Computergrafik I

Einführung in modernes OpenGL (3.3 Core)

Was ist OpenGL?

Was ist OpenGL?

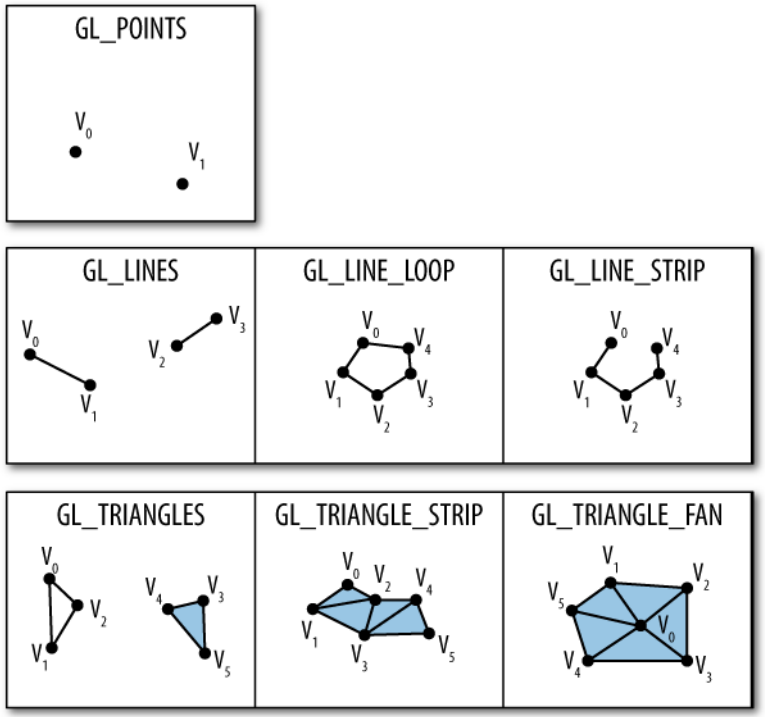
- Programmierschnittstelle zur Entwicklung von 2D/3D-Grafikanwendungen
- Plattform- und programmiersprachenunabhängig
- Besteht aus mehr als 250 Befehlen
- Keine high-level Funktionalität
- OpenGL 4.5 (seit 2014 aktuelle Version)

Grundlegende Vorgehensweise

4 Schritte zum Erfolg

1. Konstruktion

- Konstruktion von Objekten aus Primitiven
- OpenGL 3.3 bietet **Punkte**, **Linien** und **Dreiecke** als Primitiven



Quelle: https://www.safaribooksonline.com/library/view/iphone-3d-programming/9781449388133/ch02s02.html

2. Anordnung der Objekte im 3D-Raum

- Mathematische **Transformationen** (Verschiebung, Rotation, Skalierung, etc.) (Model-Matrix)
- Positionierung und Ausrichtung einer „gedachten“ **Kamera** (View-Matrix)
- **Darstellungsart** wählen (Orthogonal oder Perspektivisch) (Projection-Matrix)
- **Vertex shader**

3. Rasterisierung

- Herunterbrechen von Primitiven in sogenannte Fragmente (Pixel mit Zusatzinformationen)

4. Berechnung von Farbe und Tiefe

- Kann explizit zugewiesen werden
- Kann durch spezielle **Beleuchtungsmodelle** berechnet werden
- Durch Aufbringen einer sogenannten **Textur**
- Oder eine Kombination dieser Vorgehensweisen
- **Fragment shader**

Weitere mögliche Schritte

- Elimination von Teilen der Objekte, die durch andere Objekte verdeckt werden (**Culling**),
- Kantenglättung (**Antialiasing**),
- **Tesselation**
- Etc.

Wissenswertes

Begriffe, Datentypen, etc.

Begriffe

- Begriffe:
  - **Rendering**: Der Prozess, bei dem der Computer Bilder aus Modellen (Objekten) erzeugt (Mathematische Darstellung → Pixel)
  - **Modell**: Wird aus geometrischen Primitiven konstruiert
  - **Vertex** (pl. **Vertices**): Typischerweise ein Punkt im 3D-Raum
  - **Pixel**: Ein Punkt im 2D-Raum

Datentypen (1/2)

- Die an manche Kommandos angehängten Buchstaben deuten die verwendeten Datentypen an (z.B. `glUniform3f`, `glUniform4ui`, `glTexParameteri`, etc.)
- Um Portierbarkeit zu gewährleisten, benutzt OpenGL eigene Datentypen

Datentypen (2/2)

