# mblght

## Dokumentation

*Benjamin Graf*

*Entstanden am IMWI – Institut für Musikwissenschaft und Musikinformatik*

*Karlsruhe, September 2012*

# Inhaltsverzeichnis

# Systemarchitektur

## SuperCollider für die Lichtdatenverarbeitung

SuperCollider ist eine auf digitale Audiosignalverarbeitung ausgerichtete Softwareumgebung ursprünglich entwickelt von James McCartney[1]. Sie besteht eigentlich aus zwei mehr oder weniger eigenständigen Programmen: Der „SuperCollider Language" (kurz „sclang"), dem Interpreter für die Programmiersprache SuperCollider, und dem „SuperCollider Server" (kurz „scsynth"), einem Server-Programm, welches sämtliche Klangsynthese und -Verarbeitung übernimmt[2]. Für das Betriebssystem Mac OS X gibt es außerdem „SuperCollider.app", eine Programmierumgebung, die auf die Entwicklung von SuperCollider-Programmcode ausgerichtet ist. In diese ist „sclang" direkt integriert, sodass eine direkte Schnittstelle zum eigentlichen SuperCollider-Programm bereitgestellt wird. Der Klangsyntheseserver dagegen bietet nur eine OSC-Schnittstelle an. Über diese können direkt Klangverarbeitungs-Prozesse gestartet und gesteuert werden. Da die Befehle aber relativ komplex sein können und aufwändig zusammenzusetzen sind, bietet „sclang" viele Methoden und Objekte, mit denen sich die Vorgänge auf dem Server abstrahieren, zusammenfassen und steuern lassen.

„scsynth" ist spezialisiert darauf, effizient, schnell und mit möglichst geringer Latenz Datenströme zu verarbeiten. Sehr viele Methoden zur Generierung und Manipulation dieser Datenströme sind bereits implementiert, angefangen von mathematischen Operationen wie Multiplikation über verschiedenste Arten von Filtern bis hin zu komplexen Techniken wie Granularsynthese oder FFT. Solche signalverarbeitenden Objekte, sogenannte UGens (von „Unit Generator"), sind in aller Regel in C programmiert und werden als kompilierte Module dynamisch in den Server eingeladen[3]. Das macht die Signalverarbeitung effizient und ressourcenschonend. Außerdem kann der Server leicht um weitere Funktionen erweitert werden. Wenn nun also Signale auf dem Server zur Steuerung von Beleuchtungsequipment verwendet werden können, stehen dafür mit einem Schlag eine Vielzahl an sehr dynamisch einsetzbaren Werkzeugen zur Verfügung, um diese Signale zu generieren und zu manipulieren.

Da mit „scsynth" vornehmlich Audiodaten verarbeitet werden, laufen die meisten Datenverarbeitungsprozesse mit einer Samplerate von 44,1 kHz oder mehr ab. Um aber Prozesse wie beispielsweise die Modulation der Lautstärke eines Audiosignals effizienter berechnen zu können, gibt es neben Signalen mit dieser hohen Samplerate („Audiorate") auch solche, die mit einer erheblich geringeren Rate verarbeitet werden, der sogenannten „Controlrate". Diese errechnet sich aus dem Quotient von „Audiorate" und der (konfigurierbaren) Server-internen Blockgröße, also die Menge der in einem Block zu verarbeitenden Samples. Es wird pro Block ein Sample

---

[1] Vgl. SuperCollider Website, 30.9.2012, unter: http://supercollider.sourceforge.net/

[2] Vgl. SuperCollider Documentation – Guides – Client vs Server, 30.9.2012, unter: http://doc.sccode.org/Guides/ClientVsServer.html

[3] Writing Unit Generators, 30.9.2012, unter: http://doc.sccode.org/Guides/WritingUGens.html

der Controlrate berechnet. Bei einer Samplerate von 44,1 kHz und der Standard-Blockgröße von 64 Samples beträgt die Controlrate 689,0625 Hz.

Das menschliche Auge fasst einzelne visuelle Eindrücke ab einer Rate von ca. 25 Eindrücken pro Sekunde zu einem zusammengehörenden Ablauf zusammen. Deswegen können für die Steuerung von Lichtequipment problemlos Controlrate-Signale verwendet werden, ohne Qualitätseinbußen in Kauf nehmen zu müssen. Die allermeisten UGens können für beide Arten von Signalen verwendet werden. Wo aber nur Audio-Signale verwendbar sind, können Signale problemlos umgewandelt werden.

Der einzige wesentliche Unterschied zwischen der Handhabung von Signalen mit Audio- oder Controlrate liegt darin, wie sich Busse verhalten, die mit dem jeweiligen Signal bespielt werden. Busse dienen dazu, Server-Intern Daten weiterzureichen, zu summieren oder verteilen und schließlich über geeignete Interfaces (beispielsweise eine Soundkarte bei Audiosignalen) auszuspielen. Dabei wird auch zwischen Bussen für Audiorate und Controlrate unterschieden. Wenn auf einen Audiorate-Bus keine neuen Daten mehr gespielt werden, springt er (wie beim echten Äquivalent beispielsweise in einem Mischpult) auf den Wert 0 zurück, er trägt also kein Signal mehr. Ein Controlrate-Bus dagegen hält den zuletzt gesetzten Wert. Wenn ein Signal komplett abbricht, muss ein Bus, der beispielsweise den Datenstrom für die Helligkeit eines Scheinwerfers enthält, manuell auf 0 gesetzt werden, um den Scheinwerfer ausgehen zu lassen. In diesem Fall wäre es also besser, die Amplitude des Signalerzeugers auf 0 zu stellen. Die restlichen Eigenschaften, was beispielsweise das Summieren der Daten angeht, sind identisch.

## Die Beleuchtungstechnik – DMX

Im professionellen Bereich wird zur Lichtsteuerung meist DMX (genauer DMX512) verwendet. Dabei handelt es sich um ein Steuersignal, welches seriell 512 ganzzahlige Werte zwischen 0 und 255 (ein Byte) überträgt. Einen Wert nennt man einen Kanal, alle zusammen ein DMX-Universum. Dieses Signal wird über spezielle Kabel mit einem fünfpoligen XLR-Stecker übertragen. Die Verbindung erfolgt dabei von einem Steuergerät („Controller“) aus nacheinander zu jedem zu steuernden Gerät. Verschiedene Arten von Leuchtmittel haben dann verschiedene Eigenschaften, die über einen oder mehrere Steuerwerte gesteuert werden können. An jedem Gerät muss eine Adresse (zwischen 1 und 512) eingestellt werden, auf deren Steuerwert es reagiert. Ein einfacher (einfarbiger) Scheinwerfer kann dann beispielsweise über einen Wert in der Helligkeit geregelt werden. Geräte, bei denen mehrere Eigenschaften gesteuert werden können, brauchen eine Startadresse (den ersten Wert des Universums, auf den sie reagieren). Die weiteren Eigenschaften können dann über die darauf folgenden Kanäle gesteuert werden. Es gibt auch viele andere Geräte, die sich über DMX steuern lassen, beispielsweise Nebelmaschinen. Das DMX-Protokoll erlaubt im Idealfall die Übertragung von bis zu 44 vollen Universen pro Sekunde[4]. Diese Rate kann aufgrund der seriellen Charakteristik des Signals noch verbessert werden, indem nur ein Teil des Universums übertragen wird. Allerdings reichen in der Praxis 44 Aktualisierungen pro Sekunde aus, um eine visuell flüssig wirkende Darstellung zu ermöglichen.

[4] Vgl. Wikipedia, DMX512, Timing, 30.9.2012, unter: http://en.wikipedia.org/wiki/DMX512-A#Timing

## Die Beleuchtungstechnik – Art-Net

Art-Net ist ein Protokoll zur Übertragung von DMX-Steuerdaten über ein Ethernet-Netzwerk. Die offene Spezifikation wurde von der englischen Firma „Artistic Licence Engineering" entwickelt und im Internet veröffentlicht[5]. Inzwischen haben vielen Firmen dieses Protokoll übernommen und in ihre Produkte implementiert. Dadurch hat es eine weite Verbreitung und Bekanntheit gefunden. Es vereinfacht besonders die Übertragung von mehreren DMX-Universen ohne jeweils ein eigenes Kabel verwenden zu müssen.

Da normales Ethernet-Netzwerk verwendet wird, kann jeder Computer mit einem Netzwerkinterface verwendet werden, um Art-Net-Daten zu senden oder zu empfangen. Dabei kann auch sonstige Netzwerkhardware verwendet werden (z.B. Switches und Router) bis hin zum Drahtlos-Netzwerk. Gerade beim sogenannten intelligenten Licht, bei dem viele Funktionen auf einmal gesteuert werden müssen, ist es inzwischen häufig möglich, ein Gerät direkt per Art-Net zu steuern. Andernfalls muss aus dem per Art-Net übertragenen Universum ein richtiges DMX-Signal generiert werden. Das geschieht mithilfe sogenannter Art-Net-Nodes – das sind Geräte, die ein Art-Net-Signal über Ethernet empfangen und aus einem der übertragenen Universen ein DMX-Signal erstellen, das per DMX-Kabel ausgegeben wird. Solche Nodes gibt es von verschiedensten Herstellern und in unterschiedlichen Bauarten, teilweise auch mit der Möglichkeit, gleich mehrere DMX-Universen auszugeben.

