**2Please check! Any comments appreciated, thanks.**

1a) i) Identity

ii) Translation

iii) Reflection

iv) Scaling

v) Rotation

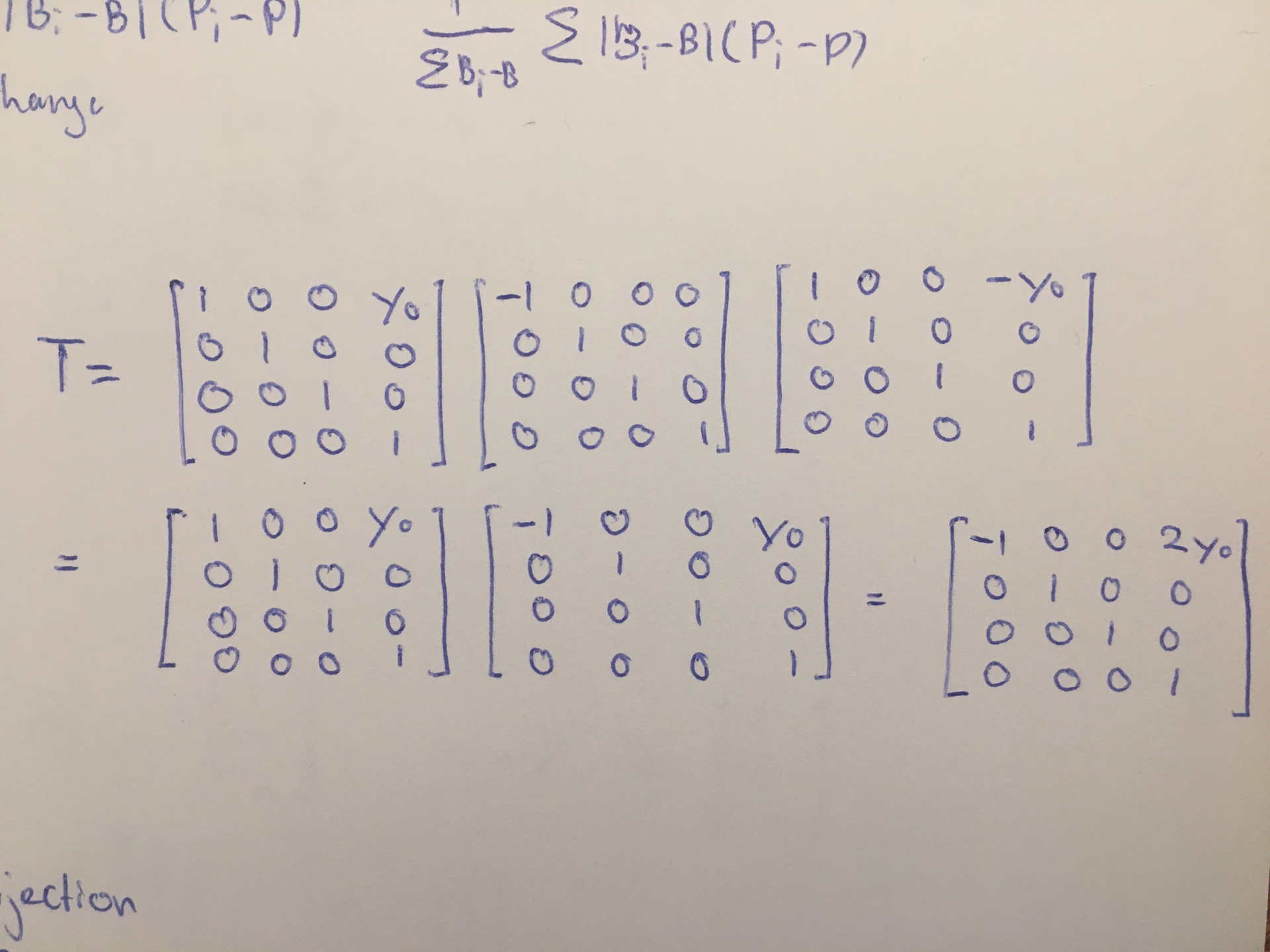
vi) Skew

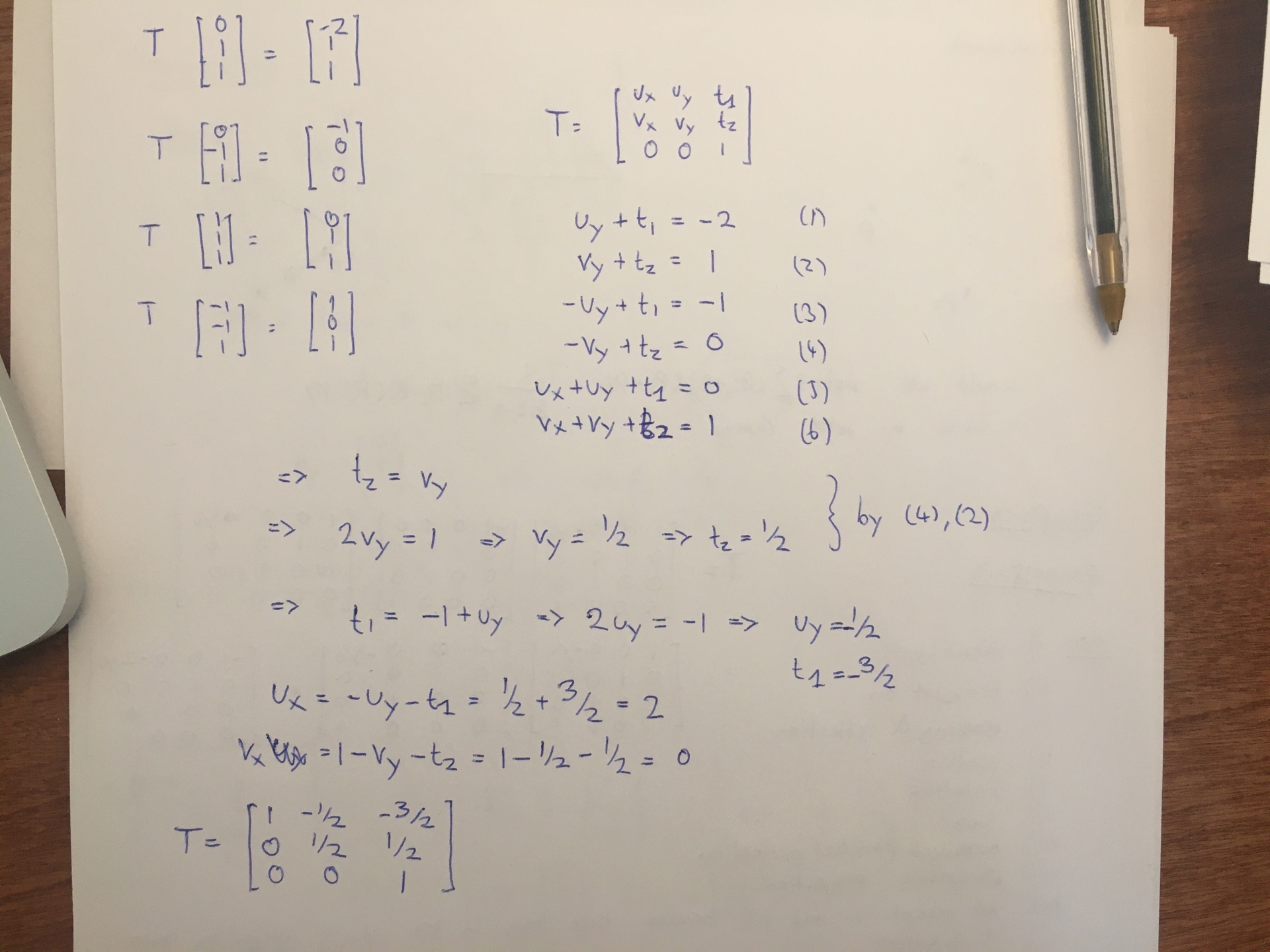
vii) Parallel (Orthographic) Projection

viii) This is a trick question. It looks like perspective projection - but not all points actually get mapped to a plane. Take (1,1,1,1)^T and (1,1,2,1)^T which the matrix maps to (1,1,-9) and (1/2,1/2,-4) respectively after dividing by the fourth component. So I would say just a translation in z followed by a scaling by 1/z (using the old z).

