Universidade de São Paulo
Instituto de Matemática e Estatística

Manual do Desenvolvedor

aMaze Unknown

Dezembro de 2010

Sumário

1 Introdução

O *aMaze Unknown* foi desenvolvido com o foco em tornar o jogo preparado para mudanças. Os esforços necessários para fazer alterações no conteúdo existente ou incluir novas funcionalidades não são grandes.

O objetivo deste documento é auxiliar desenvolvedores interessados na expansão do jogo. O conteúdo busca esclarecer as dúvidas em relação às funcionalidades implementadas e dar uma visão geral da estrutura do jogo.

2 Especificações Técnicas

O jogo é uma base jogável em rede, multiplataforma, multiplayer e preparada para ser estendida. O jogo foi baseado na plataforma Panda3D e a manipulação do código necessita conhecer, aprender e explorar a engine. Diversas das bibliotecas e rotinas utilizadas pertencem ao Panda 3D.

A engine Panda 3D suporta o desenvolvimento tanto em Python como em C++. A escolha por utilizar Python foi por ser uma linguagem simples, dinâmica e com melhor documentação no site do Panda.

A estruturação do código e das classes seguem padrões de design orientados a objetos. A escolha por essa estrutura foi para tornar simples e fácil estender as funcionalidades iniciais. A idéia baseou-se em ter um jogo preparado para mudanças.

Para a criação de modelos gráficos tridimensionais, utilizou-se do 3D Studio Max.

3 O Jogo

3.1 Estrutura

O jogo é segmentado em dois níveis: um jogo principal e diversos mini-games.

O jogo principal tem como cenário um labirinto no qual os jogadores se descolam e interagem livremente com o objetivo de ser o primeiro a conseguir escapar. Nesse trajeto, o jogador será submetido a diversas provas através de mini-games.

Nos mini-games os jogadores têm a possibilidade de enfrentar ou cooperar buscando ganhar vantagem em relação ao restante dos adversários.

3.2 Mini-Games

Os mini-games são uma oportunidade efetiva dos jogadores se enfrentarem/colaborarem e eventualmente ganharem alguma vantagem. Cada mini-game pode ser classificado em quatro classes: individuais, grupos, coletivos ou complôs.

No primeiro estilo, cada jogador escolhe suas decisões independentemente e tem o objetivo de vencer os demais jogadores; no segundo, os jogadores são separados em dois grupos e precisam se ajudar nas escolhas para vencer o grupo adversário; no estilo coletivo a principal característica é a cooperação: os jogadores devem todos colaborar para atingir um objetivo comum; e por fim o estilo complô, quando um jogador enfrenta os outros, tentando impedi-los de atingirem suas metas.

3.2.1 Um Exemplo de Mini-Game

A versão atual do jogo apresenta um mini-game coletivo de exemplo.

O cenário do mini-game é uma plataforma móvel com dois níveis. No nível superior, a plataforma é transparente e permite que o outro nível seja visto. Já no nível inferior a plataforma é opaca e em forma de labirinto.

Inicialmente, existe uma bolinha em algum ponto deste labirinto e um buraco em algum outro ponto. O objetivo é que os jogadores trabalhem em conjunto para levar a bolinha até o buraco no menor tempo possível. Os jogadores ficam no nível superior, onde se deslocam livremente, inclinando a plataforma de acordo com os respectivos pesos.

O eixo de movimento da plataforma está localizado em seu centro, por isso um jogador mais distante do centro proporciona uma maior inclinação da plataforma.

3.2.2 Como criar um Mini-Game

Inicialmente o desenvolvedor deve criar os modelos, cenários, do mini-game. Depois basta colocar as texturas na pasta *imagens* os arquivos referentes ao modelo na pasta *modelos*. Os modelos devem estar no formato *egg*, que será melhor explicado a seguir.

O passo seguinte é criar o código do mini-game. A classe MiniGame encapsula as características de um mini-game e define um conjunto de métodos abstratos padrão.

```
class MiniGame () :

        def __init__ (self) :
                self.emAndamento = True

        def run (self):
                self.carrega()
                self.inicia()
                self.mainLoop = taskMgr.add(self.loop, "loop")
                while self.emAndamento :
                        self.mainLoop.step()
                self.finaliza()
                self.limpa()

        @abstractmethod
        def carrega (self):
                raise NotImplementedError

        @abstractmethod
        def inicia (self):
                raise NotImplementedError

        @abstractmethod
        def finaliza (self):
                raise NotImplementedError

        @abstractmethod
        def loop (self) :
                raise NotImplementedError

        @abstractmethod
        def limpa (self) :
                raise NotImplementedError
```

Código 3.1: *Classe MiniGame.*

Para criar um novo mini-game, deve-se criar uma subclasse de MiniGame e implementar esses 5 métodos abstratos. Por exemplo, o mini-game implementado:

```
class MesaMovel (MiniGame) :

        def carrega (self) :
                self.posiciona_camera()
                self.carrega_modelos()
                self.setup_colisao()
                self.setup_luzes()
                ...

        def inicia (self):
                posInicial = (-4.5, 7, self.cenario.getZ() + 2.9)
                self.raizBola.setPos(posInicial)
                self.bolaV = Vec3(0,0,0)
                self.aceleracaoV = Vec3(0,0,0)

        def finaliza (self):
                taskMgr.remove(self.mainLoop)

        def loop (self, task):
                dt = task.time - task.last
                task.last = task.time

                if dt > .2: return task.cont

                self.posiciona_camera()
                for i in range(self.cHandler.getNumEntries()):
                        entry = self.cHandler.getEntry(i)
                        ...
                ...

        def limpa (self) :
                self.raizBola.remove()
                self.cenario.remove()
                messenger.send('fimMiniGame')
```

Código 3.2: *Classe MesaMovel.*

Com esse conjunto de métodos implementados, é possível iniciar o mini-game, colocá-lo em execução e finalizá-lo de forma correta e segura. Além disso, a obrigatoriedade de implementar esses métodos padroniza a criação e inserção de novos mini-games.

