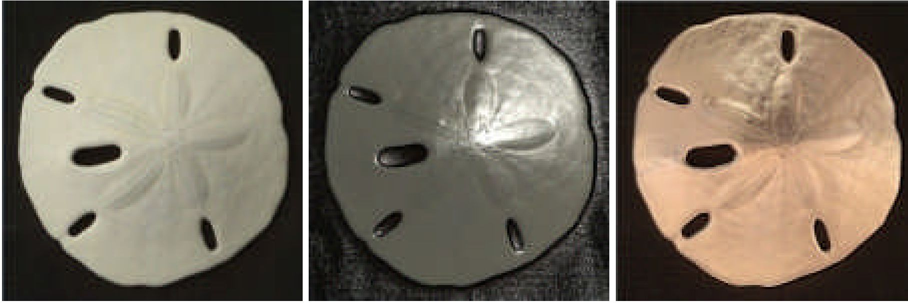
Viewing Polynomial Texture Maps Using Java



Clifford Lyon

Harvard Extension School

Independent Study in Computer Graphics

December 2004

cplyon928@comast.net

Acknowledgements

Thanks to Hanspeter Pfister for directing the project, Tom Malzbender and Dan Gelb for their willingness to share their technology and helpful hints, Henry Leitner and Bill Robinson for allowing this to happen, and Carla Schroer, Mark Mudge, and Joseph Padfield for their feedback and giving it a try.

Table of Contents

Viewing Polynomial Texture Maps Using Java 1

PTM Background 4

Java Implementation 4

Java PTM Library 5

Approach: PTM Model 5

Reading PTMs 6

Version Support 6

Display Method 7

Challenges and Alternatives 7

Supported Features 8

Ambience 8

Directional Light 8

Specular Highlights 8

Diffuse Gain 9

New Features 9

Animation 9

Point light 9

Environment Mapping 9

Appendix 11

Code Example 11

PTM Library Diagram 12

References 13

The PTM Viewer for Java was developed for an independent study in computer graphics through the Harvard Extension School. This paper highlights the structure of the Java application, some of the challenges encountered during the development process, and the features offered by the viewer. The appendix includes a working code sample for the interested coder. The images above, produced by the viewer, show a PTM lit with a directional light, with specular reflection enabled, and with environment mapping enabled.

PTM Background

Hewlett-Packard Labs developed Polynomial Texture Mapping (PTM) as a way of increasing the photorealism of texture maps (Malzbender, T., Gelb, D., & Wolters, H. 2001.) Coefficients of a biquadratic polynomial are stored per texel and used to reconstruct surface color under varying lighting conditions. Like bump mapping, this allows the perception of surface deformations. However, the PTM method is image-based, and photographs of a surface under varying lighting conditions can be used to construct these maps. Unlike bump maps, Polynomial Texture Maps (PTMs) also capture variations due to surface self-shadowing and interreflections, which enhance realism.

Java Implementation

HP implemented a PTM viewer in software using OpenGL, a graphics application programming interface (OpenGL Architecture Review Board, 2003.) The viewer can be downloaded from the PTM website at: http://www.hpl.hp.com/ptm/. The user can change the location of the light using the mouse, producing variations in the PTMs appearance. In addition to image relighting using a directional light, the viewer demonstrates novel lighting effects and material simulations. The viewer is easy to use, fast, and loads all types of PTMs. However, the viewer must be downloaded and unpacked before use. Our goal was to reproduce the functionality of the OpenGL viewer using the Java programming language (Gosling, J., Joy, B., & Steele, G. 2000.) The main benefit of a pure Java solution is accessibility across operating systems. Anyone with the Java plug-in installed and a web browser can use our viewer through the internet. We tried to create a simple, intuitive, and user-friendly interface that would demonstrate the range of PTM

effects in a few minutes using a modern web browser. We also added some new features, such as using the recovered surface normals to perform sphere environment mapping, a technique that simulates the effect of a surrounding environment on the appearance of an image.

Java PTM Library

Approach: PTM Model

We wanted a framework that made it easy to introduce new lighting effects. The design of our PTM library separates the PTM data model, transformations on the data, and presentation of the data into distinct modules. All of the lighting effects take place in the transformation module, which creates a view of the data for presentation. The user chooses which transformations to apply through a graphical user interface (GUI.) The user can adjust parameters for the transformations through the GUI's control panel. The software applies the selected transformations to the data, and presents the results to the viewer. The figure below illustrates flow of data from its raw form through transformation and finally to the viewer, who can modify the transformation parameters interactively and re-display the data.

