Introduction to Computer Graphics with WebGL

Week9

Instructor: Hooman Salamat

Color models: RGB and HSB

- HSB/HLS are two variations of a very basic color model for defining colors in desktop graphics programs that closely matches the way we perceive color.
- **Hue** defines the color itself, for example, red in distinction to blue or yellow. The values for the hue axis run from 0–360° beginning and ending with red and running through green, blue and all intermediary colors like greenish-blue, orange, purple, etc.
- **Saturation** indicates the degree to which the hue differs from a neutral gray. The values run from 0%, which is no color saturation, to 100%, which is the fullest saturation of a given hue at a given percentage of illumination.
- **Lightness** indicates the level of illumination. The values run as percentages; 0% appears black (no light) while 100% is full illumination, which washes out the color (it appears white)

Color Picker - jQuery plugin

- A simple component to select color in the same way you select color in Adobe Photoshop
- Features:
  - Flat mode - as element in page
  - Powerful controls for color selection
  - Easy to customize the look by changing some images
  - Fits into the viewport
- http://www.eyecon.ro/colorpicker/#download
-
- onChange: function (hsb, hex, rgb) {
- carColorHex = hex;
- $('#carColor div').css('backgroundColor', '#' + hex);
- },

Orthogonal Projection Matrices

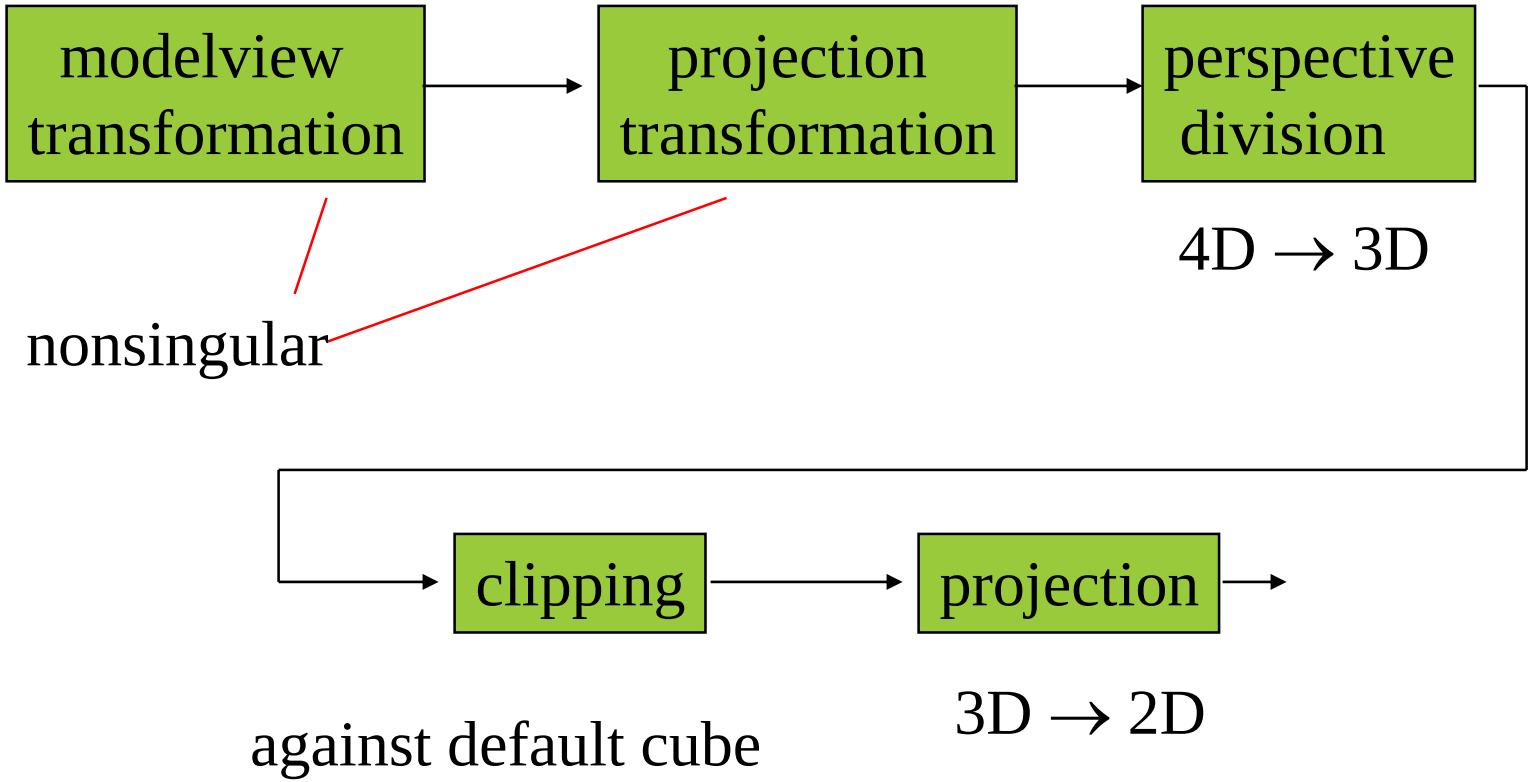
Objectives

- Derive the projection matrices used for standard orthogonal projections
- Introduce oblique projections
- Introduce projection normalization

Normalization

- Rather than derive a different projection matrix for each type of projection, we can convert all projections to orthogonal projections with the default view volume
- This strategy allows us to use standard transformations in the pipeline and makes for efficient clipping

Pipeline View



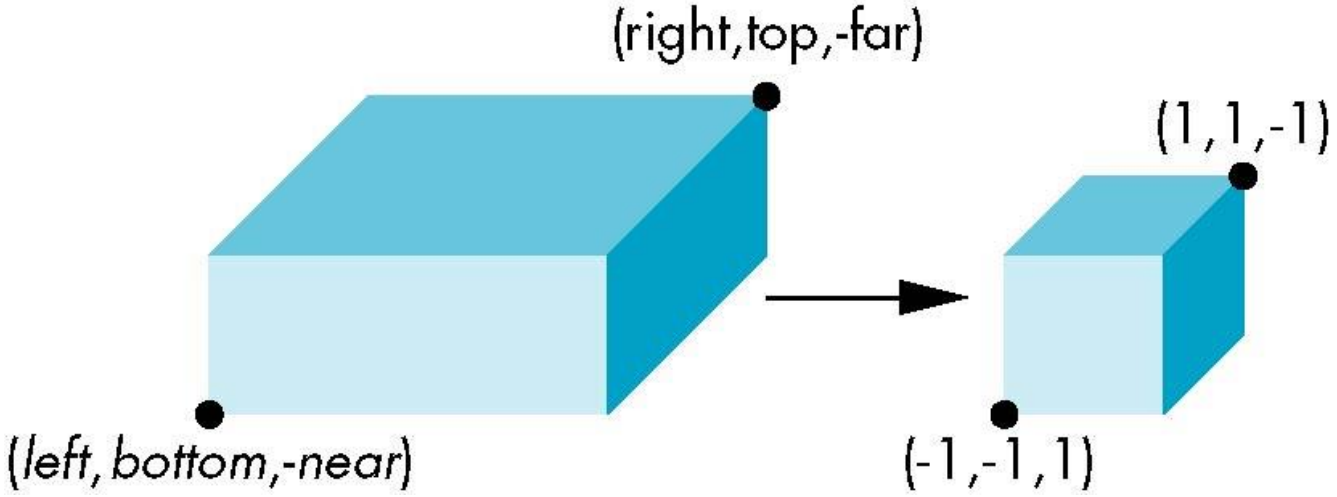
Notes

- We stay in four-dimensional homogeneous coordinates through both the modelview and projection transformations
  - Both these transformations are nonsingular
  - Default to identity matrices (orthogonal view)
- Normalization lets us clip against simple cube regardless of type of projection
- Delay final projection until end
  - Important for hidden-surface removal to retain depth information as long as possible

