Orbital and Trans-Planetary Trajectory Simulation System Architecture & Design

Rev. A – DRAFT
2022-09-22

Ahmed A. Moussa
ahmmoussa22@gmail.com

Author Note

This is an attempt to challenge my expertise in software development while applying knowledge from my favorite aerospace subjects: orbital mechanics and beyond-earth environment.

Abstract

Orbital mechanics simulations are often used to predict the position or trajectory of an orbiting object over time and to predict environmental conditions such as lighting, radiation, or atmospheric drag throughout the orbital lifetime. Aside from simulating the environment and classical mechanics of motion, satellites or spacecraft also have various subsystems that respond to the real-time parameters. These subsystems need to be simulated as well to predict the behavior of space technology post-deployment. Combining all this together is typically very challenging since either two or more complex system simulations must be intertwined into one, or data from one simulation tool is processed and further inputted into the next tool and so forth which is slow, tedious, and does not allow for real-time simulation. An orbital simulation core or "sandbox" with potential for interoperability with any program can organize and resolve the issue of partitioning complex real-time simulations. Several satellite system simulations of separate external applications may rely on the environmental and kinetic data provided by the orbital sandbox. The orbital simulation core itself has several intra-connected submodules including a physics engine, a graphics engine for visualization, and an orbital computer for certain orbital-specific computations such as orbital element resolution. First order ordinary differential equation computations are performed using Runge-Kutta methods for a high degree of accuracy. Conventionally, graphics lead to expensive pre- and post-processing procedures; however, new technological advancements have enabled the design of a very efficient rendering procedure. The system's general architecture also allows for parallel computing and efficient data storage.

Keywords: Orbital, Simulation, Systems, Real-time, Graphics, Space

Table of Contents

Symbols

Symbol	Meaning	Unit	Mathematical Definition
n	Number of Vertices		
m	Number of Objects		
M	Mass	kg	
G	Gravitational Constant	N m^2 /kg^2	6.6743 x 10^{-11}
r	Position	m	
v	Velocity	m/s	
h	Angular Momentum	kg m^2/s	Eq. A1
μ	Standard Gravitational Parameter	m^3/s^2	Eq. A2
ε	Orbital Energy	J	Eq. A3
N	Direction of Ascending Node		Eq. A4
a	Semimajor Axis	m	Eq. 2.1
e	Eccentricity		Eq. 2.2
i	Inclination	rad or deg	Eq. 2.3
Ω	Right Ascension of Ascending Node	rad or deg	Eq. 2.4
ω	Argument of Perigee	rad or deg	Eq. 2.5
t_p	Time of Perigee Passage	s	Eq. 2.6
θ	True Anomaly	rad or deg	Eq. 2.10
E	Eccentric Anomaly	rad or deg	Eq. 2.9
θ_M	Mean Anomaly	rad or deg	Eq. 2.8
C	Rotation Matrix		
I	Unit Vector		

Orbital and Trans-Planetary Trajectory Simulation System Architecture & Design

The orbital & trans-planetary trajectory simulator's objective is providing a controllably accurate and highly modular n-body simulation core for space system research and applications. This publication explores the planned system architecture and design of the simulator. The scope of the application includes simulation of rigid bodies' kinetics and thermodynamics as well as environmental conditions in outer space. The application will have a UI and simulation visuals for the user to interact with the simulator and view data in real-time. Satellite subsystems and control operations are beyond the scope of this simulator; however, the application intends to provide a local API endpoint for future external modules to interact with the simulators space sandbox.

I | ARCHITECTURE

The central application management system and entry point is referred to as the "main control loop" or "application loop". The application loop of the simulator shall depend on five primary modules: Physics Engine (PE), Graphics Engine (GE), Orbital Computer (OC), Graphical User Interface (GUI), and the Entity Manager. The PE, GE, and GUI do not interact with one another, yet they share certain information through memory controlled by the Entity Manager. The main control loop can modify control parameters or execute public methods of the PE and GE upon receiving input from the GUI. It also executes a render call on every loop cycle.

