

Challenge #6 - Projecting, Camera Placing

Computer Graphics

Presented by:

Santiago Zubieta / Jose Cortes

Teacher:

Helmuth Trefftz

Universidad EAFIT

Briefing

We'll work on projections of 3D objects to 2D planes, and how to move the visualization Camera around, all of this using the good old point * matrix transformations!

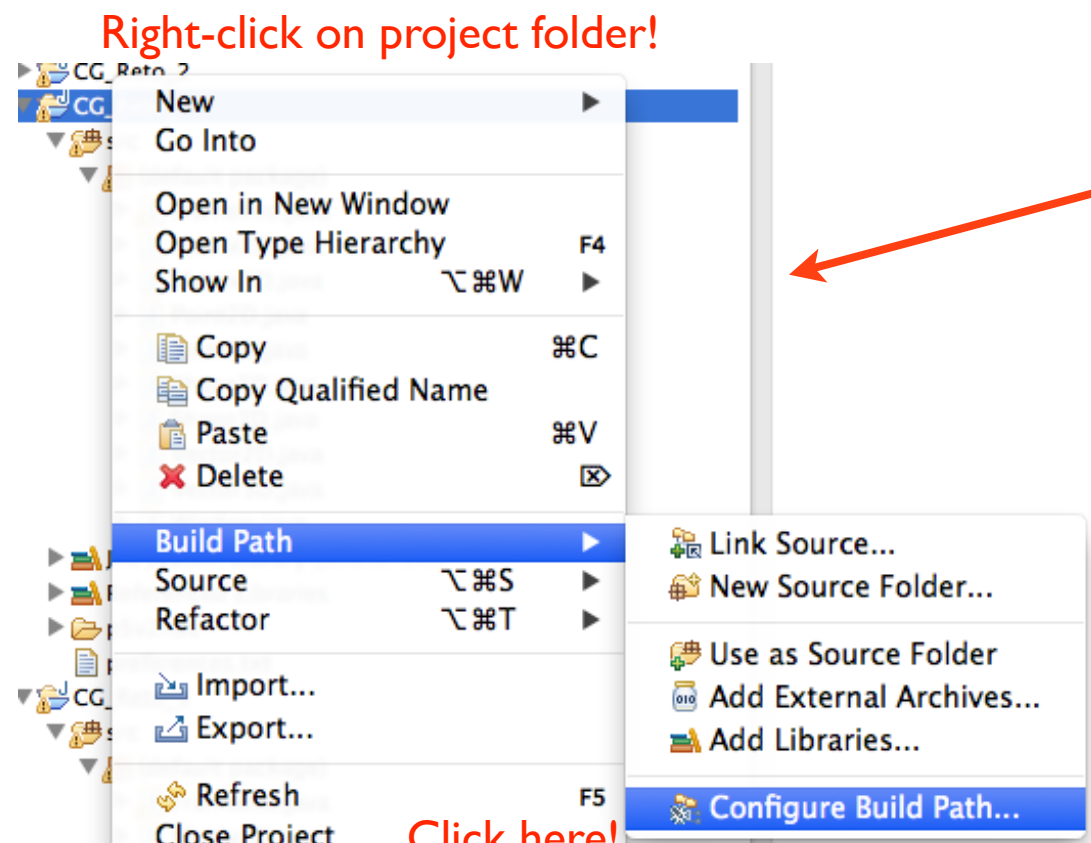
Transformation
Matrix

Point that will be
Transformed

```
public static Point3D multiplyMatrixAndPoint(Matrix3D mat, Point3D p) {  
    // It uses a 4D matrix to make us of Homogeneous Coordinates, to be  
    // ..able to translate with matrix operations  
    float pt[] = { 0, 0, 0, 0 };  
    float vals[] = { p.x, p.y, p.z, 1.0f };  
    for (int i = 0; i < 4; i++) {  
        for (int j = 0; j < 4; j++) {  
            pt[i] += mat.m[i][j] * vals[j];  
        }  
    }  
    return new Point3D(pt[0], pt[1], pt[2]);  
}
```