Orthogonal Normalization

```
ortho(left,right,bottom,top,near,far)
```

normalization ⇒ find transformation to convert specified clipping volume to default



Orthogonal Matrix

- Two steps
  - Move center to origin

    T(-(left+right)/2, -(bottom+top)/2,(near+far)/2))
  - Scale to have sides of length 2

    S(2/(left-right),2/(top-bottom),2/(near-far))

$$\mathbf{P} = \mathbf{ST} = \begin{bmatrix} \frac{2}{right-left} & 0 & 0 & -\frac{right-left}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\ 0 & 0 & \frac{2}{near-far} & \frac{far+near}{far-near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Final Projection

- Set $z = 0$
- Equivalent to the homogeneous coordinate transformation

$$\mathbf{M}_{\text{orth}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Hence, general orthogonal projection in 4D is

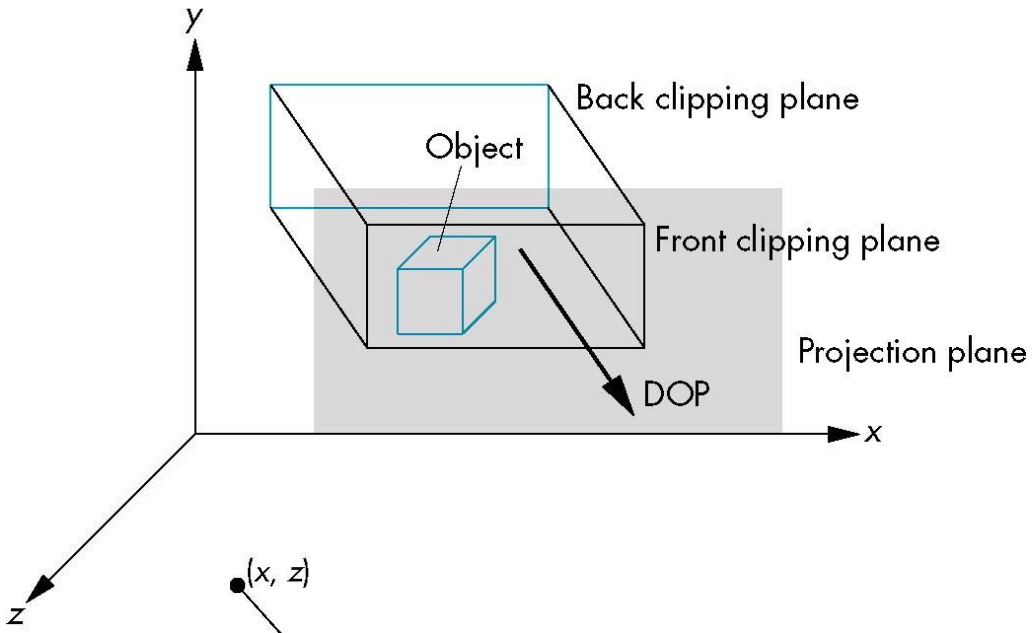
$$\mathbf{P} = \mathbf{M}_{\text{orth}}\mathbf{ST}$$

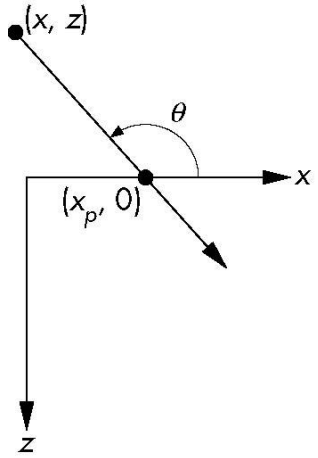
Oblique Projections

- The OpenGL projection functions cannot produce general parallel projections such as

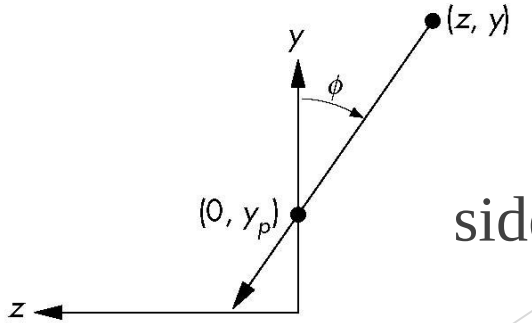
- However if we look at the example of the cube it appears that the cube has been sheared
- Oblique Projection = Shear + Orthogonal Projection

General Shear





top view



side view

Shear Matrix

xy shear (*z* values unchanged)

$$\mathbf{H}(\theta,\phi) = \begin{bmatrix} 1 & 0 & -\cot\theta & 0 \\ 0 & 1 & -\cot\varphi & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

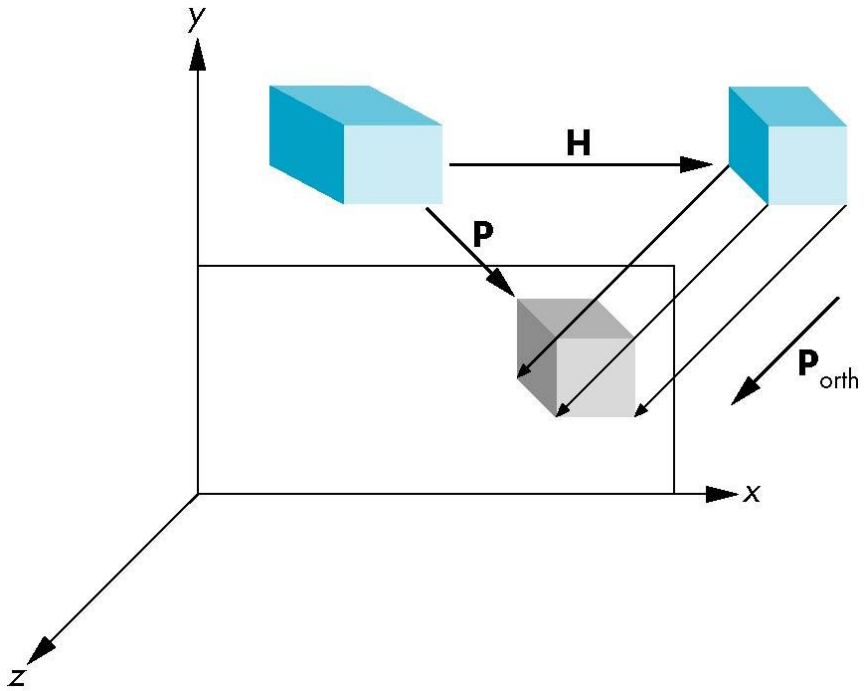
Projection matrix

General case:

$$\mathbf{P} = \mathbf{M}_{\text{orth}}\,\mathbf{H}(\theta,\phi)$$

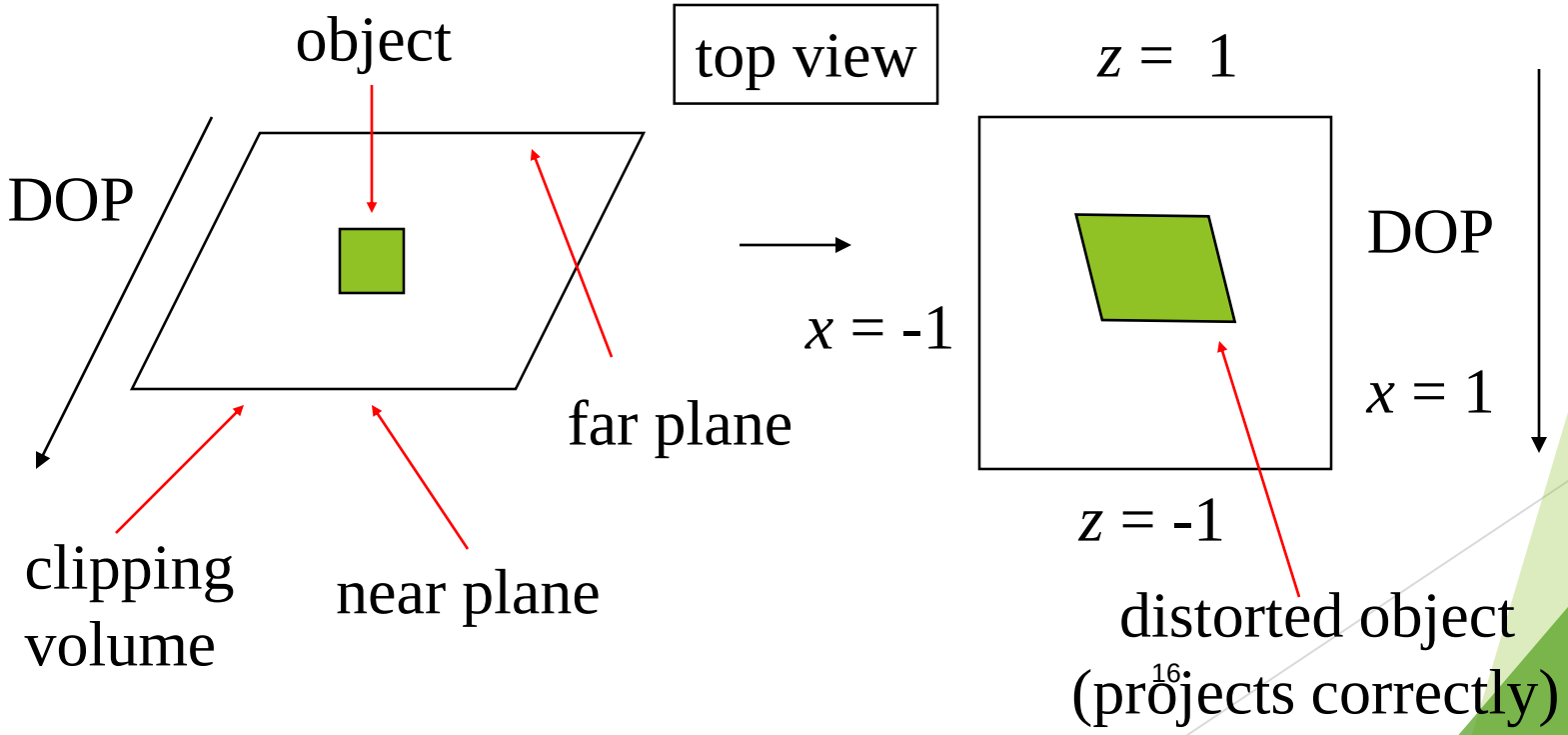
$$\mathbf{P} = \mathbf{M}_{\text{orth}}\,\mathbf{STH}(\theta,\phi)$$

Equivalency



Effect on Clipping

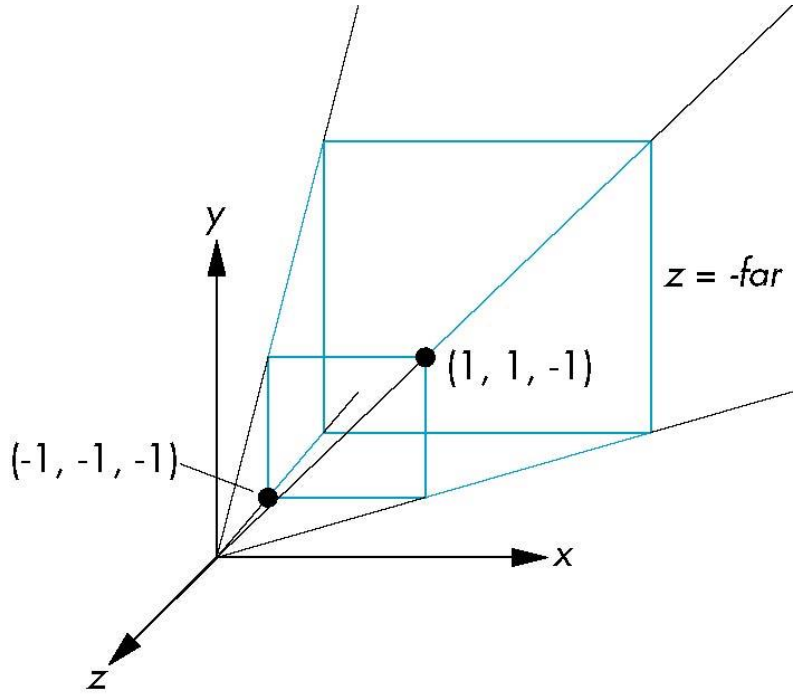
- The projection matrix **P** = **STH** transforms the original clipping volume to the default clipping volume



Simple Perspective

Consider a simple perspective with the COP at the origin, the near clipping plane at z = -1, and a 90 degree field of view determined by the planes

$x = \pm z, y = \pm z$



Perspective Matrices

Simple projection matrix in homogeneous coordinates

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Note that this matrix is independent of the far clipping plane

Generalization

$$\mathbf{N} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \alpha & \beta \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

