# Assignment #5:
# Scalar Field Visualization 3D: Direct Volume Rendering

Due October 4th, before midnight

## Goals:

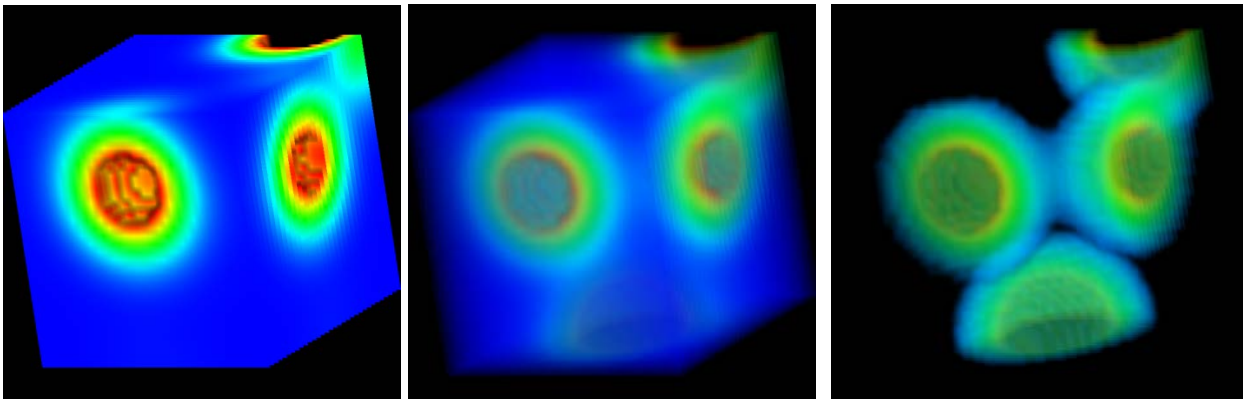
This is the continuation of Assignment 4. The goal is to implement a simple DVR -- 2D texture-based volume rendering. You will need to re-use the 3D synthetic temperature generated in Assignment 4. Your color coding, 3D slicing, and wire-frame iso-surfacing should still be functional for this Assignment.

## Tasks:

**Task 1: Writing question (20 points)**

1. There are a number of important components (or sub-tasks) needed by direct volume rendering. Please describe them. Which one is the most important sub-task, and why?
2. One issue of the image-order method (e.g., ray-casting) is that not all vorxels participate during the rendering. Describe a strategy that may involve more voxels during the ray-casting *without using the object-order approach*.

**Task 2. Texture-based Direct Volume Rendering (80 points)**



Implement the 2D texture approach of the texture-based volume rendering. Here is a more detailed instruction on how you should do this.

First, create a volume of nodes. You should have done this for the previous assignment, but you may need to re-set the resolution of your 3D grid (see below).

```
struct node Nodes[NX][NY][NZ];
```

**//Pick NX, NY, and NZ such that each is a power of 2, e.g. 64, 128, 256…**

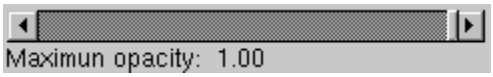
**In JS Version**

You can reuse the `Node` class in the Assignment 4. To create a 3D array in Javascript and assign `Node` values to the array, you can follow the following sample code:

```
var Nodes = new Array();

for (var i = 0; i < NX; i++) {
    Nodes[i] = new Array();
      for (var j = 0; j < NY; j++) {
      Nodes[i][j] = new Array();
            for (var k = 0; k < NZ; k++) {
            let myNode = new Node();

            // Update the value of the myNode here

            ...

            Nodes[i][j][k] = myNode;

            }
    }
}
```

For each node structure, compute its scalar value S (**which you may need to re-compute again due to the different NX, NY, and NZ**) and its associated R, G, B. Use the same scalar value function and color-mapping scheme that you have been using in the past assignments.

Create a normal slider that will determine the opacity that the visible S values will have. (**10 points**)




Add an interface to turn on and off the GL_Linear for the texture interpolation (**10 points**).


Write 3 routines, **CompositeXY()**, **CompositeXZ()**, and **CompositeYZ()**. These will composite the voxels and fill one of the following 3 texture arrays:

```
unsigned char   TextureXY[NZ][NY][NX][4];
unsigned char   TextureXZ[NY][NZ][NX][4];
unsigned char   TextureYZ[NX][NZ][NY][4];
```

These arrays hold **4** components with R, G, B channels and an alpha channel to control the opacity.

