



Universidade Federal de Juiz de Fora

Campus Juiz De Fora

Curso De Orientação a Objetos – DCC025-2021.1

Entrega Primeira Etapa

Jogo De Combate Por Turnos

Hiero Henrique Barcelos Costa -202065136A

Matheus Cardoso Faesy - 202065065A

Thaís de Jesus Soares - 202065511B

Juiz de Fora

2021

1. INTRODUÇÃO

1.1. Projeto

Nesse projeto pretendeu-se criar a implementação de um jogo com batalha de turnos. Neste, o jogador irá escolher seu personagem, que será representado por um deus, e progredir na história enfrentando diferentes tipos de inimigos que usam diferentes tipos de estratégia.

No decorrer da história ele vai se deparar com pequenas escolhas que poderão alterar drasticamente o final do jogo, essas escolhas vão contar pontos para seu final. Uma escolha boa, por exemplo, irá adicionar pontos para o final bom, enquanto uma escolha ruim pontos para o final ruim. No final da jogatina esses pontos serão contabilizados, de tal modo que cada usuário possa ter um final diferente.

Assim, este deverá manusear seus recursos e fazer as escolhas corretas para ser capaz de derrotar seus inimigos e eventualmente chegar ao fim do jogo.

1.2. Classes

O jogo irá iniciar pela classe *main*, *Jogo*, que chamará a classe *Jogador*. Essa, por sua vez, irá chamar a tela de login que vai associar o usuário ao seu objeto jogador. O objeto jogador terá, ainda, uma chave que associa ele a um progresso na história para que assim possa continuar de onde parou ao entrar e sair do jogo múltiplas vezes.

Ao enfrentar inimigos, o jogo coloca o jogador e seu deus selecionado contra um bot e seu inimigo. Nessa batalha, em seus respectivos turnos, tanto o jogador quanto o bot irão escolher habilidades que causarão dano na vida do adversário ou consumíveis que poderão aumentar tanto a própria vida quanto o poder, e o último de pé será dado como o vencedor daquela batalha.

Enquanto isso, todos os objetos das classes *Jogador*, *Deus*, *Inimigos*, *Bot*, *Habilidade* e *Consumível* vão ser armazenados na classe *Armazém* que irá salvar todos os dados e pode ser usada para chamá-los se necessário posteriormente.

Link para o repositório: <https://github.com/faesy/DCC025-Projeto2021.1.git>

2. DESENVOLVIMENTO

2.1. Classe Armazém

A classe *Armazém* foi criada tendo por objetivo propiciar a integração entre os arquivos texto, os quais irão permitir a persistência dos dados do programa, durante e entre suas execuções e as demais classes criadas, as quais vão, por sua vez, acessar essas informações.

Para tanto, foi empregada a interface padrão *Map*, bem como sua implementação padrão *HashMap*, as quais, de forma conjunta, permitiram a criação de dicionários dinâmicos, através dos quais cada instância de uma dada classe pode ser associada a uma chave, que será utilizada para buscar os dados de um objeto com relativa facilidade. Nesse sentido foram criados os seguintes atributos:

- *Deuses*, de interface padrão *Map*, declarado como atributo estático e privado que irá receber os dados referentes aos deuses existentes.

- *Habilidades*, análogo ao atributo *Deuses*, sendo responsável por receber os dados referentes às habilidades presentes no programa.

- *Jogadores*, novamente análogo ao atributo *Deuses*, recebendo os dados relativos aos jogadores cadastrados.

Ademais, é válido ressaltar que os métodos contidos na classe podem ser agrupados em três grupos principais, os quais serão expostos abaixo de forma geral.

2.1.1. Escrita nos arquivos

A priori, foi instanciado um objeto do tipo *File*, passando como argumento o caminho para o arquivo a ser manipulado. Posteriormente, verificou-se a existência do arquivo e em caso de negativa este foi inicializado vazio, por meio da função *createNewFile()*. Por fim, a escrita no arquivo foi realizada por meio da função *write()*.

2.1.1. Leitura dos arquivos

A leitura dos arquivos foi realizada de forma semelhante à escrita. Diferenciando-se, principalmente no que tange ao uso das funções *readLine()* e *ready()*, as quais leem e verificam a existência de mais linhas respectivamente. Além de serem as responsáveis por armazenar os dados lidos na estrutura *Map*.