A chamada do mini-game é feita na classe *Main* e utiliza apenas o método *run()*. O comportamento desse método é definido pelas classes abstratas que devem ser implementadas. O desenvolvedor tem total poder e liberdade de criar o mini-game da forma que quiser.

O último passo é inseri-lo no jogo, ou seja, torná-lo acessível. Para tal, basta acrescentar a classe do novo mini-game na lista de mini-games existentes no método *inicia* na classe *Main*.

```
self.minigames.append(MesaMovel)
self.minigames.append(MiniGame2)
self.minigames.append(OutroMiniGame)
self.minigames.append(MaisUmMiniGame)
...
self.minigames.append(UltimoMiniGame)
```

Código 3.3: *Lista de Mini-Games existentes.*

Quando o jogador encontrar um mini-game, será sorteada alguma classe dessa lista e o mini-game iniciará automaticamente.

```
classe = self.minigames[randint(0, len(self.minigames) - 1)]
minigame = classe(self.jogadores)
```

Código 3.4: *Sorteio de um Mini-Game.*

4 O 3D Studio Max

No desenvolvimento de jogos a qualidade gráfica é um dos diferenciais. Desenvolver um jogo com código organizado, extensível, multiplataforma e não se preocupar com a parte gráfica certamente não será uma boa escolha quando for apresentado aos usuários finais.

Existem diversas opções poderosas para modelagem gráfica tridimensional, porém a utilização não é trivial. Por combinar relativa facilidade de uso e recursos muito poderosos, optou-se por utilizar o 3D Studio Max.

Outras boas ferramentas para a modelagem gráfica e que são compatíveis com o panda são o *Blender* e o *Maya*.

4.1 Criação dos Modelos

Visando um aplicativo eficiente, a criação do ambiente foi projetada para aproveitar o máximo de desempenho, principalmente com relação às colisões, com o mínimo de perda de qualidade visual. A quantidade de vértices nos modelos é a mínima aceitável e o mapeamento de texturas foi feito com o mínimo de repetição.

Um dos problemas enfrentados na criação dos modelos é a definição da relação entre quantidade de vértices e qualidade visual. Um modelo com uma grande quantidade de faces e vértices é pesado para renderizar, enquanto um modelo com poucas faces e vértices não apresenta um visual aceitável.

A solução para lidar com esse problema foi encontrar uma ponderação entre os dois fatores que combinasse satisfatoriamente a qualidade visual do modelo final e a complexidade envolvida na renderização do mesmo.

Outra otimização realizada, na busca por um melhor desempenho, foi a adaptação dos modelos para versões mais leves e simples. Essa adaptação inclui a redução do tamanho do arquivo do modelo, através da remoção de vértices não necessários (que não são exibidos na renderização, por exemplo vértices internos a um polígono) e a unificação de objetos em que a forma de colisão teria o mesmo tratamento.

A primeira alteração permitiu a utilização de modelos mais leves e fez com que o carregamento e a renderização ficassem mais eficientes. A modificação para as colisões facilitou o tratamento. Anteriormente, era necessário verificar se existia colisão com todos os objetos que compunham o modelo (e muitas vezes o tratamento para essas colisões era o mesmo); após a alteração,

esses objetos formavam um único componente, simplificando a verificação e os tratamentos das colisões.

4.2 Criação de Novos Modelos

A criação de novos modelos pode ser feita com qualquer ferramenta que possibilite exportar o modelo criado para o formato *egg*. Os novos modelos devem ser colocados na pasta *modelos* e as respectivas texturas na pasta *imagens*.

A alteração do modelo que representa um personagem pode ser feita alterando o parâmetro que é passado para a classe *Personagem* no momento da criação. Esse parâmetro representa o nome do arquivo que define o modelo na pasta.

5 O Ambiente Panda3D

A plataforma escolhida para rodar o jogo foi o Panda 3D. "Panda3D é uma engine 3D: uma biblioteca de sub-rotinas para renderização 3D e desenvolvimento de jogos. A biblioteca é escrita em C++ com um conjunto de associações em Python. O desenvolvimento de jogos com o Panda3D geralmente consiste em escrever um programa em Python ou C++ que manipula a biblioteca Panda3D" [1].

O Panda inclui gráficos, áudio, entrada e saída, detecção de colisão e diversos outros recursos relevantes para a criação de jogos, desde uma simples árvore de renderização até sub-rotinas que facilitam a comunicação em rede.

Alguns conceitos relacionados ao Panda são detalhados a seguir.

5.1 ShowBase

O coração de uma aplicação desenvolvida com Panda3D está na classe ShowBase. Ela é a responsável por diversos objetos essenciais para a aplicação e implementa o método *run()*, que abre uma janela e começa a execução do programa. Portanto, é necessário que pelo menos uma classe estenda ShowBase, que será o ponto de partida da aplicação.

[1] Tradução livre do site oficial do Panda3D. Texto original disponível em: http://www.panda3d.org/manual/index.php/Introduction_to_Panda3D.

A classe ShowBase define, entre outros, os seguintes objetos: base, render, camera, loader, taskMgr, messenger, render2d, aspect2d e hidden. Esses objetos são responsáveis por organizar os recursos que servem de base para o jogo, por exemplo a câmera, as tarefas, as mensagens, o que será renderizado e formas simples de acesso a métodos importantes, como o de carregar um modelo.

5.2 Grafo de Cena

O Panda3D mantém uma relação, em forma árvore, dos objetos que devem ser renderizados. Um objeto será renderizado se e somente se pertencer a essa árvore. Essa árvore recebe o nome de Scene Graph, ou Grafo de Cena, e sua raiz é o objeto render.

