Transformations

Computer Graphics 2020

TA 2

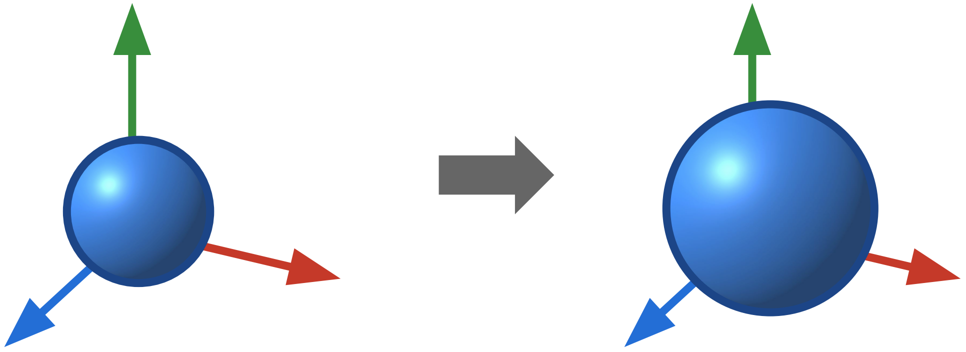
- 3D transformations recap
- Rotation about an arbitrary axis algorithm
- EX1 + The BVH format

3D Transformation Matrices

- A matrix representing some linear transformation in 3D space
- We use homogeneous coordinates to allow for translations
- Because of this, a 3D transformation matrix will have dimensions 4×4

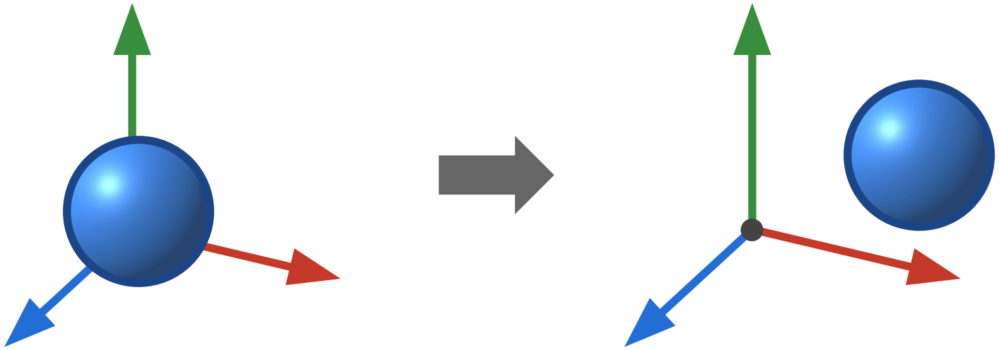
Scaling Matrix

$$\begin{bmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} ax \\ by \\ cz \\ 1 \end{bmatrix}$$



Translation Matrix

$$\begin{bmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + a \\ y + b \\ z + c \\ 1 \end{bmatrix}$$



Combining Transforms

- We can combine transformations by multiplying their associated matrices
- Let us define:

$$T = \begin{bmatrix} 1 & 0 & 0 & \mathbf{1} \\ 0 & 1 & 0 & \mathbf{1} \\ 0 & 0 & 1 & \mathbf{1} \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad S = \begin{bmatrix} \mathbf{2} & 0 & 0 & 0 \\ 0 & \mathbf{2} & 0 & 0 \\ 0 & 0 & \mathbf{2} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Does $TS = ST$?

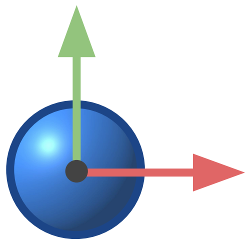
Combining Transforms

- Matrix multiplication is noncommutative
- Similarly, the order in which we apply transformations changes the result

$$TS = \begin{bmatrix} \mathbf{2} & 0 & 0 & \mathbf{1} \\ 0 & \mathbf{2} & 0 & \mathbf{1} \\ 0 & 0 & \mathbf{2} & \mathbf{1} \\ 0 & 0 & 0 & 1 \end{bmatrix} \neq \begin{bmatrix} \mathbf{2} & 0 & 0 & \mathbf{2} \\ 0 & \mathbf{2} & 0 & \mathbf{2} \\ 0 & 0 & \mathbf{2} & \mathbf{2} \\ 0 & 0 & 0 & 1 \end{bmatrix} = ST$$

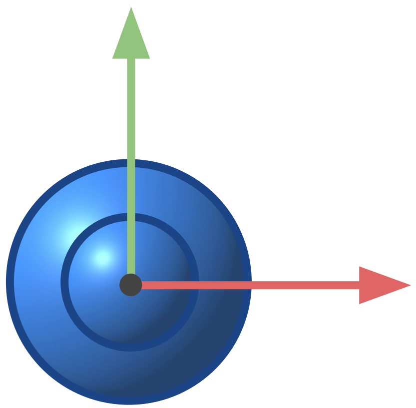
Combining Transforms

- We can also think about applying the transforms geometrically to see that order matters
- Note that like in matrix multiplication, the transformations are applied from right to left
- Consider a sphere centered on the origin:



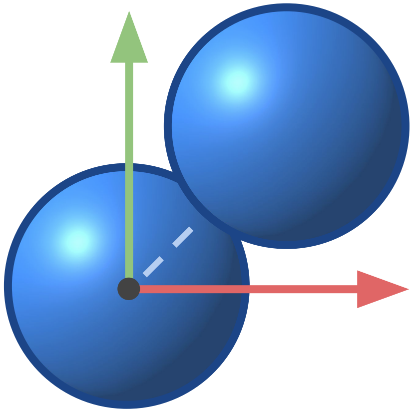
Combining Transforms

- If we apply TS to the sphere, first we scale using the scaling matrix S:



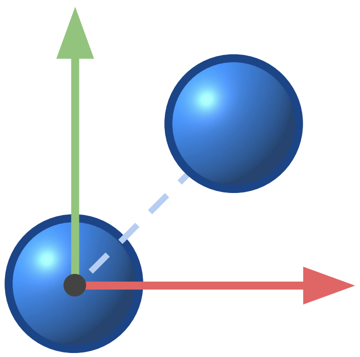
Combining Transforms

- Then we translate using the translation matrix T:



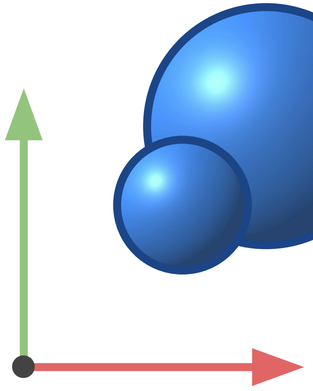
Combining Transforms

- If we apply ST to the sphere, first we translate using the translation matrix T:



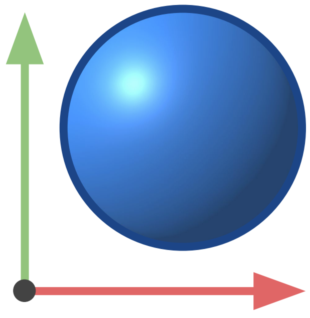
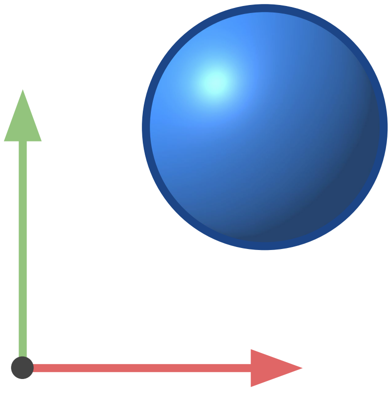
Combining Transforms

- Then we scale using $\mathbf{S}$. Note that scaling transforms everything in relation to the origin:



Combining Transforms

- We get a different result!

$$TS = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 0 & 2 & 0 & 1 \\ 0 & 0 & 2 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \neq \begin{bmatrix} 2 & 0 & 0 & 2 \\ 0 & 2 & 0 & 2 \\ 0 & 0 & 2 & 2 \\ 0 & 0 & 0 & 1 \end{bmatrix} = ST$$

Rotation Matrices

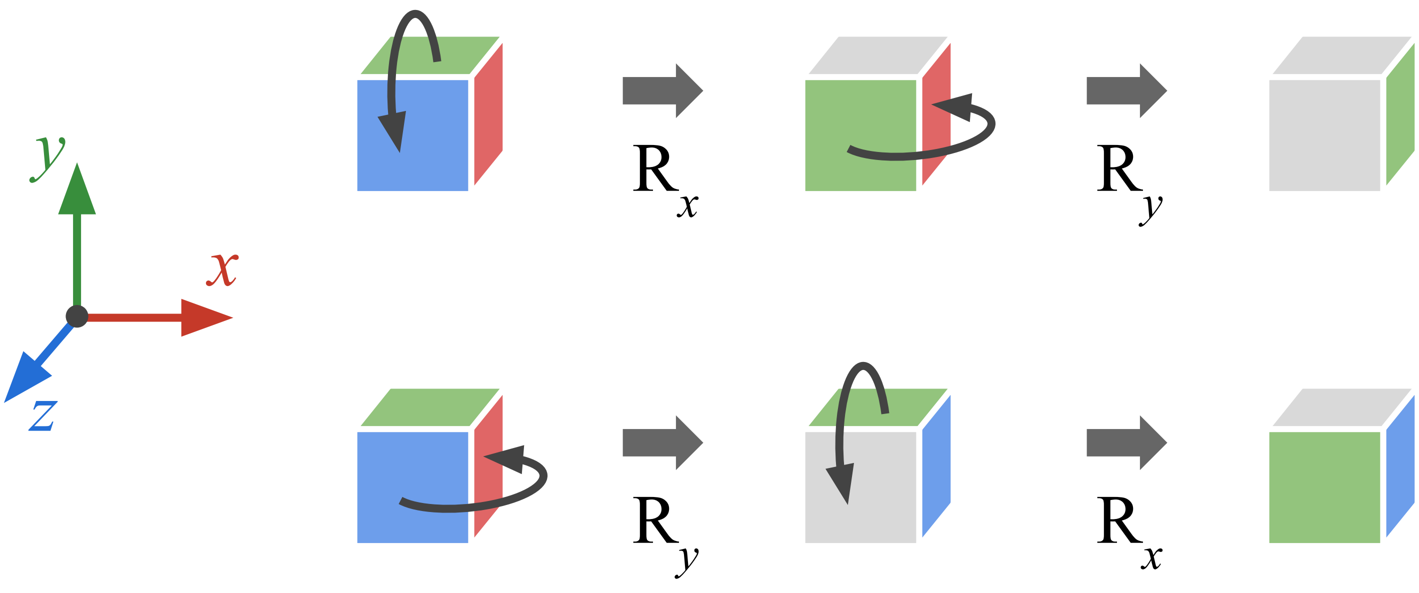
- To rotate, we can define a rotation matrix about each axis. For example, the x axis:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y\cos\theta - z\sin\theta \\ y\sin\theta + z\cos\theta \\ 1 \end{bmatrix}$$

