Álbum de fotos 3D implementado com OpenCV e OpenGL
Computação Visual
Universidade de Aveiro

Diogo Silva 60337

Resumo **– Este relatório descreve detalhadamente a estrutura, filtros e o motor da aplicação implementada em OpenGL e OpenCV. Descrevendo detalhadamente, método de manipulação, método de visualização, entre outros.**

Nota

Toda a documentação relativa a complicação encontra-se num ficheiro à parte com o nome README (aconselha-se que seja lido antes de tentar executar a aplicação desenvolvida), ou no fim deste relatório.

I. Estrutura da Aplicação

A aplicação está organizada de forma a ser reutilizável, sendo assim, existe várias componentes distintas:

1. Efeitos, manipulação de cada imagem, mais precisamente, aplicação de filtros sobre imagens, tendo um modo personalizado ou com valores de entrada por defeito
2. Modelos, contém todas as caracterísiticas necessárias para a representação de um modelo no ambiente OpenGL
3. Shaders, trata do fazer o processamento respectivo na placa gráfica
4. Utilidades, contém as utilidades para o sistema, neste momento, contém apenas MathUtils que permite manipular matrizes e vectores
5. Visualizador Gráfico (animações), manipulação de vários modelos para criar animações, contendo as respectivas acções, como primir um botão

Podendo assim facilmente adicionar um tema gráfico com novas animações, ou adicionar novos filtros ao sistema.

A. Implementação

Neste capítulo é abordado todas as implementações das classes principais, deixando de lado, classes que herdem interfaces, sendo essas interfaces abordadas.

A.1 Efeitos

Nesta secção é abordado a implementação da estrutura efeitos e não o detalhe de cada filtro, esse assunto é abordado na secção de Aspectos Importantes do Trabalho.

A class `Effect` no ficheiro `effects/effect.h|cpp` contém a interface usada por qualquer efeito, sendo que cada filtro vai ter de herdar esta classe de forma a ficar um sistema escalável, sendo que a função principal desta classe é a `ApplyEffect` que faz a respectiva manipulação de cada imagem consoante o filtro de que se trata.

Na seguinte imagem é possível verificar a implementação da respectiva classe:

```
 8  #ifndef EFFECT_H
 9  #define EFFECT_H
10
11  #include <string>
12  #include "opencv2/core/core.hpp"
13  #include "opencv2/imgproc/imgproc.hpp"
14  #include "opencv2/highgui/highgui.hpp"
15  #include "opencv2/photo/photo.hpp"
16
17  using namespace std;
18  using namespace cv;
19
20  class Effect {
21  public:
22      Effect(string previewImage, bool);
23      ~Effect();
24
25      string getPreviewImagePath();
26      virtual Mat applyEffect(Mat, vector<void*>);
27      virtual string getEffectName();
28      virtual vector<void*> readParameters();
29      virtual vector<void*> requestDefaultParameters();
30      bool custom;
31  private:
32      string image;
33      string effectName;
34  };
35
36  #endif
37
```

Fig. 1
Implementação do Effect

Para além desta classe, ainda existe outra classe que permite armazenar todos os filtros existentes fazendo a manipulação deles a partir dela, tendo a mesma a função `ApplyEffect` mas com um campo adicional que indica o filtro respectivo.

Esta classe armazena todos os filtros existentes na sua construção (ou seja, no construtor), armazenando-os todos num vector de Effect fazendo assim proveito do polimorfismo.

Como se pode ver na imagem seguinte esta classe é muito parecido à `Effect` com a excepção que tem um vector de Effect.

A.2 Modelos

Todos os modelos que precisam de ser representados em OpenGL são carregados para uma classe chamada `GraphicModel` que permite guardar a seguinte informação:

1. Número de vertíces
2. Lista de vertíces
3. Lista das normais
4. Lista das texturas
5. Matrix da imagem representada
6. Matrix da imagem carregada inicialmente
7. Valores de deslocamento, rotação e de redimensionamento
8. ID da textura actual

Como se pode verificar na imagem seguinte:

```
20  using namespace cv;
21
22  class GraphicModel {
23  public:
24      /* Coordenadas dos vertices */
25      GraphicModel();
26      ~GraphicModel();
27
28      int numVertices;
29      vector<float> arrayVertices;
30      vector<float> arrayNormais;
31      vector<float> arrayTextures;
32      string filepath;
33      Mat image, original;
34      GLuint textureID;
35      /* Parametros das transformacoes */
36      Point3_<double> desl;
37      Point3_<double> anguloRot;
38      Point3_<double> factorEsc;
39  };
40
41  #endif
```

Fig. 2
Implementação do GraphicModel

Em que estes dados todos são úteis para a representação no modelo no ambiente gráfico, a excepção da Matrix inicial carregada que serve para dar a possibilidade ao utilizador de voltar a imagem inicial.

