Reference Axes

3D space (with 6 DFs) is usually described with a reference axis.

MonoGames uses a right-hand coordinate system

positive Z towards viewer

Positive rotations about an axis are counter-clockwise looking along the axis to the origin.

Or, clockwise looking from the origin along the axis.

Place right hand around "axis" with thumb pointing along the positive direction and the finger curl is in the positive direction.
Object in 3 space

Consider an object in 3 space. It has a:

- position ( @ ) point, object's center ("anchor") location
- orientation "local reference frame"
  or axes: up, right, backward

Matrix.Identity represents a 3D reference axes with homogeneous coordinates:

(10,-5,0,1) a point and (10,-5,0,0) a vector)

A matrix (Orientation Matrix) can also represent an object's local reference frame.

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

Rows are the local reference frame vectors and center point.
Columns are their coordinates on the reference axes.

MonoGames Matrix documentation
Matrix Struct

fields  
M11, M12, ... M44  // M Row Column

methods  
CreateLookAt(...), CreateTranslation(...), 
CreateRotation?(...), CreateFromAxisAngle(...), 
CreateFromQuaternion(), CreateFromYawPitchRoll(...), 
CreateWorld(...), Invert(...), Transpose(...), Multiply(...). Lerp(...), ...

properties  
Forward, Backward, Right, Left, Up, Down, Translation

Matrix composition is order dependent. Rotations and scale before translation. Rotations about origin

<< see ConsoleMath distro for Math in a console window>>

Rotation submatrix "orientation"

Translation submatrix "position"
Forward / Backward

Matrix.Forward and Matrix.Backward may seem "reversed" for a right-handed coordinate system.

Forward is the direction of looking initially looking on –Z
The user is looking into the scene

Properties Right / Left, Up / Down are right-handed reference frame axes.

The matrix field values (M11 .. M44) are right-handed coordinates.

Matrix avatar = Matrix.Identity;

See: Matrix transforms and “handedness”
World View Projection

MonoGames has separate matrices for transformations of:

**World**
- objects specified in world coordinates (reference axes)
  - `Update(Gametime gt)` method sets up transformations.

**View**
- objects transformed to view coordinates – the camera's reference frame becomes the reference axes.
  - Often set in `LoadContent()`

**Projection**
- objects transformed to 2D display plane coordinates (3D clipping).
  - Often set in `LoadContent()`

World, View, and Projection transforms are applied in `Draw(Gametime gt)`.
Matrix.CreateLookAt(...) Assume a shape and camera set as:

Matrix.CreateLookAt(
    Vector3 cameraPosition,
    Vector3 lookAtPosition,
    Vector3 cameraUpVector)

Matrix.CreateLookAt(
    new Vector3(20, 50, 50),
    Vector3.Zero, Vector3.Up);

// Pseudocode operations
OM = Matrix.Identity; // camera's
// move camera to Position
OM *= translate(20, 50, 50)
siteVector = Vector3(Position,
    Lookat) // as a unit vector
// camera rotates X ≈ -45°
OM *= rotateX(aCosDot (siteVector, 
    OM.Forward))
//camera rotates Y ≈ -10°
OM *= rotateY(aCosDot(OM.Up, LookAt)
consoleMath, pyramid example

The consoleMath.cs program shows how to test/use MonoGames Matrices and Vectors inside a text based console program. Useful for testing.

Consider 3 Matrices: World, View and Project applied to the vertices of a 4 sided pyramid.

pyramid vertices:

\((-10.0, -5.0, 10.0), (10.0, -5.0, 10.0), (0.0, -5.0, -10.0), (0.0, 5.0, 0.0)\)
World transform

pyramid vertices:

( -5.0, -5.0, 5.0 )
( 5.0, -5.0, 5.0 )
( 0.0, -5.0, -5.0 )
( 0.0, 5.0, 0.0 )

World

<table>
<thead>
<tr>
<th></th>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>Right</td>
<td>1.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>Up</td>
<td>0.000</td>
<td>1.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>Backward</td>
<td>0.000</td>
<td>0.000</td>
<td>1.000</td>
<td>0.000</td>
</tr>
<tr>
<td>Translation</td>
<td>5.000</td>
<td>10.000</td>
<td>20.000</td>
<td>1.000</td>
</tr>
</tbody>
</table>

\[
\text{pyramidWC}_i = \text{Vector3.Transform(pyramid}[i], \text{World})
\]

