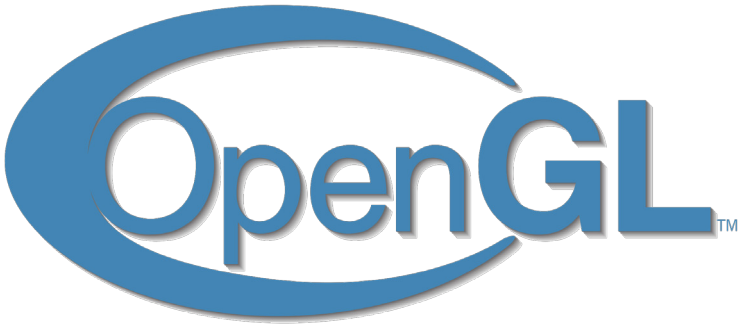
Proceedings of the advanced seminar on modern OpenGL

Chair of computer graphics and visualization

summer semester 2015



Computer Graphics
and Visualization

Preface

OpenGL (Open Graphics Library) is a hardware- and platform independent API for the generation of 2D and 3D graphics. Since version 1.0, that was officially specified by the Architecture Review Board (ARB), it has evolved to be one of the major APIs for GPU based rendering. Today, the specification contains more than 200 commands and data structures for all areas of computation with graphics cards.

During its more than 20 years long development history, the API has undergone massive changes. Initially, OpenGL was heavily based on abstraction and provided high level functionality to configure an otherwise fixed rendering pipeline. This approach was consistent with GPUs at the time where most of the functionality was hard-wired. However, modern GPUs account for the demand of high data throughput and flexibility. In order to catch up with this development, the OpenGL API changed drastically, for example by the introduction of shaders or the massive use of buffers. With recent versions of OpenGL 4, a variety of problems that used to be hard to tackle or required tricks and even misuse of the rendering pipeline, can be solved in an efficient and elegant manner.

This collection contains the work of the advanced seminar on computer graphics and visualization of the summer semester 2015 on modern GPU programming. It is divided into three areas. First, basics are laid out that provide a detailed overview on the different shader stages. Then, a selection of common problems is described and elegant solutions are shown. Finally, frameworks for general programming on the GPU and alternative APIs such as DirectX, Mantle or Metal are presented. The collection ends with an overview on Vulkan, which is considered to be the long term successor of OpenGL.

Table of Contents

1 Basics

Grundlagen Shader

L. Zepner[1]

[1]TU Dresden, Germany

Abstract

Dieses Paper gibt einen kurzen Überblick über die sechs Shader der OpenGL API. Zu jedem Shader werden die charakteristischen Eigenschaften, wie Ein- und Ausgaben, als auch Aufgaben definiert und mit knappen Beispielen verdeutlicht. Besonderer Fokus liegt dabei auf dem Unterschied zwischen den einzelnen Shadern und ihrem Beitrag zu einer Grafikanwendung.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Grundlagen Shader—Vertex-Shader, Fragment-Shader, Tesselation Control-Shader, Tesselation Evaluation-Shader, Geometry-Shader, Compute-Shader

1. Einleitung

OpenGL dient als abstrakte Schicht zwischen Anwendung und Hardware und unterteilt das Grafiksystem in verschiedene Bereiche. Durch die Verwendung der OpenGL API ist es möglich auf dieses Grafiksystem zuzugreifen und plattformunabhängig Grafikanwendungen zu entwickeln. Programmieren lassen sich die Anwendungen über Shader, welche durch fest eingebaute Teile der Pipeline verbunden sind.
Shader werden in der OpenGL Shading Language geschrieben und in einem Programm-Objekt miteinander verlinkt. Jeder Shader hat seinen eigenen Wirkungsbereich und festgelegte Ein- und Ausgabemöglichkeiten, welche im Folgenden beschrieben werden. [SWH13]

2. Die Grafikpipeline

Die Grundlage der OpenGL Programmierung ist die Verwendung von Shadern. Dabei handelt es sich um einzelne Programme, welche vom Nutzer implementiert werden können und die dadurch das Rendering der Grafik beeinflussen. Die Grundlage im Hinblick auf die Infrastruktur zwischen den einzelnen Teilen bildet die Grafikpipeline. Sie bestimmt das Verhalten und die Reihenfolge der einzelnen Phasen der Grafikprogrammierung. [RK06]
Die Figure 1 gibt einen groben Überblick über die einzelnen Abschnitte.

Die Pipeline setzt sich aus programmierbaren und festgelegten Funktionsbereichen zusammen. Die Shader werden in der Grafik durch blaue Boxen hervorgehoben, sie sind

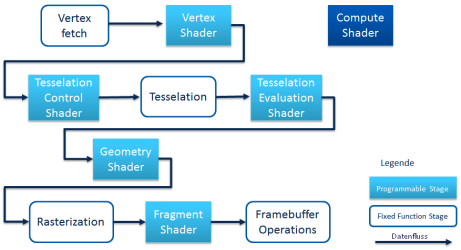


Figure 1: *Überblick über die Grafikpipeline*

der Teil, der von einem Nutzer frei definiert werden kann. Die festgelegten Bereiche, hier mit runden Ecken dargestellt, sind nicht frei programmierbar. [SWH13]
Die Grafikpipeline beinhaltet fünf integrierte Shader und einen extra Shader. Der Compute-Shader besitzt eine Sonderstellung unter den Shadern, da dieser kein Teil der Grafikpipeline ist und in der Regel nur indirekt zum Rendern von Grafiken genutzt wird. Aus diesem Grund wird er in Abschnitt 8 gesondert angesprochen.
In der Minimalkonfiguration besteht ein Shaderprogramm aus mindestens einem Vertex-Shader, welcher alleine jedoch noch keine sichtbare Ausgabe erzeugen kann, weswegen oft auch ein Fragment-Shader integriert wird. Punkte, Linien und Dreiecke bilden die Basiseinheiten von OpenGL auf welchen die Shader operieren und komplexere Primitive generieren. Dieser Prozess unterteilt sich in zwei größere Teile, welche sich in der Pipeline wiederfinden. Zuerst werden die

Primitive im Frontend verarbeitet und in Punkte, Linien oder Dreiecke umgewandelt, bevor sie in der Rasterisierungsphase an das Backend weitergegeben werden, welches mit den einzelnen Pixeldaten ein Ergebnis generiert. [SWH13]

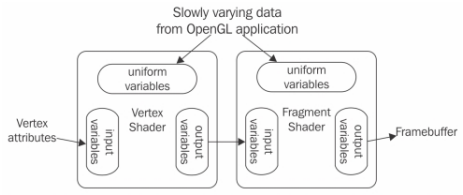


Figure 2: *Der Datenfluss in der Pipeline [SWH13]*

Die Pipeline gibt nicht nur eine Reihenfolge für die Ausführung der einzelnen Phasen an, sondern definiert auch eine feste Richtung für den Datenfluss (Abb. 2), nach welchem Eingabedaten die Pipeline komplett durchlaufen müssen. Aus diesem Grund lassen sich dich die Ein- und Ausgaben im Vorfeld strukturieren und eine Grundordnung festlegen.
Zum einen haben Shader Zugriff auf *uniform Variablen*, welche zuvor im Programm definiert wurden und von den Shadern nur gelesen, nicht jedoch geändert, werden können. Sie werden vom CPU-Programm übergeben und müssen nicht extra von einem Shader zum nächsten weiter geleitet werden, sondern können von jedem Shader in jeder Phase genutzt werden. Zum anderen existieren die Daten, welche die Pipeline durchlaufen. Sie sind mit *in* und *out* gekennzeichnet und können so von einer Phase zur nächsten übergeben werden. Dabei handelt es sich zum einen um vorher im Programm festgelegte Vertex-Attribute, welche zu Beginn der Pipeline übergeben werden müssen und zum anderen sind es Daten, die erst im Shader angelegt werden. Beispiele dafür sind Position, Farbe und Normalen. Um eine korrekte Übergabe der Daten zu gewährleisten, müssen die Ausgabedaten des einen Shaders den gleichen Namen tragen, wie die Eingabedaten des folgenden. [SWH13, sha15]

3. Der Vertex-Shader

Die Hauptaufgabe des Vertex-Shaders besteht darin Vertices zu manipulieren und dadurch Eigenschaften, wie zum Beispiel die Position, zu verändern. Er stellt die erste programmierbare Schnittstelle in der Grafikpipeline dar und ist der einzige obligatorische Teil für die Zusammensetzung eines Shaderprogramms. Seine Daten erhält der Shader über das *Vertex Fetching*, einen vorgegebenen Teil, welcher die zuvor im Programm definierten Eingabevariablen an den Vertex-Shader übergibt.
Da der Shader bei gleichen Eingabewerten identische Ergebnisse berechnet, können, um Ressourcen zu sparen, Ergebnisse wiederverwendet werden, anstatt sie doppelt zu berechnen. Verwendung findet das beim *Indexed Rendering*. Hier werden zum Beispiel verschiedene Indices festgelegt, anhand derer OpenGL feststellt, ob diese bereits in einem früheren Aufruf des Shaders genutzt wurden. Umgesetzt wird das mit Hilfe eines Caches für die Ergebnisse, so dass bei wiederholter Verwendung eines Index-Paares auf diese zugegriffen werden kann. Aus diesem Grund wird der Shader mindestens einmal pro eindeutigem Vertex-Paar aufgerufen. Die verarbeiteten oder neu angelegten Daten gibt der Vertex-Shader anschließend an den nächsten Teil der Grafikpipeline weiter. [Ver15]
Ein klassisches Beispiel für die Verwendung dieses Shaders ist ein einfacher *Pass-Through-Shader*, welcher die Eingabedaten unverändert an den nächsten Bearbeitungsblock weitergibt. Zu sehen ist ein solches Beispiel im Quellcode 1, bei welchem die erhaltene Position ohne Veränderung weitergegeben wird. Die erste Zeile gibt die Version des Shaders an. Als nächstes werden die Eingabevariablen mit dem Schlagwort *in* und die Ausgabevariable mit dem Schlagwort *out* gesetzt. [SWH13, Ver15]

Quellcode 1: *Pass-Through-Vertex-Shader*

```
#version 330
in vec4 position;
in vec4 vs_color;

out vec4 color;

void main(){
        color = vs_color;
        gl_Position = position;
}
```

Ein mögliches Ergebnis eines Vertex-Shaders in Verbindung mit einem Fragment-Shader ist in Figure 3 zu sehen. Dabei handelt es sich um eine *Per-Vertex Beleuchtung* mittels Gourand Shading.

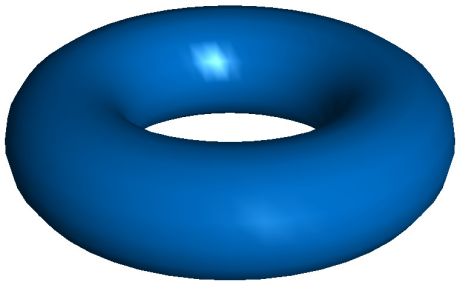


Figure 3: *Per-Vertex Beleuchtung [hig13]*

Die Tabelle 3 gibt abschließend einen Überblick über die Ein- und Ausgaben des Shaders und dessen Aufruf-Häufigkeit.

Eingabe:	Vertex Attribute
Ausgabe:	mind. Vertex Position
Aufruf-Häufigkeit:	mind. einmal pro eindeutigem Vertex Paar

Tabelle 3

4. Der Fragment-Shader

Der Fragment-Shader steht zwar in der Grafikpipeline ganz zum Schluss, jedoch ist er unerlässlich für die Ausgabe auf dem Bildschirm.
Bevor die Daten aus dem ersten Teil der Pipeline an den Fragment Shader übergeben werden können, müssen sie in der *Rasterization Phase* bearbeitet werden. Nachdem die Vertices in Linien und Dreiecke gruppiert worden sind, werden sie auf die im Display sichtbaren Bereiche beschränkt, also geclippt, und anschließend zur Rasterisierung weitergegeben. Dies bedeutet, dass die Vertex-Repräsentation der Daten in diskrete Einheiten gerastert wird, welche als Fragmente bezeichnet werden. Jedes Fragment überdeckt genau einen Pixel. Im Fragment-Shader selbst werden die Eigenschaften der Fragmente festgelegt. Dabei kann es sich zum Beispiel um die Beleuchtung der Fragmente, das Material, die RGB-Farbe oder deren Tiefeninformationen handeln. [fs15, SWH13]
Die Ein- und Ausgabe von jeweils einem Fragment machen dabei deutlich, dass der Shader einmal pro Fragment aufgerufen wird. [fs15]
Typische Anwendungsbeispiele sind verschiedene Beleuchtungsarten, sowie zum Beispiel *Bump Mapping*. Figure 4 zeigt eine *Per-Fragment Beleuchtung* mittels Phong Shading.

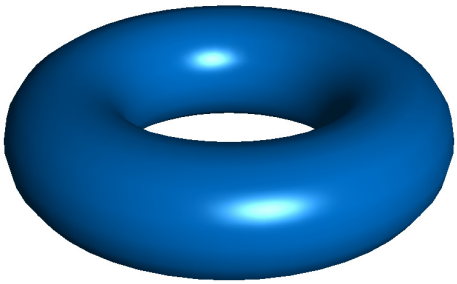


Figure 4: *Per-Fragment Beleuchtung [hig13]*

Die Tabelle 4 gibt abschließend einen Überblick über die Ein- und Ausgaben des Shaders und dessen Aufruf-Häufigkeit.

Eingabe:	ein Fragment
Ausgabe:	ein Fragment
Aufruf-Häufigkeit:	einmal pro Fragment

Tabelle 4

5. Die Tesselation-Phase

Bei der Tesselation-Phase werden Primitive höherer Ordnung in kleinere Primitive unterteilt, um damit unter anderem den Detailgrad eines Netztes zu erhöhen. Diese komplexen Primitive werden auch als *Patch* bezeichnet und es ist festgelegt, dass die Grafikkarte eine Patchgröße von mindestens 32 Vertices unterstützen muss. Diese Phase lässt sich in den Tesselation Control-Shader, die Tesselation-Engine und den Tesselation Evaluation-Shader untergliedern. [SWH13, tes15]

5.1. Der Tesselation Control-Shader

Der erste Teil der Tesslation-Phase, der Tesselation Control-Shader (TCS) bestimmt die *Tesselation-Level* pro *Patch* und verarbeitet die Eingabedaten. Dabei verarbeitet der TCS einen Patch, welcher aus einer Menge von Vertices besteht, die *Control Points* genannt werden. Im Standardfall besteht ein Patch aus drei Control Points, was sich jedoch implementierungsabhäng verändern lässt. [SWH13] Der TCS wird einmal pro Vertex eines Patches aufgerufen und schreibt bei jedem Aufruf genau einen Vertex als Ausgabe in den Ausgabepatch. Da verschiedene Aufrufe denselben Patch füllen ist es wichtig, dass die einzelnen Phasen miteinander verbunden sind und ihre Ausgabewerte miteinander teilen. [tcs15]
Bei der Bestimmung des Tesselation-Levels legt der TCS ein inneres und ein äußeres Tesselation-Level fest. Sie bestimmen wie ein Primitiv unterteilt wird. Figure 5 zeigt die Tesselation eines Primitives. In horizontaler Richtung wurde eine Unterteilung in fünf Teile vorgenommen und in vertikaler Richtung eine äußere Unterteilung von drei Teilen. Die inneren Level wurden mit neun (horizontal) und sieben (vertikal) definiert. [SWH13]

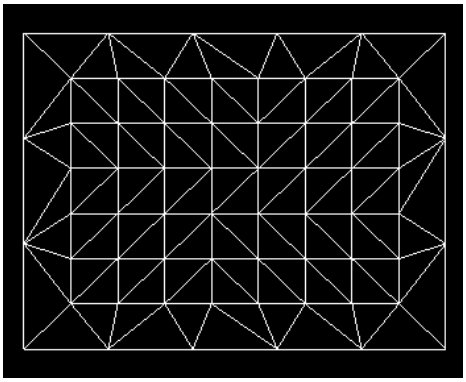


Figure 5: *Tesselierung [SWH13]*

Diese Werte werden an die nicht programmierbare Tesselation-Engine weitergereicht, welche sie verarbeitet. Sie unterteilt das Primitiv in die entsprechenden Basiseinheiten und es entsteht ein abstrakter Patch, welcher die Koordinaten der einzelnen Vertices enthält. Dieser Patch

wird anschließend an den Tesselation Evaluation-Shader (TES) weitergereicht. [tes15] Neben den Daten für die Tesselation-Engine gibt der TSC Daten direkt an den TES weiter. Werte, die den ganzen Patch betreffen, wie zum Beispiel die Farbe des Patches, werden dabei als *per patch* gekennzeichnet und müssen nicht pro Vertex an den TES gegeben werden. [SWH13]
Eine Besonderheit bei der Ausgabe des TCS ist die Beschränkung. Er kann nicht mehr als 4096 Komponenten pro Patch ausgeben. Eine Komponente ist definiert als Basiseinheit, das heißt ein float ist zum Beispiel eine Komponente, und ein drei dimensionaler Vektor zählt als drei Komponenten. [tcs15, SA15]
Die Tabelle 5.1 gibt abschließend einen Überblick über die Ein- und Ausgaben des Shaders und dessen Aufruf-Häufigkeit.

Eingabe:	Menge von Vertices
Ausgabe:	Tesselation-Level ein Vertex des neuen Patches
Aufruf-Häufigkeit:	einmal pro Vertex des Patches

Tabelle 5.1

5.2. Der Tesselation Evaluation-Shader

Der TES berechnet aus dem abstrakten Patch und den Vertexdaten, die er vom TCS bekommt, die interpolierten Positionen und verschiedenen per-Vertexdaten, welche anschließend der nächsten Pipeline-Stufe übergeben werden können. Zwar wird im Vorfeld festgelegt, ob Linien oder Dreiecksstreifen generiert werden sollen, jedoch hat der TES die Möglichkeit, dies zu überschreiben und pro Vertex Punktprimitive zu generieren. [tev15, SWH13]
Aufgerufen wird der TES mindestens einmal pro tesseliertem Vertex, es ist jedoch möglich, dass er öfter für den selben Vertex aufgerufen wird und je höher die Tesselation-Level sind, umso größer ist die Anzahl der Aufrufe. [tev15]
Zu den Anwendungsfällen der Tesselation-Phase zählen vor allem die unterschiedlichen Tesselation-Level. Dadurch ist es möglich einzelne Szenen so zu rendern, dass im Hinblick auf den Detailgrad sehr unterschiedliche Ergebnisse entstehen. Es ist sowohl möglich damit glattere Oberflächen mit höheren Tesselation-Leveln zu erzeugen, als auch gröbere mit niedrigeren Leveln. Verwendung findet dies zum Beispiel bei Szenen mit sehr nahen Objekten, welche sich durch viele Details und glatte Oberflächen auszeichnen können, aber auch Szenen mit sehr weit entfernten Objekten, welche keinen so hohen Detailgrad brauchen. Zu sehen ist dies in Figure 6. [SWH13]

Die Tabelle 5.2 gibt abschließend einen Überblick über die Ein- und Ausgaben des Shaders und dessen Aufruf-Häufigkeit.

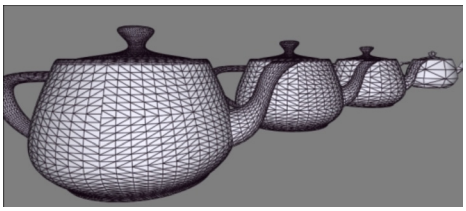


Figure 6: *Unterschiedliche Tesselation-Level auf Grund ihrer Tiefe [SWH13]*

Eingabe:	Menge von Koordinaten der Vertices
Ausgabe:	ein Vertex
Aufruf-Häufigkeit:	mind. einmal pro tesseliertem Vertex

Tabelle 5.2

6. Der Geometry-Shader

Der Geometry-Shader besitzt eine Sonderstellung unter den Shadern, da er komplette Primitive in einem Aufruf verarbeiten kann. Er hat Zugriff auf alle Vertices eines Eingangsprimitives und kann diese verändern, er kann den Primitiv-Typ ändern und aus Dreiecksstreifen zum Beispiel Viereckstreifen machen. Desweiteren ist es möglich im Geometry-Shader neue Primitive zu erzeugen oder vorhandene zu entfernen. [gs15, SWH13] Zum Verfeinern der Geometrie nutzt man in der Regel den Tesselation-Shader, während der Geometry-Shader eher beim Layered Rendering, um zum Beispiel ein Primitiv in mehreren Bildern zu rendern, oder beim Transform Feedback zum Einsatz kommt. [gs15] Als Eingabe erhält der Geometry-Shader Punkte, Linien oder Dreiecke, welche zu Punkten, Linienstreifen oder Dreiecksstreifen zusammengefügt werden. Er wird einmal pro Primitiv aufgerufen und ähnlich dem Tesselation-Shader ist der Geometry-Shader darin beschränkt, wie viele Primitive in einem Aufruf ausgegeben werden können. Dieser Maximalwert setzt sich aus der Anzahl der Komponenten pro Vertex und der Anzahl der Vertices zusammen. Das Produkt dieser beiden Werte darf das gesamte Ausgabemaximum von 1024 nicht überschreiten. Wenn zum Beispiel pro Vertex 12 Komponenten ausgegeben werden, reduziert sich die Anzahl der möglichen Vertices von 256 auf 85. [gs15, SA15]

Quellcode 2: *Pass-Through-Geometry-Shader*

```
#version 430

layout (triangles) in;
layout (triangle_strip) out;
layout (max_vertices = 3) out;

void main(void){
 int i;

  for (i = 0; i < gl_in.length(); i++){
    gl_Position = gl_in[i].gl_Position;
          EmitVertex();
          }
EndPrimitive();
}
```

Auch in seinem Aufbau unterscheidet sich der Geometry-Shader von den anderen. Der Quellcode 2 zeigt einen einfachen Pass-Through-Geometry-Shader. Nachdem die Shader Version gesetzt ist, werden für die Ein- und Ausgabevariablen die Primitive-Modi festgelegt. Der Shader erhält Dreiecke und soll Dreieckstreifen mit 3 Vertices generieren. Essentiell bei der Implementierung sind die Funktionen *EmitVertex()* und *EndPrimitive()*. Erstere wird nach jedem Vertex, der der Pipeline übergeben werden soll, aufgerufen. Dabei ist es implementierungsabhängig, wieviele Vertices zu einem Primitiv zusammengefügt werden, so dass die Funktion gegebenenfalls mehrfach aufgerufen werden muss. Wurde jeder Vertex, der ein einzelnes Primitiv ausmacht, an die Pipeline weitergereicht, wird dieser mit der Funktion *EndPrimitive()* signalisiert, dass das Primitiv abgeschlossen ist. Sollen mehrere voneinander getrennte Primitive gerendert werden, unterteilen sich die Berechnungen jeweils durch einen Aufruf der EndPrimitive()-Funktion. [SWH13]

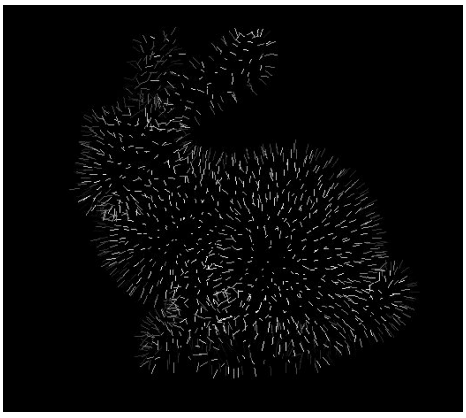


Figure 7: *Veränderung des Primitivtyps im Geometry-Shader [Wol13]*

Die Figure 7 zeigt das Ergebnis nach der Änderung des Primitivtyps. Der Shader hatte als Eingabe Dreicke, aus welchen für die Ausgabe die Normalen berechnet und als Linien ausgegeben wurden. [Wol13]
Die Tabelle 6 gibt abschließend einen Überblick über die Ein- und Ausgaben des Shaders und dessen Aufruf-Häufigkeit.

Eingabe:	Primitiv eines Typs
Ausgabe:	null – mehrere Primitive eines Typs
Aufruf-Häufigkeit:	einmal pro Primitiv

Tabelle 6

7. Geometry-Shader und Tesselation-Shader im Vergleich

Der Geometry-Shader ist ein sehr mächtiger Teil der Grafikpipeline, welcher dadurch jedoch keine gute Performance liefert. Die Verarbeitung im Shader erfolgt seriell bzw. nicht vollparallel und er ist in seiner Ausgabe beschränkt. Im Gegesatz dazu steht der Tesselation-Shader, welcher nicht so viele Funktionen vorweisen kann, dafür jedoch eine bessere Perfomance aufweist. Seine Verarbeitung ist vollparallel und obwohl er ein Limit für seine Ausgabekomponenten hat, sind die maximalen Werte höher, als die des Geometry-Shaders.
Abschließend lässt sich sagen, dass sich der Geometry-Shader vor allem zum Verändern der Geometrie einer Topologie und zum Layered Rendering eignet. Sollen zum Beispiel nicht das Netz, sondern dessen Normalen angezeigt werden, wie in Figure 7 oder soll das Netz visuell auseinander genommen werden, bietet sich die Verwendung eines Geometry-Shaders an. Der Tesselation-Shader hingegen kann zur Erzeugung vieler neuer Vertices verwendet werden, um ein Netz detailreicher oder auch detailärmer zu gestalten. [Gro15, SA15]

8. Der Compute-Shader

Außerhalb der Grafikpipeline befindet sich der Compute-Shader. Dieser soll die Rechenkapazität, die ein Grafikprozessor besitzt auszunutzen und die Aufgaben von den Prozesskernen auf die Grafikkarte umlagern, so dass auch nicht-grafische, parallele Datenverarbeitung auf dieser ausgeführt werden kann. Abseits von OpenGL gibt es bereits Möglichkeiten die Grafikprozessoren dafür zu nutzen, wie zum Beispiel OpenCL. Mit Hilfe des Compute-Shaders wird der direkte Zugriff auf die Grafikkerne jedoch in OpenGL eingegliedert. [Bai13]
Der Compute-Shader hat keine fest definierten Ein- oder Ausgaben, da er seine eigene einstufige Pipeline darstellt und keinen Vorgänger oder Nachfolger hat, der ihm Daten weitergeben könnte. Aus diesem Grund ist es nicht möglich ihn mit anderen Shadern in einem Programmobjekt

zu verlinken, sondern notwendig, dass er sein eigenes Programmobjekt definiert. Dieses kann zum Beispiel vor dem Grafikobjekt aufgerufen werden, damit die vom Compute-Shader manipulierten Daten von den anderen Shadern verwendet werden können, oder es wird für die Nachbearbeitung im Anschluss genutzt. Das sind jedoch nur zwei von vielen Möglichkeiten den Compute Shader einzusetzen.
Der Shader hat dadurch, dass er ein eigenes Programmobjekt bildet, keine Möglichkeit auf die Daten innerhalb der Grafikpipeline zuzugreifen. Das bedeutet, dass er die Daten zwischen den anderen Shadern nicht nutzen kann, sondern nur die Anfangs- oder Endrepräsentation der Daten verwendet. [cs15]
Der Shader arbeitet mit *globalen Work-Groups*, welche sich in *lokale Work-Groups* unterteilen, welche wiederum aus *Work-Items* bestehen. Eine globale Work-Group wird durch einen Aufruf an OpenGL gesendet und daraufhin in einen dreidimensionalen Raum lokaler Work-Groups unterteilt, deren Anzahl vom Nutzer festgelegt werden kann. Diese lokalen Gruppen bestehen wiederum aus einem dreidimensionalen Block von Work-Items, welche in jeweils einem Compute-Shader-Aufruf verarbeitet werden. Der Aufbau ist in Figure 8 zu sehen. Die Bezeichnung „Work Group“ steht in diesem Beispiel für eine lokale Work-Group und „Invocation“ für ein Work-Item.

