dis-tutorial

Coordinate Systems

DIS is transmitting state information about entities in the world, and the most important state information is usually where the entities are and which direction they're pointing-their position in the world. This brings up an obvious but important question: what coordinate system should we use to describe an entity's location? It's a trickier question than one might think at first. There's a simple solution that can be used in games, but that doesn't work well with real world position data. There's a much more complex answer that can be used to describe the position of items in the real world, but even that solution has a series of problems.

Game Coordinate Systems

The technology that games are similar to what the military simulation applications use, and the solution games apply to the coordinate system problem can illustrate a few things. First person like Call of Duty, World of Warcraft or Skyrim move entities in 3D worlds, just as military simulations do. The games use a 3D graphics library to position and render entities, perhaps OpenGL, Direct3D, Web3D, or the higher level higher game engine graphics such as OpenSimulator, Lumberyard, or Unreal. The graphics package has to use a master coordinate system to position entities and then draw them. Figure x is a screen capture from the Blender 3D tool that shows the simplest possible situation: a cube that's offset a few units from the universal coordinate system's origin. The overall coordinate system origin is at the intersection of the blue and red lines, and the cube has an additional, local coordinate system set at its center.

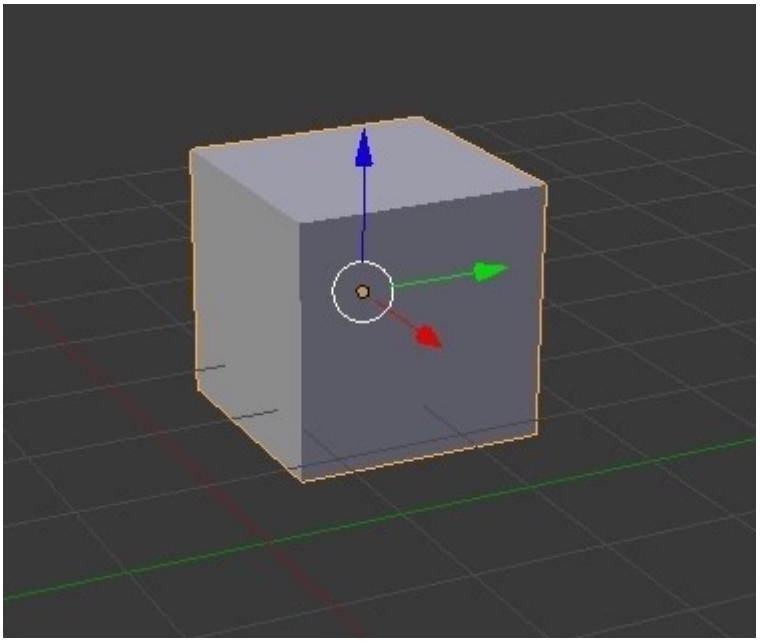


Figure x

At this point the “what coordinate system?” question seems to have a simple answer. Simplicity is a virtue in programming, so we’ll describe the game world using a coordinate system that assumes the world is infinitely flat in all directions. World of Warcraft can lay out the villages and roads used in their world on a surface that’s flat, and we’ll pick a Cartesian master coordinate system that can be used to describe the position of everything. We’ll pick a place to put the origin, and then we can describe the position of any feature in the world fairly directly. This makes the math involved not so bad.

We can also use other coordinate systems that are helpful for a player. A game engine used by a player viewing the game from a street in the village of Goldshire might find it helpful for there to be a local coordinate system with a nearby origin. See figure x.

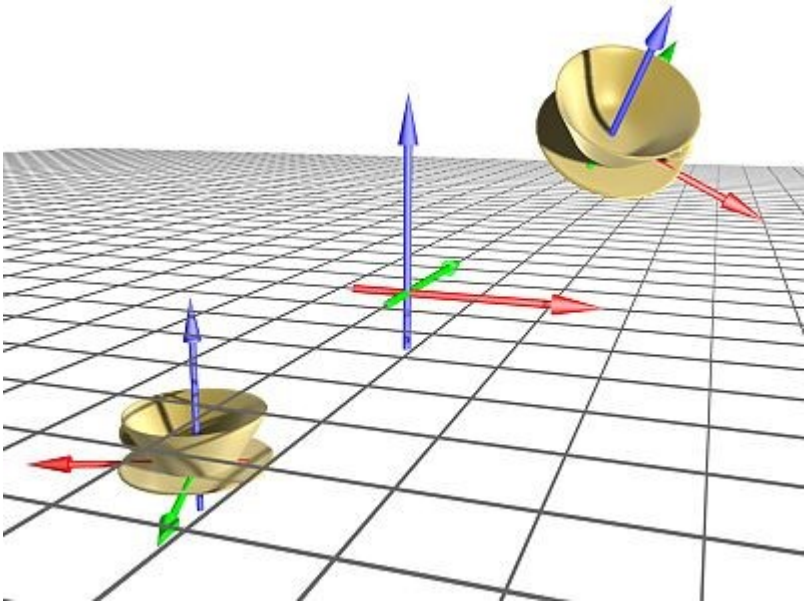


Figure x

There’s a master, global coordinate system, but we can also set up other coordinate systems because this usually makes various programming tasks simpler. There’s some math involved to express a new coordinate system in terms of the master coordinate system, but it’s not all that bad. If you didn’t sleep through the linear algebra class you can probably pull it off.

