C++ Hinweise zu Übungsblatt 3

Nachfolgend finden sich wieder Hinweise zu C++, die für Übungsblatt 3 relevant werden könnten. Wie im letzten Blatt gilt, dass diese nur kurz/beispielhaft erklärt werden.

Was sind Zeiger? (What are pointers?)

Einleitung

Wenn man Variablen anlegt, werden diese immer irgendwo im Arbeitsspeicher (RAM) gespeichert. In vielen Programmiersprachen wie Java ist mehr oder weniger verborgen, an welcher Stelle im Speicher Variablen gespeichert werden. In C++ kann man dagegen für jede Variable (egal ob z.B. im Stack oder Heap gespeichert), herausfinden, wo diese gespeichert ist – also an welcher Speicheradresse.

Speicheradresse zur Variable erhalten

Um die Speicheradresse einer Variable in C++ zu erhalten, schreibt man ein **&**-Zeichen vor dem Variablennamen:

```
// Definiere Variable:
int myVariable = 1;

// Erhalte Speicheradresse der Variable:
&myVariable;
```

Speicheradresse auf Konsole ausgeben

Die Speicheradresse einer Variable kann man sich auf die Konsole ausgeben (in Hexadezimaldarstellung):

```
// Gebe Speicheradresse auf der Konsole aus:
std::cout << &myVariable << std::endl;
```

I.d.R. hilft einem das Ausgeben aber wenig, außer um zu verstehen, was ein Zeiger genau ist (und es ist zudem Compiler- und Systemabhängig).

Speicheradresse als Variable speichern

Die Speicheradresse einer anderen Variable kann man wiederum auch in eine Variable speichern, was man *bei der Typangabe* mit einem *nachgestellten* *****-Zeichen kennzeichnen kann, also:

```
int* pointerToMyVariable = &myVariable;
```

Merke: Eine Variable, die eine Speicheradresse speichert, wird auch Zeiger oder Pointer genannt.

Wert eines Zeigers erhalten

Um wieder auf den Wert eines Pointers zugreifen zu können, kann man ein *****-Zeichen *vor dem Variablennamen eines Pointers* schreiben, also:

```
// Speichere den Wert der Variable myVariable in myOriginalVariable:
int myOriginalValue = *pointerToMyVariable;
```

In diesem Falle würde myOriginalValue wieder den Wert 1 enthalten.

Merke: Den Vorgang, den Wert an einer Speicheradresse bzw. Pointer zu erhalten, nennt man auch *Dereferenzierung*. Mithilfe von `*pointerToMyVariable` *dereferenziere* ich also den Pointer.

Zeiger als `void*`-Variable speichern

Vielleicht fragt ihr euch, warum man für einen Pointer überhaupt den Typ wie `int*` oder `Vec4f*` mitangeben muss – schließlich handelt es sich immer um eine Adresse im Arbeitsspeicher und der Arbeitsspeicher kennt schließlichgar keine Typen. Und tatsächlich braucht man keinen konkreten Typ mitangeben, sondern kann jeden Pointer auch als `void*` deklarieren:

```
void* myVoidPointer = &myVariable;
```

Wenn eine Funktion einen Parameter vom Typ `void*` besitzt, z.B.:

```
void myFunction(void* aPointer){
// [...]
}
```

dann bedeutet dass, dass man einen beliebigen Pointer an die Funktion übergeben kann. Ich könnte sie also mit einen int*-Pointer aufrufen, d.h. `myFunction(&myVariable)` aufrufen, aber ich könnte ihr z.B. auch einen Vec4f*-Pointer übergeben.

Merke: `void*` steht also für irgendeinen Pointer, egal von welchem Typ.

Wert eines `void*`-Zeigers erhalten

Warum gibt es denn trotzdem soetwas wie einen `int*`-Zeiger und `Vec4f*`-Zeiger? Das lieg daran, dass man einen void*-Pointer nicht dereferenzieren kann und folgendes ergibt einen Fehler:

```
// Erzeugt Fehler
int myOriginalValue = *myVoidPointer;
```

Der C++ Compiler weiß schließlich nicht, von welchem Typ der Wert ist, welcher an der Speicheradresse `myVoidPointer` beginnt – wie gesagt, der Arbeitsspeicher kennt keine Typen, sondern nur Bits und Bytes. Es könnte z.B. ein Wert vom Typ *int* sein, ein Wert vom Typ `Vec4f` oder ein x-beliebiges anderes Objekt.

Im Falle eines `void*`-Pointers kann man den C++ Compiler aber dennoch sagen, er soll ihn z.B. wie ein int*-Pointer behandeln, also:

```
// Funktioniert:
int originalValue = *((int*)myVoidPointer);
```

Dabei wird der `void*`-Pointer erst als int*-Pointer interpretiert (via Casting, wie aus Java bekannt), und anschließend kann man ihn mittels vorgestelltes *-Zeichen dereferenzieren.

Warum sind Pointer denn für Übungsblatt 3 wichtig?

