CMSC427 fall2017 lab3
Fan Yang

I moved the methods of computerCameraMatrix() and setCameraMatrix() from draw() to setup() and call them only when a change is being made to the camera, so that each new matrix is calculated only once. This strategy saves a lot of computation. More details on the implementation are listed below.

Compute camera matrix

The computation is the same as what we learned in class. Z axis of the camera (zc) is calculated by subtracting lookAt from At followed by normalization. X axis of the camera (xc) is the cross product of up and zc. yc is the cross product of zc and xc. The vector d is calculated by applying the transposed rotation matrix to the translation vector, as explained in the "camera matrix examples" file that Dr. Eastman uploaded on the class webpage.

Move forward and backward, slide left and right, slide up and down

When any of these moves happens, the camera frame remains the same except that the origin point is moving around, thus, my implementations for these methods are very similar. After a careful study of the camera matrix, I found that the only influence that each of the moves has on the camera matrix is that the move changes one of the coordinates of the d vector. For example, slide left or right only has influence on the x coordinate of the d vector, so my implementation for that function is as simple the code below:

```
public void slideLeftRight(float s) {
  d.x -= 10*s;
  setCameraMatrix();
}
```

For moving up and down and sliding forward and backward I just changed d.x into d.y or d.z, and the methods works perfectly.

Roll left and right, pitch up and down, yaw left and right

Again, my implementations for these methods are very similar. This is because when each of these moves happens, the origin of the camera's frame stays in the place and the only change is that the fame rotates around one of its axes. For rolling, the frame rotates around zc axis; for pitch, it rotates around xc; for yaw, it is yc. What we only need to do is to calculate the new axes from the old axes and the new d vector from the old d vector. This calculation can be easily done by using the vector transformation knowledge that we have been refreshed on recently. Below is my code for rolling camera. The code for yaw and pitch is very similar.

```
public void rollCamera(float t){
  PVector xc_new = PVector.mult(xc, cos(t)).add(PVector.mult(yc, sin(t)));
  yc = PVector.mult(xc, -sin(t)).add(PVector.mult(yc, cos(t)));
  xc = xc_new;
  float x = d.x * cos(t) + d.y * sin(t);
  d.y = -d.x * sin(t) + d.y * cos(t);
  d.x = x;
  setCameraMatrix();
}
```

Finally, home the camera back to its initial position just takes a recalculation of the camera matrix from the At and lookAt positions and the up vector.

Below is the source code for Lab3Camera class:

```
/* Lab 3
 * Lab3Camera class
 *
 * Test driver for Camera class
 * Illustrates Camera class methods
 */

Camera cam = new Camera();

void setup() {
  size(600,600,P3D);
  smooth();
  noFill();
  strokeWeight(2);
  // Compute the camera transformation matrix
  cam.computeCameraMatrix();
  // Computer and apply the camera transformation matrix
  cam.setCameraMatrix();
}

void draw() {
  background(255);
  //translate(width/2,height/2,0);
  // Modeling transformations
  pushMatrix();
  translate(120,0,0);
  box(80);
  popMatrix();
  box(160);
}

// Handle keyPressed events
// Each motion happens only once when key is pressed
void keyPressed() {

   if (key == f) // Move forward
      cam.forwardCamera( 1 );

   else if (key == "b")   // Move back
      cam.forwardCamera( -1 );

   else if (key == "l")   // turn left
      cam.slideLeftRight(-1);

   else if (key == "r")   // turn right
      cam.slideLeftRight(1);

   else if (key == "u")   // slide up
      cam.slideUpDown(-1);
```

```
    else if (key == "d")    // slide down
       cam.slideUpDown(1);

    else if (key == "t")    // roll right
      cam.rollCamera(0.05);

    else if (key == "y")    // roll left
      cam.rollCamera(-0.05);

    else if (key == "w")    // pitch down
      cam.pitchCamera(0.05);

    else if (key == "z")    // pitch up
      cam.pitchCamera(-0.05);

    else if (key == "e")    // yaw left
      cam.yawCamera(0.05);

    else if (key == "x")    // yaw right
      cam.yawCamera(-0.05);

    else if (key == "h")    // Home back to the initial camera position
      cam.homeCamera();
}
```

Below is the source code for Camera class:

```
/* Lab 3
 * Camera class
 * Maintain Camera transformation for OpenGL
 * Methods
 *
 * Note: one strategy is to keep at, LookAt and up as
 * as primary data, and update them when moving, and
 * then re-derive xc,yc,zc and d from them. This means
 * after each move you call computeCameraMatrix to refresh
 * xc,yc,zc and d
 *
 *    Fan Yang
 */

public class Camera {

  // Parameters to define camera transformation
  PVector at, lookAt, up;
  // Components of camera transformation
  PVector zc, xc, yc, d;
```

```
// Default camera parameters
public Camera(){
  at = new PVector(0,0,300);
  lookAt = new PVector(width/2,height/2,0);
  up = new PVector(0,1,0);
}

// Compute elements of camera matrix xc,yc,zc and d
public void computeCameraMatrix(){
  zc = PVector.sub(at, lookAt).normalize();
  xc = up.cross(zc).normalize();
  yc = zc.cross(xc);
  d = new PVector(-at.dot(xc), -at.dot(yc),-at.dot(zc));
}

// Apply camera matrix to the OpenGL transform matrix
// Uncomment this after computeCameraMatrix is complete
public void setCameraMatrix(){
  beginCamera();
    resetMatrix();
    applyMatrix(xc.x, xc.y, xc.z, d.x,
                yc.x, yc.y, yc.z, d.y,
                zc.x, zc.y, zc.z, d.z,
                0.0, 0.0, 0.0, 1 );
  endCamera();
}

// Move the camera forward s units
public void forwardCamera(float s) {
  d.z += 10*s;
  setCameraMatrix();
  //an alternative implementation
  //at.add(PVector.mult(zc, -10*s));
  //computeCameraMatrix();
  //setCameraMatrix();
}

public void slideLeftRight(float s) {
  d.x -= 10*s;
  setCameraMatrix();
  //an alternative implementation
  //at.add(PVector.mult(xc, 10*s));
  //lookAt.add(PVector.mult(xc, 10*s));
  //computeCameraMatrix();
  //setCameraMatrix();
}

public void slideUpDown(float s) {
  d.y -= 10*s;
  setCameraMatrix();
  //an alternative implementation
  //at.add(PVector.mult(yc, 10*s));
```

