can you explain what these three functions do?

```
def get_rays(H, W, K, c2w):
    i, j = torch.meshgrid(torch.linspace(0, W-1, W),
torch.linspace(0, H-1, H))  # pytorch's meshgrid has
indexing='ij'
    i = i.t()
    j = j.t()
    dirs = torch.stack([(i-K[0][2])/K[0][0], -(j-K[1][2])/K[1]
[1], -torch.ones_like(i)], -1)
    # Rotate ray directions from camera frame to the world
frame
    rays_d = torch.sum(dirs[..., np.newaxis, :] * c2w[:3,:3],
-1)  # dot product, equals to: [c2w.dot(dir) for dir in dirs]
    # Translate camera frame's origin to the world frame. It
is the origin of all rays.
    rays_o = c2w[:3,-1].expand(rays_d.shape)
    return rays_o, rays_d


def get_rays_np(H, W, K, c2w):
    i, j = np.meshgrid(np.arange(W, dtype=np.float32),
np.arange(H, dtype=np.float32), indexing='xy')
    dirs = np.stack([(i-K[0][2])/K[0][0], -(j-K[1][2])/K[1][1],
-np.ones_like(i)], -1)
    # Rotate ray directions from camera frame to the world
frame
    rays_d = np.sum(dirs[..., np.newaxis, :] * c2w[:3,:3], -1)
# dot product, equals to: [c2w.dot(dir) for dir in dirs]
    # Translate camera frame's origin to the world frame. It
is the origin of all rays.
    rays_o = np.broadcast_to(c2w[:3,-1],
np.shape(rays_d))
    return rays_o, rays_d


def ndc_rays(H, W, focal, near, rays_o, rays_d):
```

```
    # Shift ray origins to near plane
    t = -(near + rays_o[...,2]) / rays_d[...,2]
    rays_o = rays_o + t[...,None] * rays_d

    # Projection
    o0 = -1./(W/(2.*focal)) * rays_o[...,0] / rays_o[...,2]
    o1 = -1./(H/(2.*focal)) * rays_o[...,1] / rays_o[...,2]
    o2 = 1. + 2. * near / rays_o[...,2]

    d0 = -1./(W/(2.*focal)) * (rays_d[...,0]/rays_d[...,2] -
rays_o[...,0]/rays_o[...,2])
    d1 = -1./(H/(2.*focal)) * (rays_d[...,1]/rays_d[...,2] -
rays_o[...,1]/rays_o[...,2])
    d2 = -2. * near / rays_o[...,2]

    rays_o = torch.stack([o0,o1,o2], -1)
    rays_d = torch.stack([d0,d1,d2], -1)

    return rays_o, rays_d
```

These three functions are standard utilities from **NeRF-style rendering pipelines**. Together, they turn camera parameters into **rays in 3D space**, and optionally convert those rays into **Normalized Device Coordinates (NDC)**.

I'll explain them step by step, from intuition to math.

---

1. `get_rays(H, W, K, c2w)` — PyTorch version

What it does:

For every pixel in an image, it computes:

- `rays_o` : the **origin** of the ray (camera position in world space)
- `rays_d` : the **direction** of the ray (where that pixel looks in 3D)

These rays are used for volume rendering (e.g., NeRF).

Step-by-step

1. Create a pixel grid

python

```python
i, j = torch.meshgrid(torch.linspace(0, W-1, W),
                      torch.linspace(0, H-1, H))
i = i.t()
j = j.t()
```

- `i` : x-coordinates of pixels
- `j` : y-coordinates of pixels
- Shape: `(H, W)`

Each `(i, j)` corresponds to a pixel.

Copy code

---

2. Convert pixels → camera-space directions

python

```python
dirs = torch.stack([
    (i - K[0][2]) / K[0][0],
    -(j - K[1][2]) / K[1][1],
    -torch.ones_like(i)
], -1)
```

This uses the **camera intrinsics matrix** `K` :

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

Copy code

- `(i - cx) / fx` , `(j - cy) / fy` : converts pixels to normalized camera coordinates
- `-1` in z: camera looks down the **negative z-axis** (NeRF convention)
- Result: a direction vector for each pixel **in camera space**

---

3. Rotate directions into world space

python

```
rays_d = torch.sum(dirs[..., None, :] * c2w[:3, :3], -1)
```

- `c2w[:3, :3]` is the **rotation matrix**
- Converts directions from camera coordinates → world coordinates

Copy code

---

4. Set ray origins

```python
rays_o = c2w[:3, -1].expand(rays_d.shape)
```

- `c2w[:3, -1]` is the **camera position in world space**
- Every ray starts at the same camera origin

Copy code

Output

- `rays_o` : `(H, W, 3)` ray origins
- `rays_d` : `(H, W, 3)` ray directions

---

2. `get_rays_np(H, W, K, c2w)` — NumPy version

This function does **the exact same thing** as `get_rays`, but:

- Uses **NumPy instead of PyTorch**
- Uses `np.meshgrid(..., indexing='xy')` so no transpose is needed

Everything else (math, output meaning) is identical.

---

3. `ndc_rays(H, W, focal, near, rays_o, rays_d)` — Convert rays to NDC

What it does:

Transforms rays from world space into **Normalized Device Coordinates (NDC)**.

This is mainly used in:

- Forward-facing NeRF scenes

- Stabilizing training
- Matching OpenGL-style projection

Step-by-step

1. Move ray origins to the near plane

python

```
t = -(near + rays_o[...,2]) / rays_d[...,2]
rays_o = rays_o + t[..., None] * rays_d
```

- Intersects each ray with the **near plane**
- Ensures all rays start at the same depth
- Helps with numerical stability

Copy code

---

2. Project origins into NDC space

python

```
o0 = -1./(W/(2.*focal)) * rays_o[...,0] / rays_o[...,2]
o1 = -1./(H/(2.*focal)) * rays_o[...,1] / rays_o[...,2]
o2 = 1. + 2. * near / rays_o[...,2]
```

This:

- Applies a **perspective projection**
- Maps coordinates into the range roughly `[-1, 1]`
- Similar to what a GPU does before rasterization

Copy code

---

3. Project directions into NDC space

python

```
d0 = -1./(W/(2.*focal)) * (rays_d[...,0]/rays_d[...,2] - rays_o[...,0]/ray
d1 = -1./(H/(2.*focal)) * (rays_d[...,1]/rays_d[...,2] - rays_o[...,1]/ray
```

```
d2 = -2. * near / rays_o[...,2]
```

- Converts ray **slopes** instead of absolute directions
- Ensures rays remain consistent after projection

Copy code

Final output

python

```python
rays_o = torch.stack([o0, o1, o2], -1)
rays_d = torch.stack([d0, d1, d2], -1)
```

Now:

- Rays live in **NDC space**
- Near plane is at `z = 1`
- Far plane approaches `z = -1`

Copy code

Big Picture Summary

Function	Purpose
`get_rays`	Generate world-space rays from camera pixels (PyTorch)
`get_rays_np`	Same as above, but NumPy
`ndc_rays`	Convert rays to normalized device coordinates

If you want, I can:

- Draw a diagram of camera → rays → NDC
- Explain why NeRF uses `-z`
- Walk through a **single pixel example numerically**

↳ * `(i - cx) / fx`, `(j - cy) / fy`: converts pixels to normalized camera coordinates * `-1` in z: camera looks down the **negative z-axis** (NeRF convention) * Result: a direction vector for each pixel **in camera space**

are camera space and normalized camera coordinates same thing?

Short answer: **no — they're related, but not the same thing.**
People often blur the terms, which is why this is confusing.

Let's clear it up cleanly.

---

1. Camera space (a.k.a. camera coordinates / view space)

Camera space is a **3D coordinate system**:

- Origin: **camera center**
- Axes: defined by the camera orientation
  (in NeRF: +x right, +y up, **−z forward**)

A point in camera space looks like:

$$(x_c, y_c, z_c)$$

Example:

```text
(0, 0, -3)   # 3 units in front of the camera
```

This is a **full 3D position**, with depth.

---

2. Normalized camera coordinates (a.k.a. normalized image plane coordinates)

Normalized camera coordinates live on a **2D plane**, not full 3D space.

