Computergraphik II

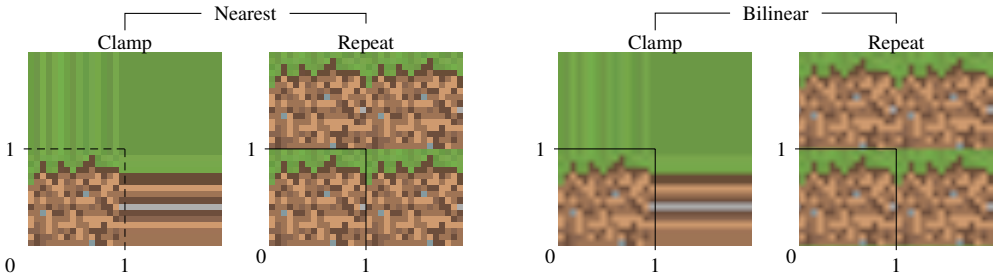
Übung 03

OpenGL Teil 3: Texturen

在本练习中，我们将介绍如何在OpenGL中进行纹理处理。 目标是使用多个纹理从简单图层开始创建基本交叉场景。 重点不应该放在效果上，而应该像以前那样关注纹理的OpenGL特定方面。

In dieser Übung werden wir uns mit der Umsetzung von Texturierung in OpenGL beschäftigen. Ziel ist es mehrere Texturen zu verwenden um, ausgehend von einer einfachen Ebene, eine rudimentäre Kreuzungsszene zu erstellen. Der Fokus soll dabei nicht auf dem Effekt liegen, sondern - wie bisher auch - auf den OpenGL-spezifischen Aspekten der Texturierung.

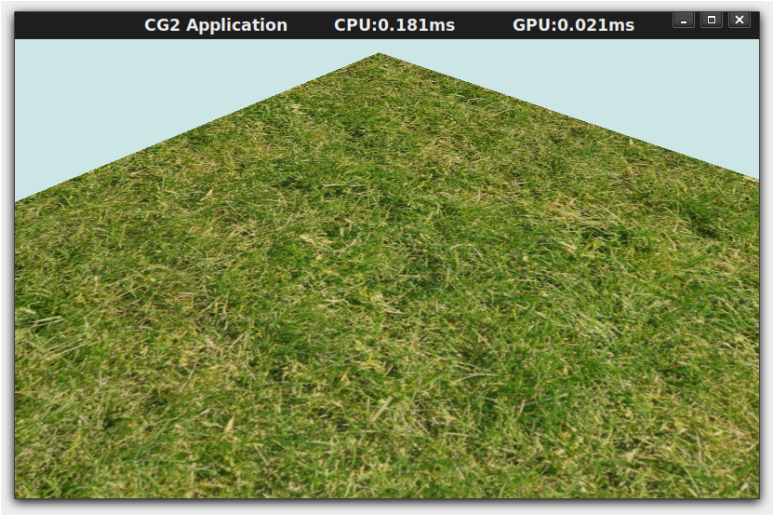
Zur Erklärung: In OpenGL sind Texturen ein-, zwei- oder dreidimensionale Arrays von bis zu vierdimensionalen (RGBA) Werten, die wir Texel (**Tex**ture **el**ements)[1] nennen. Texturen werden generell eingesetzt um räumlich höher aufgelöste Daten effizient in die Fragmentverarbeitung (in der Regel den Fragmentshader) zu bekommen. Um dies zu erreichen wird zum Beispiel jedem Vertex eine Position im Texturraum zugewiesen (Texturkoordinaten $\boldsymbol{t}$). Diese Texturkoordinaten werden dann interpoliert und als Varyings an die Fragmentverarbeitung weitergegeben, wo diese dann genutzt werden können, um die Bildinformation an der entsprechenden Stelle zu sampeln (abtasten/auswerten). Dazu werden die Texturkoordinaten in Texelkoordinaten ($\boldsymbol{T}$) umgerechnet ($\boldsymbol{T} = \boldsymbol{t} \cdot \boldsymbol{r}$ wobei $\boldsymbol{r}$ die Auflösung der Textur in Texeln ist). Da die Texturkoordinaten nicht zwingend auf genau einem Texelzentrum liegen, muss ein Vorgenen definiert werden, nach dem die Farbwerte zwischen den Texelzentren berechnet werden. Diesen Prozess nennt man auch das Filtern der Textur. In der Praxis werden zwei wesentliche Filter eingesetzt. Beim **Nearest**-Filter wird immer der Wert des nächsten Texels verwendet. Beim **Bilinearen**-Filter hingegen werden die Farbwerte der vier nächsten Texel, entsprechend dem Abstand zu diesen Texeln linear gewichtet gemischt, um den Farbwert zu ermitteln. In OpenGL werden zwei Filterregeln angegeben: Eine, wenn die Textur verkleinert wird (der Abdruck des Fragments im Texturraum ist größer als ein Texel) und eine weitere, wenn die Textur vergrößert werden muss. Der Texturraum ist, wie oben beschrieben, so definiert, dass die gesamte Texturinformation im Wertebereich $[0, 1]^n$ liegt. Der Texturraum ist deshalb aber nicht auf diesen Wertebereich beschränkt. Zusätzlich zu den Filterregeln werden deshalb noch Regeln definiert, wie mit Texturcoodinaten außerhalb von $[0, 1]^n$ umgegangen wird. Dieses sog. Wrapping (umbrechen) kann auf verschiedene Art und Weise erfolgen. Im einfachsten Fall ist außerhalb $[0, 1]^n$ Schluss, und die Texelkoordinaten werden auf die äußere Kante der Textur beschränkt (**clamp**). Dieses Vorgehen halt zur Folge, dass sich das letzte Texel immer weiter wiederholt. Alternativ dazu kann die Texturinformation auch wiederholt werden (**repeat**), um so außerhalb von $[0, 1]^n$ Werte zu generieren. Bei OpenGL können diese Modi für jede Dimension getrennt angegeben werden.



