2020
5

ISSN 1433-2620 > 24. Jahrgang >> www.digitalproduction.com
Publiziert von Pixeltown GmbH

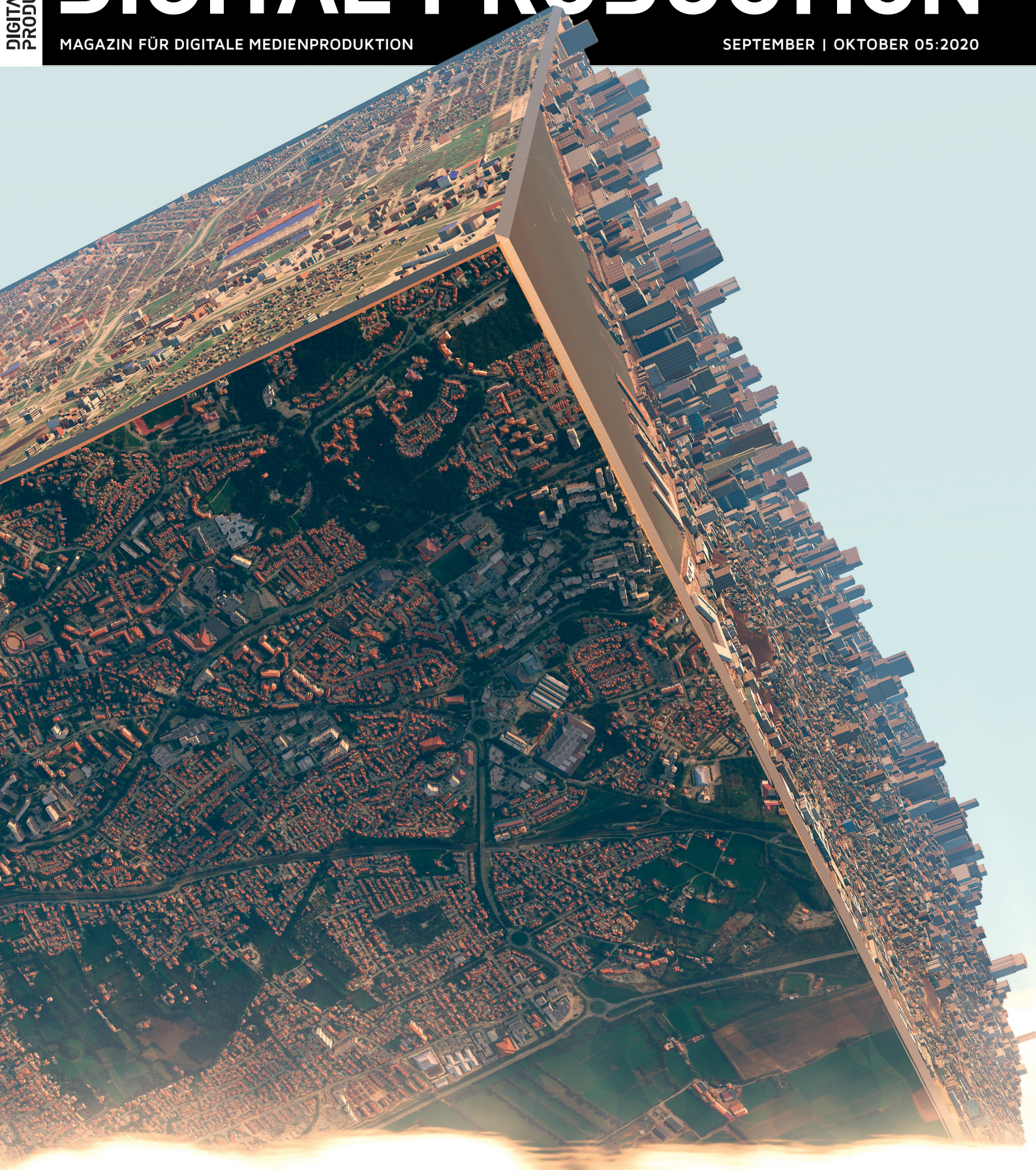
Deutschland € 17,90
Österreich € 19,–
Schweiz sfr 23,–

