1. BlendShape原理以及代码实现

2.1 原理

我们认为一张新的人脸是其他张人脸的线性组合。表示平均人脸。表示Blend Shape的权,其约束满足。



2.2 代码实现

2.2.1 数据结构

std::vector<std::vector<cv::Vec4f>> vertices; /\* Vertex coordinates. \*/

本工程的顶点存储采用Vector进行存储。Vertices[i]表示不不同的人脸形状。对于vertices[0]为平均人脸模型。

void ShapeDeformation(Vec4f\* pVertexArray, int weight[BLENDSHAPE\_NUM],

bool bUseKeyPoint, MorphableModel model)

{

/\* Getting the number of Vertex. \*/

int lVertexCount = model.Head.VertexNum;

Vec4f\* lSrcVertexArray = pVertexArray;

/\* Copy the Srcouce array to dst array. \*/

Vec4f\* lDstVertexArray = new Vec4f[lVertexCount];

memcpy(lDstVertexArray, pVertexArray, lVertexCount \* sizeof(Vec4f));

/\* Getting the channel number of a BlendShape. \*/

int lBlendShapeChannelCount = model.Head.BlendShapeNum;

lBlendShapeChannelCount = lBlendShapeChannelCount > BLENDSHAPE\_NUM ? BLENDSHAPE\_NUM : lBlendShapeChannelCount;

for (int lChannelIndex = 1; lChannelIndex < lBlendShapeChannelCount; lChannelIndex++){

int lWeight = weight[lChannelIndex];

for (int j = 0; j < lVertexCount; j++){

lDstVertexArray[j] += (model.Head.vertices[lChannelIndex][j] - lSrcVertexArray[j]) \* lWeight \* 0.01f;

}

}

memcpy(pVertexArray, lDstVertexArray, lVertexCount \* sizeof(Vec4f));

delete [] lDstVertexArray;

}

1. 基于NNLS的人脸拟合

对于很多问题最优化问题，通常可以转换为最小二乘法求解的问题。即：



由于fbx中的Blend Shape 采用2.1中的变形方式，因此原式转为：



令，。矩阵和向量均已知。同时，

且人脸形状为向量形式。故，上式转为非负最小二乘问题。



3.1 代码实现

MatrixXf A = MatrixXf::Zero(TWO\_DIMENSIONAL\_COORDINATES \* FACE\_LANDMRAK\_NUM, BLENDSHAPE\_NUM);

VectorXf b = VectorXf::Zero(TWO\_DIMENSIONAL\_COORDINATES \* FACE\_LANDMRAK\_NUM);

VectorXf x;

for(int n = 0; n < BLENDSHAPE\_NUM; n++){

for(int j = 0; j < TWO\_DIMENSIONAL\_COORDINATES; j++){

for(int i = 0; i < FACE\_LANDMRAK\_NUM; i++){

int indices = VertexIndices[i];

if( j == 0){

A(j \* FACE\_LANDMRAK\_NUM + i, n) = (float)(( (Model.Head.vertices[n + 1][indices][j] - Model.Head.vertices[0][indices][j])) \* sp.s);

}else if( j == 1){

A(j \* FACE\_LANDMRAK\_NUM + i, n) = (float)((-(Model.Head.vertices[n + 1][indices][j] - Model.Head.vertices[0][indices][j])) \* sp.s);

}

}

}

}

for(int j = 0; j < TWO\_DIMENSIONAL\_COORDINATES; j++){

for(int i = 0; i < FACE\_LANDMRAK\_NUM; i++){

int indices = VertexIndices[i];

if(j == 0){

b(j\*FACE\_LANDMRAK\_NUM + i) = vFaceLandmarksPos[i][j] - (float)((Model.Head.vertices[0][indices][j] + sp.tx) \* sp.s);

}else if(j == 1){

b(j\*FACE\_LANDMRAK\_NUM + i) = vFaceLandmarksPos[i][j] - (float)(((-Model.Head.vertices[0][indices][j]) + sp.ty) \* sp.s);

}

}

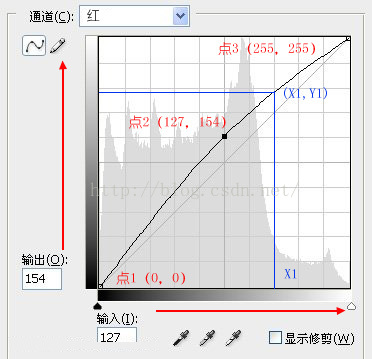
}

bool non\_singular = Eigen::NNLS<MatrixXf>::solve(A, b, x);

1. 皮肤美白

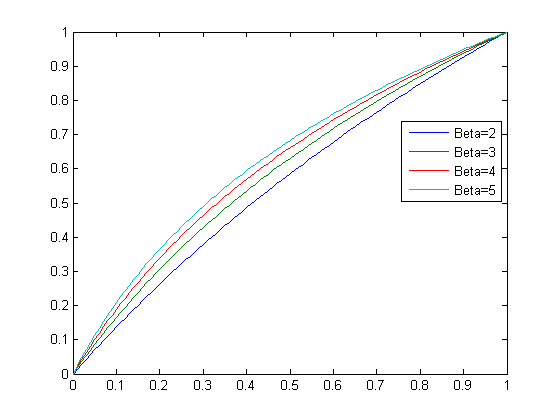
4.1 算法原理

对于很多关于皮肤的处理算法本质都是来源于Photoshop。在Photoshop中可以对图像进行曲线调整。要提高皮肤美白程度，按照一定的方式调整皮肤颜色曲线即可。如下图所示。



对于该曲线我们可以通过下面函数进行拟合。该函数的值域为0到1，因此使用过程中要对像素的范围进行变换。





4.2 代码实现

void SkinWhitening(Mat src,Mat &dst,int beta){

for (int i = 0; i < src.rows; ++i){

for (int j = 0; j < src.cols; ++j){

for(int c = 0; c < 3; c++){

float t = src.at<Vec3b>(i, j)[c] / 255.0f;

t = (float)(log(t \* (beta - 1) + 1.0f) / log(beta));

dst.at<Vec3b>(i, j)[c] = (unsigned char)(t \* 255);

}

}

}}