Volume Visualization using 2D Texture Mapping
Detailed Instructions

November 24, 2004

1 Introduction

You will experiment with volume visualization using texture mapping. The main advantage of this method is that it is relatively fast and relatively simple to implement. In fact, the method can be implemented using standard OpenGL on a platform that supports 2D texture mapping in hardware.

1.1 Layout and documentation

The following files are edited in this exercise.

```
TextureVolume/texsculpt.cpp
TextureVolume/TextureVolume.cpp
TextureVolume/TextureVolume.h
```

There is some documentation of the program controls here:

```
ComputerGraphics/TextureVolume/README
```

In addition `TextureVolume.h` contains extensive documentation in the form of comments. You can compile and run the program in the usual way. Pass `Teddybear.xml` as argument and hit 'm' in the program window to run marching cubes. This allows you to see a polygonal model of the volume.

1.2 Basic idea

Basically, a volume is a 3D lattice of points, where each point, called a voxel, has an associated value (the voxel value). In other words, a volume is a 3D database where each record has a key (x,y,z) and an associated value. Throughout this exercise we will assume that the voxel value is a grey–level intensity (just like in a grey–level image).

A volume can also be seen as a stack of images as illustrated in Figure 1 below. If the voxel values are grey–level intensities, we can now see the volume as a stack of grey–level images. This view should be very intuitive, and it leads to a simple method for volume rendering: We simply render the volume by compositing these images (henceforth called slices) back to front. However, some problems are apparent:

- Simply rendering slices on top of each other can only be used to generate two views, namely the two possible projections of the volume perpendicular to the slice planes.

- If each slice is completely opaque, we can only see the last slice that was rendered. Hence, the slices should be semi–transparent.

To solve the first problem, we texture map each slice onto a quadrilateral and then we render these texture mapped quads in such a way that they make up a sliced volume. This is illustrated both in Figure 1 and 2. As long as the slices are more or less parallel to the image plane this works well, but if we see the slices edge on, the result is awful. Therefore, the method requires us to keep three separate stacks of slices, one for each major axis. We then select and render the slice stack that is most perpendicular to the eye direction as illustrated in Figure 2.

To solve the second problem, we use alpha–compositing. To make things really simple, we simply set both colour and alpha equal to the intensity. Hence, for each pixel of each slice:

$$(R_s, G_s, B_s, A_s) = (I_t, I_t, I_t, I_t) \tag{1}$$

where subscript t indicates the texture value and s indicates the colour that results from mapping the texture onto geometry. In other words, the intensity values should simply replace the geometry colour. `glTexEnvi` is used to select this behaviour.

The slices are now rendered on top of each other so that

$$(R, G, B) = (R_s, G_s, B_s)(A_s) + (R, G, B)(1 - A_s) \tag{2}$$

In other words, the framebuffer colour value is equal to the colour value of the texture mapped polygon times its own alpha plus the original colour times one minus the texture alpha. `glBlendFunc` can be used to select this behaviour. We also want to eliminate noise. This is done by using the alpha test to reject fragments whose alpha values are beneath a given threshold.

The method works well on any hardware which implements 2D texture mapping in hardware, but it does have many drawbacks. The quality is not as good as when using other methods, and it is quite easy to see when the slicing direction changes. Furthermore, there is no real shading. All of these problems have been addressed in literature by using recent sophisticated hardware (mostly on SGIs or GeForce3) to either fill in more slices or render the slices in a more sophisticated way. 3D texture mapping can be used to implement a method where slices are always parallel to the image plane. These things are beyond the scope of this exercise, however, and generally harder to implement.