[1] Analog: Pixel = **Pic**ture **El**ement, Voxel = **Vo**lume **El**ement, etc..

说明：在OpenGL中，纹理是一维，二维或三维阵列，最多可达四维

（RGBA）我们称为纹素（纹理元素）的值。纹理通常用于将空间上更高分辨率的数据有效地提取到片段处理（通常是片段着色器）中。为了实现这一点，例如，为每个顶点分配纹理空间中的位置（纹理坐标t）。然后对这些纹理坐标进行插值并作为Varyings传递给片段处理，然后可以使用它们在适当的位置对图像信息进行采样（扫描/评估）。为了这个目的，纹理坐标是在纹理像素坐标（T）转换（T = T * R，其中R是纹理的纹理像素中的分辨率是）。由于纹理坐标不一定位于正好一个纹素中心，因此必须定义前体，之后计算纹素中心之间的颜色值。此过程也称为过滤纹理。实际上，使用了两个主要的过滤器。最近的过滤器始终使用下一个纹素的值。另一方面，在双线性滤波器中，根据到这些纹素的距离对四个最接近的纹素的颜色值进行线性加权，以确定颜色值。 ，A如果纹理减小（在纹理空间片段的印象是多于一个纹理像素更大）和另一，如果纹理必须增加：在OpenGL两个过滤规则中指定的。如上所述，纹理空间被定义为使得整个纹理信息在值范围[0; 1]谎言。因此，纹理空间不限于该范围的值。除了过滤规则，因此仍然定义规则，如同来自外部的纹理坐标[0; 1] n被处理。这就是所谓的包装（包装）可以用不同的方式完成。在最简单的情况下，在[0; 1] n完成，纹理像素坐标被夹在纹理的外边缘。这个过程意味着最后一个Texel会越来越重复。或者，可以重复纹理信息以使其超出[0; 1] n生成值。使用OpenGL，可以为每个维度单独指定这些模式。



Tastaturbelegung		
Esc	Programm beenden	
F1 / Shift F1	Wireframe	an aus
W A S D	Bewegen	
Maus	Cursor	an aus