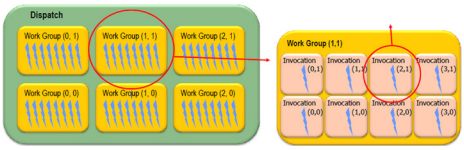


Figure 8: *Aufbau einer globalen Work-Group [Lic12]*

Da es keine fest eingebauten Ausgabevariablen im Shader gibt, muss der Datentransfer über die *Buffer Objects* oder *Images* geschehen. Um die Verarbeitung großer Datenmengen zu gewährleisten werden *Shader Storage Buffer Objects (SSBO)* verwendet. [Bai13] Sie ähneln in ihrer Handhabung den textitUniform Buffer Objects. Mit ihnen ist es möglich auf den gespeicherten Daten sowohl Lese- und Schreibzugriffe, als auch atomare Operationen auszuführen. Die geforderte Mindestgröße eines SSBOs beträgt 16 MB. [Lic12] Die Anwendungsfälle sind zum einen die große Parallelverarbeitung von Daten und zum anderen zum Beispiel das Aktualisieren von Geometrien oder Partikelsystemen, wie in Figure 9 zu sehen ist. [Bai13]

Die Tabelle 8 gibt abschließend einen Überblick über die Ein- und Ausgaben des Shaders und dessen Aufruf-Häufigkeit.

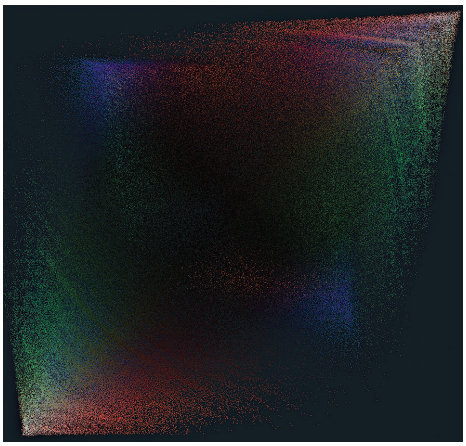


Figure 9: *(Interaktives) Partikelsystem [Bai13]*

Eingabe:	Zugriff auf Uniform Variablen Buffer Objects
Ausgabe:	direktes Schreiben in den Speicher
Aufruf-Häufigkeit:	einmal pro Work-Item

Tabelle 8

References

[Bai13] BAILEY M.: Using gpu shaders for visualization, part 3. *Computer Graphics and Applications, IEEE 33*, 3 (May 2013), 5–11. doi:10.1109/MCG.2013.49. 5, 6

[cs15] Compute-shader, 2015. URL: https://www.opengl.org/wiki/Compute_Shader. 6

[fs15] Fragment-shader, 2015. URL: https://www.opengl.org/wiki/Fragment_Shader. 3

[Gro15] GROSCH T.: Geometry- und tesselation-shader, 2015. URL: http://www.rendering.ovgu.de/rendering_media/downloads/folien/gpuprog/05_geometryshader.pdf. 5

[gs15] Geometry-shader, 2015. URL: https:///www.opengl.org/wiki/Geometry_Shader. 4

[hig13] Smooth specular highlights, 2013. URL: https://en.wikibooks.org/wiki/GLSL_Programming/GLUT/Smooth_Specular_Highlights. 2, 3

[Lic12] LICHTENBELT B.: Opengl 4.3 overview, 2012. URL: https://www.khronos.org/assets/uploads/developers/library/2012-siggraph-opengl-bof/OpenGL-4.3-Overview-SIGGRAPH_Aug12.pdf. 6

[RK06] ROST R., KESSENICH J.: *OpenGL Shading Language*. Graphics programming. Addison-Wesley, 2006. URL: https://books.google.de/books?id=-SI07OhiQMAC. 1

[SA15] SEGAL M., AKELEY K.: The opengl graphics system: A specification, 2015. URL: https://www.opengl.org/registry/doc/glspec44.core.pdf. 4, 5

[sha15] Shader, 2015. URL: https://www.opengl.org/wiki/Shader. 2

[SWH13] SELLERS G., WRIGHT R. S., HAEMEL N.: *OpenGL SuperBible: Comprehensive Tutorial and Reference*. Addison-Wesley Professional, 2013. 1, 2, 3, 4, 5

[tcs15] Tessellatio control-shader, 2015. URL: https://www.opengl.org/wiki/Tessellation_Control_Shader. 3, 4

[tes15] Tesselation, 2015. URL: https://www.opengl.org/wiki/Tessellationr. 3, 4

[tev15] Tessellation evaluation-shader, 2015. URL: https://www.opengl.org/wiki/Tessellation_Evaluation_Shader. 4

[Ver15] Vertex-shader, 2015. URL: https://www.opengl.org/wiki/Vertex_Shader. 2

[Wol13] WOLFF D.: *OpenGL 4 Shading Language Cookbook Second Edition*. Packt Publishing, 2013. 5

2 Common Applications

OpenGL Tessellation

Maximilian Richter

Abstract

This paper provides a detailed overview on how tessellation works in OpenGL 4.0 and an exemplary contribution to terrain rendering. All stages of the tessellation pipeline and their algorithms are explained, including input and output parameters of the tessellation shaders and an discussion of their limitations and issues. Approaches to tessellation based level of detail calculation are reviewed and evaluated with examination of disadvantages strategies used before tessellation was introduced had.

1. Introduction

Since OpenGL 4.0 hardware tessellation has been part of the OpenGL Core standard. Tessellation is a method of subdividing polygons into finer primitives. This allows runtime control of the GPU on how detailed surfaces should appear. For example in terrain rendering, meshes can be tessellated to generate more detail if they are closely visible and reduced in their detail when they are less visible.

This paper is considered as an introduction to OpenGL's tessellation process. The remainder of this paper is organized as follows: In section 2 an overview of how tessellation is integrated in OpenGL is given, before detailing the single stages of tessellation. This section closes with an evaluation of the limits of hardware tessellation. Approaches for adaptive terrain tessellation are described in section 3 leading to Section 4 closing the paper by summarizing the main information.

2. Tessellation in OpenGL

Surfaces desired to be tesselated, have to be declared as patches. A Patch is a collection of vertices with additional per-patch attributes. The vertices have no implied geometric ordering. Their interpretation has to be defined in the tessellation shaders. The tessellation control shader recieves the patches and computes per-vertex and per-patch attributes for an output patch. Based on tessellation levels computed by the control shader as per-patch atrributes the tessellation primitive generator generates new vertices. The position and attributes of these vertices are finally computed by the tessellation evaluation shader. The generated primitves are then used by subsequent shader stages as input. Tessellation is located between vertex shader and geometry shader in the rendering pipeline. Both the control shader and the evaluation shader are optional. If no control shader is active, the patch is directly provided by the vertex shader to the primitive generator and its tessellated by default tessellation levels. If no evaluation shader is active, no tessellation will take place. [LLKM*15, Dud12, Wol13]

2.1. Tessellation Control Shader

The control shader is the first stage of tessellation. Once an input patch is recieved from the vertex shader, the control shader is executed once per vertex of the output patch. The number of vertices in the output patch is specified at linking time by the output layout qualifier `vertices` in the control shader source code. [LLKM*15, Boe10]

Input. The control shader consumes the input patch provided by the vertex shader. This data is stored into a built-in array `gl_in`. `gl_in` is an array of structures, where each structure is holding values for one specific vertex of the input patch. The length of `gl_in` equals the maximum patch size `gl_MaxPatchVertices`. Since this length may be larger than the number of vertices of the input patch, an additional variable `gl_PatchVerticesIn` stores the actual number of input vertices. As the control shader is invokated once per output vertex, it is necessary to know which vertex the current invokation is processing and by that at which position in the output array to write in. The built-in variable `gl_InvokationID` provides this information. [LLKM*15, Bus15]

Output. After processing the input patch is written to `gl_out`. `gl_out` is an array holding structures of per-vertex values, similar to `gl_in`, but ment to be used by subsequent shaders. The control shader outputs two arrays, one for inner tessellation levels and one for outer tessellation levels. Inner tessellation levels determine how many primitves will

be generated inside the patch. They are stored as array of 2 floating point values to `gl_TessLevelInner`. As inner tessellation levels are per patch values, they potentially differ among neighbouring patches. To ensure, that the outer edges, these patches share, fit together without cracks, their subdivision has to be controlled independently. [LLKM*15, Bus15] Outer tessellation levels determine to what extent the outer edges will be subdivided. Its implemented as array of four floating point values `gl_TessLevelOuter`.

The outer triangle and the outermost inner triangle of an tessellated triangle is illustrated in figure 1. Everything inside the outermost inner triangle is tessellated according to the first inner tessellation level `i[0]`. The outer edges ere tessellated according to the first three outer tessellation levels `o[0..2]`. Contraty to the triangle the quad uses all four outer and both inner tessellation levels.

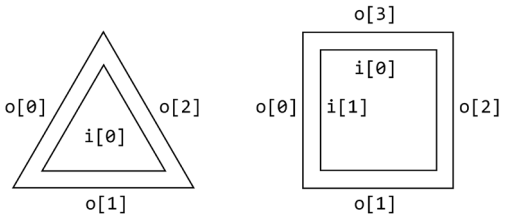


Figure 1: *Responsibility of different tessellation levels.*

2.2. Tessellation Primitive Generator

The primitive generator is the fixed function stage responsible for the actual generation of new primitives. It is not programmable, yet configurable in how the tessalation should be executed. This configuration is provided by both the control shader and the evaluation shader. The control shader sends the tessellation levels that are interpretet based on the tessellation type and the tessellation spacing. Despite being executed before the evaluation shader the primitive generator can read the tessellation type and spacing, because they are set when the shaderprogram is linked. [LLKM*15, Wol13, Bus15]

To tessellate a patch it is necessary to tessellate its edges too. Tessellating an edge means subdiving it into a series of segments. The total length of these segments equals the length of the edge. Yet they do not necessarily have to have a constant length. The length of each segment is determined by the tessellation spacing. It is specified by an input layout declaration in the evaluation shader source code, declaring `equal_spacing`, `fractional_even_spacing` or `fractional_odd_spacing` as spacing strategy. [LLKM*15, Bus15]

If equal spacing is specified, the tessellation level is rounded up to the nearest integer n and the edge is divided into n segments of equal length. Due to rounding, the tessellation level's fractional part is ignored. For continously changing tessellation levels this leads to suddely appearing primitives and flickering, as they constantly change in size. Fractional spacing's purpose is to allow more stable sized primitives.

Fractional even spacing rounds up the tessellation level to the nearest even integer n. Fractional odd spacing rounds up to the nearest odd integer n. If n equals 1 the edge will not be subdivided. Otherwise the edge will be divided into $n-2$ equal sized segments, with two additional segments of length equal to each other. The length of these two segments is determined by the fractional part of the tessellation level. Let f be the tessellation level before rounding. Then the length of the two segments, relative to the others, equals $n-f$. For the fractional even spacing shown in figure 2.II, n is rounded to 2. Since $n-f$ equals 0.5, the length of each outer segment is 25% of the inner segments' length. For the fractional odd spacing shown in figure 2.III, n is rounded to 3. This time $n-f$ equals 1.5 and the length of each outer segment is 75% of the inner segments' length. Determined by the fractional part the length of the two segments will apparently always be smaller than or equal to the length of the other segments. [LLKM*15]

Figure 2: *Three edges of equal length and tessellation factor of 1.5, tessellated by different tessellation spacings.*

OpenGL's tessellation implementation includes a collection of algorithms to subdivide surfaces. The generated primitives differ in layout and shape depending on these algorithms. Which of them is performed by the primitve generator is specified by the tessellation type: `triangles`, `quads` or `isolines`.

Triangles. For the tessellation type triangles, the primitive generator subdivides an equiliteral triangle into a collection of triangles. The area inside the original triangle is first subdivided into a set of concentric triangles. This is achieved by temporarily subdividing the three outer edges using the first inner tessellation level, generating n segments. three new Vertices are generated at the intersection of the perpendicular lines extending the vertices on each outer edge that are closest to a corner. (Fig. 3.I) If n equals 2 the inner triangle degenerates because all lines intersect at the same point in the center of the triangle, so no further subdivision is possible. (Fig. 3.II) If n equals 3 the inner triangle is the innermost concentric triangle and not further subdivided. Otherweise each edge of the inner triangle is subdivided into $n-2$ segments by generating new vertices at the intersection with perpendicular lines through the inner vertices on

the outer edge. Then the subdivision process is repeated, using the generated triangle as outer triangle. (Fig. 3.II)

After all concentric triangles are generated, the area between them is filled completely with non overlapping triangles. For each pair of adjacent inner trianges each vertex creates a triangle with the two closest adjacent vertices on the other triangle. If the innermost triangle degenerates, all vertices of the containing triangle are connected to the center, creating 6 triangles.

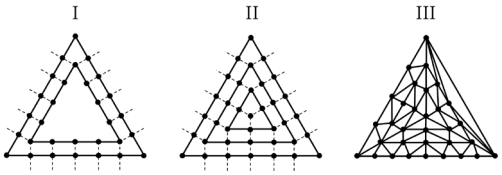


Figure 3: *Different steps of triangle tessellation. Outer tessellation levels: 4, 8, 1, 0. Inner tessellation levels: 6, 0.*

To fill the area between the outermost triangle and the outermost inner triangle the temporary subdivision of the outer edges is discarded. The edges are subdivided according the first three outer tessellation levels and the tessellation spacing. Finally the area is filled by triangles the same way as described above. (Fig. 3.III) [LLKM*15]

Quads. For the tessellation type quads, the primitive generator subdivides a rectangle into a collection of triangles. The vertical edges are subdivided into n segments using the fist inner tessellation level. The same happens to the horizontal edges using the second inner tessellation level, generating m segments. Each vertex on these edges is connected with the corresponding vertex on the parallel edge. This divides the rectangle into a grid of smaller rectangles, shown in figure 4.I. The inner region of the rectangle is the collection of all rectangles not adjacent to one of the outer edges. The boundary of this region is the inner rectangle, illustrated in figure 4.II. It degenerates, if m or n equals 2, since at least two of the edges then consist of a single point. Each of the rectangles making up the inner rectangle is then divided into two triangles. By adding a line from the vertex that is closest to the outer rectangles center to the most distant one rotational symmetry is achieved. As in figure 4.II the inner rectangle then consisits only of triangles.

It remains to fill the area between the inner and outer rectangles. Again the temporary subdivision is discarded and the outer edges are subdivided, depending on the first 4 outer tessellation evels applied counter clockwise from the left to the top edge. The area between the outer and inner rectangles is then filled by triangles, where each vertex creates a triangle with the two closest adjacent vertices on the other rectangle. (Fig. 4.III) [LLKM*15]

Isolines. For the tessellation type isolines, the primitive

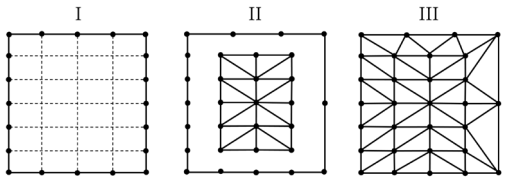


Figure 4: *Different steps of quad tessellation. Outer tessellation levels: 6, 4, 2, 3. Inner tessellation levels: 4, 6.*

generator subdivides a rectangle into a series of lines. These horizontal line-segments are connected to strips called isolines. The vertical edges of the rectangle are subdivided into n segments using the first outer tessellation level, always using equal spacing. The generated vertices are then connected horizontally to the corresponding vertex on the parallel edge. (Fig. 5.I) Like in figure 5.II, no line is drawn between the top vertex on both vertical edges. Therefore n isolines are created. Each of them is subsequently divided according to the second outer tessellation level and the tessellation spacing. (Fig. 5.III) [LLKM*15]

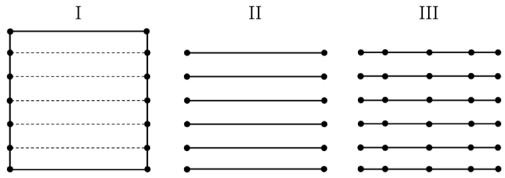


Figure 5: *Different steps of isoline tessellation. Outer tessellation levels: 6, 1.5. Inner tessellation levels are not relevant.*

2.3. Tessellation Evaluation Shader

This is the final stage of tessellation. The evaluation shader takes the abstract patch generated by the primitive generator and uses the information of the input patch to generate an output patch. The evaluation shader is invoked once per vertex of the abstract patch. So, the number of invokations depends on the tessellation levels. Each invokation produces one output vertex, therefore the number of output vertices equals the number of vertices in the abstract patch. Each invokation takes the tessellation coordinates of one vertex of the abstract patch and generates an output vertex with position and attributes of the same domain as the input patch. Tessellation coordinates represent the position of one vertex in the abstract patch as values in the range $[0,1]$. [LLKM*15, Bus15]

Input. The information of the input patch is stored in an input array `gl_in`. It receives its data from the corresponding output array `gl_out`, written by the control shader or the

vertex shader. If a control shader is active, `gl_in` equals exactly the control shader's `gl_out` output array. Otherwise the input patch is provided as vertex shader output. The structure of `gl_in` is equal to the control shaders's input array as described in 3.1, Input. The abstract patch is not entirely visible to each evaluation shader instance. As each evaluation shader invokation processes only one vertex of the abstract patch, only the tessellation coordinates of this vertex are stored as a three component vector to the variable `gl_TessCoord`.

Output. Each evaluation shader invokation outputs a series of built-in output variables, like `gl_Position` and `gl_TexCoord`, representing one vertex of the output patch. Their values are interpolated from the input patch according to the tessellation coordinares. Which interpolation algorithm is used again is specified by the tessellation type. For triangles barycentric interpolation is used, quads and isolines are interpolated linearly. The output variables behave exactly like the the equivalent vertex shader outputs. [LLKM*15]

2.4. Limitations and issues

Tessellation increases visual quality for surfaces that suffer from discretization: Curves, Noise and displacement mapping appear more detailed mith more vertices, whereas flat surfaces do not change. Therefore, despite every surface being tessellateable, its not reasonable for any kind of mesh.

OpenGL limits the tessellation levles to a maximum of 64. Cases where higher tessellation levels are needed, require alternative methods to achieve the desired level of subdivision. Patches have to be subdivided into smaller patches before tessellation is performed, what increases CPU load.

Edges, shared among neighbouring patches, potentially differ in their tessellation level. If that is the case and the generated vertices along these edges are displaced after generation, holes in the geometry appear. To avoid cracks, shared edges of equal length always have to have equal tessellation levels. Therefor the tessellation level has to be calculated depending on values identical for both edges, like the vertices' positions and the edge midpoints. [LLKM*15, Bus15, Can11]

3. Adaptive tessellation

One of the most useful applications of tessellation is the adaptive tessellation: generating vertices, and by that detail, where it contributes most to the visual quality of the scene. Former approaches to dynamic level of detail (LOD) shared multiple weaknesses: They fail to use the capabilitys of the GPU to accelerate computations. By relying on heavy preprocessing, they rather shifted load to the CPU. In this case preprocessing means generating differently detailed meshes of the same model. Harsh transitions between different detail levels cause visible artefacts. CPU load increases, as it has to determine each frame which parts of the geometry have to be drawn with a certain level of detail. Per frame transferring the choosen meshes to the GPU consumes large bandwidth capacity. [Bus15]

Terrain rendering involves large geometries to represent environmental structures. As terrains grow larger and more complex, more vertices are reqired to achieve a proper level of detail. As the number of vertices grows, performance decreases. Therefore ways to reduce the number of vertices are necessary. Terrains involve areas that contribute little to the image, being less visible due to distance or perspective. To render efficiently, the level of detail of geometry has to cohere with its visibility, leading to less visible areas being rendered with reduced detail. Transitions between different levels of detail should be imperceptivle. The continuous nature of terrain made this hard to achieve for former strategies with discrete levels of detail. Tessellation allows adaptive and continous level of detail, without the issues of former approaches. [Bus15]

To utilize tessellation, the terrain is generated as a regular grid of square patches. The tessellation level of these patches is varied depending on their visibility by the control shader. After the patches are tessellated, the generated vertices are displaced using a height map to sample from in the evaluation shader. In the following two approaches to how to calculate the tessellation level are examined. [Boe10, Can11]

3.1. Distance adaptive approach

To calculate the tessellation level for an edge, the distance of the camera to the edge is determined. Each edge's midpoint is calculated by averaging the two vertices of the edge. The midpoint is then projected to camera space to calculate the distance from edge to the the camera plane. This distance is then mapped to the range $[1,64]$ by some scaling. The outer tessellation level for this edge is the resulting value. The inner tessellation level is the average of all outer tessellation levels, therefore all outer tessellation levels have to be calculated in advance. As the order of execution of the invocations is unknown the edges can not be processed simultaneously. [Boe10, Bus15]

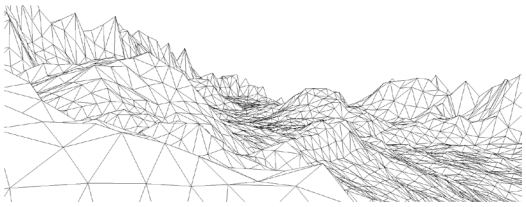


Figure 6: *Wireframe of a terrain, tesselated distance adaptive.*

The disadvantage of this approach is the independance of the size of the patches. For meshes of varying patch size,

this leads to tessellating dense regions too much, while tessellating coarse regions to less. Sone triangles generated in figure 6 cover just few pixels, as their surface is almost parallel to the view vector. There is no point in tessellating patches that cover just few pixels, but orientation is not considered in this approach. To solve these issues another heuristic considering the actual patch size and visibility is introduced. [Bus15, Can11]

3.2. Screenspace adaptive approach

This algorithm tessellates based on a target edge length, given in pixels per edge. Again each control shader invocation processes one edge. Around each edge a sphere is created, using the edge as diameter. This sphere is then projected to the screen space to calculate the screen space diameter. This diameter provides the number of pixels the edge occupies on the screen. By projecting a sphere to the screen instead of projecting an edge the result ist not depending on the orientation of the edge to the camera The tessellation level is determined by comparing this diameter to the target edge length. If the diameter length is less than the target edge length, there is no need for tessellation, since the edge is already below the target length. Otherwise the edge will be tessellated. The projected edge in figure 7 is about 6 time longer than the target length. Therefore this edge will be tessellated into 6 or more segments, depending on the tessellation spacing. The tessellation level is calculated by dividing the diameter length by the target edge length. [Bus15, Can11]

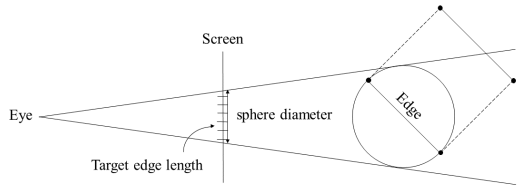


Figure 7: *Projection of a sphere, wrapped around an edge, to screen space.*

4. Conclusion

By subdividing polygons into smaller primitives tessellation allows generating new vertices where they increase visual quality most. Therefore less detailed control meshes can be sent to the GPU and be tessellated if their level of detail is not sufficient. This saves memory bandwidth and increases performance, as only highly visible meshes are rendered with high level of detail. Distant, barely visible meshes are tessellated with minimal tessellation levels.

Tessellation allows the rendering of large geometries with continous, adaptive level of detail. Former approaches only supported discrete levels of detail resulting in artefacts and high processing cost. Featues like edge spacing are unique to tessellation and allow stabilizing the primitive size for constantly changing geometries.

The advantage of screen space adaptive tessellation is that it computes tessellation levels that depend not just on the distance to the camera, but also on the size of the patch. This leads to all patches appearing at the nearly same size on the screen, providing the same level of detail to each pixel, independent from the distance to the camera.

References

[Boe10] BOESCH F.: Opengl 4 tessellation, 2010. http://codeflow.org/entries/2010/nov/07/opengl-4-tessellation/.

[Bus15] BUSHONG V.: Tessellated terrain rendering with dynamic lod, 2015. http://victorbush.com/2015/01/tessellated-terrain/.

[Can11] CANTLAY I.: Directx 11 terrain tessellation, 2011. http://developer.download.nvidia.com/assets/gamedev/files/sdk/11/TerrainTessellation_

[Dud12] DUDASH B.: Dynamic hardware tessellation basics, 2012. https://developer.nvidia.com/content/dynamic-hardware-tessellation-basics.

[LLKM*15] LICHTENBELT B., LICEA-KANE B., MERRY B., DODD C., WERNESS E., SELLERS G., ROTH G., CASTANO I., BOLZ J., HAEMEL N., BOUDIER P., DANIELL P.: *ARB_tessellation_shader*. Khronos Group, 2015. https://www.opengl.org/registry/specs/ARB/tessellation_shader.txt.

[Wol13] WOLFF D.: *OpenGL 4 Shading Language Cookbook*. Packt Publishing, 2013.

The Compute Shader

Mirko Salm

TU Dresden, Germany
Chair of Computer Graphics and Visualization

Abstract

The compute shader directly exposes general purpose computation capabilities of the graphics processing unit (GPU) and is therefore an interesting alternative to the shader stages of the traditional rendering pipeline, which are intended for more specific tasks. Since compute shaders display a more low level approach to GPU programming, several concepts like thread grouping, explicit memory accesses, and synchronization become more relevant. This paper intents to provide a coarse overview of the most important aspects and concepts with regard to compute shaders.

1. Introduction

With the increasing computational power of GPUs over the last years, interest grew to utilize their data parallelism for non-graphics tasks like machine learning and monte-carlo simulations. The concept of using the GPU for this kind of applications became known under the term *General Purpose Computing on GPUs* (GPGPU) [GPG]. While in the beginning GPGPU had to be realized using the traditional graphics pipeline which is specialized for rendering tasks, over time several frameworks occurred that specifically aimed for GPGPU application like Nvidia's CUDA [Cud15], OpenCL [OCL], and OpenACC [ACC]. Following the success of these frameworks, the concept of compute shaders was introduced in Direct3D [D3D] with the arrival of version 11.0 in 2009 and later under OpenGL [OGL] 4.3 in 2012. Compute shaders allow a similar GPGPU approach to GPU programming like the dedicated frameworks but does so in the context of graphics APIs. While it is possible to combine graphics APIs and GPGPU frameworks, several disadvantages can originate from this. First, using two different APIs with different hardware support can affect the compatibility of the application negatively or, at least, does never so in a positive way. Secondly, a certain amount of overhead is introduced by the interoperability between the two APIs. Working with a single API also means both rendering tasks and GPGPU tasks are approached in a more unified fashion.

In the following, first the historical approach to GPGPU without dedicated frameworks is shortly considered before the compute shader itself is described in the subsequent chapter.