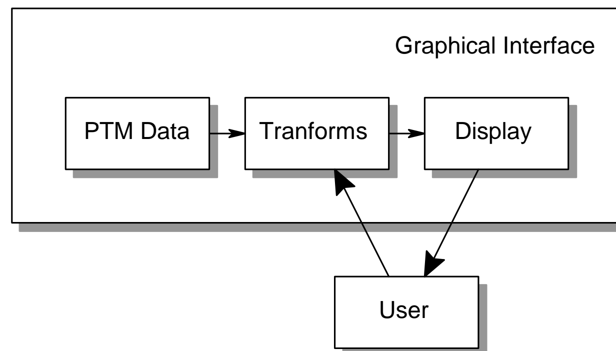


Figure 1 - PTM Model

Module separation fosters extensibility for new transformations: new lighting effects are simply new concrete *Transform* classes; the data model and presentation modules remain unchanged.

Reading PTMs

The first step in acquiring the PTM for display is reading the PTM file. Several versions of PTM files exist. The *PTMIO* module (PTM input/output) reads the PTM files into a standard *PTM* object. The *PTMIO* module provides static methods for accessing the PTM data, and hides the details of the file type from the main library. Inside the class, *PTMIO* reads the header of the file, and instantiates a concrete instance of a *PTMReader* appropriate to the file. If no appropriate parser is available, the module throws an exception. Figure 2 expands the figure above, illustrating the place of the *PTMIO* module in the data path.

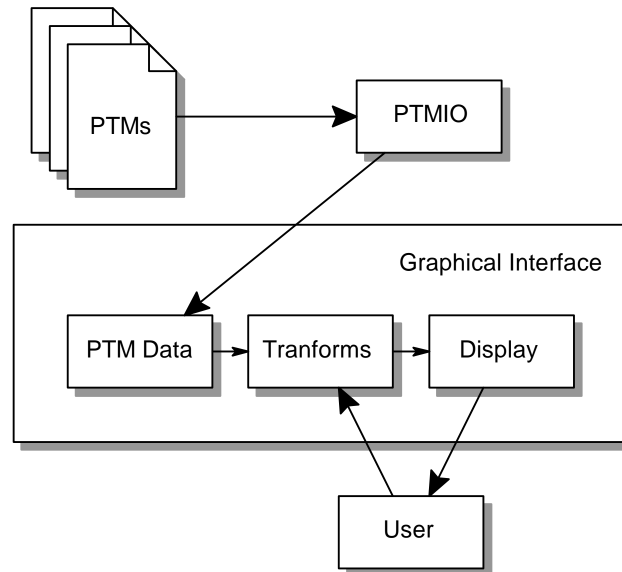


Figure 2 - PTMIO

Version Support

Currently, our software supports a subset of existing PTM file formats: LRBG and JPEG LRGB, versions 1 and 2. LRGB PTMs use a single polynomial at each pixel for all colors (red, green, and blue.) In contrast, RGB PTMs use a separate equation for each color, requiring additional operations at rendering time and additional memory. Since RGB PTMs have a bigger memory footprint and greater runtime cost, we chose LRGB types over RGB counterparts. However, the structure of the library is such that supporting concrete RGB *PTM* and *PTMReader* subclasses would reuse much of the existing framework.

Display Method

The software displays PTMs by applying a transformation to the source PTM data and writing the results to a buffer that stores the pixels of the image displayed in the viewer. The PTM object stores raw RGB color channel information and the polynomial coefficients for lighting effects. A concrete *Transform* operator iterates over the color channel and coefficient arrays, computing a destination value for each array element in the pixel buffer and writing that result. The buffer belongs to a Java *BufferedImage* class encapsulated in a display object, the *PTMCanvas*.

All of the relighting and effects happen inside of a transform operation. The operators keep a reference to the *BufferedImage* pixel buffer, and use that reference to rewrite pixel values. Once the transformation completes, the *paintImmediately()* method updates the *BufferedImage* container. The transform operators implement the re-lighting effects documented by HP Labs in their PTM paper (Malzbender, T., Gelb, D., & Wolters, H. 2001.) In addition, there are some other effects available, such as a point light source and environment mapping.

Challenges and Alternatives

We spent a great deal of time performance tuning. We evaluated OpenGL bindings for Java, the various Java 2D image implementations, and fast math packages. In the end, we settled on a pure Java implementation using a *DataBuffer* from a *BufferedImage* for display, as described above. This choice offered good performance and easy access through a web browser. All of the original performance-critical code is inside the transform classes. In the end, three things sped up the transformations and delivered the best frame rates:

Implementing floating-point calculations as integer math. Despite claims that modern chips do a fine job optimizing floating-point calculations, there was a noticeable improvement in performance using bit-shifted integer approximations for floating point numbers.

Inlining method calls in the rendering loop. This prevented a more elegant pipeline implementation, with a series of callbacks stacked together and fired during the rendering loop. However, it was worth the tradeoff for better performance.

