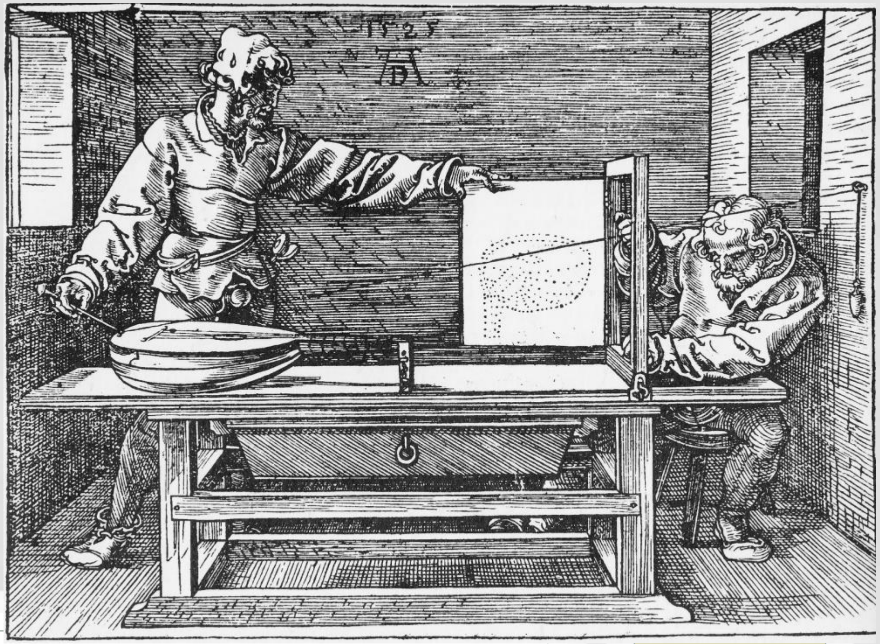


PERSPECTIVE PROJECTIONS + RASTERIZER PIPELINE

References:

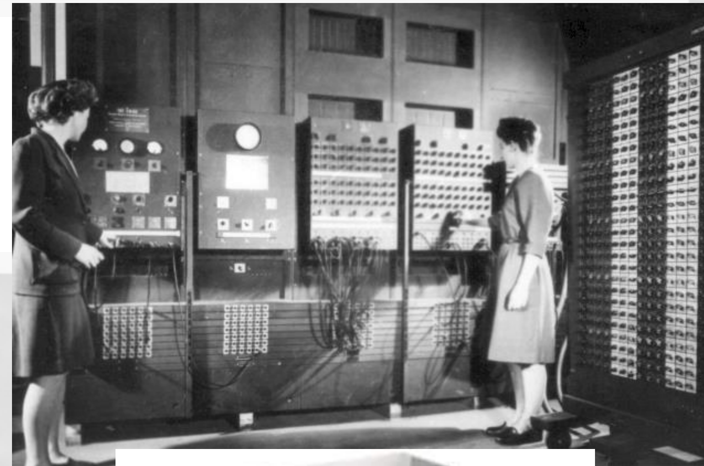
- http://www.scratchapixel.com/lessons/3d-basic-rendering/perspective-and-orthographic-projection-matrix/projection-matrix-introduction
- http://www.scratchapixel.com/lessons/3d-basic-rendering/perspective-and-orthographic-projection-matrix/building-basic-perspective-projection-matrix
- Computer Graphics: Principles and Practice in C (2nd edition)

OVERVIEW

- Lab8: Wireframe Rasterizer
  - No notion of camera
  - Orthogonal Projection
  - No filled polygons
  - No lighting
  - Just the first phase in the rasterizer **pipeline**.
- Lab9 attempts to add some / all these.
- Slides marked with a * are more likely to be on the final…

GPU HISTORY

- From ca. 1960 – 1980:
  - Workstation software-only rendering
- Ca. 1980 - 2000
  - PC's + [later] Accelerated Graphics cards
    - Fixed-function pipeline
    - Ca. 1990 = OpenGL
    - Single-core GPU's??? [check this]
- Ca. 2000 – 2006
  - Programmable shader GPU's
    - GeForce 3 (PC, Xbox [original])
    - Still Single core??? [check this too]
- Ca. 2006 – present
  - General purpose GPU's
    - GeForce 8
    - Many, many cores.
- All 4 generations use the same math
  - Just expose it differently.
  - Later generations aren't better – just faster.



