LIGHT SOURCES
LIGHTING

• So far we have seen one basic type of light that can create interesting lighting affects.

• Consider the following scene which cannot be created with the current light model.

• Why not?
BASIC LIGHT TYPES

- **Global Ambient Light** – a global color applied to all objects evenly
- **Directional Light** – an infinitely far away light source where light rays affecting the scene are all parallel
- **Point Light** – local lighting such that light rays affecting the scene are dependent on the lights position relative to the object
- **Spot Light** – local lighting such that light is focused in a direction from a point source.
MODELING GLOBAL AMBIENT LIGHT

• Global ambient light applies a base light intensity evenly to the whole scene

• Parameters:
  • $I_a$ – light intensity (RGBA)

• There is no direction to the light, because it comes from everywhere
MODELING DIRECTIONAL LIGHT

• Directional lights are infinitely far away causing light rays to be parallel within the scene.

• Parameters:
  • \( \hat{d} \) – direction vector
  • \( I_a, I_d, I_s \) - Light intensities for ambient, diffuse, and specular components (RGBA)

• Direction to the light is \( \hat{l} = -\hat{d} \)
ATTENUATION

- For local light sources, we can model **attenuation**, which is the concept that a light affects an object based on how far away from the light source it is.

- Types:
  - **Linear attenuation**, based on distance:
    \[ \alpha_l = \frac{1}{a_c + a_l d + a_q d^2} \]
    where \( d \) is the distance to the light source.
  - **Angular attenuation**, based on angle:
    \[ \alpha_a = (\hat{l} \cdot \hat{d})^{a_a} \]
    where \( \hat{l} \) is the direction from the light source and \( \hat{d} \) is the direction of the light source.
  - If both linear and angular attenuation are used, multiply them together: \( \alpha = \alpha_l \alpha_a \)
MODELING POINT LIGHTS

• Point Lights are local to the scene. A light affects objects based on their relative positions and distances.

• Parameters:
  • $\vec{p}$ – position
  • $I_a, I_d, I_s$ - Light intensities for ambient, diffuse, and specular components (RGBA)
  • $a_c, a_l, a_q$ – Linear attenuation constants

• Direction to the light is $\hat{l} = \frac{\vec{p} - \vec{x}}{||\vec{p} - \vec{x}||}$
MODELING SPOT LIGHTS

• Spot lights are local light sources that are directional

• Parameters:
  • $\mathbf{p}$ – position
  • $\mathbf{d}$ – direction
  • $\theta$ – cut-off angle
  • $I_a, I_d, I_s$ - Light intensities for ambient, diffuse, and specular components (RGBA)
  • $a_c, a_l, a_q$ – Linear attenuation constants
  • $a_a$ – Angular attenuation constant

• Direction to the light is $\hat{l} = \frac{\mathbf{p}-\mathbf{x}}{||\mathbf{p}-\mathbf{x}||}$

• Light cutoff is determined by $-\hat{l} \cdot \mathbf{d} > \cos \theta$
MODELING MATERIALS

• Recall the parameters defining a material for an object
  • $k_a, k_d, k_s$ – ambient, diffuse, and specular light coefficients
  • $\rho$ – specular exponent (shininess)
ADS LIGHTING MODEL

• Together, ambient + diffuse (Lambertian) + specular (Blinn-Phong) lighting yields the ADS lighting model
  • ADS is Ambient, Diffuse, Specular
• Now with attenuation $\alpha$

• For any light:
  • Ambient: $L_a = k_a I_a$
  • Diffuse: $L_d = \alpha k_d I_d \max(0, \hat{n} \cdot \hat{i})$
  • Specular: $L_s = \alpha k_s I_s \max(0, \hat{v} \cdot \hat{r})^\rho$

• For all lights:

$$L = \sum_{\text{Lights}} L_a + L_d + L_s$$

If using a global ambient, then lights do not contribute a local ambient portion.
EXERCISE

• Split up into teams and define a model for one of the following light types.
  • By define, I mean – pick the parameters that define the light itself (e.g., shape) and determine how to compute the $\hat{\mathbf{i}}$ vector for use in the ADS lighting model.
ARTISTIC STYLING
ARTISTIC STYLING