- As seen in the lecture, similar matrices can be constructed for the y and z axes

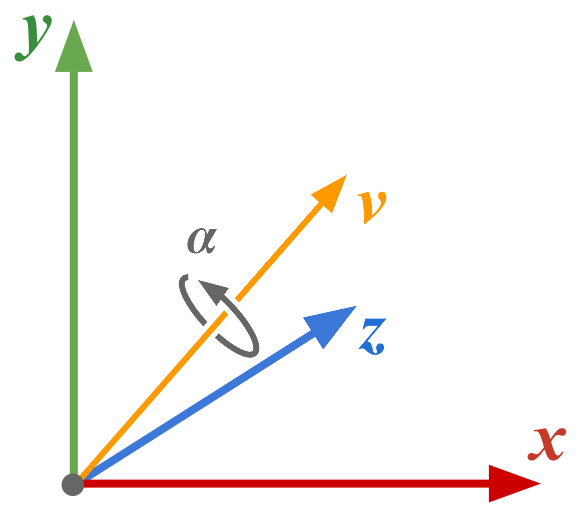
Combining Rotations

- Like scaling and translating, the order in which we combine rotations matters
- Let R_x, R_y be rotations of 90° about the x, y axes:



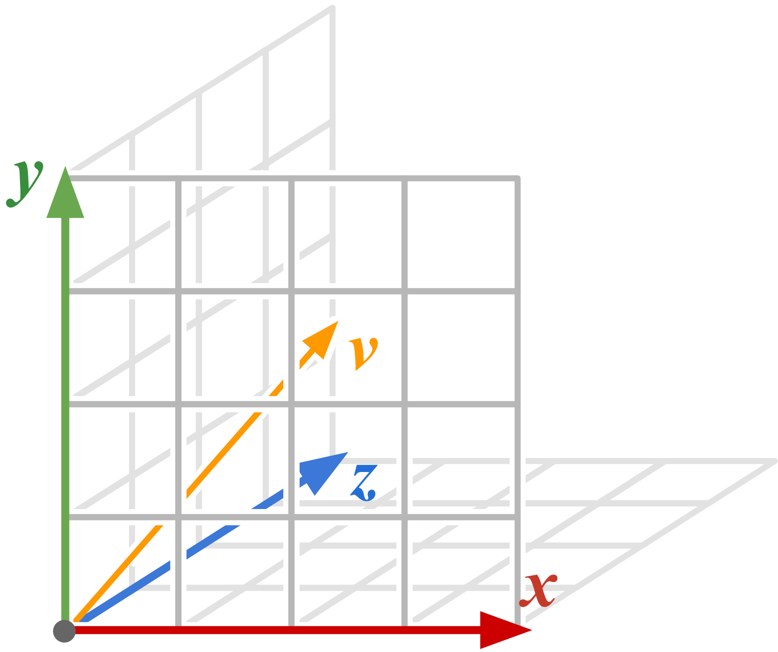
Rotation About an Arbitrary Axis

- We want to find a rotation matrix $\mathbf{R}$ that can rotate $\boldsymbol{\alpha}$ degrees about some arbitrary axis $\boldsymbol{v}$



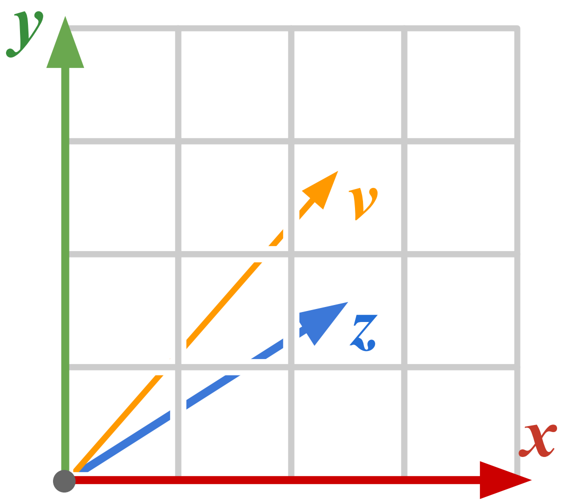
Rotation About an Arbitrary Axis

- To understand the diagram better, take a look at the planes defined by each pair of axes



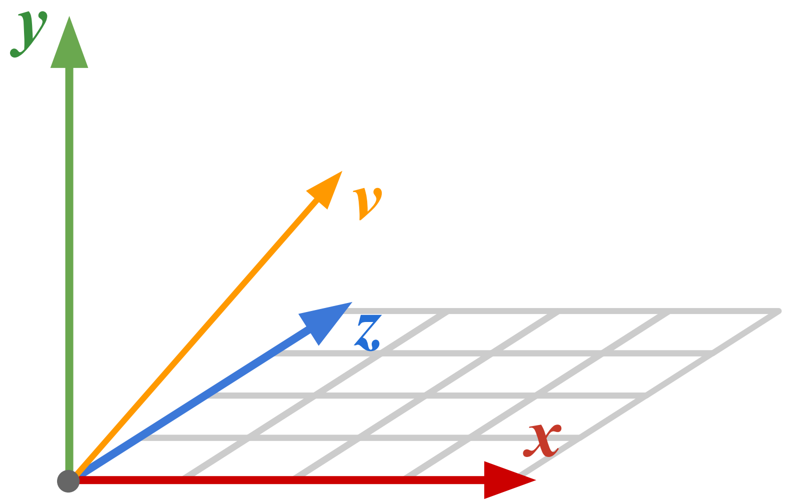
Rotation About an Arbitrary Axis

- The XY Plane:



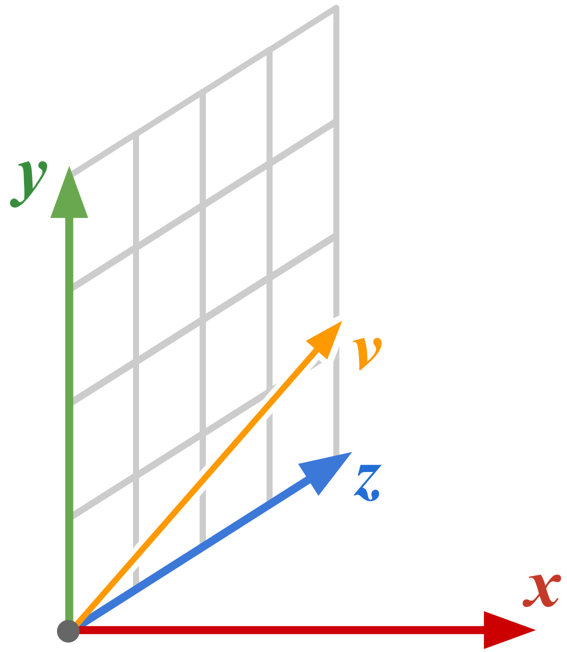
Rotation About an Arbitrary Axis

- The **XZ** Plane:



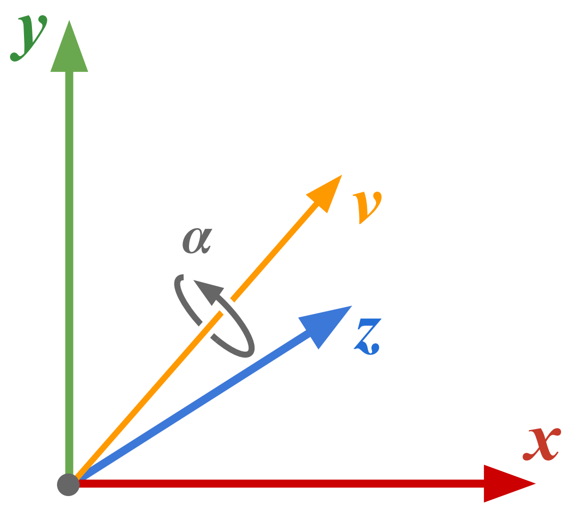
Rotation About an Arbitrary Axis

- The **YZ** Plane:



Rotation About an Arbitrary Axis

- We can assume $\boldsymbol{v}$ is normalized, because its length does not affect the rotation