( 0.0, 5.0, 25.0 )
( 10.0, 5.0, 25.0 )
( 5.0, 5.0, 15.0 )
( 5.0, 15.0, 20.0 )
View transform

Matrix.CreateLookAt( [0, 50, 50], [0, 0, 0], [0, 1, 0])

View

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>Right</td>
<td>1.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>Up</td>
<td>0.000</td>
<td>0.707</td>
<td>0.707</td>
</tr>
<tr>
<td>Backward</td>
<td>0.000</td>
<td>-0.707</td>
<td>0.707</td>
</tr>
<tr>
<td>Translation</td>
<td>0.000</td>
<td>0.000</td>
<td>-70.711</td>
</tr>
</tbody>
</table>

pyramidEC_i = Vector3.Transform(pyramidWC[i], View);

( 0.0, -14.1, -49.4)
( 10.0, -14.1, -49.4)
( 5.0, -7.0, -56.5)
( 5.0, -3.5, -45.9)
Projection transform

```csharp
Matrix.CreatePerspectiveFieldOfView(
    MathHelper.ToRadians(60.0f), 1.0f, 1, 1000);
```

Projection

<table>
<thead>
<tr>
<th></th>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>Right</td>
<td>1.732</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>Up</td>
<td>0.000</td>
<td>1.732</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>Backward</td>
<td>0.000</td>
<td>0.000</td>
<td>-1.001</td>
<td>-1.000</td>
</tr>
<tr>
<td>Translation</td>
<td>0.000</td>
<td>0.000</td>
<td>-1.001</td>
<td>0.000</td>
</tr>
</tbody>
</table>

```csharp
pyramidPC_i = Vector3.Transform(pyramidEC[i], Projection);
```

```
( 0.0, -24.4, 48.5)
( 17.3, -24.4, 48.5)
( 8.6, -12.2, 55.6)
( 8.6, -6.1, 45.0)
```

The projection coordinate has not removed z values, it has scaled and clipped to the viewing frustum.

Depth sorting occurs after lighting in the "fragment" shading passes.
Movement

Consider a movable 6DF object as a "Rigid body" with rotation and translations forces that affect its movement.

Position ( @ ) point, object's center ("anchor") location

Target (vector in forward direction ) length is distance of move

Rotation – sets "local" or "frame" axis: up, right, backward),

Steering forces:
  rotation radians about an (arbitrary) axis
  translation vector (length = magnitude of force)

"Big Bang" movement  next  \( \leftarrow \) turn + forward step

0. Object saves its current position.
1. Object "moves to origin" keeping its current rotation submatrix
2. Object makes any rotations to set its current orientation
3. Object determines its next position  \( \leftarrow \) current + step * Forward
4. Test for collisions or any other position based constraints
5. if next position OK, translate to next position
   else don't move, translate to saved position
public void updateMovableObject() {
    Vector3 startLocation, stopLocation;
    startLocation = stopLocation = Translation; // current pos
    // move to origin, rotations about center
    Orientation *= Matrix.CreateTranslation(-1 * Translation);
    Orientation *= Matrix.CreateRotationY(yaw); // rotate Y
    Orientation *= Matrix.CreateRotationX(pitch); // rotate X
    Orientation *= Matrix.CreateRotationZ(roll); // rotate Z
    // move forward & test for off terrain or collision
    stopLocation += ((step * stepSize) * Forward);
    // no collision possible if moving off terrain
    if (stage.withinRange(this.Name, stopLocation)) // terrain
        if (model.IsCollidable && collision(stopLocation))
            // collision restore start position, keep rotation
            Orientation *= Matrix.CreateTranslation(startLocation);
        else // no collision and on surface, move forward
            Orientation *= Matrix.CreateTranslation(stopLocation);
    else // off terrain, restore position, keep rotation
        Orientation *= Matrix.CreateTranslation(startLocation);
}
Agent views

First Person view
camera location = avatar Location + vertical offset + direction offset
camera orientation is same as avatar's

Follow view
camera location = avatar Location + vertical offset - direction offset
optionally camera orientation rotates negatively on Right