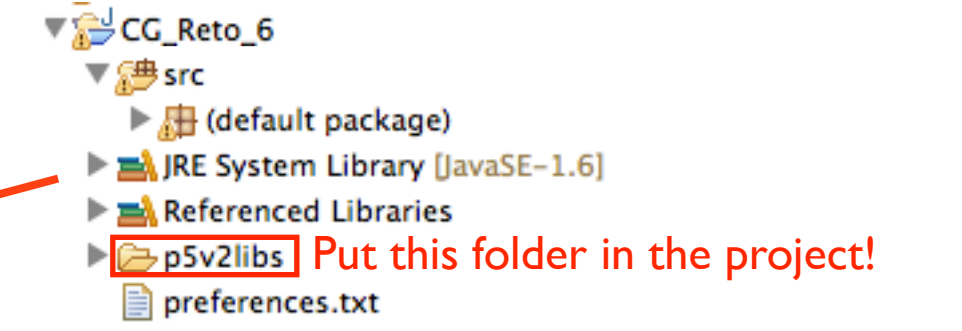
Installing Processing

Right-click on project folder!

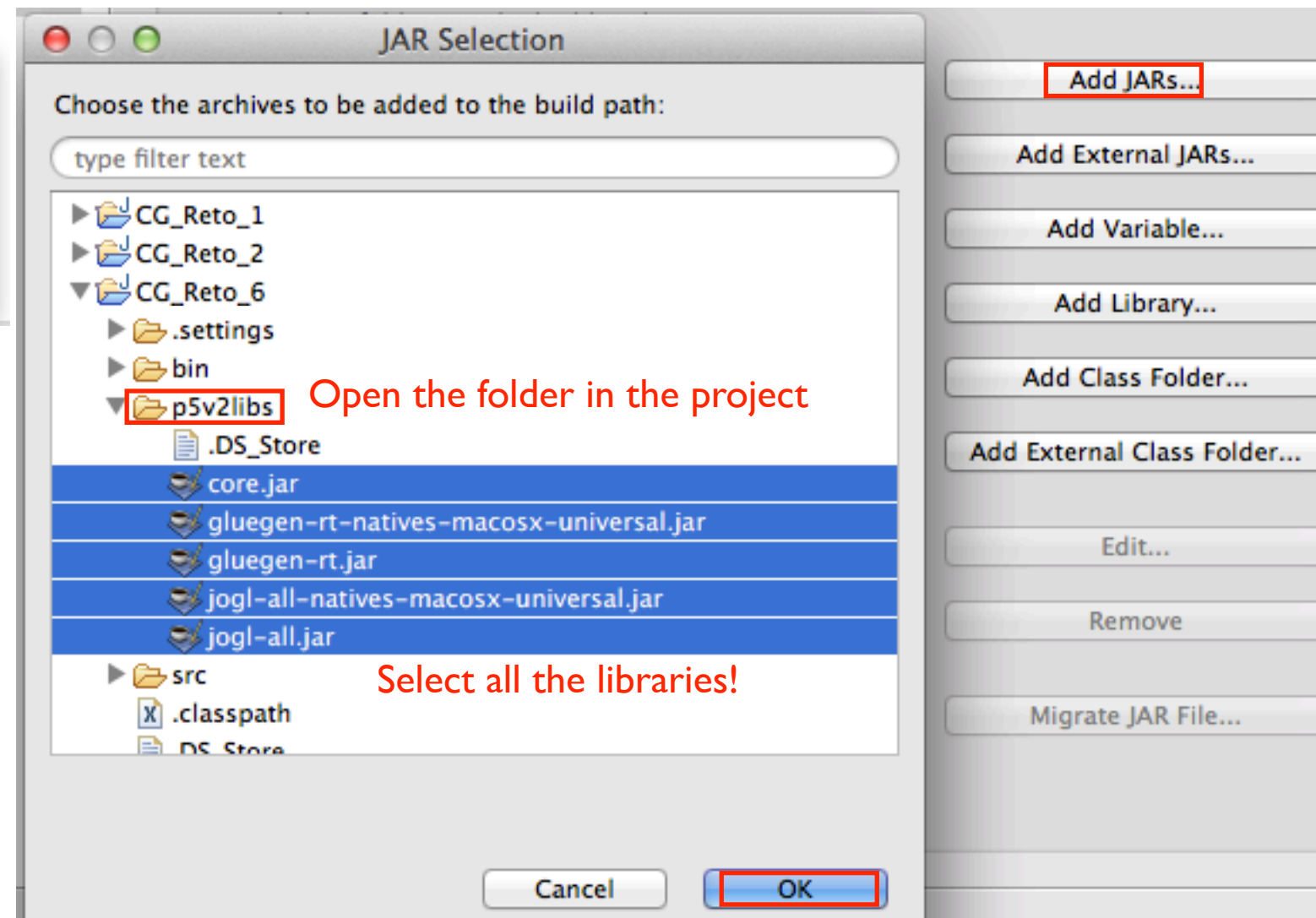


Click here!

Put this folder in the project!



Open the folder in the project



Select all the libraries!

Processing is a framework built on top of Java which allows for awesome graphical capabilities, it has its own transformations but for the purposes of this course the transformation part we're doing on our own.

Projecting

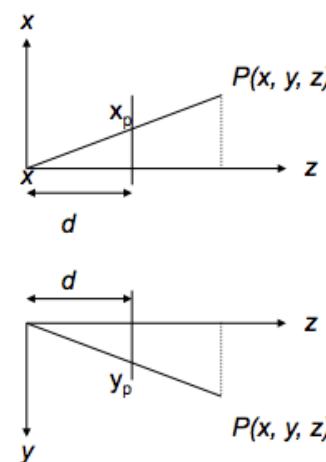
Before, we were asked to project on a plane not too far away from the origin, resulting in glitches whenever an object 'crossed' this projecting boundary. We found out that for projecting its best using a plane very far away from the origin, or at least at a point where shapes won't cross it. The result of this is being able to see the exact representation of a 3D shape with coordinates on X,Y, and Z, by drawing lines only with some projected X and Y parameters.

Another problem with projecting is the lack of a 'Z-index' when drawing, so a line drawn doesn't know if it goes behind or in front of another line, because the 3D shape was 'flattened' into 2D space, and with this loss of depth, the order of 'layers' is also lost. Using a Z-index, no matter the order you draw the shapes, they will always be on top of the shapes with lower Z-index, but without Z-index, the shapes will be drawn in the order of the code, and the last drawn shape will always be on top no matter if its original un-projected shape was behind another shape. With wire-frame drawings this goes unnoticed, but with filling it becomes immediately noticeable.

