Project 2: Transformation and Viewing

Natalie Huante 05/17/2025

Introduction

This project focuses on key transformations and viewing operations in computer graphics and is inspired by CPSC 516 - Advanced Computer Graphics course (taught my Dr. Trudi Qi) at Chapman University. The project uses OpenGL functions and Python libraries in the program. This paper is meant to outline the project's key components along with explanations of their implementations. The final project is split into 9 different tasks, each of which is outlined in detail below and is showcased in the video file attached. The main goal is to create two different versions of a scarecrow and animate its movements along with integrating other keyboard controls.

Task 1: Basic Scarecrow

- **Method** - *How each body part was placed into the appropriate positions to construct the Scarecrow.* In the following order, I apply these transformations from top down in my code:

  1. *Translate* up the Y-axis by 12.5 units. (applied to the head and the nose)
  2. *Translate* along the z-axis by 2.5 units. (applied to the nose to sit at the front of the head)
  3. *Rotate* about the x-axis by -90 degrees (applied to only the torso to stand it up parallel to the y-axis)
  4. *Translate* by -1.2 along the x-axis and -12 along the y-axis (applied to only the right leg to move it below the torso and off-center)
  5. *Rotate* about the x-axis by -90 degrees (applied to only the right leg to stand it up parallel to the y-axis before shifting it in step 4)
  6. *Translate* by 1.2 along the x-axis and -12 along the y-axis (applied to only the left leg to move it below the torso and off-center)
  7. *Rotate* about the x-axis by -90 degrees (applied to only the left leg to stand it up parallel to the y-axis before shifting it in step 6)
  8. *Translate* by -12.5 along the x-axis and 9 along the y-axis (applied to only the right arm to make it stick out on the side of the torso)
  9. *Rotate* about the y-axis by 90 degrees (applied to only the right arm to make it parallel to the x-axis before shifting it in step 8)
  10. *Translate* by 12.5 along the x-axis and 9 along the y-axis (applied to only the left arm to make it stick out on the other side of the torso)
  11. *Rotate* about the y-axis by -90 degrees (applied to only the left arm to make it parallel to the x-axis before shifting it in step 10)

The code order is represented below and it results in only certain transformations being applied to certain body parts.

Push Matrix # head and nose
- Step 1 - Translate
- Head Sphere Created
- Step 2 - Translate
- Nose Cylinder Created

Pop Matrix
Push Matrix # torso
- Step 3 - Rotate
- Torso Cylinder Created

Pop Matrix
Push Matrix # right leg
- Step 4 - Translate
- Step 5 - Rotate
- Right Leg Cylinder Created

Pop Matrix
Push Matrix … same but for left leg… Pop Matrix # left leg
Push Matrix
- Step 8 - Translate
- Step 9 - Rotate
- Left Leg Cylinder Created

Pop Matrix
Push Matrix … same but for left arm Pop Matrix # left arm

Task 2: Rotate Scarecrow's Head & Nose

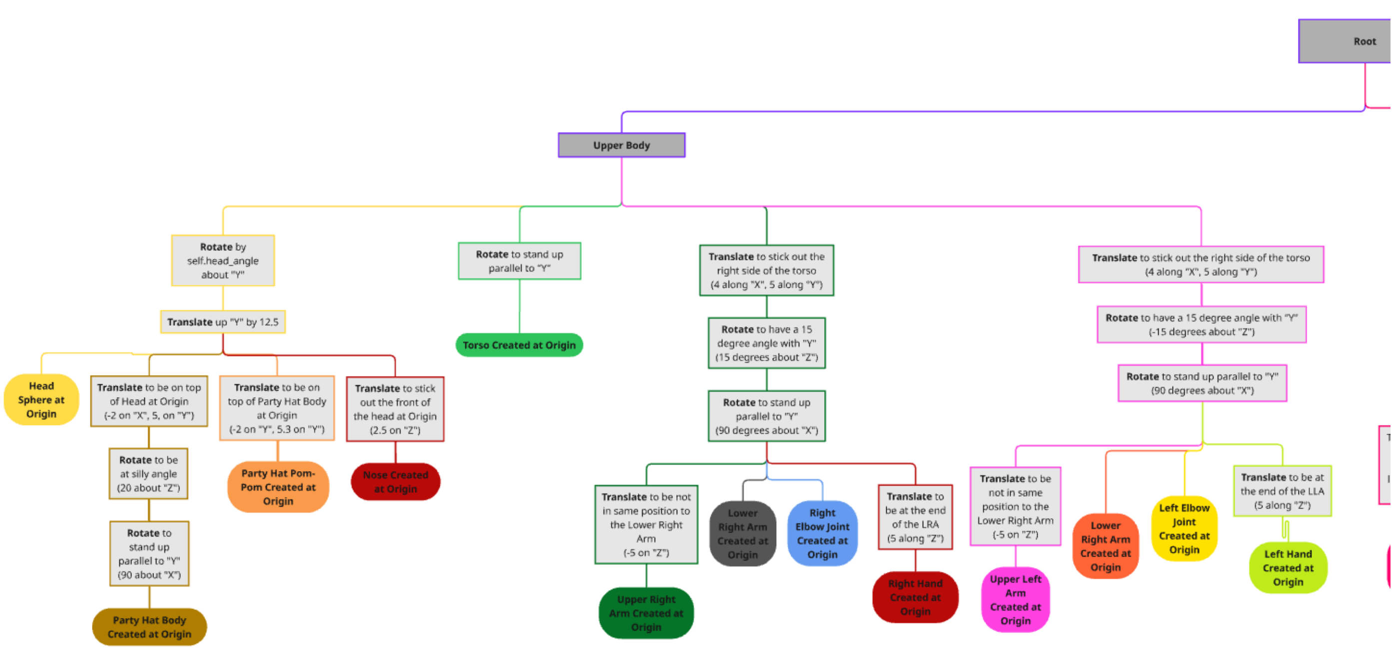
- **Method** - *How to only rotate the head and nose, including which OpenGL functions are used, incorporating them into the previous code for Task 1, and incorporating keyboard input.*