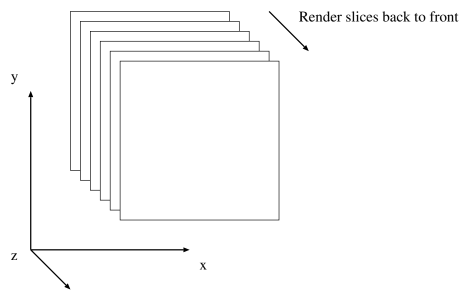

Figure 1: A volume can be represented by a set of slices

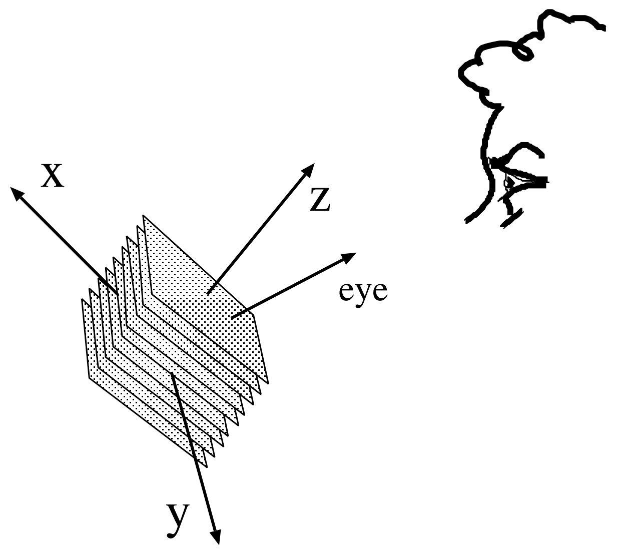

Figure 2: eye vector determines which slice direction is chosen.

2 Implementation: Texture mapping and blending

In this part of the exercise, you set up texture parameters and fill in the code to render a volume using texture mapping.

2.1 OpenGL State and Texture Name Generation

First we need to fill in the contents of

```
void TextureVolume::gl_init()
```

which is used to set up the necessary OpenGL state. You need to do the following:

- Enable 2D texture mapping, alpha test, blending, and depth test.
- Generate texture names (using `glGenTextures`) for all slices along all directions. The texture names should be stored in the vectors x_slice_textures, y_slice_textures, and z_slice_textures. To get a pointer to the first element of such a vector use e.g.

  ```
  &x_slice_textures[0]
  ```

- Set up the correct OpenGL texture mapping and the correct blending:

  ```
  glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, xxx);
  glBlendFunc(xxx, xxx);
  ```

  where xxx's are replaced with appropriate arguments.

2.2 Binding images to textures

For this question, you should modify

```
void TextureVolume::gl_bind_textures()
```

In `gl_init`, we generated texture names for the texture slices. Now, it is time to associate texture images with each slice. For each slice, we first bind the texture and then associate the texture image using a call to `glTexImage2D`. Calls to `glTexImage2D` should have the following form:

```
glTexImage2D(GL_TEXTURE_2D,
             0,
             GL_INTENSITY,
             SLICES,XSIZE,
             0,
             GL_RED,
             GL_UNSIGNED_BYTE,
             (const void*) &y_data[i].get(0,0));
```

Here `SLICES` and `XSIZE` are the width and height of the texture. `&y_data[i].get(0,0)` gets the address of the lower left pixel of slice `i`. For slices in the X and Y directions, you should replace these three arguments with the appropriate values.

For each slice, we also need to set up magnification and minification filters and values for GL_TEXTURE_WRAP_S and GL_TEXTURE_WRAP_T. You can experiment with settings for the mini– and magnification filters, but is important that you set wrapping to CLAMP. These parameters are controlled using `glTexParameteri`. If you forget to set up some of these parameters things will not work!

2.3 Rendering textures

Now it is time to render the textures. You should implement

```
void TextureVolume::gl_draw_stack(const Vec3f& eye, float iso);
```

The first chore is to set the alpha function so that we only write to the frame buffer if alpha is greater than iso. Remember that your iso value is in the range 0 to 255 while `glAlphaFunc` takes an argument between 0 and 1.