• Consider the following image

• What is different from the ADS model?
  • Outlining
  • Cool-to-warm shading
OUTLINING

• Difficult problem. Assume a triangular mesh and reason about adjacent triangles (sharing an edge) with normal \( n_0 \) and \( n_1 \)

• Consider some options:
  • Silhouette outline – boundary based on view
    • Highlight if \((\vec{v} \cdot \hat{n}_0)(\vec{v} \cdot \hat{n}_1) < 0\)
    • Only true with one front facing and one back facing triangle
  • Crease outline – sharpness of edges
    • Highlight if \(\hat{n}_0 \cdot \hat{n}_1 \leq threshold\)
  • Etc
OUTLINING

• Can also play tricks with multiple rendering passes

• Draw all front facing triangles first. Then draw all back facing triangle with distances slightly offset toward the viewer.
COOL-TO-WARM-SHADING

• A cool to warm shading gives a cartoon-like or artistic stylized look for an object.

• Easy to support.
  • Select one cool color $c_c$, e.g., blue, and one warm color $c_w$, e.g., orange.
  • Define weighting $k_w = \frac{1 + \hat{n} \cdot \hat{l}}{2}$
  • Define color $\tilde{c} = k_w c_w + (1 - k_w) c_c$
    as a weighted ratio based on the angle to a light.
GLSL DATA STRUCTURES
GLSL supports `structs`. `structs` simply group data members together:
- No methods, and can only be used with uniform data.

Example:
```cpp
struct A {
  vec3 x;
};
uniform A a;
```

Find location of the data in your application by: `glGetUniformLocation(program, "a.x")`

GLSL supports arrays as well:
- Must be statically allocated and uniform data.

Example:
```cpp
const int size = 10;
uniform A as[size];
```

Find location of the data in your application by: `glGetUniformLocation(program, "as[3].x")`
EXERCISE

• Write a GLSL struct for Material data

• Write a general GLSL struct for various light types
  • All 3 types of light can be supported with a single struct
TEXTURE MAPPING
TEXTURE MAPPING

• Say you want to visualize something like a sandy beach, wood flooring, a poster, etc.

• All of these possibilities have spatially varying surface properties — color, normal, etc.
  • These change an attribute of a surface but not its shape

• To model these, we perform texture mapping, using an image to store these details and map these values onto the surface.
TEXTURE MAPPING OVERVIEW

- We will first explore with colors, but texture mapping can be used for any properties varying across a surface, even things having nothing to do with an image.

- A big challenge with texture mapping is managing distortion and aliasing effects.

- Essentially, the problem boils down to performing a texture lookup to acquire a texture sample, e.g., for selecting a $k_d$ value in ADS lighting.
TEXTURE MAPPING BASICS

• Mathematically, we need to define a function that maps from the surface to the texture that we can compute for any pixel

• This is the **texture coordinate function** mapping a surface $S$ to the domain of the texture $T$

\[
\phi: S \rightarrow T : (x, y, z) \rightarrow (u, v)
\]

• $T$ is referred to as the **texture space**. Commonly it is the unit square $(u, v) \in [0,1]^2$

• Contrast this with $\pi$ the function mapping a surface to screen space

$$\pi: S \rightarrow \text{Screen Space}$$

$$\phi: S \rightarrow T$$

$$\pi: S \rightarrow \text{Screen Space}$$

(0,0) $u$ $v$

(1,1)
A STEP BACK

• Anyone find it odd that the mapping goes from the surface to the texture?

• Break it down from two perspectives
  • Artist perspective – needs to define all of the texture coordinates for the vertices of a model (i.e., define $\phi$)
  • Graphics engine perspective – needs to use the texture coordinates to look up in the image data
DEFINING TEXTURE COORDINATES

• Problem identical to cartographers designing maps that cover large areas of the Earth’s surface

• Ideally $\phi$ is defined with the following in mind:
  • **Bijectivity** – every point on the surface maps to a different place in the texture (but not always)
  • **Size distortion** – the scale of the texture should be approximately the same across the surface
  • **Shape distortion** – the texture itself should not be distorted. Said another way, a circle should map to a roughly circular shape in texture space.
  • **Continuity** – the number of seams should be limited
DEFINING TEXTURE COORDINATES

