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

import java.util.ArrayList;
import java.util.HashMap;

import com.badlogic.gdx.graphics.Texture;
// [omitted]
import com.blindtigergames.werescrewed.util.Util;

/**
 * A Skeleton is a node in the level tree structure. It moves platforms under it
 * as well as skeletons attached.
 *
 * @author Stewart
 *
 *         TODO: Perhaps change skeleton name, and make skeleton more like a
 *         tree (i.e. It should have a list of non-jointed entities too.)
 */

public class Skeleton extends Platform {

    // public static final int foreground = 0;
    // public static final int background = 1;
    // public static final int midground = 2;

    public PolySprite bgSprite, fgSprite;

    SimpleFrameAnimator alphaFadeAnimator;
    private final float fadeSpeed = 3f;

    protected HashMap< String, Platform > dynamicPlatformMap = new HashMap< String,
Platform >( );
    protected HashMap< String, Skeleton > childSkeletonMap = new HashMap< String,
Skeleton >( );
    protected HashMap< String, Platform > kinematicPlatformMap = new HashMap<
String, Platform >( );
    protected HashMap< String, Rope > ropeMap = new HashMap< String, Rope >( );
    protected HashMap< String, Screw > screwMap = new HashMap< String, Screw >( );
    protected HashMap< String, CheckPoint > checkpointMap = new HashMap< String,
CheckPoint >( );
    protected HashMap< String, EventTrigger > eventMap = new HashMap< String,
EventTrigger >( );
    private ArrayList< Entity > entitiesToRemove = new ArrayList< Entity >( );

    private int entityCount = 0;
```

```java
    protected RootSkeleton rootSkeleton;
    protected Skeleton parentSkeleton;

    protected boolean applyFadeToFGDecals = true;
    protected boolean isMacroSkeleton = false;
    protected boolean invisibleBGDecal = false;

    protected boolean wasInactive = false;
    protected boolean onScreen = true;
    protected boolean isUpdatable = true;

    protected boolean setChildSkeletonsToSleep = false;
    protected boolean useBoundingRect = false;
    protected boolean updatedOnce = false;
    public Rectangle boundingRect = new Rectangle( -10000, -10000, 10000, 10000 );
    protected Rectangle lastCameraRect = new Rectangle( 0, 0, 0, 0 );
    protected boolean removed = false;

    public boolean respawningDontPutToSleep = false;

    private final float MAX_FALL_POS = -5000.0f;

    // private ShapeRenderer shapeRender;

    /**
     * Constructor used by SkeletonBuilder
     *
     * @param n
     * @param pos
     * @param tex
     * @param world
     * @param bodyType
     */
    public Skeleton( String n, Vector2 pos, Texture tex, World world,
            BodyType bodyType ) {
        super( n, pos, tex, world ); // not constructing body class
        this.world = world;
        constructSkeleton( pos, bodyType );
        super.setSolid( false );
        entityType = EntityType.SKELETON;
        alphaFadeAnimator = new SimpleFrameAnimator( ).speed( 0 )
                .loop( LoopBehavior.STOP ).time( 1 );
        // shapeRender = new ShapeRenderer( );
    }
    /**
     * COnstructor to default to kinematic body type
     *
     * @param n
     * @param pos
     * @param tex
     * @param world
     */
    public Skeleton( String n, Vector2 pos, Texture tex, World world ) {
        this( n, pos, tex, world, BodyType.KinematicBody );
    }

    public void constructSkeleton( Vector2 pos, BodyType bodyType ) {
        // Skeletons have no fixtures!!
        BodyDef skeletonBodyDef = new BodyDef( );
        skeletonBodyDef.type = bodyType;

        skeletonBodyDef.position.set( pos.cpy( ).mul( Util.PIXEL_TO_BOX ) );
        body = world.createBody( skeletonBodyDef );
        body.setUserData( this );

        FixtureDef dynFixtureDef = new FixtureDef( );
        PolygonShape polygon = new PolygonShape( );
        polygon.setAsBox( 100 * Util.PIXEL_TO_BOX, 100 * Util.PIXEL_TO_BOX );
        dynFixtureDef.shape = polygon;
        dynFixtureDef.density = 5f;
        dynFixtureDef.isSensor = true;
        dynFixtureDef.filter.categoryBits = Util.CATEGORY_SKELS;
        dynFixtureDef.filter.maskBits = Util.CATEGORY_SCREWS;
        body.createFixture( dynFixtureDef );
        polygon.dispose( );
        body.setGravityScale( 0.1f );
        // this.quickfixCollisions( );
    }

