Architektur und Programmierung von Grafik- und Koprozessoren

Die Grafik Pipeline

Stefan Zellmann

Lehrstuhl für Informatik, Universität zu Köln

SS2019

Lernziele

1. **Host Interface** - die Studierenden verstehen das Zusammenspiel zwischen “Kernel Mode” und “User Mode” Gerätetreibern, die mehreren Betriebssystemprozessen den gleichzeitigen Zugriff auf die Grafikkarte erlauben.
2. **Kommandoverarbeitung** - die Studierenden verstehen den Kontrollfluss von Zeichenkommandos, die die Applikation an die Grafikkarte verschickt.
3. **Architekturen** - die Studierenden kennen Architekturprinzipien, die bei modernen Grafikkarten eine Rolle spielen.
4. **Moderne Grafik APIs** - die Vorlesungseinheit bereitet die Studierenden auf moderne Grafik APIs wie DirectX 12 oder Vulkan vor. Die Studierenden verstehen, welche Rolle Nebenläufigkeit und Synchronisation bei der Programmierung moderner Grafikkarten spielen.

Kurze Einführung in Grafik APIs

Immediate Mode vs. Retained Mode APIs

Immediate Mode APIs

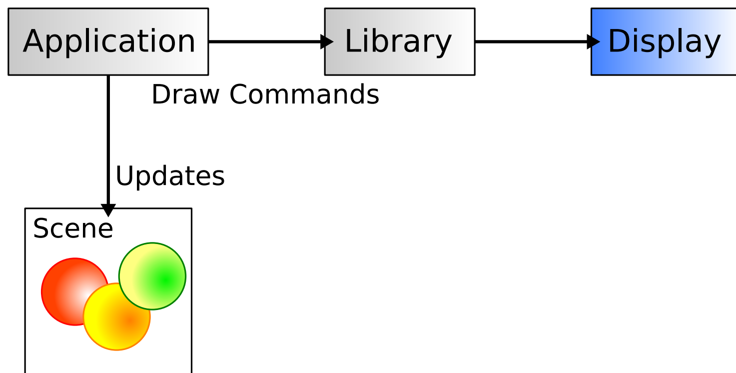


Abbildung: vgl. Windows Dev Center - Windows Graphics: Retained Mode Versus Immediate Mode.

Immediate Mode vs. Retained Mode APIs

Retained Mode APIs

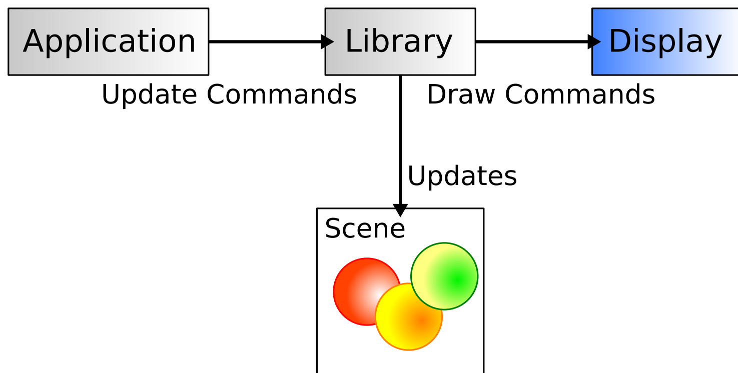


Abbildung: vgl. Windows Dev Center - Windows Graphics: Retained Mode Versus Immediate Mode.

Immediate Mode APIs

3D APIs traditionell *Immediate Mode* - wenn Programm Zeichenbefehl spezifiziert, *sieht es so aus*, als würde der Befehl unmittelbar ausgeführt.

Realität: Treiber *puffert* Kommandos und schickt Batches von Zeichenbefehlen an GPU. Batches beinhalten potentiell Zeichenbefehle von verschiedenen Threads oder sogar verschiedenen Applikationen. Komplizierte Heuristiken, um zu entscheiden, wie gepuffert wird.

Das kann teils zu überraschendem Performanzverhalten führen!

Moderne APIs (Vulkan, D3D12): nicht Immediate, Applikation steuert, wann welche Speicherbefehle *zu submittieren sind*, sowie Abhängigkeiten zwischen Kommandos + Synchronisation $\Rightarrow$ viel weniger komplexe Treiber.

Immediate Mode APIs

- Auf den folgenden Folien: OpenGL-ähnlicher Pseudo Code dient zur Illustration, Programmausschnitte unvollständig.
- Kein OpenGL Tutorial, nur Konzepte.
- Konzepte gelten auch für alte Direct3D Versionen.

