Dimitri Charneux

Rendu tp:

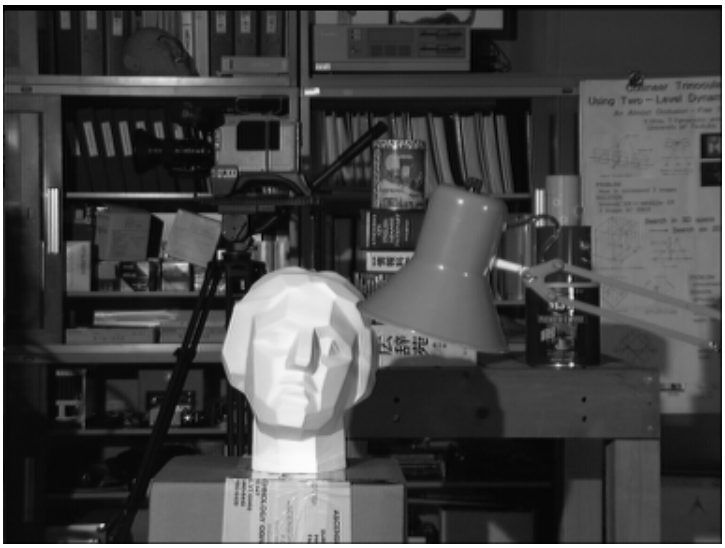
Stéréovision dense.

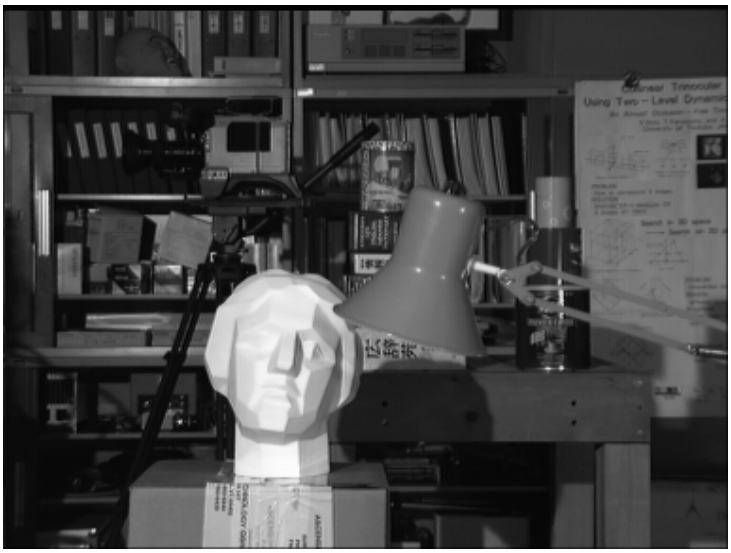
Table des matières

Introduction

Durant ce TP, nous avons travaillé sur la stéréovision dense. Cette technique qui utilise deux images, une gauche et une droite, consiste à calculer la disparité de chaque pixel entre les deux images. Pour réaliser cette méthode, il faut soit utiliser un stéréoscope canonique, soit ajouter une étape de rectification épipolaire pour revenir à une configuration canonique.

L'objectif de ce TP est donc de calculer les disparité entre les deux images suivantes représentant la même scène mais avec des caméras décalées de quelques centimètres.





1. Similarité par SSD

La première étape de ce TP a été d'analyser la fonction *iviLeftDisparityMap*. Cette fonction sert à calculer la carte de disparité des deux images en prenant l'image gauche comme référence.

Cette méthode utilise la fonction suivante que nous avons dû coder :

```
Mat iviComputeLeftSSDCost(const Mat& mLeftGray,
                          const Mat& mRightGray,
                          int iShift,
                          int iWindowHalfSize) {

      Mat mLeftSSDCost(mLeftGray.size(), CV_64F);
      for(int x = 0 ; x < mLeftGray.cols - iWindowHalfSize ; x++){
            for(int y = 0 ; y < mLeftGray.rows - iWindowHalfSize ; y++){
                  double ssd = 0.0;
                  for(int i = -iWindowHalfSize ; i <= iWindowHalfSize ; i++){
                        for(int j = -iWindowHalfSize
                                    ; j <= iWindowHalfSize ; j++){
                              double    I1    =    (double)mLeftGray.at<unsigned
char>(y+j, x+i);
                              double    I2    =    (double)mRightGray.at<unsigned
char>(y+j, x+i-iShift);
                              ssd+= pow(I1 - I2 , 2.0);
                        }
                  }
                  mLeftSSDCost.at<double>(y, x) = ssd;
            }
```

```
        }
        return mLeftSSDCost;
}
```

Cette fonction réalise le calcul de l'indice de similarité qui est constitué de la somme des différences entre chaque pixel le tout au carré.

