Python-Programmierung für Nuke

Compositor stoßen täglich auf wiederkehrende Arbeitsabläufe. Mit Python-Skripten für Nuke lassen sich für diese viel Arbeit und Zeit einsparen. Das auf den ersten Blick als Mehraufwand erscheinende Erstellen von Skripten stellt sich meist als sinnvoll heraus. Denn einmal erstellt, lassen sich Skripte schnell auf andere Situationen umschreiben und erweitern, um somit den Workflow zu beschleunigen. Daher ist es für jeden Artist von Vorteil, Skripting-Erfahrung zu haben.

von Simon Jokuschies

Powered by

SCIA INSTITUTE

Sind die Konzepte und Funktionsweisen erst einmal verstanden, kann Programmieren viel Spaß machen. Dieser Workshop bietet einen Einstieg in die Python-Programmierung für Nuke und zeigt, wie die Erweiterung des Compositingprogramms durch eigene Pytho-Skripte möglich ist. Der erste Teil zeigt essenzielle Programmierelemente in Python, mit denen Nuke im zweiten Teil erweitert wird. Hierbei entsteht ein Skript, welches die Versionierung von Nuke-Skripten regelt und den Write Node um nützliche Funktionen erweitert.

Alle Skripte gibt es zum Download in kommentierter Version inklusive Tutorial auf **www.digitalproduction.com**.

Vorneweg: Python

Zunächst ein paar Worte zu Python. Bei Python handelt es sich um eine Programmiersprache, die relativ simpel zu erlernen und sehr effektiv und flexibel einsetzbar ist. Sie wurde speziell dafür entwickelt, möglichst einfach und übersichtlich zu sein.

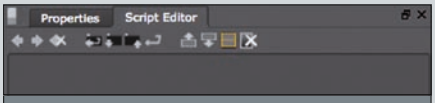
Dies macht das Erlernen angenehm und erzeugt eine steile Lernkurve. Python ist in vielen Programmen, wie etwa Nuke, Hiero, Maya, Houdini, Cinema 4D und Blender, integriert. Diese Programme lassen sich durch Python-Skripte vielfältig individuell anpassen und erweitern, wodurch sie flexibel werden. In Nuke lassen sich mithilfe von Python unter anderem Menü-Strukturen verändern und erweitern, Gizmos integrieren, eigene Panels erstellen sowie Nodes beeinflussen.

Im Laufe der letzten Jahre hat sich eine große Community gebildet, die sich mit der Erweiterung von Nuke beschäftigt.

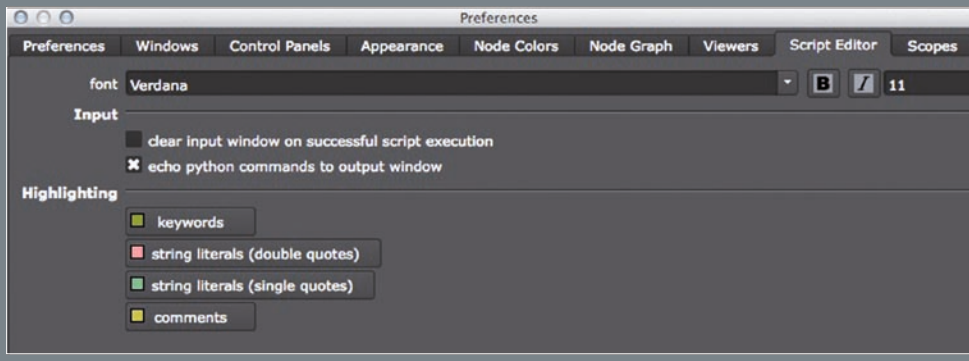
Nukepedia (**www.nukepedia.com**) ist die größte Online-Plattform, auf der Artists ihre selbst geschriebenen Skripte und Erweiterungen rund um Nuke veröffentlichen.

Der Script Editor in Nuke

Nuke liefert einen eigenen Editor zum Erstellen von Python-Skripten. Beim Wählen des Elements „Script Editor" im „Content Menu" integriert sich dieser ins Layout. Er enthält die wichtigsten Befehle zum Schreiben von Skripten und ist für die Erstellung kleiner Funktionen vollkommen ausreichend. Mit zunehmendem Umfang der Skripte ist jedoch ein Wechsel zu einem eigenständigen Edi-



Der in Nuke enthaltene Script Editor liefert die wichtigsten Befehle zum Erstellen von Python-Skripten.



Mit dem Abwählen der oberen Checkbox wird verhindert, dass ausgeführter Quelltext verschwindet.

tor wie beispielsweise Sublime, PyCharme, Notepad++, Textwrangler oder Komodo zu empfehlen. Diese bieten mehrere Vorteile: So enthalten Skript-Editoren beispielsweise eine bessere Navigation, Zeilennummern, Suchen-und-Ersetzen-Funktionen, automatische Wortvervollständigun und vieles mehr. Ein weiterer Vorteil eines Skript-Editors ist, dass im Falle eines Absturzes von Nuke der Quelltext im internen Script Editor verloren gehen würde. Befindet er sich im eigenständigen Editor, so bleibt er erhalten.

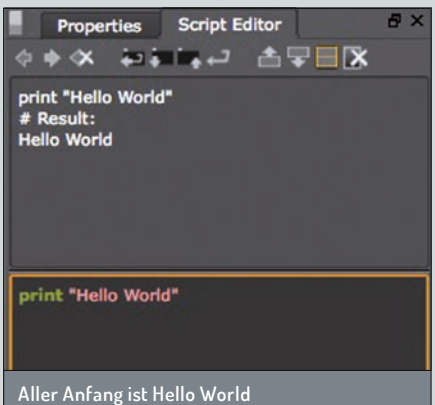
Der Script Editor von Nuke besteht aus zwei Elementen. Im unteren Bereich befindet sich das sogenannte „Input Window" für die Eingabe von Befehlen. Diese lassen sich mit Strg + Enter (Win) respektive Cmd + Enter (Mac) ausführen. In der oberen Hälfte befindet sich das „Output Window" zur Ausgabe der Befehle. Um zu verhindern, dass der ausgeführte Quelltext aus dem Input Window verschwindet, lässt sich in Nukes Einstellungen im Tab „Script Editor" die Checkbox „clear input window on successful script execution" abwählen.