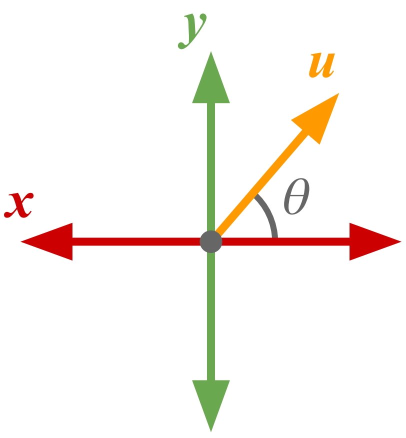


Rotation About an Arbitrary Axis

- We know how to rotate about the main coordinate axes x, y, z
- We want to find a correct combination of these rotations that we can combine to get $\mathbf{R}$
- First we need to learn about finding angles using the 2-Argument Arctangent

2-Argument Arctangent

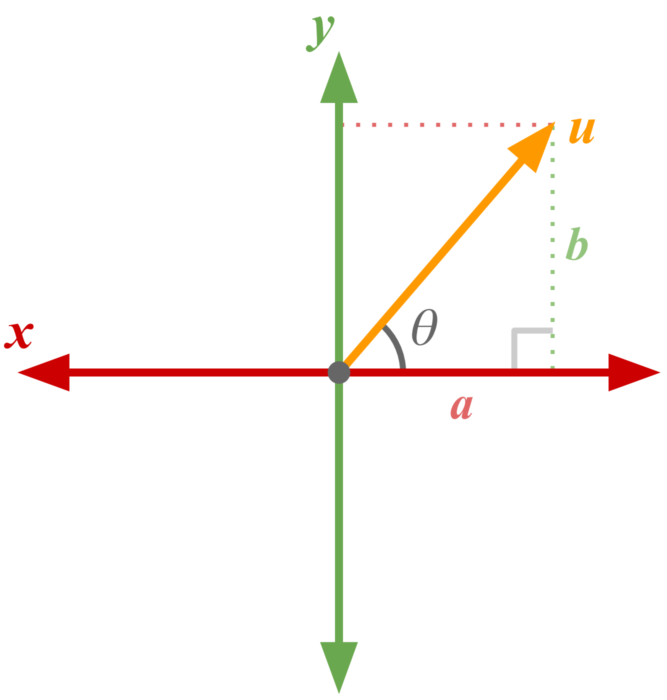
- How do we find the angle θ between a 2D vector $\boldsymbol{u} = (\boldsymbol{a}, \boldsymbol{b})$ and the x axis?



- Assume $\boldsymbol{u}$ is normalized (if not, we can normalize without changing θ)

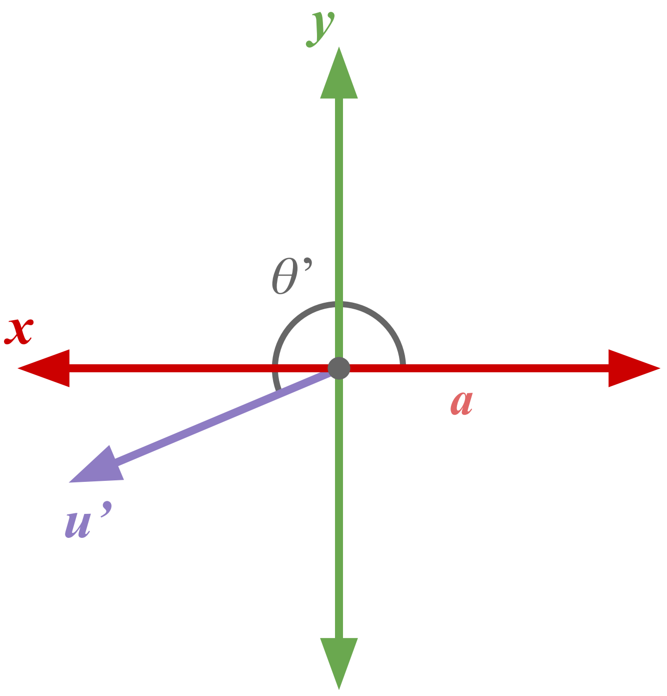
2-Argument Arctangent

- We can see that $\tan\theta = \boldsymbol{b} \,/\, \boldsymbol{a}$
- So we get $\theta = \arctan(\boldsymbol{b} \,/\, \boldsymbol{a})$



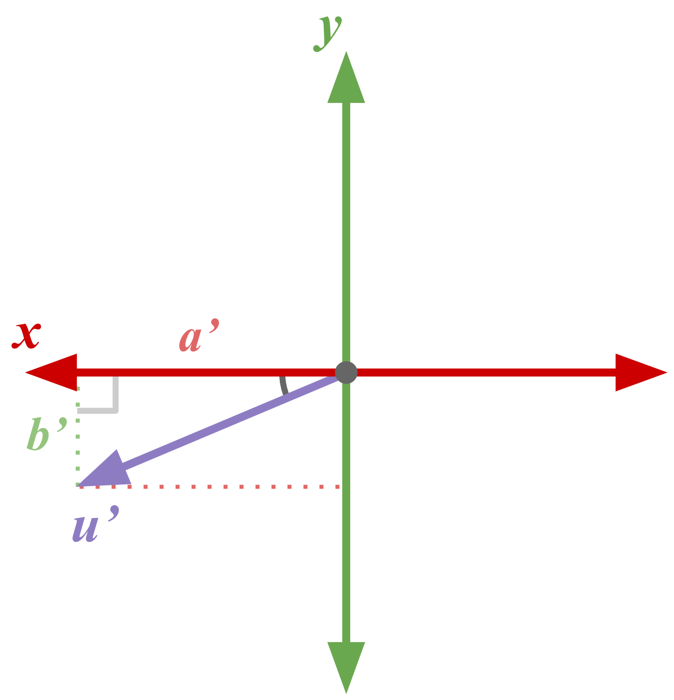
2-Argument Arctangent

- How about $\boldsymbol{u}' = (\boldsymbol{a}', \boldsymbol{b}')$?



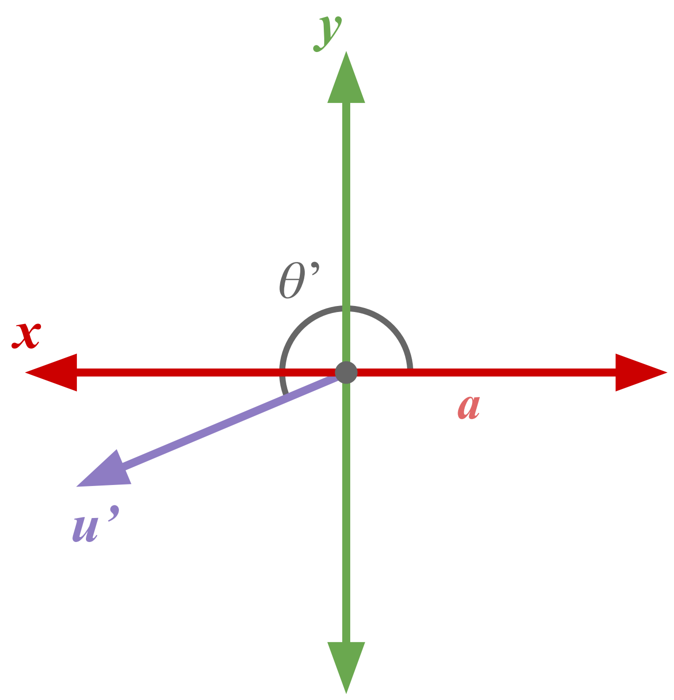
2-Argument Arctangent

- arctan(***b'*** / ***a'***) will only give us the angle between ***u'*** and the negative side of the ***x*** axis



2-Argument Arctangent

- We need to add some multiple of 90° that depends on the quadrant: $\theta' = 180° + \arctan(\boldsymbol{b'} \,/\, \boldsymbol{a'})$



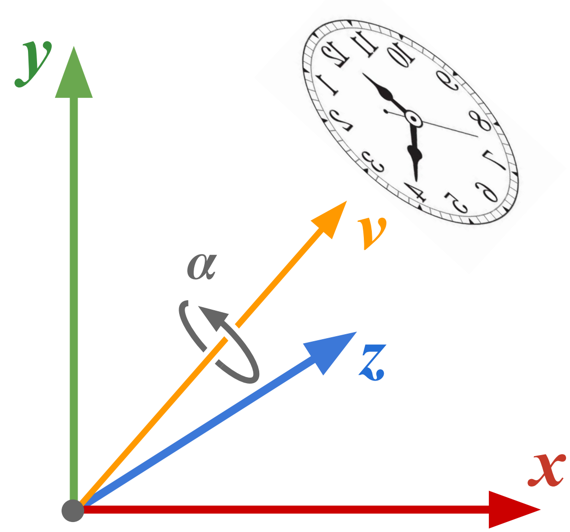
2-Argument Arctangent

- Most programming math libraries we have the ***2-Argument Arctangent*** function which returns θ, taking into account the quadrant
- Usually called *arctan2* or *atan2*
- Very useful tool to get an angle from coordinates on a 2D plane
- In Unity we can use it like so:

```
Mathf.Atan2(u.y, u.x);
```

Rotation About an Arbitrary Axis

- So how do we actually find $\mathbf{R}$?
- Denote $\boldsymbol{v} = (\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c})$, assume $\boldsymbol{v}$ is normalized
- We'll use clockwise rotations, like Unity

