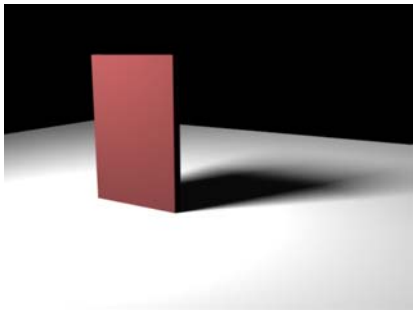
Shadow & Light

- Shadows
  - Shadow is composed of two regions
    - Umbra
    - Penumbra
  - Umbra is the solid region
    - Light completely blocked
  - Penumbra is the transition around the umbra
    - Light partially blocked
    - Only happens with extended light sources
  - Shadow is a volume
  - Cross-section of shadow volume is a projection of the object as seen from the light source



Shadow & Light

- Hard shadows
  - Umbra only
  - Assumes point light sources
- Soft shadows
  - Umbra and penumbra
- Shadows easily rendered using raytracing
  - For every point, trace ray to light and check for intersection with intermediate objects
  - For extended light sources, trace multiple rays to various points on the light source and shade according to the number of rays not blocked
- But polygon rendering does not do raytracing, so how to render shadows with OpenGL and the like?
  - Various methods which work in various cases, some hard shadows, some soft shadows, some fast, and some slow
  - There is no perfect shadow algorithm (yet) that always works

Shadow & Light

- Projective shadow textures
  - Easy but quite limited
  - Texture is an image containing the projection of the object as seen from a light direction
  - Create a texture which looks like the shadow of an object
    - Texture can be pre-computed or pre-made if relative position of light to object remains constant, and object doesn't change
    - Otherwise, could use render-to-texture methods to render view of object from light direction
  - Shadow texture has alpha set to opaque for shadow, transparent for the rest
  - To project the shadow onto another object, use projective texturemapping
  - Can fake smooth shadows by using a low-resolution texture
    - Texture interpolation between opaque and transparent areas of texture creates a fake penumbra effect

Shadow & Light

- Projective texturing:
  - Use `glTexGen*()` to let OpenGL compute texture coordinates based on vertex position (`GL_OBJECT_LINEAR`)
  - Use texture matrix to scale and position the shadow texture to match the object position, and orient the shadow texture to match the light direction

Shadow & Light

- For example: shadows of objects on terrain
  - Render terrain as normal, with textures etc.
  - Enable alpha blending
  - For each object
    - Make shadow texture of object current
    - Set projective texturemapping parameters for object position
    - Re-render terrain polygons which are likely to intersect the shadow, without any texture, purely black or gray
  - Render all objects on terrain
- Advantages of projective shadow textures:
  - Fast and easy
  - Smooth shadows
  - Good for shadows on terrain, floors, walls
- Disadvantages:
  - No self-shadowing
  - Projective shadow applied to both sides of an object
  - Shadow applied to all objects, even if between light and shadowing object

Shadow & Light

- Could improve on projective texture shadows if we can figure out if a point is in front or behind a shadow-casting surface as seen along the light direction
- The shadow mapping idea:
  - Assume spotlight or directional light
  - Render scene from a light position
  - Store depthmap in a texture (“shadow map”)
  - Render scene from camera
    - Project shadow map onto scene from light position
    - For each pixel being rendered, figure out distance to light
      - Use texture coordinate generation to set texture coordinates based on distance from light position instead of distance from origin
    - Compare distance with value from projected shadow map
    - If pixel↔light distance is less than shadow map distance, render pixel as normal, otherwise render shadowed

Shadow & Light

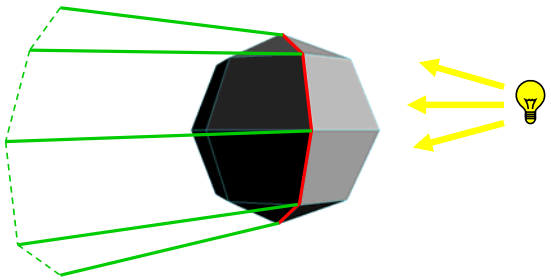
- Indexed shadow mapping
  - Instead of creating a shadow map with depths, store a unique item ID for each object seen by the light
  - When rendering scene from camera, for each pixel compare ID of object being rendered with ID in shadow map
  - If ID match, object is seen by light at that pixel, and hence pixel is lit
  - No self-shadowing

Shadow & Light

- Texture-mapping based shadows suffer from resolution problems
- Can we do shadows with geometry instead of textures?
- Yes: shadow volumes
  - A shadow is a volume
  - Construct that volume
  - Somehow figure out for each point being rendered if it is inside or outside a shadow volume