  To rotate only the head and nose, I add `glRotatef(self.head_angle, 0, 1, 0)` above the translation corresponding to Step 1 in Task 1. Since the head and the nose shapes are both in the same stack already, adding this rotation to the top of the stack will effectively apply the rotation to both shapes. The angle we rotate by is stored in `self.head_angle`, which is modified in the main file every time the 'I' and 'O' keys are held down on the keyboard. By using a variable, we can keep modifying and changing the rotation value with keyboard input and simply pass in that value to the rotation operation at each call of the `draw_Scarecrow()` function.

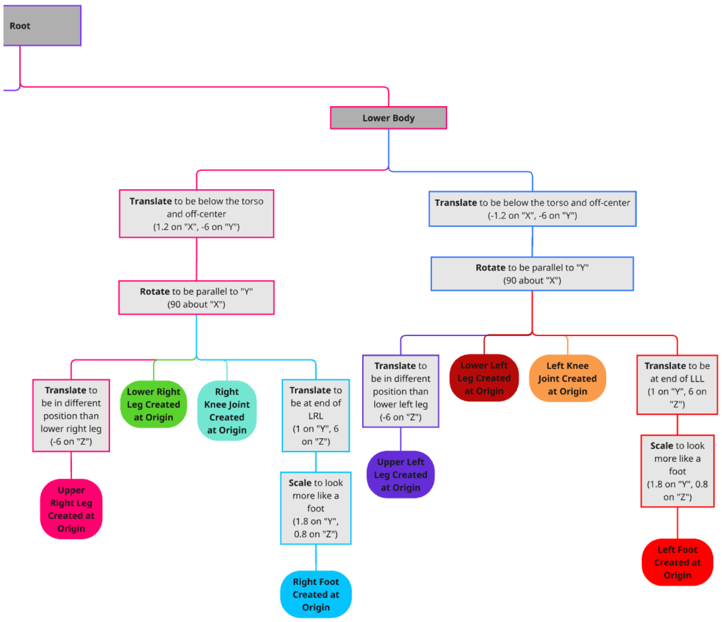
Task 3: Upgraded Scarecrow

- **Method** - *Shows the relationship between transformations with a scene graph. Describe how each body part is transformed in the upgraded Scarecrow using OpenGL.*

Scene Graph for Upper Body Transformations of Upgraded Scarecrow



Scene Graph for Lower Body Transformations of Upgraded Scarecrow



Looking at the scene graph, I will first step through the process of reading the graph for an example body part. Second, I provide a short description of some other body parts' transformations, according to the group(s) they are a part of.

Reading the Graph for the Right Foot Object - Right Foot

We read the graph bottom-up, starting at the Blue oval node that represents the creation of the right foot object. Each grey transformation node will be applied to the object below as we move up the graph. Using this logic, the first transformation applied will be a scaling to make the created cube object have similar dimensions to a foot. Then, a translation is done to move that scaled cube to the end of where the lower right leg object will be (when it is first created). Next, that translated version is rotated and translated to be aligned with the upper body (along the y-axis) and below the torso object. A similar process can be done to understand any series of transformations on the scene graph.

Brief Description for Each Body Part by Group

OpenGL Functions Used

- For each ***Rotation*** transformation, the OpenGL function glRotatef() is used in the program, where the first argument is the specified angle, and the following three arguments are 0 or 1, where only the specified axis is set to 1.
- For each ***Translate*** transformation, the OpenGL function glTranslatef() is used in the program, where the specified movements are placed in the order (x, y, z) as arguments. If no movement is specified, we will use zero as its argument to indicate no movement in that direction.
- For each ***Scale*** transformation, the OpenGL function glRotatef() is used in the program, where the specified scaling movements are placed in the order (x, y, z) as arguments. If no scaling is indicated in a certain direction, we use zero as its argument.

Example Transformation Series

- **Left Foot:** Created at Origin (CAO) → Scale → Translate → (w rest of left leg) Rotate → (w rest of left leg) Translate
- **Left Knee & Lower Left Leg:** CAO → (w rest of left leg) Rotate → (w rest of left leg) Translate
- **Upper Left Leg:** CAO → Translate → (w rest of left leg) Rotate → Translate
- **Right Foot:** CAO → Scale → Translate → (w rest of right leg) Rotate → Translate
- **Right Knee & Lower Right Leg:** CAO → (w rest of right leg) Rotate → Translate
- **Upper Right Leg:** CAO → Translate → (w rest of right leg) Rotate → Translate
- **Head:** CAO → (w rest of head objects) Translate → Rotate

