Rendering Flight Telemetry in Platform-Independent Three-Dimensional Touch-Interactive Simulations

Matthew A. Noyes*

University of Rochester, Rochester, NY, 14627

Robert Hirsh†

Johnson Space Center, Houston, TX, 77058

The proliferation of portable, touch-enabled computational devices has led to a fundamental paradigm shift in human-computer interaction. By combining the flexibility of software systems with the intuitiveness of the touch screen, new methods of data visualization become possible. This paper explores the use and architecture of 3D graphics engines to display precise digital representations of flight test scenarios using live or pre-recorded telemetry, using the commercial UNITY 3D engine as a case study. This paper focuses on both the rationale of 3D graphics engine and touch screen utilization, as well as the technical design of the telemetry-delivery system and application software.

Nomenclature

CAD	Computer Assisted Design
CGI	Computer Generated Image
CEM	Computational Electromagnetism
CFD	Computational Fluid Dynamics
HUD	Heads-Up Display
GUI	Graphical User Interface
GPU	Graphics Processing Unit
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VTB	Vertical Test Bed

*Software Engineer Intern, Spacecraft Software Engineering

†Aerospace Engineer, Spacecraft Software Engineering

I. Introduction

Three dimensional graphics are a critical asset to engineers of all disciplines. In the last decade, computer graphics have revolutionized analytic and design processes from the most massive of spacecraft down to the tiniest of microchips. Graphics play many roles in product creation depending on the development stage, from conception and prototyping to test data visualization; it also applies to physical simulations and CEM/CFD, or performance simulations like those related to propulsion systems and stress analysis.[1] However, much of today's computer-assisted design takes place using the same mouse-and-monitor graphical user interface developed decades ago. The purpose of CAD is to make the process more user-friendly, justifying exploration of new applications in human-computer interaction. Additionally, while simulations are vital in product creation they typically involve rendering based on a numerically solved set of established physical equations, rather than live telemetry from actual test vehicles.

First, a rationale was developed for the introduction of a new interface in engineering product design, followed by a motivation to apply this new interface to real-time telemetry-driven simulations. After weighing several options, a commercial rendering engine was selected and the software architecture and telemetry-routing system was designed and implemented in prototype form, concluding with performance analysis and an enumeration of future enhancements.

II. Rationale

In 1984, Apple released the Macintosh, the first commercially successful computer to use a GUI. It spawned a new wave of devices that would soon change the world. The GUI dramatically altered the consumer computing landscape; no longer bound by esoteric command line interfaces, personal computer usage skyrocketed. Computers became intuitive, incorporating real-world analogies to software domain abstractions, such as the "dragging and dropping" of files. Hardware became relatively small compared to the volume of mainframes and workstations, thus finding a place not just in business, but in the home. Systems could enhance productivity, as well as provide entertainment. The GUI-enabled desktop catalyzed a paradigm shift, transforming lives in nearly every regard.

This design choice remained constant during the following two decades. Although hardware rapidly expanded its technical specifications, performance, and efficiency, neither the GUI nor hardware portability dramatically improved. The latter has seen attempts at improvement via the design of notebook and netbook computers; while these devices are certainly more portable than desktop PCs, they are still relatively bulky and cumbersome compared to today's alternatives.

In 2007, Apple released the iPhone, again creating a ripple in the computing industry. The iPhone spawned countless alternatives, imitations, and improvements from competitors, solidifying the smartphone market. Now, a powerful computational device can fit in the palm of one's hand, the hallmark of true portability. The highly responsive touch interface once again intuitively mimics real-world concepts for data manipulation, such as 3D object translation via dragging gestures, and rotation via twisting gestures. These devices are highly versatile due to the thousands of applications that are highly accessible via centralized distribution. Private firms are recognizing the value of smartphones as flexible employee productivity boosters, warranting investigation into its use for engineering analysis. Smartphones also have applications for public outreach given their remarkable popularity.

One potential use in both areas includes three-dimensional telemetry displays. Well-executed engineering requires repeated and thorough testing to ensure product quality. In NASA's quest to lay the foundation for next-generation space missions, this involves developing new propulsion systems and rocket designs, as well as flight software guidance algorithms to operate these vehicles. Both products require operational verification on live test vehicles. As VTB flight frequency increases, telemetry analysis requires greater dedicated field test time. Due to smartphone portability, a new, intuitive telemetry display for the device provides instant datastream access. Due to its compact size, the display would have to intelligently represent this telemetry to maximize available space. A lifelike modeling of a test run best captures this requirement by condensing large amount of numerical information into a display mirroring reality.

