Lab 4
More Shaders

CSE 4431/5331.03M

Advanced Topics in 3D Computer Graphics

TA: Margarita Vinnikov

mvinni@cse.yorku.ca

Debugging Shaders

- Can't print a number from a shader, but you can "print" a colour,
  - most of our value-checking is going to be by setting the fragment shader outputs to a value that we want to test and then looking at it to guess the number.
- The trick is to isolate each variable, and test one at a time. If you try testing the whole thing you will pull your hair out and waste a lot of time.
- This is the general process:
  1. allow one variable to change value
  2. make all other variables some constant value
  3. work out the expected result on paper
  4. check the value of the variable being tested to see if it matches expected (paper) result
  5. repeat process for next variable
- http://antongerdelan.net/opengl/debugshaders.html

Shading Models

- **Flat** shading
  - compute a single colour value per triangle
- **Gouraud** shading
  - compute the colour for the vertices of a triangle and interpolate the colour values for points inside the triangle
- **Phong** shading
  - compute the colour for every surface point

Colour in Shaders

- **diffuse**: light reflected by an object in every direction. This is what we commonly call the colour of an object.
- **ambient**: used to simulate bounced lighting. It fills the areas where direct light can't be found, thereby preventing those areas from becoming too dark. Commonly this value is proportional to the diffuse colour.
- **specular**: this is light that gets reflected more strongly in a particular direction, commonly in the reflection of the light direction vector around the surface's normal. This colour is not related to the diffuse colour.
- **emissive**: the object itself emits light.

Quiz

- Ambient

- *Diffuse*

- *Specular*

- *Ambient + diffuse*

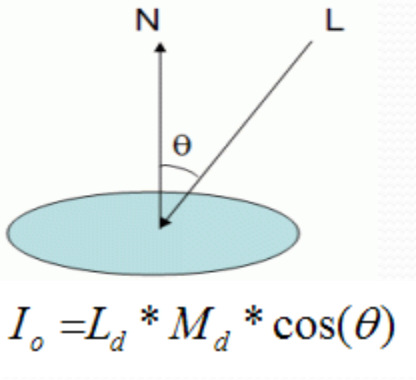
- Point Light

- Spot Light

Lights

- **Directional** light
  - all light rays are parallel, as if the light was placed infinitely far away, and distance implied no attenuation. For instance, for all practical purposes, for an observer on planet earth, the light that arrives from the sun is directional. This implies that the light direction is constant for all vertices and fragments, which makes this the easiest type of light to implement.
- **Point** lights spread their rays in all directions, just like an ordinary lamp, or even the sun if we were to model the solar system.
- **Spotlights** are point lights that only emit light in a particular set of directions. A common approach is to consider that the light volume is a cone, with its apex at the light's position. Hence, an object will only be lit if it is inside the cone.

Diffuse Light



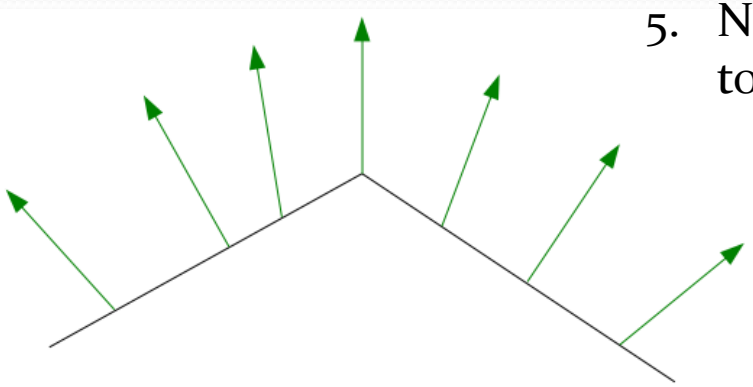
I_o is the reflected intensity,
L_d is the light's diffuse color (*gl_LightSource[o].diffuse*),
M_d is the material's diffuse coefficient (*gl_FrontMaterial.diffuse*).

