Sprite-Lib - eine Funktionsbibliothek für die Nutzung von Sprites in Processing

Kai Hinkelmann, kai.hinkelmann@fh-kiel.de

3. April 2023

1 Einleitung

Die Sprite-Lib unterstützt Euch bei der Erstellung einer Processing-Anwendung, die Sprites nutzt. Die folgenden Sprite-Typen werden unterstützt:

- `SingleSprite` - das Single-Sprite ist ein einfaches Sprite das ”nur” genau ein Image verwaltet.
- `MultiSprite` - Sprites, die mehrere Animationsframes in einer Instanz vereinen
- `AnimatedSprite` - Sprites, die mehrere Animationsframes automatisch animieren
- `SequencedSprite` - Sprites, die für die enthaltenen Animationsframes unterschiedliche Abläufe in Framenummer-Sequenzen enthalten.

2 „Hilfsklassen” und Enums

Das Package spritelib enthält einige Hilfsklassen und Enums. Im einzelnen sind das:

- `enum ANCHORTYPE`

  Bei der Darstellung eines Sprites wird eine Position auf dem Bildschirm übergeben. Ein Sprite hat ein umgebendes Rechteck, dessen Größe beim Konstruktor des Sprites mit übergeben wird. Die Lage der Darstellungsposition zum umgebenden Rechteck wird durch einen Anchor bzw. ANCHORTYPE festgelegt.

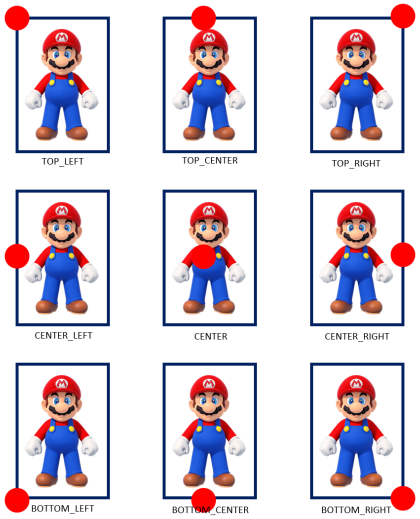


Abb. 1: ANCHORTYPEs

- `enum EVENTTYPE`

- EventTypes sind:

  - `ENTERFRAME` wird ausgelöst wenn ein Frame „betreten" wird.
  - `LEAVEFRAME` wird ausgelöst wenn ein Frame „verlassen" wird.

  Die Klasse `MultiSprite` und alle von ihr abgeleiteten Klassen lösen bei Framewechseln Events aus, die durch Setzen des Attributes `onSpriteEvent` gefangen werden können. Parameter eines Events ist ein `SpriteEvent` mit einem Attribute `eventtype` vom Typ `EVENTTYPE`.

- `Point`
  Die Klasse Point codiert ein Pixel auf dem Bildschirm mit den Koordinaten `x` und `y`. Beide sind private mit entsprechenden Gettern und Settern.

- `Size`
  Die Klasse `Size` codiert die Größe eines rechteckigen Objektes mit den Attributen `width` and `height`. Beide sind private mit entsprechenden Gettern un Settern.

- `Rectangle`
  Die Klasse `Rectangle` codiert ein Rechteck. Die Klasse setzt sich zusammen aus einem Attribut `Point topleft` und `Size size` mit Ihren entsprechenden Bedeutungen.

- `Sequence`
  Eine `Sequence` beschreibt eine Abfolge von Frames, die automatisch „abgespielt" werden. Diese Klasse wird von der Klasse `SequencedSprite` genutzt. Die Klasse `Sequence` enthält drei Attribute und zehn Methoden mit größtenteils offensichtlicher Funktionalität:

  - `private ArrayList<Integer> frames`, eine Liste vom Bitmap-Indexen.

> Zu beachten ist, dass eine Sequence nicht die Bitmaps selbst sondern nur die Indexe in der Bitmap-Liste der Superklasse `AnimSprite` enthält.

  - `private String name`, die Bezeichnung der Sequenz
  - `private String nextSequenceName`, die Bezeichnung der „Folgesequenz", also der `Sequence`, mit der die Animation fortgesetzt wird, wenn der letzte Frame erreicht ist. Hierbei kann es sich auch um die selbe Sequenz handeln. In diesem Fall entspricht das Verhalten dem des `AnimSprite`. Allerdings kann die nutzende Klasse die aktive Sequenz jederzeit per Methodenaufruf ändern.
  - `public Sequence( String name )`, Konstruktor mit dem Namen der Sequenz.
  - `public Sequence( String name, String nextSequenceName )`, Konstruktor mit dem Namen der Sequenz und dem Namen der Folgesequenz.

