COSC 414/519I: Computer Graphics

2023W2

Shan Du

Hidden-Surface Removal

- Hidden-surface-removal / visible-surface algorithms
- WebGL has a particular algorithm associated with it, the *z*-buffer algorithm.
- Hidden-surface-removal algorithms can be divided into two broad classes:
  - Object-space algorithms
  - Image-space algorithms

Hidden-Surface Removal

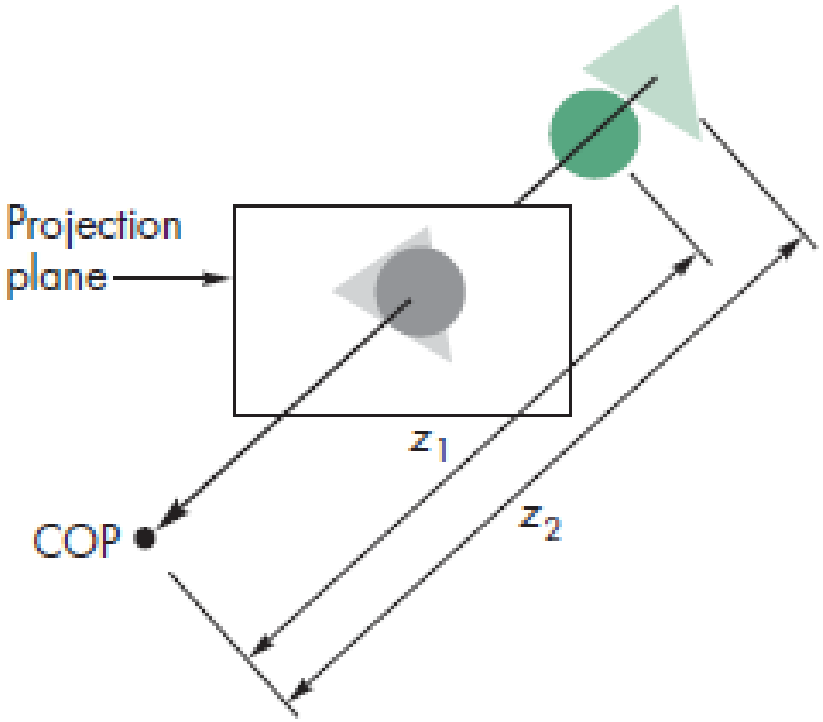
- Object-space algorithms
  - Object-space algorithms attempt to order the surfaces of the objects in the scene such that rendering surfaces in a particular order provides the correct image.
  - For example, render the back-facing surfaces first, then “paint” over them with the front surfaces and produce the correct image.
  - This class of algorithm does not work well when the objects are passed down the pipeline in an arbitrary order.

Hidden-Surface Removal

- Image-space algorithms
  - Image-space algorithms work at part of the projection process and seek to determine the relationship among object points on each projector.
  - The z-buffer algorithm is one of this type.
  - We can save partial information as each object is rendered.

Hidden-Surface Removal

- Assume objects are polygons.
- We need a depth buffer, or *z*-buffer, to store the necessary depth information as polygons are rasterized.



The *z*-buffer algorithm.

Hidden-Surface Removal

- Because we must keep depth information for each pixel in the color buffer, the z-buffer has the same spatial resolution as the color buffer.
- Its depth resolution is usually 32 bits with recent graphics cards that store this information as floating-point numbers.
- The *z*-buffer is one of the buffers that constitute the framebuffer and is usually part of the memory on the graphics card.

Hidden-Surface Removal

- The depth buffer is initialized to a value that corresponds to the farthest distance from the viewer.
- When each polygon inside the clipping volume is rasterized, the depth of each fragment - how far the corresponding point on the polygon is from the viewer – is calculated.

Hidden-Surface Removal

- If this depth is greater than the value at that fragment's location in the depth buffer, then a polygon that has already been rasterized is closer to the viewer along the projector corresponding to the fragment. Then we ignore it.
- If the depth is less than what is already is the z-buffer, then along this projector the polygon being rendered is closer than any we have seen so far. Then, we used the color of the polygon to replace the color of the pixel in the color buffer and update the depth in the *z*-buffer.

