

**Tópicos Especiais I: Modelagem, Rendering e Iluminação Baseados em Imagens
2003.1**

Manuel M. Oliveira

**Trabalho de Implementação Nº 1
3D Image Warping**

Passed in: 6/5/01

Total de Pontos do Trabalho: 100

Data para Entrega: 16/05/03 às 10:30

Objetivo

O objetivo deste trabalho de implementação é fornecer aos estudantes uma sólida compreensão sobre a técnica de 3D Image Warping. Ao concluir este trabalho, o estudante deverá:

- a) Estar familiarizado com o conceito e a manipulação de imagens com profundidade (*depth images*);
- b) Ter implementado o algoritmo para realização de 3D Image Warping direto (*forward 3D Image Warping*);
- c) Ter explorado otimizações na implementação do algoritmo de 3D Image Warping, tirando proveito de computações incrementais para sub-expressões sempre que possível;
- d) Ter implementado o algoritmo *Occlusion-Compatible Order* para determinação de visibilidade.

Descrição do Trabalho

- 1) Leia o Capítulo 3 (*A Warping Equation*, páginas 30 a 60) da tese de doutorado de Leonard McMillan Jr., intitulada *An Image-Based Approach to Three-Dimensional Computer Graphics* (<http://www.cs.unc.edu/~ibr/pubs/mcmillan-diss/mcmillan-diss.pdf>) e estude as notas de aula do dia 2/05/2003 (http://www.inf.ufrgs.br/~oliveira/Cursos/INF01179/2003_1/3D_image_warping.pdf).
- 2) Implemente um programa para realizar *forward 3D Image Warping*, seguindo as instruções presentes nas notas de aula. O seu programa deverá suportar pelo menos as seguintes funções:
 - a) Ler um arquivo no formato *ibr_tiff* contendo uma imagem com profundidade cujo nome é especificado pelo usuário na linha de comando **(10 pontos)**;
 - b) Implementar o algoritmo de *forward 3D Image Warping*, suportando translações e rotações da câmera alvo. Tanto as translações como as rotações devem ser feitas em relação ao sistema de coordenadas da câmera alvo **(50 pontos)**;
 - c) Implementar otimizações tirando proveito de computações incrementais para sub-expressões sempre que possível, conforme discutido em sala e explicado nas notas de aula **(15 pontos)**;
 - d) Implementar o algoritmo *Occlusion-Compatible Order* para determinação de visibilidade. **(25 pontos)**;

Dicas para Realização do Trabalho

1) Sobre as Imagens com Profundidade armazenadas no formato ibrtiff.

Imagens gravado neste formato contém dados sobre cor e disparidade associados a cada pixel, além dos parâmetros da câmera (centro de projeção - C_s e vetores a_s , b_s e c_s). Estas imagens podem ser lidas utilizando uma API disponível na página que contém o link para esta descrição. Instruções de como utilizar a API podem ser encontradas na documentação contida nos arquivos .h que a acompanham. Tudo o que você tem a fazer é ligar o seu programa com a biblioteca provida para a API e fazer as chamadas apropriadas.

A rotina *read* abaixo faz a leitura de um arquivo contendo uma imagem com profundidade e recupera os dados sobre cor (retornados no ponteiro *image*), disparidade (retornados no ponteiro *disparity*) e camera (retornados no ponteiro *camera*), referentes à imagem fonte cujo nome é passado para a função.

Como a função *read* sugere, os conteúdos apontados pelas variáveis *image*, *disparity*, *buff* e *dispMap* devem ser entendidos como matrizes com numero de linhas e colunas iguais a *width* e *height*, respectivamente. O conteúdo de cada uma destas matrizes é armazenadas como um vetor e, portanto, utiliza-se um endereço linear (todos os elementos da primeira linha seguidos por todos os elementos da segunda linha, ..., seguidos por todos os elementos da última linha).

