```java
package com.blindtigergames.werescrewed.entity.platforms;

import com.badlogic.gdx.graphics.Texture;
import com.badlogic.gdx.math.Vector2;
import com.badlogic.gdx.physics.box2d.BodyDef;
import com.badlogic.gdx.physics.box2d.BodyDef.BodyType;
import com.badlogic.gdx.physics.box2d.FixtureDef;
import com.badlogic.gdx.physics.box2d.Joint;
import com.badlogic.gdx.physics.box2d.PolygonShape;
import com.badlogic.gdx.physics.box2d.World;
import com.badlogic.gdx.physics.box2d.joints.RevoluteJointDef;
import com.badlogic.gdx.utils.Array;
import com.blindtigergames.werescrewed.entity.Entity;
import com.blindtigergames.werescrewed.entity.EntityDef;
import com.blindtigergames.werescrewed.entity.EntityType;
import com.blindtigergames.werescrewed.entity.Skeleton;
import com.blindtigergames.werescrewed.entity.Sprite;
import com.blindtigergames.werescrewed.level.Level;
import com.blindtigergames.werescrewed.util.Util;

/**
 * Platform Mostly just an inherited class, but complex platform uses that as
 * it's main class
 *
 * @author Ranveer / Stew
 *
 */

public class Platform extends Entity {

    // ============================================
    // Fields
    // ============================================
    protected float width, height;
    protected boolean dynamicType = false;
    protected boolean rotate = false;
    public boolean oneSided = false;
    public boolean moveable = false;
    // tileConstant is 16 for setasbox function which uses half width/height
    // creates 32x32 objects
    protected static final int tileConstant = 16;
    /**
     * Use this for any tile size calculations
     */
    public static final int tile = 32;
    protected PlatformType platType;

    /**
     * Used for kinematic movement connected to skeleton. Pixels.
     */
    protected Vector2 localPosition; // in pixels, local coordinate system
    protected Vector2 previousPosition;
    protected Vector2 prevBodyPos;
    private float localRotation; // in radians, local rot system
    protected float previousRotation;
    protected float prevBodyAngle;
    protected Vector2 localLinearVelocity; // in meters/step
    protected float localAngularVelocity; //
    protected Vector2 originPosition; // world position that this platform
                                        // spawns
                                        // at, in pixels

    private Vector2 originRelativeToSkeleton; // box meters

    protected Joint extraSkeletonJoint;
    private boolean firstStep = true;

    // ===============================================
    // Constructors
    // ===============================================

    /**
     * General purpose platform constructor for things that don't use an
     * entitydef. Currently used by PlatformBuilder and Tiled Platform
     *
     * @param name
     * @param pos
     * @param tex
     * @param world
     */
    public Platform( String name, Vector2 pos, Texture tex, World world ) {
        super( name, pos, tex, null, true );
        this.world = world;
        entityType = EntityType.PLATFORM;
        init( pos );
    }

    /**
     * Construct platforms using an EntityDef. This is used by
     * PlatformBuilder.buildComplexBody()
     *
     * @param name
     * @param type
     * @param world
     * @param pos
     * @param rot
     * @param scale
     */
    public Platform( String name, EntityDef type, World world, Vector2 pos,
            float rot, Vector2 scale ) {
        super( name, type, world, pos, rot, scale, null, true );
        entityType = EntityType.PLATFORM;
        init( pos );
    }

    /**
     * Loading a Complex platform, or used to load complex Hazard
     *
     * (no scale or rotation because its defined in entitydef)
     *
     * @param name
     * @param type
     * @param world
     * @param pos
     */

    public Platform( String name, EntityDef type, World world, Vector2 pos ) {
        super( name, type, world, pos, null );
        entityType = EntityType.PLATFORM;
        init( pos );
    }

    /**
     * Initialize things.
     *
     * @author stew
     * @param pos
     */
    void init( Vector2 pos ) {
        localPosition = new Vector2( 0, 0 );
        previousPosition = new Vector2( localPosition.x, localPosition.y );
        prevBodyPos = new Vector2( 0, 0 );
        localLinearVelocity = new Vector2( 0, 0 );
        localRotation = 0;
```

```java
        previousRotation = localRotation;
        originPosition = pos.cpy( );
        platType = PlatformType.DEFAULT; // set to default unless subclass sets
                                            // it later in a constructor
        originRelativeToSkeleton = new Vector2( );
    }

    // ============================================
    // Methods
    // ============================================

    /**
     * return localPosition Vector2 in PIXELS.