1.1 System Interconnectivity

A brief high-level overview of the system architecture and module inter-connectivity is visualized in Figure 1.1. The entity manager stores all necessary simulation object parameters, physical properties, kinetic data, and vertex data of static structures. Essentially, it's a managed shared memory pool for the PE, GE, OC, and GUI to utilize data that is updated or created by another module. As an example, the GE needs the position of each object which is computed by the PE and used by the OC for computations and the GUI for information delivery. Instead of creating copies of the same data and having to perform expensive copy operations for each compute cycle performed by the PE, it is far more effective to share the memory in a thread-safe and controlled location.

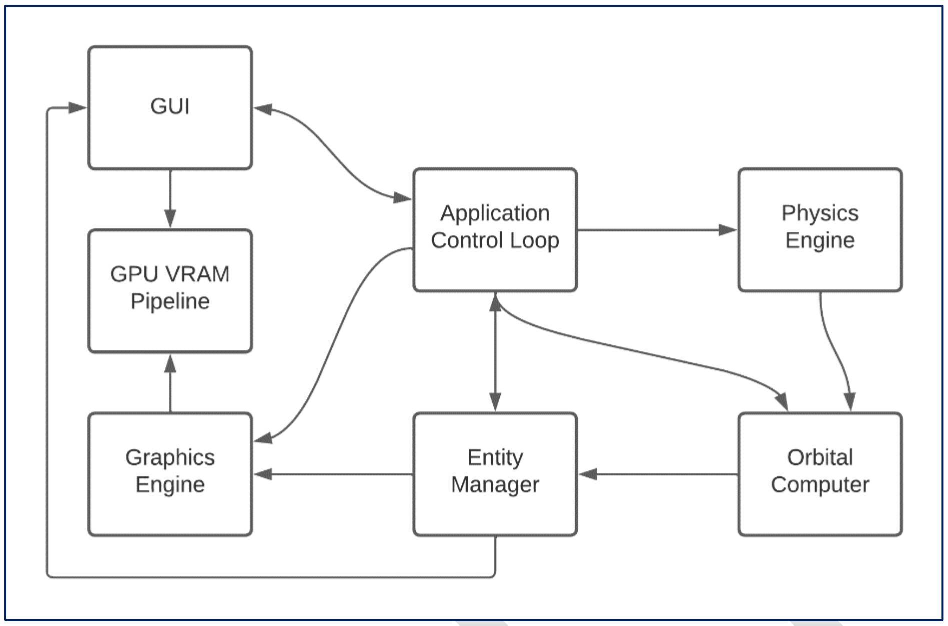

Fig. 1.1. High-level overview of the system architecture partitioning

1.2 API & External Connectors

An API & streaming server will be implemented to allow two-way communication between the simulator and external programs. Data such as system and kinetic information may be delivered to the external program upon request. The external programs may also interact with the API to modify system parameters or apply perturbations to simulation objects. One of many use-cases of this functionality includes applying "perturbations" caused by thruster engagement on a spacecraft or satellite system simulator to perform an orbital maneuver.

II | RESEARCH & DESIGN

Each module has unique methods that are optimized for fail-safe performance, interoperability, and parallel computing with reasonable minimum resource requirements. The following sections delve deeper into the design challenges of each application module. Figure 2.1 demonstrates the overall system architecture as well as the primary sub-components within each module and how they interact with one another.

