CS 4620 PA2: Pipeline

out: 30 September 2011
due: 7 October 2011

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 with the rasterization stage implemented, 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 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 and normals, 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.

We have provided the full rasterization process, and the pixel-increments work as follows. We calculate the $V_{x_{increment}}$ and $V_{y_{increment}}$ for each value, V, to be interpolated from the vertices to

the fragments. The problem can be stated as: given the position and values at each vertex, find the amount to increment the value in both the x and y directions which will satisfy the following equations:

$$P_0 = (x_0, y_0)$$
$$P_1 = (x_1, y_1)$$
$$P_2 = (x_2, y_2)$$
$$V_{p_1} - V_{p_0} = V_{x_{increment}}(x_1 - x_0) + V_{y_{increment}}(y_1 - y_0)$$
$$V_{p_2} - V_{p_0} = V_{x_{increment}}(x_2 - x_0) + V_{y_{increment}}(y_2 - y_0)$$

The following diagram should help you visualize. And the implementation of this process is avail-

$$G(x, y) = V_{x_{\text{inc}}}(x - x_0) + V_{y_{\text{inc}}}(y - x_0) + V_0$$

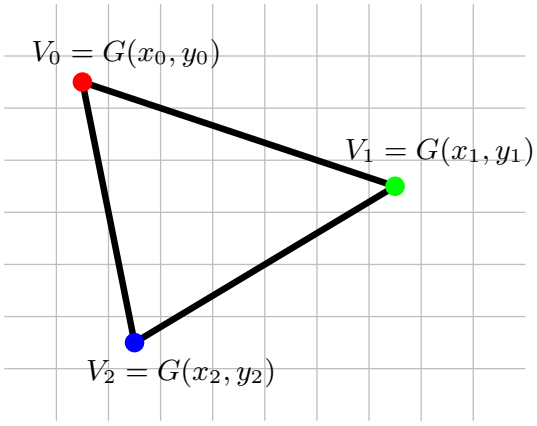

Figure 1: **Rasterization Increments**

able at `calculateInc()` in the `TriangleProperties` class.

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, and normal. 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 values specified between the Vertex and Fragment object are the means of communication between the vertex and fragment processors. Since not every vertex processor will give all the output attributes, and not every fragment processor will consume all the input attributes, not all vertex shaders will be compatible with all fragment shaders. Thus, we have included a compatibility index to specify whether a vertex processor and fragment processor will be compatible with each other. It is up to the user (you) to decide how to use all the attributes passed between the vertex and fragment processors. But here is a list of all the attributes along with guidelines about how to use them:

- Vertex : the 4D homogenous coordinate location (after applying perspective transformations) of the vertex. This value will not be interpolated for the fragment processor to consume, but will instead be used to rasterize the triangle.

- Normal : the normal at each vertex of the triangle which will be interpolated along the fragments of the triangle.
- Color : the color at each vertex of the triangle which will be interpolated along the fragments of the triangle. (Can be used to transport color sent to the vertex shader, or lighting calculations calculated on a per-vertex basis)
- Eye space position : the eye space position of each vertex of the triangle which will be interpolated along the fragments of the triangle. (Can be used to transport fragment eye-space coordinates to do per-pixel lighting)

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 achieve 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 Requirement overview

You will 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. 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.

For this assignment, the Blinn-Phong lighting model is as described in the textbook (except assume that the light's specular component is full white). The diffuse color comes from the vertex colors. 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 vertex 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, and the vertex normal 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` 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. You will not need to edit this class.

5. `TriangleProperties` contains the logic for cull testing the triangle and calculating the increments. The `initializeAndCullTest` method is called when the post-Vertex-processed triangle is sent to the rasterizer once. This method returns true if the triangle is to be rejected and false for acceptance. The `calculateInc` method is called for each value that is to be interpolated across the vertices. This method outputs the $[V_{x_{\text{increment}}}, V_{y_{\text{increment}}}]$ in the `output parameter`.

6. `FragmentProcessor` holds the code for fragment processing. You will implement sub-classes 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`.

7. `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.

8. `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`

9. `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.

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.
    - "Balls": a scene consisting of two spheres (with smooth normals). You can use it to debug your Phong shading, since the specular highlights should be easily noticeable.
    - "Cube": a scene consisting of a cube whose faces contain different colors.
    - "ship1.msh" and "ship2.msh": Two more sophisticated scenes stored in triangle meshes. Each triangle has a constant normal direction, which makes the models look faceted.
    - "ship1/ship2.msh (Smooth)": The same meshes with per-pixel interpolated surface normals. This will make them look smoother for Phong shading.
    - "bunny500.msh (Smooth)": Another mesh that's more organic than the ships.

