Visual Computing Lab 5

CHAKKA Karthik Subramanyam

CHATTERJEE Jit

TP5 - Image Formation

The objective of this practical work is to study image formation by projecting a 3D point cloud to an image plane using the pinhole camera model.

1) Pinhole camera and perspective projection

The simplest imaging device which captures the geometry of the perspective projection. Rays of light enters through an infinitely small pin hole aperture into the camera. The intersection of the light rays with the image plane form the image of the object. This kind of mapping from 3D to 2D is called perspective projection. Perspective projection with the origin in the image plane:

$$\begin{pmatrix} X_{p'} \\ Y_{p'} \\ 0 \\ 1 + Z_{p'}/f \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1/f & 1 \end{pmatrix} \begin{pmatrix} X_{p'} \\ Y_{p'} \\ Z_{p'} \\ 1 \end{pmatrix}$$

and thus:

$$\begin{pmatrix} X_{img} \\ Y_{img} \end{pmatrix} = \begin{pmatrix} X_{p'}/(1 + Z_{p'}/f) \\ Y_{p'}/(1 + Z_{p'}/f) \end{pmatrix}$$

Goal:

```
To implement the pinhole projection to project every point in the object
 to image plane.
```

Procedure:

Code Snippet:

```
void doPinholeProjection(const char* ipFilNam){
    int N;
    point3d *pt = readOff(ipFilNam, &N);
```

```
    size_t row = 500, col = 500, dim = 3;
    mat *img = allocMatMem(row, col, dim);
    from3dtoPinholeProj(pt, (unsigned)N, img, row, col);
    char projOpFilNam[MAX_FIL_NAM_SIZ];
    mkOpNameWExt(projOpFilNam, "PinHole", (char)N,
 ipFilNam, ".ppm", sizeof (projOpFilNam));
    imgMatToPgm(projOpFilNam, img, row, col, 3, (unsigned)MAX_PIX_VAL,
 eP6);
}

void from3dtoPinholeProj(point3d *pt, size_t N, mat *img, size_t row,
 size_t col){
    RigTrans *r = rigTransIn3d(pt, N, 0, 0, 0, 0, 0, 0);
    RigTrans *rp = rigTransToPerspec(r, N, ORIGINAL_IMG_F);
    free(r);
    PerspecImg *p = perspecToImage(rp, N);
    free(rp);
    PerspecImg *s = scalePerspecImage(p, N, row, col);
    free(p);
    prespecToImg(s, N, img, row, col);
    free(s);
}
```

```
RigTrans *occludeRigTrans(RigTrans *r, size_t N, size_t *nN, double del)
{
    RigTrans *p = malloc(N * sizeof (RigTrans));
    memset(p, 0, N * sizeof (RigTrans));
    size_t cnt = 0;
    for (size_t n = 0; n < N; ++n) {
        size_t c = 0;
        for (; c < N; ++c) {
            if( ( r[n].x > (r[c].x - del) ) && ( r[n].x < r[c].x + del )
 && ( r[n].y > (r[c].y - del) ) && ( r[n].y < r[c].y + del ) ){
                if(r[c].z > r[n].z){
                    break;
                }
            }
        }
        if(c == N){
            p[cnt].x = r[n].x;
            p[cnt].y = r[n].y;
            p[cnt].z = r[n].z;
            p[cnt].d = r[n].d;
            p[cnt].r = r[n].r;
```