after perspective division, the point $(x, y, z, 1)$ goes to

$$x'' = x/z$$
$$y'' = y/z$$
$$Z'' = -(\alpha+\beta/z)$$

which projects orthogonally to the desired point regardless of α and β

Picking α and β

If we pick

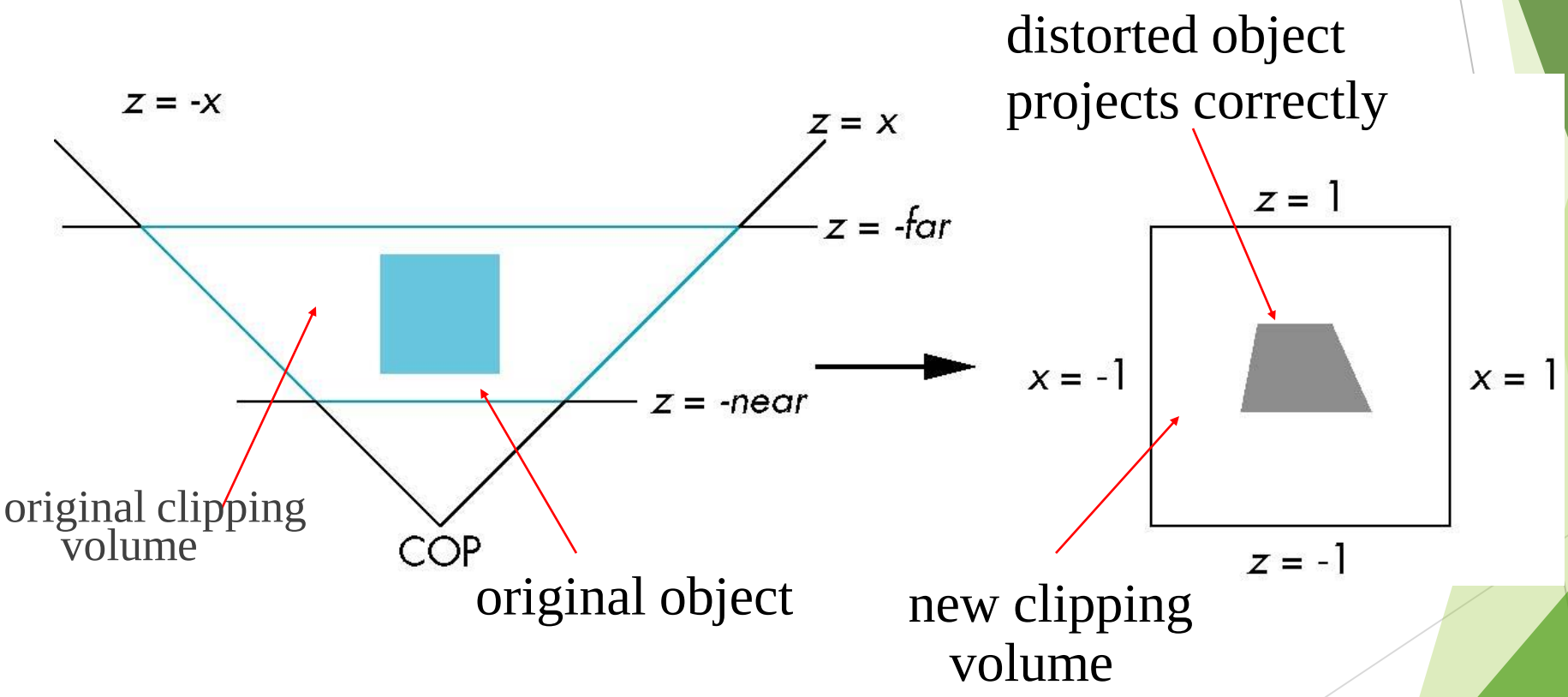
$$\alpha = \frac{\text{near} + \text{far}}{\text{far} - \text{near}}$$

$$\beta = \frac{2\text{near} * \text{far}}{\text{near} - \text{far}}$$

the near plane is mapped to $z = -1$
the far plane is mapped to $z = 1$
and the sides are mapped to $x = \pm 1, y = \pm 1$

Hence the new clipping volume is the default clipping volume

Normalization Transformation

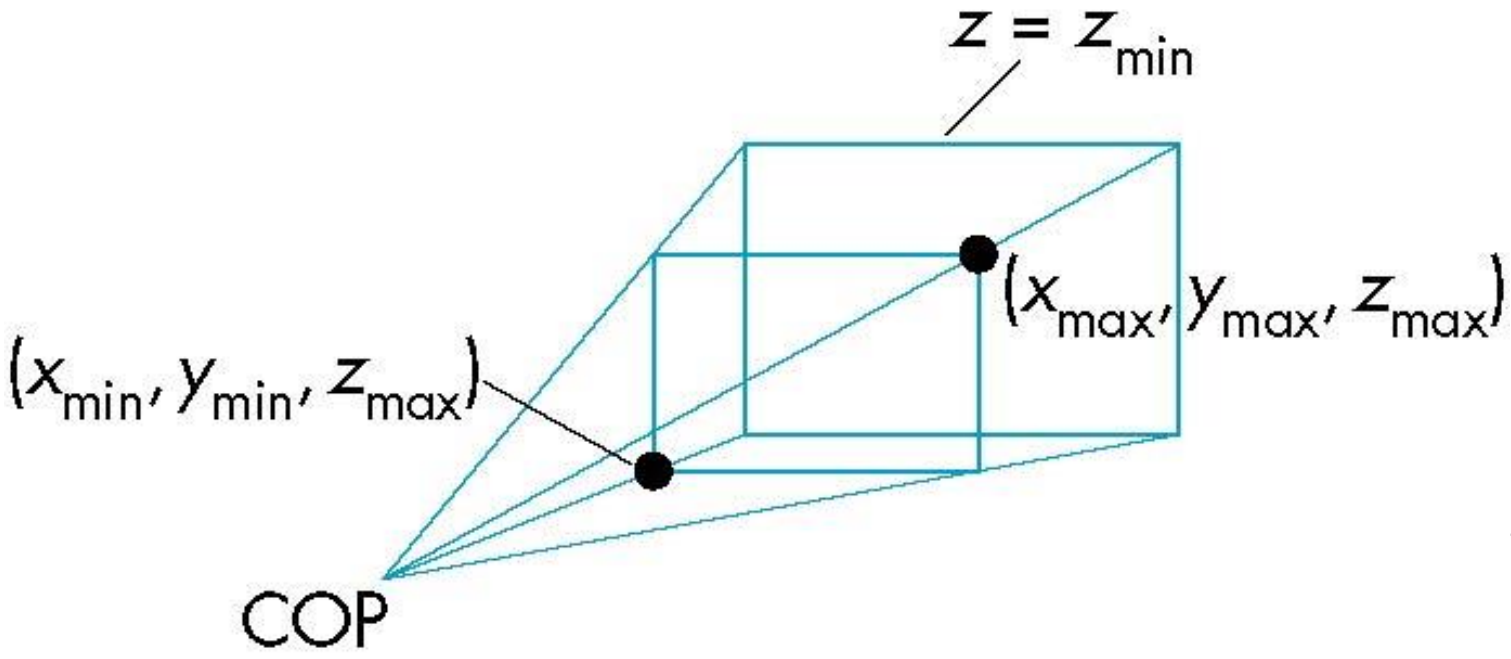


Normalization and Hidden-Surface Removal

- Although our selection of the form of the perspective matrices may appear somewhat arbitrary, it was chosen so that if $z_1 > z_2$ in the original clipping volume then the for the transformed points $z_1' > z_2'$
- Thus hidden surface removal works if we first apply the normalization transformation
- However, the formula $z'' = -(\alpha+\beta/z)$ implies that the distances are distorted by the normalization which can cause numerical problems especially if the near distance is small

WebGL Perspective

- **`gl.frustum`** allows for an unsymmetric viewing frustum (although **`gl.perspective`** does not)



OpenGL Perspective Matrix

- The normalization in **`Frustum`** requires an initial shear to form a right viewing pyramid, followed by a scaling to get the normalized perspective volume. Finally, the perspective matrix results in needing only a final orthogonal transformation

$$\mathbf{P = NSH}$$

our previously defined perspective matrix

shear and scale

Why do we do it this way?

