USING C# AND WPF TO CREATE A DYNAMIC 3D TELEMETRY THEATER AND TRAJECTORY VISUALIZATION TOOL

By Mark Reff and John O’Neal
Raytheon Missile Systems

ABSTRACT
Telemetry data from flight tests are normally plotted using MatLab™ or other third party software. For most of the trajectory and flight parameters, a static 2D or 3D line graph does not provide the proper data visualization that can be accomplished with 3D software. Commercial 3D software can be expensive and difficult to customize, and writing custom software using Direct3D or OpenGL can be complex and time consuming. These problems were overcome using C# and Windows Presentation Foundation (WPF) to quickly and easily create a 3D Telemetry Theater and Trajectory Visualization Tool to dynamically display actual and simulated flight tests.

Keywords: Telemetry, Data Visualization, Flight Test, C#, WPF, 3D

INTRODUCTION
Many organizations perform real-time hardware-in-the-loop (HIL) and computer-in-the-loop (CIL) simulations in conjunction with actual flight testing to validate flight hardware and software. The resulting telemetry data from these tests are commonly plotted on line graphs using MatLab™ or other third party software. For most of the trajectory and flight parameters, a static 2D or 3D line graph does not provide the proper data visualization that can be accomplished with 3D software. Unfortunately, existing commercial 3D software can cost thousands of dollars, be difficult or impossible to customize, and contain many unnecessary or unwanted features. Alternatively, writing custom software using Direct3D or OpenGL can be complex and time consuming.

For these reasons, the option selected for this project was the creation of a custom 3D application that included only those features requested by the users. This paper describes how C# and Windows Presentation Foundation (WPF) were used to quickly and easily create a lightweight dynamic 3D Telemetry Theater and Trajectory Visualization tool, including the processes developed for dynamic display.

INTRODUCTION TO WPF 3D GRAPHICS
Microsoft’s Windows Presentation Foundation (WPF) contains functionality to draw and manipulate 3D objects. WPF contains a Viewport3D class which is the background on which the 3D scene is rendered. A Camera class defines the viewpoint in 3D space. As such, this class contains properties to set the field of view, the look direction, and the near and far plane.
distances. WPF also has four light source classes that illuminate the 3D scene: AmbientLight, DirectionalLight, PointLight, and SpotLight. AmbientLight illuminates objects evenly, regardless of location. DirectionalLight illuminates from a specified directional vector. PointLight illuminates from a specified 3D position and contains a Range property to specify the light’s distance. SpotLight illuminates a specified cone-shaped area. These classes can be used separately or in combination with one another.

WPF also contains classes that represent the models that are moved and manipulated in 3D applications. The GeometryModel3D class, which inherits from the abstract Model3D base class, is used to display 3D objects. Multiple models can be combined into a single model using the Model3DGroup class. GeometryModel3D and Model3DGroup objects must be attached to a Visual3D subclass, either ModelVisual3D or ModelUIElement3D.

3D models contain vertices, with X, Y, and Z coordinates representing their position in 3D space. A collection of vertices is known as a mesh, and meshes are usually comprised of triangles (i.e., 3 connected vertices). Meshes are used to form the shape of the models. Depending on their level of detail, 3D models can contain thousands of vertices.

**GENERATING 3D MODELS IN WPF**

In WPF, XAML (Extensible Application Markup Language) is used to render visual UI components including 3D models. XAML is an XML, tag-based language. XAML files are connected to C# files known as “code-behind” that contain partial class definitions. For anything other than basic 3D shapes, building 3D models directly in XAML is a laborious process. Usually, the models are created using commercial 3D computer graphics software.

Once the 3D models have been created, they must be converted to XAML. Commercial and open source applications exist that import the major 3D file formats (.x, .fbx, .3ds, .obj, .blend) and convert them to compliant XAML files.

XAML 3D model files can be quite large, depending on how intricately the model is drawn, and, therefore, the number of vertices that must be included in the file. Figure 1 shows a fragment of XAML code for a 3D model of a mock missile, containing the first vertex:

```xml
<ModelUIElement3D x:Name="MissileUIElement"
    <Model3DGroup x:Name="MainBody"
        <GeometryModel3D>
            <GeometryModel3D.Geometry>
                <MeshGeometry3D Positions="-0.10101453313053, -0.0709033921942744...
                    TriangleIndices="0 1 2 3...
                    TextureCoordinates="0.0800293162465096, 0.941046416759491.../>
            </GeometryModel3D.Geometry>
        </GeometryModel3D>
    </Model3DGroup>
</ModelUIElement3D>
```

**Figure 1: XAML for a 3D Model**
Rendering 3D models in WPF has a built-in advantage over other 3D frameworks: hardware acceleration. WPF renders graphics through the graphics card, which allows the CPU to perform other processing and accelerates graphics performance. Using Windows 7 or 8, a WPF 3D application can load and process multiple XAML models with significant detail quickly (within a few seconds). Running the application in Windows XP, however, can take up to one minute to load the same models.

