**Rendering 3D display using 2D Engine, Swing**

This program is 3D display viewer program using 2D Engine, Swing.

**Process of rendering 3D on the display**

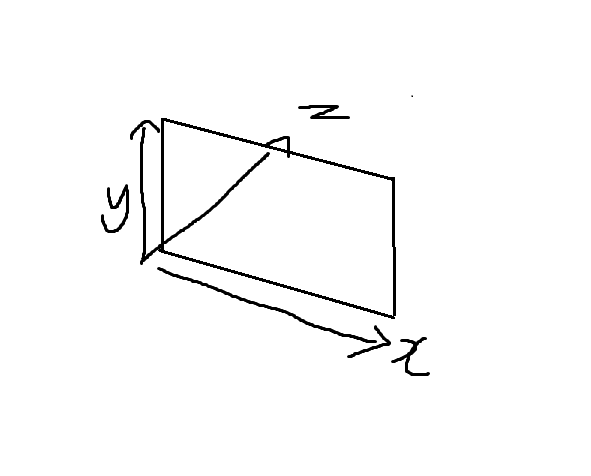
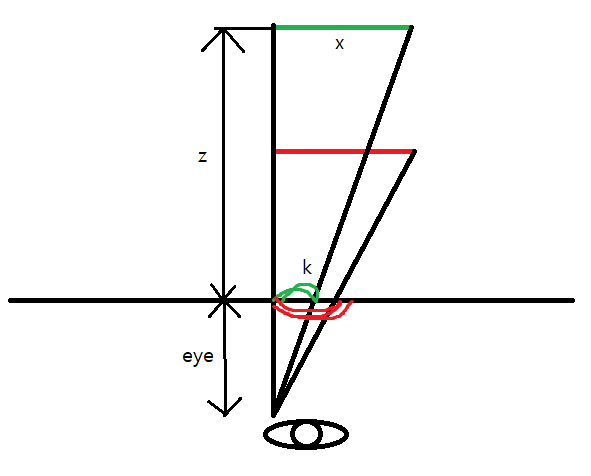
public Point get2DPoint(double eye, double mul) {  
 double x = this.x / (z + eye) \* eye \* mul;  
  
 double y = this.y / (z + eye) \* eye \* mul;  
  
 return new Point(x, y);  
}

Best of thing to need to know is that it is impossible to display 3D in 2D display screen.

So we need to change dot of 3D point to dot of 2D point.

To calculate this, we use the basic logic that uses “a proportional expression” to adapt 3D point to 2D point.

I have an image to help your appreciate.



The first picture is how to calculate a proportional expression visually.

And the second picture is what is Axis of Coordinates.

It has two lines that length is same but distance is different.

If so, it has a difference of length according to eye position and we feel.

so we can change 3d point to 2d point adapting z value, 3D coordinate.

we use the next formula:

we can calculate the final k value to use the proportional expression and map dots to 2d display.

I introduced about only x value, but same y value.

After, I just connected the result dots to see lines.

I need to program density of lines to decrease alpha value of color, but to be simple I skipped.

**Rotation of Dots**

public Point3D getRotatePoint(double yaw, double pitch) {  
 // x1 = b \* cos(pitch) - a \* sin(pitch)  
 // y1 = b \* sin(pitch) + a \* cos(pitch)  
  
 double a = this.x;  
 double b = this.y;  
 double c = this.z;  
 //double z = this.z;  
  
 double d = *degree*(pitch);  
 double d2 = *degree*(yaw);  
  
 double x1 = b \* Math.*cos*(d) - a \* Math.*sin*(d);  
  
 double y1 = b \* Math.*sin*(d) + a \* Math.*cos*(d);  
  
 double x2 = x1;  
 double y2 = c \* Math.*cos*(d2) - y1 \* Math.*sin*(d2);  
 double z2 = c \* Math.*sin*(d2) + y1 \* Math.*cos*(d2);  
  
 return new Point3D(x2, y2, z2);  
}

We can simply express how to rotate dots using trigonometric function.

when you input (x,y), you can get (x1, y1) coordinate rotated by value.