- Normalization allows for a single pipeline for both perspective and orthogonal viewing
- We stay in four dimensional homogeneous coordinates as long as possible to retain three-dimensional information needed for hidden-surface removal and shading
- We simplify clipping

Perspective Matrices

Frustum

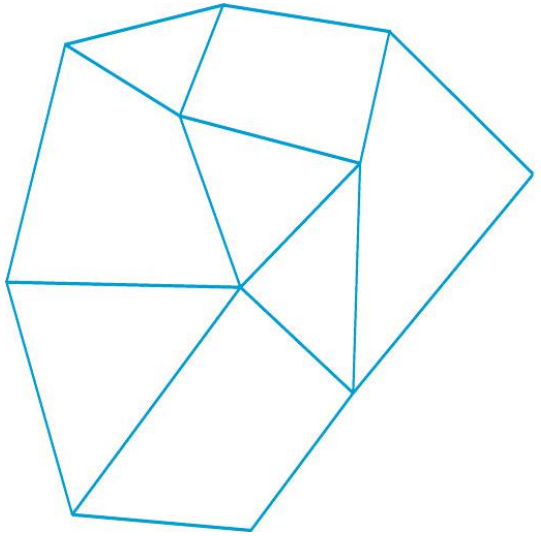
$$\mathbf{P} = \begin{bmatrix} \frac{2*near}{right-left} & 0 & \frac{right-left}{right-left} & 0 \\ 0 & \frac{2*near}{top-bottom} & \frac{top+bottom}{top-bottom} & 0 \\ 0 & 0 & -\frac{far+near}{far-near} & -\frac{2*far*near}{far-near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Perspective

$$\mathbf{P} = \begin{bmatrix} \frac{near}{right} & 0 & 0 & 0 \\ 0 & \frac{near}{top} & 0 & 0 \\ 0 & 0 & -\frac{far+near}{far-near} & -\frac{2*far*near}{far-near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

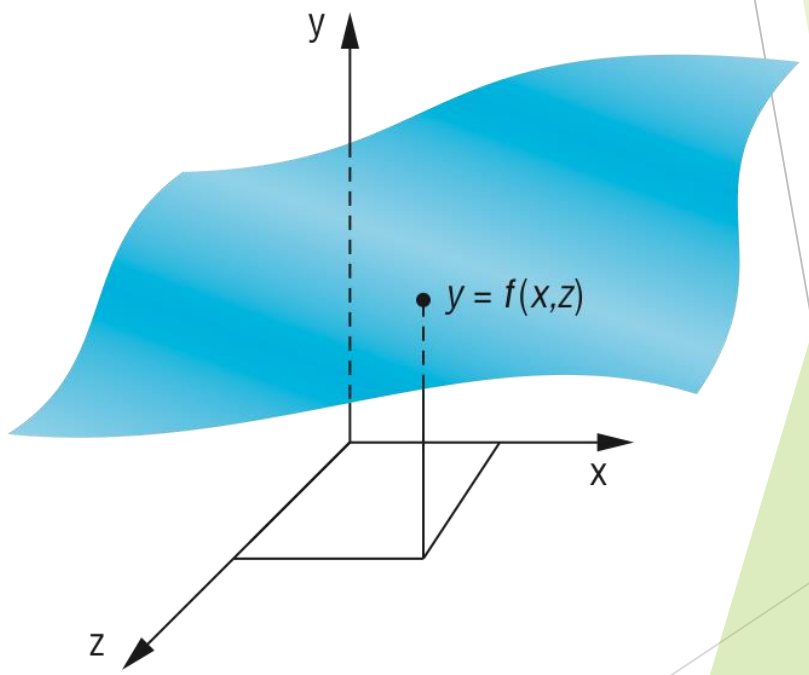
Meshes

- A mesh is a set of polygons that share vertices and edges.
- Polygonal meshes are the standard method for defining and displaying surfaces
  - Approximations to curved surfaces
  - Directly from CAD packages
  - Subdivision
- Most common are quadrilateral and triangular meshes
  - Triangle strips and fans
- We use rectangular meshes for the display of height data.
- Height data determine a surface, such as terrain, either through a function that gives the heights above a reference value



Height Fields

- For each (x, z) there is a unique y where x and z are the points on a two dimensional surface such as a rectangle. Therefore, for each (x,z), we get exactly one y.
- Such surfaces are sometimes called 2-1/2 dimensional surfaces or height fields.
- Example: use an (x,z) coordinate system to give positions on the surface of the earth, we can use function f to represent the altitude at each location.
- F is discrete based on a set of samples or measurement of experimental data
- We assume these data points are equally spaced such that
  - $X_i = X_0 + i * \Delta X$ i=0,…,nRows
  - $Z_i = Z_0 + z * \Delta Z$ z=0,…,nColumns
  - ΔX and ΔZ are the spacing between the samples in the x and z direction
  - $Y_{ij} = f(X_i, Z_j)$
- If f is known analytically, then we can sample it to obtain a set of discrete data with which to work.



Height Field

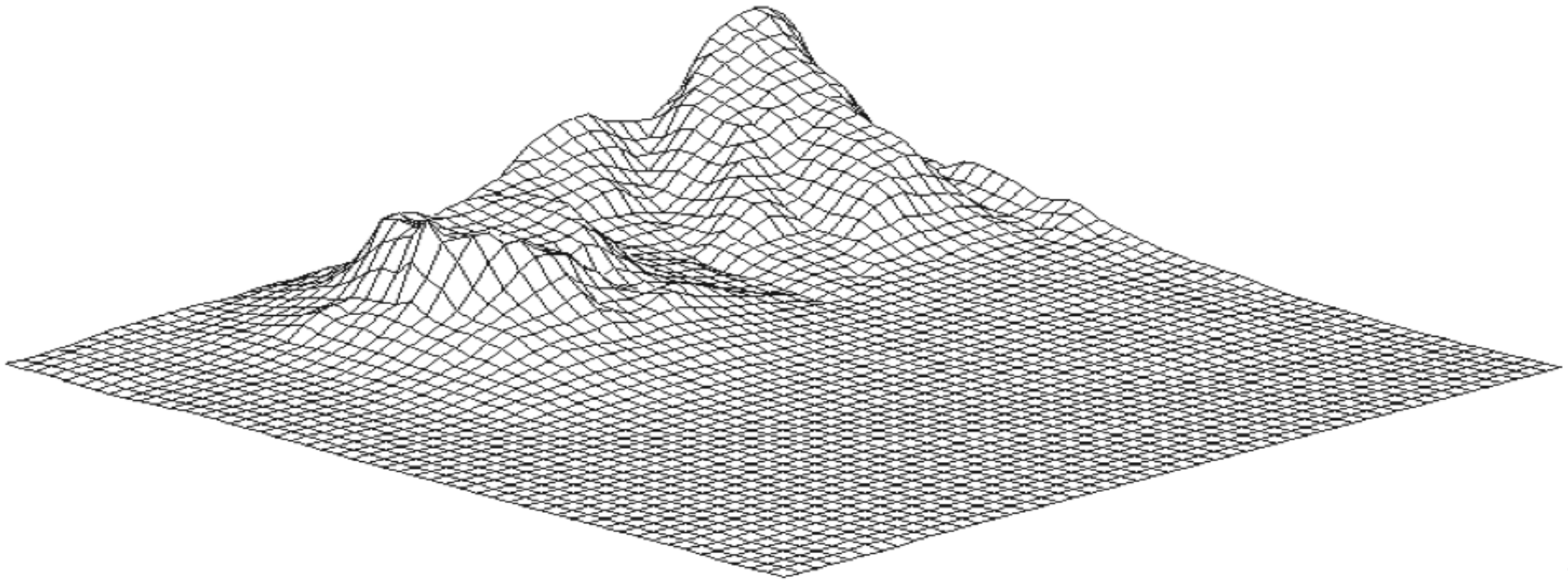
- The simplest way to display the data is to draw a line strip for each value of x and another for each value of z, thus generating nRows+nColumns line strips.
- Suppose that the height data that are in two-dimensional array data.
- We can form a single array with the data converted to vertices arranged first by rows and then by columns.

```
for(var i=0; i<nRows-1; ++i) {
        for(var j=0; j<nColumns-1;++j) {
        pointsArray.push( vec4(2*i/nRows-1, data[i][j], 2*j/nColumns-1, 1.0));
}
    }


    for(var j=0; j<nColumns-1;j++) {
      for(var i=0; i<nRows-1; i++) {
        pointsArray.push( vec4(2*i/nRows-1, data[i][j], 2*j/nColumns-1, 1.0));
}
    }
```

```
function render()
{
    gl.clear( gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
    var eye = vec3( radius*Math.sin(theta)*Math.cos(phi),
                    radius*Math.sin(theta)*Math.sin(phi),
                    radius*Math.cos(theta));
    var modelViewMatrix = lookAt( eye, at, up );
    var projectionMatrix = ortho( left, right, bottom, ytop, near, far );
    gl.uniformMatrix4fv( modelViewMatrixLoc, false, flatten(modelViewMatrix) );
    gl.uniformMatrix4fv( projectionMatrixLoc, false, flatten(projectionMatrix) );
    for(var i=0; i<nRows; i++){
      gl.uniform4fv(fColor, flatten(black));
      gl.drawArrays(gl.LINE_STRIP,i*nColumns,nColumns);
      index++;
    }
    for(var i=0; i<nColumns; i++){
      gl.uniform4fv(fColor, flatten(black));
       gl.drawArrays(gl.LINE_STRIP,i*nRows+index/2,nRows);
     }
    requestAnimFrame(render);
}
```