Essenzielle Programmierelemente in Python

Im Folgenden werden grundlegende Elemente vorgestellt, die als Basis für das Skript im zweiten Teil dienen. Da Python auf Zeilenumbrüche achtet, ist jeder mit einem Umbruchsymbol (↩) markiert – das dreispaltige Layout verhindert lange Zeilen. Die wohl bekannteste Aufgabe in jeder Programmiersprache ist eine Ausgabe mit dem Inhalt „Hello World". Hierfür tippt man in das Input Window folgende Zeile und führt den Code aus:

```
print "Hello World"
```

Der Befehl print ist eine in Python integrierte Funktion, die eine Ausgabe im Output Window erzeugt. Um einen Text ausgeben zu lassen, muss dieser in Anführungszeichen stehen.



Aller Anfang ist Hello World

Um ein Skript für andere Nutzer verständlich zu halten, ist es ratsam, den Quelltext mit Kommentaren zu versehen. Diese werden beim Ausführen des Programms nicht beachtet. Einzeilige Kommentare beginnen mit einem Raute-Zeichen, wohingegen mehrzeilige durch drei einfache Anführungszeichen umschlossen werden.

```
# this is a comment ↩
↩
'''↩
this is another comment ↩
but now over ↩
several lines ↩
'''↩
```

Ein wichtiges Element in der Programmierung stellt die Speicherung von Daten in Variablen dar. Diese können unterschiedliche Datentypen annehmen. Sogenannte Strings stellen Zeichenketten, also einen Text dar, während sogenannte Integer ganze Zahlen abbilden. Variablen lassen sich befüllen, ausgeben und miteinander verarbeiten.

Beim Befüllen einer Variable steht auf der linken Seite der Name der Variable, gefolgt von einem Gleichheitszeichen, und rechts davon der Inhalt, welcher der Variable zugewiesen wird.

```
b = 7 ↩
c = 2 ↩
print b + c ↩
```

In diesem Beispiel bekommt die Variable b den Wert 7, die Variable c den Wert 2 zugewiesen. Daraufhin wird die Summe der beiden Variablenwerte ausgegeben, sodass beim Ausführen im Output Window die Zahl 9 erscheint. Es lassen sich auch mehrere Werte in einer sogenannten list, einer Art Container, speichern.

```
colors = ["red", "green", "blue"]↩
colors.append("yellow") ↩
del colors[0] ↩
print colors ↩
```

Die erste Zeile erzeugt eine list mit dem Namen „colors", welche die Werte "red", "green" und "blue" erhält. Der Befehl append() fügt der list das Element "yellow" hinzu. Als Nächstes löscht die Funktion del den Datensatz der list mit dem Index 0. Der erste Eintrag einer list ist stets 0, daher ist das zu löschende Element "red". In der vierten Zeile wird die list ausgegeben. Im Output Window erscheint daher ["green", "blue", "yellow"]. In einer list können auch andere Datentypen wie beispielsweise Zahlen gespeichert werden. Es existieren noch einige weitere Datentypen.

Um Variablen innerhalb einer Zeichenkette auszugeben, muss folgende Besonderheit beachtet werden. Dazu ein kleines Beispiel:

```
language = "python"↩
wie = "spitze"↩
print "language ist wie"↩
```

Im Output Window wird nicht wie zu erwarten der Satz python ist spitze ausgegeben, sondern language ist wie. Da die Variablen innerhalb der Anführungszeichen stehen, werden die Variablen beim Ausführen des Skripts nicht mit ihren Variableninhalten ersetzt. Um die Variableninhalte innerhalb einer Zeichenkette anzeigen zu lassen, muss die dritte Zeile ein wenig verändert werden:

```
language = "python"↩
wie = "spitze"↩
print "%s" %language + " ist %s" %wie ↩
```

"%s" steht in der Zeichenkette für einen Platzhalter, der außerhalb der Zeichenkette mit dem Wert der Variablen language ersetzt wird. Bei letzterer Variable erfolgt dies entsprechend. Beide Zeichenketten lassen sich durch ein Pluszeichen miteinander verbinden.

Ein wichtiger Aspekt ist die Fallunterscheidung, mit der Verzweigungsmöglichkeiten entstehen. Eine Bedingung wird geprüft und bei deren Eintreten ein Befehl oder mehrere ausgeführt. Trifft sie nicht ein, wird ein anderer Befehl ausgeführt. In Python sieht das wie folgt aus:

```
a = 15 ↩
b = 10 ↩
```

```
if a == b: ↵
   print “a und b sind identisch“↵
else: ↵
   print “a und b sind unterschiedlich“↵
```

In den ersten zwei Zeilen werden Variablen erstellt und mit Werten gefüllt. Die dritte Zeile prüft, ob der Wert a dem von b entspricht. Wichtig ist hierbei das doppelte Gleichheitszeichen für einen Vergleich, denn ein einzelnes wäre die Befüllung der Variable a mit dem Wert von b. Die Bedingung wird mit einem Doppelpunkt abgeschlossen. Trifft diese zu (was in diesem Beispiel nicht der Fall ist), so wird all das, was eingerückt wurde, ausgeführt.

Die Zeileneinrückung mit der Tabulatortaste oder alternativ mit vier Leerzeichen ist die sogenannte Indentation, eine wichtige Konvention innerhalb Pythons. Viele andere Sprachen wie C, Java oder PHP umschließen entsprechende Anweisungen mit geschweiften Klammern. In Python geschieht dies durch Zeileneinrückung.