They are obtained by:

$$x_n = \frac{x_c}{z_c}, \quad y_n = \frac{y_c}{z_c}$$

or equivalently from pixels:

$$x_n = \frac{i - c_x}{f_x}, \quad y_n = \frac{j - c_y}{f_y}$$

Key properties:

- Unitless
- Independent of focal length
- Represent a **ray direction**, not a point in space

They answer:

> "In what direction does this pixel look?"

---

3. What your code is actually doing

This line:

python

```python
(i - cx) / fx, (j - cy) / fy
```

produces **normalized camera coordinates**.

Copy code

Then this:

python

```python
dirs = [..., ..., -1]
```

lifts them into camera space by assigning a depth:

Copy code

text

```text
(x_n, y_n, -1)
```

Copy code

So the vector:

text

```
[(i-cx)/fx, (j-cy)/fy, -1]
```

is:

Copy code

✅ **a direction vector in camera space**
❌ **not just normalized camera coordinates**

---

4. Why this distinction matters

Concept	Dimensionality	Has depth?	Meaning
Camera space	3D	✅	Actual coordinates relative to camera
Normalized camera coords	2D	❌	Direction on image plane
`(x_n, y_n, -1)`	3D	⚠️ fixed	Direction vector in camera space

Normalized camera coordinates **become** camera-space rays **only after** you choose a depth (here `z = -1`).

---

5. Why people mix the terms (especially in NeRF code)

In ray generation:

- We **don't care about the exact depth**
- Any point along the same ray is equivalent

So people casually say:

> "camera-space direction"

even though it's constructed from **normalized image plane coordinates**.

That's correct *in context*, but imprecise in theory.

One-sentence takeaway

Normalized camera coordinates are 2D ray directions; camera space is 3D.
Your code converts normalized camera coordinates into camera-space ray directions by setting `z = -1` **.**

If you want, I can also explain how this differs from **NDC** or **OpenGL camera space**, which adds another layer of confusion.

↳ xn\=fxi−cx,yn\=fyj−cy

what are i, Cx and Cy?

Good question — these symbols come straight from the **camera intrinsics model**.

I'll explain each one clearly and relate them back to the code and equation you wrote.

The equation you're looking at

$$x_n = \frac{i - c_x}{f_x}, \quad y_n = \frac{j - c_y}{f_y}$$

This converts **pixel coordinates** → **normalized camera coordinates**.

---

1. What is `i` (and `j`)?

- `i` : **horizontal pixel coordinate** (x-axis)
- `j` : **vertical pixel coordinate** (y-axis)

They come from the image grid.

Example (for a 640×480 image):

```text
top-left pixel:     i = 0,   j = 0
center pixel:       i ≈ 320, j ≈ 240
bottom-right pixel: i = 639, j = 479
```

In your code:

```python
i, j = meshgrid(...)
```

`i` and `j` store the pixel location of **each ray**.

Copy code

---

2. What are `Cx` and `Cy` (`c_x`, `c_y`)?

- `c_x` : **x-coordinate of the principal point**
- `c_y` : **y-coordinate of the principal point**

The **principal point** is where the camera's optical axis hits the image plane.

In an ideal camera:

```text
c_x = W / 2
c_y = H / 2
```

But in real cameras, it might be slightly off-center.

Copy code

They come from the **intrinsic matrix**:

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

So:

python

```
c_x = K[0, 2]
c_y = K[1, 2]
```

Copy code

3. Why subtract `Cx` and `Cy` ?

Pixel coordinates are measured from the **top-left corner** of the image.

But camera geometry is defined relative to the **optical center**.

So:

text

```
i - c_x   → how far left/right from the center
j - c_y   → how far up/down from the center
```

This recenters the coordinate system.

Copy code

---

4. Why divide by `f_x` and `f_y` ?

This:

- Removes the effect of focal length
- Converts pixel distances into **unitless direction slopes**

After division:

- `x_n` , `y_n` describe **ray direction**, not pixel distance
- Same scene + different focal length → same normalized coordinates

---

5. Putting it all together (intuition)

For a single pixel:

text

```
Pixel (i, j)
   ↓ subtract (c_x, c_y)
```

```
Offset from image center
   ↓ divide by (f_x, f_y)
Normalized direction
```

Then your code does:

python

```python
(x_n, y_n, -1)
```

Copy code

Which means:

> "This pixel shoots a ray forward from the camera through this direction."

One-line summary

- `i` , `j` : pixel coordinates
- `c_x` , `c_y` : image center (principal point)
- `(i - c_x) / f_x` , `(j - c_y) / f_y` : normalized camera coordinates (ray direction)

If you want, I can also explain **why the y term is negated** in your code or how this relates to OpenCV vs OpenGL conventions.

↳ I can also explain how this differs from **NDC** or **OpenGL camera space**, which adds another layer of confusion

please do.

Great — this *is* the part that trips most people up. I'll explain it in three layers and tie them directly to your code:

1. **Why the y-coordinate is negated**
2. **OpenCV vs OpenGL camera conventions**
3. **How NDC fits into all of this**

---

1. Why is the y term negated?

In your code:

python

```python
dirs = torch.stack([
    (i - cx) / fx,
    -(j - cy) / fy,
    -1
], -1)
```

The key line is:

python

```python
-(j - cy) / fy
```

Copy code

The root cause: image coordinates vs 3D coordinates

Copy code

Image space (pixels):

- Origin: **top-left**
- x increases → right
- y increases → **down**

Camera / 3D space:

- Origin: camera center
- x increases → right
- y increases → **up**

So the y-axis is **flipped**.

What the negation does

text

```text
Pixel above center:   j < cy  →  (j - cy) < 0
After negation:       y > 0   →  ray points upward
```

Without the minus sign:

- The image would be vertically flipped
- Rays would go in the wrong direction

Copy code

✅ **The negation fixes the coordinate system mismatch**

---

2. OpenCV vs OpenGL camera conventions

This is where terminology gets messy.

OpenCV convention (used by NeRF)

- Camera looks down **−Z**
- Right-handed system
- Image y-axis points down

```text
x → right
y → up
z → backward
```

That’s why your code uses:

```python
z = -1
```

This means:

Copy code

> “Shoot the ray forward”

OpenGL convention (classic graphics pipeline)

- Camera looks down **−Z** (same)
- BUT after projection, coordinates are transformed into **NDC**

OpenGL pipeline:

text

```
camera space → clip space → NDC → screen
```

Copy code

OpenCV usually **stops at camera space**.

Quick comparison

Concept	OpenCV / NeRF	OpenGL
Forward direction	–Z	–Z
Image origin	Top-left	Bottom-left
Typical use	Vision, reconstruction	Rendering
Uses NDC?	❌ (usually)	✅

---

3. How NDC fits in (your `ndc_rays` function)

NDC = **Normalized Device Coordinates**

After projection:

text

```
x, y, z ∈ [−1, 1]
```

Copy code

Why NeRF uses NDC

For **forward-facing scenes**:

- All rays originate from roughly the same direction
- NDC:
  - Normalizes depth
  - Keeps rays well-behaved
  - Improves training stability

What `ndc_rays` is doing conceptually

1. **Intersect rays with the near plane**

```python
t = -(near + rays_o[...,2]) / rays_d[...,2]
```

2. **Apply a perspective projection**

```python
x_ndc = x / z
y_ndc = y / z
```

3. **Rescale to [-1, 1]**

```python
W / (2 * focal)
H / (2 * focal)
```

After this:

- Near plane → `z = 1`
- Far plane → `z → -1`

This mirrors OpenGL's NDC (with NeRF-specific tweaks).

---

4. Putting it all together (mental model)

Ray generation pipeline in your code

```text
pixels (i, j)
   ↓ subtract (cx, cy)
image-centered pixels
   ↓ flip y
camera-aligned coordinates
   ↓ divide by focal
normalized camera coords
   ↓ set z = -1
```

```
camera-space ray direction
   ↓ rotate + translate (c2w)
world-space rays
   ↓ (optional) ndc_rays
NDC rays
```

Final takeaway (one paragraph)

- The **y negation** fixes the mismatch between image coordinates (downward y) and 3D camera coordinates (upward y).
- NeRF follows an **OpenCV-style camera model** where the camera looks down **−Z**.
- **NDC** is a *later* projection step that normalizes rays for rendering and training, similar to OpenGL's pipeline but adapted for ray tracing.

