Graphics and Game Technology

Assignment 5

Texture Mapping

and

Isosurface visualisation

1 Part 1 - Texture mapping

This assignment will familiarize you with the different aspects involved in texture mapping. The provided framework loads texture images from PPM files, a number of which are provided. The framework also sets up most of the OpenGL 2D texturing state for you. It does not apply texture coordinates; you will be asked to do that in the assignments below.

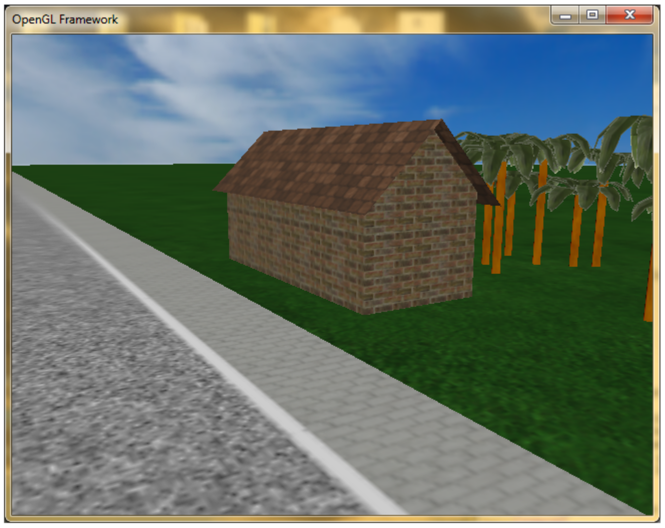


Figure 1: Screenshot of the end result: a textured house, trees, grass, road and skydome.

A number of keyboard functions are implemented by the framework application:

- `t`: toggle texturing on/off,
- `l`: render the scene using lines/wireframe,
- `p`: render the scene using polygons,
- `o`: center camera rotation to another object in the scene.

The camera position can be controlled using the mouse (left button), including zoom (right button).

The coordinate system used in the framework is as follows: the Y-axis points upwards. Imagine the X-axis pointing to the right and the Y-axis pointing up then the Z-axis points towards you. This is a so-called “right-handed” coordinate system.

1.1 3D object files

The framework includes a number of 3D objects, provided in text files with extension `.obj`. These files use a simple format, suitable for editing in a standard text editor. The format is line-oriented and any line starting with a `#` character is considered a comment line. The files declare one or more vertices (geometry) and one or more polygons (topology) using these vertices.

Vertices are declared by a single line of the format `"v <x> <y> <z>"`, giving the coordinates of the vertex. Each vertex is implicitly given an integer index, starting from 0.

Polygons are declared using multiple lines. The first line for each polygon is of the form `"p <n> <t>"` and lists the number of vertices n in the polygon and the texture identifier t. Both of these are integers. Note that the framework only supports polygons with 3 or 4 vertices (i.e. triangles and quads), as OpenGL does not really cope well with polygons with more vertices.

Then follows a line describing the color of the polygon, which is used as the object color for non-textured displaying. This line is of the form `"<r> <g> <b>"`, with three values in the range $[0, 1]$. When texturing is on, this color is combined with the color from the texture to calculate the final color.

Finally, n lines are given, one for each of the vertices in the polygon. These lines are of the form `"<vi> <s> <t>"`. Here, vi is the index of a vertex previously declared and s and t are the texture coordinates to be used for this vertex. See, for example, the file `ground.obj`, which creates a plane defined by 4 vertices. Note that an assumption is made that all polygons are planar so that a polygon normal can be easily calculated by the framework.

The texture identifier is used as an index in the array of OpenGL texture names, `texture_names`. This array contains the texture names generated at run-time for each of the loaded texture images. Take a look at function `InitGL` in `main.c` to see how this is handled.

1.2 Assignment 1.1 - Basic texturing

Complete the following tasks:

- In `DrawPolylist` add a call to `glTexCoord2f` in the correct place so that it applies texture coordinates for polygon vertices. See the lecture sheets on texture mapping for hints and code examples.

- Open the road texture `textures/road.ppm` in an image viewer, e.g. **cd textures; eog road.ppm**. This texture should be mapped on the road object (the grey strip running straight through the scene) in such a way that the full height of the image is mapped along the smallest side of the strip. Set the correct texture coordinates in `road.obj` in such a way that the texture image does not get stretched but is repeated.