Trifft die Bedingung aus Zeile 3 nicht zu (wie es in diesem Beispiel der Fall ist), erfolgt automatisch ein Sprung in die fünfte Zeile. Folglich werden alle Befehle ausgeführt, die eingerückt unter der else-Anweisung stehen. In diesem Beispiel wird also ausgegeben: a und b sind unterschiedlich. Besonders zu Beginn stößt man beim Ausführen der Skripte in Python oft auf die Fehlermeldung „IndentationError“, weil der Quelltext nicht richtig eingerückt wurde. Wäre die vierte Zeile aus dem genannten Beispiel nicht eingerückt, erschiene folglich die Fehlermeldung IndentationError.

Um wiederholt Befehle auszuführen, lassen sich diese in einer sogenannten for-Schleife aufrufen, anstatt sie repetitiv niederzuschreiben zu müssen. Angenommen, die Zahlen 0 bis 9 sollen im Output Window ausgegeben werden, so lässt sich dies wie folgt realisieren:

```
for i in range(0,10): ↵
   print i ↵
```

Mit dem Schlüsselwort „for“ beginnt die Anweisung für eine Schleife. Die Funktion range(0,10) erstellt eine list mit den Werten von 0 bis 9. Die for-Schleife durchläuft nun die list Element für Element. Mit der Variable i lässt sich innerhalb der Schleife auf das jeweilige aktive Element zugreifen. Alle auszuführenden Befehle sind eingerückt.

In diesem Fall wird der Inhalt von i ausgegeben, was im ersten Durchlauf dem Wert 0 entspricht. Danach ist das Ende der for-Schleife erreicht. Es wird auf den Anfang der Schleife zurückgesprungen, das nächste Element der list als aktives Element gesetzt und ausgegeben. Dies wiederholt sich solange, bis die list vollständig durchlaufen ist und die Schleife beendet wird.

In den bisherigen Beispielen wurde auf viele interne Funktionen von Python zurückgegriffen, wie beispielsweise die print-Funktion. Es lassen sich aber auch eigene Funktionen erstellen. Hierbei wird zwischen Funktionen mit und ohne Rückgabewert differenziert. Die Unterschiede lassen sich anhand eines kleinen Beispiels gut verdeutlichen.

```
#function with return statement ↵
def addNumbers(number1, number2): ↵
   result = number1 + number2 ↵
   return result ↵
myNumber = addNumbers(2,4) ↵
print myNumber ↵
```

Die Deklaration einer Funktion entspricht einem Bauplan über alle auszuführenden Befehle. Eine Funktion wird mit dem Schlüsselwort „def“ eingeleitet, gefolgt vom Namen der Funktion und in Klammern mit Werten, die beim Aufrufen der Funktion mitgegeben werden.

Diese sogenannten Parameter sind optional und lassen sich innerhalb der Funktion verarbeiten. Alle auszuführenden Befehle müssen eingerückt sein. In der dritten Zeile wird eine Variable result erstellt, deren Wert der Summe der Parameter number1 und number2 entspricht.

Der Wert wird durch das Schlüsselwort „return“ in Zeile 4 zurückgegeben. Die Funktion wird durch ihre Deklaration als solche nicht ausgeführt. In der fünften Zeile wird nun eine Variable myNumber erstellt, deren Inhalt dem Rückgabewert der Funktion addNumbers() mit den Parametern 2 und 4 entspricht. Die Funktion wird dadurch aufgerufen, die beiden Zahlen werden gemäß des Bauplans addiert und das Ergebnis wird an die Variable myNumber zurückgegeben, welche in der nächsten Zeile ausgegeben wird. Im Output Window erscheint daher die Zahl 6.

Eine Funktion ohne Rückgabewert verläuft etwas anders:

```
#function without return statement ↵
def outputHello(): ↵
     print “hello“ ↵
outputHello() ↵
```

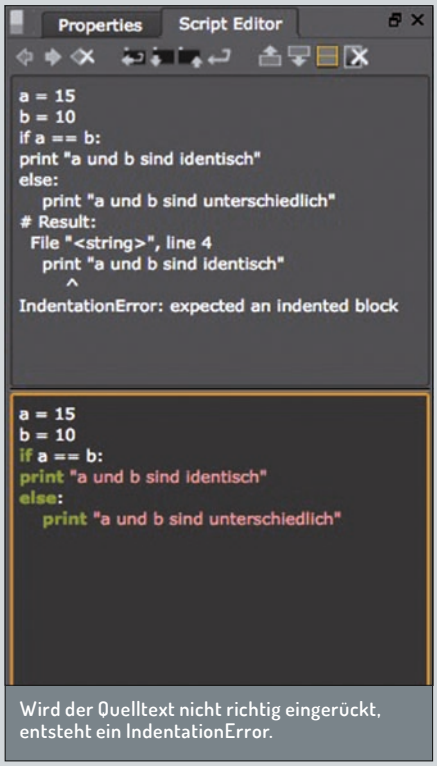
Die Funktion outputHello() wird mit dem Schlüsselwort „def“ eingeleitet. Sie beinhaltet in diesem Beispiel keine zu übergebenden Parameter. Der auszuführende Teil besteht aus nur einem Befehl, nämlich der Ausgabe von „hello“. In der vierten Zeile wird nun die Funktion aufgerufen und dadurch hello im Output Window ausgegeben.

Zusammenfassend wird bei einer Funktion mit Rückgabewert ein Wert zurückgegeben, während bei einer Funktion ohne Rückgabewert lediglich Befehle ausgeführt werden. Python lässt sich durch zusätzliche Funktionen und Elemente, die sich in sogenannten Modulen befinden, erweitern. Das Schlüsselwort hierfür lautet „import“. Python bietet werkseitig beispielsweise keine Ausgabe der aktuellen Uhrzeit. Hierfür lässt sich auf das time-Modul zurückgreifen, welches eine Funktion zum Anzeigen der Uhrzeit bereitstellt.

```
import time ↵
print time.strftime(“%H:%M:%S“) ↵
```

Die erste Zeile importiert das time-Modul, in Zeile 2 wird die im time-Modul enthaltene Funktion strftime() aufgerufen, mit der die ak-

Container: Mit lists lassen sich bequem mehrere Werte speichern.



