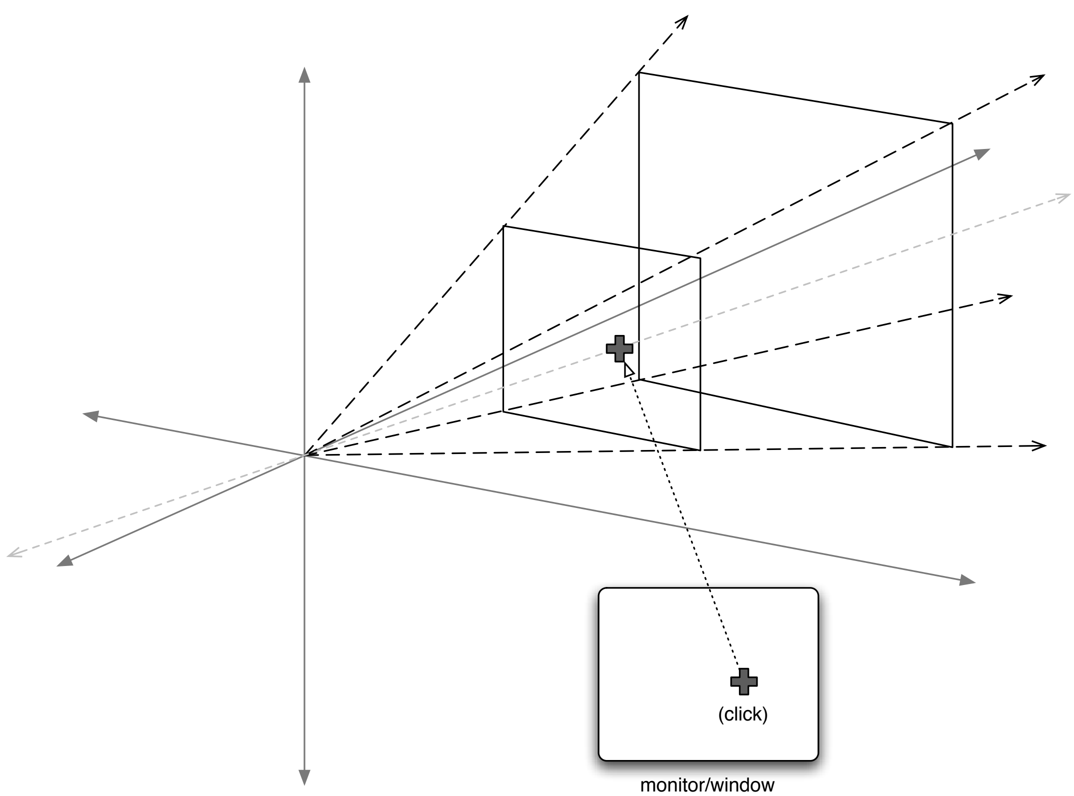
“Unproject” Explained

- Now that we’ve broken down the math for projections, we can explain the *unproject* program that was distributed earlier
- The core issue is: given a mouse event (click, move, drag), how do we translate that event’s mouse coordinates into our 3D world?
- With our projection analysis, we can phrase this question more specifically now: given a set of coordinates on the 2D viewport, what are the corresponding coordinates in 3D space?



Observations

- First off, note from the diagram that a 2D mouse click does not translate into a point, but into a ray — after all, we are adding an entire dimension
- So, we can't really go from a mouse point to a single 3D point; the best we can do is identify the *line* along which the mouse point's 3D equivalent must lie
- OpenGL's *gluUnProject()* function will help give you that line — but what you do with it after that is up to you

What *gluUnProject()* Does

- *gluUnProject()* inverts the projection calculation; instead of going from a world point to a screen point, which is what projection does…

  point in "world" ➠ model-view ➠ projection ➠ viewport ➠ point on "screen"

- …we take the screen point and go the other way:

  point on "screen" ➠ viewport^{-1} *➠ projection*$^{-1}$ *➠ model-view*$^{-1}$ *➠ point in "world"*

- With this in mind, the signature of the *gluUnProject()* function should now be pretty self-explanatory:

  *GLint gluUnProject (GLdouble winX, GLdouble winY, GLdouble winZ, const GLdouble *model, const GLdouble *proj, const GLint *view, GLdouble* objX, GLdouble* objY, GLdouble* objZ);*

gluUnProject() Double-Take

- Given what we have said so far, some parts of *gluUnProject()*'s signature may have you wondering:
  - Why does the screen ("win") point have a z-coordinate?
  - Since the result of the function is a 3D point, the output arguments are passed as pointers; so what is that integer that the function returns directly?
- We answer the second question first: not all matrices are invertible — thus, *gluUnProject()* might not succeed, in which case it will return *GL_FALSE*, with successful inversion returning *GL_TRUE*
- Now back to that z-coordinate on the "screen…"

- Recall that, during the final drawing to the viewport, we happen to not *need* the z coordinate; however, as you have seen from the matrices, we *do* get a value for the z axis…so, even though we don't use z in the final drawing, it can (and does) get calculated
- It turns out that, the way OpenGL calculates things, *winZ* == 0.0 (the screen) corresponds to *objZ* == –*N* (the near plane), and *winZ* == 1.0 corresponds to *objZ* == –*F* (the far plane)
- Since two points determine a line, we actually need to call *gluUnProject()* <u>twice</u>: once with *winZ* == 0.0, then again with *winZ* == 1.0 — this will give us the world points that correspond to the mouse click on the near and far planes, respectively

Typical *gluUnProject()* Sequence

Now that we know what *gluUnProject()* specifically does, we can sketch out its general use, given some screen coordinate *(mx, my)*:

- Invert the *my* coordinate (since the screen *y*-axis goes in the opposite direction as the 3D *y*-axis)
- Grab the current values for the three matrices: model-view, projection, and viewport
- Call *gluUnProject()* twice, once for (*mx*, *my*, 0.0) and again for (*mx*, *my*, 1.0)

- Once you have the two points, what you do next now depends on how you're representing the objects in your model
- Generally, you would test to see which objects intersect that line, then choose one of them as the "hit" object, and act accordingly
- Bilinear interpolation is useful here: since you know two endpoints, you can represent their line in terms of a single argument *u*, where $u = 0$ corresponds to the near point, and $u = 1$ corresponds to the far point

$$L(u) = nearPoint + u(farPoint - nearPoint)$$

  - In the sample program, we're testing against a fixed plane with a known *z*, so we solve for *u* using bilinear interpolation using the *z* coordinates, then use *u* to subsequently calculate *x* and *y*; the resulting (*x*, *y*, z) is the point on the plane that was "clicked on" by the mouse