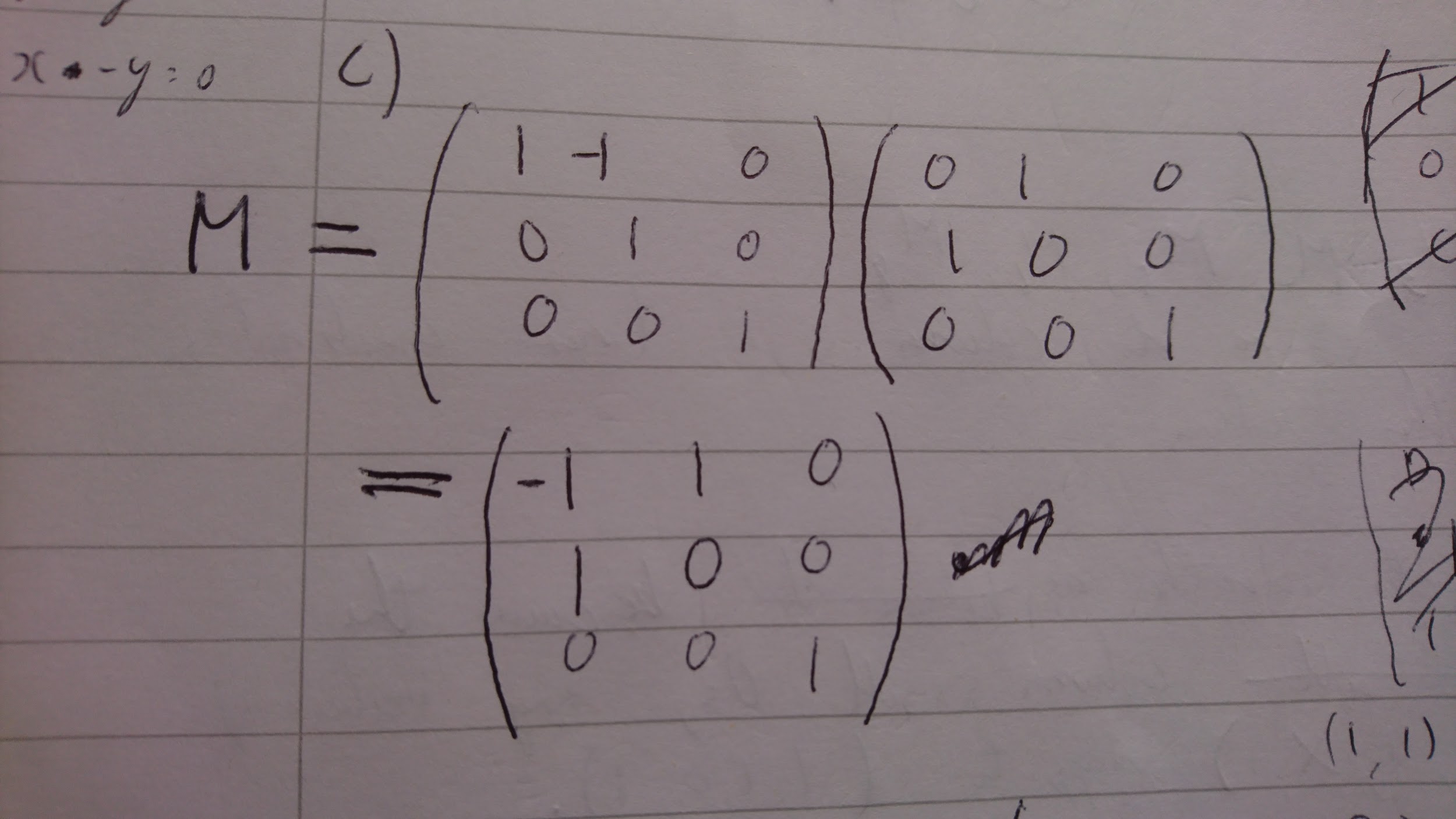
ix) i) - vi) are all invertible, because they are transformations from 3d space to 3d space, so no dimensions are lost. vii) is not invertible, because we go from 3d to 2d. viii) is invertible because it is full rank.