* RASTERIZATION PIPELINE

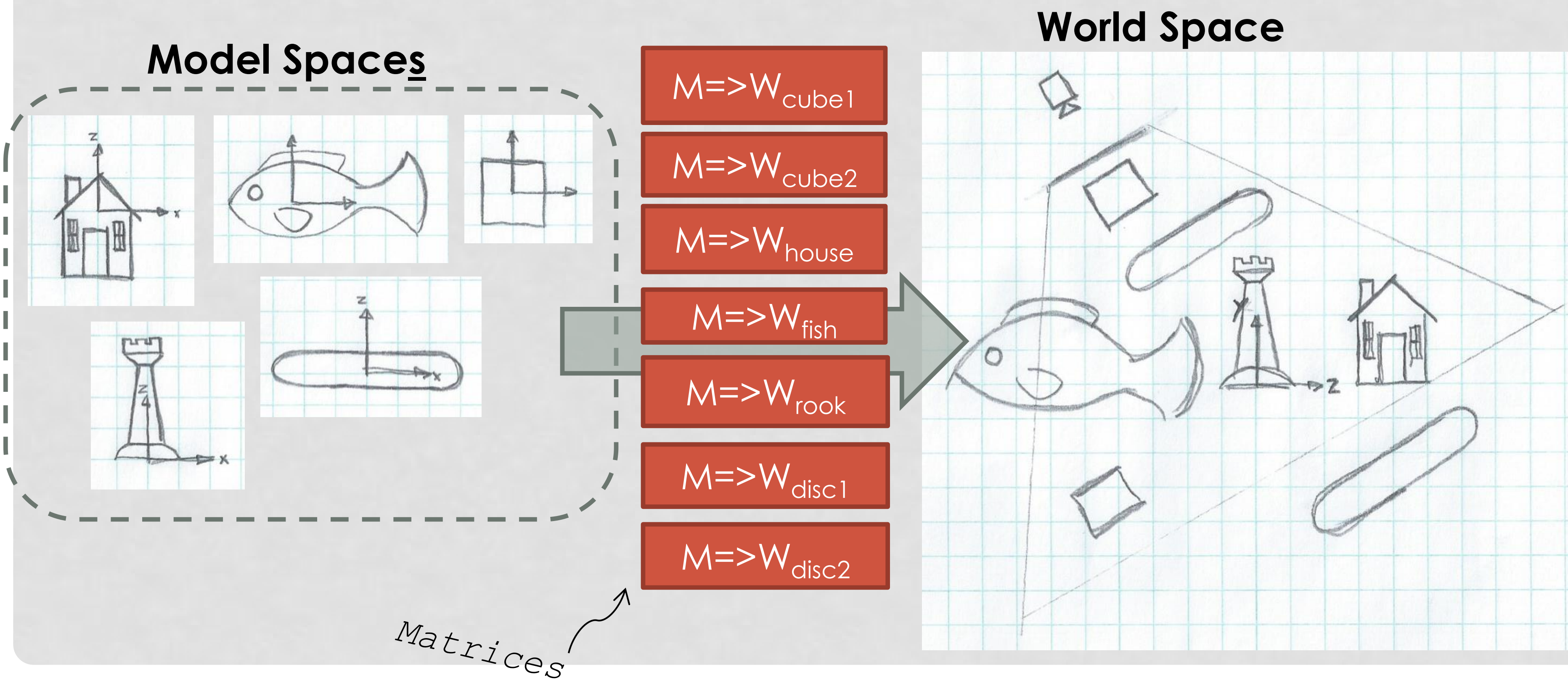
- Meshes => Pixels
  - A mesh is a collection of vertices.
  - Vertex * transformMatrix = Vertex’
  - Vertex’ is in a new **space**.
  - So far we’ve explored:
    - Model Space
    - World Space
  - The transform matrix essentially converts from one space to another.
  - The rasterizer is a long sequence of matrix transforms.