Por definição, essa relação é hierárquica. Portanto, os objetos que estiverem abaixo de um objeto X serão renderizados "em relação à" X. Isso quer dizer que algumas propriedades desses objetos, como posição, orientação e tamanho, serão calculadas sempre em relação às propriedades de X. Por exemplo, o centro do eixo de coordenadas da posição desses objetos passa a ser a posição de X.

5.3 NodePath

A classe NodePath é a unidade fundamental de interação com o grafo de cena. Todos os objetos passíveis de serem renderizados são do tipo NodePath. Um objeto é inserido no grafo de cena com o método *reparentTo(outroObjeto)*. Para que um objeto X seja independente de outros objetos, deve-se inserí-lo na árvore exatamente abaixo da raiz, ou seja, fazer *X.reparetnTo(render)*; para que X seja renderizado em relação ao objeto Y deve-se fazer *X.reparentTo(Y)*.

5.4 Tasks

As Tasks são funções especiais que são executadas a todo frame, uma após a outra, cooperativamente. Um novo frame é iniciado somente quando todas as tasks terminarem suas execuções. Uma task é definida como uma função ou um método.

```
def taskExemplo (task) :
        #Faz algo
        return task.cont
```

Código 5.1: Tasks.

5.5 Task Manager

As tasks são atribuídas ao objeto global taskMgr, que é responsável por gerenciá-las e controlar suas execuções. Para que uma nova task seja iniciada, é necessário atribuí-la ao taskMgr pelo comando *taskMgr.add(taskExemplo, "nomeTask")*.

Assim, essa task começará sua execução a partir do próximo frame. O Task Manager mantém o controle de execução de todas as tasks, podendo continuar a executá-las, removê-las da lista de tasks ativas ou fazer algo quando uma task termina sua execução.

É possível, ainda, visualizar as tasks registradas através de uma interface gráfica através do comando *taskMgr.popupControls()*. Essa opção é útil na fase de desenvolvimento, especialmente para debug.

5.6 Colisões

Uma parte importante de um jogo é o tratamento de colisões. O panda3D possui um mecanismo simples e eficiente para detectar e notificar as ocorrências de uma colisão. Mas antes disso, uma definição:

Uma colisão acontece quando um objeto, se movimentando no espaço, tenta ocupar um espaço já ocupado por outro objeto.

O programador é o responsável por definir os objetos passíveis de gerar uma colisão, os objetos que podem ser colididos e tratar as colisões detectadas no espaço.

5.6.1 Participantes

Primeiramente, vamos a duas definições:

Um objeto From é o objeto ativo da colisão, ou seja, aquele que colidiu; um objeto Into é o objeto passivo da colisão, ou seja, aquele que foi colidido.

Essa separação é importante pois o sistema mantém uma lista de todos os objetos *From* no espaço e checa, a todo frame, se eles estão colidindo com outros objetos. Um objeto estático, geralmente, é classificado como um objeto *Into*, enquanto um objeto móvel, um objeto *From*.

5.6.2 Sólidos de Colisão

O objeto central no sistema de colisões é o CollisionSolid, ou sólido de colisão. Para que um objeto no espaço seja detectado pelo sistema de colisões, é necessário que ele possua um CollisionSolid associado.

Dentre os vários tipos de sólidos de colisão, está o ColisionSphere; é o sólido mais robusto e otimizado para ser utilizado em uma colisão. Além disso, é o único sólido que é capaz de gerar e de receber uma colisão com todos os outros tipos de sólidos. Especificamente, uma parte do objeto é colidível se está no interior da esfera de colisão.

O seguinte fragmento de código ilustra a criação de um sólido de colisão e sua associação a um objeto:

```
1 sphere = CollisionSphere(0, 0, 0, 1)
2 collisionNodePath = objeto.attachNewNode(CollisionNode("esferaColisao"))
3 collisionNodePath.node().addSolid(sphere)
```

Código 5.2: Sólido de colisão.

A linha 1 define uma esfera de colisão de centro (0, 0, 0) e raio 1. A linha 2 associa um CollisionNode a um objeto, que permitirá que se defina um ou mais sólidos de colisão para aquele objeto. Na linha 3, a esfera criada é associada ao objeto. Nesse momento, a posição do centro da esfera passa a ser relativo ao centro do objeto.

Outros tipos de sólidos de colisão disponíveis: CollisionTube, CollisionInvSphere, CollisionPlane, CollisionPolygon, CollisionRay, CollisionLine, CollisonSegment e CollisonParabola [2].

Existe outro tipo de definição de objetos colidíveis, que é feita no arquivo do modelo.

5.6.3 Tratando as Colisões

Para cada colisão detectada, um objeto do tipo CollisionEntry é criado. Esse objeto carrega informações como os objetos *Into* e *From*, o ponto de colisão e o vetor normal da colisão. Essas entradas de colisão são tratadas por um tratador de colisão, um CollisionHandler. Esse objeto captura as entradas geradas e define o modo como elas serão tratadas. O tratador mais simples é o CollisionHandlerQueue, que consiste de uma fila onde as entradas

[2] Mais informações podem ser obtidas em https://www.panda3d.org/manual/index.php/Collision_Solids

são armazenadas, na ordem em que foram geradas. Um exemplo de criação e uso de uma fila:

```
fila = CollisionHandlerQueue()
...
for i in range(queue.getNumEntries()):
        entrada = queue.getEntry(i)
        # Faz alguma coisa
```

Código 5.3: Tratador colisões.

Existem vários tipos de tratadores de colisão, a saber: CollisionHandlerEvent, CollisionHandlerPusher, PhysicsCollisionHandler, CollisionHandlerFloor.

5.6.4 Registrando as Colisões

O objeto responsável por checar por colisões é o CollisionTraverser. Ele mantém a lista de objetos colidíveis e realiza a verificação de colisões entre os objetos. Todos os objetos no espaço, desde que possuam um CollisionSolid associado, são objetos *Into*, já que todos podem receber uma colisão. Para um objeto ser também um objeto *From*, é necessário registrá-lo no traverser. Para isso, é necessário também associar qual tratador de colisões será responsável por tratar as colisões geradas por esse objeto.

```
traverser = CollisionTraverser("nome no traverser")
traverser.addCollider(objetoFrom, handler)
```

Código 5.4: Registro de colisões.