Immediate Mode APIs

```
glBegin(GL_TRIANGLES);

    // Red triangle
    glColor3ub(255,0,0);
    glVertex3f(0,0,0);
    glVertex3f(1,0,0);
    glVertex3f(1,1,0);

    // Green triangle
    glColor3ub(0,255,0);
    glVertex3f(0,0,0);
    glVertex3f(1,0,0);
    glVertex3f(1,1,0);

glEnd();
```

Immediate Mode APIs

- Immediate Mode APIs bilden den Zeichenalgorithmus ab, nicht die zugrundeliegende Hardware.
- Hardware: *hochparallel*, für hohen *Durchsatz* optimiert.
  - Besonderes Merkmal von GPUs: hohe Bandbreite, dafür vergleichsweise hohe Latenz: z. B. bei Host-Interconnect (PCIe), aber auch bei GDDR Speicher.
- Immediate Mode Operationen wie die von oben hätten eine unglaublich hohe Latenz $\Rightarrow$ Pufferung nötig.
- Frage: wer puffert Daten?
  - Treiber: Produktivität.
  - Anwendungsentwickler: applikationsspezifisches Wissen $\Rightarrow$ höheres Potential für bessere Performanz.

Immediate Mode APIs

Graphic APIs, egal welche, unterstützen heute zumindest Pufferung von Daten:

```
// Only once:
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER, num_verts, verts,
             GL_STATIC_DRAW);

// Later, when rendering:
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glDrawBuffer(vbo);
```

Anwender hat aber generell wenig Kontrolle, wann Daten kopiert werden, welche Speicherbereiche genutzt werden etc.

GPU Zustandsmaschine

Traditionelle GPU APIs implementieren den Rasterisierungsalgorithmus mittels Zustandsmaschinen:

```
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, tex);
glTexImage2D(...);
glEnable(GL_LIGHT0);
glLightfv(GL_LIGHT0, GL_POSITION, pos);

glMatrixMode(GL_MODELVIEW);
glLoadMatrix(mv);
```

gefolgt von Zeichenbefehlen:

```
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glDrawBuffer(fb);
glXSwapBuffers(); // platform specific!
```

GPU Zustandsmaschine

`glActiveTexture(GL_TEXTURE0)` - aktive Textureinheit ist jetzt '`0`'.

`glEnable(GL_LIGHT0)` - aktive Lichtquelle ist jetzt '`0`'.

`glMatrixMode(GL_MODELVIEW)` - *als nächstes* wird MV Matrix verändert.

`glBindBuffer(GL_ARRAY_BUFFER, vbo)` - der gerade gebundene Buffer ist Vertex-Buffer mit *Handle* '`vbo`'.

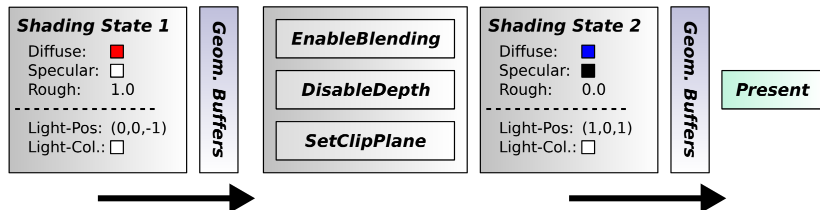
GPU Zustandsmaschine

Programmablauf exemplarisch für (altes) OpenGL:

1. Erzeuge Fenster, erzeuge OpenGL Kontext(e). Binde Kontexte an Threads.
2. Initialisiere Framebuffer, Zeichnen in Backbuffer etc.
3. Starte "Render-Loop":
    - Pushe intrinsische Kameraparameter und globale Kameratransformation auf Matrix Stack.
    - Traversiere Szenengraph (Baumstruktur, die Geometrie hierarchisch mit Transformationen / Materialeigenschaften etc. verknüpft).
        - Pushe individuelle Transformationen auf Matrix Stack.
        - Initialisiere Materialien / Texturen etc. per Geometrie.
        - Submitte Geometrie (Vertex Buffer).
    - Setze Beleuchtungsparameter, Zeichen-State (Depth Buffering, Alpha Blending, Backface Culling, ...)
    - Präsentiere Bild, indem Front- und Backbuffer vertauscht werden.

GPU Zustandsmaschine

Zustand wurde / wird in *Kontexten* gespeichert. Kontext speichert gesamten Zustand zu Zeitpunkt t.