    /**
     * Attach a platform to this skeleton that will freely rotate about the
     * center. Make sure the platform is dynamic
     *
     * @param platform
     */
    public void addPlatformRotatingCenter( Platform platform ) {
        // Default values of the builder will allow rotation with anchor at
        // center of platform
```

```
        new RevoluteJointBuilder( world ).entityA( this ).entityB( platform )
                .build( );
        addDynamicPlatform( platform );
    }

    /**
     * Attach a platform to this skeleton that rotates with a motor the platform
     * must already be set as dynamic
     *
     * @param platform
     */
    public void addPlatformRotatingCenterWithMot( Platform platform,
            float rotSpeedInMeters ) {
        // Default values of the builder will allow rotation with anchor at
        // center of platform
        new RevoluteJointBuilder( world ).entityA( this ).entityB( platform )
                .motor( true ).motorSpeed( rotSpeedInMeters ).build( );

        addDynamicPlatform( platform );
    }

    /**
     * Add a platform that will only move / rotate with skeleton Don't use this.
     * if it's fixed, you might as well add it as kinematic
     *
     * @param platform
     */
    public void addDynamicPlatformFixed( Platform platform ) {
        new RevoluteJointBuilder( world ).entityA( this ).entityB( platform )
                .limit( true ).lower( 0 ).upper( 0 ).build( );
        addDynamicPlatform( platform );
    }

    /**
     * Add a platform to this skeleton. Will determine what list to add it to
     * for you!
     *
     * @param platform
     */
    public void addPlatform( Platform platform ) {
        if ( platform.body.getType( ) == BodyType.DynamicBody )
            addDynamicPlatform( platform );
        else
            addKinematicPlatform( platform );
    }
```

```java
	public void addPlatforms( Platform... platforms ) {
	    for ( Platform p : platforms ) {
	        addPlatform( p );
	    }
	}

	public void addRope( Rope rope, boolean toJoint ) {
	    if ( toJoint ) {
	        new RevoluteJointBuilder( world ).entityA( this )
	                .entityB( rope.getFirstLink( ) ).limit( true ).lower( 0 )
	                .upper( 0 ).build( );
	    }
	    // ropes.add( rope );
	    ropeMap.put( rope.name, rope );
	}

	public boolean isMacroSkel( ) {
	    return isMacroSkeleton;
	}

	public void setMacroSkel( boolean macroSkel ) {
	    isMacroSkeleton = macroSkel;
	}

	/**
	 * 
	 * @param ss
	 *            - add stripped screw onto the skeleton
	 */
	public void addStrippedScrew( StrippedScrew ss ) {
	    addScrewForDraw( ss );
	}

	/**
	 * Add a screw to be drawn!
	 * 
	 * @param Screw
	 */
	public void addScrewForDraw( Screw s ) {
	    // screws.add(s);
	    entityCount++;
	    screwMap.put( s.name, s );
	    s.setParentSkeleton( this );
	}
```

```java
	/**
	 * add checkpoint to be drawn
	 */
	public void addCheckPoint( CheckPoint chkpt ) {
	    entityCount++;
	    checkpointMap.put( chkpt.name, chkpt );
	    chkpt.setParentSkeleton( this );
	}

	/**
	 * Simply adds a platform to the list, without explicitly attaching it to
	 * the skelington
	 * 
	 * @param Entity
	 *            platform
	 * @author stew
	 */
	public void addDynamicPlatform( Platform platform ) {
	    entityCount++;
	    // this.dynamicPlatforms.add( platform );
	    if ( dynamicPlatformMap.containsKey( platform.name ) ) {
	        platform.name = getUniqueName( platform.name );
	    }
	    dynamicPlatformMap.put( platform.name, platform );
	    platform.setParentSkeleton( this );
	    platform.setOriginRelativeToSkeleton( platform.getPosition( ).cpy( )
	            .sub( getPosition( ) ) );
	}