From the course material.

$$M_{per} \text{ (perspective matrix)} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix}$$

equivalent



□ **Projection plane:**
Orthogonal to the z axis,
distance d from the
origin.

$$\frac{x_p}{d} = \frac{x}{z}, \frac{y_p}{d} = \frac{y}{z}$$

$$x_p = \frac{d \cdot x}{z} = \frac{x}{z/d}; y_p = \frac{d \cdot y}{z} = \frac{y}{z/d}$$

1. Multiply point by Projection Matrix (resulting in all ..points being the same than before but now $w = z / d$)
2. Normalize point (divide everything by w to make it ..again 1 and all points divided by z/d)

```
public void project(){
    float dist = 1000;
    float f = 1f/dist;
    z += dist;
    float matrix[][] = {
        { 1, 0, 0, 0 },
        { 0, 1, 0, 0 },
        { 0, 0, 1, 0 },
        { 0, 0, f, 0 }
    };
    Matrix3D projectMatrix = new Matrix3D(matrix);
    Point3D translatedPoint = Matrix3D.multiplyMatrixAndPoint(projectMatrix, this);
    this.x = translatedPoint.x;
    this.y = translatedPoint.y;
    this.z = translatedPoint.z;
    this.w = translatedPoint.w;
    normalize();
}
```

Method in the Point3D Object

With Homogeneous Coordinates

The result of this matrix will be converting from

$$[X \ Y \ Z \ W]^T \text{ into } [x \ y \ z \ z/d]^T$$

AND THEN NORMALIZE!