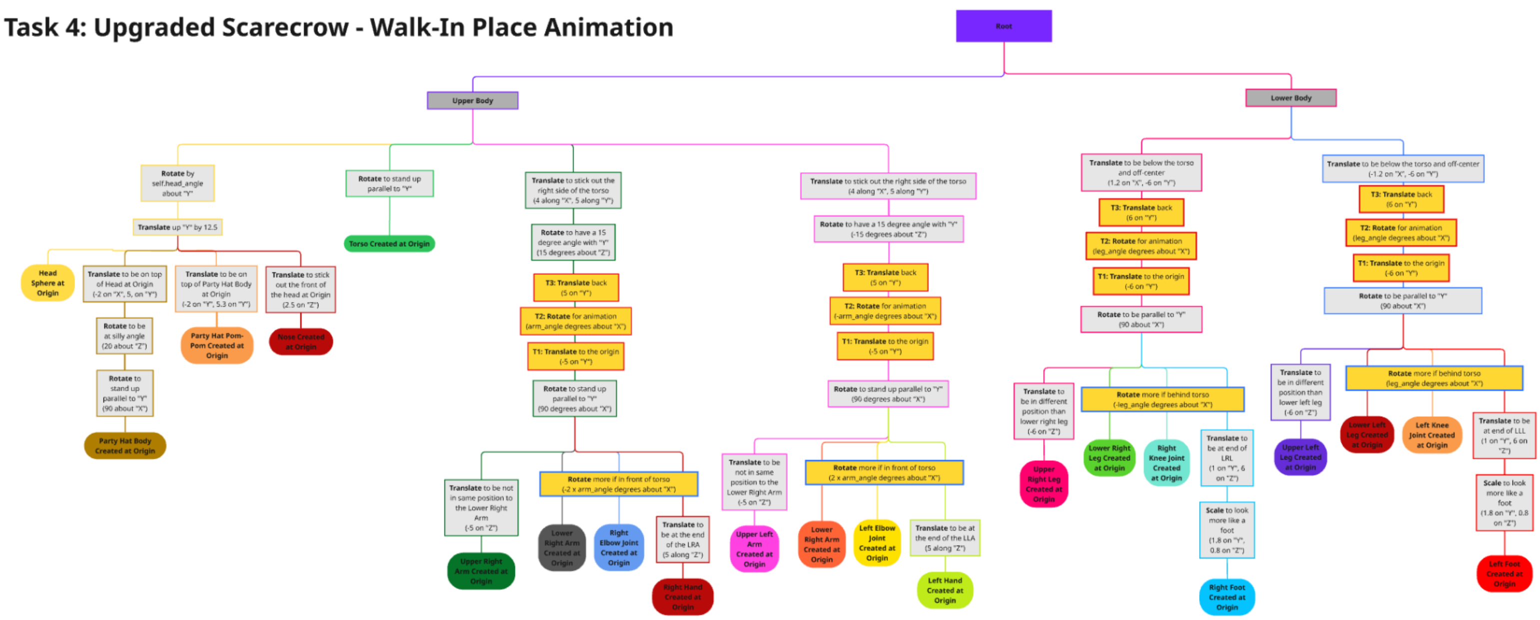
- **Party Hat Body:** CAO → Rotate → Rotate → Translate → (w rest of head objects) Translate → Rotate
- **Party Hat Pom-Pom:** CAO → Translate → (w rest of head objects) Translate → Rotate
- **Nose:** CAO → Translate → (w rest of head objects) Translate → Rotate

I do not include all of the objects in this list in an effort not to repeat the entire scene graph in bullet points.

Task 4: Walk-In Place Animation

- **Method** - *See subsections below.*

Modified Scene Graph to Include Walk-In Place Animation Transformations [shown with with gold fill]



1) *Briefly explain how you transform limbs during the forward and backward movements.*

For the overall arm and leg animations (not including the extra rotation needed for the lower limbs), three new transformations are added to each overall limb. These are placed so that they affect the entire limb, but they happen before the last translation that sets the limb in place in relation to the torso. Take the right arm as an example. The three transformations are placed above all the other transformations that affect the individual body parts within the right arm limb, so that they affect all of those individual parts (the hand, the elbow, …). Therefore, T1, T2, and T3 will transform the entire limb by the same amounts. These three new transformations are as follows [nodes with gold fill, red outline]:

- **T1: Translate(x, y, z):** This translation will move the limb to be centered at the origin. This will prepare it for T2.
- **T2: Rotate(self.leg_angle or .arm_angle, 1, 0, 0):** This rotation will make sure that the angle at which the entire limb is rotated is dependent on what the current value of the variable is. Rotation is around the x-axis for all four limbs.
- **T3: Translate(-x, -y, -z):** This translation will revert the T1 and place the entire limb (now rotated after T2) back where it was before in the world.

2) *How and when to make the lower limbs transform more than the upper limbs.*

To make only the lower limbs transform more, a single line of code is added right above where each lower limb was created at origin. Therefore, if I rotate the lower limb by x number of degrees before it gets rotated again by the transformations described in Q1 above, the final scarecrow will have a lower limb that is rotated x degrees more than the corresponding upper limb. This is applied to each lower limb (including the joints and feet/hands), and we make the following considerations to account for extra rotation only in certain scenarios [nodes with gold fill, blue outline]:

- **Lower Left Leg:** Only if the leg angle is positive (aka the left leg limb is currently swinging behind the torso) do we apply the `glRotate(self.leg_angle, 1, 0, 0)` transformation.
- **Lower Right Leg:** Only if the leg angle is negative (aka the left leg limb is currently swinging behind the torso) do we apply the `glRotate(-self.leg_angle, 1, 0, 0)` transformation.
- **Lower Right Arm:** Only if the arm angle is negative (aka the right arm limb is currently swinging in front of the torso) do we apply the `glRotate(2 * self.arm_angle, 1, 0, 0)` transformation.
- **Lower Left Arm:** Only if the arm angle is positive (aka the left arm limb is currently swinging in front of the torso) do we apply the `glRotate(-2 * self.arm_angle, 1, 0, 0)` transformation.