On parcours donc chaque colonne et chaque ligne de l'image gauche pour parcourir tout les pixels et on applique la formule de SSD suivantes :

$$SSD(x_l, y, s) = \sum_{i=-w_x}^{w_x} \sum_{j=-w_y}^{w_y} \left(\mathbf{I}_l(x_l + i, y + j) - \mathbf{I}_r(x_l + i - s, y + j) \right)^2$$

Grâce à ces méthodes, nous obtenons l'image suivante comme disparité de l'image droite par rapport à la gauche :

Maintenant que nous avons calculé la disparité de l'image de droite par rapport à l'image de gauche, nous allons voir la similarité inverse et effectuer une cohérence gauche-droite pour vérifier nos résultats.

2. Vérification gauche-droite

La première étape que nous avons réalisé pour effectuer la vérification gauche-droite a été de calculer la disparité de l'image de gauche par rapport à l'image de droite.

Pour cela, nous avons écrit deux méthode *iviRightDisparityMap* et *iviComputeRightSSDCost* qui sont les mêmes méthodes que pour la partie précédente mais avec les images de gauche et de droite inversées.

Ces deux méthodes sont fournies en annexe et nous donnent la disparité suivantes :

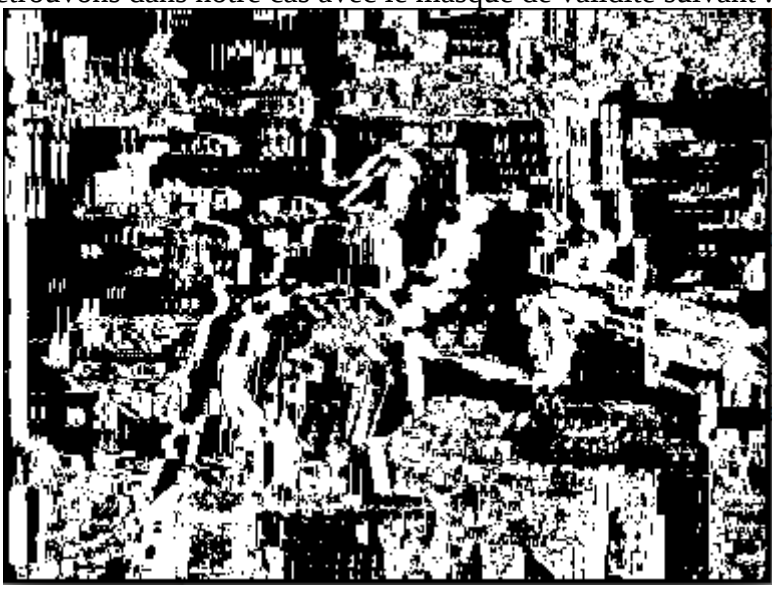
Nous avons ensuite dû coder la fonction *iviLeftRightConsistency* qui effectue une vérification gauche-droite des deux disparités calculées précédemment.

```
Mat iviLeftRightConsistency(const Mat& mLeftDisparity,
                            const Mat& mRightDisparity,
                            Mat& mValidityMask) {
    Mat mDisparity(mLeftDisparity.size(), CV_8U);
    for(int xl = 0; xl < mLeftDisparity.cols; xl++) {
      for (int y = 0; y < mLeftDisparity.rows; y++) {
        int xr = xl - (double) mLeftDisparity.at<unsigned char>(y,xl);
        if((double) mLeftDisparity.at<unsigned char>(y, xl) ==
            (double) mRightDisparity.at<unsigned char>(y,
                  xl- (double) mLeftDisparity.at<unsigned char>(y, xl))) {
          mValidityMask.at<unsigned char>(y, xl) = 0;
          mDisparity.at<unsigned char>(y, xl)
                 = (double) mLeftDisparity.at<unsigned char>(y, xl);
         } else {
          mValidityMask.at<unsigned char>(y, xl) = 255;
         }
      }
    }
    return mDisparity;
}
```

Cette fonction nous renvoie un masque de validité qui contient des pixels noirs et blancs.

Les pixels sont noirs quand la disparité calculée à la position donnée est correcte. Ils sont blancs quand la disparité est incorrecte ce qui correspond généralement aux occultations entre les deux images.