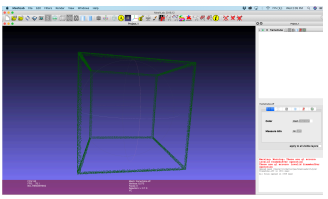
```
            p[cnt].g = r[n].g;
            p[cnt].b = r[n].b;
            ++cnt;
        }
    }
    *nN = cnt;
    RigTrans *o = malloc(cnt * sizeof (RigTrans));
    for (size_t n = 0; n < cnt; ++n) {
        o[n].x = p[n].x;
        o[n].y = p[n].y;
        o[n].z = p[n].z;
        o[n].d = p[n].d;
        o[n].r = p[n].r;
        o[n].g = p[n].g;
        o[n].b = p[n].b;
    }
    free(p);
    return o;
}
```

```
void prespecToImg(PerspecImg *pt, size_t N, mat *img, size_t row, size_t
 col){
    size_t dim = 3;
    mat val[3] = {0, 0, 0};
    memsetMatrix(img, val, row, col, dim);
    for (size_t n = 0; n < N; ++n) {
        if((pt[n].x < 0) || (pt[n].x > row)){
            continue;
        }
        if((pt[n].y < 0) || (pt[n].y > col)){
            continue;
        }
        size_t rowNo = (unsigned)pt[n].x;
        size_t colNo = (unsigned)pt[n].y;
        img[ ( rowNo * col * dim ) + ( colNo * dim ) + 0 ] = pt[n].r;
        img[ ( rowNo * col * dim ) + ( colNo * dim ) + 1 ] = pt[n].g;
        img[ ( rowNo * col * dim ) + ( colNo * dim ) + 2 ] = pt[n].b;
    }
}

PerspecImg *scalePerspecImage(PerspecImg *p, size_t N, size_t row,
 size_t col){
    PerspecImg *s = malloc(N * sizeof (PerspecImg));
    for (size_t n = 0; n < N; ++n) {
        s[n].x = (p[n].x * row) + row / 2;
```

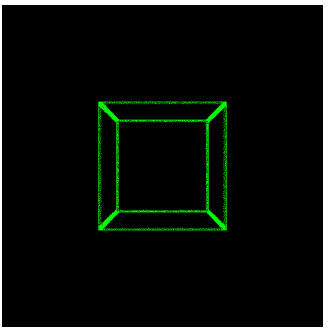
```
        s[n].y = (p[n].y * col) + col / 2;
        s[n].r = p[n].r;
        s[n].g = p[n].g;
        s[n].b = p[n].b;
    }
    return s;
}
```

Input Image: cubeFrame.off



Input Image in meshlab software (Fig 1.1)

Output Image:



Perspective projection of Fig 1.1 (Fig 1.2)

Results:

Distances and angles are not preserved. Parallel lines do not in general project to parallel lines (unless they are parallel to the image plane).

---

2) uv projection

A UV layout is a visual representation of a 3D model flattened onto a 2D plane. Each point on the 2D plane is called a UV and represents a vertex on the 3D object. In this way, all areas within the boundary of the UV layout correspond to a specific spot on the model.

Procedure:

Now we need to fit a rectangular image to the image plane by choosing the image origin and image resolution α_u and α_v with the unit of meter/pixel so that we can get pixel coordinates (u, v) of the projected point:

$$\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} X_{img}/\alpha_u + u_0 \\ Y_{img}/\alpha_v + v_0 \end{pmatrix}$$

where α_u and α_v are scaling factor, and α_u/α_v is called aspect ratio, which is 1 for pinhole camera.

Code Snippet:

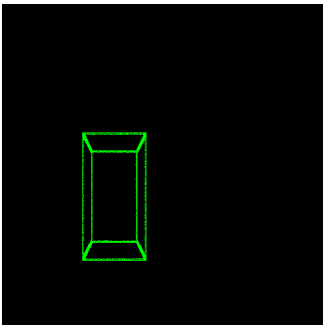
```
void doUVProjection(const char* ipFilNam, size_t resolX, size_t resolY,
 size_t x0, size_t y0){
    int N;
    point3d *pt = readOff(ipFilNam, &N);
    size_t row = 500, col = 500, dim = 3;
    mat *img = allocMatMem(row, col, dim);
    from3dtoUVProj(pt, (unsigned)N, resolX, resolY, x0, y0, img, row,
 col);
    char projOpFilNam[MAX_FIL_NAM_SIZ];
    mkOpNameWExt(projOpFilNam, "UV", (char)N, ipFilNam, ".ppm", sizeof
 (projOpFilNam));
    imgMatToPgm(projOpFilNam, img, row, col, 3, (unsigned)MAX_PIX_VAL,
 eP6);
}

void from3dtoUVProj(point3d *pt, size_t N, size_t resolX, size_t resolY,
 size_t x0, size_t y0, mat *img, size_t row, size_t col){
    RigTrans *r = rigTransIn3d(pt, N, 0, 0, 0, 0, 0, 0);
    RigTrans *rp = rigTransToPerspec(r, N, ORIGINAL_IMG_F);
    free(r);
```

```
    PerspecImg *p = perspecToImage(rp, N);
    free(rp);
    PerspecImg *s = scalePerspecImage(p, N, row, col);
    free(p);
    PerspecImg *uv = perspecToUV(s, N, resolX, resolY, x0, y0);
    free(s);
    prespecToImg(uv, N, img, row, col);
    free(uv);
}

PerspecImg *perspecToUV(PerspecImg *pt, size_t N, size_t resolX, size_t
 resolY, size_t x0, size_t y0){
    PerspecImg *uv = malloc(N * sizeof (PerspecImg));
    for (size_t n = 0; n < N; ++n) {
        uv[n].x = (pt[n].x / resolX) + x0;
        uv[n].y = (pt[n].y / resolY) + y0;
        uv[n].r = pt[n].r;
        uv[n].g = pt[n].g;
        uv[n].b = pt[n].b;
    }
    return uv;
}
```

Output Image:



uv projection of Fig 1.1 (Fig 1.3)

3) Rigid transformation

A transformation is said to be rigid if it preserves relative distances—that is to say, if P and Q are transformed to P* and Q* then the distance from P* to Q* is the same as that from P to Q.

Procedure:

Transform of a point in 3D-space:

$$\begin{pmatrix} X_{p'} \\ Y_{p'} \\ Z_{p'} \\ 1 \end{pmatrix} = \begin{pmatrix} \boldsymbol{R} & \boldsymbol{t} \\ \boldsymbol{0} & 1 \end{pmatrix} \begin{pmatrix} X_p \\ Y_p \\ Z_p \\ 1 \end{pmatrix}$$

where R is 3 × 3 rotational matrix, t a 3 × 1 translation vector, and p' the new point position after transform. In practice 3D rotation can be represented in many ways (such as Euler angle, quaternion, and axis-angle) and the calculation of the rotation matrix out of those representations varies.

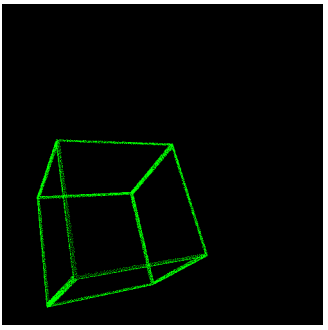
Code Snippet: Rigid Transformation and PinHole

```
void doTransformNPinholeProjection(const char* ipFilNam, double
 f, double gama, double beta, double alpha, double T_x, double
 T_y, double T_z){
    int N;
    point3d *pt = readOff(ipFilNam, &N);
    size_t row = 500, col = 500, dim = 3;
    mat *img = allocMatMem(row, col, dim);
    from3dTransformToPinholeProj(pt, (unsigned)N, f, gama, beta, alpha,
 T_x, T_y, T_z, img, row, col);
    char projOpFilNam[MAX_FIL_NAM_SIZ];
    mkOpNameWExt(projOpFilNam, "TPinHole", (char)N,
 ipFilNam, ".ppm", sizeof (projOpFilNam));
    imgMatToPgm(projOpFilNam, img, row, col, 3, (unsigned)MAX_PIX_VAL,
 ePG);
}

void from3dTransformToPinholeProj(point3d *pt, size_t N, double
 f, double gama, double beta, double alpha, double T_x, double
 T_y, double T_z, mat *img, size_t row, size_t col){
    RigTrans *r = rigTransIn3d(pt, N, gama, beta, alpha, T_x, T_y, T_z);
```

```
    RigTrans *rp = rigTransToPerspec(r, N, f);
    free(r);
    PerspecImg *p = perspecToImage(rp, N);
    free(rp);
    PerspecImg *s = scalePerspecImage(p, N, row, col);
    free(p);
    prespecToImg(s, N, img, row, col);
    free(s);
}
```

