CS 465 Homework 2b

out: Monday 17 November 2003
due: Wednesday 26 September 2003

In this assignment, you will implement several types of shading in a simple software graphics pipeline. In terms of the graphics pipeline stages we discussed in lecture, we are giving you the application and rasterizer stages of the pipeline, and your job is to implement the vertex and fragment processing stages to achieve several different kinds of shading. This is very much like the task you are faced with when using a modern programmable graphics processor such as the ones that power current high-end PC graphics boards.

This assignment is to be done individually.

Principle of operation

As discussed in lecture, the *graphics pipeline* is a sequence of processing stages that efficiently transforms a set of 3D *primitives* into a shaded rendering from a particular camera. The major stages of the pipeline are:

- Application: holds the scene being rendered in some appropriate data structure, and send a series of primitives (only triangles, in our case) to the pipeline for rendering.
- Vertex processing: transforms the primitives into screen space, optionally doing other processing, such as lighting, along the way. In our pipeline this stage is known as "triangle processing" because the only primitive type is separate triangles.
- Rasterization: takes the screen-space triangles resulting from vertex processing and generates a *fragment* for every pixel that's covered by each triangle. Also interpolates parameter values, such as colors, normals, and texture coordinates, given by the vertex processing stage to create smoothly varying parameter values for the fragments. Depending on the design of the rasterizer, it may *clip* the primitives to the view volume. Our simple rasterizer cannot handle triangles that cross the view plane, so in our pipeline the rasterizer does near-plane clipping.
- Fragment processing: processes the fragments to determine the final color for each

one and to do z buffering for hidden surface removal. Writes the results to the *framebuffer*.

- Display: displays the contents of the framebuffer where the user can see them.

For each of the types of rendering detailed in the requirements below, you need to implement a triangle processor and a fragment processor by implementing subclasses of the triangle processor and fragment processor base classes. The triangle processor takes as input three vertices with colors, normals, and texture coordinates, and it hands the triangle off to the rasterizer in the form of three screen-space vertices and three *attribute arrays* that contain the parameters to be interpolated and given to the fragment processor. The fragment processor takes as input an integer (x, y) pixel coordinate and an attribute array, and after doing the appropriate computations it sets pixels in the framebuffer as appropriate.

The attribute arrays are the means of communication between the vertex and fragment programs, and the two stages need to agree on how many attributes there are and what they mean. When the user chooses the two programs, the framework enforces agreement on the number of attributes, but the semantics are up to you. Note that our rasterizer does perspective-correct interpolation of all attributes.

The pipeline contains three transformation matrices: the Modelview matrix, which is the product of the modeling and viewing matrices we discussed in lecture, the Projection matrix, and the Viewport matrix. You'll use the Modelview matrix to transform the input coordinates (object space) to eye-space coordinates. An important feature of the pipeline is that it only allows rotations and translations in the Modelview matrix. This means that you can transform normals using the same matrix you use to transform vectors, which is a nice convenience.

Our software graphics pipeline is designed for simplicity rather than efficiency, though it does run acceptably fast for small scenes like the ones provided by the framework. The time to render a frame is heavily dominated by fragment processing (which is typical of software pipelines), so when you implement your fragment programs make every statement count! In particular, it would be a bad idea to allocate objects in the fragment program or to use unnecessary calls to the math library.

Requirements

Implement vertex and fragment programs that provide the following kinds of shading, all with hidden surface removal. The framework comes with an implementation of constant shading with no z-buffer, just to get you started.

1. Constant shading: each triangle is rendered entirely with the color of its first vertex, and the other colors, the normal, and the texture coordinates are ignored.

2. Textured constant shading: each triangle is rendered with the color of its first vertex multiplied by the current texture. The texture coordinates are used to look up in the texture. The other colors and the normals are ignored.

3. Flat shading: each triangle is rendered with a single color computed using the Blinn-Phong lighting model with the color and normal of the first vertex. The other colors and normals and the texture coordinates are ignored.

4. Gouraud shading: each triangle is rendered with colors interpolated from colors computed at the vertices. The color at each vertex is computed using the Blinn-Phong lighting model with the color and normal of that vertex. The texture coordinates are ignored.