1. The normal vector and the light direction vector (*gl_LightSource[o].position*) has to be normalized

2. Cos(θ) = lightDir . normal / (|lightDir| * |normal|)
3. We want *gl_Normal* and *lightDir* to be normalized
   | normal | = 1 | lightDir | = 1

   So that
   Cos(θ) = lightDir . normal

4. OpenGL stores the lights direction in eye space coordinates;
5. Need to transform the normal to eye space in order to compute the dot product.

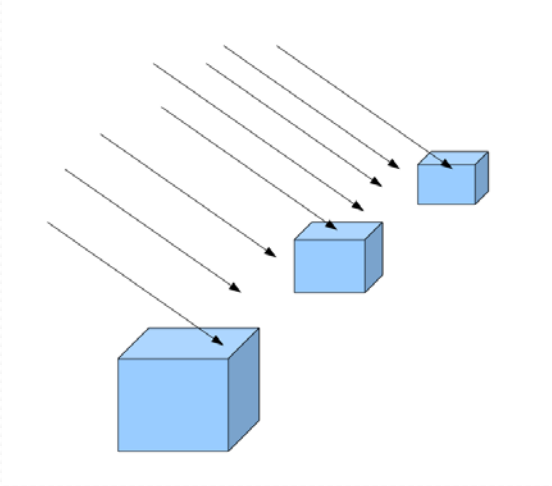


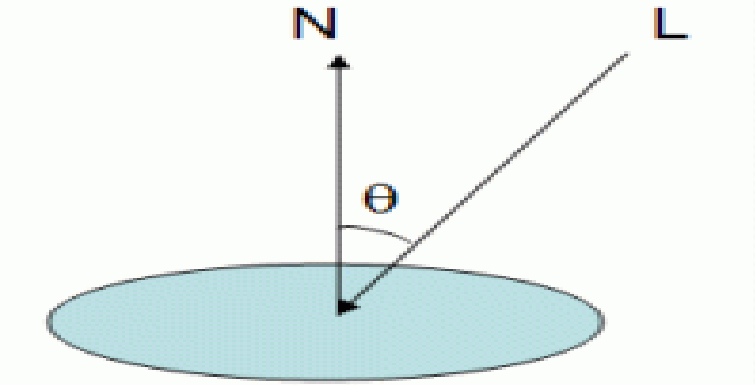
Eye Space

- To transform the normal to eye space use the *gl_NormalMatrix*.
  - This matrix is the transpose of the inverse of the 3x3 upper left sub matrix from the modelview matrix.

```
varying vec3 normal;
void main()
{
  normal = gl_NormalMatrix * gl_Normal;
   …;
}
```