It can only use in 2D but we can use it in 3D (x, y, z) applying both for vertical coordinate and for horizontal coordinate.

So the function returns new Point3D object calculated rotation inputing pitch(horizontal) and yaw(vertical) value.

degree() 함수는 컴퓨터에서 radian을 사용하므로 degree to radian 변환해주는 함수이다.

**구한 2D 점을 모니터에서 표현하기**

public Point toPosition(double midx, double midy, int onesize) {  
 return new Point(midx + x \* onesize, midy - y \* onesize);  
}

우리가 사용하는 (0, 0, 0) 같은 좌표들은 수학적으로 봤을 때 중심을 의미한다.

하지만 이것을 그대로 컴퓨터에 적용하게 되면 컴퓨터는 왼쪽 위가 (0, 0)이므로 왼쪽 위 끝을 기준으로 렌더링되며 현 컴퓨터가 1920 \* 1080 해상도를 갖고있는 상황 상 픽셀 기준 (1,0,0) (2,0,0)등의 값은 너무 미미해 잘 보이지 않는다.

따라서 이를 보정해주기 위해 위의 함수를 사용한다.

midx, 화면의 중심 x좌표

midy, 화면의 중심 y좌표

onesize, 수학적 좌표상 1칸 이동할 때 이동해야할 픽셀 수

**그래픽으로의 표현**

앞에서도 언급했듯이 Swing 그래픽 엔진을 사용한다.

먼저 중앙 패널이다.

public void drawInBuffer() {  
 Graphics2D g = (Graphics2D) buffer.getGraphics();  
  
 // 버퍼에다 싸질러녾은 잡잡한거 초기화  
 g.setColor(Color.*WHITE*);  
 g.fillRect(0, 0, buffer.getWidth(), buffer.getHeight());  
  
 // 선들을 그려낸다  
 for (Pair<Figure, Boolean> figureInfo : figures) {  
 // 색깔을 설정하고  
 Figure figure = figureInfo.getKey();  
 g.setColor(figure.getColor());  
  
 // 그려낼 점들의 위치를 받아온다.  
 List<Point3D> figures = figure.getCoordinates();  
  
 for (int i = 0; i < figures.size(); i++) {  
 Point3D position = figures.get(i);  
  
 // 현재 각도에 맞게 회전을 시켜준다. (3D 회전을 시킬때 x축을 회전하고 y축을 따로 회전시켜도 된다)  
 // 왜 이렇게하면 회전이 되는지는 구글에 많이 나와있다.  
 Point3D point3D = position.getRotatePoint(xAngle, yAngle);  
  
 // get2DPoint: 원하시면 따로 설명 해드립니다 좀 상당히 복잡한 내용ㅇ,,,,,  
 // toPosition: 쓴 좌표를 모니터의 픽셀 위치에 맞게 배치시켜준다.  
 // (실제 Point에는 (1,1) 이런 좌표가 들어있는데 이걸 모니터에 표현해야 하니까. 모니터의 (1,1) 좌표에 아무런 보정도 없이 그려버리면 눈뜨고 보기 쉽지 않을꺼다)  
 Point point = point3D.get2DPoint(eye, mul).toPosition(buffer.getWidth() / 2, buffer.getHeight() / 2, onePixelSize);  
  
 // 점에는 따로 사각형 모양을 해준다. 없애면 그냥 선만 남는다.  
 g.fillRect((int) point.x - 3, (int) point.y - 3, 6, 6);  
 }  
  
 List<Point> connect = figure.getConnects();  
  
 for (int i = 0; i < connect.size(); i++) {  
 Point point = connect.get(i);  
  
