim convertidas para pixels em 2D na tela.

//TODO: imagem de todo processo de transformação de um sistema para outro

4.7.1 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 $(0, 0)$.

//TODO: imagem de um objeto no local space.

4.7.2 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 *model* escala, rotaciona e translada o objeto para o espaço de mundo através do processo chamado transformação afim.

Se todos os objetos fossem inseridos no mundo somente usando as coordenadas locais provavelmente todos 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 com as outras e caber no espaço vazio e, por fim, transladá-la para a posição na pilha.

//TODO: imagem exemplo ilustrando o processo de inserir a caixa.

4.7.3 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.

//TODO: imagem exemplo

4.7.4 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, ou simplesmente, fora do intervalo NDC.

4.7.5 Projeção

Existem duas formas de aplicar a projeção. São elas projeção ortográfica e projeção de perspectiva.

4.7.5.1 Projeção ortográfica

//TODO: frustum - tronco de base paralela

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 retângulo formado pelo plano mais próximo e pelo plano mais distante serão apresentadas ao usuário. Todo o resto será recortado fora. //TODO: imagem do far e near plane

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.

//TODO: imagem do resultado de uma matriz orto

Para especificar a matriz ortográfica será utilizada a função do GLM definida a seguir. 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);
```

Algoritmo 4.22 – Função Ortho do GLM

//TODO: inserir imagem indicando parâmetros. Mais fácil visualizar

4.7.5.2 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. A matriz de projeção pode ser criada utilizando a seguitne função do GLM.

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

Algoritmo 4.23 – Função Perspective do GLM

4.7.6 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.

4.8 Camera

4.9 Sistema de input

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

4.9.1 GLFW Keyboard

4.9.2 GLFW Mouse

4.9.3 GLFW Game Control

4.10 Gerência de recursos

O sistema de gerência de recursos é responsável por ler e salvar todos os arquivos da Engine. Em suma, esse sistema é responsável por centralizar todas as operações de I/O e gerenciá-las num único lugar de forma que seja evitado a ocorrência de dois arquivos idênticos em memória.

4.10.1 Chunk system

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. Outro fator determinante para a escolha deste formato é a necessidade de poder gerar um mapa infinito e com carregamento dinâmico, ou seja, sem telas de loading. Isso é possível repartindo o mapa em pequenos pedaços que serão gerados, lidos e escritos sobre demanda.