Wird der Quelltext nicht richtig eingerückt, entsteht ein IndentationError.

tuelle Uhrzeit, in Stunden:Minuten:Sekunden formatiert, ausgegeben wird. Python bietet werkseitig keine Möglichkeit an, Nodes in Nuke zu bearbeiten. Daher wurde speziell das nuke-Modul konzipert.

Import nuke

Das nuke-Modul bietet zahlreiche Nuke-spezifische Funktionen. Beispielsweise lassen sich Nodes mit folgender Funktion direkt über den Script Editor erstellen:

```
nuke.createNode("Blur")
```

Die Funktion createNode() verlangt beim Aufruf als Parameter die Node-Klasse, von der ein Node-Objekt, eine sogenannte Instanz, erstellt werden soll. Der übergebene Parameter Blur erstellt somit einen Blur-Node. Mit dem Befehl selectedNodes() lassen sich alle im Nodegraph aktuell ausgewählten Nodes ermitteln:

```
print nuke.selectedNodes() ↩
```

Um den zuletzt ausgewählten Node ausgeben zu lassen, dient folgender Befehl:

```
print nuke.selectedNode() ↩
```

Die letzten beiden Befehle gaben die Node-Objekte als solche aus. Jeder Node trägt diverse Eigenschaften wie den Namen, die Position im Nodegraph und die Werte jedes einzelnen Knobs mit sich. Um beispielsweise den Namen des zuletzt ausgewählten Objekts ausgeben zu lassen, lässt sich der Befehl name() verwenden:

```
print nuke.selectedNode().name() ↩
```

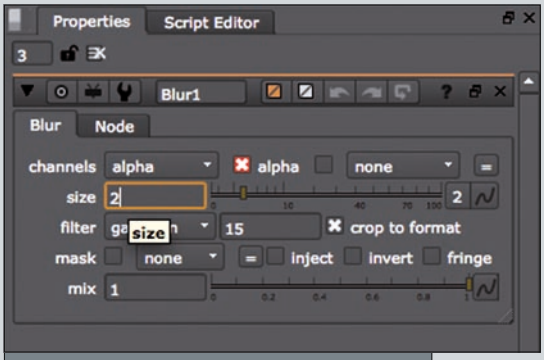
Um einen bestimmten Knob-Value eines Nodes ausgeben zu lassen, lässt sich der Node, gefolgt vom Knob-Namen in eckigen Klammern, und dem Befehl getValue() ansprechen. Die size, also die Stärke der Weichzeichnung eines ausgewählten Blur-Nodes lässt sich daher wie folgt ausgeben:

```
print nuke.selectedNode()["size"].getValue()
↩
```

Mit dem Befehl setValue() lässt sich diese Stärke verändern:

```
nuke.selectedNode()["size"].setValue(65)
↩
```

Mit dem Hovern des Mauszeigers über das gewünschte Eingabefeld jedes Knobs erscheint ein Tooltip. Der fett geschriebene Teil entspricht dabei dem jeweiligen Knob-Namen. Mit den bisherigen Kenntnissen lassen sich bereits nützliche Dinge realisieren.



Das Hovern über einen beliebigen Knob verrät den in fett geschriebenen Knob-Namen.

Aufgabe: Man benötigt alle Pfade aller Read Nodes in einem Skript

```
for node in nuke.allNodes("Read"): ↩
   print node.name() + ": " +
node["file"].getValue() ↩
```

Die Funktion nuke.allNodes() wählt alle Nodes im Nodegraph aus. Mit dem Parameter Read beschränkt sich die Auswahl auf alle Read Nodes. Die Variable node speichert den jeweils aktiven Node der for-Schleife, um ihn aufrufen zu können. Um nun den Pfad des Read Nodes ausgeben zu lassen, lässt sich der Node, gefolgt vom entsprechenden Knob-Namen file, und der Funktion getValue() aufrufen.

Aufgabe: Den Colorspace aller ausgewählten Read Nodes in sRGB ändern

```
for node in nuke.selectedNodes("Read"):
↩
   node["colorspace"].setValue("sRGB") ↩
```

Skripte bei jedem Start von Nuke ausführen

Für die Erweiterung von Nuke existieren zwei wichtige Dateien: die Python-Dateien init.py und menu.py. Sie befinden sich in einem versteckten Ordner, der sogenannten Nuke Home Directory auf der Festplatte unter dem Pfad:

```
users/username/.nuke [mac]
C:\Users\username [win]
/home/username [linux]
```

Der „username" muss durch den entsprechenden Namen des Nutzers ersetzt werden. Alle Befehle, die in der init.py stehen, werden beim Starten von Nuke ausgeführt, bevor Nukes Benutzeroberfläche geladen ist; in der menu.py, nachdem Nukes Benutzeroberfläche geladen ist. Falls diese beiden Dateien in der eigenen Nuke Home Directory noch nicht existieren, können sie manuell erstellt werden. Die menu.py lässt sich nutzen, um Nuke mit eigenen Menüs und Shortcuts zu erweitern, während init.py Plugin-Pfade setzt. Es können auch weitere menu.py und init.py in Unterordnern existieren.

Workshop – nextVersion

Das bisherige Wissen lässt sich nun praktisch anwenden, um ein Skript zu erstellen, welches die Versionierung von Nuke-Skripten vereinfacht und den Write Node erweitert. Compositor erstellen oft mehrere Versionen eines Skripts, welche sie stets manuell aufsetzen müssen.

Mit dem Skript, das den Namen „nextVersion" erhält, lassen sich das aktuelle Nuke-Skript und die ausgewählten Write Nodes automatisch um eine Versionsnummer erhöhen. Zudem wird der Write Node um nützliche Funktionen erweitert. Er erhält Schaltflächen, um den Renderpfad im Explorer anzeigen und eine Versionserhöhung sowie den Renderordner automatisch erstellen zu lassen und zudem beim Rendern ein Backup der jeweiligen Datei zu speichern. Durch Letzteres wird sichergestellt, dass eine Skript-Datei existiert, deren Ausgabe mit dem gerenderten Bild identisch ist.

