

Project 2 Report

Task 1: Create a Basic Scarecrow

Transforming Each Part

Head

- To do this, I translated upward so that the head sits on top of the torso (e.g., `glTranslatef(0, 12.5, 0)`). My reasoning was that the torso height is approximately 10, so halfway plus the sphere's radius gave about $y = 12.5$ (top of torso + radius).
- I used `gluSphere(quadratic, self.head_sphere, 32, 32)` to draw the head (line 219)

Nose

- Since the nose had to attach to the front of the head, I translated to the head's center and then forward along z by the radius of the head.
 - For instance, `glTranslatef(0, 12.5, self.head_sphere)`.
- The nose rotates around the y -axis by `head_angle` (if testing out Task 2).
- `gluCylinder(quadratic, 0.3, 0.0, 1.8, 32, 32)` (line 227) draws the conical nose.

Torso

- The torso of the scarecrow is a vertical cylinder. By default in OpenGL, a cylinder created with `gluCylinder()` extends along the local z -axis from $z = 0$ to $z = \text{height}$.
- However, for the scarecrow to appear upright in the 3D scene, the torso had to be aligned along the global y -axis (upward direction). I applied a rotation of -90° around the x -axis.
- Specifically, I used:
 - `glRotatef(-90, 1, 0, 0)` (line 232), which performs a counter-clockwise rotation about the x -axis.
- After the orientation was corrected, I drew the cylinder using:
 - `gluCylinder(quadratic, 2.5, 2.5, 10, 32, 32)` (line 234), which creates a cylinder with both base and top radius 2.5 and a height of 10.
- This resulted in a vertical cylindrical torso that stands upright and connects naturally to the head above and legs below.

Legs (Right and Left)

- I did a horizontal translation ± 1.2 along x to separate the left and right legs.
- I rotated 90° around x so that each cylinder would extend down the y -axis.
- Because each leg's height is 12, I placed them on top of the leg so that they would meet the torso at $y = 0$. Thus, I could see the bottom at negative z in local coordinates (but effectively negative y in global, after rotation).

Arms (Right and Left)

- I translated the arms near the top sides of the torso (e.g., $x = \pm(\text{torso radius}), y \sim 9$).
- I rotated the arms by $\pm 90^\circ$ around the y -axis so that the OpenGL cylinders (which by default extended along the z -axis) instead extended sideways (along the x -axis).
- This resulted in a pair of cylinders sticking out at shoulder level.

In general, I used `glPushMatrix()` at the beginning and `glPopMatrix()` at the end of each body part so that the transformations were isolated and didn't interfere with other parts. The torso was the center/anchor of the body. I put the head above the torso using a vertical translation and rotated it independently around the y -axis to simulate turning. I put the nose forward along the z -axis from the head center, and it uses the same rotation as the head. I positioned the arms by translating sideways from the torso and rotating 90° around the y -axis to align along the x -direction. Similarly, I offset the legs from the center and rotated them to align downward. I chose this specific order for creating body parts because of the scarecrow's logical hierarchy, from head down to limbs. For all of the body part transformations, I rotated before translating since OpenGL reverses the applied order.

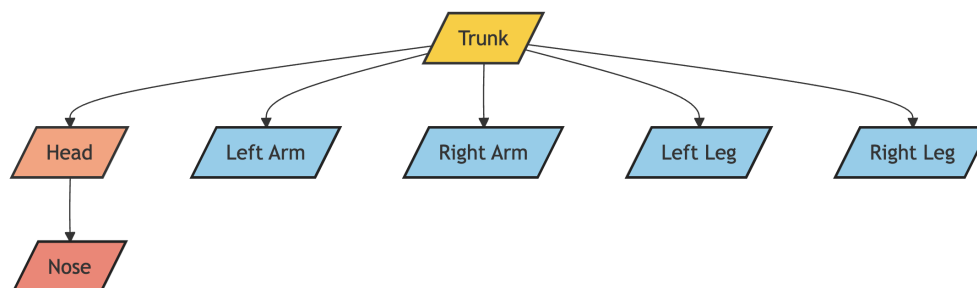


Figure 1: Order of Scarecrow Body Part Creation

Task 2: Rotate the Scarecrow's Head & Nose

The goal for this task was to allow only the head and nose to rotate left (by pressing “l”) and right (by pressing “r”) by up to $\pm 85^\circ$.

