Computergraphik II

Übung 04

OpenGL Teil 4: Shader Compositor

在本课中，我们将不再关注OpenGL特定问题，而是围绕OpenGL软件设计中经常遇到的问题：着色器。
In dieser Einheit werden wir uns weniger mit OpenGL spezifischen Problemen beschäftigen, sondern mit einem Problem, welchem man im Softwaredesign um OpenGL häufig begegnet: Die Shader.
要提出问题的解决方案，我们首先需要了解问题。 到目前为止，我们在练习中使用了两个着色器程序，一个用于表示Sponza场景中的对象，另一个用于渲染天空。
Um die Lösung des Problems vor zu stellen, müssen wir zuerst das Problem verstehen. Bisher haben wir in der Übung genau zwei Shaderprogramme verwendet, eines um die Objekte der Sponza-Szene darzustellen, und eines um den Himmel zu rendern.

Stellen Sie sich folgende Aufgabe vor: Rendern Sie ein weiteres Objekt, für welches keine Textur vorliegt, in den Innenhof der Szene. 想象一下以下任务：将没有纹理的另一个对象渲染到场景的庭院中。
我们之前的程序期望纹理绑定到反照率值。 我们怎么能解决这个问题呢？ 直觉上，有几种方法：
Unser bisheriges Programm erwartet, dass eine Textur mit den Albedowerten gebunden ist. Wie könnten wir dieses Problem lösen? Intuitiv gibt es mehrere Ansätze:

我们可以创建一个1X1像素纹理，我们在其中配置颜色。
1. Wir könnten eine 1×1-Pixel Textur erstellen, in der wir die Farbe konfigurieren.
也许我们可以通过GLSL函数测试纹理是否绑定？
2. Wir könnten vielleicht über eine GLSL-Funktion testen, ob eine Textur gebunden ist?
我们可以编写制服，无论是使用纹理还是均匀颜色。
3. Wir könnten in einem Uniform kodieren, ob die Textur oder eine konstante Farbe aus einem Uniform verwendet werden soll.
我们可以编写另一个没有纹理的着色器，并使用它来渲染无纹理的对象。
4. Wir könnten einen weiteren Shader schreiben, der ohne Textur auskommt, und diesen verwenden um untexturierte Objekte darzustellen.

Alle diese Möglichkeiten funktionieren im Prinzip, sind aber aus verschiedenen Gründen mehr oder weniger elegant, beziehungsweise praktikabel: 所有这些选项原则上都有效，但出于各种原因，或多或少优雅或实用：
纹理需要的不仅仅是颜色信息。 我们还需要纹理坐标。 此外，必须在OpenGL端组织纹理访问，纹理访问适用于更昂贵的操作。
1. Mit einer Textur wird mehr als nur die Farbinformation benötigt. Wir benötigen außerdem Texturkoordinaten. Außerdem muss auf OpenGL-Seite der Texturzugriff organisiert werden, Texturzugriffe gelten zu den teureren Operationen.
实际上，由于GLSL 430（即OpenGL 4.3）确实有一个函数纹理QueryLevels，如果没有绑定纹理，它应返回零。 在实践中，不支持此功能（Nvidia为未绑定纹理返回非零值）。
2. Es gibt seit GLSL 430 (also OpenGL 4.3) tatsächlich eine Funktion `textureQueryLevels`, welche Null zurück liefern soll, wenn keine Textur gebunden ist. In der Praxis ist diese Funktion nicht sonderlich gut unterstützt (Nvidia gibt von Null verschiedene Werte bei ungebundener Textur).
到目前为止，这种变体原则上仍然是最好的。 是否应使用纹理的测试会对性能产生负面影响。
3. Diese Variante ist prinzipiell noch die beste bisher. Der Test, ob die Textur verwendet werden soll, kann sich aber durchaus negativ auf die Performance auswirken.
单独着色器的变体原则上是最干净的解决方案。 GPU只执行我们需要的操作，我们可以非常确定支持不同的着色器。
4. Die Variante für einen separaten Shader ist prinzipiell die sauberste Lösung. Die GPU führt dabei nur die Operationen aus, die wir auch benötigen, und wir können uns ziemlich sicher sein, dass verschiedene Shader unterstützt werden.

所以我们决定变种四。
Wir entscheiden uns also für Variante vier[1].