Hidden-Surface Removal

- A depth buffer is part of the framebuffer. Consequently, the application programmer need to only enable hidden-surface removal by using

```
gl.enable(gl.DEPTH_TEST);
```

and clear it, usually at the same time as the color buffer

```
g.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
```

Hidden-Surface Removal

- Culling
  - For a convex object, such as cube, faces whose normals point away from the viewer are never visible and can be eliminated or culled before the rasterizer.
  - We can turn on culling in WebGL:

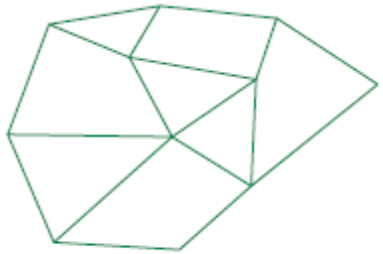
```
gl.enable(gl.CULL_FACE);
```

  - We can select which face to cull:

```
gl.cullFace(face);
```

Displaying Meshes

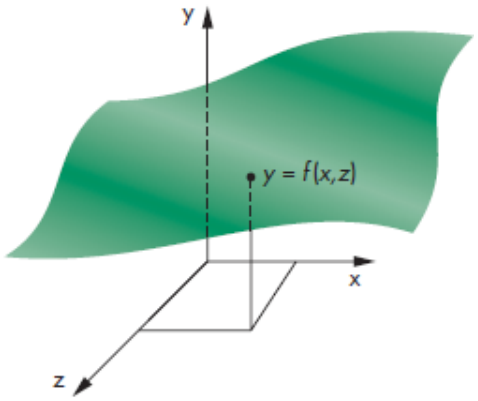
- A mesh is a set of polygons that share vertices and edges.



Mesh.

- Rectangular and triangular meshes are much simpler to work with and are useful for a wide variety of applications.
- Example:

  Use mesh to display height data.



Height field.

Displaying Meshes

- Height data determine a surface, such as terrain, either through a function that gives the heights above a reference value, such as elevations above sea level, or through samples taken at various points on the surface.
- Suppose the heights are given by *y* through a function $y = f(x, z)$, where x and z are the points on a 2D surface such as a rectangle. Thus, for each x, z, we get exactly *y*. Such surfaces are sometimes called $2\frac{1}{2}$ - dimensional surfaces or height fields.

Displaying Meshes

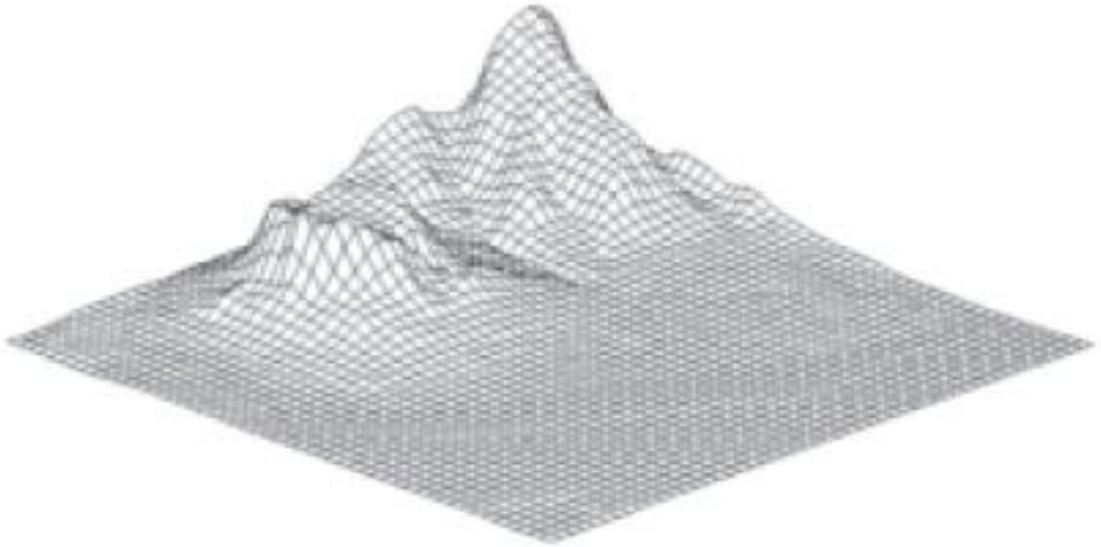
- In many situations, the function *f* is known only discretely, and we have a set of samples or measurements of the experimental data of the form $y_{ij} = f(x_i, z_j)$.
- We assume that these data points are equally spaced such that

$$x_i = x_0 + i\Delta x, i = 0, \dots, nRows - 1$$
$$z_j = z_0 + j\Delta z, j = 0, \dots, nColumns - 1$$

where Δx and Δz are the spacing between the samples.