If you want next, I can:

- Walk through a **single pixel numerically**
- Draw a coordinate diagram in ASCII
- Explain **why NeRF doesn't always use NDC**

Copy code

↪ camera space → clip space → NDC → screen

can you explain the clip space and NDC? also explain how the conversion of each step does.

Below is a **clean, end-to-end explanation of the graphics pipeline** your developer note mentions:

> **camera space → clip space → NDC → screen**

I'll explain:

1. What **clip space** is
2. What **NDC** is
3. What **each conversion step actually does mathematically and conceptually**

4. How this relates to your NeRF `ndc_rays` function

---

1. Camera space (starting point)

You already mostly understand this, but it anchors everything else.

Camera (view) space

- Origin: camera center
- Coordinates are **3D**
- Camera looks down **−Z** (OpenGL & NeRF)

A point looks like:

```scss
(x_c, y_c, z_c, 1)
```

Copy code

Rays are defined here before any projection.

---

2. Clip space — *before perspective division*

What is clip space?

Clip space is the result of applying the projection matrix to camera-space coordinates.

Mathematically:

```ini
clip_coord = ProjectionMatrix × camera_coord
```

Copy code

This produces **homogeneous coordinates**:

```scss
(x_clip, y_clip, z_clip, w_clip)
```

Copy code

Why does clip space exist?

Clip space allows the GPU to:

- Perform **frustum clipping**
- Keep perspective information in `w`
- Delay division until after clipping

This is why it's called *clip* space.

What the projection matrix does

For perspective projection, it:

- Encodes focal length (field of view)
- Encodes near/far planes
- Moves depth info into `w`

Conceptually:

```nginx
x_clip ≈ x_c
y_clip ≈ y_c
z_clip ≈ depth mapping
w_clip ≈ −z_c
```

⚠️ **Nothing is normalized yet**

---

3. Clipping (why "clip" space matters)

Before dividing by `w`, the GPU clips geometry:

A point is kept only if:

```diff
−w ≤ x_clip ≤ w
−w ≤ y_clip ≤ w
−w ≤ z_clip ≤ w
```

Anything outside is discarded.

This prevents:

Copy code

- Division by zero
- Objects behind the camera
- Geometry outside the view frustum

---

4. NDC — Normalized Device Coordinates

What is NDC?

After clipping, the GPU performs **perspective division**:

```ini
x_ndc = x_clip / w_clip
y_ndc = y_clip / w_clip
z_ndc = z_clip / w_clip
```

Now:

```css
x_ndc, y_ndc, z_ndc ∈ [-1, 1]
```

This is **NDC**.

Copy code

Why divide by `w`?

This is the step that creates **perspective**.

Effects:

- Far objects appear smaller
- Parallel lines converge
- Depth becomes nonlinear

This is the mathematical core of perspective projection.

Interpretation of NDC

Axis	Meaning
x = −1	left edge of screen
x = +1	right edge
y = −1	bottom
y = +1	top
z = −1	far plane
z = +1	near plane (OpenGL)

Everything visible fits inside a **cube**.

---

5. Screen space (final step)

NDC → screen coordinates:

```ini
x_screen = (x_ndc + 1) * width  / 2
y_screen = (1 - y_ndc) * height / 2
```

This:

- Maps `[-1, 1]` → pixel coordinates
- Flips y for image coordinates
- Produces final raster positions

---

6. Full pipeline summary (compact)

scss

```
Camera space (3D)
   ↓ Projection matrix
Clip space (4D, homogeneous)
   ↓ Clipping
Still clip space
   ↓ Perspective divide (÷ w)
NDC (3D, normalized)
   ↓ Viewport transform
Screen space (pixels)
```

---

7. How this relates to NeRF's `ndc_rays`

NeRF does **not rasterize**, so it skips:

- Clip space
- Homogeneous coordinates
- GPU clipping

Instead, it **manually recreates the math**.

What NeRF does directly

In `ndc_rays`:

```python
x_ndc = x / z
y_ndc = y / z
```

That is **exactly the perspective divide**, but done analytically.

Then it rescales by:

```python
W / (2 * focal)
H / (2 * focal)
```

Which corresponds to:

- Field of view normalization

- Projection matrix scaling terms

So NeRF:

- Skips clip space
- Skips `w`
- Goes **camera space → NDC directly**

Because rays never need clipping.

---

8. Key conceptual distinction (important)

Space	Purpose
Camera space	Geometry relative to camera
Clip space	Enable clipping using `w`
NDC	Unified, normalized viewing cube
Screen	Actual pixels

Clip space is **a GPU engineering space**, not a geometric one.

Final takeaway (one sentence)

Clip space exists to make clipping and perspective mathematically safe using homogeneous coordinates; NDC is what you get after dividing by `w`, where everything visible lies in a normalized `[-1,1]` cube.

If you want next, I can:

- Derive the **projection matrix numerically**
- Show **why** `w = -z`
- Compare **OpenGL vs DirectX NDC differences**
- Tie this back to **ray marching vs rasterization**

↳ \-w ≤ x_clip ≤ w \-w ≤ y_clip ≤ w \-w ≤ z_clip ≤ w

is w the w_clip?

Yes. **That `w` is specifically `w_clip`.**

When you see inequalities like:

```
-w ≤ x_clip ≤ w
-w ≤ y_clip ≤ w
-w ≤ z_clip ≤ w
```

the `w` there is **the same `w` component of the clip-space coordinate**:

$$(x_{\text{clip}},\ y_{\text{clip}},\ z_{\text{clip}},\ \mathbf{w}_{\text{clip}})$$

Copy code

Why this matters

After applying the projection matrix, every point is in **clip space** and represented in **homogeneous coordinates**:

$$\begin{bmatrix} x_{\text{clip}} \\ y_{\text{clip}} \\ z_{\text{clip}} \\ \mathbf{w}_{\text{clip}} \end{bmatrix}$$

Before doing the perspective divide, the GPU performs **clipping** using `w_clip`.

Only points that satisfy:

$$-w_{\text{clip}} \le x_{\text{clip}} \le w_{\text{clip}}$$

$$-w_{\text{clip}} \le y_{\text{clip}} \le w_{\text{clip}}$$

$$-w_{\text{clip}} \le z_{\text{clip}} \le w_{\text{clip}}$$

are kept.

Why clip against `±w_clip` ?

Because **perspective hasn't happened yet**.

Later, the GPU will divide:

$$x_{\text{ndc}} = \frac{x_{\text{clip}}}{w_{\text{clip}}}$$

If clipping were done *after* division, you'd risk:

- division by zero
- geometry behind the camera
- broken interpolation

Clipping against `±w_clip` guarantees that **after division**:

$$x_{\text{ndc}}, y_{\text{ndc}}, z_{\text{ndc}} \in [-1, 1]$$

What is `w_clip` conceptually?

In a standard OpenGL-style projection:

$$w_{\text{clip}} = -z_{\text{camera}}$$

So:

- Points **in front of the camera** → `w_clip > 0`
- Points **behind the camera** → `w_clip < 0` (discarded)

This is why the camera looks down **–Z**.

Pipeline recap (minimal)

```scss
camera space (x, y, z, 1)
   ↓ projection matrix
clip space (x_clip, y_clip, z_clip, w_clip)
   ↓ clip against ±w_clip
```

```
   ↓ divide by w_clip
NDC (x_ndc, y_ndc, z_ndc)
```

Copy code

One-line takeaway

Yes — the `w` in those inequalities is `w_clip`, the homogeneous coordinate produced by the projection matrix, and it is what makes safe clipping and perspective division possible.

If you want, I can also show:

- a concrete numeric example
- why DirectX clips `z` differently
- how this relates to NeRF skipping clip space

↳ homogeneous coordinates

what does homogeneous coordinates mean?

Homogeneous coordinates are a mathematical trick that lets us represent **translation, projection, and perspective** using **matrix multiplication**.

They’re the reason the graphics pipeline (and camera geometry) works so cleanly.

I’ll explain this in a structured way and tie it directly to clip space and `w`.

---

1. The core problem homogeneous coordinates solve

In normal 3D coordinates:

css

```css
(x, y, z)
```

you **cannot** represent translation with a matrix multiply alone.