## Lichtsteuerdaten aus dem Rechner – olad

Um Lichtsteuerdaten möglichst bequem im Computer verwalten zu können, gibt es die „Open Lighting Architecture"[6] (kurz OLA). Dabei handelt es sich um ein Softwarepaket, welches intern mehrere DMX-Universen verwalten und über viele verschiedene Wege senden und empfangen kann. Es gibt Treiber für verschiedene Protokolle zur Datenübertragung über Ethernet (unter anderem Art-Net) genauso wie für verschiedene Interfaces, die beispielsweise an einem USB-Port direkt ein DMX-Signal erzeugen können. Dieser Teil der Software wird zusammengefasst unter dem Namen „olad", dem OLA Daemon, und läuft als Hintergrundprozess auf einem Computer. Konfiguriert werden kann ein OLA Server über Konfigurationsdateien. Während der Laufzeit können über ein Web-Interface oder über die Kommandozeile weitere Konfigurationen vorgenommen werden, zum Beispiel welches Universum woher Daten empfängt und wohin wieder sendet. Außerdem gibt es noch Programmierschnittstellen (APIs) für C/C++ und Python, mit denen der „olad" kontrolliert und DMX-Daten ausgetauscht werden können. Damit kann „olad" als Interface zwischen eigenen Programmen und Lichtinstallationen, die verschiedenste Protokolle zur Datenübertragung nutzen, verwendet werden.

Um DMX-Daten in den Server hinein zu bekommen, gibt es verschiedene Möglichkeiten: Zum einen kann man mit beliebigen Controllern (beispielsweise Lichtsteuerpulte) DMX-Daten

---

[5] Art-Net 3, 30.9.2012, unter: http://www.artisticlicence.com/WebSiteMaster/User%20Guides/art-net.pdf

[6] Vgl. open-lighting – OLA: An open framework for DMX lighting control, 30.9.2012, unter: http://code.google.com/p/open-lighting

erzeugen und über verschiedene Protokolle in den Server einspielen. Soll aber derselbe Rechner als Controller verwendet werden, kann mit der besagte API spezielle Steuersoftware in C/C++ oder in Python programmiert werden. Im OLA-Paket sind schon einige einfache Tools enthalten, die die API verwenden, um auf verschiedene Arten DMX-Daten zu erzeugen. Es gibt vor allem für Debugging-Zwecke eine Kommandozeilen-basierte Steuerkonsole `ola_dmxconsole`. Beim Starten dieses Programms über die Kommandozeile öffnet sich eine text-basierte interaktive Steuerkonsole, in der für jeden Kanal eines Universums auf dem Server Werte gesetzt werden können. Diese Werte werden dann über alle konfigurierten Interfaces ausgegeben. Etwas einfacher ist das Programm `ola_set_dmx`, mit dem über einen einfachen Kommandozeilenbefehl Daten eines Universums geschrieben werden können. Ein Beispiel, bei dem für das erste Universum (mit der ID 0) die ersten fünf DMX-Kanäle mit den Werten 0, 10, 20, 30 und 255 gesetzt werden:

```
> ola_set_dmx --universe 0 --dmx 0,10,20,30,255
```

Für die Kommunikation aus einer anderen Software heraus gibt es das Programm `ola_streaming_client`. Wird dieses Programm von der Kommandozeile aus gestartet, ohne direkt DMX-Daten mitzugeben, öffnet sich eine Standardeingabe („stdin“). Über diese können DMX-Werte gesetzt werden. Die Eingabe der Werte eines „Frames“ wird mit einem Zeilenumbruch beendet. Durch die Standardeingabe können Programme über eine Unix Pipe relativ einfach DMX-Daten an den OLA-Server senden.

## Von SuperCollider zum Scheinwerfer

Um Steuerdaten vom SuperCollider-Server bis hin zum Scheinwerfer (oder dem zu steuernden Gerät) zu transportieren, bietet sich der folgende Weg an:

Daten, die auf dem SuperCollider-Server geschrieben werden, werden von der SuperCollider-Language mit einer bestimmten Regelmäßigkeit ausgelesen. Aus dieser heraus wird ebenfalls das Programm `ola_streaming_client` gestartet und über eine sogenannte Unix Pipe die DMX-Daten zur Standard-Eingabe des Streaming Clients übergeben. Dieser übergibt dann die Daten an den laufenden OLA Server. Steht ein Art-Net-Node zur Verfügung, kann der OLA Server nun direkt Art-Net-Daten über das Netzwerk-Interface des Rechners ausgeben. Über Standard-Netzwerk-Komponenten wird das Signal dann zum Node übertragen, der aus den Art-Net-Daten ein DMX-Signal erzeugt. Über eine entsprechende Verbindung wird dann das zu steuernde Gerät an den Node angeschlossen und gesteuert. Alternativ gibt es wie oben beschrieben auch verschiedene andere unterstützte Geräte, die teilweise sogar über USB angeschlossen werden können und direkt ein DMX-Signal ausgeben.

## Verwendete Software-Versionen

Ich habe die Lichtsteuersoftware zusammen mit folgenden Softwareversionen entwickelt und getestet:

- SuperCollider 3.5.5
- OLA Daemon Version 0.8.20
- Apple Mac OS X 10.6.8

# Quellcode-Dokumentation

In der entwickelten Software sollen Lichtsteuerdaten ähnlich wie Audio-Signale verarbeitet werden. Deshalb geschieht die eigentliche Verarbeitung der Datenströme auf `scsynth`, dem Supercollider Server. Diese Signale müssen abgetastet und an die Lichtsteuerungssoftware OLA weitergereicht werden. Das geschieht in der SuperCollider Language über einen Puffer (`DmxBuffer`), der jeweils ein DMX-Universum an Steuersignalen abfragt und an einen oder mehrere Ausgabegeräte (z.B. `OlaPipe`) weitergibt. Um nicht jede DMX-Adresse einzeln ansteuern zu müssen, können die zu steuernden Gerätschaften (`Devices`) in ihren Funktionen abstrahiert und bei einem `Patcher` registriert werden, der dann die richtigen Daten an die jeweiligen Adressen schickt.

## I. Vorbereitungen

Bevor man SuperCollider mit einer bestehenden Lichtanlage verbinden kann, muss man einige Vorbereitungen treffen, damit generierte Signale auch bei den zu steuernden Geräten ankommen.

Zuallererst sollte man natürlich sicher sein, dass alle Scheinwerfer und sonstige Geräte korrekt verkabelt, mit Strom versorgt und (falls nötig) eingeschaltet sind. Auch die Adressen der Geräte müssen bekannt oder gezielt konfiguriert sein. Wird Art-Net als Übertragungsmedium verwendet, muss man den Art-Net-Node, an den die DMX-Leitung angeschlossen ist, konfigurieren (Art-Net-Net und -Subnet sowie IP-Adresse!). Dazu muss man sicherstellen, dass die Netzwerkverbindung korrekt hergestellt werden kann und auch das eigene Netzwerkinterface korrekt konfiguriert wurde. Hier hilft als erster Test, die Verbindung zum Node mit einem Netzwerk-Ping zu testen. Die meisten USB-DMX-Interfaces müssen einfach angeschlossen werden.

Wenn alle Verbindungen hergestellt wurden, kann man den OLA-Server „olad" konfigurieren und starten. In der Konfiguration ist vor allem zu beachten, dass das Art-Net-Netz und -Subnetz eingestellt wird, auf welches der Node reagiert. Nach dem Start muss man ein olad-internes Universum anlegen und den Art-Net-Ausgabe-Port zuweisen. Das geht entweder über das Webinterface oder per Kommandozeile über die Befehle `ola_patch`[7] und `ola_artnet`[8]. Wurde alles konfiguriert, kann man die Funktion mit der `ola_dmxconsole` testen – es sollte bereits möglich sein, einzelne Kanäle zu steuern und der angeschlossene Scheinwerfer sollte entsprechend reagieren. (Als weiteres Testwerkzeug gibt es im Webinterface ein einfaches Mischpult, mit dem man die einzelnen Kanäle ansteuern kann.)

Schließlich muss man noch SuperCollider selbst installieren und die Klassen in das System einbinden. Die SuperCollider-Software erhält man auf der Webseite http://supercollider.sourceforge.net/ einschließlich Installationshinweisen für das jeweilige Sys-

[7] Vgl. Using OLA - OpenDMX.net, 30.9.2012, unter: http://opendmx.net/index.php/Using_OLA

[8] Vgl. OLA Command Line Tools - OpenDMX.net, 30.9.2012, unter: http://opendmx.net/index.php/OLA_Command_Line_Tools

tem. Die Dateien in der Anlage[9] enthalten die entwickelten Klassen und müssen in das „Extensions"-Verzeichnis der SuperCollider-Installation kopiert werden. Unter Mac OS X befindet es sich normalerweise in `~/Library/ApplicationSupport/SuperCollider/Extensions`. (Man kann sich den Pfad von SuperCollider angeben lassen, indem man den Befehl `Platform.userExtensionDir` ausführt.) Damit werden alle in den Dateien definierten Klassen automatisch geladen.

Um zu testen, ob DMX-Daten aus SuperCollider heraus beim OLA-Server ankommen, kann man das hilfreiche Kommandozeilenprogramm `ola_dmxmonitor` verwenden. Es zeigt einfach alle gerade gesetzten Werte des ausgewählten Universums an.

# II. Systemsetup

Um eine Lichtanlage sinnvoll von SuperCollider aus zu steuern, muss man erst einige Schritte zur Einrichtung solch einer Anlage vornehmen. Zuerst sollte man die Ausspielwege definieren und initiieren. Danach kann man eine Patcher-Instanz anlegen und die Ausspielwege dort registrieren. Jetzt muss man für alle real existierenden Geräte Definitionen erstellen. Schließlich legt man Instanzen für die existierenden Geräte an. Für Geräte, deren Methoden nicht bereits definiert wurden, muss man dieses noch tun.