     * 
     * @return
     */
    public Vector2 getLocalPos( ) {
        return localPosition;
    }

    /**
     * set localPosition Vector2 in PIXELS!!!
     * 
     * @param newLocalPos
     *            in PIXELS
     */
    public void setLocalPos( Vector2 newLocalPosPixel ) {
        setLocalPos( newLocalPosPixel.x, newLocalPosPixel.y );
    }

    public void setLocalPos( float xPixel, float yPixel ) {
        localPosition.x = xPixel;
        localPosition.y = yPixel;
    }

    /**
     * returns local rotation in RADIANS
     */
    public float getLocalRot( ) {
        return localRotation;
    }

    /**
     * returns previous location last time it moved
     */
    public boolean hasMoved( ) {
        Vector2 bodyPos = body.getPosition( ).mul( Util.BOX_TO_PIXEL );
        if ( previousPosition.x != localPosition.x
                || previousPosition.y != localPosition.y
                || ( body != null && ( prevBodyPos.x != bodyPos.x || prevBodyPos.y
!= bodyPos.y ) ) ) {
            return true;
        }
        return false;
    }

    /**
     * set the previous position to this position
     */
    public void setPreviousTransformation( ) {
        Vector2 bodyPos = body.getPosition( ).mul( Util.BOX_TO_PIXEL );
        previousPosition = new Vector2( localPosition.x, localPosition.y );
        if ( body != null ) {
            prevBodyPos = new Vector2( bodyPos.x, bodyPos.y );
            prevBodyAngle = body.getAngle( );
        }
        previousRotation = localRotation;
    }

    /**
     * returns previous rotation last time it rotated
     */
    public boolean hasRotated( ) {
        if ( previousRotation != localRotation
                || prevBodyAngle != body.getAngle( ) ) {
            return true;
        }
        return false;
    }

    @Override
    public void updateDecals( float deltaTime ) {
        if ( firstStep || hasMoved( ) || hasRotated( ) || this.currentMover( ) !=
null ||
                ( this.getParentSkeleton( ) != null && ( this.getParentSkeleton(
).hasMoved( ) ||
                this.getParentSkeleton( ).hasRotated( )
                || this.getParentSkeleton( ).currentMover( ) != null ) ) ) {
            Vector2 bodyPos = this.getPositionPixel( );
            float angle = this.getAngle( ), cos = ( float ) Math.cos( angle ), sin
    = ( float ) Math
                    .sin( angle );
            float x, y, r;
            Vector2 offset;
            Sprite decal;
            float a = angle * Util.RAD_TO_DEG;
            for ( int i = 0; i < fgDecals.size( ); i++ ) {
                offset = fgDecalOffsets.get( i );
                decal = fgDecals.get( i );
                r = fgDecalAngles.get( i );
                x = bodyPos.x + ( ( offset.x ) * cos ) - ( ( offset.y ) * sin );
                y = bodyPos.y + ( ( offset.y ) * cos ) + ( ( offset.x ) * sin );
                decal.setPosition( x + decal.getOriginX( ),
                        y + decal.getOriginY( ) );
                decal.setRotation( r + a );
            }
            for ( int i = 0; i < bgDecals.size( ); i++ ) {
                offset = bgDecalOffsets.get( i );
                decal = bgDecals.get( i );
                r = bgDecalAngles.get( i );
                x = bodyPos.x + ( ( offset.x ) * cos ) - ( ( offset.y ) * sin );
                y = bodyPos.y + ( ( offset.y ) * cos ) + ( ( offset.x ) * sin );
                decal.setPosition( x + decal.getOriginX( ),
                        y + decal.getOriginY( ) );
                decal.setRotation( r + a );
            }
        }
        firstStep = false;
    }

    /**
     * set local rotation in RADIAN
     *
     * @param newLocalRotRadians
     */
    public void setLocalRot( float newLocalRotRadians ) {
        localRotation = newLocalRotRadians;
    }

    /**
     * return originPosition Vector2 in PIXELS.
     *
     * @return
     */
    public Vector2 getOriginPos( ) {
        return originPosition;
    }

    /**
     * set Origin Position Vector2 in PIXELS!!!