Suffix	Datentyp	C/C++	OpenGL
b	8-bit integer	signed char	GLbyte
s	16-bit integer	short	GLshort
i	32-bit integer	int oder long	GLint, GLsizei
f	32-bit floating-point	float	GLfloat
d	64-bit floating-point	double	GLdouble
ub	8-bit unsigned integer	unsigned char	GLubyte, GLboolean
us	16-bit unsigned integer	unsigned short	GLushort
ui	32-bit unsigned integer	unsigned int	GLuint, GLenum, GLbitfield

OpenGL als Statusmaschine

- OpenGL kann in verschiedene Zustände (Modes) versetzt werden
  - Jeder Zustand bleibt solange aktiv bis er geändert wird
  - Z.B. verwendet jedes Objekt solange die zuletzt zugewiesene Textur bis eine neue gesetzt wird (vereinfacht dargestellt)
- Weitere Zustände: Textur, Lichtquellen, Linienstärke, Punktgröße, Vertex Buffer Object, Framebuffer, Shader Program, etc.
- Viele Zustände werden mit `glEnable(...)` und `glDisable(...)` an- und ausgeschaltet.
- Auch das Binden von Objekten verändert den Zustand: `glBindFramebuffer`, `glBindTexture`, `glUseProgram`, etc.
- Abfrage der zugehörigen Zustandsvariable mit `glIsEnabled(...)` oder z.B. `glGetIntegerv()`

Bibliotheken

GLFW, GLEW und GLM

GLFW (OpenGL Contex/Window Toolkit)

- Bietet Funktionen zur Erzeugung eines OpenGL-Fensters
- Bietet Funktionen zur Abfrage von Tastastur-, Fenster- und Mausereignissen
- Plattformunabhängig
- http://www.glfw.org

GLEW (OpenGL Extension Wrangler Library)

- Ermöglicht das Laden der einzelnen OpenGL-Funktionen
- Plattformunabhängig
- http://glew.sourceforge.net

GLM (OpenGL Mathematics)

- An modernes OpenGL angepasste Mathematikbibliothek
- Bietet alles für Vektoren und Matrizen bzgl. Computergrafik
- Klassen und Funktionen ähnlich benannt wie in GLSL
- Bringt viele nützliche Funktionen mit
  - Funktion zum Erstellen einer Projektionsmatrix (Perspektivisch oder Orthogonal)
  - Funktionen zur Projektion von 2D → 3D und zurück
  - Funktionen zur Schnittpunktberechnung
  - Etc.
- http://glm.g-truc.net

OpenGL-Programm

Ein einfaches OpenGL-Programm auf Basis von GLFW, GLEW und GLM

GLFW, GLEW (1/2)

- `int glfwInit()`: Initialisiert die GLFW-Bibliothek.
- `void glfwWindowHint(int target, int hint)`: Kann unter anderem benutzt werden um festzulegen, welche OpenGL-Version beim Erstellen des OpenGL-Fensters benutzt werden soll.
- `GLFWwindow* glfwCreateWindow(...)`: Erstellt das Fenster.
- `void glfwTerminate()`: Schließt unter anderem alle offenen OpenGL-Fenster und gibt Ressourcen frei.
- `void glfwMakeContextCurrent(GLFWwindow* window)`: Sorgt dafür, das im aktuellen Thread der aktuelle OpenGL-Kontext genutzt wird.
- `GLenum glewInit()`: Lädt die OpenGL-Erweiterungen (Funktionen).

GLFW, GLEW (2/2)

- `void glfwWindowShouldClose(GLFWwindow* window)`: Überprüft, ob das Fenster geschlossen werden soll (z. B. Benutzer klickt auf Fenster schließen).
- `void glfwSwapBuffers(GLFWwindow* window)`: Tauscht front- und backbuffer aus.
- `void glfwWaitEvents()`: Legt den aktuellen Thread so lange schlafen, bis ein Ereignis auftritt (zum Beispiel eine Zeicheneingabe).

Eventhandling (GLFW)

- `GLFWwindowsizefun glfwSetWindowSizeCallback(GLFWwindow* window, GLFWwindowsizefun cbfun)`: Setzt die Funktion, die bei jeder Größenveränderung des Fensters aufgerufen werden soll
  - `typedef void(* GLFWwindowsizefun)(GLFWwindow *, int, int)`
- `GLFWcharfun glfwSetCharCallback(GLFWwindow* window, GLFWcharfun cbfun)`: Setzt die Funktion, die der Eingabe von Zeichen aufgerufen werden soll
  - `typedef void(* GLFWcharfun)(GLFWwindow *, unsigned int)`

Init-, Release- und Renderfunktionen

- `bool init()`: Wird bei der Initialisierung des Programms aufgerufen (nach der Erstellung des OpenGL-Kontexts). Hier können Daten wie zum Beispiel Modelle und Texturen geladen werden oder wichtige OpenGL-Zustände verändert werden.
- `void release()`: Wird beim Beenden des Programms und in einigen Fehlerfällen aufgerufen. Dient dem Freigeben von Ressourcen.
- `void render()`: Wird Aufgerufen, wenn der Inhalt des Fensters gezeichnet werden soll.