Ambient Light

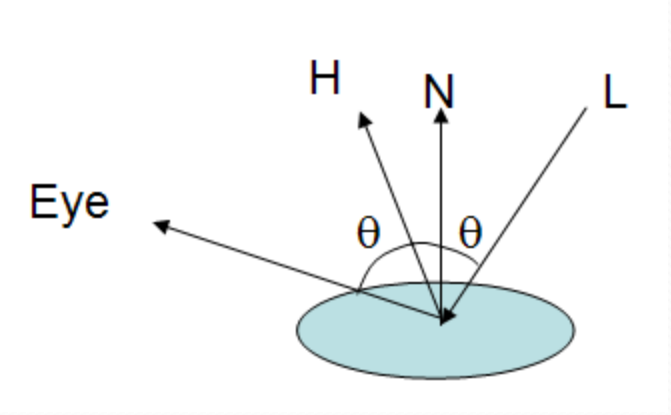




$$I_a = G_a * M_a + L_a * M_a$$

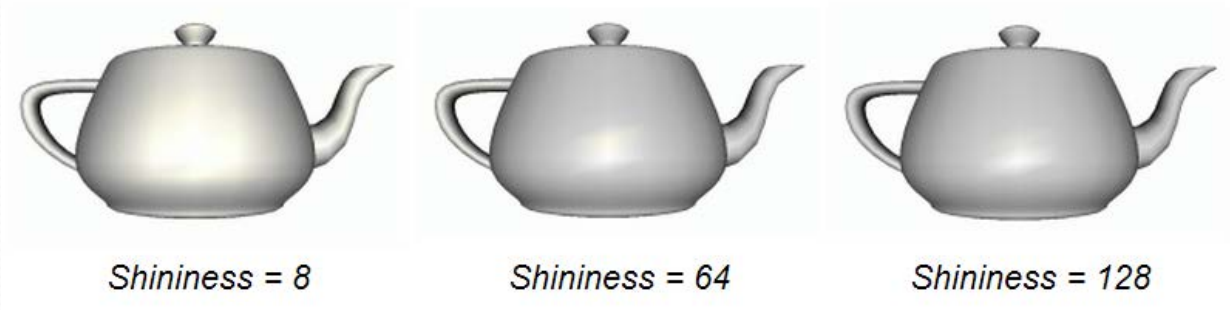
I_a is the ambient intensity,
M_a is the global ambent (gl_LightModel.ambient)
L_a is the light's ambinent color (*gl_LightSource[o].ambient*),
M_a is the material's ambient coefficient (*gl_FrontMaterial.ambient*),

Specular Lights



$$H = Eye - L$$

$$Spec = (N \cdot H)^{s} * L_s * M_s$$



Shininess = 8 Shininess = 64 Shininess = 128

Light Source Review

```
Vertex Color = emission + globalAmbient + sum(attenuation
  * spotlight * lightAmbient + (max {L.N, 0} * diffuse) +
  (max {H.N, 0} ^ shininess)*specular])
```

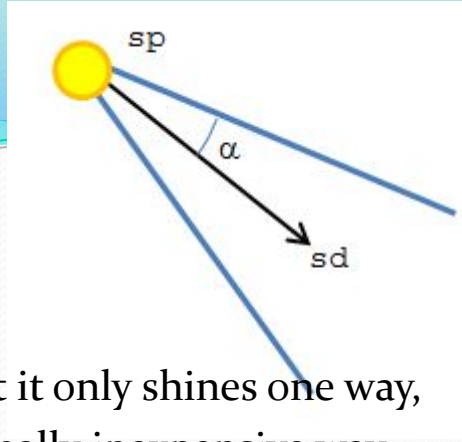
Terms	Description	
Emission	is the material's emissive color	
Global Ambient	is the ambient color*global ambient brightness	
Attenuation	is a term causing lights to become dimmer with distance	
Spotlight	is a term causing spotlight effects	
Ambient	is the light's ambient color*brightness	
Diffuse	is the light's diffuse color * the material's diffuse color	
Shininess	is the specular exponent, which tells us how shiny a surface is	
Specular	is the light's specular color * the material's specular color	
L	is the normalised(unit length) vector from the vertex we are lighting to the light	
N	is the unit length normal to our vertex	
H	is the normalised "half angle" vector, which points half way between L and the viewer (V)	

1st Task

Your Turn

- Follow instructions attached (Lab 3), implement light shaders
- Bonus: create an interesting light shader

Spotlights



- Shine a light in one specific direction.
  - It still comes from a specific point in space though.
  - In the real world we would just stick the light source in a cone so that it only shines one way,
  - Not actually calculating light rays in our GPU - do it in a computationally inexpensive way.

