

# 04\_Math

Matrices, Trigonometry, Geometry, Sign Distance Fields

- [Matrices](#)
  - [New Page](#)
- [Trigonometry](#)
- [Geometry](#)
- [Sign Distance Fields](#)
- [Cross Product](#)
- [Dot Product](#)
- [Quaternions](#)
- [raycast](#)
- [normals](#)
- [Spaces](#)
- [Questions](#)
- [Bounding Volumes](#)

# Matrices

# New Page

- ☐ **Vectors describe positions or directions within a coordinate space.**
- ☐ **Matrices describe how to transform vectors from one space to another.**

```
“ [ Xaxis.x Yaxis.x Zaxis.x Tx ]  
  [ Xaxis.y Yaxis.y Zaxis.y Ty ]  
  [ Xaxis.z Yaxis.z Zaxis.z Tz ]  
  [ 0      0      0      1 ]
```

A 4×4 transformation matrix is made up of:

- **3 direction vectors** (X, Y, Z axes — orientation)
- **1 position vector** (T — translation)
- **1 extra row** that supports **homogeneous coordinates**

Action	Result
Set <code>w = 1</code>	Vector becomes a <b>position</b> → gets full matrix transform (T * R * S)
Set <code>w = 0</code>	Vector becomes a <b>direction</b> → skips translation
Divide by <code>w</code>	Converts <b>clip space</b> → <b>NDC</b> (perspective divide)
Store something in <code>w</code>	Use <code>w</code> as a <b>packed value</b> (for instancing, motion, etc.)

YES — asking *what is a basis* means you’re ready to **understand all of 3D spacebuilding and matrix logic**. Let’s break it down clearly, intuitively, and practically for tech art ☐☐

## What Is a **Basis**?

```
“ A basis is a set of vectors that define a coordinate space.  
  It tells you:
```

- Where the **origin** is
- What direction **X, Y, Z** go
- What "**1 unit**" means in each direction

It's like building a mini coordinate system — your own local grid.

---

## □ Example: Standard Basis in 3D

The default basis in world space is:

$X = (1, 0, 0)$

$Y = (0, 1, 0)$

$Z = (0, 0, 1)$

This is called an **orthonormal basis**:

- Each axis is **unit length**
- Each axis is **perpendicular** to the others

This is what Unity uses for world space, object space (by default), and more.

---

## □□ Why Is a Basis Useful?

“ A basis lets you **build your own space** inside another space.

You can:

- Rotate it (to align to a surface)
  - Scale it (to stretch/compress space)
  - Offset it (with a position = origin)
  - Pack it into a matrix → for transforming vectors
- 

## □□ Basis = The Building Blocks of a Matrix

In a transformation matrix, the 3 main columns **are your basis vectors**:

[ Right Up Forward Position ]

[ X Y Z T ]

So:

- The **first 3 columns** = your basis vectors (local X, Y, Z)
- The **4th column** = your new origin (translation vector)

□ A matrix is just **abasis with a location**.

## □□ Real Tech Art Examples

Use Case	Basis Involved?	What It Does
Tangent space normal maps	TBN basis	Converts normals from UV space to world space
Custom pivot control	Custom basis	Reorients local X/Y/Z axes
Surface alignment shader	Basis from world normals	Builds a local space aligned to geometry
Procedural animation	Create and apply basis	Bones, aim constraints, etc.
Billboarding	Rebuild basis using camera	Rotate object to face view direction

“□ **Abasis** is a set of vectors that define a space's axes.

A **matrix** = **basis + position**.

You use it to **move**, **rotate**, or **remap** things between spaces.

- **Identity Matrix**

- Does nothing when applied to a vector.
- Basis for understanding other transformations.
- Represents standard world coordinates.

- **Uniform Scaled Matrix**

- Modify all diagonal values equally.

- Scales objects equally in all directions.
- E.g., `0.5` on all diagonal entries = mesh is scaled down by half.
- **Axis Scaled Matrix**
  - Change individual diagonal values independently.
  - Scales the object along specific axes.
  - E.g., a 3 in the X-axis = stretches object 3x in X only.
- **Mirror (Reflection) Matrix**
  - Multiply a diagonal entry by `-1` to mirror on that axis.
  - E.g., `-1` in X = flips mesh on the X-axis.
- **Matrix with Normalized Basis**
  - Basis vectors (matrix columns) are unit length and orthogonal.
  - Often used for pure rotation without scaling.
  - Each axis remains perpendicular and consistent in size.
- **Matrix with Non-Orthogonal Basis**
  - Columns are not perpendicular.
  - Causes skewing—object axes are no longer at right angles.
- **Matrix with Non-Normalized Basis**
  - Basis vectors may differ in length.
  - Can scale along arbitrary axes (not just X, Y, Z).
- **Rotation Matrices**
  - Represented by changing off-diagonal values (not just diagonal).
  - Rotate vectors in space while maintaining their magnitude.
- **Matrix Interpretation Tip**
  - Think of each matrix column as defining a new space axis.
  - Multiplying a vector by a matrix expresses it in that new space.