Vertex Buffer Objects (VBO) (1/3)

- Dienen dazu, Vertex-Daten im VRAM der Grafikkarte abzulegen
- Vertex-Daten setzen sich aus diversen Attributen zusammen. Für das Rendering interessant sind z.B. folgende:
  - position (x, y, z)
  - color (r, g, b)
  - normal (x, y, z)
  - texture coordinate (u, v)
  - etc.
- Inhalt der VBOs werden von Vertex Shadern verarbeitet

Vertex Buffer Objects (VBO) (2/3)

- Zwei Möglichkeiten zur Handhabung von Vertex-Daten
  - ein VBO pro Attribut (position, color, normal, etc.), oder
  - ein VBO für alle Vertex-Attribute (interleaving)
- Ein VBO für jedes Attribut ist einfacher zu implementieren
- Ein VBO für alle Attribute hat Performancevorteile (cache friendly)

Vertex Buffer Objects (VBO) (3/3)

Position Buffer (z.B. `std::vector<glm::vec3>`)

Index	0	1	2	...	n
Inhalt	Vertex	Vertex	Vertex	...	Vertex

Color Buffer:

Index	0	1	2	...	n
Inhalt	Color	Color	Color	...	Color

Normal Buffer:

Index	0	1	2	...	n
Inhalt	Normal	Normal	Normal	...	Normal

Index Buffer:

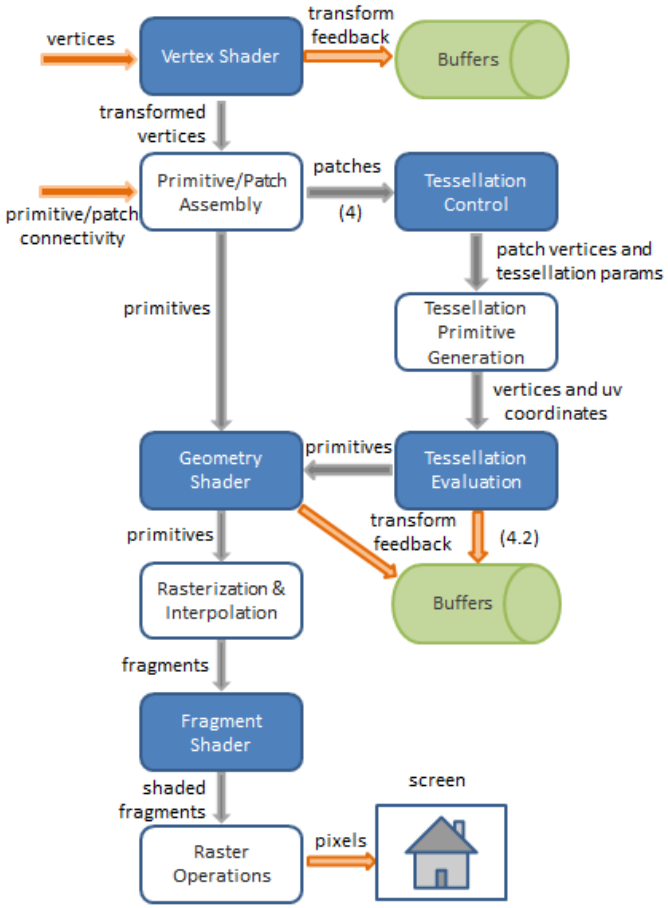
Index	0	1	2	3	4	5	...
Inhalt	0	1	2	0	2	3	...