Das Skript „nextVersion" enthält zwei Python-Dateien: eine nextVersion.py, in die die beschriebene Hauptfunktionalität ausgelagert wird, sowie eine menu.py, um das nextVersion-Skript in Nuke einzubinden.

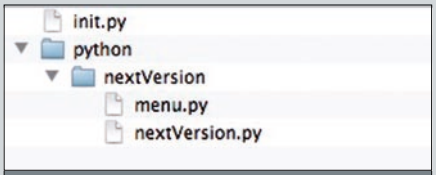
Um eine saubere Ordnerstruktur zu bewahren, wird zunächst in der Nuke Home Directory ein Ordner mit dem Namen „python" erstellt. Darin lassen sich alle selbst geschriebenen Python-Skripte ablegen. In dem Ordner wird ein neuer Ordner mit dem Namen „nextVersion" erstellt, in welchem sich die beiden Dateien menu.py und nextVersion.py befinden. Damit diese beim Start von Nuke geladen werden, muss in der init.py der Nuke Home Directory folgender Befehl hinzugefügt werden:

```
nuke.pluginAddPath("python/nextVersion") ↩
```

Die Struktur der Nuke Home Directory sollte nun der im unteren Bild entsprechen.

Die menu.py

Die menu.py im Ordner nextVersion bindet die Funktionen der Datei nextVersion.py in



Modifiziert: So sollte die angepasste Nuke Home Directory nun aussehen.

Nuke ein. Sie besteht aus folgenden Zeilen:

```
import nuke ↩
import nextVersion ↩

nuke.menu(“Nuke“).addCommand(‚Scripts/
nextVersion‘, ‚nextVersion.nextVersi-
on()‘) ↩
nuke.addOnUserCreate(nextVersion.addCu-
stomElements, nodeClass = “Write“) ↩
nuke.addKnobChanged(nextVersion.perform-
CustomAction, nodeClass = “Write“) ↩
```

Als Erstes wird das Nuke-Modul importiert. Da im Folgenden auf ihre Funktionen zugegriffen wird, muss die Datei nextVersion ebenfalls importiert werden.

Die nächste Zeile erweitert das User-Menü. Das Menü mit dem Namen „Nuke“, welches die obere Menu Bar (File, Edit, Layout, Viewer, Render, etc.) darstellt, erhält ein weiteres Element: Scripts. Dieses erhält den Eintrag „nextVersion“.

Klickt der Nutzer darauf, wird die Funktion nextVersion(), die sich in der gleichnamigen Datei befindet, aufgerufen. Mit der in Nuke bereitgestellten Funktion addOnUserCreate() lassen sich Befehle ausführen, sobald ein neuer Node erstellt wurde. Als Parameter wird die Funktion addCustomElements() mitgegeben, die sich in der Datei nextVersion.py befindet.

Dieser Befehl soll immer dann ausgeführt werden, wenn ein Write Node erstellt wird. Nuke stellt die Funktion addKnobChanged() zur Verfügung. Sobald der Nutzer einen Knob eines Nodes ändert, wird ein Befehl ausgeführt. In diesem Fall soll beim Verändern von Knobs die Funktion performCustomAction() aus der Datei nextVersion.py aufgerufen werden.

Dies soll jedoch nur eintreten, wenn der ausgewählte Node ein Write Node ist. Hiermit wird sichergestellt, dass etwas geschieht, sobald der Nutzer auf die im Folgenden selbst erstellten Buttons klickt. Die Befehle hierfür sind in der Funktion performCustomAction() in der nextVersion.py festgelegt.

Die nextVersion.py

Die Datei nextVersion.py bildet die Hauptdatei des Skripts. Als Erstes werden alle notwendigen Module importiert.

```
import nuke ↩
import nukescripts ↩
import os ↩
import sys ↩
import subprocess ↩
import platform ↩
```

Neben dem nuke-Modul bietet das nukescripts-Modul weiterführende Nuke-spezifische Funktionen. Mit den os-, sys-, subprocess- und platform-Modulen wird unter anderem die Funktionalität hergestellt, Ordner auf dem Betriebssystem öffnen und erstellen zu können sowie deren Existenz zu prüfen. Das Skript verfügt über fünf Funktionen:

- addCustomElements(): Dem Write-Node eigene Schaltflächen hinzufügen
- nextVersionPanel(): Fenster, mit dem die in der Version zu erhöhenden Write Nodes ausgewählt werden können
- saveBackup(): Backup der aktuellen Datei im Renderordner speichern
- performCustomAction(): Befehle für die erstellten Schaltflächen des Write Nodes
- nextVersion(): Aktuelles Skript sowie alle markierten Write Nodes in der Version erhöhen

Im Folgenden werden die einzelnen Funktionen erläutert.

Die Funktion addCustomElements()

```
def addCustomElements(): ↩
↩
   node = nuke.thisNode() ↩
   ↩
   #init custom elements ↩
   tab_next = nuke.Tab_Knob(“next“,
“next“) ↩
   btn_revealInFinder = nuke.PyScript_
Knob(‚reveal in explorer‘, ‚reveal in
explorer‘, ‚,) ↩
   btn_createNextVersionFolder = nuke.
PyScript_Knob(‚create next version‘,
‚create next version‘, ‚,) ↩
   btn_forceCreateDirectory = nuke.
PyScript_Knob(‚force create directory‘,
‚force create directory‘, ‚,) ↩
   chBox_saveBackup = nuke.Boolean_
Knob(‚save backup‘, ‚save backup‘) ↩
↩
   #add custom elements to node ↩
   node.addKnob(tab_next) ↩
   node.addKnob(btn_revealInFinder) ↩
   node.addKnob(btn_createNextVersion-
Folder) ↩
   node.addKnob(btn_forceCreateDirecto-
ry) ↩
   node.addKnob(chBox_saveBackup) ↩
```