*VRASTERIZATION PIPELINE, CONT.

- Major Spaces
  - **Model Space**: as the model appears relative to blender / maya axes.
  - **Camera Space**: objects are all relative to the rendering camera.
  - **Projection / Clip Space**:
    - Perspective Projection + **Homogeneous Divide**
    - Orthogonal Projection
    - Isometric Projection
    - ….
  - **Screen Space**: pixels (with depth)
- ***We go from one space to the next with a matrix.***
  - Note: The matrices I'm giving you are for a left-handed system.

* MODEL => WORLD

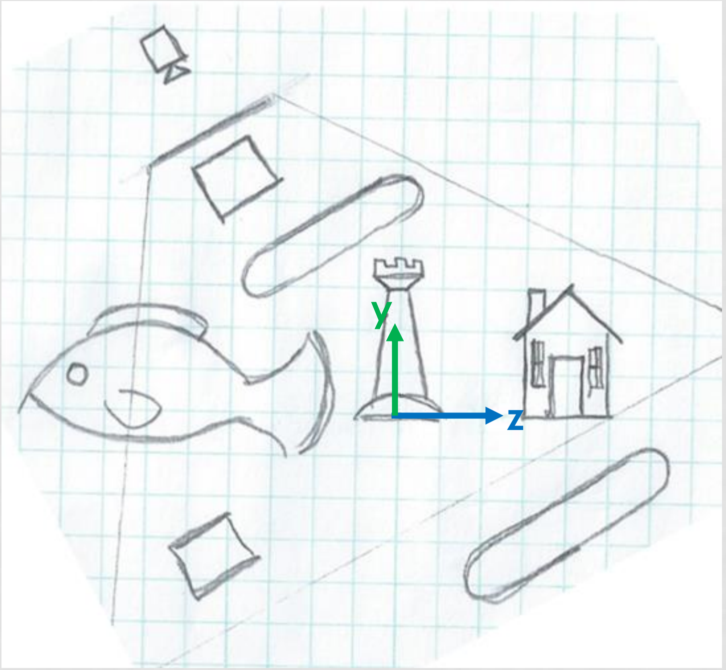
- This is what we were doing in Lab8
- This could be accomplished with a scene graph *or* a single (possibly concatenated) matrix.



* WORLD => VIEW

- Move *everything* so camera is at the origin and aligned with world axes.
- View / Camera space are the same thing.

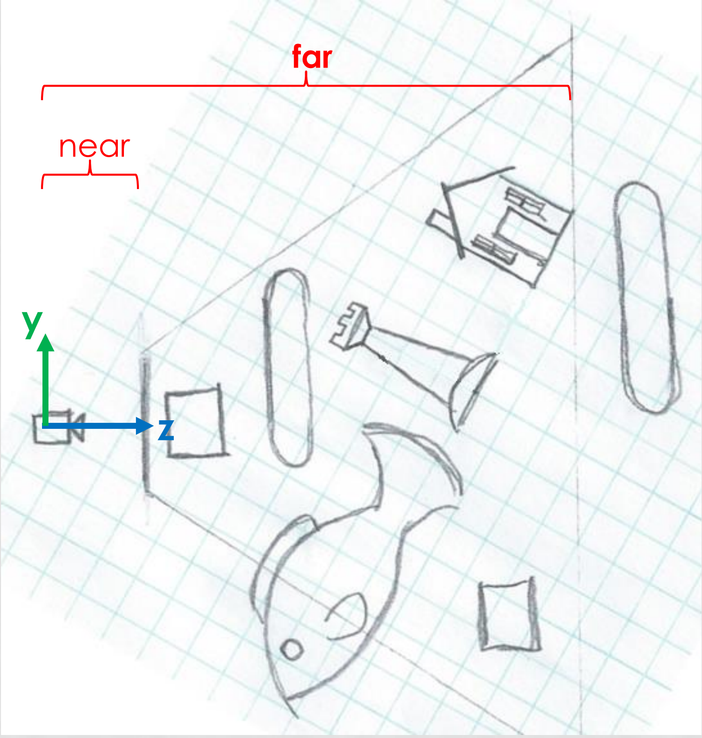
World Space



W=>V

viewplane_height

View Space



WORLD => VIEW, CONT.