Additionally, this application serves as an outreach tool by providing an interactive method for the public to watch live test flights. Designated *iMorpheus*, this application will be the primary education asset for NASA's Project Morpheus. It will allow users to view test flight simulations using live data, browse and play recorded data files, and control the virtual simulation model for self-piloted flights around a virtual JSC.

III. System Design

III.A. UNITY 3D Engine

The UNITY 3D engine powers the iMorpheus telemetry display. Traditionally used as a game development tool, UNITY comprises an ideal candidate due to its highly optimized graphical rendering agent, an incorporated physics engine for simulating state changes, integrated plugin expansion capabilities for, among other aspects, socket classes, and its cross-compilation to a diverse set of devices. UNITY-developed products run on standard web browsers, PC/Mac standalone executables, iPhone, iPod Touch, and iPad, Android devices, and XBox 360, Playstation 3, and Wii. For engineering analysis, compilation to multiple smartphones and desktop computers allows for use both on and off the field. In terms of public outreach, the application will have access to multiple consumer groups, each with its own set of unique hardware. Both iOS and Android smart phones allow for touch controls, while all three entertainment consoles support joystick motion control for local simulation user input. Additionally, both smartphones and entertainment consoles contain their own regulated app stores, removing a need for NASA to provide a distribution service and bandwidth.

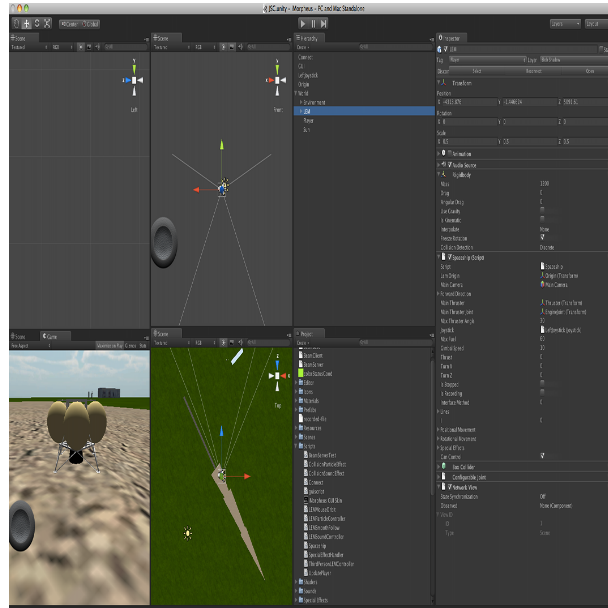


Figure 1. The UNITY editing window. This view contains, from left-to-right: viewing windows for the xy, xz, and yz planes and main camera, the GameObject hierarchy, asset hierarchy, and selected object components. The selected GameObject (VTB) contains audio source, physical control script, collider, joint, and network Components.

The engine provides a high degree of modularity and reuse due to its treatment of simulation assets. The engine models every object in the environment as a tree of *GameObjects*. A GameObject represents a specific

piece of the world with base attributes for position and orientation relative to an absolute coordinate grid, and may be outfitted with specific *Components* for enhanced functionality. For example, attaching a *Mesh Renderer* Component with an accompanying 3D model renders that model on top of the point position with the given rotation of its parent. Attaching a *Rigidbody* Component applies physics engine modeling to the object complete with customizable parameters for mass, atmospheric drag, collision detection, and so forth. A parent object's children maintain spatial relationships in the local coordinate system given a translation, rotation, or scaling of that parent, allowing small, simple objects to comprise larger, more complex ones. Within the tree of GameObjects, any subtree can be exported as a package complete with all rendering resources for use in any other UNITY application. For example, the test bed model could find use in another 3D environment, or spacecraft built by other developers could be easily imported for use in the simulated world. As a result, UNITY is ideal for collaborative work among multiple software engineers and projects.

III.B. Updating Vehicle State

III.B.1. Using Live Telemetry

There are several methods to implement networking in UNITY:

1. State Synchronization

   UNITY allows certain specific GameObject components to share data over a network stream, including position, velocity, and animation. This can be performed via a reliable delta compression algorithm, where data is only sent over the network following a state change. Data delivery supports both guaranteed (TCP-like) and uncompressed, unreliable (UDP-like) modes. Because iMorpheus supports sharing data unrelated to the formats state synchronization allows, such as fuel and throttle `floats`, the application does not support this method.