	/**
	 * Add Kinamatic platform to this Skeleton
	 * 
	 * @param Platform
	 *            that's already set as kinematic
	 */
	public void addKinematicPlatform( Platform platform ) {
	    // kinematicPlatforms.add( platform );
	    entityCount++;
	    if ( kinematicPlatformMap.containsKey( platform.name ) ) {
	        platform.name = getUniqueName( platform.name );
	    }
	    kinematicPlatformMap.put( platform.name, platform );
	    platform.setParentSkeleton( this );
	    platform.setOriginRelativeToSkeleton( platform.getPosition( ).cpy( )
                .sub( ( getPosition( ) ) ) );
    }

    public void addSteam( Steam steam ) {
        addKinematicPlatform( steam );
    }

    /**
     * Add EventTrigger to this Skeleton
     * 
     * @param event
     *            EventTrigger to be added to Skeleton
     */
    public void addEventTrigger( EventTrigger event ) {
        entityCount++;
        if ( eventMap.containsKey( event.name ) ) {
            event.name = getUniqueName( event.name );
        }
        event.setParentSkeleton( this );
        event.setOriginRelativeToSkeleton( event.getPosition( ).cpy( )
                .sub( ( getPosition( ) ) ) );
        eventMap.put( event.name, event );
    }

    public void addHazard( Hazard h ) {
        addPlatform( h );
    }

    /**
     * Add a skeleton to the sub skeleton list of this one.
     * 
     * @author stew
     */
    public void addSkeleton( Skeleton skeleton ) {
        // this.childSkeletons.add( skeleton );
        if ( this == rootSkeleton ) {
            skeleton.setMacroSkel( true );
        }
        skeleton.parentSkeleton = this;
        skeleton.rootSkeleton = this.rootSkeleton;
        childSkeletonMap.put( skeleton.name, skeleton );
        skeleton.setParentSkeleton( this );
        skeleton.setOriginRelativeToSkeleton( skeleton.getPosition( ).cpy( )
                .sub( ( getPosition( ) ) ) );
    }
```

```java

    /**
     * set skeleton to awake or not TODO: Do kinamtic platforms need sleeping?
     */
    public void setSkeletonAwakeRec( boolean isAwake ) {
        for ( Skeleton skeleton : childSkeletonMap.values( ) ) {
            skeleton.setSkeletonAwakeRec( isAwake );
        }
        for ( Platform platform : dynamicPlatformMap.values( ) ) {
            platform.body.setAwake( isAwake );
        }
        for ( Platform platform : kinematicPlatformMap.values( ) ) {
            platform.body.setAwake( isAwake );
        }
        for ( Screw screw : screwMap.values( ) ) {
            screw.body.setAwake( isAwake );
        }
        for ( CheckPoint chkpt : checkpointMap.values( ) ) {
            chkpt.body.setAwake( isAwake );
        }
    }

    /**
     * finds the skeleton with this name
     */
    public Skeleton getSubSkeletonByName( String name ) {
        if ( childSkeletonMap.containsKey( name ) ) {
            return childSkeletonMap.get( name );
        }
        return null;
    }

    public void setSkeletonEntitiesToSleepRecursively( ) {
        this.setEntitiesToSleepOnUpdate( );
        this.wasInactive = true;
        for ( Skeleton skeleton : this.childSkeletonMap.values( ) ) {
            if ( !skeleton.dontPutToSleep ) {
                if ( this.useBoundingRect ) {
                    if ( inRectangleBounds( this.boundingRect,
                            skeleton.getPositionPixel( ) ) ) {
                        skeleton.setSkeletonEntitiesToSleepRecursively( );
                        skeleton.body.setActive( true );
                        skeleton.body.setAwake( false );
                    } else {
                        skeleton.dontPutToSleep = true;
                        }
                    } else {
                        skeleton.setSkeletonEntitiesToSleepRecursively( );
                        skeleton.body.setActive( true );
                        skeleton.body.setAwake( false );
                    }
                }
            }
        }

        public boolean inRectangleBounds( Rectangle rect, Vector2 point ) {
            if ( point.x > rect.x && point.x < rect.x + rect.width
                    && point.y > rect.y && point.y < rect.y + rect.height ) {
                return true;
            }
            return false;
        }

        public boolean isRemoved( ) {
            return removed;
        }

        /**
         * This update function is ONLY called on the very root skeleton, it takes
         * care of the child sksletons
         *
         * @author stew
         */
        @Override
        public void update( float deltaTime ) {
            if ( this.getPositionPixel( ).y < MAX_FALL_POS && !this.removed ) {
                this.remove( );
            } else {
                if ( !removed ) {
                    if ( !this.removeNextStep ) {
                        super.update( deltaTime );
                        float frameRate = 1 / deltaTime;
                        isUpdatable = ( !this.isFadingSkel( ) || this.isFGFaded( ) )
                                || this.dontPutToSleep;
                        if ( useBoundingRect && updatedOnce ) {
                            boundingRect.x = this.getPositionPixel( ).x
                                    - ( boundingRect.width / 2.0f );
                            boundingRect.y = this.getPositionPixel( ).y
                                    - ( boundingRect.height / 2.0f );
                            if ( !boundingRect.overlaps( lastCameraRect ) ) {
```