Rotating the Head and Nose

OpenGL Functions Used

- I used `glPushMatrix()` to save the current transformation state and isolate the head and nose transformations.
- I applied `glRotatef(self.head_angle, 0, 1, 0)` to rotate the local coordinate system around the y -axis by the head angle (in degrees).
- I used `glTranslatef()` after the rotation to position the head above the torso and the nose in front of the face.
- I rendered the head with `gluSphere()` (line 298) and the nose with `gluCylinder()` (line 303).
- Finally, I called `glPopMatrix()` to restore the previous transformation state so that other body parts remained unaffected.

How it integrated into Task 1 code

- In the `draw_Scarecrow()` method, I created two separate transformation blocks: one for the head and one for the nose.
- In both blocks, I started with `glPushMatrix()` and applied the same `glRotatef(self.head_angle, 0, 1, 0)`.
- For the head, I translated it upward with `glTranslatef(0, 12.5, 0)` (line 296) so it sat on top of the torso, then drew it using `gluSphere()`.
- For the nose, I also applied the same rotation, then translated it forward along z using the head's radius (e.g., `glTranslatef(0, 12.5, self.head_sphere)`) and rendered it with `gluCylinder()`.
- Since both transformations were in their own matrix stack and used the same rotation, the nose stayed attached to the head and moved with it.

Keyboard Input Integration

- I declared a variable `self.head_angle` in the `Scarecrow` class to track the rotation of the head.
- In the keyboard input handler, I mapped:
 - Pressing the “i” key to decrement `head_angle` (rotating the head left).
 - Pressing the “o” key to increment `head_angle` (rotating the head right).
- I also clamped `head_angle` to stay within -85° and $+85^\circ$ to prevent excessive rotation.
- During rendering, I passed the current `head_angle` to `glRotatef()`. This allowed the head and nose to rotate smoothly upon keyboard interactions.

Task 3: Upgrade the Scarecrow (More Joints, Hands, Feet)

The goal here was to use a more sophisticated scene graph so the scarecrow could have flexible arms and legs with visible joints.

Upper Arm & Lower Arm

- Each arm was split into two segments, both half the original length (e.g., $5 + 5 = 10$ units total).
- I put a small joint (sphere) between the upper and lower arm to represent the elbow.
- The upper arm was first translated to the shoulder ($x = \pm 2.5$, $y = 0$, $z = 10$) and then rotated outward (e.g., $\pm 15^\circ$ around y) and downward/upward based on walking animation (x -axis).
- I then rotated the lower arm separately (if arm was moving backward) and drawn forward from the elbow.
- I drew a hand sphere (radius ~ 1.1) at the end after another forward translation.

Upper Leg & Lower Leg

- I also split each leg into upper and lower segments of length 6 ($6 + 6 = 12$).
- The upper leg was positioned slightly outward ($x = \pm 1.2$) and rotated forward/backward around the x -axis to simulate walking.
- A joint sphere (knee) was added at the end of the upper leg.
- The lower leg continued from the joint, with optional rotation applied if the leg was swinging backward.
- I put the foot at the end of the lower leg and drawn using `glutSolidCube(1.5)` scaled by `glScalef(1, 1.8, 0.8)` to approximate a flat foot shape.

Hands & Feet

- I added hands as green spheres (radius ~ 1.1) attached to the end of each lower arm.
- I made feet as cubes scaled to resemble shoes (shorter in height and wider in length) placed at the end of the lower leg path.