• Two general ways to match the perspectives
  • Compute them geometrically (used mostly by modeling programs for artists)
  • Store texture coordinates with vertices for quick lookup and interpolation (engine perspective)

• For methods that follow, small assumption that the object fits in the box $[-1,1]^3$

• These are the "vt"s in OBJ files.
• Your engines will read these and send them to a shader program interpolating them across a triangle
PLANAR PROJECTION

• Parallel projection of the object into texture space
  \[ \phi(x, y, z) = (u, v) \] where
  \[
  \begin{bmatrix}
  u \\
  v \\
  * \\
  1
  \end{bmatrix} = P_{ortho}
  \begin{bmatrix}
  x \\
  y \\
  z \\
  1
  \end{bmatrix}
  \]

  • Works well for flat objects with little variance in normal
  • Becomes injective for any closed shape

• Perspective projection of the object into texture space
  \[ \phi(x, y, z) = \left(\frac{\tilde{u}}{w}, \frac{\tilde{v}}{w}\right) \] where
  \[
  \begin{bmatrix}
  \tilde{u} \\
  \tilde{v} \\
  * \\
  w
  \end{bmatrix} = P_{persp}
  \begin{bmatrix}
  x \\
  y \\
  z \\
  1
  \end{bmatrix}
  \]

  • Great for shadow mapping

Example distortion with planar projection
SPHERICAL COORDINATES

• Can use the longitude and latitude (spherical coordinates \((\rho, \theta, \phi)\)) for a texture coordinate function

• Heave distortion near the poles

• Discontinuity along a single latitude

• Mapping:
  \[ \phi(x, y, z) = (u, v) \]
  \[ = \left( \frac{\pi + \text{atan2}(y, x)}{2\pi}, \frac{\pi - \text{acos} \left( \frac{z}{|x|} \right)}{\pi} \right) \]
CYLINDRICAL COORDINATES

• An option for more column-line objects is cylindrical coordinates

• Mapping:
\[ \phi(x, y, z) = (u, v) \]
\[ = \left( \frac{\pi + \text{atan2}(y, x)}{2\pi}, \frac{1 + z}{2} \right) \]
CUBE MAPS

- One way to avoid the distortions is to add discontinuity
- Cube maps provide an alternative to spherical mapping that allows textures to be a bit more uniform
- Pick face based on coordinate with the largest absolute value and the sign to select from there.

\[
\begin{align*}
\phi_{-x} (x, y, z) &= \frac{1 + (z, -y)}{2|x|} \\
\phi_{+x} (x, y, z) &= \frac{1 + (-z, -y)}{2|x|} \\
\phi_{-y} (x, y, z) &= \frac{1 + (x, -z)}{2|y|} \\
\phi_{+y} (x, y, z) &= \frac{1 + (x, +z)}{2|y|} \\
\phi_{-z} (x, y, z) &= \frac{1 + (-x, -y)}{2|z|} \\
\phi_{+z} (x, y, z) &= \frac{1 + (+x, -y)}{2|z|}
\end{align*}
\]
TEXTURE LOOKUPS

• Often texture coordinates go outside of the boundary of the image.

• Depending on the scenario, you want to handle this differently

• Options
  • Background color
  • Clamping
  • Wrapping
EXERCISE

• Define a lookup routine of \((u, v)\) for the three options
TEXTURE TRANSFORMATIONS

- Transformations can be a useful mechanism to support artists designing texture maps (e.g., translate + scale)
HANDLING CONTINUITY AND SEAMS

• Essentially the only solution is to duplicate the positional vertex data to separate the texture data.
ALIASING IN TEXTURE MAPPING

• A perspective projection introduces aliasing affects into texture mapping. Consider:
  • A pixel close to the camera has a very small footprint area in texture space
  • A pixel far from the camera actually has a very large footprint area in texture space
SOLUTIONS

• For footprints smaller than a pixel
  • Magnification

• For footprints larger than a pixel
  • Minification
MAGNIFICATION

• We need to upsample to perform magnification

• To options:
  • Select nearest texel
  • Bilinear combination of nearest 4 texels
EXERCISE

- Write an algorithm to perform bilinear sampling
MINIFICATION

• Downsampling would require knowing all of the texels filled by the footprint to determine an average.