For this project, the models were created using commercial 3D modeling software. The models were then converted to XAML using an inexpensive conversion application. Four different types of models are used in the 3D Telemetry Theater: missiles, launch platforms, targets, and backgrounds. The missile models are actually composites comprised of separate models (and therefore different XAML files): the main body, the fins, and possibly a booster. The separate sub-models are necessary so that each section can move independently. During a missile flight run in the 3D Telemetry Theater, the fins move based on the fin position telemetry values, and for missiles equipped with boosters, the booster drops from the missile body when the booster separation telemetry signal is received.

DESIGN OF THE DYNAMIC 3D TELEMETRY THEATER AND TRAJECTORY VISUALIZATION TOOL

The remainder of this document details the classes and processes that were used and developed to create the dynamic 3D Telemetry Theater and Trajectory Visualization Tool.

Dynamic Movement

The 3D Telemetry Theater accomplishes dynamic movement in two ways: by using a slider control to manually control movement and by using play/pause buttons based on a timer to automatically control movement. The slider’s increments are represented by the individual major frames of the flight run’s telemetry; each click on the slider’s forward arrow button advances one major frame in the telemetry data (and vice-versa for the slider’s backward arrow button). The slider can be manually scrolled forward and backward, which moves all the model objects forward and backward through the flight run.

For the play button, the models are moved based on a timer synchronized to the telemetry frame rate of the missile. For example, if the missile frame rate is 10 Hz, the timer starts and stops every 100 milliseconds. In order to automatically move the WPF 3D objects via the play button, the IncrementFrame method in the 3D Telemetry Theater is registered as the Rendering event handler for the C# CompositionTarget class. This method contains an instance of the C# Stopwatch timer class that is used to control the frame rate based on milliseconds. At a missile frame rate of 10 Hz, the Stopwatch timer object fires an event every 100 milliseconds; this event calls the IncrementFrame method, which increases the telemetry major frame counter by one. This results in the 3D Telemetry Theater automatically displaying 10 telemetry frames per second. The pause button simply stops the Stopwatch timer until the play button is pressed again.
3D Line Class

The ScreenSpaceLines3D class is part of the 3D Tools for WPF package. The 3D lines are created by drawing interlocking triangle meshes, which to the naked eye appear as smooth lines. The ScreenSpaceLines3D class requires a collection of 3D points that represent the points on the line, and it uses 4 x 4 matrix transformations to determine the dimensions and orientation of the line that's drawn. The color and the thickness of the lines can also be modified.

This class is used in the 3D Telemetry Theater to draw the XYZ axes and the 3D trajectory lines for the missiles and targets. In WPF 3D, as in most 3D software, the Y axis is up, the X axis is to the right, and the Z axis extends out from the screen. In the 3D Telemetry Theater, the ScreenSpaceLines3D class is also used to draw distance marker lines on the three axes.

For the trajectory lines, the XYZ coordinates of the missile and target are used. These lines are constructed and displayed dynamically, with two new XYZ position points added for every major frame in the flight run (whether the frame slider or the play/pause buttons are used to dynamically progress through the run). When the frame slider is moved backward, the collection of XYZ points passed to the ScreenSpaceLines3D object is reduced, which triggers the trajectory lines to be redrawn, thereby visually erasing the trajectory lines as the slider moves backward.

An important consideration when drawing trajectory lines is how to handle dropouts in the telemetry data. Plotting dropouts causes jagged trajectory lines. Since the users of the 3D Telemetry Theater wanted smooth trajectory lines, all dropouts were removed from the trajectory points.

Separate colors are used for each axis line and missile and target trajectory lines. A checkbox on the 3D Telemetry Theater window allows these lines to be turned on and off, which is accomplished by simply setting the thickness property in the ScreenSpacesLines3D object to 0 for off and 1 for on.

Camera Class

In WPF 3D, the Camera class encapsulates the 3D point of view (i.e., the positioning of the camera). It contains the Position, FieldOfView, LookDirection, and UpDirection properties. The Position property is the XYZ point of the camera in 3D space. The FieldOfView vector contains the camera’s horizontal field of view in degrees. The LookDirection vector specifies the direction the camera is aimed. The UpDirection vector specifies which direction is “up” from the camera’s perspective. Each model can have its own camera and every camera created must be added to the WPF Viewport3D instance.