Scene Graph Logic

- *Arms:* Started from the torso's top/side $\rightarrow$ rotated upper arm $\rightarrow$ joint sphere $\rightarrow$ rotated and translated lower arm $\rightarrow$ hand.
- *Legs:* Started from the pelvis $\rightarrow$ rotated upper leg $\rightarrow$ knee joint $\rightarrow$ rotated and translated lower leg $\rightarrow$ foot.

Typical Transform Steps for Arms and Legs

The steps below outline how I transformed each limb in the upgraded scarecrow. I applied these steps symmetrically to both sides and followed the structure as shown in the scene graph.

1. I translated to the base attachment point (e.g., shoulder for arms, hip for legs).
2. I rotated around the y -axis to angle the limb slightly outward from the torso.
3. I rotated around the x -axis to swing the upper limb forward or backward (used for walk animations).
4. I drew the upper limb as a cylinder.
5. I translated forward by the length of the upper limb to reach the joint.
6. I drew a joint sphere.
7. If needed (e.g., during walking), I applied another rotation to the lower limb segment.
8. I drew the lower limb as another cylinder.
9. I translated again to the end of the limb and drew the hand (sphere) or foot (scaled cube).

Switching Between Basic and Upgraded

Press “u” in the main loop to toggle which draw function is called (`draw_Scarecrow()` vs `draw_Scarecrow_Upgrade()`), enabling a clean switch between flat and hierarchical modeling.

Task 4: Walk-in-Place Animation

Limb Movement and Animation Logic

1) How I transformed the upper and lower arms/legs (forward vs. backward)

For both the arms and the legs, I translated from the torso or pelvis to the limb’s base position and then rotated the upper segment around the x -axis based on the current animation angle. When the limb was swinging backward, I rotated the upper segment more to keep it relatively straight. When swinging forward, I rotated the lower segment more to simulate bending at the elbow or knee.

2) When and why the lower limbs moved more than the upper I made the lower limbs (forearms and shins) rotate more than the upper ones only when the limb was moving forward (when it had a negative angle). In those cases, I doubled the angle on the lower segment to give it more visual bend. When moving backward, I kept the whole limb straight by only rotating the upper part.