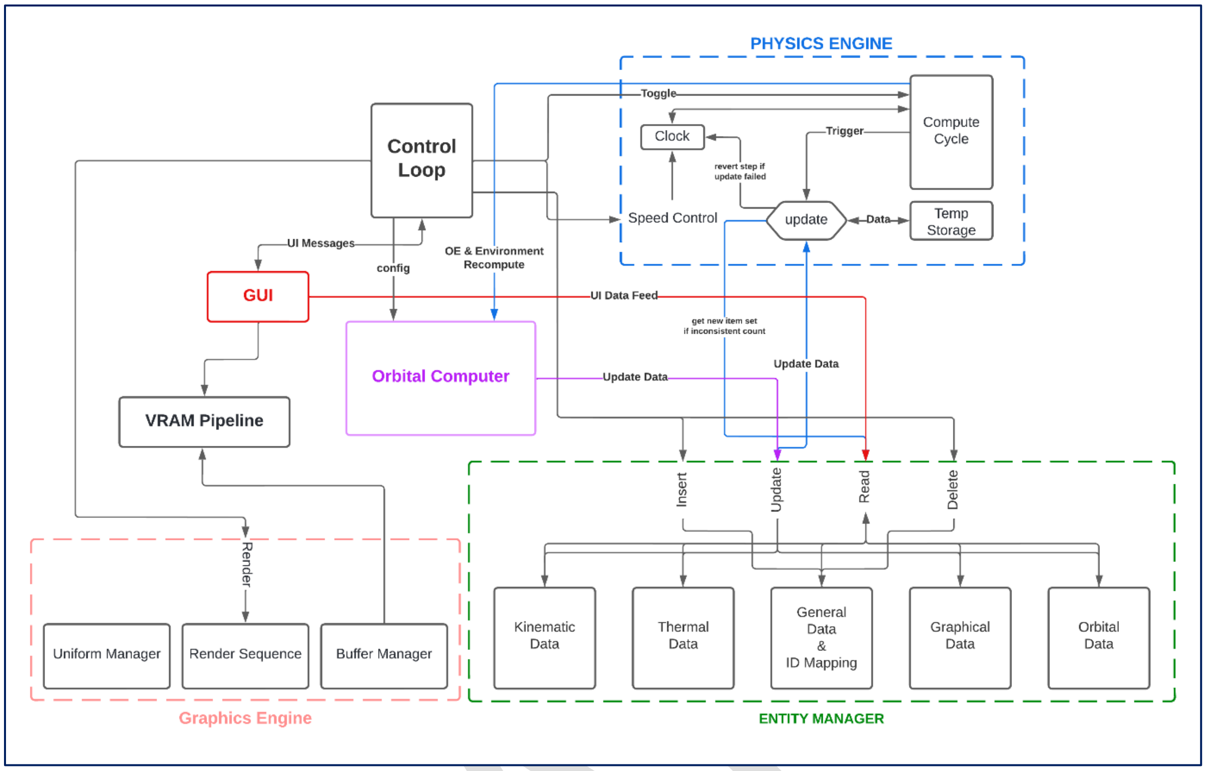

Fig. 2.1. Detailed high-level overview of the system architecture and module sub-components.

2.1 [PE] Physics Engine

The purpose of this module is to perform numerical time-bound computations for several physical properties and motion attributes throughout the simulation. To compute the change in such properties and/or attributes over time, a numerical solver is needed. The engine will allow for real-time or set-time simulation to take place in forward or reverse. Computations will be performed in a "compute cycle" that is disjointed from the main control loop.

2.1.1 Numerical solvers

Several solver options with various accuracy levels will be provided to the user including the fourth order Runge-Kutta (RK4), the Bogacki–Shampine method (RK23), and the Fehlberg method (RK45). Truly infinitely small parameter increment cannot be used to numerically solve relationships with present-day mathematical methods. Furthermore, large increments yield large accuracy error accumulation over time and setting very small increments with Euler's method[1] of numerical integration significantly hinders performance. For these reasons, various Runge-Kutta methods with low truncation error accumulation and different processing speeds are implemented to satisfy the accuracy needs of the user.