Observação: Os conteúdos tanto de cor quanto de disparidade têm a ordem de suas linhas invertidas para tirar proveito do comando de OpenGL *glDrawPixels* (no método *display* – veja item 5 a seguir “Exibindo o Resultado da Operação de Warping”).

```
//
// read a reference image using the UNC-CH IBR TIFF format
//
void Image::read(char *FileName)
{
    int i, j;
    int width, height, image_type;
    int pix_counter = 0; // to compute the center of mass of the scene
    char *description;
    unsigned char *image;
    float *disparity;
    float *normal;
    float *camera;
    float r, disp;

    if (readIBRTIFF(FileName, &image, &disparity, &camera, &normal, &width, &height &description,
        DONT_LOAD_NORMALS, &image_type) == 0) {
        rows = height;
        cols = width;
    }
    //
    // allocate space to the frame buffer and to the disparity map
    //
    buff = new rgb[height*width];
    dispMap = new float[height*width];
    //
    // Check if the system run out of memory
    //
    if (buff == NULL || dispMap == NULL) {
        cerr << "Not enough memory! " << endl;
```

```

        exit(1);
    }
//
//  copy the image data to the image buffer. The rows
//  have been intentionally reversed to take advantage
//  of the GL function glDrawPixels (in method display)
//  that fills the window from bottom to top
//  using a single block transfer from memory
//
    for (i=0; i<height; i++)
        for (j=0; j<width; j++) {
            buff[(rows-i-1)*width + j].r = image[i*width*4 + j*4 + 0];
            buff[(rows-i-1)*width + j].g = image[i*width*4 + j*4 + 1];
            buff[(rows-i-1)*width + j].b = image[i*width*4 + j*4 + 2];
        }
//
//  copy the disparity data to the disparity map buffer
//
    for (i=0; i<height; i++)
        for (j=0; j<width; j++)
            dispMap[(rows-i-1)*width + j] = disparity[i*width + j];
//
//  read the source camera parameters into the source image matrix M
//
//  Center of projection
//
    M.C.x = camera[0];
    M.C.y = camera[1];
    M.C.z = camera[2];
//
//  Vectors a, b and c, respectively
//
    M.a.x = camera[3];
    M.a.y = camera[4];
    M.a.z = camera[5];
    M.b.x = camera[6];
    M.b.y = camera[7];
    M.b.z = camera[8];
    M.c.x = camera[9];
    M.c.y = camera[10];
    M.c.z = camera[11];
//
//  free temporary memory
//
    delete []image;
    delete []disparity;
    delete []camera;
    if (normal != NULL)
        delete []normal;
    if (description != NULL)
        delete []description;
}
}

```

2) Implementando o algoritmo *forward 3D Image Warping*

Em poucas palavras, *forward 3D image warping* consiste em transferir a cor associada a cada pixel de uma imagem fonte para o plano de imagem da camera destino. As coordenadas dos pixels na imagem destino são obtidas utilizando as expressões $u = r/t$ e $v = s/t$, conforme descrito nas notas de aula. Antes de calcular u e v você precisará obter os vários termos w_{ij} , para os quais você precisará dispor das operações de produto escalar e produto vetorial.

Dada uma imagem fonte com dimensões $W_s \times H_s$, você deverá criar um “buffer” com as mesmas dimensões para conter o resultado da operação de warping (imagem destino). Note-se que, como a imagem original não muda e é necessária para a operação de warping, seu conteúdo não deve ser destruído.

Observações

- A cada novo quadro, o conteúdo da imagem destino deve ser inicializado para evitar que resquícios de imagens anteriores interfiram com a imagem atual. Uma forma eficiente de inicializar um vetor com um dado valor é o uso do comando *memset*. Neste caso, teríamos (`buff_destino, '0', 3*rows*cols`). Lembre-se que cada elemento da matrix corresponde a 3 bytes, armazenando os valores R, G e B, respectivamente.
- Os valores obtidos para as coordenadas u e v são valores em ponto flutuante e devem ser ajustados para o valor inteiro mais próximo;
- Deve-se checar se os valores de coordenadas obtidos encontram-se dentro dos limites (dimensões) da imagem destino. Caso contrário, o pixel correspondente à tal transformação é ignorado;
- Caso cada pixel da imagem fonte seja mapeado em um único pixel na imagem destino, a imagem destino apresentará “rachaduras” quando a camera destino encontrar-se mais próxima da superfície representada que a camera fonte. Para minimizar estes problemas, o rendering é feito utilizando uma técnica conhecida como “splatting”. Assim, dadas as coordenadas (u_t, v_t) do pixel na imagem destino, a mesma cor é replicada em uma vizinhança (2×2) ou (3×3) em torno de (u_t, v_t) .

