CIS 665 – GPU Programming and Architecture

Homework #2
Due: June 6/3/09

1) <u>Bezier Curves & Color Shaping</u> (30 points)

Background:

The concept of color shaping was introduced in Lecture 2 as a demonstration of shading that exceeds the capabilities of the fixed function pipeline. Recall that color shaping is a technique where lighting terms (specular, diffuse, rim, reflection, etc.) are used to mix a palette of colors. In the example presented in class, simple color shaping is achieved by passing two diffuse and specular colors to the fragment shader (Kd0, Kd1, Ks0, and Ks1), and using the specular and diffuse terms to drive a linear interpolation between the two sets of colors:

```
// Compute the diffuse term
float3 diffuse = lerp(Kd0,Kd1,diffuseLight) *lightColor*diffuseLight;

// Compute the specular term
float3 specular=lerp(Ks0,Ks1,specularLight)*lightColor*specularLight;
```

Linear interpolation (lerp) is limited in that when applied to more than two colors (using several lerp's), Mach Bands can occur in the resulting image (bands of perceptually exaggerated contrast at discontinuities in color). To support more colors and eliminate Mach Bands, we can choose a higher-order interpolation function. A favorite choice in computer graphics is the cubic polynomial. In particular, Bezier curves provide a high degree of control and continuity. An efficient formulation for calculating the Bezier curve interpolation of four values (control points) is as follows:

DeCasteljau's Algorithm:

- Given values b0, b1, b2, b3, and curve parameter t:
    1. Linearly interpolate between points b0 and b1 using parameter t.
    2. Linearly interpolate between points b1 and b2 using parameter t.
    3. Linearly interpolate between points b2 and b3 using parameter t.
    4. Linearly interpolate between the results of step 1 and 2 using parameter t.
    5. Linearly interpolate between the results of step 2 and 3 using parameter t.
    6. Linearly interpolate between the results of step 4 and 5 using parameter t.
    7. The result of step 6 is the Bezier curve point at t.

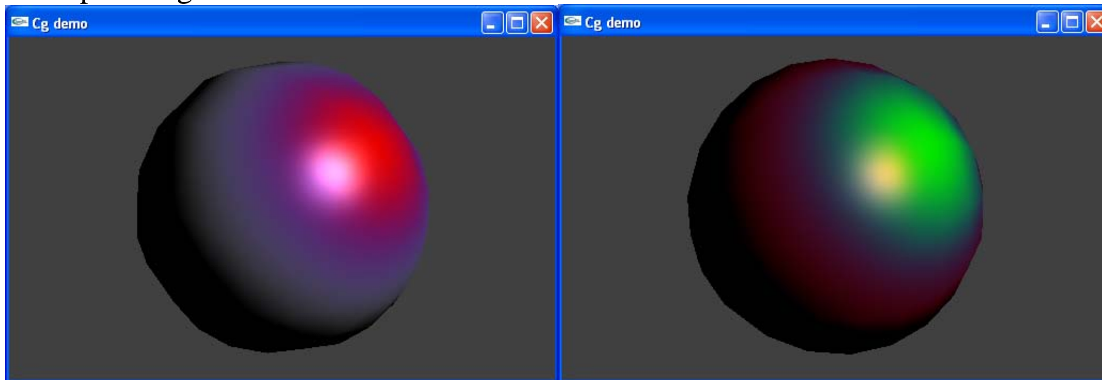
Problem:

Modify the "plastic_color_shape_fragment_sphere" program (available on the course website) to perform color shaping on the diffuse and specular lighting components using Bezier curves rather than lerp. This will entail modifying the demo.cpp code to pass Kd0, Kd1, Kd2, Kd3 , Ks0, Ks1, Ks2, and Ks3 as uniform parameters, adding a Bezier function to the fragment program, and modifying to color shaping calls as follows:

```
diffuse = Bezier(Kd0,Kd1,Kd2,Kd3,diffuseLight) * ..

specular =Bezier(Ks0,Ks1,Ks2,Ks3,specularLight)* ..
```

Example Images:



Kd0	.9	.9	.9	Ks0	.9	.9	.9	Kd0	.9	0	0	Ks0	.9	0	.9
Kd1	.9	.9	.9	Ks1	.9	.9	.9	Kd1	.9	0	0	Ks1	.9	0	.9
Kd2	0	0	.9	Ks2	0	0	.9	Kd2	0	0	.9	Ks2	.9	.9	.9
Kd3	.9	0	0	Ks3	.9	.9	.9	Kd3	0	.9	0	Ks3	.9	0	0

2) **<u>Ward Shading Model (Anisotropic Highlights)</u> (40 points)**

Background:

Read the following excerpt from *Advanced Renderman*, explaining the formulation for the Ward Shading Model:

Anisotropic materials are not uncommon. Various manufacturing processes can produce materials with microscopic grooves that are all aligned to a particular direction (picture the surface being covered with tiny half-cylinders oriented in parallel or otherwise coherently). This gives rise to anisotropic BRDFs. A number of papers have been written about anisotropic reflection models, including Kajiya (1985) and Poulin and Fournier (1990).