- `public Sequence( String name, int... elems )`, Konstruktor mit dem Namen der Sequenz und einer Folge von Bitmap-Indexen. Der Aufruf ist z.B.
  `Sequence s = new Sequence(laufen",1,2,3,2)`
- `public Sequence( String name, String nextSequenceName, int... elems )`, analog zu den beiden genannten Konstruktoren
- `public String getName()`, Getter des Namens
- `public String getNextSequenceName()`, Getter des Folgesequenz-Namens
- `public void addFrame( int frame )`, ergänzt einen einzelnen Index zu der Liste der Indexe
- `public void addRange( int from, int to )`, ergänzt alle Indexe inkl. from und to zu der Liste der Indexe
- `public int getLength()`, Getter für die Anzahl der Indexe
- public int getFrame(int i), Getter für den i.ten Index in der Liste der Indexe

3 Klassen

3.1 Sprite

Die Klasse `Sprite` ist die Basisklasse für alle folgenden Spriteklassen. Die Klasse `Sprite` selbst ist abstrakt und kann nicht instantiiert werden. Relevante Attribute und Methoden sind:

- `protected Sprite( int width, int height )`, Konstruktor der Klasse `Sprite`. Die Parameter geben die Breite und die Höhe des Sprites in Pixeln an. Bei diesem Konstruktor wird der Ankerpunkt auf `center` initialisiert.
- `protected Sprite( int width, int height, ANCHORTYPE anchor )`, Konstruktor der Klasse `Sprite`. Funktion wie oben, nur dass der Ankerpunkt mit dem Wert des dritten Parameters vorbelegt wird.

> *Beide Konstruktoren sind `protected`, können also nur von einer abgeleiteten Klasse aufgerufen werden. Da die Klasse abstrakt ist, ist das aber keine Einschränkung.*

- `public Rectangle getPlotRectangle()`, gibt das Rechteck zurück, in dem der Sprite zuletzt dargestellt wurde.
- `public Rectangle getPlotRect( int origX, int origY )`, gibt das Rechteck zurück, in dem der Sprite mit dem Zeichenpunkt (origX,origY) dargestellt wird bzw. werden würde. Dabei wird der Ankerpunkt berücksichtigt.
- `public Rectangle getPlotRect( Point orig )`, gibt das Rechteck zurück, in dem der Sprite mit dem Zeichenpunkt orig dargestellt wird bzw. werden würde. Dabei wird der Ankerpunkt berücksichtigt.

Es wäre „sauberer" in der Methode mit `new Rectangle(...)` eine neue Instanz von `Rectangle` zu erzeugen und zurück zu geben. Da diese Methode aber für jedes Sprite bei jedem `Processing.draw()` aufgerufen wird -potentiell also tausendmal oder öfter pro Sekunde- wird hier das Attribut `plotRectangle` benutzt, das auch nur für diesen Zweck eingeführt wurde.

- `public boolean isVisible()`, getter für das Attribut `visible`.

Da die Klasse `Sprite` selbst überhaupt keine Darstellungsmethode realisiert muss das Flag `visible` von den abgeleiteten Klassen berücksichtigt werden.

- `public Size getSize()`, gibt die Größe des Sprites zurück.
- `public ANCHORTYPE getAnchor()`, gibt den Ankerpunkt zurück.
- `public void draw( PApplet applet, Point pos )`, realisiert selbst keine Darstellung, muss aber von den abgeleiteten Klassen aufgerufen werden, damit das `plotRectangle` aktualisiert wird.

Die PApplet-Instanz muss übergeben werden, weil in der Methode Methoden aus der Processing-Klasse PApplet genutzt werden. Der elegantere Weg wäre hier ein sogenannter Singleton. Dieses „Design Pattern" wird aber erst zu einem späteren Zeitpunkt in der Vorlesung eingeführt.

- `show() / hide()`, setzen bzw. löschen das visible-Flag.

3.2 SingleSprite

Die Klasse `SingleSprite` ist ein „normales" Sprite, das genau ein Image / eine Bitmap enthält. Die Klasse enthält ein Attribut und zwei Methoden:

- `private PImage image`. In diesem Attribut wird das Image gespeichert. Das Image wird im Konstruktor übergeben.