Real World Coordinate Systems

We have a less pleasant situation when we’re trying to describe the position of entities in the real world.

Entities are positioned on the earth’s ellipsoid-shaped surface, and that’s even worse than using a perfectly round sphere. If we’re trying to realistically model naval warfare then we won’t be able to see a ship 100 km away from our ship because it’s below the horizon. Using a single, unified, flat coordinate system would not be realistic and if the exercise is conducted across a large geographic scale it wouldn’t provide good training. The gamer’s admirably simple approach of using a flat coordinate system for a flat world doesn’t work because the real world isn’t flat.

There are other problems. In an LVC environment, we’re trying to pull together information

from multiple sources. The location of buildings or roads is provided by maps, and the maps are often using latitude and longitude to describe their positions. These days there are petabytes of open source information on the web that's georeferenced, from coffee shop locations and roads to tribal cultural information. The positions are also usually described in latitude and longitude. The Army often uses Military Grid Reference System (MGRS) to describe the position of units. And, as mentioned above, we've got a game graphics package which we're also using to describe the position of entities, and the graphics system coordinate system is Cartesian. We may also be trying to describe the orbit of a satellite, or the relative positions of five maneuvering aircraft.

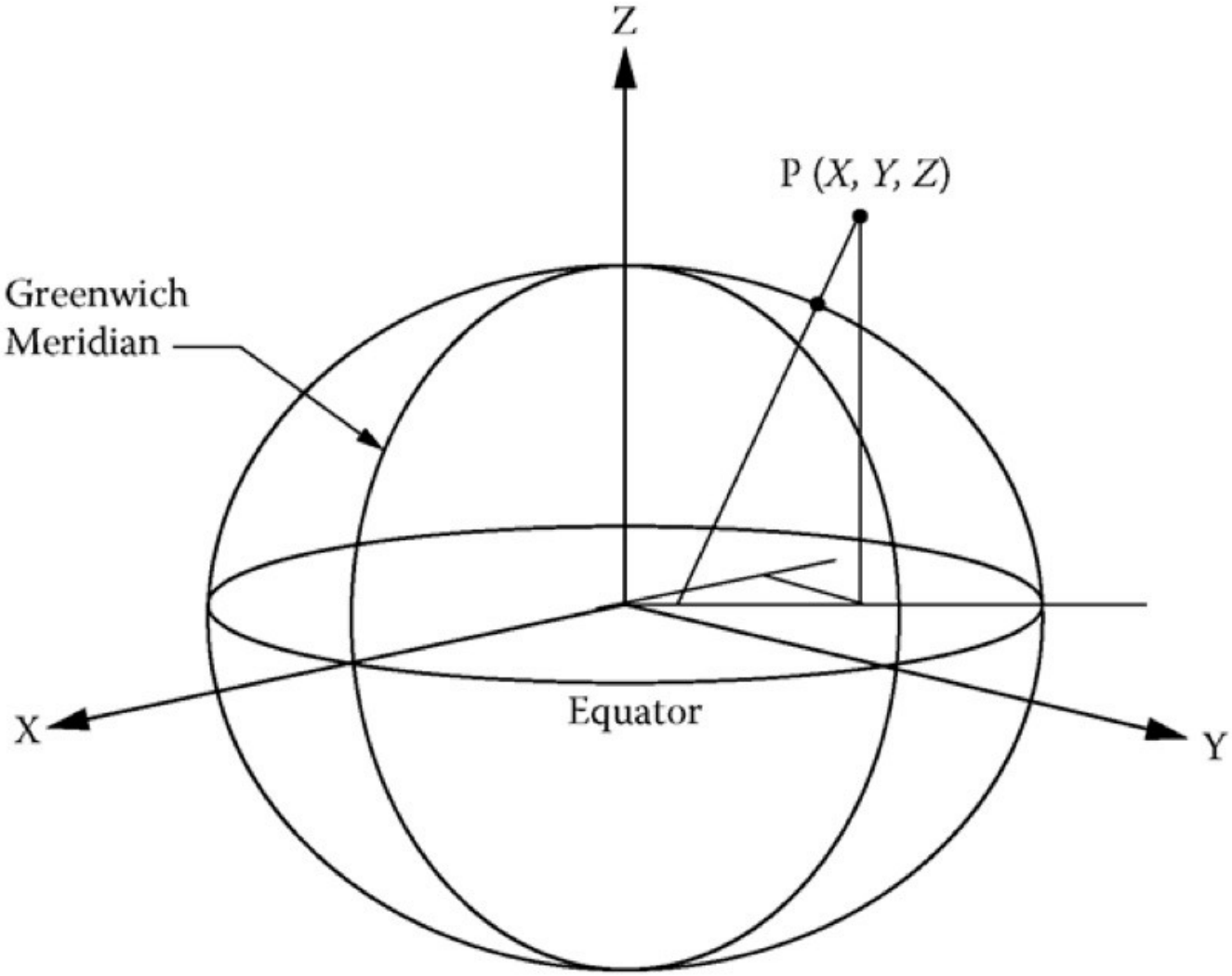
Imagine a virtual vehicle driving through a town. The vehicle is being rendered in a game engine that's using a conventional Cartestian coordinate system that is flat and rectilinear. We're using the web as a source of position data for real items in the world, and as anyone who has used Google Maps can attest those positions are almost always described in terms of latitude, longitude, and altitude. We want to be able to look up while inside the game and see the position of a constructively simulated satellite overhead, and the position of that satellite's orbit may be described using Keplerian orbital elements. We also want to arrive at a destination that's specified in MGRS. At the same time we are shot at by a simulated artillery fire system that's 30 KM away, so the curvature of the earth can't be ignored.