5. Phong shading: each triangle is rendered with colors computed using the Blinn-Phong lighting model with the color and normal interpolated from the colors and normals of the vertices. The texture coordinates are ignored.

6. Textured Phong shading: each triangle is rendered with colors computed using the Blinn-Phong lighting model with the color from the current texture map and the normal interpolated from the colors and normals of the vertices. The vertex colors are ignored (unlike in the texture-modulated constant shading).

In all cases where you are computing lighting, the lighting model is the same Blinn-Phong model as used in the ray tracer. The diffuse and ambient colors are equal, and they come from the vertex color or texture map. The specular color is a constant. There is a single directional light source, and you should use the infinite viewer approximation[1].

Framework code

The framework is a simple graphics pipeline that is modeled on the way hardware graphics pipelines work (rather than being an example of the most efficient way to write a software pipeline). The classes that are most important for this assignment are:

1. `Pipeline` coordinates the operation of the pipeline and provides the interface to the Scene classes that draw the test scenes. It contains references to all the other pieces of the pipeline: the triangle processor, the rasterizer, the fragment processor, and the framebuffer. It is here that you will find the current transformations and the lighting parameters for the Phong model.

2. `TriangleProcessor` holds the code to do triangle (vertex) processing. The main function is `triangle`, which is called to render a triangle. It takes four arrays as arguments, each of which should be three elements long: the three vertex positions,

[1] This means that in eye coordinates the direction toward the eye is always $+z$.

the vertex colors, the vertex normals, and the texture coordinates. Not every triangle processor will use all the arguments (in fact, none will use every single argument).

3. `Rasterizer` contains the algorithms for clipping and rasterization. The `rasterize` method is the entry point, which is called by the triangle processor for each triangle. It clips the triangle against the near plane, which results in zero, one, or two triangles that need to be rasterized. For each triangle it calls `rasterizeClipped`, which does the actual work of rasterization. For every fragment generated, the rasterizer calls the `fragment` method of the fragment processor. You'll need to understand the rasterization code for the discussion questions, but you can safely ignore the clipping code, which looks hairy (although it really is straightforward—it just has a lot of cases).

   Note: The rasterizer adds the z' (or screen-space z) depth as the first parameter in the parameter array it hands to the fragment processor. This means that all the other attributes are shifted by one place. For instance, `ConstColorTP` puts the RGB color into attributes 0, 1, and 2, but `TrivialColorFP` has to take them from attributes 1, 2, and 3 because the z' value is now attribute 0.

4. `FragmentProcessor` holds the code for fragment processing. The main function is `fragment`, which is called for every fragment (and therefore needs to be efficient). The arguments to `fragment` are the coordinates of the pixel it addresses and the attribute values, which are interpolated from the corresponding values supplied by the triangle processor.

5. `FrameBuffer` is a simple class to store the final image. It stores the color channels as a byte array (three bytes per pixel) and the z buffer as a float array (one float per pixel). The fragment processor can read the z buffer using the `getZ` method, and it can write to all the channels using the `set` method. The image actually gets on the screen by being drawn in `PipeView.display`.

You'll find that there are subclasses of `TriangleProcessor` and `FragmentProcessor` already there. To let you know what we have in mind, here are the combinations of triangle and fragment processor we expect you to use to implement each shading mode:

- Constant shading: `ConstColorTP` and `ColorZBufferFP`
- Textured constant shading: `TexturedTP` and `TexturedFP`
- Flat shading: `FlatShadedTP` and `ColorZBufferFP`
- Gouraud shading: `SmoothShadedTP` and `ColorZBufferFP`
- Phong shading: `FragmentShadedTP` and `PhongShadedFP`
- Textured Phong shading: `TexturedFragmentShadedTP` and `TexturedPhongFP`

The classes `MainFrame`, `GLView`, and `PipeView` are concerned with the user interface. The program comes up with a single window that shows you two viewports looking at the same scene. The left one is renderd by our software pipeline with your triangle and fragment processing code, and the one on the right is rendered by OpenGL using your PC's graphics hardware (or a software emulator in the OS if the hardware is absent or mis-configured). There are several pop-up menus across the bottom. The first two let you choose the active triangle and fragment processor. The third one lets you choose between several simple test scenes; the fourth one lets you choose among several textures; and the last one lets you choose between two ways to control the camera.

