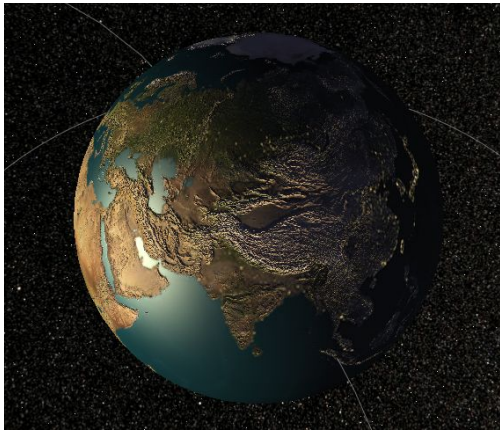
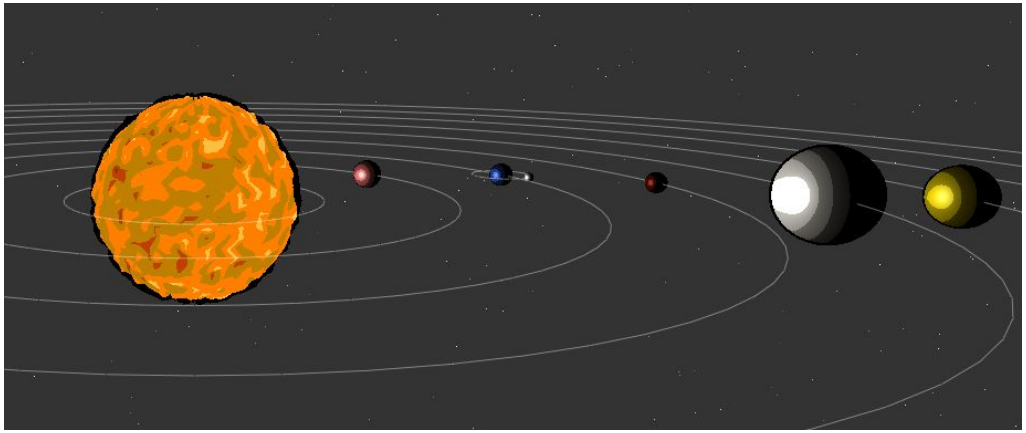




Computer Graphics Exercise

2. OpenGL Introduction

Jakob Wagner

OpenGL

- is a **specification** for a 3D graphics API
- the **implementation** is contained in display drivers (NVIDIA, AMD, Intel, Mesa)

  -> each driver can react differently to incorrect use

- is object oriented and based on a state machine -> no direct object manipulation
- very strong optimisation and prediction through implementation to reduce driver overhead
- cumbersome to program -> wrapper classes are very helpful to prevent repeating commands

- superseded by Vulkan with more explicit object and memory management
- Vulkan is even more cumbersome to program and not supported by older GPUs -> not used in exercise

OpenGL History

1992 - 1.0 immediate mode, fixed function rendering
-> vertex specification when drawing, automatic shading

2003 - 1.5 (vertex) buffer objects
-> store (vertex) data permanently on GPU memory

2004 - 2.0 introduction GLSL, programmable vertex and fragment shaders
-> shading stages can be programmed in C-like language

2008 - 3.0 Vertex Array Objects, Framebuffer Objects
-> VAOs enclose geometry definition, Framebuffers allow rendering to offscreen buffers

2009 - 3.2 update to GLSL, used in framework
-> shading language easier to understand

https://en.wikipedia.org/wiki/OpenGL
http://stevenhoward.org/opengl/versions.html

OpenGL History

1992 - 1.0

Immediate Mode: specify vertices when drawing them

application_fixed.cpp

```
glBegin(GL_TRIANGLES)   // start vertex stream for triangles
glVertex3f(...)         // 1st vertex attributes
glColor3f(..)
...
glVertex3f(..)          // kd vertex attributes
glEnd()                 // finish vertex stream
```

Fixed Function: vertex transformation and shading is done automatically
only transformation matrices are specified

```
glMatrixMode(GL_PROJECTION)   // activate modification of projection matrix
glLoadMatrixf(...)            // load new projection matrix
glMatrixMode(GL_MODELVIEW)    // modification of model and view matrix
```