1b) Move plane x=-y\_0 to yz plane, scale x coordinate, move plane back to x = -y\_0



1c) 

So this question will have many solutions, as it is not clear which corner point should map to which. To verify a solution, just plug the vertices through the transformation.



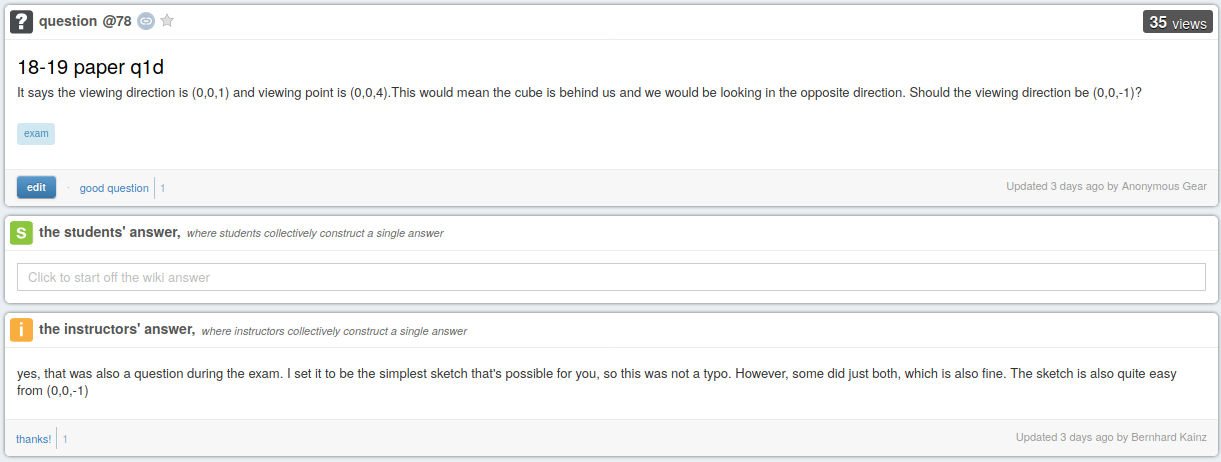
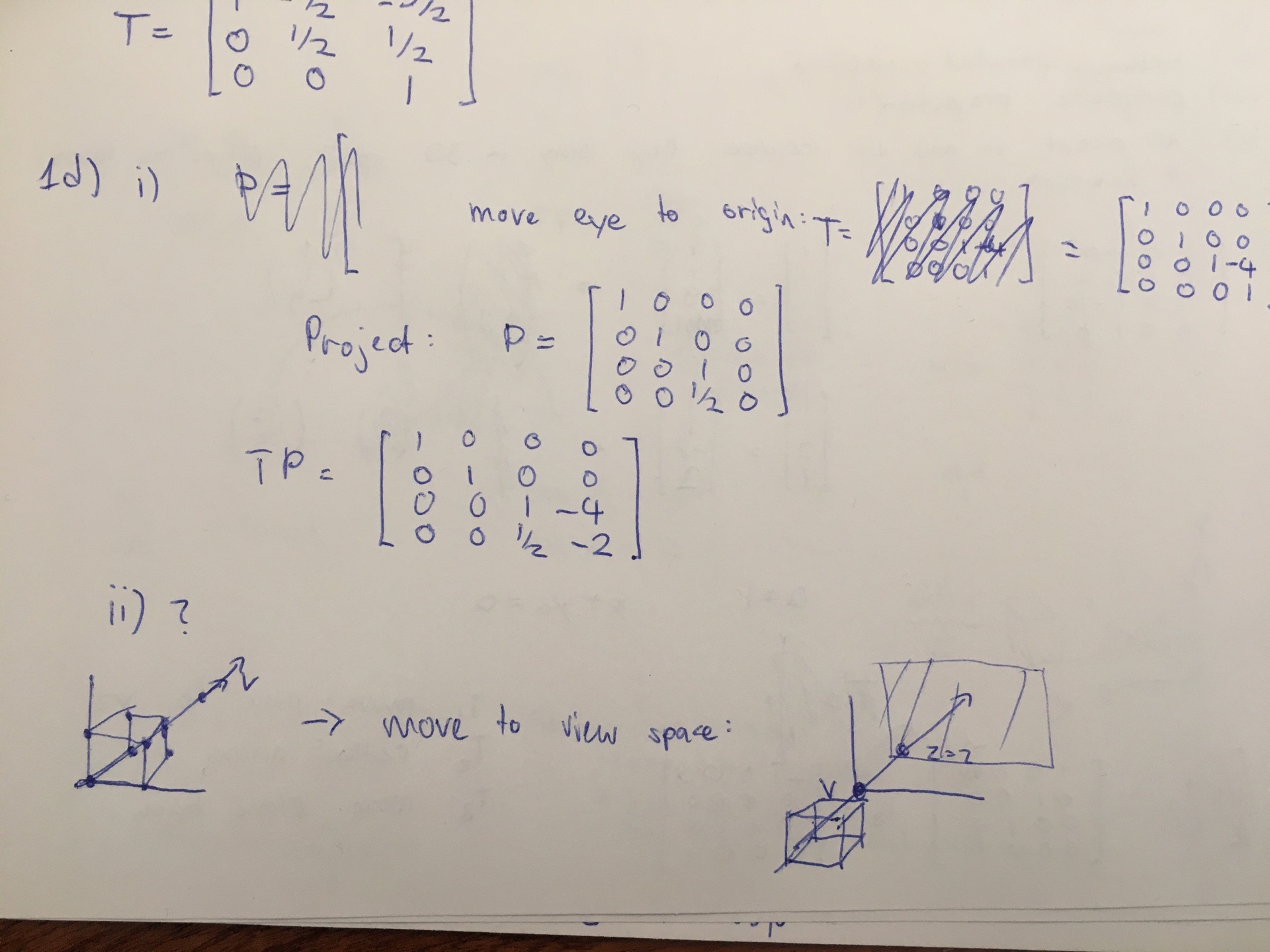
Also (skew as above + rotate anticlockwise 90deg)