Nous nous retrouvons dans notre cas avec le masque de validité suivant :



La suite du TP, qui nous demandait d'étudier un rapport technique pour amélioré le temps de calcul des SSD, n'a pas été faites par manque de temps.

Conclusion

Nous avons appris durant ce TP une technique de stéréovision dense nous permettant de calculer la disparité entre chaque pixels de deux images.

Nous avons ensuite implémenté un algorithme de vérification gauche-droite pour établir un masque de validité de nos disparité.

Annexe

Code de la méthode *iviRightDisparityMap :*

```
Mat iviRightDisparityMap(const Mat& mLeftGray,
                         const Mat& mRightGray,
                         int iMaxDisparity,
                         int iWindowHalfSize) {
     Mat mSSD(mRightGray.size(), CV_64F);
     Mat mMinSSD(mRightGray.size(), CV_64F);
     Mat mRightDisparityMap(mRightGray.size(), CV_8U);
     double dMinSSD, *pdPtr1, *pdPtr2;
     unsigned char *pucDisparity;
     int iShift, iRow, iCol;

    // Initialisation de l'image du minimum de SSD
    dMinSSD = pow((double)(2 * iWindowHalfSize + 1), 2.0) * 512.0;
    for (iRow = iWindowHalfSize;
        iRow < mMinSSD.size().height - iWindowHalfSize;
        iRow++) {
        // Pointeur sur le debut de la ligne
        pdPtr1 = mMinSSD.ptr<double>(iRow);
        // Sauter la demi fenetre non utilisee
        pdPtr1 += iWindowHalfSize;
        // Remplir le reste de la ligne
        for (iCol = iWindowHalfSize;
            iCol < mMinSSD.size().width - iWindowHalfSize;
            iCol++)
                *pdPtr1++ = dMinSSD;
    }
    // Boucler pour tous les decalages possibles
    for (iShift = 0; iShift < iMaxDisparity; iShift++) {
        // Calculer le cout SSD pour ce decalage
        mSSD = iviComputeRightSSDCost(mRightGray, mLeftGray,
                                      iShift, iWindowHalfSize);
        // Mettre a jour les valeurs minimales
        for (iRow = iWindowHalfSize;
            iRow < mMinSSD.size().height - iWindowHalfSize;
            iRow++) {
            // Pointeurs vers les debuts des lignes
            pdPtr1 = mMinSSD.ptr<double>(iRow);
            pdPtr2 = mSSD.ptr<double>(iRow);
            pucDisparity = mRightDisparityMap.ptr<unsigned char>(iRow);
            // Sauter la demi fenetre non utilisee
            pdPtr1 += iWindowHalfSize;
            pdPtr2 += iWindowHalfSize;
            pucDisparity += iWindowHalfSize;
            // Comparer sur le reste de la ligne
            for (iCol = iWindowHalfSize;
                iCol < mMinSSD.size().width - iWindowHalfSize;
                iCol++) {
                // SSD plus faible que le minimum precedent
                if (*pdPtr1 > *pdPtr2) {
                    *pucDisparity = (unsigned char)iShift;
                    *pdPtr1 = *pdPtr2;
                }
                // Pixels suivants
                pdPtr1++; pdPtr2++; pucDisparity++;
```

```
            }
        }
    }
      return mRightDisparityMap;
}
```

Code de la méthode iviComputeRightSSDCost:

```
Mat iviComputeRightSSDCost(const Mat& mLeftGray,
                           const Mat& mRightGray,
                           int iShift,
                           int iWindowHalfSize) {
      Mat mRightSSDCost(mLeftGray.size(), CV_64F);
    for(int x = 0 ; x < mRightGray.cols - iWindowHalfSize ; x++){
      for(int y = 0 ; y < mRightGray.rows - iWindowHalfSize ; y++){
            double ssd = 0.0;
            for(int i = -iWindowHalfSize ; i <= iWindowHalfSize ; i++){
                  for(int j = -iWindowHalfSize ; j <= iWindowHalfSize ; j++){
                              double    I1    =    (double)mRightGray.at<unsigned
char>(y+j, x+i);
                              double    I2    =    (double)mLeftGray.at<unsigned
char>(y+j, x+i-iShift);
                              ssd+= pow(I1 - I2 , 2.0);
                  }
            }
            mRightSSDCost.at<double>(y, x) = ssd;
      }
    }
    return mRightSSDCost;
}
```