Digitales Sommersemester 2020:

Datenvisualisierung und GPU-Computing

Übung 2: Qt, OpenGL, Colourmapping

21.04.2021

Andreas Beckert

andreas.beckert@uni-hamburg.de

Regionales Rechenzentrum, Visual Data Analysis Group,

FB Informatik Arbeitsgebiet „Scientific Visualization and Parallel Processing“

Diese Übung

- Besprechung von Übung 1b

- OpenGL in Qt
- Template für Übung 2: Ein einfaches OpenGL-Programm in Qt

- Übung 2:
  - Integration eines OpenGL-Displays in unser Programm.
  - Ziel: Erste visuelle Darstellung unseres Datenfeldes, denn: *„The purpose of computing is insight, not numbers.“* – Richard Hamming (1962, Numerical Methods for Scientists and Engineers)

Grundlagen Qt

- Qt stellt viele **Klassen** und **Typen** bereit, die die Implementierung von komplexeren C++-Programmen (mit und ohne GUI) vereinfachen, gruppiert in Modulen: https://doc.qt.io/qt-5/modules-cpp.html
  - In `QtCore` z.B.: `QString`, `QDateTime`, `QVector`, …

- GUI Applikationen können mit den „Qt Widgets“ implementiert werden, ein Beispiel ist eine einfache „Notepad“ Applikation: https://doc.qt.io/qt-5/qtwidgets-tutorials-notepad-example.html

- Weitere Beispiele: https://doc.qt.io/qt-5/qtexamplesandtutorials.html

- Wir schauen uns später ein Beispiel an, welches als Template für Übung 2 verwendet wird.

Computergrafik mit OpenGL in Qt

- Was ist wichtig für uns?

  - OpenGL in Qt
  - „Object-order rendering“: Programmierbare Rasterisierungs-Grafikpipeline auf der GPU *(!= Visualisierungs-Pipeline)*
  - Vertex Processing: Transformationen durch Matrizen
  - Fragment Processing: Texturen, Depth Test
  - Umsetzung in OpenGL: GLSL Shader

OpenGL in Qt: QOpenGLWidget

- Wir verwenden Shader-basiertes OpenGL Version 3.3/4 (Lauffähigkeit auf Heimrechnern?)
  - „**Core-Profile**“, keine fixed-function-Pipeline wie in altem OpenGL.

- In Qt5 sind die OpenGL-Funktionen sehr einfach über das „QOpenGLWidget“ verfügbar: https://doc.qt.io/qt-5/qopenglwidget.html

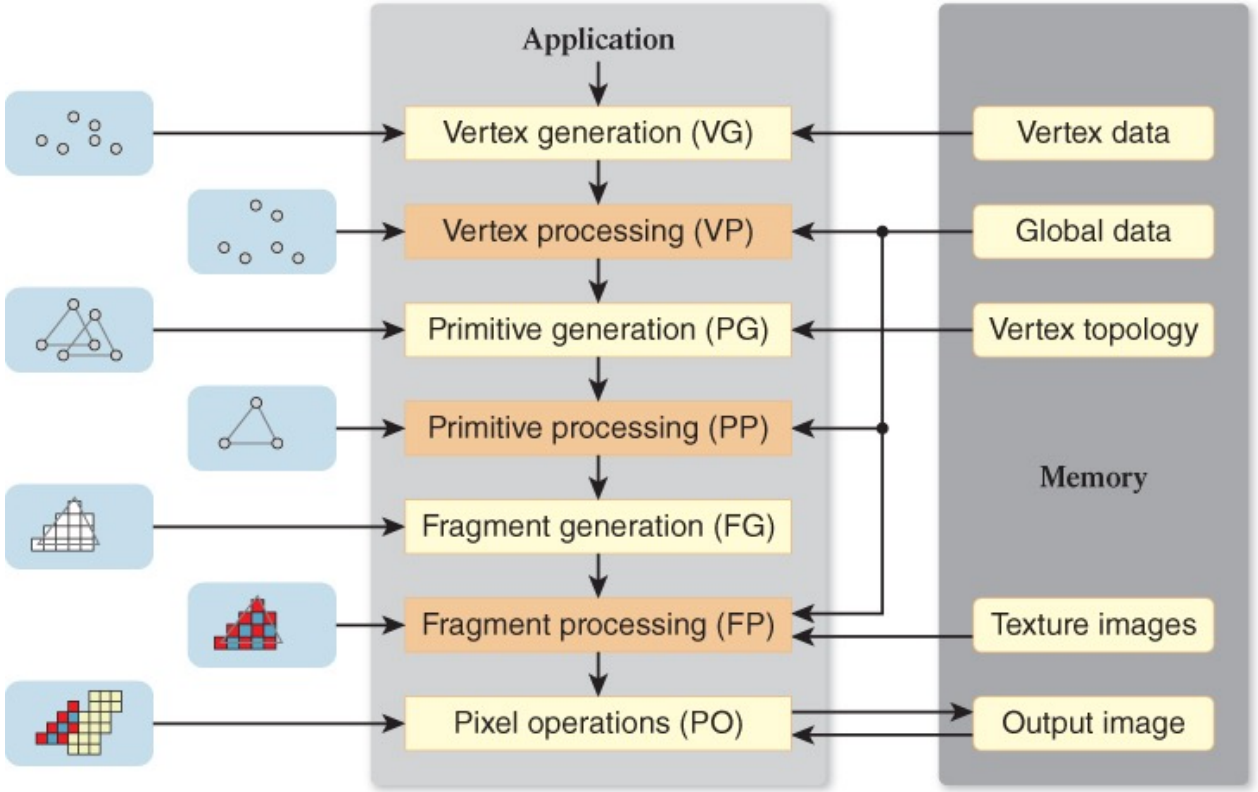
- Qt übernimmt die Erstellung des Viewports; es stellt außerdem weitreichende Abstrahierungen der OpenGL-API zur Verfügung, so dass wir uns nur um die „wichtigen“ Dinge kümmern müssen.