A partir de agora, o traverser checará, a todo frame, se o objeto objetoFrom está colidindo com alguém no espaço.

5.7 Interval

Existe um mecanismo de transição suave de valores chamado Interval. Ele consiste de alterar o valor de uma variável continuamente, sem gerar mudanças bruscas, durante um determinado intervalo de tempo. Ele possui a seguinte assinatura simplificada: *LerpInterval(duração, valorFinal, valorInicial=None)*, onde Lerp é abreviação para linearly interpolate, ou interpolação linear. Esse mecanismo possui várias versões, sendo que a mais utilizada é LerpPosInterval, que varia a posição de um objeto.

Ao iniciá-lo, o objeto irá de *posicãoInicial* até *posiçãoFinal* em *duração* segundos. Porém, esse mecanismo possui algumas limitações. A principal delas é que, iniciado o intervalo, o objeto sempre terminará na posiçãoFinal.

Mais especificamente, se esse intervalo for interrompido antes do término, o objeto será imediatamente deslocado para posiçãoFinal. Isso é um problema, pois se no caminho entre um ponto e outro ocorre uma colisão, o objeto não mais se movimentará devido ao objeto colidido e ficará inativo durante todo o tempo restante do intervalo. Isso acontece com todas as versões do LerpInterval.

Para evitar esse problema foi necessário escrever um mecanismo semelhante ao LerpInterval mas mais flexível.

A variação continua de um valor indo do valor A para o valor B é dada pela expressão

$$A + t(B - A),\ 0 \leq t \leq 1$$

Para que essa variação seja suave, é necessário variar continuamente o valor de t. Isso é feito seguindo a seguinte relação genérica:

$$t = \frac{(tempoAtual - tempoInicial)}{(tempoFinal - tempoInicial)}$$

onde *tempoFinal = tempoinicial + numeroSegundos*. Ou seja, seguindo essa relação, é possível variar o valor de t, indo de 0 a 1 durante *numeroSegundos* segundos. Mas ainda é necessário obter constantemente valores para tempoAtual e calcular os valores de t sem bloquear a Thread principal. Para isso, esse cálculo é feito em uma Task, que é executada enquanto *t != 1* (ou *tempoAtual < tempoFinal*) e que, a cada execução, atualiza o valor de *tempoInicial*. Agora basta, a cada frame, atualizar o valor da variável, fazendo variável = *A + t(B - A)*.

Como a variação é feita manualmente, é possível interrompe-la a qualquer momento sem que o objeto fique travado ou que seu valor se altere bruscamente, bastando finalizar a execução da task.

No projeto, esse novo mecanismo foi utilizado no clareamento da tela quando um mini-game é encontrado. Essa mudança é feita inserindo uma neblina na tela e aumentando sua densidade, cujo valor varia entre 0 e 1. *SetUp.fog.setExpDensity(t)*, $0 \leq t \leq 1$.

5.8 O Formato *egg*

O Panda utiliza arquivos com extensão *egg* para carregar objetos e modelos gráficos que serão renderizados. Esse tipo de arquivo consiste de uma série de comandos seqüenciais e/ou hierárquicos que obedecem à seguinte forma genérica:

```
<comando> nome { conteúdo }
```

Código 5.5: Formato dos comandos do *egg*.

O arquivo também define as texturas aplicadas ao objeto:

```
<Texture> nome {
        nomeArquivo
        [opcoes]
}
```

Código 5.6: Textura no *egg*.

e as aplica em seu seus polígonos. Um polígono é definido por:

```
<Polygon> {
        <RGBA> { 1 1 1 1 }
        <TRef> { Textura1 }
        <VertexRef> { 1 2 3 <Ref> { vertices.verts } }
}
```

Código 5.7: Polígono no *egg*.

e o comando VertexRef faz referência a vértices definidos anteriormente:

```
<Vertex> 1 {
        x y z
        <UV> { u v }
        <Normal> { a b c }
}
```

Código 5.8: Vértice no *egg*.

Um modelo em *egg* pode ser feito manualmente ou com a ajuda de um software de modelagem gráfica compatível. A lista de softwares compatíveis é extensa mas os softwares recomendados, tanto por qualidade como por facilidade de exportar arquivos *egg* (via plugins disponibilizados pelo Panda), são Maya, 3D Studio Max e Blender.

6 Comunicação dos Jogadores

A comunicação entre os jogadores deve ser precisa, eficaz e eficiente, garantindo que toda a informação necessária é trocada e que o tráfego das mensagens não seja um problema no desempenho. A precisão das informações é importante para garantir a consistência do jogo.

A estrutura escolhida para a rede é baseada na arquitetura cliente/servidor. Um jogador, o criador da sala, faz o papel de servidor e fica aguardando pela comunicação dos outros jogadores, que fazem o papel dos clientes.

A escolha por utilizar uma arquitetura que centraliza o controle e a distribuição das mensagens exige uma atenção a mais para que não se crie um gargalo de informações no servidor. Toda a comunicação dos jogadores é baseada na troca de mensagens via socket.

Os jogadores não trocam as informações diretamente. Todas as mensagens são enviadas ao servidor que tem o conhecimento de todos os clientes conectados, sabe resolver o conteúdo da mensagem e a encaminha para o destinatário correto.

As mensagens podem ser separadas em dois tipos principais: mensagens de aviso e de posicionamento. O primeiro tipo de mensagem tem função unicamente informativa e tem aplicação, por exemplo, no momento em que um novo cliente se conecta, um novo mini-game começa ou um mini-game termina. As informações de posicionamento servem para que todos os jogadores saibam onde estão os seus adversários.