3) *What angle ranges are used for the arms and legs (upper and lower).*

The arm angle and leg angle for the animation are both restricted to the range of [-30, 30]. This is handled in the main file. If the `key_l_on` is True (the walk animation is toggled on), then:

- If the scarecrow's `arm_direction` is 1, the arm is currently swinging backwards, so the `arm_angle` is increased by the `swing_speed` until it reaches the maximum of 30 degrees. Once it reaches 30, the `arm_direction` is negated (to now swing in the opposite direction), and the same process continues, but now decreasing until the angle reaches -30.
- The same is done for the scarecrow's `leg_angle`.

4) *How and when to keep the arms and the legs straight.*

The arms should be kept straight when swinging behind the torso and bend as they come in front of the torso. The legs are the opposite, staying straight when in front of the torso and bending as they go behind the torso.

I accomplish this by using the `arm_angle` and `leg_angle` to indicate where the limb would be in the rotation. If the angle is positive, it would be rotated to be in front of the torso, and vice versa for a negative angle. Therefore, I only add an extra rotation by `leg_angle` to the lower leg limbs when their respective side is behind the torso. It is important to note that they alternate swinging directions, so for the left leg, a positive `leg_angle` means it is behind the torso, while for the right leg, a negative `leg_angle` means it is behind the torso.

The same is true for the arm limbs. A positive `arm_angle` means the right arm is in front of the torso, while a negative `arm_angle` means the left arm is in front of the torso. Also, the lower arm limbs should rotate from -60 to 60, so the `glRotatef()` function, when executed, simply multiplies the current `arm_angle` by 2.

5) *<u>How to transform a joint for both arms and legs.</u>*
6) *<u>How to transform hands.</u>*
7) *<u>How to create and transform feet, which initially are cubes.</u>*

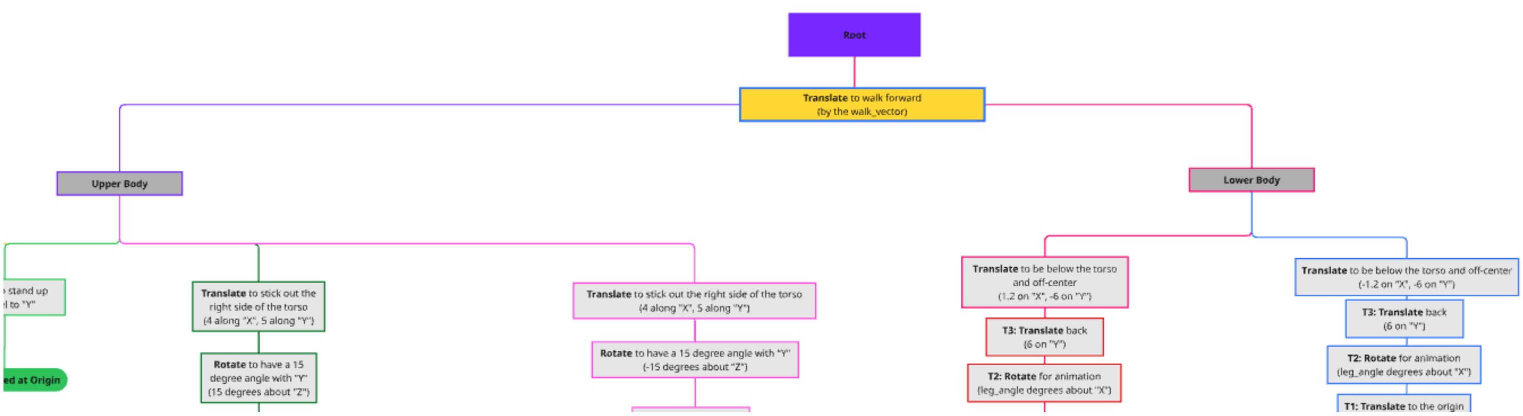
For Q5-Q7, we can look back at the explanation of the initial creation of the upgraded scarecrow and the initial scene graph to see how the feet, joints, and hands are created and transformed to be in the correct position, of the correct size, and of the correct rotation. No further transformations are needed for the animation since this new movement will be applied to each limb after they have already been constructed correctly. For example, the additions that animate the left arm as a whole will already affect the left hand and the left elbow joint due to their placement in the code.

8) *<u>How to adjust the swinging speed of the arms and legs.</u>*

Since we use the swinging speed to adjust how much to change the `leg_angle` and `arm_angle` at each iteration of the animation running, all we need to do to adjust the swinging speed of the arms and legs is to adjust the `swing_speed` variable of the scarecrow.