  You will notice that the texture seems to be drawn only once, i.e. it isn't repeated. Alter the relevant OpenGL call(s) to enable texture repeating. Hint: see function `InitGL` and the lecture sheets.

- The green ground plane actually consist of two copies of the polygonal model defined in `ground.obj`, separated by the road. Add texture coordinates to the ground plane model so that it is covered with the grass texture 20 times in one direction and 10 times in the other one, thereby again not stretching the texture image.

- The house model in the scene (in `house.obj`) is made up of wall polygons and roof polygons. Add texture coordinates to the roof polygons so that the roof texture image is repeated 3 times along the longest side and once along the other side. The tiles in the texture should be oriented the way you would expect for a roof, see Figure 1.

  There are two types of wall polygons used in the house: one rectangular, the other 5-sided. Add textures coordinates to the walls of the house model. Do this in such a way that the texture coordinates match the different wall polygons, so that the texture continues fluently from one wall to the other. The texture image should appear a total of 7 times, when counting horizontally along the four walls. The texture image should not change its aspect ratio (the ratio between width and height of the image). This means that the applied texture should never appear stretched (horizontally nor vertically) compared to the original texture image.

  Hint: make a little drawing of the house geometry together with its dimensions.

- The scene also includes what is known as a *skydome*. This is a hemisphere surrounding the scene, which acts as a sky. The skydome is not stored in a file but is procedurally generated, by function `createHemisphere()` in `geometry.c`. In the framework, displaying of the skydome is commented out in function `DrawGLScene()`. This is done to give you a clear view of the

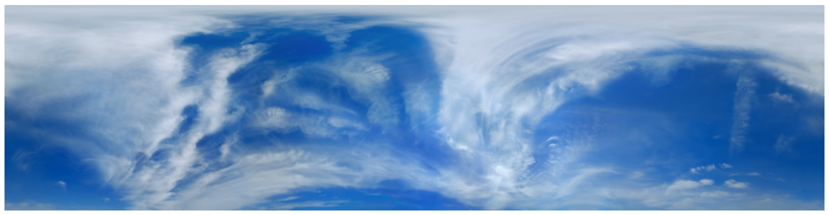


Figure 2: The sky texture that must be mapped onto the skydome.

ground plane while texturing it with the grass. Remove the commenting so that the skydome is drawn.

A texture is provided (`sky.ppm`) that should be mapped on the hemisphere, see Figure 2. If you imagine this image to be rolled up from left to right into a cylinder, followed by folding the top of the cylinder to come together in a single point, then you end up with the way the image should be mapped onto the hemisphere. I.e., the bottom of the image should run once around the bottom of the hemisphere (the horizon). The top line of the image should converge in the top of the hemisphere.

The texture coordinates for the skydome's vertices are currently all set to zero. Alter the relevant function(s) in `geometry.c` so that the correct coordinates are applied.

If you successfully set the hemisphere texture coordinates, it should be relatively straightforward to also alter function `createCylinder()`, which is used to draw the stem of a tree. Alter this function as well.

1.3 Assignment 1.2 - Mip-mapping

You might notice aliasing (or "Moiré") patterns in the different textured polygons, especially in the "sidewalk" part of the road, but also on the house walls at certain distances. These are the places where OpenGL needs to combine several texture pixels (texels) to obtain the color of a single screen pixel, so-called texture minification. The default method for this is to use linear interpolation. A better way is to use mip-mapping, as shown during the lecture.

Enable mip-mapping, by replacing the `glTexImage2D` call with a call to `gluBuild2DMipmaps` and setting/changing the relevant texturing parameters. See the man-pages of `gluBuild2DMipmaps` and `glTexParameteri` for details. Try out the different mip-mapping minification filters provides by OpenGL to notice the difference and select the one you think works best.

