CS 432 – Interactive Computer Graphics

Lecture 5 – Part 1

Picking

Reading

- Angel
  - Chapters 3-4
  - Section 10.1
- Red Book
  - Chapter 5, Appendix E

Ray Casting/Picking

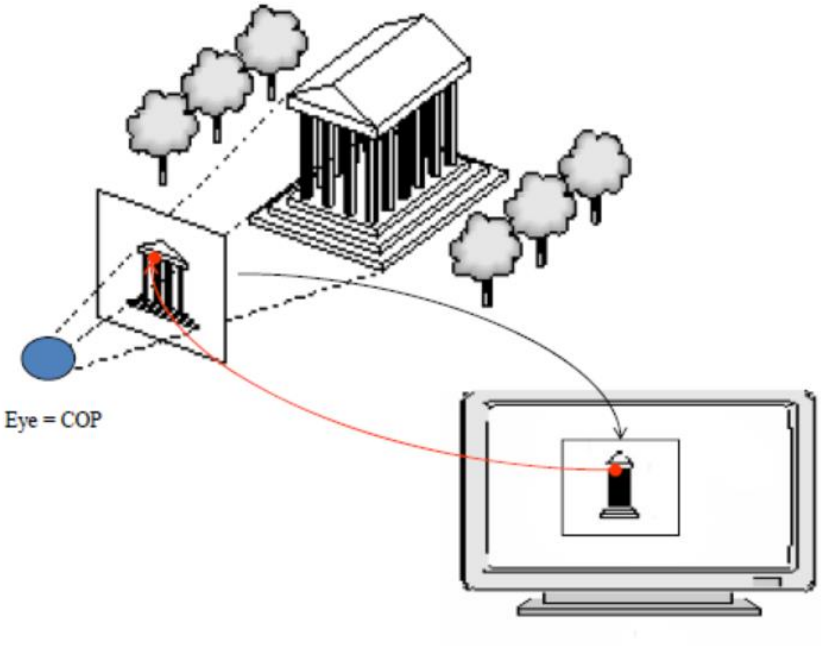
- One thing we often want to do is select objects
  - We will call this *picking*
- There are several approaches to this.
- We are going to take a *ray casting* approach since it overlaps with several other concepts:
  - Ray tracing
  - Collision detection

Ray Casting/Picking

- Our approach:
  1. Find the pixel in screen coordinates we clicked on
  2. Convert this to be the location on the front of the viewing volume
  3. Adjust for projection
  4. Trace a ray through the camera's COP in the direction of the point picked
  5. Find the first object this "infinite" ray intersects with
     - Must either move the ray to each object's model coordinates move the object to the world (via its model transformation matrix)