[[-1,-1,0],

[1,0,0],

[0,0,1]]

1d) i) Viewpoint is at (0, 0, 4). So we need to transform to view space coordinates first by translating back to the origin. Then project.



ii) Isn’t this behind us?

Yes as below. This is why they say don’t take Graphics.

2a) i) d because the angle between the vector from the viewpoint reflected across the normal and the vector from the surface to the light is the smallest.

ii) a because the angle between the normal and the vector from the surface to the light is the smallest

iii)

Not sure if valid, solved with similar triangles (as angle must be same, so valid reflection?):



∴ brightest specular point is

Alternative using online solver:

Calcualte r and v in terms of x and solve (r\*v)/(|r||v|) = 1 as the angle should be 0

x = -2/5 = -0.4

iv) The point directly beneath the light source, (-4, 0).

#

2b) i) E = 2/8 A + 6/8 B

ii) F = 1/8 C + 7/8 D

iii) G = 4/7 E + 4/7 F

iv) Interpolate linearly between E and F to get a weightjed sum of all 4 colours

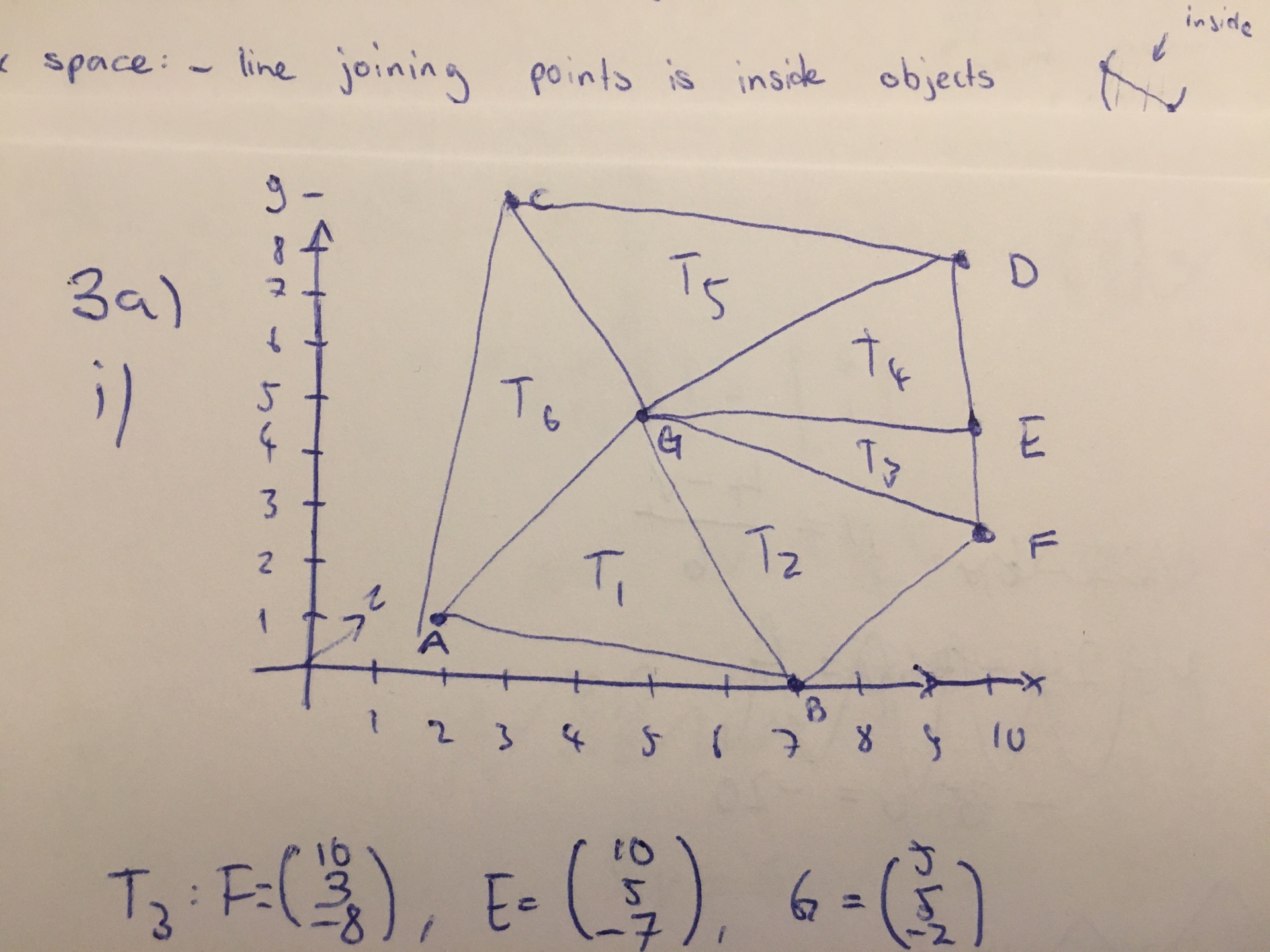
Colour at E = (219.5, 32, 0)

F = (31.25, 0, 250)

At G = (143.29, 18.29, 142.86). Should also multiply this by ⅞ because the two weights 4/7 and 4/7 sum up to 8/7 in total, increasing the intensity? -> G = (125.75, 16, 125)