1.4 Assignment 1.3 - Using textures as objects

The tree objects so far look pretty simplistic, i.e. just cylinders with spheres on top. In `DrawGLScene()` the trees are drawn with a simple for-loop that

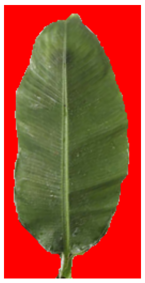


Figure 3: A texture of a single leaf from a banana palm.

draws the same polygon lists multiple times, each time with slightly different transformations to put tree objects in different locations.

We haven't done any texturing yet on the sphere, mostly because it is very similar to the skydome texturing. We are going to replace the sphere with something a bit more "leaf-like". For this, a texture image is provided of a large leaf from a banana palm, see Figure 3. To mark transparent pixels in this image, the color red is used. When the image is loaded by the framework these red pixels are replaced by fully transparent pixels.

For this part of this assignment, replace the tree spheres with simple polygonal objects that are textured using the leaf texture. For this you will have to both create some geometry and assign the relevant texture coordinates. This is most conveniently done using a separate `.obj` file. You can then read in this file and use the geometry from there. The palm texture is already loaded by the framework.

Use at least 8 vertices for a leaf and display 5 to 10 leafs at the location of the sphere (which you can remove), similar to the way a real tree's leafs would be organized. If you want to use random numbers for this, you can use function `rand_float()`, which will return a random floating-point value in the range $[0, 1]$.

See Figure 4 for a possible use of the leaf texture.

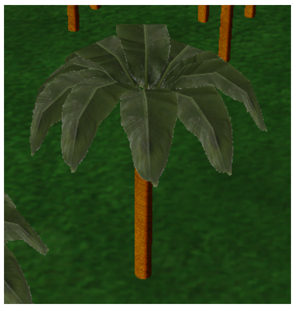


Figure 4: Multiple leafs on a palm tree.

1.5 Tips

You can view the provided texture images using an image editing/viewing package like The GIMP (`gimp`, not installed on the Linux systems) or Eye of Gnome (`eog`). With the GIMP you could even create a special test texture image to help you in setting correct texture coordinates.

2 Part 2 - Isosurface extraction

In this assignment you will explore a problem from scientific visualisation, namely the rendering of volumetric datasets. We are going to extract an isosurface, which is a polygonal surface that runs through cells of a dataset that have equal value. An example is shown in Figure 5, which shows a screenshot of what you can expect from this assignment: an isosurface extracted from a CT scan of a mummy.

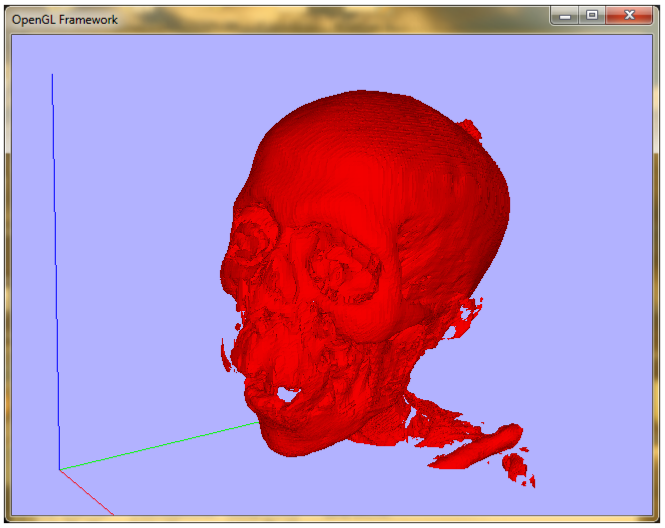


Figure 5: Screenshot of the end result: an isosurface extracted from a CT scan dataset.

The work horse in scientific visualization (and other areas) for isosurface extraction is the “marching cubes” algorithm [1]. However, although it is straightforward to implement, it is quite a lot of work to do so. Instead, we are going to implement a variation called the “marching tetrahedra” algorithm. This algorithm was invented mostly to circumvent the fact that marching cubes was patented at the time (the patent expired in 2005). Marching tetrahedra has two advantages over marching cubes: 1) it is easier to implement, 2) the resulting isosurface more closely matches the iso-values in the volume dataset. It does produce more triangles than marching cubes on the same dataset, but with today’s hardware-accelerated graphics cards this isn’t usually a problem.

We will describe the complete algorithm together with all the concepts involved.