Copy code

Example:

- Rotation ✔️ (matrix)
- Scaling ✔️ (matrix)
- **Translation** ❌ (needs addition)

Graphics pipelines want *everything* to be:

```cpp
matrix × vector
```

Homogeneous coordinates make that possible.

Copy code

---

2. The idea: add one extra dimension

We embed 3D points into **4D space**:

```css
(x, y, z)  →  (x, y, z, w)
```

This is a **homogeneous coordinate**.

Copy code

For ordinary points:

```ini
w = 1
```

Copy code

3. Why this works (translation example)

A 3D translation by `(tx, ty, tz)` becomes:

$$\begin{bmatrix}1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1\end{bmatrix}\begin{bmatrix}x \\ y \\ z \\ 1\end{bmatrix} = \begin{bmatrix}x + t_x \\ y + t_y \\ z + t_z \\ 1\end{bmatrix}$$

Translation is now **just a matrix multiply**.

That's the key benefit.

---

4. Homogeneous coordinates are not unique

The coordinates:

```css
(x, y, z, w)
```

and

Copy code

```scss
(kx, ky, kz, kw)
```

represent the **same point** (for any nonzero `k`).

Copy code

To get the actual 3D point, you **divide by** `w`:

$$(x', y', z') = \left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}\right)$$

This step is called **dehomogenization**.

---

5. Why perspective works (the magic part)

Perspective projection sets:

```ini
w = −z_camera
```

So after projection:

Copy code

$$x_{\text{ndc}} = \frac{x_{\text{clip}}}{w_{\text{clip}}} = \frac{x}{-z}$$

This:

- Makes far objects smaller
- Creates depth-dependent scaling
- Produces realistic perspective

All from a **single divide by** `w`.

---

6. Points vs directions (important distinction)

Type	Homogeneous form	Why
Point	`(x, y, z, 1)`	Affected by translation
Direction / vector	`(x, y, z, 0)`	Not affected by translation

Example:

- Ray origin → `w = 1`
- Ray direction → `w = 0`

This is exactly what your NeRF code relies on conceptually.

---

7. Why clip space uses homogeneous coordinates

Clip space coordinates are:

scss

```
(x_clip, y_clip, z_clip, w_clip)
```

Copy code

Before dividing by `w_clip`, the GPU:

- Clips against `±w_clip`
- Ensures perspective divide is safe

After division:

scss

```
(x_ndc, y_ndc, z_ndc)
```

Copy code

So homogeneous coordinates:

- Enable clipping
- Enable perspective
- Enable a unified matrix pipeline

---

8. Intuition (one-sentence mental model)

Homogeneous coordinates store “scale” in `w`, and dividing by `w` turns that scale into perspective.

---

9. Minimal takeaway

- Homogeneous coordinates add an extra dimension
- They make translation and projection linear
- `w` controls perspective and clipping
- Dividing by `w` gives back real 3D coordinates

If you want next, I can:

- Show a numeric camera → clip → NDC example
- Connect this to projective geometry
- Explain why points at infinity have `w = 0`

↳ If you want next, I can: * Derive the **projection matrix numerically** * Show **why `w = -z`** * Compare **OpenGL vs DirectX NDC differences** * Tie this back to **ray marching vs rasterization**

can you do all four items?

Absolutely — let’s do **all four**, cleanly and in order.

I’ll keep the math concrete but not overwhelming, and I’ll constantly connect it back to

what it's for.

1️⃣ Deriving the perspective projection matrix (numerically)

We want a matrix that maps **camera space → clip space**.

Camera space assumptions (OpenGL / NeRF style)

- Camera at origin
- Looking down **−Z**
- Near plane at `n`
- Far plane at `f`
- Vertical field of view `fov`
- Aspect ratio `a = width / height`

Step 1: Perspective intuition

For a point in camera space:

```scss
(x_c, y_c, z_c)
```

Perspective says:

```nginx
x_proj ∝ x_c / -z_c
y_proj ∝ y_c / -z_c
```

So we want:

- division by `z`
- but **only after clipping**

That's why we use homogeneous coordinates.

Step 2: Projection matrix (OpenGL)

The standard OpenGL perspective projection matrix is:

$$P = \begin{bmatrix} \frac{1}{a\tan(\theta/2)} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(\theta/2)} & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Apply it to:

$$(x_c, y_c, z_c, 1)$$

Result:

```ini
x_clip = x_c / (a tan(fov/2))
y_clip = y_c / tan(fov/2)
z_clip = A z_c + B
w_clip = -z_c
```

✅ **Perspective divide happens later via** `w_clip`

2️⃣ Why `w = -z` ?

Copy code

This is a *huge* conceptual point.

What `w` is doing

After projection:

```scss
(x_clip, y_clip, z_clip, w_clip)
```

Then:

Copy code

```ini
```

```
x_ndc = x_clip / w_clip
```

If:

```ini
w_clip = -z_camera
```

Then:

```ini
x_ndc = x / -z
```

That's **exactly perspective projection**.

Why the minus sign?

Because:

- Camera looks down **−Z**
- Objects in front have `z < 0`
- We want `w > 0` for visible points

So:

```ini
w = -z
```

ensures:

- visible points → `w > 0`
- behind camera → `w < 0` → clipped

Deep intuition

> `w` **stores depth so that dividing by it shrinks distant objects.**

No division → no perspective

Division by `z` → perspective

3️⃣ OpenGL vs DirectX NDC differences

This causes *a lot* of confusion.

OpenGL NDC (classic)

After perspective divide:

```css
x, y ∈ [-1, 1]
z ∈ [-1, 1]
```

- Near plane → `z = +1`
- Far plane → `z = -1`
- Right-handed
- Camera looks down –Z

DirectX / Vulkan NDC

```css
x, y ∈ [-1, 1]
z ∈ [0, 1]
```

- Near plane → `z = 0`
- Far plane → `z = 1`
- Left-handed by default (configurable)
- Better depth buffer precision

Why the difference?

Historical + hardware reasons.

DirectX chose `[0, 1]` for depth because:

- Depth buffers are unsigned
- Precision is better near the camera

Key takeaway

Feature	OpenGL	DirectX
z range	[-1, 1]	[0, 1]
Near depth	+1	0
Far depth	−1	1
Used by NeRF	✅	❌

NeRF's `ndc_rays` matches **OpenGL-style math**.

4️⃣ Ray marching vs rasterization (why NeRF skips clip space)

This ties *everything* together.

Rasterization pipeline (graphics)

```css
triangles
 → camera space
 → clip space
 → NDC
 → screen
 → pixels
```

Needs:

- Clip space

- Homogeneous coordinates
- Triangle clipping
- Interpolation

Copy code

Ray marching (NeRF)

nginx

```
pixels
 → rays
 → sample points along ray
 → evaluate volume
```

Key differences:

- No triangles
- No clipping
- No screen projection

Copy code

So NeRF:

- **does not need clip space**
- **does not need homogeneous coords**
- **does not rasterize**

Why NeRF still uses NDC

For forward-facing scenes:

- Rays are nearly parallel
- Depth values vary wildly

NDC:

- Normalizes depth
- Makes ray sampling stable
- Keeps values numerically well-behaved

That's why NeRF does:

nginx

```
camera space → NDC
```

directly, skipping clip space entirely.

Copy code

🔁 Final mental model (important)

vbnet

```
Rasterization:
camera → clip → NDC → screen

NeRF:
camera → rays → (optional) NDC → ray marching
```

Clip space is **GPU plumbing**.
NDC is **geometry normalization**.
NeRF only needs the second.

Copy code

✅ Final one-paragraph takeaway

- The projection matrix encodes perspective by putting depth into `w`
- `w = -z` makes perspective division work and enforces visibility
- OpenGL and DirectX differ mainly in **z range**
- NeRF skips clip space entirely because rays don't need clipping, but still uses NDC for numerical stability

If you want next, I can:

- Walk one point **numerically through every space**
- Show how **depth nonlinearity** comes from `1/z`
- Explain why **points at infinity have** `w = 0`

↳ Perspective says: nginx Copy code `x_proj ∝ x_c / -z_c y_proj ∝ y_c / -z_c` why?

Ah! You're asking **why perspective projection divides by `z_c` (or `-z_c`)**. Let's break this down carefully.

