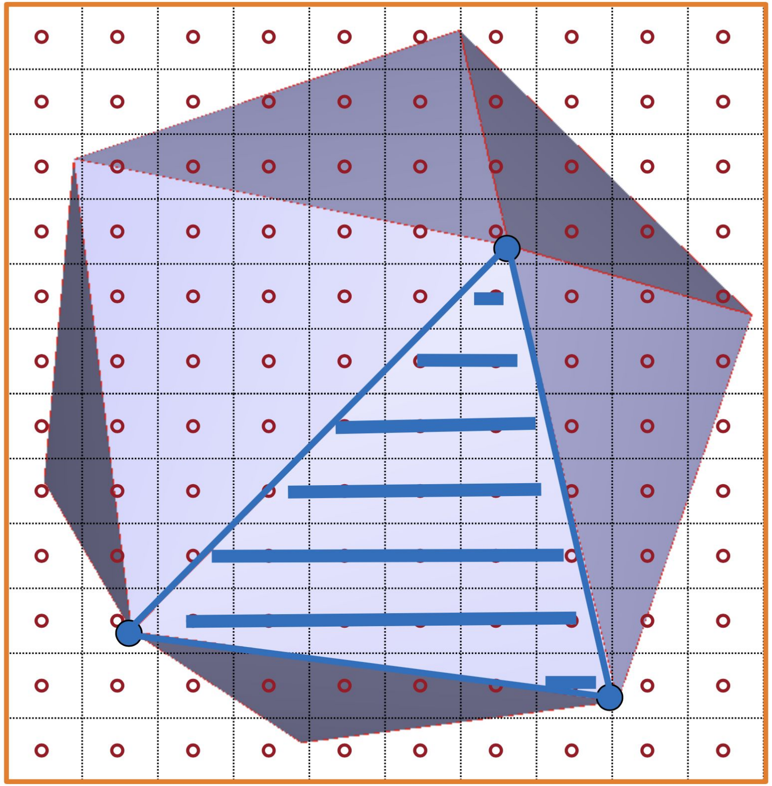
Introducing Assignment 4: Rasterizer

Jiaqi & Carlo

What is Rasterization



Efficiently render 3D primitives to a 2D image

Why Rasterization (vs. Ray Tracing)

- Less Computationally Expensive:

“For this object, decide which pixels that will have their color affected”

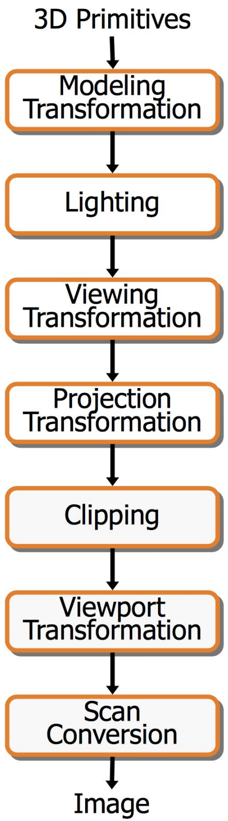
versus

“For this pixel, decide which objects affect its color”