## Ausspielwege

Das Ausspielen der Steuerdaten eines DMX-Universums geschieht über die Klasse DmxBuffer (`DmxBuffer.sc`). Eine Instanz dieser Klasse hält die Daten eines Universums vor und sendet sie über verschiedene mögliche Ausgabegeräte in regelmäßigen Abständen (standardmäßig 60 mal in der Sekunde) in die Welt. Dafür benötigt eine DmxBuffer-Instanz nun noch einen oder mehrere Ausgabewege („Output-Devices"). Davon wurden bereits zwei implementiert: `OlaPipe`, um über eine Unix Pipe Daten zu einem lokalen OLA-Server zu schicken und `RainbowSerial`, mit dem man Daten über ein spezielles Protokoll über eine serielle Verbindung zu einem Rainbowduino-Board[10] senden kann. Werden aber noch andere Ausspielwege benötigt, können diese durch selbst zu entwickelnde Klassen erweitert werden. Solch eine Klasse bzw. deren Instanz muss dabei auf drei Methoden reagieren:

- `new()`: Damit wird eine Instanz der Klasse erzeugt. Hier müssen bereits alle Initialisierungen vorgenommen werden, die nötig sind, um Daten senden zu können. Es können beliebige Argumente definiert werden, etwa um die Ausspielung genauer zu steuern.
- `send()`: Diese Methode wird regelmäßig aufgerufen, um Daten zu senden. Als Argument erhält sie eine sortierte Liste (`List`-Objekt) mit bis zu 512 Ganzzahlwerten zwischen 0 und 255, die den Werten eines DMX-Universums entsprechen.
- `close()`: Hiermit können, falls notwendig, beispielsweise Subprozesse beendet oder Datei-Handler, die offen gehalten wurden, geschlossen werden.

---

[9] Alle Dateien im Pfad /Code/mblght/sc/

[10] Rainbowduino v3.0, 30.9.2012, unter: http://www.seeedstudio.com/wiki/Rainbowduino_v3.0

## OlaPipe

Eine Instanz der Klasse `OlaPipe` erzeugt über eine sogenannte Unix Pipe eine Verbindung zum OLA Server. Diese "Pipe" wird in der Supercollider-Software über die `Pipe`-Klasse erzeugt und ist im wesentlichen ein (Kommandozeilen-) Programmaufruf aus SuperCollider heraus. Über die Standard-Texteingabe dieses Programms können dann Daten an das Programm gesendet werden. `OlaPipe` nutzt das Programm `ola_streaming_client`, das bei der Installation des OLA-Softwarepakets automatisch mit installiert wird.

Erfahrungsgemäß ist diese Verbindung bei bis zu 120 Aktualisierungen pro Sekunde stabil. Darüber hinaus kann es bei meinem Testsystem zu Abstürzen kommen. Allerdings reicht diese Rate bei weitem aus, um ein flüssig wirkendes Gesamtbild zu erstellen. Außerdem liegt sie über der maximalen Framerate von DMX.

### Benutzung

Um eine Instanz zu erzeugen, muss wieder die Methode `new` aufgerufen werden. Als Argument dieser Methode kann die ID des gewünschte DMX-Universums des OLA Servers als Ganzzahl angegeben werden. Ohne Angabe erfolgt eine Verbindung zum ersten Universum mit der ID 0. (Der Dienst `olad` sollte bereits korrekt konfiguriert und gestartet sein, siehe „I. Vorbereitungen“, S. 15).

```
p = OlaPipe.new(0)
```

Diese Instanz reagiert auf die Methode `send`, mit der Daten an das DMX-Universum gesendet werden können. Die Methode erwartet eine Liste (Objekt der Klasse `List`) mit bis zu 512 Ganzzahlwerten zwischen 0 und 255. Ein Beispiel mit der eben erzeugten Instanz:

```
var data = List.newUsing([155, 212, 0, 47]);
p.send(data);
```

So kann man die korrekte Funktionsweise ohne weitere Einrichtungsarbeit testen. Bei bestehender Verbindung sollten diese Werte im OLA-Server ankommen.

Schließlich kann man die `Pipe`-Instanz wieder beenden mit

```
p.close()
```

## RainbowSerial

Diese Klasse bietet die Möglichkeit, DMX-Daten über eine serielle Verbindung über USB zu senden. Mit einem Rainbowduino-Board[11] können diese Daten verwendet werden, um bis zu 96 RGB-LEDs anzusteuern und somit mehrfarbige Scheinwerfer zu simulieren. Bei diesem speziellen Board steht allerdings nur eine begrenzte Datenrate über die serielle Verbindung zur Verfügung. Um trotzdem eine flüssige Darstellung mit mehr als 25 Frames pro Sekunde zu erreichen, werden in diesem Fall nur 196 4-bit-Werte übertragen. Das ursprüngliche DMX-Signal (512 8-bit-Werte) wird also in der Bandbreite begrenzt. Damit reduziert sich beispielsweise bei Farbmischungen auch die Anzahl darstellbaren Farben, aber für Demonstrations- und Debugging-Zwe-

[11] Rainbowduino v3.0, 30.9.2012, unter: http://www.seeedstudio.com/wiki/Rainbowduino_v3.0

cke reicht das hier verwendete "Protokoll" aus. Auf dem Board muss die passende Firmware[12] installiert werden.

Auch wenn man grundsätzlich auch andere serielle Empfänger anschließen und die Daten beliebig interpretieren könnte, wurde diese Schnittstelle eigentlich für die Nutzung mit dem Rainbowduino-Board entwickelt. Dabei wird angenommen, dass 96 RGB-"Scheinwerfer" sequenziell adressiert mit je einem Wert für die Farbe Rot, Grün und Blau verwendet werden. Dementsprechend sollten auch im Patcher die entsprechenden Device-Typen verwendet werden.

### Benutzung

Zuerst sollte eine (physikalische) Verbindung zum Board hergestellt werden. Dabei legt das Mac OS X-Betriebssystem eine Gerätedatei im `/dev`-Verzeichnis an. Über diese wird die serielle Verbindung aufgebaut. Zuerst muss man heraus finden, ob und an welcher Stelle SuperCollider die Gerätedatei findet. Das macht man mit:

```
SerialPort.devices;
```

Man erhält eine Liste mit allen gefundenen Geräten und muss sich nun den Index des richtigen Geräteeintrags in dieser Liste merken. Dann kann eine Verbindung hergestellt werden, wobei man eben diesen Index als Argument angibt.

```
var index = 3;
r = RainbowSerial.new(index);
```

Das öffnet die serielle Verbindung und man kann nun auch über die `send()`-Methode Daten an das Board senden.

```
var data = List.newUsing([255, 0, 0, 33, 255]);
r.send(data);
```

Diese Verbindung sollte nach Benutzung wieder geschlossen werden, auch wenn SuperCollider das beim ordentlichen Beenden automatisch macht.

## DmxBuffer

Um eine Instanz der Klasse DmxBuffer zu erstellen und in der globalen Variable `b` zu speichern, genügt folgender Code:

```
b = DmxBuffer.new();
```

Intern wird bei diesem Aufruf mit der Methode `makeRunner` eine Routine erzeugt, die die gepufferten Daten ausspielt (siehe Datei `DmxBuffer.sc` Zeile 54). Dabei wird versucht, eine bestimmte Framerate zu erreichen. Diese liegt standardmäßig bei 60 Frames pro Sekunde, kann aber eingestellt werden über die Setter-Methode `fps_()`:

```
b.fps_(30)
```

Dabei ist zu beachten, dass zu höhere Framerate-Werte natürlich mehr Systemressourcen benötigen und zu hohe Werte sogar zu Abstürzen der ganzen SuperCollider-Software führen können.

[12] In der Anlage, Pfad /Code/Rainbowduino/firmwre_4bit.ino

Nun kann man direkt ein oder mehrere "Output-Devices" in Form von Instanzen der oben beschriebenen Klassen an der DmxBuffer-Instanz registrieren, indem man deren Methode `addDevice` aufruft.

```
b.addDevice(p);
```

DmxBuffer sendet dann seinen (bis jetzt noch leeren) Puffer regelmäßig über die `send`-Methode an die registrierte Device-Instanz. Es können problemlos mehrere Device-Instanzen gleichzeitig registriert werden, um einfach die gleichen Daten über mehrere Wege auszuspielen.

Um den Datenpuffer der DmxBuffer-Instanz zu füllen, steht die `set`-Methode zur Verfügung. Diese kann auf drei verschiedene Wege Daten in den Puffer schreiben:

a) Den Wert eines bestimmten Kanals des DMX-Universums setzen.
Dafür müssen zwei Argumente übergeben werden. Das erste ist der Index des zu setzenden Kanals, das zweite der Wert als Ganzzahl zwischen 0 und 255. Das folgende Beispiel setzt den Wert des 9. DMX-Kanals auf 255:

```
b.set(9, 255);
```

b) Die Werte des kompletten Puffers setzen.
Dafür muss nur ein Argument übergeben werden: Eine geordnete Liste (Klasse `List` in SuperCollider) mit allen 512 Werten des DMX-Universums als Ganzzahlen zwischen 0 und 255. Alternativ können dabei auch weniger als 512 Werte angegeben werden, wodurch die übrigen Kanäle einfach nicht gesetzt werden. In diesem Beispiel werden die ersten 100 Kanäle auf zufällige Werte zwischen 0 und 255 gesetzt:

```
var data = List.newUsing({256.rand}!100);
b.set(data);
```

c) Eine bestimmte Anzahl von Werten für aufeinander folgende Kanäle ab einem bestimmten Startkanal setzen.
Dabei muss als erstes Argument dieselbe Liste wie in b) aber mit weniger als 512 Werten und als zweites Argument der erste zu setzende Kanal angegeben werden. Beispielsweise um 10 Kanäle ab Kanal 30 auf 0 zu setzen:

```
var data = List.newUsing(0!10);
var offset = 30;
b.set(data, offset)
```

Wird der Puffer der DmxBuffer-Instanz auf diese Weise beschrieben, sollten sich die Änderungen direkt in allen registrierten Output Devices wiederspiegeln.