Output Image:



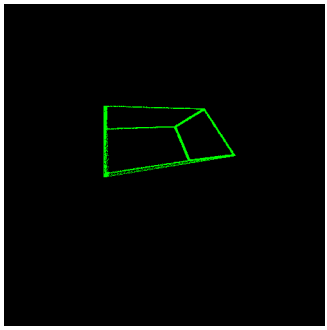
```
        Rigid Transformation and PinHole projection of Fig 1.1 (Fig 1.4)
```

Code Snippet: Rigid Transformation and uv

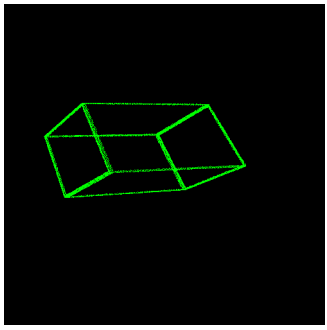
```
void doTransformNUVProjection(const char* ipFilNam, double f, double
 gama, double beta, double alpha, double T_x, double T_y, double T_z,
 size_t resolX, size_t resolY, size_t x0, size_t y0){
    int N;
    point3d *pt = readOff(ipFilNam, &N);
    size_t row = 500, col = 500, dim = 3;
    mat *img = allocMatMem(row, col, dim);
    from3dTransformtoUVProj(pt, (unsigned)N, f, gama, beta, alpha, T_x,
 T_y, T_z, resolX, resolY, x0, y0, img, row, col);
    char projOpFilNam[MAX_FIL_NAM_SIZ];
    mkOpNameWExt(projOpFilNam, "TUV", (char)N, ipFilNam, ".ppm", sizeof
 (projOpFilNam));
    imgMatToPgm(projOpFilNam, img, row, col, 3, (unsigned)MAX_PIX_VAL,
 eP6);
}
```

```
void from3dTransformtoUVProj(point3d *pt, size_t N, double f, double
 gama, double beta, double alpha, double T_x, double T_y, double T_z,
 size_t resolX, size_t resolY, size_t x0, size_t y0, mat *img, size_t
 row, size_t col){
    RigTrans *r = rigTransIn3d(pt, N, gama, beta, alpha, T_x, T_y, T_z);
    RigTrans *rp = rigTransToPerspec(r, N, f);
    free(r);
    PerspecImg *p = perspecToImage(rp, N);
    free(rp);
    PerspecImg *s = scalePerspecImage(p, N, row, col);
    free(p);
    PerspecImg *uv = perspecToUV(s, N, resolX, resolY, x0, y0);
    free(s);
    prespecToImg(uv, N, img, row, col);
    free(uv);
}
```

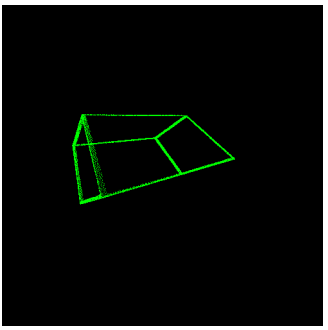
Output Image: Rigid Transformation and uv projection with different change in parameters



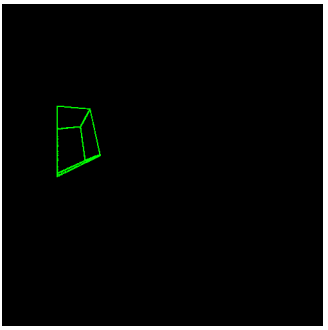
Rigid Transformation and uv projection of Fig 1.1 (Fig 1.5)



```
Rigid Transformation and uv projection with change in focal length of
 Fig 1.1 (Fig 1.6)
```



```
Rigid Transformation and uv projection with modification in object
 transformation of Fig 1.1 (Fig 1.7)
```