DIGITAL PRODUCTION

DIGITAL PRODUCTION

MAGAZIN FÜR DIGITALE MEDIENPRODUKTION

SEPTEMBER | OKTOBER 05:2020

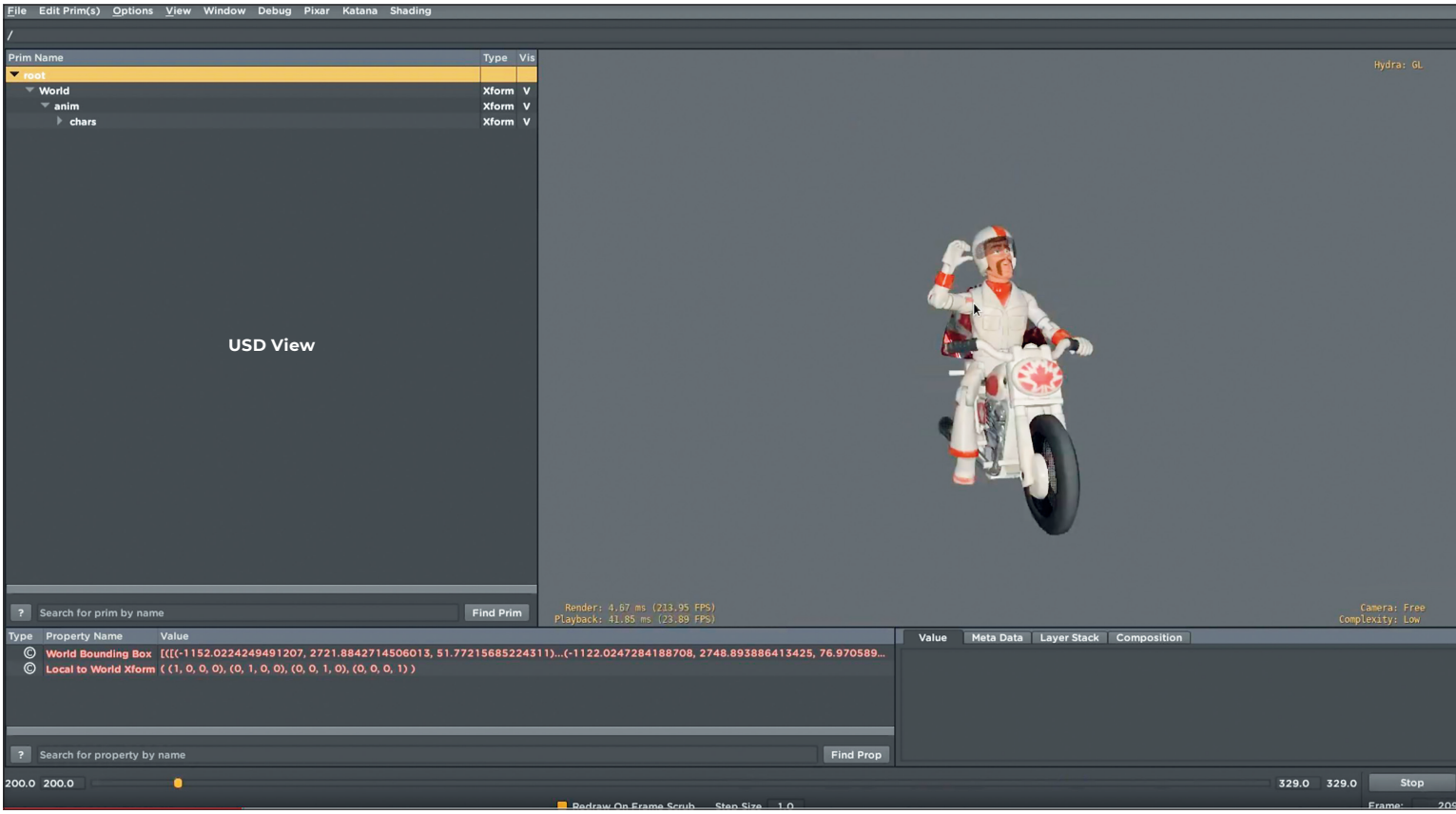


Praxis
Speedtree, MoCap,
UVPackmaster, Pano2VR

Theorie
USD, Skin in ZBrush,
Color in Avid, Asus Screens

Projekte
Love & 50 Megatons,
The Awakening, The Lander

...und mehr
Flame, Grease Pencil,
Nuke Indie und BMD 12K



Universal Scene Description (USD) – der Stand der Dinge

USD ist in aller Munde – sogar Nuke kann es mittlerweile über den NatGeo Node importieren. Und vordergründig sieht das alles sehr interessant und logisch aus. Aber was erwartet einen, wenn man den Hahn zur Pipeline aufdreht? **von Jan Walter**

In diesem Artikel möchte ich auf einige Aspekte von USD eingehen, die sich zunächst mit dem Rendering (also der Berechnung von Bildern) beschäftigen, und dann einen Ausblick geben, was wir in naher Zukunft als Benutzer in einigen Produkten bzgl. USD (Universal Scene Description) erwarten können. Als Principal R&D Engineer bei The Mill komme ich eher von der Programmierer-Seite und würde deshalb gerne mit ein paar Screenshots (eines Youtube-Videos **bit.ly/usdview**) anfangen.