O Panda possui algumas classes e funcionalidade que simplificam a criação da rede. Dentre as implementações existentes, estão classes que auxiliam tanto no momento de estabelecer conexões (na arquitetura cliente/servidor) como para transmitir informações. As funcionalidades implementadas suportam o uso dos protocolos TCP e UDP.

6.1 Formato das Mensagens

A decisão sobre quais informações devem fazer parte da mensagem é outro ponto interessante. É importante manter o tamanho da mensagem pequeno para que o tráfego na rede não seja grande. Por outro lado é necessário que toda a informação relevante esteja de alguma forma na mensagem.

O tamanho da mensagem é importante também para reduzir o tempo necessário para processá-la (extrair o conteúdo e iniciar as ações necessárias) do lado de quem a recebe. Do lado de quem envia, é importante para reduzir o tempo necessário para criar o datagrama e enviá-lo pela rede.

O formato final da mensagem contém toda e somente a informação necessária. A mensagem está organizada com a seguinte estrutura:

- Um inteiro que faz o papel de um identificador para o tipo da mensagem;
- Um inteiro que representa o id do jogador que está enviando a mensagem;
- Um inteiro que representa o id do jogador deve receber a mensagem (ou o valor -1 para indicar que é uma mensagem que deve ser enviada a todos os jogadores);
- O conteúdo da mensagem.

A estrutura inicial da mensagem é bem fixa, enquanto a parte do conteúdo pode variar bastante. No caso de mensagens informativas, apenas o identificador da mensagem é necessário, especificando qual ação tomar. No caso de mensagens de posicionamento, o conteúdo da mensagem contém as teclas que vão sendo pressionadas pelo jogador, de forma que a posição do personagem no jogo possa ser calculada.

6.2 Tráfego de Mensagens

O tráfego das mensagens na rede é um ponto que merece uma atenção especial. A grande quantidade de informações trafegando na rede pode criar um gargalo no servidor, fazendo com que a distribuição das mensagens comece a atrasar e conseqüentemente pequenos travamentos começam a ocorrer no jogo. Outro problema que pode ser conseqüência desse atraso é a imprecisão das informações recebidas; o maior tempo entre o envio e recebimento da mensagem faz com que nem sempre o que está sendo exibido na tela do jogador reflita o estado atual do jogo.

Com o objetivo de amenizar esse problema, alguns testes sugerem a solução que deve ser adotada em cada situação específica. No projeto, a solução final encontrada foi enviar mensagens somente quando um dos jogadores pressionava ou soltava uma tecla, com mensagens de correção da sincronia em intervalos regulares.

A solução ideal parecia ser enviar as mensagens com a posição e orientação do jogador em intervalos de tempo regulares, mas restava um pequeno atraso no movimento do personagem remoto e a dificuldade de controlar a animação do personagem dentro do jogo. A estrutura do Panda que é responsável por dados dois pontos, criar a animação do personagem indo de um ponto ao outro, não consegue tratar as colisões que ocorrem nesse trajeto. Assim, os personagens deixavam de colidir uns com os outros e também com o cenário.

A solução de enviar somente as informações sobre as teclas pressionadas e fazendo o envio somente no momento em que uma nova tecla era apertada ou liberada acabava com o problema da animação, já que utilizaria o mesmo do personagem local. Essa solução não cria um gargalo no servidor e o tempo de resposta é excelente, não prejudicando a precisão nem a eficiência da comunicação.

O problema dessa solução é que em certas situações ela gera uma diferença mínima entre a posição real do personagem e posição remota. A propagação desse pequeno erro, porém, causa grandes efeitos em longo prazo e dá a impressão de não haver sincronização entre a posição do personagem nos clientes.

A causa dessa pequena diferença é o modo como se trata os eventos de teclas. Localmente, esse evento é processado em paralelo ao restante do jogo, inclusive a outros eventos de tecla, enquanto remotamente essa operação torna-se seqüencial. Assim, a situação onde esse erro aparece é quando o jogador pressiona e solta rapidamente uma tecla.

Localmente, essa ação de pressionar e soltar rapidamente a tecla gera dois eventos que serão processados em paralelo. A conseqüência disso é que o tempo em que a tecla pressionada será marcada como ativa e logo em seguida como inativa pode ser menor do que o tempo que leva para o jogo passar de um frame para o outro. O resultado é que a ação causada por essa rápida ação pode não existir.

Já remotamente, no momento em que a tecla é pressiona, um pacote é gerado e enviado pela rede com essa ação, em seguida, quando a tecla é liberada, outro pacote é criado e enviado com a nova ação. O cliente que recebe essa mensagem irá processar os pacotes de forma seqüencial, portanto irá gerar o resultado da primeira ação e em seguida o da segunda. A conseqüência disso é que a primeira ação foi executada, mesmo que rapidamente, gerando uma alteração na posição ou na orientação do personagem.

O fato de localmente a ação não ter gerado nenhum efeito e remotamente a ação ter existido, mesmo que minimamente, faz com que nesse momento apareça a diferença entre as posições ou orientações. Com o decorrer do jogo, a combinação desses pequenos erros leva a uma grande diferença nas posições dos personagens.

A solução mais simples para resolver esse problema é enviar pela rede, em intervalos de tempos regulares, a posição e a orientação do personagem. Esse intervalo de tempo não pode ser muito curto para não sobrecarregar a rede, mas também não pode ser muito longo para que não existam saltos entre as posições do personagem em razão da correção.

A solução final combina o envio das teclas com o envio periódico da orientação e da posição de cada personagem para sincronizá-los. O segundo tipo de mensagem é enviado em intervalos de 0.1 segundos. Apesar de essa solução aumentar o tráfego na rede, a quantidade total de mensagens não é um problema.

7 Estrutura do Código

O problema da estrutura do código está em criar e manter um código de qualidade, bem estruturado e intuitivo. Além de ser um critério fundamental para auxiliar no requisito de extensibilidade do projeto.