2.1.1. Alteração dos arquivos

A função *atualizaJogadores(Map<String, Jogador> jogadores)*, possui um mecanismo de implementação semelhante às demais e foi elaborada visando atualizar os dados de um jogador. Para tanto, a estrutura *Map* recebida por parâmetro, a qual contém os dados antigos, bem como aqueles que sofreram alterações, é utilizada para escrever os dados dos jogadores novamente no arquivo, após este ser apagado através da função *delete()*.

2.2. Classe Bot

A classe Bot foi criada tendo por objetivo o manejo dos inimigos do jogo. Para tanto, ela apresenta um construtor inicializar e os processos e métodos para:

- Criação de inimigos.
- Destruição de inimigos.
- Ações dos inimigos durante o combate, escolhendo as habilidades deles e retornando o dano da habilidade escolhida.
- Controle da vida do inimigo durante o combate, verificando se o inimigo está morto ou não a cada ataque recebido.

2.3. Classe Consumível

A classe *Consumível* foi criada tendo por objetivo a criação de objetos poções que atuarão no jogo como fonte de vida ou poder para o personagem. Assim, ela dispõe de quatro atributos protegidos:

➤ *Carga*, do tipo inteiro, para mostrar quantas vezes ela pode ser utilizada durante o combate.

➤ *Tipo*, do tipo inteiro, para diferenciar uma espécie de poção da outra.

➤ *Nome*, do tipo String, para nomear a poção.

➤ *Descrição*, do tipo String, para explicar o funcionamento da poção.

Além dos métodos protegidos.

➤ *RecuperarVida*, para retornar os valores dos efeitos de cura momentâneo.

➤ *AumentarPoder*, para retornar os valores dos efeitos de aumento de poder momentâneos.

2.4. Classe Deus

A classe *Deus* foi criada com o objetivo de conter todas as informações sobre os deuses, possibilitando a criação dos objetos *Deus* para cada um dos avatares que o jogador terá direito a escolher equipar. Dessa forma, possui, então, os atributos privados para:

➤ Analisar a vida, através dos atributos inteiros *vidaBase*, *vidaMax*, *vidaAtual* para registrar respectivamente o modificador da vida, a vida máxima e a vida atual do *Deus*.

➤ Analisar o poder, através dos atributos inteiros *poderBase* e *poder* para registrar respectivamente o modificador de poder e o poder bruto que o deus possui.

➤ Analisar o nível e experiência, através dos atributos inteiros *nível* e *xp*.

➤ Obter o nome e descrição do *Deus*, através dos atributos *String nome* e *descricao*.

➤ Verificar se o *Deus* morreu, através do atributo booleano *morto*.

➤ Manter o controle das habilidades e das cargas de cada uma delas, através dos arranjos de habilidades e de inteiros, *habilidade* e *cargas*, respectivamente.

➤ Verifica a quantidade total de objetos *Deus*, através de um atributo estático inteiro *qtddDeuses*.

Além dos métodos:

2.4.1. Relacionados a vida do personagem

Os responsáveis pelo manejo desse aspecto são *descansar(int vida)*, cuja função é recuperar a vida e as cargas das habilidades e consumíveis, *reduzirVida(int dano)*, cuja função é reduzir a vida do personagem e, caso ele morra, fazer com que ele perca experiência, além da *recuperarVida(int recuperacao)*, cuja função é recuperar a vida.

2.4.2. Relacionados ao nível e experiência do personagem

Os responsáveis pelo manejo desses aspectos são *funcaoLvlUp()*, cuja função é verificar a quantidade de experiência do jogador e, caso necessário, atualiza seu nível, *ganharXp(int ganho)*, cuja função é adicionar o valor da experiência dada ao valor total de experiência que o personagem possui e chamar o método *funcaoLvlUp()*, *perderXp(int perda)*, cuja função é retirar o valor da experiência dada ao valor total de experiência que o personagem possui e chamar o método *funcaoLvlUp()*.

2.4.3. Relacionados ao poder do personagem

Os responsáveis pelo manejo desse aspecto é a *funcaoPoder()*, cuja função é atualizar o poder bruto do *Deus*.

2.4.4. Relacionados às habilidades

Os responsáveis pelo manejo desse aspecto são *alocarHabilidades(Habilidade[] habilidades)*, cuja função é delimitar quais são as habilidades que o Deus está utilizando, *usarHabilidade(int resposta)*, cuja função é gastar uma carga da habilidade e retornar o dano causado por ela e *verificaCarga(int slot)*, cuja função é verificar se ainda há cargas para a habilidade.

