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

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 e inteligência artificial. 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 comummente construída utilizando-se o padrão de projeto em camadas. A Figura 2 ilustra um típico diagrama da arquitetura desse 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 atende-lo conforme suas necessidades. Para este projeto, a engine utilizada será de desenvolvimento próprio.

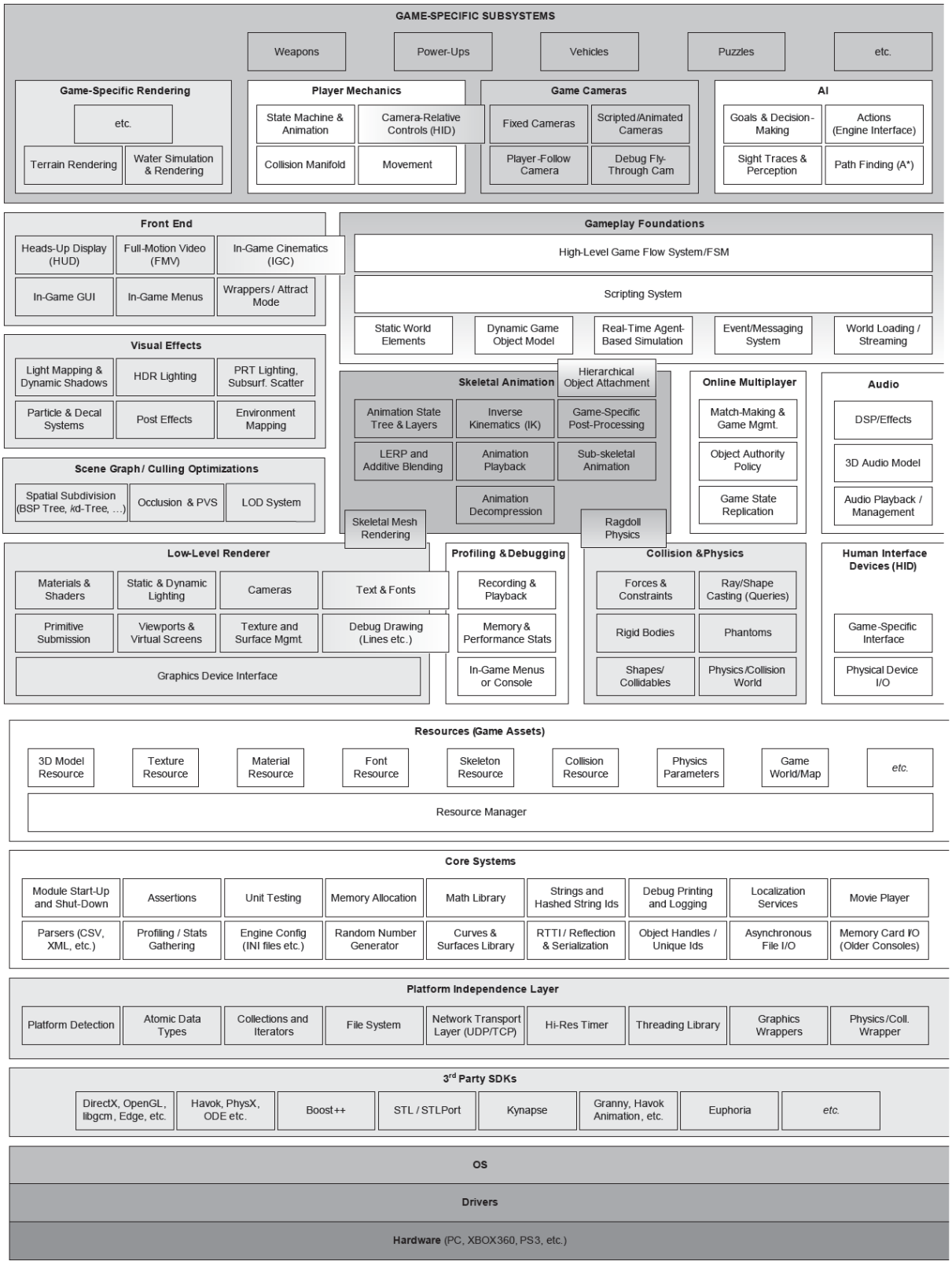

Figura 2 – Componentes de uma Game Engine. Fonte: (GREGORY, 2009)