- Take advantage of spatial coherence of 3D primitives

- But...it doesn’t model the actual behavior of light

Rasterization Pipeline



Transform into 3D world coordinate system

Illuminate according to lighting and reflectance

Transform into 3D camera coordinate system

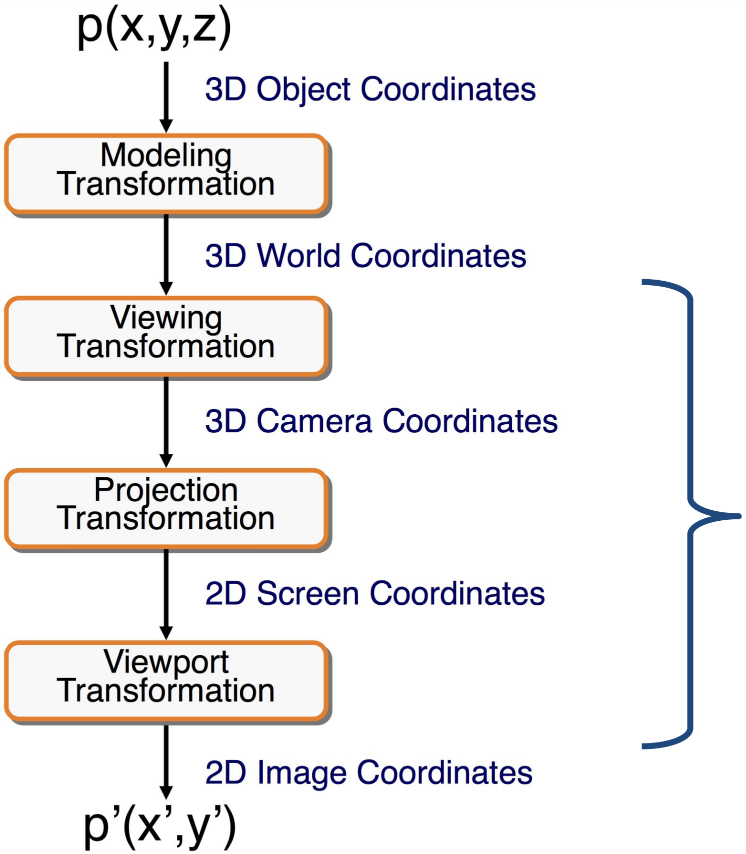
Transform into 2D camera coordinate system

Clip primitives outside camera’s view

Transform into image coordinate system

Draw pixels (includes texturing, hidden surface, ...)

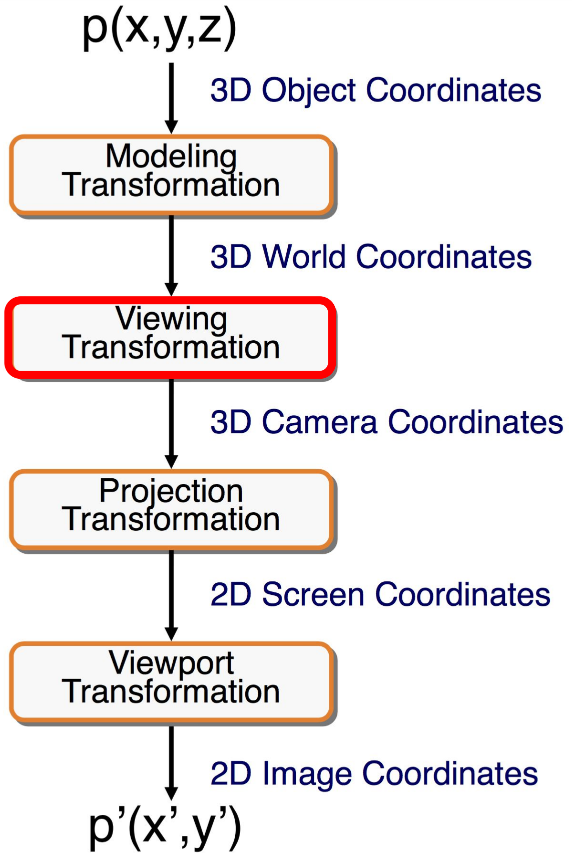
Transformation Pipeline



To be implemented in A4:

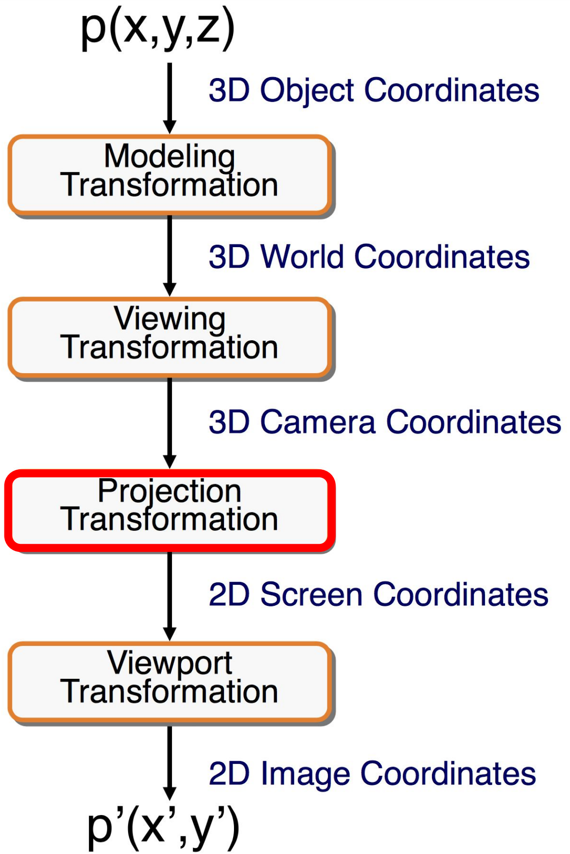
Transform a triangle with 3D world coordinates to a projected triangle with 2D image coordinates