Displaying Meshes

- 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.



Mesh plot of Honolulu data using line strips.

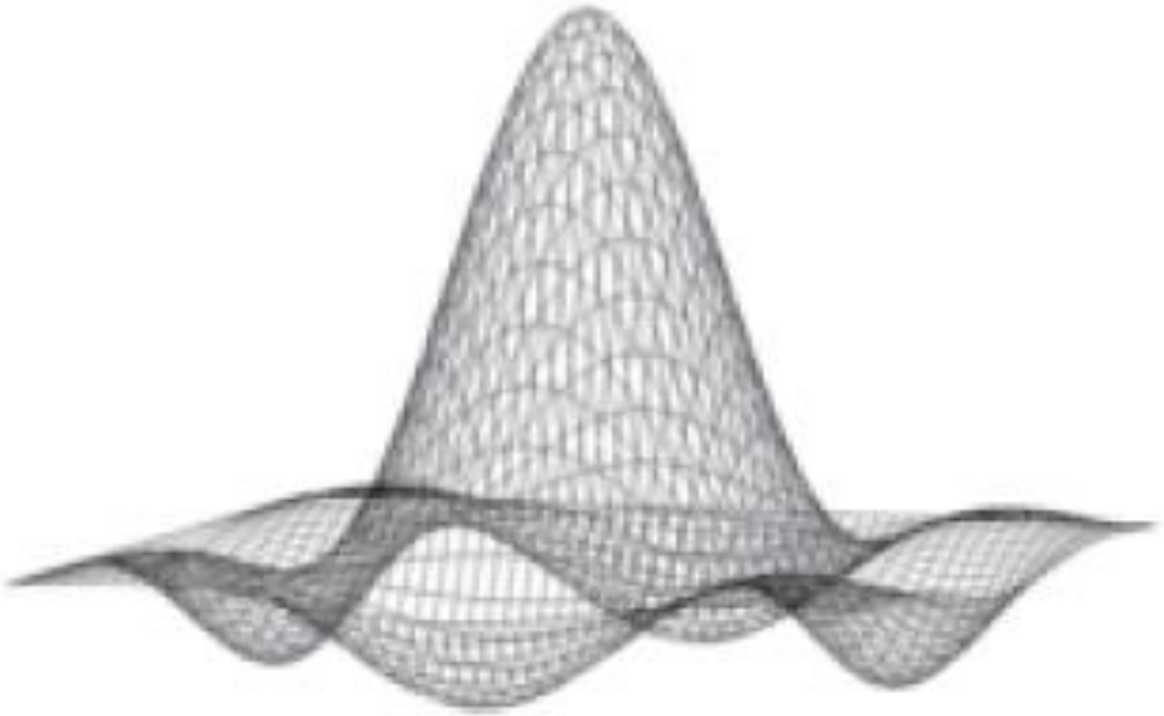
Displaying Meshes

- Suppose the height data are in a 2D array *data*, we can form a single array with the data converted to vertices:

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

- We usually scale the data over a convenient range, such as (-1,1).