```java
                                isUpdatable = false;
                                if ( !wasInactive ) {
                                    wasInactive = true;
                                    setSkeletonEntitiesToSleepRecursively( );
                                }
                            } else {
                                isUpdatable = true;
                            }
                        } else if ( !useBoundingRect && !isUpdatable
                                && this.setChildSkeletonsToSleep && !wasInactive ) {
                            setSkeletonEntitiesToSleepRecursively( );
                        }
                        updatedOnce = true;
                        if ( isUpdatable || isMacroSkeleton ) {
                            updateMover( deltaTime );
                            if ( entityType != EntityType.ROOTSKELETON
                                    && isKinematic( ) ) {
                                super.setTargetPosRotFromSkeleton( frameRate,
                                        parentSkeleton );
                            }
                        }
                        for ( EventTrigger event : eventMap.values( ) ) {
                            event.translatePosRotFromSKeleton( this );
                            // event.setTargetPosRotFromSkeleton( frameRate, this );
                        }

                        if ( isUpdatable ) {
                            for ( Rope rope : ropeMap.values( ) ) {
                                // TODO: ropes need to be able to be deleted
                                if ( wasInactive ) {
                                    boolean nextLink = true;
                                    int index = 0;
                                    if ( rope.getEndAttachment( ) != null ) {
                                        if ( !rope.getEndAttachment( ).body
                                                .isActive( ) ) {
                                            rope.getEndAttachment( ).body
                                                    .setActive( true );
                                        }
                                        // if ( rope.getEndAttachment(
                                        // ).body.isAwake( ) ) {
                                        // rope.getEndAttachment( ).body.setAwake(
                                        // false );
                                        // }
                                    }
                                    while ( nextLink ) {
```

```
                if ( !rope.getLink( index ).body.isActive( ) ) {
                    rope.getLink( index ).body
                            .setActive( true );
                }
                // if ( rope.getLink( index ).body.isAwake(
                // ) ) {
                // rope.getLink( index ).body.setAwake(
                // false );
                // }
                if ( rope.getLastLink( ) == rope
                        .getLink( index ) ) {
                    nextLink = false;
                }
                index++;
            }
        }
        rope.update( deltaTime );
    }
    for ( Platform platform : kinematicPlatformMap.values( ) ) {
        if ( platform.removeNextStep ) {
            entitiesToRemove.add( platform );
        } else {
            if ( wasInactive ) {
                if ( !platform.body.isActive( ) ) {
                    platform.body.setActive( true );
                }
                if ( platform.body.isAwake( ) ) {
                    platform.body.setAwake( false );
                }
                platform.translatePosRotFromSKeleton( this );
                platform.update( deltaTime );
            } else {
                platform.updateMover( deltaTime );
                if ( !platform.body.isActive( ) ) {
                    platform.body.setActive( true );
                }
                if ( platform.body.isAwake( ) ) {
                    platform.body.setAwake( false );
                }
                if ( platform.hasMoved( )
                        || platform.hasRotated( )
                        || hasMoved( ) || hasRotated( ) ) {
                    platform.setTargetPosRotFromSkeleton(
                            frameRate, this );
                    platform.setPreviousTransformation( );
```

