3D Graphics Engine Prototype

Rendering Content for Virtual Reality headsets

COMP3617 Virtual & Augmented Reality Assignment 2022/2023

Piotr Borowiecki (svmm25)
PIOTR.BOROWIECKI@DURHAM.AC.UK

Abstract

This paper presents a prototype 3D graphics engine designed for virtual reality (VR) headsets. The engine is capable of handling perspective projection, object transformations, tracking, physics, and distortion correction. We extended the rudimentary RenderPy engine and incorporated additional functionality, such as a dead reckoning filter, complementary filter, physics simulations, and distance-based collision detection. The engine can perform simple shading, based on vertex colour interpolation.

1 Background

Virtual reality (VR) is an exciting and rapidly growing field that allows users to experience immersive environments that can be both realistic and fantastical. VR headsets provide a window into these virtual worlds, presenting the user with a high-resolution display that tracks their head movements in real-time. To create these virtual environments, 3D graphics engines are used for rendering content and simulating physical phenomena. While using a pre-build engines, such as Unity 3D or Unreal engine allow for quick development, it may limit the originality of experiences. Starting from the ground up, on the other hand, provides greater control and flexibility, but requires a significant implementation burden. The purpose of this paper is to present a 3D graphics engine prototyp capable of rendering content for VR headsets. To achieve this, we started with the rudimentary RenderPy [1] engine and extended it to include perspective projection and object transformations. We then implemented a dead reckoning filter and complementary filter to handle positional tracking using data from an Inertial Measurement Unit (IMU), stored in a provided .csv file. We also incorporated physics simulations to handle gravity and air resistance, as well as distance-based collision detection for falling objects. Finally, we implemented distortion pre-correction. In the following sections, we will describe the methodology used to extend the engine, present the results of our experiments, and discuss the implications of our work.

2 Methodology

2.1 Rendering

The provided RenderPy [1] source code was altered and extended to implemenet dynamic rendering with real time output of the frame buffer on scree instead of output to drive. The decision to remove the ‘`Vector`‘ class from the RenderPy source code and utilize vectors created with NumPy [2] was based on the performance benefits and ease of performing related computations, such as multiplying matrices, which are inherent to the library. While it is possible to implement these features from scratch, retaining a proprietary ‘`Vector`‘ class would necessitate the creation of a proprietary ‘`Matrix`‘ class as well and the defining all relevant operations, which would be redundant given the assignment’s focus on VR. Additionally, NumPy’s underlying implementation and lower-level code enables more efficient vector operations, resulting in faster computation times and rendering. To address the problem of implementing perspective projection, it was necessary to implement the model, view, and projection matrices, utilizing the properties of homogeneous coordinates. In the context of Virtual Reality, perspective projection is preferred to orthographic projection, as it is required for the proper projection of three-dimensional objects onto a two-dimensional surface with accurate scaling and depth perception. The model matrix code can be found in ‘`object.py`‘, while both the view and projection matrices are implemented in ‘`camera.py`‘. By utilizing homogeneous coordinates and implementing these matrices from the ground up, RenderPy was extended to provide the necessary functionality for perspective projection. Translations, rotations, and scaling matrices were implemented for object transformations. These matrices were then utilized to compute the model matrix, which in turn could be composed with the view and projection matrices for efficient transformation of vertices from the model's reference frame to the screen. This allowed for the avoidance of repeated application of transformations for each rotation of the model, with the orientation being stored as a quaternion. However, for physical simulations such as the rotation of headsets due to friction and collisions, it was necessary to compute the world coordinates of the objects by utilizing the orientation, position, and scaling properties of the rendered bodies. The relevant code can be found in ‘`transform.py`‘ and ‘`quaternion.py`‘ files.

2.2 Tracking: Handling Positional Data

The provided dataset consists of 6959 records acquired from a VR headset's IMU, gathered by sequentially rotating the headset from 0 to +90 degrees, and then to −90 degrees around the x, y, and z axes. Data was recorded at a rate of $256Hz$ and includes rotational rate in degrees per second, magnetometer flux readings in Gauss (G), as well as acceleration expressed in units of g (m/s^2), where g represents the acceleration due to gravity at the Earth's surface. Alternative units were anticipated in further development of the engine, and thus, the dataset required appropriate pre-processing. The provided dataset was imported as a .csv file and the necessary pre-processing steps were implemented. The accelerometer's data was normalized, while the gyroscope readings were converted to radians per second. Additionally, the axis were re-labeled so that the up vector is y instead of z. To obtain an orientation quaternion from the gyroscope data, a tilt correction quaternion was calculated and fused with the gyroscopic orientation quaternion using a method suggested by LaValle et al [3]. The relevant code can be found in the ‘`tracker.py`‘ file. With further development in mind, the ‘`Quaternion`‘ class was implemented, including methods for performing common operations, such as addition, multiplication, computing the norm, computing the conjugate, and taking the inverse, as well as converting to rotation matrix. Calculating Euler angles from quaternions correctly is not a trivial task due to ambiguities arising from the existence of numerous conventions. Based on the algorithm proposed by Bernardes and Viollet [4], ‘`toEuler()`‘ method was implemented to account for both, extrinsic and intrinsic rotation types. The ‘`rotationOrder`‘ parameter accepts the values of either ”zxz”, ”xyx”, ”yzy”, ”zyz”, ”xzx”, or ”yxy” (Proper Euler angles), but also the values of ”xyz”, ”yzx”, ”zxy”, ”xzy”, ”zyx”, or ”yxz” (Tait-Bryan angles), representing the order in which rotations are performed. In all cases, the method returns a corresponding triple of *alpha*, *beta*, and *gamma* values, expressing angles of rotation around x, y, and z axis respectively, in radians. Rotations stemming from the intrinsic ”zyx” variant are commonly referred to as yaw, pitch, and roll. The algorithm accounts for the gimbal lock possibility and handles such cases appropriately. The authors claim around 30x speedup when compared with the popular Shuster [5] method from 1993. An additional ‘`toQuaternion()`‘ function was implemented. Sarabandi et al. [6] provide a thorough overview of existing methods, but the rotation matrix to quaternion conversion approach was followed. All relevant classes and functions can be found in the ‘`quaternion.py`‘ file.