     *
     * @param newLocalPos
     *            in PIXELS
     */
    public void setOriginPos( Vector2 newOriginPosPixel ) {
        originPosition.x = newOriginPosPixel.x;
        originPosition.y = newOriginPosPixel.y;
    }

    public void setOriginPos( float xPixel, float yPixel ) {
        originPosition.x = xPixel;
        originPosition.y = yPixel;
    }

    public Vector2 getLocLinearVel( ) {
        return localLinearVelocity;
    }

    public void setLocLinearVel( Vector2 linVelMeters ) {
        localLinearVelocity = linVelMeters.cpy( );
    }

    public void setLocLinearVel( float xMeter, float yMeter ) {
        localLinearVelocity.x = xMeter;
        localLinearVelocity.y = yMeter;
    }

    public float getLocAngularVel( ) {
        return localAngularVelocity;
    }

    public void setLocAngularVel( float angVelMeter ) {
        localAngularVelocity = angVelMeter;
    }

    @Override
    public void setAwake( ) {
        body.setAwake( true );
    }
    @Override
    public void update( float deltaTime ) {
        super.update( deltaTime );
        if ( removeNextStep ) {
            remove( );
        }
    }

    /**
     * Swap from kinematic to dynamic.
     */
    public void changeType( ) {
        dynamicType = !dynamicType;
        if ( dynamicType ) {
            body.setType( BodyType.DynamicBody );
            // Filter filter = new Filter( );
            // for ( Fixture f : body.getFixtureList( ) ) {
            // filter = f.getFilterData( );
            // // move player back to original category
            // filter.categoryBits = Util.CATEGORY_PLATFORMS;
            // // player now collides with everything
            // filter.maskBits = Util.CATEGORY_EVERYTHING;
            // f.setFilterData( filter );
            // }
        } else {
            body.setType( BodyType.KinematicBody );
            // Filter filter = new Filter( );
            // for ( Fixture f : body.getFixtureList( ) ) {
            // filter = f.getFilterData( );
            // // move player back to original category
            // filter.categoryBits = Util.CATEGORY_PLATFORMS;
            // // player now collides with everything
            // filter.maskBits = Util.CATEGORY_EVERYTHING;
            // f.setFilterData( filter );
            // }
        }

        body.setActive( false );
    }

    // This function sets the platform to 180* no matter what angle it currently
    // is
    public void setHorizontal( ) {
        body.setTransform( body.getPosition( ), ( float ) Math.toRadians( 90 ) );
    }
```

```java
    // This function sets platform to 90*
    public void setVertical( ) {
        body.setTransform( body.getPosition( ), ( float ) Math.toRadians( 180 ) );
    }

    public boolean getOneSided( ) {
        return oneSided;
    }

    public void setOneSided( boolean value ) {
        oneSided = value;
    }

    protected void rotate( ) {
        body.setAngularVelocity( 1f );
    }

    protected void rotateBy90( ) {
        float bodyAngle = body.getAngle( );
        body.setTransform( body.getPosition( ), bodyAngle + 90 );
    }

    /**
     * Returns the private member platform type for casting or whatever
     *
     * @return PLATFORMTYPE
     */
    public PlatformType getPlatformType( ) {
        return platType;
    }

    /**
     * Set this platforms type!!
     *
     * @author stew
     * @param newPlatformType
     */
    public void setPlatformType( PlatformType newPlatformType ) {
        platType = newPlatformType;
    }

    /**
     * Set the position and angle of the kinematic platform based on the parent
     * skeleton's pos/rot. Now better than ever! Use this to set a platform's
     * velocity so the platform does normal phsyics.
     *
     * @param frameRate
     *            which is typically 1/deltaTime.
