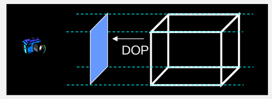
Parallel Projection

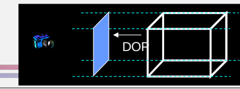
- projectors are parallel lines



Properties –

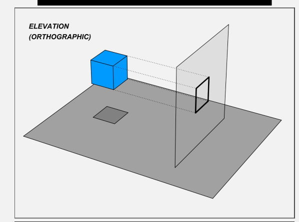
- distant objects appear the same size as near objects
- parallel lines do not converge

Parallel Projection

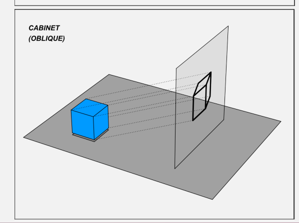


Types –

- *orthographic* – projectors are perpendicular to the projection plane



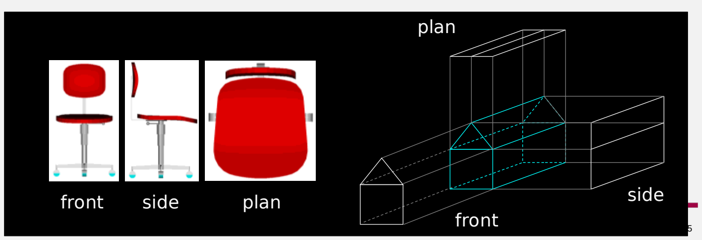
- *oblique* – projectors are not perpendicular to the projection plane



https://commons.wikimedia.org/w/index.php?curid=58480268

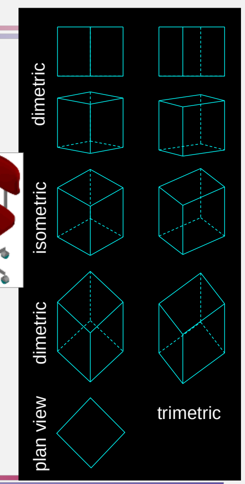
Multiview Orthographic Projection

- separate pictures from different sides
  - projection plane is parallel to one of the principal planes defined by the coordinate axes
  - all views use the same scale
- often used for engineering & architectural drawings
- accurate measurements possible
- does not provide realistic view
- need multiple drawings to get 3D feel

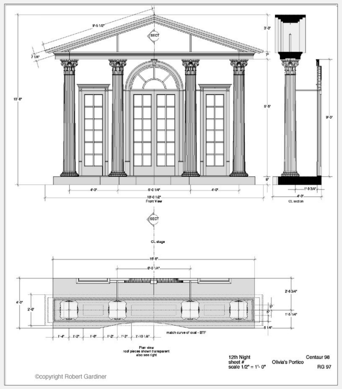


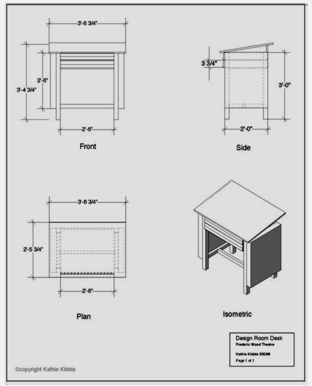
Axonometric Projections

- projection plane is not parallel to one of the principal planes defined by the coordinate axes
- *isometric* has single scale factor for all three axes
  - commonly used for catalog illustrations, patent office records, furniture design, structural design
  - illustrates 3D nature without multiple views
  - scale measurements are possible
  - lack of foreshortening creates distorted appearance
  - less useful for curved shapes
- *dimetric* has single scale factor for two axes
- *trimetric* has different scale factors for each axis

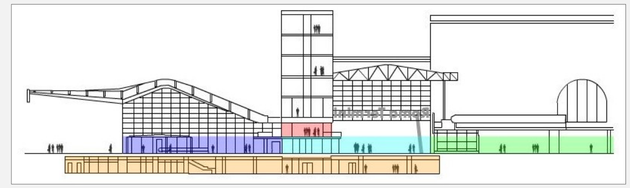


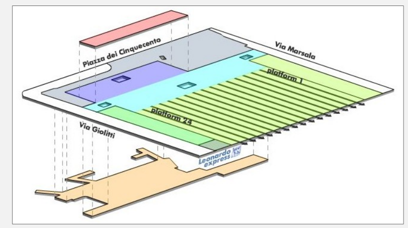
Orthographic Parallel Projections





Orthographic Parallel Projections





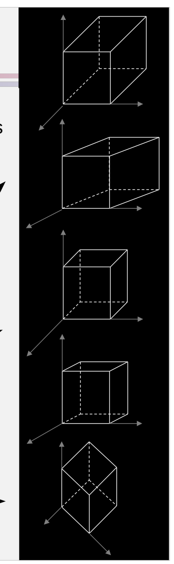
axonometric: dimetric

Isometric Parallel Projection

The Isometric Map of Midtown Manhattan, © 1989 The Manhattan Map Company
from Tufte, *Envisioning Information*, p. 37