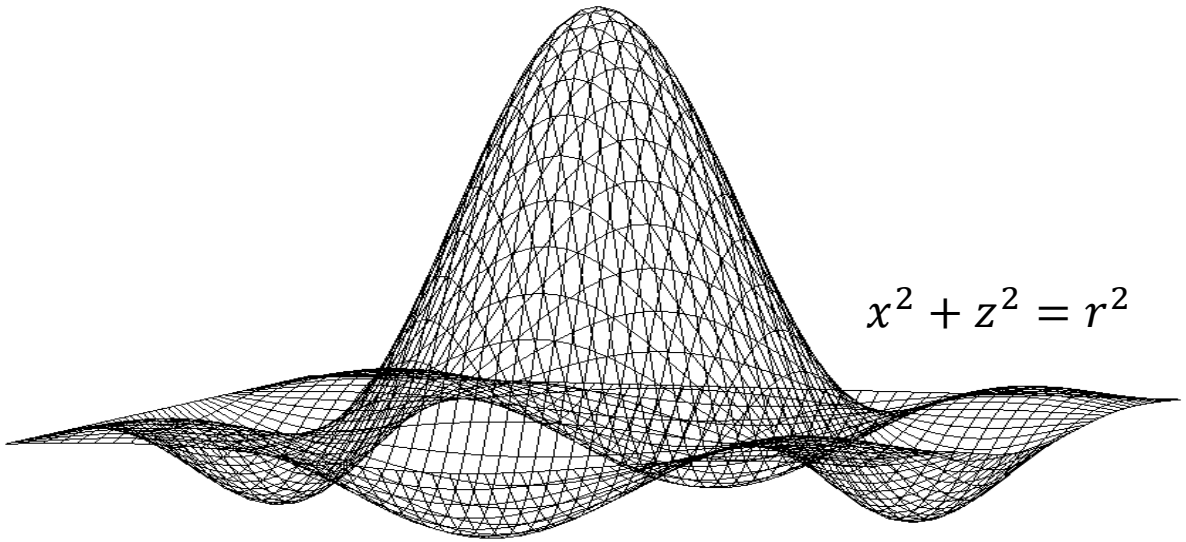
Honolulu Plot Using Line Strips



Plot 3D

- Old 2D method uses fact that data are ordered and we can render front to back
- Regard each plane of constant z as a flat surface that can block (parts of) planes behind it
- Can proceed iteratively maintaining a visible top and visible bottom
  - Lots of little line intersections
- Lots of code but avoids all 3D

Lines on Back and Hidden Faces



sombrero or Mexican hat function $(\sin \pi r)/(\pi r)$

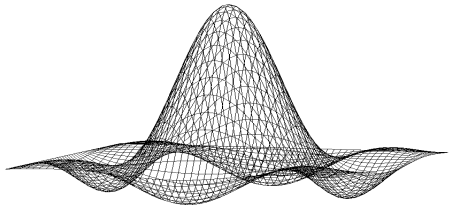
Sombrero hat

- Form a two dimensional array (as an array of one dimensional array) using this function to determine values where r is the distance to a point in the x,y plane:

```
var data = new Array(nRows);
for(var i =0; i < nRows; i++) {
  data[i]=new Array(nColumns);
}
for(var i=0; i < nRows; i++) {
    var x = Math.PI*(4*i/nRows-2.0);
    for(var j=0; j < nColumns; j++) {
        var y = Math.PI*(4*j/nRows-2.0);
        var r = Math.sqrt(x*x+y*y)
        // take care of 0/0 for r = 0
        if(r) data[i][j] = Math.sin(r)/r;
        else data[i][j] = 1;
    }
}
```

Display Meshes as line strips

```
for(var i=0; i< nRows; i++)
    gl.drawArrays( gl.LINE_STRIP, i*nColumns, nColumns );
for(var i=0; i< nColumns; i++)
    gl.drawArrays( gl.LINE_STRIP, i*nRows+pointsArray.length/2, nRows );
```

