CS 4620 Program 3: Pipeline

out: Saturday 18 October 2008
due: Tuesday 28 October 2008

1 Introduction

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.

2 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 sends 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.
- 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 we have provided a clipper to cut triangles at the near plane.
- Fragment processing: processes the fragments to determine the final color for each, to perform z-buffering for hidden surface removal, and to write the results to the *framebuffer*.
- Display: displays the contents of the framebuffer where the user can see them.

For each of rendering methods detailed below, you will implement a vertex processor and a fragment processor as subclasses of the `VertexProcessor` and `FragmentProcessor` base classes. The vertex processor's input is a vertex's position, color, normal, and texture coordinates. It returns a processed `Vertex` to the pipeline. Each `Vertex` will contain a screen space vertex position and an *attribute array* containing its parameters. They are given to the rasterizer which produces fragments whose data is interpolated from the vertex attributes. The fragment processor takes as input a `Fragment` which contains 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.

The pipeline contains three transformation matrices: the Modelview matrix, which is the product of the modelling and viewing matrices we discussed in lecture, the Projection matrix, and the Viewport matrix. You'll use the Modelview matrix to transform the input triangle data in 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, a nice convenience.

Our software graphics pipeline cannot match the performance of dedicated hardware; though for small scenes, like those in this assignment, it can acheive interactive performance. However, the time to render a frame is largely *pixel bound* meaning most of the time is spent in fragment processing. You should take care to implement your fragment programs as efficiently as possible. Make every statement count! In particular, your performance will be seriously compromised if you allocate objects in the fragment program or to use many calls to the `Math` library.

3 Requirements

Implement vertex and fragment programs to provide the following kinds of shading with hidden surface removal and support for multiple light sources.

1. Smooth Shading: each triangle is rendered with colors interpolated from the vertices. The color at each vertex is computed using the Blinn-Phong lighting model using the color and normal of that vertex.

2. Texture-modulated Smooth Shading: each triangle is shaded with the interpolated color from per-vertex shading multiplied by the current texture. The texture coordinates for each vertex are used to index into the texture.

3. Fragment Shading: each triangle is rendered using the Blinn-Phong lighting model, but the shading calculation is now done for each fragment with interpolated normals, colors, and eye-space positions from the vertices.

4. Textured Fragment Shading: per-fragment Blinn-Phong shading, but use the color from the current texture map. The vertex colors are ignored (unlike in the texture-modulated smooth shading).

*5. Reflection mapping: meshes are rendered to appear shiny by using the interpolated normals to compute a reflection vector and look up in a spherical environment map. Like the lights, the environment map should be fixed in eye space.

*6. Texture interpolation: implement bilinear interpolation for texture magnification. This is most easily implemented by modifying the `Texture` class rather than by implementing new fragment processors.

For this assignment, the Blinn-Phong lighting model is the same as the model used in the ray tracing assignment (except assume that the light's specular component is full white). The diffuse color comes from either the vertex colors or texture map. The specular color and exponent, and information about light sources (which are diffuse-colored point lights with positions in eye-space), are stored in the `Pipeline` class. You should implement local lighting: the direction to the viewer and to light sources vary across each triangle. You must also include the ambient term (which is also found in `Pipeline`) in all your shading.

4 Framework code

The framework is a simple graphics pipeline that is modelled on the way hardware graphics pipelines work. For this assignment, you should not need to modify any code outside of the triangle and fragment programs you will write. However, you will read or write data from several of the other classes.

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 pipeline stages: 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. Like OpenGL, primitives are rendered by using `Pipeline.begin` to indicate what type of primitive will be given, followed by a sequence of calls to `Pipeline.vertex` to give the vertices, then a call to `Pipeline.end`. All primitives are converted into triangles.

2. `VertexProcessor` holds the code to perform vertex processing. Your vertex processors will be derived from this class. Your will have to write implementations of the `vertex()` method in each sub-class. This method is called when rendering each vertex and it takes the vertex position, the vertex color, the vertex normal, and the texture coordinates as arguments. Not every vertex processor will use all the arguments.

   There are also two extra methods in a `VertexProcessor`: `updateLightModel` and `updateTransforms`. The first is called by the framework whenever the lighting parameters (light position, intensity, etc) change. The second is called whenever the modelview, projection, or viewport matrices change. These calls allow you to store the current transformation matrices or do some additional pre-computations in your sub-classes to be used in later computations.

3. `Clipper` contains the algorithms for clipping before rasterization. The `clip` method is the entry point, called by the pipeline for each triangle. It clips the triangle against the near plane, which results in zero, one, or two triangles that need to be rasterized.