2.5. Classe Habilidade

A classe *Habilidade* foi criada com o objetivo de conter todas as informações sobre as habilidades, possibilitando a criação dos objetos habilidades que em suma serão os ataques que os *Deus* e os *Inimigos* poderão escolher usar. Possui, então, os atributos privados:

- *Descrição*, do tipo *String*, que vai descrever como a habilidade funciona.
- *Nome*, do tipo *String*, que vai nomear a habilidade.
- *Dano*, do tipo inteiro, que diz o dano base da habilidade.
- Método protegido *dano*, que retorna o dano causado.
- *Carga*, do tipo inteiro, que expressa quantas vezes uma dada habilidade pode ser usada antes de um descanso.

2.6. Classe Inimigo

A classe *Inimigo* se assemelha a classe *Deus*, porém com intuitos opostos. Enquanto o *Deus* será controlado pelo *Jogador*, o *Inimigo* será controlado pelo programa, mais especificamente pela classe *Bot*. Logo para evitar uma repetição desnecessária mencionaremos apenas os métodos e atributos novos ou que se diferenciam de alguma forma.

- Inexistência de atributos e métodos relacionados à *Consumível*.
- Método *estabeleceVida()*, privado e void, calcula a *vidaAtual* com base no *nivelDesafio*, na *vidaBase* e na *vidaMax*.
- Método *categorizaXP(int modo)*, privado e void, calcula o *xpDada* com base no *nivelDesafio* e no *modo*.
- Método *estabelecePoder()*, privado e void, calcula o *Poder* com base no *nivelDesafio* e no *poderBase*.
- Atributo *xpDada*, privado inteiro, representa a quantidade de xp que o jogador recebe ao derrotar esse inimigo
- Atributo *nívelDesafio*, privado inteiro, número que representa a dificuldade do inimigo.

2.7. Classe Jogador

A classe *Jogador* foi desenvolvida visando gerenciar os dados relativos a um jogador. Nesse sentido, foram elaboradas as funções *logar()*, *deslogar()* e *cadastrar* (*boolean cadastrado*). Essas, torna-se válido destacar, ainda se encontram incompletas, visto que não foram integradas às interfaces gráficas. Esses métodos encontram-se descritos abaixo resumidamente.

2.7.1. Cadastrar

O funcionamento do método consiste na verificação prévia da existência de um jogador com o mesmo nome informado pelo usuário. Caso o programa encontre equivalência, ou seja, se o nome informado já estiver em uso, exige-se que o jogador tente se cadastrar novamente passando um nome e senha novos. Por fim, os dados recém inseridos são salvos no arquivo.

2.7.2. Logar

A função consiste, novamente, na verificação da existência de jogador com o mesmo nome que o informado pelo usuário. Caso a busca retorne verdadeiro, e a senha informada coincida com a armazenada, o status do jogador como logado é atualizado. Se o resultado for falso, o programa demanda que o usuário se cadastre.

2.7.3. Deslogar

O método atualiza os dados do jogador mudando seu status de logado para falso.

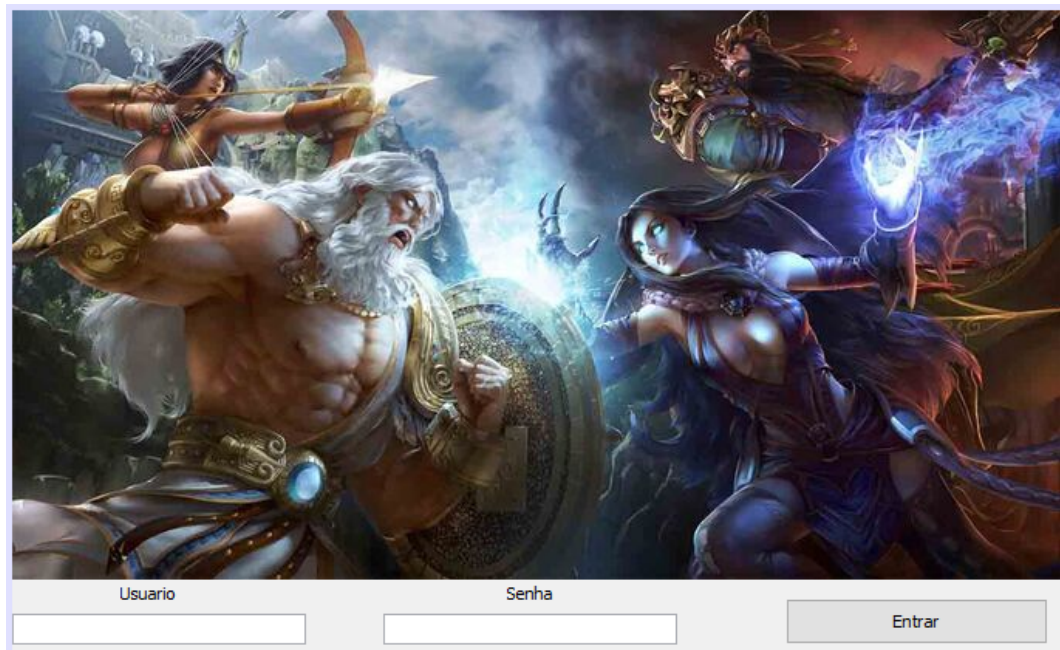
Ademais, a classe *Jogador* conta com os seguintes atributos:

- *DeusEscolhido*, instância do tipo *Deus*, a qual armazena o deus escolhido pelo jogador.
- *Nome* e *senha*, de tipo *String*, que serão responsáveis pela identificação de um jogador pré-cadastrado.
- *Cadastrado* e *logado*, variáveis booleanas que vão indicar o status do jogador quanto ao seu cadastro e presença atual no jogo respectivamente.
- *ConsumíveisEquipados[]*, arranjo de objetos do tipo *Consumível* que irá guardar os consumíveis pertencentes a um determinado jogador.

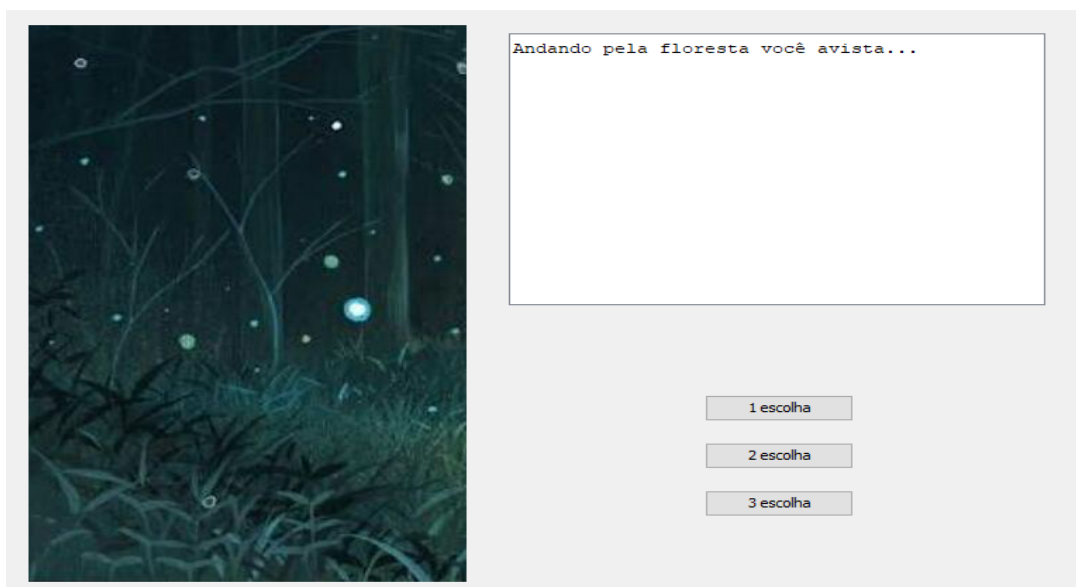
2.8. Interface gráfica

Para a entrega da primeira parte do trabalho foi preparado somente três rascunhos de interface. Essas são a interface de *Login*, a Interface do *Menu de Escolhas* e a interface de *Diálogo*. Já a interface de *Cadastro* não foi feita ainda, pois seria similar à de *Login*.