Will man zur Kontrolle den momentanen Inhalt des Puffers auslesen, kann man das mit der `get`-Methode tun. Gibt man kein Argument an, wird eine sortierte Liste mit allen 512 Werten des Puffers zurück gegeben. Man kann aber auch die Nummer eines Kanals angeben, um nur dessen Wert zu erhalten.

```
b.get(); // oder
b.get(14);
```

# Lichtequipment

## Geräteprofile

Geräte, die über Artnet/DMX gesteuert werden sollen, existieren als Instanzen der `Device`-Klasse. Da unterschiedliche Geräte sehr verschiedene Funktionen haben können (vom einfachen dimmbaren Scheinwerfer über RGB-LED-Leuchtmittel über Stroboskope bis hin zu Nebelmaschinen) muss für jeden Gerätetyp ein Profil vorhanden sein, in dem dessen Funktionen beschrieben werden. Diese Beschreibung soll dabei einigermaßen abstrakt gehalten sein, um zum einen auch komplexere Steuerungen mit wenigen Parametern vornehmen zu können. Somit kann zum Beispiel die Ausrichtung eines bewegbaren Scheinwerfers („Moving Head“) statt über Anteile am maximalen Drehwinkel des Geräts über einfache XY-Koordinaten erfolgen. Zum anderen können über diese Profile gleiche Funktionen verschiedener Geräte, die jedoch unterschiedlich anzusteuern sind, vereinheitlicht werden. Denkbar ist beispielsweise eine Funktion „Farbe“, die als Parameter RGB-Werte erhält und für alle Geräte, die Farbwechsel unterstützen, die jeweils passenden DMX-Parameter einstellt – selbst wenn ein Gerät eigentlich eine CMYK-Farbmischung vorsieht.

Einige Profile sind bereits angelegt und werden automatisch geladen. Das passiert in der Klasse `Device` in der Klassenmethode `initClass`, die beim Laden der Klasse automatisch aufgerufen wird. Darin wird wiederholt die Klassenmethode `addType` aufgerufen. Diese verlangt als Argumente zum einen einen Titel in Form eines Symbols und zum anderen ein `Event`-Objekt, in dem einige bestimmte Schlüssel mit bestimmten Werten ausgestattet werden müssen. Als Beispiel nun die Implementierung eines einfachen Dimmer-Gerätes, das lediglich einen Wert eines Geräts ändern kann (üblicherweise die Helligkeit eines einfarbigen Scheinwerfers):

```
Device.addType(\dim, (
	// die von diesem Gerät benötigte Anzahl an Kanälen:
	channels: 1,

	// Die Anzahl an Argumenten, die jede Methode (außer der Standard-Methode init) benötigt:
	// (Hier gibt es nur eine Methode, nämlich „dim“)
	numArgs: (dim: 1),

	// Eine Methode bzw. Funktion, auf die dieses Gerät reagiert - in diesem Fall eine einfache
	// Helligkeitsregelung. Der hier angegebenen Funktion wird beim Aufruf eine Referenz auf
	// die eigene Instanz sowie ein Array mit den nötigen Argumenten (hier nur eins) übergeben.
	dim: { |self, args|
		// Die Instanz der Klasse Device, die hier übergeben wurde, reagiert auf die Methode
		// setDmx, die den Wert eines bestimmten Kanals dieses Geräts setzen kann. Die
		// absolute Adresse im DMX-Universum muss hier noch nicht berücksichtigt werden!
		self.setDmx(0, (args[0] * 255).round.asInteger);
	},

	// Die Init-Methode wird bei der Device-Initialisierung automatisch aufgerufen. Sie muss
	// nicht vorhanden sein, kann aber helfen, das Gerät betriebsbereit zu machen (beispiels-
	// weise indem eine Blende geöffnet wird).
	init: { |self|
		// Hier wird das Gerät schlicht ausgeschaltet.
		self.setDmx(0, 0);
	}
));
```

Mit dieser Methode kann auch zur Laufzeit des Programms dynamisch ein neuer Typ angelegt und dann auch geladen werden. Man kann den Aufruf auch an beliebiger anderer Stelle im Quellcode ausführen. Es empfiehlt sich aber, die Gerätedefinitionen bereits beim SuperCollider-Start zu laden.

Alle bereits geladenen Typen kann man mit der Klassenmethode `typeNames` abfragen:

```
Device.typeNames();
```

### Devices

Instanzen dieser `Device`-Klasse haben nun also einen Typ, durch den ihre Funktionen festgelegt sind, und eine Adresse, unter der sie im DMX-Universum zu finden sind. Man kann eine solche Instanz einfach anlegen über die `new`-Methode mit den Argumenten Gerätetyp und Adresse, wobei der Gerätetyp ein Symbol mit der Bezeichnung des Geräts ist und die Adresse der entsprechende Ganzzahlwert. Hier wird ein Gerät des Typs „smplrgb“ mit der Adresse 10 angelegt und in der globalen Variable `d` gespeichert.

```
d = Device.new(\smplrgb, 10);
```

Um nun typenbezogene Methoden auszuführen wird die `action`-Methode der `Device`-Instanz benutzt. Diese benötigt als erstes Argument die gewünschte Typen-Methode und als zweites Argument alle Argumente der aufzurufenden Methode in Form eines Arrays.

```
// Hier wird die Farbe eines eben angelegten RGB-Farben mischenden Scheinwerfers auf Rot eingestellt
d.action(\color, [255, 0, 0]);
```

Um zu überprüfen, ob ein Gerätetyp eine bestimmte Funktion überhaupt unterstützt, gibt es die `hasMethod`-Methode:

```
d.hasMethod(\color); // liefert true
```

Den aktuellen „Zustand“ des Geräts, also die Steuerdaten, die im Moment gesetzt sind, kann man abfragen mit der Methode `getDmx`. Diese gibt eine Liste mit Werten aller Kanäle zwischen 0 und 255 zurück. Man kann, wenn nötig, auch direkt Kanaldaten setzen mit der Methode `setDmx`, die als Argumente den zu setzenden Kanal (bezogen auf dieses Gerät, nicht das ganze Universum!) und den Wert erwartet.

## Der Patcher

Ein `Patcher`-Objekt verwaltet nun alle zu steuernden Geräte in einem DMX-Universum. Es vermittelt zwischen den Methoden der am `Patcher` registrierten `Devices` und dem `DmxBuffer`, der ebenfalls am Patcher registriert wird. Das Aufrufen einer Methode eines `Devices` führt dazu, dass die zurückgegebenen Werte der einzelnen Kanäle desselben an die richtige Stelle im DMX-Universum beziehungsweise im `DmxBuffer` geschrieben werden. Nach einer korrekten Einrichtung muss sich der Nutzer keine Gedanken mehr darüber machen, welcher Steuerwert wo hin geschrieben werden muss. Dadurch werden einmal programmierte Effekte oder Szenen auch auf andere Installationen portierbar. Der `Patcher` vermittelt außerdem zwischen der Signalverarbeitung, die auf SuperColliders Serverkomponente „scsynth“ berechnet wird, und der Ausspielung, die im „Language“-Teil „sclang“ stattfindet.

## Patcher verwalten

Man initiiert eine Instanz der Klasse `Patcher` wie gewohnt mit der `new`-Methode und gibt als Argument eine ID in Form eines Titels (vom Typ Symbol) an. Wenn der Patcher beispielsweise die Beleuchtung der Bühne steuern soll, eignet sich die ID `\stage`:

```
p = Patcher.new(\stage);
```

Eine Liste aller existierenden Patcher erhält man über die Klassenmethode `all`:

```
Patcher.all();
```

So kann man auch gezielt einzelne Patcher ansprechen. Den Patcher mit der vorher vergebenen ID `\stage` erhält man so:

```
Patcher.all().at(\stage); // oder per Array-Syntax:
Patcher.all[\stage];
```

Es wird auch ein Standard- („default-“) Patcher vorgehalten. Wird zum ersten Mal ein `Patcher`-Objekt erstellt, wird dieses zum Standard-Patcher gemacht. Man kann einen bestimmten Patcher mit der Methode `makeDefault` zum Standard-Patcher machen:

```
Patcher.all().at(\stage).makeDefault;
```

Danach kann man auf diesen Patcher mit der Klassenmethode `default` zugreifen:

```
Patcher.default; // liefert nun den Patcher mit der ID \stage
```

## Puffer registrieren

Der Patcher übergibt Steuerdaten automatisch an eine `DmxBuffer`-Instanz, wenn diese an ihm registriert wurde. Dafür gibt es die Methode `addBuffer`, die als Argument den entsprechenden Puffer braucht:

```
// p ist der eben angelegte Patcher, b ein vorher erstelltes DmxBuffer-Objekt
p.addBuffer(b);
```

Eine Liste aller registrierten Puffer gibt die „Getter“-Methode `buffers` zurück.