2. Remote Procedure Calls

   Remote procedure calls allow a server to call specially defined functions on its clients, and even to pass arguments. While remote procedure calls are limited to a specific subset of possible variables (`integer`, `float`, `string`, `NetworkPlayer`, `Vector3`, `NetworkViewID`, and `Quaternion`) thereby preventing sending a single object instance containing variables of these types, it does allow for sending a theoretically infinite argument list to clients, as well as performing additional client-side work after variables have been set. However, remote procedure calls must be made from an application on a server running a UNITY application, which is not supported in our server-side architecture.

3. Custom Socket Classes

   UNITY's highly extensible scripting and plugin systems allow custom C# socket programming. It is possible to define custom scripts to read incoming socket data from any source, including those originating outside UNITY, such as the VTB telemetry delivery system. This methodology allows for the greatest level of flexibility, but also requires greater design and development. The most significant work efforts would remain server-side.

The team considered writing a UNITY application to receive raw numeric telemetry, build data values and transmit data to clients using the UNITY networking code. This would have significantly reduced development time and code complexity. While this

solution would have been ideal for internal team use, it would have been difficult to scale for a public release. For network management purposes, the team decided to focus on implementing a dedicated telemtry push server.

The Morpheus vehicle uses the ITOS tool to beam telemetry to a communication center near the launch site. The communication center forwards ITOS telemetry to the Mission Control Center, where it is republished via a Lightstreamer data adapter. Lightstreamer is a push service, refreshing live data as soon as new values are available. If data are not updated in a single refresh cycle, they are not forwarded to the client, reducing network bandwidth. This makes Lightstreamer ideal for scalable data delivery.

Lightstreamer provides clients for many different software environments, including .NET, Java, and HTML/Javascript. Due to clients' reliability on libraries unsupported by UNITY, a custom socket interface was designed for iMorpheus until Lightstreamer releases a proper UNITY client.

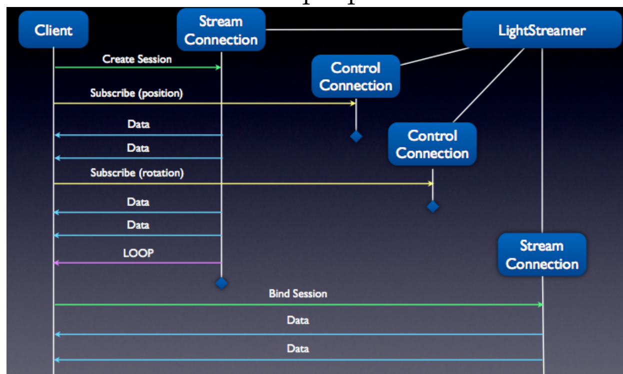


Figure 2. Control flow for LightStreamer subscriptions. Separate sockets are used for data streaming and control events. If a stream connection dies, it can be reestablished (bound) with a session ID.

Start Data Collection Thread
loop
  if Session ID == Null **then**
    Initialize New Stream Socket Session
    Store Session ID
    Subscribe to *telemetry*
  else
    Bind Stream Session Socket
  end if
  while Running **do**
    $buffer \leftarrow telemetry$
    Decode Unicode *telemetry*
    if *telemetry* arrived **then**
      Initialize *telemetry_variables*
      Assemble *telemetry_state*
      $queue \leftarrow telemetry_state$
    end if
  end while
  Unsubscribe to *telemetry*
end loop

Figure 3. Control Flow Algorithm

An iMorpheus session opens a stream connection in a background thread to the Lightstreamer server where it obtains and stores a session ID to reconnect in the event of stream closure. Lightstreamer automatically closes the stream connection once the number of bytes sent to the client exceeds a server-specified content length by default. In a real-time telemetry delivery scenario this should be avoided as it will produce a highly observable latency. The iMorpheus client requests a content length equal to the maximum integer constant; in the event of disconnect, the application will reopen a stream connection using the stored session ID. Following stream socket initialization, the application opens a control connection and submits an HTTP request over a TCP socket, which adds a subscription to all telemetry variables required for modeling state, then closes the control connection. The application will begin receiving raw bytes, which are stored in a fixed-size clientside buffer. Thread progress halts until data are received. No timeout is specified to prevent disconnection in the absence of realtime updates.

The Morpheus vehicle provides updates at 10 Hz; while Lightstreamer's maximum update frequency is 1 Hz; to compensate for this latency, the server delivers telemetry values as JSON strings, containing an `Arraylist` of 10 states. The iMorpheus application parses this object and updates vehicle state over the

course of a second. iMorpheus vehicle state will have a minimum latency of 1 second from data delivery; there will be additional lag derived from linearly interpolating the spacecraft along designated waypoints.

