Bibliotecas C no Python

Reutilização de código

Python, PHP, Ruby... O problema é que, mais cedo ou mais tarde, teremos que voltar ao mundo real e usar uma biblioteca C. Acompanhe neste artigo como usar uma biblioteca OpenGL no Python.

por José María Ruiz

Beverley Bridge – www.sxc.hu

Nos anos 70 o pessoal da engenharia de software começou a imaginar um futuro perfeito. Não seria preciso escrever programas para problemas já resolvidos – o famoso "não reinvente a roda". Começou-se a falar de objetos e componentes.

Infelizmente, essa utopia não era muito diferente da previsão de que todos iriam de foguete para o trabalho e comeriam em lanchonetes espaciais. Mas a realidade é quase tão estranha: quem imaginaria ter em casa um computador, conectado a uma rede mundial?

Hoje em dia, o tipo de código mais utilizado é o que vem na forma de bibliotecas. No mundo Linux em particular, elas geralmente foram programadas em *C* ou *C++*. Devido a inúmeros fatores, essas linguagens foram as mais populares para o desenvolvimento de bibliotecas e sistemas e continuam sendo insubstituíveis em muitos âmbitos. A linguagem C sempre se destacou por sua portabilidade (é muito simples encontrar ou criar um compilador C para uma nova arquitetura de hardware), enquanto o C++ se sobressai pelo equilíbrio entre portabilidade e capacidade para grandes projetos.

OpenGL

A biblioteca que vamos usar em nosso exemplo fornece uma série de funções que nos permite manipular objetos 3D através do *OpenGL* [1]. Espantado? Na verdade, não é tão difícil. Não iremos entrar em detalhes sobre o OpenGL, só o básico.

O OpenGL é um padrão mundial criado pelo SGI (Silicon Graphics Interactive) para o desenvolvimento de aplicações 3D interativas. Traduzido para a linguagem doméstica, isso quer dizer... Games! Todos os jogos multiplataforma que utilizam 3D foram programados com OpenGL (*DirectX*, a alternativa Microsoft, funciona apenas em sistemas Intel com Windows®).

O OpenGL tem um design simples. Os objetos são compostos por vértices (um triângulo tem 3 vértices) os quais definem áreas que podem ter uma série de propriedades como cor e brilho. Uma vez definidos os objetos, passamos a criar uma série de funções que serão chamadas quando forem produzidos os eventos. O OpenGL utiliza um loop infinito que, basicamente, se dedica a colher um evento. Por exemplo, o pressionar de uma tecla chama a função destinada a administrá-lo. Simples assim.

Evidentemente programar um jogo 3D é muito complicado, mas a base é essa: fazer as imagens da tela reagirem a eventos que possam ser disparados a qualquer momento, e que a reação seja em tempo real, ou seja, sem atraso perceptível. Quem nunca sonhou com um upgrade de placa de vídeo, em busca de mais frames por segundo, para uma experiência 3D mais realista?

Na `listagem1.c` (disponível para download em [2]) podemos ver um programa OpenGL muito simples, que será, no entanto, a base para nossa biblioteca. Para compilá-lo, devemos executar `make listagem1` no diretório em que o conteúdo do arquivo `lm21_python.tar.gz` (disponível no link [2]) for descompactado.

Para que serve nosso programa? Ele gera um cubo 3D com texturas e o gira sem parar até que pressionemos a tecla ***[2]***. Então ele pára, até que comece a girar novamente quando pressionamos a tecla ***[1]***. Com os três botões do mouse, podemos definir o ângulo em que o cubo gira.

Se compilarmos o programa dessa forma, iremos obter um executável que poderemos usar. A velocidade com que o cubo gira depende do hardware.

Para fazer essa compilação, precisaremos de algo especial: o *SWIG* [3].

SWIG

O SWIG é uma aplicativo desenvolvido em 1995, no departamento de física teórica do Laboratório Nacional Los Álamos

[4]. Lá foi desenvolvida a primeira bomba atômica e, desde então, o lugar se tornou um centro de pesquisa de ponta mundial. O objetivo ao se criar o SWIG era reduzir a complexidade de muitas aplicações científicas que exigem o uso de bibliotecas em linguagens de "baixo nível" como C ou *Fortran*.