     * @param skeleton
     *
     * @author stew
     */
    public void setTargetPosRotFromSkeleton( float frameRate, Skeleton skeleton ) {
        if ( skeleton != null ) {
            Vector2 posOnSkeleLocalMeter = originRelativeToSkeleton.cpy( ).add(
                    localPosition.cpy( ).mul( Util.PIXEL_TO_BOX ) );
            float radiusFromSkeletonMeters = posOnSkeleLocalMeter.len( );
            float newAngleFromSkeleton = skeleton.body.getAngle( )
                    + Util.angleBetweenPoints( Vector2.Zero,
                            posOnSkeleLocalMeter );

            Vector2 targetPosition = Util.PointOnCircle(
                    radiusFromSkeletonMeters, newAngleFromSkeleton,
                    skeleton.getPosition( ) ).sub( body.getPosition( ) );
            float targetRotation = localRotation + skeleton.body.getAngle( )
                    - body.getAngle( );

            body.setLinearVelocity( targetPosition.mul( frameRate ) );
            body.setAngularVelocity( targetRotation * frameRate );
        }
    }

    /**
     * This function TRANSLATES a platform, so it won't act with normal physics.
     * This is mainly used for event triggers.
     *
     * @param skeleton
     * @author stew
     */
    public void translatePosRotFromSKeleton( Skeleton skeleton ) {
        if ( skeleton != null ) {
            Vector2 posOnSkeleLocalMeter = originRelativeToSkeleton.cpy( ).add(
                    localPosition.cpy( ).mul( Util.PIXEL_TO_BOX ) );

            if ( posOnSkeleLocalMeter.equals( Vector2.Zero ) ) {
                body.setTransform( skeleton.body.getPosition( ), localRotation
                        + skeleton.body.getAngle( ) );
            } else {
                float radiusFromSkeletonMeters = posOnSkeleLocalMeter.len( );
                float newAngleFromSkeleton = skeleton.body.getAngle( );
                newAngleFromSkeleton += Util.angleBetweenPoints( Vector2.Zero,
                        posOnSkeleLocalMeter );

                Vector2 targetPosition = Util.PointOnCircle(
                        radiusFromSkeletonMeters, newAngleFromSkeleton,
                        skeleton.getPosition( ) );
                float targetRotation = localRotation + skeleton.body.getAngle( );

                body.setTransform( targetPosition, targetRotation );
            }

        }
    }

    @Override
    public void setCrushing( boolean value ) {
        crushing = value;
        oneSided = false;
    }

    public Vector2 getOriginRelativeToSkeleton( ) {
        return originRelativeToSkeleton;
    }

    public void setOriginRelativeToSkeleton( Vector2 originRelativeToSkeleton ) {
        this.originRelativeToSkeleton = originRelativeToSkeleton;
    }

    public void constructBodyFromVerts( Array< Vector2 > loadedVerts,
            Vector2 positionPixel ) {
        BodyDef bodyDef = new BodyDef( );
        bodyDef.position.set( positionPixel.mul( Util.PIXEL_TO_BOX ) );
        body = world.createBody( bodyDef );

        PolygonShape polygon = new PolygonShape( );
        Vector2[ ] verts = new Vector2[ loadedVerts.size - 1 ];

        // MAKE SURE START POINT IS IN THE MIDDLE
        // AND SECOND AND END POINT ARE THE SAME POSITION
        int i = 0;
        for ( int j = 0; j < loadedVerts.size; j++ ) {
            if ( j == loadedVerts.size - 1 )
                continue;
            Vector2 v = loadedVerts.get( j );
```

```java
            verts[ i ] = new Vector2( v.x * Util.PIXEL_TO_BOX, v.y
                    * Util.PIXEL_TO_BOX );
            ++i;
        }
        polygon.set( verts );

        FixtureDef fixture = new FixtureDef( );
        fixture.shape = polygon;

        body.createFixture( fixture );
        body.setUserData( this );

        polygon.dispose( );
    }

    /**
     * This function is used to joint a platform to a skeleton so that it stays
     * in place also this way we save the reference to that particular joint so
     * we can delete it later
     *
     * @param skel
     */
    public void addJointToSkeleton( Skeleton skel ) {
        RevoluteJointDef rjd = new RevoluteJointDef( );
        rjd.initialize( body, skel.body, body.getWorldCenter( ) );
        extraSkeletonJoint = ( Joint ) this.world.createJoint( rjd );
    }

    /**
     * Adds the joint (connected to a skeleton) to the list to remove it when
     * the Box2d world is not locked() (otherwise it crashes)
     *
     * Only really used when level loading
     */
    public void destorySkeletonJoint( ) {
        if ( extraSkeletonJoint != null ) {
            Level.jointsToRemove.add( extraSkeletonJoint );
            extraSkeletonJoint = null;
        }
    }

}
```