When data are received, they are decoded into an ASCII text string containing unicode characters for quotation marks, carriage returns and line feeds. An explicit `String.Replace()` call translates the data into well-formatted JSON. This string begins with a comma-delimited numeric sequence indicating the table number and item number of the update, followed by a vertical-pipe delimited sequence of JSON strings, each specifying a single cell in the subscribed table. Each cell corresponds to a single `Telemetry` variable. Each variable follows the following schema:

Name	Telemetry ID
Class	Telemetry Type
Identifier	ITOS Variable Status
Color	Color Mapping to Identifier
Data	Telemetry Waypoint Array

Figure 4. JSON Telemetry Schema

Value	Telemetry Data
TimeStamp	Time Data Recorded

Figure 5. JSON Waypoint Schema

The incoming JSON data was originally formatted to be deserialized by the NewtonSoft JSON library, however this library was found to be incompatible with UNITY. The data is instead decoded by MiniJSON into a `Hashtable`. Individual values are pulled from this `Hashtable` and placed into `Telemetry` objects, each storing data for a single state variable. A `Telemetry` objects array representing a single Lightstreamer response is placed on a thread-safe `Queue`. This `Queue` is simultaneously accessed by the main thread to populate vehicle state.

Due to the magnitude of data required to model vehicle state, as well as the significant overhead of using JSON strings, data are split into packets of unknown size. Each time data is read and decoded, the number of bytes received is stored and the size of the buffer updated accordingly. If a Lightstreamer table header is located in the middle of the buffer, multiple datasets have been received; the application will then store the first complete set of data in a string for parsing; the remaining bytes are copied to the beginning of the buffer and the buffer size is adjusted accordingly. Bytes stored beyond the index specified by the buffer size are not zeroed and are ignored by the application. If the application has not detected complete reception of data, it waits until more bytes are received before checking again.

After complete reception but before parsing, the application also checks the data string for a `LOOP` message. Lightstreamer sends status messages if necessary in the final cell of a table request. A `LOOP` request is sent upon reaching content length; if received the application opens a new stream connection which intelligently restores communication with the stored session ID. After splitting the table along vertical pipes and whitespace into the individual table items, the items are ready to be parsed.

All connection code is managed by an abstract superclass, hiding implementation details from the software engineer. Data are parsed using the `parseBytes()` method defined by the Lightstreamer client. The `parseBytes()` method takes individual lightstreamer table cells, decodes the JSON strings into values and assembles the values into `Telemetry` objects, where each `Telemetry` object represents a state single variable.

State updates are performed from the main thread, which grabs a reference to (but does not dequeue) the next telemetry update. If the `Queue` has `Telemetry` available, we cre-

ate a set of a local replacement variables for each piece of data to be updated. `Telemetry` values are queried fetched from a helper function, `GetTelemetryValue()`, which takes an ITOS telemetry identifier string, a queue, and the old variable to update's value. If a `Telemetry` object with the specified name parameter exists in the head of the `Queue`, its stored value is returned; otherwise, the old variable's value is returned. Since Lightstreamer is change-only, new state telemetry may not contain all possible variables; it is simpler from a logic perspective to simply assume all variables will be updated with their old values rather than explicitly checking existence outside the helper function.

NASA's commitment to public outreach often implies heavy documentation during Project Morpheus test flights. To facilitate community connection, a journalist utilizes social media websites to post updates of the vehicle's status. A system was therefore designed to retrieve and display official NASA Twitter messages inside the application, in real-time.

All Twitter fetch code was implemented within a separate Component of the main GUI GameObject. The algorithm obtains an HTML string from a Twitter Atom feed containing the latest posted message to the specified account name. This information is public and does not require user credentials. The algorithm searches for HTML tags wrapping the status message and grabs the substring between them. Finally, regular expressions remove any HTML tags within the message (e.g. embedded links) before displaying in the Messages window. The method waits for 30 seconds before checking again; if the message has not changed, it resumes its wait-check cycle.

while Running **do**
  Start Coroutine
  content ← *html*
  $msg_{new} \leftarrow content_{substring}$
  Parse msg_{new} for malformed characters
  Remove nested HTML tags
  if $msg_{new} \neq msg_{old}$ **then**
    $msg_{old} \leftarrow msg_{new}$
    $msg_{window} = msg_{window} + msg_{new}$
  end if
end while

Figure 6. Twitter Message Fetching Algorithm

The Twitter fetch code is written in Javascript to take advantage of UNITY's `WWW` class, which fetches of HTML strings at the specified URL without requiring socket management as in the case of Lightstreamer subscriptions. UNITY functions may be called only from the main thread, requiring the use of coroutines, which are easier to manage in Javascript. Unlike a thread, which allows separate and parallel execution of code, a coroutine allows parallel execution within a single thread. This approach is less efficient than threading on multi-core systems due to the overhead of switching call stacks in a single line of execution, but it is considered advantageous for its simple implementation and relatively infrequent polling.