2. The Historical Approach

Before the arrival of dedicated frameworks GPGPU programs had to be implemented using the traditional rendering pipeline. Since the stages of the rendering pipeline are designed for specific rendering related tasks like vertex or fragment processing this approach can be inconvenient to work with. On the upside, being not dependent on a specific GPGPU framework a wider range of hardware can be targeted by the application. A rather straightforward approach to implement a GPGPU program using the fragment shader is by rasterizing a quad that covers the whole viewport. Every fragment is then considered to be a thread and individual threads can be identified through interpolated 2d coordinates of the four vertices. Another, more flexible approach that also allows scattered writes to the framebuffer is to generate a stream of vertices that are rasterized as points. In this case, the thread logic can be implemented in the vertex shader using `gl_VertexID` as thread identifier. Both approaches, however, hide some of the low level concepts that are exposed to compute shaders and therefore provide less optimization potential.

3. The Compute Shader

In the following subsections various concepts that are unique to the compute shader stage are presented beginning with the *compute space* based on which individual shader invocations are identified during a compute dispatch. Subsequently, a short overview of relevant resources is given. Means of synchronization are considered afterwards before *shared variables*, which provide important optimization po-

tential, are described. At last, the practical application of compute shaders is illustrated using the example of a simple water simulation.

3.1. Compute Space

To utilize threads in a meaningful way during execution, it is necessary to identify individual shader invocations through unique IDs. These IDs are used to access and process different data in different threads or to perform different computations. The IDs assigned to a specific thread depend on its position in the *compute space* [Wik15b], an abstract 3d space all invocations in a compute dispatch reside in. The compute space is organized as a two level hierarchy where the top level consist of a 3d grid of *work groups*. Each work group in turn consists of a 3d grid of invocations/threads, forming the bottom level of the hierarchy. This concept is illustrated in figure 1.

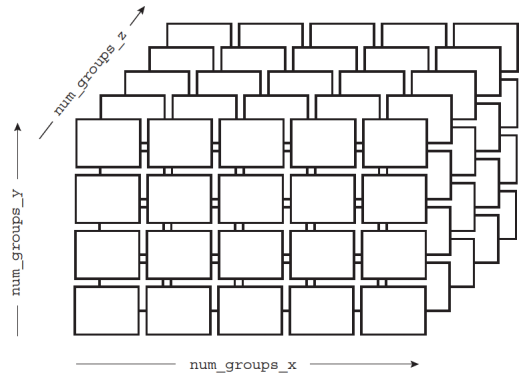


Figure 1: *The compute space illustrated as a 3d array of rectangles. The rectangles represent the individual work groups. Every work group contains the same number of threads. The threads inside each work group are organized in a 3d grid as well (not depicted here). [Com]*

The number of work groups, the *work group count*, is determined when performing a dispatch using either the `glDispatchCompute` or `glDispatchComputeIndirect` command. `glDispatchCompute` takes as input three unsigned integers describing the number of work groups in all three dimensions whereas the indirect variant takes as argument the handle of a buffer containing this three numbers. In contrast to the work group count which can be altered for every dispatch, the number of threads per work group, the *local size* or *work group size*, is statically defined in the shader and can not be dynamically changed between different dispatches. During execution every thread has excess to four different IDs that depend on its position in the compute space. The `WorkGroupID` is a three component ID describing the position of the work group the considered thread resides in and is consequently the same for every thread in a specific work group. In contrast, the `gl_LocalInvocationID` is a 3d ID that contains the position of the considered thread inside its work group. From this two IDs the `gl_GlobalInvocationID` and the `gl_LocalInvocationIndex` are derived. The `gl_GlobalInvocationID` is the most commonly used ID and contains the unique 3d position of a thread in the complete dispatch. `gl_LocalInvocationIndex` is a single component ID resulting from a dimension-wise linearization of `gl_LocalInvocationID` and can therefore only uniquely identify a thread inside its work group.

The hierarchical grouping scheme is motivated by the way the GPU processes the threads. The individual work groups are distributed over the stream processors of the GPU where they are processed in parallel and independently of each other [Cud15]. It is therefore beneficial to dispatch at least as many work groups as stream processors are available to prevent stream processors from being idle. In turn, every stream processor processes the threads inside a work group in groups of 32 threads per *warp* (Nvidia) [Cud15], 64 threads per *wavefront* (AMD) [Ocl13], or a different number depending on the hardware vendor. As a consequence, it is advisable to keep the work group size a multiple of the warp or wavefront size. This ensures that no warp or wavefront has to be padded with threads that perform no productive work. Another aspect that needs to be considered with regard to the lock-step execution of threads in warps/wavefronts is that inside a warp/wavefront diverging threads (threads that take different branches after evaluating a flow control statement) force the whole warp/wavefront to process both branches. A possible strategy to prevent this scenario would be to make the branch condition only dependent on the `gl_WorkGoupID`. Alternatively, the branch dependent computations can be kept cheap to minimize the amount of resulting overhead due to diverging threads inside a warp/wavefront.

3.2. Resources

OpenGL offers a variety of different resources types to provide input data to the various shader stages. Common shader resources are *uniform variables* [Wik15h], *Uniform Buffer Objects* (UBO) [Wik15g] and *textures* [Wik15f]. Uniform variables are a convenient way to supply shader invocations with input values that do not change over the course of a draw call or dispatch. Using UBOs multiple uniform variables can be grouped together in a single buffer. Batching multiple variables with similar update frequency together can improve performance with regard to the data transfer to the GPU since overhead induced by every issued transfer is payed only once for the whole buffer instead of for every individual variable. Textures traditionally contain image data. To improve the performance and quality of sampling operations performed on this image data, textures use 2d caching and provide dedicated filtering schemes like trilinear interpolation. These features are distinct to textures and are not

available to other resources. The resource types mentioned so far are restricted to read-only access and therefore provide no output capabilities to the shader stages. However, since no implicit output stage is associated with the computer shader like it is the case for the fragment shader, resource types that provide explicit write access are required to output results from individual compute shader invocations. In the following three resource types that offer read as well as write capabilities are described.

Shader Storage Buffer Objects (SSBO) [Wik15e] provide generic storage capabilities similar to UBOs. However, there are a couple of distinct differences that become apparent when compared to UBOs. First, SSBOs can be written while UBOs are read exclusive. It is important to consider that this write accesses are incoherent, meaning that issuing a write access alone does in no way guarantee that the issued value is visible from any reading thread at any given time without performing explicit synchronization to ensure this visibility. The required synchronization measures are outlined in the subsequent section. A second difference is that the OpenGL specification ensures the maximal size of a SSBO to be 16 MB at a minimum while for UBOs only 16 KB are guaranteed. On the down side, memory accesses to SSBOs are generally slower than to UBOs since UBOs reside in constant memory which is optimized for read only accesses. In contrast to UBOs, it is not necessary to define the actual size of an SSBO in the shader. It is however still possible to query the size of a SSBO using the GLSL intrinsic function `length()`. To avoid race conditions when trying to perform operations which require the original value in the buffer to be read, GLSL provides a number of atomic operations. However, the application of atomic operations is limited to signed and unsigned integers inside the SSBO.

In situations where the write accesses are inherently 2d in their nature, like when outputting image data from compute shader threads, it can be beneficial to use *images* [Wik15c] instead of SSBOs. Images are similar to textures in the way they store data and offer an optimized 2d caching scheme. However, they do not support mipmapping or any other kind of dedicated filtering. As for SSBOs, atomic operations can only be performed on integer formats while write accesses in general are incoherent and might require manual synchronization.

The third possibility to output from the compute shader is through *atomic counters* [Wik15a]. While atomic counters can also be realized using atomic operations on SSBOs or images, dedicated atomic counter are implicitly coherent and require no synchronization. On the down side, they can only be incremented or decrement by 1 at a time. Additionally, atomic counter are restricted to unsigned integers.

3.3. Synchronization

As already stated, incoherent memory accesses require to be explicitly synchronized to ensure visibility [Wik15d]. A value becomes visible as soon as its memory write is completed and it can be safely accessed. There are two distinct types of visibility: *external visibility* and *internal visibility*. External visibility refers to visibility between individual OpenGL commands. Since no ordering guarantees exist with regard to OpenGL commands, values written to an image by a compute dispatch are, for example, not necessarily visible to a subsequent draw call that writes the image content to a frame buffer object. For this purpose a `glMemoryBarrier` command with the appropriate flag can be placed between the dispatch and the draw call. It is important to note that the flags describe how the values are intended to be read, not how they were written. Internal visibility on the other side refers to visibility between threads of a single compute dispatch (or draw call). There are three important ingredients to the concept of internal visibility. The first one is the `coherent` qualifier which can be used for image and buffer variables. Using `coherent` ensures that completed writes, not writes that are only issued but writes that have actually written their value to physical memory, are visible to other threads. Internally this is realized by turning of cache usage for that particular variable which consequently implies that all threads are guaranteed to access the same memory when trying to read or write to the variable. To ensure that all writes a thread has issued are completed at a given point in the program flow, memory barriers have to be used. Memory barriers do not provide synchronization between individual threads. For this purpose the execution barrier `barrier();` exist, the third ingredient to realize internal visibility. `barrier`, however, can only be used to synchronize threads that share the same work group, e.g., threads that are guaranteed to run on the same stream processor. Figure 2 demonstrates the collective application of all three concepts to realize internal visibility for the content of a SSBO.

```
layout(std430, binding = 5)
coherent buffer blob// coherent memory access
{ int ssb[]; };// SSBO of undefined size
// ...
// perform some (complex) operation on SSBO:
ssb[gl_GlobalInvocationID.x] *= -1;
memoryBarrierBuffer();// complete write
barrier();// sync with threads of work group
// read from SSBO, the read value is visible:
int value = ssb[gl_GlobalInvocationID.x + 1];
// make use of value in the following
// ...
```

Figure 2: *Internal visibility for a SSBO is realized through the usage of the* `coherent` *qualifier, a memory barrier, and an execution barrier.*

3.4. Shared Variables

Shared variables is a feature that is exclusive to the compute shader and is not available to any other shader stage. They

provide a way to efficiently share intermediate results between threads of the same work group. Shared variables reside in fast on-chip memory of the individual stream processors, the so called *shared memory* [Wik15b] [Cud15]. This is the reason why only the compute shader has access to shared variables since no other shader stage features the concept of explicit work groups. Shared variables are implicitly coherent and require the shared keyword in their declaration. A common usage pattern for shared variables is a manually handled cache. For this, every thread in a work group loads data from slow global memory into fast shared memory. After issuing a memory and execution barrier all threads can then access the data of their neighbor threads through shared variables. A possible application where this concept can significantly improve performance are convolutions. In this case auxiliary threads are necessary to ensure that all required data is loaded into shared memory before each thread accesses spatially adjacent values.

3.5. Water Simulation

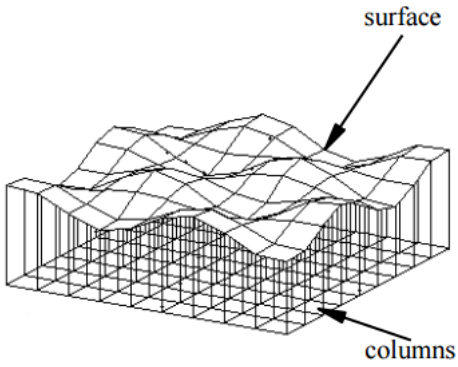


Figure 3: *Depiction of the discretization of the water body in a regular grid of quadratic columns of water. [OH95]*

In this section the application of the compute shader stage is conceptually illustrated using the example of a simple height field based water simulation as described in *Dynamic Simulation of Splashing Fluids* [OH95]. While volumetric simulations offer a greater level of realism, their application is limited to smaller bodies of water due to high memory consumption for the state variables and heavy performance requirements per update step. The underlying model of the height field based simulation described in the following is rather simplistic. The body of water is discretized in a regular grid of columns of water (see figure 3). The exchange of water between adjacent columns is realized through horizontal pipes that connect each column at the bottom of the grid with its eight neighbors. The water height is considered to be constant over the surface of each column. Differences in heights in adjacent columns result in pressure in the connecting pipes. This pressure either increases or decreases the flow through the pipes which in turn results in an increase or decrease in the water heights in the columns. In other words, the simulation is advanced using a semi-implicit Euler integration step. While the pressure that results in a change of flow is dependent on various factors like gravity and pipe diameter, in practice a single empirically determined constant is sufficient as scale factor for the integration of the flow value based on height deltas since the model does not implicitly ensure the height field to behave similar to a continuous body of water anyway. Depending on how the scale factor is chosen the fluid behaves more or less viscous.

For the compute shader based implementation the column grid is subdivided into quadratic tiles. Each tile is processed by a single work group while each column in a tile is processed by a single thread. Since shared memory is used as cache for height values and flows, a seam of perimeter threads needs to be added around each tile to ensure that the threads along the original border of the tile can access the state variables of all their neighbors through shared memory [ZPH11]. The state of the simulation as a whole is stored in two images. The first image holds all the height values while the second one stores the flows to four adjacent columns at each grid point. Only four flows are required per column since the remaining four values can be easily determined from the neighbor flows in the respective directions by flipping their signs. To prevent read after write problems, two pairs of state images are used in a ping-pong buffer fashion where input and output are swapped after every update step, e.g., after every compute dispatch. The update procedure as depicted in figure 4 is described in the following.

First every thread loads its height value and flows into shared memory and synchronizes with the other threads in the work group. Next, height deltas are calculated using the heights stored in shared memory. The deltas are in turn used to update the four flows stored at each grid point. Since the height update requires flows of adjacent grid point to be read, another synchronization is necessary to ensure that all updated flows reside in shared memory. After the height update has been performed, the new heights and flows are written to the image pair that currently serves as output. Interaction with the surface of the fluid can be modeled by explicitly applying additional pressure to pipes in an area of interest. Figure 5 shows a non-photo-realistic rendering of a fluid surface simulated using the described technique.

3.6. Conclusion

The compute shader displays a practical addition to the traditional rendering pipeline by exposing computational capabilities of the GPU on a lower level. This allows for a more direct approach when implementing GPGPU tasks in the context of graphics applications without the need to use a GPGPU framework in addition to the graphics API. Furthermore, the access to shared memory provides a powerful tool for optimization that is not available to other shader stages.

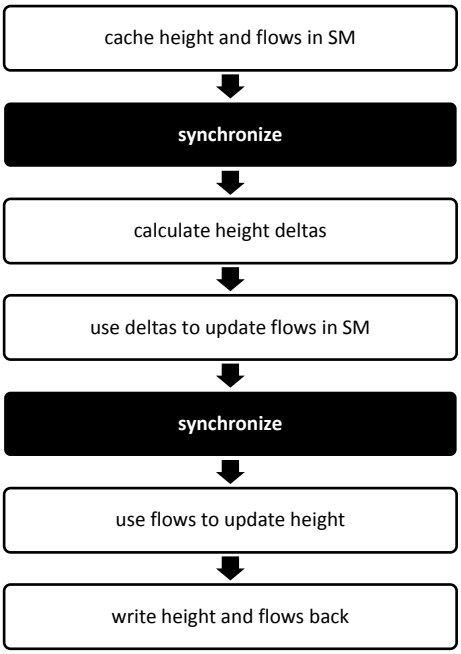


Figure 4: *High-level overview of the update procedure of the fluid simulation. 'SM' abbreviates 'shared memory'.*

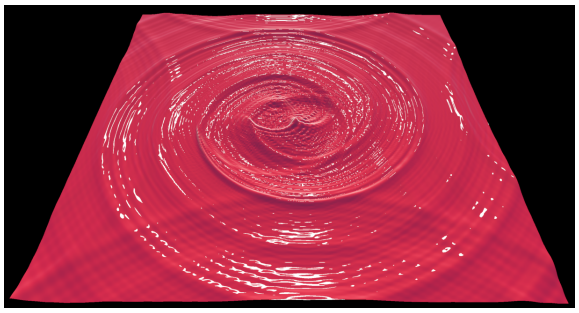


Figure 5: *Non-photo-realistic rendering of a fluid surface simulated with the height field based approach. A force footprint orbiting the center of the grid stimulates the simulation resulting in spiral-shaped wave fronts.*

On the downside, it should be noted that compute shaders are only supported by the more up-to-date graphics APIs and therefore reduce compatibility of the application with older graphics hardware when being used. However, it can be expected that this disadvantage will lose its significance over time and that the compute shader will be an integral feature of future graphics APIs.

References

[ACC] Openacc. http://www.openacc-standard.org/. 1

[Com] Compute space illustration. http://doi.ieeecomputersociety.org/cms/Computer.org/dl/mags/cg/2013/03/figures/mcg20130300054.gif. 2

[Cud15] Cuda programming guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2015. 1, 2, 4

[D3D] Direct3d. https://msdn.microsoft.com/en-us/library/windows/desktop/hh309466(v=vs.85).aspx. 1

[GPG] Gpgpu.org. http://gpgpu.org/about. 1

[OCL] Opencl. https://www.khronos.org/opencl/. 1

[Ocl13] Opencl programming guide. http://developer.amd.com/wordpress/media/2013/07/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide-rev-2.7.pdf, 2013. 2

[OGL] Opengl. https://www.opengl.org/. 1

[OH95] O'BRIEN J. F., HODGINS J. K.: Dynamic simulation of splashing fluids. In *Proceedings of the Computer Animation* (Washington, DC, USA, 1995), CA '95, IEEE Computer Society, pp. 198–. URL: http://dl.acm.org/citation.cfm?id=791214.791474. 4

[Wik15a] Atomic counter (opengl wiki). https://www.opengl.org/wiki/Atomic_Counter, 2015. 3

[Wik15b] Compute shader (opengl wiki). https://www.opengl.org/wiki/Compute_Shader, 2015. 2, 4

[Wik15c] Image load store (opengl wiki). https://www.opengl.org/wiki/Image_Load_Store, 2015. 3

[Wik15d] Memory model (opengl wiki). https://www.opengl.org/wiki/Memory_Model, 2015. 3

[Wik15e] Shader storage buffer object (opengl wiki). https://www.opengl.org/wiki/Shader_Storage_Buffer_Object, 2015. 3

[Wik15f] Texture (opengl wiki). https://www.opengl.org/wiki/Texture, 2015. 2

[Wik15g] Uniform buffer object (opengl wiki). https://www.opengl.org/wiki/Uniform_Buffer_Object, 2015. 2

[Wik15h] Uniform glsl (opengl wiki). https://www.opengl.org/wiki/Uniform_(GLSL), 2015. 2

[ZPH11] ZINK J., PETTINEO M., HOXLEY J.: *Practical rendering and computation with Direct3D 11*. CRC Press, Boca Raton, 2011. An A.K. Peters book. URL: http://opac.inria.fr/record=b1134711. 4

Parallel Sorting Algorithms on GPUs

L. Mais, TU Dresden, Germany

Abstract

This paper gives an overview about various sorting algorithms which are suitable for parallel implementation on the GPU. Especially, it focuses on the well-studied radix sort and its implementation as well as optimization techniques. The radix sort is based on the fundamental prefix sum operation that is also presented within this paper. Although sorting algorithms are widely implemented for GPGPU applications, the paper shows an example application using DirectX 11 compute shader. Finally, the paper outlines some optimization techniques, which can be used to speed up sorting algorithms on the GPU.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Algorithms, Parallel sorting—Radix sort

1. Introduction

Sorting algorithms are well-studied and widely used in all areas of computer science. In computer graphics, sorting routines are essential in applications such as shadow and transparency modeling, ray tracing as well as particle rendering and animation [MG11]. For instance, sorting helps to construct efficient acceleration structures like kd-trees and bounding volume hierarchies for ray tracing and collision detection. Besides, sorting is applied in General Purpose Computing on Graphics Processing Unit (GPGPU) applications within parallel hashing, database acceleration and data mining [MG11].

While implementing sorting algorithms, it is the objective to make full use of today's GPUs computational resources like SIMD architecture and high bandwidth. However, the highly parallel stream model also enforces us to rethink how to parallelize efficiently existing sequential solution.

2. Related Work

Harris et al. [HSO07] describe the functionality of the parallel prefix sum (Scan) and present an efficient implementation on GPU using CUDA. Besides, this work outlines the importance of the scan function as primitve for many parallel algorithms not just limited to sorting.

The radix sort implementation of Satish et al. [SHG09] was an influential and efficient solution that became part of the CUDA Data Parallel Primitive (CUDPP) library.

Later published research papers such as Ha et al. [HKS10] and Merrill et al. [MG11] refer to the work of Satish et al. as basis concept, but introduce further optimization strategies, both using CUDA. For instance, Ha et al. present a solution applying implicit parallel counting and mixed-data structure.

Moreover, a practical approach to radix sort is given by Garcia et al. [GOUR10], showing the use of sorting in the context of ray tracing using DirectX 11 compute shader.

Polok et al. [PIS15] introduce a radix sort implementation with OpenCL for a GPGPU application processing sparse matrix algebra.

Besides the focus on the radix sort algorithm, Cederman et al. [CT09] present an efficient parallel quicksort implementation using CUDA.

3. Basics of Sorting Approaches

Sorting algorithms can be divided into data-driven and data-independent approaches [KW04]. Data-driven algorithms describe iterative algorithms where the processing depends on the value of the currently considered element. Popular representatives are quicksort and heapsort. The quicksort is one of the fastest sorting algorithms and has an average complexity of $O(n \cdot \log(n))$.

Contrary, data-independent algorithms are characterized by always executing the same processing path. Consequently, the computational cost and needed communication is known in advance, which makes it easier to parallelize.

Furthermore, this group can be categorized into comparison-based and counting sorters as outlined in Ha et al. [HKS10]. Comparison-based sorters rely on sorting

networks, which sort a sequence of input values with a fixed number of steps. Within this context each step is called a pass and each pass applies a certain number of comparators in parallel. Figure 1 shows the bitonic sorting network as an example of the comparison-based sorter.

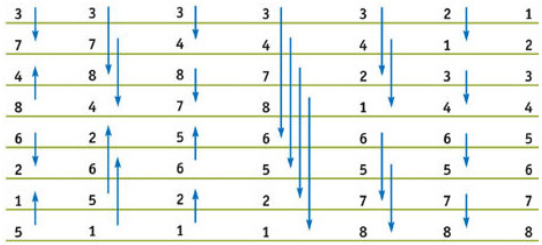


Figure 1: *Bitonic Merge Sort over eight elements [BP04]*

The idea of the bitonic sort is that a sequence is ordered by building ascending and descending subsequences of keys and merging them. The algorithm starts with $n/2$-sequences and continues by doubling the size of the sequences. Although the bitonic sort is popular, it is quite costly with a complexity of $O(n \cdot \log^2 n)$. Therefore, bitonic sort should be considered in the case of either dealing with non-integer keys or keys of variable length. Otherwise, the following described counting sorters offer a more efficient and scalable approach.

The main advantage of counting sorters is that they execute in linear complexity, which results in high performance improvements comparing with comparison-based sorters. Radix sorting is the fastest algorithm in this group and is explained in detail in the following section.

4. Radix Sort

Radix sort is a fast and stable sorting algorithm, which can be used for keys that map to integral values such as integers, strings or floating points. If the n keys consist of d digits, the complexity results in $O(n \cdot d)$.

The algorithm works by grouping the keys according to the value of a certain digit. Once the keys are grouped, they are collected again in ascending order of the groups. These two steps are repeated d times, going through the keys from the least significant to the most significant digit.

Moreover, the choice of the radix r plays an important role, because it determines the number of groups, in the following named as buckets, in which the keys get sorted. For efficient implementation, it is suitable to set $r = 2$ or a power of two $r = 2^b$. Figure 2 illustrates the functionality of the radix sort algorithm for three digit keys and a radix $r = 10$.

4.1. All-Prefix Sum (Scan) Primitive

In each iteration of the radix sort a bucket sort is performed. For a more efficient implementation the bucket sort can be

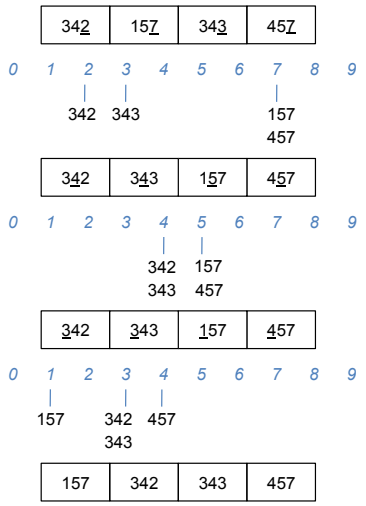


Figure 2: *Radix Sort for three-digit keys*

reduced to a parallel prefix sum operation, also called scan. The scan operation is an important data-parallel primitive, which can be applied in many cases such as stream compaction, summed-area tables and sorting as shown in [HSO07].

The scan takes a binary associative operator $\oplus$ and an identity I. It is defined as following function:

$$[a_0, a_1, ..., a_{n-1}] \rightarrow [I, a_0, (a_0 \oplus a_1, ..., (a_0 \oplus a_1 \oplus ... \oplus a_{n-2})]$$

For example, applying addition as $\oplus$ and setting the identity to zero, the ordered input array $[3,5,0,1,7]$ returns the ordered array $[0,3,8,8,9]$. More precisely, the given definition describes the exclusive scan, because it does not include the last element within the prefix sum. Contrary, there is the inclusive scan without identity, so it includes the last element in the prefix sum.

In order to implement the scan in parallel efficiently, Harris et al. [HSO07] describe a solution using a balanced binary tree. An input array of n leaves results in a tree with $\log_2 n$ levels and $O(n)$ adds per tree traversal. All operations can be executed in-place in shared memory.

The implementation takes place in two phases: the reduction phase and down-sweep phase. In the reduction phase partial sums are calculated traversing the tree from leaves to root as shown in Figure 3.

The down-sweep phase starts with inserting zero at the root and then copying the current value to the left child as well as copying the sum of the current value and the neighbour to the right child, illustrated in Figure 4.

Implementing radix sort, scan is applied multiple times in order to compute the new index of the elements in each iteration as shown in the following subsection.

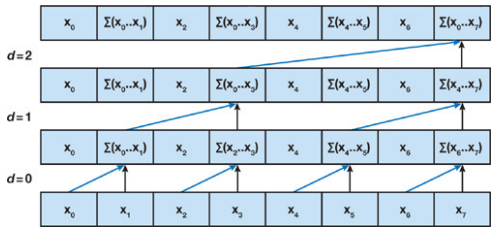


Figure 3: *Reduction phase of scan [HSO07]*

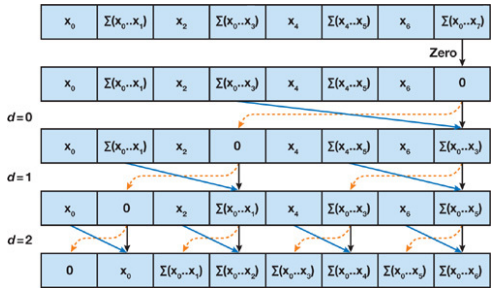


Figure 4: *Down-sweep phase of scan [HSO07]*

4.2. Basic approach for radix sort implementation

In the following, the concept of Satis et al. [SHG09] is described. In the time of publication it was the fastest known implementation of the radix sort algorithm and became part of the CUDPP library. Besides, the ideas introduced in this work are referred in many further implementations.

The input sequence of keys is divided into p blocks that can be processed in parallel. Each block sorts its data in shared memory with respect to a chosen radix 2^b.

