fragment shader

- simulate materials
- use texture data

Materials

- we can simulate light bouncing off of a material
- best done in the fragment shader if possible
- Uniform variables are often used to describe things like the positions of some light sources in the scene, which do not change from vertex to vertex.
- Varying variables are often used to describe the coordinate vector of the point (with respect to, say, the eye frame), a normal for the point
- parameters describing the material properties at the point might be uniforms or varying.
- The fragment shader then typically takes this data and simulates how light would bounce off of this material, producing a color in the image.

basic idea

- When light hits a physical material, it is scattered in various outgoing directions.
- Different kinds of materials scatter light in different patterns and this results in different appearances when observed by the eye or a camera.
  - Some materials may appear glossy while others matte.
- By simulating this scattering process, we can give a realistic physical appearance to a 3D rendered object.
- we will take a pragmatic approach here instead of a scientific one.
- we have used 3 materials this semester

example: pvc blob

- see pvc blob
  - Note that this figure just describes the result of light that comes in from the specific shown direction $\vec{l}$. For other incoming directions we would need a different blob to visualize the resulting scattering.
- the plastic will appear brightest when observed in the directions clustered about the "bounce" direction of the light: $B(\vec{l})$.
- Given any vector $\vec{w}$ (not necessarily of unit norm) and a unit normal vector $\vec{n}$, we can compute the bounce vector of $\vec{w}$ as

$$B(\vec{w}) = 2(\vec{w} \cdot \vec{n})\vec{n} - \vec{w} \tag{1}$$

basic setup

- see figure
- we assume that all light comes from a single *point light source* that is pointed to by the *light vector*, $\vec{l}$.
  - in real life, there are some small light sources (sun, candle), there are some big light sources (sky, fluorescent bulbs).
  - additionally, EVERY surface in the scene reflects light towards the point in question.
- The light hits the point $\tilde{p}$ on some triangle with normal $\vec{n}$.
- The angle between the surface normal and the vector pointing towards the light is denoted as θ.
- we want to compute the amount of light reflected towards the eye along the *view vector*, $\vec{v}$.

color

- To handle color, we just compute three reflected quantities for RGB.
- we typically represent the incoming light in RGB (often white)
- and use RGB coefficients for the material property representation.
- and just do 3 independent calculations
- Note that this use of RBG is not physically accurate
- These quantities are then directly used as the RGB values for the pixel.

more details

- We will often assume here that all coordinates of points and vectors are written with respect to the eye frame.
- note the normal does not have to be the "true normal" of the flat triangular geometry.

Diffuse

- "Diffuse" materials, like rough wood, appear equally bright when observed from all directions $\vec{v}$.
  - Thus, when calculating the color of a point on a diffuse surface, we do not need to use the $\vec{v}$ vector at all.
- Diffuse materials appear brighter when light hits them "from above", and appear dimmer when light hits them from a grazing angle.
- This is because the amount of incoming photons hitting a small, fixed sized, patch of material is proportional to $\cos(\theta) = \vec{n} \cdot \vec{l}$.

diffuse shader

```
#version 130

uniform vec3 uLight, uColor;

in vec3 vNormal;
in vec3 vPosition;

out vec4 fragColor;

void main() {
  vec3 tolight = normalize(uLight - vPosition);
  vec3 normal = normalize(vNormal);

  float diffuse = max(0.0, dot(normal, tolight));
  vec3 intensity = uColor * diffuse;

  fragColor = vec4(intensity, 1.0);
}
```

- The `normalize` call is used to make sure the interpolated vectors have unit norm.
- The `max` call is used so that we do not try to produce negative colors at points that face away from the light.
- The calculated value of `diffuse` is used to modulate the intrinsic surface color passed in as `uColor`.

extensions

- the light itself has not been given a color of its own. This too can be easily modeled with the addition of another uniform variable.
- Multiple lights can also be easily added to this code.
- indirect light can be very crudely modeled in our computation by simply adding some constant *ambient* color value to `intensity`.

- Sometimes, it may be more convenient to model the light as coming from a direction, described by a vector, instead of a from a point. In this case, when modeling such a *directional light*, the vector `toLight` would be passed in as a uniform variable.

shiny

- Many materials, such as plastics, are not diffuse;
- for a fixed incoming light distribution, they appear brighter when observed from some directions and dimmer when observed from others.
- Round objects of made up of such materials have bright highlights wherever the surface-normal points "just right".
- A simple, somewhat plausible computation that is often used in practice, is to simply calculate the light's bounce vector $B(\vec{l})$, and then compute its angle with $\vec{v}$.
- When the bounce and view vectors are well aligned, we draw a very bright pixel. When they are not well aligned, we draw a dimmer pixel.
  - done with a dot product
- To model shiny materials, we want the brightness to fall off very quickly in angle, so we then raise the dot product to a positive power

shiny shader

```
uniform vec3 uLight,  uColor;

in vec3 vNormal;
in vec3 vPosition;

out vec4 fragColor;

void main() {
  vec3 normal = normalize(vNormal);
  vec3 viewDir = normalize(-vPosition);
  vec3 lightDir = normalize(uLight - vPosition);

  float nDotL = dot(normal, lightDir);
  vec3 reflection = normalize( 2.0 * normal * nDotL - lightDir);
  float rDotV = max(0.0, dot(reflection, viewDir));
  float specular = pow(rDotV, 64.0);
  float diffuse = max(nDotL, 0.0);

  vec3 intensity =
    uColor *
    (diffuse + 0.2) +
    vec3(0.4, 0.4, 0.4) * specular;
  fragColor = vec4(intensity.x, intensity.y, intensity.z, 1.0);
}
```