One significant obstacle in this implementation is Twitter's rate limit of 150 requests per hour to individual addresses. For a single user, this is not a problem as the maximum possible number of checks performed is 120 per hour. However because limiting is IP-based, it may be enforced on entire LANs; this theory has not been tested in greater detail.

A future implementation would take advantage of the Twitter Streaming API, however due to time limitations and the unavailability of native UNITY clients, that approach was deferred to a later iteration.

III.B.2. Using Recorded Telemetry

The engine data sent over the network can also power recording playback. Recording is currently handled by placing all data variables into a plain text file, separated by newline characters. At application startup, iMor-

pheus scans the file into memory, placing each variable at its own index within a one-dimensional array. The program stores an integer representing the current position in the array. This integer powers both reading from the array, as well as directing the state of the recording interface scrubber. Dividing the array size by the total number of variables gives the total number of frames in the recording. Each frame update, the systems sets each engine state variable in the order it was recorded, incrementing the current index after each setting. If the user stops playback, a boolean flag prevents this process from occurring and the index remains at its current position. When the user scrubs through playback, the current slider value represents a specific frame in the recording. Each frame jump, the index increments or decrements accordingly by the total number of variables to ensure data remains properly aligned. The scrubber's minimum and maximum values reflect the first and last frames, respectively, so as not to produce a misaligned value, but the system also includes a clamping function to restrict legal frames to within the appropriate limit.

Because engine data resides in a text file and is very simplistic in terms of the quantity of variables to represent each frame, it occupies very little hard drive space. A typical engine data file uses about 16 megabytes per hour, compared to 4000 megabytes per hour for high-definition footage. Also, while more resolved HD images occupy more space, iMorpheus completely decouples the view from the data; graphical fidelity may approach near-photorealism without an impact on the file size or compatibility with engine data files. Furthermore, high definition footage is non-interactive, requiring the placement of multiple cameras to ensure a complete viewing angle.

III.B.3. Using Live Controls

The iMorpheus telemetry display simulates some VTB physics to provide a realistic local-sim-control scenario. The simulation is simplified due to the hardware limitations of mobile devices, yet the capability does exist for a more complex flight model.

A single Javascript source file, "VTB.js," attached to the GameObject containing the VTB rendering agent handles all state changes on the vehicle. The script specifies a set of user-defined constants for movement settings, such as maximum speed, positive and negative acceleration, drag under various conditions, and terminal velocity. The script also defines a generic set of special effects interfaces for movement along all axes: positive/negative thrust (y-axis) and positive/negative turning (x- and z-axes). The software engineer develops the special effect Components and particle renderers, attaching them to these variables. When the VTB script detects user input for movement along these axes, it simply enables the special effects Components. When movement ceases, they are deactivated. This allows for a great deal of modularity in the spacecraft model design. Generally, splitting functionality into individual Components attached to the same GameObject simplifies code maintenance, as well as improving execution speed since boolean flags are not required every frame to determine behavior; the engine simply ignores disabled Components.

The simulation course of action depends upon the user-specified interface method. If the application is using live data, the VTB script doesn't need to simulate anything as that data arrives over the network. If the application uses recorded data, the application scans a file containing engine data and updates state information accordingly. Since the default time step will be the same during recording and playback, no other synchronization code is required. Lastly, if the user is

set to use live simulation controls, the application receives input depending on the platform. For Desktops and Web Browsers, the applications checks for an external joystick connection; otherwise it uses the keyboard for controls.

UNITY allows the developer to set "Input Axes" which map string names to specific buttons and joystick movements. Depending on context, the application selects the appropriate mapping, which returns a `float` in the range $[0, 1]$. In the case of a button this is simply either endpoint; in the case of a joystick axis any value within the boundary condition. Multiplying the input value by the maximum offset from neutral gives the current vehicle state. For example, if the joystick is half-engaged along the X-axis, it will return a value of $\frac{1}{2}$ for that axis. If maximum gimbal rotation along said axis is 30 degrees, the final gimbal position for the frame will be 15 degrees, since $30 * \frac{1}{2} = 15$.

