Virtual and Augmented Reality Report

Vincent Kenny

Rendering

Static Renders

I implemented the custom function renderVector() to display important vectors, with the activation code left commented out in the getYawCorrectionQuaternion() and getTiltCorrectionQuaternion() functions of render.py.

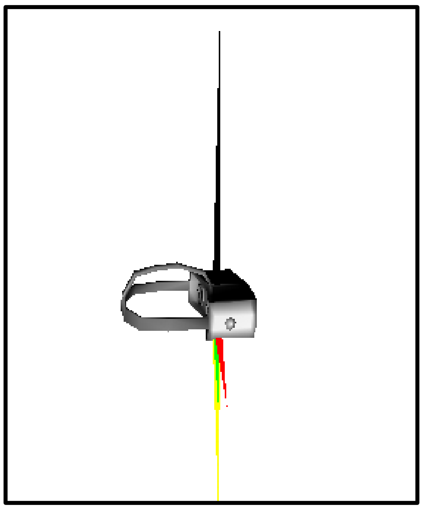

Figure 1

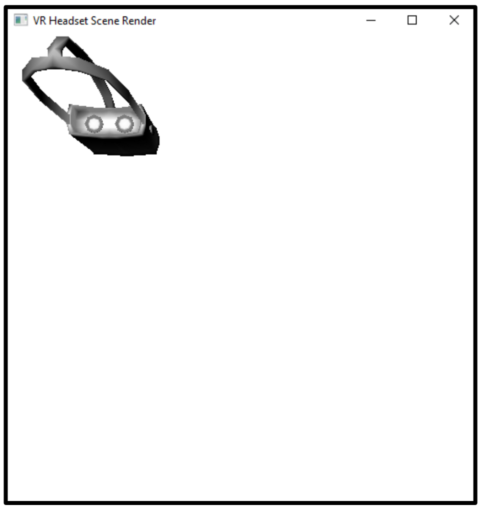

Figure 2

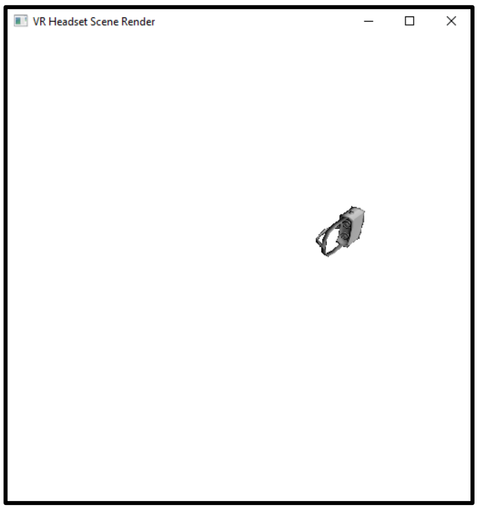

Figure 4

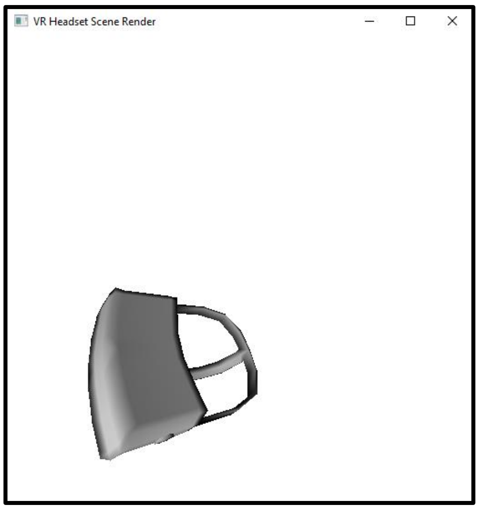

Figure 3

Figure 1 displays key vectors: a black triangle for the up-vector, a yellow triangle for the magnetic field, a green arrow for the yellow's horizontal projection, and a red arrow for north. The headset constantly attempts to align the green vector with red.

Figures 2 to 4 show different perspectives of the primary headset model: demonstrating the correct implementation of the transformation matrices.

