Wolfgang Hurst

mentalRay Shader Programmierung mit SoftImage|XSI

Die ersten Schritte zu einem Shader

Wolfgang Hurst
http://www.whurst.net
Version 1.0

Diese Seite ist leer

Das Dokument ist für die Anzeige von zwei Seiten optimiert

Inhaltsverzeichnis

Die ersten Worte

Aufbau des Dokumentes

Das Dokument ist so aufgebaut, dass man es Seite für Seite direkt nachvollziehen kann. Ich habe versucht hitziges Blättern zwischen den Kapiteln zu vermeiden. Wer Seite für Seite liest und dabei die Aktionen umsetzt, wird direkt – ohne Umweg – zum Ziel geführt.

Diese Art des Dokumentflusses hat aber auch einen kleinen Schönheitsfehler, es gibt bestimmt Momente wo man meint – „Was Zum Geier Mache Ich Hier?“, ja ganz bestimmt! Einfach machen und womöglich geht auf der nächsten Seite das Flutlicht an. Bin ich mir Sicher ☺

Der erste Teil beschäftigt sich primär mit den Tools an sich. Um irgendetwas entwickeln zu können, braucht man zwingend auch eine Landschaft die funktioniert. Weiterhin ist Hintergrundwissen auch nicht ganz so verkehrt. Beides werde ich in diesem Dokument versuchen dem Leser zu vermitteln.

Das Dokument selbst ist keine Referenz, wo man mal kurz auf Seite 42 nachschlägt wie XYZ funktioniert. Ja gut – man kann das tun, wenn man das Dokument bereits im Schlaf rückwärts mit dem Fuß an die Decke schreiben kann. Jemand der das Dokument nicht gelesen hat wird wohl nicht viel Glücklicher werden als mit den 2,8 Millionen anderer Tutorials im Internet. Es ist als Learning-By-Page-To-Page gedacht – oder so …

Einleitung

Was man braucht

Was braucht man um Shader für SoftImage bezüglich mentalRay zu programmieren ist nicht wirklich viel. Wir brauchen unbedingt ein C++ Compiler. Für Linux sollte der GCC ausreichen, für Windows sollte man den Visual C++ nehmen. Es gibt unter Windows zwar auch andere Compiler, mit denen habe ich es aber nicht probiert.

Dieses Dokument basiert auf folgender Software:

- Microsoft Windows 7 64bit
- AutoDesk SoftImage|XSI 7.5 oder 2012 – Windows 64bit
- AutoDesk SoftImage|XSI 7.5 oder 2012 – Windows 32bit (Trial reicht)
- Microsoft Visual C++ 2012 Professional

Wobei das Visual C++ 2012 bei mir noch die freie Version zum Probieren ist. Mit der Visual C++ 2010 Express Version sollte es aber auch gehen. Ich habe beide SoftImage Versionen aufgeführt, da dieses Dokument für beide gilt. Das sind halt die die ich habe …

Bei SoftImage 32bit habe ich die Trial installiert, weil wir die SDK Libs brauchen. Wer nur 32bit hat muss später bei dem Visual C++ einiges überspringen oder gedanklich ausklammern.

Im Folgenden wird erwartet das, oben genannte Software einwandfrei funktioniert. Da alle Schritte bis auf das Detail runtergebrochen werden, braucht niemand ein absoluter Spezialist auf der Software zu sein. Aber die Software sollte man unfallfrei starten können.

Funktionsweise eines mentalRay Shaders

Bevor wir uns die Pointer um die Ohren hauen ist es wichtig zu verstehen wie das Ganze funktioniert. Wenn man das nicht weiß hat man erst einmal ein paar Start Schwierigkeiten. Aber zur Beruhigung die Funktionsweise ist denkbar einfach. Viel einfacher als man eigentlich erwarten sollte.

Das Funktionsprinzip ist, ich will es mal so ausdrücken, Ray-basierend. Immer dann wenn mentalRay eine Szene rendert und ein Lichtstrahl auf unser Objekt mit unseren Shader trifft, wird unsere Funktion aufgerufen. Und unsere Funktion liefert dann die Farbe des Punktes.

Man kann geteilter Meinung sein, ob das nun eine gute Idee ist oder nicht. Eins ist aber entsprechend zu beachten: Unsere Funktion wird unter Umständen zig-tausend Mal aufgerufen. Das zwingt uns dazu möglichst wenig in unserer Funktion zu machen. Ansonsten haben wir keinen Spaß beim Rendern.

Wird zum Beispiel eine simple Textur benutzt, kann man die Textur beim ersten Ray entsprechend laden, und die nachfolgenden Rays bedienen sich dann einfach aus dem großen Array. Kein Problem. Viele Shader aber tun das nicht. Insbesondere Shader die Fraktale, Wolken oder sonstige Texturen liefern tun das nicht. Können sie auch nicht … weil wenn die Kamera näher an das Objekt kommt, würde es zum einen sehr klotzig werden zum anderen würden Bereiche berechnet die man nicht braucht.

Also unser Shader wird also bei jedem Ray aufgerufen. mentalRay über gibt uns diverse Informationen mit denen wir herausfinden können wo wir gerade sind. Und als Ergebnis haben wir eine Farbe zu liefern.

Unser Shader sollte aber auch noch eine Eigenschaft besitzen. Zwar bekommen wir von mentalRay Informationen vorgesetzt, es ist aber fatal sich Blind darauf zu verlassen. Wenn z.B. der Textur-Space nicht zugewiesen wurde, dann wird auch keiner von mentalRay geliefert. Wenn wir jetzt aber trotzdem darauf zugreifen, dann haben wir einen Crash. Das sollte unbedingt vermieden werden. Man sollte lieber alles doppelt prüfen, bevor es an allen Ecken und Enden knallt.

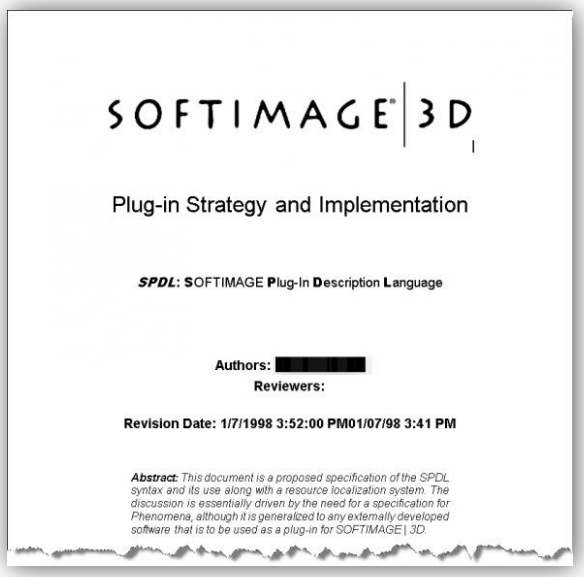
Also zusammenfassend könnte man jetzt grob die Anforderungen oder die goldenen Regeln unserer Funktion – sprich Shader – definieren:

- Shader wird bei jedem Ray aufgerufen
- Shader Funktion sollte so schnell wie irgend möglich sein
- Shader Funktionen sollten Pointer und Werte prüfen

Integration in SoftImage

Damit wir den Shader auch konfigurieren können benötigt SoftImage eine entsprechende Beschreibung des Shaders. Diese Aufgabe übernimmt eine sogenannte SPDL[1] Datei.

Ich war Lange auf der Suche nach was SPDL überhaupt bedeutet. Man will es ja kaum glauben, aber selbst wenn man den vollständigen Begriff in den Google eingibt, bekommt man quasi keine Antwort. Aber ich habe was gefunden:



SOFTIMAGE | 3D

Plug-in Strategy and Implementation

***SPDL*: SOFTIMAGE Plug-In Description Language**

Authors: [redacted]
Reviewers:

Revision Date: 1/7/1998 3:52:00 PM01/07/98 3:41 PM

Abstract: *This document is a proposed specification of the SPDL syntax and its use along with a resource localization system. The discussion is essentially driven by the need for a specification for Phenomena, although it is generalized to any externally developed software that is to be used as a plug-in for SOFTIMAGE | 3D.*

Abbildung 1 - SPDL und was es bedeutet von 1998

In einer SPDL Datei sagen wir dem SoftImage welche Inputs und Outputs wir haben. Die Inputs und Outputs kennen wir ja aus dem Rendertree, wo wir Shader miteinander verbinden können.

[1] SPDL – SoftImage Plugin Description Language – siehe http://xsisupport.com/2012/07/06/

Weiterhin legen wir in der SPDL Datei fest, welche Eigenschaften und wie unsere Dialoge des Shaders aussehen sollen. Spätestens dann wenn wir im Rendertree auf unseren Shader klicken und uns die Eigenschaften anschauen wollen kommt das SPDL zum Einsatz.

Die SPDL übernimmt aber auch die Schnittstelle zwischen SoftImage und unserer Programmierung. Das was wir in SoftImage einstellen, muss ja irgendwie den Weg in unsere Programmierung finden. Genau da greift uns die SPDL unter die Arme.

Die SPDL wird aber auch für Funktionen innerhalb unseres Eigenschaften Dialoges verwendet. Wenn wir zum Beispiel ein ganzen Bereich ausgrauen wollen, wenn wir eine Checkbox deaktivieren. Diese Funktionalität kann nur in SPDL beschrieben werden. Beispielhaft die *Transparency* oder *Reflection* Gruppe beim Phong oder Lambert Shader. Wenn ich da die Checkbox deaktiviere, wird der Rest ausgegraut.

Das Problem mit SPDL Dateien ist, das wir bei Änderungen immer das ganze SoftImage restarten müssen. Das kann einem, insbesondere beim Testen, sehr massive Nerven kosten.

Zusammenfassen, obwohl ich nicht viel geschrieben habe :

- SPDL konfiguriert die Inputs und Outputs und deren Datentypen
- SPDL gestaltet den Eigenschaftsdialog
- SPDL fügt Funktionalitäten in die UI ein

Erstellen eines vollständigen Templates

Das Erstellen eines Templates ist relativ einfach. SoftImage gibt einem alle nötigen Mittel inklusive bedienerfreundlichen UI. Damit lässt sich in wenigen Minuten ein einfaches Grundgerüst bauen. SoftImage erstellt uns ein SPDL File, die Headerdatei, eine C++ Datei und eine Visual C++ Projektdatei, bzw. ein GNU Makefile.

In der GUI von SoftImage können wir unsere Inputs und Outputs sehr einfach definieren. Und ein Knopfdruck später haben wir alles nötige für das Grundgerüst zusammen.

Kurz vor dem Start

So nun wissen wir dass wir eigentlich zwei Sachen haben die Perfekt ineinander greifen müssen. SPDL und unser C++. Weiterhin brauchen wir ein Plan … hier ein Plan:

1. Festlegung des Shaders (Textur, Material, …)
2. Festlegung der Inputs
3. Erstellen des SPDL und des Basis C++ Shaders
4. Programmieren des Shaders
5. Kompilieren des C++ Source
6. Importieren des SPDL und des Shaders
7. Testen, Testen und Testen

In diesem Dokument machen wir gleich ein Textur-Shader. Als erstes werden wir ein einfachen Input einer Farbe haben das werden wir zuerst automatisiert machen, und dann später an der SPDL selbst rumschrauben. Den Textur-Shader werden wir dann auch ausbauen. Kompilieren tun wir das Ganze auch im Detail.

Aber nun genug des Blabla … jetzt geht es los ☺

Grundlagen des SoftImage und Visual C++ Zusammenspiel

Dieses Kapitel beschreibt die nötigen Grundlagen im Detail und geht wenig auf den Shader an sich ein. Erklärt wird hier wie man ein SPDL und die Visual C++ Projekte konfiguriert

Anforderungen

Als erstes müssen wir uns Überlegen welche Inputs und Outputs wir in unserem Textur-Shader haben wollen und was er tun soll. Da es unser erster Shader sein soll, wollen wir erst einmal nur eine Farbe definieren können die unser Shader als Output liefert. Also das absolute – und wahrscheinlich sinnloseste – Minimum was man sich vorstellen kann. Es geht aber eigentlich mehr darum um den Workflow der Programme zu zeigen und zu konfigurieren, als darum jetzt einen Super Shader zu bauen.

Template Erstellen mit SoftImage

Als erstes Öffnen wir SoftImage und klicken im Menü File auf PlugIn-Manager. Im darauffolgenden Popup wählen wir den SPDLs Tab. Dort auf File -> New -> mentalRay Shader, bei SoftImage 7.5 heißt es nur Shader. Das nachfolgende Bild zeigt den Schritt

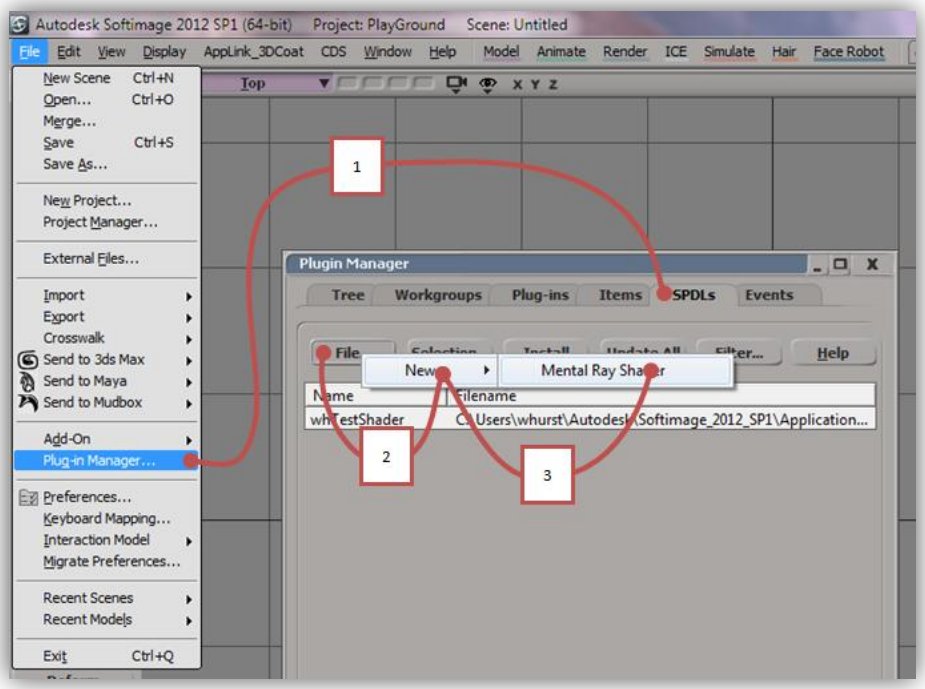


Abbildung 2 - Mit dem Plugin Manager ein SPDL erstellen

Als nächstes macht sich ein Einstellungsdialog auf wo wir unsere Einstellungen entsprechend einstellen können ☺

Zu dem Wizard könnt Ihr auch die Dokumentation von AutoDesk zu Rate ziehen. Unter http://download.autodesk.com/global/docs/softimage2012/en_us/sdkguide/index.html?url=files/cus_mrshad_wizard.htm,topicNumber=d28e25358 könnt Ihr nachschlagen.

Einstellungen des Tabs *Shader Information*

Im Tab Shader Information werden Grundparameter für das Grundgerüst und den Namen des Shaders eingestellt. Die Einstellungen dabei haben folgende Bedeutung:

Einstellungsoption	Bedeutung
Short Name	Der Name des Projektes. Dieser taucht in SoftImage im Rendertree auf, ist auch gleichzeitig der Name des Plugins und des Source Codes
Long Name	Eine etwas längere Beschreibung. Die taucht später im Materialeditor auf
Main Group	Die Main Group stellt den Typ des Shaders ein. Es gibt dort Texture, Material, Volume und andere. Es werden jeweilige Optionen in dieser Gruppe angezeigt, wenn man es umstellt
Texture Type	Wird nur angezeigt, wenn die Main Group auf Texture steht und gibt den Ausgabe Typ an. Normal steht der auf Color
Generate Init/Exit Code	Beim Generieren des Grundgerüstes wird ein Init und Exit Block erstellt. Sollte man immer aktivieren, man weiß ja nie …
Project Name	Ich tippe darauf dass das der Name der Dateien sein wird, den der Wizard dann erstellt. Bitte bitte stellt den gleichen ein wie unter Short Name. Ich weiß nicht was passiert, wenn es sich unterscheidet. Sobald man den Short Namen eingibt wird der Projekt automatisch angepasst
Output Directory	In diesem Verzeichnis landen nachher alle Grundgerüstdateien. Bitte wählt dort nicht den Default.
Generate Project	Den Knopf drücken wir erst wenn wir alles eingestellt haben!

Nach dem wir nun wissen was die Parameter bedeuten, füllen wir diese entsprechend aus. Als

- Short Name : **tutSimpleColor**
- Long Name : **Tutorial Simple Color**
- Main Group : **Texture**
- Texture Type : **Color**
- Generate Init/Exit Code : **on**
- Project Name : **tutSimpleColor**
- Output Directory : *Bitte wählt ein geeigneten Ort für eure Sourcen*

Das Ganze, bis auf den Output Path, sollte dann etwa so aussehen:

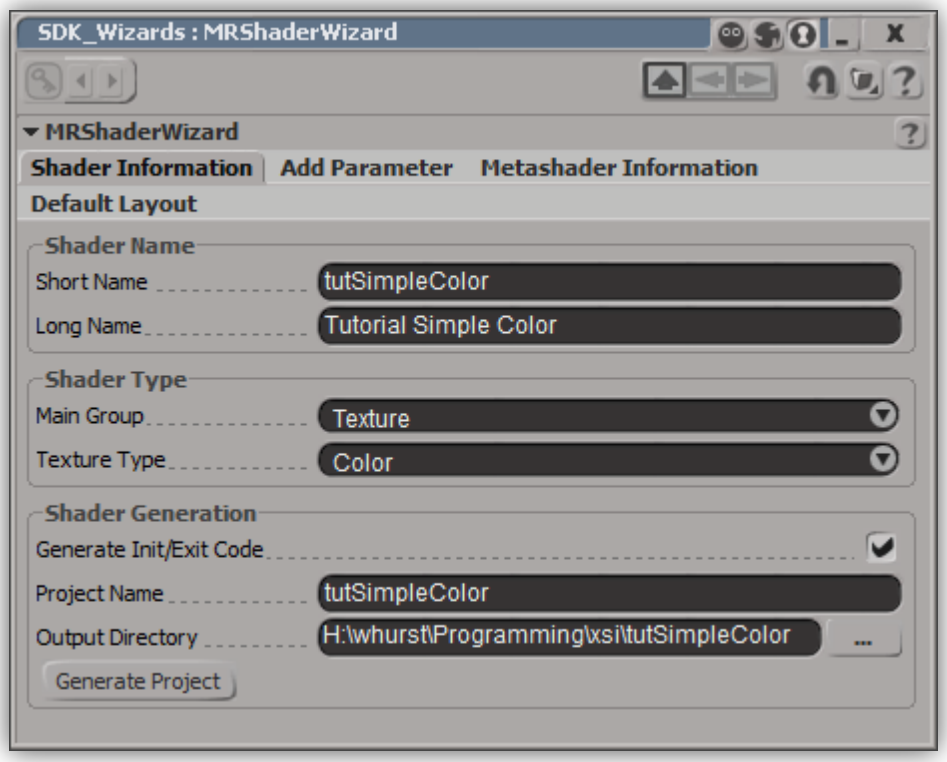


Abbildung 3 - SDK Wizard - Shader Informationen für tutSimpleColor

Einstellungen des Tabs *Add Parameter*

Da unser Shader einfach nur eine von uns eingestellte Farbe weitergeben soll, brauchen wir natürlich die Farbe. Je nachdem welchen Typ wir einstellen, verändert sich der Dialog etwas. Aber grundlegend kann man sagen:

Einstellungsoption	Bedeutung
Type	Gibt die Art des Parameters welche wir unserem Shader als Information zukommen lassen wollen
Name	Dieser Name wird intern vom Shader verwendet. Der Name legt überdies auch den Namen der Variable in unseren Shader fest
Label	Das was wir hier reinschreiben taucht später in unserer GUI auf als Bezeichnung für den Parameter
Animatable	Ist dieser Wert aktiv, kann man diesen animieren. Dabei kann man dann die Werte mittels FCurves modifizieren. Es gibt einige Typen wo man das nicht machen kann
Texturable	Wenn dieser Wert aktiv ist, kann man diesen Parameter von extern überschreiben. Dieses Flag öffnet also die Möglichkeit dort Ergebnisse von anderen Shader Komponenten hineinlaufen zulassen
Persistable	Der Wert den wir Einstellen wird gespeichert. Das ist immer eine gute Idee
Add Parameter Knopf	Den drücken wir, wenn wir oben alles eingestellt haben