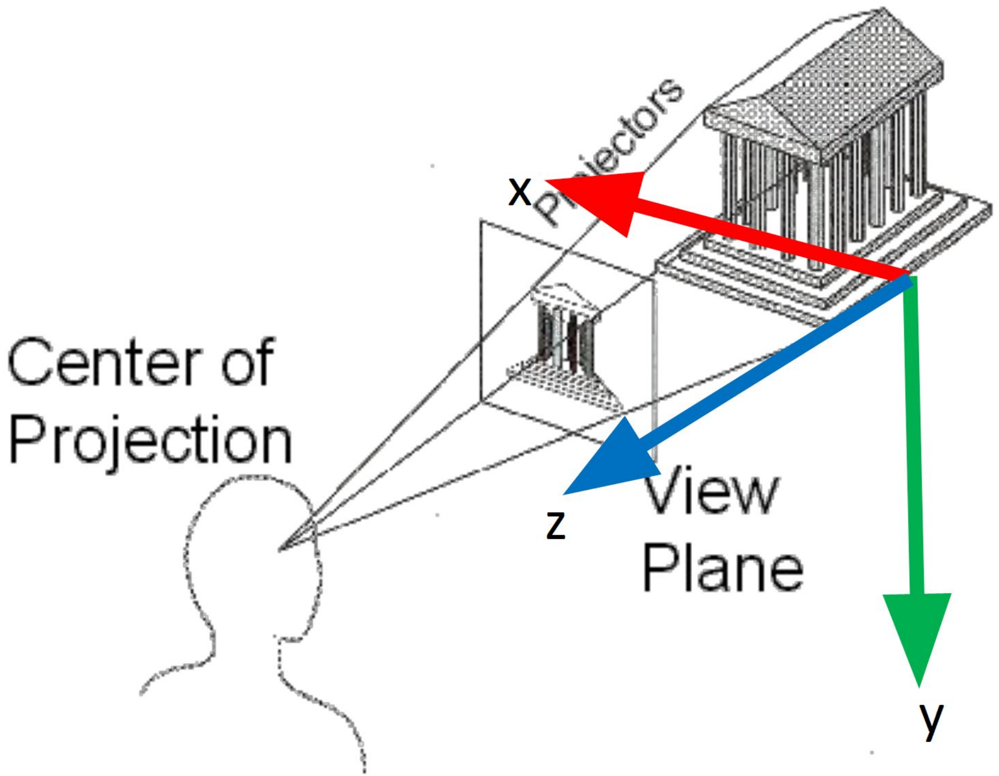
Viewing Transformation



- We represent the pose of the camera in the world space as: [R | t], in homogeneous form (4x4 matrix).
- [R | t] serves to transforms a point represented in the camera coordinate system to the world coordinate system.
- To transform a point in the world coordinate system to the camera coordinate system, we simply apply the inverse of [R | t].