A qualidade do código se relaciona com conseguir manter um código simples e bem modelado. As funcionalidades devem ter papéis bem definidos e separados, além de serem independentes umas das outras, em termos de implementação. As classes devem ser desacopladas entre si e cada uma ter uma responsabilidade específica.

Esses requisitos podem ser atingidos focando na organização e na clareza do código. A utilização de diagramas para projetar o que será desenvolvido também ajuda nesse processo.

Outra tentativa para elevar a qualidade do código é aplicar idéias para maximizar a coesão e minimizar o acoplamento entre as classes. Coesão refere-se à diversidade de responsabilidades associadas a uma classe; quanto menor a diversidade, maior a coesão. Acoplamento entre classes refere-se ao quanto duas classes são dependentes uma da outra; é uma medida do quanto uma classe conhece as outras.

A solução para maximizar a coesão é criar classes que representam apenas uma abstração e onde cada atributo descreve uma instância da classe. O problema do acoplamento pode ser resolvido com a idéia de criar classes que dependam somente da interface das outras e não da implementação.

Apesar de simples, esses detalhes tornam o código mais simples, fácil de entender, modificar, estender e reutilizar.

O critério de extensibilidade visa não somente ter um projeto passível de ser estendido, mas um projeto pensado, estruturado e preparado para tal. A inclusão de funcionalidades ou novos mini-games deve ser simples, fácil e não exigir muitos esforços por parte do desenvolvedor.

7.1 Design Patterns

Na busca por um código de qualidade, a aplicação de padrões de design orientados a objetos aparece como uma solução bem interessante. Os padrões criam um vocabulário comum entre os engenheiros de software.

O uso de padrões permite elevar o nível de abstração do sistema e possibilita discutir de um nível mais alto a estrutura do projeto. Os padrões descrevem bases de soluções para problemas que ocorrem repetidamente.

O projeto aplicou padrões visando o critério de extensibilidade. Alguns dos padrões utilizados no desenvolvimento serão detalhados a seguir.

O *Singleton* é um padrão de criação que garante que será criada somente uma instância de uma determinada classe e esta pode ser facilmente acessível de qualquer ponto do programa. No projeto, o padrão *Singleton* foi utilizado no cenário, por exemplo. O cenário é sempre único, em qualquer momento da execução do jogo e deve estar facilmente acessível.

```
class Cenario() :

        _instancia = None

        def __init__ (self) :
                if Cenario._instancia :
                        raise Cenario._instancia
                Cenario._instancia = self
```

Código 7.1: *Singleton*.

O *Template Method* é um padrão comportamental que define um método modelo. Normalmente, é um método concreto em uma classe abstrata em que cada passo do algoritmo é definido por outro método abstrato. O uso desse padrão permite que somente um passo do método seja facilmente alterado sobrescrevendo um dos métodos que compõe o *Template Method*. No projeto, o *Template Method* foi utilizado para definir o método principal da classe abstrata *Mini-Game*. As subclasses, que representam os mini-games concretos, é que são responsáveis por definir cada um dos métodos e assim determinar o comportamento do mini-game.

```
class MiniGame () :

        def __init__ (self) :
                self.emAndamento = True

        def run (self):
                self.carrega()
                self.inicia()
                self.mainLoop = taskMgr.add(self.loop, "loop")
                while self.emAndamento :
                        self.mainLoop.step()
                self.finaliza()
                self.limpa()

        @abstractmethod
        def carrega (self):
                raise NotImplementedError

        @abstractmethod
        def inicia (self):
                raise NotImplementedError

        @abstractmethod
        def finaliza (self):
                raise NotImplementedError

        @abstractmethod
        def loop (self) :
        raise NotImplementedError

        @abstractmethod
        def limpa (self) :
                raise NotImplementedError
```

Código 7.2: *Template Method.*

O *Memento* é um padrão comportamental que captura e externaliza o estado interno de um objeto sem violar o encapsulamento. Assim, garante que este estado possa ser recuperado posteriormente. No projeto, o *Memento* é aplicado quando o jogador faz a transição entre o jogo base e um mini-game qualquer. No momento em que o mini-game tem início o estado do jogo base é capturado e armazenado. Assim, ao final do mini-game, este estado é restaurado e o jogo continua de onde havia parado.

```
def transicaoMiniGame(self):
        self.estado = EstadoMemento.salvaEstado(self.jogadores)
        ...

def voltaParaJogoBase(self):
        ...
        EstadoMemento.recuperaEstado(self.jogadores, self.estado)
```

Código 7.3: *Memento*.

O *Observer* é um padrão comportamental que define uma relação de dependência entre objetos de forma que quando o estado de um objeto muda, todos os seus dependentes são notificados. No projeto, o *Observer* aparece no momento em que um mini-game é iniciado, por exemplo. Quando um jogador encontra um mini-game o estado do jogo base é alterado e todos os dependentes, no caso os jogadores, precisam ser avisados.

O *Listener* é uma implementação específica do padrão *Observer* e no projeto está presente principalmente nas classes que representam o *Cliente* e o *Servidor*.