Da wir eine Farbe haben wollen legen wir mal unsere Einstellungen fest :

- Type : **Color (alpha)**
- Name : **baseColor**
- Label : **Basis Color**
- Animatable : **on**
- Texturable : **off**
- Persistable : **on**
- Default Color : **1.0 0.0 0.0 1.0** (Knall Rot)

Danach drücken wir den Knopf Add Parameter und er taucht unten in der Liste auf. In unserem Dialog wird dabei automatisch der Name inkrementiert, nicht verwirren lassen im nachfolgenden Bild. Das ganze sollte dann in etwa so aussehen:

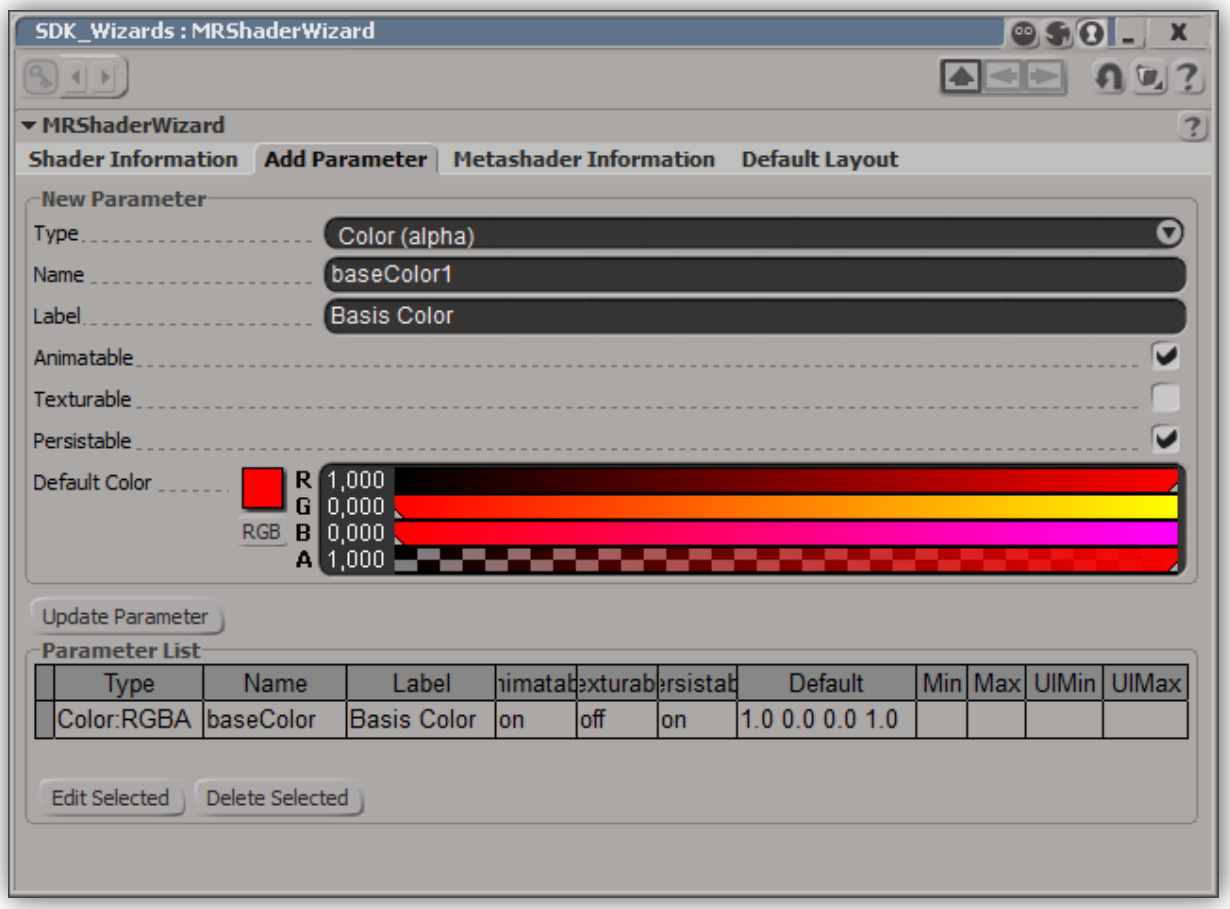


Abbildung 4 - SDK Wizard - Add Parameter für unseren tutSimpleColor

Einstellungen des Tabs *Metashader Informationen*

Die lassen wir erst einmal auf den Default Einstellungen. Ich bin mir nicht sicher ob ich in diesem Tutorial dazu kommen möchte die Werte zu beleuchten. Vielleicht viel später einmal

Einstellungen des Tabs *Default Layout*

Da wir nur einen einzigen Parameter haben, wird es jetzt massiv einfach. Auf der rechten Seite in der Liste Parameters stehen alle Parameter die wir definiert haben. In der Liste Layout ist anfänglich nichts. Damit wir aber unseren Parameter auch sehen müssen wir diesen einen jenen welchen auch in das Layout hinzufügen. Dazu markieren wir diesen und drücken unter der Parameter Liste auf den Knopf Add. Der Parameter sollte dann im Layout erscheinen.

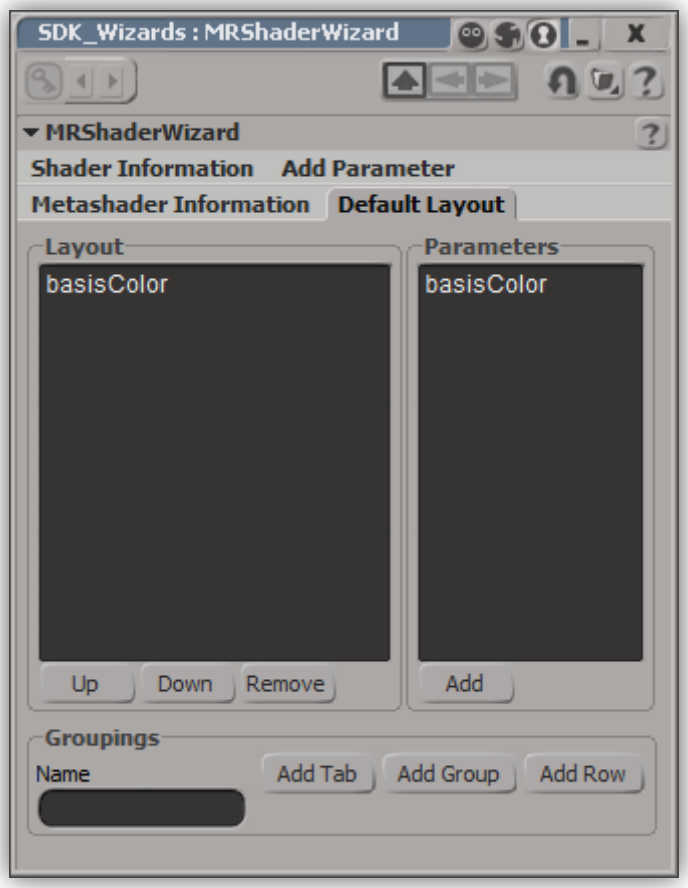


Abbildung 5 - SDK Wizard Layout Einstellungen für unseren tutSimpleColor

Wir werden später etwas mehr mit dem Layout machen. Für uns soll das jetzt erst einmal reichen. Macht ja auch bei einem Parameter nicht unbedingt Sinn

Grundgerüst erstellen

So nachdem wir unsere Einstellungen im SDK Wizard gemacht haben gehen wir auf den Tab Shader Information und drücken den Knopf Generate Project

Es passiert nichts … ja schaut im Script Editor oder auf der Statuszeile, dort steht das er es im Output Path erstellt hat. Da wir nun massiv neugierig sind, schauen wir mal in unser Output Path rein was er so gebaut hat:

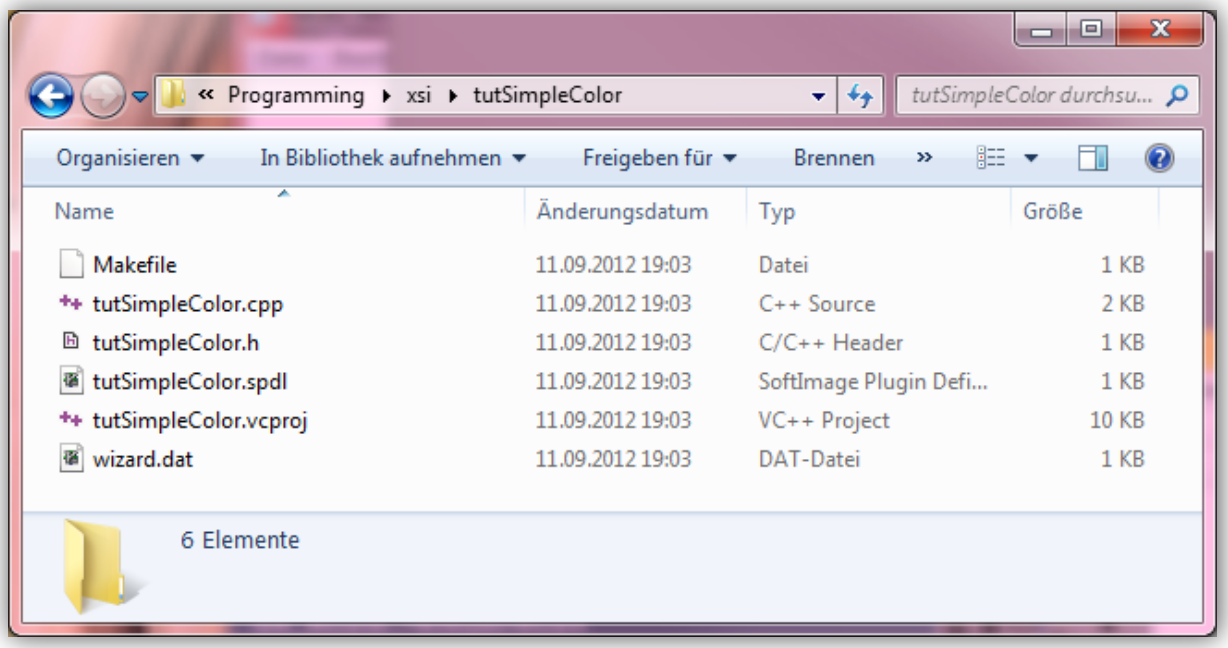


Abbildung 6 - Ergebnis im Output Path nach der Generierung von tutSimpleColor

Ganz viele Dateien. Bevor wir den Visual C++ starten, sollten wir – bevor er alles zuknüllt mit Zeug – anschauen was wir hier haben:

Dateiname	Aufgabe dieser Datei
Makefile	Das Makefile wird für UNIX benötigt. Es ist gemacht worden für den Gnu C++ Compiler
tutSimpleColor.cpp	Das ist der Source Code unseres Shaders
tutSimpleColor.h	In dieser Header Datei liegt unser Struct mit den Parametern
tutSimpleColor.spdl	Die SPDL Datei, wo das Layout, die Parameter und sonstiges definiert sind
tutSimpleColor.vcproj	Die Visual C++ Projekt Datei
wizard.dat	Irgendeine Datei für den Wizard. Bitte nicht drin rumfummeln

Importieren des Projektes in Visual C++

So wir sind jetzt in einer relativ heißen Phase. Wir importieren jetzt das fehlerhafte Visual C++ Projekt. Aber keine Angst, es ist irgendwie eine Visual C++ Projekt Datei aus längst vergangenen Tagen. Wir müssen nach dem Import manuell alles selbst einstellen, bzw. berichtigen.

Ich verwende hier ein Visual C++ 2012, es sollte aber auch mit Visual C++ 2010 Express gehen. Wir doppelklicken unser tutSimpleColor.vcproj und erwarten freudige Fehlermeldungen vom Visual C++. Man sollte sich den Text nicht durchlesen, der ist etwas komisch:

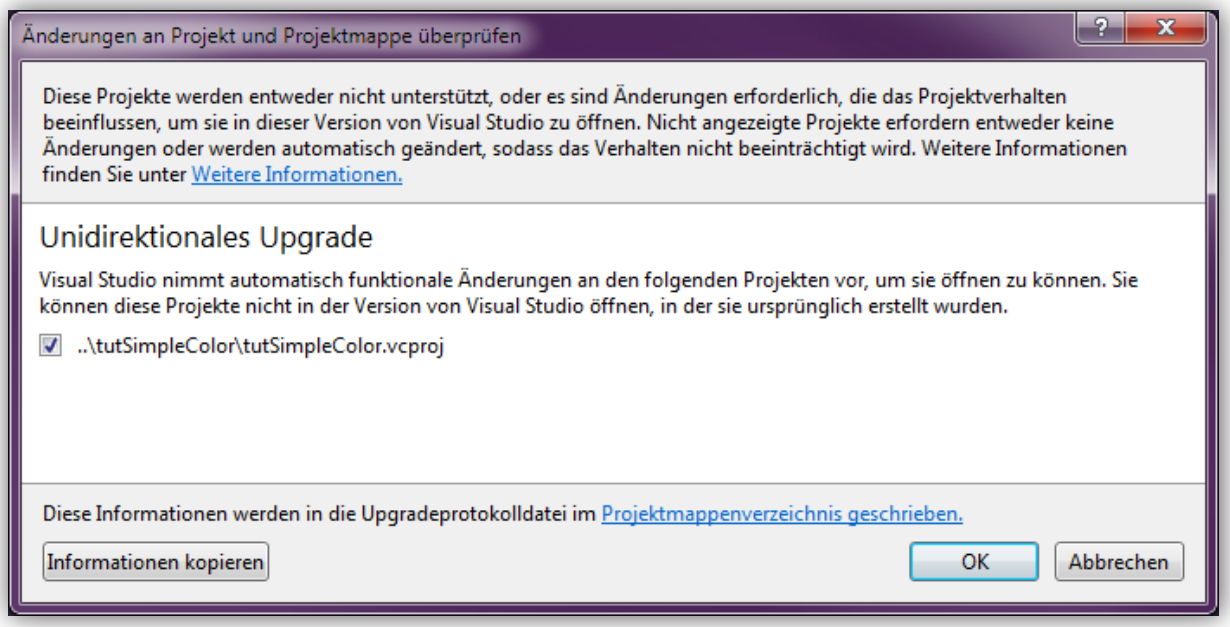


Abbildung 7 - Normale Fehlermeldung beim Import des vcproj in Visual C++

Einfach auf OK klicken bitte. Man bekommt einen relativ umfangreichen Bericht. Es sind 9 Warnungen und 5 Meldungen. In etwa. Berichte über Benutzerkontensteuerung ignorieren wir. Der Compiler Schalter /YX wurde entfernt weil es ihn nicht mehr gibt. Wir könnten jetzt nachschauen was es war, aber wenn er entfernt wurde, war er wohl sowieso nicht gut. Na gut, es waren die vorkompilierten Header, also für uns nichts Wildes. Was uns aber interessiert ist der Linker Output. Den müssen wir anpassen, weil sonst kompilieren wir später die 32bit Version über die 64bit Version, was nicht so gut ist

Default wird das Projekt als Debug Windows 32bit markiert. Wir gehen also erst einmal in die Monströsen Einstellungen unseres Compilers und ändern ein paar Dinge

Projekt Anpassungen für Debug Win32

Wir öffnen als voller Ehrfurcht unsere Projekt Eigenschaften. Auf der linken Seite im Projektmappen-Explorer klicken wir unser Projekt mit der rechten Maustaste (Kontextmenü) an und wählen ganz unten den Punkt Eigenschaften. Bitte nicht auf Projektmappe rechts klicken, dann kommt was ganz anderes. Also hier Klicken:

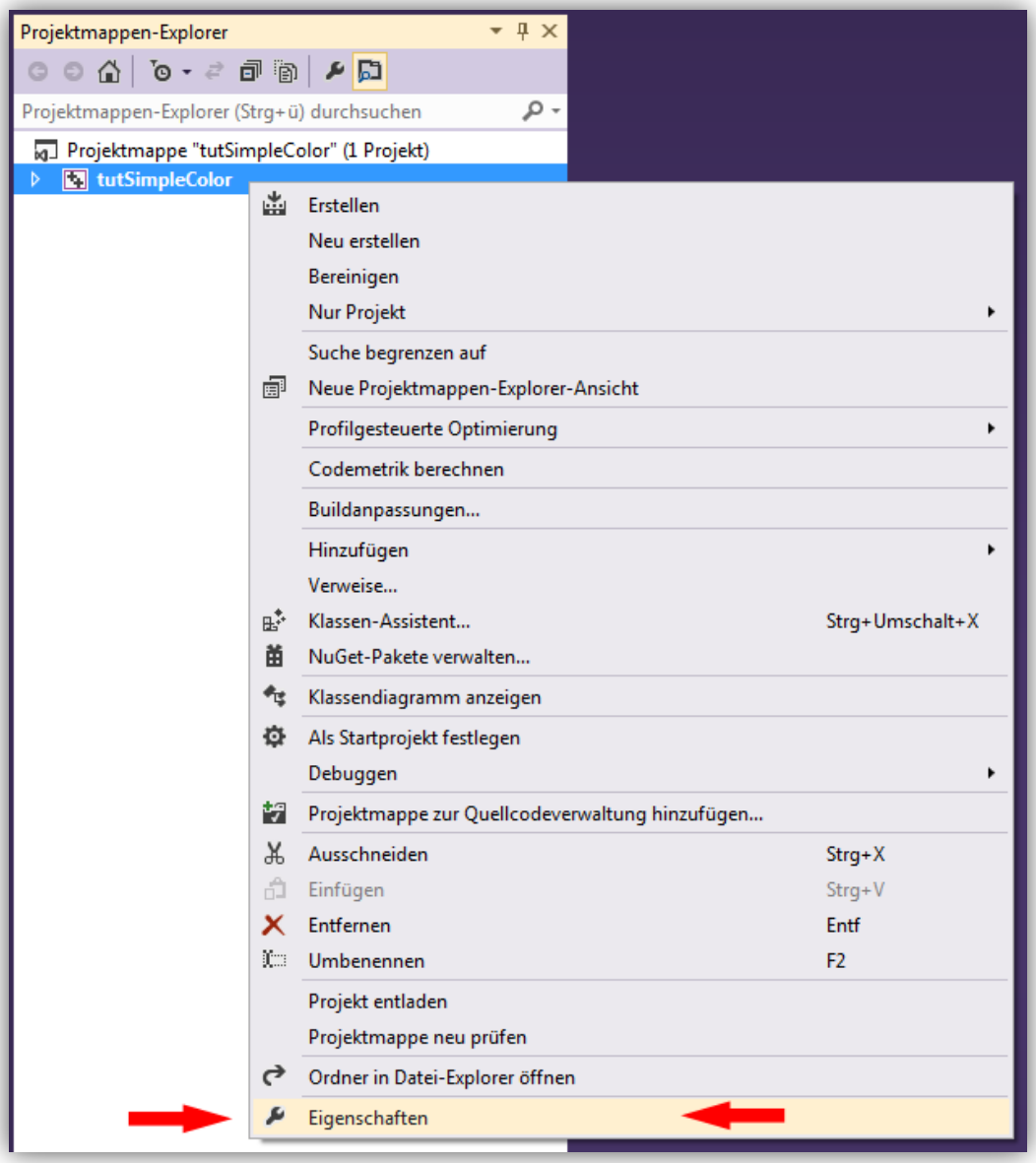


Abbildung 8 - Zu den Einstellungen unseres Projektes finden

Danach öffnet sich der Konfigurationsdialog. Wir müssen unbedingt ein paar Werte ändern.