2.1.2 Compute mechanism & resources

Historically, numerical solving methods have large compute power requirements. To leverage a computer's CPU to its fullest, the physics engine can instantiate several compute workers in parallel on separate CPU threads with a limit defined by the user. The first and required thread created will proclaim the role of the "lead thread" to perform certain unique operations before and after each compute cycle. This reduces the need for very frequent mutex or atomic operations and sets the overall application requirements to a minimum of two available threads (one for the application loop and a minimum of one for a compute worker).

The compute workers take "jobs" from a queue of simulation entities to update. Once the queue is empty, the lead thread will submit the updated entity data to the Entity Manager and repopulate the queue with the newly computed data as well as any new entities in the Entity Manager added during the completed compute cycle.

2.1.3 Control parameters

The physics engine simulation may be paused, reset, and accelerated or reversed by the user. The number of threads at the disposal of the engine may also be controlled. These parameters are accessed through class methods executed by the main application loop once it receives user execution messages through the GUI.

2.2 [GE] Graphics Engine

To visualize the simulation, millions of vertices need to be rendered by a graphical processing unit (GPU) and updated in real-time. The biggest drawback of rendering a simulation is that an object moves, every vertex position for the object must be translated. Updating every vertex individually would yield an algorithm with a maximum complexity of $O(n^2)$ where n represents the vertex count, and a variable minimum complexity of $mO(n)$ where m represents the number of simulation objects and $m >= 2$ since there must be at least two objects for gravitational interaction. Further drawbacks include the magnitude of outer space and the reflection of light. To avoid complex matrix computations per object, each vertex position is anchored relative to a center of the solar system (the barycenter or the sun). Vertex positions in outer space typically only fit in an 8-byte double typed variable without any sort of truncation. The position data must be downscaled to reasonable 4-byte float values to preserve GPU memory and reduce computation complexity. The reflection of light from object to object must also be considered such as the moon reflecting solar light.

2.2.1 Reducing render procedure complexity

Reducing the maximum and minimum complexity is possible through means of static rendering objects & GLSL GPU shader uniform updates. By presuming that the shape of an object in orbit remains static, its vertices at an anchored position of null remain unchanged throughout the simulation. The only dynamic requirement would be to instruct the GPU shader program on how to translate the static object vertices in 3D space. This can be accomplished with an O(1) operation by setting a position uniform variable in the GPU shader program rather than iterating through each element of the vertex buffer to append position data per vertex. This reduces the CPU-side minimum algorithm complexity to O(n). For the lesser-common scenario where an object's static shape must be changed, a new vertex buffer can still be generated and injected into the GPU.

2.2.2 Consequences and benefits complexity reduction

Due to the limited capabilities of the GPU shader language GLSL, it is not optimal to store object ids within the vertex buffer and have the shader refer to a variable map of positions to ids. This would statically limit the number of objects that can be rendered due to the necessary memory pre-allocation at compile-time. Consequently, every object must have its own vertex array object (VAO) and draw call to set a unique position uniform variable per object which puts more strain on the GPU; though sources [1] & [2] suggest that draw calls are not a significant performance hinderance when limited to a few thousand. Testing must be performed to analyze performance of simulations with large object counts considering that the solar system has over 100 000 objects.

Since every object needs a uniform update and draw call, the rendering procedure complexity is O(m) where *m* is the number of celestial objects in the simulation and $m <= n$ and $m << n$ in most cases unless running a particle simulation which is beyond the scope of this system. In case all static shapes must be updated at once which is a very rare and unusual simulation scenario, the maximum complexity approaches mO(n) which is the minimum equivalent to the complexity when updating every vertex individually.

2.2.3 Failed Approach – VB chunking with metadata headers

An alternative concept explored was inserting the position data at the beginning of each celestial object vertex buffer chunk in a batched VAO; however, this is not possible due to the nature of a GPU's parallel computing architecture that renders fragments in parallel without any potential consideration for previously read data within the buffer.