 //연결할 두개의 점을 가져온다.  
 Point3D pp1 = figures.get((int) point.x).getRotatePoint(xAngle, yAngle);  
 Point3D pp2 = figures.get((int) point.y).getRotatePoint(xAngle, yAngle);  
  
 // 이걸 설명하려면 원하시면 따로 해드립니다  
 if (Math.*abs*(pp1.z) > eye) {  
 eye = Math.*abs*(pp1.z) \* 3;  
 }  
  
 if (Math.*abs*(pp2.z) > eye) {  
 eye = Math.*abs*(pp2.z) \* 3;  
 }  
  
 // 위와 똑같이 모니터 픽셀에 맞게 좌표를 모니터 픽셀 위치로 변경  
 Point p = pp1.get2DPoint(eye, mul).toPosition(buffer.getWidth() / 2, buffer.getHeight() / 2, onePixelSize);  
 Point p2 = pp2.get2DPoint(eye, mul).toPosition(buffer.getWidth() / 2, buffer.getHeight() / 2, onePixelSize);  
  
 // 선을 그려준다.  
 g.drawLine((int) p.x, (int) p.y, (int) p2.x, (int) p2.y);  
 }  
 }  
}

private JPanel drawPanel = new JPanel() {  
 // 메소드를 오버라이딩 했다  
 // repaint(), paint() 등에 대해 자세한 내용은 https://blog.naver.com/jydev/220735105178  
 @Override  
 public void paint(Graphics g) {  
 // 버퍼에다가 쓴 뒤 draw 하는 이유는 업데이트 할때마다 생기는 깜빡이를 방지하기 위함  
 // 캔버스(화면)를 초기화하고, 캔버스에 그림을 그려내는 그 순간에 깜빡거림이 생김  
 drawInBuffer();  
 g.drawImage(buffer, 0, 0, null);  
 }  
};

Swing에서는 getGraphics()를 이용해 Graphics 객체를 얻으면 이를 통해 Panel또는 Frame에 원하는 도형이나 그림을 그릴 수 있다.

얻은 Graphics 객체에 화면을 회전하거나, 벡터를 추가하는 등 행동을 할 경우 화면을 업데이트 해주는 방식으로 진행한다.

단, Swing 자체의 Panel이나 Graphics 그리고 paint()를 이용하는 경우 깜빡임 현상이 발생한다.

그러므로 더블버퍼링과 비슷하게 이미지와 비슷한 BufferedImage를 생성해 임시로 결과물을 렌더링한 뒤 덮어씌우는 방식으로 진행하여 깜빡임 현상을 막는다.

또한 상단 바를 표현하기 위해 Panel을 추가로 만들어 넣은 코드가 있는데, paint 메소드를 그냥 오버라이딩 해버리는 경우 상단 바가 표현되지 않기 때문에 Panel을 새로 만들어 넣어주었다.

위의 코드에서는 먼저 버퍼로 사용하기 위한 BufferedImage buffer;를 선언하고 Panel의 사이즈와 동일하게 크기를 잡았다.

먼저 buffer를 하얀 배경으로 모두 칠한 뒤 그림을 그리기 시작한다.

Pair<Figure, Boolean>의 Figure는 실질적으로 담겨있는 선에 대한 정보이고, Boolean은 점을 클릭할 시 점의 좌표를 수정할 수 있는지 없는지를 나타낸다.

Figure에서 점들의 집합을 가져와 3D 좌표에 대한 2D 좌표를 구하고, 2D 좌표를 모니터 화면에 매핑시키도록 변환한다. 변환이 완료되면 두 점을 단순히 drawLine을 통하여 연결해준다.

이로써 화면에 벡터를 표시할 수 있게 된다.