- The coordinates of the vector $\vec{v}$ are given by -vPosition since, in eye coordinates, the eye's position coincides with the origin.
- Also notice that, in this code, the specular component is white and is unrelated to any material color. This is quite typical for plastic materials, where the highlights match the color of the incoming light, regardless of the material color.
- 9.ogl.separatespecular
- gldemos.material

Anisotropy

- The two previous material models, as well as many others, have the property of *isotropy*.

- This means that there is no preferred "grain" to the surface.
- some materials, like brushed metal, behave in an *anisotropic* manner.
  - If one takes a flat sample of such a material and spins it around its normal, the material's appearance changes.
- When modeling such a material, we will assume that a preferred tangent vector, $\vec{t}$, is somehow included in the model and given to the shader.
  - bookfig
- Kajiya and Kay made up one model based on the assumption that the surface is made up of tiny cylinders;

demos

- 9.d3d.hlslanis/
- 9.ogl.glslthinfil

Texture Mapping

- we have already seen and used texture mapping
- In basic texturing, we simply "glue" part of an image onto a triangle by specifying texture coordinates at the three vertices.
- bunch of opengl code to load a texture and set various parameters (lin/const, mipmap, wrapping rules)
- a uniform variable is used to point to the desired texture unit,
- Varying variables are used to store texture coordinates
- In this simplest incarnation, we just fetch `r,g,b` values from the texture and send them directly to the framebuffer.
  - bookfig
- Alternatively, the texture data could be interpreted as, say, the diffuse material color of the surface point, which would then be followed by the diffuse material computation described earlier

Normal Mapping

- The data from a texture can also be interpreted in more interesting ways.
- In *normal mapping*, the `r,g,b` values from a texture are interpreted as the three coordinates of the normal at the point.
- This normal data can then be used as part of some material simulation ,
- Normal data has three coordinate values, each in the range $[-1..1]$, while RGB textures store three values, each in the range $[0..1]$.
- so need some conversions.
- sometime need to deal with coordinate system conversions.

Environment Cube Maps

- Textures can also be used to model the *environment* in the distance around the object being rendered.
- In this case, we typically use 6 square textures representing the faces of a large cube surrounding the scene.
- Each texture pixel represents the color as seen along one direction in the environment.
- This is called a *cube map*. GLSL provides a cube-texture data type, `samplerCube` specifically for this purpose.
- During the shading of a point, we can treat the material at that point as a perfect mirror and fetch the environment data from the appropriate incoming direction.
- note This calculation is not completely correct, since the environment map stores the view of the scene as seen *from one point only*.
  - bookfig

e-map shader

- we calculate $B(\vec{v})$.
- This bounced vector will point points towards the environment direction which would be observed in a mirrored surface.
- By looking up the cube map using this direction, we give the surface the appearance of a mirror.

```
uniform samplerCube texUnit0;

in vec3 vNormal;
in vec4 vPosition;

out fragColor;

vec3 reflect(vec3 w, vec3 n){
  return - w + n * (dot(w, n)*2.0);
}

void main(void){
  vec3 normal = normalize(vNormal);
  vec3 reflected = reflect(normalize(vec3(-vPosition)), normal);
  vec4 texColor0 = textureCube(texUnit0,reflected);

  fragColor = vec4(texColor0.r, texColor0.g, texColor0.b, 1.0);
}
```

- `-vPosition` represents the view vector $\vec{v}$.
- `textureCube` is a special GLSL function that takes a direction vector and returns the color stored at this direction in the cube texture map.
- here we assume eye-coordinates, but frame changes may be needed
- this can be used for refraction
- 9.d3d.fresnel
- 9.ogl.glsldispersion

Projector Texture Mapping

- There are times when we wish to glue our texture onto our triangles using a *projector* model, instead of the affine gluing model
- For example, we may wish to simulate a slide projector illuminating some triangles in space.
  - facade image reprojection
  - facade bookfig
- the slide projector is modeled using 4 by 4, modelview and projection matrices, M_s and P_s.
  - see bookfig

$$\begin{bmatrix} x_t w_t \\ y_t w_t \\ - \\ w_t \end{bmatrix} = P_s M_s \begin{bmatrix} x_o \\ y_o \\ z_o \\ 1 \end{bmatrix} \tag{2}$$

- with the texture coordinates defined as $x_t = \frac{x_t w_t}{w_t}$ and $y_t = \frac{y_t w_t}{w_t}$.
- To color a point on a triangle with object coordinates $[x_o, y_o, z_o, 1]^t$, we fetch the texture data stored at location $[x_t, y_t]^t$.
- the three quantities $x_t w_t$, $y_t w_t$ and w_t are all affine functions of (x_o, y_o, z_o). Thus these quantities will be properly interpolated over a triangle when implemented as varying variables. In the fragment shader, we need to divide by w_t to obtain the actual texture coordinates.