2.2.4 Future Approach – SSBO dynamic mapping

As of OpenGL 4.3, it is possible to store and interact with data of an arbitrary size within buffer objects on the GPU's VRAM using a Shader Storage Buffer Object (SSBO). According to the Khronos OpenGL Working Group: "This extension provides the ability for OpenGL shaders to perform random access reads, writes, and atomic memory operations on variables stored in a buffer object." [3]. Only one arbitrarily sized array may be stored per SSBO (at the end of the SSBO structure in particular). This is non-limiting for the simulator since the only variable sized quantity is the number of render objects; hence an array of structures may be stored where the structures contain position data and any other object data relevant to the shader program.

This method of data insertion eliminates the need for separate VBOs per object since the object id used as a map to an SSBO slot may simply be appended to each vertex in a single batched VBO. Consequently, only one draw call is needed. Furthermore, the complexity of the CPU-side render loop may be reduced to O(1) if the SSBO's array is precompiled by the OC or GE during each PE compute cycle. The only consequence of doing so is storing duplicate data in RAM; however, the size of the data is insignificant as even with one million objects an estimated 24MB in excess will be used with raw data. If necessary, the excess storage can be halved by preprocessing the downscaled rendering position stored using float values instead of double values.

In conclusion, the rendering method introduced in this section is superior to all other methods explored in this publication due to its potential O(1) CPU-side algorithm complexity and need for only one GPU render call instead of several thousands. This method will likely be implemented in a future version of the simulator since many systems including all MacOS devices have yet to implement support for OpenGL v4.3 as of this document's publication date.

2.2.5 Orbital trajectory rendering

The trajectory of an orbit is always slowly changing due to perturbations and consequently the trajectory loop needs to be re-rendered on every render cycle. This doubles the number of draw calls needed hence the use for a batch rendering approach as described in section 2.2.4. The number of vertices in a line is far less than in a 3D object hence why re-computing the vertex buffer for every render call is not an inadequate solution. The rendering algorithm complexity for the CPU remains the same since each trajectory can be rendered per object iteration.

2.3 [OC] Orbital Computer

The purpose of the orbital computer (OC) module is to compute orbital trajectory elements and analyze upcoming orbital concerns such as environmental hazards. It may also contain other methods that don't qualify to be a part of the GE or PE.

To avoid having to run another O(n) algorithm in parallel to compute orbital elements in real-time, the physics engine worker threads will call OC methods to re-compute orbital elements in real-time without losing the compute cycle concurrency.

2.3.1 Orbital element & trajectory position computation

The computation of certain orbital elements for a given position and velocity can be accomplished through Equations 2.1 – 2.6 [4]. These elements can be used to calculate the object position at any point in the future orbital trajectory assuming no secondary perturbations exist. The trajectory computations help provide a general overview and estimate of the position and environment to come.

$$a = \frac{\mu}{2\varepsilon} \tag{2.1}$$

$$\vec{e} = \frac{\vec{v}\times\vec{h}}{\mu} - \frac{\vec{r}}{r} \tag{2.2}$$

$$i = \cos^{-1}\left(\frac{\vec{h}\cdot\vec{I_z}}{h}\right) \tag{2.3}$$

$$\Omega_0 = \cos^{-1}\left(\frac{\vec{N}\cdot\vec{I_x}}{N}\right) \Rightarrow \Omega = \begin{cases} \Omega_0 & \text{if } \vec{N}\cdot\vec{I_y} \geq 0 \\ 360^0 - \Omega_0 & \text{if } \vec{N}\cdot\vec{I_y} < 0 \end{cases} \tag{2.4}$$

$$\omega_0 = \cos^{-1}\left(\frac{\vec{N}\cdot\vec{e}}{Ne}\right) \Rightarrow \omega = \begin{cases} \omega_0 & \text{if } \vec{e}\cdot\vec{I_z} \geq 0 \\ 360^0 - \omega_0 & \text{if } \vec{e}\cdot\vec{I_z} < 0 \end{cases} \tag{2.5}$$

$$t_p = t_0 - \frac{E - e\ \sin E}{\sqrt{\mu/a^3}} \tag{2.6}$$

Equations 2.7a & 2.7b for object kinematics computation are relative to the true anomaly in orbit. The equations show a vector multiplication with a rotation matrix to get the values in the desired frame of reference. This can be useful when generating the trajectory line vertices for the