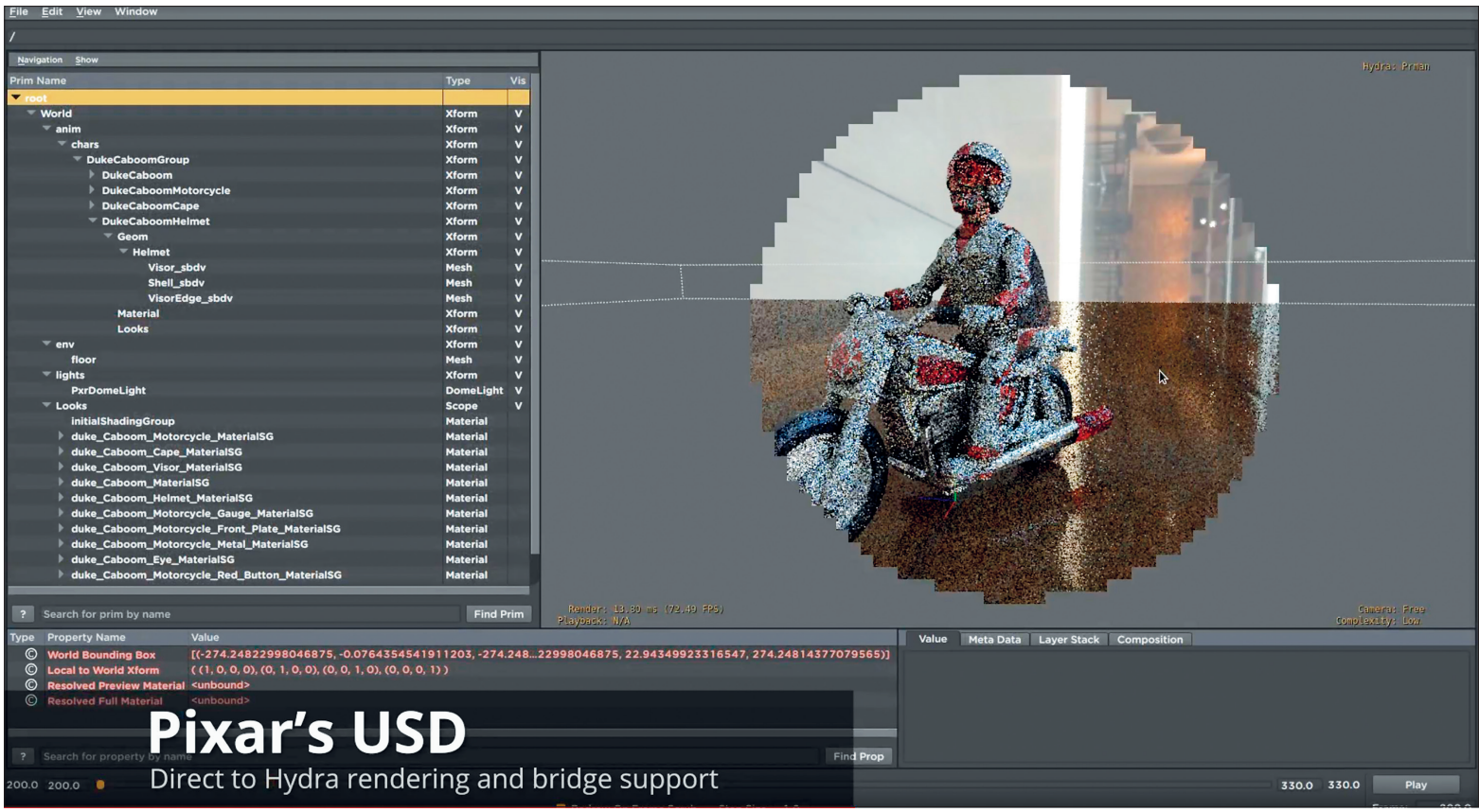
In dem Screenshot oben sehen wir ein Standalone-Programm namens USD View, das ein Programmierer erhalten würde, wenn er den Source Code von Pixar (**github.com/PixarAnimationStudios/USD**) kompilieren würde. Es ist ein Beispiel dafür, wie ein Programm aussehen könnte, das als Grundlage die USD-API (Programmierschnittstelle) und -Bibliothek nutzt. Man kann über Google auch vorkompilierte Versionen des Programms (z.B. für Windows) finden.

Was sieht man?

Oben sieht man eine Menüleiste mit verschiedenen Einträgen und Optionen. Links befindet sich ein sogenannter Outliner bzw. eine Art Verzeichnisstruktur, in der man Ordner bzw. Unterordner öffnen und wieder schließen kann. Ich werde darauf später noch mal näher eingehen.

Rechts ist ein Viewport sichtbar. Früher wäre das eine Art Fenster, in dem OpenGL (oder DirectX) benutzt wird (ähnlich wie bei Spielen), um den Benutzer mit der Szene, der Geometrie und Ähnlichem interagieren zu lassen. Die Qualität ist normalerweise nicht besonders hoch und sieht dem am Schluss berechneten Bild nur bedingt ähnlich. Es dient also eher als eine Art Vorschau. Je nachdem, wie modern ein Programm ist und welche Hardware man für die Grafikkarte voraussetzen kann, kann diese Vorschau jedoch schon sehr nahe an das fertige Bild herankommen. Siehe z.B. Blender in Version 2.8 und später.

Den unteren Teil ignorieren wir hier weitgehend, aber es gibt eine Timeline, über die man eine vorgefertigte Animation ablaufen lassen kann und die möglichst in Echtzeit (im Viewport) eine Vorschau auf den fertigen Animationsfilm bietet, an dem man gerade arbeitet.

