Pure Graphics Programming – Conectando GLSL com C++

1. Sumário

2. Introdução

Pure Graphics Programming (PGP) é uma fina camada de software sobre a arquitetura da API OpenGL 2.1. O objetivo dessa camada é facilitar o aprendizado de programação gráfica 3D com *pipeline programável*, um tópico espinhoso para iniciantes em programação gráfica. Contudo, *PGP* não tem o objetivo de fornecer uma abstração como um grafo de cena, como faz Java3D, por exemplo. O objetivo é facilitar um primeiro contato do iniciante em programação gráfica para que entenda como conectar os dados de seu programa cliente (em C++) com a memória dos programas que rodam na Unidade de Processamento Gráfico (GPU – de *Graphics Processing Unity*). O nome desses programas que rodam na GPU são conhecidos como *shaders* e são compostos de módulos que correspondem às diferentes etapas da *pipeline* de renderização da OpenGL. Os *shaders* são escritos em uma linguagem própria, chamada GLSL (*OpenGL Shadding Language*). Para saber mais sobre GLSL, acesse o sitio oficial do grupo que mantém a linguagem:

https://www.khronos.org/opengl/wiki/OpenGL_Shading_Language.

Alguns tutoriais que explicam como fazer a comunicação entre OpenGL e GLSL são:

- http://www.lighthouse3d.com/tutorials/glsl-12 tutorial/communication-opengl-shaders/
- https://pt.wikibooks.org/wiki/Programa%C3%A7%C3%A3o_com_OpenGL/Modern_OpenGL_Introduction

Antes de utilizarem PGP, recomendo que tentem fazer um exemplo simples, como desenhar um triângulo, utilizando somente OpenGL e GLSL. Neste manual, é suposto que o leitor já tenha conhecimento básico de OpenGL e GLSL e, portanto, restringe-se a explicar brevemente o princípio geral de funcionamento da PGP e como o programador pode utilizar suas camadas para criar rapidamente programas gráficos.

3. O que PGP faz?

PGP facilita a comunicação entre OpenGL e GLSL, mas não esconde OpenGL do usuário. Isso significa que você terá que saber OpenGL 2.1 para programar usando PGP. A vantagem de PGP é que automatiza algumas tarefas e simplifica o processo de comunicação entre OpenGL e GLSL. O custo dessa simplificação é a tomada de algumas suposições que podem impactar o desempenho final da aplicação. Ou seja, talvez PGP não seja a forma mais otimizada de fazer as coisas, pois o objetivo dessa camada de software é simplificar a programação gráfica para iniciantes, sem esconder o que está por trás. E isso é um desafio em tanto!

4. Como PGP faz o que Faz?

Par alcançar o seu objetivo, PGP possui duas camadas, dois níveis de programação: *ShaderPoxy* e *Scene*.

ShaderProxy é a camada responsável por facilitar a comunicação entre OpenGL e GLSL. E faz somente isso.

Scene é a camada que fornece um maior nível de abstração, mas não fornece um grafo de cena, como Java 3D. *Scene* fornece classes que permitem reduzir a quantidade de código necessária para se criar aplicações simples. A camada *Scene* é construída em cima de *ShaderProxy*.

A Tabela 1 mostra as principais classes de cada camada e quais os seus objetivos.

Tabela 1. Classes da PGP.

Classe	Camada	Objetivo
ShaderProxy	*ShaderProxy*	Facilitar a comunicação entre OpenGL e GLSL.
Object	*Scene*	Encapsula atributos e informações pré-definidas que permitem, de forma rápida, criar objetos iluminados e que podem sofrer transformações geométricas.
ShadedObject	*Scene*	Um objeto conectado com um *shader* (via *ShaderProxy*), o que permite aplicar transformações e fazer cálculos de iluminação com poucas linhas de código.
Camera	*Scene*	Encapsula transformações de câmera, de modo que seja rápido e fácil criar uma câmera e movimentá-la na cena.

Como mostrado na Tabela 1, poucas classes são fornecidas, o que contribui com o objetivo de ser uma camada de software simples para o aprendizado de programação gráfica. Se mais abstrações forem criadas, pode-se obter o efeito contrário do pretendido, pois a complexidade de compreender uma interface mais complexa se

somaria com a dificuldade de aprender o OpenGL em si, dado que o objetivo desta camada de software é servir de exemplo para aprendizes e não esconder OpenGL deles.