1. Define a direction for the light (normal)

```
vec3 spot_dir_eye = normalize (target_position_eye - light_position_eye);
float spot_dot = dot (spot_dir_eye, dir_to_surface_eye);
const float spot_arc = 1.0 - 5.0 / 90.0;
float spot_factor = 1.0;
     if (spot_dot < spot_arc) {
       spot_factor = 0.0;
     }
     ...
   diffuse_light *= spot_factor;
   speculat_light *= spot_factor;
```

http://antongerdelan.net/opengl/lights.html
http://www.lighthouse3d.com/tutorials/glsl-core-tutorial/spotlights/

Attenuation

- Basic spotlight calculation will give us a uniform circle of light on a surface if we use it to factor our diffuse and specular light components.
  - What if we want a roll-off or attenuation from centre to edge instead?

- Need to make ure that the value stays between 0 and 1 by using the clamp() function.

```
float spot_factor = (spot_dot - spot_arc) / (1.0 - spot_arc);
spot_factor = clamp (spot_factor, 0.0, 1.0);
```

Textures

Texture

- In order to perform texturing operations in GLSL need to access the texture coordinates per vertex.
  - GLSL provides some attribute variables, one for each texture unit:

```
attribute vec4 gl_MultiTexCoordo;
...
attribute vec4 gl_MultiTexCoord7;
```

- GLSL also provides access to the texture matrices for each texture unit in an uniform array.
  - uniform mat4 gl_TextureMatrix[gl_MaxTextureCoords];

Vertex shader

- Has access to the attributes defined above to get the texture coordinates specified in the OpenGL application.
- Has to compute the texture coordinate for the vertex
  - store it in the pre defined varying variable *gl_TexCoord[i]*,
  - where *i* indicates the texture unit.
- Example :

```
void main()
{
    gl_TexCoord[o] = gl_MultiTexCoordo;
    gl_Position = ftransform();
}
```

- To use the texture matrix :

```
void main() {
    gl_TexCoord[o] = gl_TextureMatrix[o] * gl_MultiTexCoordo;
    gl_Position = ftransform();
 }
```

Fragment Shader

- *In gl_TexCoord* is a varying variable
  - used in the fragment shader to access the interpolated texture coordinate.
  - Need to declare a special type of variable.
    - sampler*i*D,
    - where *i* is the dimensionality of the texture.
    - uniform sampler2D tex;
    - The user defined *tex* variable contains the texture unit we are going to use, in this case 0.
    - The function that gives a texel,
      - a pixel in the texture image, is *texture2D*.
      - This function receives a *sampler2D*, the texture coordinates, and it returns the texel value.
      - The signature is as follows: vec4 texture2D(sampler2D, vec2);
      - The returned value takes into account all the texture settings as defined in the OpenGL application, for instance the filtering, mipmap, clamp, etc…
  - Example

```
uniform sampler2D tex;
void main()
{
    vec4 color = texture2D(tex,gl_TexCoord[0].st);
    gl_FragColor = color;
}
```

  - When accessing texture coordinates use s,t,p,q.
    - Note that *r* is not used to avoid conlficts with rgb selectors

OpenGL 3.x and 4.x

- Texturing: http://lwjgl.org/wiki/index.php?title=The_Quad_textured
- VBO
  - GLuint positionIndex = 0;
    GLuint texcoordIndex = 1;
    glUseProgram(programId);
    glBindAttribLocation(programId, positionIndex, "position");
    glBindAttribLocation(programId, texcoordIndex, "texcoord");

- //Then before your glDraw*()
  - glEnableVertexAttribArray(positionIndex);
    glEnableVertexAttribArray(texcoordIndex);

  - http://www.opengl.org/wiki/Vertex_Specification#Vertex_Array_Object
  - http://en.wikipedia.org/wiki/Vertex_Buffer_Object

2nd Task

Your Turn

- Pass texture and display it in black an white

- Bonus: you can turn texture to grayscale or sepia:
  - For example:
    - // Convert to grayscale using NTSC conversion weights
    - gray =dot(txt.rgb, vec3(0.299, 0.587, 0.114));
    - sepia = vec4(gray * vec3(1.2, 1.0, 0.8), 1.0);