The following routine can be used to composite the texture at XY planes. The texture on XZ and YZ planes can be composited similarly.

```
void
CompositeXY( void )
```

```
{
   int x, y, z, zz;
   float alpha;                 /* opacity at this voxel       */
   float r, g, b;                       /* running color composite     */

   for( x = 0; x < NX; x++ )
   {
        for( y = 0; y < NY; y++ )
        {
            r = g = b = 0.;
           for( zz = 0; zz < NZ; zz++ )
           {
               /* which direction to composite:       */
               if( Zside == PLUS )
                   z = zz;
               else
                   z = ( NZ-1 ) - zz;


               if( ????? ) // determine whether the value is out of the
                           // range set by the range slider
               {
                       r = g = b = 0.;
                       alpha = 0.;
               }
               else
               {
                       r = Nodes[x][y][z].r;
                       g = Nodes[x][y][z].g;
                       b = Nodes[x][y][z].b;
                       alpha = MaxAlpha;
               }

               TextureXY[zz][y][x][0] = (unsigned char) ( 255.*r + .5 );
               TextureXY[zz][y][x][1] = (unsigned char) ( 255.*g + .5 );
               TextureXY[zz][y][x][2] = (unsigned char) ( 255.*b + .5 );
               TextureXY[zz][y][x][3] = (unsigned char) ( 255.*alpha + .5 );
           }
        }
   }
}
```

**In JS Version**

The texture arrays need to be created with the type of Uint8Array which corresponds to the **unsigned char** type of C++.

```
var TextureXY = new Array();
var TextureXZ = new Array();
var TextureYZ = new Array();
for (var i = 0; i < NX; i++) {
      TextureYZ[i] = new Uint8Array(NY * NZ * 4);

}
```

Here is the sample code for **CompositeXY()** function

```
function CompositeXY() {

  var x, y, z, zz;
  var alpha;      /* opacity at this voxel  */
  var r, g, b;      /* running color composite  */
  var MaxAlpha = 1;
  var idx = 0;
  for (x = 0; x < NX; x++) {
    for (y = 0; y < NY; y++) {
      r = g = b = 0.;
      for (zz = 0; zz < NZ; zz++) {
        /* which direction to composite:  */
        if (Zside == PLUS)
          z = zz;
        else
          z = (NZ - 1) - zz;

        if (???) // determine whether the value is out of the
        // range set by the range slider

        {
          r = g = b = 0.;
          alpha = 0.;
        }
        else {
          r = Nodes[x][y][z].r;
          g = Nodes[x][y][z].g;
          b = Nodes[x][y][z].b;
        }
        var int_r = Math.floor(Math.min(255. * r + .5, 255.));
        var int_g = Math.floor(Math.min(255. * g + .5, 255.));
        var int_b = Math.floor(Math.min(255. * b + .5, 255.));
        var int_a = Math.floor(Math.min(255. * alpha + .5, 255.));

        TextureXY[zz][idx] = int_r;
        TextureXY[zz][idx + 1] = int_g;
        TextureXY[zz][idx + 2] = int_b;
        TextureXY[zz][idx + 3] = int_a;


      }
      idx += 4;
    }
```

```
  }
}
```

Depending on the rotation of the volume, you will want to draw the parallel planes in the X, Y, or Z direction, and draw them minus-to-PLUS or plus-to-MINUS. The following code tells you which of these cases is the one that you will need.

```
/* which way is a surface facing:                                    */

const int MINUS = { 0 };
const int PLUS  = { 1 };


/* what is the major direction:
        */

#define X       0
#define Y       1
#define Z       2


        . . .


int     Major;                  /* X, Y, or Z                        */
int     Xside, Yside, Zside;    /* which side is visible, PLUS or MINUS
        */


        . . .

/**
 ** determine which sides of the cube are facing the user
 **
 ** this assumes that the rotation is being done by:
 **
 **     glRotatef( Yrot, 0., 1., 0. );
 **     glRotatef( Xrot, 1., 0., 0. );
 **
 **/


void
DetermineVisibility()
{
        float xr, yr;
        float cx, sx;
        float cy, sy;
        float nzx, nzy, nzz;    /* z component of normal for x side, y side,
and z side      */

        xr = Xrot * ( M_PI/180. );
        yr = Yrot * ( M_PI/180. );
```

```
        cx = cos( xr );
        sx = sin( xr );
        cy = cos( yr );
        sy = sin( yr );

        nzx = -sy;
        nzy =  sx * cy;
        nzz =  cx * cy;


        /* which sides of the cube are showing:
        */
        /* the Xside being shown to the user is MINUS or PLUS        */

        Xside = ( nzx > 0.  ?  PLUS  :  MINUS );
        Yside = ( nzy > 0.  ?  PLUS  :  MINUS );
        Zside = ( nzz > 0.  ?  PLUS  :  MINUS );


        /* which direction needs to be composited:                   */

        if( fabs(nzx) > fabs(nzy)  &&  fabs(nzx) > fabs(nzz)  )
                Major = X;
        else if( fabs(nzy) > fabs(nzx)  &&  fabs(nzy) > fabs(nzz)  )
                Major = Y;
        else
                Major = Z;
}
```

