In the following I will present a method for deforming three dimensional geometry using a technique relying on *radial basis functions* (RBFs). These are mathematical functions that take a real number as input argument and return a real number. RBFs can be used for creating a smooth interpolation between values known only at a discrete set of positions. The term *radial* is used because the input argument given is typically computed as the distance between a fixed position in 3D space and another position at which we would like to evaluate a certain quantity.

The tutorial will give a short introduction to the linear algebra involved. However the source code contains a working implementation of the technique which may be used as a black box.

Interpolation by radial basis functions

Assume that the value of a scalar valued function $F : \mathbb{R}^3 \rightarrow \mathbb{R}$ is known in M distinct discrete points $\mathbf{x}_i$ in three dimensional space. Then RBFs provide a means for creating a smooth interpolation function of F in the whole domain of $\mathbb{R}^3$. This function is written as a sum of M evaluations of a radial basis function $g(r_i) : \mathbb{R} \rightarrow \mathbb{R}$ where r_i is the distance between the point $\mathbf{x} = (x, y, z)$ to be evaluated and $\mathbf{x}_i$:

$$!F(\mathbf{x}) = \sum_{i=1}^{M} a_i g(||\mathbf{x} - \mathbf{x}_i||) + c_0 + c_1 x + c_2 y + c_3 z, \mathbf{x} = (x, y, z) \mathbf{(1)}$$

Here a_i are scalar coefficients and the last four terms constitute a first degree polynomial with coefficients c_0 to c_3. These terms describe an affine transformation which cannot be realised by the radial basis functions alone. From the M known function values $F(x_i, y_i, z_i) = F_i$ we can assemble a system of $M + 4$ linear equations: $\mathbf{GA} = \mathbf{F}$
where $\mathbf{F} = (F_1, F_2, \ldots, F_M, 0, 0, 0, 0)$, $\mathbf{A} = (a_1, a_2, \ldots, a_M, c_0, c_1, c_2, c_3)$ and $\mathbf{G}$ is an $(M + 4) \times (M + 4)$ matrix :

$$!\mathbf{G} = \begin{bmatrix}
g_{11} & g_{12} & \bullet & \bullet & \bullet & g_{1M} & 1 & x_1 & y_1 & z_1 \\
g_{21} & g_{22} & \bullet & \bullet & \bullet & g_{2M} & 1 & x_2 & y_2 & z_2 \\
\bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
\bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
\bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
g_{M1} & g_{M2} & \bullet & \bullet & \bullet & g_{MM} & 1 & x_M & y_M & z_M \\
1 & 1 & \bullet & \bullet & \bullet & 1 & 0 & 0 & 0 & 0 \\
x_1 & x_2 & \bullet & \bullet & \bullet & x_M & 0 & 0 & 0 & 0 \\
y_1 & y_2 & \bullet & \bullet & \bullet & y_M & 0 & 0 & 0 & 0 \\
z_1 & z_2 & \bullet & \bullet & \bullet & z_M & 0 & 0 & 0 & 0
\end{bmatrix}$$

Here $g_{ij} = g(||\mathbf{x}_i - \mathbf{x}_j||)$. A number of choices for g will result in a unique solution of the system. In this tutorial we use the *shifted log function*:

$$!g(t) = \sqrt{\log(t^2 + k^2)}, k^2 \geq 1$$

with $k = 1$. Solving the equation system for $\mathbf{A}$ gives us the coefficients to use in equation $\mathbf{(1)}$ when interpolating between known values.

Interpolating displacements

How can RBF's be used for deforming geometry? Well assume that the deformation is known for M 3D positions $\mathbf{x}_i$ and that this information is represented by a vector describing 3D displacement $\mathbf{u}_i$ of the geometry that was positioned at $\mathbf{x}_i$ in the original, undeformed state. You can think of the $\mathbf{x}_i$ positions as control points that have been moved to positions $\mathbf{x}_i + \mathbf{u}_i$. The RBF interpolation method can now be used for interpolating these displacements to other positions.

Using the notation $\mathbf{x}_i = (x_i, y_i, z_i)$ and $\mathbf{u}_i = (u_i^x, u_i^y, u_i^z)$ three linear systems are set up as above letting the displacements u be the quantity we called a in the previous section:

$$!\mathbf{G}\mathbf{A}_x = (u_1^x, u_2^x, \ldots, u_M^x, 0, 0, 0, 0)^T$$
$$!\mathbf{G}\mathbf{A}_y = (u_1^y, u_2^y, \ldots, u_M^y, 0, 0, 0, 0)^T$$
$$!\mathbf{G}\mathbf{A}_z = (u_1^z, u_2^z, \ldots, u_M^z, 0, 0, 0, 0)^T$$

where $\mathbf{G}$ is assembled as described above. Solving for $\mathbf{A}_x$, $\mathbf{A}_y$, and $\mathbf{A}_z$ involves a single matrix inversion and three matrix-vector multiplications and gives us the coefficients for interpolating displacements in all three directions by the expression (1)