현재 이 코드에서 가장 결점이 되는 코드는 다음 코드이다:

if (Math.*abs*(pp1.z) > eye) {  
 eye = Math.*abs*(pp1.z) \* 3;  
}  
  
if (Math.*abs*(pp2.z) > eye) {  
 eye = Math.*abs*(pp2.z) \* 3;  
}

현재 프로그램에는 z값이 0 미만으로 내려가 디스플레이에 표시할 수 없는 경우에 대한 처리가 되어있지 않다.

따라서 z값이 0 미만으로 내려간 값들은 표시가 제대로 안되거나 화면 전체가 깨진 것처럼 줄이 보이게 된다.

해당 문제는 상이 맺히는 디스플레이와 점과 평면간 교점을 구하여 보정하는 방법이 주를 이룰 것으로 예상된다.

하지만 위의 프로그램에선 구현하지 않아도 결과물에 큰 차이가 없어 간단한 구현을 위해 단순한해결 방법을 사용하였다.

음수가 되는 z값이 있으면 그에 맞춰 eye값도 올려줌에 따라 문제를 방지하는 것이다.

비유하여 표현하면, 디스플레이보다 뒤에 맺히는 선들을 보기 위해 디스플레이를 더 뒤로 보내는 것이다. 눈을 더 뒤로 보내는 것이다.

private GoldenList vectorVIewerPanelList = new GoldenList();

public class GoldenList extends JPanel {  
 //private TableBorder border;  
  
 private JPanel panel = new JPanel();  
 private JScrollPane pane = new JScrollPane(panel, JScrollPane.*VERTICAL\_SCROLLBAR\_AS\_NEEDED*, JScrollPane.*HORIZONTAL\_SCROLLBAR\_NEVER*);  
  
 private JLabel bigLabel = new JLabel();  
  
 public GoldenList() {  
 setLayout(new BorderLayout());  
  
 // gridbaglayout은 x,y좌표를 이용해 컴포넌트를 배치할 수 있게 해준다  
 // 비율이나 마진 등 다양한 옵션도 있다  
 // 옵션별 설명은 검색하면 나온다  
 panel.setLayout(new GridBagLayout());  
  
 add(pane, "Center");  
  
 }  
  
 public JScrollPane getScrollPane() {  
 return pane;  
 }  
  
 public void addElement(Component component) {  
 // 아래에 추가했던 공간채우기용 JLabel 지우기  
 if(panel.getComponentCount() > 0) {  
 panel.remove(panel.getComponentCount() - 1);  
 }  
  
 {  
 GridBagConstraints constraints = new GridBagConstraints();  
 //constraints.fill = GridBagConstraints.NORTHWEST;  
 // 이 옵션이 왼쪽 위를 기준으로 늘리는건가 줄이는건가 일겁니다  
 constraints.anchor = GridBagConstraints.*NORTHWEST*;  
  
 // (0, y)로 설정  
 constraints.gridx = 0;  
 constraints.gridy = panel.getComponentCount();  
  
 // 패널 추가하기  
 panel.add(component, constraints);  
 }  
  
 // weight는 비어있는 공간이 있으면 비어있는 공간을 얼마나 비율로 채울 것인가를 정하는 겁니다  
 // weight = 1이면 100%  
 // 0.5면 50%  
  
 // 이걸 왜 추가했냐면 이게 없으면 가운데서부터 컴포넌트들이 나옵니다  
 {  
 GridBagConstraints constraints = new GridBagConstraints();  
 constraints.anchor = GridBagConstraints.*NORTHWEST*;  
 constraints.gridx = 0;  
 constraints.gridy = panel.getComponentCount() + 1;  
 constraints.weightx = 1;  
 constraints.weighty = 1;  
 panel.add(bigLabel, constraints);  
 }  
 }  
  
 public int listSize() {  
 return panel.getComponentCount() - 1;  
 }  
  
 public Component getElement(int index) {  
 return panel.getComponent(index);  
 }  
}

오른쪽에 보이는 리스트와 비슷하게 생긴 화면은 벡터를 추가하면 벡터를 관리할 수 있도록 하는 패널이다. 개인적으로 선호하는 GridBagLayout을 사용했으며 JScrollPane을 이용해 너무 많은 벡터가 추가되면 스크롤하여 내리고 올릴 수 있도록 설계하였다.