Man kann einen Puffer mit der `removeBuffer`-Methode entfernen, die den Index des Objekts als Argument benötigt.

```
p.removeBuffer(0);
```

Diese Methode beendet den jeweiligen Puffer auch, wodurch wiederum dessen Ausgabegeräteinstanzen geschlossen werden sollten.

## Devices registrieren

Hat man wie oben beschrieben einige `Device`-Instanzen angelegt, kann man diese am `Patcher`-Objekt registrieren. Das geht mit der Methode `addDevice`.

```
// p enthält eine Patcher-Instanz, d eine Device-Instanz (s.o.)
p.addDevice(d);
```

Natürlich kann ein Device auch direkt beim Hinzufügen erstellt werden.

```
p.addDevice(Device.new(\smplrgb, 20));
```

Da der Patcher die `Device`-Instanzen sowieso vorhält, ist das der einfachere Weg, insbesondere dann, wenn viele Instanzen registriert werden müssen. So lassen sich auch die Adressen „algorithmisch“ vergeben, beispielsweise in einer Programmschleife. Das verringert zwar den Einrichtungsaufwand auf Programmseite, dafür müssen die Geräte korrekt programmiert werden.

Man erhält eine Liste mit allen registrierten Devices über die „Getter“-Methode `devices`:

```
p.devices();
```

## Gruppen

Um bei der späteren Steuerung gezielter Geräte ansprechen zu können, gibt es die Möglichkeit, Geräte Gruppen zuzuordnen. Damit können beispielsweise Bühnenlicht von Saallicht oder Frontalbeleuchtung und Beleuchtung von der Seite/von Hinten getrennt werden. Dazu müssen im Vorfeld die gewünschten Gruppen angelegt werden. Hier als Beispiel eine Gruppe mit dem Titel `\stage`:

```
p.addGroup(\stage); // (Name der Gruppe als Symbol)
```

Eine Liste aller bereits angelegten Gruppen gibt die Methode `groupNames` zurück:

```
p.groupNames();
```

Ein bereits instanziiertes Device kann dann mit der Methode `addDeviceToGroup` dieser Gruppe zugeordnet werden.

```
// d referenziert ein Device, welches auch schon beim Patcher registriert wurde
p.addDeviceToGroup(d, \stage);
```

Auf eine Gruppe kann nun über die „Getter“-Methode `groups` zugegriffen werden. Diese liefert erst einmal alle Gruppen in Form einer Zuordnungsliste (`IdentityDictionary` in SuperCollider) zurück. Auf eine spezielle Gruppe kann dann aber ganz einfach mit der Methode `at` oder der Array-Syntax zugegriffen werden:

```
p.groups().at(\stage); // oder
p.groups[\stage]
```

Dabei wird wiederum jeweils eine Liste mit allen der Gruppe zugeordneten Devices zurückgegeben.

Devices aus einer Gruppe entfernen kann man mit der Methode `removeDeviceFromGroup`, welche als Argumente den Index des zu entfernenden Geräts und den Gruppennamen benötigt:

```
p.removeDeviceFromGroup(0, \stage); // entfernt das erste Device aus der Gruppe \stage
```

Eine ganze Gruppe kann man löschen mit der Methode `removeGroup`:

```
p.removeGroup(\stage); // entfernt die Gruppe \stage
```

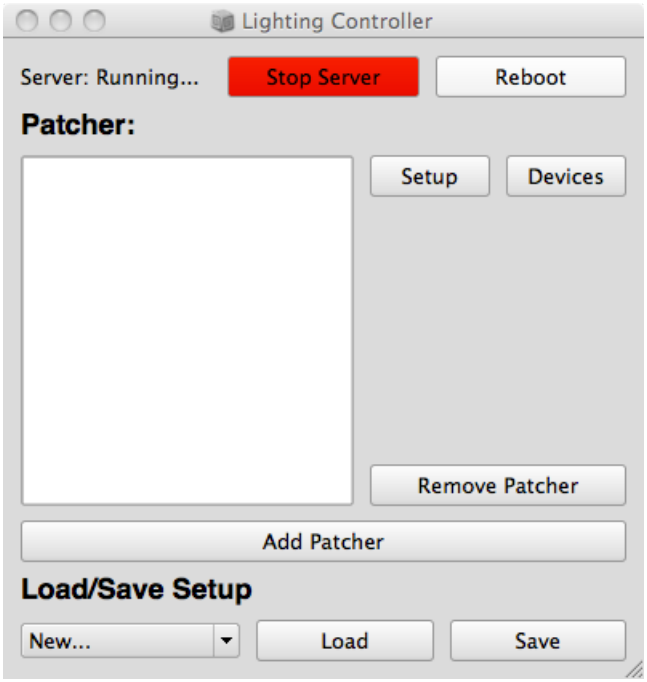
Dies löscht jedoch nur die Gruppe und die Gerätezuordnungen, nicht die Geräte selbst! Diese bleiben ganz normal am Patcher registriert.

# LGui

Um die Konfigurationsarbeit des SuperCollider-Systems bis hier hin zu erleichtern, gibt es eine kleine grafische Benutzerschnittstelle, die die wichtigsten Funktionen zur Einrichtung anbietet. Um es aufzurufen, genügt der Standard-Klasseninitiator:

```
LGui.new(); // oder kürzer: LGui();
```

## Das LGui-Hauptfenster

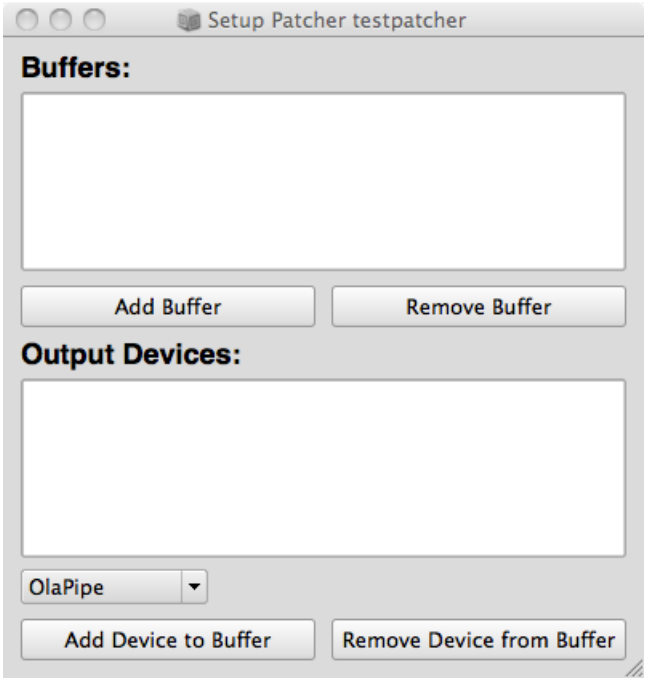



Hier stehen einige grundlegende Funktionen zur Verfügung. Von oben links nach unten rechts:

- **Server:** Der Status des „scsynth"-Servers wird angezeigt (in diesem Fall läuft er bereits). Je nach Status kann der Server gestoppt oder gestartet werden.
- **Reboot:** Hier kann das komplette System neu geladen werden. Entspricht einem „Class Recompile", bei dem im Wesentlichen SuperCollider („sclang" und „scsynth") neu gestartet wird. Wenn nicht anders konfiguriert muss man danach das LGui-System manuell neu starten!
- **Patcher:** In dem weißen Feld werden alle angelegten Patcher angezeigt (in diesem Fall keine). Wurde ein Patcher angelegt, kann man ihn markieren, um weitere Funktionen auszuführen.
- **Setup:** Ausgabeschnittstellen des ausgewählten Patchers konfigurieren (s.u.).
- **Devices:** Zu steuernde Geräte anlegen und organisieren (s.u.).
- **Remove Patcher:** Den ausgewählten Patcher löschen. Dadurch werden auch alle Devices und Ausgabe-Interfaces beendet und geschlossen/deaktiviert.

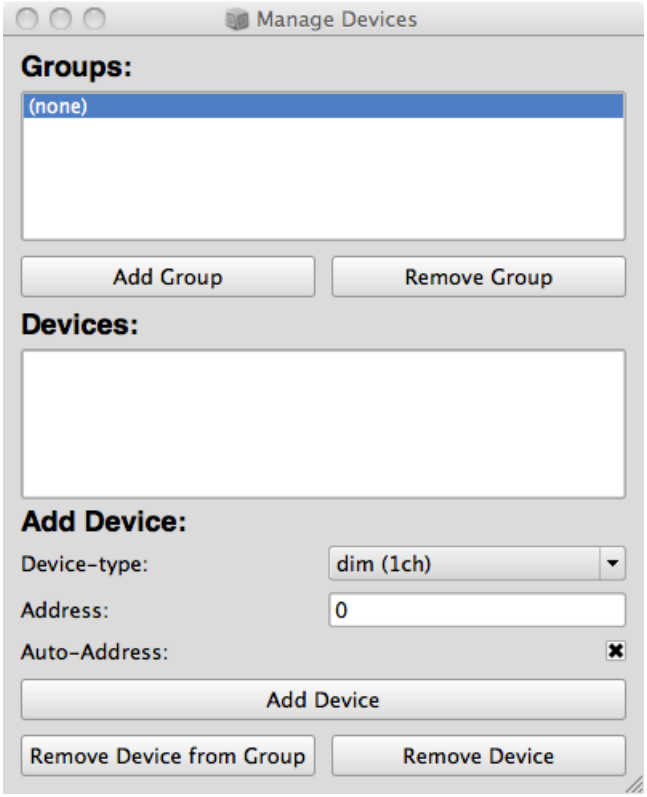
- **Add Patcher:** Einen neuen Patcher erstellen. Es öffnet sich ein Dialog, in dem ein Name vergeben werden kann.
- **Load/Save Setup:** Konfigurationen laden und speichern. Mit **Save** wird die aktuelle Konfiguration (angelegte Patcher mit allen registrierten Devices und Ausgabe-Interfaces) gespeichert. Wenn dabei in der Auswahlliste links „New...“ ausgewählt wurde, öffnet sich ein Dialog, um einen neuen Namen anzulegen und die Konfiguration darunter in einem neuen „Slot“ zu speichern. Andernfalls wird die ausgewählte Konfiguration überschrieben. Mit **Load** wird die mit der Auswahlliste links ausgewählte Konfiguration geladen (und die aktuell geladene Konfiguration überschrieben).
  Wird eine Konfiguration unter dem Namen „default“ gespeichert, wird diese beim Initialisieren des LGui-Systems automatisch geladen!

