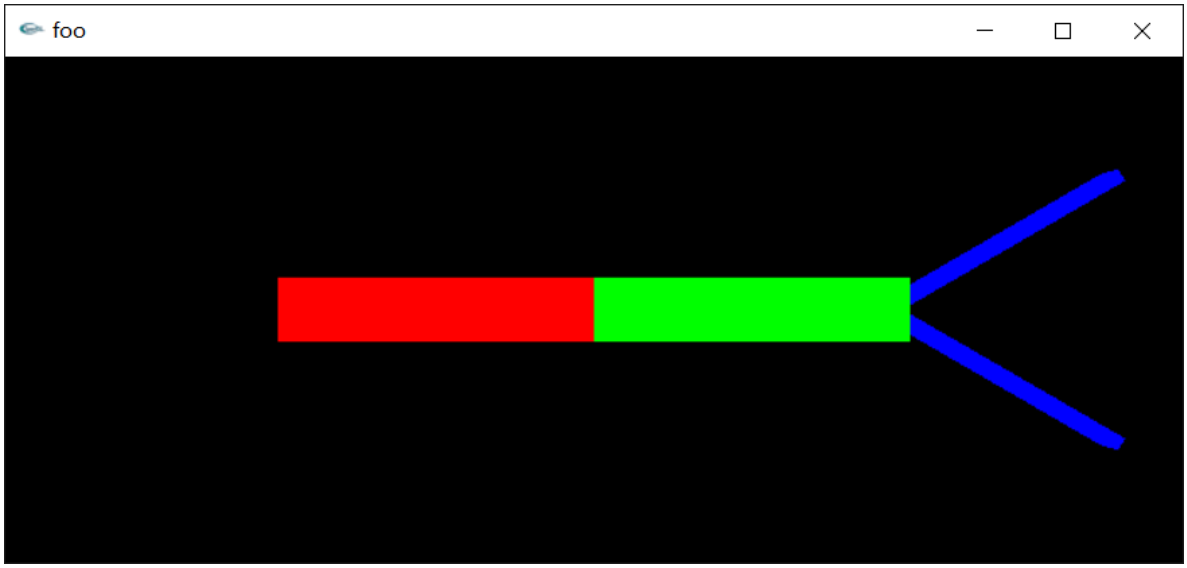
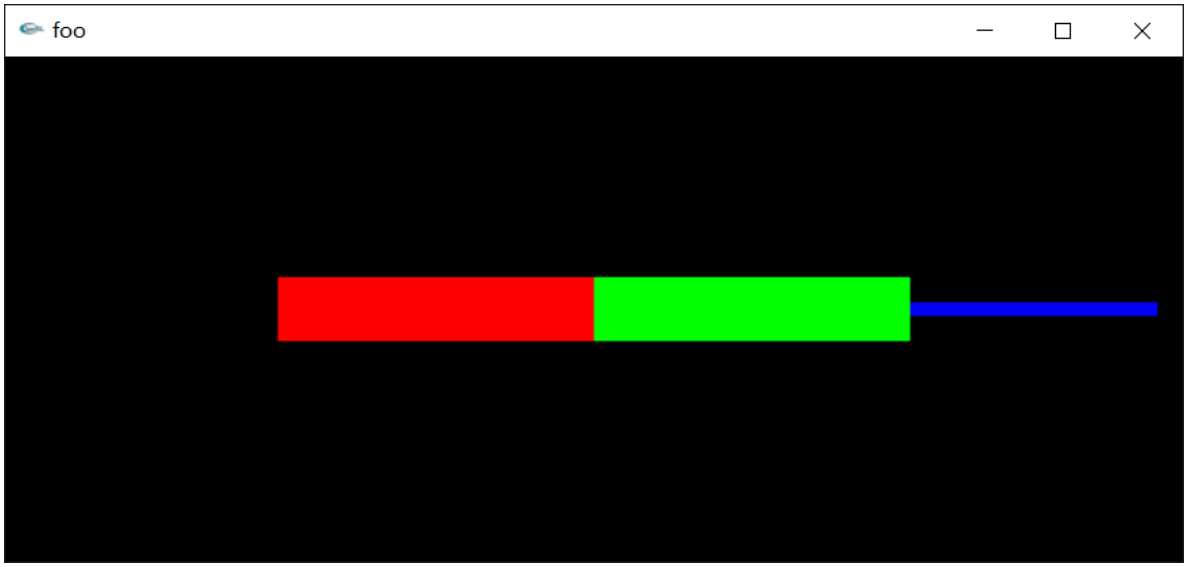