- Now, to construct the W=>V matrix…
  - T = Translate (enough to make camera at origin)
  - R = Rotate (to align the camera axes with world axes)
    - Q = Rotate world axes to camera axes

$$Q = \begin{bmatrix} \widehat{camX}_x & \widehat{camX}_y & \widehat{camX}_z & 0 \\ \widehat{camY}_x & \widehat{camY}_y & \widehat{camY}_z & 0 \\ \widehat{camZ}_x & \widehat{camZ}_y & \widehat{camZ}_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

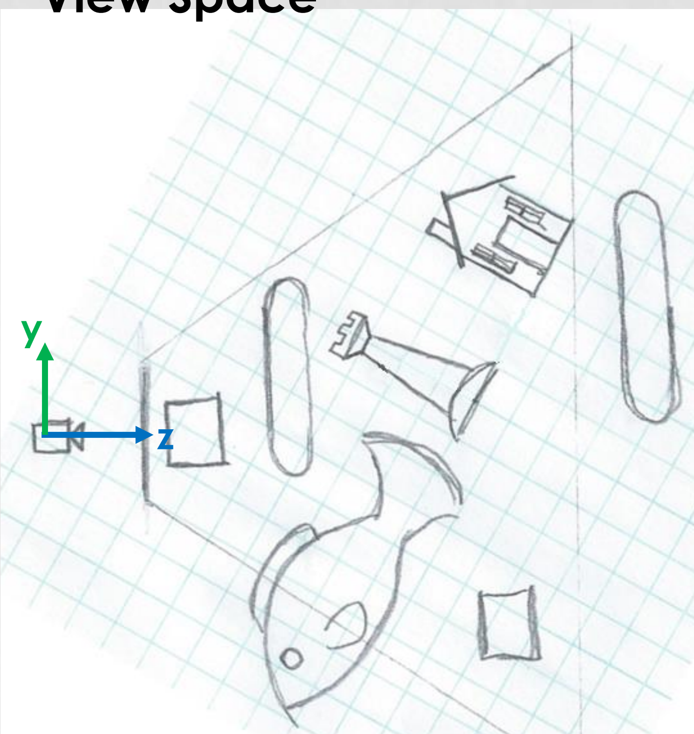
    - R is just Q^T
    - Side-note: R is the inverse of Q. Since Q is an orthonormal matrix, the transpose is a matrix
      - Ortho = all columns (or rows) are unit-length vectors
      - Normal = all columns (or rows) are perpendicular.

