Programming for Artists and Designers

Tutorial 7: Vectors and Using them with a HUD

Copy Right:
This tutorial uses ideas, code and images from the following people and places:
Tarquin @ The unreal wiki
Jeff Giles @ Sega 500
NASA @ liftoff NASA

Aims:
This tutorial aims to introduce you to concept of vectors and rotators and to get you programming a more advanced HUD.

Introduction:
Vectors are an important part of games life, so you better get used to them. They are used for many things in games and 3D graphics.
This is a brief introduction only, it is however important to have a basic grasp of trigonometry.
Check out this rant about lack of trig knowledge from some very good modders: http://wiki.beyondunreal.com/wiki/Tarquin/Trig_Rant
See the links document @ activehelix for links to sites on vectors and rotators.

Note:
You may hear people talking about scalars, so here is a definition:
1: A real number rather than a vector
2: A quantity (as mass or time) that has a magnitude describable by a real number and no direction

Scalar is usually used to describe numbers when talking about vectors.

Vectors
Dictionary Definition: A quantity that has magnitude and direction and that is commonly represented by a directed line segment whose length represents the magnitude and whose orientation in space represents the direction; *broadly:* an element of a vector space

Vectors are represented by a single symbol:
In longhand, this is underlined: v
In print bold: **v**
In coding, it's up to the coder to remember which is what type.

Generally a vector can also have two, three, four or even more components. But we are dealing with UnrealScript, where Vectors always consist of three components.

Describing Vectors
Vectors are usually described using symbols and the coordinates that everyone should be familiar with.

On a two-dimensional plane, any point (x,y) is a vector. Graphically, we often represent such a vector by drawing an arrow from the origin to the point, with the tip of the arrow resting at the point.
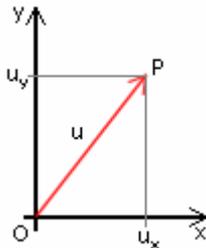
The 2d vector u with its components ux and uy

The situation for three-dimensional vectors is similar, with an ordered triplet (x,y,z) being represented by an arrow from the origin to the corresponding point in three-dimensional space. In vector language, this is described as (ux , uy , uz) to make it clear that the three numbers belong to the vector u. This looks exactly the same as the coordinates for a point. The only difference is the way we think about them.

Vectors are extremely powerful. They can represent:

Points – as seen in the above diagram
If P is a point in space, OP or in longhand with an arrow above, is the vector from O, the origin to P. Adding or subtracting these gives vectors between points.

Displacements
You could say, "the pawn has moved 64 units", or you could say "the pawn has moved 64 units to the north-east", or you could use a vector of length 64 that points in that direction. A displacement is different from a length in that it has a direction as well as a magnitude. See length of a vector below.

Velocities and Accelerations
Like displacements, these have a direction. The magnitude of the vector now represents "how much" velocity there is; in other words, the speed. To return to the earlier car example, if you say, "the car has a speed of 10 m/s", then you're giving the magnitude of the velocity, but not the direction.

Forces
When specifying an impulse on a Karma simulated actor, you can use a vector to represent the direction and magnitude of the force to be applied to that actor.

Representing Vectors as Coordinates
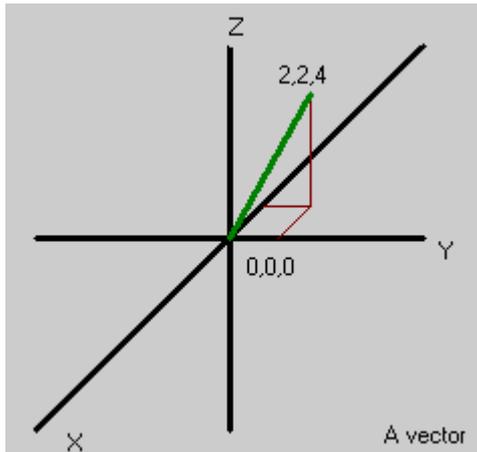
- X is positive to the right
- Y is positive going up
- Z is positive disappearing into the screen

Representing Vectors in Unreal
This may leave you wondering, if vectors represent so much how does unreal know what type of vector I am using. Well unreal doesn't know if you are specifying a position, force or velocity vector. As with many programming concepts association in purely in the programmers mind.
Unreal knows only that you have 3 float values grouped together in one structure.
The structure is found in the object file and is defined as:

```
struct Vector
{
    var() config float X, Y, Z;
};
```

float VSize (vector A) [static]
Returns the length of the vector, ||A||.
vector Normal (vector A) [static]
Returns a vector with the same direction and a length of 1.
Invert (out vector X, out vector Y, out vector Z)
vector VRand ( ) [static]
Returns a vector with random direction and a length of 1.
vector MirrorVectorByNormal (vector Vect, vector Normal) [static]
Mirrors a vector about a specified normal vector.
Think of a projectile p with a certain speed vector that hits a wall with a certain normal vector HitNormal and gets reflected without any damping. The projectile's new speed is what you get from MirrorVectorByNormal(p.Velocity, HitNormal). This can also be calculated through:
Vect - (Vect dot Normal(HitNormal)) * Normal(HitNormal) (this is used e.g. by the Ripper blades)