The simplest way to sort the keys using scan is to consider only one bit at a time, which is known as split operation. In order to decrease the number of scatters, it is not efficient to set $b = 1$ like the split operation would suggest. Instead each block creates a histogram that computes the occurences of the 2^b possible digits. For instance, sorting 32-bit integer numbers and setting the radix $r = 2^4$, the algorithm sorts 4-bit subsequences per iteration and requires 16 counting buckets. Each iteration of the radix sort is processed in four steps:

1) Each block sorts the subsequence of the keys in shared memory by using the split primitiv with respect to the i-th bit.
2) Each block writes its 2^b-entry digit histogram and its sorted data back to global memory.
3) Then a scan over the global histogram tables is performed in order to compute the global digit offset.
4) Using the scan results, each block copies its elements to their correct output position in global memory.

The split primitive, applied in the first step, places all keys with zero at the i-th bit before all keys with one while keeping the existing order of keys with equivalent value. Figure 5 illustrates the functionality of the split primitive.

Figure 5: *Scan applied in radix sort to compute the new indices as follows: if bit b is zero then current index minus number of ones with lower index; if bit b is one then total number of zeros plus scan at current index [GOUR10]*

The global $p \times 2^b$ histogram table is stored in column-major order as illustrated in Figure 6. Then a scan is applied to get the offsets for each block and digit. These results are used to determine the final indices of the locally sorted elements. If there are some keys with the same digit in one block, the corresponding local offset is added to the global one. Afterwards, all elements are written to global memory to the computed global index.

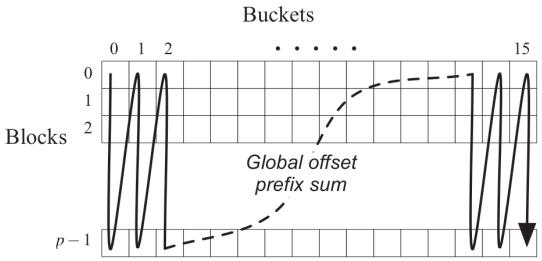


Figure 6: *Global histogram table for buckets [SHG09]*

4.3. Example implementation using Compute Shader

Radix sort is often implemented using CUDA. In the following section, an example implementation is described using DirectX 11 compute shader, which was implemented by Garcia et al. [GOUR10]. Instead of four steps per iteration according to Satish et al. [SHG09], this implementation uses three steps and avoids writing data back to global memory in the second step.

The implementation sorts 32-bit integer numbers with a radix $r = 2^4$ in 8 cycles. The input array is divided into

blocks with 512 threads. Each block again is divided into warps with k threads. Each thread processes one element of the input array at a time, particularly a 4-bit subsequences of the key.

In the local sort step each block extracts one bit and passes it to the scanBlock function. This function executes several scans and works as follows:

1) It scans the elements in groups of k threads.
2) Then, it adds the last group element to the scan result of each group and builds a new array, which is scanned again.
3) Each thread uses its thread id and the result from the last scan to determine the indices for all elements within the block.

After four split and scatter operations the 4-bit keys are sorted completely in shared memory. Then, the offset function computes the bucket offsets for each block. It counts the number of keys per bucket per block and writes it into a global bucket matrix.
Finally, a global scatter dispatch computes the global index of the elements. Therefor, each already locally sorted element get its index by adding the local offset to the global bucket offset.

Figure 7 shows a performance test of the radix sort implementation of Garcia et al. It illustrates that for under 2 million keys the compute shader can achieve a competetive performance to the CUDA implementation. However, for input sequences above 2 million the CUDA implementation is 1.7× to 2× faster. As one reason for that can be mentioned that CUDA is highly optimized for NVIDIA GPUs. Contrary, the solution using DirectX 11 compute shader is not limited to certain hardware, it can run on any compute shader compliant computing platform. Besides, the radix sort implementation using compute shader has not been focused by research in the same extent as CUDA implementations.

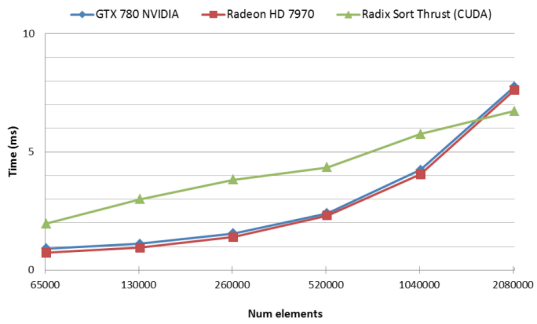


Figure 7: *DirectX 11 Compute Shader Radix Sort Performance in AMD Radeon HD 7970 and NVIDIA Geforce GTX780 GPUs compared with Thrust CUDA radix sort running in the NVIDIA Geforce GTX780 GPU. [GOUR10]*

However, this implementation has shown its advantages applied in ray tracing. Ray tracing is a technique to simulate the effect of light by tracing it through pixels in an image plane. Garcia et al. point out that memory handling can be seen as constant bottleneck. In order to realize fast memory access and fast ray-geometry intersection discovery, efficient data structures become crucial to real-time ray tracing applications.

Within the scope of the work of Garcia et al. radix sort is applied to build stackless linear bounding volume hierarchies (SLBVH). Therefor, the entire tree hierarchy is constructed for each frame. Due to the shown advantages, using efficient parallel sorting on GPUs seems to be a key factor for real-time ray tracing with dynamic scenes.

4.4. Optimization techniques

The following section presents some optimization techniques introduced in several research papers dealing with parallel sorting on GPUs. These techniques are not limited to sorting, instead they can be used for various parallel algorithms.

1) Avoid branching and loop unrolling

One option to increase performance is to reduce control instructions and condition checking. Therefore, avoiding branching and loop unrolling are possible techniques. For instance, loop unrolling can be realized by copying the inner instructions of the loop.

2) Optimize memory access pattern

Another important strategy is to use shared memory across threads as often as possible in order to make use of the high access time within the block and to hide latencies to global memory. However, avoiding bank conflicts needs to be considered.

A bank conflict occurs when multiple threads in the same warp access the same bank. As a result, the instructions are executed serialized. If n threads are accessing the same bank at the same time, there is a degree-n bank conflict which requires n times many cycles.

Concerning the scan implementation by Harris et al. [HSO07], multiple bank conflicts occured in the naive approach traversing the tree. This problem could be handled by using addressing with a variable padding. The applied offset has been increased whenever the index of the shared memory array exceeded the number of memory banks.

3) Kernel fusion

According to Merrill et al. [MG11] kernel fusion is a technique for reducing aggregate memory workload. This includes that intermediate results can be stored in register or shared memory to avoid data transfer to global memory. This approach has also been applied in the scanBlock function of Garcia's et al. implementation, described in 4.3.

4) Optimize hardware occupancy

Optimizing hardware occupancy means the efficient use of the given hardware of recent GPUs. A GPU can run 1000 to 10000 threads concurrently and parallel code is executed through warps.

As an example, the radix sort implementation of Polok et al. [PIS15] can be considered. Polok et al. could improve performance implementing variable length of segments. The length of the segment is the number of keys get processed by one block. Contrary, the implementation of Satish et al. always processes 1024 items per block. However, Polok et al. determine the length of segments as minimum to keep the underlying hardware fully utilized.

5. Evaluation

Revisiting parallel sorting algorithms on GPU has shown that radix sort is the fastest and probably best-studied sorting algorithm for GPU implementations. Although, there are other parallel implementations such as bitonic sort or quicksort, radix sort should be preferred if the keys can be mapped to integer values.

Moreover, radix sort has been efficiently implemented using APIs like DirectX and OpenGL as well as GPGPU APIs including CUDA and OpenCL. When implementing sorting within a computer graphic application, DirectX and OpenGL should be the better choice. In this case, OpenGL has also the advantage to be a multi-platform and cross-language API.

On the other hand, there are several libraries that implement radix sort for GPGPU applications such as CUDPP, Thrust or CLOGS. These solutions seem to be more performant, especially processing bigger input arrays.

6. Conclusion

This paper has outlined the development of parallel sorting algorithms on the GPU. Especially, the strategy and advantages of the radix sort algorithm have been pointed out. While describing the concept of implementing radix sort, the importance of the scan operations has been shown. Scan can be implemented efficiently in parallel and can be used for many other applications.

Furthermore, introducing some optimization techniques, it was shown that it is crucial to understand the underlaying hardware and memory access pattern in order to fully utilize the given hardware and hide memory latencies.

According to more recent research papers dealing with radix sort, it is obvious that there can be still achieved further performance improvements. Moreover, it is important to choose the suitable API and adjust existing solutions to the given use case and the further developing hardware.

References

[BP04] BUCK I., PURCELL T.: A toolkit for computation on gpus. *GPU Gems* (2004). URL: http://http.developer.nvidia.com/GPUGems/gpugems_ch37.html. 2

[CT09] CEDERMAN D., TSIGAS P.: Gpu-quicksort: A practical quicksort algorithm for graphics processors. *Journal of Experimental Algorithmics (JEA) 14*, 4 (2009). doi:http://dx.doi.org/10.1145/1498698.1564500. 1

[GOUR10] GARCIA A., O.ALVIZO, U.OLIVARES, RAMOS F.: Fast data parallel radix sort implementation in directx 11 compute shader to accelerate ray tracing algorithms. *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing* (2010). 1, 3, 4

[HKS10] HA L., KRUEGER J., SILVA C.: Implicit radix sorting on gpus. *GPU Gems 2* (2010). URL: http://www.gpucomputing.net/sites/default/files/papers/2622/ImplSorting.pdf. 1

[HSO07] HARRIS M., SENGUPTA S., OWENS J.: Parallel prefix sum (scan) with cuda. *GPU Gems 3*, 19 (2007). URL: http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html. 1, 2, 3, 4

[KW04] KIPFER P., WESTERMANN R.: Improved gpu sorting. *GPU Gems 2*, 46 (2004). URL: http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter46.html. 1

[MG11] MERRILL D., GRIMSHAW A.: High performance and scalable radix sorting: a case studiy of implementing dynamic parallelism for gpu computing. *Parallel Processing Letters 21*, 2 (2011), 245–272. 1, 4

[PIS15] POLOK L., ILA V., SMRZ P.: Fast radix sort for sparse linear algebra on gpu. *Proceedings of the High Performance Computing Symposium HPC 14*, 11 (2015). 1, 5

[SHG09] SATISH N., HARRIS M., GARLAND M.: Designing efficient sorting algorithms for manycore gpus. *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing* (2009). 1, 3

State of the art regarding order-independent transparency with modern OpenGL

Max Reitz[1]

[1]TU Dresden, Faculty of Computer Science: Hauptseminar Computergrafik und Visualisierung

Abstract

Transparency has a multitude of applications in computer graphics. The traditional method of alpha blending requires sorting the objects from back to front, which is neither trivial nor necessarily sufficient. This paper will discuss different approaches to achieve order-independent transparency using modern OpenGL, mainly McGuire's and Bavoil's order-independent blending, an unbounded A-buffer implementation, and Hybrid Transparency, by presenting them, comparing their strengths and weaknesses, and thus contemplating sensible applications.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation—Bitmap and framebuffer operations

1. Introduction

The usual approach for drawing transparent objects employed in real-time applications is the alpha blending algorithm (section 3), which is reasonably fast and accurate, if performed correctly. However, it requires all fragments to be drawn from back to front which is not trivial to achieve; surrogates like sorting whole objects instead of fragments are only sufficient for a limited range of applications.

Modern OpenGL allows a multitude of different approaches such as using truly order-independent blending [MB13] (section 4.1), automatically sorting all fragments drawn from back to front [Kub14] (section 5), or combining both methods to achieve near-accurate results while limiting memory usage to a fixed maximum [MCTB13] (section 6.2).

There is no single approach suitable for every situation, so in section 7, after having presented multiple designs with their respective strengths and drawbacks, we will finally suggest the most appropriate use case for each of them.

First, however, we will study transparency in general so the different methods can be reasonably explained.

2. Transparency in general

For the purpose of this paper, a transparent object consists of one or more transparent fragments. Each of those fragments is defined by having a *color* $U \in [0,1]^3$, a *depth* z and an *opacity* $\alpha \in [0,1]$. Conversely, the *transparency* is defined to be $1-\alpha$. Note that non-transparent fragments can be represented by choosing $\alpha = 1$.

The color of all fragments visible behind a transparent fragment (that is, all fragments with a greater depth value) will be multiplied by $1-\alpha$. The color of the fragment itself is multiplied by α when drawn. When composing all fragments for a single pixel, the values thus obtained are added.

In the following, we will consider a single pixel on the screen. The fragments generated for that pixel are to be indexed from back to front, where fragment 0 is the background (e.g. the frontmost non-transparent fragment, occluding everything behind it) and fragment n is the one most closest to the viewer.

The *visibility* of a fragment i is the factor by which its color is multiplied due to the cumulative opacity of all the fragments in front of it:

$$v_i = \prod_{j=i+1}^{n} (1-\alpha_j) \tag{1}$$

Since two fragments at the same depth value would be expected to have the same visibility, we can also represent this as a function of the depth value, where we define

$$v(z) = \begin{cases} v_i & i = \arg\max_{0 \le j \le n} z_j \le z \\ 1 & \text{if no such } i \text{ exists} \end{cases} \tag{2}$$

The color a fragment i adds to the pixel, assuming it is not occluded by other fragments (i.e., $v_i = 1$), is $C_i = \alpha_i U_i$. This is also called the *pre-multiplied* color. In the general case (i.e., v_i is arbitrary), the color added will be $v_i C_i = v_i \alpha_i U_i$.

We can now infer the general term for calculating the color of a pixel with transparent fragments:

$$\begin{aligned} C_P &= \sum_{i=0}^{n} v_i C_i \\ &= \sum_{i=0}^{n} \left(\prod_{j=i+1}^{n} (1-\alpha_j) \right) C_i \end{aligned} \tag{3}$$

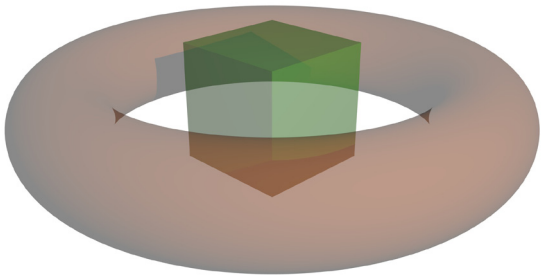


Figure 1: *Cube enclosed by a torus; neither object is strictly in front of the other*

3. Alpha blending

We can see that using term (3) for rendering transparent fragments leads to quadratic complexity $O(n^2)$, which is not desirable. Alpha blending is a way to solve this issue.

Expanding term (3) yields:

$$\begin{aligned} C_P &= 1 \cdot C_n + (1-\alpha_n)C_{n-1} + \dots \\ &+ (1-\alpha_n)\dots(1-\alpha_2)C_1 \\ &+ (1-\alpha_n)\dots(1-\alpha_2)(1-\alpha_1)C_0 \\ &= C_n + (1-\alpha_n)(C_{n-1} + (1-\alpha_{n-1})(\dots \\ &\quad + (1-\alpha_2)(C_1 + (1-\alpha_1)C_0))) \end{aligned}$$

Therefore, C_P can be accumulated by drawing from back to front. Let C_A be the color accumulated so far:

0. $C_A := C_0$
1. $C_A := C_1 + (1-\alpha_1)C_A$
2. $C_A := C_2 + (1-\alpha_2)C_A$

   $\dots$

n. $C_A := C_n + (1-\alpha_n)C_A$

Now, $C_A = C_P$. This process is referred to as alpha blending. As can be seen, it requires only $O(n)$ steps to derive C_P.

It can be implemented in OpenGL using *blending*, a process where an existing color C_d in the frame buffer is combined with the color of a newly generated fragment C_s using a *blending term* to generate the new color C_d to save in the frame buffer. In order to perform alpha blending, all transparent fragments are drawn back to front using the following blending term:

$$C_d := C_s + (1-\alpha_s)C_d \tag{4}$$

We can see that using (4) for blending will result in the desired color accumulation.

3.1. Fragment ordering

A traditional way of ensuring this ordering is by drawing the non-transparent geometry first, and then drawing all transparent objects ordered from back to front. However, e.g. if one object encloses another, one cannot define a strict order where one object's fragments are all in front of all the other object's fragments (see fig. 1).

A way to solve the issue is by ordering triangles instead of objects; but this can fail, too, since two triangles may intersect.

Therefore, both ordering objects and ordering triangles are ways to ensure this ordering, but not in the general case. Generally, it is required to sort the fragments themselves to ensure correct results.

4. Order-independent blending

The reason alpha blending requires all fragments to be sorted from back to front is not only because it has been defined that way, but can be seen in the algorithm itself. Specifically, the blending term is neither commutative nor associative.

Let $C_d \circ C_s$ be the blending term. In order to be able to draw fragments in any order, the following must obviously hold true for all C_d, C_{s_1}, and C_{s_2}:

$$(C_d \circ C_{s_1}) \circ C_{s_2} = (C_d \circ C_{s_2}) \circ C_{s_1} \tag{5}$$

At least for blending terms with a neutral element, this is true if and only if the blending term is commutative and associative, which is not the case for (4).

Examples for commutative and associative blending terms are plain addition and multiplication, and both can indeed be used for simulating some forms of transparency.

Additive blending can only add brightness to the scene, but it cannot remove any. Therefore, it is suitable for drawing things like plasma or fire.

Multiplicative blending (with factors in $[0,1]$) can only darken the scene, but it cannot add brightness, making it suitable for drawing e.g. smoke or other dark transparent objects.

When using either additive or multiplicative blending, transparent fragments may be drawn in any order. However, when using both at the same time, this is no longer the case, since mixing addition and multiplication is no longer commutative or associative.

4.1. Weighted blending

[MB13] presents a weighted commutative and associative blending operator which can approximate the effects of true transparency as defined in section 2.

The idea is finding an approximate representation of the visibility function which can be calculated for every fragment basically independently of all the other fragments for the pixel in question. In order to not require that visibility function to be normalized, the approximation is then used for creating a weighted average of the transparent fragments' colors (thus normalizing the function) and blending that average color onto the non-transparent background, using the background visibility.

First, the background visibility can be directly derived from (1) and is

$$\prod_{i=1}^{n}(1-\alpha_i) \tag{6}$$

Multiplication is commutative and associative, thus it is possible to compute this term using order-independent blending.

The weighted average of the colors of all transparent fragments is:

$$\frac{\sum_{i=1}^{n} C_i \cdot w(z_i, \alpha_i)}{\sum_{i=1}^{n} \alpha_i \cdot w(z_i, \alpha_i)} \tag{7}$$

That is, colors are weighted based on their opacity and a generic weighting term $w(z, \alpha)$, as opposed to the real case where colors are basically weighted based on the visibility term (1). Since addition as well is commutative and associative, this term can be computed using order-independent blending, too.

The weighting term is generally chosen so that its value increases with decreasing z, thus effectively simulating the effect of the visibility term. The difference is that the visibility term (1) explicitly takes the fragment order into account, whereas the weighting term does so only implicitly by assigning more weight to fragments which are closer to the viewer. An example term given in [MB13] is

$$w(z, \alpha) \;=\; \alpha \cdot \max(10^{-2}, 3 \times 10^{3} \cdot (1 - d(z))^{3})$$

where $d(z)$ is the value in `gl_FragCoord.z` in the fragment shader. As can be seen, this term does make use of α even though it is already taken into account outside of the weight function itself. This is appearently due to near-transparent fragments near the viewer otherwise receiving too much weight.

Combining (6) and (7) yields

$$C_{MB} = \frac{\sum_{i=1}^{n} C_i \cdot w(z_i, \alpha_i)}{\sum_{i=1}^{n} \alpha_i \cdot w(z_i, \alpha_i)} \left(1 - \prod_{i=1}^{n} (1 - \alpha_i)\right) + C_0 \prod_{i=1}^{n} (1 - \alpha_i) \tag{8}$$

This can be implemented by using a frame buffer with two color buffers. The first color buffer will receive every fragment's weighted color and the weight itself using additive blending, the second color buffer will receive every fragment's transparency using multiplicative blending.

After all the geometry has been drawn in this way, the first color buffer contains $\sum C_i \cdot w(z_i, \alpha_i)$ and $\sum \alpha_i \cdot w(z_i, \alpha_i)$, and the second one contains $\prod(1 - \alpha_i)$. A full-screen pass is then used to combine these values to C_{MB} as described in (8).

4.1.1. Comparison to alpha blending

An obvious advantage of this method is it being order-independent and still achieving linear complexity. Also, it is simple to implement and does not require special OpenGL features, thus being perfectly suited for e.g. mobile platforms. However, it does not solve (3).

The difference to alpha blending (and thus the general transparency term given in (3)) is that the real visibility function (2) is generally not smooth but has a step for every fragment (cf. fig. 3). The weighting term used in McGuire's and Bavoil's method on the other hand is smooth. This results in fragments with similar depth values to have nearly identical weighting using the McGuire-Bavoil method, but using (3), one can easily discern which of the fragments is in front of the other. In fact, with (3) it does not matter at all how far the fragments are apart, while it very much does matter for the McGuire-Bavoil approximation.

This has obvious drawbacks where accuracy is required, but it can also be beneficial. Imagine rendering a flame using one polygon and the smoke it produces using another one, both transparent. Flame and smoke are likely to penetrate each other, which will lead to popping artifacts when using correct alpha blending. Using the McGuire-Bavoil method, moving one polygon in front of the other will result in a smooth transition.

5. A-buffer

OpenGL generally uses the z-buffer algorithm, which means having a color buffer which can hold a single fragment's color and a depth buffer holding that fragment's depth value. When a new fragment is generated, its depth is compared to the value in the depth buffer for the respective pixel. If that depth test is passed (normally: if the new depth value is less than the old one), the new fragment's color and depth replace the old values stored (or are combined using blending).

Another solution to the same problem the z-buffer algorithm tries to solve is the A-buffer algorithm, originally invented for anti-aliasing. Here, the color buffer can store a multitude of fragments, in theory even an unlimited amount. When rendering the geometry, every new fragment is just added to the list of fragments for its pixel. Afterwards, a full-screen pass collapses the fragments collected to a single color value for each pixel. This algorithm can be used to implement efficient fragment sorting.

5.1. General implementation

First, the transparent geometry is rendered using the A-buffer algorithm. The full-screen resolve pass then sorts the fragments for each pixel from back to front and then applies alpha blending to the sorted list.

5.2. Implementation using OpenGL

There are multiple ways of implementing an A-buffer using OpenGL, but perhaps the most efficient one is using a linked list for each pixel [Kub14].

There is a 2D single-channel integer texture having the same size as the framebuffer, and a 1D four-channel integer texture (or texture buffer) which has an arbitrary size. The latter will hold the fragments (the list elements), the former will point to the list head for each pixel.

Pointing to a list element is done by specifying its index (+1, so 0 can be used to denote the end of the list). Each list element represents a fragment and has to point to the next fragment in the list, it will thus contain:

- the fragment's color (including the opacity)
- the fragment's depth
- the pointer to the next list element

The "next element" will always be the previous fragment drawn. The list head thus points to the last fragment drawn for the respective pixel.

Besides the two textures, there is an atomic counter which contains the index of the first free element in the 1D texture.

See fig. 2 for an overview.

Drawing the transparent geometry can be done without

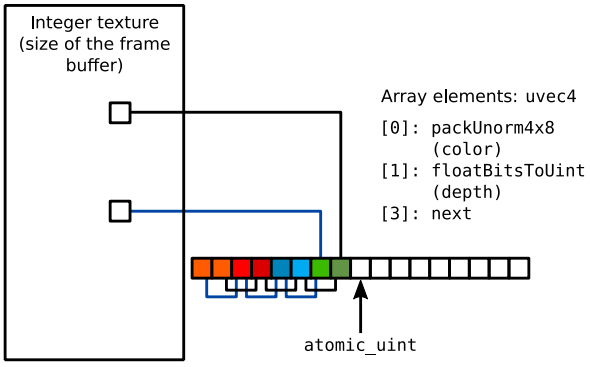


Figure 2: *Linked-list A-buffer implementation using OpenGL*

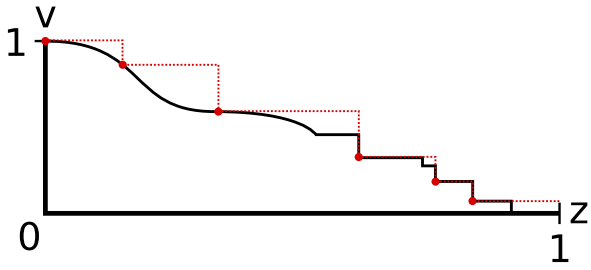


Figure 3: *Example of a visibility function, and it being reconstructed from several samples using an overestimating step function*

having a meaningful framebuffer, the textures defined above are bound as images. Whenever a fragment is generated, it needs to be entered into its respective pixel's fragment list. First, the fragment shader allocates space in the 1D texture for storing the fragment data. This is done by incrementing the atomic counter and using its previous value as the index to store the fragment into (unless that index is beyond the size of the 1D texture, in which case the fragment is discarded).

The pointer to the last fragment drawn for the current pixel is obtained by atomically exchanging the current fragment's pointer with the value stored in the 2D texture.

Then, the current fragment can be entered into the 1D texture, since all the required values (color, depth, *next* pointer) are known.

After all transparent fragments have thus been collected, a full-screen pass is performed on the two textures, gathering all fragments for a single pixel into a local array, sorting it (using e.g. insertion sort) from back to front and then performing the usual alpha blending.

5.3. Comparison

This method is nearly correct as long as the 1D texture is large enough to hold all the fragments. Its size can however be adjusted on the fly to accommodate for that. The only remaining issue is the local array used during the full-screen pass in the fragment shader: Since GLSL does not support variable-length arrays, its size must be fixed at compile time and thus only a limited number of fragments can be gathered and sorted per pixel. The main problem here is that these fragments are just a random selection of k fragments, and not necessarily the foremost (and thus most important) ones.

If the method could be made to work with an arbitrary number of fragments per pixel, it would require an unbounded amount of memory, which may not be acceptable especially for real-time applications.

So there are edge cases where this method may not perform well, but in general it is an accurate and reasonably fast method of rendering transparent objects. It only requires a single geometry pass (as opposed to e.g. depth peeling) and another full-screen pass to resolve the gathered data, basically the same as the McGuire-Bavoil method. However, due to the shaders being more complex, this A-buffer implementation will in practice still be slower than the McGuire-Bavoil method. Also, it requires OpenGL 4.2, or at least the ARB_shader_atomic_counters and the ARB_shader_image_load_store extensions, which may either be a critical issue or not a problem at all.

6. Determining the visibility function

The final group of methods presented here are based on the following idea:

1. Draw all transparent geometry and collect data for representing the visibility function $v(z)$ for each pixel.
2. Draw all transparent geometry and use $v(z)$ to explicitly determine every fragment's visibility.

6.1. Adaptive Transparency (AT)

Adaptive Transparency [SML11] represents the visibility function by a number of samples, each consisting of the visibility at a depth z (see fig. 3). Reconstructing the visibility function from those samples is done like in (2). These samples are stored in a sorted array of fixed size k.

Drawing a fragment results in a new sample being generated. Since the sample storage is limited to k samples, this generally means that an old sample has to be removed. For this, we examine which of the rectangles spanned by two samples ($v(z)$ is a step function) has the smallest area, i.e.

$$i = \arg\min_j \big((v(z_{j-1}) - v(z_j))(z_j - z_{j-1})\big) \qquad (9)$$

Those two samples are then collapsed into a single new one which replaces both. [SML11] proposes doing so by simply removing sample i.

When entering the new sample into the array, care must be taken to update the visibility values of all samples behind that new sample.