The getPerspectiveProjection() function was extended to return properly transformed z values for the Z-buffer, so that polygons will correctly cull based on camera position.

Tracking

Axis Correction

Due to the orientation of the IMU on the headset, an axis correction had to be performed when reading in data. The following mapping gave the expected results:

- x-axis -> z-axis
- y-axis -> x-axis
- z-axis -> y-axis

A separate axis correction was required for the Madgwick filter. This will be explained in its own section.

Yaw Correction Implementation

Yaw correction activates only when the horizontal magnetic field vector is within 5% of the reference vector's magnitude, reducing errors from tilt accumulations. The magnetic field vector predominantly points downward, with a smaller horizontal component. If the pitch of the headset is significantly off, then in the process of converting the magnetic field vector into the global frame by applying the body transformation, the resulting vector can end up pointing straight down. In this case the horizontal component can change directions very rapidly, due to having a very small magnitude. As a result of fewer frames in which to perform yaw correction, the gain coefficient had to be increased from 0.01 to 0.02 to maintain a sufficiently low correction delay. Through experimentation I found that a gain coefficient much higher than 0.02 would cause some degree of jitter when the headset was at rest: Therefore, gainCoefficient = 0.02 was used for both tilt and yaw correction.

I also averaged magnetometer readings over the first 200 frames to establish a reference north, benefiting from minimal drift due to the headset's relative stability for the first 700 frames.

Tilt Correction Implementation

Similarly to yaw correction, I found that rapid rotation of the headset lead to extremely inaccurate estimates of the headset's up vector. This is inevitable when using accelerometer data, as the component of acceleration from gravity cannot be separated from all other applied forces. Applying a tight threshold check for the up-vector magnitude, within 2% of g (9.81 ms^{-1}), slightly improved tilt accuracy. However, the problem was that almost all frames during a rotation of the headset had unusable up vectors, and therefore no tilt correction. Once the headset had come to a rest it then had to be corrected quickly, and this would likely cause sickness for the user who was wearing the headset. To combat these limitations, I chose to implement the Madgwick filter [1].

Madgwick Filter Implementation

I implemented the Madgwick filter [1] to enhance tilt and yaw corrections, smoothly integrating gyroscope, accelerometer, and magnetometer data. My implementation is based off Madgwick's original C code, available at [2], which had to be translated into python, and then adapted to work in a coordinate space with a different set of axes. This algorithm uses a gradient descent approach to minimise the error between the orientation from integration of gyroscope readings, and the orientation as measured by the magnetometer and accelerometer. Madgwick's equations work under the assumption that the z-axis points up, the x-axis points forward, and the y-axis points left from the headset's perspective. This is the same coordinate system in which the IMU data is expressed. However, the coordinate system in our renderer is defined with y pointing up, x pointing to the right, and z pointing out of the screen: a translation layer is needed. The quaternion which represents the mapping from the sensor's coordinate system to the renderer's coordinate system is toRenderAxesQuaternion = (-0.5 + 0.5i + 0.5j + 0.5k). Therefore, when we pass the model's orientation to the Madgwick filter, we instead pass:

toRenderAxesQuaternion.conjugate() * modelOrientationQuaternion * toRenderAxesQuaternion

which converts the model's orientation into the IMU's coordinate system. Similarly, when the Madgwick filter returns our deltaQuaternion for rotating the model, we instead return:

toRenderAxesQuaternion * deltaQuaternion * toRenderAxesQuaternion.conjugate()

to convert this rotation into the same coordinate system as the model. The result is a correct sequence of rotations "from 0 degrees, to +90 degrees and then to -90 degrees around the X, Y and Z axes".

The advantage of the Madgwick filter over traditional tilt and yaw correction methods lies in its handling of the accelerometer's up-vector, which often deviates from true vertical during lateral rotations. It's challenging to distinguish accurate readings without context, even using a filter to verify vector magnitudes close to g (9.81 ms^{-1}). The Madgwick filter combines the smoothness of dead reckoning with the enhanced precision from tilt and yaw correction.