A.3 Controlador de temas

Tal como nos efeitos, para os temas da representação gráfica existe uma classe que funciona como uma interface que permite criar outros temas herdando essa classe, que é a `Theme` no ficheiro `visualization/theme.h|cpp`.

Essa classe tem todas as funções respectivas aos movimentos básicos, tais como, mover o tema para a direita, para a esquerda, para baixo, ampliar, entre os restantes. Para a implementação de cada tema é passado os modelos gráficos de todas as imagens carregadas do directório, sendo esta classe responsável pela manipulação das suas respectivas deslocações e orientações.

Como se pode ver na imagem seguinte, temos a interface pronta a ser herdada por qualquer tema que se pretenda que seja criado.

```
class Theme {
public:
    static Theme* getInstance(vector<GraphicModel> *

    virtual ~Theme();

    virtual void initTheme(void);
    virtual void pressLeft(void);
    virtual void pressRight(void);
    virtual void pressUp(void);
    virtual void pressDown(void);
    virtual void zoomIn(void);
    virtual void zoomOut(void);
    virtual void animation_moveLeft(int);
    virtual void animation_moveRight(int);
    virtual void animation_moveUp(int);
    virtual void animation_moveDown(int);
    virtual void animation_zoomIn(int);
    virtual void animation_zoomOut(int);
    int currentPos;
    bool animationActive;
protected:
    Theme(vector<GraphicModel> * images);

    static void staticAnimation_moveLeft(int);
    static void staticAnimation_moveRight(int);
    static void staticAnimation_moveUp(int);
    static void staticAnimation_moveDown(int);
    static void staticAnimation_zoomIn(int);
    static void staticAnimation_zoomOut(int);
    static Theme * instance;

    bool animationMove;
    bool animationZoom;
    vector<GraphicModel> * c_images;
};
```

Fig. 3
Implementação do Effect

Sendo que tanto o tema Coverflow, ou Slideflow herdam esta classe.

Para além desta classe, ainda temos um controlador de temas que permite manusear todos os temas introduzidos no sistemas, tendo como um vector de `Theme`, tal como acontecia com a classe `Effects`.

Criando assim abstração ao utilizador da manipulação do vector, precisando apenas de fazer `next()` no objecto `ThemeController`.

Todos os tipos de visualização criados são abordados no capítulo de "Aspectos Importantes do Trabalho" de forma detalhada incluindo uma amostra das suas funcionalidades.

II. Aspectos Importantes do Trabalho

A. Filtros/Efeitos

Foram criados vários filtros para que se permitisse modificar qualquer imagem em tempo real com apenas um clique, sendo assim criado vários filtros, tais como: Sepia, Lomo, Tons de Cinza, Pencil Sketch, entre outros.
Considere-se para todos os filtros a seguir referidos, a seguinte imagem original:

Fig. 4
Original

A.1 Sepia

Este filtro consiste em aplicar em cada canal de um pixel um factor que é resultado do conjunto dos 3 canais, partindo disto, pode-se já concluir que não é aplicável a imagens com apenas 1 canal.

Sendo que os factores [1] aplicados a cada canal são os seguintes:

$$NovoR = 0.189R + 0.769G + 0.393B$$
$$NovoG = 0.168R + 0.686G + 0.349B$$
$$NovoB = 0.131R + 0.534G + 0.272B$$

Sendo que o resultado final depois de aplicar o filtro de Sepia é o seguinte:

Fig. 5
Sepia

Para realizar este efeito em OpenCV, é preciso ler todos os pixeis e em cada pixel substituir cada canal pelo resultado de cada equação, isto pode ser feito de forma mais simples utilizando a função transform de OpenCV que permite aplicar um kernel a matrix de uma imagem.

A implementação pode ser encontrada em `effects/types/Sepia.hpp`

A.2 Lomo

O efeito de Lomo consiste no mesmo que o Sepia mas com a diferença que o resultado do canal vermelho vai para o azul e o resultado do canal azul vai para o canal vermelho, assumindo-se então as seguintes equações.

$$NovoR = 0.131R + 0.534G + 0.272B$$
$$NovoG = 0.168R + 0.686G + 0.349B$$
$$NovoB = 0.189R + 0.769G + 0.393B$$

Sendo que o resultado do Sepia e do Lomo são bastante parecidos.