Immer, wenn man mithilfe von OpenGL Daten an die Grafikkarte überträgt, verlangt OpenGL in C++ nicht direkt die Daten (z.B. einen `std::vector`), sondern lediglich einen Pointer auf den Anfang der Daten, ebenso wie die Größe der Daten in Bytes. OpenGL kopiert sich dann selbst die Daten aus den Arbeitsspeicher im angegebenen Bereich heraus.

std::vector - Speicherlayout

Im letzten C++ Hinweisblatt hast Du bereits erfahren, dass die `std::vector`-Klasse soetwas wie eine ArrayList in Java ist und wie sie grundlegend benutzt werden kann (heißt, wie Werte/Objekte eingefügt werden und wie man sie ausließt). Es gibt zudem die Funktion `size()`, die zurückgibt, wie viele Elemente in der Liste enthalten sind - genauso wie in Java.

Um auf den darunter liegenden Buffer/Array zuzugreifen, in welchem die Daten gespeichert sind, kann man die `data()`-Funktion benutzen. Diese Funktion gibt einen Pointer auf das erste Element der Liste zurück.

```
// Liste erstellen:
std::vector<int> myList;
myList.push_back(42);
myList.push_back(7);

// Gibt Pointer auf das erste Element zurueck:
int* firstElementPointer = myList.data();

// Alternativ waere auch moeglich:
int* firstElementPointer2 = &myList[0];
```

Kontinuität: Eine wichtige Eigenschaft der `std::vector`-Klasse ist, dass alle Elemente garantiert direkt hinter einander im Arbeitsspeicher liegen. Wenn man mittels OpenGL alle Elemente einer solchen Liste auf den Grafikspeicher übertragen will (z.B. in einem VBO), dann kann man sicher sein, dass alles übertragen wird, wenn man den Pointer auf das erste Element übergibt.

sizeof-Operator

C++ besitzt einen sizeof-Operator, der für jeden Typ herausfinden kann, wie viel Bytes an Speicher dieser im Arbeitsspeicher beansprucht. Das kann manchmal ganz nützlich sein:

```
unsigned int sizeOfInt = sizeof(int); // Gibt 4 zurueck
unsigned int sizeOfFloat = sizeof(int); // Gibt 4 zurueck

// Gibt 16 zurueck, da die Vec4f-Klasse 4 * 4 Byte Attribute speichert:
unsigned int sizeOfVec4f = sizeof(Vec4f);

// Gibt auf den meisten 64-Bit Systemen 8 zurueck (Platzbedarf einer Speicheradresse):
unsigned int sizeOfPtr = sizeof(void*);
```

Typen definieren

Was ist das?

Wieder etwas, was in Java nicht geht: In C++ kann man Typen mittels `typedef` Schlüsselwort einen anderen Namen geben. Wenn euch der Name der Klasse `Vec4f` nicht gefällt und ihr `Vector` besser findet, könntet ihr einfach `typedef Vec4f Vector` schreiben und schon könnt ihr die Funktionalität der Klasse so verwenden, als hätte sie den Namen `Vector`:

```
typedef Vec4f Vector;
Vector myVec4f = Vector(1.0, 1.0, 1.0);
```

Das sollt ihr aber bitte **nicht** machen! ;-) Ebenso könntet ihr beide Typen mixen, da beide Typnamen vom Compiler als den gleichen Typ behandelt werden:

```
Vector a = Vec4f(1.0, 1.0, 1.0);
Vec4f b = Vector(1.0, 1.0, 1.0);
```

Das bitte erst recht **nicht** machen!

OpenGL definiert eigene Typen

Der Hintergrund, warum dies hier erklärt wird, ist, dass OpenGL genauso seine eigenen Typen definiert. Ein paar Beispiele sind:

```
typedef unsigned int GLenum;
typedef int GLint;
typedef unsigned int GLuint;
typedef float GLfloat;
typedef double GLdouble;
typedef void GLvoid;
```

Der Grund, warum OpenGL dies macht, ist Kompatibilität: In C++ sind nicht zwangsläufig alle primitiven Typen auf allen Systeme gleich groß (z.B. wenn man für Mikroprozessoren entwickelt, o.ä.). D.h. auf manchen Systemen werden diese Typen von OpenGL anders definiert, um die Kompatibilität mit der Grafikkarte zu wahren, damit diese beim Übergeben an die Grafikkarte genauso groß sind wie erwartet. Für heutige gängige Computer & Betriebsysteme ist dies in der Praxis mit OpenGL allerdings eher wenig von Bedeutung.

Wichtig ist nur, dass ihr euch nicht fragt, was diese komischen Typen der Parameter bedeuten, die OpenGL in seinen Funktionen nutzt, siehe z.B. die Dokumentation von `glBufferData`: https://registry.khronos.org/OpenGL-Refpages/gl4/html/glBufferData.xhtml – der `GLsizeiptr`-Typ ist übrigens auch nur eine vorzeichenbehaftete Ganzzahl, je nach System 32-Bit oder 64-Bit. Man kann als Argument also einfach ein `int` übergeben.

GLenum

Übrigens hat OpenGL ziemlich viele Konstanten vordefiniert, z.B. soetwas wie `GL_ARRAY_BUFFER`, `GL_STATIC_DRAW`, `GL_FLOAT` und vieles mehr, die lediglich eine Ganzzahl speichern. Immer wenn OpenGL ein `GLenum` haben möchte (welcher wie oben sichtbar ja als `unsigned int` definiert ist), möchte OpenGL im Normalfall einer dieser Konstanten übergeben bekommen, die i.d.R. für eine bestimmte Option oder Eigenschaft stehen.