CS 465 Program 3: Pipeline

(revised October 3, 2005)

out: 3 October 2005
due: 18 October 2005

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. In our pipeline this stage is known as "triangle processing" because the only primitives are 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. Your simple rasterizer will not be able to handle triangles that cross the view plane, so we have provided a `Clipper` which runs just before the rasterization.
- 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 triangle processor and a fragment processor as subclasses of the `TriangleProcessor` and `FragmentProcessor` base classes. The triangle processor input is three vertices, colors, normals, and texture coordinates. It returns a list of `Vertex`s to the pipeline. Each `Vertex` will contain a screen space vertex 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 is 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 (which is also typical of hardware pipelines). 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 (which in Java is very inefficient).

3 Requirements

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

1. Constant Shading: each triangle is shaded the color of a single vertex. To match the hardware defaults, this is the third vertex. All other data, the colors, the normal, and the texture coordinates, are ignored.

2. Textured Shading: each triangle is shaded with the color of a single vertex (third to match hardware) multiplied by the current texture. The texture coordinates for each vertex are used to index into 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 single (third) vertex. The other colors and normals and the texture coordinates are ignored.

4. Gouraud 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. The texture coordinates are ignored.

5. Phong Shading: each triangle is rendered using the Blinn-Phong lighting model but the colors and the normals interpolated from normals of the vertices and the model is calculated at each fragment. The texture coordinates are ignored.

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

For this assignment, the Blinn-Phong lighting model is the same as the model used in the ray tracering assignment. The diffuse color comes from either the vertex colors or texture map. The specular color, exponent and lighting direction are all constants stored statically in the `Pipeline` class. Finally, you should use the *infinite viewer approximation* for the lighting direction. This means that in eye coordinates the direction toward the eye is always $+z$.

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. The `renderTriangle()` method ties everything together.

2. `TriangleProcessor` holds the code to perform triangle processing. Your triangle processors will be sub-classes of this class. Your will have to write implementations of the `triangle()` method in each sub-class. This method is called when rendering each triangle and it takes four arrays as arguments, each of which should be three elements long: the three vertex positions, 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).

   There are also two extra methods in a `TriangleProcessor`: `updateLightModel` and `updateTransforms`. The first is called by the framework whenever the lighting parameters (light direction, 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 lighting parameters 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`.

   `FragmentProcessor` also contains a method to update the lighting model just like the `TriangleProcessor` class. Note that it does not need a method to update the transformations. Why?

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. It is external to your code and is included in the vecmath.jar file in the framework zip. Most of the mathematics calls are similar to the methods in `Vector3` and `Point3` from earlier assignments, but many are slightly different. You should look over 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. `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 $[0, 1] \times [0, 1]$.

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

   - "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.
   - "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."
   - "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 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.

10. `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; 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). However, OpenGL does not support normal interpolation (required for the Phong shading modes) without using *pixel shaders* that are very similar to what you are writing for the software pipeline. Unfortunately, the standards for pixel shaders are hardware specific and we chose not to include them fearing framework code that would crash on someone's hardware. This means that for all the modes but the Phong ones, your code should match the OpenGL window perfectly (watch that you use the third vertex when applicable). In the Phong modes, the OpenGL window will continue to use Gouraud shading which will look blocky compared to the smooth fragment shading produced by your code.

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:

- 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`

5 Setup

This project will make use of a few external libraries in addition to the framework code. We encourage you to try to setup and run the framework code early so that you can work out any diffculties before the deadline approaches. This section will explain how to setup Eclipse to properly compile and run the assignment. If you have not been using Eclipse for your programming, we

strongly recommend that you do so for this assignment[1]. Its free and can be downloaded from `http://www.eclipse.org/`.

5.1 Creating the project

1. Unzip the framework bundle into the root directory of the Eclipse workspace. By default, this is a directory called `workspace` under the Eclipse installation directory. This will create a new directory called "PA3".
2. Open Eclipse and select "File >New >Project"
3. Select "Java Project"
4. Name the project exactly "PA3" and click "Finish." The project should now be created in your workspace. The source code is all in a project sub-directory called src.

5.2 Creating Run profile

1. Open MainFrame.java in the pipeline package.
2. Select "Run >Run..."
3. Double click on the "Java Application" entry in the left hand box and a new Run configuration is created automatically.
4. Select the "Arguments" tab on the right
5. In the "VM arguments" box enter exactly this string: -Djava.library.path="./dlls/" including the quotes.
6. Click "Run". The project should now execute and the main GUI window should open. This run configuration is saved and you can use it to run to the program from now on.

6 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.

7 Extra Credit

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

[1] If you wish to use another IDE or environment, you are free to do so if you understand how to correctly configure the external libraries. The course staff will try to help you if they can, but you will ultimately be responsible for configuring environments other than Eclipse.

1. Reflections 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 cube-map. You can find some cube-map images at `http://www.debevec.org/Probes/`.

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.