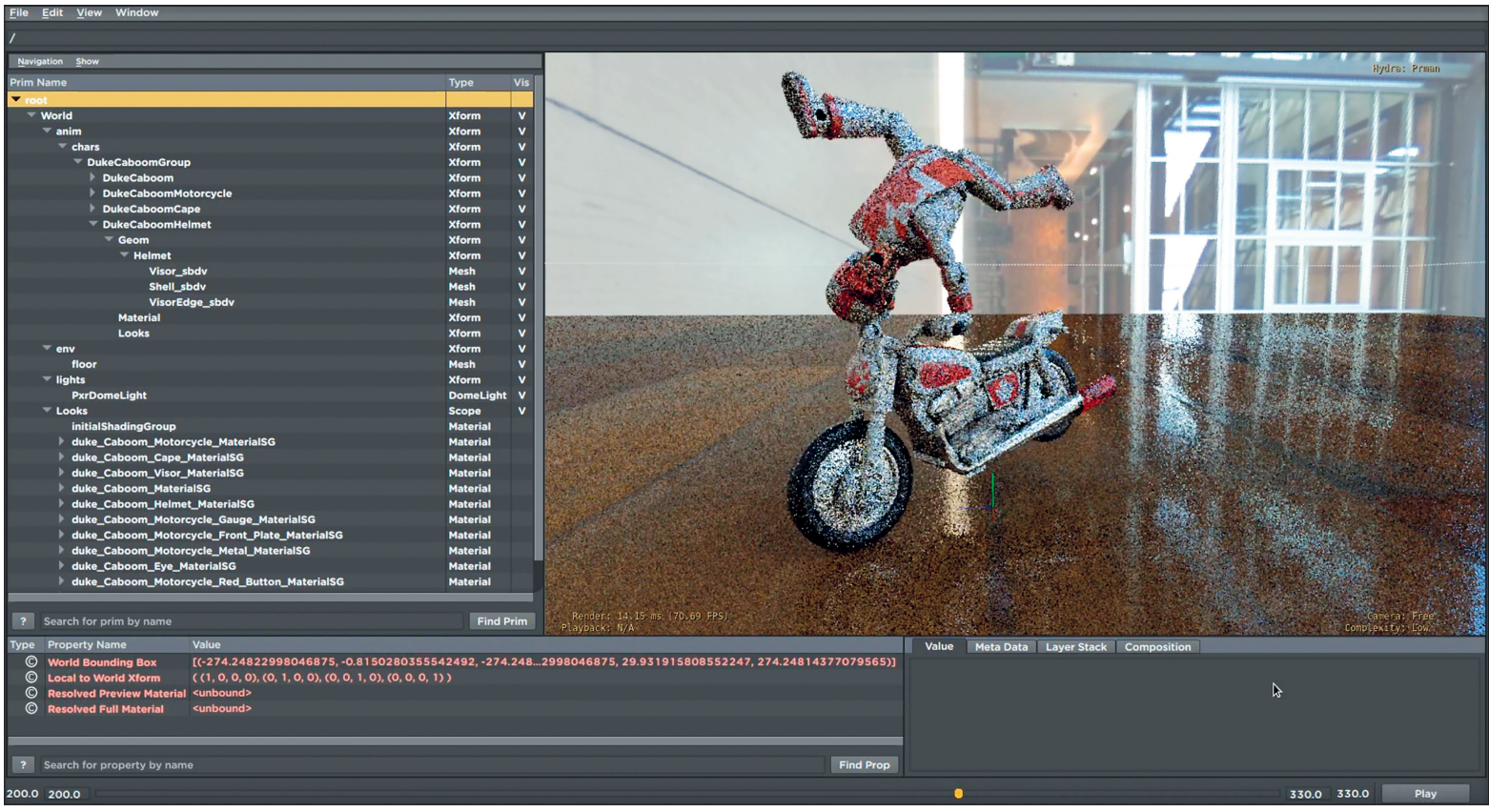


Die Berechnung startet.

Bei Sekunde 34 des Videos (siehe Screenshot oben) wird es interessant, denn plötzlich wird im Viewport ein Bild berechnet (gerendert).

Vom Mittelpunkt ausgehend werden etwas größere Rechtecke (also nicht einzelne Pixel, sondern sogenannte Buckets) berechnet, die schon eine relativ realistische Beleuchtung und deren Auswirkung auf die verwendeten Materialien geben.