Include Verzeichnisse

Der bekannte Fehler:

```
1>h:\whurst\programming\xsi\tutsimplecolor\tutsimplecolor.cpp(6): fatal error C1083: Datei
(Include) kann nicht geöffnet werden: "shader.h": No such file or directory
```

Ist ein Kennzeichen für fehlende Include Dateien. Um diesen Fehler zu reparieren ändern wir den Wert in unseren Projekt Einstellungen. In Konfigurationseigenschaften -> VC++ Verzeichnisse fügen wir in Include Verzeichnisse das Verzeichnis von SoftImages SDK Includes hinzu, das muss man halt Suchen, liegt aber etwa da:

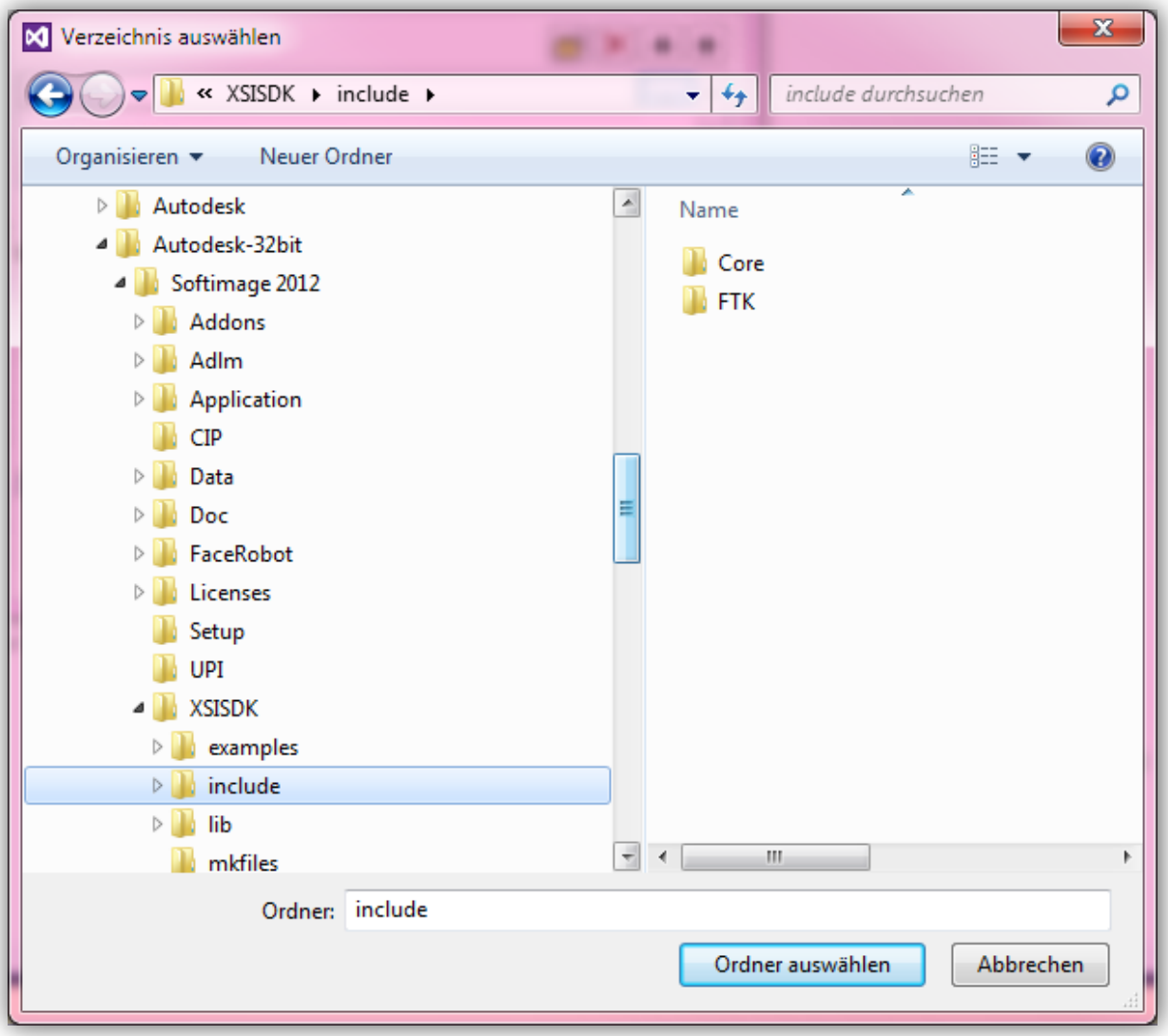


Abbildung 9 - Hinzufügen des SoftImage SDK Include 32bit Verzeichnis

Das ganze sollte dann etwa so aussehen:

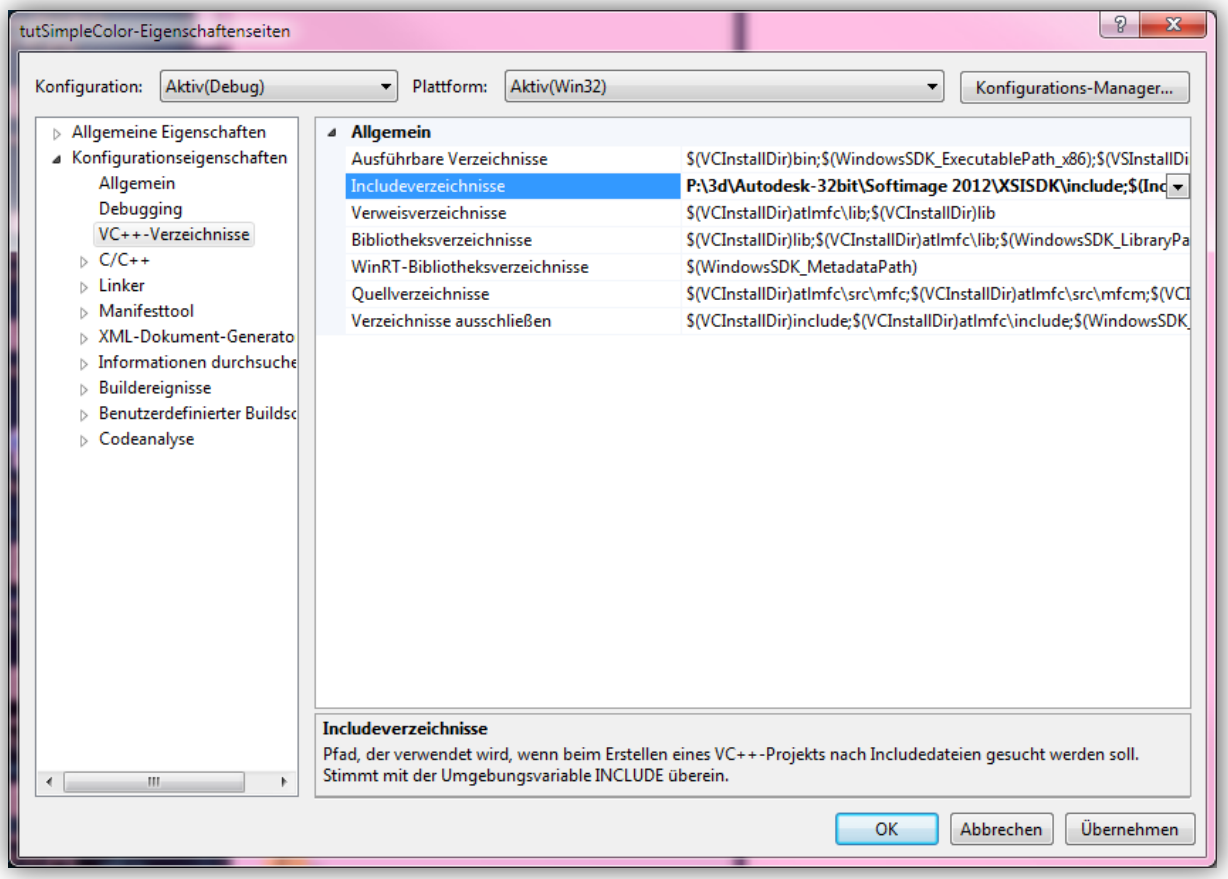


Abbildung 10 - Eigenschaften Seite nach dem Hinzufügen des Include 32bit Path

Lib Verzeichnisse

Ein weiterer bekannter Fehler ist dieser hier:

```
1>LINK : fatal error LNK1181: Eingabedatei "shader.lib" kann nicht geöffnet werden.
```

Ist ein Kennzeichen für die fehlende Lib Dateien. Um diesen Fehler zu reparieren ändern wir den Wert in unseren Projekt Einstellungen. In Konfigurationseigenschaften -> VC++ Verzeichnisse fügen wir in Bibliotheksverzeichnisse das Verzeichnis von SoftImages SDK Libs hinzu, das muss man halt Suchen, liegt aber etwa da:

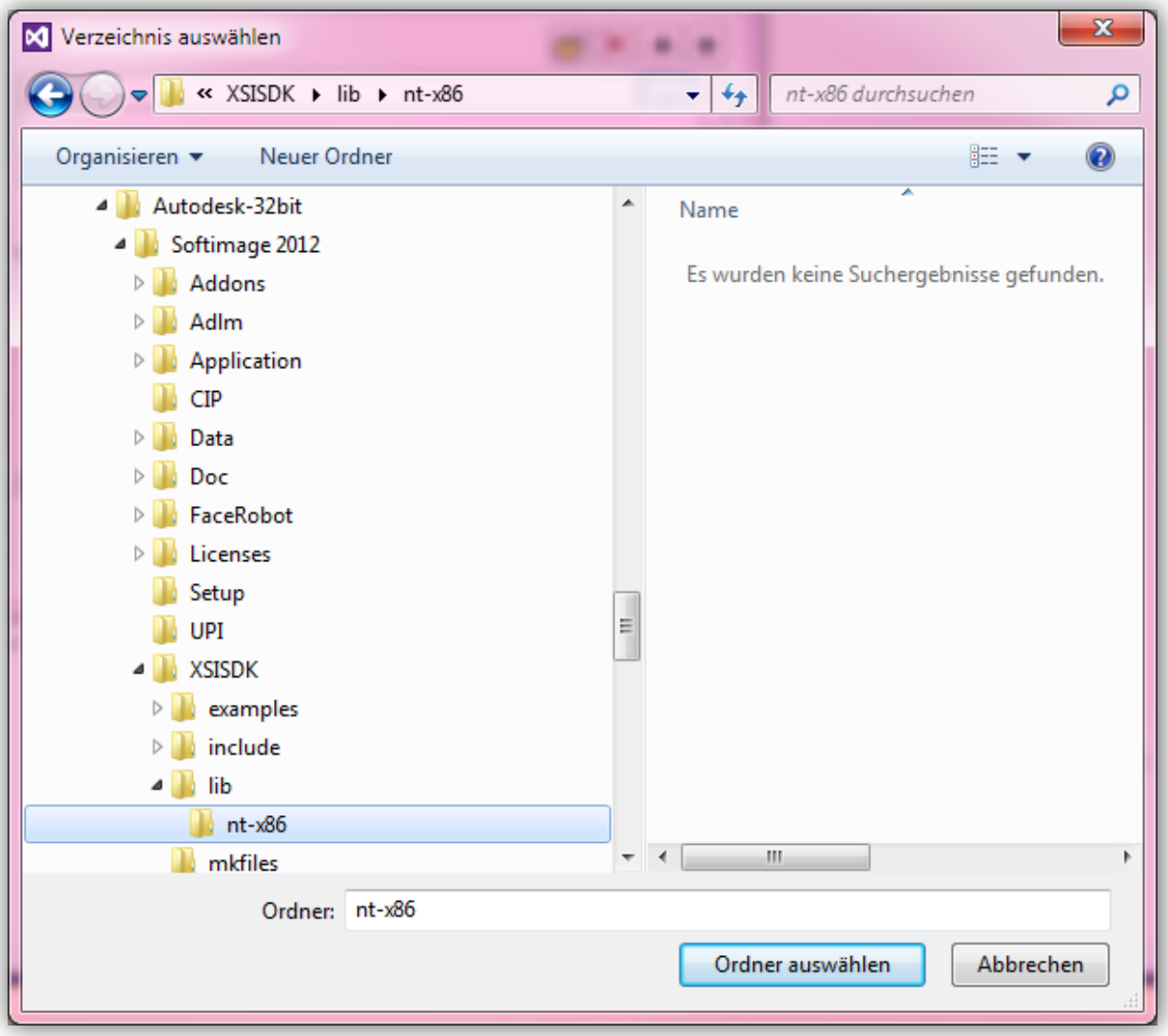


Abbildung 11 - Hinzufügen des SoftImage SDK Lib 32bit Verzeichnis

Danach sollten wir folgendes haben:

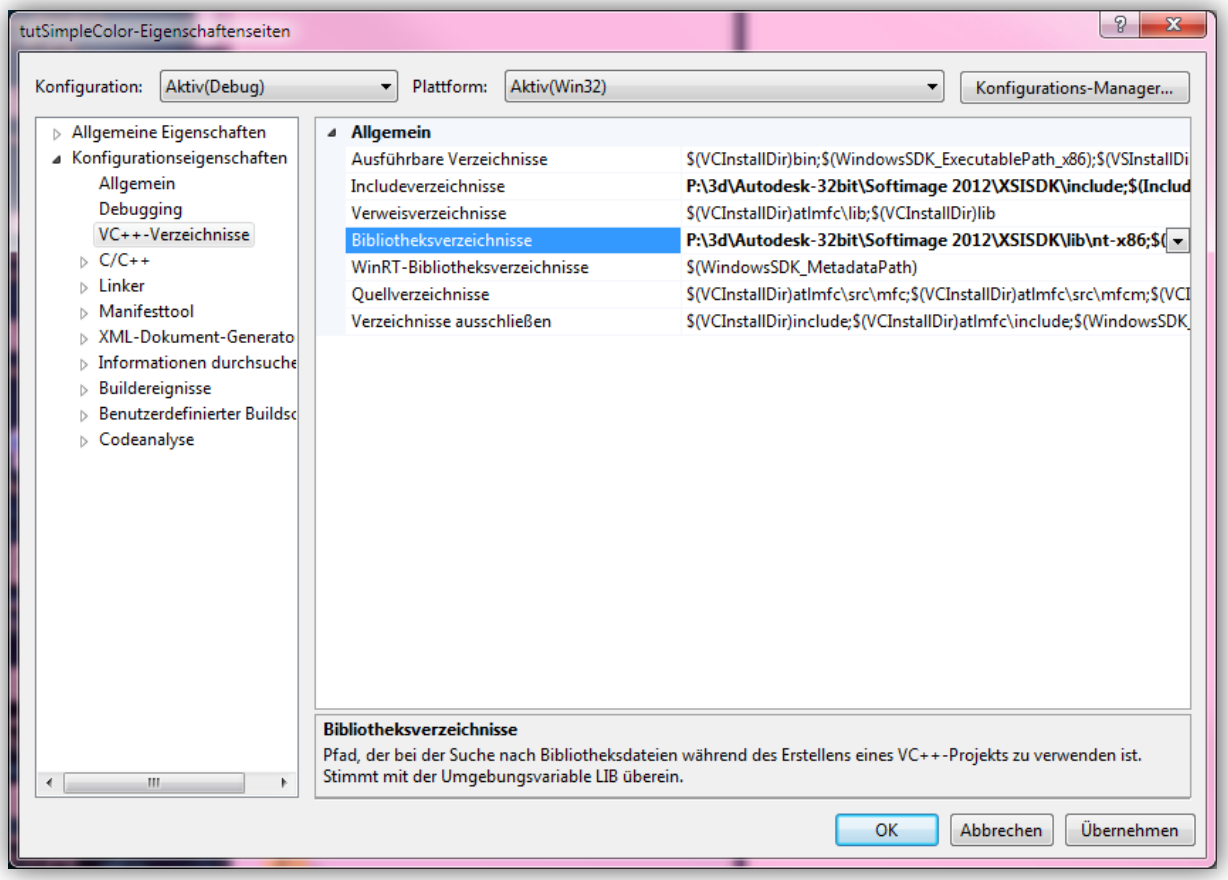


Abbildung 12 - Eigenschaften Seite nach dem Hinzufügen des Lib 32bit Path

Linker Output berichtigen

Der Linker Output passt leider überhaupt nicht. Das ist ein echtes Problem. Er nimmt nämlich unseren Projekt Path. Wenn wir später unser Target umschalten, wird die vorherige DLL Datei nämlich platt gemacht, und das ist echt schlecht. Erkennbar an folgender Meldung:

```
1>C:\....CppBuild.targets(1137,5): warning MSB8012: TargetPath
(H:\...\tutSimpleColor\.\Debug\tutSimpleColor.dll) does not match the Linker's OutputFile
property value (H:\...\xsi\tutSimpleColor\tutSimpleColor.dll). This may cause your project to
build incorrectly. To correct this, please make sure that $(OutDir), $(TargetName) and
$(TargetExt) property values match the value specified in %(Link.OutputFile).
```

In den Projekt Einstellungen müssen wir den Ausgabe Path der DLL ändern. Dazu geht man in die Projekt Eigenschaften -> Konfigurationseigenschaften -> Linker -> Allgemein und ändert dort die Ausgabe Datei um und fügt ein ./Debug/ vorne weg. Das sieht dann so aus:

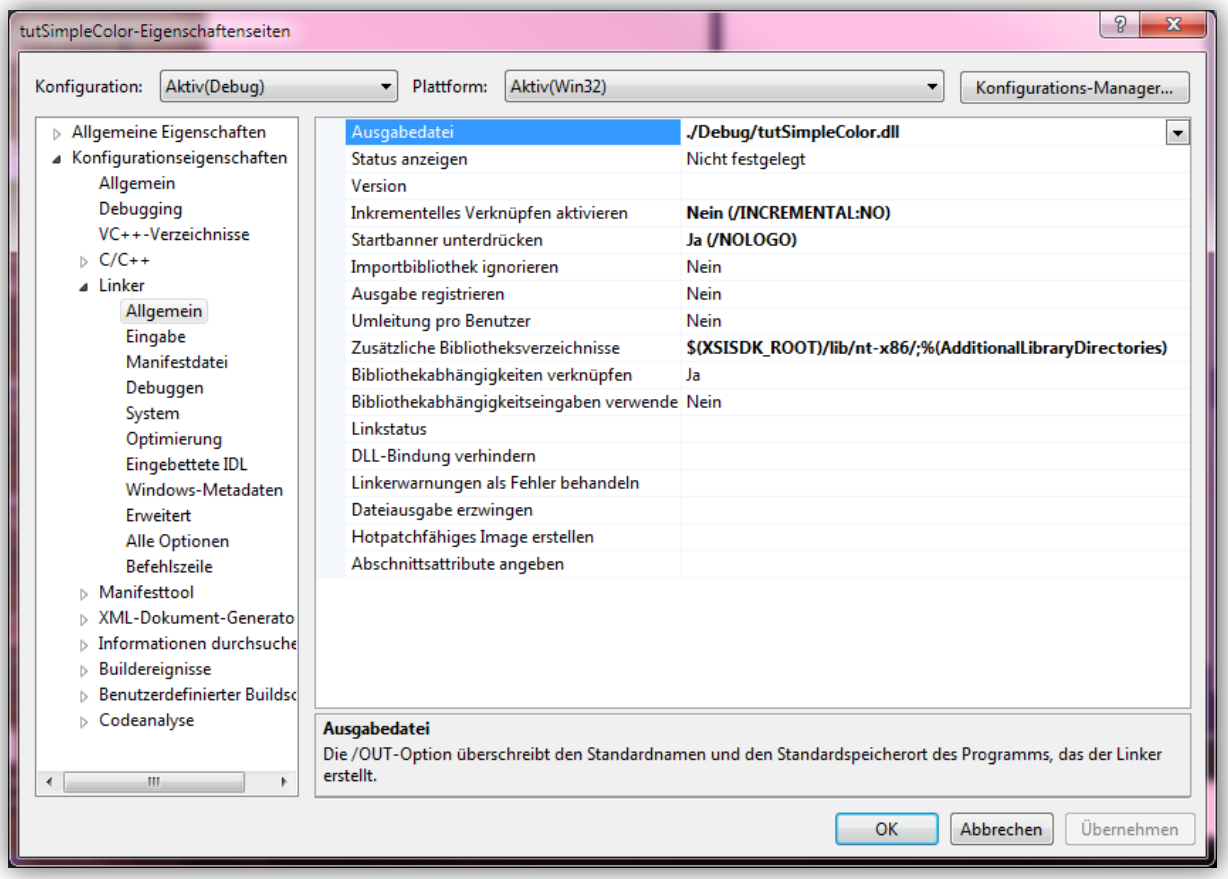


Abbildung 13 - Eigenschaften Seite nach dem Ändern des Ausgabe Path der 32bit DLL

Umschalten der Debug Informationen

Noch immer kommt uns ein Fehler entgegengehüpft:

```
1>tutSimpleColor.obj : warning LNK4075: /EDITANDCONTINUE wird aufgrund der Angabe von /OPT:LBR
ignoriert.
```

Das ist kein Beinbruch, ich will es aber weg haben. In den Eigenschaften des Projektes unter Konfigurationseigenschaften -> C/C++ -> Allgemein gibt es eine Einstellung mit dem Namen Debuginformationsformat. Das stellen wir auf nur Programmdatenbank um, etwa so:

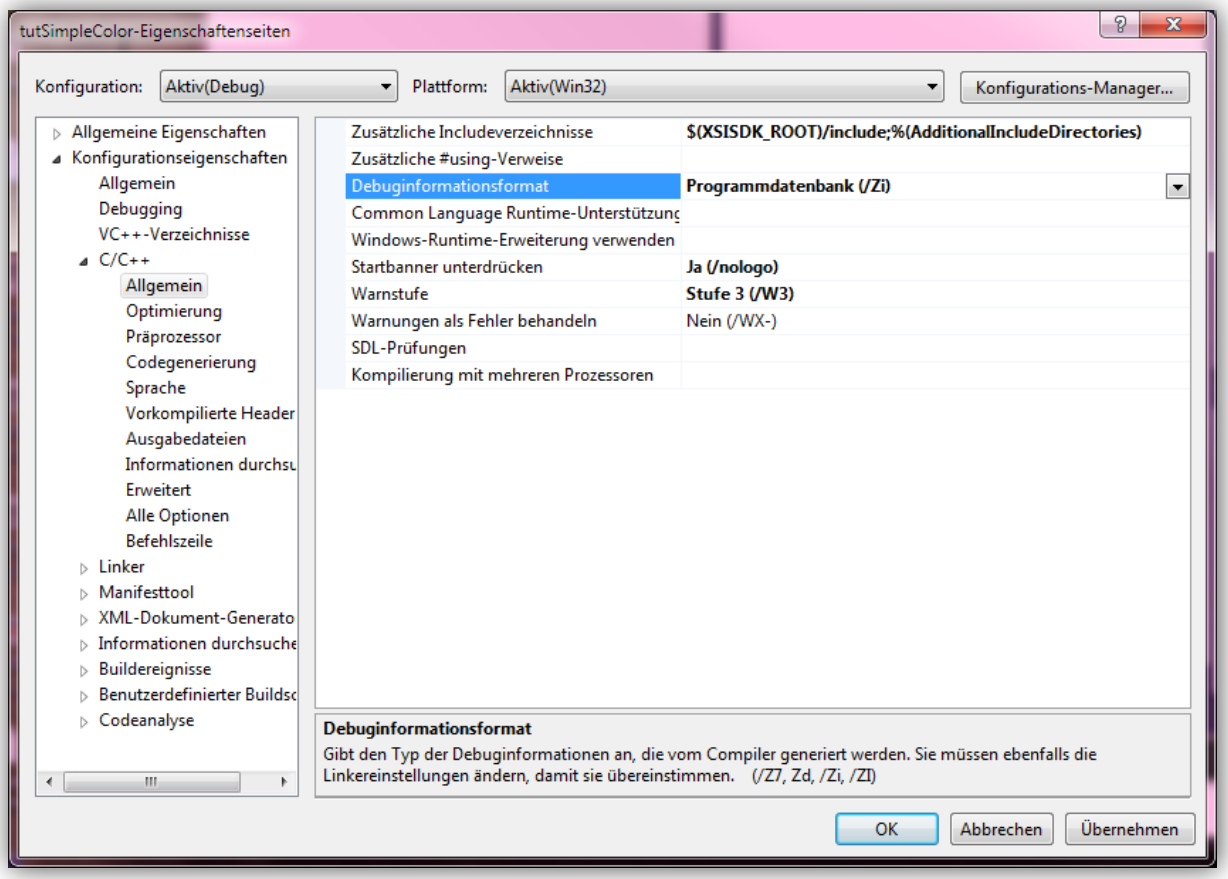


Abbildung 14 - Eigenschaften Seite nach dem Ändern der Debug Information

Das war alles. Zumindest für Visual C++ 2012

Test Kompilieren

Nachdem wir nun ein paar Änderungen gemacht haben erstellen wir nun einmal unser Projekt in dem wir mutig auf F7 drücken.

Wir sollten etwa so etwas bekommen:

```
1>------ Erstellen gestartet: Projekt: tutSimpleColor, Konfiguration: Debug Win32 ------
1>  tutSimpleColor.cpp
1>     Bibliothek ".\Debug/tutSimpleColor.lib" und Objekt ".\Debug/tutSimpleColor.exp" werden
erstellt.
1>  tutSimpleColor.vcxproj ->
H:\whurst\Programming\xsi\tutSimpleColor\.\Debug\tutSimpleColor.dll
========== Erstellen: 1 erfolgreich, 0 fehlerhaft, 0 aktuell, 0 übersprungen ==========
```

Und im Debug Ordner sollte sich jetzt eine DLL tummeln:

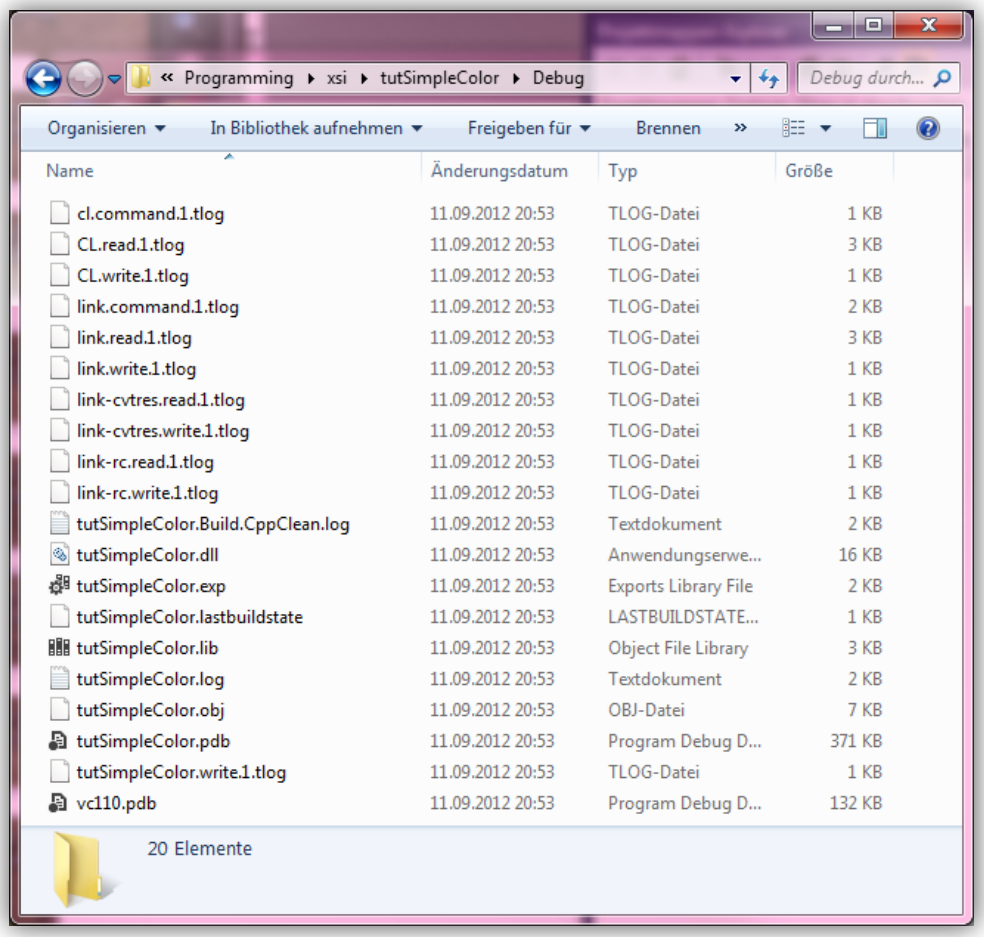


Abbildung 15 - Verzeichnis von 32bit Debug nach dem Test Compile

Das ist sie ☺ Aber nur 32bit und vollgestopft mit Debug Informationen … erst einmal egal

Projekt in neuer Form speichern

Wir sollten danach unbedingt das Projekt speichern. Im Menü auf Datei – oder Noch-Neurerer-Deutsch DATEI – den Punkt Alle Speichern wählen. Es macht sich dann ein Dialog auf innerhalb unseres Projekt Verzeichnis auf und er bittet uns den Namen einer sln Datei zu vergeben. SLN ist das neue Visual Studio Format. Wir lassen es einfach so und sagen Ja.

Damit wir keine Probleme später durch einen Unfall erleiden müssen, sollten wir jetzt die vcproj löschen. Aber nicht die vcxproj und/oder die vcxproj.user Datei !!! Für das Projekt reicht ein Doppelklick auf die SLN Datei dann aus.

Projekt Anpassungen für Release Win32

Was wir aber viel mehr brauchen ist das Release für die 32bit Version. Dazu öffnen wir unsere Projekteigenschaften und schalten oben links die Konfiguration von Aktiv(Debug) auf Release um. Danach ändern sich alle Parameter wieder. In Konfigurationseigenschaften -> Allgemein sollte jetzt als Ausgabe Verzeichnis .\Release\ stehen. Wenn man jetzt aber F7 drückt, meckert er wieder dass er die Header nicht findet. Nun ja das kennen wir bereits. Hier die Kurzform:

- VC++ Verzeichnisse -> Include Verzeichnis anpassen (siehe Seite 19)
- VC++ Verzeichnisse -> Bibliothekten Verzeichnis anpassen (siehe Seite 21)
- Linker -> Allgemein -> Ausgabe. Aber mit .\Release\ anstatt wie auf Seite 23 mit Debug

Wenn man es nun kompiliert mit F7 sollte alles OK sein. Soweit dazu

Projekt Anpassungen für Debug Win64

Es ist das gleiche Spiel wie bei Release Win32. Wir wählen den Projekt Eigenschaften Debug x64 aus. Aber jetzt Vorsicht ! Wir brauchen die Include Dateien und Lib Dateien vom 64bit SoftImage! Aber ansonsten ist es der gleiche Vorgang wie oben ☺

Der Linker Output wird mit .\Debugx64\ erweitert

Aber wenn man nun Kompiliert bekommt man einen lustigen Fehler:

```
1>.\Debugx64/tutSimpleColor.obj : fatal error LNK1112: Modul-Computertyp "X86" steht in Konflikt
mit dem Zielcomputertyp "x64".
```

Wir müssen nur noch die Plattform ändern. In unserem Projekt Eigenschaften Dialog ist rechts oben ein Button mit der Aufschrift Konfigurationsmanager.

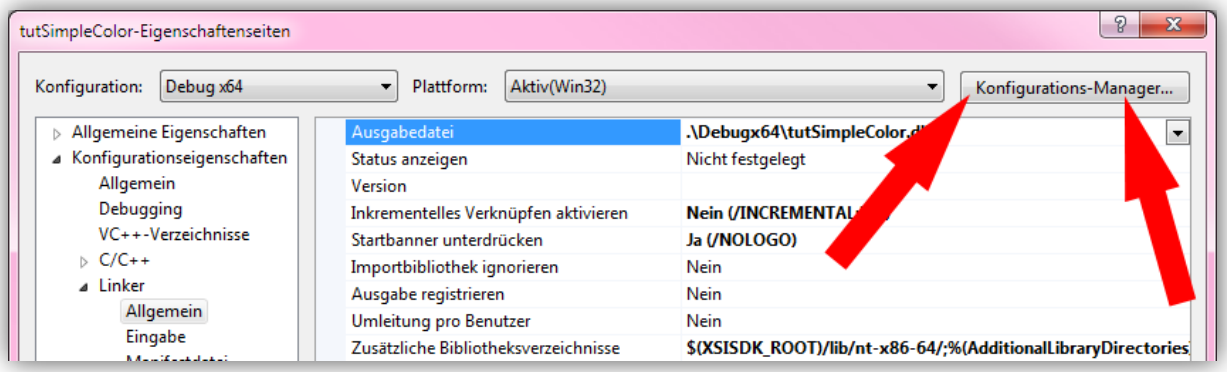


Abbildung 16 - Konfigurations-Manager Button für 64bit

Den klicken wir und es öffnet sich ein Fenster. In diesem Fenster wählen wir in Aktive Projektmappenplattform den Eintrag Neu

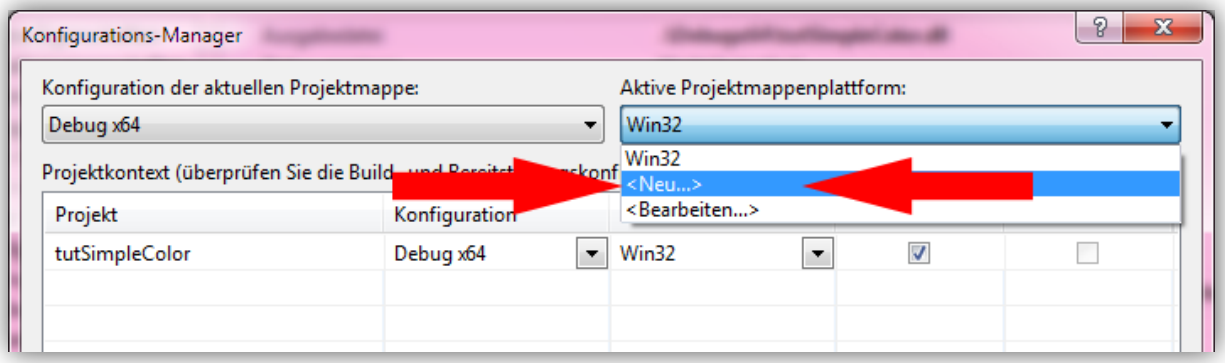


Abbildung 17 - Konfigurations-Manager neue Plattform für 64bit hinzufügen

Und es geht ein weiteres Fenster auf. Dort stellen wir den x64 ein

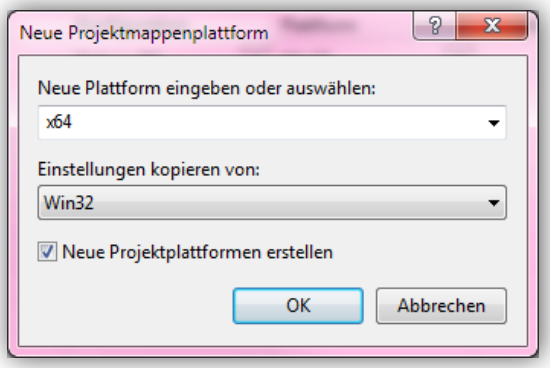


Abbildung 18 - Zielplattform auf 64bit einstellen

und klicken auf OK. Danach baut er alles für 64bit um. Wobei mir bei der Erstellung dieser Seite aufgefallen ist, dass er nun vollkommen andere Output Verzeichnisse verwendet. Wir müssen also unser Linker -> Allgemein -> Ausgabedatei noch einmal anfassen und ein x64\Debug x64, mit Leerzeichen, da rein basteln. Das ganze sieht dann so aus:

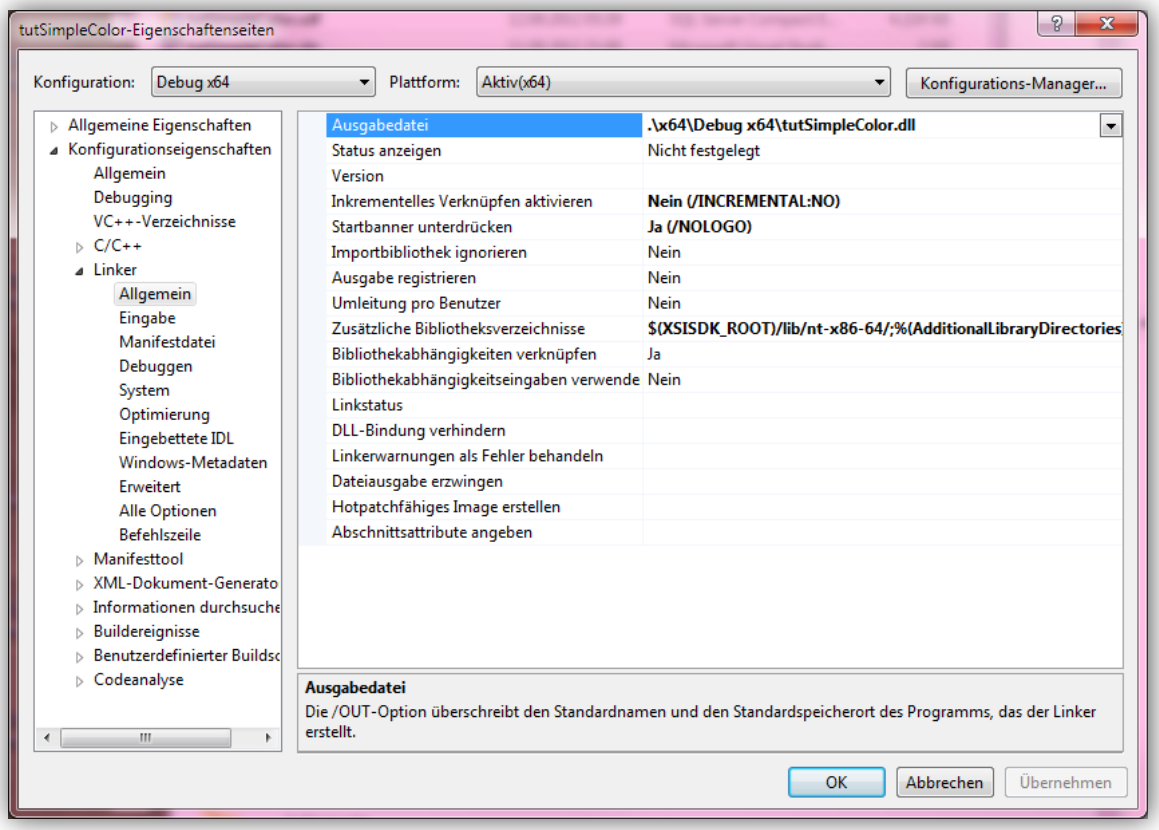


Abbildung 19 - Debug 64bit in 64bit Umgebung mit angepasster Ausgabedatei

Projekt Anpassungen für Release Win64

Man ahnt es schon … oder? Ja-ha genau … Bitte das gleiche noch einmal für die Release. Der Linker Ausgabe Ort ist dann aber ein .\x64\Release x64.

Projekt Anpassungen für Batch Lauf

So … nachdem wir nun 4 Targets gebaut haben, hätten wir gerne das alle 4 Targets auch in einem Rutsch gebaut werden. Dazu klicken wir im Menü auf ERSTELLEN und dann unten auf Batch erstellen. Es öffnet sich dann ein Fenster wo wir einstellen können, welche Targets er bauen soll.

Wir wollen von dem angebotenen Punkten aber nur 4:

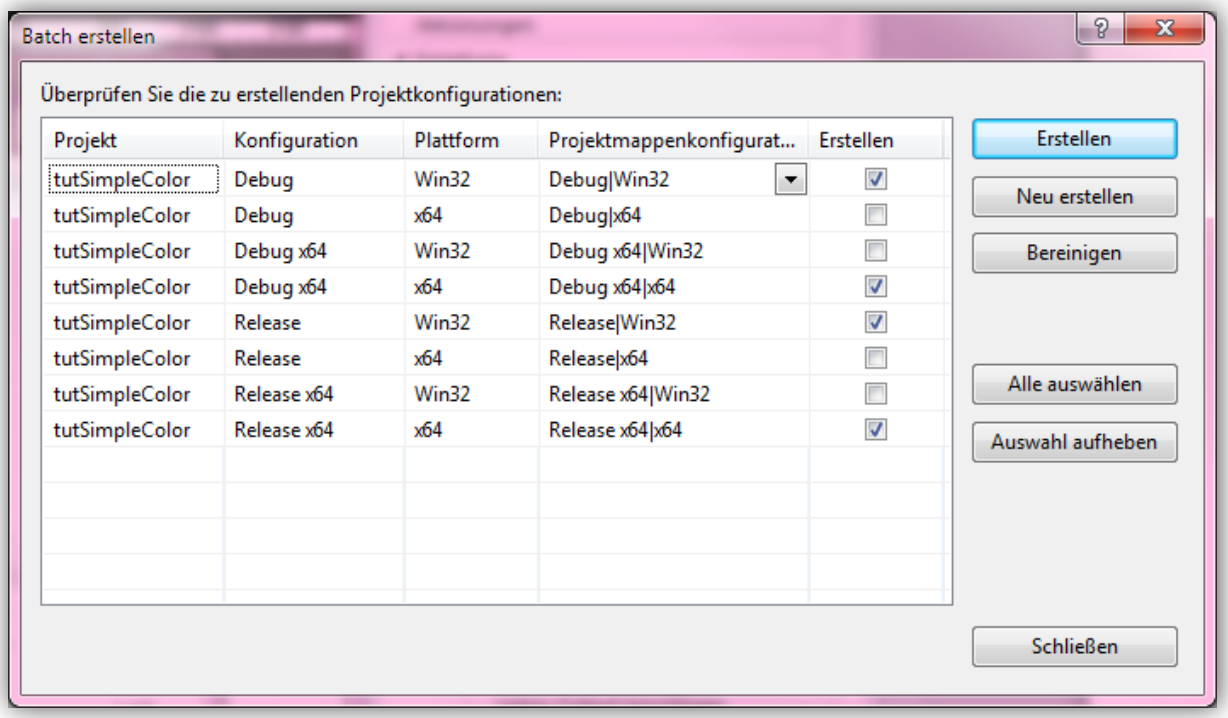


Abbildung 20 - Einstellungen zum Batchlauf

Wenn wir dann auf Erstellen klicken, werden alle Targets erstellt. Zum Glück merkt er sich unsere Einstellungen für das nächste Mal. Man sollte einmal zur Sicherheit auf Neu Erstellen klicken und sich den Output anschauen. Kommt so etwas:

```
========== Alles neu erstellen: 4 erfolgreich, 0 fehlerhaft, 0 übersprungen ==========
```

Hat man alles richtig gemacht!