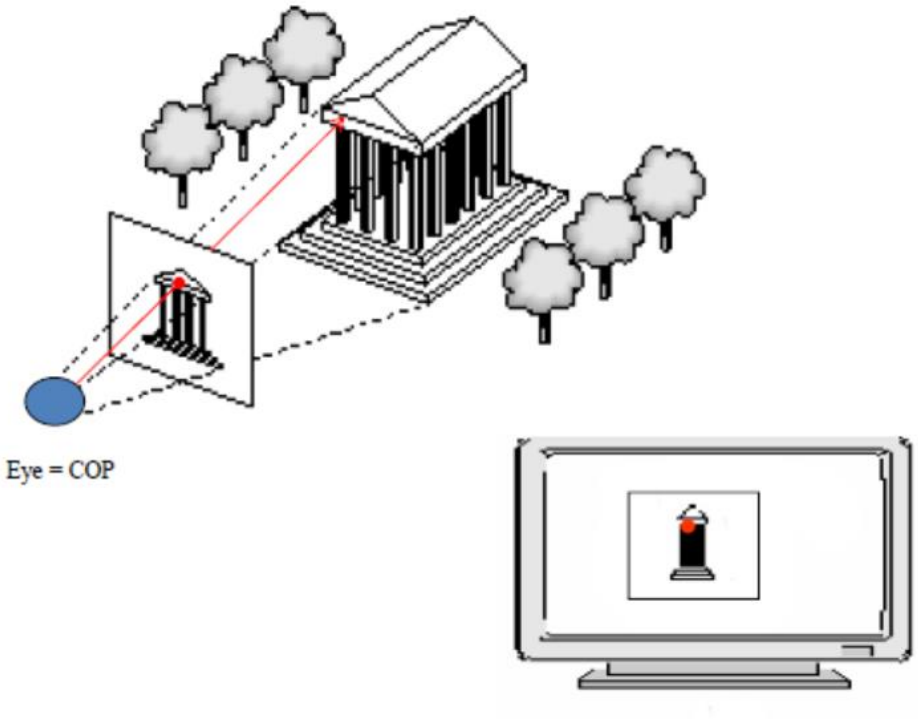
Picking

- Identify the pixel click on the near plane of the camera

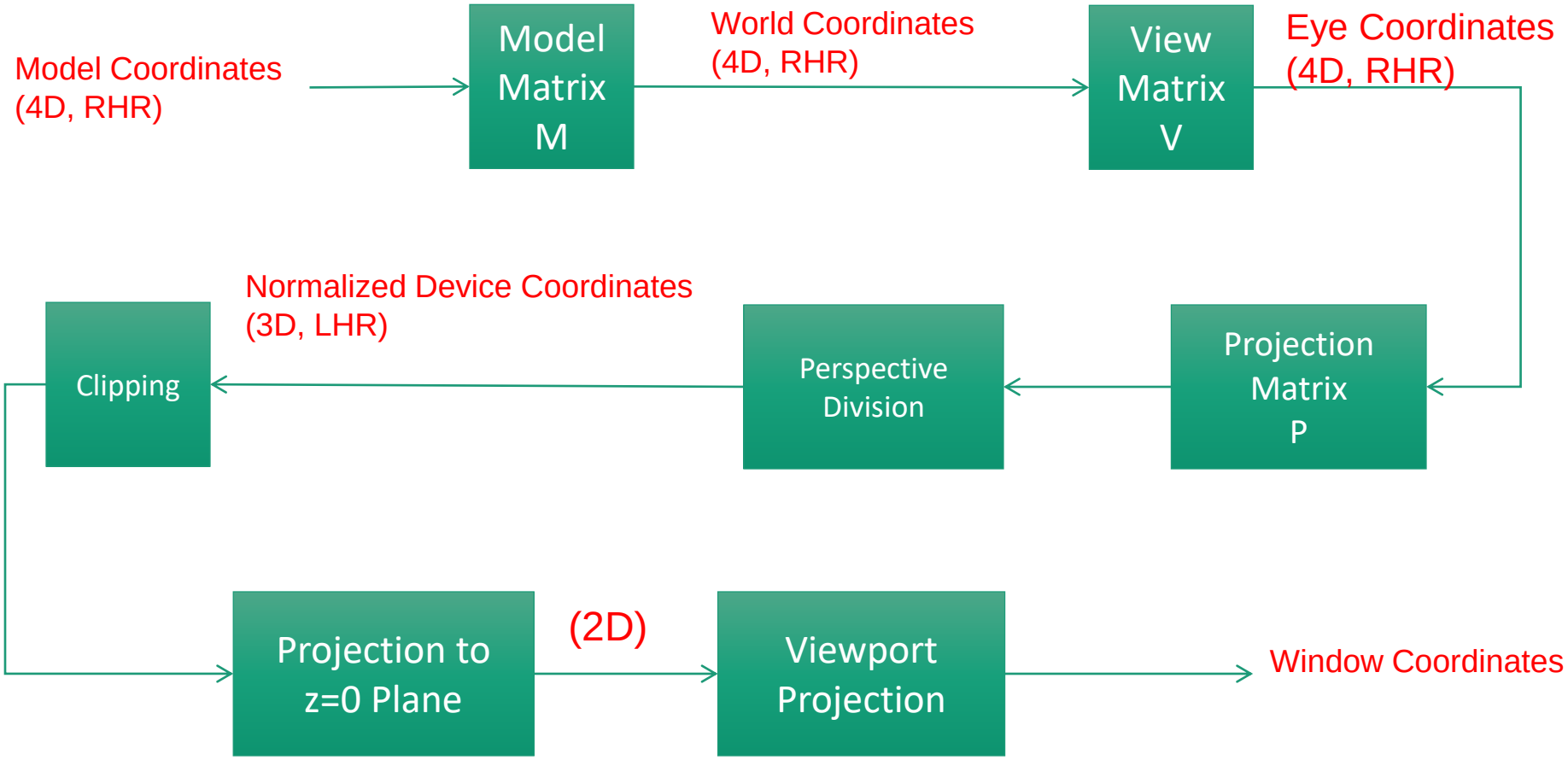


Picking

- Cast a ray through the location



Transformation Pipeline



Inverse Matrix

- Throughout this course we will need to invert many transformations
- We just defined inverses for our perspective projection and camera matrix
- But we may want a generic inverse function for 4x4 matrices
- Try to be efficient and only compute these when needed
  - Projection matrix will only change when reshape occurs
  - Camera matrix will change whenever the camera moves
  - Model matrices will change whenever a model moves