Limiting array index lookups. Java checks that array index values are inbounds each time an array element is dereferenced. However, if an array is used in a loop in such a way that the compiler can tell an out-of-bounds element will never be addressed, this check can be skipped and there is a performance lift. Beyond these general guidelines, several specific performance enhancements improved individual transformation performance (caching, piecewise approximations for continuous functions, etc.)

Supported Features

Ambience

The user can adjust the ambient light in the scene by using the “light” slider in the first panel of the control window, “general”. Moving the “light” slider from left to right increases the amount of ambient light.

Directional Light

The default setting for relighting the image is a directional light. One assumes a directional light to be an infinite distance from the object, so incoming “rays” striking each pixel are considered parallel. Given this assumption, the parameters for the biquadratic polynomial need only be determined once for the entire image, giving a very reasonable performance.

Specular Highlights

The specular highlight operation uses recovered surface normals and a standard Phong lighting equation:

$$I = I_a k_a + I_d k_d (N \cdot L) + I_s k_s (N \cdot H)^n \qquad \textbf{(1)}$$

In this implementation, the diffuse component (the second term) of the lighting equation is simply the directional PTM lighting. The control panel exposes the k_d and c terms, as well at the exponent n. The k_d term controls the amount of diffuse light in the equation. The k_s term controls the degree of specular reflection, and the exponent controls the size of the highlights (high values narrow the highlights). The ambient term is a global value and may be adjusted using “light” slider in the general control panel.

Diffuse Gain

The diffuse gain operator provides enhanced contrast, revealing greater surface detail. This effect was part of the original OpenGL implementation. The HP paper describes it in detail. The level of diffuse gain can be set through the GUI using the “dgain” slider.

New Features

Animation

The animation feature simply moves the light location automatically in a background thread.

Point light

The point light source determines biquadratic polynomial parameters for each pixel separately, factoring in the distance of the light from the source. While more expensive than a directional light, the point light source enables the user to change the distance of the light from the PTM, producing a distinct lighting effect.

Environment Mapping

Environment mapping (or reflection mapping) is a way of adding realism to a scene by allowing the surrounding environment to influence lighting. We implemented sphere environment mapping using the standard OpenGL indexing techniques and Paul Debevec’s sphere maps (http://www.debevec.org/Probes/.) The sphere map image represents the environment surrounding the PTM. Debevec’s maps combine two images to avoid the problem of poor sampling near the backward-facing directions around the circumference of the sphere. The more accurate the derived normals of the original PTM, the better the resulting effect. The software allows the user to select an environment map and to rotate it interactively around the Z-axis.

Implementation

The sphere mapping implementation uses the OpenGL method for indexing into the environment map with the reflection vector. We calculated the recovered surface normal as described in the HP PTM paper for the general case, equations (16), (17), and (18): given coefficients a0, a1, … a5, the normal is:

$$\overrightarrow{N} = (lu_0, lv_0, \sqrt{1 - lu_0{}^2, lv_0{}^2}) \qquad \textbf{(2)}$$

Where :

$$lu0 = \frac{a_2 a_4 - 2a_1 a_3}{4a_0 a_1 - a_2{}^2} \qquad \textbf{(3)}$$

$$lv0 = \frac{a_2 a_3 - 2a_0 a_4}{4a_0 a_1 - a_2{}^2} \qquad \textbf{(4)}$$

Given the normal, and the incedent light, we compute the reflection vector:

$$\vec{R} = 2\vec{N}(\vec{N} \cdot \vec{I}) - \vec{I} \qquad \textbf{(5)}$$

Define:

$$m = \sqrt{Rx^2 + Ry^2 + (Rz + 1)^2} \qquad \textbf{(6)}$$

The texture coordinates into the environment map range from 0 to 1:

$$s = \frac{R_x}{2m} + \frac{1}{2} \qquad \textbf{(7)}$$

$$t = \frac{R_y}{2m} + \frac{1}{2} \qquad \textbf{(8)}$$

The specular and directional transform operators read the value of the indexed pixel from the environment map, and factor that into the final pixel value. The “reflectance slider” on the control panel adjusts the degree of Gaussian blur and down sampling of the environment map. We combined these closely correlated parameters into a single adjustment for simplicity. Moving the “reflectance” slider from left to right increases the simulated roughness of the material, and so results in a more diffuse lighting effect.