Shadow & Light

- Constructing a shadow volume
  - Shadow volume is formed by extruding the outer edges of an illuminated surface
  - The outer edges are called “silhouette edges”
    - Silhouette edges for convex objects are edges between a polygon which faces towards the light and a polygon which faces away from the light

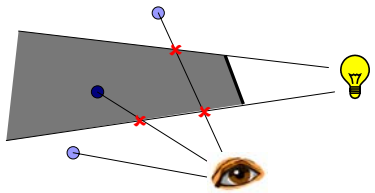


Shadow & Light

- Silhouette edges found by computing angle between polygon normals and vector from polygon to light using a dot product
  - If > 0 for one polygon of edge, and < 0 for other polygon of edge, edge is a silhouette edge
- For each silhouette edge, extrude a quad from edge vertices to infinity (or far enough) away from light along rays from light to edge vertices
- Volume is capped at the front by the polygons facing the light

Shadow & Light

- Assume camera is outside of the shadow volume
- Finding if a point is inside or outside shadow volume:
  - Take a ray from the point to the camera
  - Count how many times ray intersects shadow volume boundary
  - If odd, point is inside shadow; otherwise outside



Shadow & Light

- Counting can be done in hardware by OpenGL
  - Only need to figure out for unoccluded points visible from camera
  - Only need to count boundaries between point and camera
  - Counting of boundary crossings can be done in any order
  - Consider the shadow volume as an object being rendered as part of the scene
  - For each pixel, count how many boundaries are drawn in front of the point when seen from camera
  - Actually, don’t need the count, only the difference between number of times ray from point to camera enters shadows an number of times ray exits shadow
    - If difference is zero, point outside shadow, else inside shadow
      - Remember that camera is assumed to be outside shadow

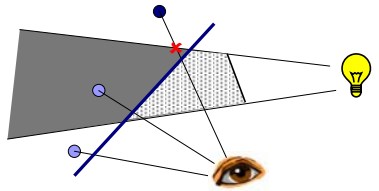
Shadow & Light

1. Draw the scene objects using only ambient lighting
   - Entire scene is in shadow
2. Disable colour and depth map writing, but keep depth testing on
3. Enable writing to the stencil buffer
4. Render front-facing polygons of shadow volume
   - Increment stencil buffer for every pixel rendered
5. Render back-facing polygons of shadow volume
   - Decrement stencil buffer for every pixel rendered
6. Enable colour buffer writing, disable stencil buffer writing
7. Enable depth testing to pass only equal depth
8. Enable stencil buffer testing to pass only pixels where stencil value is equal to 0
9. Draw the scene objects using the light
   - Overwrites pixels not in shadow with lit pixels

- Can extend to multiple lights by repeating for each light, and combining the results by accumulating the renders in an accumulation buffer, or with additive blending
- Can also start off with fully lit scene, then only overdraw pixels in shadow with ambient light only
- OpenGL2.0 allows separate stencil functions for front- and back-facing polygons, so steps 4 and 5 can be done in one pass

Shadow & Light

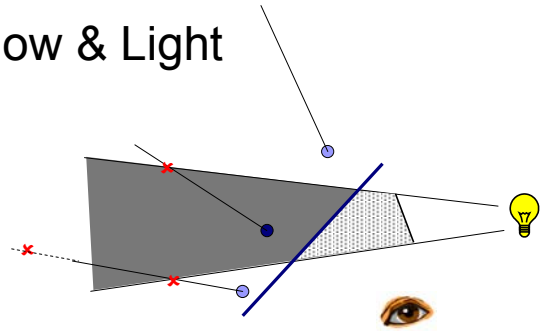
- Problem: near clipping plane may remove shadow volume boundaries near the camera
  - Results in wrong count, and hence either a missing shadow or a false shadow
  - Happens for shadows near the camera
  - Stuff near the camera is also the most important



Shadow & Light

- Fixing this is difficult
  - Several methods have been proposed
  - All have problems (don't work in some cases, precision problems, computationally expensive)
- This shadow mapping method is called "z-pass", as it counts the shadow boundaries that pass the usual depthmap test
  - That is, shadow boundaries between point and camera
- But for point-in-polyhedron testing, counting the boundaries along *any* ray from the point will work
  - Use the ray from the point *away* from the camera
  - No more near clipping plane problem!
- Instead of counting boundaries which pass the depth test (between point and camera), count boundaries which fail the depth test (between point and infinity away from the camera)
  - Called the "z-fail" method, or "Carmack's reverse"
- Added advantage: also works for camera inside shadow volume

Shadow & Light



- But, ray-volume intersection may be very far away, beyond far clipping plane
  - Same problem as z-pass, but this one is solvable
- First step: cap the volume at the far end

Shadow & Light

- Cap the shadow volume at the far end of the volume
  - Make cap by projecting polygons of shadow-casting object which face away from the light
  - Place cap far away enough so that all points which should be in shadow are within the now capped volume
  - Cap must be no further than the far clipping plane
- Wouldn't it be nice if the far clipping plane could be set at infinity…
  - But it can!