Displaying Meshes



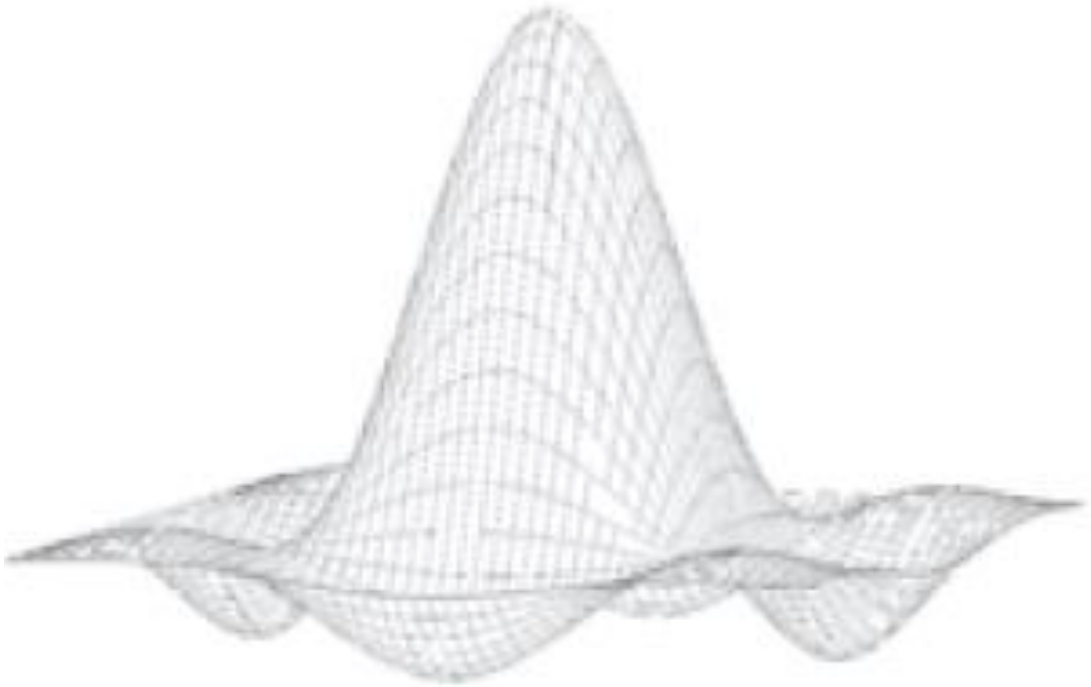
Mesh plot of sombrero function.

Displaying Meshes

- Displaying Meshes as Surfaces
  - We use filled polygons to hide the back surfaces but still display the mesh as line segments.
  - We reorganize the data so that each successive four points in the vertex array define a rectangular polygon using the values data[i][j], data[i+1][j], data[i+1][j+1], and data[i][j+1].
  - If we first render the four vertices as a triangle fan in the background color and then render the same vertices in a different colors as a line loop with hidden-surface removal enabled, the two filled triangles will hide any surfaces behind them.

Displaying Meshes

- Displaying Meshes as Surfaces



Mesh plot of sombrero using filled polygons.

Displaying Meshes

- Polygon Offset
  - Each triangle is rendered twice in the same plane, once filled and once by its edges.
  - Even though the second rendering of edges is done after the rendering of the filled triangles, numerical inaccuracies in the renderer often cause parts of second rendering to lie behind the corresponding fragments in the first rendering. This phenomenon is known as Z fighting.
  - We can avoid this problem by enabling the polygon offset mode.

Displaying Meshes

- Polygon Offset
  - This works by automatically adding an offset to the z coordinate, whose value is a function of each object's inclination with respect to the viewer's line of sight. You only need to add two lines of code to enable this function.

    1. Enabling the polygon offset function:
    gl.enable(gl.POLYGON_OFFSET_FILL);

    2. Specifying the parameter used to calculate the offset (before drawing):

    gl.polygonOffset(1.0, 1.0);