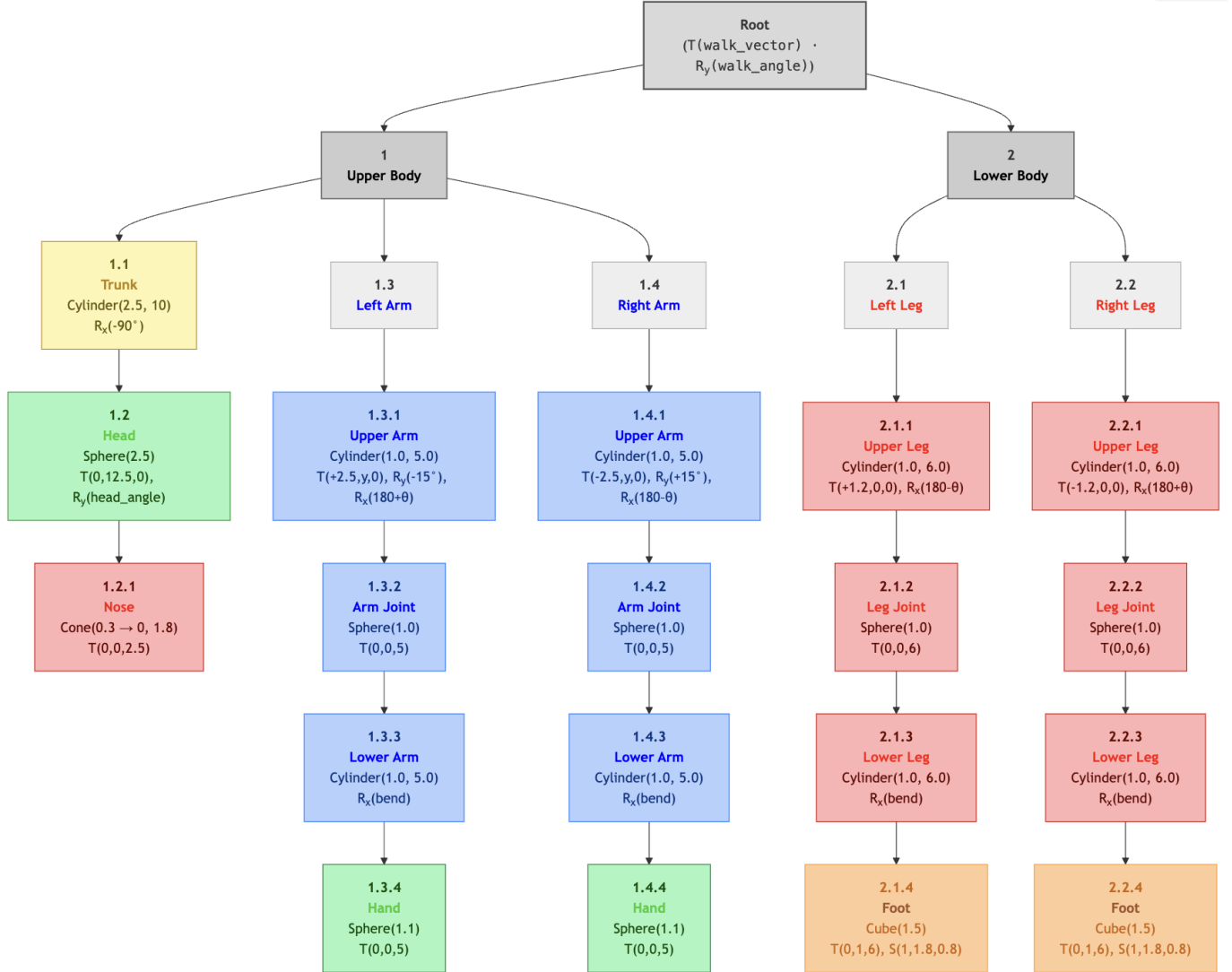


Figure 2: Scene Graph of Upgraded Scarecrow

3) Angle ranges used for arms and legs I clamped both `self.arm_angle` and `self.leg_angle` to the range $[-30^\circ, +30^\circ]$, which controlled the rotation of the upper segments. For the lower segments, I either left the angle at zero (when straight) or doubled it (when flexed forward).

4) When I kept the limbs straight I kept the limbs straight when the limb was swinging backward (positive angle). In those cases, I skipped any additional rotation of the lower segment so that the whole limb would appear stiff and extended.

5) How I transformed the joints After drawing each upper limb segment, I translated forward along the local z -axis and drew a small joint sphere. I didn't apply any rotation to the joint itself; it just marked the connection point to the next segment.

6) How I transformed the hands I made each hand a green sphere (radius ~ 1.1) at the end of the lower arm. I reached the hand position by translating forward again after drawing the lower arm. The hands rotated along with whatever rotations occurred at the lower arm.

7) How I created and transformed the feet I used `glutSolidCube(self.foot_cube)` (line 344) (with `self.foot_cube`) = 1.5 to make the feet and then applied `glScalef(1, 1.8, 0.8)` (line 343) to make them wider and flatter. After that, I translated forward and slightly upward to place the foot and then applied the scale before drawing.

8) How I adjusted the swinging speed I stored the swing speed as `self.swing_speed` (default 1.0). Each frame, I added or subtracted this value to the arm and leg angles depending on their current swing direction. When either angle reached $\pm 30^\circ$, I flipped the direction variable to reverse the swing and keep it oscillating.

Task 5: Walk in a Straight Line

1) How I transformed the entire scarecrow along a direction and speed To move the scarecrow across the scene, I translated the entire body by a running total vector called `self.walk_vector`. I updated this vector each frame by adding `self.walk_speed` times the current `walk_direction` (a unit vector). In the drawing function, I applied `glTranslatef(*self.walk_vector)` (line 292) before any other body part transformations which moved the entire figure together.