2.3 Tracking: Camera Pose Calculation

A dead reckoning filter was implemented using the gyroscope-measured rotational and the gravity-based tilt correction using the accelerometer data. The current orientation was calculated by using a previously determined orientation, starting at the identity quaternion, and re-evaluating that orientation based on an estimated angular velocity over the elapsed time. The initial orientation $q[0]$ was considered to be the identity quaternion. Accelerometer information was then included in the computation by transforming acceleration measurements into the global frame. The tilt axis was calculated, and the angle between the up vector and the vector obtained from the accelerometer was found. The complementary filter was used to fuse the gyroscope estimation and the accelerometer estimation. Upon firing up the engine, the virtual camera rotates based on the fused input data. To investigate the effect of drift compensation, different α values were tried, and their impact on drift was noted. The experimentation with different alpha values revealed that higher values lead to unstable motion due to noise present in the accelerometer data, which results in an unnatural and unpleasant experience for the user. It was observed that a trade-off must be achieved between accurately representing the motion and preserving the natural appearance of the movement. Employing a low-pass filter on the accelerometer data could potentially mitigate the effects of noise, although this approach was not attempted in this study.

2.4 Distortion Pre-Correction

Distortion pre-correction was implemented by accounting for the distortion introduced by the headset lenses. The inverse of the provided formula for the distortion pre-correction was approximated using the method suggested by LaValle (http://lavalle.pl/vr/node211.html), with the values of $c1$ and $c2$ set to 5 (equation no. 7.17). The distortion effect is clearly visible in the rendered video.

2.5 Physics

The Euler scheme was utilized to simulate the environment with a timestep of two steps per rendered frame. Collision detection was performed in two stages: a broad phase that implemented the sweep and prune algorithm on the objects' Axis-Aligned Bounding Boxes, followed by a narrow phase that investigated each candidate collision to determine the point of contact between the bounding spheres, which are calculated using Ritter's algorithm which provides a good balance between speed and nearly optimal radius. The point of contact was then used to apply the impulse to the bodies involved in the collision, which modified their linear and angular momenta. This allowed for the simulation of rotational effects, using the inertia tensor that was initially computed for the model and then transformed to the world coordinates frame using the object's model matrix. Air friction was calculated by applying forces to faces, taking into account the total (linear + angular) velocity of the face's midpoint, as well as its area in the world frame. This information was then used to compute forces and torques to determine the total contribution to momentum and angular momentum.

3 Results and Limitations

The engine successfully renders, allowing for synthesizing separate frames into a video using external software. 1,800 static frames for different orientations of the camera can be found in the attached output folder, demonstrating that implemented tracking and transformation matrices work as expected.

4 Discussion

4.1 Simple Dead Reckoning vs Including the Accelerometer

One limitation of using dead reckoning filter alone is that the tilt might be misrepresented because of the lack of corrective information from the outside world. While simple dead reckoning can be a useful method in situations where other sources of information are not available, including accelerometer data can significantly improve the accuracy of the position estimation.

4.2 Performance

Performance is satisfactory.

4.3 Positional Tracking

Positional tracking could be estimated using the accelerometer data, since it measures the acceleration of the headset.

4.4 Collision Detection using Spheres

Innacurate, but very fast.

4.5 Distortion Correction with GPUs

The provided code is very easy to parallelised. GPU shaders could easily perform the computation in parallel. In fact, this is a preferred approach, but the implementation was outside the scope of this project and constitute further work.

References

[1] Ethan Cannizzaro. *RenderPy*. Accessed on April 8, 2023. 2019. URL: `https://github.com/ecann/RenderPy`.

[2] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: `10.1038/s41586-020-2649-2`. URL: `https://doi.org/10.1038/s41586-020-2649-2`.

[3] Steven M LaValle et al. "Head tracking for the Oculus Rift". In: *2014 IEEE international conference on robotics and automation (ICRA)*. IEEE. 2014, pp. 187–194.

[4] Evandro Bernardes and Stéphane Viollet. "Quaternion to Euler angles conversion: A direct, general and computationally efficient method". In: *Plos one* 17.11 (2022), e0276302.

[5] Malcolm D Shuster et al. "A survey of attitude representations". In: *Navigation* 8.9 (1993), pp. 439–517.

[6] Soheil Sarabandi and Federico Thomas. "A survey on the computation of quaternions from rotation matrices". In: *Journal of mechanisms and robotics* 11.2 (2019).