The beta parameter in the Madgwick filter functions like the gain coefficient in tilt and yaw corrections, managing responsiveness to changes in sensor data. As both accelerometer and magnetometer data are considered together for the Madgwick filter, I found that a higher value of 0.05 could be used without introducing jittering from over correction.

Dead Reckoning vs Including Accelerometer

To measure the performance of our accelerometer-based tilt correction over a simple dead reckoning filter, I added the tiltCorrectionAngle from each frame into a list. Taking the mean of these values would tell me, on average, how far off the true tilt we were over the render.

The average tilt angle error without accelerometer-based correction was 9.53 degrees. Using accelerometer-based tilt correction, and a gain coefficient of 0.02, the average tilt angle error was 2.22 degrees. The total render time using only dead reckoning, rendering every 10th frame, was 271.7 seconds. Using gravity-based tilt correction the total render time was 273.1 seconds. This corresponds to a 0.5% increase in render time which, given the significant improvements in tracking accuracy, I would consider a worthwile performance tradeoff. This data demonstrates the effectiveness of integrating accelerometer data into the tracking system, compared to a simple dead reckoning filter.

Upon visual analysis of the renders for dead reckoning, and dead reckoning plus tilt correction, you could be mistaken in thinking that the simple dead reckoning looks better. This is because the movemenet is extremely smooth: with no reference points to conform to, the headset is free to rotate seamlessly. However, in the case of VR where there is a user wearing the headset, the advantage of "smoother" turning would be outweighed by the increased motion sickness, which is common when pitch and roll are innacurate. The render for simple dead reckoning demonstrates the result of this drift error most clearly towards the end of the video: the headset is quite clearly tilted sharply to its right, as opposed to the other three renders where the headset finishes in a near upright position.

Expected Number of Calculations

For analysis, I will consider the number of fundamental mathematical operations, such as * or sin(), in each algorithm.

Starting with dead reckoning: Calculate the gyroscope vector's magnitude and multiply by the time step for the rotation angle. Dividing the vector by its magnitude identifies the rotation axis. Assuming one operation each to square elements, add them, and take the square root gives 6 operations. Finding rotation angle brings this to 7 in addition to 3 operations to normalise the gyroscope vector using the magnitude. Determining the quaternion (w + xi + yj + zk) with w = cos(rotationAngle / 2) and x, y, z = sin(rotationAngle / 2) * rotationAxis adds 6 more operations, totalling 16 per frame for dead reckoning, making it extremely computationally efficient.

Including accelerometer correction adds further complexity. To start, we need two quaternion multiplications to convert the up-vector into the global coordinate space. Each of these quaternion multiplications involves 28 total operations. Tilt correction axis requires a single negation. To find the tilt correction angle requires a dot product calculation, which is 5 operations, and 2 further for negative arccos(). Normalising the tilt correction axis takes 9 operations and then, similarly to dead reckoning, finding the corresponding quaternion takes 6 more operations. This sums to 79 total fundamental operations which, although much higher than dead reckoning, is still sufficiently low to be fast enough for use in a realtime VR environment.

Positional Tracking

While positional tracking of the headset is possible, the accuracy will be poor without the use of further sensors. The only IMU reading that gives us any information about the headset’s translation in 3D space is the accelerometer, which gives acceleration in ms^{-2}. The approach for converting this into a displacement would involve a double integration. The drift error from a numerical integration approach would not only be non-correctable without any reference points, but would grow at a polynomial rate of x^2. This would cause the positional estimate to drift faster and faster over time, as opposed to our single gyroscope integration, in which the error grew linearly. To correct for this, there would need to be some reference point for the headset’s position within the room, similarly to that provided by the accelerometer and magnetometer for tilt and yaw. Modern VR devices use cameras in the headset, along with computer vision techniques, to track their position relative to fixed reference points in the user’s local space.

Physics System

Implementation

The headset which starts off to the right has its mass set at 0.1kg, unlike the others at 1kg, to demonstrate collision dynamics and air resistance effects on varying masses.

The two headsets that move directly away from or toward the camera are designed to showcase the LOD switching system. As a result, both headsets have gravity turned off and do not experience any air resistance. The main headset in the centre of the screen has its experienceCollisions property set to False, giving it effectively infinite mass.