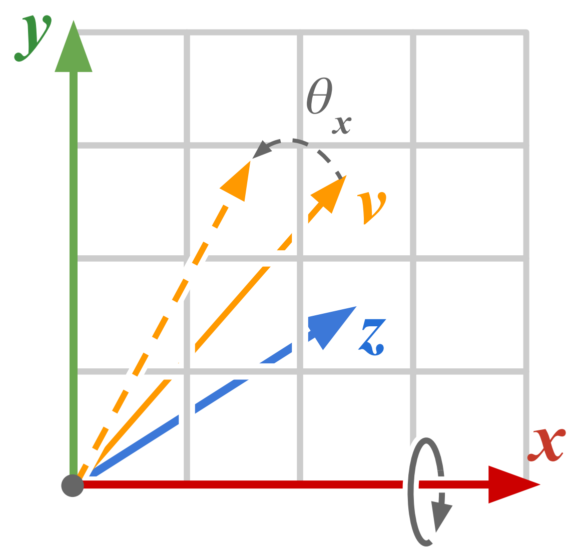


Rotation About an Arbitrary Axis

- The idea: make $\boldsymbol{v}$ coincident with one of the coordinate axes, rotate, then transform back:
  1. Rotate about the $\boldsymbol{x}$ axis into the $\mathbf{XY}$ plane
  2. Rotate about the $\boldsymbol{z}$ axis into the $\mathbf{YZ}$ plane - now $\boldsymbol{v}$ is aligned with the $\boldsymbol{y}$ axis
  3. Rotate about the $\boldsymbol{y}$ axis by $\boldsymbol{\alpha}$
  4. Apply inverse rotation about the $\boldsymbol{z}$ axis
  5. Apply inverse rotation about the $\boldsymbol{x}$ axis

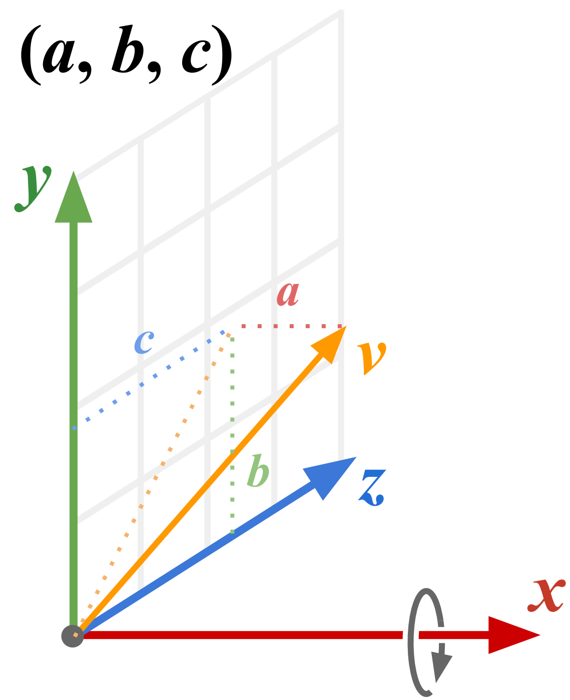
1. Rotate Into XY Plane

- We want to build a matrix $\mathbf{R}_x$ that will rotate $\boldsymbol{v}$ about the x axis into the XY plane
- How do we find the rotation angle θ_x?



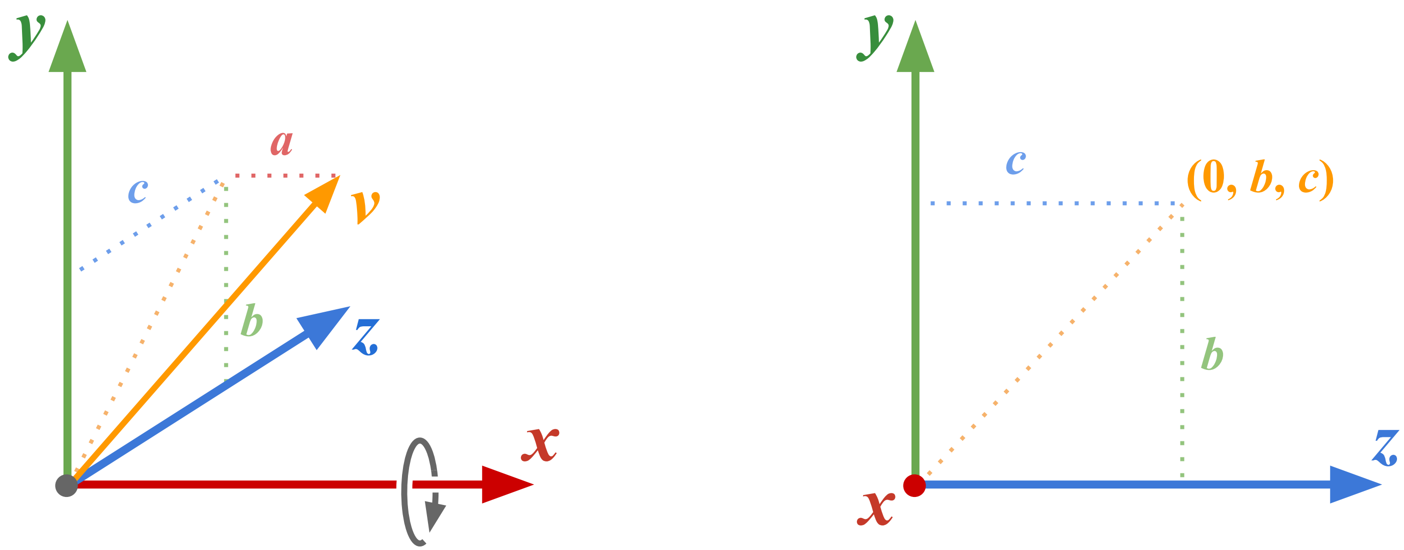
1. Rotate Into XY Plane

- We want to look at the projection of $\boldsymbol{v}$ onto the YZ plane (why?)
- Remember $\boldsymbol{v} = (a, b, c)$



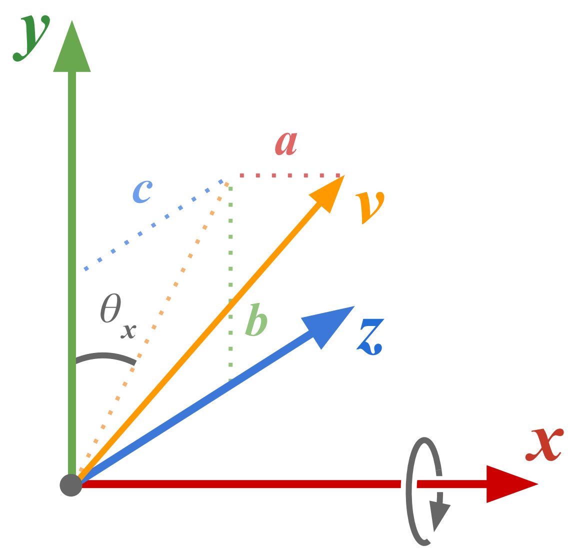
1. Rotate Into XY Plane

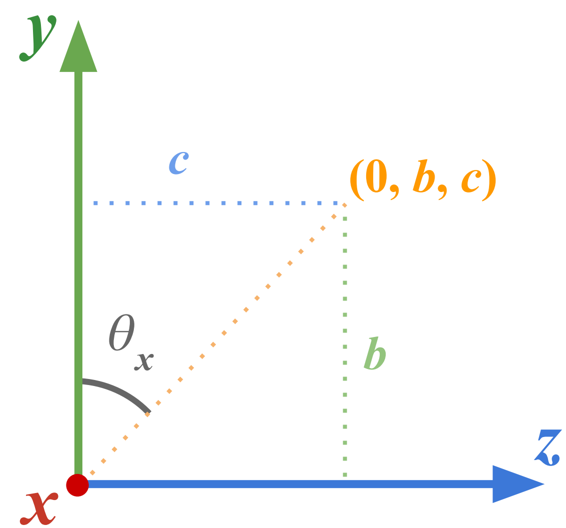
- The projection is the vector $(0, b, c)$



1. Rotate Into XY Plane

- θ_x is the angle between the y axis and the projected $\boldsymbol{v}$
- How do we calculate it?