Camera

We'll be using a camera that is at a distance from the beginning equal to 100. When this camera rotates, it will rotate around a point 100 units of distance in front of it, like if it were bounded by a sphere. If the position is changed, the point of rotation won't change, but the shapes on the screen will appear to move (but their real position remains unchanged, just the point they're being watched from). There are different commands in the keyboard that will be used to move the camera around:

```
// These are for moving the camera in a certain axis direction
// Z increases outwards, so reducing the camera Z is getting closer
if(e.getKey() == 'w') cam.dZ -= 42;
if(e.getKey() == 's') cam.dZ += 42;
// Moving in the Z axis
if(e.getKey() == 'a') cam.dX -= 10;
if(e.getKey() == 'd') cam.dX += 10;
// Moving in the X axis
if(e.getKey() == 'q') cam.dY += 10;
if(e.getKey() == 'e') cam.dY -= 10;
// Moving in the Y axis
```

Moving the Camera Position

w / s: move inwards / outwards

a / d: move towards left / right

q / e: move towards top / bottom

```
// These are for rotating the camera around a certain axis
if(e.getKey() == 'i') cam.rotateYZ(-10);
if(e.getKey() == 'k') cam.rotateYZ( 10);
// Rotating in the YZ plane, or around the X axis
if(e.getKey() == 'j') cam.rotateZX(-10);
if(e.getKey() == 'l') cam.rotateZX( 10);
// Rotating in the ZX plane, or around the Y axis
if(e.getKey() == 'u') cam.rotateXY(-10);
if(e.getKey() == 'o') cam.rotateXY( 10);
// Rotating in the XY plane, or around the Z axis
```

Rotating the Camera Angle

i / k: rotate in YZ plane

j / l: rotate in ZX plane

u / o: rotate in XY plane

```
if(e.getKey() == 'r') cam = new Camera();
// r is a special button that will reset the camera position
```

Other Camera Operations

r: reset the camera position

Camera

First, an orthonormal base is found. Three vectors that define the camera's coordinate system orientation. n is the vector of what its looking at, u is the vector that points 'above' the camera, and v is a sideways vector. The components will be the cosine-direction components of the unitary vectors, at multiplying a point by this matrix will align the point's coordinate system and the camera's too. Then the point is also translated to compensate for the shifts in the camera position, and then the point is projected!

From the course material.

$$n = \frac{N}{|N|} = (n_x, n_y, n_z) \text{ 'Looking at' vector}$$

N is a vector defined as 'kinda upwards'

$$u = \frac{V \times n}{|V \times n|} = (u_x, u_y, u_z) \text{ 'Sideways' vector}$$