Greg Ward Larson described an anisotropic reflection model in his SIGGRAPH '92 paper, "Measuring and Modeling Anisotropic Reflection" (Ward, 1992). In this paper, anisotropic specular reflection was given as

$$\frac{1}{\sqrt{\cos\theta_i \cos\theta_r}} \frac{1}{4\pi\alpha_x\alpha_y} \exp\left[-2\frac{\left(\frac{\hat{h}\cdot\hat{x}}{\alpha_x}\right)^2 + \left(\frac{\hat{h}\cdot\hat{y}}{\alpha_y}\right)^2}{1+\hat{h}\cdot\hat{n}}\right],$$

where

- θ_i is the angle between the surface normal and the direction of the light source.
- θ_r is the angle between the surface normal and the vector in the direction the light is reflected (i.e., toward the viewer).
- $\hat{x}$ and $\hat{y}$ are the two perpendicular tangent directions on the surface.
- α_x and α_y are the standard deviations of the slope in the $\hat{x}$ and $\hat{y}$ directions, respectively. We will call these `xroughness` and `yroughness`.
- $\hat{n}$ is the unit surface normal (`normalize(N)`).
- $\hat{h}$ is the half-angle between the incident and reflection rays (i.e., `H = normalize (normalize(-I) + normalize(L))`).

Problem:

Modify the "plastic_fragment_sphere" program (available on the course website) to implement Ward's formulation for anisotropic specular highlights. This will involve passing uniform xroughness and y roughness parameters to the fragment program, calculating and passing the surface tangents as varying parameters to the vertex program, and coding the above equation in the fragment program (Hint: the cosine of an angle is equivalent to the dot product of unit vectors). The tangents can be calculated as follows:

The demo.cpp code creates a sphere by discretely evaluating the following parametric equation:

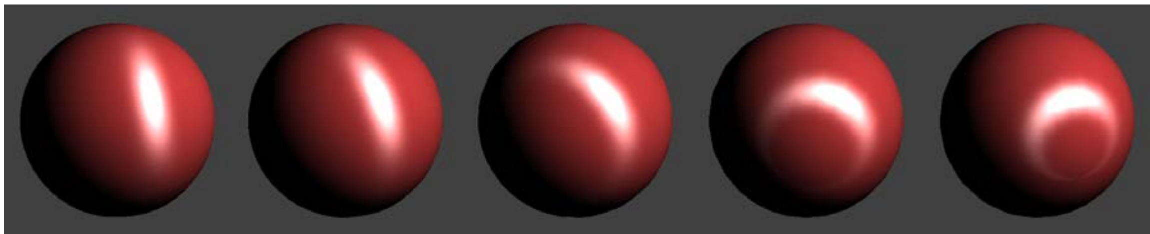
$$P_x = sin(\pi u) * sin(2\pi v)$$
$$P_y = sin(\pi u) * cos(2\pi v)$$
$$P_z = cos(\pi u)$$

The tangents can be derived by taking the partial derivates of the parametric equation with respect to u and v:

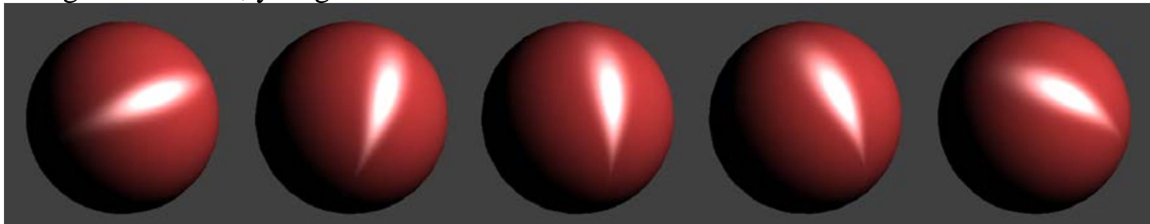
U Tangent:	V Tangent:
$dP_x/du = \pi * cos(\pi u) * sin(2\pi v)$	$dP_x/dv = sin(\pi u) * 2\pi * cos(2\pi v)$
$dP_y/du = \pi * cos(\pi u) * cos(2\pi v)$	$dP_y/dv = sin(\pi u) * -2\pi * sin(2\pi v)$
$dP_z/du = -\pi * sin(\pi u)$	$dP_z/dv = 0$