2.1 Basics

Volume dataset

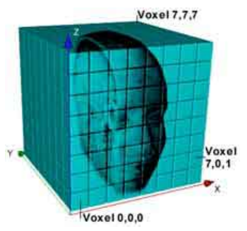


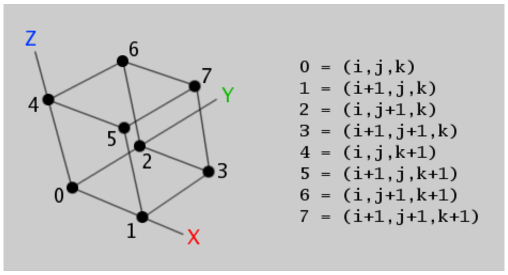
Figure 6: The voxels in a volume dataset.

A volume dataset consists of $nx \times ny \times nz$ voxels ("volumetric pixels"), see Figure 6. Each of the voxels is denoted by an index (i, j, k) with

$$0 \leq i \leq nx - 1$$
$$0 \leq j \leq ny - 1$$
$$0 \leq k \leq nz - 1$$

Cells

For the marching tetrahedra algorithm we assume that there is a single "data point" located at the center of each voxel, which has the associated voxel value. By taking together 8 neighboring data points in a $2 \times 2 \times 2$ cube shape, we create a so-called "cell" (see Figure 7).



```
0 = (i,j,k)
1 = (i+1,j,k)
2 = (i,j+1,k)
3 = (i+1,j+1,k)
4 = (i,j,k+1)
5 = (i+1,j,k+1)
6 = (i,j+1,k+1)
7 = (i+1,j+1,k+1)
```

Figure 7: A cell of the dataset, consisting of 8 neighboring data points.

Note that although we show the X, Y and Z axes here, this does not mean that data point 0 is located at the origin of the coordinate system. It merely shows that point 1 is further along the X axis with respect to point 0, point 2 is further along the Y axis, etc.

2.2 The marching tetrahedra algorithm

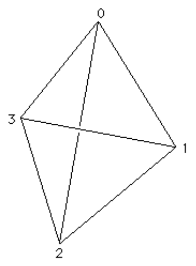


Figure 8: A tetrahedron.

A central concept in the marching tetrahedra algorithm is (not surprisingly) the *tetrahedron*. This is an object consisting of 4 vertices, 6 edges that span 4 faces (see Figure 8).

The "marching" part of the algorithm consists of iterating over all cells in the volume dataset, splitting each cell into 6 tetrahedra and generating up to 2 triangles for each tetrahedron[1]. The final isosurface is formed by collecting all the generated individual triangles into a single triangle mesh.

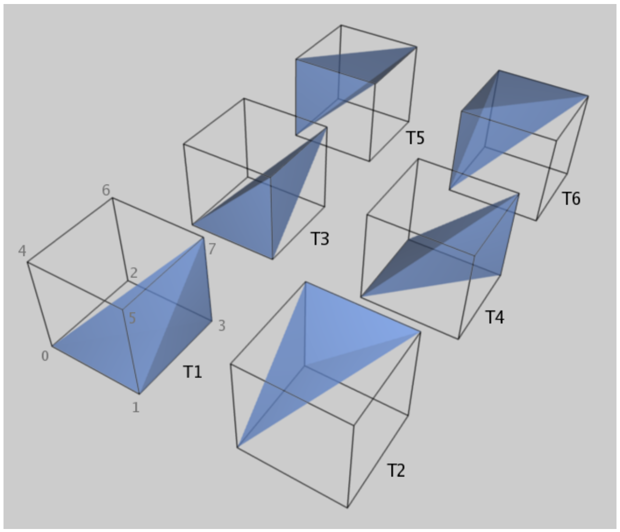


Figure 9: The 6 tetrahedra defined in a cell.

The 6 tetrahedra defined for each cell are shown in Figure 9. Note that as the cell corners correspond to volume data points, so do the vertices of each tetrahedron.

[1]There is also variations of the algorithm using 5 and 20 tetrahedra per cell.