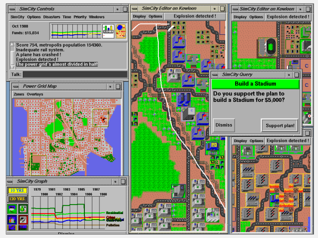
Oblique Parallel Projections

- projectors not perpendicular to projection plane
  - shows exact shape of front face for accurate measurements but still gives 3D sense
  - better than axonometric for elliptical shapes
  - choice of view may lead to distorted look
  - no foreshortening (doesn't look realistic)
- cavalier
  - perpendicular faces are projected at full scale – projected depth is same scale as width and height
  - x, y axes are drawn perpendicular to each other
  - commonly drawn with 135º or 160º angle between x and z axes
- cabinet
  - perpendicular faces are projected at 50% scale – projected depth is ½ scale of width and height
  - x, y axes are drawn perpendicular to each other
  - commonly drawn with 135º or 160º angle between x and z axes
- military
  - perpendicular faces are projected at full scale – projected depth is same scale as width and height
  - x, z axes drawn at 45º and 135º degrees
  - y axis drawn vertically

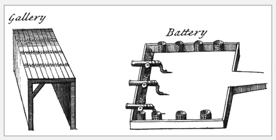


Oblique Parallel Projections

detail of a scroll by Xu Yang, 18th century



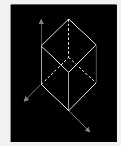
SimCity

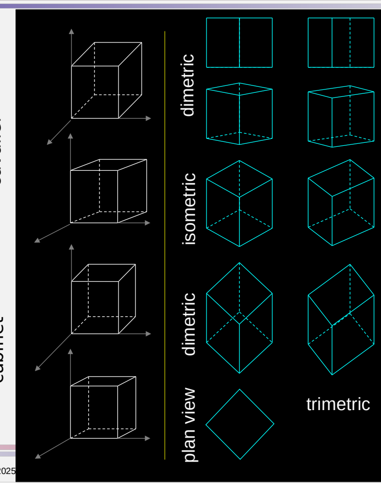


https://en.wikipedia.org/wiki/Oblique_projection#/media/File:Xu_Yang_-_Entrance_and_yard_of_a_yamen.jpg
https://en.wikipedia.org/wiki/Oblique_projection#/media/File:Perspective_cavaliere_fortification.jpg
https://en.wikipedia.org/wiki/Oblique_projection#/media/File:SimCity-Indigo.gif

Oblique vs Orthographic Projections

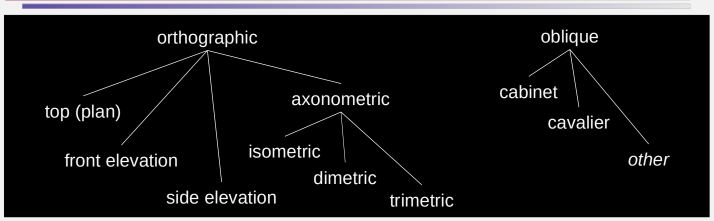
military





Parallel Projection Recap

projectors are parallel lines



projectors are perpendicular to projection plane

projectors parallel to a coordinate axis

distinguished by which axis

projectors not parallel to any coordinate axis

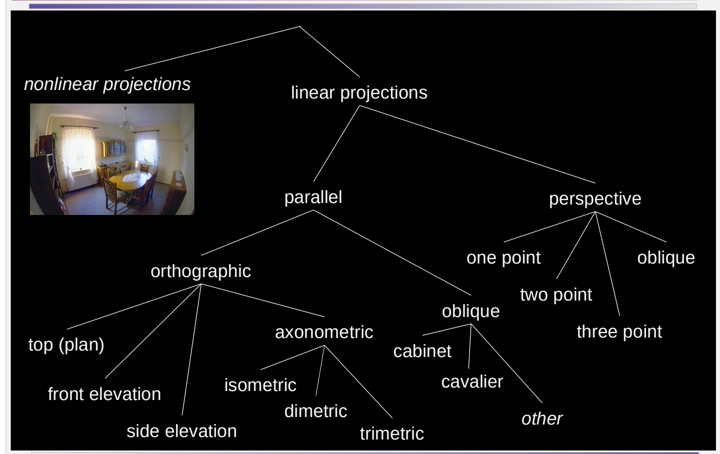
distinguished by number of different scales along axes