OpenGL History

2003 - 1.5

Vertex Buffer Objects: store vertex data in buffer on GPU memory for quick drawing

application_vbo.cpp

```
glGenBuffers(1, &m_vertex_bo)                          // generate generic buffer
glBindBuffer(GL_ARRAY_BUFFER, m_vertex_bo)             // bind this as an vertex data buffer
glBufferData(GL_ARRAY_BUFFER, size, data, usage)       // upload data to current buffer
...
glBindBuffer(GL_ARRAY_BUFFER, m_vertex_bo)             // bind vertex buffer for drawing
glEnableClientState(GL_VERTEX_ARRAY)             // enable vertex position reads from buffer
glVertexPointer(components, format, stride, offset)    // set layout of vertex positions
...    //enable and format other attributes
glDrawArrays(GL_TRIANGLES, offset, number)             //draw vertices form buffer
```

OpenGL History

2003 - 1.5

Index Buffer Objects: define the indices of vertices to draw

application_indexed.cpp

draw the same vertex multiple times without repeating the data in the vertex buffer

```
glGenBuffers(1, &index_bo)                          // generate generic buffer
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, index_bo)     // bind this as an index data buffer
glBufferData(GL_ELEMENT_ARRAY_BUFFER, size, data, usage) // upload data to current buffer
...
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, index_bo)     // bind index buffer for drawing
...    // enable vertex attributes
glDrawElements(GL_TRIANGLES, number, format, offset)  // draw indexed vertex data
```

OpenGL History

2004 - 2.0

Shader Programs: programmable vertex and fragment processing stages

application_shader.cpp

shading stages can be programmed in C-like shading language GLSL

```
glUseProgram(shader_program)        // bind shader for drawing
...                                 // draw vertices
```

In vertex shader:

```
//perform transformation manually
gl_Position = gl_ProjectionMatrix * gl_ModelViewMatrix * vec4(in_Position, 1.0)
```

In fragment shader:

```
out_Color = vec4(pass_Color, 1.0)   // set output color manually
```

OpenGL History

2004 - 2.0

Uniforms: transfer variables to shaders which are constant during drawing

application_uniform.cpp

each uniform name has a unique location in each shader program

```
//get location of uniform called “ProjectionMatrix”
GLint location = glGetUniformLocation(shader_program, "ProjectionMatrix")

glUseProgram(shader_program)        // bind shader for upload
glUniform*(location, ..., value)    // upload value to location in bound shader
```

In shader:

```
uniform mat4 ProjectionMatrix       // use like normal variable
```

OpenGL History

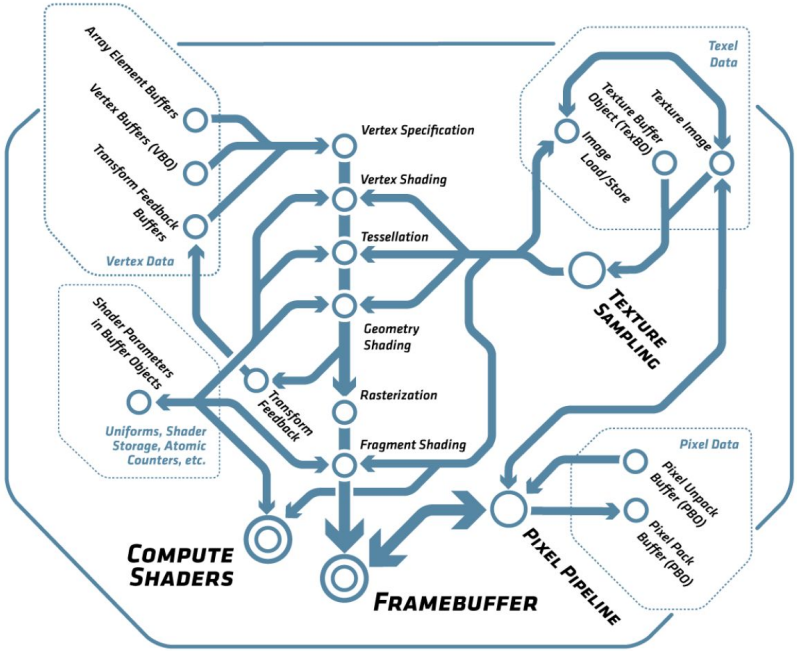
2008 - 3.0

Vertex Array Objects: encapsulate vertex data, attributes and indices within one object