Um beim Erstellen den jeweiligen Write Node ansprechen und ihm Elemente hinzufügen zu können, wird dieser in der Variable node gespeichert. Des Weiteren erhält der Write Node vier Zusatzelemente: einen Button, um den Ausgabepfad im Explorer anzuzeigen; einen Button um den Write Node in der Version zu erhöhen; einen Button, um den Renderordner automatisch erstellen zu lassen; und eine Checkbox zur Auswahl, ob beim Rendern ein Backup des aktuellen Nuke-Skripts im Renderordner gespeichert werden soll. Bei allen Initialisierungen stellt der erste Parameter in Klammern den Namen des jeweiligen Elements dar, während der zweite die Aufschrift des Elements bildet. Alle Elemente werden in einem neuen Tab mit dem Namen „next“ angeordnet. Abschließend fügt die Funktion addKnob() alle vier Elemente dem Write Node hinzu.

Die Funktion nextVersionPanel()

```
def nextVersionPanel(): ↩
   ↩
   p = nuke.Panel(‚nextVersion – Select
write nodes to increment version‘) ↩
   p.setWidth(500) ↩
   for node in nuke.allNodes(“Write“): ↩
      p.addBooleanCheckBox(““%s“ % node.
name() , 1) ↩
   return p ↩
```

Die Funktion nuke.Panel() erstellt ein Fenster, dessen Titel als Parameter übergeben wird und eine Breite von 500 Pixeln erhält. Des Weiteren wird im Fenster für jeden Write Node im Skript eine Checkbox mit dem Namen des Write Nodes erstellt und auf den Standardwert 1, also ausgewählt, gesetzt. Die Funktion wird im späteren Verlauf in der Funktion nextVersion() aufgerufen.

Die Funktion saveBackup()

```
def saveBackup(writeName): ↩
   ↩
   scriptLocation = nuke.root().name() ↩
   scriptName = os.path.
basename(scriptLocation) ↩
   fileValue = nuke.toNode(writeName)
[‚file‘].getValue() ↩
   renderPath = os.path.dirname(fileValue)
↩
   ↩
   nuke.scriptSave(renderPath + “/back-
up_“ + scriptName)↩
```

Mit der Funktion saveBackup() wird vor dem Rendervorgang eine Backup-Datei des aktuellen Nuke-Skripts im Renderordner gespeichert. Hierfür werden der Skript-Name sowie der reine Renderpfad benötigt. Mit der Funktion nuke.root().name() lässt sich der komplette Pfad des aktuellen Nuke-Skripts abrufen. Aus diesem lässt sich mittels der Funktion os.path.basename() der Name des Nuke-Skripts extrahieren. Als Nächstes wird der komplette Renderpfad inklusive Bilddatei ausgelesen, mit dessen Hilfe durch die Funktion os.path.dirname() der reine Ordnerpfad ermittelt wird. Durch diese Werte lässt sich nun im Renderordner eine Backup-Datei mit dem Namen „backup_“ und dem Namen des aktuellen Skripts speichern.

Die Funktion performCustomAction()

```
def performCustomAction(): ↩
   ↩
   node = nuke.thisNode() ↩
   knob = nuke.thisKnob() ↩
↩
   fileValue = node[“file“].getValue() ↩
   renderPath = os.path.dirname(fileValue)
↩
```

In der Funktion performCustomAction() werden die Befehle beim Drücken der selbst erstellten Buttons festgelegt. Zuerst speichert die Variable node den aktuell angeklickten Node. Zudem hält die Variable knob die aktuell betätigte Schaltfläche des Nodes fest. Des Weiteren wird der gesamte Renderpfad, der in den Write Node eingegeben wurde, gespeichert. Daraus lässt sich durch os.path.dirname() der reine Ordnerpfad ermitteln. Nun wird über den Namen der gedrückten Schaltfläche geprüft, welche Schaltfläche gedrückt wurde.

```
   #reveal in explorer ↵
   if knob.name() == "reveal in explorer": ↵
      if renderPath!="": ↵
         try: ↵
            if platform.system() == "Windows": ↵

os.startfile(renderPath) ↵
         elif platform.system() == "Darwin": ↵
                  subprocess.Popen(["open", renderPath]) ↵
         else: ↵
                  subprocess.Popen(["xdg-open", renderPath]) ↵
         except: ↵
            nuke.message("couldn't open render path. No such directory") ↵
      else: ↵
         nuke.message("Please make sure to set a render path") ↵
```

Wurde die Schaltflächte „reveal in explorer" gedrückt, wird überprüft, ob ein Renderpfad eingegeben wurde. Falls nicht, erscheint die Meldung, dass kein Renderpfad gesetzt wurde. Andernfalls wird versucht, den Ordner des Renderpfads zu öffnen.

Dies kann auf sichere Art und Weise mit der try-except-Anweisung ausgeführt werden. Es werden alle Befehle ausgeführt, die eingerückt innerhalb der try-Anweisung stehen. Tritt dabei an einer Stelle ein Fehler auf, bricht das Programm nicht ab, sondern springt zu der Anweisung, die eingerückt nach except steht. Da das Öffnen eines Ordners auf jedem Betriebssystem anders verläuft, muss mit der Funktion platform.system() zunächst das Betriebssystem, auf dem das Skript läuft, abgefragt werden.