- Bevor wir Implementierungs-Details betrachten: Grundlagen OpenGL…

Rasterization-based graphics pipeline

Graphics pipeline on recent graphics cards (GPUs)

- Programmable with shading languages: **GLSL** (OpenGL), **HLSL** (DirectX), **Nvidia CG** (both) → syntax similar to C/C++



Hughes et al. (2013)

Rasterization-based graphics pipeline in OpenGL

Result of vertex shader stage

- Vertex coordinates after perspective projection
- Additional attributes like color and texture coordinates

Simple example of a vertex shader:

```
in vec3 vertexPosition;
in vec2 vertexTexCoord;
uniform mat4 modelViewProjectionMatrix;

smooth out vec2 texCoordVS2FS;

void main()
{
gl_Position = modelViewProjectionMatrix * vec4(vertexPosition.xyz, 1);
texCoordVS2FS = vertexTexCoord;
}
```

Rasterization-based graphics pipeline in OpenGL

The transformed, attributed vertex stream is passed to the rasterizer stage

- The rasterizer maps canonical view volume to pixel coordinates
- For each triangle, the rasterizer determines the pixels covered by this triangle – for each such pixel a fragment is generated
- Per-vertex attributes are interpolated to each fragment

Simple example of a fragment shader:

```
smooth in vec2 texCoordVS2FS;
uniform sampler2D textureSampler;
layout(location = 0) out vec4 fragmentColor;

void main()
{
fragmentColor = texture(textureSampler, texCoordVS2FS);
}
```

Auf der CPU-Seite…

- Grundlegende Programmstruktur OpenGL Grafikprogramm:

```
#include <libraryheaders>

int main()
{
    createWindow(title, width, height);
    createOpenGLContext(settings);

    while (windowOpen)
    {
        while (event = newEvent())
            handleEvent(event);

        updateScene();

        drawGraphics();

        presentGraphics();
    }

    return 0;
}
```

https://open.gl

Grundlagen OpenGL Programmierung

- Wie für C++ gibt es für OpenGL viele Online-Tutorials, z.B.:
  - https://open.gl/
  - https://learnopengl.com/

- Kleines Glossary: https://learnopengl.com/Getting-started/Review

- OpenGL Reference Card: https://www.khronos.org/files/opengl46-quick-reference-card.pdf (gibt keine für v3.3...)
  - Siehe auch: https://www.khronos.org/developers/reference-cards/

- Wir benötigen das meiste davon erstmal nicht: Fokus auf minimales Programmgerüst in Qt.

Web-Rundtour (1): Was von OpenGL benötigen wir?