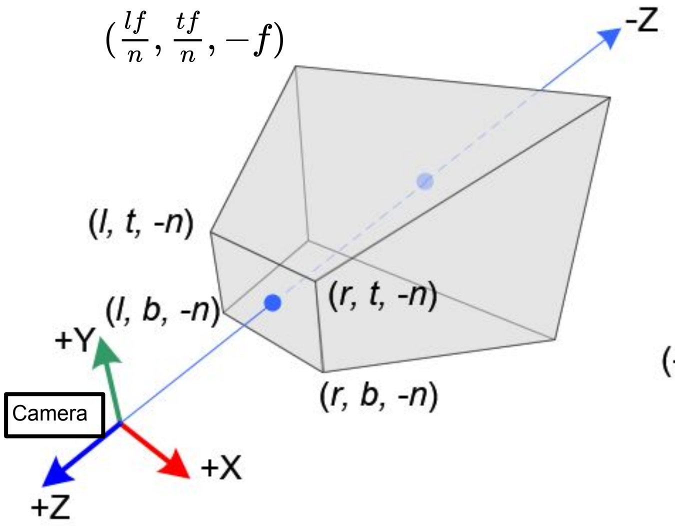
Perspective Projection Transform



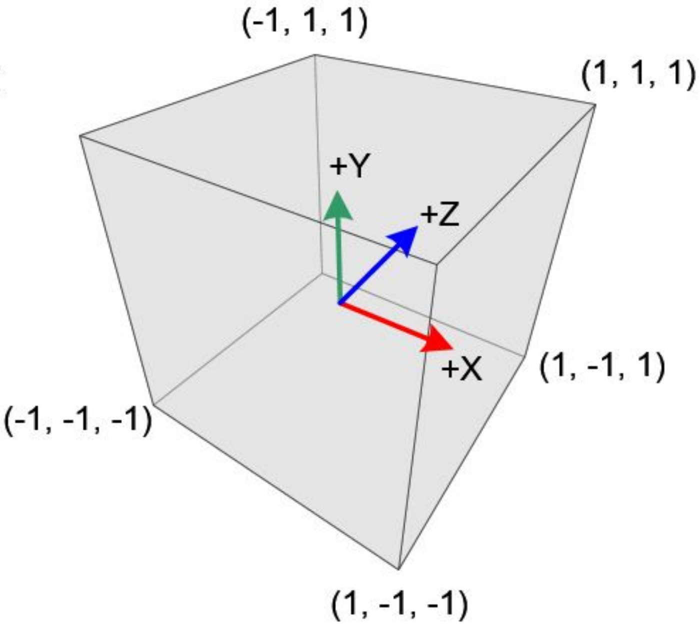


View Volumes

* n and f are usually positive values. But by convention, the near plane is located at –n and the far plane is located at –f.

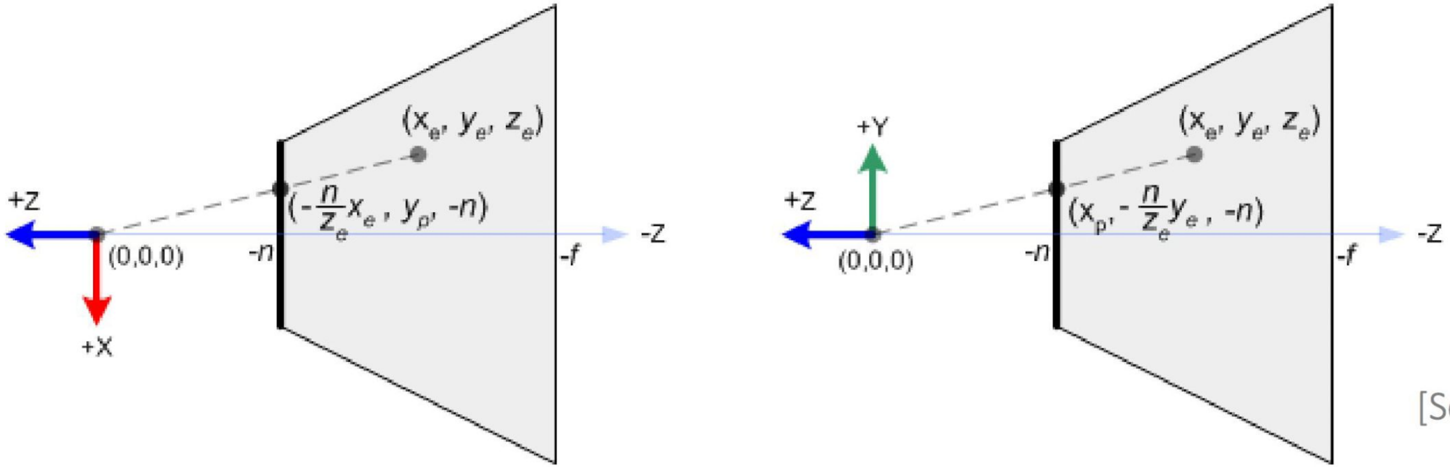


Camera Coordinates
with truncated pyramid frustum view volume defined by
([l, r] in x, [b, t] in y, and [-n, -f] in z)*



Normalized Device Coordinates (NDC)
with canonical view volume of a cube
([l, r] → [-1, 1] in x, [b, t] → [-1, 1] in y, and [-n, -f] → [-1, 1] in z)

Transform to Canonical View Volume



[Song Ho Ahn]