Inverse Matrix

- Here are some resources for computing the inverse of a 4x4 matrix:
  - http://www.cg.info.hiroshima-cu.ac.jp/~miyazaki/knowledge/teche23.html
  - http://stackoverflow.com/questions/1148309/inverting-a-4x4-matrix

Step 1: Pixel to Near Viewing Volume Plane

- Using the screen's width (ww) and height (wh) we can find where on the near plane the pixel $p = (p_x, p_y)$ is:
  - $x' = 2 * \frac{p_x}{ww} - 1$
  - $y' = 1 - 2 * \frac{p_y}{wh}$
- Now the point on near plane of the volume (in homogenous normalized device coordinates, which obeys the LHR, and thus has it's front plane at $z = -1$) is simply:

$$p_{front} = \begin{bmatrix} x' \\ y' \\ -1 \\ 1 \end{bmatrix}$$

Step 2: Adjust for Projection

- Next we must adjust for the projection.
- Let us assume that we used perspective projection.
- Our projection matrix would then be:

$$P = \begin{bmatrix} n/r & 0 & 0 & 0 \\ 0 & n/t & 0 & 0 \\ 0 & 0 & -(f+n)/(f-n) & -2fn/(f-n) \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

  - n is the near plane
  - f is the far plane
  - $t = n * \tan(\frac{fov}{2})$
  - $r = t * aspect$

Recall we specify the n value as *displacement* from the camera, $n > 0$.

Step 2: Adjust for Projection

- We can compute the inverse P^{-1} as

$$P^{-1} = \begin{bmatrix} r/n & 0 & 0 & 0 \\ 0 & t/n & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & (n-f)/(2fn) & (n+f)/(2fn) \end{bmatrix}$$

- Which we can use to find the location in camera coordinates

$$p_{cam} = P^{-1}p_{front} = \begin{bmatrix} rx'/n \\ ty'/n \\ -1 \\ 1/n \end{bmatrix}$$

- But we want our 4d coordinate to be one so

$$p_{cam} = \begin{bmatrix} rx'/n \\ ty'/n \\ -1 \\ 1/n \end{bmatrix} = \begin{bmatrix} rx' \\ ty' \\ -n \\ 1 \end{bmatrix}$$

Step 3: The probe Ray in World Coordinates

- So now we have the point we clicked on in camera coordinates, p_{cam}. Now we need it in world coordinates.
- Recall that our view matrix, V, moved and oriented the camera the result of which is that each pixel is transformed from world coordinates to camera coordinates as

$$p_{cam} = V p_{world}$$

- Therefore, to find the clicked point in world coordinates we must do the inverse

$$p_{world} = V^{-1} p_{cam}$$

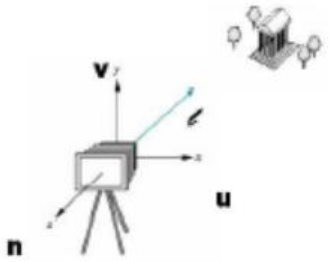
Step 3: The probe Ray in World Coordinates

- The view matrix can be manually created using just the camera coordinate system (u, v, n), and the Eye of the camera

$$V = \begin{bmatrix} u_1 & u_2 & u_3 & -u \cdot Eye \\ v_1 & v_2 & v_3 & -v \cdot Eye \\ n_1 & n_2 & n_3 & -n \cdot Eye \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Instead of inverting the matrix (may be expensive) we can invert our camera coordinate system (Eye, u, v, n) as:

$$V^{-1} = \begin{bmatrix} u_1 & v_1 & n_1 & Eye_1 \\ u_2 & v_2 & n_2 & Eye_2 \\ u_3 & v_3 & n_3 & Eye_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



- So now our point in world coordinates is:

$$p_{world} = V^{-1} p_{cam}$$

Step 4: Probing the World

- Now we have the point in world coordinate, using the camera eye (which is already specified in world coordinates) we should be able to compute the ray in world coordinates:

$$ray_{world} = p_{world} - Eye$$

- Now we need to test if this ray intersects any of these objects.
- Since our objects are typically made up of triangles we need to do *ray→triangle collision detection.*