VIEW => PROJECTION

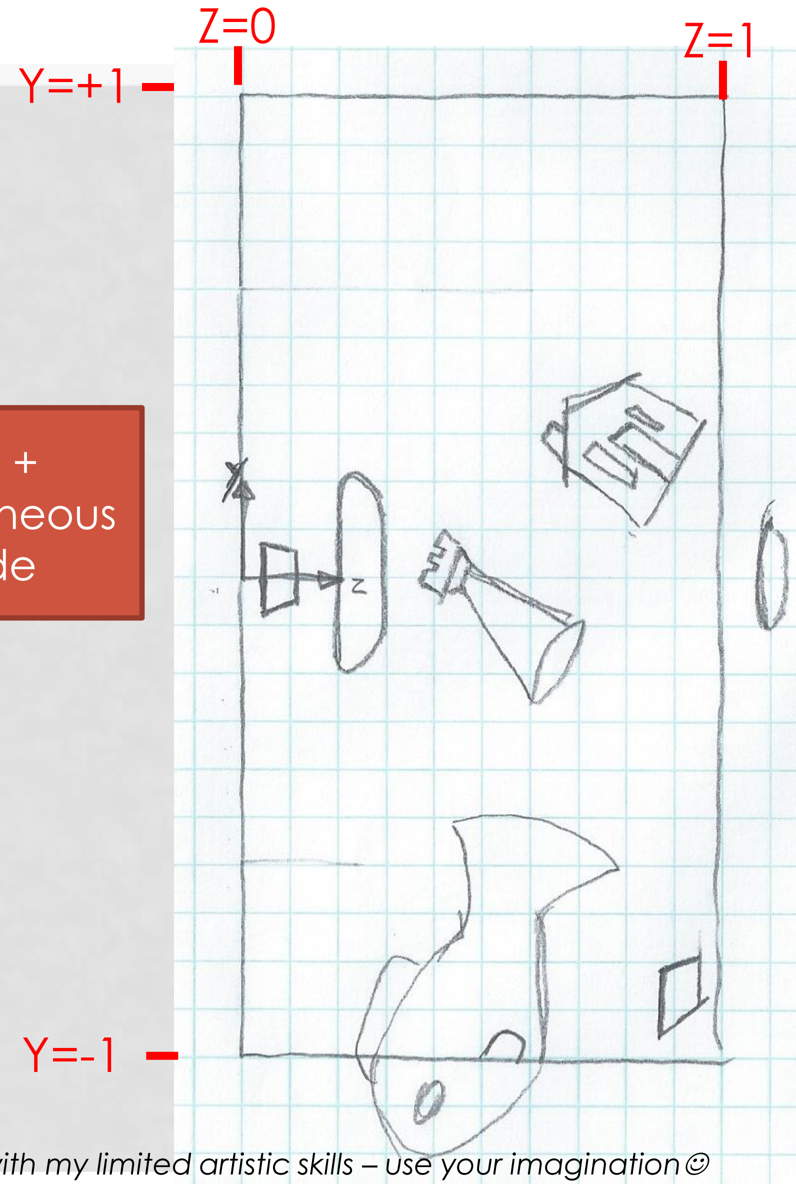
- I'm going to focus on Perspective Projection
- Goal: Introduce the perspective effect
  - Farther away objects look smaller
- We do this by compressing the **view frustrum** into a cube, along with all objects in the world.
- This is the one (and only) thing we can't do with a matrix alone.
- Normally we'd clip out invisible geometry
  - That's why this space is sometimes called **clip space**.

* VIEW => PROJECTION, CONT.

View Space



V=>P + homogeneous -divide



This is as close as I could get with my limited artistic skills – use your imagination☺

VIEW => PROJECTION, CONT.

- Here is the V => P matrix:
  - I'd like to show you the derivation, but no time…

$$V2P = \begin{bmatrix} \frac{2*near}{vpw} & 0 & 0 & 0 \\ 0 & \frac{2*near}{vph} & 0 & 0 \\ 0 & 0 & \frac{far}{far-near} & 1 \\ 0 & 0 & -\frac{far*near}{far-near} & 0 \end{bmatrix}$$

- After the transformation, the w component of points will be equal to the z component in view space.
- We need to divide all elements by w
  - This finishes the perspective transformation
  - And re-sets the w component to 1.

CLIPPING

- In a real rasterizer, we could clip polygons outside the clip space.
  - That's why what I call Perspective Space is sometimes called Clip Space.
- We'll take a simpler (slower) approach…

* CLIP => SCREEN

- Converts clip-space coordinates into screen-space.
- The x/y values are most important…
- …but the z-value is still important
  - For knowing what's in front of what
  - Useful in the rasterization stage (next)
- Main idea:
  - S = Scale in x / y direction to match screen dim.
  - T = Translate such that origin is at window origin
  - Clip2Screen = S * T

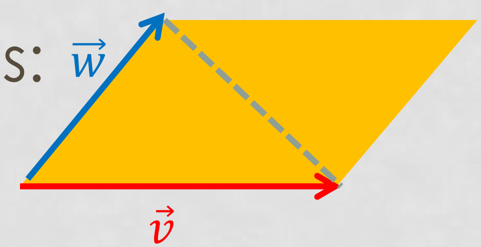
POLYGON RASTERIZATION

- The pipeline is finished
  - We have a set of polygons in screen space.
  - We now need to fill in pixel colors.
- I’m going to show you the simplest (imo) rasterization technique.
- It’s based on areas of triangles…