$$v = n \times u = (v_x, v_y, v_z) \text{ 'Upwards' vector}$$

These are all orthogonal to each other, makes what's called an orthonormal base

$$R = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

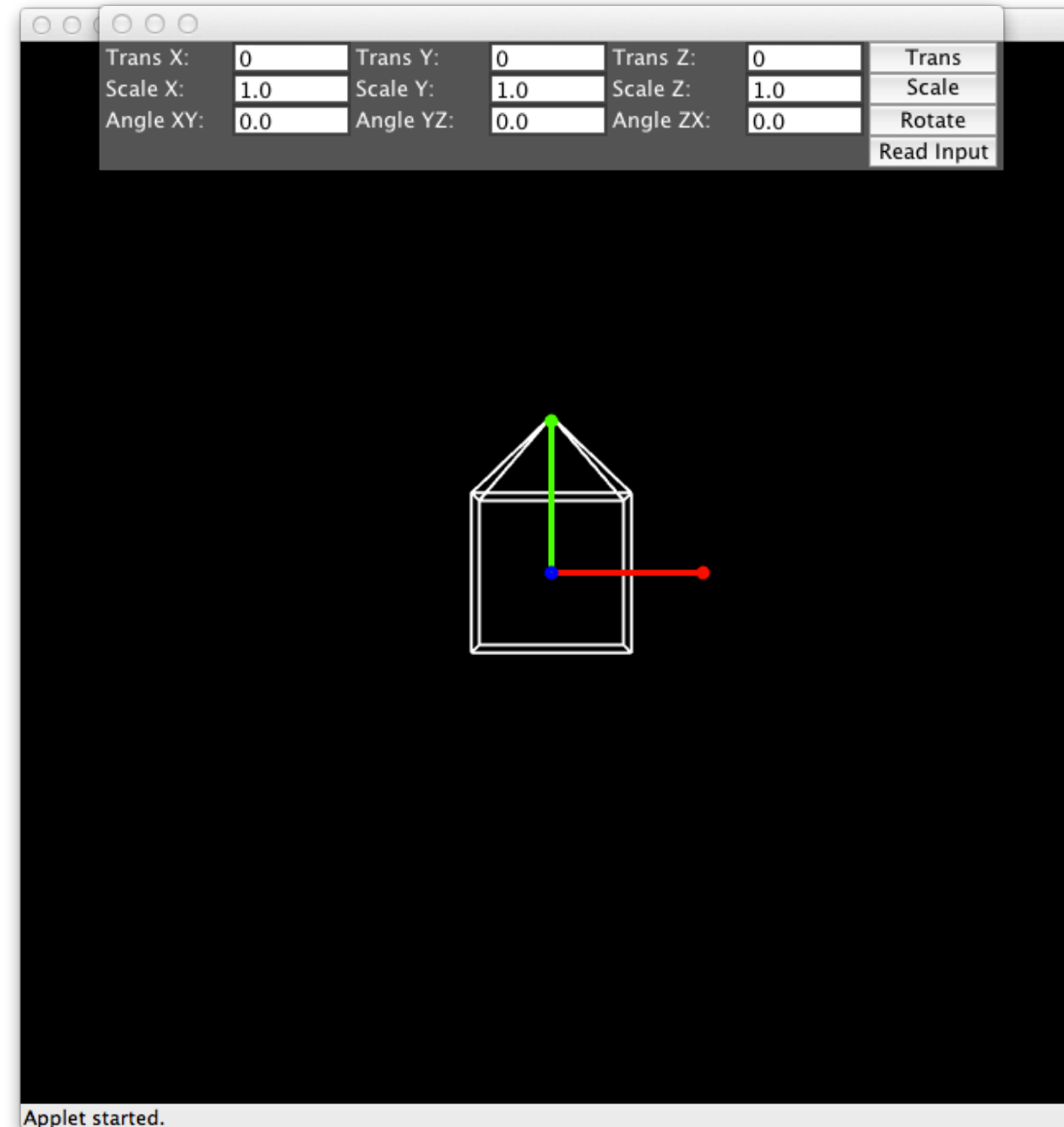
This will align the Camera Coordinate System and the normal Coordinate System, compensating for the Camera Position

```
public void align(Camera cam){
    // Every point before being drawn, is first aligned with the camera's
    // ..own orientation
    Camera camera = cam.clone();
    Vector3D n = camera.n; // The 'looking-at' vector
    n.normalize();
    Vector3D u = cam.u; // The 'kinda up' vector
    u = u.crossProduct(n); // Now a 'sideways' vector
    u.normalize();
    Vector3D v = n.crossProduct(u); // The 'real up' vector
    float matrix[][] = {
        { u.x, u.y, u.z, 0 },
        { v.x, v.y, v.z, 0 },
        { n.x, n.y, n.z, 0 },
        { 0, 0, 0, 1 }
    };
    Matrix3D camMatrix = new Matrix3D(matrix);
    Point3D adjustedPoint = Matrix3D.multiplyMatrixAndPoint(camMatrix, this);
    this.x = adjustedPoint.x;
    this.y = adjustedPoint.y;
    this.z = adjustedPoint.z;
    this.w = adjustedPoint.w;
    // ..then translated to compensate for the camera's changes in X/Y/Z axis
    this.translate(-cam.dX, -cam.dY, -cam.dZ);
    // ..and then projected to our custom plane of projection with a set distance
    // ..of 1000 (Z increases outwards the screen)
    this.project();
}
```

Method in the Point3D Object

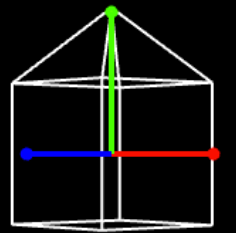
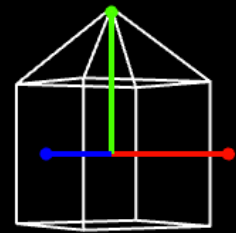
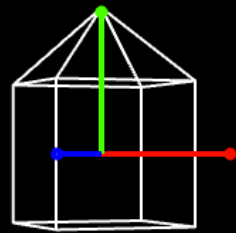
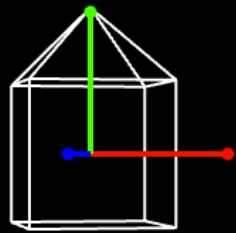
Camera