Shadow & Light

- Standard projection matrix as made by `glFrustum()` or `gluPerspective()`:

$$\mathbf{P} = \begin{bmatrix} \frac{2\times Near}{Right-Left} & 0 & \frac{Right+Left}{Right-Left} & 0 \\ 0 & \frac{2\times Near}{Top-Bottom} & \frac{Top+Bottom}{Top-Bottom} & 0 \\ 0 & 0 & -\frac{Far+Near}{Far-Near} & -\frac{2\times Far\times Near}{Far-Near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

  - Note: typically frustum is now skewed, so Right+Left = Top+Bottom = 0
- Find the limit when *Far* goes to infinity:

$$\lim_{Far\to\infty} \mathbf{P} = \mathbf{P}_{\inf} = \begin{bmatrix} \frac{2\times Near}{Right-Left} & 0 & \frac{Right+Left}{Right-Left} & 0 \\ 0 & \frac{2\times Near}{Top-Bottom} & \frac{Top+Bottom}{Top-Bottom} & 0 \\ 0 & 0 & -1 & -2\times Near \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Shadow & Light

- How to specify a vertex at infinity?
  - Remember homogeneous coordinates?
    - Vertex given by v = (x,y,z,w)
    - Normally would set w to 1
  - Vertex set at infinity by setting w = 0
    - v = (x,y,z,0), apply projection matrix, v' = (x',y',-z,-z)
    - Homogeneous division: v_window = (-x'/z, -y'/z, 1) where 1 is the value that is put in the depthmap (range [0..1])
- What about depthmap precision
  - Far clipping plane normally used to limit the depth range which is to be mapped to depthmap range (typically 16 or 24 bits)
  - Depthmap range stretched over the range of non-homogeneous (after homogeneous division) z values
  - How much extra does the depthmap have to be stretched by to include point at infinity compared with using a far plane?
  - Window depth for (0,0,-1,0) using normal projection P: Far/(Far-Near)
  - Window depth for (0,0,-1,0) using infinite projection P_{INF}: 1
  - So for example if Near = 1, Far = 100, depthmap precision squeezed by only 1%
  - Effect on depthmap precision is very small!

Shadow & Light

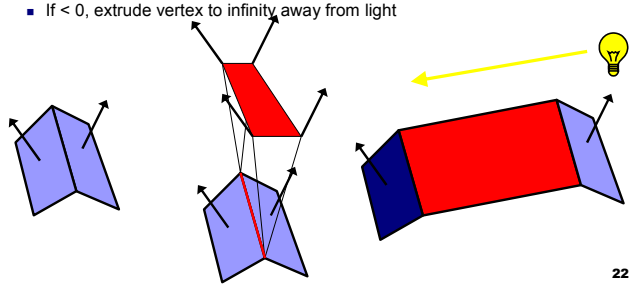
- Soft shadows with shadow volumes
  - Point in shadow volume test gives a pass or fail, resulting in hard shadows
  - For soft shadows, repeat the process with the light position changed slightly every time
  - Accumulate the results from each pass
  - Can cause a staircase effect in the penumbra of the shadow due to the discrete sampling of the light position

Shadow & Light

- Advantages of shadow volumes:
  - More accurate than texture-based methods
  - Self-shadowing
- Disadvantages:
  - More geometry to render
  - Must find silhouette edges
  - Must construct shadow volume
  - Must be careful not to get gaps in volume boundary due to numerical precision errors
  - Expensive soft-shadows

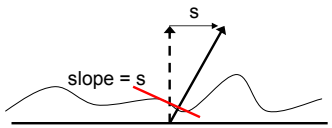
Shadow & Light

- Can actually construct shadow volume using a vertex shader
  - Vertex shaders can not create new vertices
  - As a pre-process, replace each edge of a potentially shadow-casting object with a zero-sized quad
  - Assign the normal of the polygons adjacent to edge of quad to the vertices of the quad edge
  - In vertex shader, compute dot product of vertex with vertex→light vector
    - If < 0, extrude vertex to infinity away from light



Shadow & Light

- Bump mapping
  - Polygons are flat
  - Any texture applied to a polygon is going to look flat
  - Bump mapping fakes bumps on a surface by slightly changing the normal per pixel
  - Consider a texture on a polygon as a height field
  - The change in the normal at a point from the polygon normal is given by the slope of the height field
  - Closely related to normal maps
    - Normal map contains the direction of the normal as perturbed by the bumps