3D Ray-Triangle Intersection

- Notes Based on:
  - http://www.scratchapixel.com/old/lessons/3d-basic-lessons/lesson-9-ray-triangle-intersection/ray-triangle-intersection-geometric-solution/
    - Note: There's an error. It should be `D=-dot(N,V0);` (notice the negative sign)
  - http://graphics.stanford.edu/courses/cs348b-98/gg/intersect.html

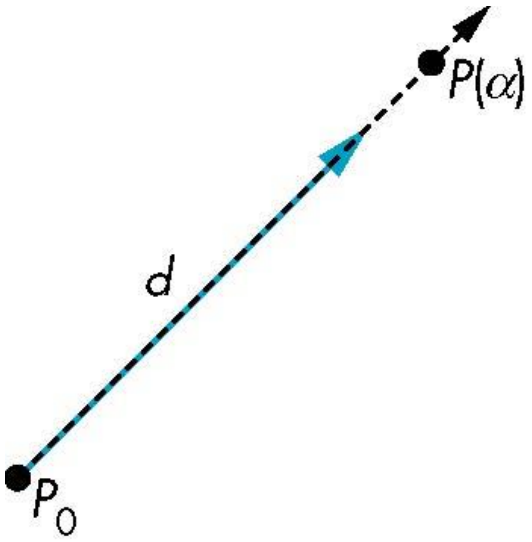
3D Ray-Triangle Intersection

- Our general approach:
  1. Determine if the ray intersects the plane that contains the triangle
  2. If it doesn't, we're done
  3. If it does, find out where, and see if that is inside of the triangle

Parametric Form of Lines

- Given an initial point P_0 and a vector $\boldsymbol{d}$ we can find any point $P(\alpha)$ on the line α units from P_0 in the direction of $\boldsymbol{d}$.
  - This is called the *parametric form* of a line

$$P(\alpha) = P_0 + \alpha \boldsymbol{d}$$

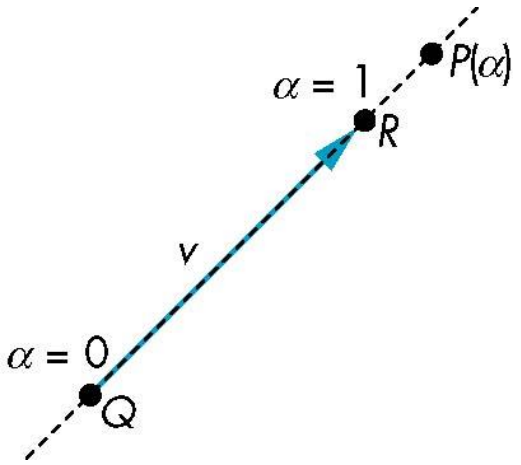


Ray and Line Segments

- If $\alpha >= 0$, then $P(\alpha)$ is a *ray* leaving P_0 in the direction $\boldsymbol{v}$
- If we use two points, Q and R, to define v, then

$$P(\alpha) = Q + \alpha(R - Q)$$
$$= Q + \alpha v$$
$$= \alpha R + (1 - \alpha)Q$$

- For $0 <= \alpha <= 1$ we get all the points on the *line segment* joining R and Q



Planes

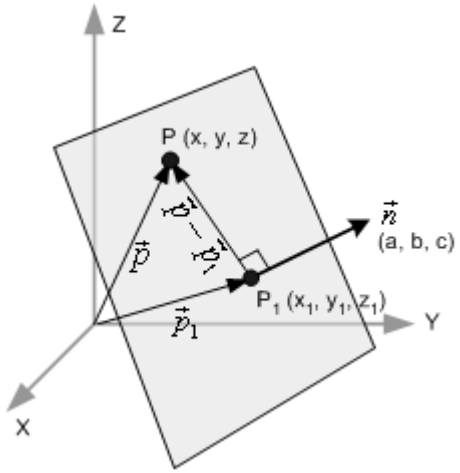
- Also relevant is the *implicit* equation for a plane:

$$N_x x + N_y y + N_z z + d = 0$$

  - Where $N = (N_x, N_y, N_z)$ is the normal of the plane and d is its distance from the origin.
- This equation will be true for all points (x, y, z)on the plane.
- We can also write this as:

$$P \cdot N + d = 0$$

where is some point on the plane
$P = (x, y, z)$



3D Ray-Plane Intersection

- Given 3 points, E, F, G, we can compute the normal of any plane:

$$N = \overrightarrow{EF} \times \overrightarrow{EG}$$

- Using one of the points, say E, we can write the equation of a plane as $E \cdot N + d = 0$.
  - And solve for $d = -E \cdot N$