[1] Variante vier hat im weiteren Verlauf der Übungen auch noch andere Vorteile :)

所以在最简单的情况下，我们会编写一个着色器，它从一个纹理中读取反照率值，另一个使用统一值中的值。 整个其他代码（光照计算，顶点数据的变换等）保持相同。 对于顶点数据，问题是不必要的，因为可以在多个程序中使用相同的着色器，因为片段着色器将被编写和维护，因此相同的代码（这又是丑陋的）。

Im einfachsten Fall würden wir also einen Shader schreiben, der die Albedowerte aus einer Textur liest und einen Anderen, der die Werte aus einem Uniform verwendet. Der gesamte andere Code (die Beleuchtungs berechnung, Transformation der Vertexdaten etc.) bleiben dabei identisch. Für die Vertexdaten erübrigt sich das Problem, da der selbe Shader in mehreren Programmen eingesetzt werden kann, für die Fragmentshader müsste so identischer Code geschrieben und gepflegt werden (Was wieder unschön ist).

Um das Problem zu umgehen verwenden wir die Link-Mechanik der OpenGL. Dabei gehen wir wie folgt vor: Wir unterscheiden zwischen Basis-Shadern und Implementations-Shadern. Wichtiger Hinweis: Das ist eine Designentscheidung für das CG2-Framework. In OpenGL sind alle Shader gleich! Basis-Shader geben die Struktur des Programms vor: Was soll gemacht werden, und in welcher Reihenfolge. Im Vertex-Shader sollen zum Beispiel die Position und alle anderen Attribute entsprechend transformiert werden. Im Fragment-Shader werden all Parameter, die zur Beleuchtung benötigt werden zusammen gesammelt, dann die Beleuchtungsberechnung durchgeführt und das Ergebnis in den Colorbuffer geschrieben:

为了解决这个问题，我们使用了OpenGL的链接机制。 这里我们进行如下：
我们区分基本着色器和实现着色器。 重要说明：这是CG2框架的设计决策。 在OpenGL中，所有着色器都是相同的！ 基本着色器指定程序的结构：应该做什么，以什么顺序。 例如，在顶点

GLSL

```glsl
#version 430 core
// Vertex Base Shader

vec4 transfrom_position_cs();
void transfrom();

void main()
{
  gl_Position=transfrom_position_cs();
  transfrom();
}
```

GLSL

```glsl
#version 430 core
// Fragment Base Shader
layout(location = 0) out vec4 frag_color;

vec4 get_albedo();
vec4 get_material_props();
vec3 get_normal();
vec3 lighting(in vec3 albedo,
              in vec4 material_props,
              in vec3 normal);

void main()
{
  frag_color.rgb = lighting(
                     get_albedo(),
                     get_material_props(),
                     get_normal());

}
```

对于各个步骤，基本着色器仅包含函数声明。 它们的实现来自实现着色器，我们可以在其他着色器中链接（如在C ++中），然后将基础着色器链接到程序。

Für die einzelnen Schritte beinhalten die Basis-Shader nur die Funktionsdeklaration. Deren Implementation kommt dann aus den Implementations-Shadern, welche wir (wie in C++) in anderen Shadern dann mit den Basis-Shadern zu einem Programm linken können.

这种方法允许我们实现不同的信息源并以各种组合使用它们。

Dieses Vorgehen erlaubt es uns verschiedene Quellen für die Informationen zu implementieren, und diese in verschiedenen Kombinationen zu verwenden.

`CG2ShaderCompositor`: CG2ShaderCompositor可以编译和链接各个着色器和程序，并为我们管理GL对象。 API通过CG2Scene传递出来，以便我们可以使用CG2Scene :: buildShaderProgram（..）创建着色器程序并使用CG2Scene :: getProgram（）检索它。

Der `CG2ShaderCompositor` übernimmt das Kompilieren und Linken der einzelnen Shader und Programme und verwaltet die GL-Objekte für uns. Das API wird dabei durch die `CG2Scene` nach außen geführt, sodass wir mit der `CG2Scene::buildShaderProgramm(..)` ein Shaderprogramm erzeugen, und diese auch mit `CG2Scene::getProgram()` wieder abrufen können.

Hinweis: Neben verschiedenen Implementations-Shadern gibt es auch verschiedene Basis-Shader. Der `CG2ShaderCompositor` erzeugt ein Programm für jede Kombination von Vertex- und Fragment-Basis-Shadern und unserer Auswahl an Implementations-Shadern.

