Implemented functions - Group 16

NOTE: We have not included the functions that were modified to render the data in low resolution during interactive mode. Furthermore, we have also excluded the ui code for the gradient magnitude controls in the TransferFunction2DView.

weight

```
    private float weight (float x)
    {
      float a = -0.5f;
      x = (x < 0) ? -x : x;
      float result = 0;
      if (x < 1.0) {
        result = (a + 2) * x * x * x - (a + 3) * x * x + 1;
      } else if (x < 2.0) {
         result = a * x * x * x - 5 * a * x * x + 8 * a * x
           - 4 * a;
      } else {
        result = 0;
      }

      return (float)result;
   }
```

cubicInterpolate

```
    private float cubicinterpolate(float g0, float g1, float g2,
      float g3, float factor) {

      float dx0 = 1 + factor;
      float dx1 = factor;
      float dx2 = 1 - factor;
      float dx3 = 2 - factor;

      float result = weight(dx0) * g0 + weight(dx1) * g1 +
        weight(dx2) * g2 + weight(dx3) * g3;

      return result;
    }
```

bicubicInterpolate

```
    private float bicubicinterpolateXY(double[] coord,int z) {

        float x = (float)coord[0];
        float y = (float)coord[1];
        //get coords for points around coord
        int x1 = (int) Math.floor(coord[0]);
        int y1 = (int) Math.floor(coord[1]);

        int x0 = x1 - 1;
        int y0 = y1 - 1;

        int x2 = x1 + 1;
        int y2 = y1 + 1;

        int x3 = x1 + 2;
        int y3 = y1 + 2;

        float t0 = cubicinterpolate(
                getVoxel(x0, y0, z),
                getVoxel(x1, y0, z),
                getVoxel(x2, y0, z),
                getVoxel(x3, y0, z),
                x - x1);
        float t1 = cubicinterpolate(
                getVoxel(x0, y1, z),
                getVoxel(x1, y1, z),
                getVoxel(x2, y1, z),
                getVoxel(x3, y1, z),
                x - x1);
        float t2 = cubicinterpolate(
                getVoxel(x0, y2, z),
                getVoxel(x1, y2, z),
                getVoxel(x2, y2, z),
                getVoxel(x3, y2, z),
                x - x1);
        float t3 = cubicinterpolate(
                getVoxel(x0, y3, z),
                getVoxel(x1, y3, z),
                getVoxel(x2, y3, z),
                getVoxel(x3, y3, z),
                x - x1);

        float result = cubicinterpolate(t0, t1, t2, t3, y - y1);

        return result;

    }
```

cubicInterpolate

```java
    public float getVoxelTriCubicInterpolate(double[] coord) {
        if (coord[0] < 1 || coord[0] > (dimX-3) || coord[1] < 1
            || coord[1] > (dimY-3)
                || coord[2] < 1 || coord[2] > (dimZ-3)) {
            return 0;
        }
        float z = (float)coord[2];

        int z1 = (int) Math.floor(coord[2]);
        int z0 = z1 - 1;
        int z2 = z1 + 1;
        int z3 = z1 + 2;

        float t0 = bicubicinterpolateXY(coord, z0);
        float t1 = bicubicinterpolateXY(coord, z1);
        float t2 = bicubicinterpolateXY(coord, z2);
        float t3 = bicubicinterpolateXY(coord, z3);

        float result = cubicinterpolate(t0, t1, t2, t3, z - z1);
        result = result < 0 ? 0 : result > 255 ? 255 : result;
        return result;
    }
```

getGradient

```java
    public VoxelGradient getGradient(double[] coord) {
        if (coord[0] < 0 || coord[0] > (dimX-2) || coord[1] < 0
            || coord[1] > (dimY-2)
                || coord[2] < 0 || coord[2] > (dimZ-2)) {
            return zero;
        }

        // Compute the rounded up/down coordinate values
        double xF = Math.floor(coord[0]);
        double xC = Math.ceil(coord[0]);
        double yF = Math.floor(coord[1]);
        double yC = Math.ceil(coord[1]);
        double zF = Math.floor(coord[2]);
        double zC = Math.ceil(coord[2]);

        float dx = (float)(coord[0] - xF);
        float dy = (float)(coord[1] - yF);
        float dz = (float)(coord[2] - zF);

        // Interpolate along the x-axis
        VoxelGradient c00 = interpolate(getGradient((int) xF,
            (int) yF, (int) zF),
                                        getGradient((int) xC,
                                            (int) yF, (int) zF),
                                            1.f - dx);
        VoxelGradient c01 = interpolate(getGradient((int) xF,
            (int) yF, (int) zC),
                                        getGradient((int) xC,
                                            (int) yF, (int) zC),
                                            1.f - dx);

        VoxelGradient c10 = interpolate(getGradient((int) xF,
            (int) yC, (int) zF),
                                        getGradient((int) xC,
                                            (int) yC, (int) zF),
                                            1.f - dx);
        VoxelGradient c11 = interpolate(getGradient((int) xF,
            (int) yC, (int) zC),
                                        getGradient((int) xC,
                                            (int) yC, (int) zC),
                                            1.f - dx);

        // Interpolate along the y-axis
        VoxelGradient c0 = interpolate(c00,c10, 1.f - dy);
        VoxelGradient c1 = interpolate(c01,c11, 1.f - dy);

        // Interpolate along the z-axis
        VoxelGradient c = interpolate(c0, c1, 1.f - dz);

        return c;
    }
```