GE; however, from a simulation perspective it is far more relevant to know where an object will be after a certain amount of time. With knowledge of the time of perigee passage "t_p", semimajor axis "a", and standard gravitational parameter "μ", the mean anomaly of the orbit can be calculated with Equation 2.8.

$$\vec{r} = \vec{C}_R \begin{bmatrix} \frac{a(1-e^2)\cos\theta}{1+e\cos\theta} \\ \frac{a(1-e^2)\sin\theta}{1+e\cos\theta} \\ 0 \end{bmatrix} \tag{2.7 (a)}$$

$$\vec{v} = \vec{C}_R \begin{bmatrix} -\sqrt{\frac{\mu}{a(1-e^2)}}\sin\theta \\ \sqrt{\frac{\mu}{a(1-e^2)}}\,(e+\cos\theta) \\ 0 \end{bmatrix} \tag{2.7 (b)}$$

Using the mean anomaly and the known eccentricity "e", the eccentric anomaly can be computed with the relation of Equation 2.9 which can further be used to compute the true anomaly of the orbit at the desired orbital time using Equation 2.10. Equation 2.9 is a transcendental equation when the eccentric anomaly is the unknown parameter thus a root-finding algorithm must be used to obtain a solution. Newton's method will be used since it is very accurate and simple to implement for fixed equations where the derivative's form is static.

$$\theta_M = \sqrt{\mu/_{a^3}}\,(t - t_0) \tag{2.8}$$

$$E - e\sin E = \theta_M \tag{2.9}$$

$$\theta = 2\tan^{-1}\left(\sqrt{\frac{1+e}{1-e}}\tan\frac{E}{2}\right) \tag{2.10}$$

2.3.2 Information methods

Methods to compute the position, velocity, energies, and more will be available for several possibilities. Most importantly, to compute the position at a certain mean anomaly of the real-time trajectory is essential to updating a vertex buffer of positions to render an orbit ellipse. Velocities and energies are also essential pieces of information for impact and momentum computations as well as orbital maneuvers.

WORKS CITED

[1] J. van Dortmont, “Optimizing Draw Call Batching Using Transient Data-Guided Texture Atlases,” thesis, 2017.

[2] R. Gesota, “How To Make Your Games Run Superfast By Using Draw Call Reduction,” *TheAppGuruz*, 21-Jun-2016. [Online]. Available: https://www.theappguruz.com/blog/learn-draw-call-reduction-and-make-your-games-run-superfast. [Accessed: 20-Sep-2022].

[3] J. Bolz, P. Daniell, C. Riccio, G. Sellers, B. Merry, and J. Kessenich, “ARB_shader_storage_buffer_object.” The Khronos Group Inc (OpenGL Registry), 28-Apr-2014.

[4] A. H. J. De Ruiter, C. J. Damaren, and J. R. Forbes, “Orbital Elements given Position and Velocity,” in Spacecraft Dynamics and control: An introduction, Oxford: Wiley, 2013, pp. 92–94.

APPENDIX A – VARIABLE DEFINITION EQUATIONS

$$\vec{h} = \vec{r} \times \vec{v} \tag{A1}$$

$$\mu = Gm_1 \tag{A2}$$

$$\varepsilon = \frac{v^2}{2} - \frac{\mu}{r} \tag{A3}$$

$$\vec{N} \cdot \vec{I_x} = N \cos \Omega \quad \text{or} \quad \vec{N} \cdot \vec{e} = Ne \cos \omega \tag{A4}$$