<u>Task 5: Walk in a Straight Line</u>

Close-Up of Node(s) Added in Sene Graph for Task 5 [shown with with gold fill]



- **Method** - *See subsections below.*

1) *<u>How to transform the entire Scarecrow along the given direction and speed.</u>*
2) *<u>How to implement on top of the previous code for Task 4.</u>*

To move the entire scarecrow in a straight line, the following additions are made:

a. Pressing `R` on the keyboard will toggle `key_r_on` in the main file, which will call the `update_walk_vector()` for the scarecrow at each iteration of the main loop.
b. In the `draw_Scarecrow_Upgrade()` function, I add a translation above all the other transformations and creations of objects for the scarecrow. Due to its location, this translation will move the entire scarecrow by its parameters after all objects have been placed, scaled, and rotated to fulfill our previous requirements from Task 4. In other words, all the body parts will be placed in their proper place (including the current position of the limbs at their current

iteration of the swinging animation) before the entire scarecrow is translated together by the same amount from the origin. The parameters of the glTranslatef() are simply the values of the walk_vector. The math behind how this is done is discussed below.

update_walk_vector () explained:

1. The current walk_vector is increased by the product of walk_speed and walk_direction. The direction stays constant at [0, 0, 1], which means the scarecrow will only walk further and further in the direction of the positive z-axis.

3) *<u>How do you synchronize the walking speed with the arm swinging speed to make the walking look realistic?</u>*

To make the walking look more realistic, I added some functions and event keys that allowed me to play with the values of the scarecrow's variables as the program ran. This helped avoid opening and closing the program repeatedly to test new combinations of swing_speed and walk_speed_mp. walk_speed_mp is a new and is used to calculate the appropriate walk_speed along with swing_speed. After testing, I found the ideal number to be 0.3. Therefore, I default the multiplier to be 0.3, but leave the event keys intact in the case that others might want to use my program to experiment with other combinations. These keys can also be used to increase the swing_speed while the program is running, which will show the instant change in both the swinging of the limbs and the walking speed.

<u>Task 6: Freeform Walk with Keyboard Control</u>

$$P_\theta := \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$Q_\theta := \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{pmatrix}$$

$$R_\theta := \begin{pmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{pmatrix}$$

- **Method** - *See subsections below.*

1) *How to rotate the walk vector mathematically, what the formula looks like, & what rotation matrix and parameters areused.*

To rotate the walk_direction vector on the rotation angle walk_angle, I do the following mathematically in the rotate_vector() function:

a. The rotation angle is first converted from degrees to radians.
b. Then, a 3x3 rotation matrix is created that follows the notation in the image on the right. We can rotate a 3x1 vector (the current walk direction) by multiplying it with this rotation matrix. To create this matrix, a 3x3 matrix is initialized with all zeroes using numpy. Then, according to the specified axis of rotation, the appropriate locations in the matrix are updated to be +/- sin or +/- cos of the rotation angle in radians. This is done using indexing.
c. Finally, the dot product is the final rotated vector.

2) *How to calculate and update the wa,lk vector and how to use it to update the position of the Scarecrow, mathematically.*

In Task 5, if the key_r_on variable was True, it would only continuously call update_walk_vector() (which at the time only extended the walk vector in a single direction). To integrate freeform walking, the walk_angle of the scarecrow is first either increased or decreased based on whether the left or right arrows are being held down. Then, the update_walk_vector() is called as it will use this now updated walk_angle value.

To incorporate this now updated walk_angle value, the update_walk_vector() function will first rotate the current walk_direction by the current walk_angle. Once this is done, the current walk_vector will be extended by the walk_speed and in the direction of the rotated walk_direction just calculated. The formula is shown below for simplicity.

walk_vector = walk_vector + (walk_speed x rotated_direction)

Since the new walk_vector is stored in the same variable as before, our previous translation transformation in Task 5 will handle this new freeform walking perfectly. The only piece left is to rotate the scarecrow to be walking in the direction it is moving. To do so, we must rotate the entire scarecrow around the y-axis after it has been constructed and before it is translated by the walk_vector. A glRotatef() transformation is placed just below the walking translation. In other words, the scarecrow is rotated about the y-axis while it is still at the origin.

3) *OpenGL implementation for freeform walking.*

Given the details in step 2, I will add only how to support continuous parameter updates in the main file. When detecting keyboard input, we can use two primary pygame methods: pygame.KEYUP and pygame.KEYDOWN. The former detects when a key is initially pressed down, and the latter detects when it is released (aka it moves back up). We can leverage these to track whether a key is being continuously held down. The format is as follows:

- A branch in our KEYDOWN conditional will set a boolean variable to True, indicating that the key has begun to be pressed.
- A corresponding branch in our KEYUP conditional will set that same boolean variable to False, indicating that the key has risen and is no longer being pressed.

We can then use the boolean variables for each continuous parameter to detect whether its corresponding action should occur at each iteration of the main loop. For example, if the key_right_on variable is True, the right arrow is currently being held down, so the scarecrow will angle more to the right as it walks.

Task 7: Switch Between Front, Side, and Back Views

- **Method** - *How the camera parameters for the side view and back view are set up.*

To implement the different camera views using the switch_view() function, I did the following:

1. Using a list of possible views [‘front’, ‘side’, ‘back’], I find the current view mode's index (position in the list). For instance, if view_mode is currently ‘side’, then the corresponding position is 1. I increment the position value by 1 to look at the position to the direct right, wrapping around to the front of the list when at the last element. This new position is then

used to find the corresponding next view mode in the list. In this case, 'back' would the the next view mode.