Using these equations, modify DrawGeometry() and ParametricEval() to calculate the U and V tangents and pass them as varying parameters to the Cg vertex program.

Example Images (Spinning sphere):



xroughness = 0.15, yroughness = 0.45



xroughness = 0.45, yroughness = 0.15

3. GPGPU: Simple Matrix Operations (30 points)

a) Implement a GPGPU algorithm in CG to add two n x m matrices
b) Implement a GPGPU algorithm in CG to subtract two n x m matrices

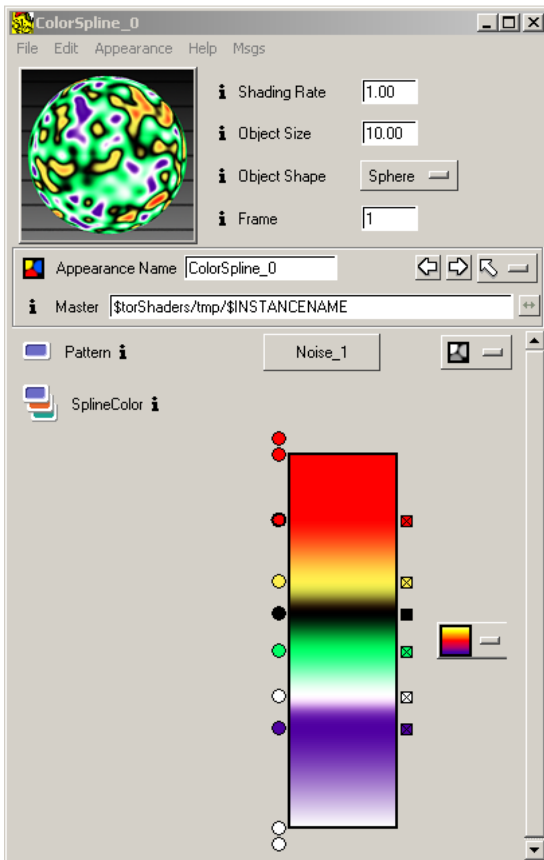
Your program should prompt the user to input a text file containing the matrices.
Your program should prompt the user the filename to save the result matrix.
(Please google c++ file reading if you're unfamiliar with it)

The file format is as follows:	Example:
Operation #start of first matrix Dimensions Data #start of second matrix Dimensions Data #end of operation	+ #start of first matrix 3 4 1 2 3 4 2 3 4 5 5 6 1 3 #start of second matrix 3 4 1 0 0 0 0 1 0 0 0 0 1 0 #end of operation

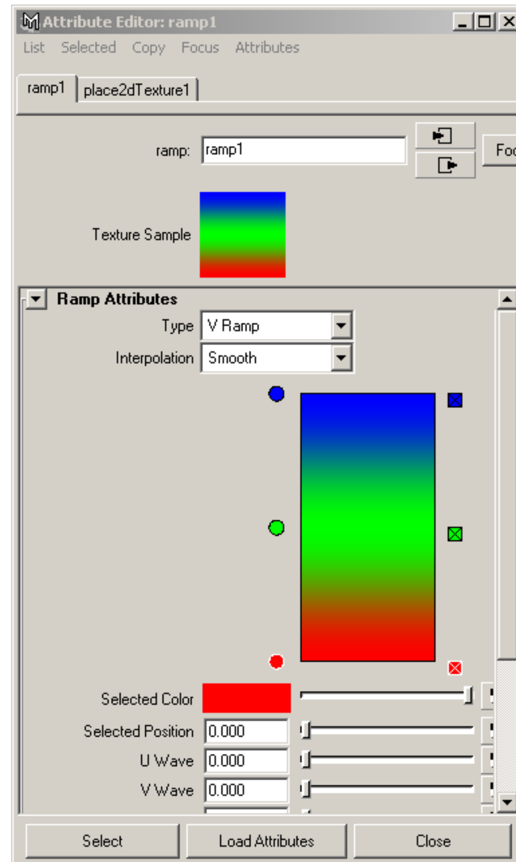
Hint:
Load the data from a file into an array. Then load that data into a texture and send it down to the GPU.