projectors not perpendicular to projection plane

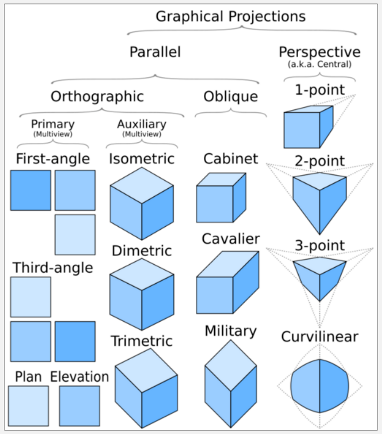
projection plane perpendicular to a coordinate axis

distinguished by scale factor of perpendicular direction

Taxonomy of Projection Types



Visual Guide to Projection Types



https://handwiki.org/wiki/File:Comparison_of_graphical_projections.svg

Projection in OpenGL

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();

…   ←   projection transform
```

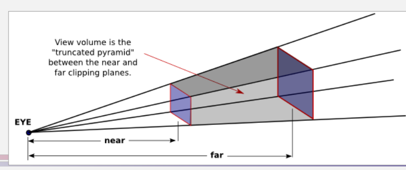
⚠ syntax is OpenGL 1.0, not WebGL

Projection Transform – Perspective

```
glFrustum(xmin,xmax,ymin,ymax,near,far)
gluPerspective(fieldOfViewAngle,aspect,near,far)
```

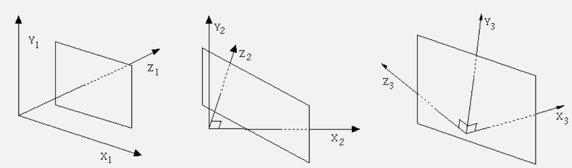
⚠ syntax is OpenGL 1.0, not WebGL

– in EC
– `xmin`,`xmax`,`ymin`,`ymax` define the view window (on the near clipping plane)
– `fieldOfViewAngle` is the angle between the bottom and top of the view volume; `aspect` is the view volume's aspect ratio (width/height)
  - aspect should generally match the aspect ratio of the viewport
– `near`, `far` specify the distance from the eye to the clipping planes
  - z coordinates are `-near` and `-far`, respectively



Projection Transform – Perspective

- one-, two-, and three-point perspective is determined by the number of zero components in the look at vector (eye → ref)
  – *n* zero components → 3-*n* point perspective



n-point perspective results from *n* principal axes intersecting the projection plane

- for oblique perspective, use `glFrustum` where the x and/or y limits are not symmetrical around 0

```
glFrustum(xmin,xmax,ymin,ymax,near,far)
gluPerspective(fieldOfViewAngle,aspect,near,far)
```

⚠ syntax is OpenGL 1.0, not WebGL

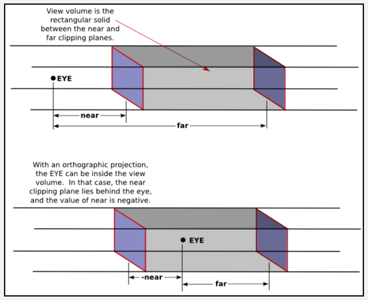
https://people.eecs.berkeley.edu/~barsky/perspective.html

Projection Transform – Orthographic

`glOrtho(xmin,xmax,ymin,ymax,near,far)`

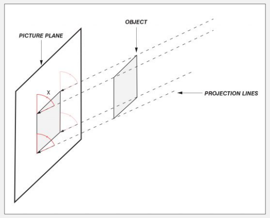
⚠ syntax is OpenGL 1.0, not WebGL

- in EC
- `xmin,xmax,ymin,ymax` define the view window (on the near clipping plane)
  - near, far specify the distance from the eye to the clipping planes
    - z coordinates are `-near` and `-far`, respectively
- type depends on the *look at* vector (eye → ref)
  - front, side, plan – along z, x, y, respectively
  - isometric – absolute value of x, y, z components are equal
  - dimetric – absolute value of two of x, y, z components are equal
  - trimetric – absolute value of x, y, z components are different



Projection Transform – Oblique Parallel

- oblique parallel projections are not directly supported by OpenGL
  - instead, shear first to create an orthogonal projection



https://blogs.ubc.ca/axonometric/visualglossary/

Recap: Viewing and Projection

⚠ syntax is OpenGL 1.0, not WebGL

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();

gluLookAt(eyeX,eyeY,eyeZ,
          refX,refY,refX,
          upX,upY,upZ);
```