Displaying Meshes

- Polygon Offset

`gl.polygonOffset(factor, units)`

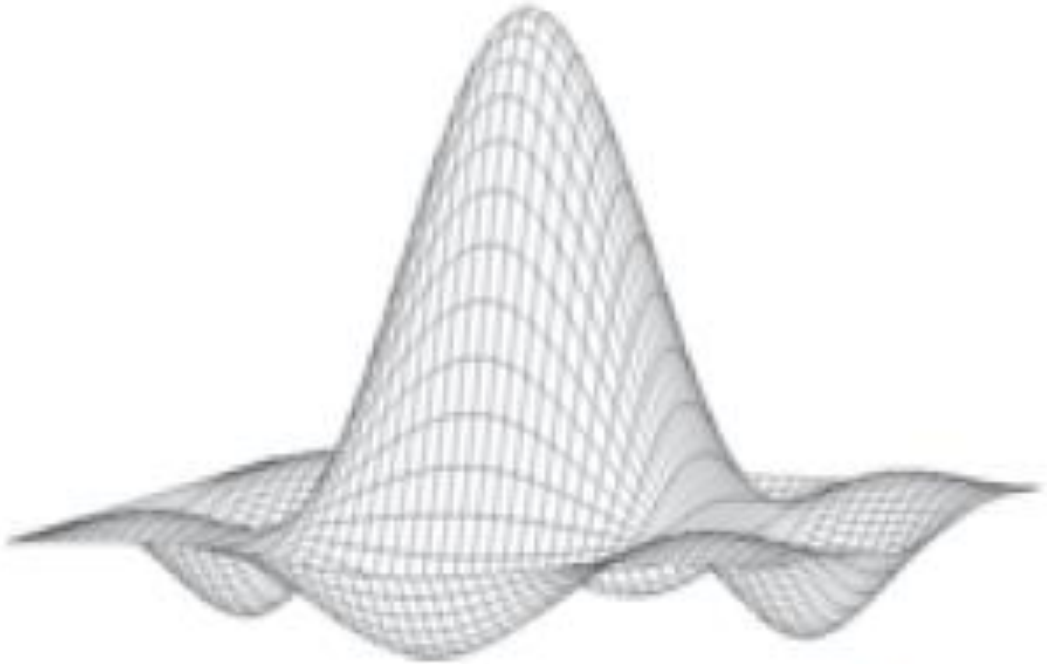
Specify the offset to be added to the z coordinate of each vertex drawn afterward. The offset is calculated with the formula m * *factor* + r * *units*, where m represents the inclination of the triangle with respect to the line of sight, and where r is the smallest difference between two z coordinates values the hardware can distinguish.

Return value None

Errors None

Displaying Meshes

- Polygon Offset



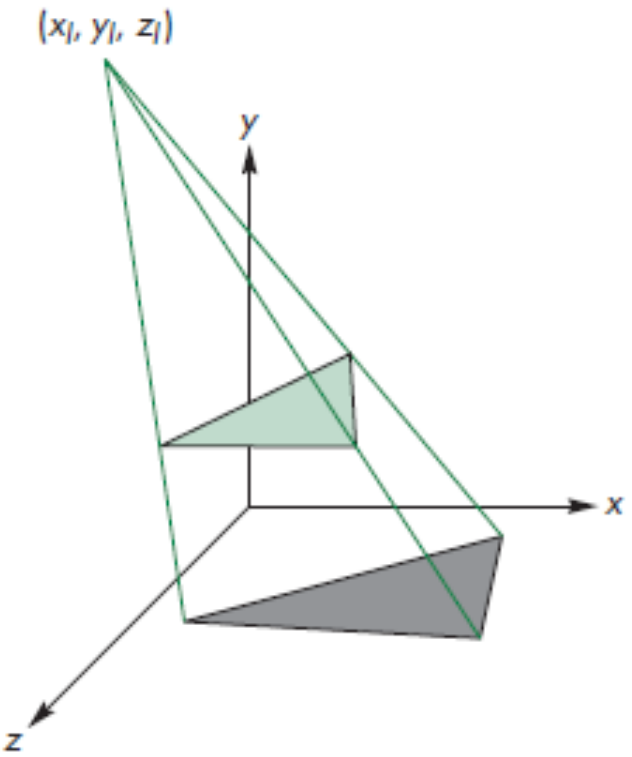
Mesh plot of sombrero using filled polygons and polygon offset.

Projections and Shadows

- Shadows are important components of realistic images and give many visual cues to the spatial relationships among the objects in a scene.
- Shadows require a light source to be present.
- A point is in shadow if it is not illuminated by any light source, or, equivalently, if a viewer at that point cannot see any light sources.

Projections and Shadows

- Projected Shadows
  - The shadow is a flat polygon, called a shadow polygon.
  - It is a projection of the original polygon onto the surface.
  - The center of projection is at the light source.