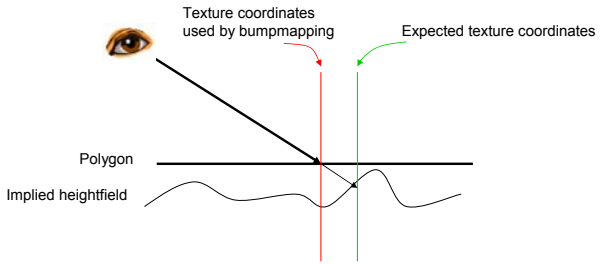


Shadow & Light

- Now quite simple to do in a fragment shader
- Previously used multi-texturing, for example:
  - Assume a diffuse lighting model
    - Light intensity is dot product of normal vector with vector to light
  - When creating a vertex, compute vertex→light vector
  - Use the vector as texture coordinates for texture unit 0
  - OpenGL will interpolate texture coordinates over polygon. If we interpret the interpolated texcoords as a vector, it will most likely no longer be normalised
  - Correct for loss of normalisation by putting a normalisation cubemap in texture unit 0
    - Cubemap contains texture whose RGB values are the components of the normalised vector as a function of vector direction
  - Compute normal map from bumpmap
  - Put grayscale normalised normal map in texture unit 1
  - Set texture unit 1 to do a dot product of the normal map with the result from texture unit 0 (the normalised interpolated vector to the light)
  - Put a colour texture in texture unit 2
  - Set texture unit 2 to modulate the texture with the intensity result from unit 1
- Also note the existence of the GL_NORMAL_MAP texture coordinate generation mode in OpenGL1.3

Shadow & Light

- Bump mapping uses the wrong texture coordinates
- Can be fixed by using a fragment shader to compute per-pixel texture offset
  - "Parallax mapping"



Shadow & Light

http://www.infiscape.com/rd.html

Shadow & Light

- Parallax mapping fails when height range is large, or at shallow viewing angles
- The ultimate solution: use raytracing to find intersection of ray with heightfield
  - Optimise a bit: do a binary search along the ray over the heightfield map to find texel where ray intersects heightfield
  - Assumes there is only one reasonable intersection point
  - "Relief mapping"
  - Can do the same thing to compute shadows
    - Use ray from light to point on surface being shaded

http://www.paralelo.com.br/arquivos/ReliefMapping.pdf

Shadow & Light

- Can place a heightfield at both sides of the polygon, and mask out uninteresting areas

Shadow & Light

- Environment mapping
  - Makes things look nice and shiny by reflecting the world
  - Create a cubemap of the environment
    - Static texture
    - Created dynamically with render-to-texture
  - Let OpenGL compute the texture coordinates for reflecting object using the GL_REFLECTION_MAP texture coordinate generation mode
    - Sets texture coordinates of vertex to eye-space reflection vector
  - Modulate with the object's base colour
  - Probably also want to add some specularity to the object surface material properties

Shadow & Light

Shadow & Light

- Lens flares
  - Lens flares only happen with real cameras
  - Lens flares are centred on a line that goes from the projected light position through the middle of the screen
  - Lens flares always appear in front of everything else, as they originate inside the camera
  - Compute the projected position of the light
  - If the light is visible, draw some quads textured with various lens flares along the line
    - Set spacing between quads proportional to the distance between the projected light position and the middle of the screen
    - Draw with alpha blending on top of the rendered scene (turn off depth testing)
  - For added effect, blend in screen-sized white quad according to how much of the light is visible for a haze effect

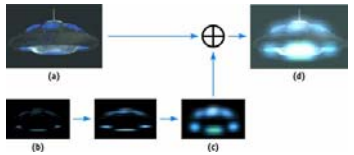
Shadow & Light

Shadow & Light

- Finding if the light is visible, and by how much
  - Render the scene
  - Compute projected light position
  - Read back area around light from colour or depth buffer
  - Count how many pixels which should overlap with light are not painted on by scene
  - Light intensity is proportional to fraction of pixels not covered by the scene
- But read-back can be slow
  - Introduces a stall, as your program has to wait for OpenGL to finish rendering the scene
  - Alternatively, compute visibility on CPU

Shadow & Light

- Glows
  - Render objects that need to glow
    - Use alpha channel to indicate glow source intensity from 0 (no glow) to 1 (lots of glow)
    - Render with blending set to multiply source colour with source alpha
  - Read back colour buffer
    - Or use render-to-texture
  - Blur the read back buffer
    - Run a blur filter over the image
    - Or if using render-to-texture, render to a low resolution texture so that scaling it up will blur it due to interpolation
  - Render the scene as normal
  - Place blurred glow image in texture (if not in texture already)
  - Blend the blurred glow image with the render by drawing screen-sized quad with blurred glow texture
    - Instead of blurring the glow image, can use the unblurred glow texture and draw the quad multiple times, each slightly offset



http://www.gamasutra.com/features/20040526/james_01.shtml