By default the application uses the following control scheme:

Joystick	Keyboard	Function
X Axis	A, D	L-R Gimbal
Z Axis	W, S	F-R Gimbal
Y Axis	Q, E	Body Rotation
Thrust Axis	Spacebar	Throttle
Thumbstick	Arrow Keys	Camera Pan

Figure 7. Live Control Scheme for Desktops

If the application is running in a touch-enabled mobile environment, the GUI will paint a virtual joystick within reach of the user's left thumb. This joystick is used for calculating gimbal position and may be manipulated within a 1x1 bounding box. The offset from stick origin will return a value $[-\frac{1}{2}, \frac{1}{2}]$. Adding $\frac{1}{2}$ maps the return value to the $[0, 1]$ domain, allowing this GUI element to be used for input.

if platform == mobile **then**
  controller ← virtual joystick
  thrust ← acceleration * throttle
  $turnX \leftarrow offset_y * gimbal_{max_angle}$
  $turnZ \leftarrow offset_x * gimbal_{max_angle}$
else
  if joystick connected **then**
    controller ← hardware joystick
    thrust ← acceleration * throttle
    $turnX \leftarrow offset_y * gimbal_{max_angle}$
    $turnZ \leftarrow offset_x * gimbal_{max_angle}$
  else
    controller ← keyboard
    $thrust \leftarrow Full * pressed$
    $turnX \leftarrow gimbal_{max_angle} * pressed$
    $turnZ \leftarrow gimbal_{max_angle} * pressed$
  end if
end if
Create quaternion target gimbal rotation
Interpolate gimbal rotation
$fuel \leftarrow Max([fuel - (timestep * throttle)], 0)$
if Has Fuel Remaining **then**
  Calculate gimbal local vec_j
  $vec_{thrust} \leftarrow vec_j * thrust * timestep$
  Apply vec_{thrust} to rigidbody
  $verticalSpeed \leftarrow |vel_y|$
  $groundSpeed \leftarrow |vel_{xz}|$
end if
if Is Recording **then**
  $file \leftarrow data$
end if

Figure 8. iMorpheus Live Controls Algorithm. Input is context-sensitive based on the type of platform and the presence of joysticks or keyboards. Physical simulation relies upon the UNITY physics engine via velocity-change force vectors, however the simulation is low-fidelity (e.g. no altitude-proportional drag coefficients).

To pan the camera, the user simply performs a drag gesture without touching the virtual joystick, which rotates the camera along a fixed radius around a pivot point GameObject in the center of the Morpheus vehicle. The camera does not rotate around the test bed's center because the model's origin does not share the same position. The effect is additive; the more fingers used to swipe, the faster the rotation will occur. This allows for both fine-grained control and high

panning speed. The exception to this effect is for two fingers, since the application also supports pinch-to-zoom. There is no default pinch-to-zoom function within UNITY; to detect pinching, one must store touch positions from the previous frame and compare to the current frame, determining if the distance vector between these two points has decreased or increased, altering the camera distance radius to match. Note that additive panning only works for iPhone and iPad hardware as current Android technology may sense a maximum of two simultaneous touch events.

III.C. Interface Design

Figure 9. The iMorpheus interface.

iMorpheus interface design incorporates a simple GUI useable in both traditional and touch-sensitive environments. The interface is context-sensitive; since there are multiple simulation modes, certain portions of the interface may not be required for the user to view at a given time; those pieces are removed from view to conserve system resources. UNITY is very efficient at three-dimensional rendering, but has an extremely high performance cost when rendering two-dimensional GUI elements. As such, some of the greatest performance increases in a UNITY application come from maximal simplification and elimination of 2D drawing.

The telemetry window provides a basic overview of standard system telemetry for which the user would find important to see. The telemetry window does not display all telemetry data transmitted from the vehicle; most of this information is rendered via state changes to the 3D model. The window gives an indication of the following variables:

Crossrange	x-axis Distance
Downrange	z-axis Distance
Altitude	y-axis Distance
Ground Speed	xz-plane Velocity Magnitude
Vertical Speed	y-axis Velocity Magnitue
Throttle	Percent Engaged
Fuel Remaining	Fuel Indicator Bar

Figure 10. iMorpheus telemetry window fields

The telemetry window displays appropriate metric system units as UNITY's coordinate system uses a meter-scale for point positions from an absolute origin.