```
                } else {
                    platform.body
                            .setLinearVelocity( Vector2.Zero );
                    platform.body.setAngularVelocity( 0.0f );
                }
                platform.update( deltaTime );
            }
        }
    }
    for ( Platform platform : dynamicPlatformMap.values( ) ) {
        if ( platform.removeNextStep ) {
            entitiesToRemove.add( platform );
        } else {
            if ( wasInactive ) {
                if ( !platform.body.isActive( ) ) {
                    platform.body.setActive( true );
                }
                if ( platform.body.isAwake( ) ) {
                    platform.body.setAwake( false );
                }
            }
            platform.updateMover( deltaTime );
            platform.update( deltaTime );
        }
    }
    for ( CheckPoint chkpt : checkpointMap.values( ) ) {
        if ( chkpt.removeNextStep ) {
            entitiesToRemove.add( chkpt );
        } else {
            if ( wasInactive ) {
                if ( !chkpt.body.isActive( ) ) {
                    chkpt.body.setActive( true );
                }
                if ( chkpt.body.isAwake( ) ) {
                    chkpt.body.setAwake( false );
                }
            }
            chkpt.update( deltaTime );
        }
    }
    for ( Screw screw : screwMap.values( ) ) {
        if ( screw.removeNextStep ) {
            entitiesToRemove.add( screw );
        } else {
            if ( wasInactive ) {
                    if ( !screw.body.isActive( ) ) {
                        screw.body.setActive( true );
                    }
                    if ( screw.body.isAwake( ) ) {
                        screw.body.setAwake( false );
                    }
                }
                screw.update( deltaTime );
            }
        }
        if ( wasInactive ) {
            if ( !body.isActive( ) ) {
                body.setActive( true );
            }
            if ( body.isAwake( ) ) {
                body.setAwake( false );
            }
            for ( Skeleton skeleton : childSkeletonMap.values( ) ) {
                if ( !skeleton.body.isActive( ) ) {
                    skeleton.body.setActive( true );
                }
                if ( skeleton.body.isAwake( ) ) {
                    skeleton.body.setAwake( false );
                }
            }
            wasInactive = false;
        }
    } else {
        if ( !wasInactive ) {
            setEntitiesToSleepOnUpdate( );
            wasInactive = true;
        }
    }

    setPreviousTransformation( );

    alphaFadeAnimator.update( deltaTime );
    Vector2 pixelPos = null;
    if ( fgSprite != null ) {
        pixelPos = getPosition( ).mul( Util.BOX_TO_PIXEL );
        fgSprite.setPosition( pixelPos.x - offset.x, pixelPos.y
                - offset.y );
        fgSprite.setRotation( MathUtils.radiansToDegrees
                * getAngle( ) );
    }
```

```
    if ( bgSprite != null ) {
        if ( pixelPos == null )
            pixelPos = getPosition( ).mul( Util.BOX_TO_PIXEL );
        bgSprite.setPosition( pixelPos.x - offset.x, pixelPos.y
                - offset.y );
        bgSprite.setRotation( MathUtils.radiansToDegrees
                * getAngle( ) );
    }
    updateDecals( deltaTime );

    // }
    // recursively update child skeletons

    for ( Skeleton skeleton : childSkeletonMap.values( ) ) {
        if ( skeleton.removeNextStep ) {
            entitiesToRemove.add( skeleton );
        } else {
            if ( !setChildSkeletonsToSleep || isUpdatable
                    || skeleton.dontPutToSleep ) {
                skeleton.update( deltaTime );
            }
        }
    }

    // remove stuff
    if ( entitiesToRemove.size( ) > 0 ) {

        for ( Entity e : entitiesToRemove ) {

            switch ( e.entityType ) {
            case SKELETON:
                Skeleton s = childSkeletonMap.remove( e.name );
                s.remove( );
                break;
            case PLATFORM:
                Platform p;
                if ( e.isKinematic( ) ) {
                    p = kinematicPlatformMap.remove( e.name );
                } else {
                    p = dynamicPlatformMap.remove( e.name );
                }
                p.remove( );
                break;
            case SCREW:
                Screw sc = screwMap.remove( e.name );
```

```
                                    sc.remove( );
                                    break;
                                case CHECKPOINT:
                                    CheckPoint chkpt = checkpointMap
                                            .remove( e.name );
                                    chkpt.setNextCheckPointInPM( );
                                    chkpt.remove( );
                                    break;
                                default:
                                    throw new RuntimeException(
                                            "You are trying to remove enity '"
                                                    + e.name
                                                    + "' but skeleton '"
                                                    + this.name
                                                    + "' can't determine it's type.
    This may be my fault for not adding a case. -stew" );
                                }
                            }
                            entitiesToRemove.clear( );
                        }
                    }
                }
            }
        }