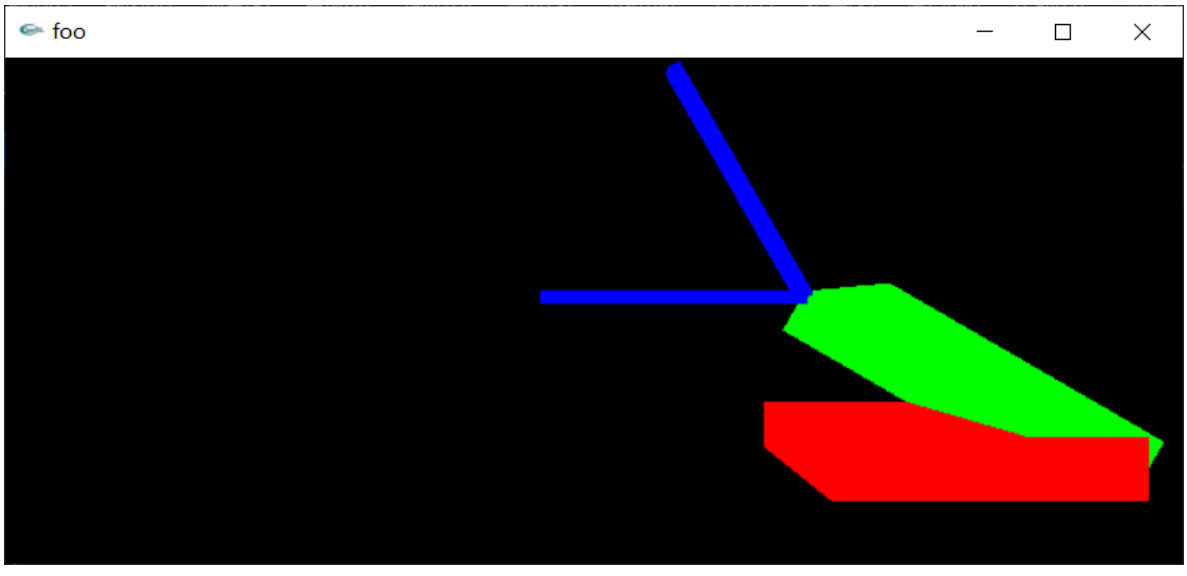
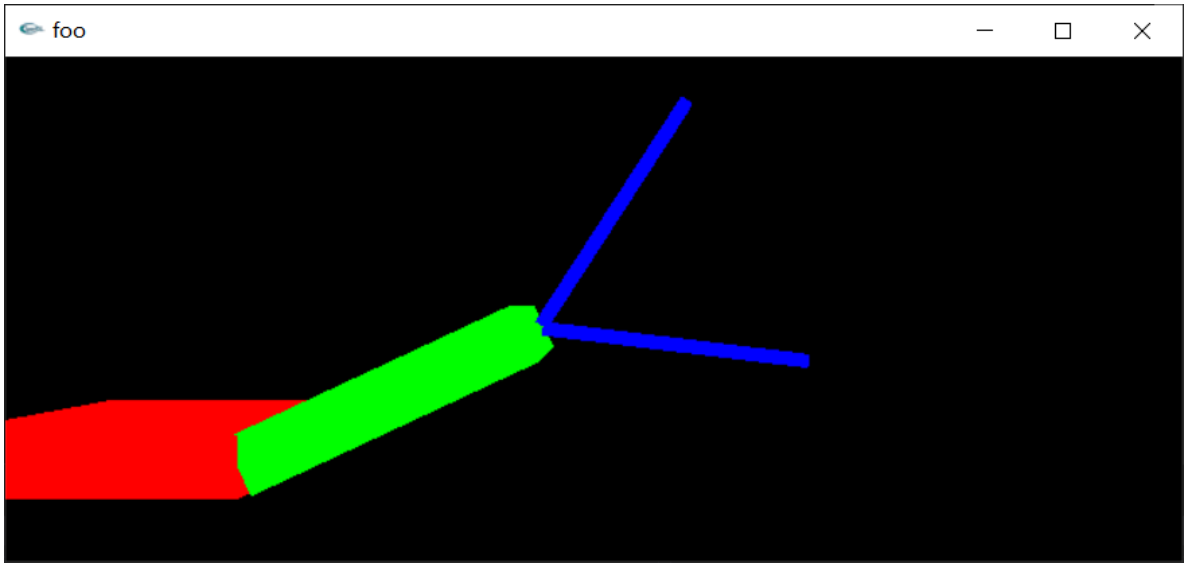