Default View at Launch



Camera

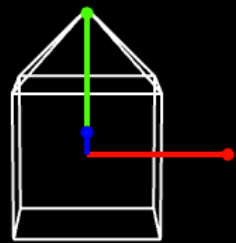
Rotating around Y several times



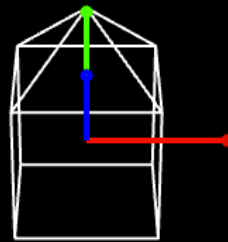
The position of the points or axis aren't changing.
only the point from which the camera observes!

Camera

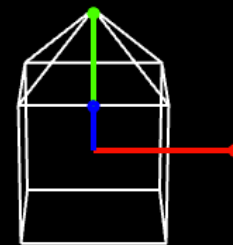
Rotating around X several times



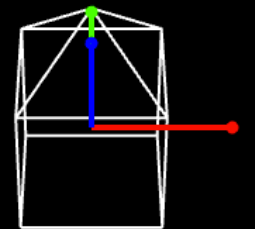
k



k



k

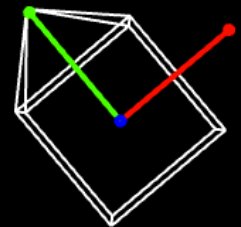
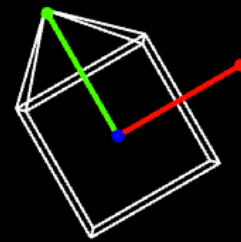
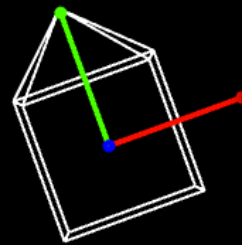
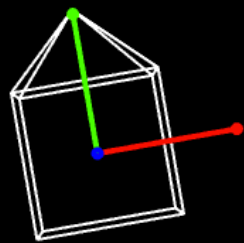


k

The position of the points or axis aren't changing.
only the point from which the camera observes!

