Computergraphik II

Übung 04

OpenGL Teil 4: Shader Compositor

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.

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.

Unser bisheriges Programm erwartet, dass eine Textur mit den Albedowerten gebunden ist. Wie könnten wir dieses Problem lösen? Intuitiv gibt es mehrere Ansätze:

1. Wir könnten eine 1×1-Pixel Textur erstellen, in der wir die Farbe konfigurieren.
2. Wir könnten vielleicht über eine GLSL-Funktion testen, ob eine Textur gebunden ist?
3. Wir könnten in einem Uniform kodieren, ob die Textur oder eine konstante Farbe aus einem Uniform verwendet werden soll.
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:

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.
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).
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.
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.

Wir entscheiden uns also für Variante vier[1].

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

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:

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

vec4 transfrom_position_cs();
void transfrom();

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

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());

}
```

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.

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

`CG2ShaderCompositor`:
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.

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.

调整实现着色器vtx_tf_static_full.vert，以便函数transform_position_cs（）和transform_varyings（）相应地转换位置和变化。注意事项： 您可以查看example.vs.glsl中的代码。 如果您已正确解决此任务，则应该能够在线框中看到白色的Sponza场景。

(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.

在frg_albedo_tex_only.fragso中调整get_albedo（）的实现，以便相应地对纹理进行采样并返回颜色。

(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.

在frg_lighting_phong.frag中调整one_light的实现（在int light_id, ...中），以返回光源编号light_id的照明方程的结果。 你可以忽略参数mp。 同样，您可以将自己定位回example.fs.glsl中的代码。

(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

渲染另一个在场景的庭院中没有纹理的对象。

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

在frg_albedo_mat_only.frag中实现适当的实现着色器。

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

调整CG2App :: init_shader（）方法以创建另一个着色器程序，该程序从材质参数中获取反照率值。

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

加载另一个对象并将其材质配置为橙色。 对于几何体，您可以使用data / models / trex.cg2vd或data / - models / suzanne.cg2vd中的对象。

(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

在一个足够复杂的场景中，大多数片段迟早会被覆盖，因为只有最终颜色缓冲区中的顶层片段才会可见。 如果由于复杂的照明计算而对片段颜色的计算花费了大量时间，则这是特别不利的。 早期深度测试只是部分有用。 虽然它可以防止处理当前片段后面的片段，但不排除该像素稍后会被另一个片段覆盖。

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.

更有效地使用早期深度测试的一种方法是首先填充深度缓冲区，然后执行实际的渲染过程。当然，这种所谓的深度预通过应该在没有精心设计的照明计算的情况下完成。

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:

过程如下：
•清空深度和颜色缓冲区（glClear）
•将深度测试设置为'小于或等于'（glDepthFunc）
•使用简化着色器渲染场景。

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

- Tiefentest auf 'gleich' stellen
  •将深度测试设置为“相等”
  •使用复杂着色器渲染场景。
- Szene unter Verwendung des komplexen Shaders rendern.

当然，如果不能进行早期深度测试，这种方法特别有用。
Dieses Verfahren ist natürlich besonders hilfreich, wenn sonst kein early-depth-test möglich wäre.[2]

Aufgaben:

实现顶点基础着色器base_vtx_position_only.vert，以便仅执行位置转换。 您可以使用实现着色器的功能。

(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.

实现fragement base shader base_frg_null.frag，以便只执行alpha测试。 从base_frg_cb.frag中删除alpha测试并打开早期深度测试！

(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.

如上所述，调整CG2App :: render_one_frame（）中的drawcalls。 之后场景看起来应该相同。 最终，帧速率略有提高。

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