• A better approximation is to perform mipmapping:
  • Store downsampled versions of an image for quick lookup. Always factors of two until the image is a single pixel large.
  • Essentially a level of detail mechanism.
  • Options are to select nearest level, or a linear combination (trilinear filtering) of closest two levels.

MIP stands for Multi in Parvo, Latin meaning a great deal in a small space.
EXERCISE

• Design an algorithm to automatically create a mipmap from an image.
  
  • Essentially, from a 2D array of colors create a new 2D array of colors such that it stores the mipmap
  
  • Need to determine how big the resultant array will be
ANISOTROPIC FILTERING

- **Anisotropic filtering** selects the mipmap level based on the shortest axis of the footprint instead of the largest and averages together several lookups.
PERSPECTIVE CORRECT INTERPOLATION

- Consider rasterization of texture coordinates based on a linear interpolation.
- An image like the following can occur
PERSPECTIVE CORRECT INTERPOLATION

• However, it should look like this

• Theories as of the difference?
PERSPECTIVE CORRECT INTERPOLATION

• Consider a scan line that occurs during rasterization.

• Note that the spacing on the projection plane is different than the spacing across the triangle.

• We need a non-linear interpolation!
PERSPECTIVE CORRECT INTERPOLATION

• Lets jump into the math to see how this should be done!

• Consider a scan line in this plane:

\[ ax + bz = c \]

• Given a point \( \langle x, z \rangle \) on the line, we can cast a ray to the origin and determine the intersection point on the view plane \( \langle p, -d \rangle \), where \( d \) is the distance to the view plane. Based on similar triangles (from perspective projection), we have:

\[
\frac{p}{x} = \frac{-d}{z}
\]
PERSPECTIVE CORRECT INTERPOLATION

• Solve for $x$:

$$x = -\frac{pz}{d}$$

• Plug into $ax + bz = c$:

$$\left(-\frac{ap}{d} + b\right)z = c$$

• For convenience, manipulate to this form:

$$\frac{1}{z} = -\frac{ap}{cd} + \frac{b}{c}$$
PERSPECTIVE CORRECT INTERPOLATION

- Consider the endpoints of the line segment \( \langle x_1, z_1 \rangle \) and \( \langle x_2, z_2 \rangle \) and their images on the view plane \( \langle p_1, -d \rangle \) and \( \langle p_2, -d \rangle \).

- Let \( p_3 = (1 - t)p_1 + tp_2 \) for some \( t \) in the range \( 0 \leq t \leq 1 \), be the \( x \)-coordinate of an interpolated point on the projection plane.

- We need to find the \( z \)-coordinate of the point \( \langle x_3, z_3 \rangle \) where the ray through \( \langle p_3, -d \rangle \) intersects the triangle.
PERSPECTIVE CORRECT INTERPOLATION

• Plug $p_3 = (1 - t)p_1 + tp_2$ into the equation \( \frac{1}{z} = -\frac{ap}{cd} + \frac{b}{c} \):

\[
\frac{1}{z_3} = -\frac{ap_3}{cd} + \frac{b}{c} = -\frac{a}{cd}p_1 (1 - t) - \frac{ap_2}{cd} t + \frac{b}{c}
\]

• Add \( \left(\frac{b}{c} - \frac{b}{c}\right) \) to the equation to reform:

\[
\frac{1}{z_3} = \left(-\frac{ap_1}{cd} + \frac{b}{c}\right)(1 - t) + \left(-\frac{ap_2}{cd} + \frac{b}{c}\right)t = \frac{1}{z_1} (1 - t) + \frac{1}{z_2}t
\]

• It interpolates with the reciprocal of the depth!
Now consider, vertex attributes, like a texture coordinate $\vec{t}$

Given two depth values $z_1$ and $z_2$ and attribute values $\vec{t}_1$ and $\vec{t}_2$, we expect an interpolated value $\vec{t}_3$ to have depth $z_3$ with the following relation:

$$\frac{\vec{t}_3 - \vec{t}_1}{\vec{t}_2 - \vec{t}_1} = \frac{z_3 - z_1}{z_2 - z_1}$$
PERSPECTIVE CORRECT INTERPOLATION