5. Onde encontrar e onde baixar PGP?

PGP está disponível no site https://github.com/professorgilzamir/pgp. Devido estar em uma versão bastante instável, a interface de suas classes ainda é passível de alteração, mas este manual se manterá atualizado de acordo com as alterações em código que por ventura venham a ocorrer.

6. Por que PGP?

PGP foi pensada para ser utilizada como exemplo em uma disciplina introdutória e inicial de Computação Gráfica. O objetivo da disciplina é abordar os conceitos de computação gráfica de forma prática e teórica, simultaneamente.

7. Exemplos

A melhor forma de utilizar PGP é baixando a sua última versão disponível no github: https://github.com/professorgilzamir/pgp. Ao fazer download ou clonar este repositório, você estará com a última versão disponível da PGP. Após baixar o repositório, você encontrará no diretório *src* um *template* de projeto nomeado como *project_template.zip.* Para iniciar um novo projeto, você precisa descompactar este arquivo. Nisso será gerado um diretório que conterá um projeto básico. Contudo, você pode utilizar qualquer um dos muitos exemplos disponíveis. Leia a documentação oficial da PGP em https://github.com/professorgilzamir/pgp. Na página principal é explicado o que cada grupo de exemplos faz.

Mãos à massa!! Espere um pouco! É preciso primeiro saber se o seu computador está preparado para rodar PGP e OpenGL.

8. Requerimentos

Para rodar os exemplos disponíveis no sitio do projeto, qualquer computador com suporte a OpenGL 2.1 pode ser utilizado. Os sistemas operacionais recomendados são GNU/Linux e MacOSX. Contudo, os exemplos também funcionam no Windows 10 e há suporte para compilação neste sistema operacional com a ajuda de software de terceiro. Veja como rodar PGP no seu computador em https://github.com/professorgilzamir/pgp/blob/master/src/README.md.

9. Arquitetura do Programa com *ShaderProxy*

A arquitetura de uma aplicação usando PGP segue o objetivo de diminuir a quantidade de código necessária para se realizara comunicação entre OpenGL e GLSL. A Figura 1 mostra que um programa em OpenGL deve mapear seus dados diretamente para GLSL por meio de atributos (dados que variam entre vértices) e dados uniformes (que não variam entre vértices). Observe que dados qualificados com *varying* não são da alçada de PGP, pois são dados trocados entre *VertexShader* e *FragmentShader*. O

problema da arquitetura na Figura 1, é que o programa cliente em OpenGL deve realizar todo o trabalho burocrático de mapeamento de dados de diferentes tipos entre OpenGL e GLSL. A solução para isso é ilustrada na Figura 2.