We're trying to integrate positions of entities that are described by multiple coordinate systems and the organizations that describe the position of things in the world don't know or care what our simulation is using as a coordinate system. The system World of Warcraft uses–a flat, rectilinear coordinate system for their entire game world–falls apart when it is integrated with data that comes from an ellipsoid-shaped world. (At least if you're not a member of the Flat Earth Society.)

So what coordinate system should be used by simulations to integrate all these data sources, while also making game graphics workable?

DIS is used in many domains, including sea, subsurface, air, land, and space. If the simulation's geographic extent is large enough the curvature of the earth can't be ignored. A simulation limited to land operations might choose MGRS, but this does not work well for aircraft simulations, where we usually want to do physics calculations in a rectilinear system. Naval operations might choose latitude/longitude, but this does not work well for air operations or space operations. We should also settle on either metric or English units. These goals are in conflict with each other, and some tradeoffs have to be made.

DIS chose to use a Cartesian coordinate system with its origin at the center of the earth, and to use meters as the unit of measurement. The X-axis of this coordinate system points out from the center of the earth and intersects the surface of the earth at the equator and prime meridian. The Y-axis likewise intersects the earth's surface at the equator, but at 90 degrees east longitude. The Z-axis points up through the north pole. This coordinate system rotates with the earth; it is sometimes called "Earth-Centered, Earth-Fixed" (ECEF).



This gives us a rectilinear coordinate system with an origin at the center of the earth. With some math (described later) we can now position an entity on the surface of the earth. The World of Warcraft solution is admirably simple but not realistic, while the DIS coordinate system and description of the world is more complex, but more useful.

It seems like an odd choice at first glance. The geocentric coordinate system is, in isolation, not very convenient. Suppose we want to move an entity on the surface of the earth one meter northwest. How much do we change the values of the X, Y, and Z axes in the geocentric coordinate system to accomplish this in our simulation? That's not intuitively obvious. The key caveat is that the position of entities described in this DIS coordinate system are only needed when a state update is *sent on the network*. Our simulation can use whatever coordinate system it likes internally, but must transmit its entity positions to other simulations using the geocentric coordinate system. What we need is a way to convert to and from our internal coordinate system to the global geocentric coordinate system.

The advantage of using a geocentric coordinate system is that, with some math, we can convert it to and from other popular coordinate systems. There are equations to convert from a position described with latitude, longitude, and altitude to the geocentric coordinate system and back again. The same for the DIS coordinate system and MGRS. The approach DIS uses to to allow simulations to use any coordinate system they like internally–whatever that simulation

feels is convenient–but when transmitting state information on the network, convert from that coordinate system to the DIS geocentric coordinate system. Likewise, when we receive a state update that describes the location of an entity, we have to convert from the geocentric coordinate system to whatever coordinate system we use internally. What is needed to pull this off is the math to convert between the coordinate system of convenience and the DIS geocentric coordinate system.