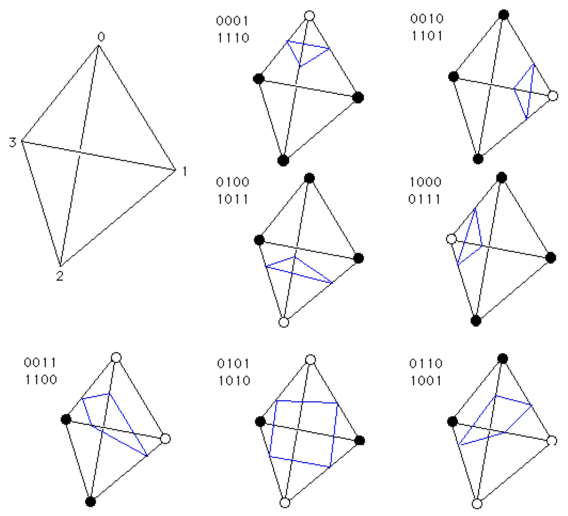


Figure 10: The different cases for a tetrahedron's vertex values, together with the generated triangles/quads. The cases 0000 and 1111, for which no triangles are generated, are not shown.

As mentioned above, for each tetrahedron either 0, 1 or 2 triangles are generated, depending on the voxel values associated with the vertices of the tetrahedron. For each tetrahedron vertex we compare the associated value against the iso-value for which we wish to extract the isosurface and associate a one-bit value with that vertex. The data value is either less than or equal to the iso-value (0), or it is greater than the iso-value (1). As there are 4 vertices in each tetrahedron, a 4-bit value B can be computed for the whole tetrahedron.

The possible bit-value cases for B are shown in Figure 10 (except for the cases 0000 and 1111). The black and white spheres represent that data values are either less-than-or-equal or greater-than the iso-value.

For every edge that has one vertex shown in black and the other in white, the isosurface intersects that edge somewhere. By determining these intersection points for all black-white edges, and by creating triangles using the intersection points (as shown in Figure 10), we can build up an isosurface.

The first four cases in Figure 10 each generate a single triangle, while the bottom three cases generate a quad (which we split into two triangles). There are two cases not shown in Figure 10: those for which all 4 vertices have the same bit-value. For these cases no triangles are generated, as the isosurface does not intersect the tetrahedron edges.

Summarizing: a high-level description of the algorithm is:

```
For each cell C in the dataset:

    For each tetrahedron T in C:
```

```
Determine bit-value B by comparing the 4 vertex data values
with the iso-value of interest;

Use B to select the appropriate case;

Compute isosurface intersection points on edges
of T (if any) and create 1 or 2 triangles using
the intersection points;
```

Note that we treat cases whose bits patterns are each other's inverses, e.g. 0110 and 1001, in the same way. It is also possible to treat them separately, with the goal of generating consistently oriented triangles with respect to their front and back side. One choice would be to always make a triangle's front face the data points with the highest values. We could then enable back-face culling on the triangles, and still get a consistent isosurface. For this assignment, you can treat the pairs of cases equally, because the triangles will be rendered without backface-culling and using the same material for front and back side. This effectively makes the isosurface two-sided.

2.3 The framework

The framework for this assignment has skeleton code to render an isosurface. It has an option for using vertex/normals arrays for rendering and provides a framerate counter. It also loads volume datasets, of course. Run the framework using the command `./main <dataset> <n>`, where `<dataset>` is the name of a dataset input file, and `<n>` is the initial iso-value.

The following keys are available in the framework application:

- `a`: toggle vertex arrays on/off,
- `b`: toggle backface culling on/off,
- `f`: toggle framerate display on/off,
- `i`: set iso-value; usage: first type the iso-value, then press `i`,
- `c`: visualize surface-vertices as cubes,
- `p`: visualize surface-vertices as points,
- `s`: visualize surface as triangle surface,
- `[`: decrease iso-value,
- `]`: increase iso-value,

The camera position can be controlled using the mouse (left button), including zoom (right button).

Assignment 2.1: the basic algorithm

Implement the marching tetrahedra algorithm, which consists of the following tasks:

- Implement `get_cell` in `volume.c`. You can use function `voxel2idx` to convert a voxel index (i, j, k) into an index of the array `volume`, which holds the volume data values.
- Implement `generate_cell_triangles` and `generate_tetrahedron_triangles` in `marching_tetrahedra.c`. For the tetrahedron cases that generate a quad, split the quad into two triangles (it is your choice how to do this).