## Das Setup-Fenster




Hier werden die Ausgabe-Schnittstellen definiert. Zuerst muss ein Puffer über den Button **Add Buffer** angelegt werden. In dem weißen Feld oben wird dieser dann aufgelistet. Der ausgewählte Puffer kann über **Remove Buffer** entfernt werden. Wenn ein Puffer angelegt und ausgewählt wurde, kann man darunter ein Ausgabegerät anlegen und am Puffer registrieren, indem man den gewünschten „Treiber“ in der Auswahlliste auswählt und den Button **Add Device to Buffer** drückt. Dabei öffnet sich ein Dialog, der eventuell zu vergebende Eigenschaften dieses Geräts bestimmen lässt, bei „OlaPipe“ beispielsweise die Nummer des Ziel-Universums. Mittels **Remove Device From Buffer** kann dieses Ausgabegerät angehalten und entfernt werden.

### Das Fenster Devices




Hier werden Licht-Geräte für diesen Patcher angelegt und Gruppen zugeordnet. Zuerst kann man eine oder mehrere Gruppen mit dem Button **Add Group** anlegen. Alle angelegten Gruppen werden oben aufgelistet. „(none)“ ist ein Platzhalter für „keine Gruppe“. Mit **Remove Group** kann eine Gruppe wieder gelöscht werden. Direkt darunter werden alle angelegten Ausgabegeräte (Scheinwerfer usw.) angezeigt. Um ein Ausgabegerät anzulegen, wählt man dessen Typ in der Auswahlliste bei **Device-type** aus und gibt die Adresse bei **Address** an. Ist der Haken bei **Auto-Address** gesetzt, wird für das nächste Gerät automatisch die nächste freie Adresse eingetragen. Ein Klick auf den Button **Add Device** legt das Gerät an und registriert es am Patcher. Wurde davor oben eine Gruppe ausgewählt, wird es auch dieser Gruppe zugeordnet. Mit dem Button **Remove Device From Group** kann man ein Gerät aus einer Gruppe löschen, es bleibt aber dem Patcher erhalten. Mit **Remove Device** wird ein Gerät vollständig gelöscht.

# III. Licht steuern

## Methoden aufrufen – Messaging

Wenn man seine Umgebung bis hier hin korrekt eingerichtet hat, kann man die Methoden der Geräteinstanzen direkt über den Patcher aufrufen. Direkt aus „sclang“ heraus geht das mit der Methode `message` der Patcher-Instanz. Diese erwartet als einziges Argument ein Objekt vom Typ `Event`. Das ist im wesentlichen eine ungeordnete Sammlung von Schlüssel-Wert-Paaren, bei der die Schlüssel immer Symbole sind, die Werte jedoch jedes Art von Objekt sein können. Um eine Methode aufzurufen, muss das `Event` zumindest zwei Schlüssel enthalten: `\method`, welches

den Namen der aufzurufenden Methode (als Symbol) enthalten muss und `\data`, ein `Array` der Daten, die der Methode übergeben werden müssen. Ein sehr einfacher Aufruf kann so aussehen:

```
p.message((method: \dim, data: [1]));
```

Dieser Befehl ruft für alle `Devices`, für die die Methode `\dim` implementiert wurde, dieselbe auf und übergibt den Wert 1. In aller Regel sollte das sämtliche (einfachen) Scheinwerfer auf voller Leistung leuchten lassen.

Optional sind noch weitere Schlüssel möglich:

- `\device`: Ein Index als Ganzzahl oder mehrere Indizes als Ganzzahlarray. Begrenzt Methodenaufrufe auf Geräte mit dem/den angegebenen Index/Indizes (beziehend auf die Reihenfolge der Registrierung der Devices, da diese in einer geordneten Liste verwaltet werden). Kann mit `\group` kombiniert werden.
- `\group`: Ein Symbol mit einem Gruppenname. Begrenzt Methodenaufrufe auf Geräte, die dieser Gruppe zugeordnet sind. Kann mit `\device` kombiniert werden.

Ein komplexeres Beispiel:

```
p.message(
       (method: \color,
        data: [0, 0, 1],
        group: \ring,
        device: (0,2..10) )
);
```

Dieses Beispiel ruft die Methode `\color` auf alle betreffenden Geräte in der Gruppe `\ring` mit einem geraden Index zwischen 0 und 10 und den Daten `[0, 0, 1]` auf – es sollte also jeder zweite Scheinwerfer blau leuchten.

Bei einer Kombination des `\group`- und `\device`-Schlüssels, wie hier zu sehen, bezieht sich die Indexnummer auf lediglich die Geräte in dieser speziellen Gruppe, da die interne Geräteliste zuerst nach dieser eingeschränkt wird.

## Pattern-System

Events können in SuperCollider auch direkt mit der Methode `play` abgespielt werden. Dafür muss ein Event-Typ festgelegt worden sein, der definiert, mit welcher Aktion ein bestimmtes Event auf die `play`-Methode reagiert[13]. In der `Patcher`-Klasse wurde so ein Event-Typ mit der Bezeichnung „light“ angelegt. Ein Event wie das oben beschriebene muss also nicht direkt an den Patcher übergeben werden, sondern es kann auch die Methode `play` aufgerufen werden, wenn der Typ `\light` festgelegt wurde.

Events werden in SuperCollider sehr häufig in Verbindung mit der Pattern-Bibliothek benutzt. Mit dieser lassen sich auf einfache Art und Weise komplexe Patterns erstellen und ausführen. Beim Abspielen eines Patterns erzeugt SuperCollider für jeden Schritt ein Event.

[13] Event – addEventType, 30.9.2012, unter: http://doc.sccode.org/Classes/Event.html#*addEventType

Wenn man nun also in in einem Pattern den Typ `\light` angibt, wird eine solche Nachricht an einen Patcher gesendet. Wird kein Patcher mittels des Schlüssels `\patcher` und der Patcher-ID angegeben, wird die Nachricht an den Standard-Patcher geschickt. Die restlichen Schlüssel entsprechen dem `Event`-Objekt oben. Das folgende Beispiel wechselt die Farbe jedes Scheinwerfers, der am Patcher `\stage` registriert und der Gruppe `\ring` zugeordnet ist, 10 mal für jeweils eine Sekunde von Rot auf Grün und zurück:

```
Pbind(
        \type, \light,
        \patcher, \stage,
        \group, \ring,
        \method, \color,
        \data, Pseq([ [1, 0, 0], [0, 0, 1] ], 10),
        \dur, 1
).play;
```

## Lichtsteuerung vom Klangsyntheseserver „scsynth“

Steuerdaten, die auf dem Klangsyntheseserver „scsynth“ erzeugt werden, müssen anders behandelt werden. Da „scsynth“ eigentlich ein eigenes Programm getrennt von „sclang“ darstellt, stehen die Daten, die auf dem Server berechnet werden, nicht direkt in der Language zur Verfügung. Der Patcher übernimmt hier die Schnittstellenverwaltung und sorgt dafür, dass ein entsprechender Ausspielweg zur Verfügung gestellt wird.

Dazu reserviert der Patcher für jede Methode eines jeden Devices sogenannte Controlrate-Busse, genauer je einen Mehr-Kanal-Bus pro Methode mit genau soviel Kanälen wie die Methode Argumente empfängt. Diese Busse funktionieren wie Busse für Audiodaten, arbeiten aber mit einer geringeren Sample-Rate, um Rechenzeit zu sparen. Für den Anwendungszweck der Lichtsteuerung reicht das aber völlig aus.

Diese Busse können nun von verschiedenen Signalgeneratoren ganz normal „bespielt“ werden. Der Patcher greift die Daten der Busse regelmäßig ab, führt die jeweiligen Methoden der angesprochenen Geräte aus und schreibt die generierten Steuerdaten in dem DMX-Puffer. Das alles passiert im wesentlichen transparent für den Anwender. Um die jeweiligen Instanzen der Busse zu erhalten, bietet der Patcher verschiedene Methoden an:

- `busesForMethod`: Liefert eine Liste mit Bussen für Methoden von allen registrierten Devices, die auf die jeweilige Methode reagieren. Benötigt als Argument die gewünschte Methode als Symbol.
- `busesForGroupMethod`: Liefert eine Liste mit Bussen für Methoden von allen einer bestimmten Gruppe zugeordneten Devices, die auf die jeweilige Methode reagieren. Benötigt als Argumente den Namen der Gruppe und der Methode jeweils als Symbol.