This process of gathering the data can be implemented in OpenGL using one (array) texture for storing the visibility values, and another one for storing the respective depth values. The fragment shader then accesses both as images and simply implements the algorithm presented above. Once the samples have been gathered, the transparent geometry is drawn once again, with the fragment shader reconstructing the visibility function from the samples according to (2).

Finally, note that there are other ways of sampling the visibility function than by assuming it to be a step function. For continuous transparent objects like fog or dust, working in fourier space may yield better results.

6.1.1. Performance considerations

Iterating through the sample array, removing samples and inserting new ones requires exclusive access to the part of the textures containing the sample data for a single pixel. This is generally not guaranteed.

In the past two years, Intel has been more or less heavily pushing their OpenGL extension for ordering fragment shader invocations on the same pixel (INTEL_fragment_shader_ordering) [Sal13]. Using this extension trivially solves the the problem, and does not cost much performance (according to Intel, that is). It is available on new Intel GPUs (starting from Haswell) and AMD (at least on the GCN architecture), although a test case on an AMD GCN card did not produce the desired results.

Another way to ensure correct ordering is using spinlocks. This can be done by binding single-channel integer textures as images and then implementing spinlocks using atomic image accesses. There are special considerations to be taken into account when implementing spin locks in a fragment shader, though (multiple shader instances share a single program counter), and it does cost performance.

6.1.2. Quality considerations

Interestingly, the reference implementation of adaptive transparency does not scan the whole sample array for samples to remove, but only the second half of it. According to comments in the source code, this is due to two reasons:

1. Doing so is faster, because it obviously only takes half the time to find a sample to remove.
2. Doing so yields better results.

Apparently removing samples which are close to the viewer is generally not good, even if the algorithm deems them unimportant. Pursuing this realization even further will then lead to Hybrid Transparency and Multi-Layer Alpha Blending.

6.2. Hybrid Transparency (HT)

Hybrid Transparency [MCTB13] builds on Adaptive Transparency and improves both its quality and its performance. In contrast to AT, HT always removes the last sample (the one with the greatest depth value) when a new one is generated (if that new one is in front of that last sample).

Samples are stored in a bounded A-buffer, that is an A-buffer which is capable of storing at most k fragments per pixel. Generally, this is implemented by using an array texture with k layers. Therefore, a sample does not only consist of its visibility and its depth, but also of its color, so that every sample directly represents the fragment it is based on.

The visibility of a fragment (or sample) is not represented directly by its actual visibility during the first pass, but only by its opacity. A full-screen pass is necessary after the first pass to calculate the actual visibilities.

All fragments that are evicted from the A-buffer are blended using the McGuire-Bavoil method in a separate color buffer. This process called *tail blending* is based on the assumption that these fragments do not add much to the final pixel color since they are not among the k nearest fragments, so using a lower-quality blending method is sufficient.

Because all the necessary data for the k nearest fragments has been gathered already, the second geometry pass is unnecessary. Instead, another full-screen pass calculates the pixel color using alpha blending (the fragments are kept sorted from back to front) for the k nearest fragments, and then applies the result onto the commutatively blended tail.

As a side note, [HBT14] presents a method for more efficient tail blending. Here, it is not fragment colors that are stored but shading parameters. The real shading is performed only during and after the first HT pass. For the k nearest fragments, high-quality shading is performed, whereas every tail fragment will be shaded with lower quality.

6.2.1. Performance considerations

See [Kub14] for an efficient implementation using OpenGL (called *Atomic Loop* there). Basically, using atomic operations on images, it is possible to atomically insert new fragments into a list that is already sorted from back to front (and keep it sorted). With OpenGL's standard 32-bit atomic operations, two geometry passes are necessary, whereas with 64-bit atomic operations, one pass suffices (since color and depth together take up 64 bits). Alternatively, as mentioned by [SV14], when using Intel's extension for fragment shader ordering, one pass will be enough, too.

Considering its runtime, this method is thus basically as fast as the unbounded A-buffer variant using linked lists.

However, HT requires much more memory for a medium amount of transparent fragments drawn, since it always takes space for k fragments per pixel, whereas the unbounded A-buffer variant does not. On the other hand, this fixed amount of memory may be preferable for real-time applications or if the number of transparent fragments drawn is very high.

As a side note, [SV14] (Multi-Layer Alpha Blending) is basically the same as HT, only that it requires Intel's fragment shader ordering extension and that it performs tail blending using standard alpha blending; which is better if the fragments are roughly ordered, but obviously much worse if they are not.

6.2.2. Quality considerations

For sufficiently large k (e.g. $k \geq 8$), HT is virtually indistinguishable from the accurate result; this is not necessarily true for the "unbounded" A-buffer, due to the implementation issues mentioned. It is especially suited for scenes with a large number of transparent fragments due to the tail blending.

7. Conclusion

In conclusion, each of the methods presented here is appropriate for a different range of use cases:

- McGuire's and Bavoil's order-independent blending is useful for applications that do not require perfect accuracy, or that desire a smooth visibility function.

Figure 4: *McGuire's and Bavoil's method*

Figure 5: *Unbounded A-buffer*

Figure 6: *Adaptive Transparency*
(note the discontinuity at the eyebrows)

Figure 7: *Hybrid Transparency*

- The unbounded A-buffer requires recent OpenGL extensions and is slower (but has the same time complexity), but generally accurate and requires only as much memory as transparent fragments are drawn.
- Hybrid Transparency requires recent OpenGL extensions, too, is about as fast as the unbounded A-buffer but requires a fixed amount of memory and is thus suitable for real-time applications with a large number of transparent fragments. The results it produces are generally visually indistinguishable from the accurate results, and due to an implementation limitation may even surpass the quality achieved using an unbounded A-buffer (again, when drawing a large number of fragments).

All of these methods can easily be integrated into existing renderers, only the fragment shaders need to be adapted accordingly in terms of where and how the newly generated fragments are written. Implementing deferred shading as described in [HBT14] is more invasive.

References

[MB13] MCGUIRE M., BAVOIL L.: Weighted Blended Order-Independent Transparency. *Journal of Computer Graphics Technologies* Vol. 2, No. 2, 2013. 1, 2, 3

[Kub14] KUBISCH C.: Order Independent Transparency In OpenGL 4.x. *GPU Technology Conference* 2014. 1, 3, 5

[MCTB13] MAULE M., COMBA J., TORCHELSEN R., BASTOS R.: Hybrid Transparency. 1, 5

[SML11] SALVI M., MONTGOMERY J., LEFOHN A.: Adaptive Transparency.
(GDC 2011 slides and reference code: https://software.intel.com/en-us/articles/adaptive-transparency) 4

[Sal13] SALVI M.: Pixel Synchronization: Solving Old Graphics Problems with New Data Structures. *SIGGRAPH* 2013. 5

[HBT14] HILLESLAND K. E., BILODEAU B., THIBIEROZ N.: Deferred Shading for Order-Independent Transparency. *EUROGRAPHICS* 2014. 5, 6

[SV14] SALVI M., VAIDYANATHAN K.: Multi-Layer Alpha Blending. 5

Global illumination in real-time

Josef Schulz

Abstract

The calculation of global illumination in real-time is not a solved problem yet. There exist a lot of approaches to simplify the physical equation for calculating the illumination fast enough to reach interactive frame rates. This paper gives an overview of common techniques and describes an algorithm, which is mainly based on two simplifications to speed up the calculation of illumination. On one hand, the geometry will be approximated by voxels and on the other hand the global illumination will be calculated in a separate frustum. The size of the frustum directly influences the complexity of the calculations.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation—global illumination, scene voxelization

1. Introduction

Illumination in computer graphics is a necessary technique for shading objects and it also improves the perceptual impact of the resulting images. Geometrical optics is a physical model, which describes the propagation of light in terms of rays. Interaction between light and matter are restricted to absorption, scattering, refraction and reflection, by this model. Other effects like diffraction or polarization are not included. They only occur in very small scales, which will not be considered here. Features like direct and indirect illumination produce soft shadows and they can be derived from geometrical optics.

Resulting equations are heavy to compute, for this reason in most programs local illumination will be used for shading objects and simulating effects of light. Nowadays there are many approximating approaches that are fast enough to generate the global illumination for real-time applications.

This work begins with an overview of existing techniques and afterwards describes an algorithm in detail. The considered algorithm accelerates the necessary calculations for indirect lighting by simplifying geometry through voxels. In addition, not the entire scene is divided into voxels, therefor a specific view-frustum will be used. Screen-space describes techniques which calculate the indirect light bounces in the view-frustum of the camera. The described algorithm is inspired by this idea and extends it by defining a separated frustum for the voxelization of the scene.

A fast voxelization and ray-voxel intersection-tests are needed. Both problems are solved and will be explained in this paper.

After introducing the related work, two forms of the rendering equation will be described. Various algorithmic solutions are explained subsequently. In Section 5, the voxel-based algorithm [THGM11] will be presented in detail and the paper ends with a conclusion.

2. Related Work

A state of the art report about global illumination in real-time is [RDGK12]. This paper gives an overview of the equations and common approaches.

The basic render equation was introduced by Kajiya [Kaj86] and the name Monte Carlo represents a numeric solution method for integrals, which performs much better than Riemann sums. A number of light-ways are produced by a raytracer and these light-ways will be used for solving the equation with the Monte-Carlo-technique. Bidirectional path tracing is a kind of raytracer which was introduced by [LW93]. These both implementations introduced the basics for generating global illumination. Reaching interactive framerates needs more approximations, because the generation of million light-ways is an expensive problem. The paper [WKB*02] describes a fast CPU based raytracer which can be used, but the framerates are still slow. A GPU based version was introduced by [PBMH02]. Another similar algorithm is called Photonmapping [Jen96]. It is a two pass technique. In the first pass photons are emitted and traced in the scene. The bounces of the photons will be stored in a

texture called Photon map. In the second pass, viewrays collect the photons. The paper [JKRY12] describes a histogram based photonmap to reach interactive framerates. Another approch introduces a fast construction method of a kd-tree on the GPU [ZHWG08] to accelerate the intersection tests.

A set of approaches use voxels to improve the calculation of the indirected light bounces as described above. One method for building such a voxelgrid in real-time is described in the paper [FFCC00]. The paper [THGM11] introduced the described algorithm. The basic idea of the voxelization algorithm bases on the work [ED06] and they extend it with a texture-atlas.

3. Shadows and light-sources

In reality, shadows are a resulting effect of illumination and they are important for scene understanding, because the generated images look more natural. If local illumination is used, shadows are often approximated by two basic algorithms, shadow volume and shadow mapping. The resulting shadows of shadow volume are just hard shadows, because the algorithm uses the stencil-buffer to draw the shadows. With shadow mapping soft shadows are possible and will be achieved mostly with smoothing.

To create a shadow-map the scene has to be rendered multiple times. For every light a z-Buffer needs to be filled and after the so called shadow maps are created, the actual image can be produced in a last step. Coordinates must be transformed in the light-coordinate-systems for every object. Afterwards the coordinates can be compared and the light occlusion can be detected. The quality of the resulting shadows depends on the size of the shadow maps.

In computer graphics, various light source types are differentiated and not all of them are physically plausible. Basic light sources are pointlights, directional and spotlights. All three have no defined surface, as opposed to more realistic light sources like sphere or area lights. Soft shadows are caused by indirect lighting and a different strength concealment of a light source. The latter requires light sources with surfaces.

4. Render equation

The equations described below limit the interactions of light and matter to those at the object boundaries. In the following equation the term L_0 describes the outgoing radiance at a point x on the surface, in the direction ω:

$$L_0(x,\omega) = L_e(x,\omega) + L_r(x,\omega) \tag{1}$$

The outgoing radiance L_0 is the sum of the emitted radiance L_e and the reflected radiance L_r. The reflected radiance can be described as the following integral:

$$L_r(x,\omega) = \int_{\Omega^+} L_i(x,\omega_i) f_r(x,\omega_i \to \omega) \langle N(x),\omega_i\rangle^+ d\omega_i \tag{2}$$

Ω^+ is a half-hemisphere at the position x, in the direction of the surface normal $N(x)$. From every direction $\omega_i \in \Omega^+$ incoming light $L_i(x,\omega_i)$ is weighted by the function f_r and the dot product $\langle,\rangle^+$. The plus symbol defines that the dot product is clamped to zero and f_r is the `brdf`, short for bidirectional reflectance distribution function. `brdf`-functions describe how light will be reflected at the opaque surfaces. Figure 1 shows the hemisphere and one incoming light ray and the outgoing radiance. Normally there is more than one incoming ray and they can come in from any direction of the hemisphere.

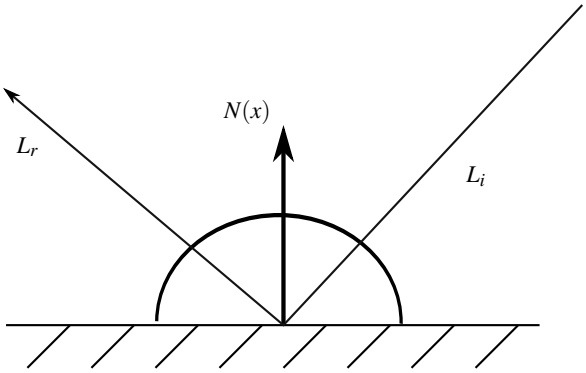


Figure 1: *This picture shows one arrow for the incoming Radiance from one direction, the surface normal and the outgoing radiance.*

There exists another way to define the equation for the reflected radiance. Instead of integrate over a hemisphere, it is possible to integrate over the total scene surface S:

$$L_r(x,\omega) = \int_S L(s,\omega_i) f_r(x,\omega_i \to \omega) \langle N(x),\omega_i\rangle^+ G(x,s) ds \tag{3}$$

Therefore the term $G(x,s)$ is needed to describe the visibility from a point on a surface s to the point x. G is called the geometry term and the direction $\omega_i = \omega'/||\omega'||$ can be calculated as the difference vector $\omega' = s - x$. The geometry term is defined as the following:

$$G(x,s) = \frac{\langle N(s),-\omega_i\rangle^+ V(x,s)}{||s-x||^2} \tag{4}$$

$V(x,s)$ defines the visibility from s to x in the equation 4, if an object is between x and s the visibility term is zero or otherwise one.

/

Solving these equations is necessary to compute global illumination. A short overview of common techniques will be given in the following section.

5. Common Techniques

This section gives an overview of four classic ways to calculate the global illumination. It is just a short summery of chosen techniques and there differences, for more detailed information the paper [RDGK12] is a good alternative. Finite elements is an example method, which is mainly structured like the second render equation 3. Following the Monte-Carlo technique for solving integrals will be described and a set of implementation options will be presented. The section ends with instant radiosity, which presents a further set of algorithms.

5.1. Finite Elements

Finite Elements or radiosity was introduces by the paper [GTGB84]. The surface of the scene will be discretize in finite surface patches at first. And after that, between every single patch the light transport have to be calculated. Every diffuse patch needs to store a value for the radiosity. For every non-diffuse patch the directional distribution of incoming and outgoing light needs to be stored.

To calculate the transport of light, some linear systems of equations needs to solved. The resulting form factors denote the amount of light, which will be transported between the patches.

5.2. Monte-Carlo technique

An integral could be solved numerical with Riemann sums. When the dimension gets bigger the calculations costs explode. An alternative to the Riemann-sums is the Monte-Carlo technique. The problem is mapped to an expectation problem:

$$F = E[\widehat{f}] = \frac{1}{N}\sum_{i=1}^{N}\frac{f(x_i)}{p(x_i)} \tag{5}$$

The equation 5 shows the Monte-Carlo estimator, the variable N defines the number of samples and $p(x_i)$ is the probability to select the sample x_i. Finally, the expectation value of an estimation function $\widehat{f}$ corresponds to the solution of the integral. A raytracer is used to produce a set of light-paths. Every path can be considered, as a sample and the final solution can be calculated. To produce among of such paths, pathtracer can be used. A pathtracer is a kind of raytracer, which traces paths from the eye and the light into the scene. After some bounces the tracer terminates and at the end eye- and lightpaths will be combined which each other. This approach produces many more paths in less time than a typical raytracer.

5.3. Photonmapping

Photonmapping is a similar approach to the pathtracer discussed before. In contrast to it, photonmapping needs two passes. In the first pass, photons are traced across the scene and if they interact with surfaces, they loose energy, which is stored in the so called photonmap. It is a kind of texture, which is needed for the second pass.

In the second pass, from every pixel a ray gets shot in the scene and if it intersects an object, the photons near the intersection will be collected. The collection of the photons can be a problem, if the intersection is really close to an edge, photons can be collected which are normally not visible.

5.4. Instant radiosity

Instant radiosity is nearly equal to the photonmapping approach. Like photonmapping two passes are needed. In the first pass photons will be traced across the scene. The photon hitpoints will be considered as so called virtual point lights. And the scene will be lit by these. Derived from this idea shadow maps can be used to determine locations for the virtual point lights.

6. Voxel-based Global Illumination

This section explains the voxel-based algorithm [THGM11] in detail. Monte-Carlo integration 5 will be used to solve the equation 1. Light paths are generated with a raytracer, but path-tracer could be implemented as well. Calculations are accelerated by simplifying geometry, this is done by voxels. A binary representation of the grid is stored in an RGBA-texture. The voxel-grid is generated only for a separate view frustum. Position and rotation of this frustum is completely arbitrary. The viewport is mapped to the texture-coordinates. The depth of the frustum is discretized and the discrete units will be called slices. Every bit of a 32bit texel indicates one cell in the voxel grid. Zero represents an empty voxel, while one stands for a filled voxel. Figure 2 illustrates this idea for simplifications only with 16 slices per texel.

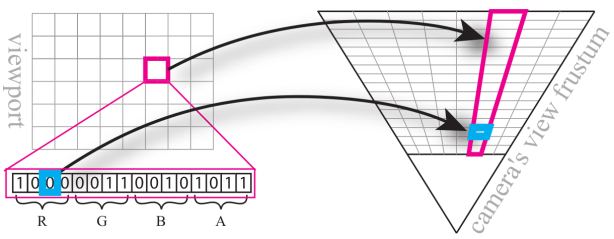


Figure 2: *This picture shows the encoding of the grid, for simplifications 4 bits are assigned for every color-channel.*

In order to increase the number of slices, multiple render targets can be used. The following two sections present two different variants to produce such voxel-grids. After that the

/

intersection-test will be described and in the last part of this section all parts will be combined to the complete algorithm.

6.1. z-Buffer method

Eisemann et al. [ED06] introduced the binary representation of the grid with textures and describes a method bases on two z-Buffers to create such a grid. While the scene is rendered, in one z-Buffer the shortest distance will be stored and the largest distance in the second one. After that for every texel the boundary of the objects can insert into the grid. Producing the grid with this method results in a lot of holes in it, more than one render pass is needed to create a complete closed representation of the geometry.

6.2. Atlas-texture method

Thiedemann [THGM11] propose a method, which uses a texture-fetch step to generating the grid. All objects need to be stored in an atlas-texture and the mapping should be predefined to accelerate the calculation. Another problem based on this strategy is the restriction to moderate object deformations. The atlas captures world-coordinates for every texel. In the next step, a pointcloud is generated for every texel of the atlas-texture. While rendering these points, the voxel-grid gets filled. Important is the size of the atlas-texture, if it is to small, holes can be generated like in the z-Buffer method, but a huge texture decreases the performance of the algorithm.

6.3. Intersection test

After the voxel-grid is created, a mip-map hierarchy will be constructed. The number of slices is constant for every mip-map level. The hierarchy of mip-maps represents a tree structure. A node in this tree is represented by a column of slices. Every node is created by logical `OR`-operation of two bitmasks, shown in figure 3.

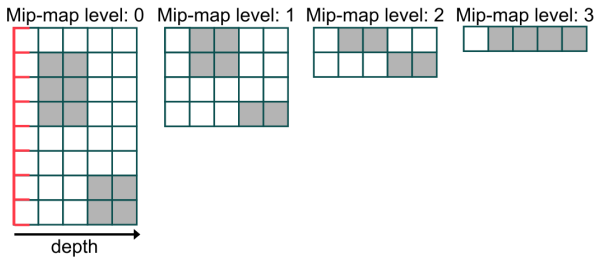


Figure 3: *The creation of the mip-maps with the `OR`-operator is represented.*

If the mip-maps are created, the intersection-test can be done by traverse the tree. In order to check for intersections, two buffers will be needed. One buffer contains the mapped ray and the second contains the intersection. Figure 4 shows the intersection test in the two-dimensional case. In the first step the ray is projected in the ray-bitmask and the intersection-mask is filled by using the `AND`-operator for the ray-bitmask and the highest mip-map level. For projecting the ray, its coordinates must be transformed in normal device coordinates first. The intersection with bounding-box needs to be checked before. All levels need to be traversed until a collision is detected or the ray-mask is empty. In the last case no intersection can be found.

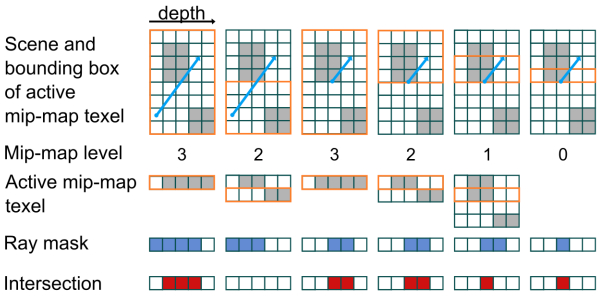


Figure 4: *Traversal the hierarchy in two dimensions.*

Currently the intersection test can only check for an intersection, but the first one is necessary for the algorithm. Searching for the first bit which is set in the ray and in the lowest level of the mip-map hierarchy solves this problem. If the direction of the ray is opposite to the direction of the voxel-frustum, the last bit has to found instead of the first one. Extracting the last bit first is one possible solution for that problem.

6.4. Complete algorithm

Before the algorithm can be described one more construction is needed. At this moment a voxel-grid can be build and an intersection test can be implemented in the fragment-shader. For the indirect bounces additional information, like normals or direct illumination needs to be stored as well. An reflective-shadow-map (`RSM`) must be generated for every light source. `RSM`s was introduced by Dachsbacher and Stamminger [DS05]. A `RSM` stores positions, normals and the direct illumination for the mapped part of the scene. Normal shadow maps are used for spotlights and cube maps for point lights.

The following lines describe in which way the indirect illumination can be calculated and figure 5 illustrates this approach. At first voxel-grid and `RSM`s will be constructed for every light source. After that the point x must be estimated. For solving the equation 1 N samples will be traced in the upper hemisphere. Therefore the voxels along the ray are traversed and if a hitpoint is found, the position will be compared with the positions stored in the `RSM`s.

For comparing the positions a $\varepsilon = \max(v, \frac{s}{\cos\alpha}, \frac{z}{z_{near}})$

/

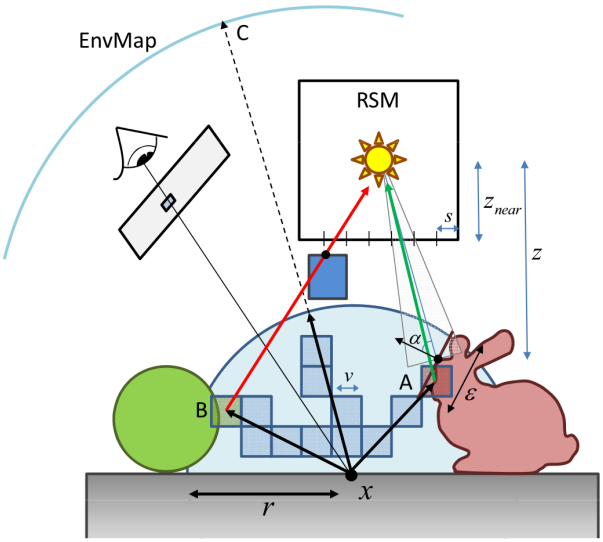


Figure 5: *Near-field indirect illumination approach.*

Figure 6: *Single bounce indirect illumination for the same scene with different radii. Frame rates from left to right: 30 fps, 27.7 fps, 25 fps.*

The algorithm use a lot of render passes and with every pass grows the overhead of the render-pipeline. As already mentioned above the method can be extend to an iterative pathtracer, but the presented technique is to slow for interactive applications. Recursive functions in shaders would help to avoid the overhead and could make such algorithms much faster.

will be defined. If the difference between both positions is smaller as the defined ε, the stored direct illumination will be used for the calculation of the indirect illumination. Otherwise there exists an element between the selected voxel and the light source and the voxel is located in the shadow of this element. In figure 5 point A shows a valid voxel and B shows a voxel, which is located in shadow. To accelerate the calculation a radius *r* is defined, if the ray does not reach a filled voxel the intersection test will be aborted. In this case the color of an environment Map will be used to define the intensity of light for this ray.

In contrast to normal image-space approaches, the voxel-frustum is defined separately. For that reason elements which blocks the light and which are not in the frustum of the camera can be detected as well.

7. Conclusion

The presented algorithms are a concise selection from the state of the art report [RDGK12]. A lot of other methods exists, for example some ignore the bias of the equation or simulate the light effects with an antiradiance approach [DSDD07]. Diffusion approaches are an other interesting idea to calculate the distribution of light. Not all of them are physically correct, but the results are pretty and interactive frame rates are an ambitious goal for global illumination on standard hardware.

Figure 6 shows the results of the explained voxel based algorithm for different radii. Even for this simple scene the frame rates are low and no other effects like physical simulation will be calculated. Static elements does not need to recompute every time, but without dynamic global illumination can be precomputed accurately an stored in textures.