1. Recording Scrubber

   The recording scrubber consists of a play/pause button and a horizontal slider indicating simulation progress through a recorded engine data file. Pressing the play/pause button halts simulation progress but does not halt the engine itself; it is still possible to interact with the application via changing modes or panning the camera around the vehicle. Users may interact with the scrubber to step forward and backward very smoothly through the simulation via a mouse on the Desktop version, or via touch on the mobile version. Users in "Use Live Data" mode can observe this rewind action if connected to a server performing this action, but cannot affect the recording itself. Client-side cameras and features may still be controlled by those users, however. This view appears only in "Use Recorded Data" mode.

2. Messages Display

The messages display allows users to view instant-message data sent from the live VTB test site. This allows application users to stay up-to-date on the finer details of flight tests difficult to clearly render in the main display.

3. Touch Controls Display

   While the Desktop version of iMorpheus may be piloted via keyboard or joystick, the mobile version does not support these hardware interfaces. Instead the application renders a two-dimensional joystick texture movable within a pre-defined rectangle on the screen, snapping back to its center when released. By measuring and normalizing the offset from this center point, mobile users may control local simulation flight with a software stick. Also in the mobile version, a vertical slider (not pictured) controls throttle engagement.

Users control local simulations by choosing a throttle setting and manipulating engine gimbal direction.

1. (x,y,z) vector coordinates for vehicle position
2. (x,y,z,w) quaternion coordinates for vehicle rotation
3. (x,y,z) vector coordinates for vehicle velocity
4. (a,b) angle offset for thruster rotation
5. vehicle fuel remaining
6. vehicle throttle engaged
7. mission elapsed time

Figure 11. This algorithm requires only 15 variables to operate properly.

Clients do not calculate state changes, they only read them; however, clients are responsible for initializing special effects. Since special effects are throttle-dependent, the client analyzes throttle data and determines whether to activate its components. This reduces both network traffic and server-side computational time.

III.D. Performance Analysis

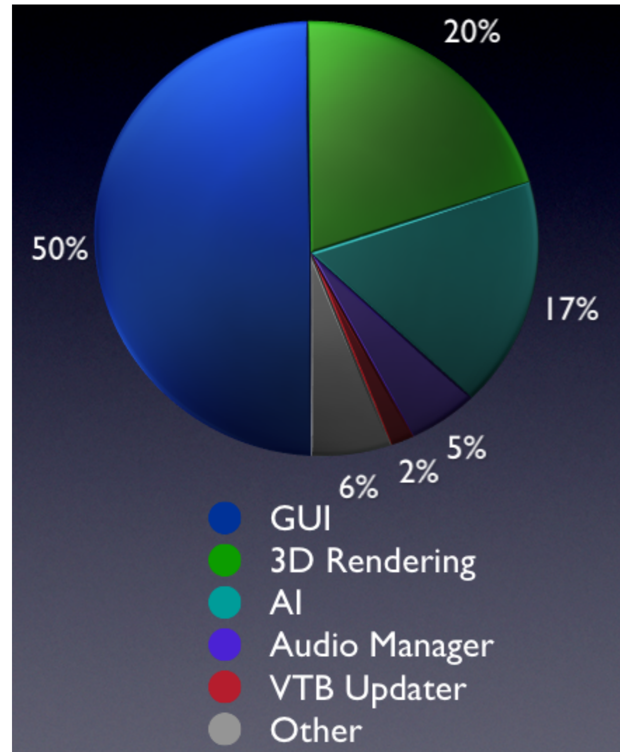


Figure 12. UNITY Desktop performance.

Performance enhancement for iMorpheus falls into two main areas: draw calls and mesh complexity. A draw call executes for each object in a given scene, during which the renderer takes the 3D model data for a structure and places it on the screen according to the position, rotation, and scale associated with its parent GameObject. Each individual draw call has a large impact on performance, so draw call reduction represents one of the best ways to improve performance. There are several ways to accomplish this:

1. Batching

Draw call batching occurs when the multiple objects that would normally have their draw calls executed separately have them execute together as if the two objects (usually spatially separated) comprise a single model. A subset of this, known as "static batching" only occurs between stationary models. Because the majority of iMorpheus scenery contains stationary objects such as rockets, buildings, and terrain, all these objects may be batched together and vastly improve frame rate. This does not apply to the VTB as its components move relative to the absolute coordinate frame.