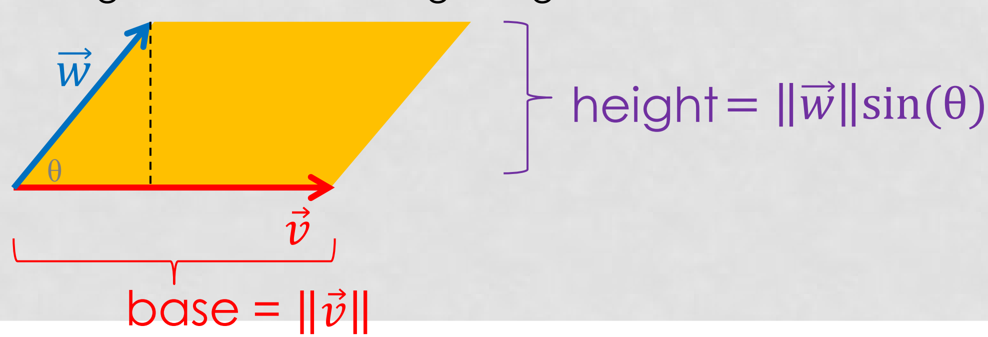
*AREA OF TRIANGLE (IN 3D)

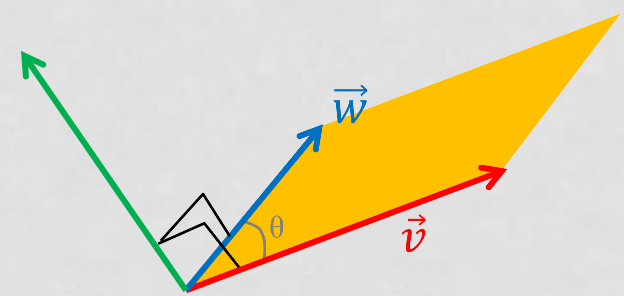
- A cross product property: $\|\vec{v}\ x\ \vec{w}\| = \|\vec{v}\|\|\vec{w}\|\sin(\theta)$
- Imagine a parallelogram with sides $\vec{v}$ and $\vec{w}$
- Recall: area of a parallelogram is base * height
- So…the area is $\|\vec{v}\| * (\|\vec{w}\| \sin(\theta))$
- Which is just $\|\vec{v}\ x\ \vec{w}\|$
- The area of the triangle indicated here is:

$$\frac{\|\vec{v}\ x\ \vec{w}\|}{2}$$



the parallelogram viewed along the green arrow





* BARYCENTRIC COORDINATES

- If you did the bonus on Lab5, this may look familiar.
- Suppose you are given:
  - 3 points ($\vec{A}, \vec{B}, and\ \vec{C}$) that make a triangle (none are equal)
  - A single point $\vec{P}$
    - For this problem, assume $\vec{P}$ lies upon the plane defined by the 3 points.
    - Determining this would make a good final exam problem…
- We want to determine if $\vec{P}$ is within the triangle or not.

* BARYCENTRIC COORDINATES, CONT.