O *Command Dispatch* é um padrão comportamental exclusivamente para linguagens dinâmicas que encapsula um comando como um objeto. No projeto, o *Command Dispatch* foi aplicado na classe *Menu* para definir a rotação para o menu correto sem necessitar de criar vários condicionais para tal. O fato de a linguagem ser dinâmica permite chamar diretamente o método associado ao comando.

```
class Menu () :

def __init__ (self, main) :
        ...
        self.opcoesMenu = {
                SAIR_APRESENTACAO   : self.goToPrincipal,
                INICIAR_PRINCIPAL   : self.goToIniciar,
                OPCOES_PRINCIPAL    : self.goToOpcoes,
                SAIR_PRINCIPAL      : self.finaliza,
                CREDITOS_OPCOES     : self.goToCreditos,
                SOM_OPCOES          : self.finaliza,
                VOLTAR_OPCOES       : self.goToPrincipal,
                CRIAR_INICIAR       : self.finaliza,
                PROCURAR_INICIAR    : self.finaliza,
                VOLTAR_INICIAR      : self.goToBackPrincipal,
                VOLTAR_CREDITOS     : self.goToOpcoes,
        }
        ...

def goTo (self, menu) :
        func = self.opcoesMenu[menu]
        func()

def goToPrincipal (self) :
        self.menu.hprInterval(1, (0, 90, -90)).start()
        self.menuAtual = MENU_PRINCIPAL

def goToOpcoes (self) :
        self.menu.hprInterval(1, (0, 180, -90)).start()
        self.menuAtual = MENU_OPCOES

def goToBackPrincipal (self) :
        self.menu.hprInterval(1, (0, 90, -90)).start()
        self.menuAtual = MENU_PRINCIPAL

def goToIniciar (self) :
        self.menu.hprInterval(1, (0, 0, -90)).start()
        self.menuAtual = MENU_INICIAR

def goToCreditos (self) :
        self.menu.hprInterval(1, (0, 270, -90)).start()
        self.menuAtual = MENU_CREDITOS
```

Código 7.4: *Command Dispatch.*

8 Conceitos Envolvidos

8.1 Jogadores não-humanos

O problema dos jogadores não-humanos está em desenvolver comportamentos inteligentes com uma baixa quantidade de processamento. O jogador não-humano deve simular o comportamento de um jogador humano; para isso ele deve saber tomar as atitudes adequadas em cada situação de forma simples.

O desenvolvimento desse jogador faz uso de técnicas de inteligência artificial para a tomada de decisão. A idéia é criar um agente inteligente que faça a melhor escolha localmente. Dessa forma o processamento final é simplificado já que somente as situações vizinhas precisam ser analisadas.

Apesar de o agente tomar a melhor decisão local, esta escolha é baseada em uma função objetivo global e tenta minimizar o valor dessa função. Dessa forma, a decisão do agente pode não ser a melhor decisão global, mas se aproxima de forma satisfatória dela.

8.1.1 Grafos e Busca em Largura

O primeiro passo para a criação do agente é levar o modelo contínuo do mapa para o mundo discreto. Nesse mapeamento, o movimento do agente é baseado em uma estrutura 4-conexa.

A representação do mapa no mini-game foi feita utilizando um grafo. Cada vértice do grafo representa uma posição no mapa real e as arestas representam uma vizinhança no mapa real e que não existe uma parede entre elas. Todas as arestas são bi-direcionais.

Inicialmente, o agente conhece o mapa e a menor distância partindo de cada um dos vértices até o buraco onde a bolinha deve ser levada. Para tal, o pré-processamento do mini-game inclui uma busca em largura para encontrar os melhores caminhos.

8.1.2 Subida de Encostas

O algoritmo escolhido para a tomada de decisão no mini-game foi o de *subida de encosta*, também conhecido por *subida da montanha* ou *hill climbing*. Essa é uma estratégia simples e popular de busca heurística baseada na busca em profundidade. É um algoritmo de busca local que não tem interesse no caminho percorrido, apenas na configuração final.

A idéia heurística por trás do algoritmo é que o número de passos para atingir um objetivo é inversamente proporcional ao tamanho deste passo. Assim, a decisão busca escolher o estado que leva para mais perto da solução. A implementação feita é baseada no *subida de encosta pela trilha mais íngreme*, que examina todos os vizinhos de um estado e escolhe aquele que está mais próximo da solução.

O algoritmo consiste em uma repetição que percorre o espaço de estados no sentido do valor decrescente e termina quando encontra um vale em que nenhum vizinho tem um valor mais baixo. O pseudocódigo é o seguinte:

```
SubidaDeEncosta (problema)
        estadoAtual <- estadoInicial(problema)
        enquanto true
                proxEstado <- melhorVizinho(estadoAtual)
                se valor(proxEstado) >= valor(estadoAtual)
                        então retorna estadoAtual
                estadoAtual <- proxEstado
```

Código 8.1: *Subida de Encosta.*

Algumas das vantagens desse algoritmo estão na simplicidade de processamento e na utilização de espaço de armazenamento. Por ser um algoritmo sem memória, as transições são Markovianas, nenhuma informação sobre o caminho percorrido precisa ser guardada, apenas o estado atual e o valor da função objetivo são armazenados, fazendo com que tenha um baixo consumo de memória. Em relação ao processamento, é simples porque não examina antecipadamente nenhum estado além dos vizinhos imediatos.

Esse algoritmo possui três situações em que não obtém sucesso: máximos (ou mínimos) locais, platôs e cordilheiras. Na primeira situação, dependendo do estado inicial, é possível que o algoritmo fique preso em máximos (ou mínimos) locais e não chegar a estados melhores e distantes; no caso de platôs pode-se encontrar uma área onde todos os vizinhos têm o mesmo valor e assim a escolha da melhor direção não pode ser feita localmente; por último, é uma região do espaço mais alta que os vizinhos e que não é possível de ser atravessada em um único passo.

Apesar de esses problemas serem bastante graves, no caso do mini-game não é preciso se preocupar diretamente com nenhum deles. O fato de o agente participar do jogo em conjunto com os jogadores humanos faz com que situações como essa sejam apenas momentâneas. A interação dos outros jogadores rapidamente tira a bolinha de situações que poderiam deixar o agente sem saber o que fazer.

8.2 Modelagem Física

O problema da modelagem física está relacionado com a tentativa de aproximar o comportamento dos fenômenos físicos presentes no jogo do comportamento real. Para isso, a modelagem requer muito cuidado na aplicação de conceitos de física.

8.2.1 Movimento da Plataforma

No mini-game, o movimento da plataforma é definido de acordo com a posição de cada um dos jogadores. O cálculo considera a força peso atuando sobre cada jogador e calcula a força resultante para determinar a inclinação da plataforma.

A força resultante é calculada sobre o eixo de movimento da plataforma, que está localizado em seu centro. Por isso, a contribuição do peso de um jogador mais distante do centro é maior do que a de um jogador próximo.