    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 (see Figures 2). The left one is rendered by our software pipeline and the right one is rendered by OpenGL using your computer'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; 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`
- Fragment shading: `FragmentShadedVP` and `PhongShadedFP`

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

Please submit your assignment in the usual way via CMS. Include a .zip file of the entire `pipeline` package tree. Include any additional data files that are needed. As always, document your code and include a brief readme file (txt or PDF) that describes any special structure or additions/omissions that we should be aware of.

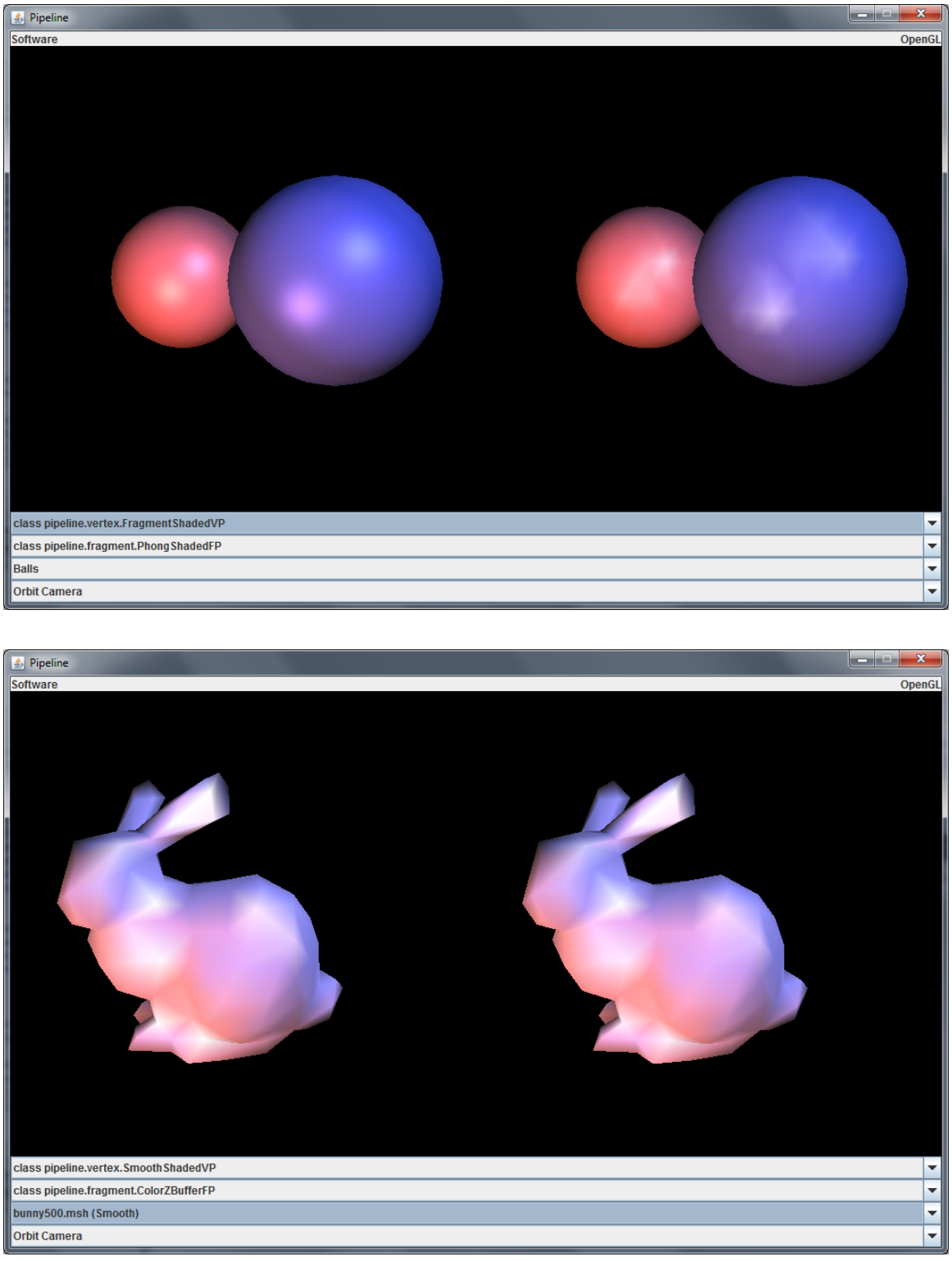

Figure 2: **Screenshots of correct implementations for reference**