If an object with a negative mass collides with an object with an equal and opposite positive mass, they will annihilate. This has no application to the render for this coursework, but I’m sure it would be appreciated by some physicists.

Within the physicsEnvironment class, the coefficient of restitution was set to 0.5. This gave a visually consistent render with an appropriate "bounciness" for all headsets.

Collision Detection

Spherical bounds provide efficient collision detection by simply comparing the vector magnitude between object centres to the sum of their radii. However, there is still room for improvement in my detection algorithm. For the relatively low number of headsets in my simulation it was sufficient to check the position of every object from every other object in each frame. This grows with complexity (n * (n – 1)) / 2 for n objects, so for the seven headsets in my render there were 21 comparisons per frame. If the number of headsets became large, then it could necessitate the use of a more advanced collision detection algorithm such as sweep and prune. This algorithm is effective in unbounded or infinite spaces and whilst its worst-case complexity is also O(n^2), its average case is closer to O(n * log(n)) for relatively even object distributions.

An issue with using spherical bounds for collisions is that it does not accurately represent the geometry of the headset model. The ratios of the lengths of the headset's bounding box in the x-y-z dimensions are approximately 16:10:22, found using Render -> Show Box Corners in MeshLab. From this we can see that the headset is over twice as long as it is tall, hence a spherical bounding shape will always either protrude or come short in one part of the model. This will cause models to either pass through each other on collision, or have visible space between them, both of which will break immersion. A more suitable bounding shape would either be a cuboid, or a more intricate shape that closely follows the model's geometry. This improved simulation accuracy would come with an increased computational demand for computing intersection between two bounding shapes, and the resulting forces applied to the colliding objects.

Level of Detail (LOD)

Implementation

I found that a full-scale simplification of the model in MeshLab caused inconsistent, and noticeable differences in lighting on the front face of the headset. Due to the lighting intensity on a face being a gradient between the intensity at each of its vertices, removing the polygons whose vertices all lie on the front of the headset resulted in the absence of completely bright faces when the headset faced the light source directly. Instead, only the vertices were brightly lit, which produced a noticeably different appearance and made the LOD switching appear much more abrupt. To fix this I deselected the front eight polygons before performing the quadratic edge collapse decimation. This maintained the front on lighting appearance in exchange for a slight loss of detail elsewhere in the headset.

Performance

The full 6959 frame render, without using the LOD system to reduce quality of distant models, took 56.65 minutes on my desktop machine. With LOD enabled, the same render took just 34.22 minutes, a 39.6% decrease in average frame generation time. This render time corresponds to an average time per frame of 0.488 seconds without LOD, and 0.295 seconds with LOD.

Improvements

I have already implemented near and far plane culling, setting the far plane at z = -150, a distance where headset models occupy just one pixel, leading to a smooth pop-out effect as models pass this boundary. To improve performance of our renderer further, it could be advantageous to implement view frustrum culling, where any model which lies entirely outside of the visible space is not drawn at all by the renderer. Due to gravity in the physics simulation, most of the headsets will fall out of the bottom of the frustrum and never come back into view but will remain within the near and far planes. Therefore, culling these models could save a significant amount of compute time per frame.

As a model moves away from the camera, the quality of lighting could be reduced. The renderer uses vertex colour interpolation to smooth the lighting transition between adjacent polygons. For distant models, where each polygon may be smaller than a pixel, a more simplistic flat shading technique would reduce lighting complexity, with minimal loss of visual quality.

Whilst not particularly important for this render, due to all headset models having equal size, a screen space LOD system could reduce render time for smaller objects. The detail of a model would be reduced according to the number of pixels that it takes up on the screen, as opposed to its distance from the camera. Such an approach promotes a more seamless and subtle transition in quality: larger models would maintain higher levels of detail at greater distances, owing to their increased visibility.

References

[1] Madgwick, S., 2010. An efficient orientation filter for inertial and inertial/magnetic sensor arrays. Report x-io and University of Bristol (UK), 25, pp.113-118.

[2] https://x-io.co.uk/open-source-imu-and-ahrs-algorithms/