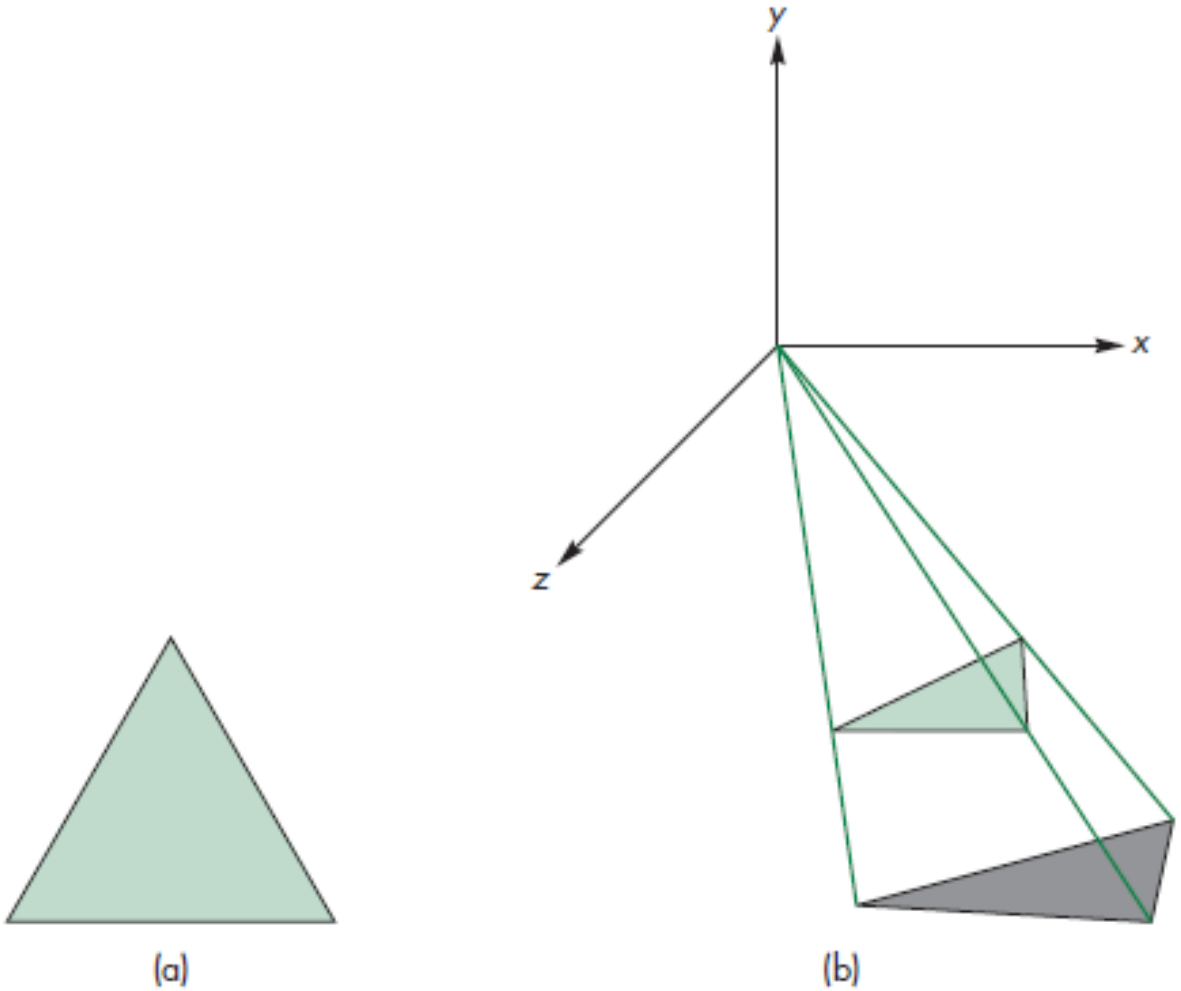


Shadow from a single polygon.

Projections and Shadows

- Projected Shadows
  - If we do a projection onto the plane of a surface in a frame in which the light source is at the origin, we obtain the vertices of the shadow polygon.
  - These vertices must then be converted back to a representation in the object frame.

Projections and Shadows



Shadow polygon projection. (a) From a light source. (b) With source moved to the origin.

Projections and Shadows

- Projected Shadows
  - Suppose we start a light source at (x_l, y_l, z_l).
  - We put the light source at the origin by using a translation matrix $\mathrm{T}(-x_l, -y_l, -z_l)$, then we have a simple perspective projection through the origin. The projection matrix is

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

Projections and Shadows

- Projected Shadows
  - Finally, we translate everything back with $\mathrm{T}(x_l, y_l, z_l)$. The concatenation of this matrix and the two translation matrices projects the vertex (x, y, z) to

$$x_p = x_l - \frac{x - x_l}{(y - y_l)/y_l}$$

$$y_p = 0$$

$$z_p = z_l - \frac{z - z_l}{(y - y_l)/y_l}.$$

Projections and Shadows

- Although we are performing a projection with respect to the light source, the matrix that we use is the model-view matrix.