Infelizmente, trabalhar com essas linguagens é complicado. Por exemplo, desenvolver uma interface gráfica, imprescindível nos dias de hoje, é um tormento. Também são linguagens complicadas, que devem ser usadas apenas por experts – e a maior parte dos cientistas de Los Álamos não são especialistas em programação. Por isso temos bibliotecas em C e Fortran potentíssimas, mas cujo público-alvo não possui capacidade técnica para usá-las.

O que fizeram os engenheiros de Los Álamos para simplificar o uso dessas potentes ferramentas? Se dedicaram a criar bibliotecas que pudessem ser manipuladas a partir de linguagens mais claras, como *Perl* ou Python. Esse processo, além de trabalhoso, é altamente repetitivo, de maneira que decidiram não ter mais que passar por ele. Então, desenvolveram o SWIG que, de forma quase automática, gera o código necessário para se utilizar bibliotecas C em linguagens de "alto nível" (desde Python até *OCaml*, passando por *Java* ou *C#*).

O "de forma quase automática" é essencial. Teremos apenas que gerar um arquivo de interface (terminado com *.i*) e o SWIG gerará um arquivo `<nome>_wrap.c`, que será compilado junto ao original para criar a biblioteca. Dessa forma obteremos uma biblioteca dinâmica (um arquivo `.so`, que vemos aos milhares em `/usr/lib`), assim como uma biblioteca na linguagem de destino. No nosso caso, essa linguagem será Python.

Exemplo

A `listagem1.c` é um programa de exemplo em OpenGL. É basicamente um cubo, que gira e responde a eventos. Ele possui uma série de propriedades que não podem ser alteradas, como a velocidade de giro e a cor. Na `listagem3.c` (download em [2]) temos a biblioteca equivalente necessária para nossos propósitos. Vamos examiná-la.

Para começar, por se tratar de uma biblioteca, ela não pode ter a função `main`, por isso a renomeamos para `arranque()`. Foram incorporadas novas funções com o objetivo de podermos modificar algumas características do objeto (cor, velocidade, título da janela...).

O OpenGL é composto pelas bibliotecas *GL*, *GLU* e *GLUT*. Essa última é muito importante, mas, infelizmente, a biblioteca GLUT padrão tem um defeito (ou, um problema, dependendo de como olharmos). A função `glutMainLoop()` é o loop infinito de gestão de eventos. Podemos entrar nela, mas jamais sair. Quando se dispara o evento que provoca a saída do loop, o programa termina. Qualquer coisa que seja colocada depois da chamada ao `glutMainLoop()` não será mais executada.

Isso não parece muito lógico, não é mesmo? E se quisermos iniciar o gerenciador de eventos, pará-lo e iniciá-lo novamente? Como não somos os únicos com esse "problema", alguns valentes, na melhor tradição do Software Livre, criaram o *FreeGLUT* [5]. Basicamente, ele é um GLUT que permite sair do loop interno do `glutMainLoop()` sem que o programa seja finalizado, entre outros recursos.

E por que queremos que isso ocorra? Como precisamos de uma biblioteca, queremos que a função `arranque()` volte ao programa principal (o intérprete de Python) assim que finalize a execução, ao invés de finalizar todo o programa.

Esse é um problema típico que podemos encontrar ao criarmos bibliotecas para o Python usando bibliotecas C. Às vezes, o que queremos usar são as funções de um programa já existente (por exemplo, imagine ter acesso às funções do *Mozilla* ou *Firefox* para *RSS* ou análise de *HTML*), de modo que os passos serão: eliminar os pontos de saída do programa (geralmente os `exit()`) e eliminar o `main`.

Assim já temos nossa biblioteca, pronta para executar o SWIG sobre ela.

Arquivo de interface

O SWIG necessita de um arquivo que diga quais funções e definições de dados devem ser exportadas para o Python. Esse arquivo se parece muito com os arquivos de cabeçalho (*headers*) em C. Dê uma olhada em `/usr/include` para ver alguns deles. O arquivo começa com a sentença

```
%module listagem3
```

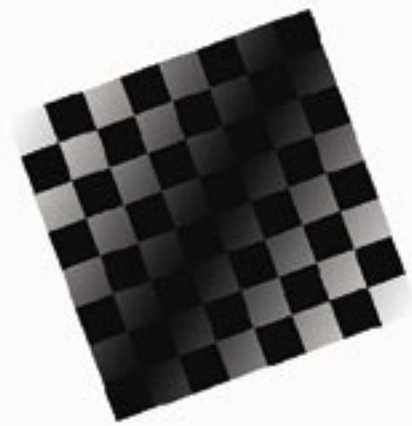

Figura 1: O cubo inicial, da maneira como é visto na janela do programa de exemplo.