• Substituting our prior determination of $z_3$ into this equation and solving for $\hat{t}_3$:

$$\hat{t}_3 = \frac{\hat{t}_1 z_2 (1 - t) + \hat{t}_2 z_1 t}{z_2 (1 - t) + z_1 t}$$

• Multiply numerator and denominator by $\frac{1}{z_1 z_2}$:

$$\hat{t}_3 = \frac{\frac{\hat{t}_1}{z_1} (1 - t) + \frac{\hat{t}_2}{z_2} t}{\frac{1}{z_1} (1 - t) + \frac{1}{z_2} t} = z_3 \left( \frac{\hat{t}_1}{z_1} (1 - t) + \frac{\hat{t}_2}{z_2} t \right)$$

• So what is the point? While $\hat{t}$ cannot be interpolated across the triangle, $\frac{\hat{t}}{z}$ can be!

After interpolation, divide by $\frac{1}{z}$
Final piece of the puzzle. Carry along an extra 1 into the perspective divide to give you the reciprocal of the depth to use for perspective correction. Recall the perspective divide and that $w$ would store the depth value:

$$\begin{bmatrix}
    u \\
    v \\
    1 \\
    x \\
    y \\
    z \\
    w
\end{bmatrix} \rightarrow \begin{bmatrix}
    u/w \\
    v/w \\
    1/w \\
    x/w \\
    y/w \\
    z/w \\
    1
\end{bmatrix}$$

This type of interpolation is performed automatically on the GPU.
OPENGL TEXTURE METHODS
FIRST STEP

• A first step will be to import images into your program. Options:
  • Qt's QImageReader
  • Simple OpenGL Image Library

• You will need to associate texture coordinates with vertex data (e.g., from OBJ file)
SENDING THE IMAGE DATA TO THE GPU

• This is typically handled internally to SOIL, for example. Also, a slightly different version will be used for automatic mipmaps.

• Generate a name for the texture:
  Gluint tid;
glGenTextures(1, &tid);

• Bind and copy data:
  glBindTexture(GL_TEXTURE_2D, tid);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB,
width, height, 0, GL_BGR, GL_UNSIGNED_BYTE, data);
SENDING THE IMAGE DATA TO THE GPU

• Other versions of the functions exist.

• You need to extrapolate based on the library, mipmapping, etc.

• Recommendations:
  • Use SOIL2 library
  • Encapsulate creation of image data and id in a function or class.
USING THE TEXTURE IN A SHADER

• Must have a uniform sampler variable:
  ```
  uniform sampler2D tex_sampler;
  ```

• Then to use invoke the build-in texture function:
  ```
  vec2 tex_coord;
  vec4 color = texture(tex_sampler, tex_coord);
  ```
SETTING UP SAMPLERS FROM THE APPLICATION

• Set an active texture and bind:
  `glActiveTexture(GL_TEXTURE0); // OR GL_TEXTURE_i`
  `glBindTexture(GL_TEXTURE_2D, tid);`

• Setup the sampler by sending a uniform integer with value 0 (or i)
  • This means a `glGetUniformLocation()` and `glUniform1i()` pair

• The number of texture units depends on the GPU
WRAPPING AND TILING

• Bind the texture and:
  
  ```
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
  ```

• Options:
  • GL_REPEAT – repeating pattern
  • GL_MIRRORED_REPEAT – alternating pattern between normal and reversed
  • GL_CLAMP_TO_EDGE – clamp to [0,1]
  • GL_CLAMP_TO_BORDER – border color that can be separately set

• Must set the wrapping/tiling setting separately for the second coordinate with
  GL_TEXTURE_WRAP_T

• Default is GL_REPEAT
SETTING MINIFICATION PARAMETER

• Bind the texture and:
  `glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);`

• Other options
  • `GL_NEAREST_MIPMAP_NEAREST` – nearest texel from nearest mipmap
  • `GL_LINEAR_MIPMAP_NEAREST` – bilinear filter on the nearest mipmap
  • `GL_NEAREST_MIPMAP_LINEAR` – linear filter between two closest mipmap levels
  • `GL_LINEAR_MIPMAP_LINEAR` – trilinear filtering
  • `GL_NEAREST` – no mipmap, nearest texel
  • `GL_LINEAR` – no mipmap, bilinear filtering