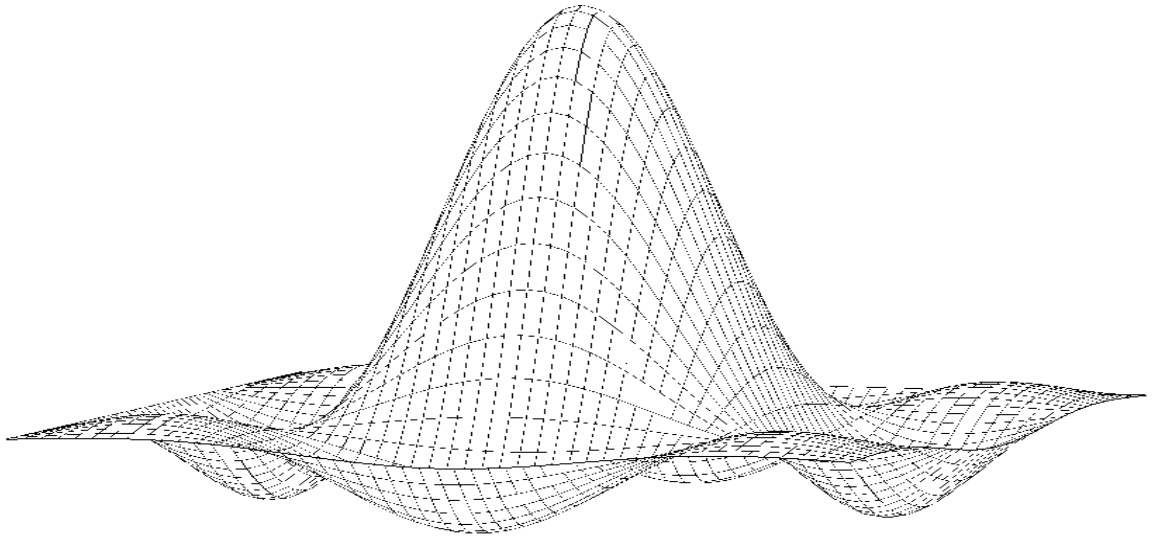


Display Meshes Using Polygons

- We can use four adjacent data points to form a quadrilateral and thus two triangles which can be shaded
- But what if we want to see the grid?
- We can display each quadrilateral twice
  - First as two filled white triangles
  - Second as a black line loop

```
for(var i=0; i<pointsArray.length; i+=4) {
        gl.uniform4fv(fColor, flatten(red));
        gl.drawArrays( gl.TRIANGLE_FAN, i, 4 );
        gl.uniform4fv(fColor, flatten(black));
        gl.drawArrays( gl.LINE_LOOP, i, 4 );}
```

Hat Using Triangles and Lines



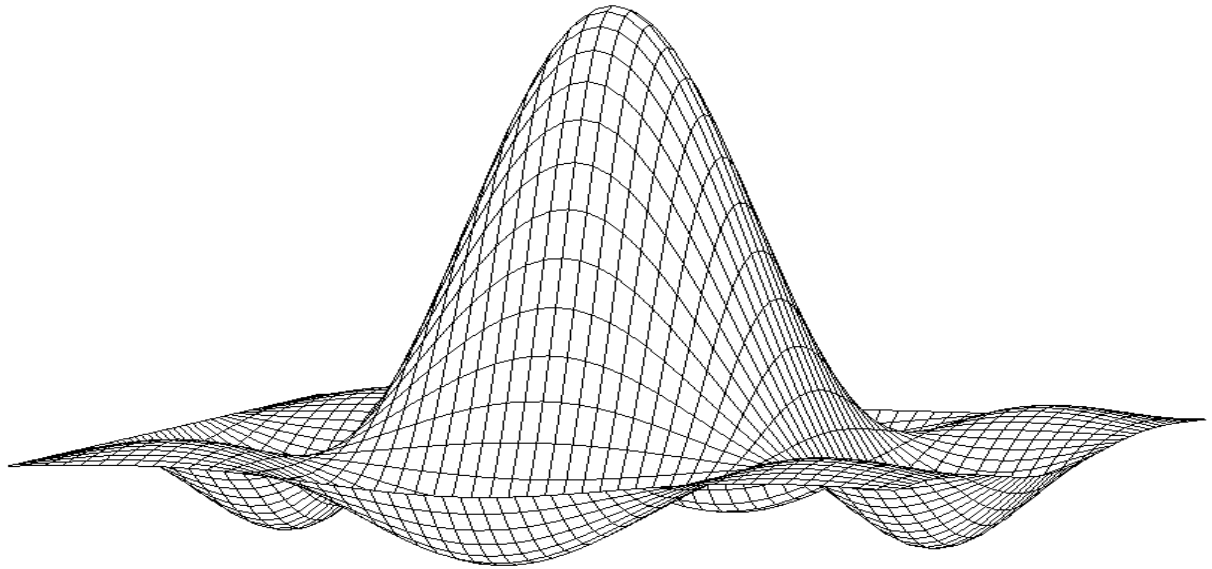
Polygon Offset

- If we draw both a polygon and a line loop, then each triangle is rendered twice in the same plane, once filled and once by its edges.
- Even though we draw the polygon first followed by the lines, small numerical errors cause some of fragments on the line to be display behind the corresponding fragment on the triangle
- We can avoid this problem by enabling the polygon offset mode
- Polygon offset (gl.polygonOffset) moves fragments slight away from camera

```
gl.enable(gl.POLYGON_OFFSET_FILL);
gl.polygonOffset(1.0, 2.0);
```

- The two parameters in gl.polygonOffset are the slope of the polygon and a difference in depth between the fragments from the polygon and the fragments from the line segment.

Hat with Polygon Offset



Other Mesh Issues

- How do we construct a mesh from disparate data (unstructured points)
- Technologies such as laser scans can produced tens of millions of such points
- Chapter 12: Delaunay triangulation
- Can we use one triangle strip for an entire 2D mesh?
- Mesh simplification

Flashlight in the Eye Graphics

- When do we not see shadows in a real scene? We must understand the interaction between light and materials.
- A point is in the shadow if it's not illuminated by any light source or if the viewer at that point cannot see any light sources.
- When the only light source is a point source at the eye or center of projection, Shadows are behind objects and not visible. This lighting strategy has been called "flashlight in the eye" model.
- Shadows are a global rendering issue
  - Is a surface visible from a source
  - May be obscured by other objects

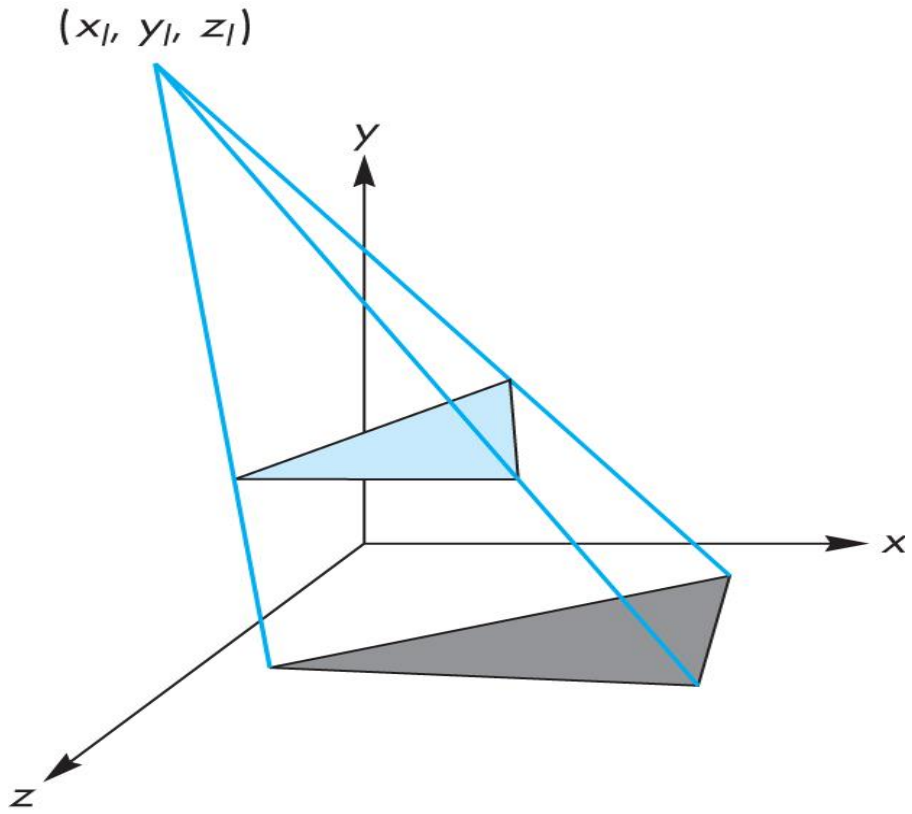
Shadows in Pipeline Renders

- Note that shadows are generated automatically by a ray tracers
  - feeler rays will detect if no light reaches a point
  - need all objects to be available
- Pipeline renderers work an object at a time so shadows are not automatic
  - can use some tricks: projective shadows
  - multi-rendering: shadow maps and shadow volumes