Appendix

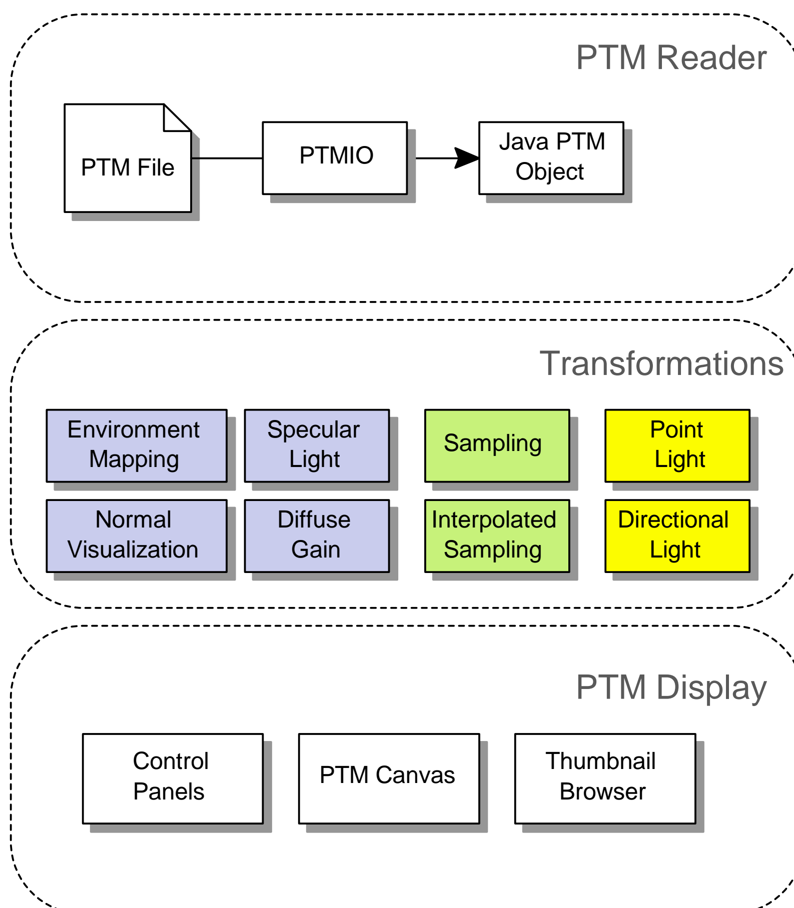
Code Example

This is a code sample demonstrating a "hello world" PTM application using the project library.

```
import java.awt.event.*;
import java.io.*;
import javax.swing.*;
import jpview.gui.PTMCanvas;
import jpview.io.PTMIO;
import jpview.ptms.PTM;
import jpview.transforms.*;

public class Example implements MouseMotionListener {
   PixelTransformOp pixelTransformOp = new DirectionalLightOp();
   PTMCanvas canvas = null;
   PTM ptm = null;
   int mouseX = 0;
   int mouseY = 0;

   public Example( String ptmFileName ) {
      try {
         ptm = PTMIO.getPTMParser(new FileInputStream(new File(ptmFileName))).readPTM();
         canvas = PTMCanvas.createPTMCanvas ( ptm.getWidth() , ptm.getHeight(), PTMCanvas.BUFFERED_IMAGE );
         pixelTransformOp.transformPixels ( canvas.getPixels(), ptm, ptm.getWidth()/2 , ptm.getHeight()/2 );
      } catch ( Exception e ) {
         e.printStackTrace();
      }
   }

   public void mouseDragged(MouseEvent e) {
      mouseX = Math.min(Math.max(e.getX(),0),ptm.getWidth());
      mouseY = Math.min(Math.max(e.getY(),0),ptm.getHeight());
      pixelTransformOp.transformPixels(canvas.getPixels(), ptm, mouseX, mouseY );
      canvas.paintImmediately(0,0,canvas.getWidth(),canvas.getHeight());
   }

   private static void createAndShowGUI(String name) {
      Example example = new Example(name);
      JFrame.setDefaultLookAndFeelDecorated(true);
      JFrame frame = new JFrame("Example");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.getContentPane().add(example.canvas);
      frame.addMouseMotionListener(example);
      frame.pack();
      frame.setVisible(true);
   }

   public static void main(String[] args) {
      try {
         final String name = args[0];
         SwingUtilities.invokeLater(new Runnable() {
            public void run() {
               createAndShowGUI(name);
            }
         });
      } catch (Exception e ) {
         e.printStackTrace();
      }
   }
}
```

PTM Library Diagram

The diagram below shows the layers of the library along the data path. The PTM Reader reads the data and stores the model; the transformations present a view of the data to the GUI. While not a strict implementation of the model-view-controller pattern, it uses a similar separation of concerns.



References

OpenGL Architecture Review Board. (2003). *OpenGL Programming Guide* (4th ed.) Boston: Addison-Wesley.

Malzbener, T., Gelb, D., & Wolters, H. (2001). Polynomial Texture Maps. *Siggraph 2001 Proceedings.* Retrieved May 1, 2004, from http://www.hpl.hp.com/research/ptm/papers/ptm.pdf

Gosling, J., Joy, B., & Steele, G. (2000). *The Java Language Specification: Vol. . The Java Language Specification* (2nd ed.) Boston: Addison-Wesley.