1️⃣ The intuitive reason: "things farther away look smaller"

Perspective projection is literally trying to model **how the human eye or a camera sees the world**:

- If an object is **twice as far away**, it should appear **half as big**.
- That means the projected coordinate on the image plane should shrink proportionally to distance.

Imagine a simple pinhole camera

- Camera at origin
- Image plane at `z = -f` (negative because camera looks down –Z)
- Point in space: `(x_c, y_c, z_c)`

Draw a line from the point to the camera center.
Where it intersects the image plane?

- By similar triangles:

$$\frac{x_{\text{proj}}}{-z_{\text{c}}} = \frac{x_c}{z_{\text{image plane}}}$$

- If we set image plane at z = –1 (NeRF convention):

$$x_{\text{proj}} = \frac{x_c}{-z_c}, \quad y_{\text{proj}} = \frac{y_c}{-z_c}$$

✅ That's exactly the formula you saw.

2️⃣ Step-by-step with triangles

markdown

```
       *
      /|
     / |
    /  | y_c
   /   |
  C----+
       x_c
```

- `C` = camera center
- `*` = 3D point
- Image plane in front of camera

By **similar triangles**:

$$\frac{\text{coordinate on image plane}}{\text{image plane depth}} = \frac{\text{world coordinate}}{\text{depth of point}}$$

Copy code

- Rearranging gives:

$$x_{\text{image}} = x_c \frac{z_{\text{plane}}}{z_c}$$

- In NeRF/OpenGL, `z_plane = -1` →

$$x_{\text{proj}} = x_c / - z_c$$

3️⃣ The minus sign

- Camera looks **down −Z**
- Points in front of camera: `z_c < 0`
- Dividing by `-z_c` makes the projected x, y **positive in the right direction**

4️⃣ Intuition in one sentence

Perspective projection divides by `z` to make **distant objects appear smaller**, exactly like how real cameras and human eyes work.

If you want, I can **draw a simple ASCII diagram showing the similar triangles** so you can "see" why it's `/ -z_c`. It really makes this click.

Do you want me to do that?

how can I save the conversation of this chat?

You have a few ways to save this conversation, depending on what you want to do with it:

1️⃣ Copy & Paste (quickest)

- Simply **select all the text** in the chat window.
- **Copy** and **paste** it into:
  - A Word document
  - Google Docs
  - Notepad or any text editor
- You can also format it if you want headings and code blocks.

2️⃣ Use the "Export" or "Download" feature (if available)

- Some chat platforms (including OpenAI web chat) let you **export the conversation as a `.txt` or `.pdf`**.
- Look for buttons like **"Export"**, **"Download"**, or **"Save chat"** in the chat window.

3️⃣ Screenshots

- For short sections or diagrams, you can **take screenshots**.
- On Windows: `Win + Shift + S` → Snip tool
- On Mac: `Cmd + Shift + 4` → Select area
- On phones: usual screenshot gesture

4️⃣ Save in a coding-friendly format

If you want to keep code blocks intact:

- Paste the conversation into **Markdown** or `.md` file
- Use triple backticks for code:

markdown

```
```python
# your code here
```

yaml

```
---

💡 **Tip:** If you plan to revisit later for learning, saving as Markdown

---

If you want, I can make a **single nicely formatted version of this entire

Do you want me to do that?
```

↳ * Show how **depth nonlinearity** comes from `1/z` * Explain why **points at infinity have `w = 0`**

can you do these two?

Absolutely! Let's tackle these **two concepts** carefully and step by step.

Copy code

1️⃣ Depth nonlinearity from `1/z`
```

Step 1: Recall perspective projection

For a point `(x, y, z)` in camera space, the **perspective divide** gives:

$$x_{\text{ndc}} = \frac{x}{-z}, \quad y_{\text{ndc}} = \frac{y}{-z}$$

Similarly, the depth `z` is transformed (in OpenGL-style NDC) like:

$$z_{\text{ndc}} = A + \frac{B}{z}$$

Where `A` and `B` depend on the **near** and **far plane**.

Step 2: Why depth is nonlinear

Look at the mapping from **camera depth** `z_c` to **NDC depth** `z_ndc`:

```nginx
z_ndc ∝ 1 / z_c
```

Copy code

- Near points (`z_c ≈ near`) → NDC changes **rapidly**
- Far points (`z_c ≈ far`) → NDC changes **slowly**

✅ This is **depth nonlinearity**.

- Far objects get **compressed** in NDC.
- Near objects get **stretched**.

Step 3: Intuition

- Depth buffers in graphics are more precise near the camera — that's why games often have "z-fighting" far away.
- The **1/z mapping** comes directly from the perspective divide: distant points divide by a larger z → smaller effect on projected depth.

Step 4: Connection to NeRF

NeRF's `ndc_rays` does something similar:

python

```python
o2 = 1. + 2. * near / rays_o[...,2]
d2 = -2. * near / rays_o[...,2]
```

- You can see the `/ rays_o[...,2]` term — this is the **1/z depth nonlinearity**.
- It ensures forward-facing rays are well-distributed in the NDC cube.

Copy code

2️⃣ Points at infinity have `w = 0`

Step 1: Homogeneous coordinates refresher

In homogeneous coordinates:

css

```css
(x, y, z, w)
```

- 3D point → `w = 1`
- Direction (vector) → `w = 0`

Copy code

Step 2: Why w = 0 represents infinity

Dehomogenization:

$$(x_{\text{real}}, y_{\text{real}}, z_{\text{real}}) = \frac{(x, y, z)}{w}$$

- If `w = 1` → normal point
- If `w = 0` → division by zero → **point at infinity**

Intuition:

- Rays that "go forever" have **no finite endpoint**, so `w = 0` encodes that the point is infinitely far along a direction.

Step 3: Connection to rays and directions

- In NeRF:
  - Ray origin → `(x, y, z, 1)` (finite)

- Ray direction → `(dx, dy, dz, 0)` (infinite along that direction)
- Homogeneous coordinates allow **one matrix multiply** to move both origin and direction:
  - Directions (`w=0`) are **not affected by translation**
  - Points (`w=1`) **are affected by translation**

✅ One-sentence takeaway

- **Depth nonlinearity** comes from perspective projection: farther objects divide by a larger `z` → compressed depth.
- **Points at infinity** have `w=0` in homogeneous coordinates because dividing by zero encodes "no finite endpoint" — exactly what you need for ray directions.

If you want, I can also **draw a small diagram showing 1/z depth compression and points at infinity as vectors** — it makes both of these ideas visually obvious.

Do you want me to do that?

I think we had a great discussion. Could you reorganize all concepts and details from the beginning of this chat session entirely so reader can understand all concepts easily? Don't loss any concept or details.

Absolutely! I can reorganize the entire discussion into a **single, structured, self-contained document**, keeping **all the concepts, code references, and details**, in a logical flow so that any reader can follow from start to finish. I'll break it into sections with headings, explanations, math, and examples, just like a well-organized tutorial.

Here's the full, reorganized version:

Understanding Ray Generation, Camera Coordinates, and Projection in NeRF

This document consolidates all the concepts we discussed in this chat session, including ray generation, camera coordinates, normalized coordinates, homogeneous coordinates, clip space, NDC, and their connections to NeRF.

1️⃣ Ray Generation from a Camera

In NeRF and similar 3D rendering pipelines, we first generate **rays** from the camera into the scene.

