Introduction to Visual Computing

Assignment# 2

Basics of Computer Graphics

February 24, 2020

Description

With the knowledge that you have gained from the lecture on the basics of computer graphics, you are ready to model a simple object in the 3D space, transform the object, and find its projection onto the viewing plane.

Objectives

- Transform a 3D box (scale, rotate, translate).
- Render the result on the screen using perspective projection.

Specific Challenges

Implementing the perspective projection given the eye position, understanding the homogeneous coordinates and implementing the transformation matrices.

Preliminary steps

- Install Processing, if not already done.
- Setup the Processing environment in 2D rendering mode as the following example:

```
void settings() {
  size (400, 400, P2D);
}
void setup() {
}
void draw() {
  line (200, 200, 400, 400);
}
```

Part I

Perspective Projection

To display a 3D box on the screen, you need to find the position of its 8 vertices on the 2D plane and then simply connect them with 2D lines. So let's start with projecting one 3D point on the screen.

Step 1 – Represent a point

- As shown below, create a class called `My2DPoint` with x and y fields, and a class called `My3DPoint` with x, y, z.

```
class My2DPoint {
  float x;
  float y;
  My2DPoint(float x, float y) {
    this.x = x;
    this.y = y;
  }
}
```

```
class My3DPoint {
  float x;
  float y;
  float z;
  My3DPoint(float x, float y, float z) {
    this.x = x;
    this.y = y;
    this.z = z;
  }
}
```

Note

In Processing, the `PVector` datatype is used to describe 2D and 3D positions. However, for this assignment, in order to clearly distinguish between the 2D and 3D spaces, you make your own classes.

Step 2 – Projecting the point

Now you need to implement a function called `projectPoint` that takes a point in 3D space and the center of projection (eye position), and returns the perspective projection of it on the screen.

```
My2DPoint projectPoint(My3DPoint eye, My3DPoint p) {
    //Complete the code!
}
```

To implement such function you should derive the formulas for the coordinates (x_p, y_p) of a projected (homogeneous) point $(x, y, z, 1)$ seen from the eye coordinates (e_x, e_y, e_z) by first translating the point into the eye reference frame (transformation $\mathbf{T}$), then projecting it (transformation $\mathbf{P}$). As you have seen in the lecture:

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & -e_x \\ 0 & 1 & 0 & -e_y \\ 0 & 0 & 1 & -e_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

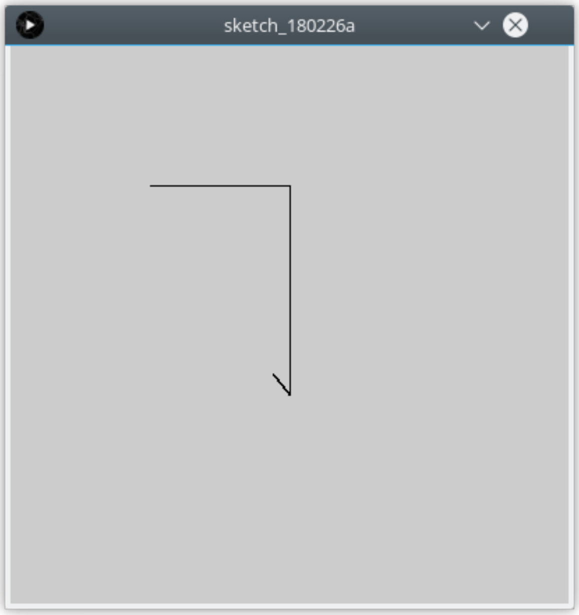
$$\mathbf{P} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{-e_z} & 0 \end{bmatrix}$$

$$\begin{bmatrix} \mathbf{x_p} \\ \mathbf{y_p} \\ -e_z \\ 1 \end{bmatrix} \cong \mathbf{PT} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Try your code out with the following sample:

```
void settings() {
  size(400, 400, P2D);
}

void setup() {
}

void draw() {
  My3DPoint eye = new My3DPoint(-100, -100, -5000);

  My2DPoint p1 = projectPoint(eye, new My3DPoint(0, 0, 0));
  My2DPoint p2 = projectPoint(eye, new My3DPoint(100, 0, 0));
  My2DPoint p3 = projectPoint(eye, new My3DPoint(100, 150, 0));
  My2DPoint p4 = projectPoint(eye, new My3DPoint(100, 150, 300));

  line (p1.x, p1.y, p2.x, p2.y);
  line (p2.x, p2.y, p3.x, p3.y);
  line (p3.x, p3.y, p4.x, p4.y);
}
```

You should obtain:

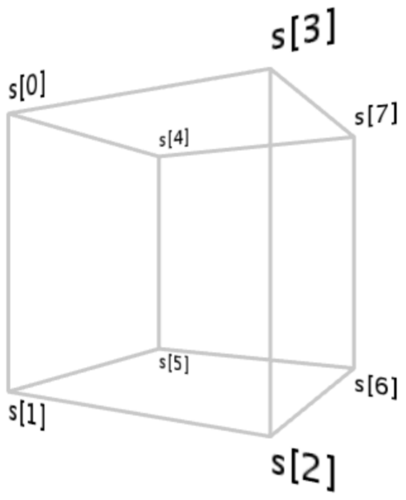


Step 3 – Projecting a cuboid

Now that you have a function to project a point, you just need to apply it to the 8 vertices of your cuboid and draw the result:

- Create a class called `My2DBox` with an array of 8 `My2DPoint` and a `render()` method to draw the 12 edges. Use the ordering that is given in the figure below.

```
class My2DBox {
  My2DPoint[] s;
  My2DBox(My2DPoint[] s) {
      this.s = s;
  }
  void render(){
      // Complete the code! use only line(x1, y1, x2, y2) built-in function.
  }
}
```



- Copy the code below which gives you a class called `My3DBox` with an array of `My3DPoint` and the constructor that takes the position and dimensions of the box, and initializes x, y, and z values of 8 vertices:

```
class My3DBox {
  My3DPoint[] p;
  My3DBox(My3DPoint origin, float dimX, float dimY, float dimZ){
    float x = origin.x;
    float y = origin.y;
    float z = origin.z;
    this.p = new My3DPoint[]{new My3DPoint(x,y+dimY,z+dimZ),
                             new My3DPoint(x,y,z+dimZ),
                             new My3DPoint(x+dimX,y,z+dimZ),
                             new My3DPoint(x+dimX,y+dimY,z+dimZ),
                             new My3DPoint(x,y+dimY,z),
                             origin,
                             new My3DPoint(x+dimX,y,z),
                             new My3DPoint(x+dimX,y+dimY,z)
                            };
  }
  My3DBox(My3DPoint[] p) {
    this.p = p;
  }
}
```

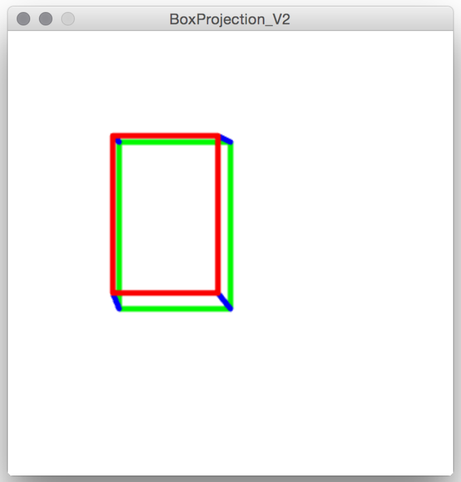
- Use your `projectPoint` function to create a `projectBox` function that takes the eye position and a `My3DBox` object and returns its projection as `My2DBox`.

```
My2DBox projectBox (My3DPoint eye, My3DBox box) {
    // Complete the code!
}
```

- Make sample instances of `My3DBox`, project them on the screen and try to visually make sense of the results. An example is given in the code below.

```
void settings() {
 size(400, 400, P2D);
}

void setup() {
}

void draw() {
  My3DPoint eye = new My3DPoint(-100, -100, -5000);
  My3DPoint origin = new My3DPoint(0, 0, 0); //The first vertex of your cuboid
  My3DBox input3DBox = new My3DBox(origin, 100,150,300);
  projectBox(eye, input3DBox).render();
}
```

You should get a drawing similar to the figure below (if you haven't specified the colors of the lines, they are all black).



Note

Curious about how to change the color of the lines? Check `https://processing.org/reference/`

Note

The color of the back lines might cover the color of the front lines. This is because you are using the P2D environment in which there is no concept of depth and the visible color is the one of the last entity that has been painted. In 3D environment, such as P3D, algorithms take care of rendering objects respecting their depth order.

Part II

Transformations

In this part you will scale, rotate and translate your box using 4x4 transformation matrices. This should be accomplished through the following steps: (1) make the homogeneous representation of the 3D box (that is, of all its vertices), (2) make the transformation matrix, (3) multiply each vertex by the transformation matrix, (4) bring back the results to non-homogeneous coordinates, and (5) project the result on the screen.

Step 1 – Homogeneous Representation

Add to your application the following function. It takes a `My3DPoint` as input and adds a final coordinate of 1, returning a 1x4 matrix.

```
float[] homogeneous3DPoint (My3DPoint p) {
  float[] result = {p.x, p.y, p.z , 1};
  return result;
}
```

Step 2 – Transformation Matrices

Create a set of functions that make the transformation matrices for scaling, translation, and rotation around one of the three axes. One of the functions is given, you have to complete the others.

```
float[][]  rotateXMatrix(float angle) {
  return(new float[][] {{1, 0 , 0 , 0},
                        {0, cos(angle), -sin(angle) , 0},
                        {0, sin(angle) , cos(angle) , 0},
                        {0, 0 , 0 , 1}});
}

float[][]  rotateYMatrix(float angle) {
   // Complete the code!
}

float[][]  rotateZMatrix(float angle) {
   // Complete the code!
}

float[][]  scaleMatrix(float x, float y, float z) {
   // Complete the code!
}

float[][]  translationMatrix(float x, float y, float z) {
   // Complete the code!
}
```