application_vao.cpp

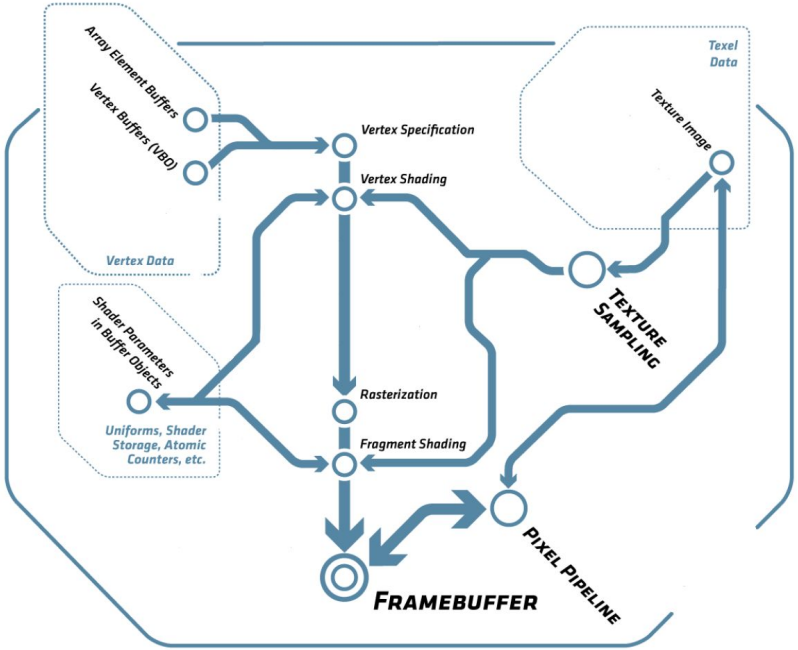
```
glBindBuffer(GL_ARRAY_BUFFER, vertex buffer)          // bind buffer with vertex data

glBindVertexArray(vertex_array)           // bind VAO for attribute and index attaching

glEnableVertexAttribArray(index)          // activate an attribute
// attach bound vertex buffer to bound VAO and define attribute format
glVertexAttribPointer(index, components, format, GL_FALSE, stride, offset)

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, index_buffer)   // bind index buffer to VAO

...

glBindVertexArray(vertex_array)           // bind VAO for drawing
// draw indexed vertex data from bound VAO as triangles using bound shader program
glDrawElements(GL_TRIANGLES, number, format, offset);
```

Pipeline (complete)



https://flzen.wordpress.com/2014/12/22/opengl-4-5-pipeline-from-khronos-and-related-chart/

Pipeline (relevant)