Installation des SPDL und der DLL in SoftImage

Leider kann man mal nicht einfach so den Shader aktualisieren im laufendem SoftImage Betrieb. Man muss es immer Restarten. Was die Sache relativ nervig macht.

Ich weiß aber auch dass wir bisher noch keine einzige Zeile programmiert haben. Ich möchte aber dieses Thema vorziehen um den späteren Dokumentfluss nicht mit Belanglosigkeiten, wie der Installation, zu stören. Ich habe dieses Thema explizit wieder nach oben – also hier her – verschoben, weil ich es da unten überhaupt nicht mochte.

Wir werden jetzt unseren Null Shader, der nichts tut außer zu existieren, in SoftImage installieren. Danach zeige ich euch wie wir den Updaten und wie man es am coolsten machen kann um den Developer-Turn-Around möglichst klein zu halten.

Als erstes brauchen wir ein leeren Temporäres Verzeichnis, wo ist vollkommen egal. In dieses Verzeichnis kopieren wir aus unserem Projekt die SPDL Datei und die zur SoftImage Version passende DLL Datei aus den Verzeichnissen. Sprich am Schluss liegen in unserem Temporären Verzeichnis zwei Dateien:

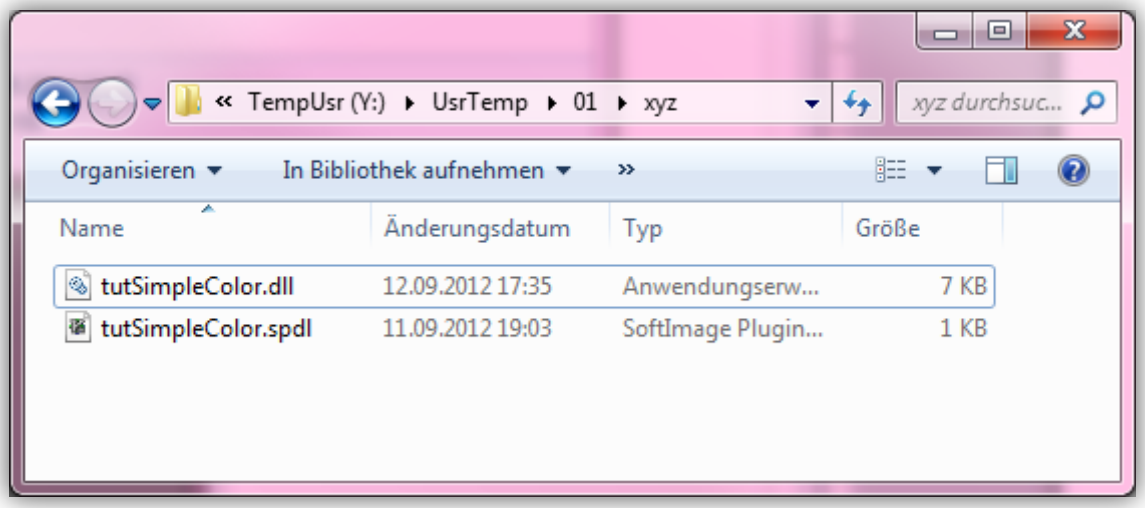


Abbildung 21 - Inhalt unseres Temporären Verzeichnisses

Danach öffnen wir unser SoftImage und gehen wieder in den Plugin-Manager. Dort wählt man den bekannten SPDLs Tab, das hatten wir auf Seite 10 schon einmal. Jetzt wählt man Install und wählt den Temporären Path aus. Das Ziel der Installation soll unser User sein und Klicken dann auf den Install Knopf:

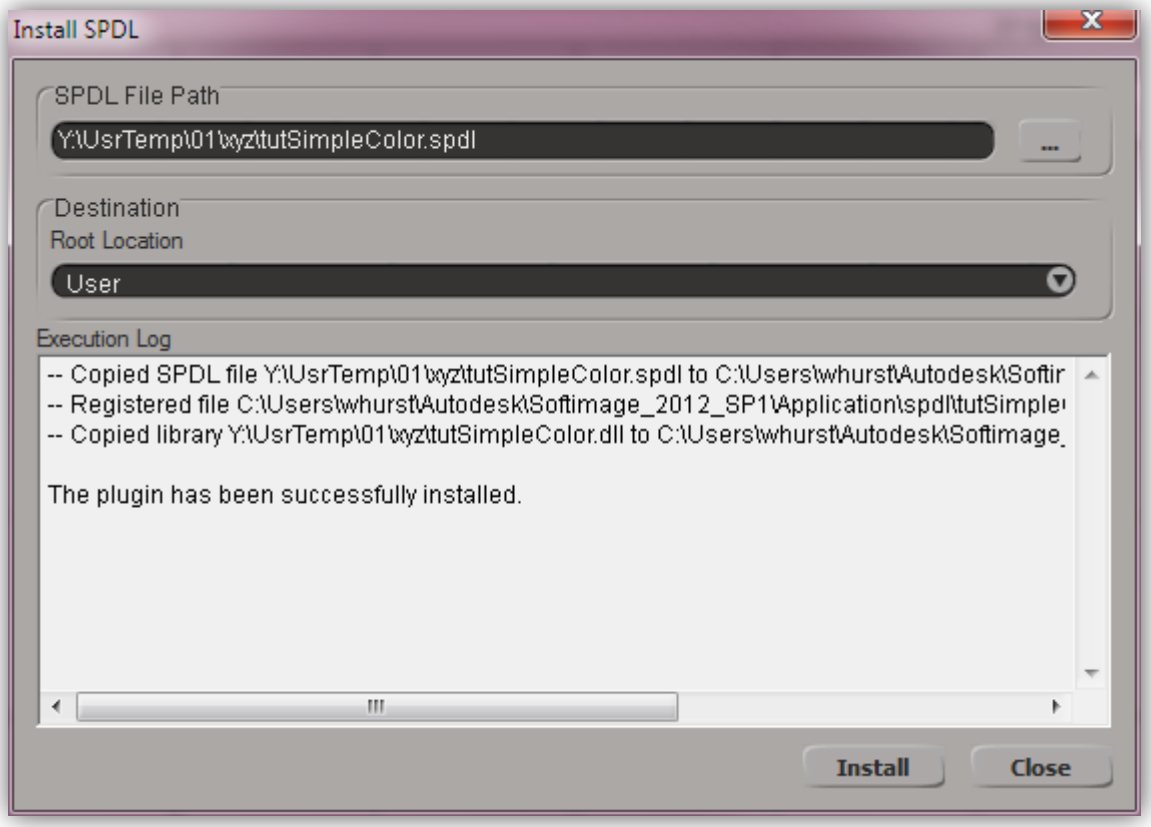


Abbildung 22 - Installation unseres Null Shaders in SoftImage

Man sieht dass er das SPDL und die DLL in unseren User Verzeichnis kopiert. Das Verzeichnis wohin er es kopiert sollte man sich merken. Das Temporäre Verzeichnis können wir nun wieder wegwerfen. Zum Testen ob unser Shader auch von SoftImage genommen wird müssen wir SoftImage jetzt verlassen und neu starten. Ja das wird uns noch Nerven kosten. Danach öffnet man den Material Editor und wählt mal das Scene Default Material aus und sucht mal nach tut im Suchfeld. Es taucht dann auf. Wir ziehen es in unser Rendertree und Doppelklicken es. Vóilà … alles da:

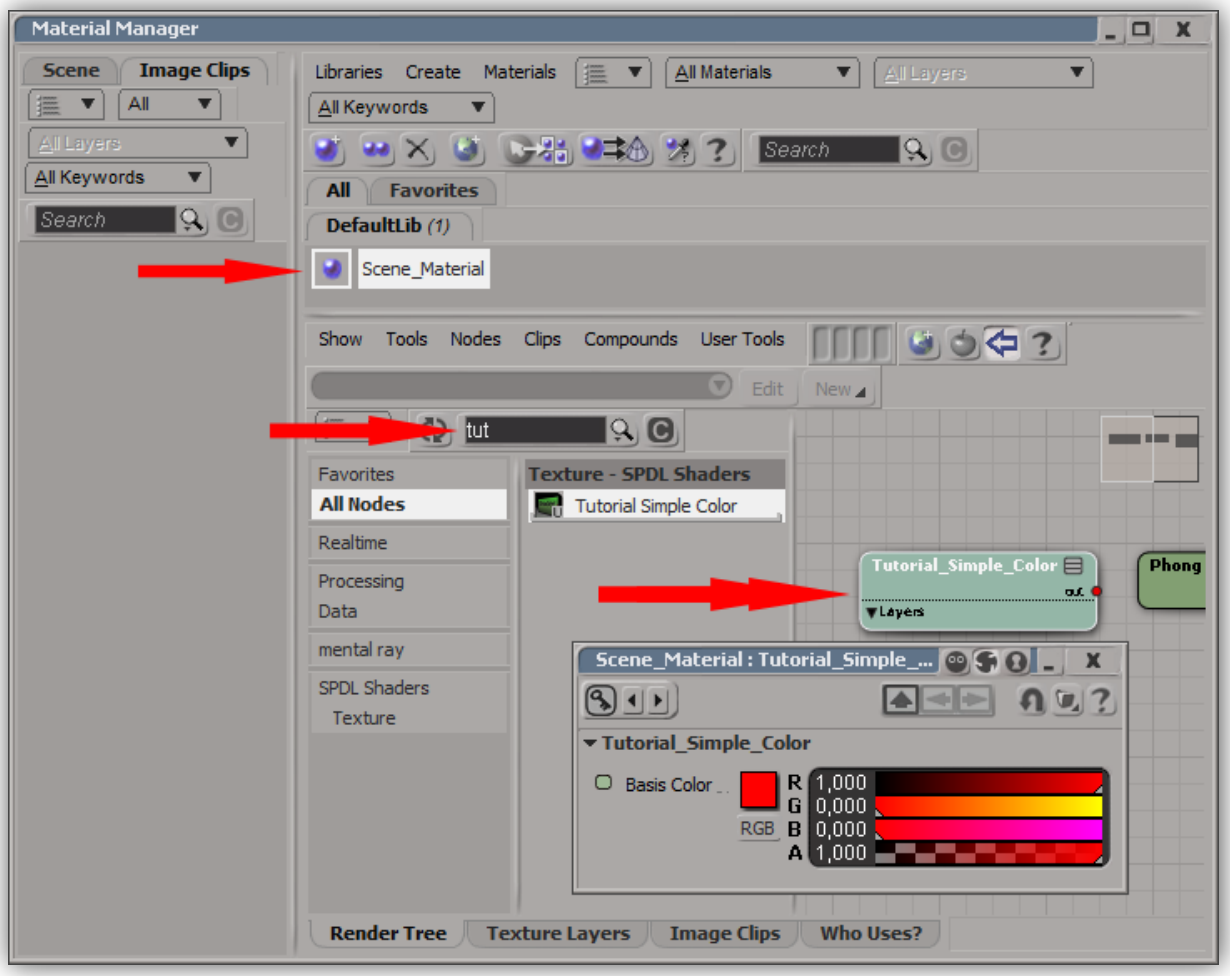


Abbildung 23 - Unser Null Shader im SoftImage Render Tree

Updaten des bereits installieren Shaders

Diesen Schritt will ich unbedingt vorher, bevor wir anfangen, machen. Wenn man nun ein Shader installiert hat, gibt es natürlich die Möglichkeit diesen im Plugin Manager zu deinstallieren um ihn dann wie oben beschrieben wieder zu installieren. Das ist aber – gelinde gesagt – sehr doof und umständlich. Nein wir tun das anders.

Wir beenden SoftImage und kopieren aus unserem Visual C++ Projekt Verzeichnis die SPDL und die DLL Datei einfach in unser SoftImage User Verzeichnis hinein und überschreiben die alten Dateien. Für das was wir hier in diesem Dokument tun werden, reicht das vollkommen hin. In der Abbildung 22 sagt er uns ja wohin er die kopiert, also C:\users\....

Test Szene erstellen

Wir brauchen unbedingt zum Testen eine Test Szene. Man kann sich eine eigene Bauen, oder folgendes im Script Editor zur Ausführung bringen:

```
GetViewCamera(1, "Camera");
SetDisplayMode("Camera", "shaded");
SetValue("Camera.camdisp.headlight", true, null);
SetValue("Camera.camdisp.wireontopunsel", true, null);
CreatePrim("Grid", "MeshSurface", null, null);
SetValue("grid.grid.ulength", 20, null);
SetValue("grid.grid.vlength", 20, null);
CreateProjection("grid", siTxtPlanarXZ, siTxtDefaultPlanarXZ, null, "Texture_Projection", null, null, null);
CreatePrim("Cylinder", "MeshSurface", null, null);
SetValue("cylinder.cylinder.height", 2, null);
SetValue("cylinder.cylinder.radius", 2, null);
SetValue("cylinder.polymsh.geom.subdivu", 24, null);
SetValue("cylinder.polymsh.geom.subdivv", 1, null);
SetValue("cylinder.polymsh.geom.subdivbase", 1, null);
CreateProjection("cylinder", siTxtCylindrical, siTxtDefaultCylindrical, null, "Texture_Projection", null, null,
null);
TranslatePivot("cylinder", 0, -1, 0);
Translate(null, 10, 0, 0, siAbsolute, siView, siObj, siY, true, null, null, null, null, true, null, -1, null,
0, null);
CreatePrim("Cube", "MeshSurface", null, null);
SetValue("cube.cube.length", 2, null);
CreateProjection("cube", siTxtCubic, siTxtDefaultCubic, null, "Texture_Projection", null, null, null);
Translate(null, 0, 0.040099980219741, 0, siRelative, siView, siObj, siXYZ, null, null, null, null, null, null,
null, null, null, null, 0, null);
TranslatePivot("cube", 0, -0.959900019780259, 0);
Translate(null, 10, 0, 0, siAbsolute, siView, siObj, siY, true, null, null, null, null, true, null, -
0.959900019780259, null, 0, null);
Translate(null, -5.88420360721443, 0, 2.46959417381453, siRelative, siView, siObj, siXYZ, null, null, null,
null, null, true, null, null, null, 0, null);
Rotate(null, 0, -33.3232405938332, 0, siRelative, siAdd, siObj, siXYZ, null, null, null, true, -
5.88420360721442, null, 2.46959417381453, 0, null);
CreatePrim("Sphere", "MeshSurface", null, null);
SetValue("sphere.sphere.radius", 3, null);
SetValue("sphere.polymsh.geom.subdivu", 24, null);
SetValue("sphere.polymsh.geom.subdivv", 12, null);
CreateProjection("sphere", siTxtSpherical, null, null, "Texture_Projection", null, null, null);
TranslatePivot("sphere", 0, -2.93185165257814, 0);
Translate(null, 0, 5.77439715164259E-02, 0, siRelative, siView, siObj, siY, null, null, null, null, null, true,
null, -2.93185165257814, null, 0, null);
TranslatePivot("sphere", 0, -2.94225602848357, 0);
Translate(null, 1.96592582628907, 0, -0.25881904510252, siAbsolute, siView, siObj, siY, true, null, null, null,
null, true, null, -2.94225602848357, null, 0, null);
Translate(null, 4.88688096192385, 0, 3.91642712413012, siRelative, siView, siObj, siY, null, null, null, null,
null, true, null, null, null, 0, null);
SelectObj("light", null, null);
DeleteObj("light");
GetPrimLight("Infinite.Preset", "Infinite", "", null, null, null);
Translate(null, 0, 0, 14.0779584072338, siRelative, siView, siObj, siXYZ, null, null, null, null, null, null,
null, null, null, null, 0, null);
Translate(null, -8.78506891194228, 0, 0, siRelative, siView, siObj, siXYZ, null, null, null, null, null, null,
null, null, null, null, 0, null);
Translate(null, 0, 5.68994497646868, 0, siRelative, siView, siObj, siXYZ, null, null, null, null, null, null,
null, null, null, null, 0, null);
Rotate(null, -26.7622969405865, 0, 0, siRelative, siAdd, siObj, siXYZ, null, null, null, null, null, null,
null, 0, null);
Rotate(null, 0, -40.1095979419455, 0, siRelative, siAdd, siObj, siXYZ, null, null, null, null, null, null,
null, 0, null);
SelectObj("grid", null, true);
ApplyShader("$XSI_DSPRESETS\\Shaders\\Material\\Constant.Preset", "", null, "", siLetLocalMaterialsOverlap);
SetValue("Sources.Materials.DefaultLib.Material.Name", "TestMaterial", null);
SelectObj("cube", null, true);
ToggleSelection("cylinder", null, true);
ToggleSelection("sphere", null, true);
AssignMaterial("grid.TestMaterial,cube,cylinder,sphere", siLetLocalMaterialsOverlap);
```

Resümee

So, ich habe euch durch die Hölle geschickt. Aber wir haben in der kurzen Zeit ganz viel gelernt, obwohl wir noch nicht einmal eine Zeile Code eingegeben haben. Unter UNIX wären wir schon 18 Mal fertig, dafür haben die Windows Leute eine schönere GUI … hahaha

Aber was wir nun wissen ist:

1. Mit SoftImage erstellen wir ein Template des Shaders mit allen Parametern und dem Layout
2. Mit Visual C++ biegen wir das Projekt für Debug/Release und 32/64bit zurecht
3. Mit SoftImage importieren und registrieren wir unseren Shader
4. Den installierten Shader updaten, wenn man ihn neu gebaut hat

Diese vier Schritte haben wir jetzt im Blut. Das schöne ist, das wir uns nun um die Programmierung kümmern können ohne lästige Unterbrechungen. Und es geht auch gleich los.

Diese Seite ist leer

Das Dokument ist für die Anzeige von zwei Seiten optimiert

mentalRay Shader Programmierung

Durch den SDK Wizard von SoftImage bekommen wir bereits einen vollkommen fertigen Rumpf geliefert. Den wollen wir uns mal anschauen. Dabei möchte ich die Teile, die wir später oder überhaupt nicht in diesem Dokument bearbeiten vorziehen.