Camera

Rotating around Z several times



u

u

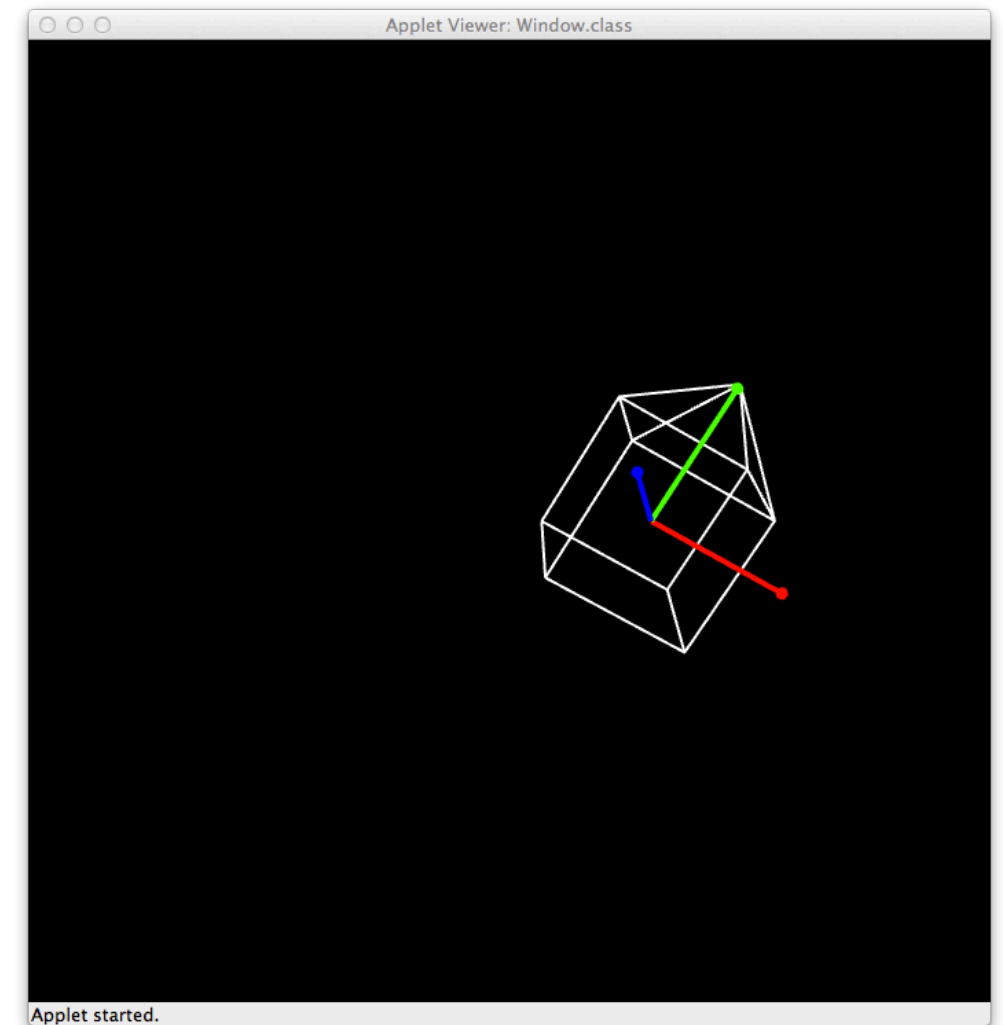
u

u

The position of the points or axis aren't changing.
only the point from which the camera observes!

Camera

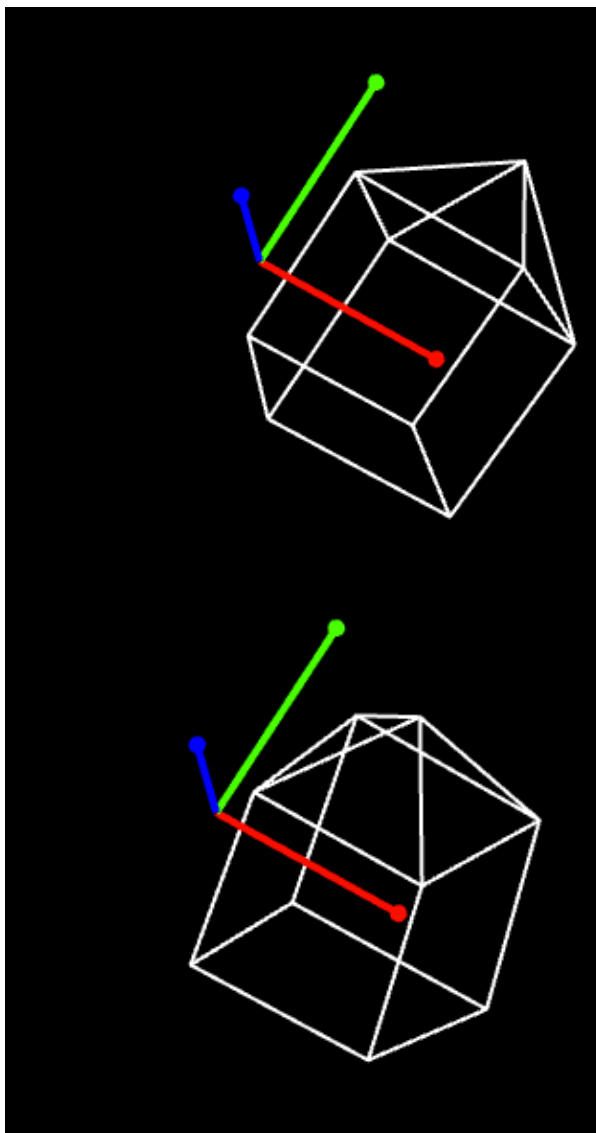
At this moment, the position of the house or the axis are still the same than at the beginning, but the camera has been moved horizontally to the left and it has been rotated quite a few times.



moved in x axis

If transformations are done at this point, they won't happen in the usual axis, but these will also reflect the changes in the camera orientation

rotated around x axis



Thanks for your time!

More challenges to follow!