É importante notar que este é um sistema crítico e deve ser meticulosamente ajustado para as necessidades de cada projeto. Um arquivo muito grande pode gerar um atraso de leitura e escrita muito grande, 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 cada operação de I/O.

Com estes detalhes abordados a implementação se faz com duas classes principais: Chunk e ChunkManager. //TODO: check class names

4.10.2 A classe Resource Manager

4.11 Aúdio

A implementação de aúdio foi feita utilizando a mesma família de bibliotecas da Khronos, o OpenAL, voltada especificamente para áudio.

4.11.1 Integrando o OpenAL

5 Inteligência artifical

O sistema responsável pela IA da Engine é parte fundamental do sistema. Sem ela o jogo iria ser um conjunto de imagens estáticas distribuídas à esmo na tela. Em essência IA é o que da vida a todos os inimigos e personagens do jogo, tornando-o dinâmico e interativo para o usuário. //TODO: https://www.gamedev.net/articles/programming/artificial-intelligence/the-total-beginners-guide-to-game-ai-r4942/

5.1 Utility AI

Em economia, *utility* é uma medida da satisfação relativa de, ou desejo de, consumo de vários bens e serviços. Dado essa medida, alguém poderá expressar o acréscimo ou queda 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 de onde será extraído os valores que quantificam o ambiente e por fim calculado o seu retorno indicando o utility daquela ação.

//TODO: ref to http://www.intrinsicalgorithm.com/media/2010GDC-DaveMark-KevinDill-UtilityTheory.pdf

5.1.1 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 atribuido a cada dessas ações uma função de pontuação. Ao final, compara-se o valor calculado (utility) das funções e o maior deles representa a ação que será executada.

//TODO: imagem com o gráfico mostrando a interpolação das funções

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

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

- Perseguir: $returndistancia^3$

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 a seguir.

//TODO: gráfico das funções

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 abaixo demonstra três situações em que, para cada conjunto de parâmetros tem-se uma ação que é decidida usando a maior utility.

Vida do agente : Distância até o inimigo :

Atirar Abrigar-se Perserguir 00 00

//TODO: good example https://alastaira.wordpress.com/2013/01/25/at-a-glance-functions-for-modelling-utility-based-game-ai/ //https://www.gamasutra.com/blogs/JakobRasmusse

5.2 Finite State Machine

5.3 Árvore de decisões

5.4 Pathfinding

Existem muitas maneiras de se implementar o algoritmo de pathfindind. Alguns funcionam muito bem para sistemas estáticos onde tem-se conhecimento do mundo e outros nem tanto. Para ambientes dinâmicos não pode-se, por exemplo, assumir waypoints estáticas com caminhos pré-calculados pois o ambiente muda a todo momento.