Next you must determine which set of slices to draw. `eye` is a vector that points from the centre of the volume towards the eye. We pick the stack whose slices are most perpendicular to `eye`. We also need to decide in which order to draw the slices. Figure out how to do this.

Having decided which slices to draw and in which order, it is time to actually draw the slices. Each slice is drawn as a single texture mapped quad. Remember to bind the texture before drawing and to assign texture coordinates to the vertices of the quad. To get correct scaling, use the vector `slice_scale` which contains the x, y, and z distances between two consecutive slices.

To make the program simpler and more readable, you can put the code for drawing a single slice in the `draw_x_slice`, `draw_y_slice` and `draw_z_slice` functions. These are inline functions, so they reside in `TextureVolume.h`

3 Part II: Volume editing

3.1 zpick

It is frequently useful to find the world coordinates corresponding to a pixel on the screen. Of course, the pixel has only a 2D location, but if we use the z–buffer value of the pixel, each pixel is in fact a 3D point. We can use the OpenGL utility toolkit function `gluUnproject` to unproject the point thus obtaining the world position of a rendered point on the screen.

Your task is to implement the function

```
bool zpick(const Vec2i& p, Vec3f& wp);
```

in `texsculpt.cpp`. The first parameter, `p`, is the 2D point whose corresponding z–value we want to obtain. The second parameter, `wp`, should be assigned the world coordinate corresponding to `p`.

The function should perform the following steps

- Obtain the z value of the pixel at position p in the framebuffer. Use `glReadPixels` to read from the framebuffer. Important: `glReadpixels` assumes that the origin of the screen x,y coordinate system is the lower left corner. The origin of `p` (as returned by glut) is the *top* left corner. You have to get the viewport size (use `glGetIntegerv`) to fix this.
- If the z value corresponds to the far clipping plane (use `glGetFloatv`) to obtain the depth range, the function should return false.
- You will now have to obtain the projection and modelview matrices. This is easily done using `glGetDoublev`, but recall that you need to set up the modelview matrix – this can be done by calling

  ```
  ball->set_gl_modelview();
  ```
- call `gluUnProject` to get the world coordinate point `wp`. Like `glReadPixels`, `gluUnProject` assumes that the lower left corner is the origin.
- Assign to `wp` the world point corresponding to `p`, and return true.

3.2 Manipulation

The code goes in

```
void  TextureVolume::manipulate(const Vec3f p, int N, Manip man)
```

where the first parameter is the point we wish to manipulate and the second parameter is the size of the tool. The final parameter indicates what manipulation should be performed.

Let pi be the integer part of p. Now the range we are interested in is (p[0]-N, p[1]-N, p[2]-N) to (p[0]+N, p[1]+N, p[2]+N). For each voxel position in that range, we perform a simple operation depending on the value of the third parameter man:

- PERLIN_ADD

$$V \leftarrow V + T - VT \tag{3}$$

- PERLIN_SUB
$$V \leftarrow V - VT \tag{4}$$
- MAX_ADD
$$V \leftarrow \max(V, T) \tag{5}$$
- MIN_SUB
$$V \leftarrow \min(V, 1 - T) \tag{6}$$
- ERASE
$$V \leftarrow 0 \tag{7}$$

where T is the value of the tool at the voxel position, and V is the value of the voxel. Before performing this operation you should scale the values of T and V to the range $[0, 1]$ and then scale back before you set the value.

The value of T is

```
T = max(0, 1-||x-p||/N);
```

where x is the voxel position.

Remember to scale V back to the range $[0, 255]$ and that you can use

```
bool TextureVolume::point_in_vol(const Vec3i& p)
```

to test if a point is inside or outside the volume. If you do not use this function your program will crash. Use

```
void TextureVolume::set(const CGLA::Vec3i& p, Byte val)
```

to write to the volume. This function will update all three sets of slices. Call `gl_bind_textures` when you have changed the volume so that the changed volume is actually used.

jab '04