Würde man nicht mit der Szene interagieren bzw. nicht die Timeline (sprich

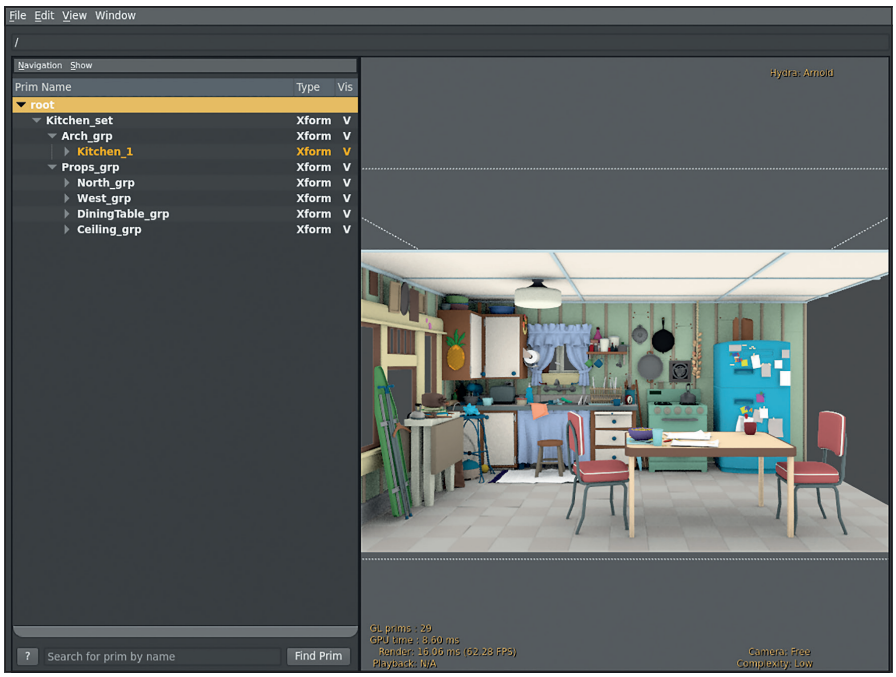


Animation) ablaufen lassen, so würde sich dieser Prozess wiederholen, d.h. die gleichen Rechtecke würden immer feiner werden. Nicht die Größe würde kleiner werden, sondern die darin enthaltenen Pixel würden langsam vom Noise befreit werden und dem endgültigen Bild immer ähnlicher, d.h. das Bild konvergiert langsam zu dem endgültigen Resultat.

Braucht's Buckets?

Es müssen übrigens nicht unbedingt Buckets sein, die gerendert werden, sondern es könnten auch einzelne Pixel direkt berechnet werden – nur sind halt im Video größere Bereiche zu sehen. Dies kann historische oder technische Gründe haben oder auch bei einzelnen Renderern ein- bzw. ausgeschaltet werden. Cycles, der Renderer, der mit Blender ausgeliefert wird, erlaubt beides: das Rendern mit einer einstellbaren Bucket-Größe bzw. das Berechnen von einzelnen Pixeln zu einem Gesamtbild, das dann immer feiner, also von Noise befreit wird. Im Falle von Cycles – dort werden Buckets auch Tiles genannt – kann sich die eingestellte Größe positiv bzw. negativ auf die benötigte Renderzeit auswirken bzw. Einfluss auf den Speicherverbrauch während der Bildberechnung haben.

Selbst Motion Blur (Bewegungsunschärfe – z.B. 0:44/1:01 im Video) oder Depth of Field (Tiefenunschärfe) wären relativ schnell