1. Eine einfache Textur

在该项目中有一个新类CG2Image，该类允许您以通常的文件格式（PNG，TGA，JPG，BMP，...）加载图像。

Im Projekt gibt es eine neue Klasse `CG2Image`, diese Klasse erlaubt es Ihnen Bilder in den üblichen Dateiformaten (PNG,TGA,JPG,BMP,...) zu laden.[2]

使用CG2App :: init_gl_state（）中准备的CG2Image加载imagedata / textures / grass.jpg。

(a) Nutzen Sie das, in `CG2App::init_gl_state()` vorbereitete, `CG2Image` um das Bild `data/textures/grass.jpg` zu laden.

用tex_grass创建一个新纹理并将其绑定到GL_TEXTURE_2D。（GlGenTextures，glBindTexture）

(b) Erzeugen Sie in `tex_grass` eine neue Textur und binden Sie diese in `GL_TEXTURE_2D`.(`glGenTextures`,`glBindTexture`)

从加载的数据初始化纹理。（GlTexImage2D）

(c) Initialisieren Sie die Textur aus den geladenen Daten. (`glTexImage2D`)

将缩小和放大的滤镜模式设置为GL_LINEAR。（GlTexParameteri）

(d) Setzen Sie den Filtermodus zur Verkleinerung und Vergrößerung auf `GL_LINEAR`. (`glTexParameteri`)

Damit ist die Textur von der Client Seite her initialisiert und kann im Shader verwendet werden. Um auf die Textur im Shader zugreifen zu können, benötigen wir ein entsprechendes Konstrukt, um die Textur zu identifizieren. Texturen werden dabei als Uniforms eines besonderen Typs repräsentiert, einem sogenannten Sampler, und im Falle einer 2D-Textur einem `sampler2D`.

这将从客户端初始化纹理，并可在着色器中使用。 要在着色器中访问纹理，我们需要一个构造来识别纹理。 纹理表示为特殊类型的uniform，即所谓的采样器，在2D纹理的情况下，表示为sampler2D。

GLSL

```
layout(binding = 0) // Auswahl der Textureinheit 0 (spaeter mehr)
uniform sampler2D my_texture;
```

采样器my_texture现在允许我们在texture0上采样2D纹理（因此得名）。

Der Sampler `my_texture` erlaubt es uns jetzt die 2D-Textur an Textureinheit `0` zu sampeln (daher der Name).

(e) Nutzen Sie den Sampler um die Materialfarbe aus der Textur zu sampeln. Die Texturkoordinaten erhalten Sie aus dem Varying `in vec2 vtex`.

使用采样器从纹理中采样材质颜色。 纹理坐标是从vec2 vtex中的Varying获得的。

GLSL

```
vec4 texture(sampler2D sampler, vec2 P)
```

Sampelt die Textur an der gegebenen Position `P` und gibt das Ergebnis als RGBA Vektor zurück.

在给定位置P处对纹理进行采样，并将结果作为RGBA向量返回。

缩放纹理坐标，使纹理看起来彼此相邻十次（矩形的设计使得顶点具有纹理坐标（0；0），（1；0），（1；1）和（0；1））。

(f) Skalieren Sie die Texturkoordinaten so, dass die Textur zehn mal gekachelt nebeneinander erscheint (das Rechteck ist so angelegt, dass die Ecken die Texturkoordinaten $(0,0)$, $(1,0)$, $(1,1)$ und $(0,1)$ haben).

[2] Intern verwenden wir dazu die stb_image Bibliothek https://github.com/nothings/stb

Texturen werden in OpenGL als Objekte modelliert und bilden die zweite große Objektfamilie neben den Buffern. Eine Textur kann (und muss) ähnlich wie ein Buffer gebunden werden. Texturen kapseln neben den eigentlichen Daten auch den eingestellten Filter- und Wrapping-Modus. Es werden neben den klassischen 2D-Texturen auch 1D- und 3D-Texturen auch spezielle Cubemap-Texturen unterstützt.

`void glGenTextures(GLsizei n, GLuint* textures)`
Erzeugt analog zu `glGenBuffers` `n` Texturnamen und speichert diese in `textures`.