/

References

[DS05] DACHSBACHER C., STAMMINGER M.: Reflective shadow maps. In *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2005), I3D '05, ACM, pp. 203–231. URL: http://doi.acm.org/10.1145/1053427.1053460, doi:10.1145/1053427.1053460. 4

[DSDD07] DACHSBACHER C., STAMMINGER M., DRETTAKIS G., DURAND F.: Implicit visibility and antiradiance for interactive global illumination. In *ACM SIGGRAPH 2007 Papers* (New York, NY, USA, 2007), SIGGRAPH '07, ACM. URL: http://doi.acm.org/10.1145/1275808.1276453, doi:10.1145/1275808.1276453. 5

[ED06] EISEMANN E., DÉCORET X.: Fast scene voxelization and applications. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (2006), ACM SIGGRAPH, pp. 71–78. URL: http://maverick.inria.fr/Publications/2006/ED06. 2, 4

[FFCC00] FANG S., FANG S., CHEN H., CHEN H.: Hardware accelerated voxelization. *Computers and Graphics 24* (2000), 200–0. 2

[GTGB84] GORAL C. M., TORRANCE K. E., GREENBERG D. P., BATTAILE B.: Modeling the interaction of light between diffuse surfaces. *SIGGRAPH Comput. Graph. 18*, 3 (Jan. 1984), 213–222. URL: http://doi.acm.org/10.1145/964965.808601, doi:10.1145/964965.808601. 3

[Jen96] JENSEN H. W.: Global illumination using photon maps. In *Proceedings of the Eurographics Workshop on Rendering Techniques '96* (London, UK, UK, 1996), Springer-Verlag, pp. 21–30. URL: http://dl.acm.org/citation.cfm?id=275458.275461. 1

[JKRY12] JÖNSSON D., KRONANDER J., ROPINSKI T., YNNERMAN A.: Historygrams: Enabling Interactive Global Illumination in Direct Volume Rendering using Photon Mapping. *IEEE Transactions on Visualization and Computer Graphics (TVCG) 18*, 12 (2012), 2364–2371. 2

[Kaj86] KAJIYA J. T.: The rendering equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1986), SIGGRAPH '86, ACM, pp. 143–150. URL: http://doi.acm.org/10.1145/15922.15902, doi:10.1145/15922.15902. 1

[LW93] LAFORTUNE E., WILLEMS Y. D.: Bi-directional Path Tracing. 1

[PBMH02] PURCELL T. J., BUCK I., MARK W. R., HANRAHAN P.: Ray tracing on programmable graphics hardware. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 2002), SIGGRAPH '02, ACM, pp. 703–712. URL: http://doi.acm.org/10.1145/566570.566640, doi:10.1145/566570.566640. 1

[RDGK12] RITSCHEL T., DACHSBACHER C., GROSCH T., KAUTZ J.: The state of the art in interactive global illumination. *Comput. Graph. Forum 31*, 1 (Feb. 2012), 160–188. URL: http://dx.doi.org/10.1111/j.1467-8659.2012.02093.x, doi:10.1111/j.1467-8659.2012.02093.x. 1, 3, 5

[THGM11] THIEDEMANN S., HENRICH N., GROSCH T., MÜLLER S.: Voxel-based global illumination. In *Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2011), I3D '11, ACM, pp. 103–110. URL: http://doi.acm.org/10.1145/1944745.1944763, doi:10.1145/1944745.1944763. 1, 2, 3, 4

[WKB*02] WALD I., KOLLIG T., BENTHIN C., KELLER A., SLUSALLEK P.: Interactive global illumination using fast ray tracing. In *Proceedings of the 13th Eurographics Workshop on Rendering* (Aire-la-Ville, Switzerland, Switzerland, 2002), EGRW '02, Eurographics Association, pp. 15–24. URL: http://dl.acm.org/citation.cfm?id=581896.581899. 1

[ZHWG08] ZHOU K., HOU Q., WANG R., GUO B.: Real-time kd-tree construction on graphics hardware. In *ACM SIGGRAPH Asia 2008 Papers* (New York, NY, USA, 2008), SIGGRAPH Asia '08, ACM, pp. 126:1–126:11. URL: http://doi.acm.org/10.1145/1457515.1409079, doi:10.1145/1457515.1409079. 2

(This page intentionally left blank.)

3 Frameworks And Technical View

Frameworks für GPGPU-Programmierung

Tobias Anker

Abstract

Es soll veranschaulicht werden, wie es möglich ist mittels GPGPU-Frameworks mathematische Aufgaben außerhalb von Grafik-Berechnungen über die GPU zu verarbeiten, um somit aufwändige Berechnungen mitunter deutlich zu beschleunigen. Dazu werden die drei Frameworks CUDA, OpenCL und OpenACC betrachtet und jeweils der allgemeine Ablauf beschrieben.

1. Einleitung

GPGPU steht für General Purpose Computation on Graphics Processing Unit, was so viel bedeutet, wie die Nutzung der Grafikkarte für verschiedenste Berechnungen. Der Grundstein hierzu wurde gelegt, indem den Programmierern die Möglichkeit zur Verfügung gestellt wurde, die für die grafische Darstellung verantwortlichen Shader selber zu programmieren und diesen Shader-Code auf der Grafikkarte direkt auszuführen. Um dies zu erreichen wurden sowohl OpenGL und Direct3D, als auch die Verarbeitungspipeline auf der Grafikkarte erweitert. An dieser Stelle entstand die Idee diese Erweiterungen auch für andere Zwecke einzusetzen.

Table 1: *Vergleich von CPU und GPU*

CPU	GPU
- Optimiert für serielle Abarbeitung	- Optimiert für parallele Abarbeitung
- Bis 8 Kerne bei 3,5 GHz (Core i7-5960X) - Bis 384 GFLOPS (Core i7-5960X)	- 3072 Kerne bei 1 GHz (Nvidia Titan X) - Bis,6600 GFLOPs (Titan X)
- RAM: bis 50 GB/s bis 64 GB niedrige Latenz	- VRAM: bis 336,5 GB/s (Titan X) bis 12 GB hohe Latenz

Tabelle 1 zeigt eine Gegengenüberstellung von CPU und GPU. Dargestellt sind hier die Werte von zwei aktuellen Prozessoren, aber auch schon in den Anfängen der GPGPU-Programmierung gab es deutliche Unterschiede zwischen CPUs und GPUs. Auf der einen Seite die CPUs, welche wenige extrem schnelle und optimierte Kerne besitzen und auf der anderen Seite die GPUs, die dagegen sehr viele Kerne besitzten, welche vergleichsweise langsam sind und keinerlei Single-Thread-Optimierung besitzen. Des weiteren können CPUs auf einen großen relativ langsamen Arbeitsspeicher zurückgreifen, welcher jedoch über eine sehr gering Latenz verfügt. Der Speicher einer Grafikkarte ist, im Vergleich zum Arbeitsspeicher, zwar von seiner Datenrate her deutlich schneller, aber dafür kleiner und nicht einfach erweiterbar und mit einer vielfach höheren Latenz behaftet. Beide besitzen also jeweils Eigenschaften, in denen sie besser sind, als der andere und dem entsprechend gibt es auch Aufgaben, für die der eine besser geeignet ist, als der andere. Es ist also sehr von Vorteil, wenn es möglich wäre, die GPUs genau für die Aufgaben einzusetzten, in denen sie den CPUs überlegen sind.

Das Problem war Anfangs, dass GPUs nur über OpenGL und Direct3D angesprochen werden konnten. Zwar war es theoretisch auch machbar beispielsweise Matrizen in eine Textur umzuwandeln und diese dann mit dem Fragment-Shader zu bearbeiten, also beispielsweise zwei Matrizen bzw. Texturen miteinander zu verrechnen, aber dies ist mit zusätzlichen Aufwand verbunden. Darüber hinaus benötigt man viele Komponenten der OpenGL-Pipeline nicht für ganz gewöhnliche Matrizenberechnungen ohne grafische Ausgabe. Im Gegenzug hatte man allerdings den Bedarf an zusätzlichen Funktionen, um mehr Kontrolle über den genauen Verarbeitungsprozess zu haben und somit die Programme besser für die jeweilige Aufgabe optimieren zu können. Um diesen Anforderungen gerecht zu werden, war es erforderlich neue Frameworks einzuführen, was zur Entstehung von CUDA, OpenCL und OpenACC führte.

2. CUDA

CUDA steht für Compute Unified Device Architecture und wurde von Nvidia für ihre eigenen Grafikkarten entwickelt. Die erste Version erschien im November 2006, zeitgleich

mit der Veröffentlichung der Geforce 8000 Serie. Mittlerweile existiert die Version 7 von CUDA. Der Funktionsumfang der Programmierschnittstelle ist abhängig von der verwendeten Grafikkarte, da sich mit fast jeder Generation die Architektur, Speichergrößen, Kernanzahl etc. geändert haben. CUDA-Programme sind in C++ - ähnlicher Sprache verfasst und benötigen zum kompilieren den nvcc-Compiler von Nvidia. Damit CUDA nicht ausschließlich an C/C++ gebunden ist, wurden seit seiner Erscheinung Wrapper für verschiedenste Programmiersprachen geschrieben, wie beispielsweise für Java oder Python. Diese sind jedoch gegenüber der C-Version deutlich eingeschränkt in ihrem Funktionsumfang.

Zum besseren Verständnis der weiteren Ausführung werden im Folgenden Begrifflichkeiten geklärt. Die Grundlage jeder GPU sind die Berechnungseinheiten. Hierbei gibt es die hierarchische Unterscheidung zwischen Threads und Blöcken. Ein Thread ist die kleinste Bearbeitungseinheit. Sie ist im Falle einer GPU sehr primitiv aufgebaut und besteht aus einigen Registern für die Operanden, einer Integer- und Float-Berechnungseinheit und Registern für die Ergebnisse. Mehrere dieser Threads bilden zusammen einen Block. Bei der Kepler-Architektur von Nvidia befinden sich bis zu 192 dieser Threads in einem Block. Dieser wiederrum besitzt neben den eigentlichen Kernen bzw. Threads einige Recheneinheiten für Double-Werte, einen Shared-Memory auf den alle Threads innerhalb dieses dazugehörigen Blocks zugreifen können und diverse Buffer und Register. Von diesen Blöcken befinden sich wiederrum mehrere auf einer GPU, zusammen mit einem großen L2-Cache, Memory-Controllern und diversen anderen Komponenten.

Desweiteren werden im nachfolgenden Text die Begriffe “Host” und “Device” verwendet. Der Host ist dabei das Synonym für die CPU und Device entspricht der GPU bzw. Grafikkarte.

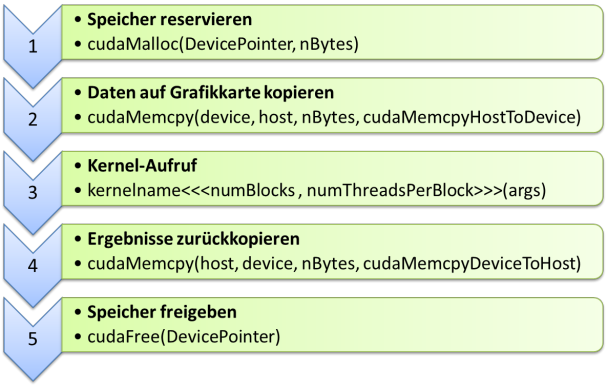


Figure 1: *Grundlegender Ablauf eines CUDA-Programms*

Der Grundaufbau eines CUDA-Programms besteht aus 5 Teilen, welche in Abbildung 1 dargestellt sind. Neben diesen gibt es in CUDA weitere mehr Möglichkeiten und seit Version 6.0 auch die Funktion die Kopiervorgänge zwischen Arbeitsspeicher und Grafikkartenspeicher dem Compiler zu überlassen. Um den grundlegenden Aufbau zu verdeutlichen, wird diese Funktion an dieser Stelle nicht behandelt.

Zu Beginn muss der benötigte Speicher auf der Grafikkarte reserviert werden. Realisiert wird dies mittels der Methode *cudaMalloc(...)*, welcher beim Aufruf ein Pointer übergeben werden muss, der später auf den reservierten Grafikkartenspeicher zeiget und die Anzahl der zu reservierenden Bytes. Diese Methode muss für jeden Speicherbereich ausgeführt werden, auf welchen das Programm während der Abarbeitung zugreift und muss bei jedem Aufruf mit einem anderen Pointer verbunden werden. Anschließend müssen sämtliche Daten auf das Device kopiert werden mit Hilfe von *cudaMemcpy(...)*. Hierbei werden zuerst das Ziel und anschließend die Quelle übergeben. Als Ziel dient der Pointer, welcher auf den allokierten Speicher aus Schritt 1 zeigt und als Quelle das Objekt, welches die zu kopierenden Daten beinhaltet. Zusätzlich werden noch die Anzahl der zu kopierenden Bytes benötigt und das Flag *cudaMemcpyHostToDevice*, damit der Compiler und das Programm die Information erhalten, in welche Richtung die Daten fließen und auf welchem Gerät sich das Ziel bzw. die Quelle befindet. Ebenso könnte sich die Quelle auf GPU 1 und das Ziel auf GPU 2 befinden. In diesem Fall muss das Flag auf *cudaMemcpyDeviceToDevice* gesetzt werden.

In Schritt 3 wird der Kernel aufgerufen, in welchem sich der auf der GPU auszuführende Code befindet. Bei ihm handelt es sich um eine Methode, die in jedem Thread der GPU identisch ist und exakt gleich abgearbeitet wird. Der Aufruf erfolgt ähnlich wie bei einer normalen Methode mittels Kernelnamen und in runden Klammern die Pointer, welche auf die Daten im Grafikkartenspeicher verweisen. Im Gegensatz zu einem normalen Methodenaufruf müssen hier jedoch noch in dreifachen spitzen Klammern die Anzahl der Blöcke und die Anzahl der Threads pro Block angegeben werden, auf denen der Kernel ausgeführt werden soll. Es besteht die Möglichkeit deutlich mehr Blöcke und Threads anzugeben, als in physischer Form auf der GPU vorhanden sind. Die Angabe erfolgt daher in Form von virtuellen Blöcken und Threads. Sobald ein physischer Thread oder Block einen virtuellen fertig abgearbeitet hat oder alle Kernel innerhalb des Blocks beendet sind, bezieht dieser den nächsten virtuellen. Sowohl bei der Blockanzahl, als auch bei der Threadanzahl, ist es möglich ein dim3-Objekt zu übergeben. Dabei handelt es sich um ein Struct-Objekt, mit dem eine Ausdehnung in x-, y- und z-Richtung definiert wird. Auf die Abarbeitung auf der GPU hat dies keinen Einfluss, aber es erleichtert dem Programmierer die Arbeit auf 2- und 3-dimensionalen Arrays innerhalb des Kernels. Die Kernel-Methode benötigt zusätzlich zu einer normalen Methode noch ein vorangestelltes “__global__”. Dies kann gleichgesetzt werden mit einem “public” und gibt an, dass der Kernel auf vom Host aus aufgerufen werden kann. Alternativ gibt es noch “__private__” was bedeutet, dass dieser Kernel nur innerhalb eines anderen

Kernels aufgrufen werden kann und entspricht damit einem “private”.

Es wurde erwähnt, dass jeder Thread den gleichen Kernel-Code ausführt. Um jedem Thread eine andere Aufgabe innerhalb des Kernels zuzuweisen oder einen bestimmten Bereich eines Arrays zuzuordnen, benötigt der Thread seine Position innerhalb des durch die virtuellen Blöcke und Threads aufgespannten Gitters. Für diesen Zweck existieren Variablen, die innerhalb des Kernels abgefragt werden können. Mittels der Variablen *blockIdx.x*, *blockIdx.y* und *blockIdx.z* kann der Thread erfragen, in welchem virtuellen Block in x-, y- und z-Richtung er sich befindet, mit *blockDim.x*, *blockDim.y* und *blockDim.z* wie viele Threads ein Block in den jeweiligen Richtungen besitzt und mit *threadIdx.x*, *threadIdx.y* und *threadIdx.z* an welcher Position er innerhalb seines Blocks liegt. Mit Hilfe dieser Werte ist es möglich innerhalb des Kernels für jeden Thread eine ID zu berechnen. Für den eindimensionalen Fall sähe dies so aus: int tid = threadIdx.x + blockIdx.x * blockDim.x.

Sobald der Kernel mit seiner Aufgabe fertig ist, kann mittels *cudaMemcpy* das Ergebnis von der Grafikkarte wieder zurück in den Arbeitsspeicher übertragen werden. Im Unterschied zu dem Kopiervorgang von Host zu Device, ligt hier die Zielvariable auf dem Host und als Quelle muss der Pointer angegeben werden, welcher auf das Ergebnis im Grafikspeicher verweist. Dem entsprechend ist das Flag *cudaMemcpyDeviceToHost* anzugeben. Nach dem Abschluss aller Kernel und Kopiervorgänge ist der anfangs reservierte Speicher wieder freizugeben. Dafür ruft man für jeden Pointer, der auf den Grafikspeicher zeigt, die Methode *cudaFree* auf.

3. OpenACC

Auf Grund dessen, dass viele Vorgänge innerhalb eines CUDA-Programms sich immer wieder relativ ähnlich wiederholen, wie das reservieren und kopieren von benötigten Variablen, wurde versuchte derartige Aufgaben auf den Compiler auszulagern und den benötigten Code auf ein Minimum zu reduzieren. Um dies zu erreichen wurde OpenACC entwickelt und im November 2011 veröffentlicht. Es unterstützt sowohl Nvidia- , als auch AMD-Grafikkarten und wird in C/C++ programmiert. Um ein OpenACC-Programm kompilieren zu können benötigte man bisher den PGI- oder CRAY-Compiler, aber auch der GCC-Compiler soll ab Version 5.0 erstmals über eine OpenACC-Unterstützung verfügen. OpenACC funktioniert pragmabasiert, das heißt, dass nur eine Zeile *#pragma acc <direktive> [Klausel[, Klausel]...]* über dem zu parallelisierenden Bereich hinzufügt werden muss. Die Klauseln sind optionale Zusätze wie beispielsweise eine Reduktions-Klausel, falls eine parallelisierte Schleife eine Variable am Ende über allen Threads aufaddieren soll. Außerdem ist es hierüber möglich gezielt Daten auf die Grafikkarte zu übertragen, wenn man die Planung der Ausführung nicht dem Compiler überlassen will. Wichtiger hingegen sind die Direktiven, da an dieser Stelle immer etwas angegeben werden muss. Die Beiden, die an dieser Stelle betrachtet werden, sind *parallel* bzw. *parallel loop* und *kernels*. Mit *parallel* wird dem Compiler während des kopilierens mitgeteilt, dass der nachfolgende Code-Abschnitt definitiv parallelisierbar ist und mittels GPU verarbeitet werden soll. Die Direktive *parallel loop* ist eine Erweiterung des Ganzen und konkretisiert den Befehl, indem gesagt wird, dass es sich beim zu parallelisierenden Bereich um eine Schleife handelt. Mit Hilfe dieser Zusatzinformationen ist es möglich sowohl den build-Vorgang zu beschleunigen, als auch einen Geschwindigkeitszuwachs bei der Programmausführung zu erzielen. Die einfachste und allgemeinste Direktive ist *kernels*. Hiermit wird der Compiler dazu angewiesen, im nachfolgenden Abschnitt alle parallelisierbaren Abschnitte zu suchen und diese auf der Grafikkarte auszuführen. Dies kann auch über sehr große Abschnitte angewendet werden, verlängert die Zeit für das kompilieren des Programms und erzeugt möglicherweise keine optimalen Ergebnisse.

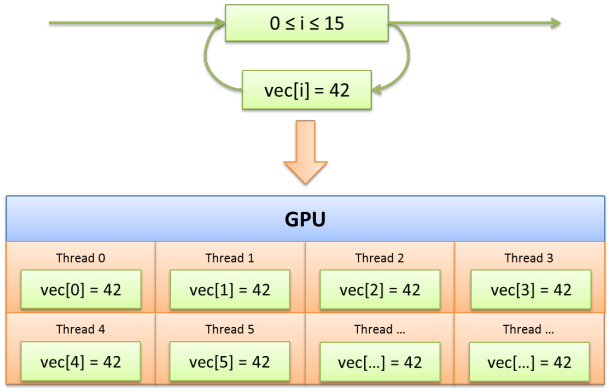


Figure 2: *Beispiel eines OpenACC-Ablaufs*

Abbildung 2 soll schematisch zeigen, wie OpenACC in etwa vorgeht, anhand einer Schleife, welche 16 mal durchlaufen wird und dabei jedes der 16 Felder eines Array mit 42 initialisiert. Wenn nun über einer derartigen For-Schleife ein *#pragma acc parallel loop* gesetzt wird, würde OpenACC diese Schleife, wie in der Abbildung dargestellt, zerlegen und jeden einzelnen der Schleifendruchläufe separat auf einem Thread der Grafikkarte ausführen.

4. OpenCL

OpenCL ähnelt einer Mischung zwischen CUDA und OpenGL und wurde 2009 veröffentlicht. Am 3.Mörz 2015 ist Version 2.1 erschienen. OpenCl unterstützt neben Nvidia- und AMD-Grafikkarten auch zusätzlich Prozessoren mit SSE3-Erweiterung von Intel und AMD und auch exotischere Hardware, wie beispielsweise der Intel Xeon Phi.

Auch hier wird die in Kapitel 2 genannte Herarchie der

Threads und Blöcke verwendet, wobei bei OpenCL die Threads “Work-Items” und Blöcke “Work-Groups” genannt werden. Der Hintergrund hierfür liegt darin begründet, dass OpenCL nicht nur auf Grafikkarten funktioniert. Im Gegensatz zu einer GPU lässt sich beispielsweise eine CPU nicht in Blöcke unterteilen.

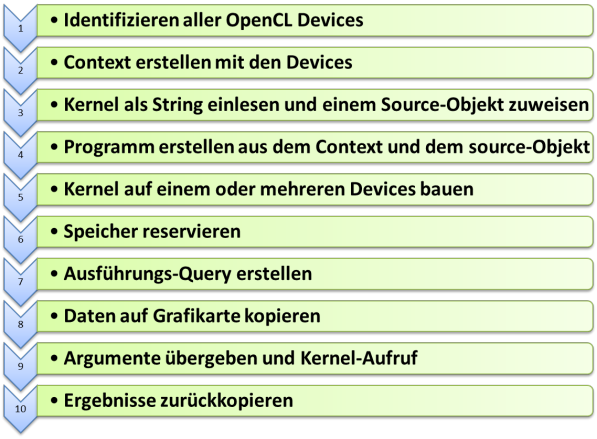


Figure 3: *Grundlegender Ablauf eines OpenCL-Programms*

Wie in Abbildung 3 zu sehen ist, ist der Ablauf eines OpenCL-Prgramm etwas umfangreicher als der eines CUDA-Programms. Die Schritte 6 - 10 stellen die bereits aus Kapitel 2 bekannten Vorgänge des reservierens und kopierens von Speicher und der Kernel-Ausführung dar. Die Schritte 1 - 5 sind nötig, da OpenCL auf vielen unterschiedlichen Geräten funktionieren muss.

Zu Beginn müssen alle OpenCL-fähigen Devices mittels der Methode *getDevices(...)* identifiziert und in einer Liste gespeichert werden. Diese können auch je nach Art gefiltert werden. Beispielsweise ist es möglich mit der Flag *CL_DEVICE_TYPE_GPU* alle kompatiblen Einheiten vom Typ GPU ausgeben zu lassen. Anschließend wird ein Context-Objekt erstellt, welchem die vorher erstellte Liste von Devices übergeben wird und welches als zentrales Verbindungsglied aller nachfolgenden von OpenCL benötigen Objekte dient.

Im nachfolgenden 3. Schritt kann der Kernel-Code geladen werden. Der Kernel liegt hierbei als String vor oder kann alternativ aus einer externen Datei als String eingelesen werden. Auch dieser besitzt Besonderheiten. Zum einen muss bei den übergebenen Argumenten die Herachie berücksichtigt werden. Im Falle einer Grafikkarte ist der Grafik-Speicher auf der Karte der *global*-Speicher und der Shared-Memory, was etwa einem L1-/L2-Cache beziechnet und pro Block vorhanden ist, wäre der *lokal*-Speicher. Dem entsprechend ist es bei OpenCL nötig dem Kernel mitzuteilen, wo sich die Daten befinden, indem ein *global* oder *lokal* vor der übergebenen Variable steht. Eine weiterere Besonderheit besteht in der Art, wie die einzelnen Work-Items ihre Position im aufgespannten Gitter ermitteln. Hierzu reicht ein *get_global_id(0)*, um die globale ID in X-Richtung abzufragen. Die Begrifflichkeit *global_id* besagt, dass es sich dabei um die ID über alle Work-Groups hinweg handelt und die *0* bezeichnet dabei die X-Achse. Genau so lässt sich mit einer *1* die globale Postion in Y-Richtung oder mittels *get_local_id(0)* die ID innerhalb einer Work-Group in X-Richtung ermitteln.

Dieser Kernel-Code wird alleine oder zusammen mit weiteren Kerneln einem Source-Objekt hinzugefügt und anschließend zusammen mit dem Context aus Schritt 2 zu einem Programm-Objekt verbunden. Auf diesem Objekt wird in Schritt 5 die Methode *build(devices)* aufgerufen und ihr eine Liste mit allen Devices übergeben, auf welchen das Programm und damit der Kernel-Code kompiliert werden soll. Der weitere Ablauf ist ähnlich zu dem aus Kapitel 2. In Schritt 6 wird anschließend der benötigte Speicher auf der Grafikkarte reservert, indem ein Buffer-Objekt erzeugt wird, welches das Context-Objekt, eine Flag für die Regelung von Schreib-Lese-Zugriffsrechten und der Byte-Anzahl übergeben bekommt. Die nachfolgenden Schritte benötigen eine CommandQueue, welche für jedes Device erstellt werden muss und in welcher alle nachfolgenden Aufgaben eingereiht werden. Dieser werden alle Kopierbefehle von Host zu Device mittels der Methode *enqueueWriteBuffer(...)* übergeben. Nachdem alle Daten auf das Device übertragen worden, kann der Kernel aufgerufen werden. Der Kernel-Aufruf sieht dabei so aus, dass zuerst ein Kernel-Objekt erzeugt wird, welchem das gebaut Programm aus Schritt 5 und ein Name für den Kernel übergeben wird. Anschließend werden alle Argumente übergeben und der Kernel mit einigen Parametern an die CommandQueue übergeben. Bei OpenCL wird, neben einem Offset, welcher einen Versatz bei der im Kernel abfragbaren global-Id erzeugt, die Gesamtanzahl aller Work-Items und anschließend die Anzahl von Work-Items pro Work-Group angegeben. Abschließend wird mittels *finish()*-Methode, aufgerufen durch die CoammanQueue, gewartet bis der Kernel mit seiner Aufgabe fertig ist. Nach erfolgreichen Abschluss können im letzten Schritt durch die Methode *enqueueReadBuffer(...)* alle Ergebnisse wieder zurück in den Arbeitsspeicher kopiert werden.

5. Gegenüberstellung

Die Tabelle in Abbildung 4 soll lediglich als kleine Zusammenfassung der drei betrachteten Frameworks dienen mit einigen Kernaussagen der letzten drei Abschnitte. Alle beitzen sowohl Vor- als auch Nachteile und es ist dem Programmierer je nach Aufgabe überlassen, welches der drei er für seine Aufgabe als geeignetsten ansieht. Der größte Nachteil von CUDA besteht darin, dass ausschließlich Nvidia-Grafikkarten unterstützt werden. Dies erzeugt wiederum auch Vorteile gegenüber OpenCL auf Nvidia-Grafikkarten. OpenCL ist an eine recht einfache Speicherherarchie gebunden, welche universell auf verschiedenste Hardware ange-

	CUDA	OpenACC	OpenCL
Plattformen	GPUs von Nvidia	GPUs von Nvidia und AMD	GPUs von Nvidia und AMD, CPUs von Intel und AMD, Intel Xeon Phi, ...
Compiler	nvcc	PGI, gcc 5.0, Cray-Compiler	gcc , clcc
Vorteil:	- Sehr effizient auf Nvidia-GPUs	- Sehr schnell und einfach zu implementieren	- Sehr breite Unterstützung
Nachteil	- Nur auf Nvidia-GPUs - Nicht OpenSource	- Relativ begrenzte Möglichkeiten der Anpassung - Je nach Aufgabe mehr oder weniger effizient	- Auf Nvidia-GPUs schlechter als CUDA

Figure 4: *Gegenüberstellung der 3 Frameworks*

wendet werden kann und kann daher nicht auf sehr spezifische Buffer einer Grafikkarte oder CPU zugreifen. Auf Nvidia-Karten können daher Komponenten wie beispielsweise die Texture-Buffer nicht für die Optimierung von Berechnungen mit einbezogen werden, was in diesem Fall nur mit CUDA möglich ist. Auf Grund dessen, dass für derartige Optimierungen recht gute Hardware-Kenntnissen benötigen und sich der Geschwindigkeitszuwachs nur selten gegenüber des investierten Programmieraufwands lohnt, reichen in den meisten Fällen die grundlegenden Speicher und Buffer, welche auch von OpenCL verwendet werden. OpenACC ist jedoch nicht so universell anwendbar wie OpenCL, unterstützt dafür gegenüber CUDA auch AMD-Grafikkarten. Der minimal nötige Programmieraufwand für die Parallelisierung mittels GPU ist bei OpenACC mit mindestens einer nötigen Programmzeile im Vergleich zu den anderen Beiden auch am geringsten. Der Hauptteil der Parallelisierungsarbeit wird dem Compiler überlassen, wodurch nur minimale Programmierkenntenisse nötig sind und sehr wenig Zeit in die GPU-Programmierung investiert werden muss.

