**2D 엔진에서 3D 렌더링하기**

아래 프로그램은 2D 디스플레이에 3D를 띄우는 프로그램입니다.

By GoldenMine (<http://goldenmine.kr>)

개발의 발단

Unity와 같은 좋은 게임 엔진들은 많고 사용법도 편하고 생산성마저 좋다.

그렇지만 단순히 엔진을 사용하는 것 보다는 직접 어느정도 구현은 할 줄 아는 것이 근본적인 실력이라고 생각하였다. 구현을 할 줄 모르면서 다른 API를 가져다 쓰게 되면, 그 API을 개발한 사람 없이는 내가 개발을 할 수 없게 되기 때문이다.

적어도 API를 구현한 사람을 뛰어넘고 더 어려운 것을 구현할 수 있으려면 적어도 이정도는 구현해 봐야 되지 않을까 생각했고 이것은 위 생각에 대한 결과물이다.

다만 물리 엔진은 필연적으로 버그가 많이 생길 수 밖에 없고 세부적인 상황에 대해 모두 대응하기란 어렵다. Unity를 짤 필요 까진 없지만 토대까지는 충분히 구현해볼 가치가 있다고 생각한다.

**버전 1**

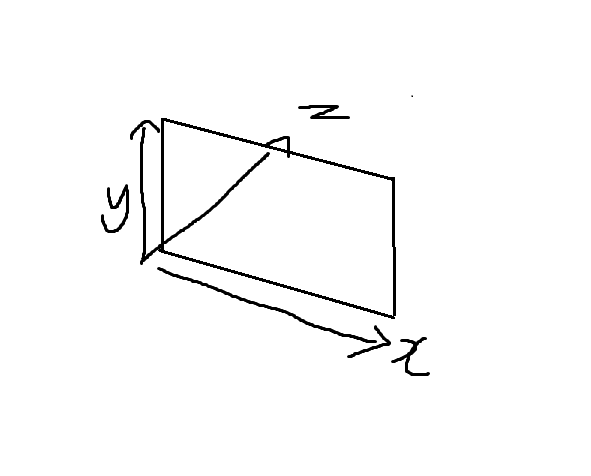
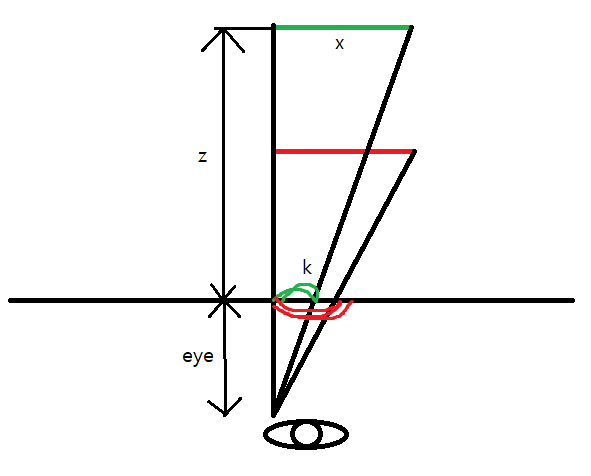
**3D를 디스플레이에 표현하는 과정**

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);  
}

가장 중요한 문제중 하나는 3D로 표현된 점을 2D 디스플레이에 표현하는 게 어렵다는 것이다.

그래서 3D 점을 2D 점으로 바꾸는 과정이 필요하다.

위 프로그램에서는 간단하게 비례식을 사용하여 3D 점을 2D 점으로 변환한다.



왼쪽 사진은 비례식을 사용해 어떻게 2D 디스플레이에 보여줄 것인지에 대한 사진이다.

즉 비례식을 사용하여 길이는 같지만 거리가 멀면 작아보이는 효과를 만들어줄 수 있다.

오른쪽 사진은 간단한 좌표계를 표현한 사진이다.

결론적으로 위와 같은 비례식을 통하여 다음과 같은 공식을 얻을 수 있다.

1차원을 기준으로 하였지만, 확장하여 2차원에서도 가능하다.

그리고 이렇게 나온 결과 점들을 선으로 연결하게 된다.

**점의 회전**

public Point3D getRotatePoint(double yaw, double pitch) {  
 double a = this.x;  
 double b = this.y;  
 double c = 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);  
}

삼각함수를 이용하여 간단하게 구현할 수 있다.

(x, y)는 초기 좌표, (x1, y1)는 변환된 좌표이다.

위의 코드는 2차원에서 사용 가능한 좌표이지만, 회전은 (x,y) 묶어서 따로, (y,z) 묶어서 따로 두 번 계산해주면 3차원에서도 회전을 만들어 낼 수 있다. 그래서 위의 코드에선 최종적으로 x2, y2, z2 변수가 있다.