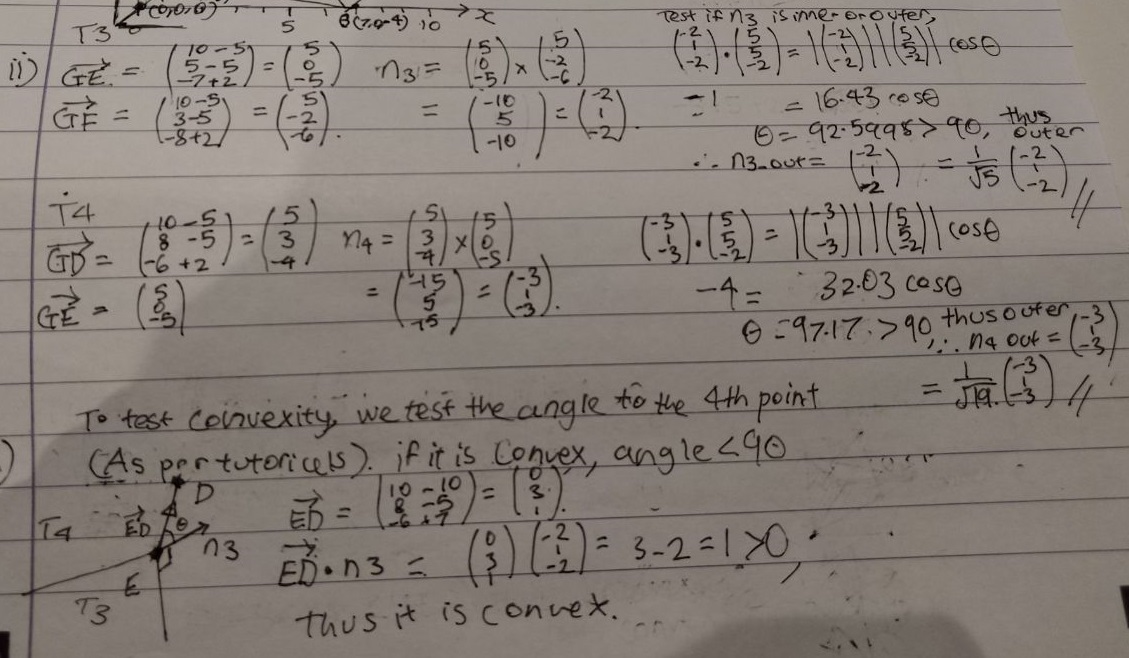
v) Vertex shader

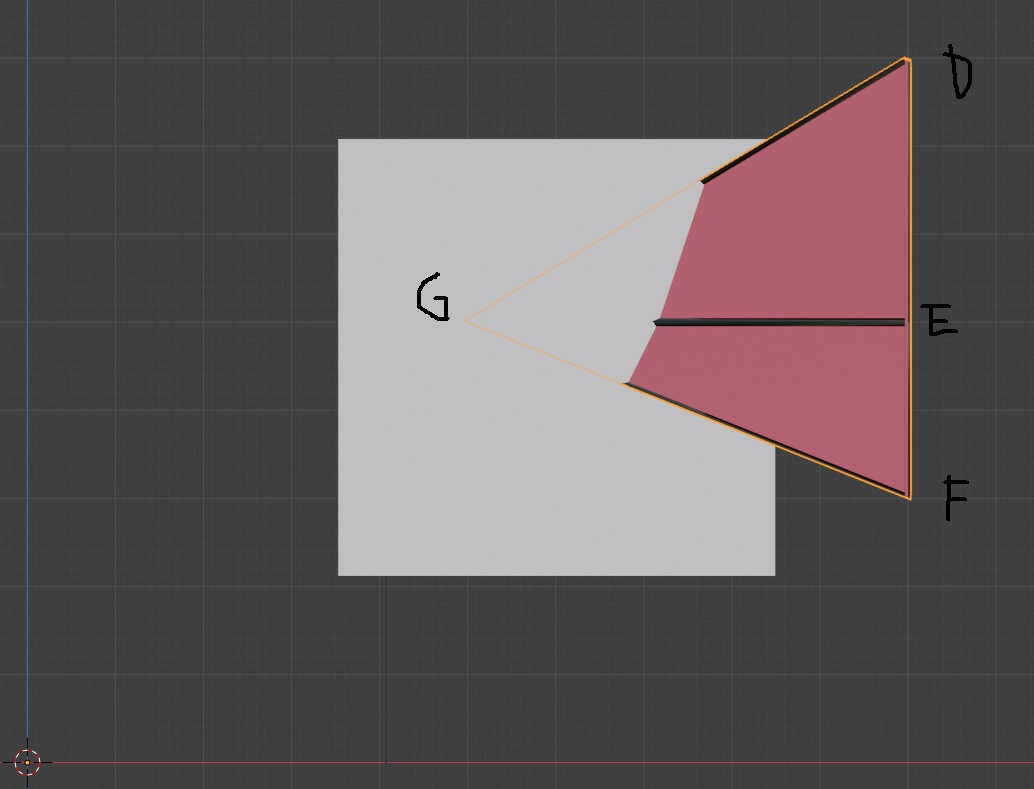
3a)i) Just draw everything without a z component