Beachtet, dass PImage -wie jedes Objekt- ein Referenzparameter ist. Wenn also das als Parameter genutzte Image nachträglich geändert wird dann hat das auch Auswirkung auf den Sprite.

- `public SingleSprite( PImage image )`, Konstruktor der Klasse `SingleSprite`. Der Parameter enthält eine Referenz auf eine Image-Instanz, die das Bitmap des Sprites enthält. Dem Konstruktor der Superklasse werden die Ausmaße des Images als Größenparameter übergeben.

- `public void draw( PApplet applet, Point newPos )`, überschriebene draw-Methode der Superklasse. Diese Methode plottet das Image auf die übergebene Position. Da die draw-Methode der Superklasse aufgerufen wird ist mit der getPlotRect-Methode der Superklasse das „genutzte" Rechteck verfügbar.

3.2.1 Beispiel-Code

```
public class SingleSpriteExample
{
    public class Main extends PApplet
    {
        PImage ballImage;
        SingleSprite aSingleSprite;
        Point drawpoint = new Point( 100, 100 );

        public static void main( String[] args )
        {
            PApplet.main( Main.class );
        }

        @Override
        public void settings()
        {
            setSize( 800, 600 );
        }

        @Override
        public void setup()
        {
            super.setup();
            ballImage = loadImage( "ball.png" );
            aSingleSprite = new SingleSprite( ballImage );
        }

        @Override
        public void draw()
        {
            background( 255 );
            aSingleSprite.draw( this, drawpoint );
        }
    }
}
```

3.2.2 MultiSprite

Die Klasse MultiSprite enthält im Gegensatz zum SingleSprite ein oder mehrere Bitmaps. Immer eines der Bitmaps ist das „aktuelle" Bitmap, das beim Aufruf der draw-Methode dargestellt wird. Die Klasse enthält drei Attribute und zwölf Methoden:

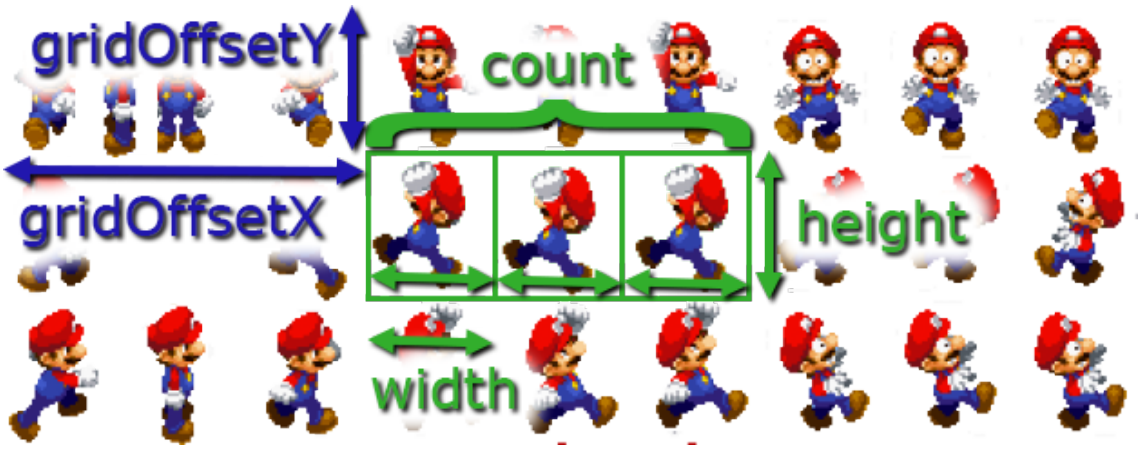
- `private int currentFrame`, enthält den Index des aktuellen Bitmaps. Der Index wird in der Klasse nicht automatisch verändert sondern wird von der nutzenden Klasse gesetzt.
- `private ArrayList<PImage> frames`, enthält Referenzen auf die verschiedenen Bitmaps bzw. Instanzen der PImage-Klasse.
- `private SpriteEvent onSpriteEvent`, enthält einen optionalen Eventhandler, der, falls er gesetzt ist, einmal vor der Darstellung eines Bitmaps und einmal danach aufgerufen wird. Mit diesem Mechanismus können z.B. Sprites synchronisiert werden.
- `public MultiSprite( int width, int height )`, (leerer) Konstruktor der Klasse. Die Angaben zu Breite und Höhe werden an den Konstruktor der Superklasse übergeben.
- `public MultiSprite( int width, int height, ANCHORTYPE anchor )`, (leerer) Konstruktor der Klasse. Die Angaben zu Breite, Höhe und Anchortype werden an den Konstruktor der Superklasse übergeben.
- `public int getCurrentFrame()`, liefert den Index des aktuellen Bitmaps zurück.
- `public List<PImage> getFrames()`, liefert die Liste der Bitmaps zurück.
- `public void setOnSpriteEvent( SpriteEvent onSpriteEvent )`, setzt den SpriteEvent-Handler.
- `public void draw( PApplet applet, Point newPos )`, stellt das aktuelle Bitmap an der Position `newPos` dar.