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**

Swing 그래픽 엔진을 사용한다.

중앙 패널(버퍼에 그림 그리기)

public void drawInBuffer() {  
 Graphics2D g = (Graphics2D) buffer.getGraphics();  
 g.setColor(Color.*WHITE*);  
 g.fillRect(0, 0, buffer.getWidth(), buffer.getHeight());  
 for (Pair<Figure, Boolean> figureInfo : figures) {  
 Figure figure = figureInfo.getKey();  
 g.setColor(figure.getColor());  
 List<Point3D> figures = figure.getCoordinates();  
 for (int i = 0; i < figures.size(); i++) {  
 Point3D position = figures.get(i);  
 Point3D point3D = position.getRotatePoint(xAngle, yAngle);  
 Point point = point3D.get2DPoint(eye, mul).toPosition(buffer.getWidth() / 2, buffer.getHeight() / 2, onePixelSize);  
 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);  
 Point3D pp1 = figures.get((int) point.x).getRotatePoint(xAngle, yAngle);  
 Point3D pp2 = figures.get((int) point.y).getRotatePoint(xAngle, yAngle);  
 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() {  
 @Override  
 public void paint(Graphics g) {  
 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 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());  
 panel.setLayout(new GridBagLayout());  
 add(pane, "Center");  
 }  
 public JScrollPane getScrollPane() {  
 return pane;  
 }  
 public void addElement(Component component) {  
 if(panel.getComponentCount() > 0) {  
 panel.remove(panel.getComponentCount() - 1);  
 }  
 {  
 GridBagConstraints constraints = new GridBagConstraints();  
 constraints.anchor = GridBagConstraints.*NORTHWEST*;  
 constraints.gridx = 0;  
 constraints.gridy = panel.getComponentCount();  
 panel.add(component, constraints);  
 }  
 {  
 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..");  
 menuItemAdd.setAccelerator(KeyStroke.*getKeyStroke*(  
 KeyEvent.*VK\_V*, InputEvent.*ALT\_MASK*));  
 menuAdd.add(menuItemAdd);  
 menuBar.add(menuAdd);  
 menuItemAdd.addActionListener(e -> vectorAddDialog.setVisible(true));  
 setJMenuBar(menuBar);  
}

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

**벡터 수정**

int adaptX = 0;  
int adaptY = 50;  
@Override  
public void mouseClicked(MouseEvent e) {  
 int index = 0;  
 Loop:  
 for (Pair<Figure, Boolean> figureInfo : figures) {  
 Figure figure = figureInfo.getKey();  
 if (figureInfo.getValue()) {  
 for (Point3D point3D : figure.getCoordinates()) {  
 Point point = point3D.getRotatePoint(xAngle, yAngle).get2DPoint(eye, mul).toPosition(buffer.getWidth() / 2, buffer.getHeight() / 2, onePixelSize);  
 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) {  
 yAngle += (e.getX() - lastX) / 2;  
 xAngle -= (e.getY() - lastY) / 2;  
  
 repaint();  
 }  
  
 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) {   
 if (mul >= 0.1) {  
 mul -= 0.1;  
 }  
 } else {  
 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.

**버전 2**

버전 1에서 몇몇 코드를 리팩토링하였고 게임처럼 프레임마다 화면을 업데이트 하도록 바꾸었다.

프레임마다 화면을 업데이트 하지 않으면 낮은 확률로 ConcurrentModificationException 등 동기화 관련 문제가 뜨기 때문이다. Synchronized는 성능이 떨어지므로 결국 바꾸게 되었다.

중력, 운동량 보존 법칙을 이용한 충돌시 이후 속도 계산을 구현하였다.

**이후 코드는 모두 코틀린으로 작성되었다.**

**중력**

override val lastPos: Vector3d = Vector3d(currentPos)  
override val acceleration = Vector3d(0.0, 0.0, -gravity, true)  
override fun calculateNextPosition() {  
 velocity += acceleration  
 lastPos.copyFrom(currentPos)  
 currentPos.addThis(velocity)  
}

중력의 경우는 단순하다. (x,y는 동서남북, z는 상하로 가정)

속도 벡터(v), 가속도 벡터(a)를 두고 a 벡터의 z 요소에 고정 값 -9.8을 넣어주면 된다.

단, 프로그램 상 9.8은 너무 큰 값이므로 실제론 작은 값으로 넣었다.

일정 시간마다 v벡터에 a벡터를 계속 더하도록 설계하였으며

v벡터또한 실제 position에 값을 더해주는 방식으로 실제 가속도가 위치에 미치는 현상을 구현하였다.

**운동량 보존 법칙 – 충돌 계산**

먼저 운동량 보존 법칙을 이용해 두 물체들을 튕겨내기 전에 해야 할 것은 두 물체가 충돌했는 지부터 계산해야 한다.