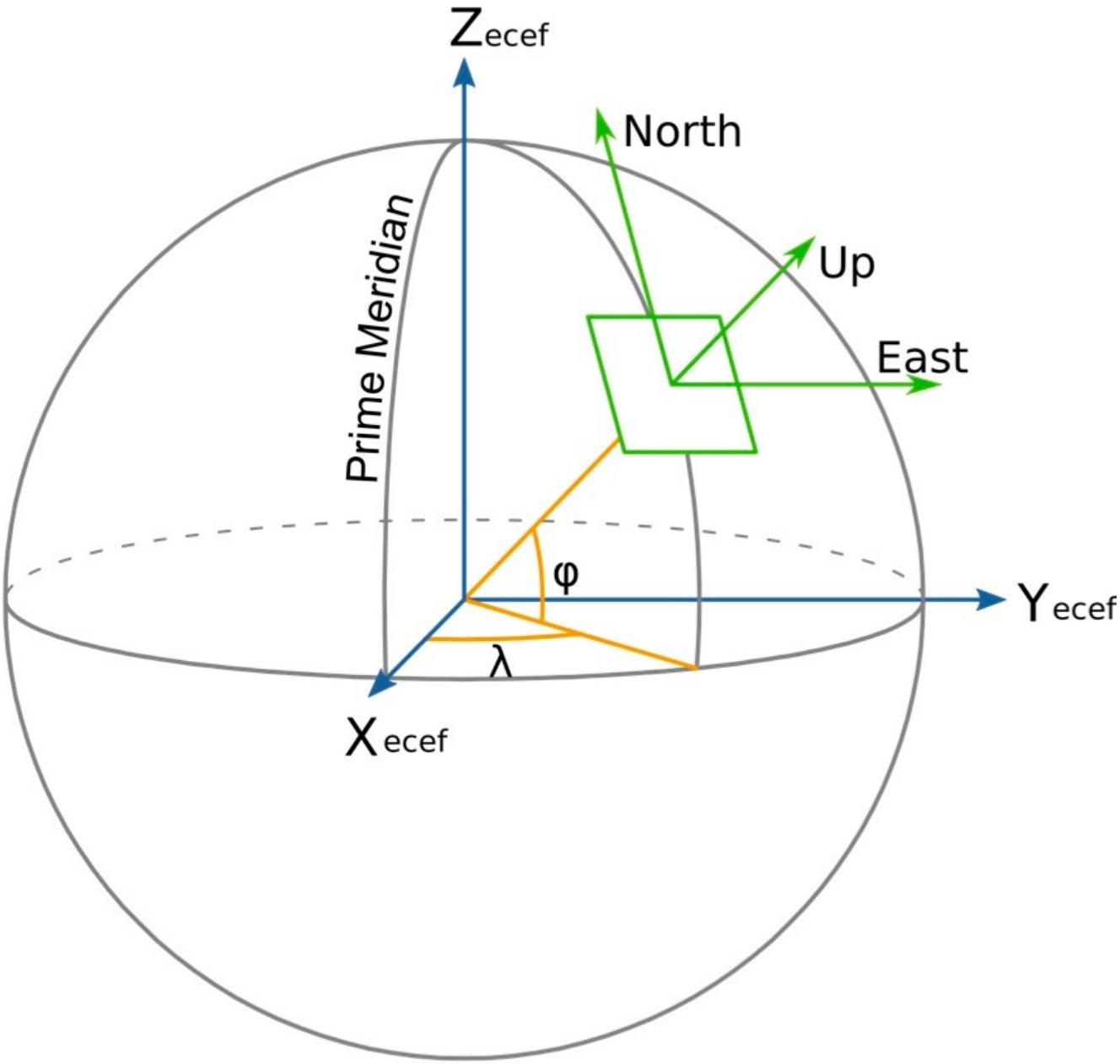
Datums

This is where it starts to get really tricky, at least for me. The geocentric coordinate system is OK, but where is the surface of the earth? The geocentric coordinate system by itself doesn't describe that. We also need a mathematical model for what the shape of the earth is. Modeling the earth as a perfectly round sphere will run into problems.

Position

Most simulations use a local, rectilinear coordinate system for physics and for drawing nearby entities, very similar to what World of Warcraft does. They describe and draw the position of nearby entities using a flat coordinate system. Then, before sending the position of the entity to the network, the simulation converts it from the local coordinate system to the global, geocentric coordinate system. The math to do these operations is well-understood and efficient.

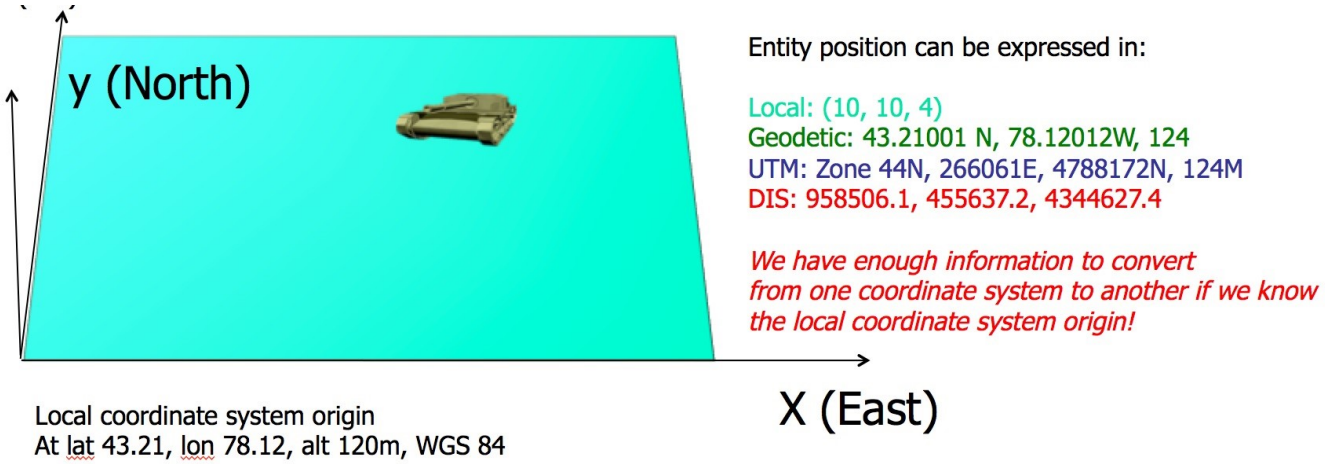
For example, a simulation might find it convenient to set up a local, flat, rectilinear coordinate system with its origin at a given latitude, longitude, and altitude, tangent to the surface of the earth.