GPU Zustandsmaschine

Kontexte von GPU und API verwaltet, Benutzer kann Zustand durch API Calls verändern.

```
// GPU Kontext (implizit)
{
    blending    : no
    depthtest   : yes
    diffuse     : (255,0,0)
    specular    : (255,255,255)
    rough       : 1.0
    light1      : disabled;
    lightpos1   : (0,0,1)
    ligntcol1   : (255,255,255)
    clipplane1  : no
}

// Programm:
glEnable(GL_LIGHT1);
glLightfv(GL_LIGHT1, GL_POSITION, { 0,0,-1 });
```

GPU Zustandsmaschine

- GPU Kontexte sind sehr schwergewichtig, speichern gesamten Rendering State.
- Frühe API Versionen: nicht CPU Multi-Threading "aware". Ein Kontext pro Thread, kein Sharing, kein Austausch und keine Synchronisation.
- Umschalten zwischen Threads: binde ganz anderen Kontext, unklar, welcher State sich unterscheidet.

Ressourcen Handles

- Ressourcen werden über *Handles* verwaltet. Einfache Datentypen (`GLint`, `GLenum`), teils Objekte mit opaker Struktur (z. B. GLXContext).
- Kein direkter Zugriff auf GPU Speicher, keine Pointer etc.
- Ressourcen sind an Kontexte gebunden, Teilen von Ressourcen durch mehrere Threads oder Prozesse schwergewichtig.

Zeichenbefehle

Der eigentliche Algorithmus steckte in den Zeichenbefehlen und war in den Anfangszeiten von GPUs außer über den *State* nicht beeinflussbar.

```
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glDrawBuffer(fb);
glXSwapBuffers();
```

Zeichenbefehle

```
glFinish();        // or
glFlush();         // or
glXSwapBuffers(); // or ..
```

Verschiedene Kommandos (Zeichnen, State Change, etc.) laufen beim API auf. Es ist dem API / Treiber überlassen, wann die Befehle ausgeführt werden (direkt oder später). Kommandos wie `SwapBuffer()`, `Flush()`, `Finish()` führen dazu, dass die Pipeline stalled und die aufgelaufenen Befehle (in endlicher Zeit, nicht notwendigerweise sofort) abgearbeitet werden müssen.

Shader Programme

Spätere GPUs: Shader Programme, die den Ablauf des Rasterisierungsalgorithmus *auf der GPU selbst* beeinflussten (vgl. Algorithmus RASTERISIERUNG):

```
activate_vertex_buffer(handle)
draw_triangles() {
    for (each vertex)
        vertex_shader();                    /*----*/

    triangle_setup();

    for (each triangle)
        scan_convert();
        for (each fragment)
            for (each light)
                fragment_shader();          /*----*/
            depth_test();
            alpha_blending();
}
present();
```

Shader Programme

- C-artige Programme, die auf GPU ablaufen. Werden an dedizierten Einsprungpunkten in die Grafik Pipeline eingehängt.
- Mit *Vertex Shadern* kann die Vertex Transformationsphase (Model-View und Perspektivische Transformation, Clipping, Transformation in normalisierte Gerätekoordinaten) *modifiziert* werden.
- Geometrie Shader schließen sich an Vertex Phase an. Programm erhält ein Vertex als Input und emittiert weitere Vertices.
- Mit Fragment Shadern kann die Fragment Phase modifiziert werden, im Rahmen derer die Fragment Farbe bestimmt wird.

Shader Programme

Nomenklatur

Bezeichnungen weichen je nach API und Konvention etwas ab.
Manchmal werden die Konzepte *Shader* und *Programm* vermischt.
D3D nennt Fragment Shader *Pixel Shader*.

Wir verwenden die *OpenGL* Konvention: *Vertex Shader*, *Fragment Shader* und ggf. weitere bilden zusammen ein *Shader Programm*.

Shader Programme

Shading Languages

- Es existieren eine Reihe:
  - GLSL ("GLSlang") - Teil des OpenGL Standards.
  - HLSL - Microsoft DirectX Shading Language.
  - Nvidia Cg - identisch mit HLSL, jedoch kompatibel mit OpenGL.
  - API-neutrale Sprachen, z. B. OpenShadingLanguage; Sprachen für spezielle APIs wie z. B. Pixars RenderMan.
- `GL_ARB_fragment_program` (OpenGL 1.3) - erste OpenGL Spezifikation für programmierbare Fragment Stage, Assembler-artig, kein Branching etc.

Shader Programme

- Shader Programme werden hochparallel ausgeführt.
  - Implizites paralleles Programmiermodell.
  - Keinerlei Zugriff auf andere Shader Instanzen.
- Vulkan / Modernes OpenGL: minimaler Vertex Shader verpflichtend (außer Compute). Ohne minimalen Fragment Shader kein Bild (manchmal Ergebnis kein Bild).