**In JS Version**

We can have a similar version for JS

```
/**
 ** determine which sides of the cube are facing the user
 **
 ** this assumes that the rotation is being done by:
 **
 ** transform.angleX;
 ** transform.angleY;
 **
 **/


function DetermineVisibility() {
  var xr, yr;
  var cx, sx;
  var cy, sy;
  var nzx, nzy, nzz;  /* z component of normal for x side, y side, and z side */

  xr = transform.angleX;
  yr = transform.angleY;
```

```
  cx = Math.cos(xr);
  sx = Math.sin(xr);
  cy = Math.cos(yr);
  sy = Math.sin(yr);

  nzx = -sy;
  nzy = sx * cy;
  nzz = cx * cy;



  /* which sides of the cube are showing:       */
  /* the Xside being shown to the user is MINUS or PLUS   */

  Xside = (nzx > 0. ? PLUS : MINUS);
  Yside = (nzy > 0. ? PLUS : MINUS);
  Zside = (nzz > 0. ? PLUS : MINUS);


  /* which direction needs to be composited:      */

  if (Math.abs(nzx) > Math.abs(nzy) && Math.abs(nzx) > Math.abs(nzz))
    Major = X;
  else if (Math.abs(nzy) > Math.abs(nzx) && Math.abs(nzy) > Math.abs(nzz))
    Major = Y;
  else
    Major = Z;


}
```

To draw the obtained texture arrays using the OpenGL texture mapping functionality. Here is what you will do.

```
glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP );
glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP );
glTexEnvf( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE );

filter = GL_NEAREST;
if( Bilinear )
        filter = GL_LINEAR;

glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, filter );
glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, filter );
glPixelStorei( GL_UNPACK_ALIGNMENT, 1 );
glEnable( GL_TEXTURE_2D );
```

```
glBlendFunc( GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA );
glEnable( GL_BLEND );


if( Major == Z )
{
        if( Zside == PLUS )
        {
                z0 = -10.;
                dz = 20. / (float)( NZ - 1 );
        }
        else
        {
                z0 = 10.;
                dz = -20. / (float)( NZ - 1 );
        }

        for( z = 0, zcoord = z0; z < NZ; z++, zcoord += dz )
        {
                glTexImage2D( GL_TEXTURE_2D, 0, 4, NX, NY, 0, GL_RGBA,
GL_UNSIGNED_BYTE, &TextureXY[z][0][0][0] );

                glBegin( GL_QUADS );

                glTexCoord2f( 0., 0. );
                glVertex3f( -10., -10., zcoord );

                glTexCoord2f( 1., 0. );
                glVertex3f( 10., -10., zcoord );

                glTexCoord2f( 1., 1. );
                glVertex3f( 10., 10., zcoord );

                glTexCoord2f( 0., 1. );
                glVertex3f( -10., 10., zcoord );

                glEnd();
        }
}

. . .
```

## In JS Version

The main differences between the skeleton code between Assignment 4 and 5 is the vertex and fragment shaders. If you want to reuse what you had in the Assignment 4, you should update the vertex and fragment shaders to make it working with textures. The below example provides the similar OpenglGL functionality to draw the obtained texture arrays:

```
// define size and format of level 0
  const level = 0;
  const internalFormat = gl.RGBA;
```

```
const border = 0;
const format = gl.RGBA;
const type = gl.UNSIGNED_BYTE;

// for Z major
var z0, dz;
if (Major == Z) {

  if (Zside == PLUS) {
    z0 = -1.0;
    dz = 2. / (NZ - 1);
  }
  else {
    z0 = 1.0;
    dz = -2. / (NZ - 1);
  }
  var z;
  var zcoord;
  for (z = 0, zcoord = z0; z < NZ; z++ , zcoord += dz) {
    const targetTexture = gl.createTexture();
    gl.bindTexture(gl.TEXTURE_2D, targetTexture);

    gl.texImage2D(gl.TEXTURE_2D, level, internalFormat, NX, NY, border, format,
      type, TextureXY[z]);
    gl.texParameterf(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.CLAMP_TO_EDGE);
    gl.texParameterf(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.CLAMP_TO_EDGE);
    gl.texParameterf(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.LINEAR);
    gl.texParameterf(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);
    gl.pixelStorei(gl.UNPACK_ALIGNMENT, 1);

    gl.blendFunc(gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA);
    gl.enable(gl.BLEND);

    var textureCoordinates = [
      0.0, 0.0,
      1.0, 0.0,
      1.0, 1.0,
      0.0, 1.0,
    ];

    var positions = [
      -1., -1., zcoord,
      -1., 1., zcoord,
      1., 1., zcoord,
      1., -1., zcoord,
```

```
      ];

      var indices = [
        0, 1, 2,
        0, 2, 3,
      ];

      const buffers = initBuffers(positions, textureCoordinates, indices);

      draw_texture_buffers(targetTexture, buffers, modelViewMatrix,
projectionMatrix);


    }
  }
```