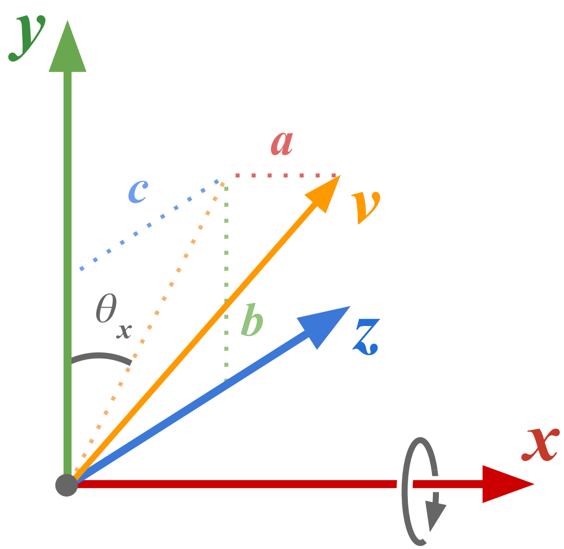


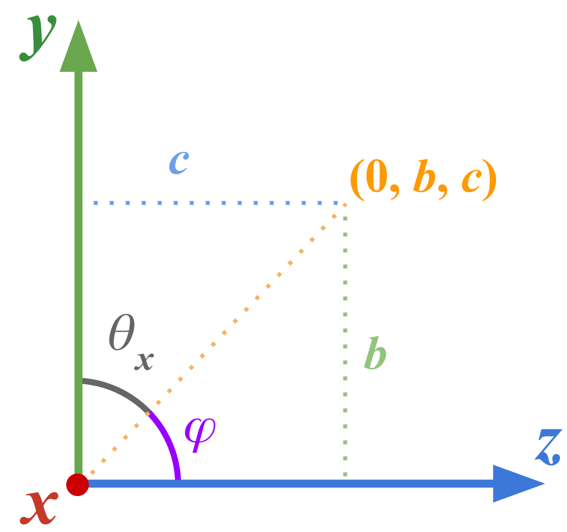


1. Rotate Into XY Plane

- We know that $\varphi = \mathrm{atan2}(b, \mathrm{c}) \quad \Rightarrow$

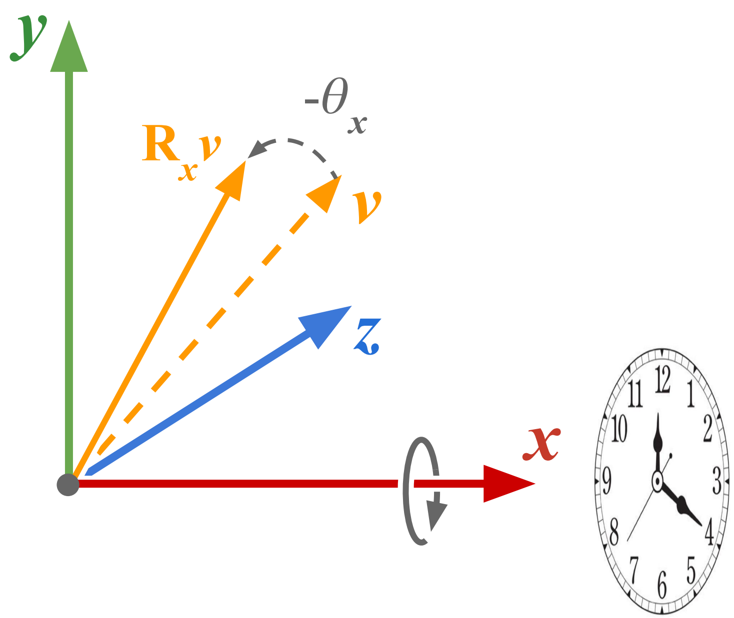
$$\theta_x = 90° - \varphi = 90° - \mathrm{atan2}(b, c)$$





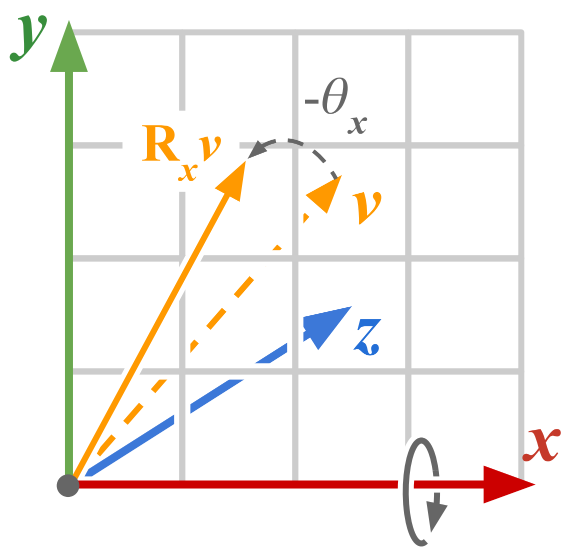
1. Rotate Into XY Plane

- We can now construct a rotation matrix $\mathbf{R}_x$
- Remember we are using clockwise rotations, so we must actually rotate by $-\theta_x$!



1. Rotate Into XY Plane

- We apply $\mathbf{R}_x$ to $\boldsymbol{v}$ and it lands on the XY plane
- What are its coordinates?



1. Rotate Into XY Plane

- Denote $\mathbf{R}_x \boldsymbol{v} = (a', b', c')$, we want to find a', b', c'
- We rotated around the x axis, so the x coordinate remains the same: $a' = a$
- We also know that $\mathbf{R}_x \boldsymbol{v}$ is on the XY plane, and that means $c' = \mathbf{0}$

1. Rotate Into XY Plane

- Rotation preserves lengths
- Remember we assumed is $\boldsymbol{v}$ normalized:

$$\sqrt{a^2 + b^2 + c^2} = \|v\| = 1 = \|R_x v\| = \sqrt{a'^2 + b'^2 + c'^2}$$

- Everything is equal to 1, we can square both sides

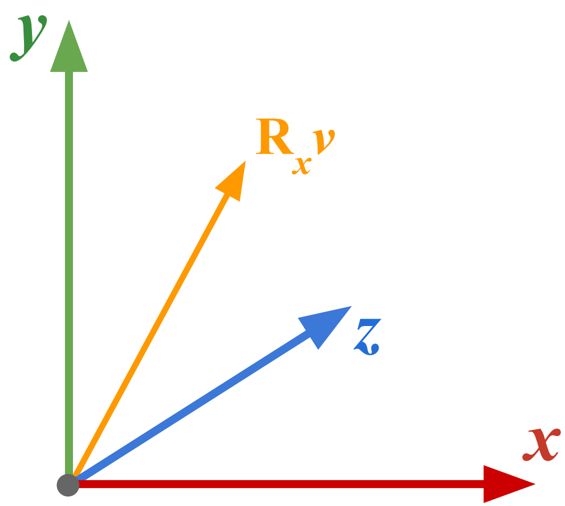
$$\Rightarrow a^2 + b^2 + c^2 = a'^2 + b'^2 + c'^2 = a^2 + b'^2 + 0$$

$$\Rightarrow b'^2 = b^2 + c^2$$

$$\Rightarrow b' = \sqrt{b^2 + c^2}$$

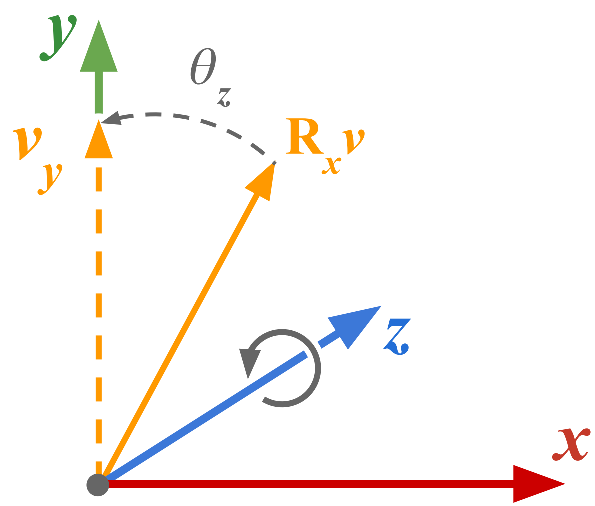
1. Rotate Into XY Plane

- We have found: $\mathbf{R}_x \boldsymbol{v} = (\boldsymbol{a}, \sqrt{\boldsymbol{b}^2 + \boldsymbol{c}^2}, 0)$



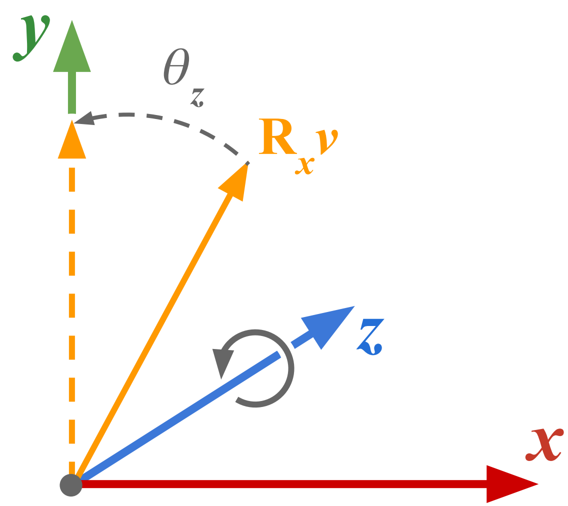
2. Rotate Into YZ Plane

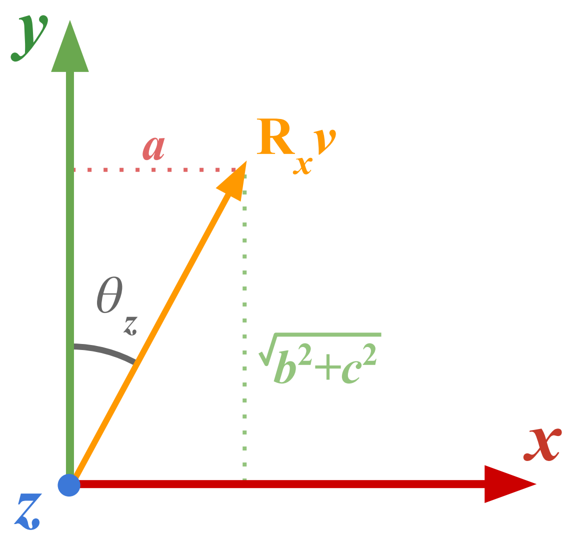
- Now we need $\mathbf{R}_z$ that will rotate about the z axis into the YZ plane
- How do we find the rotation angle θ_z?