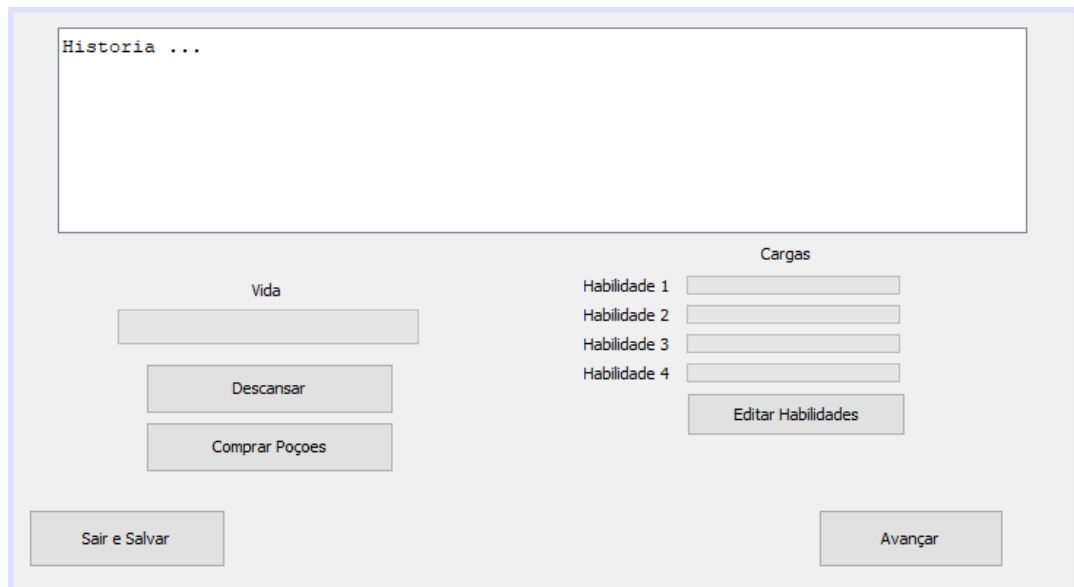
➤ Rascunho da Interface de *Login* desenvolvida



➤ Rascunho da Interface de *Diálogo* desenvolvida

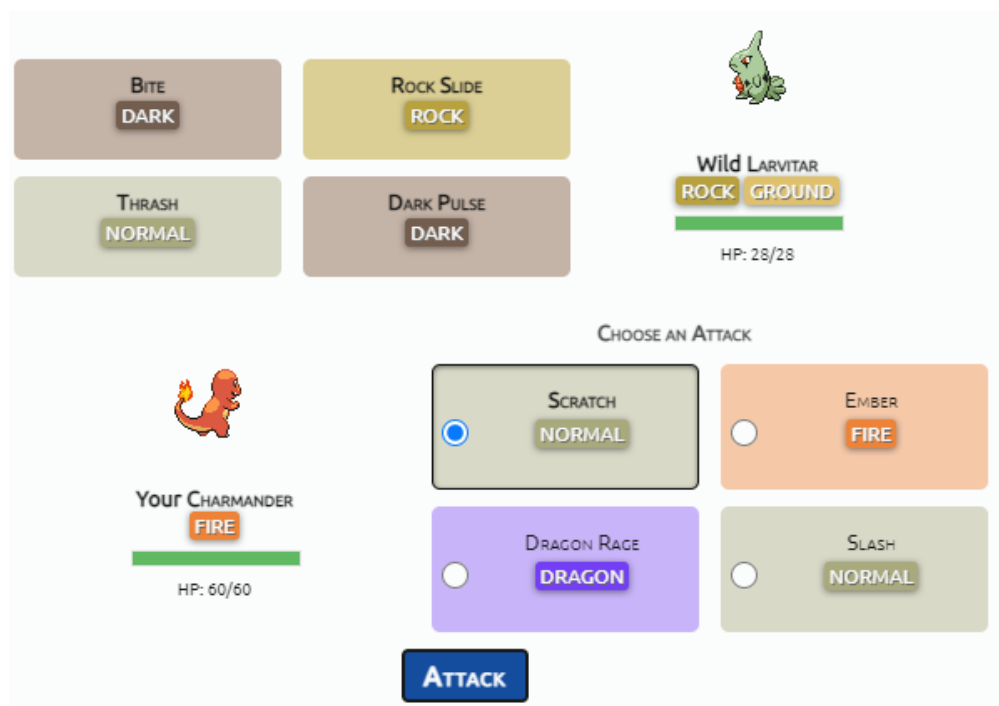


➤ Rascunho da Interface de *MenuDeEscolhas* desenvolvida:







Ademais, segue abaixo algumas interfaces ainda por fazer e imagens destas em jogos similares.

➤ Menu De Combate – Tem como função realizar o combate de turnos do jogo.