PyTorch function:

python

```python
def get_rays(H, W, K, c2w):
    i, j = torch.meshgrid(torch.linspace(0, W-1, W), torch.linspace(0, H-1
    i = i.t()
    j = j.t()
    dirs = torch.stack([(i-K[0][2])/K[0][0], -(j-K[1][2])/K[1][1], -torch.
    rays_d = torch.sum(dirs[..., np.newaxis, :] * c2w[:3,:3], -1)
    rays_o = c2w[:3,-1].expand(rays_d.shape)
    return rays_o, rays_d
```

NumPy version:

python

```python
def get_rays_np(H, W, K, c2w):
    i, j = np.meshgrid(np.arange(W, dtype=np.float32), np.arange(H, dtype=
    dirs = np.stack([(i-K[0][2])/K[0][0], -(j-K[1][2])/K[1][1], -np.ones_l
    rays_d = np.sum(dirs[..., np.newaxis, :] * c2w[:3,:3], -1)
    rays_o = np.broadcast_to(c2w[:3,-1], np.shape(rays_d))
    return rays_o, rays_d
```

Explanation of the code:

1. **Pixel grids (`i` , `j`)**
   - `i` : horizontal pixel coordinates (0 → W-1)
   - `j` : vertical pixel coordinates (0 → H-1)
   - Generated using `meshgrid` .
2. **Camera intrinsics (`K`)**
   - Focal lengths: `f_x = K[0,0]` , `f_y = K[1,1]`
   - Principal point: `c_x = K[0,2]` , `c_y = K[1,2]`
   - These define how pixels map to directions.
3. **Normalized camera coordinates**
   - Formula:

$$x_n = \frac{i - c_x}{f_x}, \quad y_n = -\frac{j - c_y}{f_y}, \quad z_n = -1$$

   - **Negation of y**: because image coordinates grow **down**, but camera space y grows **up**.
   - Result: ray direction in **camera coordinates**.
4. **World coordinates**
   - Rotate using camera-to-world matrix `c2w[:3,:3]` .
   - Translate origin to camera center `c2w[:3,-1]` .

Copy code

Key concepts:

Symbol	Meaning
`i` , `j`	Pixel coordinates (image grid)
`c_x` , `c_y`	Principal point of the camera (image center)
`f_x` , `f_y`	Focal lengths in pixels
`dirs`	Ray directions in camera space
`rays_o`	Ray origins in world space
`rays_d`	Ray directions in world space

2️⃣ Normalized Device Coordinates (NDC) and `ndc_rays`

python

```python
def ndc_rays(H, W, focal, near, rays_o, rays_d):
    t = -(near + rays_o[...,2]) / rays_d[...,2]
    rays_o = rays_o + t[...,None] * rays_d

    o0 = -1./(W/(2.*focal)) * rays_o[...,0] / rays_o[...,2]
    o1 = -1./(H/(2.*focal)) * rays_o[...,1] / rays_o[...,2]
    o2 = 1. + 2. * near / rays_o[...,2]

    d0 = -1./(W/(2.*focal)) * (rays_d[...,0]/rays_d[...,2] - rays_o[...,0]
    d1 = -1./(H/(2.*focal)) * (rays_d[...,1]/rays_d[...,2] - rays_o[...,1]
    d2 = -2. * near / rays_o[...,2]

    rays_o = torch.stack([o0,o1,o2], -1)
    rays_d = torch.stack([d0,d1,d2], -1)

    return rays_o, rays_d
```

What this does:

1. **Shift ray origins to near plane**:
   Ensures all rays start at the near clipping plane.
2. **Apply perspective projection**:
   - Divide x, y by z (`x/z` , `y/z`) → creates perspective (1/z depth nonlinearity)
   - Rescale to NDC cube [-1, 1]
3. **Compute new ray directions in NDC**.

Why use NDC?

- Stabilizes ray marching for forward-facing scenes.
- Depth is **nonlinear**, compressed for far points, expanded near the camera.
- Makes training NeRF networks more stable.

Copy code

Depth nonlinearity

From perspective:

$$x_{\text{proj}} = \frac{x_c}{-z_c}, \quad y_{\text{proj}} = \frac{y_c}{-z_c}$$

- Far points (`z_c` large) → smaller projected change
- Near points (`z_c` small) → larger projected change

This is the **1/z effect**.

3️⃣ Homogeneous Coordinates

Definition:

- Represent 3D points in 4D: `(x, y, z, w)`
- Points: `w = 1`
- Directions: `w = 0`
- Dehomogenize (convert back to 3D):

$$(x', y', z') = (x/w, y/w, z/w)$$

Why homogeneous coordinates?

- Allows **translation, rotation, scaling, and perspective projection** all with matrix multiplication.
- Directions (`w=0`) are not translated, only rotated.
- Points (`w=1`) are affected by translation.

Points at infinity:

- `w = 0` → dehomogenization gives division by zero → point at infinity
- Used to represent **rays/directions** in graphics and NeRF.

4️⃣ Camera → Clip Space → NDC → Screen Pipeline

Camera space:

- 3D coordinates relative to camera

- Camera looks down **−Z**
- Rays are defined here

Clip space:

- Apply **projection matrix**

\text{clip_coord} = P \cdot \text{camera_coord}

- Result: homogeneous `(x_clip, y_clip, z_clip, w_clip)`
- Clipping is done against ±w_clip before perspective divide
- Example: `w_clip = -z_c`

NDC:

- Divide by `w_clip` :

$$x_{\text{ndc}} = x_{\text{clip}}/w_{\text{clip}}, \quad y_{\text{ndc}} = y_{\text{clip}}/w_{\text{clip}}, \quad z_{\text{ndc}} = z_{\text{clip}}/w_{\text{clip}}$$

- Now all coordinates are in cube [-1, 1]
- Perspective is applied automatically via division by z

Screen space:

- Map NDC [-1, 1] → pixel coordinates

$$x_{\text{screen}} = \frac{(x_{\text{ndc}} + 1)}{2} W, \quad y_{\text{screen}} = \frac{(1 - y_{\text{ndc}})}{2} H$$

OpenGL vs DirectX NDC:

Feature	OpenGL	DirectX
z range	[-1, 1]	[0, 1]
Near plane	+1	0
Far plane	-1	1
Right-handed	Yes	Left-handed by default

Why NeRF skips clip space:

- Rays do not need triangle clipping
- No rasterization → no GPU pipeline needed
- Only NDC is used for **numerical stability**

5️⃣ Perspective Divide and Depth Nonlinearity

- Division by z after projection:

$$x_{\text{proj}} = x/-z, \quad y_{\text{proj}} = y/-z$$

- Far objects shrink → **depth compression**
- Near objects expand → **better precision near camera**
- Depth nonlinearity comes from the `/ z` term in the projection

---

6️⃣ Summary of Symbols and Concepts

Symbol	Meaning
`i`, `j`	Pixel coordinates
`c_x`, `c_y`	Principal point (image center)
`f_x`, `f_y`	Focal length in pixels
`dirs`	Ray directions in camera space
`rays_o`	Ray origins in world space
`rays_d`	Ray directions in world space
`w` / `w_clip`	Homogeneous coordinate, controls perspective
NDC	Normalized Device Coordinates, after perspective divide
Clip space	Homogeneous coordinates before division, used for clipping
Points at infinity	`w = 0`

7️⃣ Key Takeaways

1. **Camera space** defines rays relative to the camera.
2. **Normalized camera coordinates** are obtained by centering pixels and dividing by focal length.
3. **Y-axis negation** corrects the difference between image (downward) and camera (upward) coordinates.
4. **Homogeneous coordinates** enable perspective projection and translation using matrices.
5. **Clip space** stores points in homogeneous coordinates before perspective divide; `w_clip` controls perspective and clipping.
6. **NDC** is the normalized cube after perspective divide; `1/z` depth nonlinearity arises here.
7. **Points at infinity** have `w = 0` → used for ray directions.
8. **NeRF skips clip space**, applying NDC directly to rays for numerical stability.

✅ This is now a **complete, self-contained explanation** connecting:

- Ray generation
- Camera intrinsics
- Coordinate systems (camera, world, NDC)
- Homogeneous coordinates
- Clip space and perspective
- Depth nonlinearity
- NeRF-specific simplifications

If you want, I can also **draw a single diagram showing camera → NDC → rays with 1/z depth** so it becomes **fully visual** and intuitive.

Do you want me to do that?

↳ represent the **same point** (for any nonzero `k`).

what do you mean they represent the same point?

Great question — this is *the* key subtle idea behind homogeneous coordinates.

When I said:

> (x, y, z, w) and (kx, ky, kz, kw) represent **the same point** (for any nonzero k)

I meant **the same point in ordinary 3D space after you convert back**.

Let's unpack that very carefully.

1️⃣ How a homogeneous point becomes a 3D point

A homogeneous coordinate:

$$(x, y, z, w)$$

represents the **actual 3D point**:

$$\left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}\right)$$

This step is called **dehomogenization**.

2️⃣ Why scaling doesn't change the point

Now take any nonzero scalar k.