2. Rotate Into YZ Plane

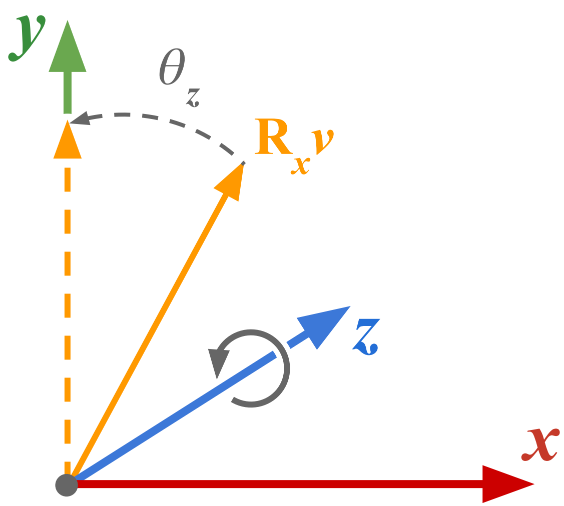
- $\mathbf{R}_x \boldsymbol{v}$ is already on the XY plane
- θ_z is the angle between the y axis and $\mathbf{R}_x \boldsymbol{v}$

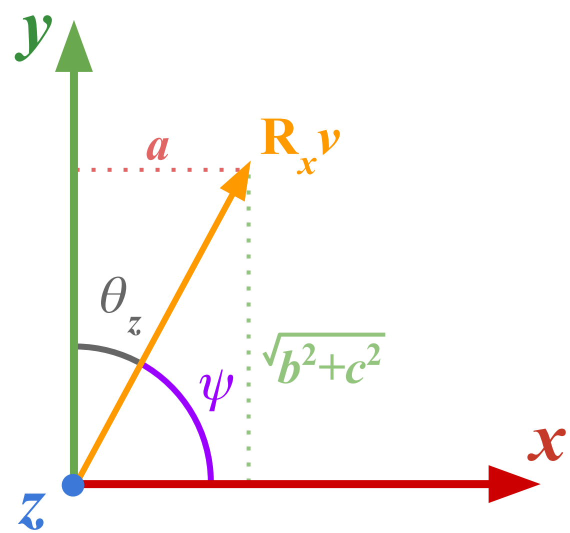




2. Rotate Into YZ Plane

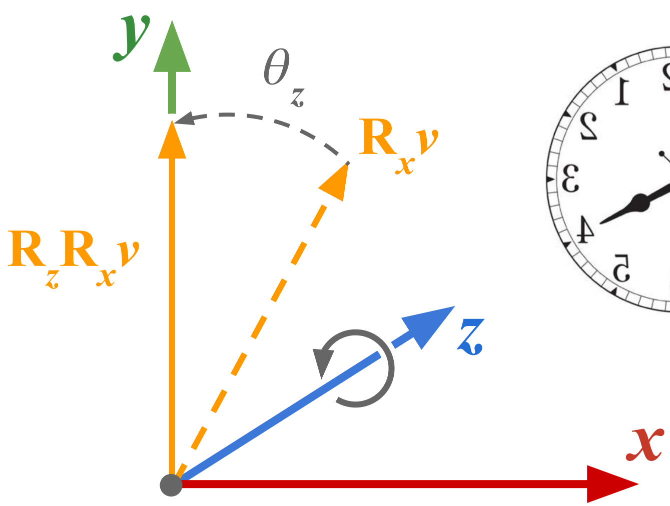
- We know that $\psi = \mathrm{atan2}(\sqrt{b^2+c^2}, a)$
- So we get: $\theta_z = 90° - \psi = 90° - \mathrm{atan2}(\sqrt{b^2+c^2}, a)$

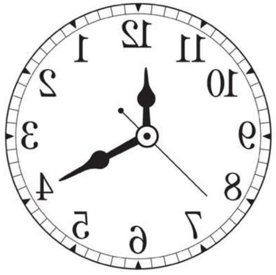




2. Rotate Into YZ Plane

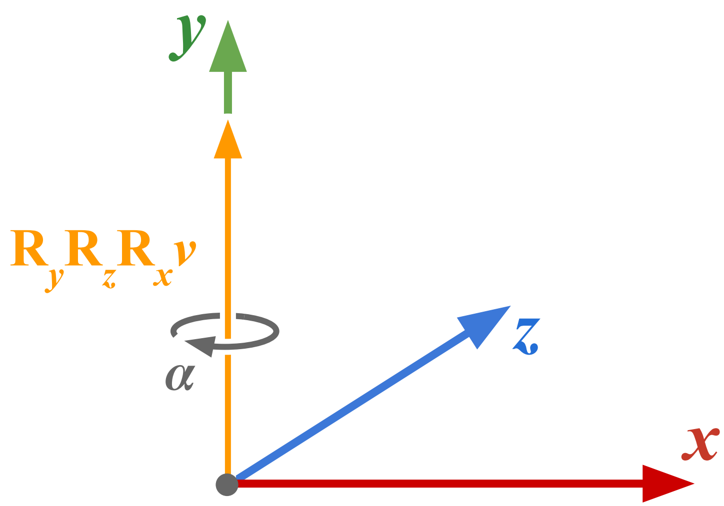
- We apply $\mathbf{R}_z$ to $\mathbf{R}_x\boldsymbol{v}$ and it lands on the y axis!
- Note that the rotation is already clockwise in relation to the z axis





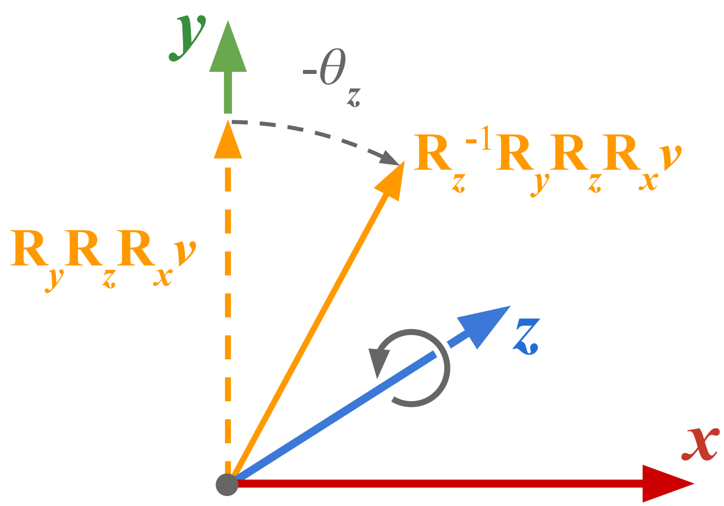
3. Rotate about the y axis by α

- Now we can create a matrix $\mathbf{R}_y$ to rotate by our target angle $\boldsymbol{\alpha}$ around the y axis, which is coincident with $\mathbf{R}_z\mathbf{R}_x\boldsymbol{v}$



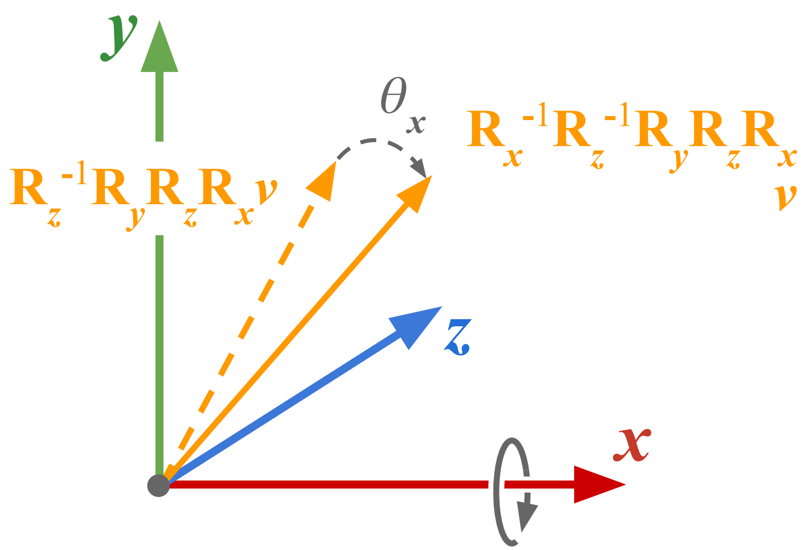
4. Apply Inverse z Rotation

- All we need to do now is rotate everything back to place
- First, apply a rotation of $-\theta_z$ about z, which is $\mathbf{R}_z^{-1}$



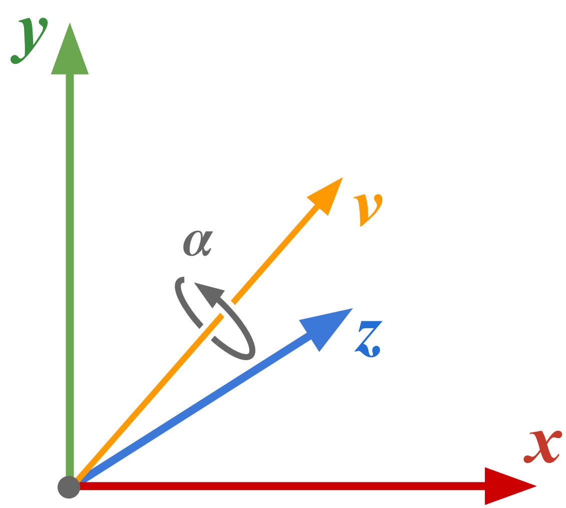
5. Apply Inverse x Rotation

- Finally, apply a rotation of θ_x about the x axis (remember we originally rotated by $-\theta_x$), which is actually the matrix $\mathbf{R}_x^{-1}$



Rotation About an Arbitrary Axis

- We have found $\mathbf{R} = \mathbf{R}_x^{-1}\mathbf{R}_z^{-1}\mathbf{R}_y\,\mathbf{R}_z\,\mathbf{R}_x$ that gives a rotation of $\boldsymbol{\alpha}$ about $\boldsymbol{v}$!