The OpenGL viewport configures itself based on the classes that are selected in the first two menus. Its behavior closely approximates what you should see in the software window (but only for valid combinations of triangle and fragment processors). Note that, since OpenGL doesn't easily support Phong shading, the renderings will not match when you select the Phong shading modes – your highlight will look much nicer. Even for the valid modes the two images will not be pixel-for-pixel identical, because the OpenGL standard purposely leaves many implementation details unspecified.

The classes `Camera`, `Geometry`, and `Scene` and its subclasses make up the application code that feeds triangles into the pipeline. The different scenes are:

- "Balls": a scene consisting of two spheres, represented by two different numbers of triangles. Their vertices share normals, so this scene exercises smooth shading (all the other scenes have flat faces). The texture coordinates are just the x and y coordinates of the vertex position. Most useful for testing the Flat, Gouraud, and two Phong shading modes.
- "Cube": a scene consisting of a cube with faces of different colors. This scene is most useful for testing the two texture mapping modes.
- "ship1.msh" and "ship2.msh": triangle meshes that are read in from files. They have texture coordinates that also come from the files, and each model is designed to look right with the correspondingly named texture selected. They are useful for testing flat shading and the textured modes.
- "Maze": a randomly generated maze. This is the one model that's meant to be used with the "Flythrough" camera mode. The shaded modes don't produce very nice looking results with this scene the way the lights are set up, so this is best viewed with the textured constant shading mode.

To control the camera in the "Orbit Camera" mode, click and drag in the window to rotate the model, and shift-click and drag to move the camera closer or farther away. In the "Flythrough" mode, click and drag to rotate the camera in place, and shift-click and hold to move forward. You can steer while moving forward by moving the mouse around.

The classes `Matrix4f` and `Texture` are utility classes that you'll need to use. `Matrix4f` (not to be confused with `javax.vecmath.Matrix4f`) is a very simple 4x4 matrix class. It does not even have inverse or transpose operations, because we don't need them for this assignment. You will only need to use the `*Multiply` and `*Compose` operations. `Texture` is the (extremely simple) class that reads in a texture image and stores it in an array. To look up the color at a particular texture coordinate, use the `sample` method, which accepts a two-dimensional point (u, v) in the unit square $[0, 1] \times [0, 1]$.

Discussion questions

1. The code in `Rasterizer.rasterize` is lacking in comments. Add explanatory documentation to the code that answers the following questions.
   (a) Which part of the routine is triangle setup and which part is the rasterization loop?
   (b) Where is back-face culling?
   (c) Where is the perspective division?
   (d) Where is perspective correction for the attributes being done? Where is the extra pseudo-attribute that is needed for perspective correction?
   (e) Does this code work by interpolating barycentric coordinates or by interpolating edge equations?
2. The rasterizer does not do far plane clipping (it only clips to the near plane). Do you nonetheless see objects clipped at the far plane? Why is this?

Handing in

Hand in in the usual way via CMS: a flat `.zip` file with the Java source files only—particularly, no `.class` files. If you did extra credit, include any additional data files that are needed. Don't forget to include a version of `Rasterizer.java` with the comments requested in the discussion questions.

Extra credit

All kinds of clever pipeline rendering tricks are accessible directly from this framework. Some examples:

1. Reflecitons from planar surfaces using two-pass rendering. You can simulate mirror-like reflections in a flat surface by making a rendering of your scene from a viewpoint

reflected across the plane, reading that image back from the framebuffer, and then using it as a texture to draw the plane from the normal viewpoint. Foley et al. has a discussion of this technique.

2. Environment mapping. Read up on environment mapping in Shirley or Foley et al., then implement a fragment processor that renders a shiny metal material by generating texture coordinates to index into a cubemap. You can find some cubemap images at `http://www.debevec.org/Probes/`.

If you do extra credit, you will probably want to extend or modify the scenes we have provided, in order to demonstrate your extra feature.

We recommend talking to us about your proposed extra credit first so we can steer you towards interesting mappings and make sure we agree that it would be worth extra credit.

Let us re-emphasize that, as always, extra credit is only for programs that correctly implement the basic requirements.