Rotators

Rotators are very similar to vectors. They are again defined in the object fie and even the struct they are defined in looks very similar to the vector struct:

```
struct Rotator
  {
     var() config int Pitch, Yaw, Roll;
  };
```

OK, so what is pitch yaw and roll well we I will show you with the aid of these handy NASA diagrams (see
http://liftoff.msfc.nasa.gov/academy/rocket_sci/shuttle/attitude/pyr.html)
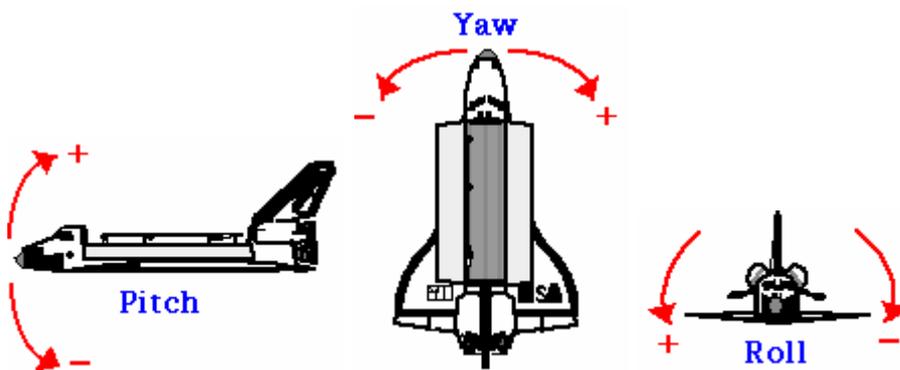
Pitch = up and down – Rotation about the X axis
Yaw = side to side – Rotation about the Y axis
Roll = tilting either left or right – Rotation about the Z axis

Each direction is divided in to 65535 Unreal Units (UU) rather than 360 degrees.
Every Actor has a rotator that defines where they are facing.
They have an automatic conversion ability against vectors, you can cast a rotator to a vector and vice versa. So they can be converted in vector and multiplied against scalar values. Therefore any math that can be performed on vectors can be performed on rotators.
This is because a vector has direction it can store the facing of an object, a rotator doesn't necessarily have direction until it is used in conjunction with a vector.

The following rotator function are also defined in the object.uc file:
GetAxes (rotator A, out vector X, out vector Y, out vector Z) [static]
Returns the three vectors X, Y and Z, where X is forward direction, Y points right and Z points upwards. (relative to the rotation expressed by A)

rotator RotRand (optional bool bRoll) [static]
Returns a random rotator. If bRoll is False the Roll component of the rotator is 0.
rotator OrthoRotation (vector X, vector Y, vector Z) [static]

Returns a rotator from three orthogonal vectors.
rotator Normalize (rotator Rot) [static]
Returns the corresponding rotator with components between -32768 and 32768.

Quaternions

This is a very basic what a quaternion is.
When we store rotators as vectors they don't have a roll. When rotators get cast to vectors they loose their roll component, when they get cast back they gain a roll of 0. A quaternion is basically a vector with an additional roll component. Useful for avoiding gimbal-lock…this is the loss of roll when you are looking directly up or down.

Putting this into Practice

Let's code a HUD that will modify the game a lot. We will add radar. This is uses code provided and copyrighted under terms found on the sega500 website by Jeff Giles - http://gamestudies.cdis.org/~jgiles/
His full sample code and images can be downloaded from there, week 11.
The radar will use the idea of vectors, after we need to place icons on the map that represent other players/bots.
I have added some comments throughout to provide further insights into Jeff Giles code.

```
//=======================================================
// cvgHUD – HUD for our game. Has radar.
//=======================================================

class cvgHUD extends HudBTeamDeathMatch;

//direct import of textures from file
#exec texture IMPORT name=RADAR FILE=Textures\radar.bmp
GROUP="HUD" MIPS=OFF FLAGS=2
#exec texture IMPORT name=BLIP FILE=Textures\blip.bmp
GROUP="HUD" MIPS=OFF FLAGS=2
```

We also need to define some variables for our class, so the next line is:

```
var texture radartex,bliptex,tex;
var int radarTL_X, radarTL_Y;
var float radarScale, range;
```

The next stage is to draw the HUD:

```
function DrawHUD(Canvas c)
{
    local color tempColor;
    local byte tempstyle;

    tempColor = c.DrawColor; //storing the values
```

```
    tempstyle= c.Style;

    radarTL_X=c.ClipX/2 - radartex.USize/2*radarscale;
    radarTL_Y=c.ClipY - radartex.VSize;

    super.DrawHUD(c);
    c.DrawColor = WhiteColor;//restore values
    c.Style = ERenderStyle.STY_Translucent;

    DrawRadar(c);
    DrawBlips(c);

    c.DrawColor = tempColor;//restore values
    c.Style = tempstyle;
}
```