4. <u>Extra Credit: Color Splines</u> (25 points)

While lerp and Bezier curves have provided our Cg shaders with rudimentary color shaping abilities, most production level shading uses a more power formulation: the Color Spline (A spline is a series of curves combined end to end). Conceptually, a one can think of a Color Spline as a list of colors and positions that define knots along an interpolation function. For Maya users, you may recognize Color Splines as "Ramps". For Renderman Arist Tools (RAT) users, you'd recognize them from the ColorSpline slim template:



Renderman Artist Tools' Color Spline



Maya Hypershade's Ramp

We will be reproducing the implementation of the Renderman Artist Tools' (RAT) Color Spline (developed by Pixar Animation Studios). Their Color Spline is actually two splines, one which interpolates the positions of the color knots, and one which interpolates the actual colors of the knots. Each is implemented as a "Catmull-Rom" spline.

Catmull-Rom Splines

Catmull-Rom splines are composed of a list of input values which form the beginning and end control points of a series of Bezier curves. The inner two control points of each Bezier curve are interpolated from the surrounding input values via the following formula:

Given input points P_0 through P_n, the four Bezier curve control points for each pair P_{i-1} and P_i are computed as follows:

$$b0 = P_{i-1}$$
$$b1 = P_{i-1} + ((P_i - P_{i-2}) / 6)$$
$$b2 = P_i - ((P_{i+1} - P_{i-1}) / 6)$$
$$b3 = P_i$$

Note that this formula requires points P_{-1} and P_{n+1}, which are not given. These points can be extrapolated using the following equation:

$$P_{-1} = (P_1 - P_0) / 2$$
$$P_{n+1} = (P_n - P_{n-1}) / 2$$

For the purpose of a color splines, it is useful to index the Catmull-Rom spline with a parameter t that ranges from 0 to 1. Using this parameter we can decide which of the spline curves to evaluate and at what value u. If there are N input points to the Catmull-Rom spline, there will be N-1 curves. Thus, we will evaluate the max(floor((N-1)*t)+1,N-1)th curve at value u = ((N-1)*t) - floor((N-1)*t).

RAT Color Splines

The user provides a Color spline with N rgb colors (C_0 through C_{n-1}), N float positions (K_0 through K_{n-1} ,between 0 and 1), and a parameter t used to index the spline. A new paramter k is calculated as catmull_rom(K_0 through K_{n-1} , t). The final color is calculated as catmull_rom(C_0 through C_{n-1} , k).

In practice, support for an arbitrary N colors and positions is cumbersome on a GPU. For simplicity we will implement Color Splines where N = 8 (Any more knots runs the risk of causing aliasing anyway). Modify your solution to problem 1 to perform color spline color shaping. You will need to pass Kd0 through Kd7, Kd0_k though Kd7_k, Ks0 through Ks7, and Ks0_k though Ks7_k to the fragment program. You will need to write a catmull_rom function (for floats and float3's), as well as a color_spline function in the fragment program. They should be called as follows:

```
diffuse = color_spline(Kd0,Kd1,Kd2,Kd3,Kd4,Kd5,Kd6,Kd7,Kd8,
          Kd0_k,Kd1_k,Kd2_k,Kd3_k,Kd4_k,Kd5_k,Kd6_k,Kd7_k,Kd8_k,
          diffuseLight) * ..

specular = color_spline(Ks0,Ks1,Ks2,Ks3,Ks4,Ks5,Ks6,Ks7,Ks8,
           Ks0_k,Ks1_k,Ks2_k,Ks3_k,Ks4_k,Ks5_k,Ks6_k,Ks7_k,Ks8_k,
           specularLight) * ..
```

References

1. Apodaca A., Gritz L., Barzel R., Calahan S., Hanson C., Johnson S. *Advanced Renderman – Creating CGI for Motion Pictures*. Morgan Kaufman Publishers, 2000.

2. Ward, G., *Measuring and Modeling Anisotropic Reflection*. Computer Graphics, Vol. 26, No. 2, July 1992.

3. C. Everitt. *Order-Independent Transparency*. http://developer.nvidia.com/object/Interactive_Order_Transparency.html

4. Henning Mortveit, Reflections from Bumpy Surfaces. Shader X3. Charles River Media, Inc. 2005.