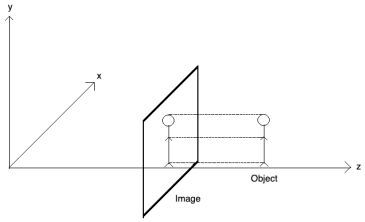
```
Rigid Transformation and uv projection with change in resolution of Fig
 1.1 (Fig 1.8)
```

Results:

Increasing the focal length of the image will increase the size of the image formed and vice versa. It also effects the perspective effect. Modification of object tranformation will result in rotated image. Changing the resolution of the image will result in zoomed in or zoomed out image. Changing the position results in shift.

4) Orthogonal projection

It is the projection of a 3D object onto a plane by a set of parallel rays orthogonal to the image plane. It is a form of parallel projection, in which all the projection lines are orthogonal to the projection plane, resulting in every plane of the scene appearing in affine transformation on the viewing surface. It is the limit of perspective projection as f − > ∞ (i.e., f /Z − > 1).



x=X, y = Y, (drop Z)

$$\begin{bmatrix} x_h \\ y_h \\ z_h \\ w \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

using w = 1,

$$x = \frac{x_h}{w} = X \quad y = \frac{y_h}{w} = Y$$

Procedure:

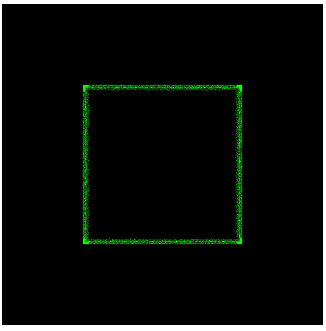
Code Snippet: Orthogonal projection

```
void doOrthProjection(const char* ipFilNam){
    int N;
    point3d *pt = readOff(ipFilNam, &N);
    size_t row = 500, col = 500, dim = 3;
    mat *img = allocMatMem(row, col, dim);
    from3dtoOrthProj(pt, (unsigned)N, img, row, col);
    char projOpFilNam[MAX_FIL_NAM_SIZ];
```

```
    mkOpNameWExt(projOpFilNam, "Orth", (char)N, ipFilNam, ".ppm", sizeof
 (projOpFilNam));
    imgMatToPgm(projOpFilNam, img, row, col, 3, (unsigned)MAX_PIX_VAL,
 eP6);
}

void from3dtoOrthProj(point3d *pt, size_t N, mat *img, size_t row,
 size_t col){
    RigTrans *r = rigTransIn3d(pt, N, 0, 0, 0, 0, 0, 0);
    RigTrans *rp = rigTransToPerspec(r, N, 20);
    free(r);
    PerspecImg *p = perspecToImage(rp, N);
    free(rp);
    PerspecImg *s = scalePerspecImage(p, N, row, col);
    free(p);
    PerspecImg *uv = perspecToUV(s, N, 1, 1, 0, 0);
    free(s);
    prespecToImg(uv, N, img, row, col);
    free(uv);
}
```

Output Image:



Orthogonal projection of Fig 1.1 (Fig 1.6)