2. Using this next view mode, the camera's parameters are updated as follows, where each tuple represents the (x, y, z) positions of the parameter:
    a. Front View
        i. Camera Position (eye_pos): (0, 10, 50)
        ii. Point the Camera Looks Toward (look_at): (0, 10, -1)
    b. Side View
        i. Camera Position (eye_pos): (50, 10, 0)
        ii. Point the Camera Looks Toward (look_at): (-1, 10, 0)
    c. Back View
        i. Camera Position (eye_pos): (60, 30, -80)
        ii. Point the Camera Looks Toward (look_at): (37.5, 15, 10)
    d. The view-up vector (which indicates where the top of the camera should face) always remains at (0, 1, 0), meaning the top of the camera is always right-side-up as it faces the positive y-direction.

*** Note: These parameters are hard-coded since we want the camera to remain unchanged while in this view (that is until we get keyboard input in further tasks). ***

Task 8: Dynamic Viewing with Keyboard Control

- **Method** - *See subsections below.*

1) *How to rotate the camera's gaze vector by an angle, mathematically. What formulas to use, how to construct the matrix, & how to calculate the new look-at point.*

To implement the control of the camera, I did the following:

1. To allow keyboard control for the camera's gaze position, I use similar techniques as before to handle continuous parameter updates where holding the `W/S` keys increase/decrease the `tilt_angle_vertical` parameter and the `A/D` keys increase/decrease the `tilt_angle_horizontal` parameter.
2. Using these two parameters, the `update_view()` function needs to calculate the new gaze vector, which is need to calculate the new look at point. This is done by…
    a. Calculating the current gaze vector using the formula below:

       camera_gaze_vector = look_at_point - camera_eye_position

    b. Determining the rotation axis for the vertical tilt. The default is the x-axis and is changed for the z-axis when in side view. *** Note: In the future, this should be further improved to be based on the position and current gaze of the camera rather than the view mode the camera is in. ***
    c. Using the previously implemented `rotate_vector()` function, rotate the current gaze vector from step a by the `tilt_angle_vertical` around the determined rotation axis.

d. Rotate the resulting the vector from step c again, but now by the `tilt_angle_horizontal` by the y-axis. This will return the final rotated gaze vector.
e. Now that we have the new gaze vector, we use the formula below to calculate the new `look_at` point. Essentially, we start the point at the current camera position and move it given the gaze vector's direction and length.

new_look_at_point = camera_eye_position + new_gaze_vector

*** Note: Since we are only changing the gaze, the eye position is left as is for now. ***

3. The new `look_at` point and `eye_pos` values from `update_view()` are then used in the main file to call the `gluLookAt()` function.

2) How to update the camera's position to make it move forward and backward along the gaze vector based on a distance, mathematically.

To implement the zooming controls of the camera (aka moving it forward and backward along the gaze vector, I made the following additions:

1. To allow keyboard control for zooming in and out, I used the same technique as in Q1 above, except with the `Q/E` keys increasing/decreasing the `zoom_distance` parameter.
2. To integrate this zoom variable into the existing code, I add to the end of the `update_view()` function. After calculating the new `look_at` point, I do the following to calculate the new eye position of the camera since the gaze will remain the same but the position will change as the zoom increases or decreases.
    a. Normalize the new gaze vector so we can dismiss its original length and focus instead on it's direction.
    b. Use the normalized gaze to move the camera in the direction of the gaze. The `zoom_distance` variable determines how much to move in this direction (how much zoom to apply).

*** Note: Step b is also applied to the new look_at point from before to also move the look at point in the same direction by the same amount. Without this step, the camera will reach a point where zooming too far in will cause its gaze to flip 180 degrees (as the look at point is now behind it). ***

3) OpenGL implementation for both camera tilting and zooming with continuous input.

For more details, see Q3 under Task 6. The same approach is used to handle continuous motion, with the exception of which keys are being listened for and which parameters they will update.

Extra Credit: First-Person Dynamic View

- **Method** - *How to calculate the head position and facing direction mathematically and update the camera model during Scarecrow's dynamic walking (walking and turning around and looking around).*

To implement the fpv for the scarecrow, I did the following:

1. In the `switch_view()` function, update the view modes available. The list is now `['front', 'side', 'back', 'first-person']`. If the new view is fpv, the prior implementation will not work the

camera's parameters need to be calculated based on the scarecrow's current position in the world and not are typically changing. We will discuss this calculation later.

2. As of now, the update_view() function is called every iteration of the main function and calls gluLookAt() with the returned new look_at and eye_pos vectors. Since camera control is not accessible from the fpv view and the camera's parameters are only based on the scarecrow's current position, the previous update_view() code is placed into the else block of a conditional. By doing so, we can use a different approach to calculate these parameters when in fpv or continue to use this previous code in all other cases. To keep things simple, I extract this fpv approach into a helper function called update_fpv(). Therefore, the if block in this conditional only calls update_fpv() and assigns the returned values to eye_pos and look_at.
3. These are the only changes needed to integrate the fpv view into the existing code.

update_fpv() explained:

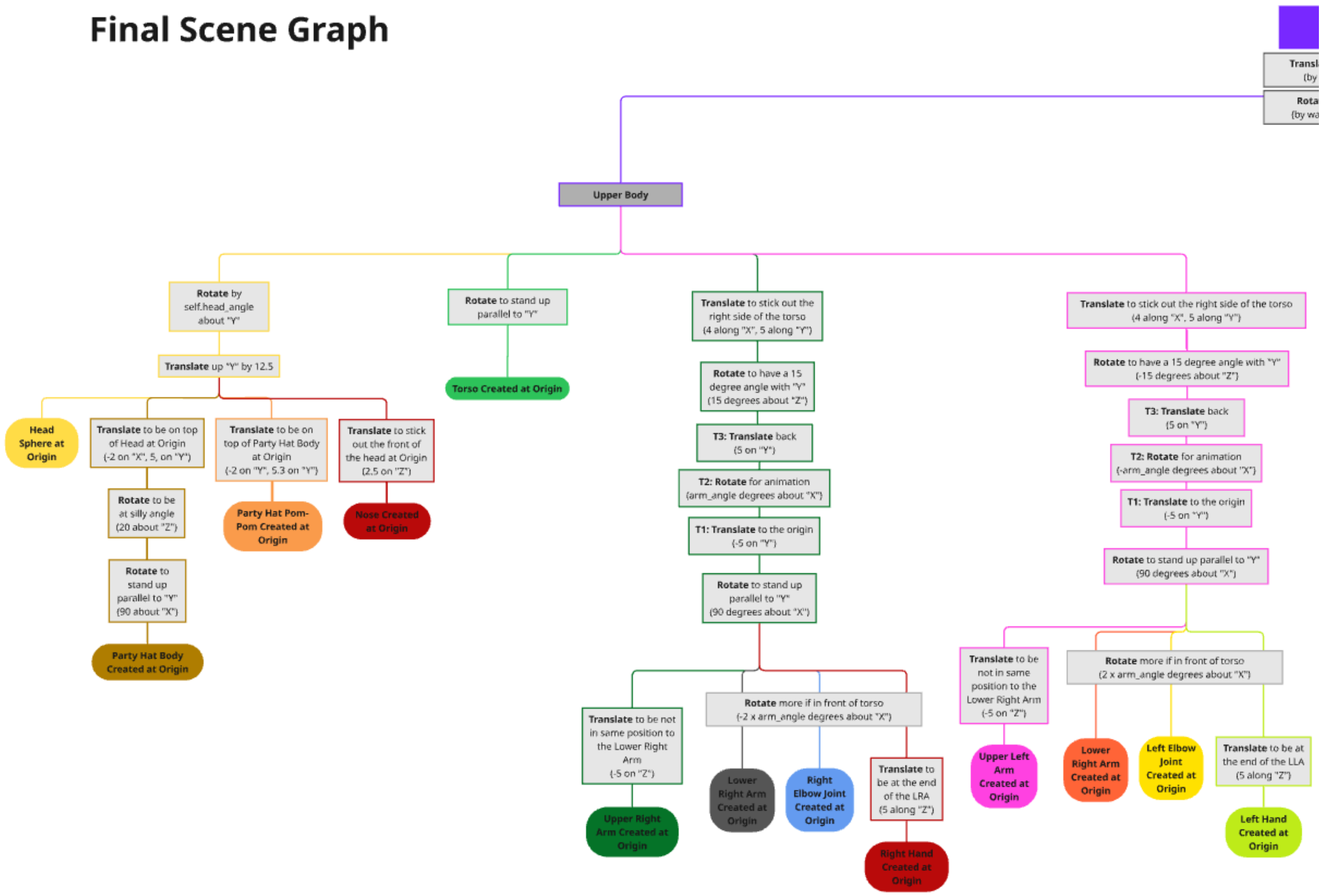
1. Finds the current position of the scarecrow head. This will guide the new camera eye position.
    a. Start at the point (0, 13, 3). When the scarecrow is still positioned at the origin, this will be right above the nose and a tad in front of the head (so as to not be stuck within the head object).
    b. Rotate this point vector around the y-axis by the current head_angle and then the walk_angle. This will make the point rotate around the head to be in line with where the scarecrow's current gaze will be facing.
    c. Translate this rotated point vector by the current walk_vector to adjust for where the scarecrow will be ultimately translated to due to walking around. This resulting point vector will be returned as the new eye_pos.
2. Finds the current gaze of the scarecrow's head. This will guide the new camera look at point.
    a. Start with a unit gaze vector looking in the positive z-axis direction. This is the scarecrow's initial gaze direction before keyboard control.
    b. Apply the same rotations to the gaze vector as those applied in step 1b above. This will result in the correct gaze vector after considering rotations of the head along and the entire body when walking around.
    c. The new look_at point is calculated by taking the new eye_pos from step 1 and adding twice the new gaze to it. This means the camera will be looking at the point 2 units away and in the direction of the calculated gaze. This point is returned as the new look_at point.