TraceRayComposite

```
int traceRayComposite(double[] entryPoint, double[]
    exitPoint, double[] rayVector, double sampleStep) {
    double[] lightVector = new double[3];

    //the light vector is directed toward the view point
        (which is the source of the light)
    // another light vector would be possible
    VectorMath.setVector(lightVector, rayVector[0],
        rayVector[1], rayVector[2]);

    //Initialization of the colors as floating point values
```

```
        double r, g, b;
        r = g = b = 0.0;
        double alpha = 0.0;
        double opacity = 0;


        TFColor voxel_color = new TFColor();
        TFColor colorAux = new TFColor();

        // Compute the number of times we need to sample
        double distance = VectorMath.distance(entryPoint,
            exitPoint);
        int nrSamples = 1 + (int) Math.floor(distance /
            sampleStep);
        //the current position is initialized as the entry point
        double[] currentPos = new double[3];
        double[] increments = new double[3];
        VectorMath.setVector(increments, rayVector[0] *
            sampleStep, rayVector[1] * sampleStep, rayVector[2]
            * sampleStep);
        VectorMath.setVector(currentPos, entryPoint[0],
            entryPoint[1], entryPoint[2]);

        if (compositingMode || tf2dMode) {
            voxel_color = computeColorTF(currentPos, increments,
                nrSamples, lightVector, rayVector);
        }

        r = voxel_color.r;
        g = voxel_color.g;
        b = voxel_color.b;
        alpha = voxel_color.a;;

        //computes the color
        int color = computeImageColor(r,g,b,alpha);
        return color;
    }
```

TraceRayISO

```
   int traceRayIso(double[] entryPoint, double[] exitPoint,
       double[] rayVector, double sampleStep) {

        double[] lightVector = new double[3];
        //We define the light vector as directed toward the view
            point (which is the source of the light)
        // another light vector would be possible
         VectorMath.setVector(lightVector, rayVector[0],
             rayVector[1], rayVector[2]);

        //Initialization of the colors as floating point values
        double r, g, b;
        r = g = b = 0.0;
        double alpha = 0.0;
        double opacity = 0;

        // To be Implemented this function right now just gives
            back a constant color
        //compute the increment and the number of samples
        double[] increments = new double[3];
        VectorMath.setVector(increments, rayVector[0] *
            sampleStep, rayVector[1] * sampleStep, rayVector[2]
            * sampleStep);

        // Compute the number of times we need to sample
        int nrSamples = 1 + (int)
            Math.floor(VectorMath.distance(entryPoint,
            exitPoint) / sampleStep);

        //the current position is initialized as the entry point
        double[] currentPos = new double[3];
        VectorMath.setVector(currentPos, entryPoint[0],
            entryPoint[1], entryPoint[2]);
        r = g = b = alpha = 0;
        do {
            double value =
                volume.getVoxelLinearInterpolate(currentPos);
            if (value > iso_value) {

                bisection_accuracy(currentPos, increments,
                    sampleStep, value, iso_value);

                // Found isosurface: Use value to compute color
                    and then break
                // isoColor contains the isosurface color from
                    the interface
                VoxelGradient gradient =
                    gradients.getGradient(currentPos);
                TFColor color = isoColor;
                if (shadingMode) {
                    color = this.computePhongShading(isoColor,
                        gradient, lightVector, rayVector);
                }

                r = color.r;
                g = color.g;
                b = color.b;
                alpha = 1.0;

                break;
            }
            for (int i = 0; i < 3; i++) {
                currentPos[i] += increments[i];
            }
            nrSamples--;
        } while (nrSamples > 0);

        //computes the color
        int color = computeImageColor(r,g,b,alpha);
        return color;
    }
```