        /**
         * removes the bodies and joints of all the skeletons children
         */
        @Override
        public void remove( ) {
            for ( Skeleton skeleton : childSkeletonMap.values( ) ) {
                skeleton.remove( );
            }
            childSkeletonMap.clear( );
            for ( Platform p : dynamicPlatformMap.values( ) ) {
                p.remove( );
            }
            dynamicPlatformMap.clear( );
            for ( Platform p : kinematicPlatformMap.values( ) ) {
                p.remove( );
            }
            kinematicPlatformMap.clear( );
            for ( Screw screw : screwMap.values( ) ) {
                screw.remove( );
            }
```

```
            screwMap.clear( );
            for ( CheckPoint chkpt : checkpointMap.values( ) ) {
                chkpt.setNextCheckPointInPM( );
                chkpt.remove( );
            }
            checkpointMap.clear( );
            for ( EventTrigger event : eventMap.values( ) ) {
                event.remove( );
            }
            eventMap.clear( );
            // for ( Rope rope : ropeMap.values( ) ) {
            // boolean nextLink = true;
            // int index = 0;
            // if ( rope.getEndAttachment( ) != null ) {
            // while ( rope.getEndAttachment( ).body.getJointList( ).iterator(
            // ).hasNext( ) ) {
            // world.destroyJoint( body.getJointList( ).get( 0 ).joint );
            // }
            // world.destroyBody( rope.getEndAttachment( ).body );
            // }
            // while ( nextLink ) {
            // world.destroyBody( rope.getLink( index ).body );
            // if ( rope.getLastLink( ) == rope.getLink( index ) ) {
            // nextLink = false;
            // }
            // index++;
            // }
            // }
            // while ( body.getJointList( ).iterator( ).hasNext( ) ) {
            // world.destroyJoint( body.getJointList( ).get( 0 ).joint );
            // }
            body.setActive( false );
            body.setAwake( true );
            // world.destroyBody( body );
            // this.fgDecals.clear( );
            // this.bgDecals.clear( );
            // this.bgSprite = null;
            // this.fgSprite = null;
            this.removed = true;
        }

        /**
         * this skeleton has gone to bed, put its entities to sleep instead of
         * updating the entities movements and such and delete them if necessary
         */
```

```
    private void setEntitiesToSleepOnUpdate( ) {
        if ( !this.removeNextStep ) {
            for ( Platform platform : kinematicPlatformMap.values( ) ) {
                if ( platform.removeNextStep ) {
                    entitiesToRemove.add( platform );
                } else if ( !platform.dontPutToSleep ) {
                    platform.body.setAwake( true );
                    platform.body.setActive( false );
                }
            }
            for ( Platform platform : dynamicPlatformMap.values( ) ) {
                if ( platform.removeNextStep ) {
                    entitiesToRemove.add( platform );
                } else {
                    platform.body.setAwake( true );
                    platform.body.setActive( false );
                }
            }
            for ( CheckPoint chkpt : checkpointMap.values( ) ) {
                if ( chkpt.removeNextStep ) {
                    entitiesToRemove.add( chkpt );
                } else {
                    chkpt.body.setActive( true );
                    chkpt.body.setAwake( false );
                }
            }
            for ( Screw screw : screwMap.values( ) ) {
                if ( screw.removeNextStep ) {
                    entitiesToRemove.add( screw );
                } else if ( !screw.dontPutToSleep ) {
                    if ( this.useBoundingRect ) {
                        if ( inRectangleBounds( this.boundingRect,
                                screw.getPositionPixel( ) ) ) {
                            if ( screw.getDepth( ) >= 0 ) {
                                screw.body.setAwake( true );
                                screw.body.setActive( false );
                            } else {
                                screw.dontPutToSleep = true;
                            }
                        } else {
                            screw.dontPutToSleep = true;
                        }
                    } else {
                        screw.body.setAwake( true );
                        screw.body.setActive( false );
```

```
                    }
                }
            }
            for ( Rope rope : ropeMap.values( ) ) {
                // TODO: ropes need to be able to be deleted
                boolean nextLink = true;
                int index = 0;
                if ( rope.getEndAttachment( ) != null ) {
                    // rope.getEndAttachment( ).body.setAwake( true );
                    rope.getEndAttachment( ).body.setActive( false );
                }
                while ( nextLink ) {
                    // rope.getLink( index ).body.setAwake( true );
                    rope.getLink( index ).body.setActive( false );
                    if ( rope.getLastLink( ) == rope.getLink( index ) ) {
                        nextLink = false;
                    }
                    index++;
                }
            }
        }
    }