In the 3D Telemetry Theater, the AbstractCamera class is the base class for the theater’s 3D cameras (and it contains an instance of the WPF 3D Camera class). The subclasses are the specific camera perspectives used in the application, such as the MissileCamera, TargetCamera, and LaunchCamera classes. Cameras in WPF 3D can be movable or stationary.

MissileCamera is a movable camera and can be thought of as a “chase” or tracking camera, following alongside the missile model as it moves through 3D space. As a result, every time the
missile moves (i.e., its XYZ position changes), the position of the MissileCamera must be updated with the missile’s position coordinates added to the MissileCamera’s distance from the missile model. LaunchCamera is a stationary camera and although its position does not change, its LookDirection changes to allow the camera to continue aiming at the missile as it moves. The LookDirection for this perspective is simply the inverse of the missile’s current position.

Rotating, Zooming, and Panning

The 3D Telemetry Theater’s CameraPerspective class allows mouse movements to perform rotation, pan, and zoom functions on the 3D camera. For rotation, the camera is rotated around its focused object (i.e., the missile or target), which has the visual effect of rotating the entire 3D scene. In 3D programming, the camera can fully rotate at any angle. Full camera rotation with a single mouse movement works well when the 3D world is small and encompasses a single object because it is easy to visually re-orient the camera back to any position. But the authors discovered that for a large world with multiple objects, like this application’s missile flight run, it is difficult to visually re-orient the camera using full camera rotation with a single mouse movement. As a result, the 3D Telemetry Theater uses two mouse movements to separate the horizontal and vertical rotation.

Full horizontal rotation is performed by dragging the mouse horizontally left or right while pressing the left mouse button. This rotation is around the Y (i.e., up) axis and the rotation amount is based on the amount of movement in the mouse’s X position. Using that amount, the camera’s X and Z values are calculated with sine and cosine in C# as shown in figure 2.

\[
\begin{align*}
\text{double CameraPositionX} & = \text{MissilePositionX} + (\text{CameraDistance} \times \text{Math.Cos(RotationAmount)}); \\
\text{double CameraPositionZ} & = \text{MissilePositionZ} + (\text{CameraDistance} \times \text{Math.Sin(RotationAmount)});
\end{align*}
\]

Figure 2: Camera rotation calculations

Instead of full vertical rotation, the 3D Telemetry Theater performs a 180 degree rotation by moving the camera over and under the missiles/targets by increasing and decreasing the camera’s height (Y position), while simultaneously decreasing the camera’s X and Z positions to position the camera directly above or below the missile or target. This is accomplished by dragging the mouse vertically up or down while pressing the right mouse button.

The 3D Telemetry Theater has two zoom modes. The first is a near distance zoom, controlled by moving the mouse scroll wheel up and down. This zoom simply moves the camera forward and backward by increasing and decreasing the camera’s X and Z positions by a small amount. For longer distances, the 3D Telemetry Theater has a quick zoom option. This is controlled by two WPF buttons on the main Window, one button for zooming in and one for zooming out. These buttons control the camera in the same manner as the mouse scroll wheel, except the X and Z positions are increased by a larger amount. The height (Y position) of the camera is also increased and decreased, due to the longer distances covered. In addition, since this zoom is
controlled by GUI buttons and not the mouse, the camera position additions and subtractions must be performed on a background thread so the UI thread does not become locked.

Panning is accomplished using the left, right, up, and down arrow keys. Pressing the up and down arrow keys performs a calculation that increases or decreases the Y (up) value of the camera’s position by adding or subtracting the product of the UpDirection vector and a fixed key press amount from the camera’s position. This causes the camera to pan up and down. Correspondingly, pressing the left and right arrow keys causes the camera to pan left and right. This movement is performed by computing a right vector, which is calculated as the product of the camera’s LookDirection and UpDirection vectors, multiplied by the fixed key press amount. This value is then added or subtracted from the camera’s position.