4. `Rasterizer` will contains the algorithm for rasterization. The `rasterize` method is the entry point called by the pipeline for each clipped triangle. The rasterizer outputs a list of fragments that are sent to the fragment processor.

5. `FragmentProcessor` holds the code for fragment processing. You will implement subclasses of this class to perform fragment processing. The main function is `fragment` and is called for every fragment (and therefore needs to be efficient). The arguments to `fragment` are a `Fragment` and the `Framebuffer`. The fragment structure stores coordinates of the pixel it addresses and the attribute values interpolated from the triangle vertices by the rasterizer. When the fragment program is done processing the fragment, the resultant color (if visible) should be set in the `FrameBuffer`.

   The current texture map is also set in `FragmentProcessor.texture`.

6. `FrameBuffer` stores 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 is drawn in `PipeView.display()` method.

7. `javax.vecmath.*` is Sun's Java vector math library. You can find the API for vecmath at

   `http://java.sun.com/products/java-media/3D/forDevelopers/J3D_1_3_API/j3dapi/index.html`

8. `Matrix4f` (not to be confused with `javax.vecmath.Matrix4f`) is a very simplified 4x4 matrix class for this assignment. You will only need to use the `*Multiply` and `*Compose` operations.

9. `Texture` handles texture objects. 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 $[0, 1] \times [0, 1]$.

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

    - "Simple Triangle": a scene with a single triangle. The color varies across its surface from full red to full green to full blue at each vertex. The texture coordinates vary in a similar, but not identical fashion.
    - "Balls": a scene consisting of two spheres with spherical normals. The texture coordinates are the x and y coordinates of the vertex position. These will be useful in debugging Phong shading, since the specular highlights are clearly noticeable.
    - "Cube": a scene consisting of a cube with faces of different colors. The texture coordinates vary from $(0, 0)$ to $(1, 1)$ across each face.
    - "ship1.msh" and "ship2.msh": triangle meshes that are read in from files. The stored meshes have texture coordinates specific to textures that match the ship designs called "ship1.png" and "ship2.png."
    - "* (Smooth)": The same meshes, except the normals are averaged for each vertex. This will make them look smoother for Phong shading and reflection mapping.
    - "bunny500.msh (Smooth)": Another mesh that's more organic than the ships - primarily to show off reflection mapping. Note that this mesh does not have texture coordinates, so normal texture mapping won't work for it.

- "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; this is best viewed with the textured smooth 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.

11. `MainFrame`, `GLView`, and `PipeView` are concerned with the user interface. `MainFrame` contains the main method of this assignment and will initialize the GUI. The program comes up with a single window that shows you two viewports containing the same scene. The left one is rendered by our software pipeline and the right one is rendered by OpenGL using your PC's graphics hardware. There are several drop down boxes 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 (the last 3 textures are spherical environment maps, meant for reflection mapping); 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. It configures OpenGL to match the behavior expected from your code (but only for valid combinations of triangle and fragment processors) as closely as possible. Because it uses the fixed-function shading in OpenGL, which is not capable of many of the shading methods you'll be implementing, it will match your code exactly (though not actually pixel-for-pixel) only for the smooth-shaded modes.

Also included are several stub classes for the triangle and fragment shaders that you will need to implement for this assignment. We expect that each of the shading modes described above will be implemented by the following pairs of shading processors:

- Smooth shading: `SmoothShadedVP` and `ColorZBufferFP`
- Textured smooth shading: `TexturedSmoothShadedVP` and `TexturedFP`
- Fragment shading: `FragmentShadedVP` and `PhongShadedFP`
- Textured Fragment shading: `TexturedFragmentShadedVP` and `TexturedPhongFP`
- * Reflection mapping: `FragmentShadedVP` and `ReflectionMapFP`

To get you started, we've provided `ConstColorVP` and `TrivialColorFP` as very basic shaders, so make sure you understand them. A good first step is probably to implement `ColorZBufferFP` and test it using `ConstColorVP` with the two-spheres scene.

5 Handing In

Hand in in the usual way via CMS. Include a .zip file of the entire `pipeline` package tree. If you did extra credit, include any additional data files that are needed.

6 Extra Credit

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

1. Better texture filtering. With some additional information passed from the rasterizer, it is possible to implement MIP mapping or anisotropic texture filtering fairly easily in this pipeline.
2. Fancier shading. With some modifications to the scenes, you could implement bump mapping, normal mapping, displacement mapping, or other mechanisms for adding details to surfaces.
3. Efficient rasterization. Rather than going over the entire bounding box of a triangle, there are far more efficient ways to rasterize triangles. The resulting code is often messy, though, so make sure you document what you're doing as well as any sources for the algorithm you're using.

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.