    /**
     *
     * @param batch
     * @param camera
     */
    @Override
    public void drawFGDecals( SpriteBatch batch, Camera camera ) {
        if ( !removed && !removeNextStep ) {
            for ( Sprite decal : fgDecals ) {
                if ( decal.alpha >= 0.25 ) {
                    if ( decal.getBoundingRectangle( ).overlaps(
                            camera.getBounds( ) ) ) {
                        decal.draw( batch );
                    }
                }
            }
        }
    }

    @Override
    public void draw( SpriteBatch batch, float deltaTime, Camera camera ) {
        if ( !removed && !removeNextStep ) {
            // if ( this.useBoundingRect ) {
            // shapeRender.setProjectionMatrix( camera.combined( ) );
            // shapeRender.begin( ShapeType.Rectangle );
            // shapeRender.rect( boundingRect.x, boundingRect.y,
            // boundingRect.width,
            // boundingRect.height );
            // shapeRender.end( );
            // }
            super.draw( batch, deltaTime, camera );
            if ( visible ) {
                drawChildren( batch, deltaTime, camera );
                if ( fgSprite != null && alphaFadeAnimator.getTime( ) > 0 ) {
                    fgSprite.setAlpha( alphaFadeAnimator.getTime( ) );
                    // batch.setColor( c.r, c.g, c.b, fgAlphaAnimator.getTime( )
                    // );
                    // fgSprite.draw( batch );
                    // batch.setColor( c.r, c.g, c.b, oldAlpha );
                }
                if ( applyFadeToFGDecals ) {
                    if ( name.equals( "head_skeleton" ) )
                        getAngle( );
                    fadeFGDecals( );
                }
            }
        }
    }

    private void drawChildren( SpriteBatch batch, float deltaTime, Camera camera ) {
        if ( !removed && !removeNextStep ) {
            lastCameraRect = camera.getBounds( );
            if ( !wasInactive && isUpdatable ) {
                for ( EventTrigger et : eventMap.values( ) ) {
                    et.draw( batch, deltaTime, camera );
                }
                for ( Screw screw : screwMap.values( ) ) {
                    if ( !screw.getRemoveNextStep( ) ) {
                        screw.draw( batch, deltaTime, camera );
                    }
                }
                for ( Platform p : dynamicPlatformMap.values( ) ) {
                    drawPlatform( p, batch, deltaTime, camera );
                }
                for ( Platform p : kinematicPlatformMap.values( ) ) {
                    drawPlatform( p, batch, deltaTime, camera );
                }
```

```
                for ( CheckPoint chkpt : checkpointMap.values( ) ) {
                    if ( !chkpt.getRemoveNextStep( ) ) {
                        chkpt.draw( batch, deltaTime, camera );
                    }
                }
                for ( Rope rope : ropeMap.values( ) ) {
                    rope.draw( batch, deltaTime, camera );
                }
            }
            // draw the entities of the parent skeleton before recursing through
            // the
            // child skeletons
            // if ( isUpdatable || isMacroSkeleton )
            {
                for ( Skeleton skeleton : childSkeletonMap.values( ) ) {
                    if ( !setChildSkeletonsToSleep || isUpdatable
                            || skeleton.dontPutToSleep ) {
                        skeleton.draw( batch, deltaTime, camera );
                    }
                }
            }
        }
    }

    /**
     *
     * @param batch
     * @param camera
     */
    @Override
    public void drawBGDecals( SpriteBatch batch, Camera camera ) {
        if ( !removed && !removeNextStep ) {
            for ( Sprite decal : bgDecals ) {
                if ( decal.getBoundingRectangle( )
                        .overlaps( camera.getBounds( ) ) ) {
                    if ( !invisibleBGDecal ) {
                        decal.draw( batch );
                    }
                }
            }
        }
    }

    /**
     * Draw each child. Tiled platforms have unique draw calls. Platforms can be
```

```java
         * hazards as well
         */
        private void drawPlatform( Platform platform, SpriteBatch batch,
                float deltaTime, Camera camera ) {
            platform.draw( batch, deltaTime, camera );
        }

        public boolean getWasInactive( ) {
            return wasInactive;
        }

        public void setUseBoundingRect( boolean setting ) {
            useBoundingRect = setting;
        }

        public boolean getIsUsingBoundingBox( ) {
            return useBoundingRect;
        }

        public boolean isUpdatable( ) {
            return isUpdatable;
        }

        private String getUniqueName( String nonUniqueName ) {
            return nonUniqueName + "-NON-UNIQUE-NAME_" + entityCount;
        }