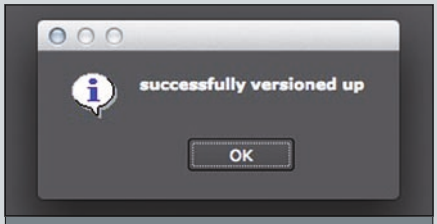
```
   #create next version ↵
   if knob.name() == "create next version": ↵
      if renderPath != "": ↵
         nukescripts.clear_selection_recursive() ↵
         node.setSelected(True) ↵
         nukescripts.version_up() ↵
         node.setSelected(False) ↵
↵
         fileValue = node["file"].getValue() ↵
         renderPath = os.path.dirname(fileValue) ↵
↵
         if not os.path.isdir(renderPath): ↵
            os.makedirs(renderPath) ↵
            nuke.message("successfully versioned up") ↵
         else: ↵
            nuke.message("Renderfolder ‚%s' seems to exist" % renderPath) ↵
      else: ↵
         nuke.message("Please make sure to set a render path") ↵
```

Hat der Nutzer den Button „create next Version" gedrückt, so wird als Erstes geprüft, ob ein Renderpfad existiert. Falls nicht, wird dies als Meldung ausgegeben. Falls ja, so wird der Write Node in der Version erhöht. Dazu darf nur der jeweilige Write Node ausgewählt sein.

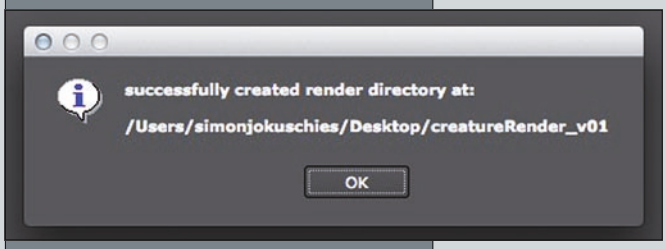
Daher werden zuerst alle ausgewählten Nodes deselektiert, anschließend wird der Write Node ausgewählt, in der Version erhöht und daraufhin wieder abgewählt.

Anschließend wird der reine Ordnerpfad bestimmt und überprüft, ob dieser auf der Festplatte existiert. Falls nicht, wird er erstellt und eine entsprechende Meldung ausgegeben.

Falls er bereits existiert, wird dies in einer Meldung festgehalten.



Der ausgewählte Write Node wird in der Version erhöht und der Renderordner automatisch erstellt.



Der Renderordner wird automatisch erstellt.

```
   #force create directory ↵
   if knob.name() == "force create directory": ↵
      if renderPath != "":↵
         if not os.path.isdir(renderPath): ↵
            os.makedirs(renderPath) ↵
            nuke.message("successfully created render directory at: \n\n%s" % renderPath) ↵
         else: ↵
            nuke.message("render directory exists") ↵
      else: ↵
         nuke.message("Please make sure to set a render path") ↵
```

Mit dem Drücken der Schaltfläche „force create directory" wird geprüft, ob der reine Ordnerpfad des Write Nodes existiert. Notfalls wird er automatisch erstellt. Dies erspart das manuelle Erstellen des Ordners.

```
   #save backup ↵
   if knob.name() == "save backup": ↵
      if knob.getValue() == 1.0: ↵
         node["beforeRender"].setValue("nextVersion.saveBackup(‚%s')" % node.name()) ↵
      else: ↵
         node["beforeRender"].setValue("") ↵
```

In jedem Write Node können zusätzlich Befehle festgelegt werden, die vor oder nach dem Rendern ausgeführt werden sollen. Ist die Checkbox „save Backup" gewählt, erhält sie den Wert 1.0. Dadurch wird im Write Node unter dem Tab „Python" im Knob „beforeRender" der Befehl saveBackup() registriert. Dies bewirkt, dass die definierte Funktion saveBackup() vor dem Rendern aufgerufen und im Renderordner eine Backup-Datei des aktuellen Nuke-Skripts gespeichert wird. Ist die Checkbox abgewählt, so wird dieser Befehl im Knob „beforeRender" entfernt.

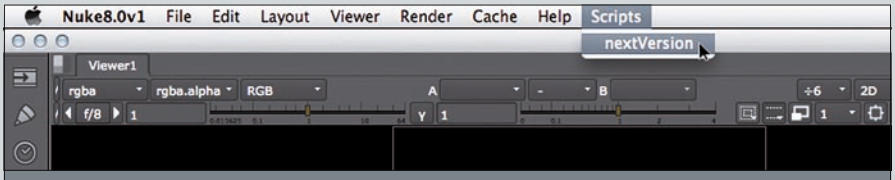
Die Funktion nextVersion():

```
def nextVersion():

   p = nextVersionPanel() ↵
   if p.show(): ↵
      #script version up ↵
      nukescripts.script_version_up() ↵
      #writeNode version up and create folder ↵
      for node in nuke.allNodes("Write"): ↵
      name = node.name() ↵
                                    if p.value(name) == 1: ↵

      node.knob("create next version").execute() ↵
```

Die Funktion nextVersion() ruft die Funktion nextVersionPanel() auf, welche wie festgelegt ein Fenster mit einer Checkbox für jeden Write Node zur Versionserhöhung bereitstellt.

Beim Klick auf OK wird das Nuke-Skript in der Version erhöht. Zudem wird für jeden Write Node, dessen Checkbox im Fenster aktiviert wurde, ein Klick auf den Button „create next version" simuliert. Dadurch werden die im Fenster ausgewählten Write Nodes in ihrer Version erhöht und die Renderordner werden automatisch erstellt.

Damit ist nextVersion komplett. Die Menu Bar von Nuke enthält nun einen neuen Eintrag: Scripts/nextVersion. Klickt der Nutzer darauf, wird die Funktion nextVersion() aufgerufen und es erscheint ein Fenster mit je einer Checkbox für jeden Write Node zur

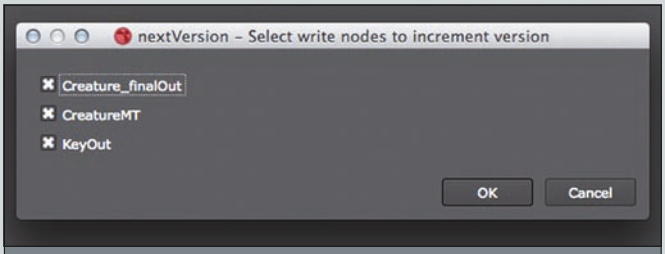


Nukes Menu Bar enthält nun eine weitere Kategorie – Scripts –, in der sich die Funktion nextVersion befindet.