1. Compute the area of the triangle

$$area(ABC) = \frac{\|(\vec{C} - \vec{A})\ x\ (\vec{B}\ -\ \vec{A})\|}{2}$$

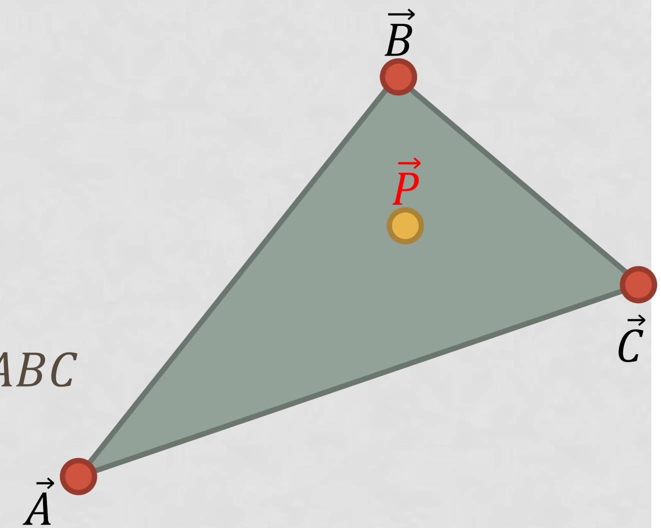
2. $\vec{P}$ is within the triangle iff:

$$area(ABC) - \varepsilon \leq area(PBC) + area(PAC) + area(PAB) \leq area(ABC) + \varepsilon$$

$$\varepsilon\ \approx 0.00001$$

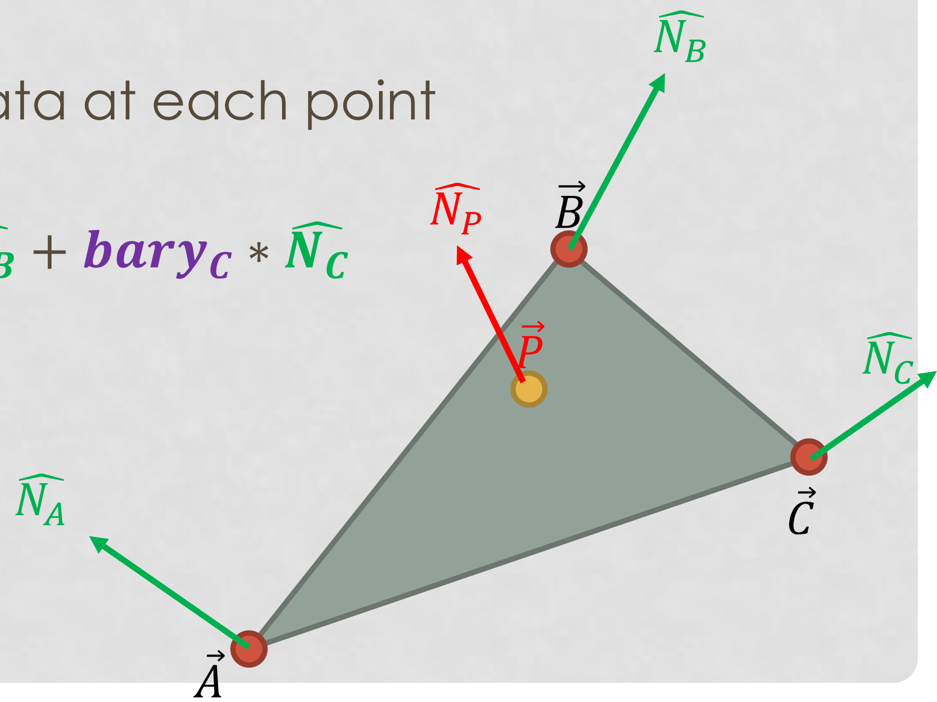
3. The barycentric coordinates are:

- $bary_A = \frac{area(PBC)}{area(ABC)}$
- $bary_B = \frac{area(PAC)}{area(ABC)}$
- $bary_C = \frac{area(PAB)}{area(ABC)}$
- Note: $bary_A + bary_B + bary_C\ \approx 1.0\ if\ P\ within\ ABC$