Coordinate Systems

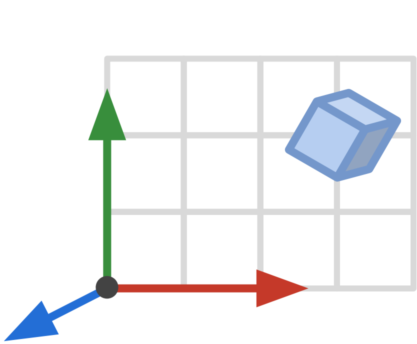
- When talking about transforms, we must understand which coordinate system we are working in
- The same transformation may produce different results in different coordinate systems
- A transformation matrix can be used to transfer between coordinate systems

Frames of Reference

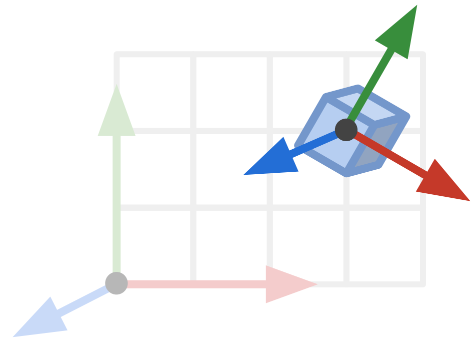
- ***World Space*** or ***Global Space*** is a coordinate system from the viewpoint of our world (in Unity, the scene), centered on the world origin **(0, 0, 0)**
- ***Object Space*** or ***Local Space*** is a coordinate system from the viewpoint of a specific object, centered on the object itself
- A ***World to Object*** transformation matrix can be used to transfer between them (and vice versa)

Frames of Reference

- Consider a cube rotated 30° about the z axis and translated to $(3, 2, 0)$:



World Space



Object Space

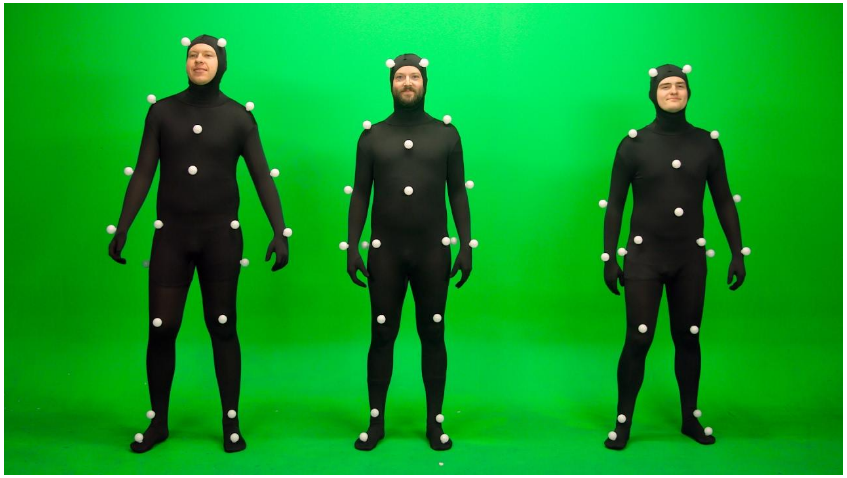
- In world space its coordinates are $(3, 2, 0)$. Object space is centered on it, so it is at $(0, 0, 0)$

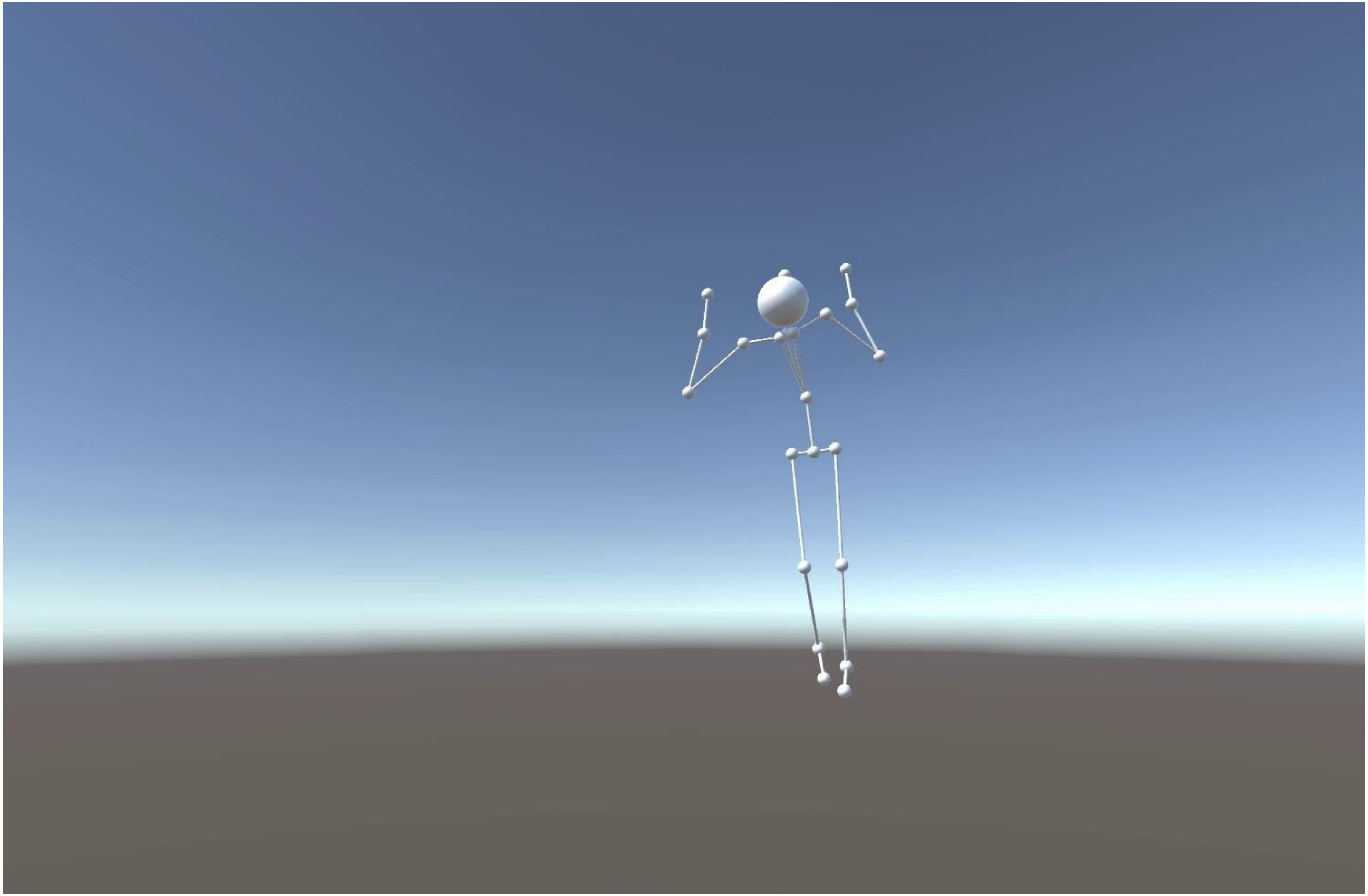
EX1

- In this exercise you will use motion capture data to draw and animate a simple 3D character on screen
- The goal of this exercise is to learn about animation and 3D transformations
- You **must** submit this exercise in pairs

Motion Capture

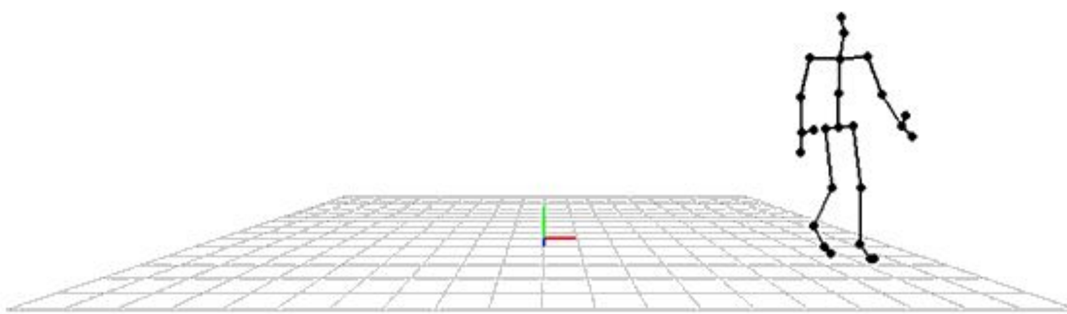
- ***Motion Capture*** generally refers to recording actions of (usually) human actors, and using that information to animate digital character models in 2D or 3D computer animation





The BVH Format

- The BVH file format was originally developed by Biovision as a way to store motion capture data
- Every BVH file contains two sections - ***Hierarchy*** and ***Motion***



The BVH Format

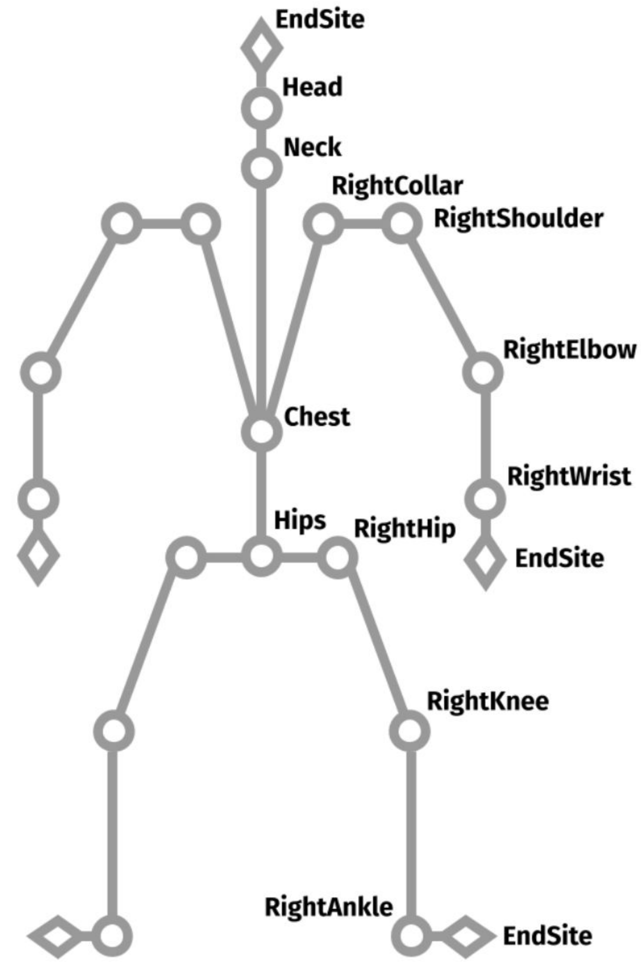
```
HIERARCHY
ROOT Hips
{
    OFFSET 0.0123 37.4256 -0.07071
    CHANNELS 6 Xposition Yposition Zposition Zrotation ...
    JOINT Chest
    { ... } // children joints of chest
     ... // more children joints of hips
}

MOTION
Frames: 30
Frame Time: 0.033333
 8.03   35.01   88.36   -3.41   14.78   -2.33   10.11   ...
 7.81   35.10   86.47   -3.78   12.94   -3.02   10.23   ...
 ... // rest of frame channel data
```