Figura 2: Com o teclado podemos mudar a posição e a direção do giro do cubo.

que declara o nome do módulo e deve ser igual ao nome de nossa biblioteca C. Esse nome será exportado para cada linguagem, de maneira que se corresponda com sua estrutura de organização. No caso do Python, o nome do módulo se corresponderá com um pacote.

O resultado final da execução do SWIG será um arquivo chamado `listagem3_wrap.c`. É um arquivo em C, que define uma série de funções que permitirão o acesso à biblioteca `listagem3.c`, definidas no arquivo `listagem3.i`.

Algumas das definições que serão colocadas no arquivo `listagem3.i` devolvem dados de tipos definidos em OpenGL, motivo pelo qual temos que dizer ao SWIG que o arquivo `listagem3_wrap.c` deve incorporar o cabeçalho:

```
#include <GL/glut.h>
```

Isso se faz mediante a declaração:

```
%{
#include <GL/glut.h>
%}
```

Entre os símbolos `%{` e `%}` pode ser introduzida qualquer declaração extra ou incluído código que não exista no arquivo `listagem3.c`. Digamos que essa é uma maneira de isolar um código específico para SWIG, relacionado às bibliotecas originais.

Bem, está quase terminado. Falta apenas declarar os tipos de dados e cabeçalhos necessários. Podemos ver as definições no arquivo `listagem3.i` (download em [2]). Basicamente são os cabeçalhos das funções que queremos usar no Python com a palavra `extern` na frente. Essa palavra diz ao SWIG que são funções que existem em um diretório externo.

Palavras mágicas

A "poção" está quase pronta. Agora só temos que usar as palavras mágicas do arquivo `Makefile` [2] que foi usado na compilação da `listagem1.c`. Agora é preciso executar o comando `make`. Isso vai executar os seguintes comandos:

```
swig -python listagem3.i
gcc -c -fPIC listagem3.c $(INCS)
gcc -c -fPIC listagem3_wrap.c$(PYINCS) $(INCS)
gcc -shared listagem3.clistagem3_wrap.c -o _listagem3.so ➥
$(INCS) $(PYINCS) $(LIBS) $(PYLIBS)
```

O primeiro comando gera o arquivo `listagem3_wrap.c`, bem como o `listagem3.py`. Há uma opção, `-python`, que poderia ser trocada por `-perl` ou `-java`. Essa opção determina a linguagem que será utilizada para gerar a biblioteca.

Obviamente, esses arquivos não são nada mais que código a ser compilado. Disso se encarregam os três comandos seguintes. O primeiro compila nossa biblioteca, o segundo os arquivos gerados pelo SWIG e o terceiro une tudo para gerar a biblioteca dinâmica.

Esses `$(INC)` ou `$(PYLIBS)` são opções de compilação guardadas em variáveis de controle no `Makefile`. Variam de sistema para sistema, mas as que aparecem em `Makefile` são as mais comuns.

Nova biblioteca

Por incrível que pareça, terminamos. Já temos nossa biblioteca. Se dermos uma olhada no diretório em que realizamos todas as ações, veremos um arquivo chamado `_listagem3.so`. É uma biblioteca dinâmica que pode ser carregada a qualquer momento. O que estamos esperando? Vamos iniciar o intérprete Python:

```
> python
Python 2.4 (#2, Apr 3 2006.22:24:02)
[GCC 3.4.2 [FreeBSD] 20040728] on freebsd5
Type "help", "copyright"." credits" or "license" for more ➥
information.
>>> import listagem3
>>>
```

Com o comando `import`, acabamos de carregar a biblioteca e a temos pronta para uso. Vamos começar pela ação mais simples: iniciar o programa com as opções padrão:

```
>>>listagem3.arranque()
```

Ao teclarmos ***[Enter]***, aparecerá uma janela na tela com um objeto que se move numa grande velocidade. É o nosso cubo, se movendo rapidamente. Se pressionarmos a tecla ***[2]*** o cubo pára. Para colocá-lo em movimento novamente, devemos pressionar ***[1]***.