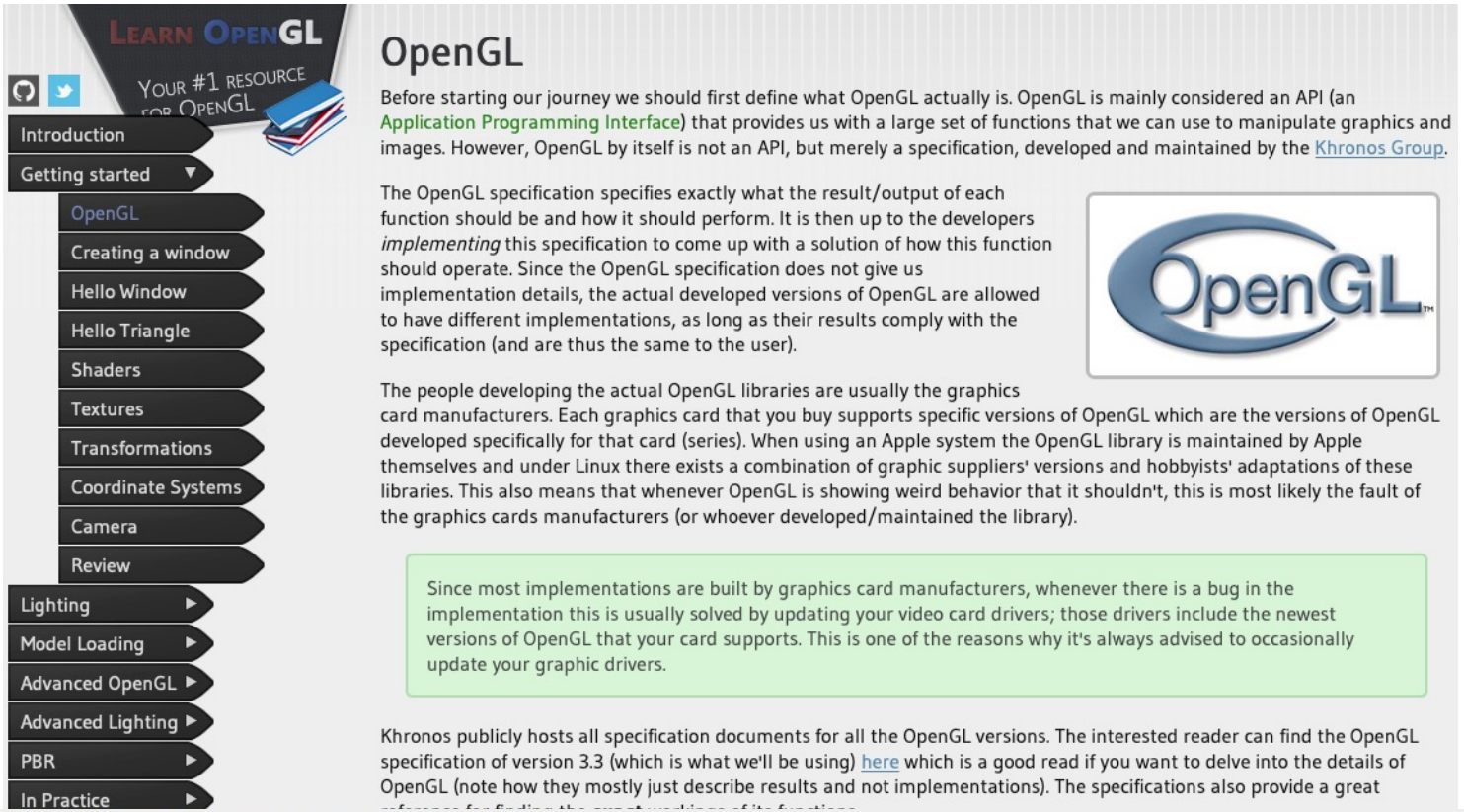
- https://learnopengl.com/Getting-started/OpenGL
  - Intro
  - Core-profile vs Immediate mode
  - State machine
  - Objects

- https://learnopengl.com/Getting-started/Hello-Triangle
  - Intro
  - Vertex Input
  - (Vertex Shader, Fragment Shader)
  - Linking Vertex Attributes
  - Vertex Array Object
  - The triangle we‘ve all been waiting for

Web-Rundtour (2): Was von OpenGL benötigen wir?

- https://learnopengl.com/Getting-started/Shaders
  - Intro
  - Types
  - Ins and outs
  - Uniforms
  - More attributes!