Start with:

$$(x, y, z, w)$$

Scale it:

$$(kx, ky, kz, kw)$$

Now dehomogenize **both**.

Original:

$$\left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}\right)$$

Scaled:

$$\left(\frac{kx}{kw}, \frac{ky}{kw}, \frac{kz}{kw}\right) = \left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}\right)$$

✅ **Exactly the same 3D point**

That's what "they represent the same point" means.

3️⃣ Concrete numeric example

Take the 3D point:

$$(2, 4, 6)$$

One homogeneous representation:

$$(2, 4, 6, 1)$$

Another:

$$(4, 8, 12, 2)$$

Another:

$$(1, 2, 3, 0.5)$$

Dehomogenize all of them:

- $(2/1, 4/1, 6/1) = (2, 4, 6)$
- $(4/2, 8/2, 12/2) = (2, 4, 6)$
- $(1/0.5, 2/0.5, 3/0.5) = (2, 4, 6)$

✅ Same 3D point every time

4️⃣ Geometric intuition (very important)

Think of homogeneous coordinates as **rays from the origin in 4D space**.

All points along this ray:

$$(x, y, z, w),\ (2x, 2y, 2z, 2w),\ (0.5x, 0.5y, 0.5z, 0.5w)$$

represent **one single 3D point**.

So a 3D point is not a single 4D vector — it's an **entire equivalence class** of vectors.

5️⃣ Why this is useful (not just abstract math)

This property allows:

- Perspective projection to work via division by `w`
- Clipping against `±w`
- Points at infinity (`w = 0`)
- A single matrix pipeline for all transformations

Without this "same point under scaling" rule, none of that would work.

6️⃣ Why directions are special (`w = 0`)

If:

$$(x, y, z, 0)$$

You **cannot divide by** `w` → no finite 3D point.

That means:

- The point is at **infinity**
- It represents a **direction**, not a location

This is why:

- Ray origins use `w = 1`
- Ray directions use `w = 0`

7️⃣ One-sentence takeaway

Homogeneous coordinates represent 3D points up to scale: multiplying all components by the same nonzero number does not change the actual point after dividing by `w` **.**

If you want, I can also:

- Connect this directly to **projective geometry**
- Show how this explains **vanishing points**
- Tie it back explicitly to **clip space and NDC**

Sure

Great — I'll finish the picture by connecting **homogeneous coordinates → projective geometry → vanishing points → clip space & NDC**.
This will make the "same point under scaling" idea feel *inevitable*, not arbitrary.

1️⃣ Homogeneous coordinates = projective geometry (the big idea)

In **Euclidean geometry**, a point is a single coordinate:

```css
( x, y, z )
```

In **projective geometry**, a point is an **equivalence class**:

Copy code

```css
(x, y, z, w)  ~  (kx, ky, kz, kw)   for k ≠ 0
```

All scaled versions represent the **same geometric point**.

Copy code

Why?

Because the *actual* point is defined only after **dehomogenization**:

$$(x/w,\ y/w,\ z/w)$$

So projective geometry:

- treats scale as meaningless
- keeps only **ratios**
- naturally models perspective

This is exactly how cameras work.

2️⃣ Points at infinity (why `w = 0` exists)

In Euclidean space:

- Parallel lines **never meet**

In projective geometry:

- Parallel lines **meet at infinity**

That "point at infinity" is encoded as:

```css
(x, y, z, 0)
```

Why?

Because you *cannot* divide by `w = 0` → the point has **no finite location**.

Direction vectors = points at infinity

Example:

```scss
(1, 0, 0, 0) → direction along +x
(0, 1, 0, 0) → direction along +y
```

These are not locations — they are **directions**.

This is why in graphics:

- **Points** use `w = 1`

- **Directions / rays** use `w = 0`

And why ray directions are unaffected by translation.

3️⃣ Vanishing points (the visual payoff)

Now let's connect this to what you *see*.

Example: railroad tracks

- The rails are parallel in 3D
- In an image, they **converge to a point**

That point is a **vanishing point**.

What is it mathematically?

It is the projection of a **direction vector**:

```scss
(dx, dy, dz, 0)
```

Copy code

After projection:

- `w` stays 0
- the direction maps to a point at infinity in projective space
- after normalization, it appears as a finite image location

So:

> **Vanishing points are points at infinity made visible by perspective projection.**

This only works because homogeneous coordinates allow `w = 0`.

4️⃣ Why scaling represents the same point (deep intuition)

Think of homogeneous coordinates as describing a **ray from the origin** in 4D:

css

```
(x, y, z, w)
```

All points along this ray:

Copy code

```scss
(x, y, z, w)
(2x, 2y, 2z, 2w)
(0.5x, 0.5y, 0.5z, 0.5w)
```

represent **one geometric point**.

Only the **direction** of the 4D vector matters — not its length.

That's why:

- scaling doesn't change the point
- division by `w` collapses the ray into a 3D point

5️⃣ How this explains clip space

Now we return to graphics.

After projection:

Copy code

```scss
(x_clip, y_clip, z_clip, w_clip)
```

This is **still projective space**.

The GPU does **not** immediately divide by `w_clip`.

Why?

Because:

- `(x, y, z, w)` and `(2x, 2y, 2z, 2w)` are the same point
- clipping must respect that equivalence

So clipping is done using:

```
-w_clip ≤ x_clip ≤ w_clip
-w_clip ≤ y_clip ≤ w_clip
-w_clip ≤ z_clip ≤ w_clip
```

These inequalities are **scale-invariant**.

That only works because points are defined **up to scale**.

Copy code

6️⃣ Perspective divide = choosing a representative

When the GPU performs:

ini

```ini
x_ndc = x_clip / w_clip
y_ndc = y_clip / w_clip
z_ndc = z_clip / w_clip
```

it is doing this:

> "Choose the representative of this equivalence class where `w = 1`."

Copy code

That's all the perspective divide is.

Before divide:

scss

```scss
(2x, 2y, 2z, 2w)
```

After divide:

Copy code

bash

```bash
(x/w, y/w, z/w)
```

Same point. Different representation.

Copy code

7️⃣ Why `w = -z` gives perspective

In the projection matrix:

```ini
w_clip = -z_camera
```

So after divide:

Copy code

```ini
x_ndc = x / -z
```

This is **exactly** the pinhole camera equation.

Copy code

Perspective emerges automatically because:

- farther points have larger |z|
- dividing by z shrinks them

No special “perspective logic” is needed — projective geometry does it for free.

8️⃣ How NeRF fits perfectly into this

NeRF:

- does not rasterize
- does not clip triangles
- does not need full homogeneous machinery

But it **still relies on the same math**:

- ray directions ≈ points at infinity (`w = 0`)
- ray origins ≈ finite points (`w = 1`)
- NDC ≈ post–perspective divide space
- depth nonlinearity ≈ `1/z`

NeRF simply performs these steps **analytically**, instead of via matrices and clip space.

9️⃣ Final mental model (lock this in)

- Homogeneous coordinates describe **points up to scale**
- Scaling does **not** change the point
- `w = 0` encodes infinity (directions)
- Perspective is just **division by w**
- Clip space exists to safely manage this representation
- NDC is the normalized result after choosing `w = 1`

One-sentence master takeaway

Homogeneous coordinates work because geometry cares about ratios, not scale — and perspective, infinity, clipping, and ray directions all fall out naturally from that single idea.

If you want, next we can:

- walk **one 3D point numerically** through *every* space
- show **vanishing points in an actual image**
- connect this to **multi-view geometry & epipolar lines**

Just tell me which one you want.

↳ Image plane at `z = -f` (negative because camera looks down –Z)

is f positive? Is the image plane before camera or after camera?

Great question — this goes straight to **camera sign conventions**, and it's something that *everyone* trips over at first.

I'll answer it carefully and unambiguously.

Short answer (then we'll justify it)

- **Yes, the focal length `f` is positive.**
- In the **OpenGL / NeRF camera convention**, the **image plane is in front of the camera**, at

$$z = -f$$

- The negative sign comes from the **choice of coordinate axes**, not from `f` being negative.

1️⃣ Why `f` is always positive

The **focal length** is a **physical distance**:

- Distance from the camera center (pinhole) to the image plane
- Distances are **non-negative by definition**

So:

```nginx
f > 0
```

Always.