Perspective Projection Matrix

- The matrix that does the desired transformation from frustum view volume to canonical view volume:

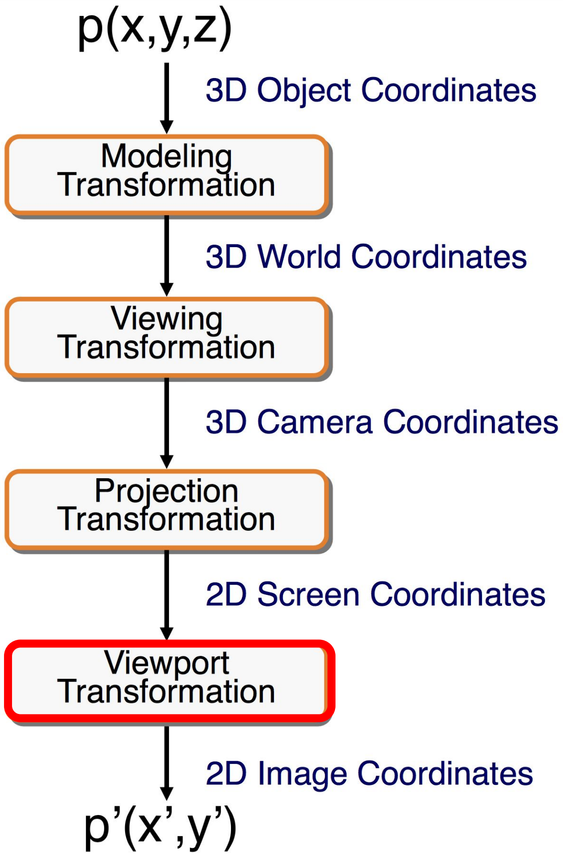
$$\begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Perspective Projection Matrix (Cont')

- What is the fourth dimension?
  - This matrix is in homogeneous form and it should be multiplied with 4D homogeneous coordinates.
  - To lift a 3D nonhomogeneous coordinate, (x,y,z) → (x, y, z, 1). Then you get (x', y', z', w) after applying the transformation.
  - To project a 4D homogeneous coordinate to a 3D nonhomogeneous coordinate:
    (x', y', z', w) → (x'/w, y'/w, z'/w)
  - Especially note that by the design of the transformation matrix, w = -z
  - if camera space z is outside (near, far), skip the triangle because it shouldn't be seen.

$$\begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Viewport Transformation



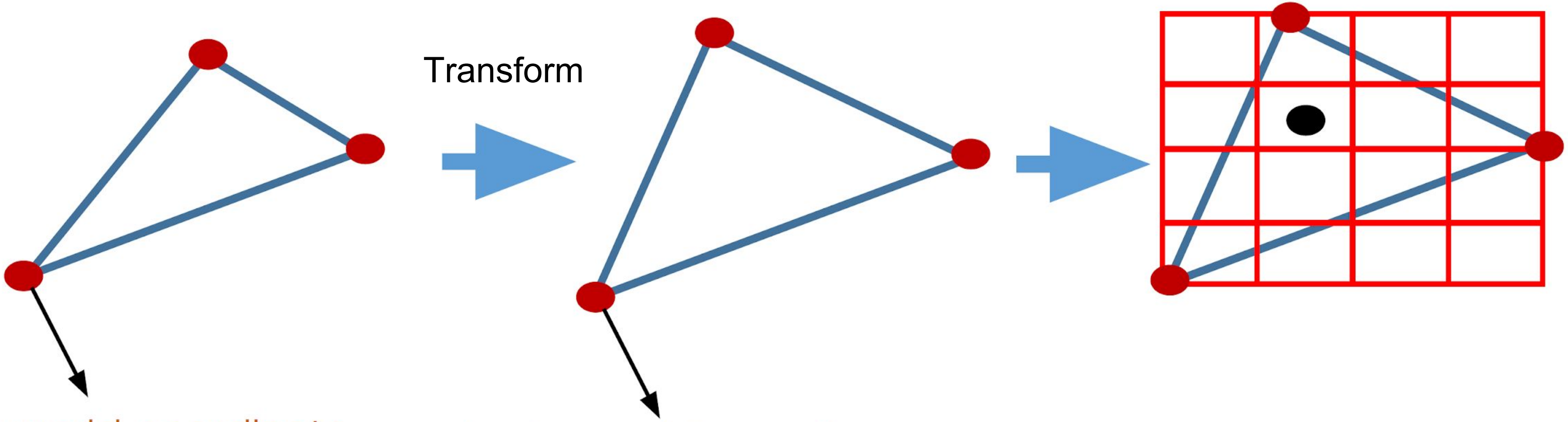
- Transform Normalized Device Coordinates (NDC) to screen space/image space:
  - x: [-1, 1] → [0, image width]
  - y: [-1, 1] → [0, image height]

- Any locations that are outside the screen/image should not be rendered.