- https://learnopengl.com/Getting-started/Textures
  - Intro
  - Texture Wrapping
  - Texture Filtering (without Mipmaps)
  - Generating a texture
  - Applying textures
  - Texture Units



Core-profile vs Immediate mode

In the old days, using OpenGL meant developing in immediate mode (often referred to as the fixed function pipeline) which was an easy-to-use method for drawing graphics. Most of the functionality of OpenGL was hidden inside the library and developers did not have much control over how OpenGL does its calculations. Developers eventually got hungry for more flexibility and over time the specifications became more flexible as a result; developers gained more control over their graphics. The immediate mode is really easy to use and understand, but it is also extremely inefficient. For that reason the specification started to deprecate immediate mode functionality from version 3.2 onwards and started motivating developers to develop in OpenGL's core-profile mode, which is a division of OpenGL's specification that removed all old deprecated functionality.

https://learnopengl.com

Besprechung Übung 1: Datenquelle

Für die Datenerzeugung in createData() bedienen wir uns einer Funktion, die in der Grafik- und Visualisierungsforschung weit verbreitet ist. Der „synthetische Tornado“ von Prof. Roger Crawfis (Ohio State University) erzeugt ein einfaches Vektorfeld („Wind“) mit drei Raumkomponenten auf einem kartesischen Gitter. Der C-Code (Achtung: kein C++) ist hier verfügbar: http://web.cse.ohio-state.edu/~crawfis.3/Data/Tornado/

- Wie sind die Funktionsargumente dieser Funktion gestaltet?
- Wie können wir diese Funktion in unser Programm integrieren, möglichst ohne sie zu verändern?
- Welche Array-Repräsentation wählen wir am besten für unser kartesisches Gitter cartesianDataGrid, um darin Werte aus der Crawfis-Funktion zu speichern (wie wird das Array in der Crawfis-Funktion beschrieben)? Welche Auswirkung hat das auf die Implementierung von getDataValue?
- Ziel: Für ein Gitter von 16^3 Gitterpunkten, welche Werte enthält Layer 10?

Lösung Übung 1: Ausgabe Layer 10 (X-Komponente):

-0.0843209	-0.0790582	-0.0720934	-0.0635956	-0.0539166	-0.0436149	-0.0334217	-0.0241202	-0.0163447	-0.0103752	-0.00603471	-0.00275505	0.000231973	0.00368401	0.00820563	0.0141962
-0.0818559	-0.0747353	-0.0656496	-0.0547291	-0.0423585	-0.0292563	-0.0164792	-0.00341018	0.00672456	0.0112185	0.0114309	0.0122398	0.0124416	0.0131565	0.0151862	0.0189873
-0.0777421	-0.0688216	-0.0575784	-0.044024	-0.0284905	-0.00487042	0.017493	0.0353892	0.0467109	0.0504546	0.0472596	0.0390582	0.0281652	0.0245515	0.0238244	0.0253666
-0.0727162	-0.0622371	-0.0490191	-0.0317419	-0.0040213	0.0244702	0.051466	0.0734571	0.0866887	0.0892179	0.0822939	0.0694478	0.0543856	0.0396848	0.0329283	0.0323655
-0.0675303	-0.0559846	-0.0413302	-0.0150164	0.0151602	0.0481329	0.0815163	0.110067	0.126649	0.127026	0.113963	0.0944578	0.0744171	0.0568251	0.0425709	0.0389082
-0.062849	-0.0510093	-0.0321982	-0.0059714	0.0252421	0.0622116	0.103798	0.143345	0.166565	0.162596	0.138719	0.11005	0.0851083	0.0659254	0.051968	0.0439591
-0.0591116	-0.0479652	-0.0292299	-0.00608744	0.0228718	0.060438	0.10962	0.167382	0.206338	0.19162	0.147861	0.109223	0.0825092	0.065163	0.0541982	0.0475389
-0.0564027	-0.0465237	-0.0323423	-0.0154089	0.00605166	0.0356125	0.0811225	0.158243	0.244754	0.193487	0.120668	0.0827983	0.0638001	0.0542065	0.0497545	0.0483725
-0.054409	-0.047225	-0.0394454	-0.0312313	-0.0222669	-0.0118327	0.00213798	0.0288361	0.191629	0.0529733	0.0316057	0.0294178	0.0317708	0.0358864	0.040868	0.0463386
-0.0525269	-0.0483196	-0.0471625	-0.0480876	-0.0525199	-0.0635068	-0.0886686	-0.150078	-0.256902	-0.16242	-0.0732544	-0.0291946	-0.00254172	0.0163829	0.0312934	0.0438708
-0.0501064	-0.0474839	-0.0518983	-0.0600084	-0.0738027	-0.0966206	-0.133396	-0.185323	-0.220881	-0.185711	-0.119294	-0.0652913	-0.026369	0.00249826	0.0250483	0.0435277
-0.0467186	-0.0451115	-0.0510675	-0.0632667	-0.0808764	-0.105304	-0.136407	-0.167717	-0.181256	-0.161515	-0.11903	-0.0735846	-0.0341329	-0.00171264	0.0249459	0.0466236
-0.0423152	-0.040723	-0.0436235	-0.0571201	-0.0745385	-0.0956411	-0.118227	-0.136448	-0.141368	-0.127328	-0.0980468	-0.0624639	-0.0273777	0.0042987	0.0320124	0.0509977
-0.0372302	-0.0347958	-0.0352351	-0.0427284	-0.0579993	-0.0746004	-0.090155	-0.10063	-0.101418	-0.0900453	-0.0681081	-0.0399898	-0.0100246	0.0190188	0.0426581	0.0571547
-0.0320702	-0.028209	-0.0268046	-0.0279096	-0.0343833	-0.0468197	-0.0572113	-0.0629303	-0.0614451	-0.0515216	-0.0339481	-0.0110625	0.0144368	0.0366078	0.0516069	0.0641242
-0.0275719	-0.0219666	-0.0184344	-0.0168735	-0.0169033	-0.017756	-0.0216761	-0.0243237	-0.021461	-0.0124096	0.00239983	0.0204946	0.0347159	0.048355	0.0605444	0.0708218