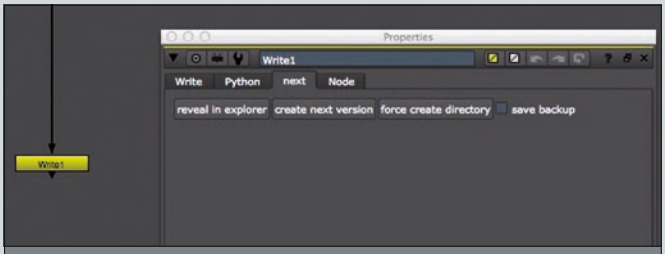
Versionserhöhung. Zudem trägt jeder Write Node nun die selbst erstellten Elemente im eigenen Tab „next".

Mit nextVersion konnten einige repetitive Aufgaben, wie etwa die Versionierung, die Speicherung von Backup-Dateien und das Erstellen von Renderordnern automatisiert werden. Das Skript ist jedoch nicht fertig. Es kann leicht modifiziert und erweitert werden. Zudem ist es wichtig, mögliche fehlerhafte Eingaben des Nutzers abzufangen. Zudem ist es wichtig, mögliche fehlerhafte Eingaben des Nutzers abzufangen. Das Skript beinhaltet noch lange nicht alle Fallunterscheidungen zum Abfangen fehlerhafter Eingaben des Nutzers, da dies den Rahmen des Artikels sprengen würde. Viel wichtiger war es, einen Einstieg in Python zu geben und aufzuzeigen, wie die Programmiersprache angewandt werden kann, um Nuke zu erweitern und den Workflow zu beschleunigen. › ei

Simon Jokuschies (www.Leafpictures.de) studierte Informationstechnologie und Gestaltung an der Fachhochschule Lübeck sowie ein Auslandssemester Digital Media an der Auckland University of Technology in Neuseeland. Er arbeitet bei der nhb Video GmbH im Bereich der Postproduktion als Compositor. Seine Schwerpunkte liegen in den Bereichen Compositing, 3D und Programmierung. Zudem ist er Gründer von pixiliti, dem Onlinenetzwerk für Medienstudenten und Kreative (www.pixiliti.de).



Im nextVersion-Fenster lassen sich bequem Versionserhöhungen für mehrere Write Nodes vornehmen.



Jeder Write Node trägt nun zusätzliche Schaltflächen und wird um nützliche Funktionen ergänzt.

Und nun in einem Schwung: die gesamte nextversion.py

```python
import nuke
import nukescripts
import os
import sys
import subprocess
import platform

def addCustomElements():

    node = nuke.thisNode()

    #init custom elements
    tab_next = nuke.Tab_Knob("next", "next")
    btn_revealInFinder = nuke.PyScript_Knob('reveal in explorer', 'reveal in explorer', '')
    btn_createNextVersionFolder = nuke.PyScript_Knob('create next version', 'create next version', '')
    btn_forceCreateDirectory = nuke.PyScript_Knob('force create directory', 'force create directory', '')
    chBox_saveBackup = nuke.Boolean_Knob('save backup', 'save backup')

    #add custom elements to node
    node.addKnob(tab_next)
    node.addKnob(btn_revealInFinder)
    node.addKnob(btn_createNextVersionFolder)
    node.addKnob(btn_forceCreateDirectory)
    node.addKnob(chBox_saveBackup)

def nextVersionPanel():

    p = nuke.Panel('nextVersion - Select write nodes to increment version')
    p.setWidth(500)
    for node in nuke.allNodes("Write"):
        p.addBooleanCheckBox("%s" % node.name() , 1)
    return p

def saveBackup(writeName):

    scriptLocation = nuke.root().name()
    scriptName = os.path.basename(scriptLocation)
    fileValue = nuke.toNode(writeName)['file'].getValue()
    renderPath = os.path.dirname(fileValue)

    nuke.scriptSave(renderPath + "/backup_" + scriptName)

def performCustomAction():

    node = nuke.thisNode()
    knob = nuke.thisKnob()

    fileValue = node["file"].getValue()
    renderPath = os.path.dirname(fileValue)

    #reveal in explorer
    if knob.name() == "reveal in explorer":
        if renderPath!="":
            try:
                if platform.system() == "Windows":
                    os.startfile(renderPath)
                elif platform.system() == "Darwin":
                    subprocess.Popen(["open", renderPath])
                else:
                    subprocess.Popen(["xdg-open", renderPath])
            except:
                nuke.message("couldn't open render path. No such directory")
        else:
            nuke.message("Please make sure to set a render path")

    #create next version
    if knob.name() == "create next version":
        if renderPath != "":
            nukescripts.clear_selection_recursive()
            node.setSelected(True)
            nukescripts.version_up()
            node.setSelected(False)

            fileValue = node["file"].getValue()
            renderPath = os.path.dirname(fileValue)

            if not os.path.isdir(renderPath):
                os.makedirs(renderPath)
                nuke.message("successfully versioned up")
            else:
                nuke.message("Renderfolder '%s' seems to exist" % renderPath)
        else:
            nuke.message("Please make sure to set a render path")

    #force create directory
    if knob.name() == "force create directory":
        if renderPath != "":
            if not os.path.isdir(renderPath):
                os.makedirs(renderPath)
                nuke.message("successfully created render directory at: \n\n%s" % renderPath)
            else:
                nuke.message("render directory exists")
        else:
            nuke.message("Please make sure to set a render path")

    #save backup
    if knob.name() == "save backup":
        if knob.getValue() == 1.0:
            node["beforeRender"].setValue("nextVersion.saveBackup('%s')" % node.name())
        else:
            node["beforeRender"].setValue("")

def nextVersion():

    p = nextVersionPanel()
    if p.show():
        #script version up
        nukescripts.script_version_up()
        #writeNode version up and create folder
        for node in nuke.allNodes("Write"):
            name = node.name()
            if p.value(name) == 1:
                node.knob("create next version").execute()
```