This coordinate system is rectilinear and doesn't take into account the curvature of the earth, but for most simulation purposes it works when entities are within a few kilometers of each other. More importantly, for the most part it's mathematically tractable, and easy to work with in the context of most graphics and physics packages. We can make the local simulation's coordinate system co-extensive with the graphics package coordinate system. If we're using a 3D graphics system like Unity or X3D we can make the graphics system coordinate system match that of the tangent plane we set up. We can easily move an entity one meter along the X-axis in the local coordinate system. When we describe the position of the entity to other simulations by sending a Protocol Data Unit (PDU), called the Entity State PDU (ESPDU), we convert the position of the entity from the local coordinate system to the global, geocentric coordinate system.

Many simulations use a East, North, Up (ENU) mapping convention for the local coordinate system axes, with east along the X-axis, north along the Y-axis, and Z pointing up from the surface of the earth. There's not much agreement on which way the coordinate axes point,

though. Another popular convention is NED, north, east, down for X, Y, and Z. Aircraft often use a local coordinate system that puts the origin at the CG of the aircraft, with the X-axis pointing out the nose, the Y-axis out the right wing, and the Z-axis pointing down. Then the position and orientation of the aircraft are described in the context of another local coordinate system. The position of a weapon on the wing of the aircraft needs to undergo several conversions, from the aircraft-centric coordinate system, to the local coordinate system that describes the position of the aircraft, and from there to the geocentric coordinate system. These conventions can be accommodated with enough math.



In this example, the position of a tank entity is described in several different coordinate systems. In the local coordinate system–the coordinate system used for most physics and graphics–it's at (10, 10, 4), and that local coordinate system has its origin at latitude 43.21, longitude 78.12, at an altitude 120 meters above the geoid described by WGS-84. In geodetic coordinates the tank is at latitude 43.21, longitude 78.12 (plus a little for both, to reflect the offset from the origin) and altitude 124. In UTM it's zone 44N, 266061E, 44788172N, 124m. In DIS coordinates it's at (958506, 455637, 4344627). Each of the positions describes the same point in space using different coordinate systems, and we can (with enough math) translate between them.

DIS simulations usually do all their local physics calculations and graphics displays in a local coordinate system which has been picked for the programmer's convenience. When the ESPDU is being prepared to be sent, the position of the entity in the local coordinate system is transformed to the DIS global coordinate system, and then set in the ESPDU. When received by the simulation on the other side, that simulation translates from the global coordinate system to whatever its own local coordinate system is.

Coordinate System Standards Problems

There are a few wrinkles in this. While the geocentric coordinate system origin is placed at the center of the earth, the coordinate system does not by itself define where the surface of the earth is. The earth is not a sphere, but rather a somewhat flattened egg-shaped surface. There are several mathematical models, called "datums," used to describe what the shape of the earth is, usually in the form of an oblate spheriod. Today the most popular of these is called

WGS-84, which is also the model for the shape of the earth used in GPS. It's not exact; the real world's mean sea level can differ from the geoid defined by WGS-84 by 0-5 meters. That may not sound like much when working on a planetary scale, but simulations that are modeling kinetic weapons require high precision for entity locations. A shot by a cannon that's off by two meters may be a clear miss.

Some maps use datums other than WGS-84 for the shape of the earth. The Australians use a datum called GDA-94 on their maps because that works well for modeling the layout of land in Australia, and that replaced an earlier datum called AGD-84. Historically the Japanese and Indians have also used their own datums for mapping, as have many nations, and even states within the US. The datums they chose often predate the creation of WGS-84 and GPS, and were picked because they tend to work well for the region of the earth that the map describes. Datums have been argued about for centuries, and after many decades of map publishing it's quite easy to come across a map that does not use WGS-84. Even though the US government has a National Geospatial Intelligence Agency responsible for mapping, the fact is that LVC simulations often use maps or geo-referenced entities from many non-government sources that use different map datums.

The problem is that using different datums also causes the models for the shape of the earth to differ. Imagine a single entity that has its position described as being 36.5973° N, 121.8731° W. The latitude and longitude lines are fixed to the surface of the earth model, and two maps are using different datums. See what can result in figure X. The datums describe two different 3D surfaces. An entity described with the same latitude and longitude will be in two different locations in 3D space. Their locations depend on the datum the map uses in addition to the reported latitude and longitude.

Figure X. (two curves, describe a surface. same lat/lon, different 3D positions.)