Besprechung Übung 1: Zusatzfragen

Das von der Crawfis-Funktion erzeugt Datenfeld enthält synthetische „Windgeschwindigkeiten“ in den drei Raumrichtungen.

- Wie unterscheiden sich die drei Windkomponenten in ihrer Größenordnung?
- Schreibe eine Funktion, die den Betrag der 3D-Windgeschwindigkeit berechnet. Wie unterscheidet sich diese Größe von den einzelnen Komponenten?
- Wie groß sind Minimum und Maximum der 3D-Geschwindigkeit bei Zeitschritt t=0?
- Wie ändern sich diese Werte mit der Zeit?

OpenGL in Qt: Template

- Was benötigen wir für ein erstes Visualisierungs-Programm?

  1. OpenGL Viewport
  2. Transformation aus einem „World space“ in den „Clip space“ (model-view-projection Matrix)
  3. Vertices definieren und an die GPU senden
  4. Texturen definieren und an die GPU senden

- Template für die Übung: Beispiel für Punkte 1, 2, 3.
  - Siehe zip-Datei im Moodle.
  - Das Template definiert einen OpenGL Viewport, erzeugt Geometriedaten (Vertices) für einen „Würfel“ („Bounding Box“ für unsere Daten), und implementiert eine (sehr) simple Maus-Interaktion, um den Würfel zu drehen.

Geometriedaten im Template: C++ Teil

- Geometriedaten können in Qt komfortabel mit QOpenGLBuffer verwendet werden : https://doc.qt.io/qt-5/qopenglbuffer.html

```
float mydata[numEntries] = ..; // some data array

QOpenGLBuffer vertexBuffer(QOpenGLBuffer::VertexBuffer);
vertexBuffer.create();
vertexBuffer.bind();
vertexBuffer.allocate(myData, numEntries * sizeOf(float));
vertexBuffer.release();

// .. store config in "vertex array object" for straightforward access
```

Erstellen eines Vertex Buffers aus einem Array.

```
shaderProgram.bind();
vertexArrayObject.bind();
// .. set more inputs for shader program execution

// .. OpenGL draw commands to execute graphics pipeline
```

Verwendung eines Vertex Buffers beim Rendering.

