**Initial Setup:**

Windows Instructions:

Step by step guide for setting up Panda3D in VS Code on Windows

MacOS Instructions:

Step by step guide for setting up Panda3D in VS Code on Windows

**Basic Programs:**

Movement.py:

Enables camera movement. Move the mouse to look around (clamped so it can’t look more than straight up or straight down). Control the translation of the camera by w: forward, s: backward, a: left, d: right, c: down, space: up. All movements are relative to the current rotation of the camera. Also has a basic model of a landscape. All other programs (unless specified otherwise) contain this functionality (movement and landscape).

**Sphere Collision:**

SphereRendered.py:

Creates a sphere made from squares. Supports changing the radius, position, and precision (how many rectangles are used to represent the sphere).

Sphere+Movement.py:

Same as SphereRendered.py but supports movement of spheres. There is a physicsUpdate function that is called at at “fixed” time steps and the spheres store their own velocities. Ever physicsUpdate the spheres are moved by velocity\*deltaTime. As far as I can tell, the way that panda3D handles delays for functions is to start the delay after the function has finished running (e.g. if physicsUpdates is set to 50fps, and it takes 10ms to run, it will be called every 30ms). That is why we use the built in clockObject instead of relying on the specified delay.

2SphereCollision.py:

Simulates the collision of 2 spheres. Supports setting the position, radius, precision (of render), mass, and velocity of the spheres. Not 100% sure that the mass works correctly in all situations.

nSphereCollision.py:

Same as 2SphereCollision.py except supports a list of spheres instead of 2 spheres. Checks for collisions between all pairs of spheres (O(n^2)).

SphereCollisionDisplay.py:

10 simulations that are looped through. There are 4 spheres of set sizes that are set to specific positions and velocities every 8 seconds. This was used as a display piece at the coop employer appreciation breakfast. Camera control is disabled, the camera automatically moves in a circle.

**Box Collision 3D:**

BoxRendered.py:

Box that can have a set scale, position, and rotation.

BoxMovement.py:

Same as BoxRendered.py, but support velocity and rotational velocity that are updated every physicsUpdate frame. Also keeps track of the positions of the vertices and the faces of the box. Finding the vertices is done using a matrix.

BoxMovement-Trig.py:

Same as BoxMovement.py except uses basic trig functions to keep track of the vertices of the box (instead of matrices). Inefficient and messy - may be useful for understanding what is actually happening when using the matrices.

CollisionDetection-Matrix.py:

2 boxes, which can be given a size, position, and rotation (each a vector 3). If these boxes ever come into contact with each other, it will be detected, the boxes will stop moving and rotating, and the type of collision will be printed to the console. Both edge-edge and point-face collision will be detected (other types of collision have an infinitesimally small chance of occurring). See ColisionDetection-Trig for more details on the differences between them.

CollisionDetection-Trig.py:

Same as Collisiondetection-Matrix.py except uses a different method to keep track of the vertices of the boxes. In order to detect a collision, we need to know the positions of all eight vertices of the box. When the box is translated or scaled (not currently supported), it is relatively easier to do. In order to keep track of the vertices when the box is rotated, CollisionDetection-Matrix.py uses a matrix. Panda3D stores the matrix of the box object which can be accessed through .getMat(). This matrix can then be applied to each of the vertices. This program finds these points using basic trig functions (details can be found in the comments of the code). This approach is far less efficient and significantly messier, but may be helpful to understand what is actually going on.

**Box Collision 2D:**

2DMovement.py:

Similar to Movement.py except for 2d. Camera uses an orthogonal lens, mouse controls do not move the camera, and wasd are used for up, down, left, and right respectively. Contains a default landscape in the background to serve as a point of reference. All 2D scripts contain this functionality unless specified otherwise.

2DSquareNoMovement.py:

Renders a square to a screen with a set scale, position, and rotation.

2DSquare.py:

Square with scale, position, and rotation, as well as velocity and rotational velocity updated every physics frame.

2DBoxCollision.py:

Detects the collision of two boxes. Only one case is needed in 2D (point-edge). This is very similar to the 3D point-face case (except only checking 2 dimensions). If the boxes collide they will stop moving.

**Cube on Plane:**

RampForces.py: Detects collision of the cube with plane, calculates the bounce by assuming perfectly elastic collision, amount of damping can be adjusted, currently set to subtract 2/3 of vertical velocity. Collisions are no longer detected if the cube is touching and velocity is in the same direction as the ramp.

RampImpluse.py: Detects collision of cube and plane, assumes no bounce by giving impulse to cancel out normal velocity to the ramp. Calculates friction by assuming gravity, normal force and friction are the same, then checking if it has exceeded the limit of frictional force by using mu.

**Pendulum:**

Pendulum v1.py (Conventional Forces): Calculates the position of the sphere using forces. Not perfect since it is not fixed to a point so drifts down slowly over time.

Pendulum v1.1.py (Angular Velocity and Acceleration): Calculates the position of the sphere using angular acceleration and velocity.

**Other:**

Course Topic Ideas:

List of things that were considered for possible inclusion in course.