注意：除了各种实现着色器之外，还有不同的基本着色器。 CG2ShaderCompositor为顶点和片段基础着色器的每个组合以及我们选择的实现着色器创建一个程序。

1. Implementation-Shader

Ziel dieser Aufgabe ist es, die bestehenden Shader `example.fs.glsl` bzw. `example.vs.glsl` in die Shader-Compositor-Struktur zu überführen.

(a) Passen Sie den Implementations-Shader `vtx_tf_static_full.vert` so an, dass die Funktionen `transform_position_cs()` und `transform_varyings()` entsprechend die Position und die Varyings transformieren. Hinweise:
Sie können Sich an dem Code aus dem `example.vs.glsl` orientieren. Wenn Sie diese Aufgabe korrekt gelöst haben, sollten Sie im Wireframe die Sponza-Szene in weiß erkennen können.

(b) Passen Sie die Implementation von `get_albedo()` in `frg_albedo_tex_only.frag` so an, dass die Textur entsprechend gesampelt wird und die Farbe zurückgegeben wird.

(c) Passen Sie die Implementation von `one_light(in int light_id, ...)` in `frg_lighting_phong.frag` so an, dass das Ergebnis der Beleuchtungsgleichung für die Lichtquelle mit der Nummer `light_id` zurückgegeben wird. Den Parameter `mp` können Sie ignorieren. Auch hier können Sie sich wieder an dem Code in `example.fs.glsl` orientieren.

2. Nicht Texturierte Objekte

Rendern Sie ein weiteres Objekt, für welches keine Textur vorliegt, in den Innenhof der Szene.

(a) Implementieren Sie den entsprechenden Implementations-Shader in `frg_albedo_mat_only.frag`.

(b) Passen Sie die Methode `CG2App::init_shader()` so an, dass ein weiteres Shaderprogramm angelegt wird, welches den Albedo-Wert aus den Materialparametern entnimmt.

(c) Laden Sie ein weiteres Objekt und konfigurieren Sie dessen Material so, dass es in einem Orange erscheint. Für die Geometrie können Sie entweder das Objekt in `data/models/trex.cg2vd` oder `data/models/suzanne.cg2vd` verwenden.

3. Basis-Shader: Depth-Pre-Pass

Bei einer ausreichend komplexen Szene werden die meisten Fragmente früher oder später überschrieben, da nur die 'oberste Schicht' Fragmente im finalen Colorbuffer zu sehen ist. Das ist insbesondere dann ungünstig, wenn die Berechnung der Fragmentfarbe durch die komplexe Beleuchtungsberechnung viel Zeit in Anspruch genommen hat. Ein early-depth-test ist dabei nur bedingt hilfreich. Er verhindert zwar, dass Fragmente, die hinter dem aktuellen Fragment liegen verarbeitet werden, schließt aber nicht aus, dass das Pixel später von einem anderen Fragment überschrieben wird.

Eine Möglichkeit den early-depth-test effektiver zu nutzen wäre es, zuerst den Tiefenpuffer zu füllen, und dann den eigentlichen Renderpass durchzuführen. Dieser sogenannte depth-pre-pass sollte dafür natürlich ohne die aufwändige Beleuchtungsberechnung durchgeführt werden.

Der Ablauf ist dann wie folgt:

- Tiefen und Farbbuffer clearen (`glClear`)
- Tiefentest auf 'kleiner-gleich' stellen (`glDepthFunc`)
- Szene unter Verwendung des vereinfachten Shaders rendern.

- Tiefentest auf 'gleich' stellen
- Szene unter Verwendung des komplexen Shaders rendern.

Dieses Verfahren ist natürlich besonders hilfreich, wenn sonst kein early-depth-test möglich wäre.[2]

Aufgaben:

(a) Implementieren Sie den Vertex-Basis-Shader `base_vtx_position_only.vert` so, dass nur die Positionstransformation durchgeführt wird. Sie können dabei auf die Funktionen der Implementations-Shader zurückgreifen.

(b) Implementieren Sie den Fragement-Basis-Shader `base_frg_null.frag` so, dass nur der Alphatest durchgeführt wird. Entfernen Sie den Alphatest aus `base_frg_cb.frag` und schalten Sie den early-depth-test ein!

(c) Passen Sie die Drawcalls in `CG2App::render_one_frame()` wie oben beschrieben an. Die Szene sollte danach identisch aussehen. Eventuell verbessert sich die Framerate geringfügig.

[2] Durch den Alpha-Test in unserer Szene und das damit verbundene `discard` ist kein early-depth-test möglich.