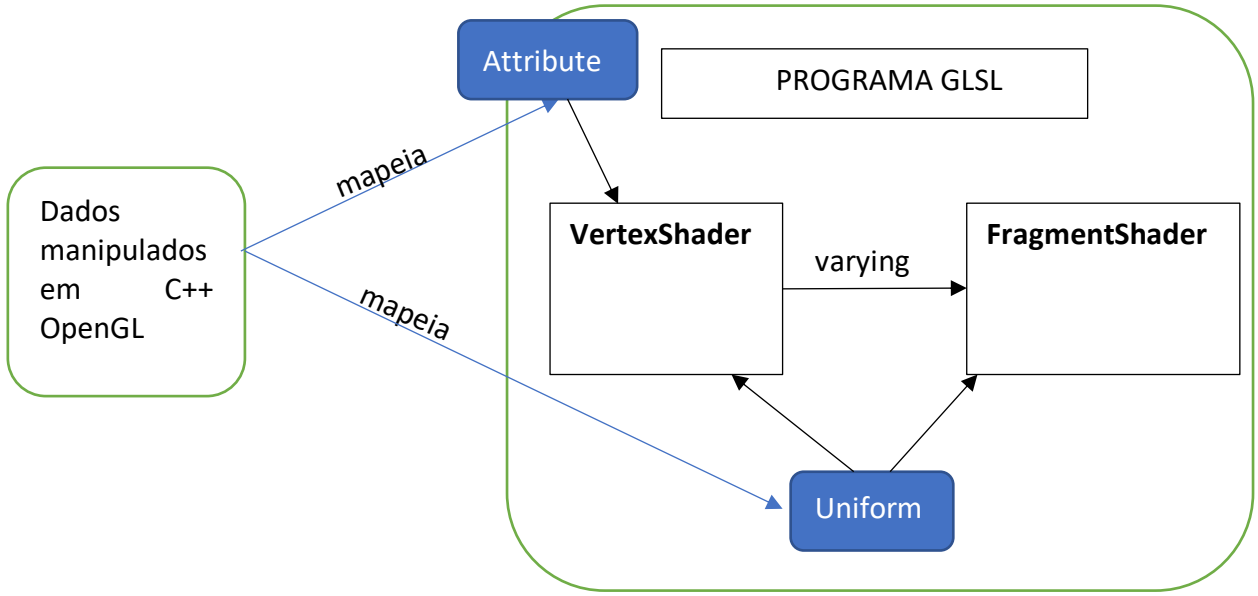

Figura 1. Arquitetura de uma aplicação OpenGL típica sem PGP.

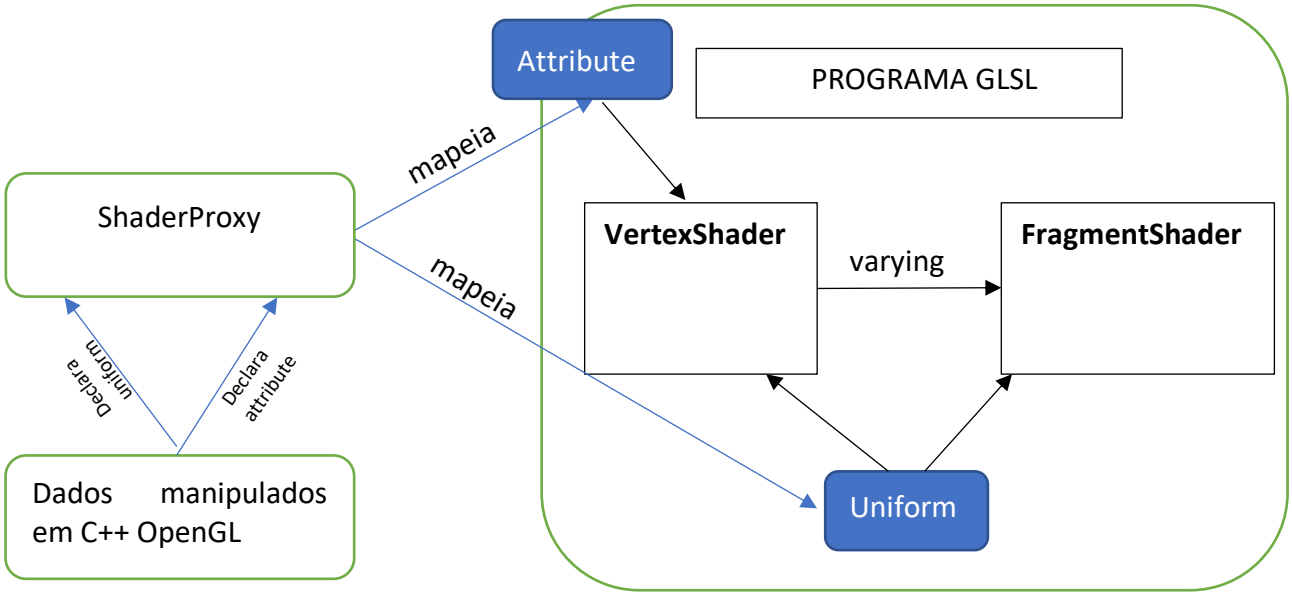

Figura 2. Arquitetura de uma aplicação com PGP.

Como mostrado na Figura 2, A aplicação com PGP apenas declara quais dos seus dados correspondem à atributos e quais correspondem à *uniforms*. O trabalho pesado de criação de *buffer* e envio de dados para os *buffers* é feita pela camada *ShaderProxy*.

Assim, o programador pode se concentrar mais na lógica da interação do programa e nas funcionalidades gráficas implementadas nos *shaders* do que no trabalho burocrático ocasionado pelo mapeamento entre dados OpenGL e GLSLS.