- Bonus: use texture to pass information about lighting coeffiecients
  - http://antongerdelan.net/opengl/phongtextures.html

- Bonus: use texture to discard fragments
  - See the following link for example
    - http://www.opengl.org/sdk/docs/tutorials/ClockworkCoders/discard.php

Bump Mapping

Putting it all together

Bump Mapping

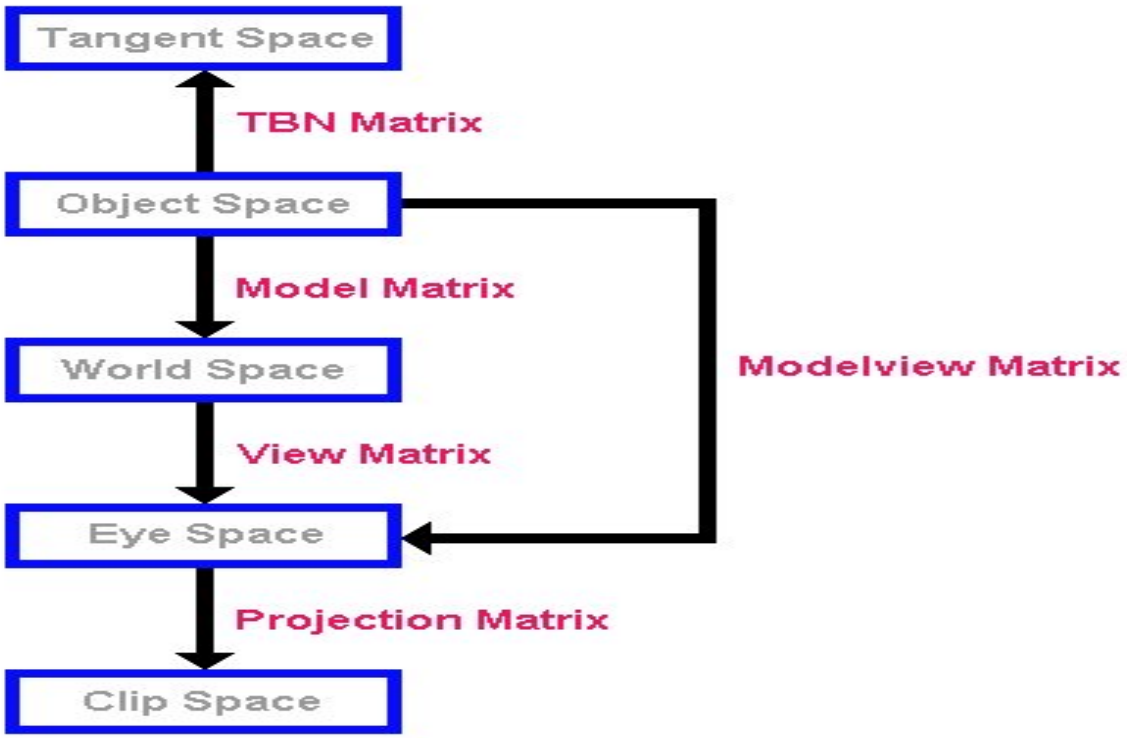
"A rendering technique that simulates the appearance of bumps, wrinkles, or surface irregularities by perturbation surface narmals to performing lighting calculations" (orange book)

- Modulating the surface normals before the lighting is applied
- Increasing realism without increasing geometric complexity
- Simulates surface irregularities
- Will not show on the silhouette edges
- To apply small effects
- Main work will be done in fragment shader