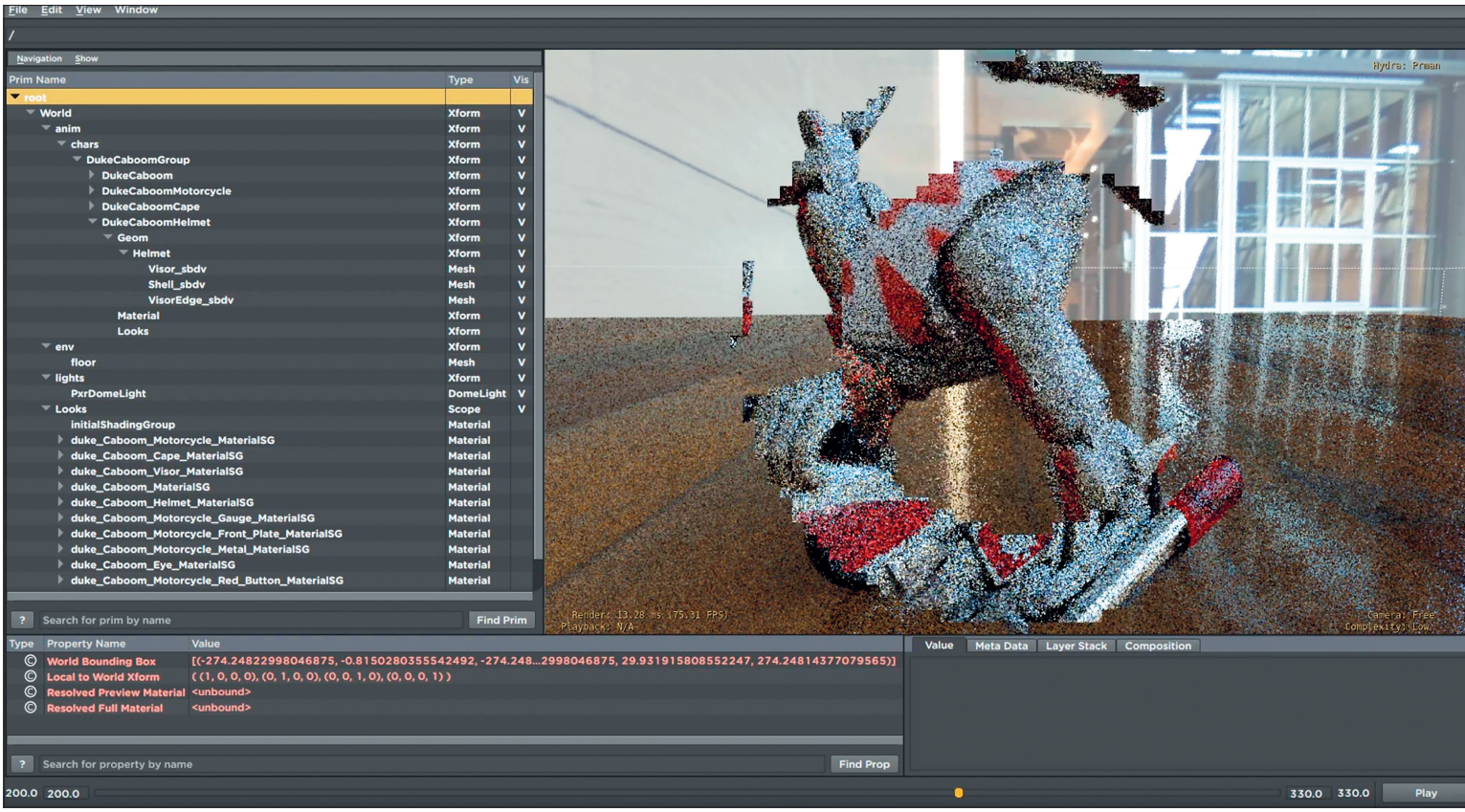


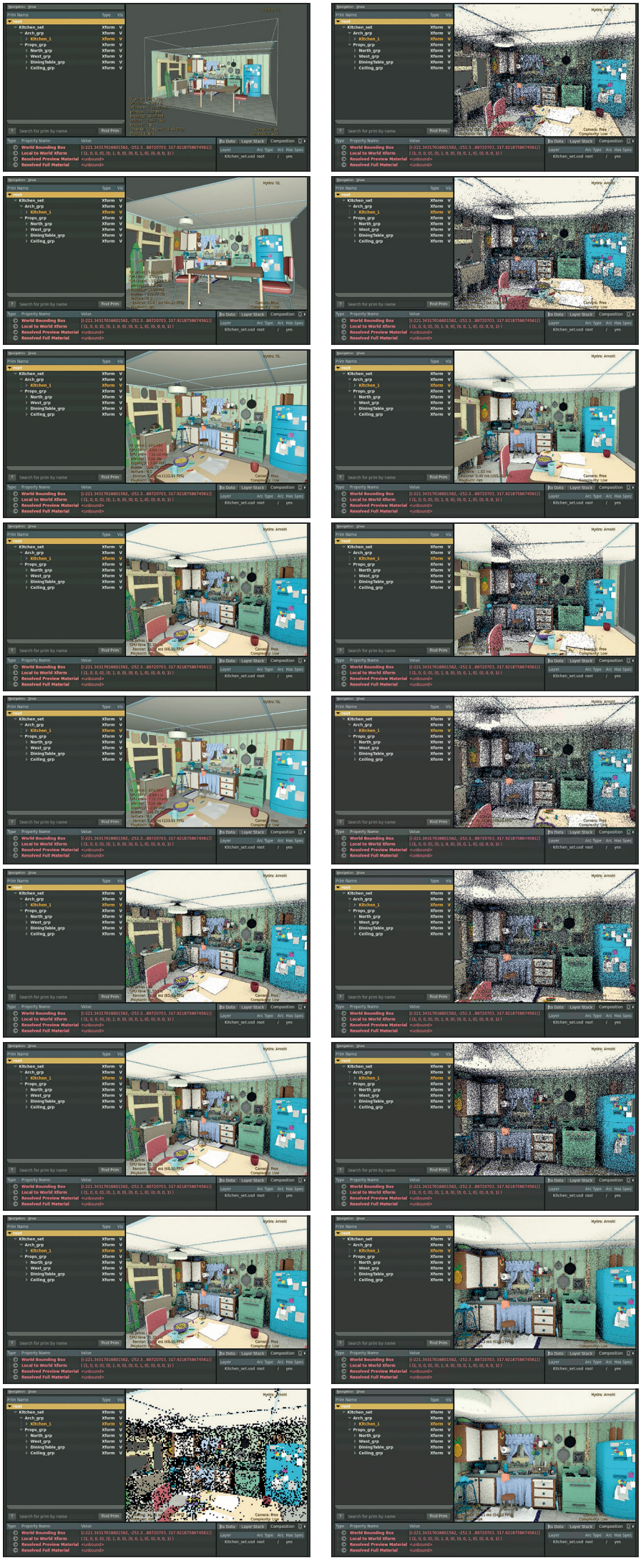
Eine Szene mit Tiefenunschärfe

(u.U. in Echtzeit) im Viewport sichtbar, ähnlich wie wir das aus anderen Programmen unter dem Namen IPR (Interactive Progressive Rendering) kennen.

Aber welcher Renderer genau ist denn nun verantwortlich für diese andere Art der Bildberechnung? In dem oben genannten Video ist es Renderman von Pixar (oder auch PRMan). Ein Aspekt von USD (aus programmiertechnischer Sicht) ist allerdings, dass es jeder beliebige Renderer sein kann. Mir ist u.a. bekannt, dass Arnold und Redshift an einer solchen USD-Integration arbeiten. Der Screenshot oben zeigt, wie z.B. Arnold direkt in den Viewport rendern kann. Über ein Plug-in kann USD View erweitert werden, sodass ein zusätzlicher Renderer integriert werden kann.

Im Kontext von USD nennt man das Imaging Framework (also den Teil zur Bilderzeugung)





Hydra (siehe auch meinen englischen Artikel: **github.com/PixarAnimationStudios/USD**) und ein solches Renderer-spezifisches Plug-in Hydra Render Delegate (siehe z.B. für den Renderer Embree: **bit.ly/pixar_embree**). Im Prinzip geht es darum, möglichst schnell einen Farbwert für entweder ein einzelnes Pixel oder mehrere Farben für Pixel in einem Bucket (Rechteck von Pixeln) zu berechnen. Hier wird es schnell technisch, wie das vonstatten gehen kann.