How big of a deal is this? Some of the datums described above could result in a difference of up to 200 m from WGS-84. If you place a map that uses one of those datums into an environment that assumes WGS-84 you'll see terrain feature discrepancies even though latitude and longitude for the feature are identical on the two maps. There may be differences of hundreds of meters. This is illustrated in the figure below. The position of the Texas capitol building is described using identical latitude and longitude on all maps, but because the model for the shape of the earth differs between the maps, the position of the capitol building also differs.

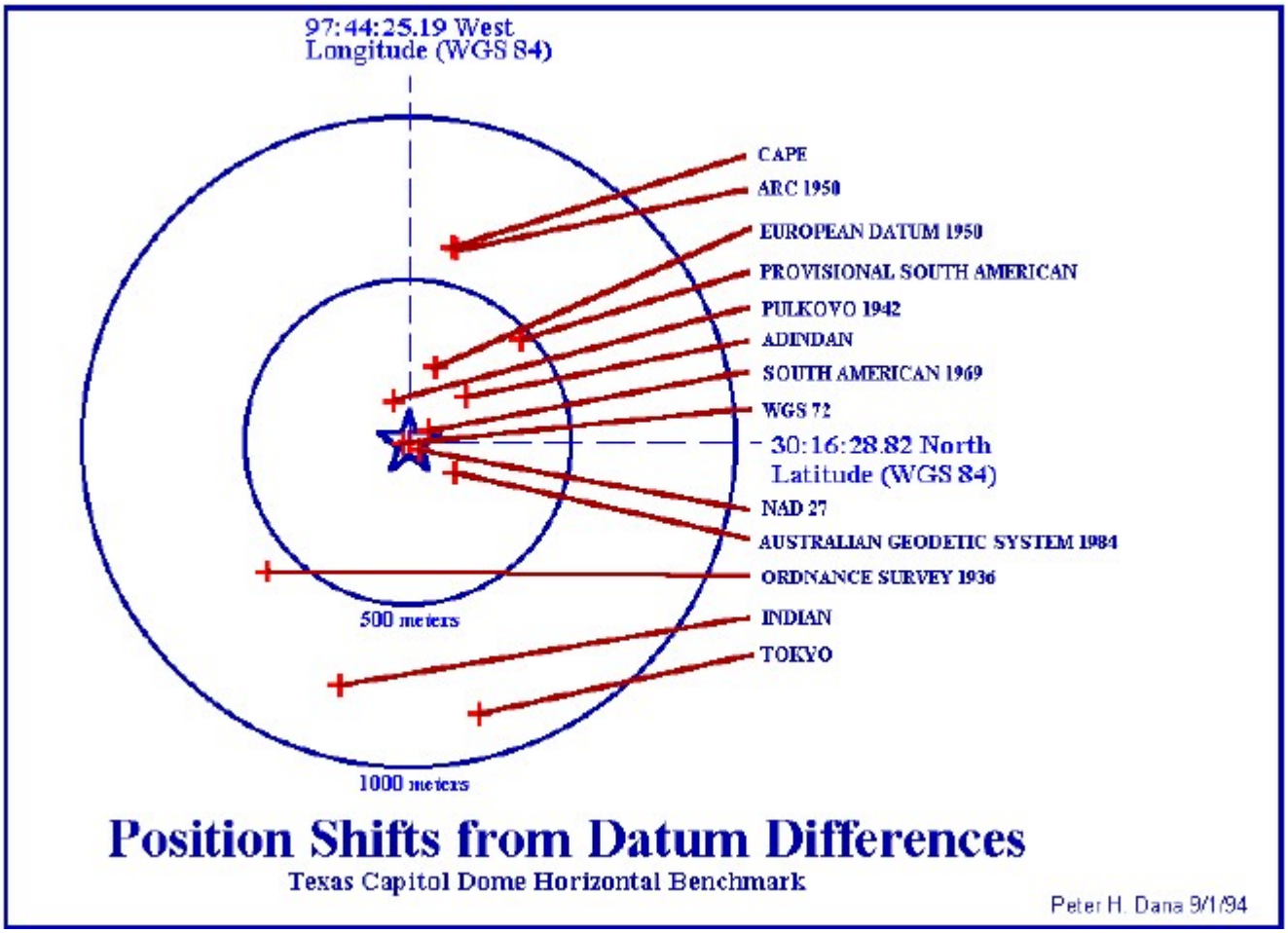


Figure x

A particular map's datam mismatch with other maps may manifest itself in LVC simulations when a convoy is shown on a map driving near a road, but with a 20 meter offset. The position of the vehicles is a live component, reported using GPS, which uses WGS-84. The map the entities are displayed on uses a different datum. When GPS reports latitude and longitude it winds up being in a different 3D location than the map's concept of the road's location.

There are also some computational issues. When using units of meters, geocentric coordinate system values can be over six million meters, and this can cause some numeric precision problems if using single precision floating point numbers. Doing the math required to convert between coordinate systems discussed below can result in computational roundoff errors. Stick to *double* precision.

Terrain

The earth is not smooth, and terrain can rise above or below the geoid i.e., Mount Everest, the Dead Sea, or the bottom of the Atlantic Ocean often do.

