Capítulo 3

Processo de construção de um MMORPG com Python

Roberto Fagá Jr, Isadora Trombeta Marques Calderan

Abstract

Building a MMORPG game is a long and hard task, as it uses different concepts from many areas. This chapter shows the construction process of a simple game project, helping in building a MMORPG client game. The process uses computer theories such as Oriented Object Programing and Networks, and also uses the language Python to create games.

Resumo

Construir um MMORPG é uma árdua e longa tarefa, por conter aspectos diferentes de tantas áreas. Este capítulo aborda a construção de um projeto simples dessa categoria de jogos, com o objetivo de ensinar a construção de um jogo, e de utilizar técnicas teóricas da computação como Programação Orientada a Objetos e Redes de Computadores, além do uso da plataforma Python para construção de jogos.

3.1. Introdução e Software Livre para desenvolvimento de jogos

Os jogos têm evoluído em sua complexidade e esforço de produção, sendo que diversos jogos comerciais atualmente demandam esforço maior do que a produção de filmes de Hollywood. Esforço que pode ser maior para a modalidade de jogos MMOG - *Multiplayer Massive Online Games*, na qual muitos jogadores participam jogando em um mesmo ambiente, criando mundos virtuais para milhares de jogadores.

A área de criação de jogos é multidisciplinar e demanda esforços de diversas áreas, como *Design* Gráfico, Computação Gráfica e Inteligência Artificial. Na realidade, o desenvolvimento de um jogo atualmente requer profissionais de diferentes áreas, como roteiristas ou mesmo profissionais especialistas em Computação Distribuída.

3.1.1. Jogos com Python

A linguagem Python possui uma sintaxe simplificada que facilita sua aprendizagem, sendo uma linguagem completa e de escrita rápida. Para a construção do jogo poderiam ser utilizadas diversas outras bibliotecas para jogos em Python, muitas de excelente qualidade, como: Pyglet (Holkner, 2010), Panda3D (The Panda3D development team, 2010), Soya (SOYA3D, 2010), PyOgre (PYTHON-OGRE, 2010) e PyODE (PYODE, 2010), apresentadas na Tabela 3.1.

Biblioteca	Características	Dependências	Binding
Pygame	Utiliza SDL com algumas funções adicionais para gráficos 2D	SDL	SDL
Pyglet	Utiliza OpenGL com o propósito de uma biblioteca para gráficos 2D, que pode integrar com 3D	OpenGL, LibAV caso utilize vídeos	OpenGL
Panda3D	É uma engine 3D escrita em C++, usada comercialmente com suporte a shader	Panda3D em C++	Panda3D em C++
Soya3D	Engine 3D pura em Python, que utiliza diversas bibliotecas em código nativo	OpenGL, OpenAL, GLEW, SDL, Cal3D, PIL, ODE	-
PyOgre	Bem utilizada em Computação Gráfica, utiliza OpenGL e DirectX	OGRE, OpenGL e/ou DirectX	OGRE
PyODE	Engine para cálculos de física em geral	ODE	ODE

Tabela 3.1. Comparação de bibliotecas para jogos em Python.

A biblioteca que será utilizada durante o desenvolvimento será a Pygame (PYGAME, 2010e), pois apresenta uma abordagem simples para produção de jogos, utilizando muitos recursos da linguagem Python. O uso de outras bibliotecas das apresentadas acima e de outras encontradas pela Internet torna o projeto o quão completo quanto for preciso.

3.1.2. Software Livre para jogos

As tecnologias utilizadas neste capítulo são Software Livre, um aspecto interessante para a produção de jogos e desenvolvimento de aplicações. O uso de Software Livre em jogos sofre algumas resistências, pois em geral o lucro financeiro das empresas de jogos são exclusivamente dos usuários com a venda de Software, enquanto que empresas de aplicações em Software Livre têm seu lucro distribuído entre usuários e outras empresas, além dele nem sempre estar vinculado à venda do produto, mas sim ao suporte ou à manutenção.

Analisado sob a perspectiva de compartilhamento de conhecimento, o uso

de Software Livre para jogos é positivo, pois a demanda de tecnologias e algoritmos necessários para a produção de um jogo é muito elevada. As tecnologias livres têm grande importância para reduzir o custo da produção de um jogo, para melhorar a qualidade com tecnologias modernas e para compartilhar o conhecimento e as técnicas de criação de jogos.

Outra motivação para uso de Software Livre é que a produção de jogos comerciais do gênero MMORPG são de alto custo, passando dos 10 milhões de dólares (Carpenter, 2010). Investimentos de alto custo para produção de jogos, considerando o cenário brasileiro, não são comuns, sendo então um dos caminhos para a produção de jogos com qualidade e com custo reduzido o uso de tecnologias livres.

3.1.3. Começando do básico

Muitos interessados na área de desenvolvimento de jogos acabam pretendendo projetos grandiosos logo de início, como um MMORPG! Na verdade, o gênero não influencia totalmente na complexidade do projeto, mas sim as características de um jogo. Existem centenas, milhares de MMORPGs e outros jogos do tipo *massive*, como os listados na Web [1], por que então fazer somente mais um?

Antes de começar um projeto complexo e grandioso, é muito interessante (e motivador) começar um projeto simples, ainda que seja usado somente de aprendizado. Neste capítulo será apresentado o processo para construção de um MMORPG simplificado que, apesar de utilizar uma *engine* 2D (o coração do jogo) e não possuir muitos recursos dos atuais MMOs, possui uma estrutura básica que permite o entendimento dos jogos dessa categoria.

Este capítulo trata da produção de um jogo MMORPG - *Multiplayer Massive Online Role Playing Game*, gênero de jogos que criam mundos virtuais para que os jogadores vivenciem personagens nesses mundos. O jogo é composto de cliente e servidor, sendo detalhada neste capítulo apenas a construção do cliente, pois é a que mais utiliza recursos de desenvolvimento de jogos. O kit inicial pode ser baixado em <http://pygame-mmorpg.googlecode.com/files/curso.zip>, juntamente com os aquivos fonte passo a passo.

A Seção 3.2 apresenta uma breve introdução a linguagem Python, que pode ser ignorada para quem já conhece a linguagem. As Seções 3.3 à 3.7 apresentam o passo a passo da construção do cliente de jogo. O capítulo finaliza-se com aprofundamentos na biblioteca Pygame e um exemplo prático de integração com redes sociais na Seção 3.8.

3.2. Conhecendo Python

Considerando-se que o aprendizado de algoritmos escritos em linguagem natural, como em português, normalmente antecede o aprendizado de linguagens de programação, é muito mais comum pensar em programas nos algoritmos em si.

[1] Uma lista com diversos MMOs pode ser encontrada em <http://mmohut.com/gamelist>, Acesso em 19 Fev. 2010

Algumas linguagens acabam tendo grande tempo de aprendizado pela distância entre uma linguagem próxima a uma natural e a sintaxe da mesma. A sintaxe da linguagem Python é muito semelhante a uma linguagem natural em formato de algoritmo, já que dispensa a utilização de chaves para demarcação de blocos, o uso de ponto-e-vírgula para fim de linha, dentre outros fatores que serão expostos nesta seção.

Python é uma plataforma de programação voltada para a agilidade de produção de aplicações, contando com uma sintaxe simplificada e orientada a objetos (Python Foundation, 2010). Muitas vezes é classificada como uma linguagem de script [2] por ser ágil, mas têm todos os recursos para a construção de grandes aplicações (Reis, 2004). A plataforma bem como a maior parte das bibliotecas utilizadas neste capítulo é Software Livre, o que garante a economia e a qualidade do código.

Lutz (2003) citam algumas vantagens da plataforma Python, dentre elas:

- **Qualidade de Software:** O código Python foi desenvolvido para ser próximo a linguagem natural, além de ser simplificado. Por isso, a linguagem se torna fácil para manutenção e alterações.

- **Produtividade:** O código fica em tamanho reduzido se comparado a código Java ou C, por consequência é necessário menos esforço para escrever, corrigir erros ou mantê-lo.

- **Portabilidade:** Assim como outras linguagens multiplataforma, um código Python funciona em diferentes plataformas, sendo possível rodar um mesmo código em Windows, Linux ou dispositivos móveis.

- **Bibliotecas:** Uma das vantagens em relação a outras linguagens ágeis (ou de *script*) é a grande coleção de bibliotecas que a plataforma Python tem. É possível encontrar bibliotecas que fazem quase todas as tarefas que são necessárias para uma aplicação, como é o caso de bibliotecas para jogos por exemplo. Muitas bibliotecas são apenas *bindings*, ou seja, apenas mapeiam funções para bibliotecas em código nativo do sistema operacional, garantindo ótimo desempenho.

- **Componentes de integração:** É possível integrar código Python com outras linguagens e tecnologias, como C, C++, Java, .NET, Corba ou também utilizar interfaces como SOAP. Muitas aplicações utilizam abordagens híbridas, como em jogos, em que muitas vezes o desenvolvimento da parte crítica do jogo, que consome de fato muito processamento, é realizada com linguagens compiladas nativamente como C e C++, e todo o restante que não utiliza grande processamento é feito com linguagens ágeis, como Python.