Der Rumpf des Shader Quelltext im Detail

Init, Exit und Versions Code

Man erinnert sich dunkel, bei dem SDK Wizard gab es eine Checkbox ob wir Init/Exit Funktion generieren will, siehe Seite 11. Diese Funktionen sehen nun wie folgt aus:

```
26 extern "C" DLLEXPORT void
27 tutSimpleColor_init
28 (
29         miState                         *state,
30         tutSimpleColor_t        *params,
31         miBoolean                       *inst_init_req
32         )
33 {
34         if( params == NULL )
35         {
36                 // TODO: Shader global initialization code goes here (if needed)
37
38                 // Request a per-instance shader initialization as well (set to
miFALSE if not needed)
39                 *inst_init_req = miTRUE;
40         }
41         else
42         {
43                 // TODO: Shader instance-specific initialization code goes here (if
needed)
44         }
45 }
46
47
48 extern "C" DLLEXPORT void
49 tutSimpleColor_exit
50 (
51         miState                         *state,
52         tutSimpleColor_t        *params
53         )
54 {
55         if( params == NULL )
56         {
57                 // TODO: Shader global cleanup code goes here (if needed)
58         }
59         else
60         {
61                 // TODO: Shader instance-specific cleanup code goes here (if needed)
62         }
63 }
64
65
66 extern "C" DLLEXPORT int
67 tutSimpleColor_version( )
68 {
69         return( 1 );
70 }
```

Sollte ein Shader zum Beispiel bei einer Initialisierung globalen Speicher oder wilde Rechnungen veranstalten, dann gehören die dort rein. Da wir in unserem Beispiel das nicht tun, gehe ich auch nicht weiter auf die Funktionen ein

Kopf der Datei

Der Kopf der Datei includiert die shader.h von mentalRay und unsere eigene Header Datei.

```
1  // C++ Source Code Generated by Softimage Shader Wizard
2
3  ////////////////////////////////////////////////////////////////
4  // Includes
5
6  #include <shader.h>
7  #include "tutSimpleColor.h"
8
9  ////////////////////////////////////////////////////////////////
10 // Implementation
11
```

Unsere eigene Header Datei

Unsere Headerdatei definiert eine Struct wo wir unsere Parameter die wir im SDK Wizard erstellt haben wiederfinden. Auf Seite 13 haben wir eine Farbe hinzugefügt. Die ist nun auch hier:

```
1  // C Header File Generated by Softimage Shader Wizard
2
3  #ifndef TUTSIMPLECOLOR_H
4  #define TUTSIMPLECOLOR_H
5
6  #include <shader.h>
7
8  ////////////////////////////////////////////////////////////////
9  // Type definition
10
11 typedef struct
12 {
13         miColor                         baseColor;                      // Basis Color
14 } tutSimpleColor_t;
15
16 #endif // TUTSIMPLECOLOR_H
```

Unsere Farbe hat den Datentyp miColor. Das ist ein miScalar für r, g, b und a, wobei ein miScalar ein einfacher float ist. Das steht unter anderem in der mentalRay shader.h

Shader Haupteinsprung Funktion

So jetzt kommen wir endlich mal zu unserem Shader und zu der Funktion um die es geht. Die Funktion wird bei jedem Ray aufgerufen, wird Gedenken an meinen warnenden Ausführungen auf Seite 6

Hier die Funktion um die es geht:

```
12 extern "C" DLLEXPORT miBoolean
13 tutSimpleColor
14 (
15         miColor                         *result,
16         miState                         *state,
17         tutSimpleColor_t        *params
18         )
19 {
20         // TODO: Shader main code goes here
21
22         return( miTRUE );
23 }
```

Auch wenn man es überhaupt nicht lesen kann, wer auch immer diese Formatierung erfunden hat aber den sollte man mit Lisp bestrafen. Unsere Funktion heißt tutSimpleColor und wirft ein miBoolean zurück und bekommt von mentalRay drei Parameter

Rückgabe Wert unseres Shaders

Der Rückgabewert ist ein miBoolean, ein miBoolean ist ein einfacher int. Man sollte aber entweder miTRUE oder miFALSE verwenden. Wenn man miTRUE zurück gibt, signalisiert man mentalRay das wir unsere Arbeit erfolgreich gemacht haben.

Übergabe Parameter: result

Result ist ein Zeiger auf ein miColor. Wir haben im SDK Wizard auf Seite 11 gesagt, dass wir gerne eine Farbe zurückgeben wollen. Wenn wir was anderen gesagt hätten, würde dort jetzt auch ein anderer Typ sein. Diesen result Zeiger, da es nicht viel Rechenkraft kostet, sollten wir immer gegen NULL testen, wer weiß … Das gilt im Übrigen für alle Zeiger die wir von unbekannten bekommen.

Wenn unser Shader fertig ist, legen wir in result unser Ergebnis ab und gehen mit miTRUE aus der Funktion raus. mentalRay weiß dann, dass die Farbe dann da irgendwo im Speicher liegt

Übergabe Parameter: state

State ist ein Zeiger auf eine sehr große Struct die alle nötigen Informationen beinhaltet. Da unsere Funktion immer nur per Ray aufgerufen wird, finden wir in diesem state die Informationen wo wir sind, welchen Render Modus wir haben und vieles mehr. In der shader.h von mentalRay findet sich eine quasi endlose Liste von Parametern. Zur Sicherheit sollten wir diesen Zeiger auch gegen NULL testen.

Übergabe Parameter: params

In diesem Parameter bekommen wir unsere Struct geliefert, in der unsere Parameter die wir in der SoftImage GUI eingestellt haben. Auch diesen Zeiger sollten wir vorher prüfen. Die Parameter müssen mit den mi_eval_* Funktionen lokal geholt werden. Ein direkter Zugriff auf params kann und wird fatale Folgen haben!

Unser Test Shader gibt was zurück

Wir wollen uns jetzt erst einmal mit dem Rückgabewert beschäftigen. Wir programmieren unseren Shader jetzt mal so um das er nur die Farbe Grün zurückgibt. Dabei ignorieren wir mal den Input Parameter

```
12 extern "C" DLLEXPORT miBoolean
13 tutSimpleColor
14 (
15         miColor                         *result,
16         miState                         *state,
17         tutSimpleColor_t        *params
18         )
19 {
20         // TODO: Shader main code goes here
21
22         if (result == NULL) return (miTRUE);
23         if (state == NULL) return (miTRUE);
24         if (params == NULL) return (miTRUE);
25
26         result->r = 0.0f;
27         result->g = 1.0f;
28         result->b = 0.0f;
29         result->a = 1.0f;
30
31         return( miTRUE );
32 }
```

Danach gehen wir hin und erstellen im Batch alle Targets wie auf Seite 29 beschrieben. Man achte bitte auf die Checkboxen, bei mir hat er - durch Zufall - welche vergessen. Danach updaten wir den Shader wie auf Seite 33 beschrieben. Und lassen unsere Szene mit eingehängten Shader rendern:

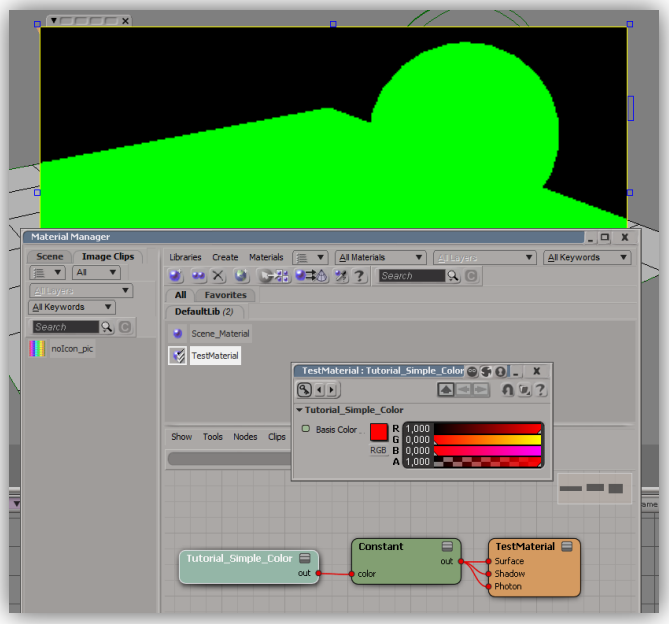


Abbildung 24 - Ergebnis nach der Rückgabe der Farbe Grün

Der aufmerksame Leser wird feststellen: Da haut doch was nicht hin. Genau ☺ Da ist was faul.

Der einfachste Shader überhaupt

Ja im obigen Beispiel haben wir ja auch nur die Farbe Grün in das result geschrieben. Den Input Parameter haben wir weder Verarbeitet, noch wird er von mentalRay benutzt. Warum sollte er. Was wir jetzt machen müssen ist uns die Farbe aus dem Parameter holen. Dazu haben wir ja den Übergabe Parameter params, da liegt ja unsere Farbe drin.

Um dort aber an die Farbe zu kommen, muss man sich den Parameter lokal via mentalRay ziehen. Das funktioniert mit den mi_eval Funktionen. Wir dürfen nicht direkt auf die params zugreifen! Immer daran denken!

Um sich die basisColor zu holen, wir wissen bereits es ist ein miColor, benutzen wir auch mi_eval_color in folgender Form:

```
miTYP meineKleineLokaleVariable = *mi_eval_TYP (&params->derParameterVomSDKWizardUndDerStruct);
```

Wenn wir dann die Farbe haben, können wir diese in unser result übertragen:

```
26          miColor baseColor = *mi_eval_color (&params->baseColor);
27
28          result->r = baseColor.r;
29          result->g = baseColor.g;
30          result->b = baseColor.b;
31          result->a = baseColor.a;
```

Man beachte dass baseColor lokal zwar genauso heißt wie der Parameter, aber er hat nichts mit dem Parameter zu tun. Durch mi_eval_* holen wir den Wert des Parameters quasi aus dem Render Tree erst ab. Der liegt da nämlich nicht so rum. Das Ergebnis nach dem Update ist nun:

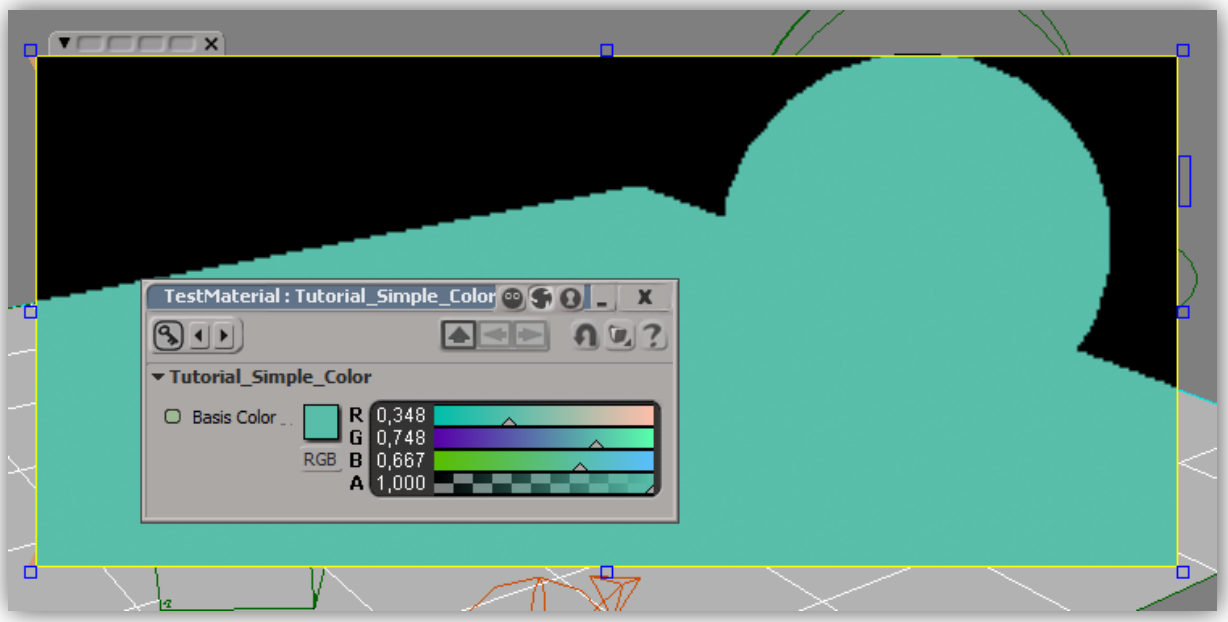


Abbildung 25 - Resultat nachdem der Input mit dem Output von uns verbunden wurde

Es geht! Die Farbe die wir einstellen, holen wir uns im Shader in unsere lokale Variable und werfen die Werte in das result.

Hier noch einmal die komplette Shader Funktion:

```
12 extern "C" DLLEXPORT miBoolean
13 tutSimpleColor
14 (
15         miColor                         *result,
16         miState                         *state,
17         tutSimpleColor_t        *params
18         )
19 {
20         // TODO: Shader main code goes here
21
22         if (result == NULL) return (miTRUE);
23         if (state == NULL) return (miTRUE);
24         if (params == NULL) return (miTRUE);
25
26         miColor baseColor = *mi_eval_color (&params->baseColor);
27
28         result->r = baseColor.r;
29         result->g = baseColor.g;
30         result->b = baseColor.b;
31         result->a = baseColor.a;
32
33         return( miTRUE );
34 }
```

Das war zum einen Einfach und Langweilig ☺

Wir wollen nun mal zwei Farben durch Zufall mischen. Wir brauchen aber dafür ein neuen Input Parameter und ein Zufalls“ding“ wo wir dann entscheiden können welche der beiden Farben wir in unser result tun werden. Aber kümmern wir uns erst einmal um weitere Parameter.

Parameter manuell im SPDL hinzufügen

Als erstes öffnen wir unser SPDL, es liegt mit im Projekt von Visual C++. Wir haben hier mehrere Bereiche, was aber gleich ins Auge fällt sind diese GUID Dinger. Um die kümmern wir uns mal.

GUID

GUID ist ein Hashwert der bei der Erstellung eines solchen quasi Weltweit einmalig ist. Selbst wenn zwei Leute gleichzeitig eine GUID generieren ist sie unterschiedlich. Ok sagen wir so, die Wahrscheinlichkeit dass es zwei identische GUIDs gibt ist so dermaßen Gering, das man es vernachlässigen kann. In die GUID wird normalerweise das Datum, die Zeit, die Netzwerk Adresse und hin und wieder auch der Benutzername gehasht und noch einmal gehasht und gehasht. Da schon allein Zeit (Mikrosekunden) und Netzwerk Adresse (MAC) schon ziemlich einzigartig sind sind die anderen Werte nur noch Kosmetik.

Diese Einzigartigkeit benutzt SoftImage aber für die Verbindungen zwischen den Shadern und für den Shader selbst. SoftImage merkt sich nicht etwa den Namen, das wäre ja auch blöd, nein er kennt nur die GUID. Kann man einfach ausprobieren indem man in Zeile 4 die GUID ändert und das SoftImage Test Szene läd, es fliegt einem sofort um die Ohren. Gleiches gilt für Output und Input.

Das bedeutet im Umkehrschluss aber auch, solange wir die existierenden GUID nicht ändern, können wir da drinnen machen was wir wollen. Und genau das haben wir vor.

GUID selbst generieren

Es wird ein GUID Generator für Windows beim Visual C++ mitgeliefert. Dazu findet man im Menü TOOLS den Punkt GUID generieren. Einmal angeklickt öffnet sich ein Programm wo wir munter GUIDs erstellen können. Unter UNIX gibt es auf der CLI uuidgen und das generiert auch gleich mehrere wenn man will.

Überprüfen der SPDL

Das manuelle Hinzufügen von Parametern, oder anderes, ohne den SDK Wizard ist natürlich gespickt mit Fallen und Stolpersteinen. Es gibt ein SoftImage CLI Programm mit dem wir die SPDL prüfen können. Dazu öffnen wir einen CLI, unter Windows gibt es speziell dafür ein Eintrag im Startmenü – Windows 8 User dürften jetzt anfangen zu weinen:

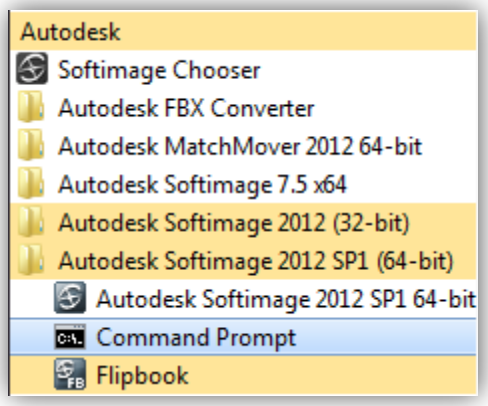


Abbildung 26 - Starten der CLI unter Windows im SoftImage Environment

Das Programm nennt sich spdlcheck und will den Namen der SPDL Datei. Unter Windows 7 können wir die Datei mit Drag und Drop in die CLI ziehen. Das geht aber leider nur vom Explorer aus. Das ganze sieht dann etwa so aus:

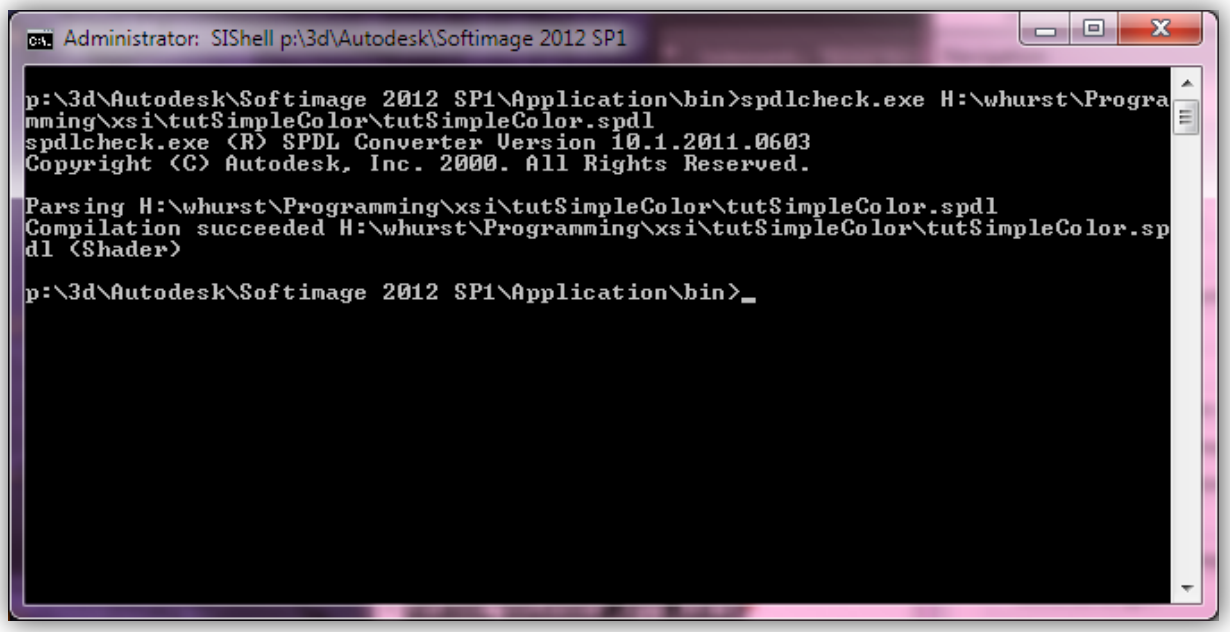


Abbildung 27 - spdlcheck unter Window

Wenn unser SPDL nicht korrekt ist, wird uns SoftImage beim Starten sehr stark anmeckern und auch gleich den Render Tree entsorgen, dann auf keinen Fall speichern. Die meisten Tippfehler kommen von vergessenen Klammern und Semikolons. So jetzt sind wir bereit für einen weiteren Parameter.

Kopieren eines Inputs

Wir wollen den bereits existierenden Parameter im Bereich PropertySet mit dem Namen baseColor kopieren, bzw. duplizieren. Dazu kopieren wir den Block einfach da drunter. Wir müssen aber den Namen des Parameters ändern und natürlich auch die GUID. Mit Value legen wir das Default fest. Da nehme ich mal Blau. Der ganze PropertySet Block sieht dann so aus:

```
 5 PropertySet "tutSimpleColor_pset"
 6 {
 7         Parameter "out" output
 8         {
 9                 GUID = "{28DAA117-D2C8-41C0-8ED9-0705D7B40988}";
10                 Type = color;
11         }
12         Parameter "baseColor" input
13         {
14                 GUID = "{8C3D1C69-BBFC-48CC-A8A3-EF404C169E48}";
15                 Type = color;
16                 Value = 1.0 0.0 0.0 1.0;
17         }
18         Parameter "secondaryColor" input
19         {
20                 GUID = "{0065048D-A567-4ADB-8BD2-A103177998AE}";
21                 Type = color;
22                 Value = 0.0 0.0 1.0 1.0;
23         }
24 }
```

Die GUID generiert man selbst, die wird sich jetzt vermutlich von meiner Unterscheiden. Das ist zumindest dann normal.

GUI Typen spezifizieren

Bisher weiß SoftImage nur, das es ein Input Typen mit dem Namen secondaryColor gibt. Das würde sogar gehen. Aber wir sehen ihn nicht und können auch keine Werte dort reinlaufen lassen. Das wird gerne von internen Variablen gemacht, man kann im SPDL auch Programmieren – was wir hier jetzt aber nicht machen. Damit SoftImage weiß wie er den Typ darstellen soll, müssen wir es ihm sagen in dem wir einfach im Bereich Defaults unsere baseColor kopieren:

```
41 Defaults
42 {
43         baseColor
44         {
45                 Name = "Basis Color";
46                 UIType = "rgba";
47         }
48         secondaryColor
49         {
50                 Name = "Second Color";
51                 UIType = "rgba";
52         }
53 }
```

Jetzt weiß SoftImage das er es darstellen sollte mit den 4 Farbbalken und es soll im Dialog die Bezeichnung Second Color stehen

Layout definieren

Ganz so glücklich ist unser SoftImage noch nicht. Wir haben zwar jetzt gesagt wie es aussehen soll, aber noch nicht wo wir es haben wollen. Wir müssen den Parameter in das Layout hinzufügen, wenn wir wollen dass es in der GUI zu sehen ist, und das wollen wir ja. Im Bereich Layout fügen wir es also einfach dazu:

```
55 Layout "Default"
56 {
57          baseColor;
58          secondaryColor;
59 }
```

Updaten einer existierenden und bereits installierten SPDL

Beim Updaten der SPL macht man das gleiche wie beim Updaten der DLL siehe Seite 33 nur das wir es jetzt in das SPDL Verzeichnis werfen. Unter Windows sollte man sich diese Verzeichnisse als Verknüpfung auf den Desktop legen und es dort einfach reinwerfen. So mach ich es und das klappt ganz gut. Zur Not scheibt man sich ein Script.