BVH Hierarchy Section

- Contains a hierarchical data structure (a tree) in which each node represents a ***Joint*** in a skeleton
- The segment between two joints is called a ***Bone***
- Joints with no children are called ***End Sites*** (and technically aren't joints at all!)
- One ***Root Joint*** (the root of the tree)

Example BVH Skeleton Hierarchy

Hips (Root)
- ↳ **Chest**
  - ↳ **Neck**
    - ↳ **Head**
      - ↳ **EndSite**
  - ↳ **RightCollar**
    - ↳ **RightShoulder**
      - ↳ **RightElbow**
        - ↳ **RightWrist**
          - ↳ **EndSite**
  - ↳ **LeftCollar**
    - ↳ ...Rest of Left Arm
- ↳ **RightHip**
  - ↳ **RightKnee**
    - ↳ **RightAnkle**
      - ↳ **EndSite**
- ↳ **LeftHip**
- ↳ ...Rest of Left Leg

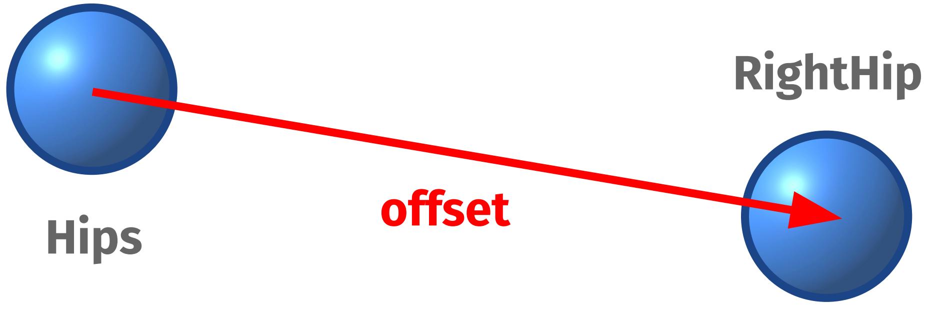


BVH Joints

- Each joint node in the tree has 3 fields:
  - **Offset** - the position of the joint relative to the parent joint's location (3D vector)
  - **Channels** - what transformation information in the motion data will be used to animate this joint
  - **Children** - a list of joints under this node in the hierarchy (like a regular tree node)

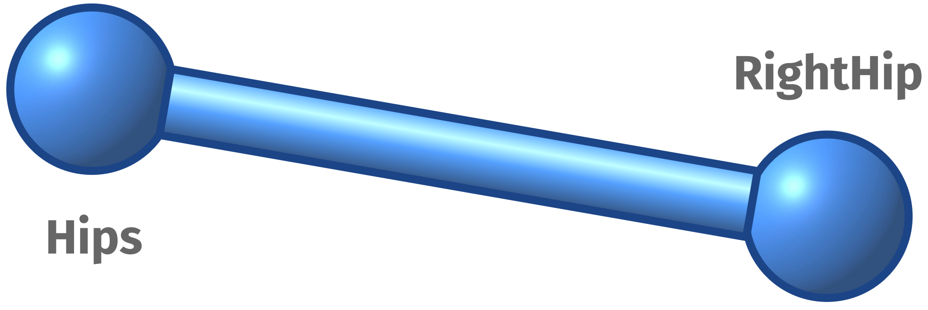
Building the Skeleton

- In order to build the skeleton (in its base pose) we need to iterate over the hierarchy tree
- For each joint, we calculate its final 3D position by adding the offset to the parent's position
- At each joint we will draw a sphere:



Building the Skeleton

- Between each joint and its parent we will draw a cylinder representing the bone
- The length of this bone can be derived from the offset



BVH Motion Section

- The motion section contains list of ***Keyframes***, Each one representing the skeleton's pose at a point in time
- A keyframe is just an array of `float` values
- Each keyframe contains ***Channels*** that correspond to angles around specific axes of specific rotations of specific joints
- The root joint also has position channels

BVH Motion Section

- Also contains the number of frames in the animation and length of each frame in seconds
- Example motion section:

```
MOTION
Frames: 30
Frame Time: 0.033333
 8.03   35.01   88.36   -3.41   14.78   -2.33   10.11   ...
 7.81   35.10   86.47   -3.78   12.94   -3.02   10.23   ...
```

Hips Xpos	Hips Ypos	Hips Zpos	Hips Zrot	Hips Xrot	Hips Yrot	Chest Zrot	...

Animating the Skeleton

- At each frame we need to adjust the skeleton's pose, according to the keyframe channel data
- First we need to determine each joint's **local space** transformation $\mathbf{M} = \mathbf{TRS}$, where:
  - $\mathbf{T}$ is the translation matrix
  - $\mathbf{R}$ is the rotation matrix
  - $\mathbf{S}$ is the scaling matrix

Animating the Skeleton

- The BVH format only allows for rigid transformations, there is no scaling: $\mathbf{S} = \mathbf{I}_4$
- The rotation matrix $\mathbf{R}$ needs to be constructed from the keyframe channel data
- To get $\mathbf{T}$ we need to construct a translation matrix from the joint's offset, given in the hierarchy section

Animating the Skeleton

- Each joint has 3 rotation channels associated with it, corresponding to angles about the x, y, z axes
- Ordering matters! Use the order specified in the hierarchy section for the joint
- For example, for a joint with channels:

```
Zrotation Xrotation Yrotation
```

The full rotation matrix is: $\mathbf{R} = \mathbf{R}_z \mathbf{R}_x \mathbf{R}_y$

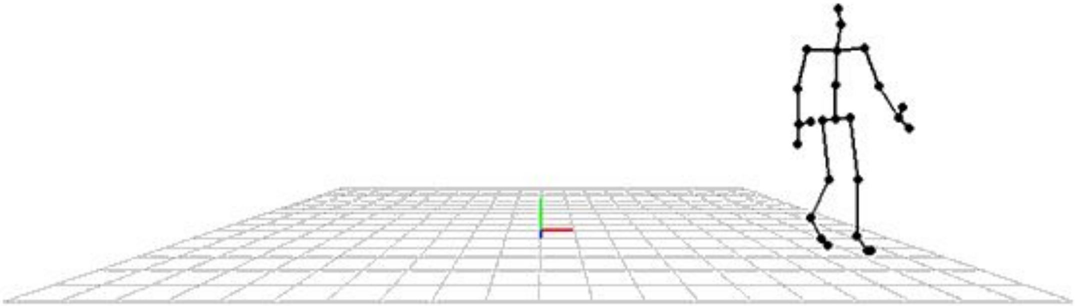
Animating the Skeleton

- The local transform $\mathbf{M}$ describes its orientation in local space, which in turn is subject to its parent's orientation
- To get the global transform $\mathbf{M'}$ for a given joint, the local transform $\mathbf{M}$ needs to be pre-multiplied by its parent's global transform, which is also obtained by pre-multiplying:

$$\mathbf{M'}_{\mathbf{RightAnkle}} = \mathbf{M}_{\mathbf{Hips}}\mathbf{M}_{\mathbf{RightHip}}\mathbf{M}_{\mathbf{RightKnee}}\mathbf{M}_{\mathbf{RightAnkle}}$$

Animating the Skeleton

- Finally we can use the the global matrix **M'** to transform each joint and animate the skeleton at each frame



List<T>

- C# List data structure, can hold items of any type

```
List<int> myList = new List<int>();
myList.Add(1);
myList.Add(5);

print(myList[1]);      // Prints "5"
print(myList.Count);   // Prints "2" (length of the list)

foreach (int item in myList)
{
    print(item); // Prints all list items
}
```

- Have a look at the [List<T> class reference]

Unity `Matrix4×4`

- Unity has a class representing a 4×4 matrix
- To declare an identity matrix:

```
Matrix4×4 a = Matrix4×4.identity
```

- A few common matrix operations:

$\mathbf{det(A)}$	`a.determinant`	`1.0`
$\mathbf{A}\boldsymbol{v}$	`a.MultiplyVector(v)`	`v`
$\mathbf{A}_{\mathbf{2,1}}$	`a.m21`	`0.0`
$\mathbf{A \cdot B}$	`a * b`	-

MatrixUtils

- In the exercise you will be given a class that creates and applies transformation matrices
- For example, to scale an object by 2, rotate it 45° about the x axis and translate it to (1, 2, 3):

```
Matrix4x4 t = MatrixUtils.Translate(new Vector3(1, 2, 3));
Matrix4x4 r = MatrixUtils.RotateX(90);
Matrix4x4 s = MatrixUtils.Scale(new Vector3(2, 2, 2));

MatrixUtils.ApplyTransform(gameObject, t * r * s);
```

BVHParser

- You will also be given a class that parses a BVH file and returns a BVHData object

```
BVHParser parser = new BVHParser();

BVHData data = parser.Parse(BVHFile);
```

```
public class BVHData
{
    BVHJoint rootJoint; // Root BVHJoint object
    int numFrames; // Number of frames in the animation
    float frameLength; // Length of each frame in seconds
    List<float[]> keyframes; // Keyframe data for animating
}
```

Good luck!