• Also, can set the `GL_TEXTURE_MAG_FILTER` with `GL_NEAREST` or `GL_LINEAR` for magnification parameter
ANISOTROPIC FILTERING

• If your graphics card supports it. You can turn this on by:
  
  ```
  glTexParameterf(GL_TEXTURE_2D, 
                  GL_TEXTURE_MAX_ANISOTROPY_EXT, anisoSetting);
  ```

• This is in an extension, read more to provide necessary precautions for use
  (i.e., if-statement determining support level)
MOST IMPORTANTLY

• Read chapter 5 of Computer Graphics Programming!

• Understand what concept you want, and then look up the appropriate API call or color format
ADVANCED TEXTURE MAPPING
AS MENTIONED PREVIOUSLY...

• Texture mapping is extremely powerful in creating detail.

• It conceptually imports arbitrary surface data for lookup to alter a shader's output.

• Consider:
SPECULAR MAPPING

• To perform **specular mapping**, in the fragment shader – look up the specular component $k_s$ from a texture.
**BUMP MAPPING**

- **Bump mapping** can refer to an altering of the normals along a surface. A most basic version would be to alter the magnitude of the normal by a value looked up in a texture – simple scaling.
NORMAL MAPPING

• However, bump mapping alone is outdated, and limited. A better approach is to specify the normals directly with texture information, i.e., normal mapping.
NORMAL MAPPING

• Lets look at how normal mapping works.

• A normal map stores the normal along the surface, but stores it in a different coordinate frame, than you expect. The space is called **tangent space** and is a coordinate frame perpendicular to the model surface.

• We need to convert the normal from this space into the camera space for lighting. Then the ADS model proceeds as expected.

• To convert a color to a normal:

\[ \vec{n} = 2\vec{c} - 1 \]

• Can you see why the map is very blue?
NORMAL MAPPING

• Tangent space is a space in which the normal of a surface is always the z-axis. This is unique at every point (vertex) on the surface.

• The other components of the coordinate frame are the tangent and bitangent vectors.

• The three vectors together allow us to convert a normal to camera coordinates.
NORMAL MAPPING

• To compute the tangent and bitangent vectors for a triangle, we can express two edges of a triangle in tangent space:

\[ E_1 = \Delta u_1 \hat{t} + \Delta v_1 \hat{b} \]
\[ E_2 = \Delta u_2 \hat{t} + \Delta v_2 \hat{b} \]
NORMAL MAPPING

• We can reform this as a matrix multiplication:

\[
\begin{bmatrix}
E_1 \\
E_2
\end{bmatrix} = \begin{bmatrix}
\Delta u_1 & \Delta v_1 \\
\Delta u_2 & \Delta v_2
\end{bmatrix} \begin{bmatrix}
\vec{t} \\
\vec{b}
\end{bmatrix}
\]

• And we can invert the texture matrix to solve for \(\vec{t}\) and \(\vec{b}\):

\[
\begin{bmatrix}
\Delta u_1 & \Delta v_1 \\
\Delta u_2 & \Delta v_2
\end{bmatrix}^{-1} \begin{bmatrix}
E_1 \\
E_2
\end{bmatrix} = \begin{bmatrix}
\vec{t} \\
\vec{b}
\end{bmatrix}
\]

• To approximate the vertex tangent and bitangent, we can average the tangent/bitangent's of all adjacent triangles. Additionally, orthonormalize them.
NORMAL MAPPING

• Technically speaking, we only need the tangent vector as \( \vec{b} = \vec{n} \times \vec{t} \)
• The tangent vector is another vertex attribute in your vertex data
• Now to put it together in a shader program:
  • In the vertex shader
    • Form the coordinate transform to place the tangent space in camera coordinates (with the inverse-transpose of the model-view matrix). This matrix, called the TBN matrix can be interpolated across the triangle
  • In the fragment shader
    • Look up the normal from the texture
    • Use the TBN matrix to morph it into camera space in order to perform lighting computations
• Alternatively, you can transform lighting and view coordinates in the vertex shader into tangent space and perform ADS lighting in this frame of reference in your fragment shader (can potentially be faster)
NORMAL MAPPING

• Normal mapping can be made simpler with the Assimp library, as it computes tangents for models that do not come with them.
PARALLAX MAPPING

- Parallax mapping is a technique for offsetting texture coordinate lookups based on height values (just like parallax viewing in physics)
DISPLACEMENT MAPPING

• Unlike the previous techniques that do not alter the surface geometry, *displacement mapping* alters the shape of a surface.