GridBagLayout을 일반적으로 사용하면 가운데에 몰리게 되는데, 이를 위쪽으로 배치하기 위해 weight가 1인 빈 패널을 하나 만들어줌으로써 위쪽으로 몰리는 효과를 주도록 하였다.

public void addMenus() {  
 JMenu menuAdd = new JMenu("add");  
  
 JMenuItem menuItemAdd = new JMenuItem("add new vector..");  
 // Alt + V 키보드 누르면 똑같은 효과가 남  
 menuItemAdd.setAccelerator(KeyStroke.*getKeyStroke*(  
 KeyEvent.*VK\_V*, InputEvent.*ALT\_MASK*));  
 menuAdd.add(menuItemAdd);  
  
 menuBar.add(menuAdd);  
  
 // 버튼을 클릭했을때 벡터 추가 화면을 열어준다  
 menuItemAdd.addActionListener(e -> vectorAddDialog.setVisible(true));  
  
 // 메뉴바 설정  
 setJMenuBar(menuBar);  
}

상단 패널은 간단하게 JMenu, JMenuItem등 간단하게 Swing 자체에서 제공하는 메뉴바를 이용해 구성했다.

**벡터 수정**

int adaptX = 0;  
int adaptY = 50;  
  
// 클릭했을때 점들의 실제 좌표랑 마우스 클릭 좌표랑 비교해본다  
@Override  
public void mouseClicked(MouseEvent e) {  
 int index = 0;  
  
 // 현재 존재하는 모든 점들을 루프 돌려본다  
 Loop:  
 for (Pair<Figure, Boolean> figureInfo : figures) {  
 Figure figure = figureInfo.getKey();  
 if (figureInfo.getValue()) {  
 for (Point3D point3D : figure.getCoordinates()) {  
 //각 점의 실제 위치를 구한다  
 Point point = point3D.getRotatePoint(xAngle, yAngle).get2DPoint(eye, mul).toPosition(buffer.getWidth() / 2, buffer.getHeight() / 2, onePixelSize);  
 //System.out.println(point + " // " + e.getX() + ", " + e.getY());  
 // 실제 점의 좌표와 클릭한 마우스간 차이가 10픽셀 이내라면 해당 벡터를 수정할 수 있게 해준다  
 if (Math.*abs*(Math.*abs*(point.x - e.getX()) - adaptX) <= 10 && Math.*abs*(Math.*abs*(point.y - e.getY()) - adaptY) <= 10) {  
 currentEditing = index;  
 vectorEditDialog.setVector(figure.getCoordinates().get(1), figure.getColor());  
 vectorEditDialog.setVisible(true);  
 break Loop;  
 }  
 }  
 }  
 index++;  
 }  
}

점을 클릭하면 벡터를 수정할 수 있는 기능이 있다. 이 기능은 get2DPoint()의 함수를 잘 만들었으므로 간단하게 구현할 수 있다.

단순히 모든 Figure에 들어 있는 모든 점들을 루프 돌려보고 이 점들의 2D 매핑 Point를 구하여 해당 Point의 x와 y값 픽셀 차이가 10 이내이면 클릭한 것으로 판정한다.

adaptX, adaptY는 패널이 메뉴바 등에 의해 좀 더 아래에 렌더링 되는데, 마우스의 위치는 이를 보정해 상대적인 결과를 출력해주지 않는다. 이를 보정하기 위해 적절한 값을 대입해 테스트해가며 좋은 값을 넣어 주었다.