Copy code

If you ever see a "negative focal length," that's not physics — it's a **coordinate convention issue**.

2️⃣ Camera coordinate system being used

In NeRF and OpenGL-style camera space:

```css
x → right
y → up
z → backward
```

That means:

- The camera is at the origin `(0, 0, 0)`

Copy code

- The camera **looks along −Z**
- Points in front of the camera have `z < 0`

This is the key fact.

3️⃣ Where is the image plane?

The image plane must be placed:

- **In front of the camera**
- Along the viewing direction
- At a distance `f` from the camera center

Since the viewing direction is **−Z**, the image plane is at:

$$z = -f$$

So:

- `f` is positive
- `z = -f` is negative because "forward" is negative Z

4️⃣ Why not put the image plane at `z = +f` ?

You *could*, but then:

- The camera would be looking down **+Z**
- Or the image would be mirrored
- Or all your projection equations would flip signs

Many computer vision systems (e.g., some OpenCV formulations) *do* use +Z forward — they just adjust the math accordingly.

NeRF follows the **OpenGL convention**, so:

forward = −Z
image plane = z = −f

5️⃣ Pinhole projection (this explains everything)

Let a 3D point in camera space be:

$$(x_c, y_c, z_c), \quad z_c < 0$$

Image plane at:

$$z = -f$$

By similar triangles:

$$\frac{x_{\text{image}}}{-f} = \frac{x_c}{z_c}$$

So:

$$x_{\text{image}} = f \frac{x_c}{-z_c}$$

Same for y.

This is where the familiar formula comes from:

```text
x_proj = x / -z
y_proj = y / -z
```

No negative focal length anywhere — only a **negative z**.

Copy code

6️⃣ How this appears in NeRF code

In your ray-generation code:

```python
dirs = [(i - cx)/fx, -(j - cy)/fy, -1]
```

This corresponds to:

Copy code

- Image plane at `z = -1`
- Focal length implicitly absorbed into `(i - cx)/fx`
- Ray points forward because `z = -1 < 0`

Again:

- `f > 0`
- Image plane is **in front of the camera**
- Negative z is just "forward"

7️⃣ Common source of confusion (important)

People sometimes say:

> "The image plane is behind the camera in OpenGL"

What they *mean* is:

- The **virtual image plane** used for math is at `z = −f`
- Not the physical sensor layout
- This avoids image flipping and keeps equations simple

It does **not** mean the camera sees backward.

✅ Final clear statement

- **The focal length** `f` **is positive**
- **The image plane is in front of the camera**
- **It is at** `z = −f` **because the camera looks down −Z**
- The negative sign comes from the coordinate system, not from the optics

If you want next, I can:

- compare this directly to **OpenCV's +Z-forward convention**
- show how the equations change if the image plane were at `+f`
- reconcile this with **physical camera sensors**

Just tell me 👍

↪ The standard OpenGL perspective projection matrix is: P\=\ [1atan(θ/2)00001tan(θ/2)0000−f+nf−n−2fnf−n00−10\]P = \\begin{bmatrix} \\frac{1}{a \\tan(\\theta/2)} & 0 & 0 & 0 \\\\ 0 & \\frac{1}{\\tan(\\theta/2)} & 0 & 0 \\\\ 0 & 0 & -\\frac{f+n}{...

Why do we have perspective projection matrix like this?

Excellent question. This matrix looks mysterious until you see that **every term is there to satisfy a very specific geometric requirement**.
I'll derive it **from requirements**, not magic, and explain **why each row has exactly that form**.

We'll use the **OpenGL / NeRF convention** throughout.

The goal of the perspective projection matrix

We want a matrix P that transforms:

```css
camera space  →  clip space
```

Copy code

such that after perspective divide:

```css
clip space / w  →  NDC
```

Copy code

the following are true:

Required properties

1. **Perspective**:

$$x_{\text{ndc}} = \frac{x}{-z}, \quad y_{\text{ndc}} = \frac{y}{-z}$$

2. **Correct field of view** (FOV)
3. **Correct aspect ratio**
4. **Near and far planes map to fixed NDC z values**
5. **Points in front of camera have positive w**

Everything in the matrix exists to enforce one of these.

Step 1: Force perspective with `w = −z`

This is the *most important step*.

We want the perspective divide to produce:

$$x_{\text{ndc}} = \frac{x_{\text{clip}}}{w_{\text{clip}}} = \frac{x}{-z}$$

That means:

$$w_{\text{clip}} = -z$$

So the **last row must be**:

$$\begin{bmatrix} 0 & 0 & -1 & 0 \end{bmatrix}$$

Because:

$$w_{\text{clip}} = 0 \cdot x + 0 \cdot y - 1 \cdot z + 0 \cdot 1 = -z$$

✅ This single row creates perspective.

Step 2: Scale x and y by field of view

We want points on the **edge of the view frustum** to map to:

ini

```ini
x_ndc = ±1
y_ndc = ±1
```

Copy code

Vertical FOV

Let:

- vertical FOV = θ
- near plane at $z = -n$

At the top of the frustum:

$$y = n \tan(\theta/2)$$

After projection:

$$y_{\text{ndc}} = \frac{y_{\text{clip}}}{w_{\text{clip}}} = \frac{Ay}{-z}$$

At the near plane:

$$z = -n \Rightarrow y_{\text{ndc}} = \frac{An\tan(\theta/2)}{n} = A\tan(\theta/2)$$

We want:

$$y_{\text{ndc}} = 1 \Rightarrow A = \frac{1}{\tan(\theta/2)}$$

So the **y scaling term** must be:

$$\frac{1}{\tan(\theta/2)}$$

Aspect ratio correction (x direction)

If:

$$a = \frac{\text{width}}{\text{height}}$$

Then horizontal FOV is wider by factor a.

So x must be scaled by:

$$\frac{1}{a\tan(\theta/2)}$$

This gives the first two rows:

$$\begin{bmatrix} \frac{1}{a\tan(\theta/2)} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(\theta/2)} & 0 & 0 \end{bmatrix}$$

Step 3: Mapping depth (near & far planes)

This is the trickiest part.

We want:

Camera z	NDC z (OpenGL)
$z = -n$	$+1$
$z = -f$	-1

Assume:

$$z_{\text{clip}} = Az + B \quad \text{and} \quad w_{\text{clip}} = -z$$

After divide:

$$z_{\text{ndc}} = \frac{Az + B}{-z}$$

Apply constraints

Near plane: $z = -n$

$$1 = \frac{-An + B}{n} \Rightarrow B = n(1 + A)$$

Far plane: $z = -f$

$$-1 = \frac{-Af + B}{f} \Rightarrow B = f(A - 1)$$

Solve the system

Set them equal:

$$n(1 + A) = f(A - 1)$$

Solving:

$$A = -\frac{f + n}{f - n}$$

$$B = -\frac{2fn}{f - n}$$

This gives the third row:

$$[0 \quad 0 \quad -\frac{f+n}{f-n} \quad -\frac{2fn}{f-n}]$$

Step 4: Assemble the full matrix

Putting everything together:

$$P = \begin{bmatrix} \frac{1}{a\tan(\theta/2)} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(\theta/2)} & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

What each row does (intuitively)

Row	Purpose
Row 1	Horizontal FOV + aspect ratio
Row 2	Vertical FOV
Row 3	Nonlinear depth mapping
Row 4	Perspective divide (`w = -z`)

Why this matrix *has* to look like this

Because:

- Perspective **requires division by z**
- Division by z requires **homogeneous coordinates**
- Homogeneous coordinates require **w = -z**

- FOV requires **tan(θ/2)**
- Depth mapping requires **two constraints → two constants**

There is **no freedom left** after imposing the geometry.

Key insight (very important)

> **The perspective projection matrix is not arbitrary — it is the unique matrix that satisfies the pinhole camera model, perspective division, and near/far depth constraints simultaneously.**

One-sentence takeaway

This matrix exists because perspective is "divide by z," and homogeneous coordinates are the only way to encode that division using linear algebra before the divide actually happens.

If you want next, I can:

- derive the **DirectX version** side-by-side
- show how this matrix changes for **infinite far plane**
- derive it **starting from similar triangles only**

Just tell me which one you want.