• This can be implemented with tessellation and geometry shaders.
GENERATING TEXTURE MAPS

• **Physically-based Rendering** – highly enhanced lighting model based in physical light interactions. Usually helps "bake" texture maps and combine other features like "roughness"
SHADOW MAPPING
WHY IS THIS PICTURE NOT REALISTIC?

• Spot all of the ways it is lacking detail?

• Main problem – you can’t tell if the object is on the plane or not.

• Solution – shadows!
HOW ABOUT NOW?
SHADOWING IN A RAY TRACER

• To compute shadows, perform an additional ray trace during color determination from the point of collision to the light source.

• If an object is hit, then this light does not interact at that point, and only ambient light is added (or none), i.e., it is in the shadow.
SHADOW MAPPING TECHNIQUES

• Projective shadows use the projective transform to map a surface onto a plane
  • Possibly can be precomputed or an object and drawn like any other polygon with another color.
• Advantages/disadvantages?
**SHADOW MAPPING TECHNIQUES**

- Shadow volume techniques identify the volume of space which is cast in shadow and darkens objects within the volume.
  - Can be implemented using geometry shaders and the stencil buffer.
- Advantages/disadvantages?
SHADOW MAPPING
OVERVIEW

• Observation: anything that cannot be "seen" by a light is in the shadow
• This seems really similar to z-buffering!
• Why?
SHADOW MAPPING
OVERVIEW

• Shadow mapping renders the scene twice (called passes)
  • Pass one – render the scene from the perspective of the light
  • Copy the depth buffer into a texture map
  • Pass two – render the scene normally, but with an altered fragment shader that incorporates shadow information
SHADOW MAPPING
PHASE ONE

• Configure the frame buffer to only save depth into a shadow texture
• Disable color output
• Rasterize objects from the light's view
**SHADOW MAPPING**  
**PASS ONE SHADERS**

**Vertex Shader**
```glsl
#version 330

in vec3 vpos;

uniform mat4 shadowMVP;

void main() {
  gl_Position = shadowMVP * vec4(vpos, 1);
}
```

**Fragment Shader**
```glsl
#version 330

void main() {}  
```
SHADOW MAPPING
PASS TWO

• We will render the entire scene again, but use the shadow texture when lighting
• We will use the same transformation from the first phase with an additional adjustment — converting from camera space to texture space for our depth values

\[ \text{shadowMVP}_2 = B \ast \text{shadowMVP}_1, \]

where

\[
B = \begin{bmatrix}
0.5 & 0 & 0 & 0.5 \\
0 & 0.5 & 0 & 0.5 \\
0 & 0 & 0.5 & 0.5 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

Note, this is a matrix version of normal to color conversion:

\[
\vec{c} = \frac{n+1}{2}
\]
SHADOW MAPPING
PASS TWO

• Enable shadow texture for lookup
• Enable color output
• Pass along the shadow matrix
• Perform regular operations (e.g., construct MVP matrix, render all objects, etc.)
SHADOW MAPPING
PASS TWO VERTEX SHADER

#version 330

// ...all the normal in/out/uniform variables

out vec4 shadowCoord;
uniform mat4 shadowMVP2;

void main() {
    // ...all the normal vertex operations for out data
    shadowCoord = shadowMVP2 * vec4(vpos, 1);
}
#version 330

// ...all the normal in/out/uniform variables

in vec4 shadowCoord;

layout (binding=0) uniform sampler2DShadow shTex;

void main() {
    // ...all the normal setup of l, n, v, h
    float notInShadow = textureProj(shTex, shadowCoord);
    if (notInShadow == 1.0) {
        // ...all the normal lighting. We are not in the shadow.
    } else {
        // ...only the ambient lighting. We are in the shadow.
    }
}

OpenGL has a special sampler for shadow mapping. The binding relays which GL_TEXTUREi unit to use, instead of passing a uniform from the application.