Vertex Shader

Einfacher Vertex Shader (**lose** basierend auf *OpenGL Shading Language* (GLSL)) . Applikation verwaltet Viewing Transformationsmatrix und perspektivische Transformationsmatrix und übergibt sie an alle *Vertex Shader Instanzen*.

```
attribute vec3 position;
in mat4 view_mat;
in mat4 proj_mat;

void main() {
  gl_Position = view_mat * proj_mat * vec4(position, 1.0);
}
```

Vertex Shader

Benutzerspezifische globale Attribute:

```
attribute vec3 position;
in mat4 view_mat; // global
in mat4 proj_mat; // global

in float time;    // global

void main() {
  position.x += cos(time);
  position.y += sin(time);
  gl_Position = view_mat * proj_mat * vec4(position, 1.0);
}
```

Vertex Shader

Benutzerdefinierte *per Vertex Attribute*:

```
attribute vec3 position; // per vertex
in mat4 view_mat;
in mat4 proj_mat;

attribute vec3 normal;   // per vertex

void main() {
    ...
    gl_Normal = transform(normal, view_mat, proj_mat);
    ...
}
```

Vertex Shader

Vertex Output geht später durch Raster Engines ⇒ **benutzerdefinierte Attribute werden während Scan Konvertierung interpoliert!**

`varying`: interpolierte Attribute stehen später in Fragment Phase zur Verfügung.

```
attribute vec3 position;
in mat4 view_mat;
in mat4 proj_mat;

varying vec3 normal;
varying vec2 uv;

void main() {
    ...
}
```

Fragment Shader

Einfacher Fragment Shader mit diffusem Beleuchtungsmodell. varying Attribute aus Vertex Shader. Fragment-spezifischer Input (z. B. light_dir).

```
// Simple diffuse shader
in sampler2D tex;
in vec3 light_dir;
varying vec3 normal;
varying vec2 uv;
out vec4 frag_color;

void main() {
  // 2-D texture access
  vec3 diff = tex2D(tex, uv);

  // Simple (single-sided) Lambert shading
  diff *= max(0.0, dot(light_dir, normal));

  // Write result
  frag_color = vec4(diff, 1.0);
}
```

Shader Programme

- Neue APIs: mehr programmierbare Pipeline Stages.
  - Geometrie Shader: folgen auf Vertex Stage. Programmatisch Erzeugung weiterer Polygone.
  - Tesselation Shader: niedrig aufgelöste Geometrie kann durch *Tesselierung* verfeinert werden, z. B. mit Bezier Patches (*Subdivision Surface* Algorithmen).
  - Weitere programmierbare Stages z. B. im Zusammenhang mit Multisampling.
- Wir gehen vereinfachend nur von Vertex und Fragment Shadern aus.

Shader Programme

Shader werden *zur Programmlaufzeit* kompiliert und gelinkt.

```
GLuint vs = glCreateShader(GL_VERTEX_SHADER);
GLuint fs = glCreateShader(GL_FRAGMENT_SHADER);

glShaderSource(vs, strlen(vert_shader_str),
               vert_shader_str, 0);
glCompileShader(vs);
glShaderSource(fs, strlen(frag_shader_str),
               frag_shader_str, 0);
glCompileShader(fs);

GLuint prog = glCreateProgram();
glAttachShader(prog, vs);
glAttachShader(prog, fs);

glLinkProgram(prog);

glUseProgram(prog);
```

Shader Programme

- Laufzeitkompilation und Linking.
  - Laufzeit Overhead (wegen Caching meistens nur erster Programmlauf).
  - OpenGL Runtime muss Compiler integrieren.
  - Traditionell sehr fehleranfällig - IHVs (Independent Hardware Vendors) legen Sprachstandard unterschiedlich aus.
  - Treiber muss Compiler-generierten Intermediärcode in Maschinencode übersetzen.
  - Shader Programme stehen im Klartext im Binary, können einfach mit Hex-Editor ausgelesen werden.
- Vulkan API: Shader werden zur Compile Zeit in optimierten Intermediärcode (SPIR-V) übersetzt.

Hierarchische Pipeline

Alte Versionen von Direct3D und OpenGL hatten hierarchische Pipelines:

- Stack für Transformationsmatrizen.
- Stack für generellen State.
- Wende Transformationen / Materialien etc. auf Teile der Geometrie an, indem State / Transformationen auf Stack `gepush()`ed werden. Wenn entsprechende Teile der Geometrie gezeichnet, `pop()` vom Stack.

Hierarchische Pipeline

Neue APIs gehen davon aus, dass die Applikation den hierarchischen State samt Transformationen verwaltet (z. B. Szenengraph Bibliothek), und dass Transformationen als uniforme Variablen an *Shader Instanzen* übergeben werden.

Kompletter Matrix Stack in OpenGL deprecated. Viele Betriebssystemhersteller (insb. Microsoft, Apple) unterstützen nur sehr alte OpenGL Versionen, daher noch viel legacy Code, der auf alten OpenGL Konzepten aufbaut.