Geometriedaten im Template: GLSL Teil

- Der Vertex Shader braucht derzeit nur die Transformation in den Clip Space vornehmen:

```
uniform mat4 mvpMatrix; // model-view-projection matrix
in vec4 vertexPosition;

void main()
{
    gl_Position = mvpMatrix * vertexPosition;
}
```

- Der Fragment Shader färbt alle Fragmente identisch ein:

```
layout(location = 0) out vec4 fragColor;

void main()
{
    fragColor = vec4(1, 0, 0, 1);
}
```

Texturen in Qt (betrachte als „GPU-Arrays“)

- Texturen können in Qt komfortabel mit QOpenGLTexture verwendet werden: https://doc.qt.io/qt-5/qopengltexture.html

```
QImage img = ..; // some image

QOpenGLTexture texture(QOpenGLTexture::Target2D);
texture.create();
texture.setWrapMode(QOpenGLTexture::ClampToEdge);
texture.setData(img);

// .. do stuff ..

texture.destroy(); // destroy at end of program
```

Erstellen einer Textur aus einem QImage.

```
const int textureUnit = 0; // select a texture unit
texture.bind(textureUnit);
shaderProgram.setUniformValue("colorMappingTexture", textureUnit);

// .. OpenGL draw commands ..
```

Verwendung einer Textur beim Rendering.

Texturen: GLSL Teil

- Der Vertex Shader muss nun zusätzlich noch Texturkoordinaten für jeden transformierten Vertex ausgeben:

```
...
smooth out vec2 texCoord;

void main()
{
    gl_Position = mvpMatrix * vertexPosition;
    texCoord = ...;
}
```

- Der Fragment Shader führt einen „look-up“ in der Textur durch:

```
uniform sampler2D colorMappingTexture;
smooth in vec2 texCoord;
layout(location = 0) out vec4 fragColor;

void main()
{
    fragColor = texture2D(colorMappingTexture, texCoord);
}
```

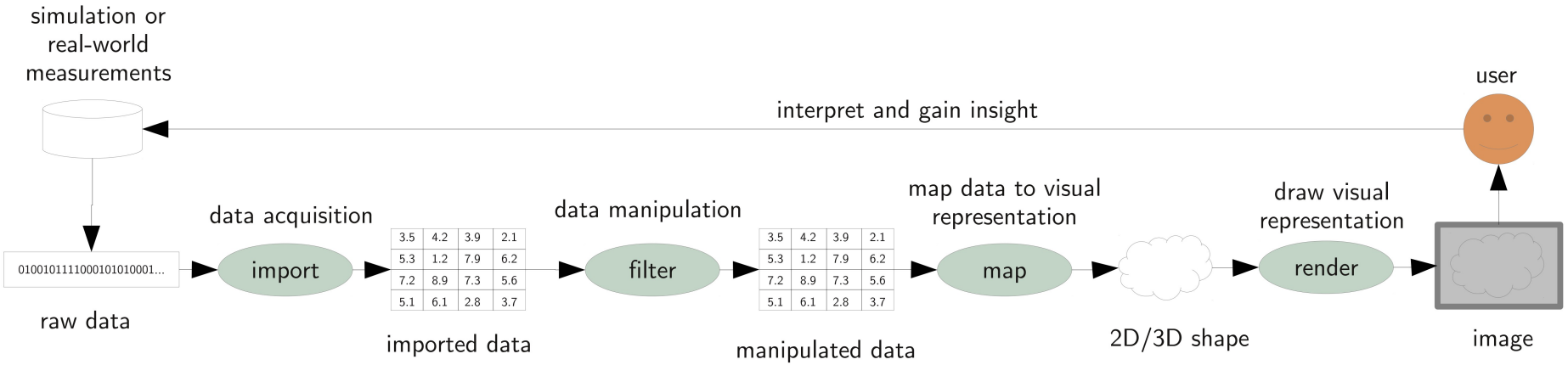
Übung 2: Colour Mapping

Als erste Visualisierungsmethode für unser Datenquelle wollen wir ein simples „Colour Mapping“ für einen 2D-Schnitt durch das 3D Datengitter implementieren. Die Skalarwerte der Komponenten der synthetischen Windgeschwindigkeit werden dabei durch eine Farbtabelle (Colourmap) in einen Farbwert überführt. Das erzeugte Bild wird mittels OpenGL dargestellt.

Ziel der Übung ist eine erste einfache Implementierung einer kompletten Visualisierungspipeline:

- Data Source: aus Übung 1.
- Mapper: Extraktion eines Schnitts aus dem Datengitter und Überführung in ein Bild.
- Rendering: Via OpenGL.
- Image presentation: Übernimmt Qt für uns.