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

Esse código sendo executado ao longo do tempo é representado pela figura:

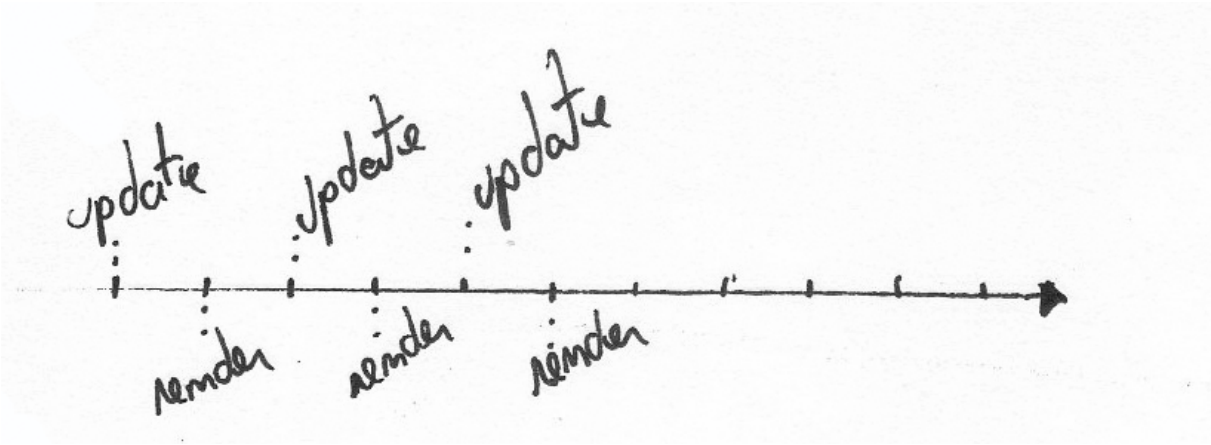

Figura 3 – Execução do game loop básico ao longo do tempo

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 representado pela sigla FPS (*Frames per second*). Cada execução do método `update` significa um passo no tempo do jogo (*timestep*). Da mesma forma que o relógio move em tiques de um segundo em um segundo, o tempo do jogo move em tiques de `update` em `update`.

4.2 Sistema de atualização

O sistema de atualização é responsável por controlar o aspecto lógico da engine. Nele ocorre 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 é 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 para o jogo.

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 poderes de processamento distintos gerou o problema descrito acima. Por exemplo, defina-se 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 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 chamamos o método `update` e portanto o personagem passa a mover-se 600 pixels por segundo ($60FPS * 10px$). 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:

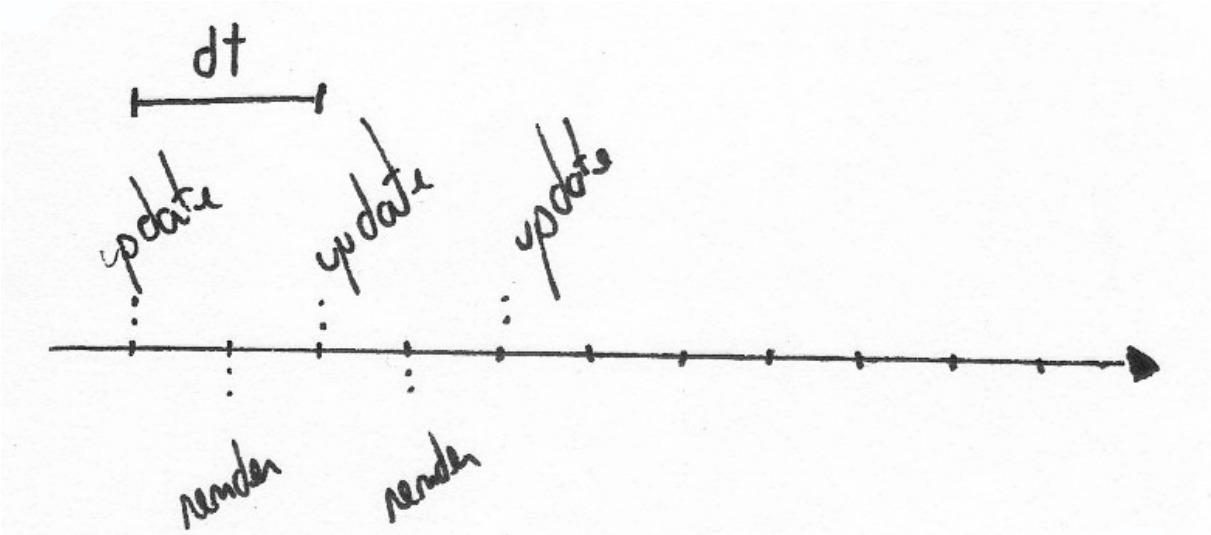

Figura 4 – Execução do game loop com timestep fixo ao longo do tempo

4.2.2 Timestep variável

Tornando a taxa de atualização Δt dinâmica ao invés de fixa ela passa a ser independente do FPS. Isso é possível medindo quanto tempo se passou entre um frame e outro. Dessa forma o game loop fica:

```
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:
//TODO: Insert Ilu. 3

Embora tenha-se resolvido o problema anterior de dois computadores com poder de processamento diferente, 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 tivermos um pico de performance, será avançado um tempo muito grande e um passo do personagem que era para ser $10 pixels$, passa a ser $10px + atraso$ e este efeito chamado de *stuttering* 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 deveria ocorrer entre o ponto atual e o próximo.

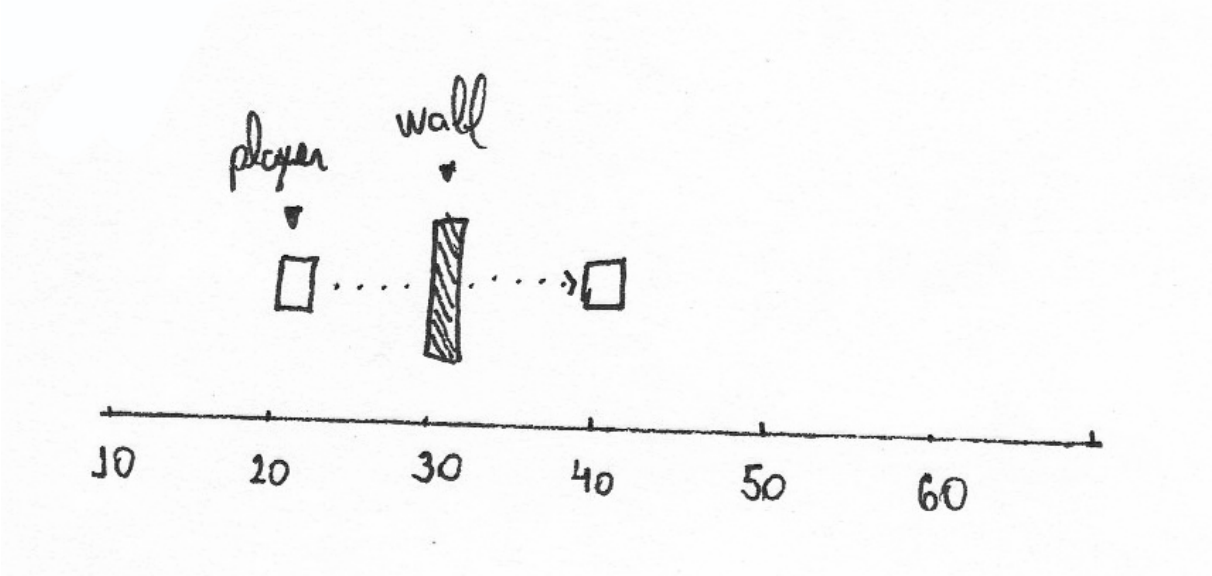

Figura 5 – Objeto ignorando colisão devido um timestep muito grande

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

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: //TODO: Insert Ilu. 6

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, 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 são atrelados à frequência do monitor que, atualmente, variam de 30 Hz até 144 Hz e para todos os efeitos são o 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 controle total sobre o SO. 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 e, sem nenhum tratamento, este efeito também causa *stuttering*. Ao chamarmos o método `render` consecutivamente e não atualizarmos a posição do objeto em cada chamada, para o usuário a impressão é de um sistema engasgado com um 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 a seguir 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 renderizado a interpolação da posição desse objeto, tornando seu movimento muito mais suave para

o usuário.

//TODO: Insert Ilu 7

//TODO: Insert Ilu 5

E o código final do game loop 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 pela 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 por si só são aquilo que na essência compõem o sistema de renderização.

4.4.1 Core-profile e Immediate mode

Immediate mode (legacy) é o modo antigo de se operar com OpenGL. Hoje ele é depreciado e substituído pelo modo Core-profile. No modo antigo as funções eram mais fáceis de usar, com muitas da funcionalidades já abstraídas pela API. Entretanto, por serem 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, nele 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 enorme máquina de estados, possuindo uma enorme coleção de variáveis que definem como operar no estado vigente. Chama-se as funções para configurar o estado atual, passando buffers e outras informações para então renderizar o estado atual.

4.4.3 Hello window

A primeira peça necessária para usar o OpenGL é criar um contexto e uma janela onde renderizar. O processo de criação da janela é especifico de cada SO e faz-se necessário uso de outra API para abstrair esse passo. Neste documento será utilizado a biblioteca `GLFW`. O código a seguir demonstra esse processo e é explicado a seguir:

```
1 public void init() {
2     if (!glfwInit())
3         System.err.println("Could not initialize GLFW.");
4
5     glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
6     glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
```

```
7       glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE)
            ;
8       glfwWindowHint(GLFW_RESIZABLE, GL_FALSE);
9
10      id = glfwCreateWindow(width, height, name,  NULL,  NULL);
11      glfwMakeContextCurrent(id);
12
13      glfwSetWindowPos(id, 2000, 60);
14      glfwShowWindow(id);
15      GL.createCapabilities();
16      glfwSwapInterval(1); //VSYNC
17
18      glViewport(0,0, width, height);
19      glEnable(GL_CULL_FACE);
20      glEnable(GL_BLEND);
21      glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
22  }
```

Algoritmo 4.6 – Inicialização da janela e contexto OpenGL

Primeiramente inicializa-se o GLFW chamando `glfwInit`. Em seguida usa-se a função `glfwWindowHint` para adicionar atributos a janela, sendo o primeiro argumento o nome do atributo e o segundo argumento o valor que se deseja atribuir. Aqui foram inseridos os atributos `GLFW_CONTEXT_VERSION_MAJOR` e `GLFW_CONTEXT_VERSION_MINOR` para indicar qual a versão ideal e mínima para rodar a aplicação. No caso essa versão é a 3.3. O atributo `GLFW_OPENGL_PROFILE` altera entre o modo core profile e immediate mode (Legacy). Cria-se então a janela com o método `glfwCreateWindow` e usa-se o id retornado para torná-lo o contexto vigente com `glfwMakeContextCurrent`. A função `glfwShowWindow` 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.

4.4.4 OpenGL pipeline

A fim de poder desenhar objetos na janela é necessário abordar os *buffer objects* e *pipeline* presentes no OpenGL. Os *buffer objects* são estruturas responsáveis por transmitir e receber dados da GPU. Existem 9 tipos de objetos e estes são divididos em duas categorias:

Regular objects

Objetos dessa categoria contem dados.

- Buffer object

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

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 3D, 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.

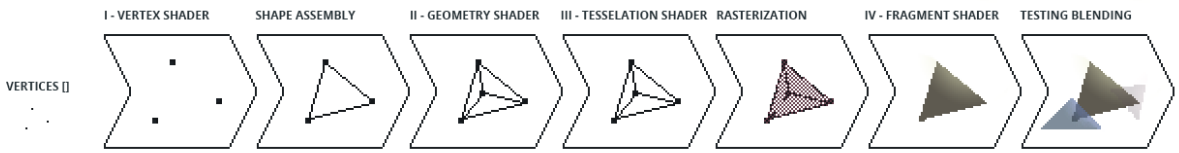

Figura 6 – Etapas do pipeline

Na etapa I o *vertex shader* é responsável, principalmente, por receber um único vértice e processar sua coordenada cartesiana de um sistema para outro. Por exemplo, as coordenadas finais desse objeto vão mudar conforme a câmera se movimento 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 6 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 6 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()* e habilita-se a localização do atributo do vértice no *vertex shader* através da função *glEnableVertexAttribArray()*. 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()`.