계산을 단순하게 하기 위해 점(공), 직육면체로 된 물체만 있다고 가정했다.

**점과 직육면체 사이의 충돌 계산**

public CollideResult checkCollision(Dot dot, SkyRectangle rectangle) {  
 final Vector3d rectangleSize = rectangle.getSize();  
 final Vector3d startPos = rectangle.getCurrentPos();  
 final Vector3d finishPos = startPos.add(rectangleSize);  
 final Vector3d lineDirection = dot.getVelocity();   
 final Vector3d linePos = dot.getCurrentPos();  
 final Vector3d linePosLast = dot.getLastPos();  
 /\* 중략 \*/  
 final Vector3d faceZDirection = new Vector3d(0, 0, 1, false);  
 /\* 중략 \*/  
 final Vector3d intersectZ = *collide*(faceZDirection, startPos, lineDirection, linePos);  
 if(lineDirection.getZ() >= 0 && intersectZ.between(startPos, finishPos, *EPSILON*) && intersectZ.between(linePosLast, linePos, *EPSILON*)) return new CollideResult(intersectZ, Face.*Z*);  
 /\* 중략 \*/  
 return null;  
 }

(위 코드는 레거시 코드가 자바여서 자바로 구현되어 있다, java.kr.goldenmine.DisplayGUI: 133)

fun collide(faceDirection: Vector3d, facePos: Vector3d, lineDirection: Vector3d, linePos: Vector3d): Vector3d {  
 val t = -(linePos.dot(faceDirection) - facePos.dot(faceDirection)) / lineDirection.dot(faceDirection)  
 return lineDirection \* t + linePos   
}

기하와 벡터에서 다뤘던 평면과 직선 사이의 교점 개념을 이용했다.

평면을 만들 때 평면의 법선 벡터 하나와 평면을 이루는 점의 위치 하나면 만들 수 있다.

따라서 직육면체의 6면은 다음과 같은 법선 벡터중 하나만 가지게 된다: x, (1, 0, 0) / y, (0, 1, 0) / z, (0, 0, 1) 그리고 점의 위치는 startPos, finishPos가 담당하게 된다.

이렇게 해서 교점을 구했으면 해당하는 교점이 이전 위치와 현재 위치 사이에서 발생한 교점인지 확인하고, 정육면체 안에서 교점이 생겼는지도 확인한다. (평면은 일반적으로 크기가 무한하므로 교점의 위치를 한 번 더 확인해야 한다)

충돌했을 때 운동량 계산

fun applyNewVelocity(rectangle: SkyRectangle, rectangle2: SkyRectangle) {  
 val rectangleWeight = *getWeight*(rectangle) \* *ADAPT\_WEIGHT* val rectangle2Weight = *getWeight*(rectangle2) \* *ADAPT\_WEIGHT* val e = 0.4  
  
 val lastRectangleVelocity = rectangle.velocity.clone()  
 val lastRectangle2Velocity = rectangle2.velocity.clone()  
  
 rectangle.velocity -= rectangle2Weight \* (1 + e) / (rectangleWeight + rectangle2Weight) \* (lastRectangleVelocity - lastRectangle2Velocity)  
 rectangle2.velocity += rectangleWeight \* (1 + e) / (rectangleWeight + rectangle2Weight) \* (lastRectangleVelocity - lastRectangle2Velocity)  
}

반발 계수 e = (v1’ – v2’) / (v2 – v1)

m1 \* v1 + m2 \* v2 = m1 \* v1’ + m2 \* v2’

임을 이용하여 점과 평면간 새로운 속도를 각 물체에 적용해주게 된다.

공과 정육면체 사이에는 충돌 계수 0.7, 정육면체 사이에는 0.4가 적용되었다.

직육면체와 직육면체 사이의 계산

public boolean checkCollision(SkyRectangle rec1, SkyRectangle rec2) {  
 Vector3d rec1StartPos = rec1.getCurrentPos();  
 Vector3d rec1FinishPos = rec1StartPos.add(rec1.getSize());  
 Vector3d rec2StartPos = rec2.getCurrentPos();  
 Vector3d rec2FinishPos = rec2StartPos.add(rec2.getSize());  
 return rec1.getCoordinates().stream().anyMatch(it -> it.between(rec2StartPos, rec2FinishPos, -*EPSILON*)) ||  
 rec2.getCoordinates().stream().anyMatch(it -> it.between(rec1StartPos, rec1FinishPos, -*EPSILON*));  
}

충돌 여부를 파악하는 것은 간단하다. 단순히 직육면체의 8개의 점에 대하여 그 점중 하나라도 다른 직육면체 안에 들어가 있는 지 파악하면 된다.

[References]

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

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