Raytracing vs. Path Tracing

Bei Echtzeitanwendungen bzw. bei Grafikkarten spricht man oft von Raytracing, bei modernen Renderern, die weniger mit realtime vermarktet werden, von bi- bzw. unidirectional Path Tracing. Diese Unterscheidung mag verwirrend sein, deshalb möchte ich kurz auf einige Aspekte eingehen.

Stark vereinfacht geht es beim Rendering um Reflexionen und Refraktionen. Eine Reflexion kann man sich wie einen Gummiball vorstellen, der gegen eine Wand geworfen wird und abprallt, in mehr oder weniger die Gegenrichtung, je nach dem Winkel, in dem er auf der Wand auftrifft. Steht man direkt vor der Wand, wird der Ball direkt auf den Werfer zurück reflektiert. Geschieht dies in einem Winkel, also z.B. schräg von der Seite, wechselt der Ball zwar immer noch die Richtung beim Abprall von der Wand, fliegt aber tendenziell vom Werfer weg. Eine Refraktion kann man sich veranschaulichen, indem man sich z.B. vornimmt einen Fisch mit einem Pfeil im Wasser zu treffen. Zielt man direkt von oben ins Wasser wird man vermutlich das Ziel gut treffen, der Pfeil wird kaum abgelenkt und fliegt mehr oder weniger gerade auf den Fisch zu. Geschieht dies in einem Winkel, also z.B. schräg von oben nach unten, fliegt der Pfeil vermutlich daneben, weil unsere Sichtstrahlen an der Wasseroberfläche gebrochen werden. Sie gehen nicht mehr direkt geradeaus, sondern werden leicht zum Grund des Wasserbeckens abgelenkt.

Beim Rendering passiert etwas Ähnliches. Vom Auge bzw. der Kamera werden Strahlen (Rays) in die Szene ausgesendet. Diese treffen auf einen Gegenstand. Das Material dieses Objekts bestimmt, ob eine Reflexion oder Refraktion oder beides berechnet werden muss. Ist die Oberfläche rau, z.B. bei einer Duschkabine, dann kann man durch das Objekt nicht so gut hindurchsehen, die Gegenstände dahinter sind nur verschwommen wahrzunehmen. Ähnlich verhält es sich mit einem Spiegel, dessen Oberfläche z.B. durch Kondenswasser die eintreffenden Strahlen in mehrere Richtungen reflektiert.

Beim traditionellen Raytracing werden hierfür mehrere Strahlen reflektiert bzw. refraktiert. Wenn man z.B. dreimal bei einem Auftreffen auf einen Gegenstand 10 neue Strahlen berechnen würde, die dann weiterverfolgt werden, dann hätte man beim ersten Mal 10 neue Strahlen generiert, beim zweiten Mal würde jeder dieser 10 neuen Strahlen 10 weitere erzeugen, also insgesamt 100. Beim dritten Gegenstand hätten wir bereits 1.000 Strahlen zu verfolgen. Stark vereinfacht würden all diese Strahlen eine Farbe erzeugen, würden jeweils an der Stelle, wo ein Gegenstand getroffen wurde, zu einer einzigen Farbe gemischt und schließlich für den ersten Strahl zu einer einzigen Farbe vermengt. Um Aliasing zu vermeiden (Antialiasing), würde man sogar noch mehr Strahlen per Pixel, sogenannte Samples, aussenden und zu einer Farbe mischen. Sagen wir, wir verwenden 8 Samples, dann hätten wir bei nur dreimaliger Strahlenverfolgung bereits 8.000 Rays erzeugt.

Beim Path Tracing geschieht im Prinzip dasselbe, aber ein wichtiger Unterschied ist, dass man versucht, die Anzahl der neuen Strahlen zu begrenzen, d.h. man versendet lieber mehr Samples per Pixel, dafür werden aber von einem Strahl nur (vereinfacht) maximal zwei neue Strahlen erzeugt. In unserem Beispiel hätten wir beim ersten Gegenstand 2 neue Strahlen erzeugt, beim zweiten Mal 4, beim dritten Mal 8 Strahlen. Man könnte also 1.000 Samples verwenden, um eine ähnliche Qualität im finalen Bild zu erreichen.

In Bezug auf USD ist dies aber ein wesentlicher Unterschied. Uns ging es ja darum, möglichst schnell einen Farbwert für ein Pixel zu berechnen. Jedes weitere Sample (pro Pixel) würde die bereits gezeigte Farbe ein wenig modifizieren, aber nicht grundsätzlich stark verändern. Es würde über die Zeit der Eindruck entstehen, dass weniger Noise zu sehen ist, aber wenn der Benutzer mit der Szene interagieren möchte, kann das Rendering jederzeit abgebrochen werden, die Szene z.B. gedreht und mit dem Rendering von vorne begonnen werden, ohne auf die Berechnung der noch ausstehenden Strahlen wie beim traditionellen Ray Tracing warten zu müssen. Vielleicht erinnert sich noch der eine oder andere an eine Zeit, wo für einen Gegenstand aus rauem Glas oder Metal ein paar Pixel sehr lange dauerten, während andere Teile des Bildes schon längst fertig gerendert waren?