```
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

4.4.6 OpenGL Shading Language (GLSL)

5 Desenvolvimento do trabalho

5.1 Mecânicas

5.1.1 Sistema de caça ativa

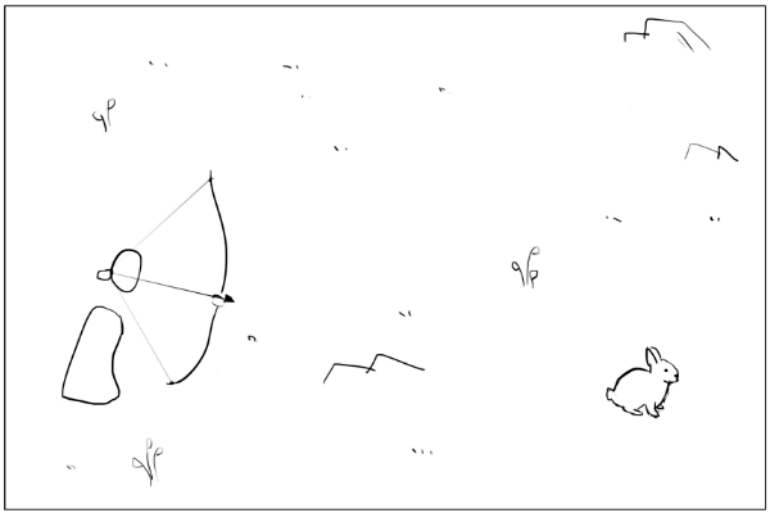

Figura 7 – Jogador caçando um coelho com arco e flecha

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

5.1.2 Sistema de armadilhas

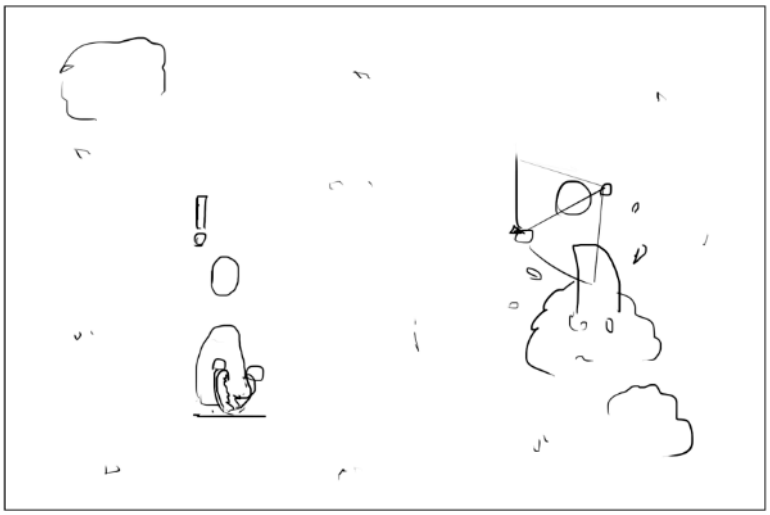

Figura 8 – Jogador pronto para atacar alvo que ficou preso na armadilha de chão

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

5.1.3 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.

5.1.4 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.