Diese beiden Methoden sind hilfreich beim Schreiben von SynthDef-Code, also Code, der zum Server geschickt wird und dort die Signalverarbeitung ansteuert. Um beispielsweise alle Scheinwerfer die die Methode `\color` kennen rot leuchten zu lassen, kann man folgenden Code ausführen:

```
SynthDef(\test, {
	var color = [1, 0, 0]; // die Methode \color empfängt rgb-werte
	p.busesForMethod(\color).do({ |bus|
		Out.kr(bus, color);
	});
	0; // Funktion muss Zahlenwert zurückgeben
}).play;
```

Hier wird zuerst ein Signal erzeugt, das, wenn man es der Methode `\color` übergibt, der Farbe Rot entspricht. Es enthält drei Werte, beinhaltet also quasi drei Kanäle. Dieses Signal wird dann auf jeden Bus (der ja auch bereits drei Kanäle vorhält) einzeln ausgegeben.

Um dieses rote Licht pulsieren zu lassen, genügt nun schon folgende modifizierte Version:

```
SynthDef(\test,{
	// das „Farbsignal“ wird mit einer langsamen schwingenden Sinuswelle moduliert
	var color = [1, 0, 0] * SinOsc.kr(1/3, mul: 0.5, add: 0.5);
	p.busesForMethod(\color).do({ |bus|
		Out.kr(bus, color);
	});
}).play;
```

Und dank SuperColliders „Multichannel-Expansion“ kann man nun recht einfach eine etwas komplexere Farbmischung betreiben:

```
SynthDef(\test, {
	var color = SinOsc.kr({0.4.rand}!3 + 0.1, pi.rand, mul: 0.5, add: 0.5);
	p.busesForMethod(\color).do({ |bus, i|
		Out.kr(bus, (color - (0.02 * i)).fold(0, 1));
	});
}).play;
```

# Lighting UGens

Um auf komplexe aber oft benutzte Signalverarbeitungsalgorithmen einfach zugreifen zu können, wurden einige Pseudo-UGens geschaffen, die speziell für die Lichtsteuerung zu verwenden sind.

UGens („Unit Generators“) sind in SuperCollider einzelne kleinere Signalverarbeitungs-Objekte innerhalb des Klangsyntheseservers „scsynth“. In aller Regel wird versucht mit einem UGen eine spezielle Aufgabe zu erledigen, also die Komplexität gering zu halten. Per SuperCollider-Language können dann umfangreiche Signalverarbeitungswege aus diesen „einfachsten Teilchen“ zusammengeschaltet werden, um schließlich komplexe DSP-Prozesse herstellen zu können. Statt wie eigentlich vorgesehen UGens in C zu implementieren ist es auch möglich, sogenannte Pseudo-UGens zu schreiben. Diese sind nur in der SuperCollider-Language implementiert. Das sind einfach Klassen, die sich gleich wie „echte“ UGens verhalten, aber eigentlich schon vorhandene UGens kombinieren um eine komplexere Funktionalität bereit zu stellen. Das hat den Vorteil, dass DSP-Code in ein leicht wiederverwendbares Format gebracht werden kann. Die UGens nicht in C implementieren zu müssen, spart dabei Zeit und Mühen und beschleunigt damit den Entwicklungsprozess, allerdings auf Kosten der benötigten Rechenleistung.

Solche UGens reagieren meist auf die Methoden `ar` und `kr`, die anzeigen, dass ein Audio- oder Kontrolldatenstrom verarbeitet werden soll. Da für die Lichtsteuerung generell nur Kontrolldatenströme verwendet werden sollen, wurde jeweils nur die Methode `kr` implementiert.

Die implementierten UGens sind:

- `Hsv2rgb.kr(h, s, v)`: Wandelt HSV-Farbwerte in RGB-Werte um.
  - `h` – „Hue", also Färbung oder auch „Farbwinkel" zwischen 0 und 1
  - `s` – „Saturation", also Sättigung des Farbwertes
  - `v` – „Value", also Helligkeit. Gibt Array mit drei Werten für Rot, Grün und Blau zurück

  **Ausgabe:** Ein Array mit den drei Werten für den Rot-, Grün- und Blau-Kanal.

- `MultiPanAz.kr(channels, in, pos, width, ori)`: Angelehnt an das SuperCollider-UGen `PanAz`. Verschiebt ein Lichtobjekt in einem Mehrkanal-Panorama bestehend aus 2 oder mehr Scheinwerfern, die in einer Linie (oder Ring) mit gleichem Abstand zueinander positioniert sind. Das Signal der Lichtquelle kann dabei mehrere Kanäle enthalten, beispielsweise ein RGB-Farbsignal. Zu beachten ist, dass bei Licht keine „Phantom-Lichtquellen" zwischen den Scheinwerfern entstehen wie es bei Audio der Fall wäre. Deswegen ist die kleinste erreichbare „Auflösung" die der positionierten Scheinwerfer.
  - `channels` – Die Anzahl an im Panorama vorhandenen Kanälen
  - `in` – Das Eingangssignal, kann mehrere Kanäle enthalten
  - `pos` – Die Position, an welche die Lichtquelle verschoben werden soll, zwischen 0 und 2, wobei 2 wieder der Ausgangsposition (0) entspricht
  - `width` – Breite des über mehrere Scheinwerfer verteilte Quellsignals. Aus eben genannten Gründen macht eine Breite von weniger als 2 nicht viel Sinn, größere Werte verteilen eine Lichtquelle auf ebenso viele nebeneinander liegende Scheinwerfer
  - `ori` – Orientierung des Panoramas. 0, wenn erster Scheinwerfer vorne mittig aufgebaut ist, -0.5 wenn er um einen halben Scheinwerferabstand nach links versetzt ist, 0.5 bei Versetzung nach Rechts usw.

  **Ausgabe:** Ein Array mit den Steuersignalen für jeden Scheinwerfer des Panoramas. Wenn das Eingangssignal aus mehreren Kanälen besteht, werden für jeden Scheinwerfer des Panoramas ebenso viele Kanäle ausgegeben. Das Ausgabearray bleibt dabei aber eindimensional mit allen Kanäle hintereinander

- `Rotator.kr(channels, group, in, rotation, width, lag)`: Ähnlich wie bei `MultiPanAz` geht es darum, Licht in einem Mehrkanaligen Panorama zu positionieren. Bei `Rotator` wird allerdings nicht nur eine einzelne Lichtquelle positioniert, sondern es kann ein ganzes schon vorhandenes Panorama verschoben bzw. rotiert werden. Es ist vergleichbar mit `SplayAz`, bei dem ein sogenanntes „Soundfield" rotiert wird.
  - `channels` – Die Anzahl der im Panorama verhandenen Kanäle
  - `group` – Die Anzahl der existierenden Kanäle, die jeweils zu einer „Lichtquelle" zusammengefasst werden (bspw. drei bei einem RGB-Farbsignal)
  - `in` – Das Eingangssignal, muss insgesamt aus `channels * group` Kanälen bestehen
  - `rotation` – Der Rotationswinkel zwischen 0 und 2, wobei 2 wieder der Ausgangsposition entspricht
  - `width` – Die Breite eines jeden rotierten Signals – damit kann das sich ergebende „Lichtfeld" geglättet werden. Das ist wichtig, da sonst bei Bewegungen (Modulation

des Parameters `rotation`) ein Pulsieren entsteht: Wird eine Lichtquelle, die sich nur über einen Scheinwerfer ausdehnt, zwischen zwei Scheinwerfern positioniert, leuchten beide gleichzeitig aber mit reduzierter Leuchtkraft – anders als bei Audio werden diese aber nicht in der Rezeption zum Originalsignal zusammengefasst
  - `lag` – Eine Glättung aller erzeugten Lichtsignale über die Zeit, ebenfalls um den gerade erwähnten Effekt auszugleichen

  **Ausgabe:** Ein Array mit den Steuersignalen für jeden Scheinwerfer des Panoramas, siehe `MultiPanAz`.

- `Mirror.kr(channels, groups, in, wet)`: „Spiegelt" eine Reihen von Signalen um ihren Mittelpunkt. Angewandt auf linear positionierte Scheinwerfer erscheint ein Lichtobjekt, das ursprünglich ganz links positioniert war, ganz rechts und so weiter. Funktioniert auch gut mit ringförmig positionierten Scheinwerfern.
  - `channels` – Die Anzahl der verwendeten Kanäle
  - `groups` – Die Anzahl der Kanäle, die jeweils zusammengefasst werden
  - `in` – Das Eingangssignal, sollte aus channels * groups Kanälen bestehen
  - `wet` – Der Anteil des gespiegelten Signals am erzeugten Signal, 0 bedeutet keine Spiegelung (nur Original-Signal), 1 bedeutet nur Spiegelung (kein Original-Signal)

  **Ausgabe:** Ein Array mit den Steuersignalen für jeden Scheinwerfer des Panoramas, siehe `MultiPanAz`.

- `Blitzen.kr(channels, groups, rate, pos, width, in, ori)`: Eigentlich ein Effektgerät. Erzeugt ein „Blitzlichtgewitter" über mehrere Ausgabekanäle, welches ähnlich wie bei `MultiPanAz` auf einen bestimmten Bereich der Ausgabekanäle beschränkt werden kann. Die Blitze werden dabei als zufällig verteilte Impulse erzeugt, deren Häufigkeit jedoch beeinflusst werden kann. Ihre Verteilung im Panorama wird gesteuert über eine Sinusartige Verteilungsglocke. Damit hat sie in der Mitte ein Maximum und nimmt zu den Seiten hin ab.
  - `channels` – Die Anzahl erzeugter Kanäle
  - `groups` – Die Anzahl der Signale, die pro Kanal zusammengefasst werden
  - `rate` – Die Rate, also Häufigkeit der Blitz-Ereignisse. Im wesentlichen die Frequenz in Hz eines `Dust`-Objekts, welches die Blitze auslöst. Nimmt zu den Seiten hin ab
  - `pos` – Die Position des Zentrums verteilt auf die (linear anzuordnenden) Kanäle von 0 bis 2, siehe auch `MultiPanAz`
  - `width` – Die Breite des Effekts von 0 bis zur Anzahl der erzeugten Kanäle
  - `in` – (Optional) Ein Eingangssignal. Wenn vorhanden wird dieses durch die Blitze moduliert, sonst wird intern ein Signal erzeugt
  - `ori` – Die Orientierung der existierenden Kanäle, siehe auch `MultiPanAz`

  **Ausgabe:** Ein Array mit den Steuersignalen für jeden Scheinwerfer des Panoramas, siehe `MultiPanAz`.

- `Pan2d.kr(gridx, gridy, in, posx, posy, width)`: Positioniert ein Lichtobjekt auf einem zweidimensionalen Raster gleichmäßig angeordneter Scheinwerfer.