Let me know if you'd like a diagram or Unity version of any of these!

# Trigonometry

# Geometry



# Sign Distance Fields

# Cross Product

# Dot Product

# Quaternions

raycast

normals

# Spaces

## Tier 1 - Cartesian Coordinate Spaces

Space	Dim.	Why It's Core	Aliases	Translation Vector	What It Means
<b>UV Space</b>	2D	Texture sampling, blending, animation, atlases, UI masking	Texture space, Texcoord space, 2D UVs	(0.5, 0.5)	The center of a texture or sprite (50% across U and V)
<b>Object Space</b>	3D	Local mesh space — used in modeling, rigging, deformation	Local space, Model space, Mesh space	(1, 0, 0)	1 unit to the right of the object's pivot (local X axis)
<b>World Space</b>	3D	Scene-level effects, decals, VFX, projection, positioning	Global space, Scene space, Absolute space	(10, 2, -5)	A point 10 units right, 2 up, 5 back from world origin
<b>Camera Space</b>	3D	Needed for physical camera simulation, lighting, billboard, fresnel	View space, Eye space, View-relative space	(0, 0, -3)	A point 3 units in front of the camera
<b>Screen Space</b>	2D	UI layout, screen shaders, post-process FX tied to resolution	Pixel space, Raster space, Framebuffer space	(1920, 1080)	Bottom-right corner of a 1920x1080 screen
<b>Viewport Space</b>	2D	Resolution-independent layout, fullscreen FX, responsive shaders	Normalized screen space, 0-1 screen space, Screen UVs	(0.25, 0.75)	25% from the left, 75% from the bottom of the screen (normalized)

Space	Dim.	Why It's Core	Aliases	Translation Vector	What It Means
Tangent Space	3D	Local shading basis — used for normal maps, decals, and detail lighting	Surface space, Texture tangent space, TBN space	(0, 0, 1)	A direction pointing "out" from the surface (local Z axis)
Bone Space	3D	Used in rigging, skinning, animation — defines how vertices are deformed	Joint space, Skinning space, Bind pose space	(0, 1.2, 0)	A vertex 1.2 units above a bone's local origin (e.g., a joint's Y axis)

Space	Origin
UV Space	Bottom-left of the texture → (0, 0) in UV coords
Object Space	The pivot point of the object (usually center or base in modeling)
World Space	The global scene origin — Unity's (0, 0, 0) in the world
Camera Space	The camera's position → camera is at (0, 0, 0) and looks down -Z
Screen Space	Bottom-left of the screen (0,0) in Unity (unless it's top-left in other systems)
Viewport Space	Bottom-left of screen in normalized space → (0.0, 0.0)
Tangent Space	The origin is the current surface point, defined per-fragment or per-vertex
Bone Space	The origin is the base of the bone/joint — local to that bone

## Tier 2

Space	Dim.	Why It's Useful	Also Known As / Aliases	Example Vector	What It Means
-------	------	-----------------	-------------------------	----------------	---------------



<b>Clip Space</b>	4D	Final stage before rasterization; output of MVP transform in vertex shader	Homogeneous clip space, Post-projection space	$(-1, 1, 0.5, 1)$	Top-left corner of screen in clip space (with depth = 0.5)
<b>NDC Space</b>	3D	Screen-aligned normalized coords after perspective divide	Normalized Device Coordinates	$(0, 0, 0.5)$	Center of screen in normalized screen space (Z = 0.5 depth)
<b>Local-to-Parent</b>	3D	Relative transforms for procedural rigs or hierarchy animation	Parent-relative space, Local hierarchy space	$(0, 1, 0)$	1 unit above the parent bone or object
<b>Bone Bind Pose Space</b>	3D	Base pose of mesh before animation; used to calculate skin deformation	Inverse bind space, Rest pose	$(0.2, 0.5, 0)$	Original vertex position relative to its joint's bind pose
<b>Camera-Relative Space</b>	3D	Used for large-world precision, eye-relative motion, and virtual cameras	Eye-relative, Floating origin space	$(0, 0, -50000)$	A point 50,000 units in front of the camera — used for planetary scale scenes
<b>Screen UV Space</b>	2D	For fullscreen shaders and distortion FX	NDC → UV, Normalized screen space	$(0.5, 0.5)$	Center of the screen (used in post-processing and image effects)
<b>Spherical / Polar Space</b>	2D/3D	Custom projections, procedural effects, panorama/fisheye mapping	Angular space, Radial coordinates	$(\theta = 1.57, r = 3)$	A point 3 units away at 90° from the origin (polar coordinates)

# Questions

Sweet! Here are **a few 3D math-flavored Python coding interview questions** tailored for tech art — with increasing difficulty. I can go over answers/solutions with you too whenever you're ready.