3D Ray-Plane Intersection

- We want to find the point P cast by a ray such that it is on the plane.
- If this point is to be on the plane then:

$$N_x \cdot P_x + N_y \cdot P_y + N_z \cdot P_z + d = 0$$

- Given the rays origin O and direction D we can write this point in parametric form as:

$$P = O + tD$$

- And via substitution we get:

$$N_x(O_x + tD_x) + N_y(O_y + tD_y) + N_z(O_z + tD_y) + d = 0$$

3D Ray-Plane Intersection

$$N_x(O_x + tD_x) + N_y(O_y + tD_y) + N_z(O_z + tD_y) + d = 0$$

- Solving for t….

$$t(N_xD_x + N_yD_y + N_zD_y) + (N_xO_x + N_yO_y + N_zO_z + d) = 0$$
$$t(N \cdot D) + (N \cdot O + d) = 0$$

- So:

$$\mathrm{t} = -\frac{N \cdot O + d}{N \cdot D}$$

- And recall that
  - $N = (F - E)\ x\ (G - E)$
  - O is the origin of the ray
  - $d\ =\ -E \cdot N$
- Once we have t we can get the position P:

$$P\ =\ O + tD$$

The Dot Product

- In the prior derivation we often used the **dot product** $A \cdot B = \left(A_x B_x + A_y B_y + A_z B_z\right)$
- The dot product can also be computed using the magnitude of A and B and the angle between them, θ

$$A \cdot B = \|A\|\|B\| cos\theta$$

- Therefore given the dot product we can compute the angle between the vectors using the inverse cosine

$$\theta = \cos^{-1}\left(\frac{A \cdot B}{\|A\|\|B\|}\right)$$

The Dot Product

$$A \cdot B = \|A\|\|B\|cos\theta$$

- And if both vectors are unit length then this becomes:

$$A \cdot B = cos\theta$$

- Why is this important?
- We can look quickly at the dot product of two vectors and determine:
  - Are they pointing in the same direction?
  - Are they orthogonal?
  - Is their angle between $90 < \theta < 270$?

3D Ray-Plane Intersection

- Back to Ray-Plane intersection…
- Right away we can determine if the ray and plane will ever intersect:
  - $N \cdot D = 0$
    - The normal and the ray direction are perpendicular
    - Therefor the triangle and the ray are parallel and can never intersect
  - $t < 0$
    - The ray is pointing away from the triangle and therefore won't intersect it
  - $0 \leq t \leq front\ plane$
    - You may want to require that the time of intersection is beyond the front plane of the projection volume (more on this later)
    - Otherwise you may detect a collision that's in front of the *eye* of the camera but behind the front viewing plane.

3D Ray-Triangle Intersection

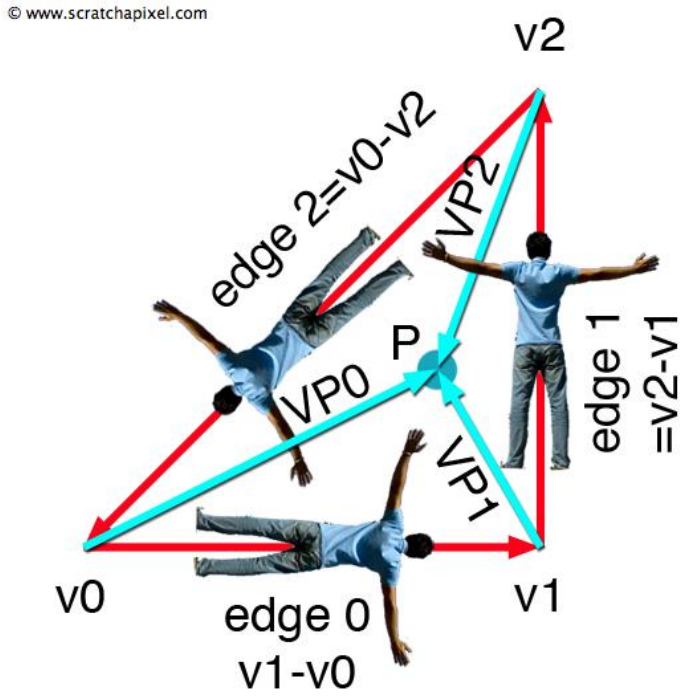
- If we pass the early cases, we must then verify that the intersection point is within the bounds of the triangle
- We can do this using the “inside-outside” test against each edge of the triangle

3D Ray-Triangle Intersection

- Inside-outside Algorithm
  - For each edge of our triangle, proceeding in the same direction (say, clockwise)
    - Get a vector from the first vertex to the second vertex of the edge
    - Get a vector from the first vertex to the intersection point
    - Compute the cross product between the two vectors
    - Compute the dot product between the triangle's normal and this cross product
  - If all of the dot products are non-negative, then we are inside of the triangle!