Vertex Array Objects (VAO)

- Das Arbeiten mit VBOs alleine ist mit viel Aufwand verbunden.
- Vor dem Rendern müssen diese aktiviert sein und mit dem entsprechenden Shader verknüpft sein (Stichwort „state machine“).
- VAOs merken sich alle Aktivierungen und Verknüpfungen
- Man muss daher nur einmal die Aktivierung und Verknüpfung tätigen und muss danach nur noch das VOA aktivieren und rendern

Shader (1/3)

- **Shader** stellen einige Stufen der Grafikpipeline dar
- Verarbeiten Daten
- **Vertex Shader** transformieren Vertices (grob gesagt) (→ NDC)
- **Fragment Shader** erzeugen die endgültige Farbe eines Pixels (grob gesagt)
- Weitere Shader-Typen vorhanden
- **Shader Programme** verknüpfen Shader und übergeben Daten
- Klasse **GLSLProgram**



Quelle: http://www.lighthouse3d.com/tutorials/glsl-tutorial/pipeline33/

Shader (2/3)

- **Shader** verarbeiten die Daten aus den Vertex Buffer Objects
- Über das **Shader Program** können „globale“ Variablen für Shader gesetzt werden (z.B. `glUniform3f(...)`). Beispiele für solche globalen Variablen:
  - Position oder Richtung einer Lichtquelle
  - Ein Zeitwert für Animationen innerhalb von Shadern
  - Faktor für die Oberflächenbeschaffenheit des Phong-Beleuchtungsmodells
  - Etc. pp.

Shader (3/3)

Vertex Shader

```
#version 330

in vec3 position;
in vec3 color;
in vec3 normal;

uniform mat4 mvp;
uniform mat3 nm;
uniform vec3 lightDirection;

out vec3 fragmentColor;

void main()
{
   vec3 n  = normalize(nm * normal);
   float intensity = max(dot(n, lightDirection), 0.0);
   fragmentColor = color * intensity;
   gl_Position   = mvp * vec4(position,  1.0);
}
```

Fragment Shader

```
#version 330

out vec3 fragColor;

in vec3 fragmentColor;

void main()
{
    fragColor = fragmentColor;
}
```

Sourcecode

- Siehe Praktikum

Objekt-Transformationen

Objekt-Transformationen

```
// Ein einfaches Beispiel eine Objekt-Transformation
glm::mat4 model(1.0f);

model = glm::translate(model, 0.0f, 0.0f, -8.0f);
model = glm::rotate(model, 45.0f, glm::vec3(1.0f, 0.0f, 0.0f));
model = glm::scale(model, glm::vec3(0.5f));

// bind shader program
program.bind();

// calculate model view projection matrix (mvp)
glm::mat4 mvp = projection * view * model;

// set mvp as uniform to the shader program
program.setUniform("mvp", mvp);

// render object
glBindVertexArrayObject(vao);
glDrawElements(…);
glBindVertexArrayObject(0);
```

Transformationen und Projektionen (1/3)

- Modelle müssen im 3D-Raum platziert bzw. bewegt werden (Transformationen) und die 3D-Szene in die 2D Bildschirmebene projiziert werden.
- Zur Berechnung werden 4x4 Matrizen benutzt (warum und wie genau Kapitel 4).
- Aufteilung der Transformation in **projection**, **view** und (mehrere) **model** Matrizen (siehe Kapitel 3)

Transformationen und Projektionen (2/3)

- Model matrix für ein Objekt kann sich aus anderen model Matrizen zusammensetzen
- Beispiel:
  - Ein Auto hat vier gleiche Räder, von denen jedes mit je 5 gleichen Schrauben an der Achse befestigt wird.
  - Es gibt also sinnvollerweise je eine Routine die eine Schraube bzw. ein Rad (ohne Schraube) in sog. lokalen Koordinaten zeichnet.
  - Wenn das Auto gezeichnet wird, soll also die Rad-Zeichen-Routine viermal mit einer jeweils anderen Transformation (Verschiebung) aufgerufen werden. Mit dem Zeichnen jedes Rades soll analog die Schrauben-Zeichen-Routine 5-mal aufgerufen werden.

Transformationen und Projektionen (3/3)

- Nur Räder und Chassis zeichnen
  - 1. Zeichne das Chassis
  - 2. Merke die momentane Position (Mitte)
  - 3. Verschiebe zum rechten vorderen Rad
  - 4. Zeichne ein Rad
  - 5. Vergesse die letzte Verschiebung und gehe zurück (zur Mitte)
  - 6. Merke die momentane Position (Mitte)
  - 7. Verschiebe zum linken vorderen Rad
  - 8. Zeichne ein Rad
  - 9. ...

Hilfreiche Ressourcen

- Bibliotheken
  - http://glew.sourceforge.net/
  - http://www.glfw.org/
  - http://glm.g-truc.net/
- Dokumentation OpenGL
  - http://docs.gl/ (wir verwenden 3.3 core)
- Tutorials
  - https://open.gl/
  - http://learnopengl.com/
  - http://www.opengl-tutorial.org/
- Modeller
  - https://www.blender.org