[2] Linguagens de script são normalmente assim chamadas por serem utilizadas para conectar aplicações ou automatizar tarefas.

A identação, ou recuo que é realizado no código fonte para demarcar um bloco, é necessária em Python pois apenas ela demarca quando inicia e termina um bloco de alguma estrutura, como um if ou um for, bem como funções. Também não há necessidade de declaração de variáveis, ou seja, não é necessário determinar previamente o tipo das mesmas.

O tipo da **variável** é definido no momento de sua atribuição, de acordo com o tipo do valor a ser recebido (se é um texto, a variável é então do tipo texto, se número inteiro, é do tipo inteiro, e assim por diante). Diferentemente de outras linguagens, essas variáveis tratam-se apenas de ponteiros para objetos, o que permite que a uma mesma variável possa ser atribuído um texto em um momento e em outro ser atribuído um número inteiro.

A seguir é apresentado um breve resumo de algumas características da linguagem, na prática.

3.2.1. Terminal interativo e primeiros passos

Além da compilação de código fonte, arquivos de extensão ".py", é possível executar python em um teminal interativo. O terminal (interpreter interface no Linux, Python command line no Windows) possibilita que a cada linha de código Python digitada, o sistema já o compile e execute na máquina, possibilitando correção de erros e aprendizado enquanto se desenvolve um código. A Figura 3.1 mostra o terminal sendo inicializado.

```
[faga@fagahm ~]$ python
Python 2.6.4 (r264:75706, Jan 25 2010, 08:55:26)
[GCC 4.4.2 20091208 (prerelease)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Figura 3.1. Terminal interativo do Python no Linux.

Python considera o que é digitado para o tipo do objeto, podendo trazer diferentes resultados para os operadores aritméticos comuns. Então o tipo numérico pode ser inteiro, *float/double*, ou cadeia de caracteres (*string*) se digitado entre aspas simples ou duplas:

```
>>> 7+2
9
>>> 8*4-2
30
>>> 7/2
3
>>> 7/2.0
3.5
>>> 7%2
1
>>> variavel = "abc"*2
>>> variavel
'abcabc'
>>> "abc"+"def"
'abcdef'
```

As cadeias de caracteres também podem ser somadas e multiplicadas. É possível notar que o objeto "variavel" assumiu ser do tipo *string* no momento da atribuição, quando também começou a existir.

Existem ainda os operadores de comparação, que comparam dois objetos de acordo com alguma relação. Além dos tradicionais, destacam-se os de lógica (negação, ***not***, e ***and***, ou ***or***), de igualdade absoluta ***is*** e de pertencer ao conjunto do outro objeto ***in***:

```
>>> a = 4
>>> b = 7*4
>>> a == b
False
>>> a > b
False
>>> 0 <= 0
True
>>> a == b/7
True
>>> "abc" == "abC"
False
>>> "abc" == "abc"
True
>>>
>>> "abc" is "abd"
False
>>> "abc" is "abc"
True
>>> "abc"*50 is "abc"*50
False
>>> "abc"*50 == "abc"*50
True
>>> "abc" in "abcabc"
True
>>> "abc" not in "and"
True
>>> "abc" == "abd" or 1 == 1 and True
True
```

O comparador de igualdade absoluta ***is*** compara se os objetos são de mesma instância, apontam para o mesmo endereço de memória, ao contrário do comparador de igualdade de valor **==**. Isso pode ser notado quando comparada *strings* de mesmo valor mas de comprimento grande, pois a plataforma Python instancia objetos diferentes nesse caso.

Os tipos *True* e *False* (note a primeira letra maiúscula) denominam o tipo booleano verdadeiro ou falso. Outro importante "operador" é o **#**, usado para comentários.

3.2.2. Funções e módulos

Existem quatro funções fundamentais para quem está iniciando em Python: ***print***, para imprimir na tela (permite também que não seja usado parênteses para passar os parâmetros), ***type***, para determinar o tipo atual do objeto, ***dir***, para exibir os métodos de um objeto, e ***help***, para exibir documentação da classe do objeto:

```
>>> print "abc",1
abc 1
>>> type(1)
<type 'int'>
```

```
>>> type("abc")
<type 'str'>
>>> dir('abc')
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__',
'__getslice__', '__gt__', '__hash__', '__init__', '__le__', '__len__', '__lt__',
'__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '_formatter_field_name_split', '_formatter_parser',
'capitalize', 'center', 'count', 'decode', 'encode', 'endswith', 'expandtabs',
 'find', 'format', 'index', 'isalnum', 'isalpha', 'isdigit', 'islower',
 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip',
'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit',
 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title',
 'translate', 'upper', 'zfill']
>>> 'abc'.find('b')
1
```

Os métodos podem ser acessados com uso do "." (ponto), e os métodos com dois *underscores* nas extremidades são privados, por convenção ("__"). Para se importar módulos, usa-se o ***import*** seguido do nome do módulo desejado:

```
>>> import os
>>> os.listdir('.')
['tango-icon-theme-extras', 'tango-icon-theme']
```

3.2.3. Estruturas de dados e de controle

Além dos tipos básicos (inteiro, ponto flutuante, booleano e string), Python também conta com alguns tipos de dados especiais. Enquanto em outras linguagens de programação costuma-se utilizar vetores para fazer listas de elementos, Python possui três tipos "nativos" para tal finalidade: listas, tuplas e dicionários. Os três tipos podem armazenar quaisquer tipos de objetos, podendo até mesmo uma mesma lista ter diferentes tipos de objetos.

Cada tipo de lista possui um modo diferente de se representar, e enquanto que as listas (representadas por **[]**) e tuplas (representadas por **()**) operam como coleções de objetos, os dicionários traduzem um objeto em outro (representados por {}). As listas são coleções dinâmicas, as tuplas são coleções estáticas:

```
>>> lista = [1, "dois", 3.0, [4, 5]]
>>> lista[2]
3.0
>>> outralista = [999, 7777]
>>> lista + outralista
[1, 'dois', 3.0, [4, 5], 999, 7777]
>>> lista * 2
[1, 'dois', 3.0, [4, 5], 1, 'dois', 3.0, [4, 5]]
>>> lista += [1]
>>> lista
[1, 'dois', 3.0, [4, 5], 1]
>>> lista[2] = 1
>>> tupla = (1,2,3)
>>> tupla += (1)
>>> tupla[2] = 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> dicionario = {'a':1, 'b':2}
>>> dicionario['a'] + dicionario['b']
3
```

```
>>> len(lista)
5
```

Uma função padrão do Python utilizada para essas coleções é a ***len***, que retorna a quantidade de objetos. É possível notar no código acima também que a recursão é possível, ou seja, é permitido que listas tenham listas como objetos. Outro ponto é o erro retornado quando se tenta alterar um valor de uma tupla.

Existem três principais estruturas de controle em Python usadas neste capítulo: ***if*** / ***else*** / ***elif***, ***while*** e ***for***.

A identação em Python deve ser realizada obrigatoriamente para demarcar o bloco, como por exemplo identar dentro de um ***if*** para executar o código aninhado somente se a condição for verdadeira (entende-se como verdadeiro o tipo *True* ou algum valor que não seja zero ou vazio):

```
>>> if 1:
...     print "ok"
...
ok
>>> if 0 or [] or {}: #todos sao como False
...     print "ok"
...
>>> if [1,2,3]:
...     print "ok"
...
ok
>>> if 1 and 2 and not False:
...     print "ok"
...
ok
>>> var = 5
>>> while var > 0:
...     var-=1
...     print var
...
4
3
2
1
0
```

A estrutura ***while*** executa o bloco enquanto a condição estiver verdadeira. A estrutura ***for*** executa um bloco atribuindo a uma variável cada um dos valores da coleção de objetos fornecida:

```
>>> range(5)
[0, 1, 2, 3, 4]
>>> for i in range(5):
...     print i
...
0
1
2
3
4
>>> lista = ["abc", "def", "qualquer", "coisa"]
>>> for elemento in lista:
...     print elemento
...
abc
def
qualquer
```

```
coisa
```

Como a estrutura ***for*** considera apenas coleções, a função ***range*** produz uma coleção de inteiros sequenciais.

3.2.4. Considerações finais de Python

Até o momento o essencial de Python já está apresentado, mas para agilidade e encorajamento no desenvolvimento de aplicações Python, são ainda necessárias algumas dicas.

A identação demarca as estruturas de dados em Python, mas ela exige alguns cuidados. Por exemplo, o compilador só reconhece a identação como igual caso haja a mesma quantidade de espaços e tabulações (originadas por exemplo pela tecla *tab* do teclado). Então é recomendado seguir sempre um mesmo padrão, como dois espaços ou quatro espaços, ou sempre usar tabulação.

Existem ainda funções úteis como a função de leitura de arquivos ***file***. A função ***file*** abre um arquivo de dados no sistema e, de acordo com o segundo parâmetro, de escrita ou leitura, retornando um objeto que manipula o arquivo:

```
>>> handler = file("arquivo.dat")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory: 'arquivo.dat'
>>> handler = file("arquivo.dat", "w")
>>> handler = file("arquivo.dat", "w")
>>> handler.write("agora da certo")
>>> handler.close()
>>> handler = file("arquivo.dat")
>>> handler.readline()
'agora da certo'
>>> handler.close()
```

O parâmetro **'w'** define modo de escrita, mas poderia ser utilizado **'a'** para escrever no final de um arquivo já existente ou criar um novo caso não exista. Quando o segundo parâmetro é omitido, o arquivo é aberto como leitura, mas poderia ser utilizado também **'r'**. Em Python, quando omitimos algum parâmetro de uma função ou de um método, é assumido o valor padrão do mesmo.

Para a criação de uma função, utiliza-se a palavra def para demarcar que é um bloco de uma função, seguido do nome da função a ser dado e, entre parênteses, uma lista dos parâmetros a serem dados, que são nomes para variáveis:

```
>>> def funcao(var1, var2):
...     print var1, var2
...
>>> funcao(1,2)
1 2
>>> funcao("oi", "tchau")
oi tchau
```

Existem diversos outros aspectos interessantes, como criação de classes para orientação a objetos e tratamento de erros. Recomenda-se o aprofundamento em materiais mais específicos, como Reis (2004) e Lutz (2003).

3.3. Hello World em PyGame

O primeiro jogo a ser construído é extremamente sem graça: consiste em uma janela preta, que não faz absolutamente nada. Mas é a estrutura básica de um jogo, um local para o mesmo aparecer. É recomendado que, quando importar o módulo **pygame** também seja invocado a função *init()*, que inicializa os módulos da Pygame:

```
>>> import pygame
>>> pygame.init()
(6, 0)
```

A função retorna uma tupla de dois valores, sendo o primeiro de módulos da Pygame inicializados com sucesso e o segundo dos que apresentaram alguma falha. Caso isso ocorra, verifique novamente a instalação da Pygame.

Para os que já possuem experiência em criar janelas de jogos, sabem a dificuldade que é realizar tal tarefa em diversas linguagens. Entretanto, em Pygame é extremamente simples, através do módulo **pygame.display** (PYGAME, 2010a):

```
>>> dir(pygame.display)
['Info', '_pygame_C_API', '__pygameinit__', '__doc__', '__file__', '__name__',
'flip', 'get_active', 'get_caption', 'get_driver', 'get_init', 'get_surface',
'get_wm_info', 'gl_get_attribute', 'gl_set_attribute', 'iconify', 'init',
'list_modes', 'mode_ok', 'quit', 'set_caption', 'set_gamma', 'set_gamma_ramp',
'set_icon', 'set_mode', 'set_palette', 'toggle_fullscreen', 'update']
```

O módulo possui diversas funções para trabalhar com a tela em que o jogo está sendo executado. Por hora, apenas a função *set_mode* será explicada e utilizada. Essa função cria uma superfície de desenho da Pygame em uma janela (caso não seja em tela cheia), de acordo com os parâmetros passados, sendo o primeiro uma tupla de dois valores com a resolução da tela (em pixels):

```
>>> screen = pygame.display.set_mode((640, 480))
>>> screen
<Surface(640x480x32 SW)>
```

A função retorna um objeto do tipo Surface, que é uma superfície de desenho da Pygame. Na documentação disponível na Internet (PYGAME, 2010a) é possível ver as opções do segundo parâmetro, que pode ser por exemplo para que o jogo seja executado em tela cheia. Quando esse código for executado, uma janela preta aparecerá na tela, que nem poderá ser fechada. Isso ocorre pois não foi associado o evento "fechar janela" para de fato fechar a janela.

O próximo passo será construirmos um *loop*, ou seja, a estrutura que repetirá infinitamente até que o jogo seja encerrado. A grande maioria dos jogos utiliza-se de *loops*, nos quais cada estrutura representa uma parte do jogo, e assim é possível desenhar os objetos necessários na tela, movimentar personagens, tratar eventos dos usuários e do jogo, dentre outras rotinas que precisam ser executadas constantemente em um jogo.

É necessário controlarmos a quantidade de quadros por segundo do *loop*, pois como os computadores atuais possuem alto poder de processamento, a jogabilidade pode ser prejudicada caso a velocidade do jogo fique muito rápida.

Para isso, a Pygame conta com o módulo **time**, o qual possui o objeto *Clock* para controlar a quantidade de quadros por segundo, dentre outras utilidades:

```
>>> clock = pygame.time.Clock()
>>> type(clock)
<type 'Clock'>
```

Com o uso do método *tick*, o objeto controla o número máximo de quadros, sendo o único parâmetro a quantidade máxima de quadros por segundo. Dessa maneira, agora surge a primeira aplicação, na qual será utilizada uma variável ***running*** para condicionar o bloco de *loop*, que por enquanto não possuirá final já que a aplicação ainda não trata o evento de fechar a janela. O código resultante, que é recomendado agora que seja colocado em um arquivo fonte separado, segue abaixo:

```
import pygame
pygame.init()
running = True
screen = pygame.display.set_mode((640,480))
clock = pygame.time.Clock()
while running:
        clock.tick(7)
```

O código acima irá produzir uma janela com o fundo preto e travada, ou seja, não fecha. O motivo é porque não estamos lidando ainda com os eventos da PyGame, que inclui o ato de fechar a janela. No caso então, o loop "while running" está *rodando* infinitamente.

3.4. Carregando informações do jogo

Para o jogo que está sendo construído, é necessário armazenar e carregar seu mapa, bem como suas imagens estáticas. O mapa do jogo é baseado em uma grande matriz de números inteiros, na qual cada número representa o tipo de imagem que aquele quadrado é. Dessa maneira, é preciso armazenar essa matriz de números em algum arquivo.

Aplicações em geral utilizam diferentes técnicas para armazenar e carregar informações. Uma delas é criar um formato próprio baseado em texto ou *bytes* e construir um interpretador para processar essas informações de alguma forma. Outra é utilizando alguma biblioteca para isso, como um *parser* XML ou o módulo **cPickle**.

O módulo **cPickle** armazena e carrega objetos Python em arquivos ou *strings*. Além dos tipos básicos de Python, como listas e dicionários, esse módulo também armazena objetos de classes novas, tornando-o útil para ser utilizado para salvar e carregar o estado atual dos jogos. Dois métodos serão úteis no projeto: ***dump*** e ***load*** (Doll, 2002). O método ***load*** carrega as informações de um arquivo dado como parâmetro, enquanto que o método ***dump*** armazena no segundo parâmetro (arquivo) o objeto passado como primero parâmetro:

```
>>> lista = [1.0, 2, "tres"]
>>> import cPickle
>>> cPickle.dump(lista, file("arquivo.dat", "w"))
```

```
>>> nova_lista = cPickle.load(file("arquivo.dat"))
>>> nova_lista
[1.0, 2, 'tres']
>>> nova_lista is lista
False
>>> nova_lista == lista
True
```

Como o módulo permite carregar e salvar objetos Python em arquivos, é possível salvar estado do jogo, perfil de jogador preferências de usuários, apesar de existirem módulos mais específicos para trabalhar com preferências de aplicações em Python. Outra questão é que os objetos são salvos em formato texto, o que pode tornar fácil a edição do estado do jogo, por exemplo.

3.5. Criando mapa baseado em quadros

Em MMORPGs um importante e usual componente é o mapa do jogo, em que diversos jogadores (até milhares) habitam um mesmo mapa, um mesmo mundo, e é necessário espaço para todos – seja para habitar, perseguir inimigos ou encontrar tesouros. Para mapas extensos, os mapas podem ser representados por uma matriz de valores, na qual cada elemento determina como é uma parte do mapa.

A Figura 3.2 mostra um pedaço de um mapa dividido em 15 quadros, chamados de *tiles*. Os quadros A e B são das mesmas imagens, bem como C e D, mas por estarem posicionadas de maneira apropriada geram um mapa contínuo.

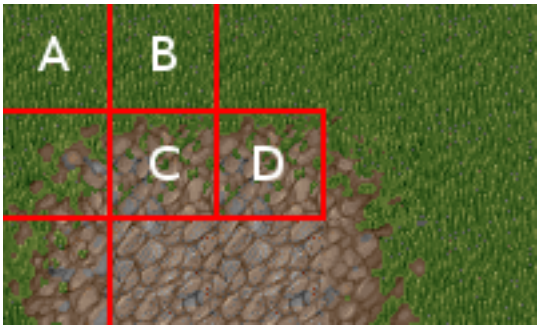


Figura 3.2. Imagens repetidas, *tiles*, formando uma imagem maior para o mapa.

Após a criação do terreno, posicionam-se objetos pelo mapa, como árvores e pedras por exemplo. A Figura 3.3 mostra um mapa que utiliza um conjunto de texturas, já com árvores e pedras. Observando em detalhes, é possível ver a repetição dos quadros no lado direito da imagem.

Apesar de mapas normalmente utilizarem também altura além de posicionamento, principalmente em jogos 3D, o projeto deste capítulo utilizará apenas altura uniforme para facilitar o desenvolvimento, permitindo o uso de números inteiros para representar o mapa. Cada inteiro é equivalente a um *tile*, e a posição do número na matriz determina a posição no mapa global do jogo.

A dimensão da matriz (tamanho) é o total de quadros que o mapa tem, que pode ser multiplicado pela dimensão de cada quadro para se ter idéia do total de

Figura 3.3. Mapa construído com *tiles*, árvores e pedras.

pixels do mapa. Para o jogo em construção, os *tiles* têm dimensão 40 x 40 pixels, e já existe um mapa construído de 500 x 500, que pode ser carregado no diretório "data":