Projective Shadows

- Oldest methods
  - Used in flight simulators to provide visual clues
- Projection of a polygon is a polygon called a **shadow polygon**
  - **Shadow is a flat polygon**
  - **Shadow is a projection of the original polygon onto the surface.**
- Given a point light source and a polygon, the vertices of the shadow polygon are the projections of the original polygon's vertices from a point source onto a surface (y=0)

Shadow Polygon



Computing Shadow Vertex

1. Source at (x_l, y_l, z_l)
2. Vertex at (x, y, z)
3. Consider simple case of shadow projected onto ground at $(x_p, 0, z_p)$
4. Translate source to origin with $T(-x_l, -y_l, -z_l)$
5. Perspective projection

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & \frac{1}{-y_l} & 0 & 0 \end{bmatrix}$$

6. Translate back

```
 light = vec3(0.0, 2.0, 0.0);
// matrix for shadow projection
    m = mat4();
    m[3][3] = 0;
    m[3][1] = -1/light[1];
    at = vec3(0.0, 0.0, 0.0);
    up = vec3(0.0, 1.0, 0.0);
    eye = vec3(1.0, 1.0, 1.0);


        // model-view matrix for square
    modelViewMatrix = lookAt(eye, at, up);
    // send color and matrix for square then render
    gl.uniformMatrix4fv( modelViewMatrixLoc, false, flatten(modelViewMatrix) );
    gl.uniform4fv(fColor, flatten(red));
    gl.drawArrays(gl.TRIANGLE_FAN, 0, 4);
    // rotate light source
    light[0] = Math.sin(theta);
    light[2] = Math.cos(theta);
    // model-view matrix for shadow then render
    modelViewMatrix = mult(modelViewMatrix, translate(light[0], light[1], light[2]));
    modelViewMatrix = mult(modelViewMatrix, m);
    modelViewMatrix = mult(modelViewMatrix, translate(-light[0], -light[1],-light[2]));
    // send color and matrix for shadow
    gl.uniformMatrix4fv( modelViewMatrixLoc, false, flatten(modelViewMatrix) );
    gl.uniform4fv(fColor, flatten(black));
    gl.drawArrays(gl.TRIANGLE_FAN, 0, 4);
```

Shadow Process

1. Put two identical triangles and their colors on GPU (black for shadow triangle)
2. Compute two model view matrices as uniforms
3. Send model view matrix for original triangle
4. Render original triangle
5. Send second model view matrix
6. Render shadow triangle

- Note shadow triangle undergoes two transformations
- Note hidden surface removal takes care of depth issues

Generalized Shadows

- Approach was OK for shadows on a single flat surface
- Note with geometry shader we can have the shader create the second triangle
- Cannot handle shadows on general objects
- Exist a variety of other methods based on same basic idea
- We'll pursue methods based on projective textures

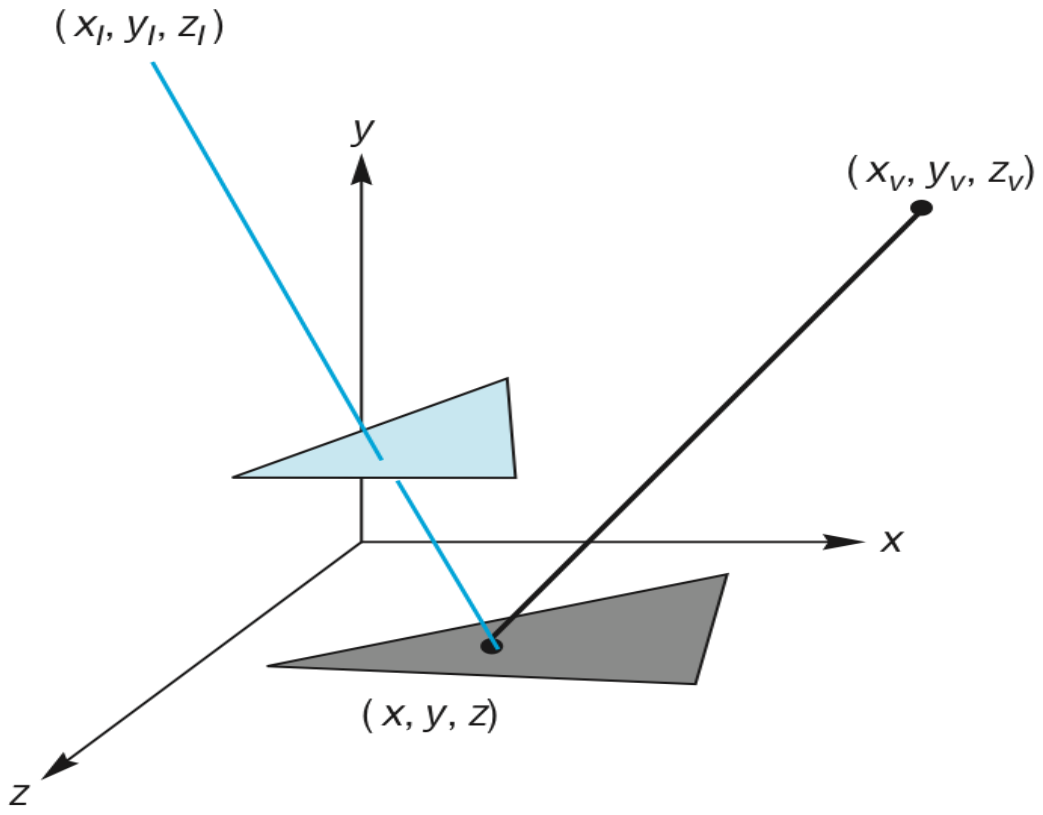
Image Based Lighting

- We can project a texture onto the surface in which case the are treating the texture as a “slide projector”
- This technique the basis of projective textures and image based lighting
- Supported in desktop OpenGL and GLSL through four dimensional texture coordinates
- Not yet in WebGL

Shadow Maps

- If we render a scene from a light source, the depth buffer will contain the distances from the source to nearest lit fragment.
- We can store these depths in a texture called a **depth map** or **shadow map**
- Note that although we don't care about the image in the shadow map, if we render with some light, anything lit is not in shadow.
- Form a shadow map for each source

Shadow Mapping



Final Rendering

- During the final rendering we compare the distance from the fragment to the light source with the distance in the shadow map
- If the depth in the shadow map is less than the distance from the fragment to the source the fragment is in shadow (from this source)
- Otherwise we use the rendered color

Implementation

- Requires multiple renderings
- We will look at render-to-texture later
  - gives us a method to save the results of a rendering as a texture
  - almost all work done in the shaders

Shadow Volumes