↖ viewing transform

```
glPushMatrix();
…
glPopMatrix();
glPushMatrix();
…
glPopMatrix();

…
```

} define the scene (modeling transforms and primitives)

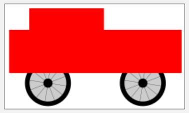
```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();

…
```

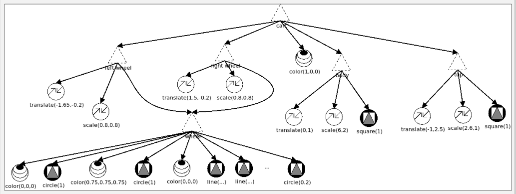
← projection transform

```
glFrustum(xmin,xmax,ymin,ymax,
          near,far)
gluPerspective(fieldOfViewAngle,
               aspect,near,far)

glOrtho(xmin,xmax,ymin,ymax,
        near,far)
```

follow `glOrtho` with shear (shear first) for oblique parallel

Hierarchical Modeling

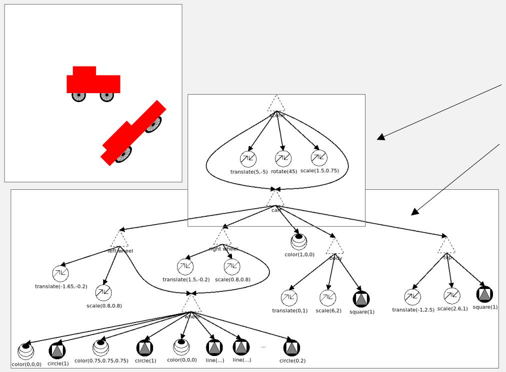




```
private void drawCart ( GL2 gl2 ) {
  // left wheel
  gl2.glPushMatrix();
  gl2.glTranslatef(-1.65f,-0.2f,0f);
  gl2.glScalef(0.8f,0.8f,1f);
  drawWheel(gl2);
  gl2.glPopMatrix();

  // right wheel
  gl2.glPushMatrix();
  gl2.glTranslatef(1.5f,-0.2f,0f);
  gl2.glScalef(0.8f,0.8f,1f);
  drawWheel(gl2);
  gl2.glPopMatrix();

  gl2.glColor3f(1f,0f,0f);

  // body of cart
  gl2.glPushMatrix();
  gl2.glTranslatef(0f,1f,0f);
  gl2.glScalef(6f,2f,1f);
  Drawing2D.filledSquare(gl2,1);
  gl2.glPopMatrix();

  // top of cart
  gl2.glPushMatrix();
  gl2.glTranslatef(-1f,2.5f,0f);
  gl2.glScalef(2.6f,1f,1f);
  Drawing2D.filledSquare(gl2,1);
  gl2.glPopMatrix();
}
```

⚠ syntax is OpenGL 1.0, not WebGL

Hierarchical Modeling



```
drawCart(gl2);

gl2.glPushMatrix();
gl2.glTranslatef(5f,-5f,0f);
gl2.glRotate(45,0f,0f,5f);
gl2.glScale(1.5f,.75f,1f);
drawCart(gl2);
gl2.glPopMatrix();
```

```
private void drawCart ( GL2 gl2 ) {
  // left wheel
  gl2.glPushMatrix();
  gl2.glTranslatef(-1.65f,-0.2f,0f);
  gl2.glScalef(0.8f,0.8f,1f);
  drawWheel(gl2);
  gl2.glPopMatrix();

  // right wheel
  gl2.glPushMatrix();
  gl2.glTranslatef(1.5f,-0.2f,0f);
  gl2.glScalef(0.8f,0.8f,1f);
  drawWheel(gl2);
  gl2.glPopMatrix();

  gl2.glColor3f(1f,0f,0f);

  // body of cart
  gl2.glPushMatrix();
  gl2.glTranslatef(0f,1f,0f);
  gl2.glScalef(6f,2f,1f);
  Drawing2D.filledSquare(gl2,1);
  gl2.glPopMatrix();

  // top of cart
  gl2.glPushMatrix();
  gl2.glTranslatef(-1f,2.5f,0f);
  gl2.glScalef(2.6f,1f,1f);
  Drawing2D.filledSquare(gl2,1);
  gl2.glPopMatrix();
}
```