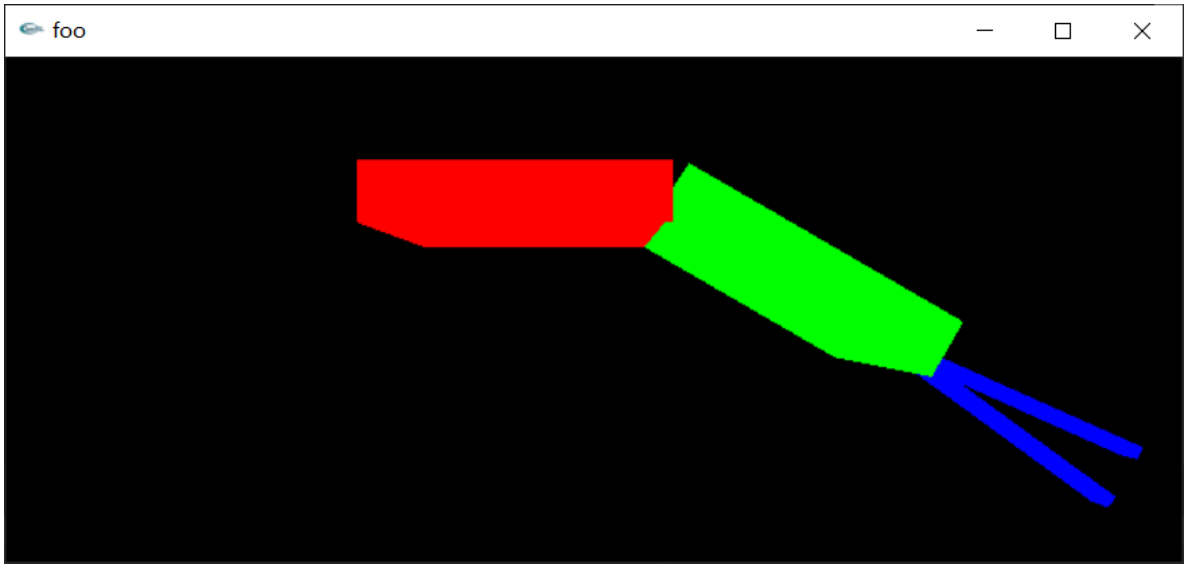
# Picture