Assignments

2. **Vertex Specification**

   define the elements being drawn

3. **Shader Programs**

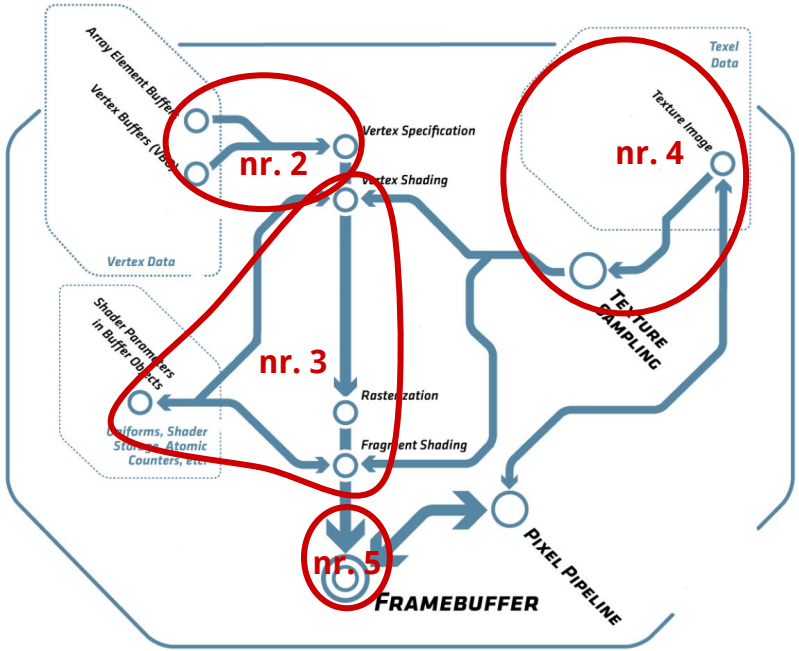
   define how to draw elements

4. **Texture Objects**

   define and use textures

5. **Framebuffer Object**

   define on what to draw elements



Framework

- on Github: https://github.com/wobakj/OpenGLFramework
- installation instructions on https://github.com/wobakj/OpenGLFramework/wiki/Installation
- tested on Linux (makefile), Windows (MSVC 2013) and Mac OS (CLion, XCode, makefile)
- requires CMake 2.8.12 and OpenGL 3.2
- pressing “R” during run-time reloads the shaders
- Pressing “Q” or “ESC” closes the application
- supports import of .obj models and png & tga textures (but not all compressions)
- path to resources can be set in first command line argument

Troubleshooting

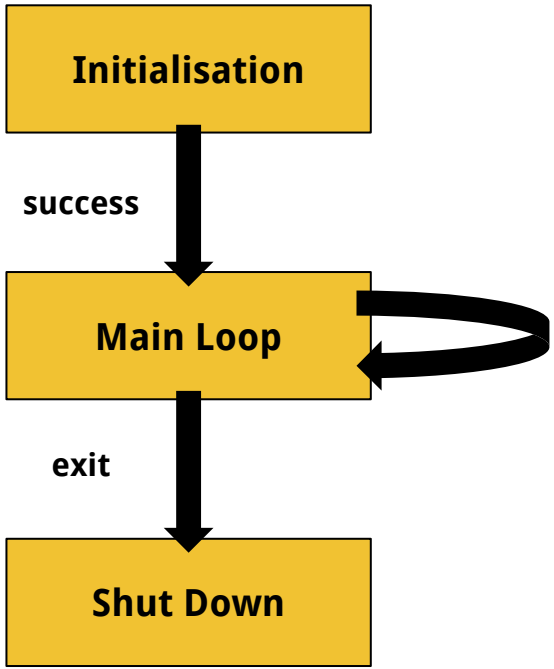
throws runtime_error when a gl* function fails and outputs function, error and arguments

- find function entry in the function reference and check reasons for error
- backtrace crash point with gdb: enter “gdb application name”, then run by entering “r”
- when crashed, enter “bt” to backtrace exception, find mentioned function name and corresponding source file and line

throws exception and outputs error message when shader compilation or linking fails

- when load at startup fails, logic_error is thrown
- when loading with “R” during runtime fails, no exception is thrown and shader can be reloaded after problems was fixed
- error message displays sourcefile, line and error