Need

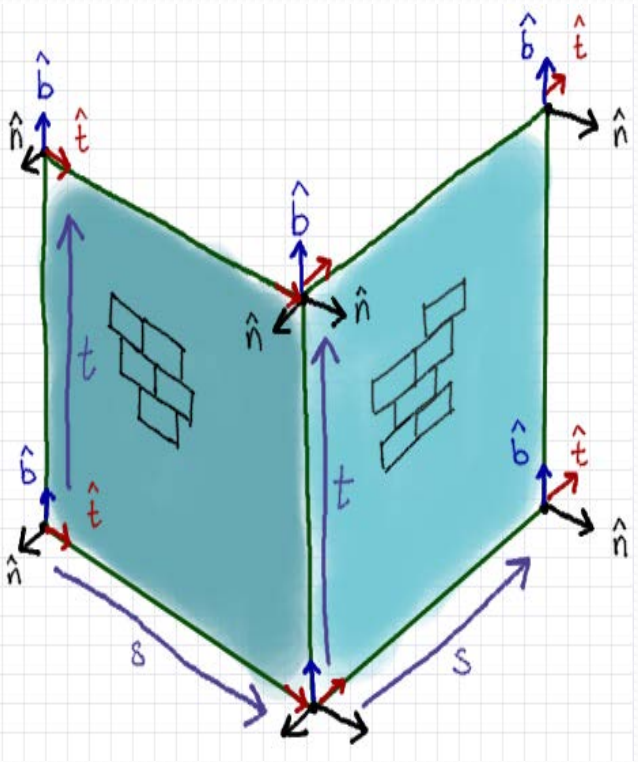
- Valid surface normals at each fragment location
- Light source
- Viewing direction vector

Coordinate Spaces



Surface Local Coordinate Space (Tangent Space)

- Varies over a rendered object
- Each point is at (0,0,0) and the unpertubrated surface normal is (0,0,1)
- Converted:
  1. Light direction
  2. Viewing direction
  3. Computed pertubrated normal
- Need to construct transformation matrix at each vertex
- How?



Why use tangent space?
Why not declare all in world space?

- Many per pixel lighting techniques and other shaders require normals and other height information declared at each pixel point.
  - Need to have one normal vector at each texel and the n axis will vary for each texel
    - Like a bumpy surface defined on a flat plane.
  - If these normals were declared in the world space coordinate system -> need to rotate these normals every time the model is rotated
    - Lights, camera and other objects will be defined in world space and will move independent of the object.
      - This would mean thousands or even millions of object to world matrix transformations will need to take place at the pixel level
- Instead of
  - converting thousand or millions of surface normals to world space
  - convert tens or hundreds of light/camera/vertex positions to tangent space and do all our calculations in tangent space as required
  - the matrices required for the transformations can be pre calculated.
  - They only need to be recalculated when the positions of the vertices change with respect to each other.

Deriving world to tangent space transformation matrix

1. Define u, v and n basis vectors in terms of the world space coordinate system.
   - Refer to the u, v, n vectors as the Tangent (T), Bi-Normal(B) and Normal (N).

$$M_{wt} = \begin{pmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \\ N_x & N_y & N_z \end{pmatrix}$$

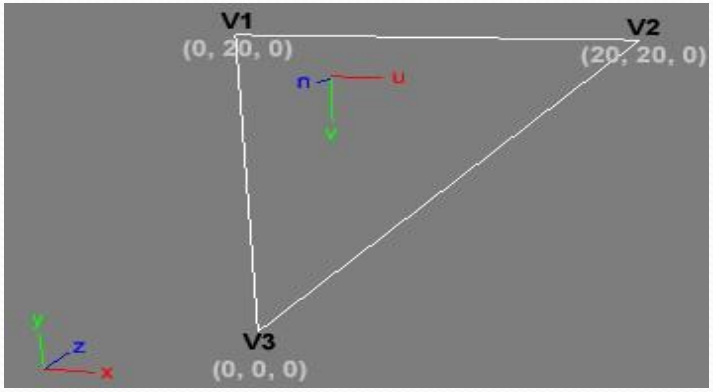
Point in tangent space (*Posts*) :

$$P_{ts} = Pws \; X \; Mwt$$

- *T, B, N* vectors (and all basis vectors) will always be at right angles to each other.
  - All we need to do is derive any two vectors
  - The third will be a cross product of the previous two.