2) How I implemented this on top of Task 4 I built on Task 4 (walk-in-place) by adding actual translation in Task 5. In Task 4, I only animated the arms and legs by oscillating their angles. For Task 5, I added real movement by calling `update_walk_vector()` (line 192) each frame. This updated the global position of the scarecrow. I also used the rotation from Task 6 (`self.walk_angle`) so that the scarecrow could turn, and I used it to rotate the direction vector before translating.

3) How I synchronized walking speed with arm swinging speed To keep the walking animation realistic, I made sure that each step forward corresponded to a full swing of the limbs. I matched the limb swing frequency to the rate at which `self.walk_vector` changed. I tuned `self.swing_speed` so that the arms and legs reached their maximum angle of ($\pm 30^\circ$) each step cycle. This made it appear as though the scarecrow's motion was coordinated with its stride (one forward swing per translation step).

Task 6: Freeform Walk with Keyboard Control

The goal of this task was to make the scarecrow turn left/right while walking.

1) How I rotated walk_direction using walk_angle To let the scarecrow turn as it walked, I rotated the initial base direction vector $[0, 0, 1]$ around the y -axis by `walk_angle` degrees. I used a 3×3 rotation matrix for y -axis rotation:

$$R_y(\theta) = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix}$$

I made this matrix with `numpy`, where θ was the angle in radians converted from `walk_angle`. I applied this rotation to the base vector using matrix multiplication to get the new `walk_direction`, which always stayed normalized.

2) How I calculated and used walk_vector Each frame, I updated `self.walk_vector` by adding the current direction vector scaled by the walk speed (line 196):

```
walk_vector += walk_speed * walk_direction
```

This gave me a running position in world space. Then, inside the drawing function, I called `glTranslatef(*self.walk_vector)` before rendering the scarecrow. This effectively shifted the entire figure along the walk path, allowing it to move in real-time based on rotation and speed.

3) How I implemented freeform walking in OpenGL I modified the main input loop to listen for arrow key presses. I adjusted `walk_angle` by a few degrees for whenever a left or right arrow was pressed. I then recalculated the rotated direction using my rotation matrix and updated the `walk_direction` accordingly.

At the same time, when freeform walking was enabled (via the “r” key), I continuously called `update_walk_vector()` each frame to apply forward motion. In the drawing code, I first rotated the scarecrow by `walk_angle` around the y -axis and then translated it using `walk_vector`. This resulted in the figure being both turned and moved in the intended direction. Overall, this approach allowed my scarecrow to have smooth directional changes and synchronized motion based on user control.

Task 7: Switch between front, side, and back views

Side View For the side view, I set the camera’s eye position to $[50.0, 0.0, 0.0]$, which placed it 50 units to the right of the origin, looking directly toward the center of the scene. I kept the look-at point at $[0.0, 0.0, 0.0]$, so the camera faced the scarecrow from the right. The view-up vector was $[0.0, 1.0, 0.0]$ to maintain vertical orientation. This setup resulted in a clean, orthogonal side perspective with no tilt.

Back View For the back view, I moved the eye to a slightly elevated and diagonally offset position: $[90.0, 20.0, -120.0]$. This placed the camera behind and above the scarecrow, angled toward the scene. I used a look-at point of $[0.0, 10.0, 0.0]$ so that the camera would focus around the torso level. I kept the view-up vector as $[0.0, 1.0, 0.0]$. As a result, the view

was similar to a third-person camera in a game and helped the visuals for viewing walking movement from behind.

Task 8: Dynamic Viewing with Keyboard Control

1) How I rotated the camera’s gaze vector (mathematically) To implement camera tilting, I started with a base gaze vector (e.g., $[0, 0, -1]$ for front view), and rotated it using two angles:

- `tilt_angle_horizontal` rotated the gaze left/right around the y -axis (line 133)
- `tilt_angle_vertical` rotated the resulting vector up/down around the x -axis (line 136).