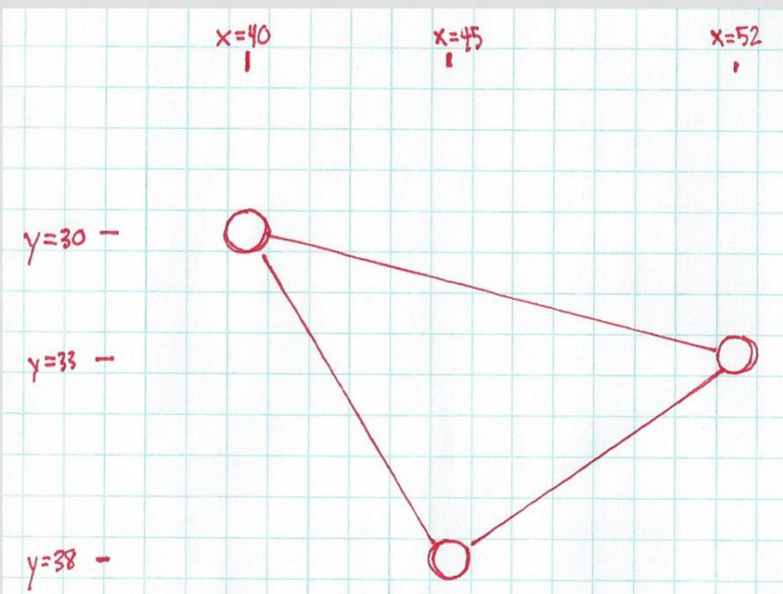
* BARYCENTRIC COORDINATES, CONT.

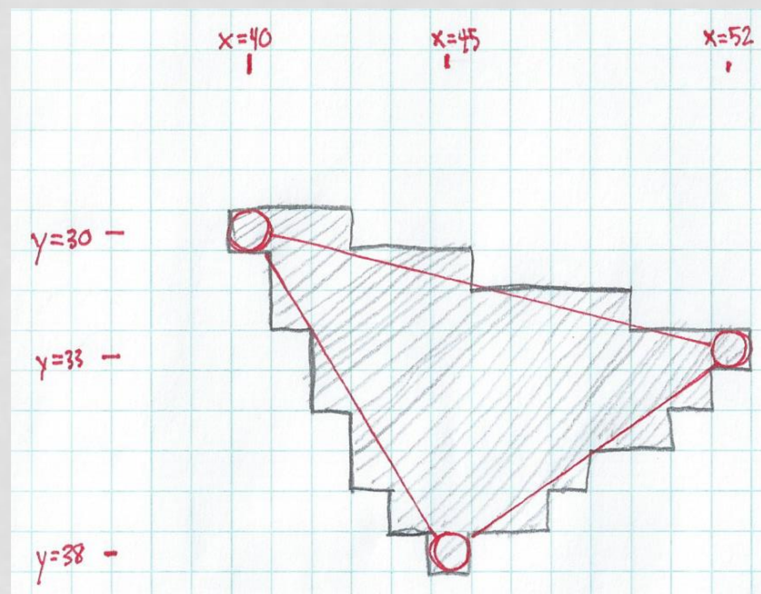
- Barycentric coordinates aren't just useful for hit-detection (as they were within the raytracer)
- They can be used as **weight values** for interpolating data.
- Example: normals
  - Suppose we have normal data at each point
    - Hint: vn lines in the obj file.
  - $\widehat{N_P} = bary_A * \widehat{N_A} + bary_B * \widehat{N_B} + bary_C * \widehat{N_C}$



TRIANGLE RASTERIZATION

- We can also use barycentric coordinates for triangle rasterization.
  - Bounding box to limit candidate pixels
  - Test each internal pixel (as P) against the barycentric test.





A LOOK FORWARD TO ETGG2801

- You *might* touch on these detail again
  - More thoroughly☺
- You definitely will be exposed to:
  - OpenGL
    - Matrix-based (but sort of hidden)
    - Shader-based
      - We don't use the "old-school" pipeline here
      - It's still very much alive in the guts of OpenGL, though.
      - Shaders = mini-programs to control:
        - Lighting
        - Geometry distortions
        - Skeletal animation
        - FSAA
        - ...
    - Bindings in Python / C / Java / etc.
- The class is fast-paced
- Summer project?
  - http://www.opengl-tutorial.org/ [C-based]
  - http://pyopengl.sourceforge.net/context/tutorials/index.html [Python-based]
  - Something else – use your google-fu!

THE END OF ETGG1803!!!

- (Almost☺) Just get through finals week.
- It’s been a pleasure – I appreciate all your hard work this semester!!