The source code

In the source code accompanying this tutorial you will find the class RBFInterpolator which has an interface like this:

```
class RBFInterpolator
{
public:
    RBFInterpolator();
    ~RBFInterpolator();

    //create an interpolation function f that obeys F_i = f(x_i, y_i, z_i)
    RBFInterpolator(vector x, vector y, vector z, vector F);

    //specify new function values F_i while keeping the same
    void UpdateFunctionValues(vector F);

    //evaluate the interpolation function f at the 3D position (x,y,z)
    real interpolate(float x, float y, float z);

private:
            ...
};
```

This class implements the interpolation method described above using the newmat matrix library. It is quite easy to use: just fill stl::vectors with the x_i, y_i and z_i components of the positions where the value F is known and another stl::vector with the F_i values. Then pass these vectors to the RBFInterpolator constructor, and it will be ready to interpolate. The F value at any position is then evaluated by calling the 'interpolate' function. If some of the F_i values change at any time, the interpolator can be quickly updated using the 'UpdateFunctionValues' method.

We want to deform a triangle surface mesh. These are stored in a class TriangleMesh, and loaded from OBJ files.
In the source code the allocation of stl::vectors describing the control points and the initialisation of RBFInterpolators looks like this:

```
void loadMeshAndSetupControlPoints()
{
    // open an OBJ file to deform
    string sourceOBJ = "test_dragon.obj";
    undeformedMesh = new TriangleMesh(sourceOBJ);
    deformedMesh = new TriangleMesh(sourceOBJ);

    // we want 11 control points which we place at different vertex positions
    const int numControlPoints = 11;
```

```
    const int verticesPerControlPoint = ((int)undeformedMesh-
>getParticles().size())/numControlPoints;

    for (int i = 0; i<numControlPoints; i++)
    {
        Vector3 pos = undeformedMesh->getParticles()
[i*verticesPerControlPoint].getPos();
        controlPointPosX.push_back(pos[0]);
        controlPointPosY.push_back(pos[1]);
        controlPointPosZ.push_back(pos[2]);
    }

    // allocate vectors for storing displacements
    for (unsigned int i = 0; i<controlPointPosX.size();  i++)
    {
        controlPointDisplacementX.push_back(0.0f);
        controlPointDisplacementY.push_back(0.0f);
        controlPointDisplacementZ.push_back(0.0f);
    }

    // initialize interpolation functions
    rbfX = RBFInterpolator(controlPointPosX, controlPointPosY,
                                            controlPointPosZ,
controlPointDisplacementX );
    rbfY = RBFInterpolator(controlPointPosX, controlPointPosY,
                                            controlPointPosZ,
controlPointDisplacementY );
    rbfZ = RBFInterpolator(controlPointPosX, controlPointPosY,
                                            controlPointPosZ,
controlPointDisplacementZ );
}
```

Now all displacements are set to zero vectors – not terribly exciting! To make it a bit more fun we can vary the displacements with time:

```
    // move control points
    for (unsigned int i = 0; i < controlPointPosX.size(); i++ )
    {
        controlPointDisplacementX[i] =
displacementMagnitude*cosf(time+i*timeOffset);
        controlPointDisplacementY[i] = displacementMagnitude*sinf(2.0f*
(time+i*timeOffset));
        controlPointDisplacementZ[i] = displacementMagnitude*sinf(4.0f*
(time+i*timeOffset));
    }

    // update the control points based on the new control point positions
    rbfX.UpdateFunctionValues(controlPointDisplacementX);
    rbfY.UpdateFunctionValues(controlPointDisplacementY);
    rbfZ.UpdateFunctionValues(controlPointDisplacementZ);

    // deform the object to render
    deformObject(deformedMesh, undeformedMesh);
```

Here the function ‘deformObject’ looks like this:

```
// Code for deforming the mesh 'initialObject' based on the current interpolation
functions (global variables).
// The deformed vertex positions will be stored in the mesh 'res'
// The triangle connectivity is assumed to be already correct in 'res'
void deformObject(TriangleMesh* res, TriangleMesh* initialObject)
{
    for (unsigned int i = 0; i < res->getParticles().size(); i++)
    {
        Vector3 oldpos = initialObject->getParticles()[i].getPos();

        Vector3 newpos;
        newpos[0] = oldpos[0] + rbfX.interpolate(oldpos[0], oldpos[1],
oldpos[2]);
        newpos[1] = oldpos[1] + rbfY.interpolate(oldpos[0], oldpos[1],
oldpos[2]);
        newpos[2] = oldpos[2] + rbfZ.interpolate(oldpos[0], oldpos[1],
oldpos[2]);

        res->getParticles()[i].setPos(newpos);
    }
}
```

That’s it!!!