A.3 Tons de cinza

Este filtro consiste em converter uma imagem em que cada pixel corresponde a um conjunto de 3 canais, para apenas um canal.

Sendo que em OpenCV isso se faz de uma forma bastante simples.

```
cvtColor(in, image_out, CV_BGR2GRAY);
```

A.4 Troca de Canais

A troca de canais consiste em percorrer todos os pixeis 1 a 1 e trocar um canal por outro, por exemplo, o vermelho pelo azul e o azul pelo vermelho.

Na aplicação foi implementado este efeito mas em que a troca efectuada seria o canal vermelho pelo verde e vice-versa, sendo que se uma imagem for completamenta vermelha, depois de aplicado este efeito, vai passar a ser toda verde.

A.5 Iluminação

Para aumentar ou diminuir a iluminação de uma imagem a partir dos canais de cores vermelho, verde e azul é díficil, sendo então que se converte primeiro a imagem para o formato YCbCr.

Depois da imagem estar convertida, altera-se directamente o canal Y (canal respectivo da luminosidade), ou seja, multiplica-se por um determinado factor, um Y superior representa uma imagem mais clara e um Y inferior representa uma imagem mais escura.

Voltando depois a converter a imagem em RGB.

A.6 Saturação

Para aumentar ou diminuir a saturação de uma imagem a partir dos canais de cores vermelho, verde e azul é díficil, sendo então que se converte primeiro a imagem para o formato YCbCr, tal como se fazia para

iluminação, mas agora em vez de manipular o canal Y, vai se manipular os canais Cb e Cr que representa as diferenças de azul e as diferenças de vermelho, respectivamente.

A.7 Vignette

O efeito Vignette consiste em fazer desvanecer as bordas de uma imagem, tornando-as mais escuras.

Sendo assim pode-se obter este efeito de um forma muito simples, percorrer todos os pixeis da imagem em que para cada pixel calcula-se a distância ao centro, quanto maior a distância, mais escuro a imagem tem de ser, obtendo assim o efeito de Vignette.

Para escurecer a imagem, foi multiplicado pelo inverso da distância cada canal de cores, tendo em conta que todos os canais a 0, corresponde ao preto, quando menor for o valor de cada canal, mais escuro fica a imagem.

Tendo em conta isto, foi obtido o seguinte resultado:

Fig. 6
Vignette

A.8 Pencil Sketch

O efeito pencil sketch tenta imitar o efeito de ter um artista a pintar uma imagem usando um pincel (trabalho manual).

Para obter este efeito foi aplicado o seguinte procedimento:

1. Sobel sobre X
2. Sobel sobre Y
3. Potência de 2 sobre o resultado de cada Sobel e somar
4. Conversão em níveis de cinzento
5. Aplicar o negativo a imagem de forma a tornar o resultado real

Aplicar o sobel serve para criar os efeitos dos riscos do pincel, fazem a potência e somar serve para juntar apenas os resultados de ambos os Sobels, sendo que depois era preciso converter para os tons habituais que se trabalha normalmente, neste caso, para tons de cinzento, só que ainda não se tem o efeito pretendido pois a imagem encontra-se invertida, sendo assim, aplica-se o negativo da imagem para obter um resultado realístico.

Sendo que o resultado final deste processo é o seguinte:

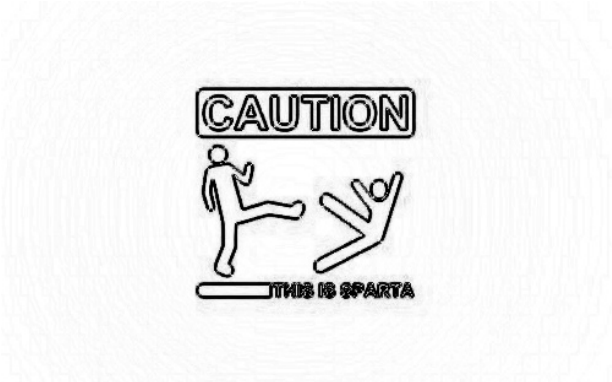

Fig. 7
Pencil Sketch

A.9 Cartoon

Este efeito tem como objectivo obter uma imagem realistica, por exemplo de uma paisagem, e torna-la num cartoon (algo perto de desenho animado).

Para obter este efeito foi efecutado em cada pixel uma média dos seus valores a volta sobrepondo o valor actual [2], fazendo assim com que pequenas alterações de valores nos canais de cor desapareceçam, fazendo com que a imagem contenha apenas variações de cores mais bruscas, tal como são os desenhos animados.