*** Note: If you look at the bottom of the main file, a draw_world_scenery() function is called after the draw_Scarecrow_Upgrade() function. This function will call upon a few helper functions in the Scarecrow class to populate the immediate area around the origin with some objects to make freeform walking in the fpv view more entertaining and to make the scarecrow's movements more obvious. ***

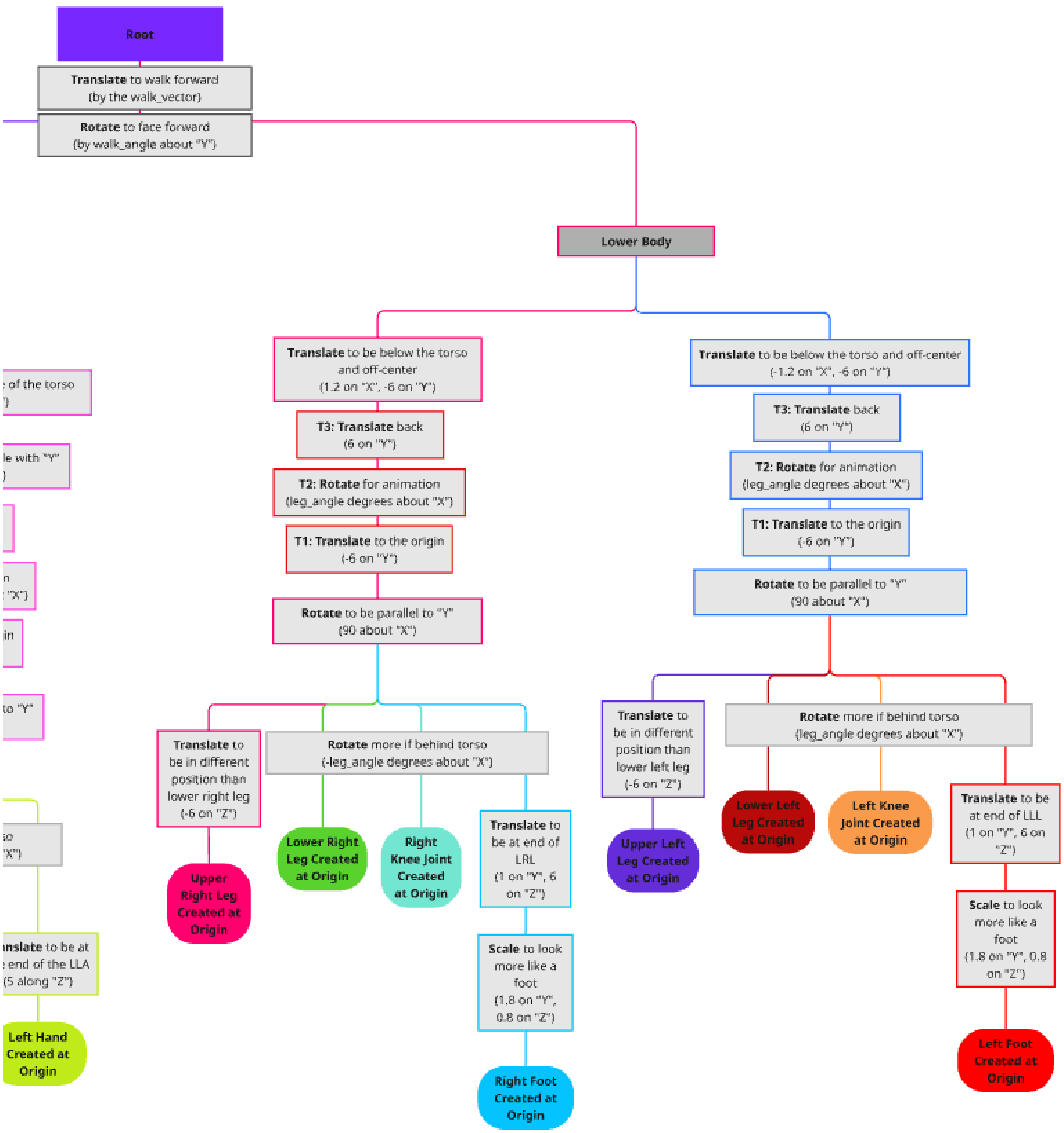
Final Video Submission

- Included in the submission folder as an MP4 file.
- All results methods are included in the same video file and are labeled accordingly.

<u>Upper Body Final Scene Graph</u>



Upper Body Final Scene Graph



Keyboard Controls Table

Key Input	Movement Function	Supports Continuous Input
i	Rotate Scarecrow's head to its left	Yes
o	Rotate Scarecrow's head to its right	Yes
u	Toggle the version of Scarecrow shown: basic, upgraded	No
l	Toggle walk-in place animation	No
r	Toggle freeform walking animation	No
←	When freeform walking, turn Scarecrow to its left	Yes
→	When freeform walking, turn Scarecrow to its right	Yes
a/d	Tilt the camera to look to its left and right (disabled for fpv)	Yes
w/d	Tilt the camera to look up and down (disabled for fpv)	Yes
q/e	Zoom the camera in/out by moving it along the gaze vector (disabled for fpv)	Yes
Space	Switch between the different camera views: front, side, back, fpv	No
0	Reset the view to its original parameters	No
1/2	When in freeform, decrease/increase the swing speed by 0.5 units	No
3/4	When in freeform, decrease/increase the walk speed multiplier by 0.1 units	No
5	When in freeform, reset the scarecrow to its original walk parameters and position at the origin.	No

References

- RGB Color Coder
- NumPy Random
- CPSC 515 Class Github
- NumPy Array Functions
- PyGame Documentation
- OpenAI's ChatGPT: I used ChatGPT for the last section of the extra credit section and research purposes throughout the project. Specifically, I used it to generate some random positions for my world's scenery (box stacks, trees, and orbs) so that they would be well disbursed. I opted for this instead of finding suitable x, z coordinates on my own, since I felt this was okay to pass off to the chat. I also used it for research purposes, or in other words, easily finding which method or function I was looking for within the libraries used. This mostly helped in terms of efficiency when completing this project.