Bird’s eye view
camera Location = avatar Location + vertical offset
camera Orientation is rotated 1.57 radians to look down at avatar
Camera's updateViewMatrix()

```java
public void updateViewMatrix() {
    // called in Stage's Update(...)
    switch (cameraCase) {
        case CameraEnum.TopDownCamera:
            viewMatrix = Matrix.CreateLookAt(
                new Vector3(terrainCenter, terrainCenter * 2, terrainCenter),
                new Vector3(terrainCenter, 0, terrainCenter),
                new Vector3(0, 0, -1));
            break;
        case CameraEnum.FirstCamera:
            viewMatrix = Matrix.CreateLookAt(agent.Translation,
                agent.Translation + agent.Forward, agent.Orientation.Up);
            viewMatrix *= Matrix.CreateTranslation(0, -offset, 0);
            break;
        case CameraEnum.FollowCamera:
            viewMatrix = Matrix.CreateLookAt(agent.Translation,
                agent.Translation + agent.Forward, agent.Orientation.Up);
            viewMatrix *= Matrix.CreateTranslation(0, -2 * offset, -8 * offset);
            break;
        case CameraEnum.AboveCamera:
            viewMatrix = Matrix.CreateLookAt(
                new Vector3(agent.Translation.X,
                agent.Translation, new Vector3(0, 0, -1));
            break;
    }
}
```
View constraints

In a 6DF environment (spaceship, submarine) the "avatar" views have no constraints.

In a terrain / gravity environment avatar view (walking) is constrained.

limit roll  range \( \pm \frac{\text{MathHelper.Pi}}{3} \) for 60\(^\circ\) "head tilt"

limit pitch range \( \pm \frac{\text{MathHelper.Pi}}{2} \) for 90\(^\circ\) looking down

\( \pm \frac{\text{MathHelper.Pi}}{6} \) for 30\(^\circ\) looking up

Object Attachments

Any object can be attached (tethered – oriented – rotated, translated) to another object.

avatars can have properties – weapons, spells, lives
views can be through windows – textured wall with a "hole"
Tours: camera animation

Consider a path (an array of Matrix where each Matrix is a position and orientation (orientation matrix) of "nodes" in the linear "graph".

As a function of time Update(GameTime) can place the current camera at the next "node" in the path.

To smooth animation between nodes intermediate "nodes" can be calculated using Matrix.Lerp(...)

Lerp – linear interperts between two matrices based on an amount (0..1).

```csharp
public static Matrix Lerp ( Matrix matrix1, Matrix matrix2, float amount )

public static Vector3 Lerp (Vector3 vector1, Vector3 vector2, float amount )
```

<< see Tour.cs example in Tour-distro >>
Tour / animation path tool

Design a tour or animation path.

Need: ordered collection of transformation matrices

Matrix [] pathName

Approach:

walk avatar along path and "click" append wayPoints to pathName
have user keyboard event (say 'P') bound to addPath()
addPath() appends avatar's transformation matrix as a waypoint to pathName
have user keyboard event (say 'F') bound to fileWrite("fileName")
fileWrite("fileName") writes pathName to file
file can be used by "game" to initialize a Matrix [] pathName
Update(...) calls updatePath(...) to move object to new waypoint.
Vector2 struct

constructors
Vector2(float)  Vector2(float, float)

properties
Vector2.X  Vector2.Y
Vector2.One  Vector2.Zero  Vector2.UnitX  Vector2.UnitY

methods  // examples
static void Clamp(Vector2 value, Vector2 min, Vector2 max)
static float Distance(Vector2 v1, Vector2 v2)
static float Dot(Vector2 v1, Vector2 v2)
    with unit vectors result is the cosine of shared angle
float Length()
static Vector2 Lerp(Vector2 v1, Vector2 v2, float amount)
    v1 + (v2 - v1) * amount
static Vector2 Negate(Vector2 v)  //  v = v * -1
void Normalize()  Vector2 Normalize(Vector2 v1)
Vector2 Reflect(Vector2 v, Vector2 normal)
Vector2 Transform(Vector2 v1, Matrix m)
Vector3, Vector4, Color

constructors  Vector3(float)  Vector3(float, float, float, float)

properties
   Vector3.Forward, Backward, Up, Down, Left, Right

methods
   static Vector3 Cross(Vector3 v1, Vector3 v2)

constructors  Vector4 (float)  Vector4(float, float, float, float)


constructors  Color(int r, int g, int b)  // 0 to 255
               Color(float r, float g, float b)  // 0.0 to 1.0
               Color(int r, int g, int b, int a)
               Color(float r, float g, float b, float a)
               Color(Vector3)  Color(Vector4)

properties  many color properties …
Terrain following

Assume:

Avatar has moved "forward" onto a surface (aPos)
Its (x, z) values are "good enough" need to adjust height. The height can be interpolated.

Avatar Up has been oriented to surface.Up

// interpolate avatarPosition – assume is in top surface
   // Y difference on X
   // Y difference on Z
aPos.Y = A.Y + xY + zY;

// or, solve plane equation for y 0 = A_x + B_y + C_z + D
// 0 = A + B + C - ( A*pos.X + B*pos.Y + C*pos.Z), or
// aPos = point on plane, N = plane’s normal : D = dot(-N, aPos)
Float A, B, C, D;
A = N.X;  B = N.Y;  C = N.Z;
aPos.Y = ( A + B + C - A * aPos.X - C * aPos.Z) / B;
Dot && Cross revisited

Consider 2 unit vectors, v1 and v2

\[ v1 = (1, 0, 0) \] // red
\[ v2 = (0, 0, 1) \] // blue

Two vectors define a triangular plane
Their cross product is the normal to the plane. // green

If the vectors are co-linear (same or opposite directions) the cross product is 0 – not a valid axis of rotation.

The dot product returns the cosine of the angle between the vectors.

The arcCosine converts to a radian value representing the angle between the vectors.

\[
\text{Vector3.dot}(v1, v2) > 0 \quad \text{angle} < 90° \\
\text{Vector3.dot}(v1, v2) < 0 \quad \text{angle} > 90° \\
\text{Vector3.dot}(v1, v2) == 0 \quad \text{angle} == 90° \\
\text{Vector3.dot}(v1, v2) == 1 \quad \text{angle} == 0° \\
\text{Vector3.dot}(v1, v2) == -1 \quad \text{angle} == 180°
\]
Vector3.Dot(v1, v2)

Consider the angle between two vectors

A (0.00, 0.00, -1.00)  // Forward
B (0.50, 0.00, 0.87)
or 12 B's equally rotated on Y

Note Vector3.Dot(A, B) is valid for 0 to π values when converted
to arcCosine (converts to radian values).

For angles π to 2π you need to adjust by 2π - arcCosine

<table>
<thead>
<tr>
<th>radian</th>
<th>aCos(dotProduct)</th>
<th>dotProduct</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.52</td>
<td>0.52</td>
<td>0.87</td>
</tr>
<tr>
<td>1.05</td>
<td>1.05</td>
<td>0.50</td>
</tr>
<tr>
<td>1.57</td>
<td>1.57</td>
<td>0.00</td>
</tr>
<tr>
<td>2.09</td>
<td>2.09</td>
<td>-0.50</td>
</tr>
<tr>
<td>2.62</td>
<td>2.62</td>
<td>-0.87</td>
</tr>
<tr>
<td>3.14</td>
<td>3.14</td>
<td>-1.00</td>
</tr>
<tr>
<td>3.67</td>
<td>2.62</td>
<td>-0.87</td>
</tr>
<tr>
<td>4.19</td>
<td>2.09</td>
<td>-0.50</td>
</tr>
<tr>
<td>4.71</td>
<td>1.57</td>
<td>0.00</td>
</tr>
<tr>
<td>5.24</td>
<td>1.05</td>
<td>0.50</td>
</tr>
<tr>
<td>5.76</td>
<td>0.52</td>
<td>0.87</td>
</tr>
<tr>
<td>6.28</td>
<td>0.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>
Orienting with guide vector