**화면 회전**

addMouseMotionListener(new MouseMotionAdapter() {  
 @Override  
 public void mouseDragged(MouseEvent e) {  
 if (lastX != -1 && lastY != -1) {  
 // 현재 좌표 - 이전 좌표 이렇게 해서 Angle에 더해주거나 빼주는 것  
 // 부호를 바꾸면 반대로 간다  
 yAngle += (e.getX() - lastX) / 2;  
 xAngle -= (e.getY() - lastY) / 2;  
  
 // 화면 업데이트  
 repaint();  
 }  
  
 // 이전 좌표 업데이트  
 lastX = e.getX();  
 lastY = e.getY();  
 }  
  
  
});

매우 단순한 기법이다. 핵심 회전 코드는 앞서 설명한 getRotatePoint()에 모두 구현되어 있다.

가로축, 세로축 에 대한 각각의 회전 정도 변수를 놓고, 마우스를 가로로 움직이면 세로축의 회전 정도를 더하거나 빼고, 세로로 움직이면 가로축의 회전 정도를 더하거나 뺀다.

움직이면 화면의 업데이트가 필요하므로 repaint()를 호출해 전체적으로 화면을 업데이트하도록 한다.

**줌 인 줌 아웃**

addMouseWheelListener(e -> {  
  
 if (e.getWheelRotation() == 1) { // 휠 위쪽 아래쪽 방향에 따라 값이 1이거나 -1이다.  
 // 여담으로 노트북 제스처로도 되더라  
 if (mul >= 0.1) {  
 mul -= 0.1;  
 }  
 } else { /\* if e.getWheelRotation() == -1 \*/  
 if (mul < 20) {  
 mul += 0.1;  
 }  
 }  
 repaint();  
});

휠을 올리거나 내리면 mul 값을 올리거나 내려 배율을 조정하게 된다.

위에서는 mul의 값을 0에서 20 사이의 값 사이로만 가능하도록 하였다.

이 mul 값은 단순히 결과가 될 x와 y 좌표에 곱해주기만 하는 역할을 한다. 즉 중심을 기준으로 거리가 더 늘어나거나 줄어드는 효과를 보여준다.

맨 위의 get2DPoint에서 곱하는 mul 값이 이 mul 값이다.

그리고 휠을 옮겼으면 화면의 업데이트가 필요하므로 repaint()를 호출해 전체적으로 화면을 업데이트 하도록 설계하였다.

**내적의 계산**

if (points.size() >= 2) {  
 Point3D result = new Point3D(1, 1, 1);  
 for (int i = 0; i < points.size(); i++) {  
 Point3D p3d = points.get(i);  
 result.x \*= p3d.x;  
 result.y \*= p3d.y;  
 result.z \*= p3d.z;  
 }  
  
 double sum = result.x + result.y + result.z;  
 JOptionPane.*showMessageDialog*(instance, "counts: " + points.size() + ", result: " + sum);  
  
 initializePoints();  
}

이 프로그램은 벡터의 값을 입력받을 때 x, y, z 성분 3개를 입력 받는다.

성분 내적의 계산은 간단한데, 두 벡터의 값을 입력받아 각 성분끼리 곱하여 각 성분을 더하기만 하면 된다.

따라서 오른쪽 벡터 관리 패널에서 벡터를 선택하면 단순히 x는 x끼리, y는 y끼리, z는 z끼리 모두 곱해 더할 뿐이다.

**외적의 계산**

public Point3D out3D(Point3D p3d) {  
 return new Point3D(y \* p3d.z - z \* p3d.y, z \* p3d.y - x \* p3d.z, x \* p3d.y - y \* p3d.x);  
}

성분 외적의 계산은 다음과 같은 공식을 따른다:

다음과 같은 공식을 이용하여 새로운 성분 벡터를 구하고, 이를 추가할 수 있도록 Dialog를 띄워주도록 설계하였다.

[References]

점 회전 <https://en.wikipedia.org/wiki/Rotation_(mathematics)>

벡터의 외적 <https://j1w2k3.tistory.com/635>