2. Vertex Reduction

   Three-dimensional models are rendered via a series of vertices connected by edges, which in turn are connected by faces, or *triangles*. The greater the number of vertices, the greater the complexity of the model and the greater time required to render it. Modern desktop and laptop systems contain very robust GPUs that may easily render hundred of thousands of vertices and hundreds of draw calls. However, smartphones are very limited in computational resources (especially the iPad, which has very poor 3D performance given its standard computer screen resolution). In general, the iPad cannot render more than 30,000 vertices in one scene without significant slowdown. For the iPhone 3GS and older, than standard vertex count should not exceed 10,000 vertices. As such, implementing scaled-down models for older hardware could significantly improve performance on those devices.

3. 2D Rendering Optimization

   UNITY's 2D graphics engine for producing in-application HUDs and menus is peculiarly more CPU-intensive than its 3D-rendering counterpart. As indicated in FIGURE 12, GUI rendering utilizes half of all application CPU allocation.

   UNITY renders GUI objects placed in specially named scripting functions executed several times per frame. Furthermore, each GUI component by itself requires at least one draw call. As the primary numerical telemetry-display tool, GUIs form an essential portion of iMorpheus; optimization in this domain promises the greatest augmentation of performance. There are freely available scripting libraries, such as GUIManager 2, that utilize pseudo-2D GUITexture and GUIText GameObjects to create the illusion of a 2D engine, while combining all rendering functions into a single draw call. Investigation of these libraries will be essential to running efficiently on older hardware.

IV. Further Research

Further research on iMorpheus resides in one of two domains:

IV.A. Optimization

From an optimization standpoint, reducing the GUI interface to a single draw call will greatly improve performance on older, mobile hardware. Additionally, the implementation of a binary encoder rather than text encoder for recording engine data would greatly reduce file size on disk.

IV.B. Feature Enhancement

From an enhancement perspective, researching more features congruent with engineering analysis, such as higher-fidelity vehicle models with interior mechanisms, telemetry to model state changes in those mechanisms,

and graphical feedback for failure events such as applying a blinking "warning" texture. Clicking or touching the affected part could display more information about the problem cause from the telemetry perspective. Also, the ability to show or hide spacecraft "layers" (circuits, fuel lines, life support sensors, etc.) so that different engineering teams may focus on different subsets of the telemetry, would better utilize the groundwork already in place for this application. The infrastructure has already been implemented for any telemetry transmitted from the vehicle, all that remains is a comparable client-side mapping. Finally, adding more detail and structures to the terrain allows for a more realistic and immersive simulation, which is important for a public release.

V. Conclusion

Three dimensional graphics displays and touch-based interfaces represent initial steps toward intuitive human-computer interaction by allowing users to manipulate virtual environments in a manner analogous to everyday human experience. The less a user must "learn" to fully utilize system capabilities, the more intuitive the system. iMorpheus serves to represent complicated flight telemetry as an interactive simulation using real-time data, recorded data, and local user control, with the ability to forward this information to many devices simultaneously. This software product has applications in both engineering analysis and public outreach due to its ease of developing complex yet user-friendly interfaces its the wide platform base. Due to the speedy addition of new features, iMorpheus will soon have a role to play in NASA's mission.

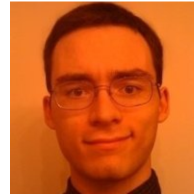
Acknowledgments

I would like to thank the following for their contributions, without whom I would be unable to pursue this project:

- **Robert Hirsh**, who worked closely on iMorpheus design.
- **Lui Wang**, who provided support in various other projects for NASA.
- **Bruce Hochstetler**, **David Goeken**, and **Jennifer Morehead** who handled server-side configuration of Lightstreamer.
- **Israel Mendez**, who helped in optimizing performance.
- **Sean Shriner**, who contributed to building layout and the message feed backend
- **USRA**, for offering me the opportunity to contribute to human spaceflight.
- And to the **NASA family**, for accepting me into this wonderful team.

References

[1]K. Sathyanarayana and G.V.V. Kumar. Evolution of computer graphics and its impact on engineering product development. In *Computer Graphics, Imaging and Visualisation, 2008. CGIV '08. Fifth International Conference on*, pages 32 –37, 2008.



Matthew A. Noyes is a student employee at NASA Johnson Space Center. He currently attends the University of Rochester in Rochester, NY pursuing a B.S. degree in Physics. He received the Outstanding JSC Intern award from JSC Education Office in the Fall of 2010. When he is not researching at Johnson Space Center and doing public outreach for NASA, he enjoys building robots, rockets and airplanes and gazing at the stars during a later night at the local observatory. His research interests include embedded and mobile application development and robotics hardware and software design.