// Return radian for rotation of unit Vector3 initial to desired
// using reference vector guide.
// Determines positive rotation < Math.PI or
// positive rotation >= Math.PI using guide
// Guide is orthogonal vector positive rotation is towards
float adjustedAcosDotGuide(Vector3 desired, Vector3 initial,
    Vector3 guide) {
    float aCosDot;
    aCosDot = (float) Math.Acos(Vector3.Dot(desired, initial));
    if ( Vector3.Distance(desired, guide) >
        Vector3.Distance(initial, guide))
        return (float) (2*Math.PI - aCosDot);
    else
        return -aCosDot;
}

<table>
<thead>
<tr>
<th>Vector</th>
<th>Rot Axis</th>
<th>Guide</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward</td>
<td>X</td>
<td>Down</td>
</tr>
<tr>
<td>Forward</td>
<td>Y</td>
<td>Left</td>
</tr>
<tr>
<td>Right</td>
<td>Y</td>
<td>Forward</td>
</tr>
<tr>
<td>Right</td>
<td>Z</td>
<td>Up</td>
</tr>
<tr>
<td>Up</td>
<td>X</td>
<td>Forward</td>
</tr>
<tr>
<td>Up</td>
<td>Z</td>
<td>Left</td>
</tr>
</tbody>
</table>
Orienting with cross product

// Use the cross product of initial and desired to get a guide.
// With rotations on "UP", an XZ plane.
// Cross product is positive for rotation 0..Math.PI radians.
// This approach used in AGMGSKv7
// in Object3d.turnToFace(Vector3 target)

float adjustedAcosDotCross(Vector3 desired, Vector3 initial) {
    float aCosDot;
    Vector3 axis;
    aCosDot = (float) Math.Acos(Vector3.Dot(desired, initial));
    axis = Vector3.Cross(desired, initial);
    if (axis.X + axis.Y + axis.Z < 0)
        return (float) (2 * Math.PI - aCosDot);
    else
        return -aCosDot;
}
Orienting Vectors with arctangent

2 vectors (v1, v2) can be oriented using the arctangent if rotate on Y. Here v1 is orient to v2.

double dx = v2.X - v1.X;
double dz = v2.Z - v1.Z;
double arcTan = 0.0;
   // avoid a divide by 0
   if (dx == 0) {  // same or opposite orientations ?
      if (dz > 0) // same orientation
         return 0;
      else if (dz < 0) // opposite orientations
         return Math.PI; // rotate on Y 180°
   }
   else {  // dx isn't 0 so can divide by it
      arcTan = Math.Atan(dz / dx);
      if (dx < 0)
         return 2 * Math.PI - arcTan * 2.0;
      else
         return arcTan * -2.0;
   }
**Orient twice**

When orienting vectors are not in the same plane.

**Orient on 2 dimensions – example below**

1. orient Forwards (blues) into same plane
   rotate on their Cross product
2. orient Ups (greens) – rotate on their cross product

With 3D drawing tool – you can "see", "plan" your operations...
Project vector to plane

Aligning vectors \((v_1, v_2)\) are often not in same plane. Need to project \(v_1\) into same plane as \(v_2\) – to get rotational radian

\(w\) is the projection of vector \(v\) onto a plane with normal \(n\).

\[ w = v - (v \cdot n)n \]

// Project vector onto the plane given plane's normal
// return normalized projected vector.
Vector3 projectVectorToPlane(Vector3 v3, Vector3 pNormal) {
    Vector3 result = new Vector3();
    pNormal.Normalize();
    result = Vector3.Subtract(v3,
                            Vector3.Multiply(pNormal, Vector3.Dot(v3, pNormal)) );
    return Vector3.Normalize(result);
}

See also XNA 4 Plane structure and its members.

Terrain following, solid Obj3d – “a tank”

<table>
<thead>
<tr>
<th>Vertex</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>30</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>60</td>
<td>-10</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>90</td>
<td>-5</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>30</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>60</td>
<td>-20</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>90</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

All vectors are normalized

\[ P_{abc} = V_{AB} + V_{AC} \]

\[ R_{abc} = \text{Cross}(N_{abc}, P_{abc}) \]

Assume obj3d is on plane \( \text{abc} \) but its \( N_{\text{obj3d}} \) (Up) is not oriented correctly to “orient” obj3d into plane \( \text{abc} \)

rotate \( N_{\text{obj3d}} \) \( \text{aDotCos}(N_{\text{obj3d}}, N_{\text{abc}}) \) radians on axis \( R_{abc} \).