Terrain is a tricky problem in itself and outside the scope (for now) of this document. Simulations need precise placement of objects, often to sub-meter accuracy. Getting agreement on this between simulations that use terrain information from different sources is very difficult. Most simulations hack this lack of accuracy by using *ground clamping*. If an entity

such as a tank is described by a companion simulation as being a meter above the ground on the local simulation, the local simulation will simply force it to be drawn as in contact with the ground. This avoids the problem of “hover tanks” that appear to float above the terrain, an artifact that would undermine user confidence in the simulation. But, this also means that the position of the entity differs from what the simulation that owns the entity is describing.

Prefab Packages

There are several packages that convert between the coordinate systems discussed above– geocentric, geodetic, MGRS, and the local coordinate system attached to a maneuvering F-16’s center of gravity. One popular package is the SEDRIS SRM package.

Sedris SRM site

The SEDRIS site includes tutorials about the theory behind the process and also tutorials about using the Java and C++ packages they provide. If you’re doing serious work with position, orientation, and velocity, then a prefab package that handles the conversions is highly recommended. You probably shouldn’t trust the math I discuss below all that much.

Coordinate System Transformations: Latitude and Longitude to Geocentric

Especially in naval simulations it’s popular to describe the position of entities using latitude, longitude, and altitude. We need to convert between latitude and longitude and the geocentric coordinate system.

Shut up and give me the equation

To convert latitude, longitude, and altitude to the DIS geocentric (“Earth-Centered, Earth Fixed”) coordinate system:

$$X = \left(\frac{a}{\sqrt{\cos^2 \alpha + \frac{b^2}{a^2} \sin^2 \alpha}} + h \right) \cos \alpha \ \cos \omega ,$$

$$Y = \left(\frac{a}{\sqrt{\cos^2 \alpha + \frac{b^2}{a^2} \sin^2 \alpha}} + h \right) \cos \alpha \ \sin \omega ,$$

$$Z = \left(\frac{b}{\sqrt{\frac{a^2}{b^2} \cos^2 \alpha + \sin^2 \alpha}} + h \right) \sin \alpha .$$

Remember, angles are in radians here. Alpha is latitude, omega is the longitude, a is the semi-major axis of the WGS-84 specification, 6378137, and b, the semi-minor axis of WGS-84, is 6356752.3142.

Converting from DIS coordinates to latitude, longitude, and altitude is a little trickier.

First, longitude:

$$\cos \omega = \frac{X}{\sqrt{X^2 + Y^2}} , \quad \sin \omega = \frac{Y}{\sqrt{X^2 + Y^2}} ;$$

Next, latitude. This can be done iteratively for better precision, but one iteration gives about five decimal places of accuracy:

$$\alpha \simeq \tan^{-1} \left(\frac{a^2}{b^2} \frac{Z}{\sqrt{X^2 + Y^2}} \right)$$

Finally, altitude:

$$h = \frac{\sqrt{X^2 + Y^2}}{\cos\alpha} - \frac{a^2}{\sqrt{a^2\cos^2\alpha + b^2\sin^2\alpha}}.$$

By “Give Me the Equation” I Meant “Give Me the Source Code”