As we cannot use GL_QUASD in WebGL, we need to create three buffers to store vertex positions, texture coordinates and indices to draw our textures. This task is accomplished through the `initBuffers` function.

```
/**
 * Create three buffers for the texture data
 * @param {*} positions
 * @param {*} textureCoordinates
 * @param {*} indices
 */
function initBuffers(positions, textureCoordinates, indices) {
  // Vertex positions
  const positionBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer);
  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(positions), gl.STATIC_DRAW);

  // Texture coordinates
  const textureCoordBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, textureCoordBuffer);
  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(textureCoordinates),
    gl.STATIC_DRAW);

  // indices
  const indexBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);
  gl.bufferData(gl.ELEMENT_ARRAY_BUFFER,
    new Uint16Array(indices), gl.STATIC_DRAW);

  return {
```

```
    position: positionBuffer,
    textureCoord: textureCoordBuffer,
    indices: indexBuffer,
  };

}
```

And use draw_texture_buffers(targetTexture, buffers, modelViewMatrix, projectionMatrix) to draw the textures to the screen

```
/**
 *  Draw textures to the screen
 * @param {*} targetTexture
 * @param {*} buffers
 * @param {*} modelViewMatrix
 * @param {*} projectionMatrix
 */
function draw_texture_buffers(targetTexture, buffers, modelViewMatrix,
projectionMatrix) {


  // Tell WebGL how to pull out the positions from the position
  // buffer into the vertexPosition attribute
  {
    const numComponents = 3;
    const type = gl.FLOAT;
    const normalize = false;
    const stride = 0;
    const offset = 0;
    gl.bindBuffer(gl.ARRAY_BUFFER, buffers.position);
    gl.vertexAttribPointer(
      programInfo.attribLocations.vertexPosition,
      numComponents,
      type,
      normalize,
      stride,
      offset);
    gl.enableVertexAttribArray(
      programInfo.attribLocations.vertexPosition);
  }

  // Tell WebGL how to pull out the texture coordinates from
  // the texture coordinate buffer into the textureCoord attribute.
  {
    const numComponents = 2;
```

```
    const type = gl.FLOAT;
    const normalize = false;
    const stride = 0;
    const offset = 0;
    gl.bindBuffer(gl.ARRAY_BUFFER, buffers.textureCoord);
    gl.vertexAttribPointer(
      programInfo.attribLocations.textureCoord,
      numComponents,
      type,
      normalize,
      stride,
      offset);
    gl.enableVertexAttribArray(
      programInfo.attribLocations.textureCoord);
  }

  // Tell WebGL which indices to use to index the vertices
  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, buffers.indices);

  // Tell WebGL to use our program when drawing

  gl.useProgram(programInfo.program);

  // Set the shader uniforms

  gl.uniformMatrix4fv(
    programInfo.uniformLocations.projectionMatrix,
    false,
    projectionMatrix);
  gl.uniformMatrix4fv(
    programInfo.uniformLocations.modelViewMatrix,
    false,
    modelViewMatrix);

  // Specify the texture to map onto the faces.

  // Tell WebGL we want to affect texture unit 0
  gl.activeTexture(gl.TEXTURE0);

  // Bind the texture to texture unit 0
  gl.bindTexture(gl.TEXTURE_2D, targetTexture);

  // Tell the shader we bound the texture to texture unit 0
  gl.uniform1i(programInfo.uniformLocations.uSampler, 0);
```

```
  {
    const vertexCount = 6;
    const type = gl.UNSIGNED_SHORT;
    const offset = 0;
    gl.drawElements(gl.TRIANGLES, vertexCount, type, offset);
  }
}
```

## Grades:

| *Tasks* | *Total points* |
|---|---|
| 1 | 20 |
| 2 | 80 |
| | |
| | |