ii) Calculate the normal using cross product, then check whether the angle between the vector from the view origin to a point in the triangle (eg. G - O) is more or less than 0; <0 then its the outer, >0 then it’s the inner. See calculation.

iii) Check that its convex by checking the angle between a normal and the vector connecting two points.





2 iv) Convex, from the drawing: A and G are both at lower absolute z-values than B and C, so there is a higher “ridge” between A and G that drops backwards in the C and B direction

*2 v) Each step of -1 in the z direction changes x by 1.5 and y by 0.5, so it will intersect T1 or T2? Not sure how to solve this.*

We’re given the direction of the viewing ray. Can use this and do a line-plane intersection with each triangle. If then it intersects

3b) i)

* Shadows: For each pixel, send a ray towards the light, only illuminate if no intersection
* Reflection: In a recursive fashion, reflect rays across surfaces and add to the color of the current pixel.
* Transparency with refraction: Same as reflection, but instead of reflecting the ray, shift the direction slightly based on the material.

Ray tracing “fakes” global illumination with tricks, like shadows etc, but in radiosity, we actually treat each pixel (patch) as a light source and calculate true global illumination that way. And so ambient is now a function of incident light.

ii) Set up rays: dir = Pixel - Origin

For i = 0 to maxDepth:

Closest\_i = intersect(dir, all objects)

If (closest\_i.hit is None):

return

For each lightsource:

Shadow\_i = intersect(l - closest\_i.pos, all\_objects)

If (shadow\_i.hit):

Colour = ambient

Else:

Colour = ambient + specular + diffuse

dir = reflect(dir, closest\_i.normal)

Termination: max depth or nothing intersect

iii)

* Max reflection depth
* Nothing was intersected
* The last object doesn’t reflect light

4a) Supersampling: Render the whole image at a higher resolution. Then each pixel in the actual image maps to multiple in the super sampled picture - so take their average.

Convolution Filtering: Render the image normally. Then move a kernel over the image which takes the average of its pixels, and sets the center to be that average.

b) Supersampling:

Pros:

* Looks good

Cons:

* Slow
* Does not work for line drawings (ie. a very thin line will be just 1 pixel in the super sampled image, so if we then take the average of pixels around it the line is basically lost)

Convolution Filtering:

Pros:

* Fast

Cons:

* Lose accuracy: We’re blurring the image

c)

Frame buffer I

Initialise z-buffer A: same size as I, 1 everywhere

For each triangle T:

Rasterise T to produce fragments

For each fragment (x, y) with depth z and colour C:

If (z < A[x,y])

A[x,y] = z

I[x,y] = C

d) All opaque fragments need to be rendered first and then render the translucent ones. If the translucent object is behind an opaque object, it won’t be rendered because of the z-buffer value will be higher. But if it is rendered (i.e. the translucent object is in front), we can simply use the alpha value to blend the two colours together.

e)i) Jagged edges? No idea.

Jaggies I think

f) Use an unweighted convolution filter (3x3 grid) that takes the averages. Use background colour if the line doesn’t touch the pixel, or foreground colour if it does.

Then at (5,3): 5/9 (100, 100, 100)^T + 4/9 (128, 200, 100)^T = (112, 144, 100)

Then at (10,5): 6/9 (100, 100, 100)^T + 3/9 (128, 200, 100)^T = (109, 133, 100)