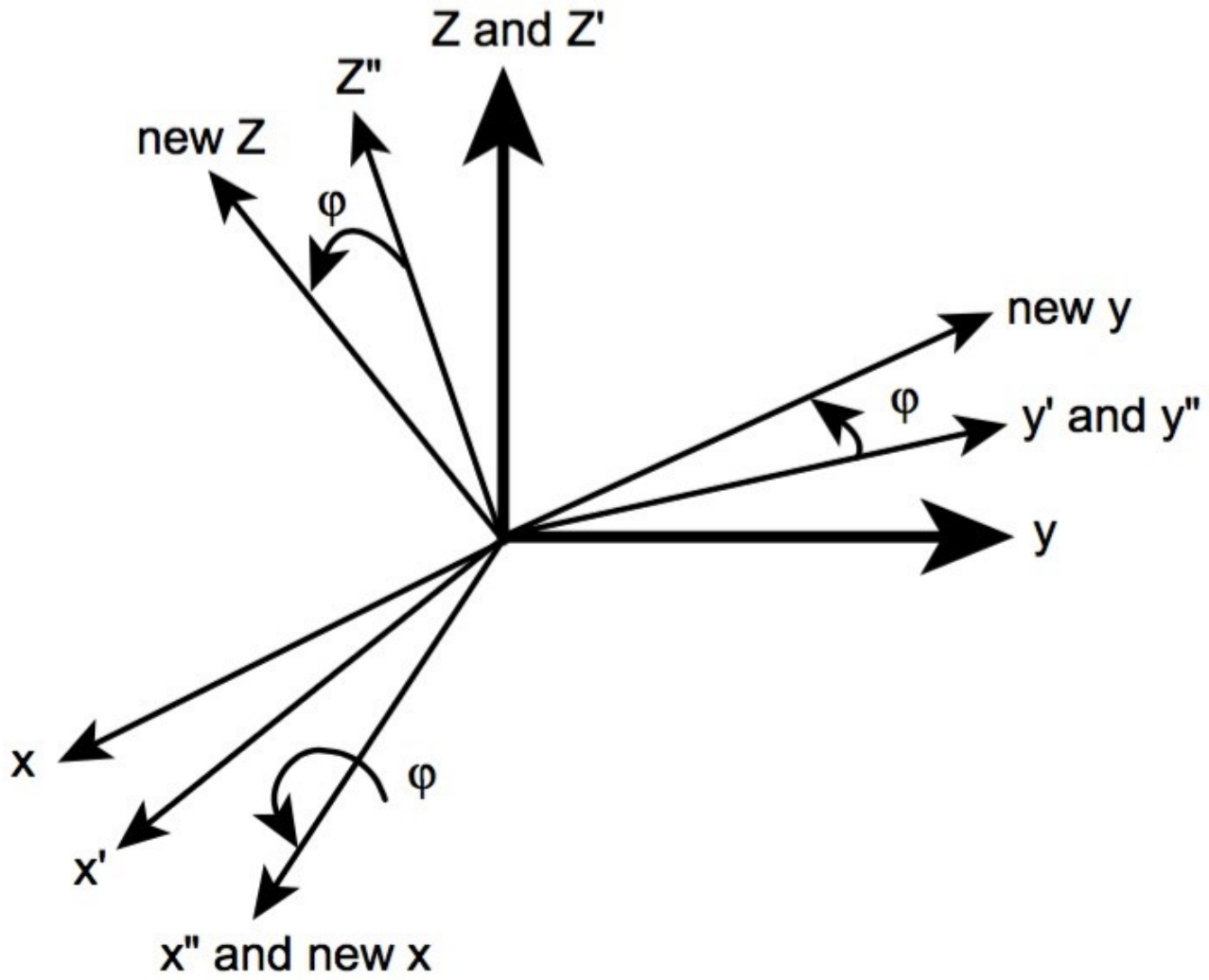
A Javascript implementation of coordinate conversion is here

The Javascript code to set up a local tangent plane coordinate system is here

Orientation

We can place an entity in the world, but how do we know which way it’s facing? In the case of DIS, the convention is to express entity location in terms of sequential rotations about coordinate axes.

The record expressing orientation has fields for psi, theta, and phi. These represent angles, expressed in radians, in the entity’s global coordinate system. First, rotate psi radians around the z-axis which points out the north pole in the DIS coordinate system. Then rotate theta radians around the y-axis, and finally phi radians around the x-axis. The final state, after three rotations, is shown in the image below:



The Austalian Defense Force has published a fine paper on the mathematics involved, including the use of quaternions to aid in computation. See the Kok paper below in "further readings."

https://discussions.sisostds.org/index.htm?A3=ind0210&L=Z-ARCHIVE-SISO-ENUM-2002&E=quoted-printable&P=7277&B=–&T=text%2Fhtml;%20charset=UTF-8&pending=

Consider two cases: we want convert a position and orientation of a vehicle at the Naval Postgraduate School in Monterey to the standard used by DIS, and convert the values we get from DIS in a state update to position, with roll, pitch, and heading. It's at 36.5973° N, 121.8731° W, altitude 5 m, and it's pointing +20 degrees from north, with a 5° roll and a 10° pitch.

First of all, we need the postion of the entity as expressed in DIS (aka ECEF) geocentric coordinates. The equation for this is above. There's also an online calculator at an online source. The ECEF coordinates are (-2707135.985, -4353750.737, 3781611.558) for that latitude, longitude, and altitude. Next we need to find a coordinate base system rotated to the same orientation as the vehicle (using the NED convention, North=x, East=y, Down=z.)

Tangent Planes

Set up a local tanget plane

Entity Local Coordinate Systems

In addition to the global coordinate system, which are used to position entities in the real world, DIS sometimes uses a local coordinate system to describe items relative to the entity in question. A local coordinate system has its origin at the center of the entity's *bounding volume*. A bounding volume is a closed volume that completely contains the entity. If there's a tank with a complex shape, then a bounding volume might be a box large enough to completely contain the tank. It's useful for creating a computationally efficient algorithms that discover things like entity collisions, and for certain graphics calculations relating to view frustums. In the context of DIS the local coordinate system can be used to describe where, specificaly, a munition impacted on an entity.

The local coordinate system's x-axis points out in front of the entity, the y-axis out the right hand side, and the z-axis points down, in a conventional right-handed coordinate system arrangement.

Further Reading

Sedris SRM package: Sedris SRM site

SRM Tutorial: Youtube Tutorial

SRM Tutorial, hardcopy: Hardcopy slides

DTIC manual for coordinate system transformations: DTIC Manual

Coordinate System Transformation theory: Book Chapter

"Using rotations to build aerospace coordinate systems", Kok: Australian Defence Force paper

WGS precision: https://tools.ietf.org/html/rfc7946

Datums effect on positons: http://www.geo.utexas.edu/courses/371c/Lectures/Fall14/Datums_GCSs_Spring14.pdf

---