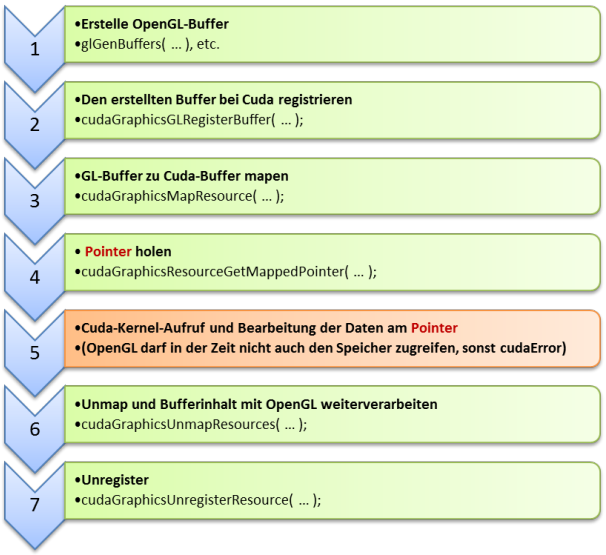


Figure 5: *Zusammenarbeit zwischen CUDA und OpenGL*

6. Verbindung mit OpenGL

Aus dem Grund heraus, dass sowohl OpenGL, als auch die drei vorgestellten Frameworks auf dem gleichen Grafikkartenspeicher arbeiten, wurden auch Möglichkeiten geschaffen diese Framework mit OpenGL zusammen arbeiten zu lassen. Auf diesem Weg ist man beispielsweise in der Lage Daten aus dem Pixel- oder Texture-Buffer von OpenGL mittels eines CUDA- oder OpenCL-Kernels zu bearbeiten. Bei allen drei Fremaworks ist der Ablauf sehr ähnlich.

Für CUDA ist dies in Abbildung 5 dargestellt. Als Erstes wird ein OpenGL-Buffer-Objekt erzeugt und gefüllt. Dabei kann es sich sowohl um einen Texture-, als auch einen Pixel-Buffer handeln. Anschließend wird ein leeres CUDA-Buffer-Objekt erzeugt und beim OpenGL-Buffer registriert. Dabei werden dem CUDA-Objekt alle nötigen Information zugewiesen, wie Position und Größe des OpenGL-Buffers im Speicher, sodass sich dieser an der gleichen Stelle im Speicher befindet. Im dritten Schritt wird nun einen Art lock-Befehl ausgeführt. Ab diesem Zeitpunkt gehört der Inhalt des OpenGL-Buffers dem CUDA-Buffer, welcher sich auf Grund des vorherigen Schrittes bereits an der gleichen Speicherposition befindet und OpenGL wird es verboten auf diesen Inhalt zugreifen, da CUDA sonst einen Fehler erzeugen würde. Im anschließenden Schritt muss der Pointer auf das Objekt abgefragt werden, um Zugriff auf den Speicherbereich zu erhalten. Nachfolgend ist es möglich einen CUDA-Kernel aufzurufen, welchem dieser Pointer übergeben wird, um innerhalb des Kernels den Bufferinhalt zu bearbeiten. Sobald der Kernel seine Aufgabe erfolgreich beendet hat, kann der Speicherbereich in Schritt 6 wieder freigegeben werden, sodass OpenGL wieder Zugriff erhält und den Inhalt grafisch ausgeben kann. Danach bleibt die Möglichkeit

entweder diesen Vorgang ab Schritt 3 zu wiederholen, oder die Buffer-Registrierung aus Schritt 2 aufzuheben.

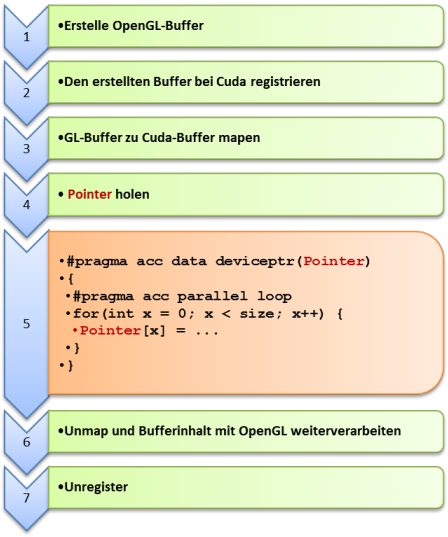


Figure 6: *Zusammenarbeit zwischen OpenACC und OpenGL*

Im Falle von OpenACC gibt es die Möglichkeit es über CUDA mit OpenGL zu verbinden, wie Abbildung 6 zeigt. Die Abfolge ist dementspechend ähnlich zur CUDA-Vairante. Der einzige Unterschied liegt darin, den Kernel durch eine OpenACC-optimierte Schleife zu ersetzen, welche dann statt des CUDA-Kernels die Daten im Buffer bearbeitet. Um dies zu ermöglichen muss zusätzlich noch eine *#pragma acc data deviceptr* angegeben werden, damit OpenACC die Information erhält, dass die Daten bereits auf der Grafikkarte liegen und es dementsprechend auf die Daten des übergebenen Pointers zugreift.

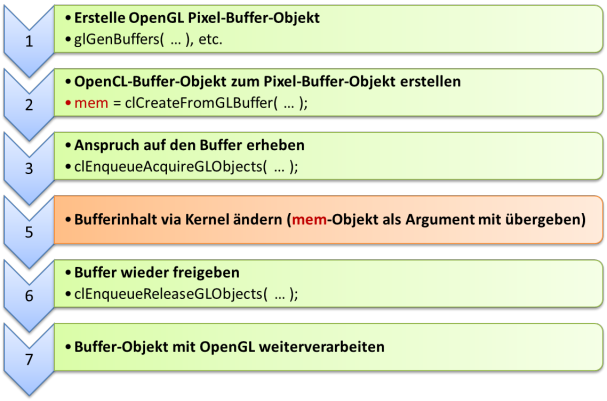


Figure 7: *Zusammenarbeit zwischen OpenCL und OpenGL*

Auch OpenCL weist Ähnlichkeiten zu CUDA auf, wie in Abbildung 7 zu sehen ist. Der einzige Unterschied, neben den unterschiedlichen Methodennamen, liegt darin, dass kein eigener Buffer erstellt wird und anschließend an den OpenGL-Buffer angepasst wird, sondern direkt aus dem OpenGL-Bufferobjekt das OpenCL-Objekt generiert wird. Danach ist auch hier der Ablauf wie bei CUDA. Der Buffer wird gelockt, der OpenCL-Kernel mit dem generierten Buffer-Objekt aufgerufen und nach erfolgreicher Bearbeitung wird der Buffer wieder freigegeben und kann von OpenGL grafisch ausgeben werden.

References

[Gas] GASTER B. R.:. URL: https://www.khronos.org/registry/cl/specs/opencl-cplusplus-1.1.pdf.

[ho] HEISE ONLINE:. URL: http://www.heise.de/newsticker/meldung/Intel-Achtkernprozessor-fuer-Desktop-PCs-2304737.html.

[(In] (INTEL) M. S.:. URL: https://software.intel.com/en-us/articles/opencl-and-opengl-interoperability-tutoria.

[Lar] LARKIN J.:. URL: http://on-demand.gputechconf.com/gtc/2013/presentations/S3076-Getting-Started-with-OpenACC.pdf.

[MSa] MICHAEL SCHWABL STEFAN HASSLACHER M. H.:. URL: http://www.cosy.sbg.ac.at/~held/teaching/wiss_arbeiten/slides_09-10/CUDA.pdf.

[MSb] MICHAEL SCHWABL STEFAN HASSLACHER M. H.:. URL: http://www.cosy.sbg.ac.at/~held/teaching/wiss_arbeiten/slides_09-10/CUDA.pdf.

[nvia] NVIDIA:. URL: http://www.nvidia.de/object/geforce-gtx-titan-x-de.html#pdpContent=2.

[nvib] NVIDIA:. URL: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#axzz3YgXUdUjC.

[Sta] STAM J.:. URL: http://www.nvidia.com/content/gtc/documents/1055_gtc09.pdf.

[Thi] THIEL P.:. URL: https://www.matse.itc.rwth-aachen.de/dienste/public/show_document.php?id=7165.

[Tre] TREVETT N.:. URL: https://www.khronos.org/assets/uploads/developers/library/overview/opencl_overview.pdf.

[unba] UNBEKANNT:. URL: http://simpleopencl.blogspot.de/2013/06/tutorial-simple-start-with-opencl-and-c.html.

[unbb] UNBEKANNT:. URL: http://dhruba.name/2012/10/06/opencl-cookbook-hello-world-using-cpp-host-binding/.

[vO] VAN OOSTEN J.:. URL: http://www.3dgep.com/opengl-interoperability-with-cuda.

[Wik] WIKI G.:. URL: https://gcc.gnu.org/wiki/OpenACC.

[Woo] WOOLLEY C.:. URL: http://www.cc.gatech.edu/~vetter/keeneland/tutorial-2011-04-14/06-intro_to_opencl.pdf.

API Alternatives to OpenGL

Martin Rühlicke

Computer Graphics and Visualization (CGV) lab, Faculty of Computer Science, TU Dresden, Germany

This paper introduces four alternatives to the commonly used OpenGL API. It describes the advantages and disadvantages of DirectX 11, the upcomming DirectX 12 and the relatively new solutions Mantle from AMD and Metal from Apple. In addition to a short presentation of these API's a comparison of performance and available platforms is given. Furthermore the differences between high-level-API's and the new low-level-API's are described by pointing out differences, benefits and drawbacks.

1. Introduction

Over the last years OpenGL and DirectX became the market-leading solutions for creating computer graphics. Since the beginning of 2013 the situation changed, when AMD introduced their new low-level graphics API called Mantle [Rig14]. Soon Apple followed with its own low-level solution called Metal [Era14].
Nowadays Mantle is already out of date through the development of the new DirectX 12 and the announcement of the OpenGL successor Vulkan [Dav15].
For the different OpenGL alternatives a short description of features, benefits and drawbacks as well as a brief overview of supporting platforms, engines and games is given. Besides this introduction a comparison between the two API-types, high-level and low-level, is one of the main topics of this article.
The paper describes DirectX 11, DirectX 12, AMD's Mantle and Apples Metal whereas in the first section DirectX 11 is addressed to be the only representation of a high-level API. One topic besides a short overview is the description of new rendering-features in the upcoming version 11.3.
Subsequently the first low-level API, Mantle, is discussed by giving some information about the development history followed by the advantages compared to high-level API's and a description of the main features of Mantle. Finally a short overview why Mantle is already out of date is given.
The next section introduces the upcoming DirectX 12. The new systems for efficient graphic calculation are described in short followed by a brief overview of supporting platforms and engines.
Finally Apples Metal is described by also showing the features, advantages and disadvantages.
After a comparison of the four described API's a short conclusion and a prospect to the future is given.

2. Related work

A detailed introduction to DirectX 11 is given in [Oos14] where the main features and some code examples are described. More information about the new main features of DirectX 11.3 respectively DirectX 12, and the parallel development of both API's can be found in [Smi14].
Sources of information about Mantle are on the one hand the official presentation of the API from the GDC14 ([Rig14]) with in detail descriptions of the main features and advantages compared to low-level API's like DirectX 11. On the other hand some related articles with performance tests can be found at ([Dem13], [Fis13], [Sau14a]). Since the official developer website of Mantle is only available for registered developers this article does not use this source of information.
In [Che15] a short but detailed introduction to DirectX 12 and a description of the differences to DirectX 11 is given. Additionally there are also performance tests made by Intel with their Asteroid demo ([Lau14]) and by 3DMark with their API Overhead Test ([And15], [Fut15]) who are related to DirectX 12. As mentioned above [Smi14] is also a good resource of information concerning DirectX 12, since the new features of DirectX 12 and DirectX 11 are the same.
The major source of information concerning Metal is the official developers page ([Inc15a]) where all features and their use are described in detail with code examples. Additionally Appleinsider ([Era14]) has a detailed overview which covers the main features, some performance tests and a short look

to the future of Metal. An introduction into programming with Metal can be found in the video recording of Warren Moores presentation ([Moo15]) of Metal and Swift.

3. Microsoft's high-level API DirectX 11

DirectX 11 is the latest iteration of Microsoft's popular graphic API and is currently available for Windows 8.1 in version 11.2. Microsoft released DirectX 11 on October 22, 2009 exclusively for Windows 7 and Windows Vista. The next iteration DirectX 11.1 was released exclusive for Windows 8.
Like its predecessors DirectX 11 is also a high-level API with limited hardware access. Parallel to DirectX 12 Microsoft will also release DirectX 11.3 which will have the same features as DirectX 12 but without the overhead-reduction. It is meant as the high-level counterpart to the low-level DirectX 12.

3.1. New features of DirectX 11

Besides some improvements of the rendering pipeline the main feature Microsoft introduces in DirectX 11 is the support of Shader Model 5.0. This leads to two new types of shaders available in DirectX 11 [Ste08].
The first type is the compute shader which runs outside of the rendering pipeline and is able to perform general parallel program executions according to the SIMD-principle. Since the architecture of modern GPUs is designed to compute a lot of tasks simultaneously, work which can easily parallelized such as physic simulations or ai calculations can be done more efficient than on the CPU.
The second new shader type in DirectX 11 is the tessellation shader which is divided in three stages. In the graphic pipeline the tessellation shader is located after the vertex shader and before the geometry shader.
First a hull shader calculates a tessellation factor which defines the granularity of the subdivision. This gives developers more control over the tessellation and allows to save performance. Since only parts of an object, who are close to the camera need to be detailed, the tessellation factor has to differ according to the distance. Figure 1 shows an example of this partial tessellation where the stones of the road are getting flat and less detailed in the distance.
The next stage of the tessellation shader is the tesselator where the subdivision of the triangles based on the tessellation factor is performed.
Subsequently the domain shader calculates the final vertex attributes based on the tesselator data and sends them to the geometry shader.

3.2. Features coming with DirectX 11.3

Microsoft announced some new major features for DirectX 11.3 and DirectX 12 such as Rasterizer Ordered Views (ROV) that allow the developers to easily and completely control the order of the rasterization. This is especially for Order Independent Transparency (OIT) a huge improvement. Without ROVs a hardware-intensive sort of the fragments is needed to compute the transparency.

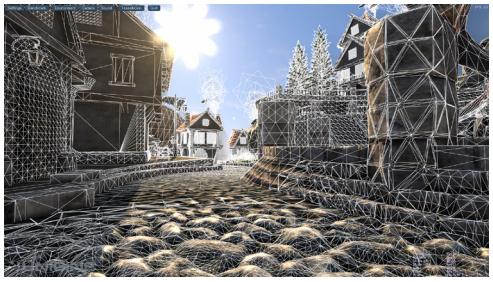


Figure 1: Tesselation of an object with different tessellation factors in relation to the cameradistance.

Due to the resulting long rendering times, the use of OIT is often restricted to scopes where no real-time graphics are needed, such as CAD [Smi14]. With the introduction of ROVs the sorting is no longer needed and the rendering time decrease so that also real-time renderings are possible [Smi14].
Another new feature are Volume Tiled Resources (VTR). In DirectX 11.2 Microsoft introduced tiled resources who are partially streamed to the GPU. The technique is similar to OpenGL's spares textures and both methods go back to the idea of megatextures, invented by John Carmack [sil13]. Normally these resources are very large, such as a non tileable texture for a whole terrain and only the parts currently visible are getting sent to the GPU. VTR are the 3d pendant of tiled resources and intended for voxels. So the visible parts of a huge voxel-model can now be streamed to the GPU.

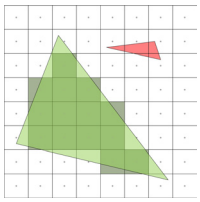


Figure 2: Visualization of the rasterization process with the standard rasterizer.

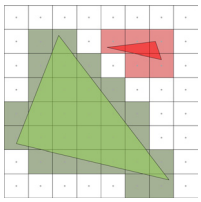


Figure 3: Visualization of the rasterization process with conservative rasterization.

Finally Conservative Rasterization (CR) is a new and more accurate rasterization technique. Normally a pixel belongs to a polygon if its center is inside the polygons borders (figure 2). This leads to inaccuracy if only a small part of a pixel covers the polygon or the polygon is to small to cover one pixel. With CR all four edges of each pixel are checked if they are inside a polygon. If one of the checks is true the pixel is assigned to these polygon (figure 3). This results in a better accuracy for tasks such as collision detection or voxelization. A drawback of this method is the increased performance-consumption because up to four individual checks per pixel have to be made instead of one check as per the standard rasterization.

The reason that the last two major features are designed to improve voxel-operations is that Nvidia recently introduced a new lighting technology for the maxwell-hardwarearchitecture called Voxel Global Illumination (VXGI) [Gmb14]. VXGI converts the currently visible scene to voxels (figure 4) and emits light-rays from the light-sources. For each light-ray up to one reflection on the voxels is calculated which leads to an accurate and relatively cheap lighting simulation (figure 5) [Gmb14]. VXGI is currently one of the most accurate methods to render lighting in real-time.

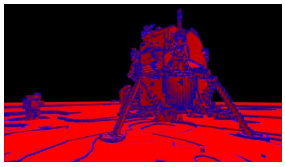


Figure 4: Visualisation of a voxelized scene.

Figure 5: Scene with lighting simulation using VXGI.

3.3. DirectX 11 supporting developers

Because DirectX 11 is Microsoft-exclusive it exclusively runs on Windows versions since Vista, on XBox 360 and XBox One. Most of the current engines, such as Unreal Engine 4, CryEngine, Frostbite 3 or Unity Engine and also most games of the last years are supporting the API [Wik15].

4. AMD's low-level API Mantle

Mantle was one of the first low-level API's available when it was released in late 2013. The idea for the development dates back to 2008 when Johan Andersson, technical director of DICE, which was formerly known for the Battlefield franchise, asked several companies to develop such an API. Since AMD was the only one who agreed, both worked close together while planning and developing Mantle.

4.1. Advantages of low-level API's

Compared to high-level APIs, low-level systems allow a more direct access to the hardware which makes it possible to control the memory usage and the rendering process of the GPU. On the one hand this reduces the CPU overhead and the costs of draw calls but on the other hand the complexity of the code and the effort to create efficient applications grows.
Nowadays programming close to the hardware is the standard for console games development in order to create nice looking graphics with the limited power of relatively old hardware [Dem13]. This is possible because the hardware of a specific console always stays the same. However on PC the direct hardware access is often limited by the operation system because of the hardware abstraction which is a result of the number of different hardware combinations. Low-level API's such as Mantle give developers the possibility to work close to the hardware also on PC.

4.2. Features of Mantle

AMD's goals were to improve GPU-performance, implement a better way of multithreading and to increase the number of draw calls per frame up to 100.000 [Rig14]. Another important point affects the portability of applications made with Mantle. To achieve this AMD ensured to develop Mantle platform-independent and support Intel and Nvidia hardware, if they would release proper drivers. To give developers even more appeal, Mantle also supports shaders written in HLSL, which is one of the most common shader-languages. The outcome of this is that the port of a DirectX or OpenGL game requires very little effort. As a result Mantle was received very well and after the announcement about 40 developer studios confirmed to support the new API [TG14].
Mantle achieves its increase in performance by several improvements compared to DirectX or OpenGL. One is to give the application access to nearly the whole GPU memory without any other software layer in between. This leads to the possibility of controlling the use of the memory without restrictions and consequentially reducing the memory tracking.
Furthermore a new resource system with only memory and images instead of the common and more specific objects such as index buffers, vertex buffers, constant buffers or texture arrays results in a simplified management of resources.
Mantle also supports prebuild data structures which lead to shorter loading times and better efficiency due to the fact that the GPU can just use them instead of compiling them each time they are needed.

4.3. Mantle-supporting developers and Mantles future

Because of the partnership with DICE, their in-house game engine Frostbite 3 was one of the first supporting the Mantle API [Fis13], [Sau14a]. Nowadays also Epics Unreal Engine 3, Cryteks Cryengine and several other companies such as Oxide Games with their Nitrous Engine support Mantle.
Although Mantle was a success with some AAA-games in development, AMDs vice president Raja Koduri announced in March 2015 that the development of Mantle will be stopped [Dav15]. Developers already working on Mantle-applications will still get support by AMD but he recommends to use the upcoming DirectX 12 or the OpenGL successor Vulkan instead. The reasons for this decision were not published but Koduri states that the Mantle API will be the base for the development of Vulkan and DirectX 12 in which AMD is closely involved.

5. Microsoft's low-level API DirectX 12

DirectX 12 was announced on the Game Developers Conference in March 2014 and will ship together with Windows 10 around summer 2015 [Col14]. It will be available on PC's and phones running with Windows and the XBox One.
Currently the specifications are already set up and developers are able to start using DirectX 12, so that the first applications are expected at the end of 2015.

Compared to DirectX 11 there are a lot of changes and improvements. The major difference is that DirectX 12 is like Mantle also a low-level API and allows more direct access to the hardware. In contrast to earlier DirectX releases the main focus of DirectX 12 is not on new graphic effects such as the tessellation-support in DirectX 11 but on an increase of performance or a decrease in power consumption [Yeu14]. DirectX 12 introduces also new rendering features which were already described in section 3.2, since they will also be available for DirectX 11.3.

5.1. Low-level features in DirectX 12

DirectX 12 achieves its improvements in performance with some new features such as Pipeline State Objects (PSO) which improves the rendering pipeline to be a closer representation of the underlying hardware. In DirectX 11 the order of the pipline stages is fixed and each stage has to finish its execution until the next stage can start. As a result the DirectX 11 pipeline is a relatively abstract representation of the hardware which causes some overhead.
PSOs allow to define and combine pipeline stages in one object which is unchangeable at runtime and can get directly translated to hardware instructions. On runtime it is possible to dynamically change between PSOs with very little effort, as they only need to get copied instead of getting recalculated.
Another feature are the command lists that improve the multithreading compared to DirectX 11 and therefore replace the old system. Command lists simply contain all instructions for one task on the GPU. The list defines which PSO to use, which textures and buffers are needed and also some general draw call arguments. With the use of multithreading these lists are pre-calculated and then sent serially to the GPU.
Descriptor heap and descriptor tables replace the resource binding model of DirectX 11 where resources are bound to fixed slots of the pipeline stages. If more resources are needed they have to get rebounded and subsequently a new draw call has to be done.
In DirectX 12 all resources are stored together in the descriptor heap instead of in individual objects. The descriptor tables represent subsets of the whole heap and define which resources are needed for one draw call. Like the PSOs the descriptor tables can also get switched with little effort.

5.2. DirectX 12-supporting developers

Since DirectX became one of the leading APIs in computer graphics over the last years, all major developer studios and their game engines are going to support DirectX 12 [Cal15][J K15][Bur15][Rom15]. For example the Unreal Engine, the Cryengine, or the Unity Engine support DirectX 12, and even a few DirectX 12-games, such as Fable Legends are already confirmed.
A drawback is that DirectX 12 will be exclusive for Windows 10, which leaves out Apple-, Android- and Linux-systems, the Playstation 4 or the Nintendo Wii U.

6. Apple's low-level API Metal

Metal was announced by Apple during the WWDC 2014 as a competitor to OpenGL ES [Era14]. Currently OpenGL ES is the standard for mobile graphics on iOS and Android, but with Metal, Apple tries to establish its own API.
About nine months before Apple introduced the new iPhones 5s and 5c. One of the main differences between them is that the iPhone 5s has a 64 bit CPU while the 5c has just a 32 bit version. However the benefit in performance is relatively small and the iPhone 5s was a lot more expensive than the 5c. Nevertheless more 5s were sold resulting that nowadays a lot of people are already using a 64 bit CPU.
Apple used this with the exclusive release of Metal for just the 64 bit CPU A7 and its successors. That means Metal is only available on iPhone 5s, the iPhone 6 generation and other mobile devices using iOS 8 and an A7, A8 or A8X CPU. On the WWDC15 Apple announced to release Metal also on OS X El Capitan [Bro15].
The name "Metal" should underline that its a low-level API where the developers can work on the lowest possible level at the bare metal of the hardware.

6.1. Features of Metal

Like in DirectX 12 and Mantle, Metal's main goals are to reduce CPU overhead and increase performance. Since it is a mobile API, it should also reduce the power consumption and not heat up the hardware too much. Furthermore Metal should achieve efficient multithreading and support complex non-graphic calculations on the GPU.
There are four main concepts Metal is based on where the first is called low-overhead interface. Metal uses interfaces to represent a single GPU which gives developers complete control over the asynchronous behaviour of the GPU and allow efficient multithreading. This reduces performance bottlenecks such as implicit state validation. the second concept affects the memory- and resource management, where the number of different resource types is reduced to just buffer and texture objects. Buffers can store data such as shader or vertex data while textures store images and some additional information like the pixel format. This concept is similar to the resource concepts of Mantle and DirectX 12 and leads to a simplified and more general resource-management.
The support of both, graphic- and compute-operations is another concept of Metal. These operations are implemented using the same data structures which allows the possibility to share resources between shaders, compute operations and runtime interface. Additionally Metal introduces the new Metal-Shading Language which is based on C++11 to write efficient shaders.
To achieve faster loading times, Metal allows to precompile shaders during the build process of an application. On runtime these shaders are just copied to the GPU which is very fast. Furthermore this method allows better code generation and easier debugging of the shaders [Inc15b].
The main characteristic of Metal is that it is completely based on type independent interfaces which can be assembled to command queues. The command queues define the instructions and the order of execution for the GPU.

6.2. Metal supporting developers

The resonance from the industry concerning the announcement of Metal was very positive [Ste14] and for example Epics Unreal Engine 4 already supports the API. Furthermore the Frostbite 3 from DICE, the Cryengine from Crytek and Unity Technologies Unity Engine also support Metal [Ste14]. There are also already some games using Metal, such as Vainglory or Asphalt 8, in the App-store available.
One major drawback for Metal might be that it is not just Apple exclusive but also 64 bit CPU exclusive which limits the target group in a so far not to be underestimated way.

7. Evaluation

Since low-level API's are still relatively new on the market it is difficult to evaluate the benefits and drawbacks compared to high-level versions under realistic conditions. Only few applications are by now available and most of them use Mantle since it was the first low-level API on the market. For DirectX 12 and Metal there are up to now no meaningful tests besides the official marketing material from Microsoft respectively Apple and their partners. Thus statements such as a CPU overhead of only 1.6 % with Metal compared to 30 % [Sau14b] with OpenGL ES represent no real life conditions.
Most of the available benchmarks and technic demos show a really significant increase in draw calls compared to DirectX 11 or OpenGL. For example figure 6 shows the API Overhead Test from 3D Mark where a raise of draw calls by a factor around 20 between DirectX 11 and DirectX 12 is visible, while the difference between Mantle and DirectX 12 is about 10%. The reason Mantle is listed in only one of the three tests is that it is only available on AMD hardware. Similar results shows the GFX Bench Metal test from Kishonti (figure 7) where Metal accomplishes about 3.5 times more draw calls than OpenGL ES.