**Model Class**

Model is the abstract base class for the 3D Telemetry Theater models. The base constructor, as shown in figure 3, instantiates the model with a container object, viewport object and 6DOF (Six Degrees of Freedom) data interface. The 6DOF data interface provides methods for accessing the telemetry data for the positions and rotations of the model in 3D space over time.

```
public Model(ContainerUIElement container, Viewport3D viewport, ISixDOFData iSixDOFData)
```

**Figure 3: Model Base Constructor**

Movement of objects in the viewport is accomplished via the virtual `Move` method. Here, a `Transform3DGroup` variable is created, which represents a collection of the three matrix transformations: rotate, translate and scale. A code snippet is shown in figure 4 below. Note that the order in which the transformations are applied is important. For example, rotation about X, then Y, then Z will have a different orientation than rotation about Y, then X, then Z. This is because the `Transform3DGroup` class is an *ordered* collection, and matrix multiplication is not communicative (i.e., XYZ rotation ≠ YXZ rotation). Also note that transforms in the collection are applied from first to last.

```
Transform3DGroup transformGroup = new Transform3DGroup();
// Can also use vectors
ScaleTransform3D scaleTransform = new ScaleTransform3D(scaleX, scaleY, scaleZ);
TranslateTransform3D translateTransform = new TranslateTransform3D(posX, posY, posZ);

RotateTransform3D xRotate = new RotateTransform3D();
RotateTransform3D yRotate = new RotateTransform3D();
RotateTransform3D zRotate = new RotateTransform3D();

// rotx, roty and rotz must be in degrees, not radians.
// rotations about x, y and z
xRotate.CenterX = x;
```
MovableText Class

In the 3D Telemetry Theater, movable text is used to denote the missile and target types, to label
the X, Y, and Z axes, and to mark the axes distance lines. This class allows 2D text to move in
conjunction with 3D objects. Because the text is displayed in 2D, it, like any text, remains
parallel to the screen surface and is therefore easier to read than 3D text. Because it is a 2D
component, the WPF TextBox used to display the text cannot be added to the Viewport3D
which contains the 3D objects. Instead, it must be added to a transparent WPF Canvas, which is
overlaid on the Viewport3D.

The MovableText class accomplishes movement of 2D text in 3D space by computing the
transform from 3D space to the inner 2D space of the Viewport3D. In 3D computer graphics,
three matrices make working with these transformations easier: the World matrix, the View
matrix, and the Projection matrix. The World matrix represents where the models are
located in 3D space, the View matrix represents where the camera position is located, and the
Projection matrix represents the location on the 2D display screen. The MovableText class
multiplies the World matrix, the View matrix, the Projection matrix, and the 3D point position of the model object or 3D line to determine the 2D point position for the TextBox.

**Background Class**

Background is a subclass of Model and contains the sky and surface backgrounds for the 3D Telemetry Theater. This class creates a sky dome, which is a half sphere and contains thousands of vertices. The sky dome uses image texture files that contains artifacts (sun, clouds, stars, etc.) and allows sky options that represent different times of the day: daytime (with the sun and clouds), evening (with darker clouds) and nighttime (with stars).

The surfaces drawn for the 3D Theater are separate image files and represent land or water. The Background class contains options for a desert or an ocean surface. These image files contain artifacts such as hills or waves. Figure 5 shows the 3D Theater with the sky dome background and a mock missile in flight.

![Figure 5: Sky dome background](image-url)
This class also contains a background option that displays the X, Y, and Z axes on an off-white background. The three axes are drawn using the `ScreenSpaceLines3D` class discussed earlier. Each axis is drawn with a different color and is clearly marked with movable text that move with the axes when the users rotate the 3D scene. The axes also contain movable text distance marker lines. Figure 6 shows the XYZ Axes background with a mock trajectory line.

![Figure 6: XYZ Axes Background](image)

When the sky background is selected, the sky model rotates and moves up and down as the missile and target models move. Because the sky artifacts, such as clouds, stars, etc. rotate and move with the sky model, the user is given a sensation of the missile and target models moving through the sky. When the missile and target models move upwards (i.e. their Y position increases), the sky moves downwards, and vice-versa. When the missile and target models move forward (i.e., their X and/or Z positions increase), the yaw of the sky model is increased, which causes the sky to rotate around the Y axis.
CONCLUSION

This paper has demonstrated that using C# and WPF 3D is an excellent option for creating a dynamic 3D Telemetry Theater and Trajectory Visualization Tool. As opposed to purchasing commercial 3D software, developing a 3D application in C# and WPF can be done quickly for little cost. In addition, WPF 3D Graphics is easier to use and understand than attempting the steep learning curve involved with DirectX, OpenGL, and GPU shaders. Finally, WPF allows seamless blending of 3D and 2D content. Using C# and WPF allowed the authors full control to create a lightweight dynamic 3D Telemetry Theater application.

REFERENCES