Creating the tangent space matrix for a face

- **Step 1:** Calculate any two edge vectors.
  - $E_{2\text{-}1} = V_2 - V_1 = (20\text{-}0, 20\text{-}20, 0\text{-}0)$ & $(1\text{-}0, 0\text{-}0) = (20, 0, 0)$ & $(1, 0)$
    $E_{3\text{-}1} = V_3 - V_1 = (0\text{-}0, 0\text{-}20, 0\text{-}0)$ & $(0\text{-}0, 1\text{-}0) = (0, -20, 0)$ & $(0, 0)$
- **Step 2:** Calculate the T vector
  - $T = E_{2\text{-}1}.xyz \,/\, E_{2\text{-}1}.u = (20/1\,,\ 0/1\,,\ 0/1) = (20, 0, 0)$
  - $T = Normalize(T) = (1, 0, 0)$
  - Note: If the value of *E2-1.u* value is 0, then we perform the same calculation on *E3-1*. It does not matter which edge we use, all we are looking for is the direction of increasing u component across the face.



Vertex	*Position (x, y, z)*	*Texture coordinate (u, v)*
V1	*(0, 20, 0)*	(0, 0)
V2	*(20, 20, 0)*	(1, 0)
V3	*(0, 0, 0)*	(0, 1)

Creating the tangent space matrix for a face 2

- **Step 3:** Calculate the Normal vector (N)
  - Cross product of the edge vectors $E_{2\text{-}1}$ and $E_{3\text{-}1}$
  - $N = CrossProduct(E_{2\text{-}1}, E_{3\text{-}1}) = CrossProduct\ (\ (20, 0, 0)\ ,\ (0, -20, 0)\ ) = (0, 0, -400)$
    $N = Normalize(N) = (0, 0, -1)$
- **Step 4:** Calculate the Bi-Normal vector (B)
- *T, B, N* vectors are always at right angles to each other
  - Derive the third from by just doing a cross product of the previous two. Therefore
  - $B = CrossProduct\ (\ T\ ,\ N\ ) = CrossProduct\ (\ (1, 0, 0)\ ,\ (0, 0, -1)\ ) = (0, 1, 0)$
- **Step 5:** Build M_{wt} from *T, B, N*

$$M_{wt} = \begin{pmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \\ N_x & N_y & N_z \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{pmatrix}$$

More Theory

- http://www.terathon.com/code/tangent.html
- http://antongerdelan.net/opengl/normal_mapping.html

Application Setup

- Application must send
  - Vertex position
  - Surface normal
  - Tangent vector
    - Passed as vertex attribute
  - The surface rendered
  - Other uniform variables
    - LightPosition
    - SurfaceColor

```
void PassNormalsToShader ()
 {
    //Load Texture

    // Pass the texture/texture to shader
     glActiveTexture(GL_TEXTURE0);
     glBindTexture(GL_TEXTURE_2D, uiTextureN);
     glUniform1i(p,0);  // The texture in the second slot
}
void generatTangent( float p[4][3], float s[4][2],float tangent[3])
{

       float v1[] = {p[2][0]-p[1][0],p[2][1]-p[1][1],p[2][2]-p[1][2]};
       float v2[] = {p[3][0]-p[1][0],p[3][1]-p[1][1],p[3][2]-p[1][2]};

       float st1[]= {s[2][0]-s[1][0],s[2][1]-s[1][1]};
       float st2[] ={s[2][0]-s[1][0],s[2][1]-s[1][1]};

       float coef = 1/ (st1[0] * st2[1] - st2[0] * st1[1]);

       tangent[0] = coef * ((v1[0] * st2[1])  + (v2[0] * -st1[1]));
       tangent[1] = coef * ((v1[1] * st2[1])  + (v2[1] * -st1[1]));
       tangent[2] = coef * ((v1[2] * st2[1])  + (v2[2] * -st1[1]));
}
```