```
>>> import cPickle
>>> tilemap = cPickle.load(file("data/tilemap.txt"))
>>> len(tilemap)
500
>>> len(tilemap[1])
500
```

O mapa presente no arquivo "tilemap.txt" possui uma matriz de inteiros variando de 0 a 23, na qual cada número é o correspondente a imagem tile**N**.png do diretório "imagens". Dessa maneira, o arquivo tile0.png é o que representa o *tile* de código 0 da matriz, e assim por diante.

Para representar a posição de árvores, casas e pedras, é utilizado um outro arquivo de mapa, o "objmap.txt". Também de dimensão 500 x 500 quadros, esse arquivo representa se há algum objeto em cada posição:

```
>>> objmap = cPickle.load(file("data/objmap.txt"))
>>> objmap[0][0]
255
>>> len(objmap)
500
```

O mapeamento dos números funciona de maneira análoga aos arquivos tileN.png, mas agora obj**N**.png, sendo, por convenção o número 255 representante de espaço vazio no mapa. Outro número reservado que não é uma imagem é o 254, que determina um espaço ocupado mas sem uma imagem necessariamente – árvores por exemplo ocupam mais que um quadro, que 40 pixels de altura.

O módulo **pygame.image** (PYGAME, 2010d) possibilita carregar imagens de formatos diversos (como PNG, JPEG e BMP) em uma *Surface*, uma superfície de desenho da Pygame. Para carregar imagens no formato Pygame utiliza-se a função ***load*** do módulo **pygame.image**:

```
>>> import pygame
>>> pygame.image.load(file('imagens/tile0.png'))
<Surface(40x40x24 SW)>
```

É necessário apenas um parâmetro, o arquivo da imagem. Como o jogo carrega diversos arquivos de imagens, com nomes sequenciais, podemos usar um *for* do Python para carregar todas as imagens:

```
>>> tile = []
>>> obj = []
>>> for i in range(24):
...     tile +=[pygame.image.load( file('imagens/tile%s.png'%i) )]
...     obj +=[pygame.image.load( file('imagens/obj%s.png'%i) )]
...
>>> tile
[<Surface(40x40x24 SW)>, <Surface(40x40x32 SW)>, <Surface(40x40x32 SW)>,
<Surface(40x40x32 SW)>, <Surface(40x40x32 SW)>, <Surface(40x40x32 SW)>,
<Surface(40x40x32 SW)>, <Surface(40x40x32 SW)>, <Surface(40x40x32 SW)>,
<Surface(40x40x32 SW)>, <Surface(40x40x32 SW)>, <Surface(40x40x32 SW)>,
<Surface(40x40x32 SW)>, <Surface(40x40x32 SW)>, <Surface(40x40x32 SW)>,
<Surface(40x40x32 SW)>, <Surface(40x41x32 SW)>, <Surface(40x40x32 SW)>,
<Surface(40x40x32 SW)>, <Surface(40x40x32 SW)>, <Surface(40x40x32 SW)>,
<Surface(40x40x32 SW)>, <Surface(40x40x32 SW)>, <Surface(40x40x32 SW)>]
```

No código acima, duas listas são inicializadas vazias (*tile = []* e *obj = []*) e o *for* é executado carregando os arquivos "tileN.png" e "objN.png", variando o N de 0 a 23, em superfícies da PyGame.

A superfície em PyGame pode ser combinada com outras superfícies, com o método ***blit***, recebendo como primeiro parâmetro o objeto a ser combinado sobre ele, e como segunda parâmetro a coordenada, em pixels, de onde a superfície deve ser combinada.

```
>>> pygame.init()
(6, 0)
>>> screen = pygame.display.set_mode((640,480))
>>> screen.blit(tile[0], (0,0))
<rect(0, 0, 40, 40)>
```

É possível que não seja exibido nada em tela com a execução acima ainda, pois a execução da PyGame em terminal interativo funciona de modo diferente da execução em arquivo fonte.

A coordenada para combinar imagens é fornecida em pixels, enquanto que no mapa para o projeto proposto possui coordenadas em *tiles*. Por este motivo,

é interessante criar-se três constantes (em Python, declara-se variáveis em caixa alta para apenas simbolizar constantes):

```
>>> WIDTH, HEIGHT, TILE = 16, 11, 40
```

As "constantes" (**WIDTH, HEIGHT, TILE**) são referentes à largura e à altura em quadros, e ao tamanho em pixels de cada *tile*. O tamanho em pixels é o tamanho de cada *tile*, e como tratam-se de 640 pixels na horizontal, cabem então 16 quadros de 40 pixels (16 x 40 = 640 pixels). Na vertical cabem 12 quadros, mas por hora serão utilizados 11 quadros apenas, sendo o último uma área da tela reservada para impressão de textos de interação entre o jogador e o jogo.

Pode-se resumir que a tela é dividida em uma *matriz de renderização* de **WIDTH** por **HEIGHT** posições, no caso 16 x 11 posições, na qual cada posição é um quadro de imagem. O loop do jogo deve então percorrer cada uma dessas posições para desenhar o quadrado correspondente, primeiro sendo as texturas do mapa (*tilemap*) depois os objetos (*objmap*), pois os objetos devem ser renderizados depois para ficarem por cima do terreno:

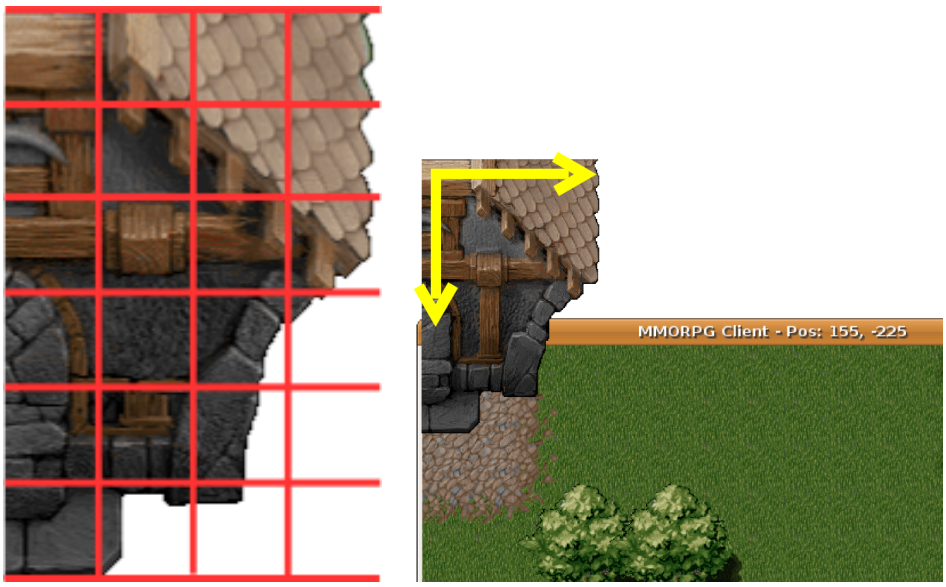
```
>>> while 1:
...     for y in range(HEIGHT):
...         for x in range(WIDTH):
...             screen.blit(tile[1], (x*TILE, y*TILE))
...     pygame.display.flip()
...
```

No caso, em um *loop* de jogo, o que o trecho acima faz é imprimir o tile[1] (uma textura de gramado) em todas as 16 x 11 posições da janela, e a função ***flip*** do módulo display atualiza toda a tela, ou seja, renderiza todos os objetos na tela a cada chamada. Para finalizar esse loop infinito a combinação de teclas "CTRL + C" finaliza o loop. Nota para a posição "(x***TILE**, y***TILE**)", que retorna a posição (x,y) da matriz 16 x 11 posições, e multiplica o tamanho do quadro para obter-se o valor da posição em pixels na tela.

Quando objetos que ocupam mais do que um quadro (40 x 40 pixels) forem renderizados, existe o problema de que os objetos antes do ponto de origem devem exibir as partes que sobrepõem, quando perto do extremo. Considerando o objeto da Figura 3.4.a como exemplo, e o ponto (0,0) da PyGame como sendo o canto superior esquerdo da janela, existe o momento em que tal objeto está fora da matriz de renderização, como exibido na Figura 3.4.b. Para solucionar tal problema, é possível realizar a renderização dos objetos antes to ponto (0,0), garantindo que os pedaços dos objetos não presentes na matriz de renderização apareçam.

Uma vez que no projeto do jogo só existem objetos até o número 23, acima só existem o espaço vazio ou o espaço preenchido, que não serão renderizados. Considerando o *loop* mostrado na seção anterior, o novo *loop* do jogo com os objetos antes do (0,0) e com valor numérico menor que 24 estabelece-se assim:

```
while running:
  clock.tick(7)
  for y in range(HEIGHT):
```



(a) Tamanho de 6 x 4 quadros de 40 pixels.

(b) Parte fora da matriz de renderização.

Figura 3.4. Objeto presente no jogo que representa parte de uma casa

```
    for x in range(WIDTH):
      screen.blit(tile[tilemap[y][x]], (x*TILE, y*TILE))
  # novo #for# para os objetos com as devidas restricoes.
  for y in range(-6, HEIGHT):
    for x in range(-4, WIDTH):
      if objmap[y][x] < 24:
        screen.blit(obj[objmap[y][x]], (x*TILE, y*TILE))
  pygame.display.flip()
```

É interessante notar que a ordem do laço dos objetos deve vir depois que o do mapa, pois os objetos devem ser mixados por cima do maap. Com a união do que foi realizado até o momento, é possível já carregar o mapa e os objetos na janela do jogo, sendo exibida uma imagem estática de árvores e grama. O código é exibido a seguir:

```
import pygame, cPickle
pygame.init()

tilemap = cPickle.load(file('data/tilemap.txt', 'r'))
objmap = cPickle.load(file('data/objmap.txt', 'r'))
running = True
screen = pygame.display.set_mode((640,480))
clock = pygame.time.Clock()

WIDTH, HEIGHT, TILE = 16, 11, 40

tile, obj = [], []

for i in range(24):
  tile += [pygame.image.load(file('imagens/tile%s.png'%i))]
  obj += [pygame.image.load(file('imagens/obj%s.png'%i))]

while running:
  clock.tick(7)
  for y in range(HEIGHT):
```

```
    for x in range(WIDTH):
      screen.blit(tile[tilemap[y][x]], (x*TILE, y*TILE))
  for y in range(-6, HEIGHT):
    for x in range(-4, WIDTH):
      if objmap[y][x] < 24:
        screen.blit(obj[objmap[y][x]], (x*TILE, y*TILE))
  pygame.display.flip()
```

3.6. Tratamento de eventos do usuário e da aplicação

Um dos pontos necessários para a usabilidade em jogos é a captura das ações do jogador, como joystick, mouse e teclado. O módulo **pygame.event** (PYGAME, 2010b) trata desses tipos de eventos, funcionando como uma fila de eventos em que cada evento capturado é um objeto adicionado a uma lista Python. A função ***get*** retorna essa lista, deixando a fila da Pygame vazia novamente para novas ações:

```
>>> import pygame
>>> pygame.init()
(6, 0)
>>> pygame.event.get()
[]
```

Para se capturar no terminal interativo, basta inicializar a tela e verificar os eventos capturados:

```
>>> screen = pygame.display.set_mode((640,480))
>>> pygame.event.get()
[<Event(3-KeyUp {'scancode': 0, 'key': 300, 'mod': 4096})>,
<Event(1-ActiveEvent {'state': 2, 'gain': 1})>,
<Event(2-KeyDown {'scancode': 38, 'key': 97, 'unicode': u'a', 'mod': 4096})>,
<Event(3-KeyUp {'scancode': 38, 'key': 97, 'mod': 4096})>,
<Event(2-KeyDown {'scancode': 56, 'key': 98, 'unicode': u'b', 'mod': 4096})>,
<Event(3-KeyUp {'scancode': 56, 'key': 98, 'mod': 4096})>,
<Event(2-KeyDown {'scancode': 54, 'key': 99, 'unicode': u'c', 'mod': 4096})>,
<Event(3-KeyUp {'scancode': 54, 'key': 99, 'mod': 4096})>,
<Event(1-ActiveEvent {'state': 2, 'gain': 0})>,
<Event(3-KeyUp {'scancode': 0, 'key': 300, 'mod': 4096})>]
```

A lista retornada contém objetos do tipo ***Event***, a classe da Pygame que simboliza um evento. É possível encontrar uma lista oficial dos tipos de eventos na documentação oficial (PYGAME, 2010b). Cada objeto tem como atributos *type*, o tipo do evento em valor numérico e um dicionário de atributos de acordo com o evento:

```
>>> eventos = pygame.event.get()
>>> evento = eventos[0]
>>> evento
<Event(1-ActiveEvent {'state': 4, 'gain': 1})>
>>> evento.type
1
>>> evento.state
4
>>> evento.gain
1
```

As chaves do dicionário podem ser acessadas como atributos do objeto. Outra questão é utilizar nomes de constantes ao invés de valores numéricos. To-

das as constantes da Pygame estão no módulo **pygame.locals**, podendo serem importadas do seguinte modo:

```
>>> from pygame.locals import *
```

Ainda sobre os eventos, é possível criar objetos de tipos personalizados e colocá-los na fila com os outros eventos com a função ***post***:

```
>>> e = pygame.event.Event(25, {'ano': 2009})
>>> pygame.event.post(e)
>>> pygame.event.get(25)
[<Event(25-UserEvent {'ano': 2009})>]
>>> e.ano
2009
```

Para o jogo então, pode-se definir alguns novos eventos, que ajudarão no desenvolvimento futuro. Ao invés de considerar os novos eventos números, é possível também definir então constantes para estes:

```
GOTO = 24
MESSAGE = 25
SETOBJECT = 26
SYSTEM = 27
GETINFO = 28
```

O **GOTO** é um evento de deslocamento do personagem, **MESSAGE** é usado para mensagens ao jogador, **SETOBJECT** cria ou destrói um objeto em uma coordenada, **SYSTEM** é usado para mensagens do sistema (como erros) e usando **GETINFO** obtém-se informações do servidor de um objeto no mapa.

3.6.1. Movimentação da tela do jogo

Para movimentação da tela do jogo, é necessário que a matriz de renderização se desloque pelos mapas, tanto de tiles quanto de objetos. A matriz representa apenas um pedaço do mapa, portanto a posição ao redor do jogador pode ser representada como um descolamento da matriz pelos pontos do mapa.

Como o jogador deve se situar no centro da tela, é interessante definir uma coordenada referente ao centro para facilitar o desenvolvimento, subtraindo 1 de cada valor pois o personagem ocupa mais do que um quadro, além das variáveis **slide_x** e **slide_y** para denominar o deslocamento da tela (horizontal e vertical respectivamente) com valor inicial de 0 (começando no canto superior esquerdo do mapa):

```
MIDDLE = WIDTH/2 - 1 , HEIGHT/2 - 1
slide_x = 0
slide_y = 0
```

O motivo da subtração na constante **MIDDLE** é puramente estético, quando o jogo estiver em fase final essa constante pode ser alterada para ver os diferentes posicionamentos do personagem do jogador.

Para movimentar o mapa, será criada a função ***goto***, a qual desloca o mapa para as coordenadas (x,y) passadas como parâmetro, referentes à posição do jogador. Por se tratarem de variáveis globais, que pertencem a todo o código, é

necessário declarar que as variáveis já inicializadas **slide_x** e **slide_y** são globais. Então deve-se posicioná-las na coordenada de parâmetro (x,y) e subtrair o ponto central, pois o deslocamento é do personagem e a visualização deve ser realizada no centro da tela:

```
def goto(x, y):
  global slide_x, slide_y
  slide_x = x - MIDDLE[0]
  slide_y = y - MIDDLE[1]
  pygame.display.set_caption("MMORPG Client - Pos: %2d, %2d"%(x, y))
```

A função ***set_caption*** muda o título da tela, nesse caso imprimindo a posição X,Y do personagem. A Figura 6 mostra a matriz do mapa e a tela do jogo ocupando apenas uma parte, o que seria a matriz de renderização ou deslocamento. As variáveis **slide_x** e **slide_y** deslocam a matriz do ponto (0,0) do mapa (representado por B) até o canto superior esquerdo da tela.

A constante **MIDDLE**, composta por dois valores, é referente ao deslocamento do personagem, ponto B, em relação ao canto da tela. Isso é necessário pois tem-se o objetivo de renderizar a região ao redor do personagem, colocando-o no centro da tela. Porém o jogo deve tratar a posição do jogador e não do canto da tela para tratamento de colisões com outros objetos (o que colide é o jogador, e não o canto da tela).

A posição do personagem no mapa será sempre dada pela coordenada (*slide_x + MIDDLE[0] , slide_y + MIDDLE[1]*). O valor do deslocamento da tela pode também ser negativo, nesse caso deslocando a matriz de renderização da direita para a esquerda e de cima para baixo, de acordo com o eixo negativo da Pygame.

Com a função ***goto***, é possível movimentar o jogador no mapa. Existem dois problemas comuns que novos programadores de jogos normalmente enfrentam:

1. Movimentos coerentes, que consiste em fazer com que o personagem se movimente a uma certa velocidade e em todas as direções.

2. Aceleração e velocidade do personagem, pois o jogador espera que enquanto estiver pressionando a tecla direcional para a movimentação o personagem irá se movimentando a uma velocidade constante ou acelerando (mais comum em jogos de corrida).

Pode-se solucionar os problemas utilizando uma variável que determine a velocidade do personagem nos eixos X e Y, por se tratar de um jogo em 2D. A velocidade é então uma tupla de dois valores, variando entre -1, 0 e 1 cada (produzir valores diferentes poderiam possibilitar o personagem correr e andar, por exemplo). A Figura 3.5 mostra a direção do personagem para cada valor de tupla.

Desse modo, pode-se definir uma constante que mapeie as setas do teclado diretamente para uma aceleração de algum eixo, permitindo que o botão direito,

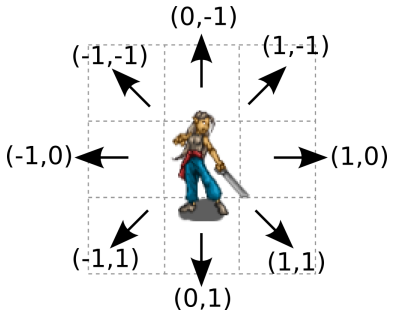


Figura 3.5. Direções de movimentação do personagem com as velocidades de deslocamento, considerando o eixo positivo para baixo e para a direita, no mesmo formato que a Pygame.

por exemplo, acelere em (1, 0), ou seja, 1 a direita no eixo X, 0 no eixo Y. Ao apertar um botão, a velocidade do personagem será somada com a aceleração da tecla correspondente, quando se soltar, a aceleração correspondente subtrai-se. A constante MOVES pode ser definida como:

```
MOVES = {
K_RIGHT: ( 1, 0),
K_LEFT : (-1, 0),
K_UP   : ( 0,-1),
K_DOWN : ( 0, 1)
}
```

Para realizar a logística do movimento, pode-se definir uma nova função, ***move***, para criar um evento do tipo **GOTO**, ser enviado à Pygame e ser processado para então chamar a função ***goto***. O motivo desse ciclo é para permitir que o servidor valide o movimento, ou seja, o jogador tenta realizar um movimento, envia ao servidor que retorna com um evento Pygame permitindo ou não o movimento. A função ***move*** está descrita a seguir:

```
from pygame.event import Event
def move((inc_x, inc_y)):
  if inc_x != 0 or inc_y != 0:
    e = Event(GOTO, {'x': slide_x + inc_x + MIDDLE[0], 'y': slide_y + inc_y + MIDDLE[1]})
    pygame.event.post(e)
```

Se a velocidade não é (0,0), cria um novo evento ***GOTO***, que possui dois atributos: x e y. Cada atributo representa a coordenada de destino, e portanto é necessário somar a velocidade ao deslocamento da tela (*slide_x* ou *slide_y*) e o deslocamento do personagem em relação ao canto superior esquerdo da tela, em cada um dos eixos horizontal e vertical. Criado o evento, a função post adiciona o evento na fila da Pygame.

3.6.2. Cliente manipulando eventos

Com a lógica da movimentação construída, resta manipular os eventos, que deve ser feito no *loop* do jogo. Para deixar o código mais claro, pode-se criar uma função ***handle***, que retira os eventos da fila e trata de acordo com o tipo:

```
moving = (0,0) #cria-se uma variavel para manter a velocidade atual
```

```
def handle():
  global running, moving
  for e in pygame.event.get():
    if e.type == QUIT:
      running = False
    elif e.type == KEYDOWN:
      if e.key in MOVES.keys():
        moving = (moving[0] + MOVES[e.key][0], moving[1] + MOVES[e.key][1])
    elif e.type == KEYUP:
      if e.key in MOVES.keys():
        moving = (moving[0] - MOVES[e.key][0], moving[1] - MOVES[e.key][1])
    elif e.type == GOTO:
      goto(e.x, e.y)
```

O tratamento apresentado é referente a fechar o jogo, quando o jogador pressiona e solta uma tecla (**KEYDOWN** e **KEYUP** respectivamente) e quando o sistema reconhece um evento do tipo **GOTO**, que denomina onde o personagem deve se posicionar.

O *loop*, com a movimentação do jogador, também fica diferente, pois agora a tela será renderizada em relação ao deslocamento da tela, além de ser necessário o tratamento de eventos:

```
while running:
  clock.tick(7)
  handle()
  move(moving)
  for y in range(HEIGHT):
    for x in range(WIDTH):
      i,j = (slide_x+x,slide_y+y)
      screen.blit(tile[tilemap[j][i]], (x*TILE, y*TILE))
  for y in range(-6,HEIGHT):
    for x in range(-4,WIDTH):
      i, j = (x+slide_x,slide_y+y)
      if objmap[j][i] < 24:
        screen.blit(obj[objmap[j][i]], (x*TILE, y*TILE))
  pygame.display.flip()
```

As variáveis **i** e **j** são na realidade a posição na tela que varia (**x** e **y** do *loop*) somadas ao deslocamento da tela, a fim de colocar o objeto ou o *tile* da coordenada do mapa (i,j) na coordenada correta da tela (x,y). As chamadas na função ***handle*** e depois na ***move*** garantem a manipulação de eventos e movimentação do personagem.

O código final do cliente que permite já a movimentação da tela está descrito a seguir:

```
01| import pygame, cPickle
02| from pygame.locals import *
03| from pygame.event import Event
04| tilemap = cPickle.load(file('data/tilemap.txt', 'r'))
05| objmap = cPickle.load(file('data/objmap.txt', 'r'))
06| running = True
07| screen = pygame.display.set_mode((640,480))
08| clock = pygame.time.Clock()
09|
10| GOTO = 24
11| MESSAGE = 25
12| SETOBJECT = 26
13| SYSTEM = 27
14| GETINFO = 28
15|
```

```
16| WIDTH, HEIGHT, TILE = 16, 12, 40
17| MIDDLE = WIDTH/2 - 1 , HEIGHT/2 - 1
18| tile, obj = [], []
19|
20| for i in range(24):
21|   tile += [pygame.image.load(file('imagens/tile%s.png'%i))]
22|   obj += [pygame.image.load(file('imagens/obj%s.png'%i))]
23|
24| slide_x = 0
25| slide_y = 0
26|
27| def goto(x, y):
28|   global slide_x, slide_y
29|   slide_x = x - MIDDLE[0]
30|   slide_y = y - MIDDLE[1]
31|   pygame.display.set_caption("MMORPG Client - Pos: %2d, %2d"%(x, y))
32|
33| def move((inc_x, inc_y)):
34|   if inc_x != 0 or inc_y != 0:
35|     e = Event(GOTO, {'x': slide_x + inc_x + MIDDLE[0],
                        'y': slide_y + inc_y + MIDDLE[1]})
36|     pygame.event.post(e)
37|
38| MOVES = {
39| K_RIGHT: ( 1, 0),
40| K_LEFT : (-1, 0),
41| K_UP   : ( 0,-1),
42| K_DOWN : ( 0, 1)
43| }
44|
45| moving = (0,0)
46| def handle():
47|   global running, moving
48|   for e in pygame.event.get():
49|     if e.type == QUIT:
50|       running = False
51|     elif e.type == KEYDOWN:
52|       if e.key in MOVES.keys():
53|         moving = (moving[0] + MOVES[e.key][0], moving[1] + MOVES[e.key][1])
54|     elif e.type == KEYUP:
55|       if e.key in MOVES.keys():
56|         moving = (moving[0] - MOVES[e.key][0], moving[1] - MOVES[e.key][1])
57|     elif e.type == GOTO:
58|       goto(e.x, e.y)
59|
60| while running:
61|   clock.tick(7)
62|   handle()
63|   move(moving)
64|   for y in range(HEIGHT):
65|     for x in range(WIDTH):
66|       i,j = (slide_x+x,slide_y+y)
67|       screen.blit(tile[tilemap[j][i]], (x*TILE, y*TILE))
68|   for y in range(-6,HEIGHT):
69|     for x in range(-4,WIDTH):
70|       i, j = (x+slide_x,slide_y+y)
71|       if objmap[j][i] < 24:
72|         screen.blit(obj[objmap[j][i]], (x*TILE, y*TILE))
73|   pygame.display.flip()
```

3.7. Conexão cliente-servidor e threads

A comunicação entre dispositivos computacionais tem se intensificado, através de redes de computadores que conectam os dispositivos entre si com o objetivo de compartilhar serviços. Para a comunicação, utilizam-se protocolos de rede, que definem o formato e a ordem das mensagens trocadas entre dois ou mais

dispositivos computacionais bem como ações realizadas na transmissão e recebimento de uma mensagem (Kurose and Ross, 2005).

Um protocolo de rede muito utilizado, que também será utilizado pelo jogo, é o TCP (Kurose and Ross, 2005, p.178), que utiliza uma conexão orientada, ou seja, as aplicações que o utilizam trocam mensagens de controle para auxiliar no controle do fluxo de pacotes.

Dois modelos bastante comuns para utilização de rede são de cliente-servidor e Peer-to-Peer (P2P). Cliente-servidor baseia-se em sistemas conectando em um mesmo dispositivo, o servidor, que centraliza as informações (ainda que existam vários servidores, para o cliente é visto apenas um servidor). Redes P2P são sistemas de nós capazes de se interconectarem e se organizarem em topologias de redes (Androutsellis-Theotokis and Spinellis, 2004).

O modelo cliente-servidor será utilizado no jogo pois o servidor agirá como entidade centralizadora e controladora dos jogadores. A segurança em jogos online é um assunto que ainda recorre aos atuais jogos, pois falhas encontradas podem dar vantagens a um jogador, tornando o jogo desmotivante em um MMORPG, quando por exemplo um jogador consegue mais pontos apenas com uma falha.

Existem sempre pontos positivos e negativos em cada aspecto de segurança. Utilizar um servidor para validar movimentação e ações do jogador auxilia na coerência do jogo, já que em geral os servidores não podem ser alterados pelos jogadores, mas os clientes sim. O quanto se processa no servidor dependerá da capacidade que se espera utilizar do servidor em recursos computacionais e o quão comprometida a jogabilidade pode estar. Por exemplo, o jogador pode apenas falar para onde ele está indo e o servidor responder que ele pode, ou o jogador passar o valor do clique do mouse e o servidor responde onde ele vai.

Outro aspecto muito comum em mundos virtuais é utilização de *spoofing*, em que um jogador consegue se passar por outro, roubando itens virtuais, causando problemas sociais no mundo virtual (como por exemplo tomar decisões quando o jogador tem um cargo importante), dentre outros problemas. A utilização de um sistema de autenticação, nomes únicos para jogadores e até mesmo utilização de chaves assimétricas pode auxiliar no combate a esse tipo de prática.

3.7.1. Threads em jogos

Um assunto muito abordado principalmente na área de Sistemas Operacionais é a concorrência de processos. Os sistemas operacionais utilizados atualmente podem fazer diversas atividades ao mesmo tempo, e a utilização de processos possibilita isso. Cada processo utiliza recursos próprios, com espaço de endereçamento e de memória próprios (Tanenbaum and Woodhull, 2006).

O sistema operacional alterna entre os processos para que todos possam ser executados em um curto período de tempo, tornando-os praticamente paralelos. Alguns sistemas também utilizam mais que um processador para executar processos em paralelo.

Dentro de um processo existe uma única *thread* por padrão, mas pode haver mais de uma a fim de compartilhar o mesmo espaço de endereçamento, ou outros recursos em comum. A grande diferença de processos e *threads* é que um processo é usado para agrupar recursos e threads são programadas para execução na CPU (Tanenbaum and Woodhull, 2006), normalmente utilizando os mesmos recursos como a memória principal.

A utilização de *threads* em jogos é fundamental, pois normalmente é necessário o processamento paralelo de informações, como renderização gráfica e manipulação dos eventos do usuário que é o caso do cliente do projeto. O servidor do projeto também utiliza diversas *threads*, uma para cada cliente a fim de manter a comunicação síncrona

Em Python, as funções podem ser executadas diretamente em *threads* a parte. Para isso, pode se usar o módulo **thread** com a função ***start_new_thread***:

```
>>> import thread
>>> def teste():
...     print "hello mundo"
...
>>> thread.start_new_thread(teste, tuple())
-1211667568
hello mundo
```

Foi necessário utilizar a função ***tuple***, que retorna uma tupla vazia, para passar que não há parâmetros na função. O retornado antes da impressão do texto, o número que varia a cada chamada, é o código da *thread*, que pode ser utilizado para manipulá-la.

3.7.2. Conectando o cliente ao servidor

Uma maneira interessante e fácil de criar um modelo de conexão cliente-servidor em Python é utilizando socket, que é uma porta entre o processo da aplicação e o protocolo de comunicação (Kurose and Ross, 2005, p. 114).

Em Python, importa-se o mesmo módulo para o servidor e para o cliente, o módulo **socket**. Após criar-se uma conexão TCP, o servidor é inicializado com o método ***bind***, para "escutar" na porta determinada no parâmetro passado (uma tupla com o primeiro valor correspondente ao IP e o segundo à porta). Criando-se então uma *thread* para aceitar clientes (método ***accept***) e receber mensagens do objeto de conexão criado no aceite (método ***recv*, com quantos *bytes* de buffer como parâmetro**). Já o cliente apenas cria a conexão TCP, chama a conexão a um servidor com o método ***connect***, também com IP e porta em uma tupla como parâmetro, e utiliza livremente os métodos ***recv*** e ***send***. O código a seguir ilustra um exemplo de criação de um servidor na mesma máquina em que se chama o cliente:

```
>>> import socket, thread
>>> server = socket.socket()
>>> server.bind(("",5050))
>>> server.listen(1)
>>> def server_loop():
...     connection, info = server.accept()
...     msg = connection.recv(9999)
```

```
...     print "server: " + msg
...
>>> thread.start_new_thread(server_loop, tuple())
-1224254608
>>> client = socket.socket()
>>> client.connect(("localhost",5050))
>>> client.send("olá")
4
server: olá
```

O método ***recv*** e ***send*** estão presentes tanto em servidor como cliente, sendo o ***recv*** trava a função ou o trecho de código até receber algum ***send*** do outro lado da conexão. No exemplo mostrado é possível notar ainda que o servidor não passa IP como parâmetro no ***bind*** (aceitando qualquer IP para conexão) e o método ***send*** retorna o total de bytes enviados com sucesso.

Com o conhecimento dos módulos de conexão (**socket**) e de *threads* (**thread**), é possível então conectar o cliente a um servidor já pronto para facilitar a criação do jogo. Recomenda-se para os interessados explorar o código do servidor e buscar melhorias e alterações, uma vez que o código é extenso mas não complicado.

A seguir será colocado o código final do cliente com as linhas numeradas, sendo explicado os novos pontos após o código:

```
 1| import pygame, cPickle, thread, socket
 2| from pygame.locals import *
 3| pygame.init()
 4| from pygame.event import Event
 5|
 6| GOTO = 24 # levar o seu personagem para posicao do evento
 7| MESSAGE = 25 # mensagem para o cliente
 8| SETOBJECT = 26 # colocar um objeto no mapa
 9| SYSTEM = 27 # mensagem do sistema, provavelmente um erro
10| GETINFO = 28 # requisitar informacoes em um ponto do mapa
11|
12| DESTINY = ("127.0.0.1", 5000)
13| WIDTH, HEIGHT, TILE = 16, 11, 40
14| MIDDLE = WIDTH/2 - 1 , HEIGHT/2 - 1
15| MYID = Event(SYSTEM, {'id': 15, 'name': 'anonimo'})
16|
17| tilemap = cPickle.load(file('data/tilemap.txt', 'r'))
18| objmap = cPickle.load(file('data/objmap.txt', 'r'))
19| running = True
20| screen = pygame.display.set_mode((640,480))
21| clock = pygame.time.Clock()
22| tile, obj = [], []
23|
24| for i in range(24):
25|   tile += [pygame.image.load(file('imagens/tile%s.png'%i))]
26|   obj += [pygame.image.load(file('imagens/obj%s.png'%i))]
27|
28| font = pygame.font.SysFont("default", 16)
29| def settext(msg):
30|   global text
31|   text = pygame.Surface((640, 40))
32|   text.blit(font.render(msg, True, (255,255,255)), (5,12))
33|
34|
35| settext("")
36| slide_x = 0
37| slide_y = 0
38|
39| def goto(x, y):
40|   global slide_x, slide_y
```

```
  slide_x = x - MIDDLE[0]
  slide_y = y - MIDDLE[1]
  pygame.display.set_caption("MMORPG Client - Pos: %2d, %2d"%(x, y))

def move((inc_x, inc_y)):
  if inc_x != 0 or inc_y != 0:
    e = Event(GOTO, {'x': slide_x + inc_x + MIDDLE[0],
                     'y': slide_y + inc_y + MIDDLE[1]})
    send(str(e))

def listener():
  while running:
    msg = conn.recv(999999)
    if msg != "":
      try:
        for m in eval(msg):
          pygame.event.post(m)
      except:
        print "Servidor: ",msg

def send(event):
  conn.send(str(event))

conn = socket.socket()
conn.connect(DESTINY)
thread.start_new_thread(listener, tuple())
conn.send(str(MYID))

MOVES = {
  K_RIGHT: ( 1, 0),
  K_LEFT : (-1, 0),
  K_UP   : ( 0,-1),
  K_DOWN : ( 0, 1)
}

moving = (0,0)

def handle():
  global running, moving
  for e in pygame.event.get():
    if e.type == QUIT:
      running = False
    elif e.type == KEYDOWN:
      if e.key in MOVES.keys():
        moving = (moving[0] + MOVES[e.key][0], moving[1] + MOVES[e.key][1])
    elif e.type == KEYUP:
      if e.key in MOVES.keys():
        moving = (moving[0] - MOVES[e.key][0], moving[1] - MOVES[e.key][1])
    elif e.type == MOUSEBUTTONDOWN:
      pos = slide_x + e.pos[0]/TILE, slide_y + e.pos[1]/TILE
      send(Event(GETINFO, {'pos':pos}))
    elif e.type == GOTO:
      goto(e.x, e.y)
    elif e.type == MESSAGE:
      settext(e.msg)
    elif e.type == SYSTEM:
      print e.msg
    elif e.type == SETOBJECT:
      objmap[e.y][e.x] = e.obj

while running:
  clock.tick(7)
  handle()
  move(moving)
  for y in range(HEIGHT):
    for x in range(WIDTH):
      i,j = (slide_x+x,slide_y+y)
      screen.blit(tile[tilemap[j][i]], (x*TILE, y*TILE))
  for y in range(-6,HEIGHT):
    for x in range(-4,WIDTH):
```

