CS 1566 Final Project Report:

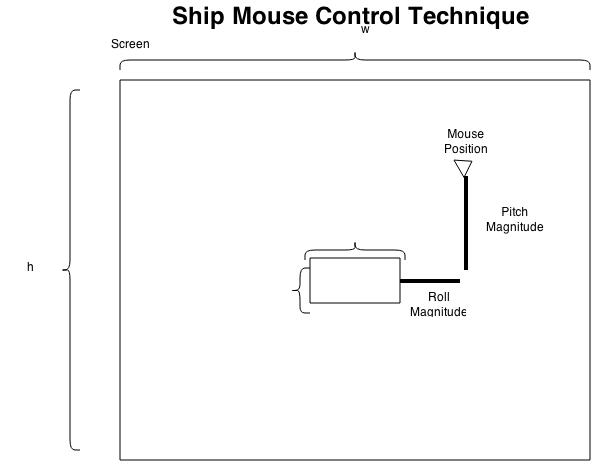
1. **Introduction**

Our project is a simulation of a procedurally generated planetary system. The user is allowed to fly through this system using a freely moving camera that can be turned and accelerated like a spaceship.

The spaceship has a 2D reticle overlaid onto the 3D view, which gives metrics about orientation and speed to the user.

1. **Camera and Skybox**

Implementation Technique



**Space** – Stop Motion (full stop)

**W** – Accelerate Forward

**S** – Accelerate Backward

**D** – Yaw Right

**A** – Yaw Left

Mouse Move Up – Pitch Up

Mouse Move Down – Pitch Down

Mouse Move Left – Roll Left

Mouse Move Right – Roll Right

Mouse Motion Over the window is used for looking up/down and rolling and is calculated as an offset from the center, excluding a small dead zone where the mouse can rest with no motion applied.

The mouse triggers movement as long as it is not located in a dead-zone in the center of the window.

Mathematically, all of the camera movement was accomplished by utilizing a 3x3 Rodrigues rotation matrix to rotate a vector (or a point) around another vector. These simple operations, when combined with the mouse, created a very intuitive user control scheme.

* Pitch was implemented by taking the cross product of the look direction and up vectors, and rotating about the resultant axis.
* Roll was implemented by rotating the up vector around the look direction vector for the camera.
* Yaw was implemented by rotating the direction vector around the up vector.

These calculations led to values for the gluLookat call which properly oriented the ship in the correct position.

1. **User Interface**
2. **Planetary Movement**

Planets in this simulation move in a manner that approximates Keplerian motion. Since our simulation does not incorporate mass and gravity, motion is along an ellipse following the equal time equal areas principle from Kepler’s second law. Precisely calculating the areas would require expensive integration, so I instead calculate areas using trigonometry. This introduces some error as I end up underestimating the area of each slice, but it results in motion that shows planets slowing as they move further from their orbital focus, and faster when they move closer.

Orbits are also defined by a normal, after the position from the focus is calculated, the point is rotated about the axis that results from the cross product of the default orbit plane normal which is (0,0,1) and the normal defined for that orbital plane.

All planetary positions are calculated separately from the rendering sequence, this avoids relying excessively on the matrix stack and better facilitates tasks like Raytracing.

In addition to the orbits, the planets rotate about their own axes, this is handled by storing a randomly generated rotation axis and incrementing an angle based on a randomly generated rotation rate

1. **Texturing and Planet Appearance**