Essentially performs a texture lookup and depth comparison. 1.0 means the depth of the fragment is closer to the light, 0.0 means it is in the shadow.
SHADOW MAPPING
APPLICATION

- Needs to provide the glue and the direction that orchestrates the phases.
- Over the next few slides, I highlight some key aspects to the application side.
  Please see the book for a full treatment.
SHADOW MAPPING APPLICATION

• Creating the shadow buffer – where the result of pass one goes
  
  GLuint sfb;
  glGenFramebuffers(1, &sfb);

• Creating the shadow texture
  
  GLuint shadowTex;
  glGenTextures(1, &shadowTex);
  glBindTexture(GL_TEXTURE_2D, shadowTex);
  glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT32, screenWidth, screenHeight, 0, GL_DEPTH_COMPONENT, GL_FLOAT, 0);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE, GL_COMPARE_REF_TO_TEXTURE);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_FUNC, GL_LEQUAL);

  Note, the internal format/format is a depth component only
  
  This sets up a comparison of depth for the textureProj method.
SHADOW MAPPING
APPLICATION

• To setup the light view matrix: choose the eye as the light position and the at vector to point to the center of the scene
• To setup the light projection matrix: choose a perspective view with an appropriate angle and far depth
• Then the shadow matrix per model will be:

\[ lightP \ast lightV \ast M \]

where \( M \) is the model matrix
SHADOW MAPPING APPLICATION

• To orchestrate the passes:
  // Set the framebuffer of pass one to output depths to a texture
  glBindFramebuffer(GL_FRAMEBUFFER, sfb);
  glFramebufferTexture(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, shadowTex, 0);
  glDrawBuffer(GL_NONE);  // No colors
  glEnable(GL_DEPTH_TEST);  // Depth!

  passOne();

  // Restore default targets
  glBindFramebuffer(GL_FRAMEBUFFER, 0);
  glActiveTexture(GL_TEXTURE0);  // Recall binding=0 from fragment shader
  glBindTexture(GL_TEXTURE_2D, shadowTex);
  glDrawBuffer(GL_FRONT);  // Color!

  passTwo();
• Nothing special about the application side of `passOne()` and `passTwo()` except to ensure that the correct `shadowMVP` matrix is sent to the GPU
EXERCISE

• How can you perform shadow mapping for multiple lights?
• How can you perform shadow mapping for a directional light?
SHADOW MAPPING

ALIASING

- Shadow acne
- Thoughts as to the cause?
- To solve:
  - Use `glEnable(GL_POLYGON_OFFSET_FILL)` and `glPolygonOffset()` to adjust triangle depth to be closer to the light
SHADOW MAPPING
ALIASING

• Peter panning

• Thoughts as to the cause?

• To solve:
  • Adjust the polygon offset – too small and shadow acne is seen, while too large and peter panning occurs

• For repeated shadowing, set texture wrapping policy to clamp
SHADOW MAPPING

ALIASING

• Jagged shadow edges

• Thoughts as to the cause

• To solve:
  • Move the light closer to the scene
  • Soft shadows!
WHAT IS DISTINCTIVELY DIFFERENT HERE?

• Shadows are **soft**, i.e., there is a **penumbra** affect
  
  • A shadow is softer the further from the light it is
A simple technique exists to generate soft shadows called Percentage Closer Filtering.

In this, we sample the shadow map at several locations to estimate a percentage within the shadow.

This is used to lighten/darken the light contribution.
SOFT SHADOWING
PERCENTAGE CLOSER FILTERING (PCF)

• Look up texels within a neighborhood to determine percentage

• Policy choices:
  • All texels within a "radius"
  • Sample nearby texels
  • Dithering

• Small modification to fragment shader required
  • Don’t forget to multiply the diffuse/specular components by the non-shadow percentage.
EXERCISE

• How can you use multiple passes to solve the following?
  • Rasterizing a reflective surface
  • Determining which object a user clicks on
  • Truly minimize the lighting computations done (1 per pixel, not 1 per fragment)