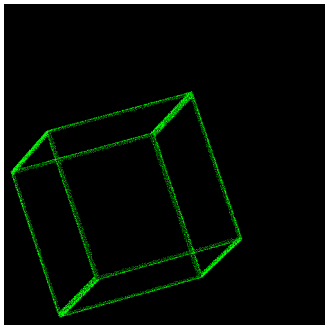
Code Snippet: Rigid Transformation and Orthogonal projection

```
void doTransormNOrthProjection(const char* ipFilNam, double gama, double
 beta, double alpha, double T_x, double T_y, double T_z){
    int N;
    point3d *pt = readOff(ipFilNam, &N);
    size_t row = 500, col = 500, dim = 3;
```

```
    mat *img = allocMatMem(row, col, dim);
    from3dTransformtoOrthProj(pt, (unsigned)N, gama, beta, alpha, T_x,
 T_y, T_z, img, row, col);
    char projOpFilNam[MAX_FIL_NAM_SIZ];
    mkOpNameWExt(projOpFilNam, "TOrth", (char)N,
 ipFilNam, ".ppm", sizeof (projOpFilNam));
    imgMatToPgm(projOpFilNam, img, row, col, 3, (unsigned)MAX_PIX_VAL,
 eP6);
}

void from3dTransformtoOrthProj(point3d *pt, size_t N, double
 gama, double beta, double alpha, double T_x, double T_y, double T_z,
 mat *img, size_t row, size_t col){
    RigTrans *r = rigTransIn3d(pt, N, gama, beta, alpha, T_x, T_y, T_z);
    RigTrans *rp = rigTransToPerspec(r, N, 20);
    free(r);
    PerspecImg *p = perspecToImage(rp, N);
    free(rp);
    PerspecImg *s = scalePerspecImage(p, N, row, col);
    free(p);
    PerspecImg *uv = perspecToUV(s, N, 1, 1, 0, 0);
    free(s);
    prespecToImg(uv, N, img, row, col);
    free(uv);
}
```

Output Image:



```
    Rigid Transformation and Orthogonal projection of Fig 1.1 (Fig 1.7)
```

Results:

Parallel lines project to parallel lines. Size does not change with distance from the camera.

---

5) Occlusion

Occlusion often occurs when two or more objects come too close and seemingly merge or combine with each other. Occlusion means that there is something we want to see in an image, but can't due to some property or some event. Occlusion is the one which blocks our view. So if the objects change its position with respect to the camera where some aspect of the object hides, computer will recognize different edges or pixels or any other features.

Procedure:

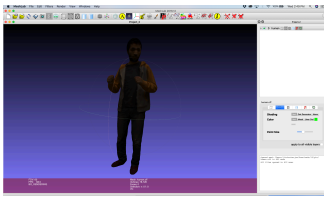
Code Snippet:

```
void doOcclusion(const char* ipFilNam){
    int N;
    point3d *pt = readOff(ipFilNam, &N);
    size_t row = 500, col = 500, dim = 3;
    mat *img = allocMatMem(row, col, dim);
    from3dtoPinholeProjWOcclusion(pt, (unsigned)N, img, row, col);
    char projOpFilNam[MAX_FIL_NAM_SIZ];
    mkOpNameWExt(projOpFilNam, "Occ", (char)N, ipFilNam, ".ppm", sizeof
 (projOpFilNam));
    imgMatToPgm(projOpFilNam, img, row, col, 3, (unsigned)MAX_PIX_VAL,
 eP6);
}

void from3dtoPinholeProjWOcclusion(point3d *pt, size_t N, mat *img,
 size_t row, size_t col){
    RigTrans *r = rigTransIn3d(pt, N, 0, 0, 0, 0, 0, 0);
    size_t nN;
    RigTrans *p = occludeRigTrans(r, N, &nN, 0.002);
    free(r);
    RigTrans *rp = rigTransToPerspec(p, nN, ORIGINAL_IMG_F);
    free(p);
    PerspecImg *pp = perspecToImage(rp, nN);
    free(rp);
```

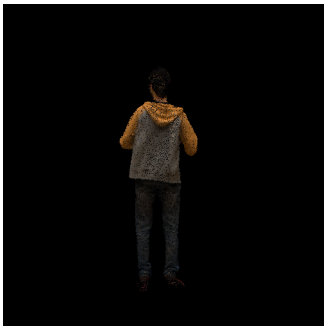
```
    PerspecImg *s = scalePerspecImage(pp, nN, row, col);
    free(pp);
    prespecToImg(s, nN, img, row, col);
    free(s);
}
```

Input Image: human.off



Input Image in meshlab software (Fig 1.8)

Output Image:



Occlusion of Fig 1.8 (Fig 1.9)

Conclusion:

Hence, in the above outputs we have shown different types of Image Formation techniques.