3) Implementando Otimizações

Para implementar as otimizações, basta analisar as sub-expressões que entram na composição dos termos r , s e t e tentar minimizar o número e o custo das operações aritméticas envolvidas. Por exemplo, operações de multiplicação devem ser substituídas por operações de adição sempre que possível.

4) Implementação do Algoritmo de visibilidade

A implementação do algoritmo de visibilidade consiste em calcular o epipolo na imagem fonte, de acordo com o descrito nas notas de aula. Para tanto, você precisará obter a matrix inversa da camera fonte (M_s^{-1}). Uma vez obtidas as coordenadas do epipolo (após a divisão perspectiva), verifique em quantas regiões o plano de imagem fonte deverá ser subdividido (de uma a quatro regiões).

Caso o valor de e_z seja negativo, cada uma das regiões deve ter seus pixels transformados partindo-se **do epipolo para as bordas da imagem fonte**. Caso contrário, as transformações devem ser aplicadas *iniciando-se nas bordas e seguindo em direção ao epipolo*.

5) Exibindo o Resultado da Operação de Warping

Apesar da implementação do algoritmo de 3D Image Warping não necessitar das operações de transformações geométricas e projeções disponíveis em OpenGL, o uso de algumas funções

providas por OpenGL e GLUT podem simplificar a tarefa de exibição dos resultados. Por exemplo, a função abaixo é suficiente para exibir o conteúdo da imagem destino armazenado na variável *< buff imagem destino >*. Detalhes sobre os comandos abaixo podem ser encontrados no Livro Vermelho de OpenGL.

```
void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glDrawPixels(cols, rows, GL_RGB, GL_UNSIGNED_BYTE, <buff imagem destino>);
    glutSwapBuffers();
}
```

6) Criando uma Janela para Exibir os Resultados

O fragmento de código a seguir mostra como criar uma janela usando GLUT para suportar imagens RGB com *double buffering*. Também ilustra como obter o nome do arquivo como argumento passado para o programa na linha de comando.

```
int main(int argc, char *argv[])
{
    if (argc < 2) {
        cerr << "Usage: warp <reference image file> " << endl << endl;
        exit(1);
    }
    //
    // Read image from a file name specified as the 1st argument to the program
    //
    W.read(argv[1]);
    //
    // initialize GLut display mode: an RGB frame buffer with double buffering
    //
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
    glutInitWindowPosition(0, 0);
    glutInitWindowSize(W.cols, W.rows);
    glutCreateWindow("3D Image Warping");
    glutDisplayFunc(<nome da sua função para display>);
    glutReshapeFunc(<nome da sua função para display>);

    glutMotionFunc(<nome da sua função para mouse motion>);
    glutMouseFunc(<nome da sua função para mouse>);
    glutKeyboardFunc(<nome da sua função para tratamento de teclado>);
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

    glutMainLoop();
    return 0;
}
```

7) Implementando Translações e Rotações da Câmera Destino (em relação ao sistema de coordenadas da câmera destino)

Translações da câmera destino são obtidas simplesmente transladando-se seu centro de projeção (C_t). Os demais parâmetros da câmera permanecem inalterados. Para transladar ao longo do eixo X, garanta que o movimento seja feito na direção especificada pelo vetor a_t . Para

translações ao longo do eixo Y, utilize a direção do vetor b_t e para o eixo Z, a direção definida pelo vetor perpendicular a a_t e b_t .

Rotações da câmera destino são obtidas mantendo-se C_t fixo e rotacionando-se os vetores a_t , b_t e c_t .

Bom Trabalho e Boa Sorte.