```
110|       i, j = (x+slide_x,slide_y+y)
111|       if objmap[j][i] < 24:
112|         screen.blit(obj[objmap[j][i]], (x*TILE, y*TILE))
113|   screen.blit(text, (0,440))
114|   pygame.display.flip()
115|
116| conn.send("bye bye")
117| conn.close()
```

Para a conexão do cliente, define-se duas constantes: uma tupla com o IP e porta do servidor e outra constante com a identificação do usuário, requisitada pelo servidor e como evento da PyGame, linhas 12 e 15. O parâmetro ***id*** da linha 15 determina qual o personagem a ser representado, podendo variar de 14 a 19 conforme definição do servidor.

A movimentação do personagem agora deverá ser validada pelo servidor. Isso ocorre para facilitar a criação do código e, principalmente, para que a validação dos movimentos dos jogadores seja controlada pelo servidor, que funciona como uma entidade supervisora para que não hajam movimentos fraudulentos, como por exemplo um jogador andar mais do que o possível ou atravessar paredes.

O cliente apenas enviará o evento do tipo GOTO para o servidor, ao invés de processar localmente. O que será processado pela função ***goto*** local serão apenas as mensagens recebidas pelo servidor, que darão as coordenadas para onde o personagem de fato deve ir, como é visto nas linhas 45 a 48. Existe uma nova função também, ***send***, que transforma qualquer objeto em *string* e envia ao servidor, nas linhas 60 e 61.

É necessário também criar-se uma função para esperar mensagens do servidor. Todas as mensagens são eventos do tipo Pygame, se não forem são mensagens para o desenvolvedor. Portanto, as linhas 50 a 58 apresentam o laço de recebimento de mensagens, em que tenta-se colocar na fila da Pygame como objetos *Event* o que é recebido (54 a 56), ou imprime caso ocorra algum erro (linha 58). Ainda sobre a conexão, as linhas 63 a 66 conectam-se no servidor, abre uma *thread* para o laço de recebimento de mensagens e envia a identificação do jogador.

Outro ponto novo no código é o uso do módulo **pygame.font** (PYGAME, 2010c), apresentado nas linhas 28 a 35, em que cria-se uma fonte padrão do sistema (linha 28) e define-se a função para renderizar texto em uma superfície de Pygame (linhas 29 a 32). O texto é renderizado na linha 32 em uma superfície preta criada na linha 31, sendo o texto branco ((255,255,255) em R,G,B é branco) e na posição (5,12). A função ***settext*** é chamada no tratamento de eventos, e adiciona-se a inclusão da superfície de texto no *loop* do jogo (linha 113).

A última grande alteração é no laço de eventos do jogo, nas linhas 77 a 98. Os novos tratamentos, a partir da linha 88 são:

- Captura do ponteiro do mouse quando ocorre um clique (linha 88), enviando um evento do tipo ***GETINFO*** para o servidor (linha 90) com a posição

correspondente ao mapa (linha 89);

- Recebimento de mensagem (linha 93), imprimindo no jogo por se tratar de uma mensagem para o cliente;
- Recebimento de mensagem de sistema (linha 95), imprimindo apenas no terminal por ser mensagem ao desenvolvedor; e
- Mudança de um objeto no mapa (linhas 97 e 99), mudando o valor no mapa de objetos da mensagem recebida do servidor. É desse modo que é realizado o movimento dos outros personagens.

O resultado final pode ser visto na Figura 3.6.

Figura 3.6. Captura de tela do cliente final.

3.8. Considerações Finais

O objetivo deste capítulo foi construir um cliente MMORPG a fim de explorar alguns conceitos de desenvolvimento de jogos com a biblioteca Pygame e a plataforma Python. O projeto do jogo MMORPG pode ainda ser melhorado em diversos aspectos, bem como derivado para outros gêneros de jogos que podem necessitar de algumas técnicas distintas.

Alguns módulos da Pygame possuem recursos muito interessantes, que podem ser explorados em profundidade com a documentação da Pygame oficial. Em especial, o módulo **pygame.mixer** manipula áudio para o jogo, **pygame.sprite** para manipular animações (para fazer por exemplo objetos animados), **pygame.movie** que permite a execução de vídeos, **pygame.transform** para trabalhar com edições em geral de **Surfaces**, além de módulos de entrada e saída como o **pygame.joystick** que permite uso de joysticks.

Uma constante dificuldade que novos programadores de Pygame encontram é a criação de interfaces gráficas, já que a Pygame por padrão não possui widgets, ou seja, elementos de interface gráfica prontos para uso, como caixa de diálogo para seleção de um arquivo, botões, menus dentre outras estruturas. Existem alguns toolkits direcionados a Pygame, sendo um de grande destaque a **PGU** (Rogers, 2009), que utiliza Pygame nativo e suporta temas para a interface gráfica, contando com um grande número de widgets.

Outra questão é a incorporação de diferentes bibliotecas para Python para o jogo. Por exemplo, aproveitando o cenário de um MMORPG, em que diversos jogadores interagem entre si, é oportuno citar um caso de uso com a biblioteca para uma rede social. É interessante imaginar cenários em que é possível se integrar jogos com redes sociais (como fazem os *social games*), e mais ainda jogos do tipo "MMO".

Jogos que reúnem grupos de participantes, também conhecidos como *social games*, vêm tendo grande destaque dentro das redes sociais presentes na internet, ampliando a função original de interação entre usuários por meio de recados e mensagens. O sucesso desse tipo de jogo pode ser associado à popularidade do site em que o aplicativo se encontra e/ou a quantidade de participantes integrados que ele possui. Alguns social games populares são FarmVille [3], Café World [4] e BuddyPoke [5].

Considerando a biblioteca disponível para o Twitter [6] [7] é possível com apenas 3 linhas de código conectar e ficar pronto para uso:

```
>>> import twitter
>>> twitconn = twitter.Api(username='twitter_login', password='twitter_pass')
```

A biblioteca acima pode, por exemplo, pegar os amigos do usuário ou mesmo postar uma nova mensagem no twitter:

```
>>> for friend in twitconn.GetFriends()
>>> for friend in twitconn.GetFriends():
...     print friend.GetScreenName(),
...
anonimo mmorpgfeed santosfc
>>> twitconn.PostUpdate("Estou jogando!")
<twitter.Status object at 0x931dbcc>
```

[3] http://www.farmville.com/
[4] http://www.zynga.com/games/index.php?game=cafeworld
[5] http://www.buddypoke.com/
[6] http://www.twitter.com
[7] disponível em <http://code.google.com/p/python-twitter/> (Acesso em 15 Fev. 2010)

Diversas outras opções estão disponíveis tanto na biblioteca do Twitter (digite "help(twitter)" para documentação) como em outras, tanto para redes sociais na Internet como para os fins mais diversos possíveis. Cabe ao leitor então começar um projeto próprio (ou derivar do MMORPG apresentado) tentando criar um jogo inovador, mesmo que apenas para aprendizado.

Referências

Androutsellis-Theotokis, S. and Spinellis, D. (2004). A survey of peer-to-peer content distribution technologies. *ACM Comput. Surv.*, 36(4):335–371.

Carpenter (2010). Applying risk analysis to play-balance rpgs.

Doll, S. (2002). Get yourself into a python cpickle.

Holkner, A. (2010). pyglet.

Kurose, J. F. and Ross, K. W. (2005). *Redes de Computadores ea Internet: uma abordagem top-down*. Pearson Addison Wesley.

Lutz, M. (2003). *Learning Python*. O'Reilly & Associates, Inc., Sebastopol, CA, USA.

PYGAME (2010a). display - pygame documentation.

PYGAME (2010b). event - pygame documentation.

PYGAME (2010c). font - pygame documentation.

PYGAME (2010d). image - pygame documentation.

PYGAME (2010e). News pygame - python game development.

PYODE (2010). Pyode.

Python Foundation (2010). Python programming language – official website.

PYTHON-OGRE (2010). Python-ogre - high performance gaming and graphics library for python.

Reis, C. R. (2004). *Python na Prática: Um curso objetivo de programação em Python*. Async Open Source.

Rogers, P. (2009). Pgu - phil's pygame utilities.

SOYA3D (2010). Soya3d.

Tanenbaum, A. S. and Woodhull, A. S. (2006). *Operating Systems Design and Implementation*. Prentice Hall.

The Panda3D development team (2010). Panda3d - free 3d game engine.