- When doing projector texture mapping, we do not need to pass any texture coordinates as attribute variables to our vertex shader.
- We simply use the object coordinates already available to us.
- We do need to pass in, using uniform variables, the necessary projector matrices.

projector shaders

```
#version 330
uniform mat4 uModelViewMatrix;
uniform mat4 uProjMatrix;

uniform mat4 uSProjMatrix;
uniform mat4 uSModelViewMatrix;

in vec4 aPosition;
out  vec4  vTexCoord;

void main(){
  vTexCoord= uSProjMatrix * uSModelViewMatrix * aPosition;
  gl_Position = uProjMatrix * uModelViewMatrix * aPosition;
}
```

```
#version 330

uniform sampler2D vTexUnit0;

in vec4 vTexCoord;
out fragColor

void main(void){
  vec2 tex2;
  tex2.x = vTexCoord.x/vTexCoord.w;
  tex2.y = vTexCoord.y/vTexCoord.w;
  vec4 texColor0 = texture2D(vTexUnit0, tex2);
  fragColor=  texColor0;
}
```

- Conveniently, OpenGL even gives us a special call `texture2DProj(vTexUnit0, pTexCoord)`, that actually does the divide for us.
- Inconveniently, when designing our slide projector matrix `uSProjMatrix`, we have to deal with the fact that the canonical texture image domain in OpenGL is the *unit square* whose lower left and upper right corners have coordinates $[0, 0]^t$ and $[1, 1]^t$ respectively, instead of the *canonical square* from $[-1, -1]^t$ to $[1, 1]^t$ used for the display window.
- demo: 9.ogl.hwshadowmap

Multipass

- More interesting rendering effects can be obtained using multiple rendering passes over the geometry in the scene.
- In this approach, the results of all but the final pass are stored offline and not drawn to the screen.
- To do this, the data is rendered into something called a *frameBufferObject*, or FBO.
- After rendering, the FBO data is then loaded as a texture, and thus can be used as input data in the next rendering pass.
- silly demo: 9.ogl.pbuffertotexture

reflection Mapping

- In this case, we want one of the rendered objects to be mirrored and reflect the rest of the scene.

- To accomplish this, we first render the rest of the scene as seen from, say, the center of the mirrored object.
- Since the scene is 360 degrees, we need to render six images from the chosen viewpoint, looking right, left, back, forth, up and down. Each of these six images has a 90 degree vertical and horizontal field of view.
- This data is then transferred over to a cube map.
- The mirrored object can now be rendered using the environment cube map above
- demo
- bookfig

Shadow Mapping

- The idea is to first create and store a z-buffered image from the point of view of the light, and then compare what we see in our view to what the light saw it its view.
- If a point observed by the eye is not observed by the light, then there must be some occluding object in between, and we should draw that point as if it were in shadow.
- bookfig
- In a first pass, we render into an FBO the scene as observed from some camera whose origin coincides with the position of the point light source. Let us model this camera transform as:

$$\begin{bmatrix} x_t w_t \\ y_t w_t \\ z_t w_t \\ w_t \end{bmatrix} = P_s M_s \begin{bmatrix} x_o \\ y_o \\ z_o \\ 1 \end{bmatrix}$$

  for appropriate matrices, P_s and M_s.
- During this first pass, we render the scene to an FBO using M_s as the modelview matrix and P_s as the projection matrix.
- In the FBO, we store, not the color of the point, but rather its z_t value.
- Due to z-buffering, the data stored at a pixel in the FBO represents the z_t value of the *geometry closest to the light* along the relevant line of sight. This FBO is then transferred to a texture.
- During the second rendering pass, we render our desired image from the eye's point of view, but for each pixel, we check and see if the point we are observing was also observed by the light, or if it was blocked by something closer in the light's view.
- To do this, we use the same computation that was done with projector texture mapping
- Doing so, in the fragment shader, we can obtain the varying variables x_t y_t and z_t associated with the point $[x_o, y_o, z_o, 1]^t$.
- We then compare this z_t value with the z_t value stored at $[x_t, y_t]^t$ in the texture.
- If these values agree then we are looking at a point that was also seen by the light; such a point is not in shadow and should be shaded accordingly. Conversely, if these values disagree, then the point we are looking at was occluded in the light's image, is in shadow, and should be shaded as such.
- demo: 9.ogl.hwshadowmap

bonus demos

- demo: 9.d3d.depthoffield
- demo: 9.gl.dynamicao