A camada *ShaderProxy* está encapsulada no *namespace shaderutils.* Além da classe *ShaderProxy* em si, há vários métodos acessórios que procuram simplificar por meio de parâmetros *default* as chamadas de métodos equivalentes em *OpenGL*. A compilação e ligação dos *shaders*, por exemplo, é automática. Consulte os exemplos e os comentários de código para entender como cada método funciona. A camada *Scene* está encapsulada no *namespace scene*.

10. Um Primeiro Exemplo

A primeira coisa a se fazer para se criar um programa em OpenGL é criar uma versão básica em GLSL de seus *shaders* e adaptá-la ao longo do desenvolvimento para as necessidades que forem surgindo. Ou então utilizar *shaders* já desenvolvidos com suporte a gráficos complexos. Para manter as coisas simples, vamos criar um projeto em um diretório separado, nomeado como *exemplo*. Dentro deste diretório, cole os diretórios *lib* e *includes* presentes no repositório da PGP no *Github*. Crie outro diretório chamado *src*, onde você salvará os próximos arquivos. Agora siga os próximos passos para realizar um exemplo simples de desenho de um triângulo.

Dentro de **exemplo/src**, salve os seguintes arquivos (disponíveis no repositório da PGP do *Github*): *Makefile, Makefile.linux, Makefile.windows, Makefile.macosx, scene.hpp, shaderutils.hpp, matrixmath.hpp e utils.hpp.* Além desses arquivos, adicione os binários das bibliotecas necessárias para OpenGL 2.1 funcionar em seu sistema operacional. Recomendo adicionar no diretório do código-fonte do programa todos seguintes arquivos para ter compatibilidade nos sistemas operacionais mais usados atualmente: ***glew32.dll***, ***libGLEW.a*** e ***libglew32.dll.a***. Estes arquivos podem todos serem encontrados no repositório da PGP no *Github*.

Crie dois arquivos: *shader.vs* e *shader.fs*. No primeiro, adicione o código do seu *VertexShader*:

```
#version 120
atribute vec2 posicao;
void main(void)
{
          //posicao final do ponto
          gl_Position = vec4(posicao, 0, 1);
}
```

No segundo arquivo, coloque o código do *Fragment Shader*:

```
#version 120

void main(void)
{
          //Define a cor do pixel como vermelho
        gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

Agora crie um arquivo *ex.cpp* dentro de *exemplo/src,* que conterá o código de nosso programa. Para utilizar a interface *ShaderProxy*, deve-se incluir em *ex.cpp* o cabeçalho *shaderutils.hpp*:

```
#include “shaderutils.hpp”
```

Além disso, para facilitar a digitação do código, pode-se declarar o uso do espaço de nome *shaderutils*:

```
using  namespace  shaderutils;
```

Em seguida, declaramos em escopo global a variável do tipo *ShaderProxy* que fará a comunicação entre OpenGL e GLSLS:

```
ShaderProxy *proxy = 0;
```

Crie o vetor que contém os dados do triângulo que queremos desenhar:

```
GLfloat pontos[] = {
                0.0f, 0.0f,
                0.5f, 0.0f,
                0.0f, 0.5f
};
```

Depois crie uma função na qual criaremos o objeto *ShaderProxy* e realizaremos a declaração do atributo GLSL que deve ser mapeado aos pontos do triângulo que queremos desenhar:

```
void init() {

         proxy = new ShaderProxy(genProgram("shader.vs",
         "shader.fs"));

         proxy->setAttribute("posicao", pontos, sizeof(pontos), 2);
}
```

Para criar uma instância do objeto *ShaderProxy*, deve-se antes criar um programa GLSL e informar o identificador do programa GLSL no construtor da classe *ShaderProxy*. A função *genProgram* (disponível em *shaderutils.hpp*) compila e liga os *shaders* cujos caminhos são indicados com duas *strings* como argumento.

O próximo passo é a criação de uma função que invocará as funções necessárias para limpar a tela e desenhar o nosso triângulo:

```
void onDraw() {
         glClearColor(0.8, 0.8, 0.8, 1.0);
        glClear(GL_COLOR_BUFFER_BIT);
//Ativa o programa do proxy
         proxy->useProgram();
//desenha os pontos como um triângulo
         proxy->drawArrays(GL_TRIANGLES);
         glutSwapBuffers();
}
```

Finalmente, criamos o contexto gráfico necessário para o nosso programa:

```
int main(int argc, char **argv) {
          glutInit(&argc, argv);
          glutInitDisplayMode(GLUT_RGBA|GLUT_DOUBLE);
          glutInitWindowSize(400, 400);
          glutCreateWindow("Hello ShaderProxy");

          init();

          glutDisplayFunc(onDraw);
          glutMainLoop();

          proxy->close();
          delete proxy;
          proxy = 0;
          return 1;
}
```

Utilizamos GLUT como gerenciador de janelas neste exemplo. O código completo se encontra em http://github.com/professorgilzamir/pgp/src/exemplo.

Além do exemplo usado neste tutorial, o repositório de código da PGP possui muitos outros exemplos divididos em capítulos. Explore os capítulos de forma progressiva, do diretório cap1_ex* ao diretório cap10_ex*, para perceber diferentes funcionalidades em ação.

11. Utilizando a camada Scene: a classe *ShadedObject*

Se quisermos adicionar transformações de forma rápida no exemplo anterior, podemos utilizar a classe *ShadedObject*. Um objeto *ShadedObject* é associado a *ShaderProxy* e a um objeto de dados interno que facilita a realização de transformações geométricas e de cálculos de iluminação no objeto. Contudo, o objeto interno trabalha com informações tridimensionais, mesmo se quisermos apenas desenhar um triângulo.

No exemplo desta seção, vamos alterar o programa anterior de modo que o triângulo gire em torno da origem sempre que uma tecla for pressiona. A primeira coisa a ser alterara é o *vertex shader* (arquivo *shader.vs*). No *vertex shader*, precisamos adicionar uma matriz que representará a transformação de rotação:

```
#version 120

attribute vec2 posicao;
uniform mat4 transform;

void main()
{
          gl_Position = transform * vec4(posicao, 0, 1);
}
```

O nome da matriz adicionada foi *tranform* e é do tipo matrix 4x4, representada por *mat4*.

O *fragmente shader* não precisa ser alterado. Precisamos agora alterar o arquivo *ex.cpp*. Neste arquivo, adicione o cabeçalho *scene.hpp*:

```
#include “scene.hpp”
```

Ainda em *ex.cpp*, declare o *namespace scene*:

```
using namespace scene;
```

Continuando no arquivo *ex.*cpp, em escopo global e depois da declaração do *namespace scene*, declare a variável *triangle* e a inicialize com o valor zero:

```
ShadedObject *triangle = 0;
```

No método *init*, criamos o objeto *ShadedObject* depois da criação e configuração do objeto *ShaderProxy*. Depois de criarmos o objeto *ShaderProxy* e atribuirmos para o ponteiro *triangle*, devemos indicar que qualquer transformação no objeto deve afetar a variável *transform* declarada e usada em nosso *vertex.vs*. Isso é feito alterando a variável *transformName* do objeto *ShadedObject*. O método *init* modificado completo fica do seguinte modo:

```
void init() {
         proxy = new ShaderProxy(genProgram("shader.vs",
"shader.fs"));
          proxy->useProgram();
          proxy->setAttribute("posicao", pontos, sizeof(pontos), 2);
          triangle = new ShadedObject("triangle", proxy);
          triangle->transformName = "transform";
          triangle->update();
}
```

Agora vamos criar uma nova função que será executada sempre que qualquer tecla do teclado for pressionada. Chamaremos essa função de *onKeyPress*. O método *onKeyPress* realiza uma rotação no triângulo. Para uma rotação no plano bidimensional, deve-se realizar a função *rotateZ(ângulo_rotacao)* do objeto *ShadedObject* que se quer rotacionar:

```
void onKeyPress(unsigned char key, int x, int y){
          triangle->data.rotateZ(1);
          triangle->update();
          glutPostRedisplay();
}
```

Observe que, uma vez que associamos o *ShaderProxy* ao objeto *ShadedObject* apontado por *triangle*, toda transformação que for feita por meio de *triangle* afetará o resultado final dos dados mapeados pelo *ShadedProxy* relacionado.

Finalmente, alteramos o método *main* apenas para registrar que a função *onKeyPress* deve ser executada sempre que uma tecla for pressionada:

```
int main(int argc, char **argv) {
          glutInit(&argc, argv);
          glutInitDisplayMode(GLUT_RGBA|GLUT_DOUBLE);
          glutInitWindowSize(400, 400);
          glutCreateWindow("Hello ShaderProxy");

          init();

          glutDisplayFunc(onDraw);
          glutKeyboardFunc(onKeyPress);
          glutMainLoop();

          delete triangle;
          triangle = 0;

          proxy->close();
          delete proxy;
          proxy = 0;
          return 1;
}
```

O código completo se encontra em http://github.com/professorgilzamir/pgp/src/exemplo_objeto.

12. Utilizando a camada Scene: a classe Camera

Podemos ainda alterar detalhes de nossa visualização, como a forma de projeção dos objetos a serem desenhados. Para manter a simplicidade do código, continuaremos com o exemplo bidimensional de desenhar um triângulo. Continuando o exemplo da Seção 11, vamos criar uma câmera que altera a escala das medidas das coordenadas da OpenGL. Por padrão, os valores de coordenadas devem variar no intervalo [-1, 1] em cada dimensão. Para alterarmos isso no caso bidimensional, vamos utilizar uma câmera com projeção ortográfica. Mas primeiramente vamos alterar o nosso *VertexShader* par:

```
#version 120

attribute vec2 posicao;
uniform mat4 transform;
uniform mat4 view;
uniform mat4 projection;

void main()
{
   gl_Position = projection * view * transform * vec4(posicao, -1, 1);
}
```

Para trabalharmos com câmera, nosso *shader* precisa ter duas matrizes: *view* e *projection*. Estas matrizes definem a orientação e a posição da câmera, no caso da *view*, e o tipo de projeção realizada pela câmera, no caso, a matriz *projection*. Agora, precisamos alterar o arquivo *ex.cpp* de modo a criarmos o objeto Camera que altera estas variáveis. Dessa forma, depois da declaração da variável do tipo *ShadedObject**, declare a variável que apontará para a nossa câmera:

```
Camera *camera = 0;
```

Vamos mudar a escala do intervalo de valores das coordenadas, no lugar de trabalharmos com coordenadas variando no intervalo [-1, 1], iremos trabalhar com valores de coordenadas no intervalo [-100, 100]. Antes disso, altere os dados do nosso triângulo para que suas coordenadas correspondam ao novo intervalo:

```
GLfloat pontos[] = {
          0.0f, 0.0f,
          50.0f, 0.0f,
          50.0f, 50.0f
};
```

Altere a função *init* de modo a instanciar a nova câmera e modifica-la para realizar uma projeção ortográfica:

```
void init() {
           proxy = new ShaderProxy(genProgram("shader.vs",
          "shader.fs"));
           proxy->useProgram();
           proxy->setAttribute("posicao", pontos, sizeof(pontos), 2);
           triangle = new ShadedObject("triangle", proxy);
           triangle->transformName = "transform";
           triangle->update();
           camera = new Camera(proxy);
           camera->setOrtho(-100.0f, 100.0f, -100.0f, 100.0f, 0.001,
          1000);
           camera->update();
}
```

Observe que, assim como quando alteramos o objeto *ShadedObject*, toda vez que alterarmos os dados da câmera, devemos executar o método *update*. Finalmente, alteramos o método *main* para desalocar a memória usada pela câmera:

```
int main(int argc, char **argv) {
          glutInit(&argc, argv);
          glutInitDisplayMode(GLUT_RGBA|GLUT_DOUBLE);
          glutInitWindowSize(400, 400);
          glutCreateWindow("Hello ShaderProxy");

          init();

          glutDisplayFunc(onDraw);
          glutKeyboardFunc(onKeyPress);
          glutMainLoop();

          delete camera;
          camera = 0;

          delete triangle;
          triangle = 0;

          proxy->close();
          delete proxy;
          proxy = 0;
          return 1;
}
```

O código completo se encontra em http://github.com/professorgilzamir/pgp/src/exemplo_camera. Exemplos mais complexos podem ser encontrados no mesmo repositório no *Github*.

13. Considerações Finais

A versão atual da PGP já cumpre seu objetivo de simplificar sobremaneira a criação dos primeiros exemplos de programas que usam OpenGL. No entanto, a interface do software poderá sofrer alterações para incorporar novas funcionalidades.

Tanto este manual quanto o código da PGP serão aperfeiçoados até que se chegue a uma versão estável do software. A versão da PGP ainda se encontra bastante instável, assim sua *interface* pode mudar com o tempo, mas mantendo o máximo de retrocompatibilidade possível.