Implementation Hints for Transform

- In A4's code, we already pre-computed viewMat := projMat * inv([R | t])
- Your job is to correctly apply viewMat to transform triangle in 3D world coordinates to projected triangle in Normalized Device Coordinates (NDC), and then re-scale it to renderer space coordinates.

Pipeline of Rendering a Triangle



In the world coordinate system: verts[], normals[], uvs[](optional), material(optional).

In the world coordinate system: verts[], normals[], uvs[](optional), material(optional).
In the camera coordinate system: projectedVerts[].

Pipeline of Rendering a Triangle (Cont')

For a pixel (x, y) in the bounding box of the projected vertices:

1. Determine whether it's inside the triangle (barycentric coordinates). If not, go to the next pixel.
2. Use barycentric coordinates to interpolate z'/w (aka. z coord in NDC) for the pixel.
3. If the interpolated z'/w is not smaller (closer) than the value in z buffer for this pixel (smallest depth encountered so far), go to the next pixel.
4. If the pixel survives, render the pixel!

Pipeline of Rendering a Pixel (Example)

To render a pixel, we need the following ingredients.

- Normal of the pixel in the world coordinate system (i.e. interpolate using the three vertex normals and barycentric coordinates).
- Position of the pixel in the world coordinate system (i.e. interpolate using the three vertex positions and barycentric coordinates).
- View position (where your camera/eye is, in the world coordinate system).
- Light position(s) (where the light source is, in the world coordinate system).
- Material of the pixel:
  • case 1: material is uniform or per-vertex (k_a, k_d, k_s shininess).
  • case 2: texture maps. (we need UV coordinates to look up (k_a, k_d, k_s shininess) of the pixel). UV coordinates can also be interpolated using the three vertex UV coordinates and barycentric coordinates).

Apply Phong Reflection Model with the ingredients above to get color.

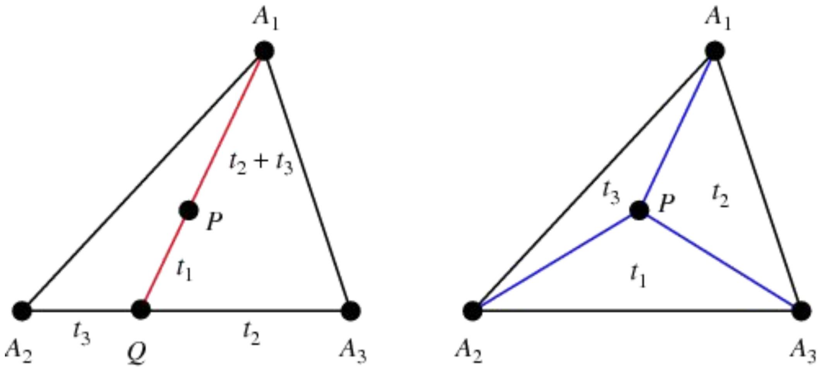
Texture Mapping

- UV coordinates
  - They specify the location of a vertex in the texture map.
  - Not defined for all meshes! Make sure to check whether uvs[] is defined or not.
- Normal Mapping
  - Store normal vector information to an image
  - Still use UV coordinates to specify the location of a vertex in the map.
  - In A4, you are asked to implement XYZ normal mapping:
    - Assuming RGB is in [0, 1], XYZ = 2*RGB - 1
    - Normalize XYZ to obtain unit normal vector
  - Other types, such as tangent space normal mapping

Barycentric Coordinates

Any point in the triangle can be represented as a convex combination of the three vertices

- Q is a linear combination of A2 and A3
- P is a linear combination of Q and A



See slides 30-33 at
https://www.cs.drexel.edu/~david/Classes/CS430/Lectures/L-10_NURBSDrawing.pdf
for efficient 2D algorithm.