  Use function `interpolate_points` to place intersection points (i.e. triangle vertices). As it is, this function will put intersection points halfway between tetrahedron edge vertices.
- Implement `FillArrayWithIsosurface` in `main.c`, which should iterate over the volume cells and add the resulting isosurface triangles to the vertex/normal arrays, using `AddVertexToArray`. Call the appropriate function(s) from the list above. Use flat shading (i.e. set the vertex normals to be equal to the triangle normal). Output the total number of triangles generated for the isosurface using `printf`.

Tips

- The framework comes with a number of datasets to test your code. Initially test your implementation using the smaller datasets and only when you are confident that your code works correctly, try one of the larger sets. You will not be able to deduce anything from a large jumbled mass of triangles for which you don't know what the correct output should be. The simple test cases (such as sphere) generate a predictable output and are easier to comprehend when you debug your code.
- The file `v3math.h` provides functions to work with points/vectors.
- Don't worry too much about optimizing your code. The goal here is to make you familiar with isosurface extraction, not optimization.

Assignment 2.2: interpolated intersection points

When you implement the first part of the assignment successfully, you might notice that the results are still quite "blocky". This is because we initially put intersection points halfway the tetrahedron edges, which isn't entirely accurate. Suppose that we extract an isosurface for iso-value 50 and we encounter a tetrahedron that has one vertex with value 10 and three vertices with value 60. We create one triangle in this case, with intersection points (and thus triangle vertices) halfway each of the edges running from the vertex with value 10 to the vertices with value 60. It makes much more sense to put the intersection points closer to the vertices with value 60, as the iso-value is closer to that value. If we want to extract a surface for a value of 60 instead of 50, we simply want to use the actual cell data points for the triangle vertices as these have values equal to the isovalue.

To make the isosurface match the voxel values in the volume more closely, we need to place the intersection points based on the voxel values of edge endpoints. For this, change function `interpolate_points` to use linear interpolation. Hint: use the difference in endpoint values.

Assignment 2.3: questions

Be prepared to answer the following questions when your implementation is graded:

- What happens for case 0011/1100 when two of the tetrahedron's vertex values are equal to the iso-value of interest and the other two are greater than the iso-value? Is this a bad thing? Hint: think of the kind of output generated in this case.
- We render the isosurface two-sided here, that is, without back-face culling. If the isosurface forms a closed surface, there would be no need for this, as the inside/back-side of the surface would never be visible. For what situation does marching tetrahedra *not* produce a closed isosurface?
- What optimizations to the marching tetrahedra algorithm as described in Section 2.2 can you think of? Are there computations that are performed redundantly? Is there an opportunity for parallel processing?

3 Grading

With respect to part 1: *When you send in your submission, don't forget to include the* `.obj` *files!*

You do not need to submit the `.vtk` datasets.

3.1 Part 1: texturing

You receive up to 6 points for assignment 1.1, basic texturing:

- 0.5 points for correctly setting texture coordinates;
- 0.5 points for a correct road texture;
- 0.5 points for a correct ground texture;
- 0.5 points for correctly repeating textures;
- 2 points for correctly texturing the house;
- 1.5 points for a correct skydome texture;

- 0.5 points for a correct cylindrical tree stem texture.

You receive up to 1 point for assignment 1.2: adding and enabling mip-mapping.

You receive up to 3 points for assignment 1.3: the tree leafs and their texturing.

3.2 Part 2: isosurface extraction

You receive up to :

- 0.5 point for correctly implementing `get_cell`,
- 1.5 points for `generate_cell_triangles`,
- 3 points for `generate_tetrahedron_triangles`,
- 1.5 points for `FillArrayWithIsosurface`,
- 2 point for the linear interpolation of intersection points,
- 1.5 points **total** for correct answers to the three questions.

Points may be deducted if your code is not well outlined or commented!

References

[1] W. E. Lorensen and H. E. Cline, "Marching cubes: A high resolution 3D surface construction algorithm," *SIGGRAPH Comput. Graph.*, vol. 21, pp. 163–169, Aug. 1987.