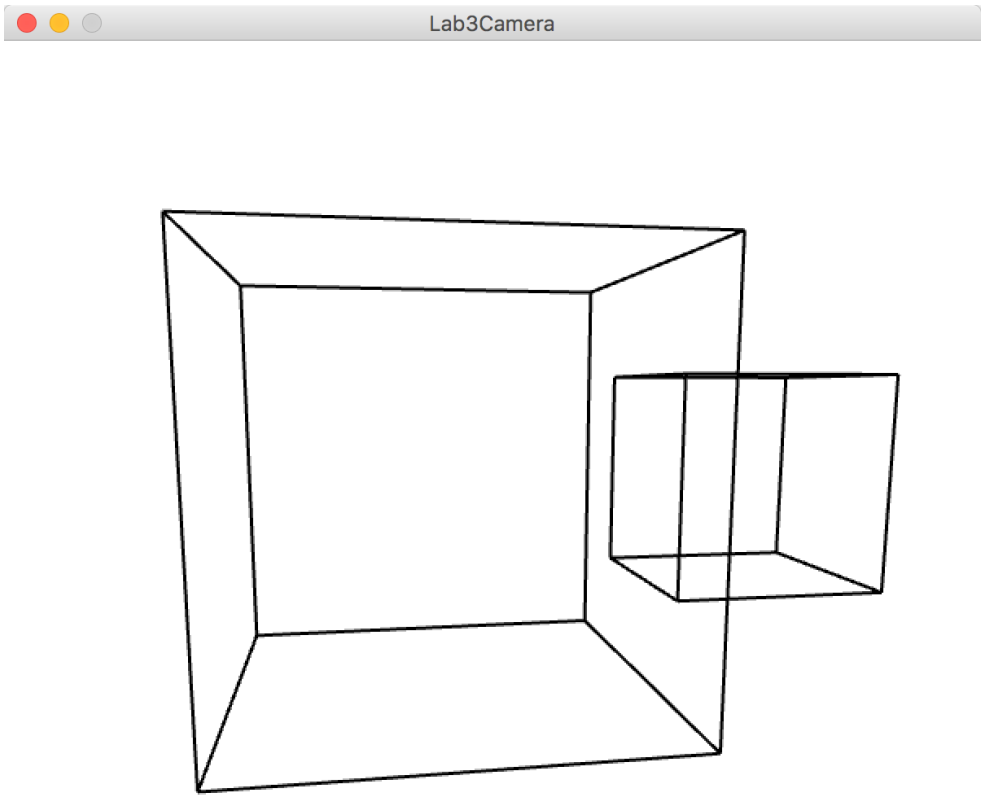
```
    //lookAt.add(PVector.mult(yc, 10*s));
    //computeCameraMatrix();
    //setCameraMatrix();
  }

  // Rotate the camera around camera axis (zc)
  // positive t radians
  public void rollCamera(float t){
    PVector xc_new = PVector.mult(xc, cos(t)).add(PVector.mult(yc, sin(t)));
    yc = PVector.mult(xc, -sin(t)).add(PVector.mult(yc, cos(t)));
    xc = xc_new;
    float x = d.x * cos(t) + d.y * sin(t);
    d.y = -d.x * sin(t) + d.y * cos(t);
    d.x = x;
    setCameraMatrix();
  }

  public void pitchCamera(float s){
    PVector yc_new = PVector.mult(yc, cos(s)).add(PVector.mult(zc, sin(s)));
    zc = PVector.mult(yc, -sin(s)).add(PVector.mult(zc, cos(s)));
    yc = yc_new;
    float y = d.y * cos(s) + d.z * sin(s);
    d.z = -d.y * sin(s) + d.z * cos(s);
    d.y = y;
    setCameraMatrix();
  }

  public void yawCamera(float s){
    PVector zc_new = PVector.mult(zc, cos(s)).add(PVector.mult(xc, sin(s)));
    xc = PVector.mult(zc, -sin(s)).add(PVector.mult(xc, cos(s)));
    zc = zc_new;
    float z = d.z * cos(s) + d.x * sin(s);
    d.x = -d.z * sin(s) + d.x * cos(s);
    d.z = z;
    setCameraMatrix();
  }

  public void homeCamera(){
    // Compute and apply the original camera matrix from at, lookAt and up
    cam.computeCameraMatrix();
    cam.setCameraMatrix();
  }
}
```


Lab3Camera