3D Ray-Triangle Intersection

- This can be generalized for ray-polygon intersection
  - Just need to do the “inside-outside” test again all edges of the polygon in question



3D Collision Detection in Homogenous Coordinates

- **Note: Dot and Cross Products don't mean anything in homogenous coordinates. Therefore copy the 4D points to 3D versions before using them:**
  - For each input, make a corresponding 3D vector, copying the x, y, z values
    - `vec4 point0_4d;`
    - `vec3 point0_3d;`
    - `point0_3d.x = point0_4d.x;`
    - `point0_3d.y = point0_4d.y;`
    - `point0_3d.z = point0_4d.z;`
    - `Etc..`

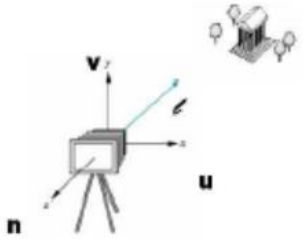
Step 4: Probing the World

- Ok so now we know how to test if a ray intersects a triangle.
- So to figure out which object (if any) you clicked on, take your ray (that's now in world coordinates) and for each object
  - Find out if it hit any of the objects triangles (you'll have to first put them in world coordinates, $p_{world} = Mp_{model}$).
  - Keep track of the earliest (smallest t) hit.
- If at least one object was hit, choose the one that was earliest.
- Note: Now you may add a `testCollision` method to our `Drawable` objects

Step 4: Probing the World

- Note: This is slow and there's better ways
  - We should only deal with objects in our clipping volume
  - Instead of testing against polygons, we can approximate objects with bounding objects and see if we intersect them
    - Bounding boxes
    - Spheres

Summary



1. Take the pixel that was clicked on and find it's location on the front of the camera:

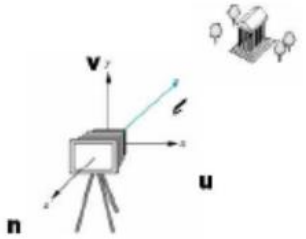
$$p_{front} = \begin{bmatrix} x' \\ y' \\ -1 \\ 1 \end{bmatrix} = \begin{bmatrix} 2\left(\frac{p_x}{ww}\right) - 1 \\ 1 - 2\left(\frac{p_y}{wh}\right) \\ -1 \\ 1 \end{bmatrix}$$

2. Reverse project this point to find it's location in the view volume

$$p_{cam} = \begin{bmatrix} rx' \\ ty' \\ -n \\ 1 \end{bmatrix}$$

Where $t = \tan\left(\frac{fov}{2}\right), r = t * aspect$

Summary



3. Put the point in the world by reversing the view matrix

$$p_{world} = \begin{bmatrix} u_1 & v_1 & n_1 & Eye_1 \\ u_2 & v_2 & n_2 & Eye_2 \\ u_3 & v_3 & n_3 & Eye_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} p_{cam}$$

4. Now for every triangle in every object
   1. Put the vertices v of the triangle in the world via $v_{world} = Mv$
   2. Test if the ray intersects that triangle and compute the intersection time/location.
5. Choose the closest intersection point (smallest time) that is greater than the near plane.

Collision Detection

- All of these same ideas can be used for collision detection.
- However, with collision detection we often want to test if we collided with a *line segment* not a *ray*
- The only difference here, is that for a line segment to intersect a plane, $0 \leq t \leq 1$

Collision Detection

- Example: Colliding the camera with objects.
- Given the camera's current eye and the plane normal, n, moving in the direction of $-n$ would create a line segment with endpoints $(eye, eye - n)$
- In addition, you have the "offset" of the front plane, so maybe we want to take into consideration the line segment $(eye, eye - n + front)$
- Now we test this segment against all triangles of all objects.
- If any of them intersect this line segment within $0 \leq t \leq 1$ then we can't move there and should remain at position eye