Depois ainda se aplicou Canny para denotar onde essas diferenças apareciam, de forma a dar o contorno ao cartoon.

A.10 Canny

É apenas aplicação directa da função de OpenCV para detecção de arestas, permitindo o utilizador alterar valores.

III. Modos de visualização

Foram criados dois modos de visualização distintos, Coverflow e Slideflow.

Ambos os modos de visualização tem a selecção de filtros/efeitos, abertura da câmera, ampliação total da imagem e os botões de interacção com o utilizador implementados da mesma forma, sendo que não é possível alterar sem mexer na estrutura.

Tudo o resto é possível alterar criando apenas uma classe que herde a class `Theme`.
Os efeitos de ampliação total e representação da câmera aparecem no capítulo resultado final, tudo o resto apenas é possível executando a aplicação.

A. *Coverflow*

Coverflow consiste em gerar as imagens todas numa linha em que o utilizador tem a possibilidade de andar apenas da esquerda para a direita ou da direita para a esquerda.

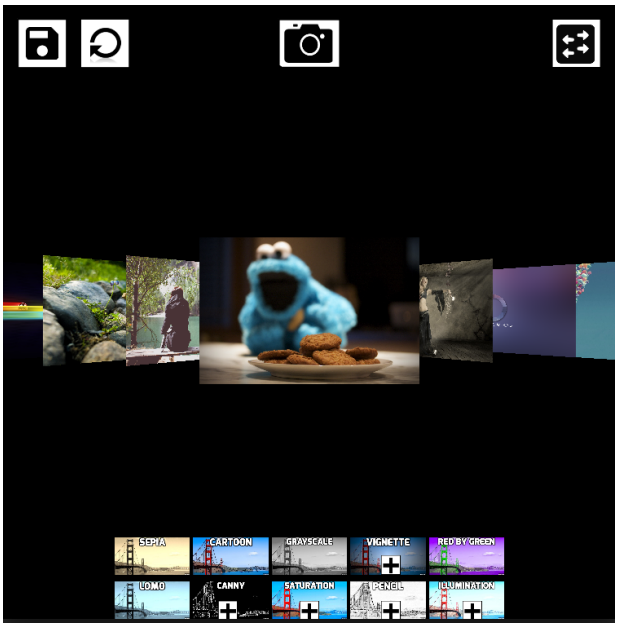

Fig. 8
Coverflow

B. *Slideshow*

Slideflow consiste em gerar as imagens todas numa matrix de n columas por m linhas, neste caso, foi implementado com 3 linhas em que o número de colunas é determinado consoante o número de imagens.

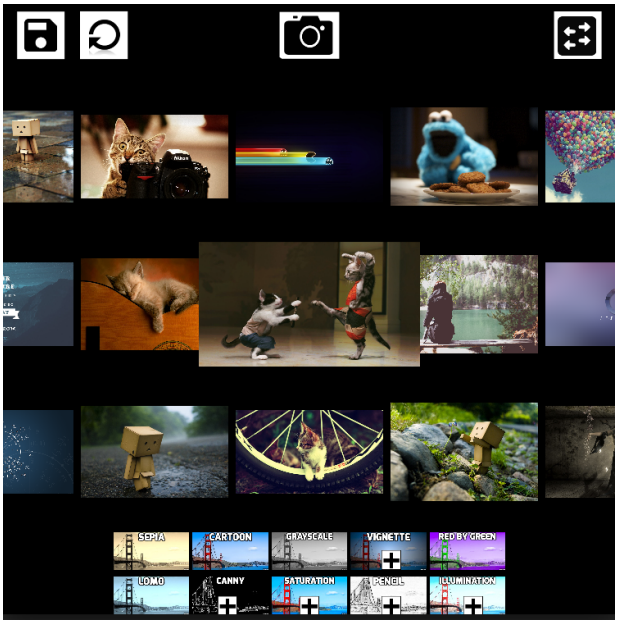

Fig. 9
Slideflow

IV. Resultado Final

O resultado final obtido foi o pretendido, sendo que é possível aplicar filtro sobre as imagens, utilizar a webcam aplicando filtros sobre a mesma, guardar a imagem seleccionada.

Um exemplo da imagem da webcam com o filtro aplicado em que a frente da webcam tem uma mão a segurar num telemóvel, mas com o filtro de Pencil Sketch.