`void glBindTexture(GLenum target, GLuint texture)`
Bindet `texture` an das angegebene `target`. Für uns sind folgende Targets von Interesse:
`GL_TEXTURE_2D`: für 2D-Texturen
`GL_TEXTURE_CUBE_MAP`: für CubeMaps (kommen später)

```
void glTexImage2D(GLenum target, GLint level,
    GLint internalformat,GLsizei width, GLsizei height, GLint border,
    GLenum format, GLenum type, const GLvoid * data);
```
Legt eine neue Textur an und füllt sie aus den übergebenen Daten. Die Parameterliste lässt sich in drei Abschnitte aufteilen:

- `target` gibt das Binding Target an, an dem die entsprechende Textur gebunden ist. Texturen haben Schichten[a]. Diese sogenannten `level` entsprechen den einzelnen Schichten der Mip-Map-Pyramide. Der `level` Parameter gibt die Schicht an, die gerade angelegt wird. Setzen Sie diesen Parameter `0`.
- Der zweite Abschnitt befasst sich mit dem Format und der Auflösung der Textur:
  - `internalformat` Beschreibt die Struktur der Texel in der Textur. Der Einfachheit halber verwenden wir einen der folgenden Werte: `GL_RED`,`GL_RG`,`GL_RGB`,`GL_RGBA`. Bei diesen Werten ist nur die Anzahl der Kanäle spezifiziert, und es wird der OpenGL Implementation überlassen, wie viele Bit pro Kanal verwendet werden (in der Regel 8 Bit).
  - `width` und `height` geben die Auflösung der Textur an.
  - Der Parameter `border` ist ein Relikt aus alten Zeiten und muss immer `0` sein.
- Der letzte Teil befasst sich mit den Daten, die wir übertragen wollen. Format gibt, wie das interne Format, an wie viele Kanäle wir übertragen wollen. (`format` und `internalformat` müssen nicht übereinstimmen. Die GL wird die Daten entsprechend konvertieren.) Mit `type` geben wir den Typ der einzelnen Werte an (`GL_UNSIGNED_BYTE`,`GL_UNSIGNED_SHORT`,`GL_UNSIGNED_INT`,`GL_FLOAT`,`...`), und `data` ist ein Zeiger auf die Daten aus denen gelesen wird (Auch die Typen werden im Zweifelsfall konvertiert).

`void glTexParameteri(GLenum target, Glenum pname, GLint param)`
Setzt den Integerparameter `pname` für die an `target` gebundene Textur auf `param`. Hier eine (hoffnungslos unvollständige) Liste der Parameter und ihrer möglichen Werte:
`GL_TEXTURE_MIN_FILTER`/`GL_TEXTURE_MAG_FILTER` setzt den Filter-Modus bei Verkleinerung/Vergrößerung:
`GL_NEAREST`: Nearest-Filter.
`GL_LINEAR`: Bilinearer-Filter.
`GL_TEXTURE_WRAP_S`/`GL_TEXTURE_WRAP_T` setzt den Wrapmodus für die S- bzw. T-Richtung[b]
`GL_CLAMP_TO_EDGE`: Clamp-Modus wie oben beschrieben.
`GL_REPEAT`: Repeat-Modus, wie oben beschrieben.
`GL_MIRRORED_REPEAT`: Mirroed-Repeat (können Sie ja mal ausprobieren)

[a] Wie Oger.
[b] S und T sind X und Y im Texturraum

到目前为止，我们只使用了一种纹理来编码材质的颜色。 通过这种设置，我们可以纹理不同的对象，我们只需要将匹配的纹理绑定到GL_TEXTURE_2D，然后我们就可以使用同一个着色器程序显示不同的纹理对象。 仅当我们想要在着色器中一次使用多个纹理时才变得困难。

到目前为止，我们无法绑定多个2D纹理，但是采样器前面的布局（binding = 0）给出了一个指示，即可以使用多个sampler2D（类似于UBO）。 但是，对于纹理，由于历史原因，选择的结构与UBO不同。