After Drawing the HUD we define our functions

This function draws the Radar image to the canvas (screen)

```
function DrawRadar(Canvas c)
{
    c.SetPos(radarTL_X, radarTL_Y);
    c.DrawTile(radartex, radartex.USize*radarscale,
radartex.VSize*radarscale,
              0,0,radartex.USize, radartex.VSize);

}
```

The next user defined function makes used of the following actor method:
UT2003 :: Object >> Actor (Methods)

With the following function we need to iterate through all actors and draw there position on the radar. Luckily for us unreal script provides an easy mechanism for us to do that:

- UnrealScript's "foreach" command makes it easy to deal with large groups of actors, for example all of the actors in a level, or all of the actors within a certain distance of another actor. "foreach" works in conjunction with a special kind of function called an "iterator" function whose purpose is to iterate through a list of actors

This is essentially a 'for' loop that iterates over a set of data. This set can be anything. It is not like a for loop that must increment and test a condition. This is simply iterating over a group of objects.

- foreach AllActors(class 'Actor', A)

The first parameter in all "foreach" commands is a constant class, which specifies what kinds of actors to search. You can use this to limit the search to, for example, all Pawns in the level exclusively.

There are a fair number of iterator methods to choose from. 11 if you count the one in the object class. We are concerned with the following iterator fnction:

RadiusActors(class<Actor> BaseClass, out Actor Actor, float Radius, optional vector Loc) [native, final, iterator]
- Returns all actors within a give radius. Iterates through all actors within a specified radius of the specified location (or if none is specified, this actor's location).

- Without that optional last parameter, this call is limited to a specific position – generally the pawns starting parameter

Slow like AllActors(). Use CollidingActors() or VisibleCollidingActors() instead if desired actor types are visible (not bHidden) and in the collision hash (bCollideActors is true).

I store the magnitude of this vector. It's the distance to the current pawn. Why divided by 8? - 1024/8 = 128. 128 = The radius of the bitmap.

Next we create a rotator based on this vector. This functionally gives us it's direction, relative to us.

Our view is *off* by 90 degrees. This correction will help us re-align the 'points' as we rotate them. So we need to make a 90` degree correction and adjust the new rotation

Then we convert the Rotator back to a vector and do a scalar multiplication on this vector by the stored magnitude. We now have a translated point (something we can display on-screen).

```
// Draw the blips on the radar screen
function DrawBlips(Canvas c)
{
    local Pawn testpawn;
    local rotator newRotation, correction;
    local vector newPosition;
    local float magnitude;

    foreach RadiusActors(class'Pawn', testpawn, range,
PawnOwner.Location)
    {
        if(testpawn != pawnOwner)  //don't check against the
owner
        {
            magnitude = vsize(testpawn.Location-
PawnOwner.Location)/8;
            newRotation =rotator(testpawn.Location-
PawnOwner.Location);
            correction.yaw = 65535/4;
```

```
            newRotation.yaw -= (PawnOwner.Rotation.yaw
+correction.yaw);

            newPosition=vector(newRotation)*magnitude;
            //set the draw position & scale it
            c.SetPos(radarscale*(newPosition.x +
radartex.USize/2) + radarTL_X,
                    radarscale*(newPosition.y +
radartex.VSize/2) + radarTL_Y );
            //place the blip on the radar
            c.DrawIcon(bliptex,radarScale);
        }
    }
}
```

With the following function we are setting the draw position of blips on the
radar. All we need to do is set our position with each iteration and move the
draw position down some.

```
function DrawTargetRectical (Canvas c)
{
   local vector x,y,z, direction;
   local float distance;
   local Pawn targetP;
   local int XPos, YPos;

   GetAxes(pawnowner.Rotation,x,y,z);
   foreach
RadiusActors(class'pawn',targetP,range,PawnOwner.Location)
   {
        direction= targetP.Location-PawnOwner.Location;
        distance = VSize(direction);
        //direction= vector(Normalize( rotator(direction)));
        direction= direction/distance;

        if(direction dot x > 0.7)  // are they in front of me
        {
            Xpos = c.ClipX/2 * (1+1.4*(direction dot Y));
            Ypos = c.ClipY/2 * (1-1.4*(direction dot Z));

            c.DrawColor=redcolor;
            c.SetPos( Xpos-16, Ypos-16);
            c.DrawIcon(target, 1);
        }
    }
}
```

Our default properties just set the texture variables and drawscales for them.

```
defaultproperties
```

```
{
    radartex=Texture'RADAR'
    bliptex=Texture'BLIP'
    target=Texture'Crosshairs.HUD.Crosshair_Circle2'
    radarScale=0.75
    range=2048
}
```

```
    radartex=Texture'RADAR'
    bliptex=Texture'BLIP'
    target=Texture'Crosshairs.HUD.Crosshair_Circle2'
    radarScale=0.75
    range=2048
```