I used two standard 3×3 rotation matrices. First, for horizontal rotation:

$$R_y(\theta) = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix}$$

Then, for vertical tilt:

$$R_x(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\phi) & -\sin(\phi) \\ 0 & \sin(\phi) & \cos(\phi) \end{bmatrix}$$

I applied these in sequence to the base gaze vector, using `numpy.dot()` (or the `@` operator) in Python. I normalized the result to get a unit gaze direction, then computed the look-at point as (line 154):

$$\text{look_at} = \text{eye_pos} + (\text{gaze_unit_vector} \times 50)$$

This created a dynamic view that turned smoothly with key input.

2) How I moved the camera forward and backward along the gaze vector To support zooming, I adjusted the camera’s eye position by moving it along the current gaze direction:

$$\text{new_eye_pos} = \text{eye_pos} + (\text{gaze_unit_vector} \times \text{zoom_distance})$$

`zoom_distance` was a scalar value, and I updated it based on key input. If it was positive, the camera moved forward. If it was negative, it moved backward. I updated the eye position before computing the look-at point so that the entire camera moved in and out along its current direction of view.

3) My OpenGL implementation for tilting and zooming with keyboard input I implemented camera tilt and zoom inside the `update_view()` method of the `Camera` class. I updated the horizontal and vertical tilt angles using the keys:

- “a” and “d” rotated the gaze left and right (horizontal).
- “w” and “s” tilted the camera up and down (vertical).

I changed `tilt_angle_horizontal` and `tilt_angle_vertical` incrementally based on which key was pressed. For zooming, I used “q” to move forward and “e” to move backward, which adjusted `zoom_distance`. I recalculated these values every frame, and I passed the resulting eye and look-at points to `gluLookAt()` to update the view matrix.

Extra Credit: First-Person View

To simulate a first-person camera mounted on the scarecrow’s head, I calculated both the head position and facing direction based on two angles: 1) the scarecrow’s body orientation (`walk_angle`) and 2) the head turn angle (`head_angle`).

Head Position Calculation I calculated the head position as the current world position of the scarecrow plus an offset for head height (line 106):

$$\text{head_center} = \text{walk_vector} + [0, 12.5, 0]$$

This placed the camera just above the torso which aligned with where the head was drawn. I used this value as the camera’s `eye` position.

Facing Direction Calculation To calculate the gaze direction, I started with a base vector pointing forward in local space: `[0, 0, 1]`. I then applied two rotations to it:

1. I first rotated it by `walk_angle` around the *y*-axis to account for the body’s orientation.
2. I then applied a second rotation by `head_angle` around the *y*-axis to account for the head turning independently.

Mathematically, this looked like:

$$\text{gaze_dir} = R_y(\text{head_angle}) \cdot R_y(\text{walk_angle}) \cdot [0, 0, 1]$$

I normalized the resulting vector and used it to calculate the look-at point:

$$\text{look_at} = \text{head_center} + (\text{gaze_dir} \times 10)$$

This resulted in a camera that looked straight ahead from the scarecrow’s head in whatever direction it was facing and turning.

OpenGL Implementation Inside `Camera.update_view()`, I checked if the current view mode was “first_person”. If so, I:

- Retrieved `walk_vector` and added a fixed height offset to get the head position.
- Used two rotation matrices (implemented via `build_y_rotation_matrix()`) (lines 36-43) to apply both the body and head angles to a forward-pointing vector.
- Normalized the result and computed a look-at point 10 units ahead.
- Returned the resulting `eye` and `look_at` positions to update the camera via `gluLookAt()`.

This made the camera follow the scarecrow’s motion and orientation while dynamically adjusting as the head turned. It behaved like a fully head-mounted perspective, as it was able to move and turn realistically during walk animations.

Sources

In-class notes and code used to assist in entire project, ChatGPT to help with Latex formatting and debugging (specifically task 4 with getting the joints to move the right way), Mason Pennell as in-class collaborator (we did our own code and then compared logic)