- `gridx` – die Anzahl der Bildpunkte in x-Richtung
- `gridy` – die Anzahl der Bildpunkte in y-Richtung
- `in` – Das zu positionierende Lichtobjekt, kann aus einem oder mehreren Kanälen bestehen
- `posx` – Die Position in x-Richtung zwischen 0 und 1
- `posy` – Die Position in y-Richtung zwischen 0 und 1
- `width` – Die Ausdehung des Lichtobjektes zwischen 0 (sehr geringe Ausdehnung, beinah nur ein Bildpunkt) und 1 (Ausdehnung über die ganze Rasterebene).

**Ausgabe:** Ein Array mit den Kanälen für jeden Bildpunkt des Rasters nacheinander von links nach rechts und von oben nach unten. Ist das Eingabesignal ein Array (also aus mehreren Kanälen bestehend), enthält auch jedes ausgegebene Bildpunkt-Signal ein Array derselben Größe.

# NodeProxys und ProxyChains

In SuperCollider gibt es viele Möglichkeiten, Signalverarbeitungsprozesse auf dem Server zu verwalten. Im einfachsten Fall wird über eine Funktion in „sclang“ ein sogenannter Node auf dem Server erzeugt und zur weiteren Steuerung in der Language ein Objekt der Synth-Klasse erzeugt, über das auf den Node zugegriffen werden kann. Komfortabler sind allerdings beispielsweise NodeProxy-Objekte. Ein NodeProxy repräsentiert einen signalverarbeitenden Prozess auf dem Server und bietet noch einige bequeme Methoden zur Steuerung dieses Prozesses an. Beispielsweise kann die Funktion, die den Signalfluss definiert, zur Laufzeit ausgetauscht werden. Es können auch mehrere solcher Funktionen flexibel hintereinander geschaltet und, wenn Audiosignale erzeugt werden, diese automatisch auf einen Audioausgabe-Bus geschickt werden. Mehrere solcher NodeProxys können ebenfalls miteinander „verschaltet“ werden, um komplexe Signalwege zu erzeugen, ähnlich wie bei einer Kette von Effektgeräten im Audio-Bereich.

## Die Klasse NodeProxy

Das letzte Beispiel von „Lichtsteuerung vom Klangsyntheseserver ‚scsynth‘“ (S. 29) kann folgendermaßen mit einem NodeProxy verwendet werden:

```
// p enthält eine Referenz auf einen Patcher
n = NodeProxy.control().source({
        var color = SinOsc.kr({0.4.rand}!3 + 0.1, pi.rand, mul: 0.5, add: 0.5);
        p.busesForMethod(\color).do({ |bus, i|
                Out.kr(bus, (color - (0.02 * i)).fold(0, 1));
        });
});
```

Hier wird der NodeProxy erzeugt und als Signalverarbeitungsprozess eine Funktion übergeben, die der Funktion von oben entspricht. Die Methode `play` muss hier nicht verwendet werden, da diese nur nötig ist, wenn Signale über ein Interface ausgegeben werden sollen (was in diesem Fall aber durch das UGen `Out.kr` geschieht).

Um nun beispielsweise statt einer Sinus- eine Sägezahn-Wellenform zu verwenden, kann einfach die Funktion ausgetauscht werden:

```
n.source = {
        var color = LFSaw.kr({0.4.rand}!3 + 0.1, 2.0.rand, mul: 0.5, add: 0.5);
        p.busesForMethod(\color).do({ |bus, i|
                Out.kr(bus, (color - (0.02 * i)).fold(0, 1));
        });
};
```

## Die Klasse ProxyChain

Um die Verwaltung mehrerer solcher NodeProxys, die nacheinander ein Signal verarbeiten sollen, zu vereinfachen, wurde die Klasse ProxyChain implementiert. Im wesentlichen übernimmt ein ProxyChain-Objekt die Verwaltung des Signalflusses der registrierten NodeProxy-Objekte untereinander und erlaubt ein dynamisches Hinzufügen und Wegnehmen einzelner NodeProxies sowie ein Verändern der Reihenfolge. Ein ProxyChain-Objekt legt man an mit der `new`-Methode:

```
c = ProxyChain.new();
```

Nun kann man einen oder mehrere NodeProxys mit der Methode `add` hinzufügen. Um die Verwaltung zu vereinfachen, gibt man jedem NodeProxy einen Namen.

```
// der erste Proxy erzeugt eine Sinuswelle mit der Frequenz 0.25 Hz und Werten zwischen 0 und 1
var np = NodeProxy.control().source = { SinOsc.kr(1/4, 0, 0.5, 0.5) };
// der zweite Proxy gibt ein Signal, das er bekommt (\in.kr) auf dem ersten dim-bus des Patchers
//     in p aus.
var out = NodeProxy.control().source = { Out.kr(p.busesForMethod(\dim)[0], \in.kr) };
c.add(\quelle, np);
c.add(\ausgabe, out);
```

Das genügt schon, um ein Signal zu erzeugen und an den Patcher auszugeben. Die Verknüpfung der beiden NodeProxys (die Weitergabe des Signals von `sp` an `out`) wird automatisch von der ProxyChain übernommen. Den aktuellen Zustand der ProxyChain kann man mit der Methode `printChain` anzeigen lassen:

```
c.printChain; // erzeugt folgende Ausgabe:

  Node 0 (\quelle): NodeProxy.control(localhost, 1)
  Node 1 (\ausgabe): NodeProxy.control(localhost, 1)
  a ProxyChain
```

Auf die einzelnen Nodes kann man nun über die Methode `at` zugreifen. Diese erwartet entweder den Namen des Nodes (als Symbol) oder den Index (als Ganzzahl).

```
c.at(\quelle); // oder
c.at(0); // oder in Array-Schreibweise
c[\ausgabe];
```

Und genauso kann man auch die Quell-Funktion eines Nodes austauschen:

```
c[\quelle].source = { LFSaw.kr(1/4, 0, 0.5, 0.5) };
```

Wenn man jetzt einen weiteren Node hinzufügt, wird dieser jedoch an das Ende der Signalkette hinzugefügt. Das kann man verhindern, indem man den Ausgabenode mit der Methode `stickToBottom` an der letzten Stelle fixiert.

```
c.stickToBottom(\ausgabe);
```

Da man in aller Regel zumindest einen Node braucht, der generierte Daten in den Patcher ausspielt, kann man sich diesen automatisch erzeugen lassen. Dazu verwendet man die Methode `addPlayer`:

```
var patcher = Patcher.default; // a Patcher
var group = nil; // a group of devices on the patcher, can be nil or Symbol of groupname
var method = \dim; // the method to react to
var channels = 1; // the number of channels to output (usually the number of devices)
var chansPerMethod; // the number of channels the method reacts to
c.addPlayer(patcher, group, method, channels, chansPerMethod)
```

Damit wird an der letzten Stelle in der ProxyChain ein solcher Ausspiel-Node angelegt und „fixiert". Dieser lässt sich danach aber wie einen „normal" angelegten Node behandeln.

Weitere Methoden sind:

- `addAfter(index, name, nodeProxy)` – Fügt einen NodeProxy nach einem anderen ein. **index** – der Index, nach dem der Proxy eingefügt werden soll; **name, nodeProxy** – wie oben.
- `addBefore(index, name, nodeProxy)` – Fügt einen NodeProxy vor einem anderen ein, Argumente siehe `addAfter`.
- `addFirst(name, nodeProxy)` – Fügt einen NodeProxy an erster Stelle ein. Argumente siehe oben (`add`).
- `addLast(name, nodeProxy)` – Fügt einen NodeProxy an erster Stelle ein. Argumente siehe oben (`add`).
- `clear(sure)` – Leert die ganze ProxyChain, löscht und stoppt alle NodeProxys. Muss bestätigt werden, indem als Argument `sure` der Wert `true` gesetzt wird.
- `moveDown(index)` – Bewegt einen NodeProxy um eine Stelle nach hinten. **index** – Ganzzahliger Index des zu bewegenden NodeProxys.
- `moveUp(index)` – Bewegt einen NodeProxy um eine Stelle nach vorne. **index** – Ganzzahliger Index des zu bewegenden NodeProxys.
- `remove(index)` – Entfernt und stoppt einen NodeProxy aus der ProxyChain. **index** – Ganzzahliger Index des zu löschenden NodeProxys.
- `stickToTop(index)` – Fixiert den mit **index** (Ganzzahl) definierten Proxy an der ersten Stelle.
- `unStickFromBottom()` – Löst die Fixierung des letzten NodeProxys der ProxyChain.
- `unStickFromTop()` – Löst die Fixierung des ersten NodeProxys der ProxyChain.