Recap: Viewing and Projection

⚠ syntax is OpenGL 1.0, not WebGL

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();

gluLookAt(eyeX,eyeY,eyeZ,
          refX,refY,refX,
          upX,upY,upZ);
```

← viewing transform

```
glPushMatrix();
…
glPopMatrix();
glPushMatrix();
…
glPopMatrix();

…
```

} define the scene (modeling transforms and primitives)

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();

…
```

← projection transform

```
glFrustum(xmin,xmax,ymin,ymax,
          near,far)
gluPerspective(fieldOfViewAngle,
               aspect,near,far)

glOrtho(xmin,xmax,ymin,ymax,
        near,far)
```

follow `glOrtho` with shear (shear first) for oblique parallel

Transformations in WebGL

- the programmable pipeline does not maintain modelview or projection matrices
  - now up to the programmer!
- three tasks
  - managing viewing pipeline transforms (modeling, viewing, projection) in JavaScript
  - supplying transforms as parameters to the vertex shader
  - applying transforms in the vertex shader

Managing Viewing Pipeline Transforms

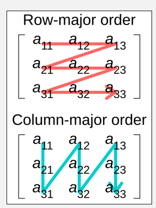
- similar to OpenGL 1.0, we'll keep track of projection and modelview matrices
  - maintained in JavaScript, then passed to the vertex shader

Using `glMatrix`

- `glMatrix` is a free JavaScript library implementing vector and matrix math
  - include in program with
    `<script src="gl-matrix-min.js"></script>` adjust path as appropriate
    - goes in <head> section

Using `glMatrix`

- defines types `vec2`, `vec3`, `vec4`, `mat3`, `mat4` for vectors and matrices
  - these are JavaScript types rather than GLSL types even though they have the same names
  - really just 1D arrays (regular JavaScript arrays or typed arrays of type `Float32Array`) with right number of elements
    - can pass an array with right number of elements whenever parameter is one of the vector or matrix types
    - matrix types use column-major order (compatible with WebGL)



Using `glMatrix`

- functions
  - *`type`*`.create()` creates a typed array of the appropriate length
    - default is 0s for `vec`*`i`*, identity for `mat`*`i`*
  - *`type`*`.clone(`*`param`*`)` creates a copy of *param*

create projection and modelview matrices to replicate what OpenGL 1.0 maintains –

```
let modelview = mat4.create();     // identity
let projection = mat4.create();    // identity
```

Using `glMatrix`

- functions – transforms
  - transform functions set the value of the first parameter (which must have been allocated previously) instead of returning result
    - `mat4.multiply(A,B,C)`
    - `mat4.translate(A,B,[tx,ty,tz])`
    - `mat4.scale(A,B,[sx,sy,sz])`
    - `mat4.rotateX(A,B,radians)`
    - `mat4.rotateY(A,B,radians)`
    - `mat4.rotateZ(A,B,radians)`
    - `mat4.rotate(A,B,radians,[px,py,pz])`
      - axis of rotation is vector (0,0,0) → (px,py,pz)
    - `mat4.identity(A)`
      - sets A to the identity matrix

usage example – to achieve the effect of `glTranslatef(dx,dy,dz)` as a modeling transform, use

```
mat4.translate(modelview,
               modelview,[dx,dy,dz])
```


Using `glMatrix`

- functions – viewing and projection
  - all set A (which must have been allocated previously) to the matrix defined
  - `mat4.lookAt(A,[eyex,eyey,eyez],[refx,refy,refz],[upx,upy,upz])`

with the modelview matrix as A, this is equivalent to

```
glLoadIdentity();
gluLookAt(eyex,eyey,eyez,refx,refy,refz,upx,upy,upz);
```

  - `mat4.ortho(A,left,right,bottom,top,near,far)`
  - `mat4.frustum(A,left,right,bottom,top,near,far)`
  - `mat4.perspective(A,fieldOfView,aspect,near,far)`
    - `fieldOfView` in radians

with the projection matrix as A, these are equivalent to

```
glLoadIdentity();
glOrtho(left,right,bottom,top,near,far);
etc
```