---

## ☐☐ Easy: Vector Basics

**1. Find the direction vector between two points. Normalize it.**

```
# Given:
point_a = [3, 2, 1]
point_b = [6, 4, 5]

# Write a function:
# get_normalized_direction(a, b) -> unit direction vector from a to b
```

---

## ☐☐ Medium: Dot & Cross Product Logic

**2. Determine if an object is facing a target.**

```
# Given:
# - forward: the normalized forward vector of an object
# - to_target: vector from object to target position

# Write a function:
# is_facing(forward, to_target) -> True if angle between < 45 degrees
```

---

**3. Create an orthonormal basis given one direction vector.**

```
# Given:
# - a normalized forward vector

# Write a function:
```

```
# generate_basis(forward) -> (right, up)
# Such that forward, right, up are all orthogonal and normalized
```

## ☐☐ Hard: Matrix Transformations

### 4. Apply a transformation matrix to a 3D point.

```
# Given:
# - a 4x4 matrix (numpy array)
# - a 3D point [x, y, z]

# Write a function:
# transform_point(matrix, point) -> transformed 3D point
# Use homogeneous coordinates internally
```

### 5. World to Local Conversion

```
# Given:
# - world_matrix of an object
# - world position of a point

# Write a function:
# convert_to_local(world_matrix, world_point) -> point in object's local space
```

## ☐☐ Bonus: Rotation Math (Quaternions)

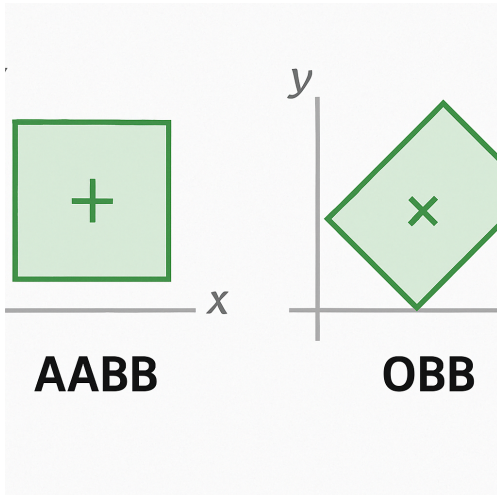
### 6. Interpolate between two rotations using quaternions.

```
# Given:
# - rotation_a and rotation_b as Euler angles
# - a blend value t in [0, 1]

# Write a function:
# blend_rotations(a, b, t) -> resulting Euler rotation
```

Let me know which one you want to tackle first, and I can walk through it with you or just check your answer!

# Bounding Volumes



Yes, this is indeed a classic **Axis-Aligned Bounding Box (AABB)** intersection question, which is **very relevant** for technical artists — especially in areas like:

- Collision detection
- Shader bounding box calculations
- AR/VR hit tests
- Culling logic
- Level-of-detail systems
- Raycasting optimizations

## □□ Problem Breakdown

You're given two AABBs defined as:

```
box = [left, bottom, right, top]
```

And you're told:

- The boxes do **not** intersect if they only **touch at the edges**.
- They **must have area of overlap** (i.e., true intersection, not adjacency).

# ☐ Intersection Logic

Two AABBs intersect **if and only if**:

```
not (  
    box1.right <= box2.left or  
    box1.left >= box2.right or  
    box1.top <= box2.bottom or  
    box1.bottom >= box2.top  
)
```

## ☐☐ Why Tech Artists Should Know This

Tech artists frequently deal with:

- Bounding boxes for **visual effects triggering**
- Geometry occlusion
- Placement tools
- Region-based optimizations
- Unity/Unreal custom tools

Even shaders can use similar logic for intersection checks in screen space or world space.

---

Would you like a Python example for this, or maybe a visual explanation or diagram?

Great question — this is a **key detail** in many AABB problems, especially for tech artists dealing with precision and hitboxes.

## ☐☐ What It Means:

When the problem says:

“The boxes do **not** intersect if they only **touch at the edges**”

...it means **boxes must share actual area** (non-zero width and height of overlap) to count as "intersecting."

# ☐ Intersecting

Two boxes **intersect** if they overlap **inside** — like this:

```
+-----+
|      |
|  A   |
|  +-----+
|  |   B   |
+---+-----+
```

Here, A and B overlap **partially**, so it's an intersection.

---

# ☐ Not Intersecting — Edge Touch

Boxes that **just touch** on the side like this:

```
+-----+
| A |
+-----+
      +-----+
      | B |
      +-----+
```

Here, box A's **right side equals** box B's **left side** — they **touch**, but **do not overlap**, so this is **not** an intersection.

Same applies if they touch **top to bottom**, or at **corners**.

---

# ☐☐ Tech Artist Analogy

Think of this like placing two tiles next to each other on a grid. If they're only **flush**, there's no actual shared area — they're adjacent, not overlapping.

This matters a lot in:

- Physics engines (collision vs proximity)
- Raycasting hitboxes
- Shader region effects

- Bounding-box culling
- 

Want me to draw an ASCII or image diagram to show edge vs overlap visually?