5.1.5 Cantar músicas

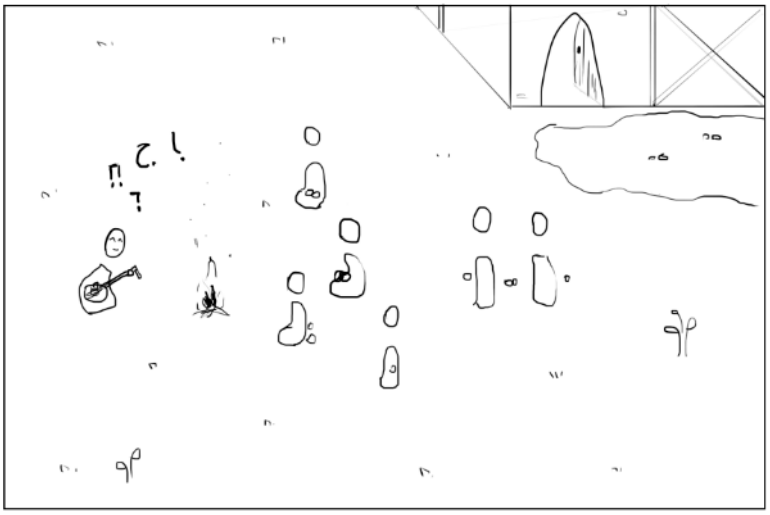

Figura 9 – Jogador cantando em um vilarejo

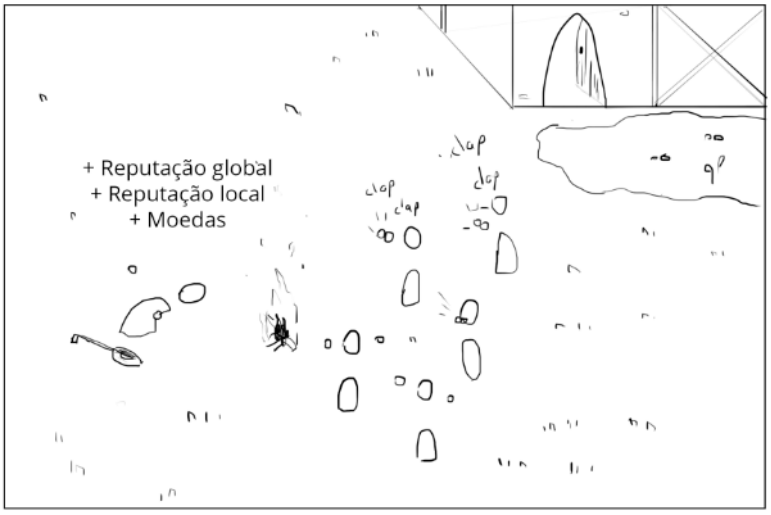

Figura 10 – Jogador sendo prestigiado após término da música

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.

5.1.6 Compor músicas

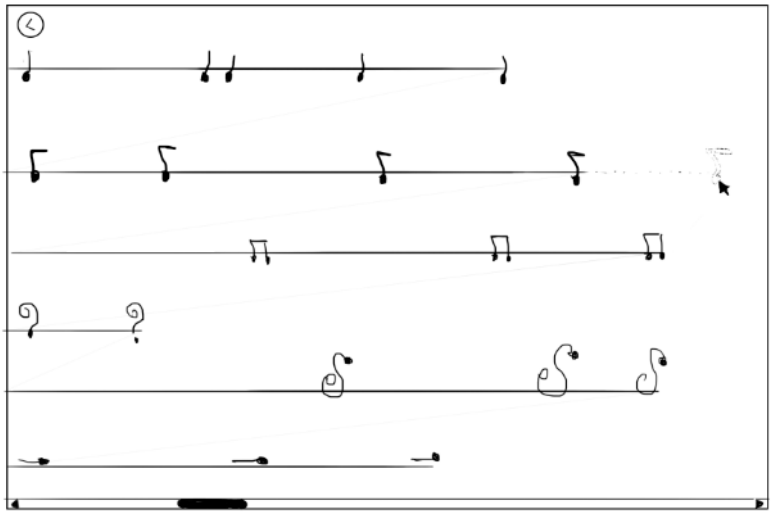

Figura 11 – Esboço do menu para compor músicas

ASD.

5.1.7 Contar histórias

ASD.

5.1.8 Compor histórias

ASD.

5.1.9 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.

5.1.10 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.

5.1.11 Domar animais

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

6 Experimentos e resultados

7 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 12, 13 e 14.

GREGORY, J. *Game Engine Architecture*. [S.l.]: Taylor and Francis Group, 2009. ISBN 9781439865262. Citado 5 vezes nas páginas 4, 19, 20, 21 e 25.

KHRONOS. *OpenGL Specifications*. 2017. Disponível em: <https://www.khronos.org/registry/OpenGL/specs/gl/>. Citado na página 13.

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 10.

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 13.

ROGERS, S. *Level Up*. [S.l.]: Edgard Blucher, 2016. ISBN 9788521207009. Citado na página 14.

VRIES, J. de. *Learn OpenGL*. 2. ed. [s.n.], 2015. Disponível em: <https://learnopengl.com/book/offline%20learnopengl.pdf>. Citado 3 vezes nas páginas 13, 27 e 29.