> *Auch hier wird die PApplet-Instanz wie oben erläutert genutzt.*

- `protected int getNextFrame()`, diese Methode wird innerhalb der Klasse aufgerufen, um den Index des Images zu ermitteln, das auf den aktuellen Index folgt. Im Falle des `MultiSprites` ist das der zyklisch nächste in der Liste der Bitmaps. Ableitungen von MultiSprite wie `SequencedSprite` können durch Überschreiben das Verhalten entsprechend ändern.
- `public void nextFrame()`, diese Methode kann von der nutzenden Klasse aufgerufen werden um das nächste Bitmaps darzustellen.

> *Die Methode* ***`nextFrame`*** *muss nicht aufgerufen werden, bzw. ist nur dann zu nutzen, wenn für die animierte Darstellung das zyklische Verhalten gewünscht ist. Alternativ kann mit der Methode* ***`setCurrentFrame`*** *auf der gewünschte Frameindex gesetzt werden.*

- `public void addFrames( PApplet applet, PImage source, int gridOffsetX, int gridOffsetY, int count )`, ergänzt ein oder mehrere Bitmaps zu der internen Liste der Bitmaps. Alle zu ergänzenden Bitmaps müssen dabei aus einem Image stammen, in einer Zeile angeordnet sein und die Größe haben, die beim Konstruktor angegeben wurde. Die folgende Grafik verdeutlicht die Zusammenhänge (so gut es geht...):



- `public void addFrameCopy( int source )` ergänzt eine Kopie eines bereits genutzten Bitmap. Der Parameter `source` ist dabei der Index der zu kopierenden Bitmap. Diese Funktion ist besonders nützlich beim Anlegen von zyklischen Animationssequenzen, also z.B. 1-2-3-4-3-2-1.

- `public void clearFrames()` löscht die Liste der Bitmaps.

3.2.3 Beispiel-Code (relevanter Ausschnitt)

```
...
public class MultiSpriteScene extends Scene
{
    private PImage spritemap;
    private MultiSprite boySprite;
    private Point aPoint = new Point( 100, 100 );

    public BoyScene( ScenedApp app )
    {
        super( app );
        spritemap = app.loadImage( "mario.png" );
        boySprite = new MultiSprite( 64, 64 );
    }

    @Override
    void init()
    {
        super.init();
        boySprite.addFrames( app, spritemap, 64, 128, 3 );
```

```
    }

    @Override
    void draw()
    {
        super.draw();
        app.background( 255 );
        boySprite.draw( app, aPoint );
        if ( pApp.millis() % 1000 == 999 )
           boySprite.nextFrame();
    }
    ...
}
...
```

3.2.4 AnimSprite

Die Klasse erbt von `MultiSprite`. Die Erweiterung zu `MultiSprite` ist, dass AnimSprite mit einer, dem Konstruktor übergebenen Taktung automatisch den jeweils nächsten Frame aktiviert. Die Klasse enthält zwei Attribute und vier Methoden:

- `private final int framerate` enthält die Geschwindigkeit mit der die Bitmaps „durchgeschaltet" werden in Millisekunden. Beispiele:
  - `framerate = 1000;` ⇒ Ein Bild pro Sekunden
  - `framerate = 500;` ⇒ Zwei Bilder pro Sekunden
- `private int nextFrameAt = 0;` Attribut, in dem sich die Klasse „merkt", wann der nächste Bildwechsel ausgeführt werden muss.
- `public AnimSprite( int width, int height, int framerate )` und texttttpublic AnimSprite(int width, int height, int framerate, ANCHORTYPE anchor) Konstruktoren mit den offensichtlichen Funktionalitäten.
- `public void draw( PApplet applet, Point newPos )` überschreibende draw-Methode. Innerhalb dieser draw-Methode wird der Bildwechsel ausgeführt.
- `public void resetAnimCounter()` setzt den internen Zähler auf 0. Das führt zu einem sofortigen Bildwechsel beim nächsten draw-Aufruf.