Formalmente, dados n jogadores, seja P_i a posição o i-ésimo jogador em relação ao centro da plataforma. Considerando que todos os jogadores têm o mesmo peso, a força resultante F é o vetor cuja direção é dada por:

$$F = P_1 + P_2 + \cdots + P_n = \sum_{n}^{i=1} P_i$$

8.2.2 Inércia

A inércia é uma propriedade que diz que a velocidade de um corpo não é alterada se nenhuma força externa ao sistema agir sobre ele ou se a resultante das forças agindo for nula. Como conseqüência, se o corpo está parado, ele continuará parado e se estiver em movimento, continuará em movimento e sua velocidade permanecerá constante.

No caso do mini-game, a inércia está presente no movimento da bolinha sobre a plataforma. A única força externa agindo é a resultante dos pesos dos jogadores. Quando a resultante é não nula, a velocidade da bolinha aumenta de acordo com a inclinação causada na plataforma.

A ação da força sobre a bolinha parada simplesmente faz com que a bolinha passe a se mover na direção da inclinação. No caso em que a bolinha já está em movimento e a direção da inclinação não se altera, o movimento da bolinha não sofre alteração no sentido, mas altera a velocidade (aumentando caso a inclinação aumente ou reduzindo caso a inclinação diminua). Por último, no caso em que a bolinha está em movimento e a direção da inclinação

é alterada, a bolinha vai aos poucos perdendo velocidade no sentido do movimento antigo e ganhando velocidade no sentido da nova inclinação, fazendo com que o movimento aparente se aproxime cada vez mais da inclinação atual da plataforma.

8.2.3 Colisões Inelásticas

A colisão inelástica ocorre quando a energia cinética não se conserva após a colisão. No caso do mini-game, pode-se observar esta propriedade na colisão entre a bolinha e as paredes do labirinto.

Quando a bolinha colide com a parede, a velocidade após a colisão é inferior à velocidade inicial. Nessa situação, o momento do sistema se conserva, mas a alteração na velocidade da bolinha indica uma variação na energia cinética do sistema. Parte dessa energia cinética é convertida em outros tipos de energia, como interna e térmica, que não foram aproveitadas.

8.2.4 Rotação da Bolinha

A rotação da bolinha em 3D é a modelagem física mais delicada do projeto. A modelagem da rotação necessita das especificações da posição e da orientação da bolinha. Especificar a posição é relativamente simples, utilizando coordenadas cartesianas ou translações em relação a uma origem conhecida. Especificar a orientação, porém, não é trivial.

Inicialmente, a orientação poderia ser dada como rotações em relação a uma origem conhecida. Porém, não é assim tão simples. Por exemplo, porque as rotações tridimensionais não comutam, implicando que a ordem de aplicação das operações nos diferentes eixos afeta o resultado final. Não é impossível representar as rotações utilizando os eixos cartesianos e ângulos, porém uma solução mais simples foi adotada: aplicar *quatérnios*.

Quatérnio é uma ramificação da matemática que generaliza o cálculo vetorial e os números complexos. Um quatérnio de rotação é determinado pelo cosseno do ângulo de rotação (parte real) e o eixo de rotação (parte imaginária).

$$q = (\cos\theta, (e_x, e_y, e_z))$$

A representação de um ponto $\vec{r}$ sobre o qual será aplicada uma rotação será um quatérnio com parte real nula, $p = (0, \vec{r})$. A rotação será representada por um quatérnio unitário $q = (s, \vec{v})$. O resultado da rotação de p por q é dado por:

$$\underline{R_q(p) = qp}$$
$$q$$

Outra vantagem da representação utilizando quatérnios é que a própria álgebra dos quatérnios realiza naturalmente a composição de rotações.

O Panda possui recursos que implementam a interpolação de orientações. Assim, dadas duas orientações representadas por quatérnios, o Panda determina os quatérnios que representam a seqüência intermediária de orientações.

9 Organização do Código

A estrutura do código é melhor entendida quando se conhece a responsabilidade de cada uma das classes. A seguir está brevemente comentada a função de cada classe do jogo.

As classes *Cliente* e *Servidor* representam a base da comunicação em rede. Entre as responsabilidades delas está o estabelecimento e manutenção da conexão e a criação e tratamento do envio e recebimento de mensagens.

As telas iniciais de escolha entre cliente ou servidor são representadas pelas classes *TelaProcura* e *TelaCria*, respectivamente. A classe *Teclado* auxilia na captura da entrada de dados do usuário. As demais telas do menu estão na classe *Menu*.

Os personagens do jogo são representados pela classe *Personagem*, que é genérica para todos os tipos de jogadores e define a movimentação e o tratamento das colisões. As implementações específicas do jogador presente, jogadores remotos e agentes inteligentes são definidas respectivamente pelas classes *Jogador*, *Inimigo* e *Agente*.

A classe *Fila* é a representação de uma estrutura do tipo fila que é utilizada pelo grafo, definido na classe *Grafo*, que serve para a tomada de decisão do agente.

A classe *Cenario* representa uma abstração para os cenários do jogo e a classe *Labirinto* é a instância que representa o cenário do jogo base. As definições da câmera e sua movimentação, que segue o personagem, estão na classe *Camera*.

A abstração de um mini-game é representada pela classe *MiniGame*, que apresenta os métodos necessários para a criação de um mini-game. Na classe *MesaMovel* está a implementação de um mini-game de exemplo.

A classe *EstadoMemento* armazena o estado do jogo base enquanto há um mini-game em andamento e a classe *SetUp* é responsável pelas definições iniciais das teclas, luz, câmera, posição das entradas para os mini-games e os efeitos do jogo base. Por último, a classe *Main* é a classe principal do jogo, iniciando as estruturas, coordenando o laço principal onde os personagens são movimentados, a comunicação entre as classes do jogo e as transições entre jogo base e mini-games.