        /**
         * Delete a child skeleton by name. Recursively tries to find the child
         * skele.
         *
         * @param skeleName
         *            searches all skeletons under this skeleton
         */
        public void deleteSkeletonByName( String skeleName ) {
            for ( Skeleton s : childSkeletonMap.values( ) ) {
                if ( s.name.equals( skeleName ) ) {
                    rootSkeleton.destroySkeleton( s );
                    break;
                } else {
                    s.deleteSkeletonByName( skeleName );
                }
            }
        }
```

```java
        /**
         * Deletes this skeleton, Potentially creates null pointers, please don't
         * directly call this, instead add your skeleton-to-be-deleted to root using
         * RootSkeleton.deleteSkeleton(Skeleton)
         */
        @Override
        public void dispose( ) {
            for ( Platform platform : dynamicPlatformMap.values( ) ) {
                platform.body.getWorld( ).destroyBody( platform.body );
            }
            dynamicPlatformMap.clear( );
            for ( Platform platform : kinematicPlatformMap.values( ) ) {
                platform.body.getWorld( ).destroyBody( platform.body );
            }
            kinematicPlatformMap.clear( );
            for ( Rope rope : ropeMap.values( ) ) {
                rope.dispose( );
            }
            ropeMap.clear( );
            for ( Screw screw : screwMap.values( ) ) {
                screw.dispose( );
            }
            for ( CheckPoint chkpt : checkpointMap.values( ) ) {
                chkpt.dispose( );
            }
            screwMap.clear( );
            for ( EventTrigger et : eventMap.values( ) ) {
                et.dispose( );
            }
            eventMap.clear( );
            for ( CheckPoint chkpt : checkpointMap.values( ) ) {
                chkpt.dispose( );
            }
            checkpointMap.clear( );
            super.dispose( );
        }

        /**
         * Generally for debug purposes
         *
         * @param angleInRadians
         */
        public void rotateBy( float angleInRadians ) {
            setLocalRot( getLocalRot( ) + angleInRadians );
        }
```

```
    public void setChildSkeletonsToSleepProperty( boolean setting ) {
        setChildSkeletonsToSleep = setting;
    }

    /**
     * For debugging
     *
     * @param xPixel
     * @param yPixel
     */
    public void translateBy( float xPixel, float yPixel ) {
        setLocalPos( getLocalPos( ).add( xPixel, yPixel ) );
    }

    /**
     * A less recursive get root function!
     *
     * @return Root skeleton of this skeleton
     */
    public RootSkeleton getRoot( ) {
        return rootSkeleton;
    }

    /**
     *
     * @param hasTransparency
     *            true if you want to see into the robot
     */
    public void setFade( boolean hasTransparency ) {
        float speed = fadeSpeed;
        // if ( !hasTransparency ){
        // Gdx.app.log("stageSkeleton","NO TRANSPARENCY");
        // }
        if ( hasTransparency ) {
            speed = -fadeSpeed;
        }
        /*
         * else{ if(name.equals("stageSkeleton")){
         *
         * //speed = fadeSpeed; } } if(name.equals("stageSkeleton"))
         * Gdx.app.log(
         * "stageSkeleton","Speed: "+speed+" Time:"+alphaFadeAnimator.getTime(
         * ));
         */
```

```
        alphaFadeAnimator.speed( speed );
    }

    private void fadeFGDecals( ) {
        float alpha = alphaFadeAnimator.getTime( );
        alpha *= alpha;
        for ( Sprite decal : fgDecals ) {
            if ( decal.getAlpha( ) != alpha ) {
                decal.setAlpha( alpha );
            }
        }
    }

    public void setFgFade( boolean applyFadeToFGDecals ) {
        this.applyFadeToFGDecals = applyFadeToFGDecals;
    }

    public boolean isFGFaded( ) {
        return alphaFadeAnimator.getTime( ) < 1;
    }

    public boolean isFadingSkel( ) {
        return applyFadeToFGDecals;
    }

    public EventTrigger getEvent( String eventName ) {
        return eventMap.get( eventName );
    }
}
```