3.2.5 Beispiel-Code (relevanter Ausschnitt)

```
...
public class MultiSpriteScene extends Scene
{
```

```
    private PImage spritemap;
    private AnimSprite boySprite;
    private Point aPoint = new Point( 100, 100 );

    public BoyScene( ScenedApp app )
    {
        super( app );
        spritemap = app.loadImage( "mario.png" );
        boySprite = new AnimSprite( 64, 64 , 100 );
    }

    @Override
    void init()
    {
        super.init();
        boySprite.addFrames( app, spritemap, 64, 128, 3 );
    }

    @Override
    void draw()
    {
        super.draw();
        app.background( 255 );
        boySprite.draw( app, aPoint );
    }
    ...
}
...
```

3.3 SequencedSprite

die von `AnimSprite` abgeleitete Klasse `SequencedSprite` führt komplexe Animationen anhand von Sequenzen durch. Die Sequenzen werden in der Hilfsklasse `Sequence` (s.o.) gespeichert. Von den eingestellten Sequenzen ist immer eine die „aktuelle". Diese wird in der angebenen Frequenz in der Reihenfolge der Indexe wiedergegeben. Wenn die aktuelle Sequenz beendet ist und eine Folgesequenz in der aktuellen Sequenz angegeben ist, dann wird die genannte Sequenz die aktuelle. Andernfalls wird die aktuelle Sequenz von vorne gestartet.
Die Klasse `SequencedSprite` hat drei Attribute und vier Methoden:

- `HashMap<String, Sequence> sequences`, die Hashmap `sequences` enthält alle Sequencen die mit ihrem jeweiligen Namen indiziert sind.
- `private Sequence currentSequence`, Referenz auf die aktuelle Sequenz

- `private int currentFrameNr`, aktueller Bitmap-Index in der aktuellen Sequenz
- `public SequencedSprite( int width, int height, int framerate )` und `public SequencedSprite( int width, int height, int framerate, ANCHORTYPE anchor )`, Konstruktoren mit den offensichtlichen Funktionalitäten
- `public void addSequence( Sequence sequence )`, ergänzt eine Sequenz zu der Hashmap mit den Sequenzen. Falls bereits eine Sequenz gleichen Namens in der Hashmap eingestellt ist wird der Aufruf ignoriert.
- `public void gotoSequence( String sequenceName )`, startet die genannte Sequenz.
-

3.3.1 Beispiel-Code (relevante Ausschnitte)

```
public class MarioSequenced
{
    private SequencedSprite marioSprite = new SequencedSprite( 71, 80, 100 );

    public void init( PApplet app, String filename )
    {
        PImage spriteMap = app.loadImage( filename );
        marioSprite.addFrames( app, spriteMap, 0, 41, 6 );  // run and stop south
        marioSprite.addFrames( app, spriteMap, 0, 121, 6 ); // run and stop west
        marioSprite.addFrames( app, spriteMap, 0, 201, 6 ); // run and stop east
        marioSprite.addFrames( app, spriteMap, 0, 281, 6 ); // run and stop north
        marioSprite.addFrames( app, spriteMap, 0, 361, 6 ); // sleep and panic

        marioSprite.addSequence( new Sequence( "run south", "run south", 0, 1, 2, 1 ) );
        marioSprite.addSequence( new Sequence( "stop south", "stop south", 3, 4, 5, 4 ) );

        marioSprite.addSequence( new Sequence( "run west", "run west", 6, 7, 8, 7 ) );
        marioSprite.addSequence( new Sequence( "stop west", "stop west", 9, 10, 11, 10 ) );

        marioSprite.addSequence( new Sequence( "run east", "run east", 12, 13, 14, 13 ) );
        marioSprite.addSequence( new Sequence( "stop east", "stop east", 15, 16, 17, 16 ) );

        marioSprite.addSequence( new Sequence( "run north", "run north", 18, 19, 20, 19 ) );
        marioSprite.addSequence( new Sequence( "stop north", "stop north", 21, 22, 23, 22 ) );

        marioSprite.addSequence( new Sequence( "bored", "bored", 24, 25, 26, 25 ) );

        marioSprite.addSequence( new Sequence( "panic_intro", "panic", 28 ) );
```

```
        marioSprite.addSequence( new Sequence( "panic", "panic", 27, 29 ) );

        marioSprite.gotoSequence( "bored" );
    }

    ...

    public void draw( PApplet app, Point pos )
    {
        marioSprite.draw( app, pos );
    }
}
...
```