Übung 2: Colour Mapping



Übung 2: Colour Mapping

- Verwende das Qt-OpenGL-Template.
- Füge die Datenquelle aus Übung 1 hinzu.
  - Wir fügen alle Elemente der Pipeline zum `OpenGLDisplayWidget` hinzu, also auch eine Instanz der Datenquelle.
  - Die Datenquelle kann in der Methode `initVisualizationPipeline()` des Widgets initialisiert werden. Zum Testen kann hier die Textausgabe aus Übung 1 aufgerufen werden.
- Füge dem Projekt zwei neue Klassen `HorizontalSliceToImageMapper` und `HorizontalSliceRenderer` hinzu.
  - Auch von diesen Klassen werden Instanzen in `OpenGLDisplayWidget` angelegt, sie werden in `initVisualizationPipeline()` erzeugt und im Destruktor freigegeben. Der Rendering-Aufruf erfolgt analog zum Aufruf der Bounding-Box-Methode.

Übung 2: `HorizontalSliceToImageMapper`

In einer klassischen Pipeline-Architektur werden die einzelnen Module „hintereinandergeschaltet“.

- Unser Mapper bekommt dafür einfacherweise eine Methode `setDataSource()`, die als Argument einen Zeiger auf eine Datenquelle erhält, hinter die sich das Modul in der Pipeline „hängt“.
- Als Ausgabe soll der Mapper ein Bild eines angefragten Gitterlevels ausgeben: Eine Methode „`QImage mapSliceToImage(int iz)`“ soll, analog zu der Textausgabe in Übung 1, ein Bild erstellen und als Typ „`QImage`“ zurückgeben.
  - Verwende vorerst nur die Komponente „0“ (also „x-Wind“) der Daten.
  - Wir verwenden eine einfache Transferfunktion: Werte über 0 werden auf den roten Farbkanal gemappt, Werte unter 0 auf den blauen. Eine brauchbare Skalierung auf Rotwerte/Blauwerte im Bereich 0..255 erhält man, wenn die Fließkommawerte mit dem Faktor „3*255“ skaliert.
  - Tipp: Verwende den Typ `QColor` sowie die Methode `setPixelColor()` von QImage.

Übung 2: `HorizontalSliceRenderer`

Der Renderer für das im Mapper erstellte Bild wird am besten analog zum Renderer für die Bounding Box erstellt.

- Für den Aufbau der Pipeline bietet sich eine Methode „`setMapper()`“ an, die als Argument einen Zeiger auf einen Mapper bekommt.
- Eine Methode „`drawImage()`“ zeichnet, analog zur Bounding Box, das Bild in den OpenGL Viewport.
  - Als ersten Schritt zeichne nur den Rahmen für das Bild. Dies benötigt nur minimale Änderungen vom Bounding-Box-Code (copy&paste…).
  - Welcher Code (C++ sowie GLSL im Shader) ist nötig, um das Bild vom Mapper in eine Textur zu überführen und darzustellen (zunächst nur für den Schnitt bei iz = 0)?
  - Tipp: Die Vertex-Koordinaten können in diesem einfachen Fall als Texturkoordinaten „recycelt“ werden, so dass keine zusätzlichen Texturkoordinaten an den Vertex-Shader übergeben werden müssen.

Übung 2: Datenexploration

- Wenn das Bild sichtbar ist, füge eine Methode „`moveSlice(int steps)`“ hinzu, die die z-Position des Schnittes verändert. Diese Methode muss also die Geometrie des Slices aktualisieren sowie ein neues Bild vom Mapper anfordern. Die Methode soll über die Cursor-Tasten (up/down) gesteuert werden (siehe `OpenGLDisplayWidget::keyPressEvent()`).
  - Tipps: Die Daten eines Vertex Buffers können einfach mit „`allocate()`“ überschrieben werden. Bei Verwendung einer Textur mit `QImage` muss die Textur hingegen erst „zerstört“ und neu erstellt werden, bevor „`setData()`“ wieder angewendet werden kann.

Sobald der Schnitt sichtbar und verschiebbar ist, erkunde das Datenfeld:

- Was verrät die Visualisierung über die Windgeschwindigkeiten (in x-Richtung) im Feld?
- Wie ändert sich die Geschwindigkeit mit der Höhe?