不是将纹理对象绑定到几个GL_TEXTURE_2D目标之一，而是在绑定纹理之前的状态中选择许多Texture Units 中的一个。 纹理单元表示所有纹理类型（GL_TEXTURE_1D，GL_TEXTURE_2D，GL_TEXTURE_3D等）的一组目标，并且纹理可以绑定到这些目标中的每一个。 但是，在着色器程序中，只允许访问纹理单元的一个纹理。因此，您可能不会使用具有相同layout(binding = 0) 的sampler2D和sampler3D，但您可以使用来自不同单元的两个采样器。

2. Mehrere Texturen

Bisher haben wir genau eine Textur verwendet, um die Farbe des Materials zu kodieren. Mit diesem Aufbau können wir verschiedene Objekte texturieren, wir müssen einfach immer die passende Textur an `GL_TEXTURE_2D` binden und können dann mit ein und dem selben Shaderprogramm verschiedene, texturierte Objekte darstellen. Schwierig wird es erst, wenn wir mehr als nur eine Textur gleichzeitig im Shader verwenden wollen. Wir haben bisher keine Möglichkeit mehr als eine 2D-Textur zu binden, das `layout(binding = 0)` vor dem Sampler gibt uns aber schon einen Hinweis, dass es möglich ist (ähnlich wie bei UBOs) mehr als einen `sampler2D` zu verwenden. Bei Texturen wird, aus historischen Gründen, allerdings eine andere Abstraktion als bei den UBOs gewählt[3]. Anstatt das Texturobjekt an eines von mehreren `GL_TEXTURE_2D` Targets zu binden wird im State eine von vielen **Texture Units** ausgewählt, bevor die Textur gebunden wird. Eine Texture Unit repräsentiert einen Satz Targets für alle Texturtypen (`GL_TEXTURE_1D`,`GL_TEXTURE_2D`,`GL_TEXTURE_3D`, etc.), und an jedes dieser Targets darf eine Textur gebunden sein. Im Shaderprogramm darf aber nur auf eine Textur einer Texture Unit zugegriffen werden. Sie dürfen also nicht einen `sampler2D` und `sampler3D` mit dem identischen `layout(binding = 0)` verwenden, können aber zwei Sampler aus verschiedenen Units verwenden.

此任务的目标是使用三个（相对）低分辨率纹理创建更复杂的纹理。一个纹理（distribution.png）编码其他两个纹理的混合比例（red通道中的grass.png和绿色通道中的pebble.png）

Ziel dieser Aufgabe ist es, aus drei (relativ) niedrig aufgelösten Texturen eine komplexere Texturierung zu schaffen. Dabei kodiert eine Textur (`distribution.png`) das Mischverhältnis der beiden anderen Texturen (`grass.png` im Rotkanal und `pebble.png` im Grünkanal).

扩展方法CG2App :: init_gl_state（），以便除了草纹理之外，图像data / textures / pebble.jpg和data / textures / distribution.png也会加载到相应的纹理tex_pebble和tex_distribution中。 设置草纹理的参数。

(a) Erweitern Sie die Methode `CG2App::init_gl_state()` so, dass neben der Grastextur auch die Bilder `data/textures/pebble.jpg` und `data/textures/distribution.png` in die entsprechenden Texturen `tex_pebble` und `tex_distribution` geladen werden. Setzen Sie die Parameter wie bei der Grastextur.

纹理绑定到以下纹理单位：tex_grass到GL_TEXTURE0，tex_pebble到GL_TEXTURE1，tex_distribution到GL_TEXTURE2。

(b) Binden Sie die Texturen an die Folgenden Texture Units: `tex_grass` an `GL_TEXTURE0`,`tex_pebble` an `GL_TEXTURE1` und ,`tex_distribution` an `GL_TEXTURE2`.

调整着色器以创建另外两个sampler2D，其中bindings1和2用于砾石和分布纹理。

(c) Passen Sie den Shader so an, dass zwei weitere `sampler2D` mit den Bindings `1` und `2` für die Schotter und die Distribution-Textur angelegt werden.