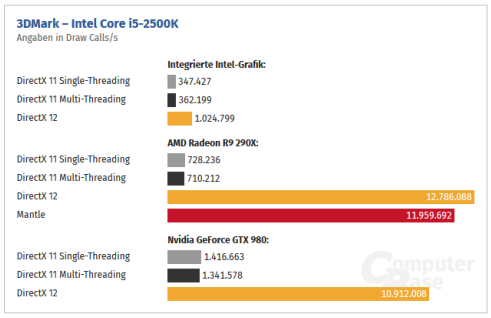


Figure 6: Performance-comparison between DirectX 11, DirectX 12 and Mantle with 3DMarks API Overhead Benchmark.

This looks like being a huge benefit for low-level API's and justify the additional effort in development but these tests just show the ideal case for low-level API's where only draw calls are important. For example the 3D Mark test uses very simple geometry (cubes) and no complex graphic effects but one draw call for every single cube to draw thousands of them. GFX Bench Metal uses a similar technique to show the reduction of CPU-overhead and the resulting increase in draw calls.

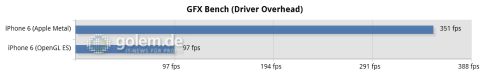


Figure 7: Result of the GFX Bench Metal - driver overhead test.

However drawcalls are only one single part in a complex graphic application since there are also other operations such as shaders and lighting calculations. A more appropriate benchmark is the "Manhattan"-test which is also a part of GFX Bench Metal. Here the hardware has to calculate a lot of graphic effects, animations and lights which uses nearly the full capacity of the GPU with both API's, Metal and OpenGL ES. As a result the difference between both API's is relatively small (figure 8). Because of various optimizations in the API, Metal is still a bit more performant but the difference to OpenGL ES is only 1.5 frames per second.

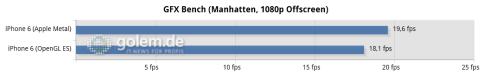


Figure 8: Result of the GFX Bench Metal - Manhattan test.

The same holds for the technic demos that were showed at the persentations of the various API's. For example Intel presented a demo called "Asteroids" (figure 9) where 50.000 individual asteroids are rendered by either DirectX 11 or DirectX 12. On the one hand this demo shows an increase in the frame rate by about 50% or on the other hand a reduction in power consumption around 50% if the frame rate is fixed.

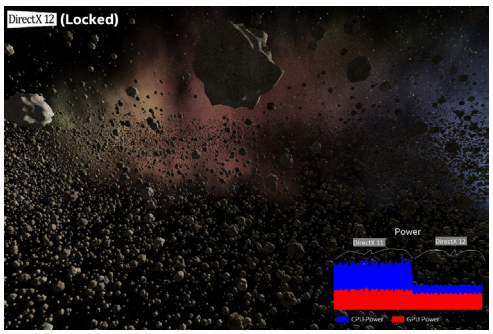


Figure 9: Comparison of power consumption between DirectX 11 and DirectX 12 with fixed frames per second.

However Epics Unreal Engine 4 techdemo "Zen Garden", which demonstrated the possibilities of Metal, shows besides the use of more draw calls also some graphic effects and is therefore a bit more representative for real applications. Though this demo does not show details such as frame rate or power consumption.
The demo uses 4x MSAA and has an average of 4.000 draw

calls per frame. The user can interact directly with about 5.000 leaves of a tree which are calculated physically correct, while in another scene around 3.500 individual butterflies are calculated.
Besides the increase in draw calls some optimizations of the low-level API's also increase performance. One example is the concept of command lists in DirectX 12 which supports better multithreading. The resulting increase in performance can be seen in figure 10 where the top part shows a task calculated in DirectX 11 while the bottom part represents DirectX 12. However this is again an official graphic published by Microsoft and thus might not represent a real use case.

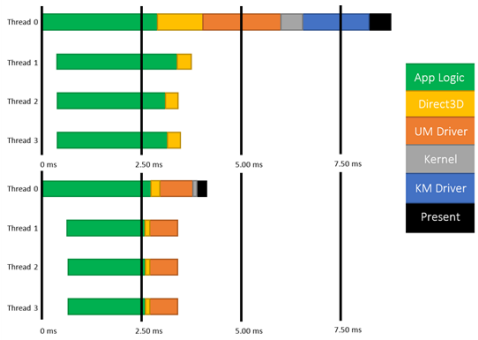


Figure 10: Multithreading in DirectX 11 (top) compared to multithreading in DirectX 12 (bottom). With DirectX 12 the orange parts are even distributed over all available threads.

8. Conclusion

Low-level API's have one huge benefit compared to high-level API's since they reduce the CPU overhead and thus allow more draw calls. However this positive effect comes with a lot more effort and responsibility for the developers. Since low-level API's give developers full control over memory management the complexity of low-level applications increases significantly.
Therefore developers should be aware that such applications will need more time, man-power and knowledge during the development. So if an application does not need to be such performant and optimized, a high-level API like OpenGL or DirectX 11 might be the better choice. Thus high-level API's will remain besides low-level versions as more general and lightweight alternatives.
The choice whether to use DirectX 12, Metal or the OpenGL successor Vulkan, mainly depends on the platforms the application should be available on.
The features of all low-level API's described in this paper are mostly similar and differ just a bit in the reduction of CPU overhead.
One huge drawback is that at the moment there is no platform-independent low-level API available, since DirectX 12 will be Windows 10 exclusive and Metal is iOS respectively OS X exclusive. The only multiplatform low-level API by now is Mantle, which should not be used by developers anymore, since AMD now focuses on DirectX 12 and Vulkan. Therefore Vulkan is likely to become the first successful platform-independent low-level API.

References

[And15] Wolfgang Andermahr. *DirectX 12 im Test vor Mantle und AMD vor Nvidia*. 2015. URL: http://www.computerbase.de/2015-03/directx-12-im-test-vor-mantle-und-amd-vor-nvidia/ (visited on 05/30/2015).

[Bro15] Mitchel Broussard. *Apple Announces Metal for OS X El Capitan*. 2015. URL: http://www.macrumors.com/2015/06/08/apple-announces-metal-for-os-x/ (visited on 07/09/2015).

[Bur15] Brian Burke. *CryEngine Gets Its First DirectX 12 Game at BUILD 2015*. 2015. URL: http://blogs.nvidia.com/blog/2015/05/01/directx-12-cryengine/ (visited on 07/09/2015).

[Cal15] John Callaham. *Unity Engine will target PC games first for supporting DirectX 12 and Windows 10*. 2015. URL: http://www.windowscentral.com/unity-engine-will-target-pc-games-first-supporting-directx-12-and-windows-10 (visited on 07/09/2015).

[Che15] Edward Chester. *DirectX 12 vs DirectX 11: What's New?* 2015. URL: http://www.trustedreviews.com/opinions/directx-12-vs-directx-11-what-s-new (visited on 05/30/2015).

[Col14] Olin Coles. *GDC14: Microsoft DirectX 12 Debuts*. 2014. URL: http://benchmarkreviews.com/14098/gdc14-microsoft-directx-12-debuts/ (visited on 07/09/2015).

[Dav15] Samantha Davis. *On APIs and the future of Mantle*. 2015. URL: https://community.amd.com/community/gaming/blog/2015/05/12/on-apis-and-the-future-of-mantle (visited on 07/09/2015).

[Dem13] Charlie Demerjian. *AMD's Mantle is the biggest change to gaming in a decade*. 2013. URL: http://semiaccurate.com/2013/09/30/amds-mantle-biggest-change-gaming-decade/ (visited on 05/30/2015).

[Era14] Daniel Eran Dilger. *Inside Metal: How Apple plans to unlock the secret graphics performance of the A7 chip*. 2014. URL: http://appleinsider.com/articles/14/06/16/inside-metal-how-apple-plans-to-unlock-the-secret-graphics-performance-the-a7-chip (visited on 05/30/2015).

[Fis13] Martin Fischer. *APU13: Mantle soll schnellste 3D-Schnittstelle werden*. 2013. URL: http://www.heise.de/newsticker/meldung/APU13-Mantle-soll-schnellste-3D-Schnittstelle-werden-2044480.html (visited on 05/30/2015).

[Fut15] Futuremark. *API Overhead feature test*. 2015. URL: http://www.futuremark.com/benchmarks/3dmark#api-overhead (visited on 05/30/2015).

[Gmb14] NVIDIA GmbH. *VXGI Technologie*. 2014. URL: http://www.nvidia.de/object/vxgi-technology-de.html (visited on 07/09/2015).

[Inc15a] Apple Inc. *About Metal and This Guide*. 2015. URL: https://developer.apple.com/library/ios/documentation/Miscellaneous/Conceptual/MetalProgrammingGuide/Introduction/Introduction.html (visited on 05/30/2015).

[Inc15b] Apple Inc. *Fundamental Metal Concepts*. 2015. URL: https://developer.apple.com/library/ios/documentation/Miscellaneous/Conceptual/MetalProgrammingGuide/Device/Device.html (visited on 07/09/2015).

[J K15] George J. King. *DirectX 12: Maxwell-Techdemo mit der Unreal Engine 4 von Microsoft & Lionhead*. 2015. URL: http://www.pcgameshardware.de/DirectX-12-Software-255525/News/DirectX-12-Maxwell-Techdemo-Unreal-Engine-4-Microsoft-Lionhead-1136525/ (visited on 07/09/2015).

[Lau14] Andrew Lauritzen. *SIGGRAPH 2014: DirectX 12 on Intel*. 2014. URL: https://software.intel.com/en-us/blogs/2014/08/11/siggraph-2014-directx-12-on-intel (visited on 05/30/2015).

[Moo15] Warren Moore. *An Introduction to 3D Graphics with Metal in Swift*. 2015. URL: https://realm.io/news/3d-graphics-metal-swift/ (visited on 05/30/2015).

[Oos14] Jeremiah van Oosten. *Introduction to DirectX 11*. 2014. URL: http://www.3dgep.com/introduction-to-directx-11/ (visited on 05/30/2015).

[Rig14] Guennadi Riguer. *Mantle - Introducing a new API for Graphics - AMD at GDC14*. 2014. URL: http://de.slideshare.net/DevCentralAMD/mantle-introducing-a-new-api-for-graphics-amd-at-gdc14 (visited on 05/30/2015).

[Rom15] Sal Romano. *Square Enix showcases Witch Chapter 0 [Cry] Luminous Engine DirectX 12 tech demo*. 2015. URL: http://gematsu.com/2015/04/square-enix-witch-chapter-0-cry-luminous-engine-directx-12-tech-demo (visited on 07/09/2015).

[Sau14a] Marc Sauter. *AMDs Mantle-API im Test: Der Prozessor-Katalysator*. 2014. URL: http://www.golem.de/news/amds-mantle-api-im-test-der-prozessor-katalysator-1402-104261.html (visited on 05/30/2015).

[Sau14b] Marc Sauter. *Apple-Metal-API hat nur 1,6 Prozent CPU-Overhead*. 2014. URL: http://www.golem.de/news/unreal-engine-4-apple-metal-api-hat-nur-1-6-prozent-cpu-overhead-1408-108520.html (visited on 05/30/2015).

[sil13] silverspaceship. *Sparse Virtual Textures*. 2013. URL: http://silverspaceship.com/src/svt/ (visited on 07/09/2015).

[Smi14] Ryan Smith. *Microsoft Details Direct3D 11.3 & 12 New Rendering Features*. 2014. URL: http://www.anandtech.com/show/8544/microsoft-details-direct3d-113-12-new-features (visited on 06/09/2015).

[Ste08] Peter Steinlechner. *Microsoft bringt DirectX 11 und renoviert Spieledienste (U.)* 2008. URL: http://www.golem.de/0807/61251.html (visited on 07/09/2015).

[Ste14] Peter Steinlechner. *Entwicklerlob für 3D-Schnittstelle Metal*. 2014. URL: http://www.golem.de/news/apple-entwicklerlob-fuer-3d-schnittstelle-metal-1406-106902.html (visited on 05/30/2015).

[TG14] Tactical Gaming (TG). *Mantle – AMD Says 40 Development Studios Are On Board*. 2014. URL: http://www.tacticalgaming.net/hq/news/2014/rmantle-amd-says-40-development-studios-board (visited on 07/09/2015).

[Wik15] Wikipedia. *List of games with DirectX 11 support*. 2015. URL: https://en.wikipedia.org/wiki/List_of_games_with_DirectX_11_support (visited on 07/09/2015).

[Yeu14] Andrew Yeung. *DirectX 12 - High Performance and High Power Savings*. 2014. URL: http://blogs.msdn.com/b/directx/archive/2014/08/13/directx-12-high-performance-and-high-power-savings.aspx (visited on 07/09/2015).

Hauptseminar: Review of the advancing progress of Vulkan developement

R.Ledermueller[1,2] [1]TU Dresden Germany

[2]Institut for Software and Multimedia Technics, Computer Graphics and Visualization

Abstract

"Vulkan is the new generation, open standard API for high-efficiency access to graphics and compute on modern GPUs." [3] The new design allows the use of muliple GPUs, controlled by multiple threads with low overhead. The layered architecture minimize error handling in release mode and the better memory accessibility allows a more efficient usage.

Categories and Subject Descriptors (according to ACM CCS):
C.0 [Computer Systems Organization]: Hardware/software interfaces—Vulkan

1. Introduction

After 20 years evolving OpenGL the Khronos Group started the discussion for Vulkan in June 2014. The architectures of CPUs and GPUs changes a lot and way of programming too. Vulkan is designed for modern architectures and offers high potential for developers. It is vendor and system independent. The paper explain 5 major improvements targeted in Vulkan. The sections Instance and Device and Pipeline characterize the new handling of hardware capabilities. Memory and Queues, Buffers and Commands describe the resource and command management. Vulkan separates debugging and release mode. Layers improve the administration of debug information. To reduce the overhead once more Vulkan skip the parsing of shader code. Instead it uses a 32bit stream called SPIR-V.
Unfortunately there is no specification out yet, so the given API-calls and attributes can change. This paper gives an overview of one year developing Vulkan.

2. Improvements

2.1. Instance and Device

An application can create any amount of Vulkan-Instances. An instance create the "Loader"-Layer, which gives debug information about memory allocation. The developers highly recommend to use this layer.

```
vkCreateInstance(&appInfo, ..., &instance);
vkEnumerateGpus(instance, ...,
&gpuCount, &gpus);
vkGetGpuInfo(gpu[0], ..., &infoSize, &info);
vkGetMultiGpuCompatibility(gpuA, gpuB, ...);
```

The code shows a typical Vulkan initialization. In many cases there are more than one resources to process graphic output. Vulkan will distribute functions look for devices, configure them and create a handle for later use like in the presented code. In the best case a GPU-Performance of two chip-producers can added linearly.

2.2. Queues, Buffers and Commands

In 2015 a desktop CPU has eight or more cores. To use all of them effective, the programmer builds command-buffers of the traditional OpenGL-calls. Once finished building a

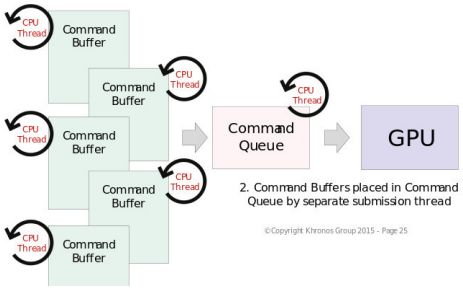


Figure 1: *Principle of using the command buffers. [4]*

buffer the application adds the buffer to a queue. It is possible to build multiple queues and switch between the queues. Manipulating the active queue is another way but the driver didn't do any synchronization. For example a queue generates a texture and overwrites an image. The Commands are changed to read the texture. The result is a model with a half-generated surface.

2.2.1. Code example

Like in figure 1 a thread creates the buffer first.

```
vkCreateCommandBuffer(device, &info, &cmdBuf);
```

Vulkan instructions have a similar semantic. The API needs information to create objects. Some come from previously defined Vulkan-objects (the device in this case). Other data is defined directly by the programmer. The info variable references in this context to a CreateCommandBufferInfo structure. The layout of these info-structures are not final yet. For a CommandBuffer it can be the type of optimization, the maximum command-count or a callback function for error handling. A DeviceCreateInfo structure has a defined count of queues, extension information and the state of the validation-laver for this device. The cmdBuf-variable is the handle of the created command buffer. With the handle commands can pushed in the buffer. The following example shows a simple render stage with a given Pipeline-object and Memory-layout.

```
vkCmdBeginRenderPass(cmdBuffer, &beginInfo);
vkCmdBindPipeline(cmdBuffer, ...);
vkCmdBindDescriptorSets(cmdBuffer, ...);
vkCmdDraw(cmdBuffer, ...);
vkCmdEndRenderPass(cmdBuffer, ...);
```

The GPU can be configured with just two commands and enables more draw-calls [1]. There will be many more command-functions. Finally the buffer has to be added to a queue and that queue must be activated.

```
vkQueueSubmit(queue, 1, cmdBuffers, ...);
```

Once the queue is submitted and all data is in GPU memory, the application can generate draws with low CPU overhead(see 4).

2.3. Pipeline

There is no global pipeline in Vulkan. A programmer creates a pipeline with multiple shaders.

```
vkCreateShader(device, ...);
```

The command compiles a given shader-code for a device. A Vulkan-driver accept the new SPIR-V language(see 3). With multiple pipelines it is possible to switch fast between them. A pipeline-object is ready to run after creation. So the programmer decides which states are mutable in the CreatePipelineInfo. Other states cannot changed after creation.

```
vkCreateGraphicsPipeline(device, ...);
```

The instruction parameters deliver all the shaders, the states(like blending, culling, depth-testing, etc.) and the used memory and variables via descriptors and descriptor-layout.

2.4. Memory

One goal is the separation of content, memory-layout and references. A memory object has a defined size. Within the object many different information are distributed over the whole size. The layout defines how the allocated size is parted and can accessed through descriptors. A descriptor has to be defined in a descriptor pool first. Then many descriptors can be packed in a descriptor set. So it is possible to swap complete objects very fast with a single command. Every pipeline stage has defined descriptor sets. By switching pipeline-objects descriptor sets used by both pipeline-objects remain. That avoids memory allocation and freeing.

2.5. Layers

Different layers are important to debug different tasks. In Vulkan there are two layers activated in any case. The Application layer is on the top, where the programmer build own debug messages. Then there are several layers which can be activated via Vulkan calls and customized in callback-functions. On the bottom is the driver-layer. It is possible to create additional layers through extensions. The list of layers isn't final yet. All currently available are in figure 2.

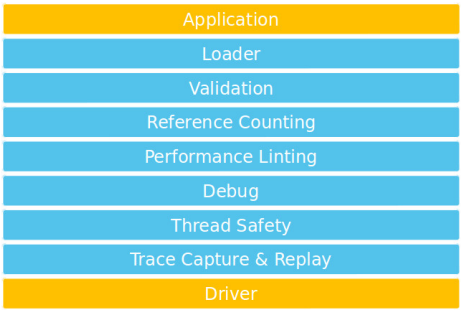


Figure 2: *A list of the actual planned layers.(Valve Vulkan Session) [5]*

2.6. Problems

With great power comes great responsibility. The programmer has to manage all threads, the devices, the memory, the synchronisation and the debug layers. Thats slows down the developing process. To solve these problems Vulkan will come with a SDK and third party tools like the debugging tool GLAVE [10]. Use utility libraries or a Vulkan optimized engine to boost the developement speed. For a better understanding and faster driver delopement Intel will release an open-source drivers [1].

3. SPIR-V

SPIR-V (Standard Portable Intermediate Representation) is a open cross API standard for graphical-shader stages and compute kernels. In OpenGL a shader-code is saved as a string in the executable or in an external file, so both save the plain source code. That code has to be parsed during runtime. With SPIR-V the compilation is reorganized. The parsing is transferred to the development cycle. Every programmer can write their code in the preferred language and compiles the code to SPIR-V. Then the driver has to compile only the SPIR-V code at runtime with less error handling. That keeps the driver simpler and smaller. SPIR-V is high level enough to be platform and system independent. Register allocation and scheduling is done by the driver.

3.1. Structure

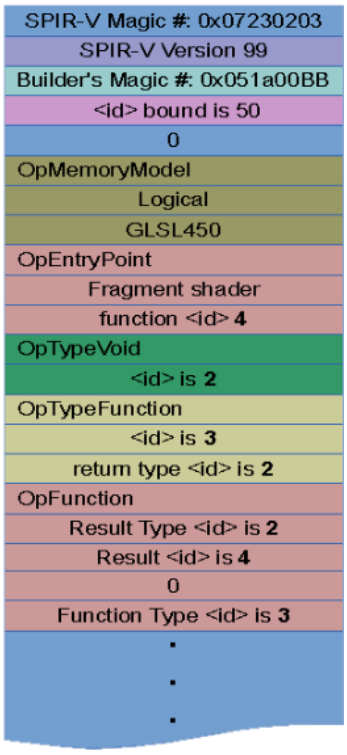


Figure 3: *An example SPIR-V code from Khronos. [9]*

A SPIR-V stream consists one or more modules. A module starts with a magic number(big or little endian format) followed by the SPIR-V Version, the builder-ID, the highest ID and the instruction schema. The main body is followed by the module header and is filled with instructions. Figure 3 shows a module with little endian, Version 0.99 the normal instruction schema(16 bit word count and 16 bit instruction number) and the highest reference is 50. The related C-code can be found in the specification [9]. For debugging the programmer can add decorators to the variables. So ids can bound to strings. This is important to connect the shader-variables with the application. The instructions encode all information to build a syntax tree or basic blocks.

3.2. Conversion example

Thus SPIR-V is a stream the definition hierarchy of the following struct changed. In C-Code the toplevel is defined first and then the tree goes from the top to the bottom.

```
struct {
 mat3x4;
 vec4[6];
 int;
};
```

So the given struct can represented through the tree diagram shown in figure 4. In a stream only defined objects can be

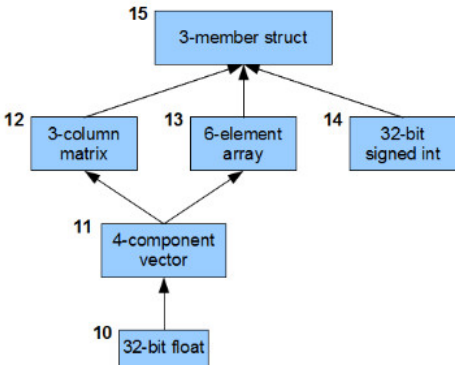


Figure 4: *The struct in a tree diagram. [5]*

used. So the deklaration order changed from bottom to top. The resulting SPIR-V code will be as following:

```
10: OpTypeFloat 32
11: OpTypeVector 10 4
12: OpTypeMatrix 11 3
13: OpTypeArray 11 6
14: OpTypeInt 32 1
15: OpTypeStruct 12 13 14
```

In the specification OpTypeFloat is defined with a length of 3. The first line is the instruction itself (length and instruction number) The second line is defined as result id. In this case it is 10. The last line is for the bit length of the data. With the OpVariable command memory can be allocated. OpVariable has a length of 4 or more. The first is type-id (10 for the float), then resulting id of that variable followed by the storage type (Uniform, Private, Constant, ...). Optionally OpVariable can have initial values. The variable can be manipulated for example with OpStore.

4. Demonstrations

To underline the advancing process there are some demonstrations online yet. Figure 5 and 6 show the big advantage

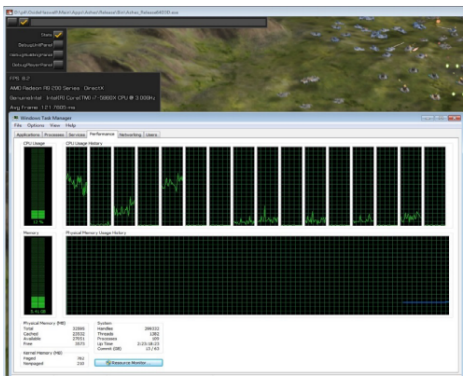


Figure 5: *Ashes of Singularity with OpenGL. [5]*

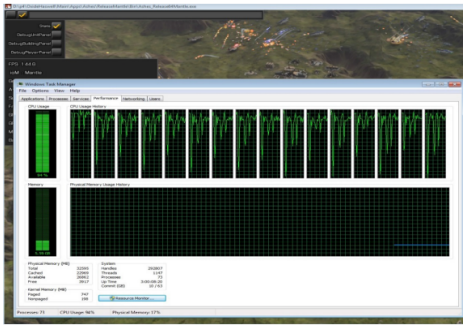


Figure 6: *Ashes of Singularity with Vulkan. [5]*

in Vulkan. Much more Draw-Calls are possible. In OpenGL the CPU is only at 12%. Too much time is waisted waiting for GPU finishing graphics or computing. In Vulkan the CPU can create many CommandBuffers. If the Queue is big enough, the CPU can do some other calulations while the GPU process the Queue.
ARM has written a benchmark to test an Arndale Octa. They used 1000 Meshes and 3 Textures. In figure 7 Vulkan reduces the cycles spent in the driver about 79% [5].

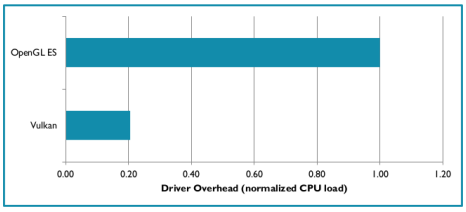


Figure 7: *A microbenchmark from ARM. [5]*

5. Summary

Vulkan offers new features to recognize hardware. It uses command-buffers that are filled in multiple threads. With a new memory management big data can be transferred with a single access. The exchange is controlled by the programmer. So data not used the same time can have the same space. The programmer is responsible for the error handling in different layers. The driver haven't parse source code, cause it reads SPIR-V. Multiple pipelines are useful to change many attributes fast. The resulting new programming overhead can be minimized through many new tools. For scientific simulations Vulkan offers much space for optimizations. Nearly all game-engines have already announced that they will support Vulkan. [1]

References

[1] GDC Vulkan Youtube (30.06.2015)
https://www.youtube.com/watch?v=QF7gENO6CI8
2, 4

[2] webLinkFont Vulkan and SPIR-V session - Youtube (30.06.2015)
https://youtu.be/qKbtrVEhaw8

[3] Vulkan Introduction (30.06.2015)
https://www.khronos.org/vulkan 1

[4] Vulkan @Multicore 2015 (30.06.2015)
https://www.khronos.org/developers/library/2015-multicore 1

[5] Vulkan @GDC 2015 (30.06.2015)
https://www.khronos.org/developers/library/2015-multicore 2, 3, 4

[6] WebGL Specifications (30.06.2015)
https://www.khronos.org/registry/webgl/specs/latest/1.0/

[7] WebGL 2.0 Specifications (30.06.2015)
https://www.khronos.org/registry/webgl/specs/latest/2.0/

[8] WebGL Security Whitepaper 30.06.2015
https://www.khronos.org/webgl/security/

[9] SPIR-V Specifications (30.06.2015)
https://www.khronos.org/registry/spir-v/specs/1.0/SPIRV.html 3

[10] GLAVE Demo YouTube (12.07.2015)
https://www.youtube.com/watch?v=miZmas6sGqM
2