Step 3 – Matrix Product

Create a function called `matrixProduct` that takes a 4x4 matrix and a 4x1 vector and returns their dot product.

```
float[] matrixProduct(float[][] a, float[] b) {
    //Complete the code!
}
```

Create a function called `transformBox` that takes an instance of `My3DBox` and a transformation matrix (4x4), and returns another `My3DBox` with transformed vertices. Notice that after multiplying each vertex by the transformation matrix you need to remove the 4th coordinate (bringing it back to the non-homogeneous coordinate). A function called `euclidian3DPoint` with the following description will do that.

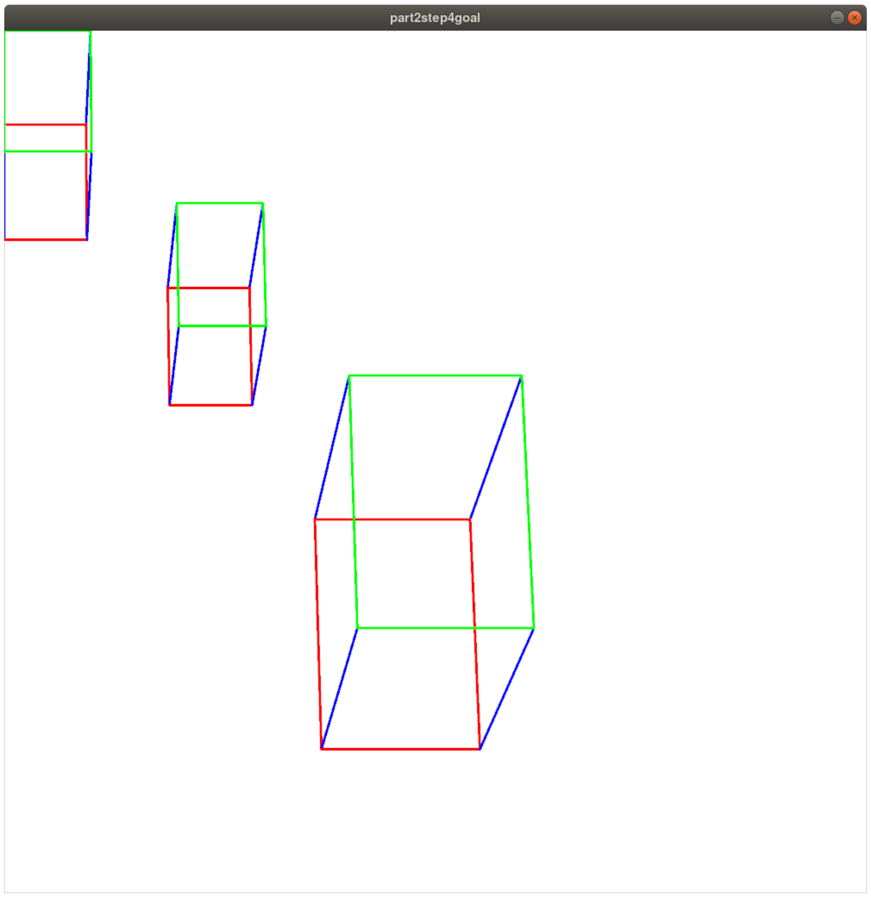
```
My3DBox transformBox(My3DBox box, float[][] transformMatrix) {
    //Complete the code! You need to use the euclidian3DPoint() function given below.
}
```

```
My3DPoint euclidian3DPoint (float[] a) {
  My3DPoint result = new My3DPoint(a[0]/a[3], a[1]/a[3], a[2]/a[3]);
  return result;
}
```

Step 4 – Integration

Use the projection function from Part I and render the transformed box on the screen. Make several instances and try to make sense of the results visually. One example is given here, which should make a drawing similar to the figure below.

```
void settings() {
  size(1000, 1000, P2D);
}

void setup () {
}

void draw() {
  background(255, 255, 255);
  My3DPoint eye = new My3DPoint(0, 0, -5000);
  My3DPoint origin = new My3DPoint(0, 0, 0);
  My3DBox input3DBox = new My3DBox(origin, 100, 150, 300);


  //rotated around x
  float[][] transform1 = rotateXMatrix(-PI/8);
  input3DBox = transformBox(input3DBox, transform1);
  projectBox(eye, input3DBox).render();

  //rotated and translated
  float[][] transform2 = translationMatrix(200, 200, 0);
  input3DBox = transformBox(input3DBox, transform2);
  projectBox(eye, input3DBox).render();

  //rotated, translated, and scaled
  float[][] transform3 = scaleMatrix(2, 2, 2);
  input3DBox = transformBox(input3DBox, transform3);
  projectBox(eye, input3DBox).render();

}
```


part2step4goal