O processo de pathfinding no ambiente 2D consiste em primeiro mapear uma seção do mapa para uma matriz. Cada

6 Procedural Content Generation

Procedural Content Generation (PCG) é o termo cunhado para descrever processos de geração de dados aleatórios ou pseudo-aleatórios em jogos. Esses dados são utilizados para gerar desde terrenos até modelos 3D de maneira algorítmica invés de manual.

6.1 Random noise

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 dado. //TODO: good source: https://thebookofshaders.com/12/ http://pcgbook.com/

6.1.1 Perlin noise

Perlin noise é um tipo de ruído gradiente concebido por Ken Perlin em 1983 //-TODO: cite. O algoritmo tem inicio definindo um grid de dimensão n. O processo mais básico consiste em definir o grid com uma série de vetores gradiente, computar o produto escalar entre vetores gradiente de distância e então realizar uma interpolação entre esses valores.

6.2 Geração de terrenos

A geração de terreno utilizada nessa engine consiste em um ...

6.2.1 Florestas

7 Criação de um protótipo

7.1 Mecânicas

7.1.1 Sistema de caça ativa

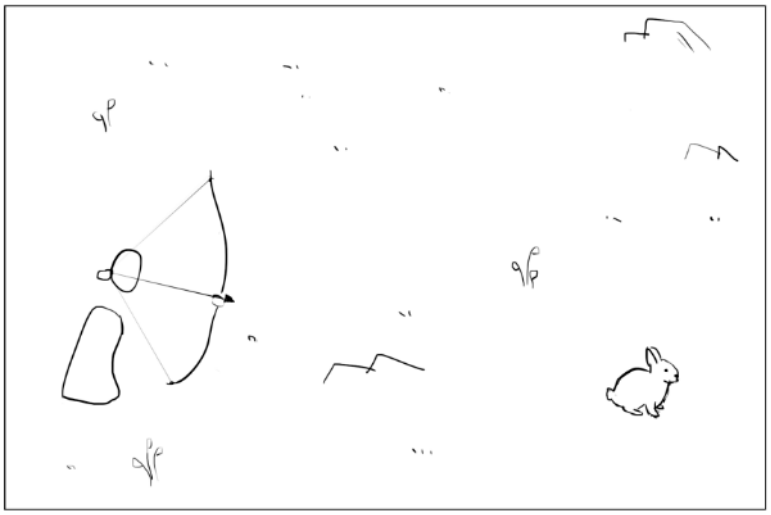

Figura 22 – Jogador caçando um coelho com arco e flecha

O sistema de caça é dado pelo seguinte fluxo de ações:

1. Jogador avista animal em ambiente selvagem
2. Ao se aproximar o animal pode perceber sua presença. Caso seja notado, a presa irá executar uma animação indicando que está desconfiado.
3. Se o jogador continuar avançando o animal irá tentar fugir
4. Senão o jogador executa um ataque quando estiver ao alcance

7.1.2 Sistema de armadilhas

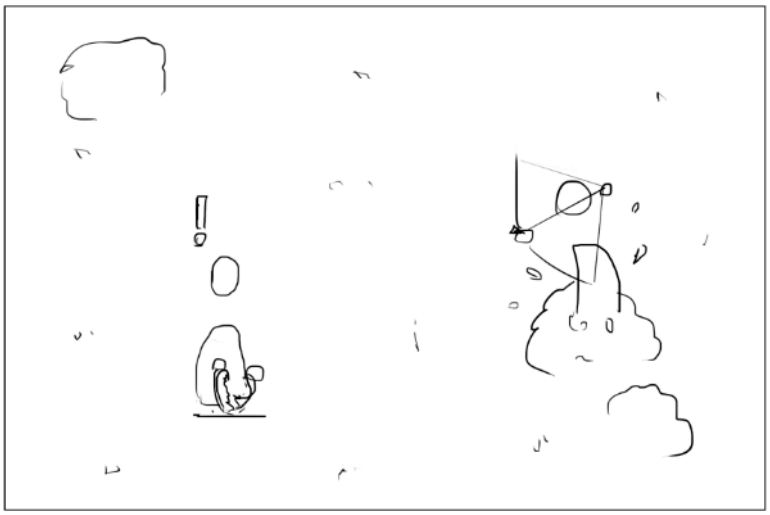

Figura 23 – Jogador pronto para atacar alvo que ficou preso na armadilha de chão

O sistema de armadilhas é dado pelo seguinte fluxo de ações:

1. Jogador aciona armadilha no local desejado
2. Qualquer animal ou inimigo pode acionar a armadilha e acionar o seu efeito

7.1.3 Sistema de Combate

O sistema de combate é delimitado pela arma escolhida. Cada arma implica em um estilo de combate totalmente diferente e só pode ser utilizada ao aprender esse estilo com um mestre. Os mestres podem ser encontrados aleatoriamente pelo mapa em grandes cidades de cada reino.

7.1.4 Sistema de apadrinhamento

Ao atingir um nível significante de reputação o jogador pode tentar ganhar um apadrinhamento de um rei ou lorde. Ganhando assim uma quantia semanal e benefícios como alimento e teto para poder compor suas músicas.

7.1.5 Cantar músicas

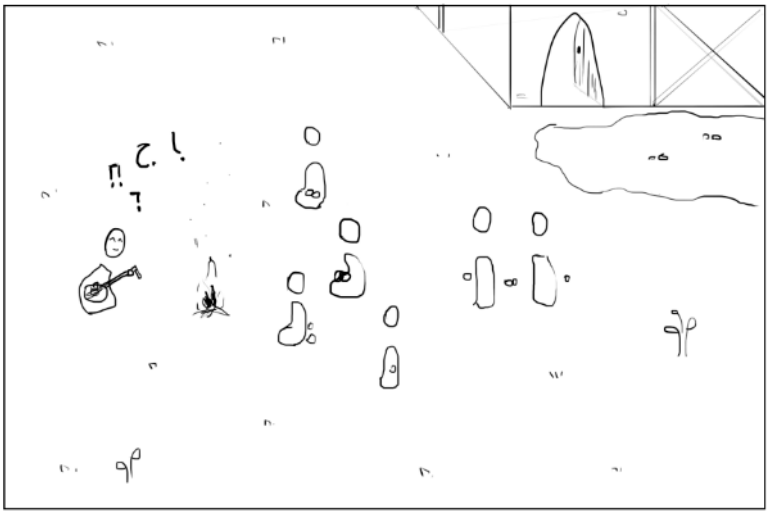

Figura 24 – Jogador cantando em um vilarejo

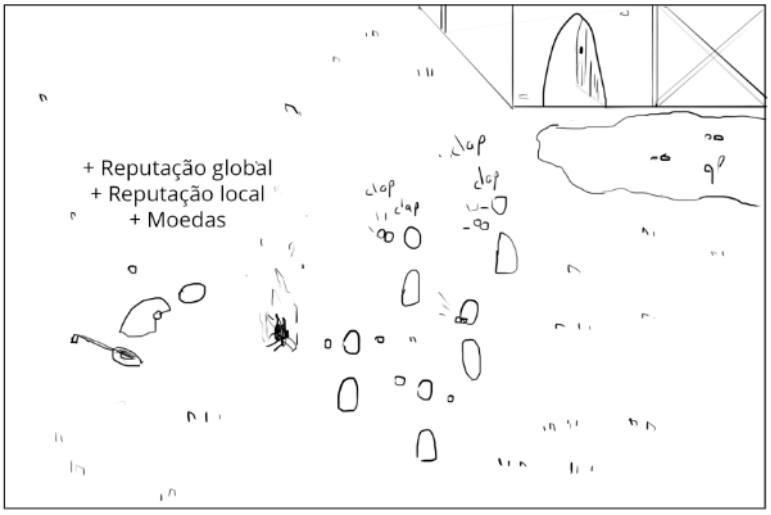

Figura 25 – Jogador sendo prestigiado após término da música