Bisection Accuracy

```
      // Given the current sample position, increment vector of the sample (vector from previous sample to current sample) and sample Step.
   // Previous sample value and current sample value, isovalue value
    // The function should search for a position where the iso_value passes that it is more precise.
   void bisection_accuracy (double[] currentPos, double[] increments,double sampleStep, double value, float iso_value) {

        double[] prevPos = new double[3];
        //check if iso_value is before or after currentPos
        for (int i = 0; i < 3; i++)
        {
           prevPos[i] = currentPos[i] - increments[i] *sampleStep;
        }
        double prevValue = volume.getVoxelLinearInterpolate(prevPos);
        if ((prevValue > iso_value) == (value > iso_value)) {
            sampleStep *= -1;
            for (int i = 0; i < 3; i++)
            {
```

```
            prevPos[i] = currentPos[i] - increments[i]
                *sampleStep;
         }
         prevValue =
             volume.getVoxelLinearInterpolate(prevPos);
         if ((prevValue > iso_value) == (value > iso_value)) {
             return; // iso_value is not in range
         }
     }

     // check if nextPos is a vallid position
     if (prevPos[0] < 0 || prevPos[0] > (volume.getDimX()-2)
         || prevPos[1] < 0 || prevPos[1] >
         (volume.getDimY()-2)
             || prevPos[2] < 0 || prevPos[2] >
                 (volume.getDimZ()-2)) {
         return;
     }

     bisection_accuracy(currentPos, increments, sampleStep,
         prevValue, value, iso_value, 25);

}

//wrapper for the actual bisection_accuracy search. This
    method first checks if the iso_value is in its search
    range.
//Then it gets the initial previous value and finally it
    calls the actual bisection_accuracy search method.
void bisection_accuracy (double[] currentPos, double[]
    increments,double sampleStep, double previousvalue,double
    value, float iso_value, int depth) {
     if (Math.abs(value - iso_value) < 0.001) {
         return;
     }
     if (depth < 0) {
        return;
     }

     sampleStep *= 0.5;

     // we are past the iso point thus go to the other
         direction
     if ((previousvalue > iso_value) != (value > iso_value)) {
         sampleStep *= -1;
     }
```

```

        // goto the midpoint
        for (int i = 0; i < 3; i++)
        {
            currentPos[i] += increments[i] *sampleStep;
        }
        double nextValue =
            volume.getVoxelLinearInterpolate(currentPos);
        bisection_accuracy(currentPos, increments, sampleStep,
            value, nextValue,iso_value, --depth);
    }
```

ComputePhongShading

```
    TFColor computePhongShading(TFColor voxel_color,
        VoxelGradient gradient, double[] lightVector,
            double[] rayVector) {
        if (gradient.mag < 0.0001)
            return voxel_color;
        // In a 3D scalar field, the gradient evaluated on an
            isosurface is the (unnormalized) normal
        double[] normal = {gradient.x / gradient.mag, gradient.y
            / gradient.mag, gradient.z / gradient.mag};

        // Given parameters for our phong material
        double k_a = 0.1;
        double k_d = 0.7;
        double k_s = 0.2;
        double alpha = 100;

        //make sure the the normal is facing the viewer
        double diffuse = VectorMath.dotproduct(normal,
            lightVector);
        if (diffuse < 0) {
            normal[0] *= -1;
            normal[1] *= -1;
            normal[2] *= -1;
        }
        diffuse = VectorMath.dotproduct(normal, lightVector);

        // Computing the halfway vector
        double[] vecR = {
            2 * diffuse * normal[0] - lightVector[0],
            2 * diffuse * normal[1] - lightVector[1],
            2 * diffuse * normal[2] - lightVector[2]
        };
        double specular = Math.pow(VectorMath.dotproduct(rayVector, vecR), alpha);

        TFColor color = new TFColor(0,0,0,voxel_color.a);
        color.r = (k_a * voxel_color.r) + (k_d * diffuse * voxel_color.r) + (k_s * specular);
        color.g = (k_a * voxel_color.g) + (k_d * diffuse * voxel_color.g) + (k_s * specular);
        color.b = (k_a * voxel_color.b) + (k_d * diffuse * voxel_color.b) + (k_s * specular);

        return color;
    }
```

ComputeOpacity2DTF

```
public double computeOpacity2DTF(double voxelValue, double gradMagnitude) {
    double opacity = 0.0;

    // Angle (in radians) between the triangle radius and max gradient magnitude
    double theta = Math.atan(tFunc2D.radius / gradients.getMaxGradientMagnitude());

    // Angle (in radians) between the voxel and center of the base of the triangle intensity
    double dCenter = Math.abs(voxelValue - tFunc2D.baseIntensity);
    double voxelAngle = Math.atan(dCenter / gradMagnitude);

    // Assign an opacity if the voxel is located inside the specified triangle
    if(voxelAngle < theta && gradMagnitude <= tFunc2D.maxMagnitude && gradMagnitude >= tFunc2D.minMagnitude) {
        double centerDist = voxelAngle / theta;
        opacity = 1 - centerDist;
    } // If not, the voxel will be transparent

    return opacity;
}
```