- Use
  - glVertexAttrib to set tonagent for a given set of vertices.
  - Or
  - glVertexAttribPointer sets the *location of the tangents* for every vertex

Normal Maps

- Normals on a per pixel basis
  - use a texture map.
  - store normal vectors, not colors.
- letting the red, green and blue components of the texture equal x, y and z respectively.
- The color components must lie between 0 and 1.
- r = (x+1)/2;
  g = (y+1)/2;
  b = (z+1)/2;
- The normals are in tangent space.
  - Normal points up in the z direction.
  - RGB color for the straight up normal is (0.5, 0.5, 1.0).
  - This is why normal maps are a blueish color.
- Make your own tutorial
  - http://www.bencloward.com/tutorials_normal_maps11.shtml

Gimp or Photoshop

- https://code.google.com/p/gimp-normalmap/
- for GIMP, which you can download. The trick then, is to create the height-map. We will be using the same texture coordinates as the regular textures, so I just converted one of my textures to greyscale and drew over it with a grey paintbrush to flatten-out the surfaces (see above image). You can see that this might take a bit of trial-and-error to get it looking good.
- The normal map creation procedure looks for differences in height to work out which way the normals should point. These are encoded as RGB colours with a range of 0 to 1. Our normals will need to be -1 to 1, so we will remember to modify this when we sample it, later. It's also possible to write a little function to generate the normals from just a height-map, of course.

Vertex Shader

- Computing
  - Surface-local light direction
  - Surface-local eye direction

```
varying vec4 passcolor; //The vertex color passed
varying vec3 LightDir; //The transformed light direction, to pass to the fragment shader
attribute vec3 tangent;
 void main()
 {
      //Put the color in a varying variable
      passcolor = gl_Color;
       //Put the vertex in the position passed
        gl_Position = ftransform();
        //Use the first set of texture coordinates in the fragment shader
        gl_TexCoord[o] = gl_MultiTexCoordo;
      // transform the normal into eye space and normalize the result */
         vec3 n = normalize(gl_NormalMatrix * gl_Normal);
      /* first transform the tangent into eye space - use gl_NormalMatrix and normalize the result */
      vec3 t = normalize(gl_NormalMatrix * tangent);
      //calculate binormal vector by ferforming cross product between normal and tangent
      vec3 b = cross(n, t);

      vec3 vVertex = vec3(gl_ModelViewMatrix * gl_Vertex);
      vec3 tmpVec = gl_LightSource[o].position.xyz - vVertex;

      LightDir.x = dot(tmpVec, t);
      LightDir.y = dot(tmpVec, b);
      LightDir.z = dot(tmpVec, n);
}
```

Fragment Shader

- Read the normals from texture shader
- Pertubrate the texel color based on the normal direction

```
uniform sampler2D BumpTex; //The bump-map
varying vec4 passcolor; //Receiving the vertex color from the vertex shader
varying vec3 LightDir; //Receiving the transformed light direction

 void main()
 {
        //Get the norm of the bump-map from BumpTex by using texture2D()and gl_TexCoord[0].xy
        vec3 BumpNorm = vec3(texture2D(BumpTex, gl_TexCoord[0].xy));
         //Expand the bump-map into a normalized signed vector (actually in a ranges of [-1,1])
        BumpNorm = (BumpNorm -0.5) * 2.0;

       vec3 lVec = normalize(LightDir);
      float diffuse = max( dot(lVec, BumpNorm), 0.0 );

         gl_FragColor = vec4(diffuse*passcolor.rgb, passcolor.w);

 }
```

3rd Task

Your Turn

- Implement Bump Mapping
- Bonus:
  - Modify to work with different lights
  - Modify to work with textured surface
  - Modify to work without normal texture

- Submit your code as follows:

  submit 4431 lab4 filename(s)