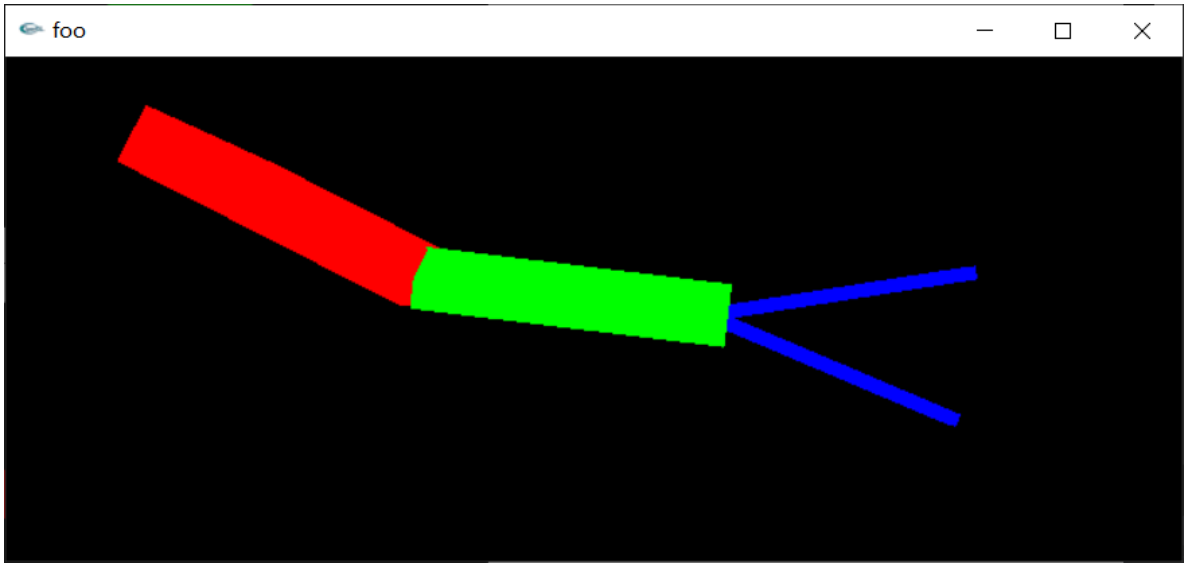



foo


foo


foo

# Code

### Experiment with the elbow and finger angles to determine reasonable maximum and minimum joint angles. Implement a hard coded maximum and minimum joint angles for these joints.

I set the maximum rotation angle of the robot arm to 150°.

```
case 'e':
        if (elbow < 150)
        {
            elbow = (elbow + 5) % 360;
        }
        glutPostRedisplay();
        break;
    case 'E':
        if (elbow > -150)
        {
            elbow = (elbow - 5) % 360;
        }
        glutPostRedisplay();
        break;
```

# glColor3fv function

```
void WINAPI glColor3fv(
   const GLfloat *v
);
```

## Parameters

- *v*

  A pointer to an array that contains red, green, and blue values.

## Remarks

The GL stores both a current single-valued color index and a current four-valued RGBA color. **glcolor** sets a new four-valued RGBA color. **glcolor3** variants specify new red, green, and blue values explicitly and set the current alpha value to 1.0 (full intensity) implicitly.

Current color values are stored in floating-point format, with unspecified mantissa and exponent sizes. Unsigned integer color components, when specified, are linearly mapped to floating-point values such that the largest representable value maps to 1.0 (full intensity), and 0 maps to 0.0 (zero intensity). Signed integer color components, when specified, are linearly mapped to floating-point values such that the most positive representable value maps to 1.0, and the most negative representable value maps to -1.0. (Note that this mapping does not convert 0 precisely to 0.0.) Floating-point values are mapped directly.

Neither floating-point nor signed integer values are clamped to the range [0,1] before the current color is updated. However, color components are clamped to this range before they are interpolated or written into a color buffer.

# glTranslatef function

```
void WINAPI glTranslatef(
   GLfloat x,
   GLfloat y,
   GLfloat z
);
```

## Parameters

- *x*

  The *x* coordinate of a translation vector.

- *y*

  The *y* coordinate of a translation vector.

- *z*

  The *z* coordinate of a translation vector.