Se usarmos os botões do mouse enquanto o cubo gira, veremos que cada botão muda a direção de giro para um sentido diferente. Se estivermos cansados do cubo ou com enjôos porque o computador é muito rápido, é só pressionar ***[Q]***: a janela desaparecerá. Voltamos a ver o prompt `>>>` do Python. Isso já é um sucesso: conseguimos executar uma função C que interage com o OpenGL a partir do Python. Mas, mesmo assim, não fizemos nada – poderíamos ter feito o mesmo com um shell script!

Vamos interagir mais com a biblioteca. Se olharmos o arquivo de interface `listagem3.i`, veremos as funções que estão disponíveis para uso a partir do Python. Vamos mudar a velocidade de giro para podermos ver o cubo.

```
>>> listagem3.setTiempoEspera(10000000)
>>> listagem3.arranque()
```

Para não complicar o código, foi introduzido um loop que faz determinado número de iterações na função `display()` do `listagem3.c`. Essa função é invocada pelo OpenGL para desenhar os objetos na tela. Se nela for introduzido um atraso, irá demorar mais para realizar o seu trabalho. Isso se traduz em uma visualização mais clara do cubo girando, se conseguirmos realizar o atraso adequado.

No nosso sistema, um valor de 10.000.000 de iterações faz com que o cubo gire de maneira suave. Evidentemente, por ser um loop, depende da velocidade do sistema. Não é a maneira mais elegante de introduzir um atraso, mas é a mais simples e curta (nosso espaço aqui é limitado...).

A variável `espera` no arquivo `listagem3.c` é do tipo *INT*. Isso significa que seu valor máximo será um pouco mais de 4.000.000.000. O que ocorrerá se introduzirmos um atraso de 5.000.000.000?

```
>>> listagem3.setTiempoEspera(5000000000)
Traceback (most recent call last): file "<stdin>", ➥
line 1, in ?
OverflowError: argument number 1: long int too large to ➥
convert to int
>>>
```

O Python permanece dizendo que a instrução não pode ser realizada. Uma falha desse tipo em um programa C teria levado a uma falha irreversível que abortaria o programa. No Python não. Simplesmente voltamos ao intérprete. Essa é uma das razões pelas quais é interessante usar bibliotecas C no Python. Eliminamos muitos erros fatais porque o Python se encarrega de verificar por nós os tipos de dados e é capaz de recuperar-se dos erros.

Vamos brincar com as cores do cubo. O cubo tem 6 lados. O normal seria atribuir uma cor para cada um. Mas o OpenGL dá cores aos vértices. Como o cubo tem 8 vértices, teremos que assinalar 8 cores. Que cor terá cada face do cubo se cada vértice tem uma cor? Pois, seguindo o jargão do OpenGL, terá uma cor "interpolada" – uma mescla das quatro cores dos vértices.

Uma das nossas funções permite definir a cor dos vértices (`setColor()`), de modo que deixaremos as faces cinzas. Daremos a um vértice a cor preta e ao seguinte, branca. No `exemplo.py` podemos ver um programa que atribui as cores branca e preta alternadamente aos vértices. Mas o resultado não é de todo cinza, por quê?

Porque depende da maneira que os vértices foram definidos. Os vértices 0 e 1 não têm porquê estar no sentido correto. Pode-se brincar com o código do `exemplo.py` para tentar obter outras cores.

Conclusão

Não é tão complicado acessar milhares de bibliotecas C a partir do Python. Na própria documentação do SWIG, aparece um exemplo de como gerar uma biblioteca Python de maneira quase automática para a famosa biblioteca *GD* de desenho 2D.

Com certeza, na próxima vez que virmos uma nova biblioteca para Python, iremos nos perguntar se os autores usaram SWIG – pode ser inclusive que tenha sido feita por nós. ■

Informações

[1] OpenGL: http://www.opengl.org

[2] Download de códigos deste artigo: http://www.linuxmagazine.com.br/issue/21/lm21_python.tar.gz[2]

[3] Projeto SWIG: http://www.swig.org[1]

[4] Laboratório Nacional Los Álamos: http://www.lanl.gov

[5] Projeto FreeGLUT: http://freeglut.sourceforge.net

Autor

José María Ruíz está finalizando seu projeto de conclusão de curso de Engenharia Técnica em Informática de Sistemas e está há mais de sete anos usando e desenvolvendo Software Livre, há dois anos no FreeBSD. Atualmente, trabalha na Animatika, empresa de software livre.