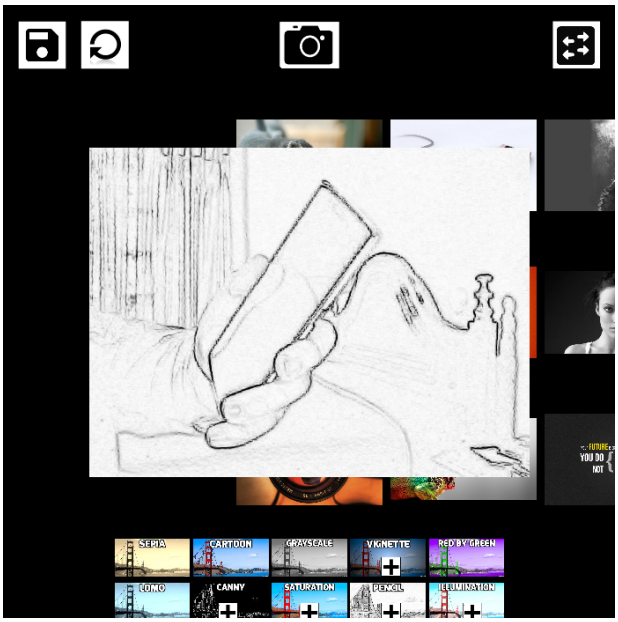

Fig. 10
Camera com Pencil Sketch

Também se pode verificar que a função de zoom utilizado o scroll pode dar jeito para ver uma imagem com melhor resolução.

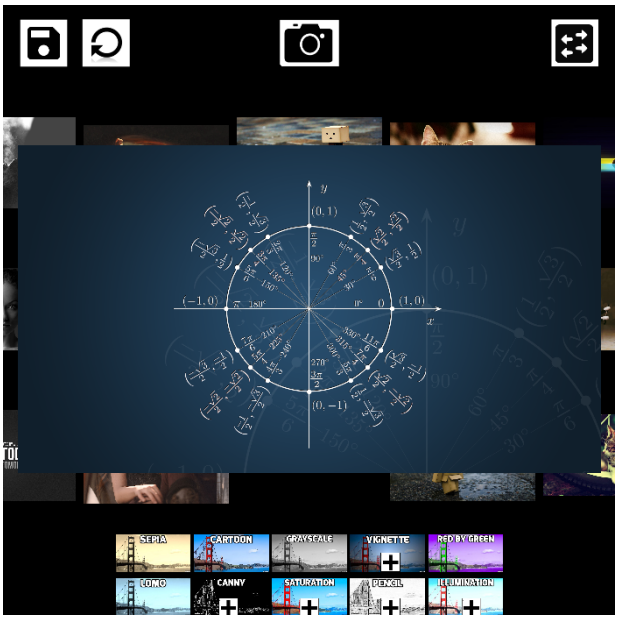

Fig. 11
Zoom in

Sendo este o resultado final obtido na aplicação.

V. Compilação e Execução

Código fonte disponível em:
`https://github.com/dbtds/slideshow3d-cvgl`
Podendo ser obtido através do comando: `git clone https://github.com/dbtds/slideshow3d-cvgl.git`
Este projecto foi desenvolvido unicamente em ambiente Linux, sendo assim, não foi criado qualquer ficheiro executável para ambiente Windows, apesar que também é possível faze-lo alterando apenas a forma de compilação.

Note-se que todas as próximas indicações foram apenas realizadas em ambiente Linux no Ubuntu 14.04 64 bits.

Software essencial para compilar e correr o programa:

1. build-essential
2. qt4-qmake
3. libglew-dev
4. freeglut3-dev
5. libboost-system-dev
6. libboost-filesystem-dev
7. opencv (pode-se ter que compilar esta biblioteca a partir do source)

Para obter cada dependência basta executar:
`sudo apt-get install dependency_name`
Após a obtenção de todas estas dependências, basta executar o Makefile na pasta principal (root).
Executando em bash `/ $ make`
Para executar o programa, basta ir a pasta gerada bin, e correr o executável com o nome chess. Executando em bash `/bin $ ./slideshow3d example_images/`
Já existe um executável pré-gerado apenas para 64 bits dentro da pasta bin com o nome chess_bin64, executando da mesma forma: Em bash `/bin $ ./slideshow3d_bin64 example_images/`

Bibliografia

[1] Paul Varcholik, "Real-time 3d rendering with directx and hlsl", `https://books.google.pt/books?id=GY-AAwAAQBAJ`.

[2] Belisarius, "Image segmentation", `http://stackoverflow.com/questions/4831813/image-segmentation-using-mean-shift-explained`.