➤ Menu de Seleção – Tem como função selecionar o seu *Deus* no início do jogo.

POKEMON #	LEVEL / Exp / TYPE	MOVES
 CHROME FALINKS	LEVEL: 15 Exp: 15,000 FIGHTING	Tackle Rock Smash Close Combat Megahorn IN TEAM CHANGE MOVES
 CHROME SCORBUNNY	LEVEL: 13 Exp: 13,000 FIRE	Tackle Ember Quick Attack Double Kick EVOLVE IN TEAM CHANGE MOVES
 CHROME CASTFORM (SUNNY FORM)	LEVEL: 5 Exp: 5,000 FIRE	Tackle Ember Flame Wheel Flamethrower IN TEAM CHANGE MOVES
 CHROME HAWLUCA	LEVEL: 87 Exp: 87,004 FIGHTING FLYING	Flying Press Aerial Ace Sky Attack Wing Attack IN TEAM CHANGE MOVES
Showing 1 to 6 out of 6 matches		Karate Chop

➤ Seleção de Habilidades – Tem como função propiciar a troca de habilidades do seu *Deus*.

Total Coins: 69,688


 Charmander
FIRE

Rearrange Attacks

Price of attacks vary depending on the level of the attacks will cost less for fire

Z-Moves are marked with * and can only be taught to a fire type pokemon. These can only be taught to a fire type pokemon.

Please check the Attack De

Change Attacks

Scratch

Ember

Dragon Rage

Slash

Shadow Punch - 30000
 Shadow Sneak - 20000
 *Shattered Psyche - 250000
 Sheer Cold - 15000
 Shell Side Arm - 58000
 Shell Trap - 45000
 Shock Wave - 30000
 Signal Beam - 40000
 Silver Wind - 30000
 Sing - 25000
 *Sinister Arrow Raid - 250000
 Sizzly Slide - 27000
 Sketch - 10000
 Skitter Smack - 32000
 Skull Bash - 65000
 Sky Attack - 71000
 Sky Drop - 30000
 Sky Uppercut - 39000
 Slam - 30000

chosen. That is, fire

may be taught to a

CLOSE

3. EXPECTATIVAS

Segue abaixo algumas ideias que dependendo da progressão e facilidade do desenvolvimento do trabalho podem vir a ser implementadas.

- Itens: Armaduras e armas que o jogador poderá equipar em seu *Deus* conforme sua vontade. Aumentado suas resistências, *vida* e *dano*.
- Resistências: A resistência seria um novo atributo da classe *Deus* que ajudaria a mitigar (diminuir) o dano recebido.
- Vulnerabilidades: A vulnerabilidade seria um novo atributo da classe *Deus* que representa a fraqueza em relação a um tipo de dano.
- Tipos Elementais (Deuses e Danos): Os deuses, inimigos e danos pertenceriam a um tipo de elemento específico o que abriria espaço para novas estratégias uma vez que novas fraquezas e pontos fortes foram adicionados.
- Outros Consumíveis: Adicionar consumíveis que poderiam aumentar a resistência contra determinado elemento
- Loja: Implementar uma loja e uma moeda. Que seriam usados para comprar e vender itens e consumíveis.
- Balanceamento: Balancear o jogo após implementar as mudanças, de forma a garantir que as alterações nas propriedades iniciais do jogo não resultem num jogo muito fácil/muito difícil.
- Implementar dois tipos diferentes de usuários, o usuário premium que irá ter acesso a todos os personagens iniciais e a toda história e o usuário base que irá somente ter acesso a alguns personagens iniciais e a uma parte inicial da história.
- Implementar diferentes modos de dificuldades.

4. CONCLUSÃO

No fim do projeto terá de ser apresentado um protótipo de jogo que:

- Possui a capacidade de diferenciar entre os usuários.
- Salva o progresso do jogo.
- Possibilita a escolha de um Deus, ou mais, caso seja um jogador premium.
- Possibilita ao jogador a chance de escolher suas ações durante a progressão da história, as quais irão influenciar no destino final.
- Diversos itens e habilidades que proporcionam um estilo de jogo próprio para cada jogador.

Por fim, o projeto, quando finalizado, apresentará jogo com uma aventura épica de um Deus em sua jornada para concluir seu objetivo final. Possuindo vários desafios para proporcionar um momento agradável de diversão aos usuários que se dispõem a gastar parte de seu tempo livre desbravando a história do jogo.