## Remarks

The **glTranslatef** function produces the translation specified by (*x*, *y*, *z*). The translation vector is used to compute a 4x4 translation matrix:

$$\begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The current matrix is multiplied by this translation matrix, with the product replacing the current matrix. That is, if M is the current matrix and T is the translation matrix, then M is replaced with M T.

# glPushMatrix function

```
void WINAPI glPushMatrix(void);
```

## Remarks

There is a stack of matrices for each of the matrix modes. In GL_MODELVIEW mode, the stack depth is at least 32. In the other two modes, GL_PROJECTION and GL_TEXTURE, the depth is at least 2. The current matrix in any mode is the matrix on the top of the stack for that mode.

The **glPushMatrix** function pushes the current matrix stack down by one, duplicating the current matrix. That is, after a **glPushMatrix** call, the matrix on the top of the stack is identical to the one below it. The **glPopMatrix** function pops the current matrix stack, replacing the current matrix with the one below it on the stack. Initially, each of the stacks contains one matrix, an identity matrix.

# glRotatef function

```
void WINAPI glRotatef(
   GLfloat angle,
   GLfloat x,
   GLfloat y,
   GLfloat z
);
```

## Parameters

- *angle*

  The angle of rotation, in degrees.

- *x*

  The *x* coordinate of a vector.

- *y*

  The *y* coordinate of a vector.

- *z*

  The *z* coordinate of a vector.

## Remarks

The **glRotatef** function computes a matrix that performs a counterclockwise rotation of angle degrees about the vector from the origin through the point (x, y, z).

The current matrix is multiplied by this rotation matrix, with the product replacing the current matrix. That is, if M is the current matrix and R is the translation matrix, then M is replaced with M R.

If the matrix mode is either GL_MODELVIEW or GL_PROJECTION, all objects drawn after **glRotatef** is called are rotated. Use **glPushMatrix** and **glPopMatrix** to save and restore the unrotated coordinate system.

# glScalef function

```
void WINAPI glScalef(
   GLfloat x,
   GLfloat y,
   GLfloat z
);
```

## Parameters

- *x*

  Scale factors along the *x* axis.

- *y*

  Scale factors along the *y* axis.

- *z*

Scale factors along the *z* axis.

### Remarks

The **glScalef** function produces a general scaling along the *x*, *y*, and *z* axes. The three arguments indicate the desired scale factors along each of the three axes. The resulting matrix appears in the following image.

$$\begin{pmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The current matrix is multiplied by this scale matrix, with the product replacing the current matrix. That is, if M is the current matrix and S is the scale matrix, then M is replaced with M S.

## glPopMatrix function

```
void WINAPI glPopMatrix(void);
```

### Remarks

There is a stack of matrices for each of the matrix modes. In GL_MODELVIEW mode, the stack depth is at least 32. In the other two modes, GL_PROJECTION and GL_TEXTURE, the depth is at least 2. The current matrix in any mode is the matrix on the top of the stack for that mode.

The **glPushMatrix** function pushes the current matrix stack down by one, duplicating the current matrix. That is, after a **glPushMatrix** call, the matrix on the top of the stack is identical to the one below it. The **glPopMatrix** function pops the current matrix stack, replacing the current matrix with the one below it on the stack. Initially, each of the stacks contains one matrix, an identity matrix.

# Experience

## Basic 2D Transformations

Translation: – x′ = x + tx – y′ = y + ty

Scale: – x′ = x * sx – y′ = y * sy

Shear: – x′ = x + hx*y* – *y′* = *y* + *hy*x

Rotation: – x′ = x*cosΘ* - *y*sinΘ – y′ = x*sinΘ* + *y*cosΘ

## Homogeneous Coordinates

represent coordinates in 2 dimensions with a 3-vector

Convenient coordinate system to represent many useful transformations

## Matrix Composition

After correctly ordering the matrices

Multiply matrices together

What results is one matrix – store it (on stack)!

Multiply this matrix by the vector of each vertex

All vertices easily transformed with one matrix multiply