Unser neues SPDL updaten wir jetzt einmal und bestaunen unser Werk:

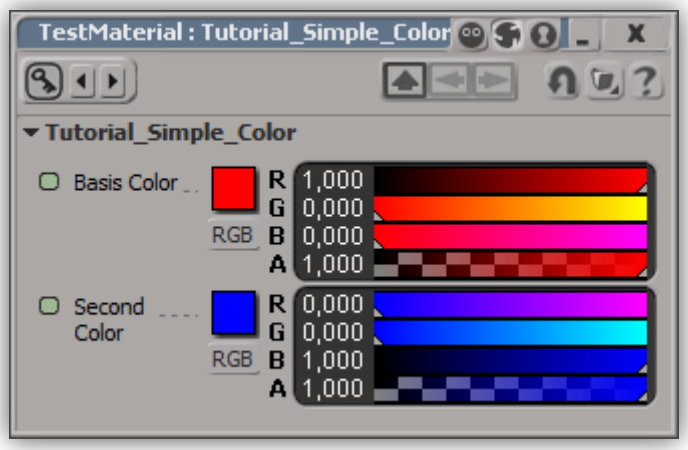


Abbildung 28 - Resultat der zwei Farben nach der SPDL Erweiterung

Parameter manuell im Header hinzufügen

Was wir gerade getan haben kann unter Umständen massiv nach hinten losgehen. Wir haben ein Input definiert, aber unseren alten Shader benutzt, der davon gar nichts weiß. Das ist verdammt grob fahrlässig und sollte man nie tun.

Wir müssen diesen Parameter mit in unsere Struct einbauen. Dabei ist die Reihenfolge wichtig. Die Reihenfolge entspricht der des Bereiches PropertySet aus der SPDL. Wer es anders macht wird lustige Effekte bekommen, die reichen dann bis zum Crash von mentalRay. Das wollen wir ja nicht.

Wir fügen also in unserer Headerdatei den Wert hinzu und formulieren auch gleich alle Beschriftungen um. Damit tut wir so, als wenn der SDK Wizard uns die Headerdatei erstellt hätte:

```
1  // C Header File Generated by Softimage Shader Wizard
2
3  #ifndef TUTSIMPLECOLOR_H
4  #define TUTSIMPLECOLOR_H
5
6  #include <shader.h>
7
8  ////////////////////////////////////////////////////////////
9  // Type definition
10
11 typedef struct
12 {
13         miColor                         baseColor;                      // Basis Color
14         miColor                         secondaryColor;                 // Second Color
15 } tutSimpleColor_t;
16
17 #endif // TUTSIMPLECOLOR_H
```

Danach erstellen wir alle Targets neu und Updaten das DLL zur Sicherheit.

Der Random Color Shader

So nachdem wir nun unseren Shader beigebracht haben ein zweiten Input zu haben. Wollen wir nun mittels Zufallswert zwischen den beiden Farben hin und her schalten. Wir brauchen wieder unsere mi_eval Funktion um die Farbe aus den Parametern zu holen und dann ein simplen if um das result zu füllen. Um ein Random Wert zwischen 0 und 1 zu bekommen verwende ich mi_random. Etwa so:

```
26          miColor baseColor = *mi_eval_color (&params->baseColor);
27          miColor secondaryColor = *mi_eval_color (&params->secondaryColor);
28
29          if (mi_random() > 0.5f) {
30                  result->r = baseColor.r;
31                  result->g = baseColor.g;
32                  result->b = baseColor.b;
33                  result->a = baseColor.a;
34          } else {
35                  result->r = secondaryColor.r;
36                  result->g = secondaryColor.g;
37                  result->b = secondaryColor.b;
38                  result->a = secondaryColor.a;
39          }
40
41          return( miTRUE );
42  }
```

Als Ergebnis bekomme ich nun für jeden Punkt zufällig mal die eine oder andere Farbe. Dabei ist die Feinheit abhängig von den Render Einstellungen. Bei Super-Low werden auch weniger Rays abgeschossen, somit grobkörniger, bei Top-Banana[2] ist es entsprechend feiner, wie das Bild beweist:

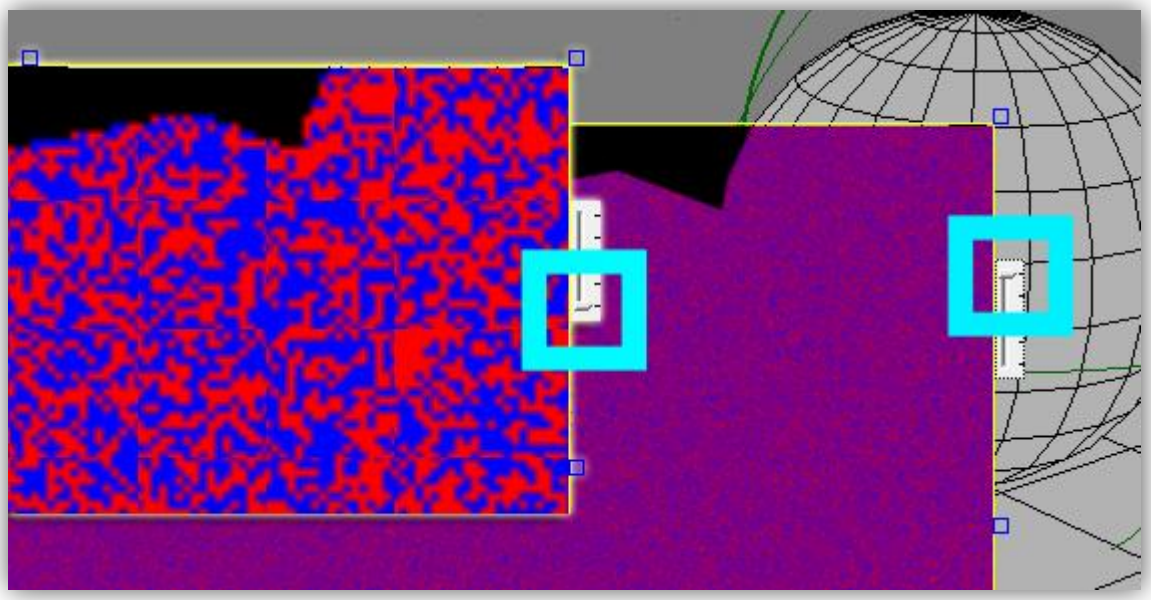


Abbildung 29 - Random Color Quick Render mit verschiedenen Auflösungen

Unser Shader wird ja bei jedem Ray aufgerufen. Daher ist die Auflösung der „Textur“ die wir liefern vollkommen unbegrenzt. Was aber auch bedeutet, je höher die Auflösung ist, desto mehr hat unser Shader zu tun.

[2] Top-Banana heißt die höchste Einstellung im Quick-Render – Ganz offiziell sogar ☺

Andere Texturen in unseren Shader hineinlaufen lassen

Wir kennen ja im Render Tree die Möglichkeit den Output eines Shaders als Input in ein anderen Shader einfließen zu lassen. Wenn wir unseren Shader aufklappen, sehen wir aber keine Inputs. Wir wollen nun erreichen dass wir anstatt der Basis Color auch andere Shader nehmen können. Dazu müssen wir SoftImage mitteilen, dass ein Parameter auch von extern gefüttert werden kann. Das können wir in unserem SPDL machen. Dort fügen wir im Input Parameter im Bereich PropertySet einfach den Parameter Texturable hinzu und setzen diesen auf true oder on. Auf Seite 13 wo wir den Input im SDK Wizard erstellt haben, was dieser Parameter nicht gesetzt. Wir editieren also unser SPDL wie folgt:

```
12          Parameter "baseColor" input
13          {
14                  GUID = "{8C3D1C69-BBFC-48CC-A8A3-EF404C169E48}";
15                  Type = color;
16                  Value = 1.0 0.0 0.0 1.0;
17                  Texturable = true;
18          }
19          Parameter "secondaryColor" input
20          {
21                  GUID = "{0065048D-A567-4ADB-8BD2-A103177998AE}";
22                  Type = color;
23                  Value = 0.0 0.0 1.0 1.0;
24                  Texturable = true;
25          }
```

Und Updaten nur unser SPDL.

Wir brauchen unseren Shader selbst nicht anfassen. Das nun die Farbe eigentlich auch von woanders herkommen kann, ist in unserem Shader vollkommen egal.

Wir holen uns ja die Farbe immer über die params und mi_eval Funktionen. mentalRay besorgt uns dann schon die Farbe die wir wollen. Dabei geht mentalRay diesen Baum selbst ab. Der Ray trifft auf ein Objekt, das hat ein Material, um die Farbe des Punktes zu bestimmen ruft mentalRay die Shader die in diesem Tree sind rekursiv auf. Bis es am Ende angekommen ist.

Dieser Shader legt das Ergebnis in result, in unserem Falle wäre das das Grid. Der nächste Shader bekommt das Ergebnis dann in sein Input gelegt, in unserem Falle wäre das der Parameter secondaryColor.

Aber auch unser Shader liefert eine Farbe und legt es in Result, welches der nächste Shader, also in unserem Falle das Constant, die Farbe weiter verarbeitet. Und so weiter …

Erst durch diesen Mechanismus wird der Render Tree eigentlich erst zu einem Tree. Um das ganze nun in einem Bild zusammenzufassen, hab ich mal eins gemacht:

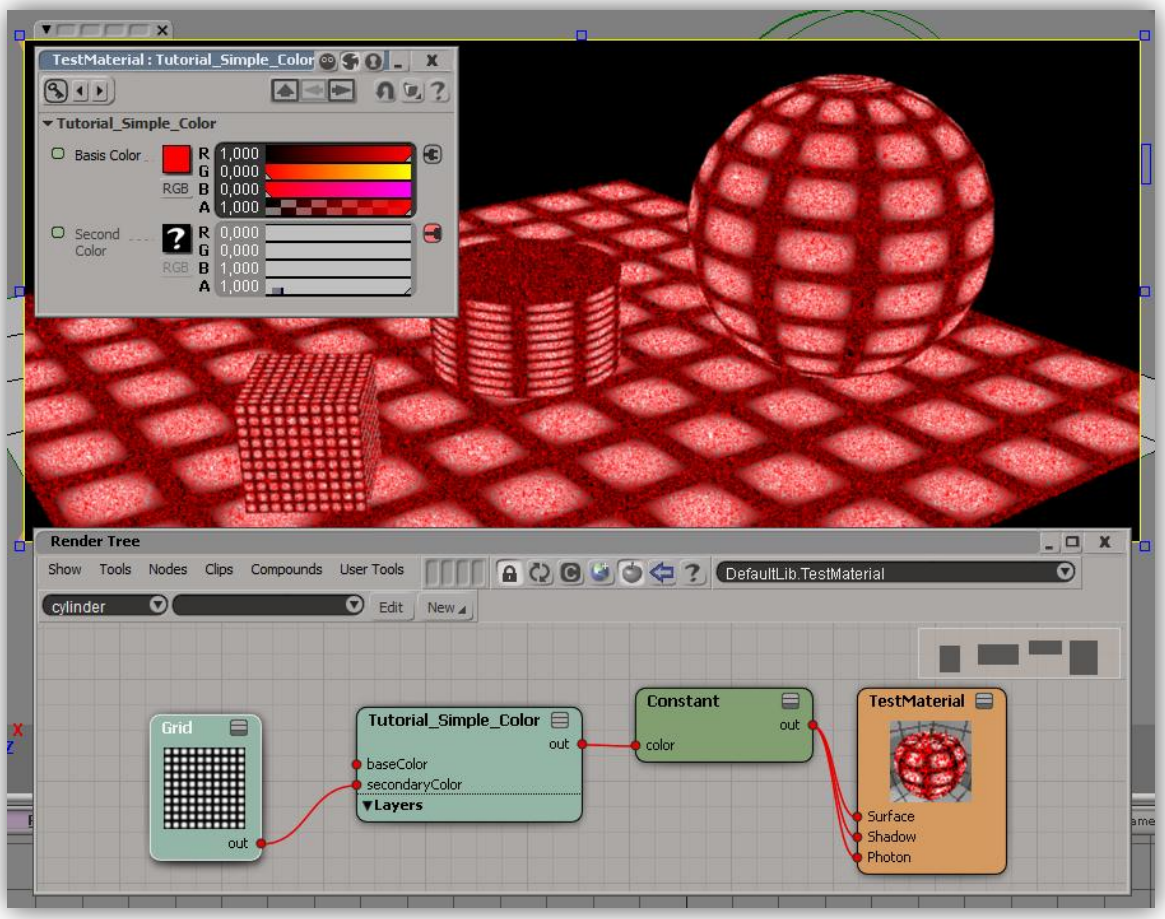


Abbildung 30 - Unser Shader mit einem Grid Shader verbunden

Man sieht deutlich das Grid durchscheinen. Unser Shader arbeitet wie vorher auch. Er nimmt mal die eine oder andere Farbe, je nachdem was Random gerade macht. Aber mal nimmt er unsere baseColor Rot, mal nimmt er die Farbe die vom Grid geliefert wird.

Das dort überall ein Grid auftaucht liegt am Texturespace, den jedes Objekt hat. Das Szenen-Erstell-Script auf Seite 34 erstellt überall eins.

Ein Shader mit Mustern bauen

Erste gedanken

So jetzt haben wir schon so viel gemacht, aber eine Frage quält uns noch. Wieso schafft es der Grid zum Beispiel ein Muster zu machen? Woher zum Teufel weiß der wann er eine Linie zeichnen soll?

Nun, der Grid weiß das auch nicht. Und er zeichnet auch keine Linie als solches. Nein, das könnte er auch gar nicht, weil er ja nie weiß wo und wann welcher Ray gerade kommt. Ja aber wie …

Ganz einfach. Wenn ein Shader aufgerufen wird, bekommen wir von mentalRay den stats Parameter mit. In dieser großen Struct steht unter anderem auch drin auf welche UV er gestoßen ist. Und das ist das Geheimnis. Jeder Ray terminiert auf einem Objekt ja immer irgendwie auf UV Koordinaten. Dagegen kann man sich überhaupt nicht wehren. Das ist zwangsläufig immer so. Und genau da fängt es an interessant zu werden.

Der Grid macht nichts anderes als zu sagen: Also wenn U zwischen 0,020 und 0,025 liegt, dann nimm Blau, ansonsten Schwarz. Fertig ist die Linie. Jetzt wird der eine oder andere sagen, das geht doch gar nicht. Doch geht. Genauso.

Ok das wollen wir mal testen. Dazu müssen wir aber wissen dass wir von mentalRay ein Array von Texturespaces bekommen werden. Das Problem an der Sache ist, das wir Blind auf den Inhalt zugreifen sollen, ohne zu prüfen ob da überhaupt was liegt. Wir sollten den unbedingt vorher fragen ob er nicht NULL ist.

Die Texturespaces liegen im Parameter state->tex_list und das ist ein Pointer auf ein miVector, aber in Wirklichkeit ist es ein Pointer auf ein Array mit Vektoren. Das erkennt man an dem Suffix _list. Wir können ja durchaus mehrere Texturespaces pro Objekt definieren. Den Index, den wir verwenden sollen, beziehen wir aus einem weiteren Input Parameter aus dem SPDL.

Aus diesem miVector der aus x, y, z besteht können wir unser UV auslesen. Ja das W auch, aber das brauchen wir gerade nicht. Das UV geht immer von 0,0 bis 1,0. Immer. Wenn wir im SoftImage unser Texturespace verkleinern oder vergrößern, machen wir automatisch unsere Textur kleiner oder größer. Wir haben allerdings nicht wirklich die Möglichkeit festzustellen die wievielte Wiederholung das nun ist, wie auch … wir kennen die Szene ja auch nicht.

Ok dann wollen wir mal ein Grid bauen. Ich will gerne ein einfaches Kreuz ausgeben. Wir brauchen in unserer GUI unbedingt dieses Texturespace Ding. Ohne dem sind wir aufgeschmissen. Wir greifen also zu unserem SPDL und machen ein solches GUI-Control da rein, zum Glück hilft uns da SPDL etwas.

Texturespace Control hinzufügen

Um ein Texturespace Control hinzuzufügen, sind wieder die gleichen Schritte nötig, wie auf Seite 43, wo wir ein Parameter hinzugefügt haben. Nur haben wir es jetzt mit anderen Typen und Werten zu tun. Im PropertySet hängen wir diesen Block an:

```
26          Parameter "texSpace" input
27          {
28                  GUID = "{179A06A8-6301-44C6-811B-C2772613D17F}";
29                  Type = texturespace;
30                  UI "filter" = "{14C06872-FEB9-11d0-9748-00A0243E36B9}";
31          }
```

In den Defaults bestimmen wir noch wie es aussehen soll:

```
61          texSpace
62          {
63                  Name = "Texture Space";
64                  UIType = "TextureSpaceItem.TextureSpaceItem.1";
65          }
```

Und im Layout tragen wir es auch noch ein.

```
72          texSpace;
```

Danach sollte es dann so aussehen:

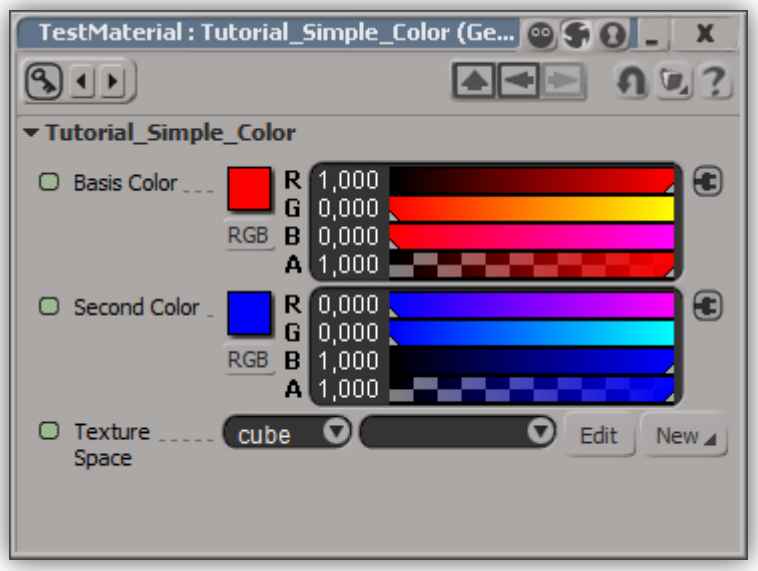


Abbildung 31 - Texturespace Control via SPDL hinzugefügt

Jetzt müssen wir diesen Input Parameter mit dem Namen texSpace mit in unsere Header Datei aufnehmen. Der Parameter mit dem Type texturespace ist ein einfacher int. Der int gibt uns später den Index auf das tex_list Array zurück. Unser Header sollte also erweitert werden:

```
15          int                             texSpace;                       // Texture Space
```

Programmierung des Kreuz Shaders

Dazu werde ich mir erst einmal UV holen und dann prüfen ob U bzw. V innerhalb von 0,4 und 0,6 ist. Wenn ja dann nehme ich die secondaryColor, ansonsten baseColor. Also ein blaues Kreuz auf rotem Grund. Das UV bekomme ich nun aus dem texSpace Parameter, der als Index auf die Liste dient.