Scene Graphs und Caching

Wenn man mal sehr weit zurück in die Geschichte der Computergrafik geht, findet man z.B. OpenGL, das vor ca. 28 Jahren das erste Mal in Erscheinung trat. Dieses hatte damals große Ähnlichkeit mit Pixars RIB-Dateiformat (Renderman Interface Bytestream). Im Prinzip finden sich hier bereits die Grundlagen für hardwareunterstütztes Viewport Rendering (schnell, für die Interaktion des Artists mit der Szene, Modellierung, Animation etc.) und die Berechnung des finalen Bildes (sobald der Artist fertig ist), was länger dauern kann und eventuell auf eine Renderfarm mit vielen CPUs ausgelagert wird.

Basierend auf OpenGL gab es zwei weitere Bibliotheken, Iris Inventor (später Open Inventor) und Iris Performer, die beide einen Scene Graph einführten. Inventor tat dies vor allem, um mehr Struktur in eine komplexe Szene zu bringen, half aber auch mit der Interaktion über sogenannte Controller Objects oder Manipulators, die in Blender z.B. Gizmos genannt werden. Performer führte den Scene Graph eher aus Performance-Gründen ein, die mit Multithreading und mehreren CPUs zu tun hatten.

Aus heutiger Sicht macht es eher Sinn, sich Gedanken über z.B. den Hauptspeicher eines Rechners und den Speicher auf der Grafikkarte Gedanken zu machen. Der Transfer vom Hauptspeicher zum Speicher der Grafikkarte ist eine der zeitaufwendigsten Operationen, die man als Programmierer deshalb möglichst selten ausführen sollte.

Hier kommt Caching ins Spiel. Man kann sich vorstellen, dass ein Attribut im Szenengraphen einen Teil der Szene (oder einen Unterbaum des Szenengraphen) sichtbar bzw. unsichtbar machen kann oder dass das gleiche Objekt in mehreren Detailstufen (Level of Detail – LOD) benutzt werden kann.

Diese Änderungen müssen zwischen dem Hauptspeicher und der Grafikkarte kommuniziert und abgeglichen werden, es wäre aber fatal, dies bei jeder Änderung für den gesamten Szenengraphen zu tun. Deshalb gibt es die Möglichkeit, Teile des Graphen als out-of-date (dirty) zu markieren und die Kommunikation mit der Grafikkarte so lange hinauszuzögern (um z.B. mehrere Änderungen zu sammeln), bis sie sich nicht mehr vermeiden lässt.

Wettervorhersage

Es ist unmöglich, detailliert vorherzusagen, wie sich USD in Zukunft in kommerzieller Software ausbreiten wird und welche Aspekte von den verschiedenen Softwareherstellern unterstützt bzw. ausgebaut werden. Ich wage die These, dass zunächst die Renderer von USD profitieren werden, da in naher Zukunft nicht mehr Plug-ins und Add-ons für jedes DCC-Tool (Digital Content Creation) geschrieben werden müssen, sondern nur noch ein Hydra Render Delegate Plug-in und eine Handvoll Skripte, die dem DCC-Tool ermöglichen, ein User Interface für verschiedenste Renderer-spezifischen Parameter zu generieren und dem Benutzer zur Verfügung zu stellen.

Langfristig erwarte ich, dass mehr und mehr DCCs intern auf eine Datenstruktur umstellen, die kompatibel zu den Vorgaben von USD ist. Diese Änderungen sind weniger sichtbar für den Benutzer, es sei denn, es fällt Funktionalität früherer Versionen weg bzw. der Workflow wird überarbeitet, um den Vorgaben von USD besser zu entsprechen.

Für Houdini konnte man das bereits auf der Siggraph 2019 sehen (z.B. Scott Keating stellte Solaris und Karma vor: **bit.ly/karma_solaris**). Momentan existiert der alte Workflow noch, bis USD besser getestet und integriert wird, aber es ist zu vermuten, dass in Zukunft USD immer wichtiger werden wird und der alte Workflow langsam immer unsichtbarer in der grafischen Benutzeroberfläche wird.

Tangent Animation hat, laut dem CG Channel, Blenders Renderer Cycles bereits im Viewport von Houdini zum Laufen gebracht, und ich arbeite an einem ähnlichen Projekt für meinen Rust-basierten Renderer rs-pbrt (**rs-pbrt.org**). **› ei**

Jan Walter hat nach dem Studium der Informatik einige Zeit an VR/AR-Projekten in Berlin gearbeitet, bis es ihn ins Ausland zog. In den Niederlanden, England und den USA hat er für mehrere Film/Werbe-Firmen gearbeitet, u.a. auch an Blender, bevor dieses Open Source wurde. Momentan arbeitet er für The Mill (Technicolor) von zu Hause aus. **janwalter.org**

Links

Mehr zu Render Delegates und die entsprechenden Links und Videos:
▷ **https://www.janwalter.org/jekyll/rendering/usd/hydra/2020/06/14/usd-hydra-render-delegate.html**

Physically Based Rendering (PBR) mit Rust:
▷ **https://www.rs-pbrt.org/**

Testszenen zum Download – für Appleseed, Arnold, Cycles, Indigo (PIGS), LuxCoreRender, Luxrender, Maxwell, Mental Ray, PBRT, PRMan und Radiance:
▷ **https://www.janwalter.org/download/**