使用分布纹理（cd）混合两个纹理（cg；cp）的颜色值。 确保直接访问分布纹理（不缩放纹理坐标）。 草和砾石纹理都应该使用平铺。

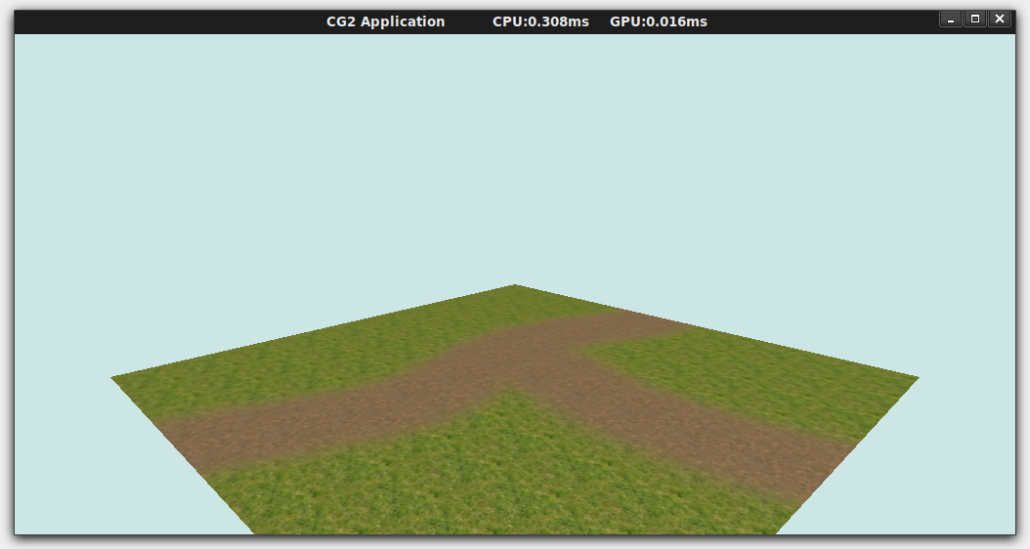
(d) Mischen Sie die Farbwerte der beiden Texturen ($\boldsymbol{c_g}, \boldsymbol{c_p}$) unter Verwendung der Distributiontextur ($\boldsymbol{c_d}$). Achten Sie darauf, dass auf die Distributiontextur direkt (also ohne Skalierung der Texturkoordinaten) zugegriffen werden soll. Die Gras und die Schottertextur sollen aber beide gekachelt angewendet werden.

$$\boldsymbol{c_{dR}} \cdot \boldsymbol{c_g} + \boldsymbol{c_{dG}} \cdot \boldsymbol{c_p} = \boldsymbol{c}_m$$

`glActiveTexture(GLenum unit);`

Selektiert `unit` als die aktive Texture Unit. Alle folgenden texturspezifischen GL-Kommandos werden sich auf diese Texture Unit beziehen. Wichtig: Der Parameter `unit` muss dabei `GL_TEXTURE0`,`GL_TEXTURE1`,... sein. Nicht der Integer $0, 1, 2, ...$, wobei `GL_TEXTURE`i = `GL_TEXTURE0`+i ist.

[3] Multitexturing ist älter als UBOs oder andere Array-Targets, viel älter.



3. Zusatzaufgabe:

在Destribution纹理的蓝色通道中，编码地形的高度（Y方向的偏移）。 由于我们无法在片段着色器中移动片段，因此必须已在顶点着色器中应用此信息。 使用顶点着色器中的分布纹理相应地移动顶点的对象空间位置。尽管在Worldspace中有正确的位置，为什么结果不能令人信服？

Im Blaukanal der Destributiontextur ist die Höhe des Terrains (Verschiebung in Y-Richtung) kodiert. Da wir Fragmente im Fragmentshader nicht verschieben können, muss diese Information bereits im Vertexshader angewendet werden. Nutzen Sie die Distributiontextur im Vertexshader um die Objectspace-Position der Vertices entsprechend zu verschieben. Warum sieht das Ergebnis, trotz korrekter Positionen im Worldspace nicht überzeugend aus?