Hier die komplette Shader Funktion dazu:

```
22          if (result == NULL) return (miTRUE);
23          if (state == NULL) return (miTRUE);
24          if (params == NULL) return (miTRUE);
25          if (state->tex_list == NULL) return (miTRUE);
26
27          miColor baseColor = *mi_eval_color (&params->baseColor);
28          miColor secondaryColor = *mi_eval_color (&params->secondaryColor);
29          int texSpace = *mi_eval_integer (&params->texSpace);
30
31          if (texSpace < 0) return (miTRUE);
32
33          miScalar u = state->tex_list[texSpace].x;
34          miScalar v = state->tex_list[texSpace].y;
35
36          if ((u > 0.4 && u < 0.6) || (v > 0.4 && v < 0.6)) {
37                  result->r = secondaryColor.r;
38                  result->g = secondaryColor.g;
39                  result->b = secondaryColor.b;
40                  result->a = secondaryColor.a;
41          } else {
42                  result->r = baseColor.r;
43                  result->g = baseColor.g;
44                  result->b = baseColor.b;
45                  result->a = baseColor.a;
46          }
47
48          return( miTRUE );
49  }
```

In Zeile 31 prüfe ich explizit noch einmal ob der Wert nicht doch Negativ ist. Ich teste gerne alles doppelt. Wobei ich anmerken muss das die mi_eval Funktionen auch ein NULL liefern könnten. Das habe ich bisher verschwiegen, aber wenn ich ich wäre, würde ich selbst das prüfen. Je weniger mein Code crashen kann, desto besser kann ich schlafen. Man könnte obiges aber auch massiv kürzen, aber wir wollen ja was lernen und kein Kryptographen Kurs absolvieren.

Wenn man das Ding nun mit allen Änderungen durch den Compiler jagt und alles aktualisiert, dann kann man auf dem nächsten Bild schön sehen das ich am Poly-Grid 4 Texture Spaces erstellt hab, die man dann einfach wechseln kann und mein Kreuz sich entsprechend anpasst.

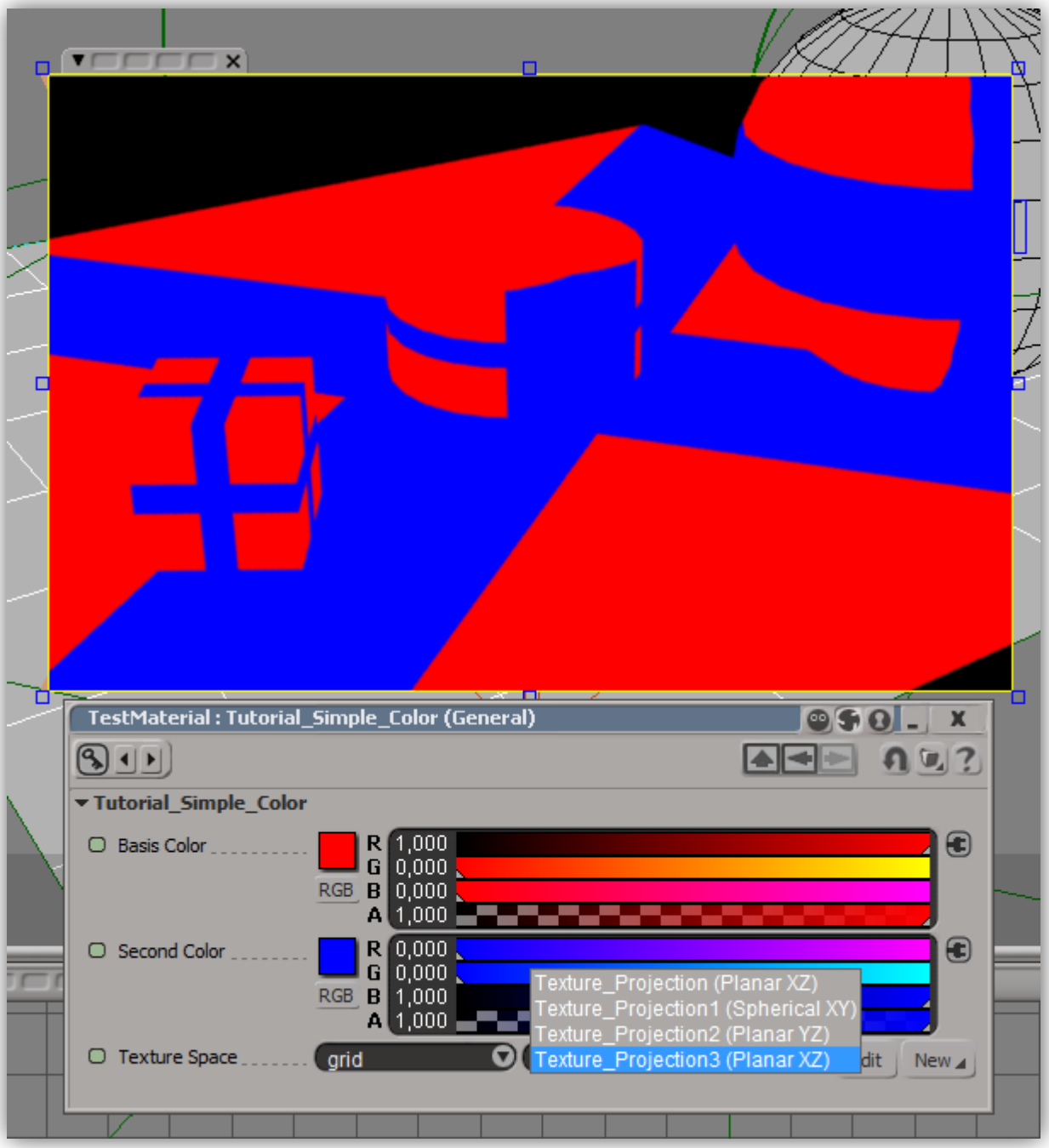


Abbildung 32 - Unser Kreuz Shader mit mehreren Texture Spaces und in Aktion

Und genau so macht es der SoftImage Grid Shader auch. Gut ja – er hat ein paar mehr Optionen. Aber die Funktionsweise ist genau die gleiche.

Einstellbare Stichstärken

Ja wir wollen aber auch die Dicke der Stiche, wenn man sie so nennen will, anpassen können. Ja gut, kein Problem. Dazu brauchen wir zwei weitere Scalar Parameter und eine andere if Zeile. Also wieder SPDL in den PropertySet:

```
32          Parameter "widthU" input {
33                  GUID = "{5C9FF168-49E6-4FBB-AA71-8D0A58FD2A71}";
34                  Type = scalar;
35                  Value = 0.1;
36                  Value Minimum = 0.0;
37                  Value Maximum = 0.5;
38          }
39          Parameter "widthV" input {
40                  GUID = "{E15DD3E6-989A-4BBC-A6EB-24DC93FCD487}";
41                  Type = scalar;
42                  Value = 0.1;
43                  Value Minimum = 0.0;
44                  Value Maximum = 0.5;
45          }
```

Das Display:

```
80          widthU {
81                  Name = "Width U";
82                  UIRange = 0.0 To 0.5;
83          }
84          widthV {
85                  Name = "Width V";
86                  UIRange = 0.0 To 0.5;
87          }
```

Und das Layout:

```
95          widthU;
96          widthV;
```

In unserer Header Datei fügen wir jetzt noch die zwei hinzu:

```
16          miScalar                        widthU;                 // Width in U
17          miScalar                        widthV;                 // Width in V
```

Und im Shader müssen wir natürlich auch noch was tun:

```
33          miScalar u = state->tex_list[texSpace].x;
34          miScalar v = state->tex_list[texSpace].y;
35
36          miScalar widthU = *mi_eval_scalar (&params->widthU);
37          miScalar widthV = *mi_eval_scalar (&params->widthV);
38
39          miScalar halfU = widthU / 2.0f;
40          miScalar halfV = widthV / 2.0f;
41
42          if ((u > (0.5 - halfU) && u < (0.5 + halfU)) || (v > (0.5 - halfV) && v < (0.5 +
halfV))) {
43                  result->r = secondaryColor.r;
44                  result->g = secondaryColor.g;
```

Und wir können auch ein anderes Material nehmen … ja das geht … sieht dann so aus:

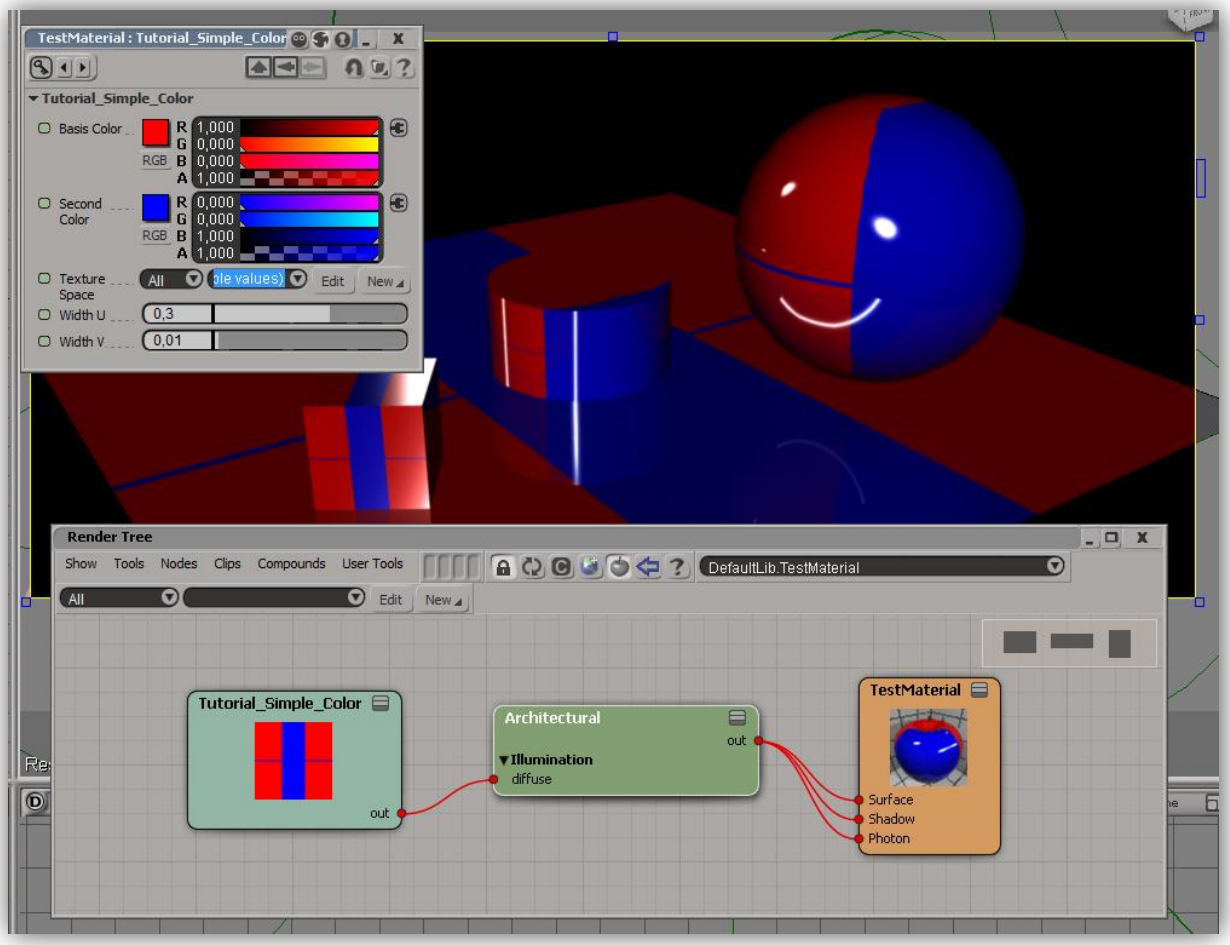


Abbildung 33 - Unser Kreuz Shader mit variablen Strich Stärken und mit Architectural

Wir könnten jetzt die Strichstärke zum Beispiel animieren. Oder die Farben mit Wolken, Steinen oder sonstigem versehen. Alles geht.

Resümee

So ich hoffe der kleine Ausflug hat Gefallen gefunden. Wir haben in diesem kleinen Abschnitt nun gelernt:

1. Der generelle Aufbau des Shader Source Code
2. Die Funktionsweise der Input Parameter und deren Zusammenhang im SPDL
3. Die Funktionsweise des Output
4. Das manuelle editieren einer SPDL und der Header Datei
5. Zugriff auf den Texture Space über den state Parameter
6. Ein paar funktionierende Shader

Das sollte erst einmal genügen ☺

Aber das wichtigste was wir gesehen haben war: Die Erweiterung eines Shaders war ja nun echt ein Kinderspiel, mal kurz zwei Scalars da reingefummelt übersetzt und gut ist. Das ganze sollte nicht mehr länger als 2-6 Minuten dauern. Jetzt ☺

Mehr Informationen zu SPDL

Ja die gibt es. Wer bei einem SPDL nicht mehr weiterkommt, kann entweder die SDK Hilfe Datei von SoftImage 7.5 lesen, an der Hilfe HTML von SoftImage 2012 verzweifeln – weil da fehlt einiges – oder einfach mal in den AutoDesk SoftImage Programm Verzeichnis unter Application\spdl schauen. Dort liegen quasi alle SPDL Dateien der eingebauten Shader rum. Dort kann man sich dann anschauen, wie zum Beispiel Integer, Boolean gehen, oder ganz andere wilde Geschichten.

Die Dateien sind auch mein erster Anlaufpunkt wenn ich was ganz unnormales machen will. Den C3DGradientNode.spdl kann ich empfehlen wenn man mal was Abgefahrenes sehen will.

Ansonsten gibt es im Internet womöglich die eine oder andere Quelle, wo man noch etwas erfahren kann. Was ich euch gegeben habe, war nur ein Einstieg, der Rest kommt entweder im zweiten Teil oder müsst Ihr euch selbst suchen ☺

Diese Seite ist leer

Das Dokument ist für die Anzeige von zwei Seiten optimiert

Der vollständige Quelltext

tutSimpleColor.spdl

```
# SPDL Generated by Softimage Shader Wizard
SPDL
Version = "2.0.0.0";
Reference = "{8A57287D-5630-45BA-9ADC-12DF9A803778}";
PropertySet "tutSimpleColor_pset"
{
        Parameter "out" output
        {
                GUID = "{28DAA117-D2C8-41C0-8ED9-0705D7B40988}";
                Type = color;
        }
        Parameter "baseColor" input
        {
                GUID = "{8C3D1C69-BBFC-48CC-A8A3-EF404C169E48}";
                Type = color;
                Value = 1.0 0.0 0.0 1.0;
                Texturable = true;
        }
        Parameter "secondaryColor" input
        {
                GUID = "{0065048D-A567-4ADB-8BD2-A103177998AE}";
                Type = color;
                Value = 0.0 0.0 1.0 1.0;
                Texturable = true;
        }
        Parameter "texSpace" input
        {
                GUID = "{279A08A8-6301-44C6-811B-C2772613D17F}";
                Type = texturespace;
                UI "filter" = "{04C00872-FEB9-11d0-9748-00A0243E36B9}";
        }
        Parameter "widthU" input {
                GUID = "{5C9FF168-49E6-4FBB-AA71-8D0A58FD2A71}";
                Type = scalar;
                Value = 0.1;
                Value Minimum = 0.0;
                Value Maximum = 0.5;
        }
        Parameter "widthV" input {
                GUID = "{E15DD3E6-989A-4BBC-A6EB-24DC93FCD487}";
                Type = scalar;
                Value = 0.1;
                Value Minimum = 0.0;
                Value Maximum = 0.5;
        }
}

MetaShader "tutSimpleColor_meta"
{
        Name = "Tutorial Simple Color";
        Type = texture;
        Renderer "mental ray"
        {
                Name = "tutSimpleColor";
                FileName = "tutSimpleColor";
                Options
                {
                        "version" = 1;
                }
        }
}

Defaults
{
        baseColor
        {
                Name = "Basis Color";
                UIType = "rgba";
        }
```

```
70          secondaryColor
71          {
72                  Name = "Second Color";
73                  UIType = "rgba";
74          }
75          texSpace
76          {
77                  Name = "Texture Space";
78                  UIType = "TextureSpaceItem.TextureSpaceItem.1";
79          }
80          widthU {
81                  Name = "Width U";
82                  UIRange = 0.0 To 0.5;
83          }
84          widthV {
85                  Name = "Width V";
86                  UIRange = 0.0 To 0.5;
87          }
88  }
89
90  Layout "Default"
91  {
92          baseColor;
93          secondaryColor;
94          texSpace;
95          widthU;
96          widthV;
97  }
98
99  Plugin = Shader
100 {
101         FileName = "tutSimpleColor";
102 }
```

tutSimpleColor.h

```
1  // C Header File Generated by Softimage Shader Wizard
2
3  #ifndef TUTSIMPLECOLOR_H
4  #define TUTSIMPLECOLOR_H
5
6  #include <shader.h>
7
8  ////////////////////////////////////////////////////////////
9  // Type definition
10
11 typedef struct
12 {
13         miColor                         baseColor;                      // Basis Color
14         miColor                         secondaryColor;                 // Second Color
15         int                            texSpace;                       // Texture Space
16         miScalar                        widthU;                 // Width in U
17         miScalar                        widthV;                 // Width in V
18 } tutSimpleColor_t;
19
20 #endif // TUTSIMPLECOLOR_H
```

tutSimpleColor.cpp

```
// C++ Source Code Generated by Softimage Shader Wizard

////////////////////////////////////////////////////////////////
// Includes

#include <shader.h>
#include "tutSimpleColor.h"

////////////////////////////////////////////////////////////////
// Implementation

extern "C" DLLEXPORT miBoolean
tutSimpleColor
(
        miColor                         *result,
        miState                         *state,
        tutSimpleColor_t        *params
        )
{
        // TODO: Shader main code goes here

        if (result == NULL) return (miTRUE);
        if (state == NULL) return (miTRUE);
        if (params == NULL) return (miTRUE);
        if (state->tex_list == NULL) return (miTRUE);

        miColor baseColor = *mi_eval_color (&params->baseColor);
        miColor secondaryColor = *mi_eval_color (&params->secondaryColor);
        int texSpace = *mi_eval_integer (&params->texSpace);

        if (texSpace < 0) return (miTRUE);

        miScalar u = state->tex_list[texSpace].x;
        miScalar v = state->tex_list[texSpace].y;

        miScalar widthU = *mi_eval_scalar (&params->widthU);
        miScalar widthV = *mi_eval_scalar (&params->widthV);

        miScalar halfU = widthU / 2.0f;
        miScalar halfV = widthV / 2.0f;

        if ((u > (0.5 - halfU) && u < (0.5 + halfU)) || (v > (0.5 - halfV) && v < (0.5 +
halfV))) {
                result->r = secondaryColor.r;
                result->g = secondaryColor.g;
                result->b = secondaryColor.b;
                result->a = secondaryColor.a;
        } else {
                result->r = baseColor.r;
                result->g = baseColor.g;
                result->b = baseColor.b;
                result->a = baseColor.a;
        }

        return( miTRUE );
}


extern "C" DLLEXPORT void
tutSimpleColor_init
(
        miState                         *state,
        tutSimpleColor_t        *params,
        miBoolean                       *inst_init_req
        )
{
        if( params == NULL )
        {
                // TODO: Shader global initialization code goes here (if needed)

                // Request a per-instance shader initialization as well (set to miFALSE if
not needed)
```

```
71                  *inst_init_req = miTRUE;
72          }
73          else
74          {
75                  // TODO: Shader instance-specific initialization code goes here (if needed)
76          }
77  }
78
79
80  extern "C" DLLEXPORT void
81  tutSimpleColor_exit
82  (
83          miState                         *state,
84          tutSimpleColor_t        *params
85          )
86  {
87          if( params == NULL )
88          {
89                  // TODO: Shader global cleanup code goes here (if needed)
90          }
91          else
92          {
93                  // TODO: Shader instance-specific cleanup code goes here (if needed)
94          }
95  }
96
97
98  extern "C" DLLEXPORT int
99  tutSimpleColor_version( )
100 {
101         return( 1 );
102 }
```

Danke

So dann will ich dieses Dokument mal abschließen!

Ich bedanke mich für das Lesen und sofern Ihr Lust hattet auch dem Durcharbeiten

Wer inhaltliche Fehler, oder Fehler im Quelltext findet – da dürften dank Copy und Paste keine sein, sagt einfach Bescheid und das Problem wird gefixt.

Wer Linksschrifterrors findet, darf sie sich an die Wand nageln ☺ haha

Kommentare jeglicher Form kann man auf meiner Webseite im Feedback Forum machen. Demnächst kommt auch eine Kommentarfunktion auf den Seiten – hab aber gerade keine Zeit musste so ein Shader Dokument schreiben … ☺ Oder im XSI Forum

Englische Übersetzung … hmmm … ja … well that will be depend on feedback, if the feedback be well, than maybe, if not than nope ☺

Nun denn …

Happy Shading With SoftImage XSI !!!

Wolfgang Hurst

Sonstige Verzeichnisse

Abbildungsverzeichnis