Cada vilarejo ou cidade tem um nível de interesse próprio em determinados instrumentos. Quanto maior o interesse, maior serão as chances de receber uma boa quantia em ouro pela apresentação. As músicas podem ser tocadas em fogueiras que o próprio jogador pode criar em volta da cidade, ou em tavernas, festivais e praças.

7.1.6 Compor músicas

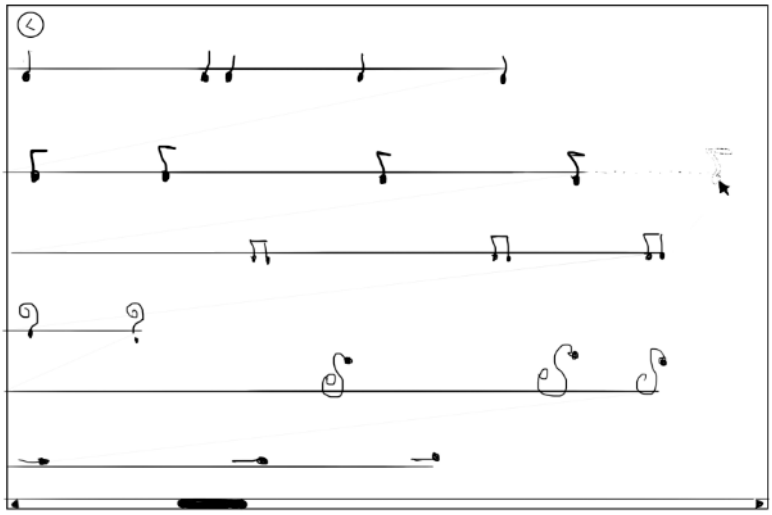

Figura 26 – Esboço do menu para compor músicas

ASD.

7.1.7 Contar histórias

ASD.

7.1.8 Compor histórias

ASD.

7.1.9 Atributos dos vilarejos e cidades

Cada cidade e vilarejo possui os seguintes atributos:

Lista de interesses instrumentais Cada vilarejo tem uma preferência por um instrumento. Quanto maior o interesse, maior a recompensa dada ao tocar músicas com aquele instrumento.

7.1.10 Atributos do jogador

Fome Ao longo do tempo o jogador precisa se alimentar para manter-se vivo. A fome é dividida em três estágios: Sem fome, fome controlável e faminto. Cada um dos estágios é apresentado ao jogador em forma de uma animação de andar diferente. Quanto mais faminto o personagem está mais lento e curvado ele irá andar, até que ela chegue em zero e culmine na morte do personagem.

7.1.11 Domar animais

Alguns animais podem ser domados através dos instrumentos. Basta cantar próximo a eles então eles serão domados.

8 Experimentos e resultados

8.1 Protótipo

8.2 Otimizações e testes de desempenho

9 Conclusão

Referências

CHANDLER, H. M. *Manual de Produção de Jogos Digitais*. 9. ed. Bookman, 2012. ISBN 9788540701847. Disponível em: <http://books.google.com.br/books?id=Ifdu3B9C7jEC>. Citado 3 vezes nas páginas 13, 14 e 15.

GREGORY, J. *Game Engine Architecture*. [S.l.]: Taylor and Francis Group, 2009. ISBN 9781439865262. Citado 5 vezes nas páginas 4, 20, 21, 22 e 26.

KHRONOS. *OpenGL Specifications*. 2017. Disponível em: <https://www.khronos.org/registry/OpenGL/specs/gl/>. Citado na página 14.

KHRONOS. *History of OpenGL*. 2018. Disponível em: <https://www.khronos.org/opengl/wiki/History_of_OpenGL#OpenGL_3.0_.282008.29>. Citado na página 28.

KHRONOS. *OpenGL Object*. 2018. Disponível em: <https://www.khronos.org/opengl/wiki/OpenGL_Object>. Citado na página 30.

KHRONOS. *OpenGL shader compilation*. 2018. Disponível em: <https://www.khronos.org/opengl/wiki/Shader_Compilation>. Citado na página 35.

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 11.

NYSTROM, R. *Game Programming Patterns*. [S.l.]: Genever Benning, 2014. ISBN 0990582906. Citado na página 25.

OPENGL. *About OpenGL*. 2017. Disponível em: <https://www.opengl.org/about/>. Citado na página 14.

ROGERS, S. *Level Up*. [S.l.]: Edgard Blucher, 2016. ISBN 9788521207009. Citado na página 15.

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 46.

VRIES, J. de. *Learn OpenGL*. 2. ed. [s.n.], 2015. Disponível em: <https://learnopengl.com/book/offline%20learnopengl.pdf>. Citado 3 vezes nas páginas 14, 28 e 31.

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 49.