Application flow

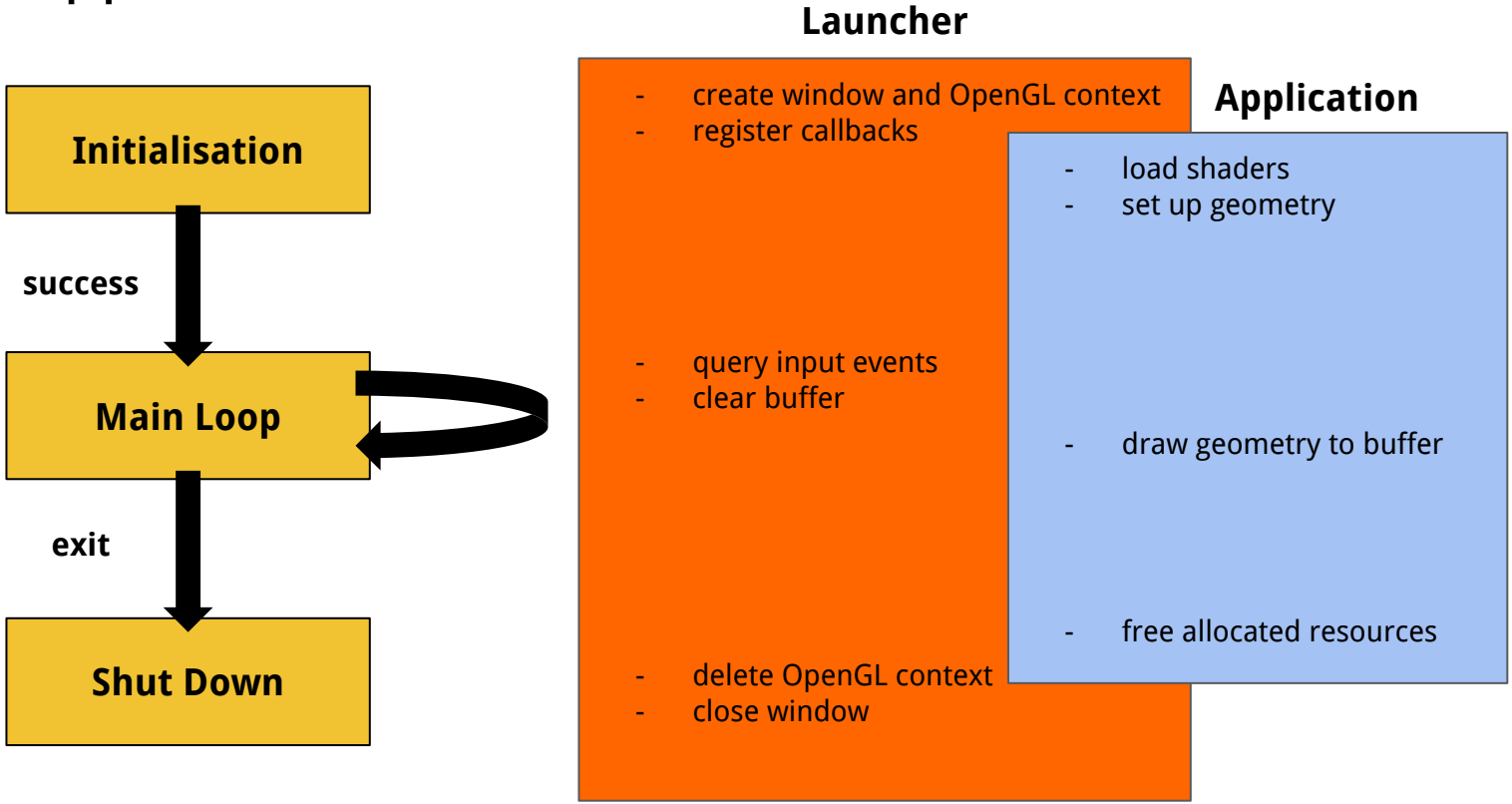


- create window and OpenGL context
- register callbacks
- load shaders
- set up geometry

- query input events
- clear buffer
- draw geometry to buffer

- free allocated resources
- delete OpenGL context
- close window

Application flow



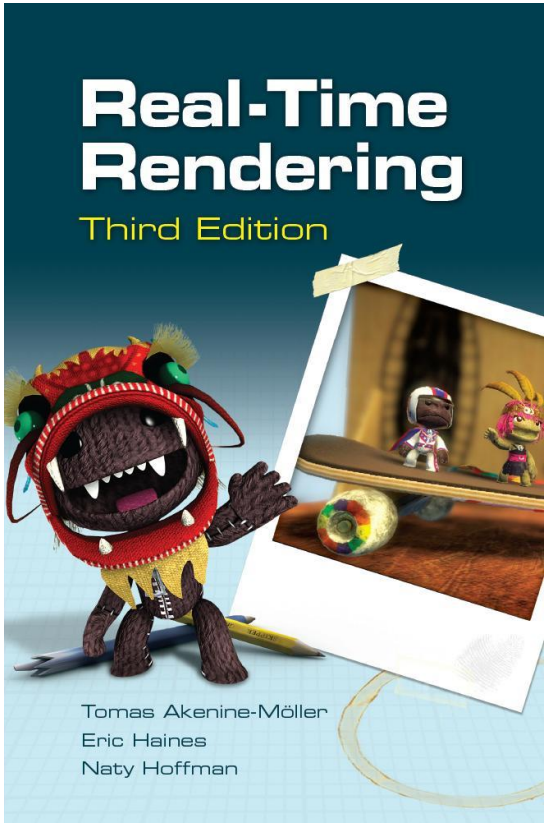
Assignment 1 – Planet Addition

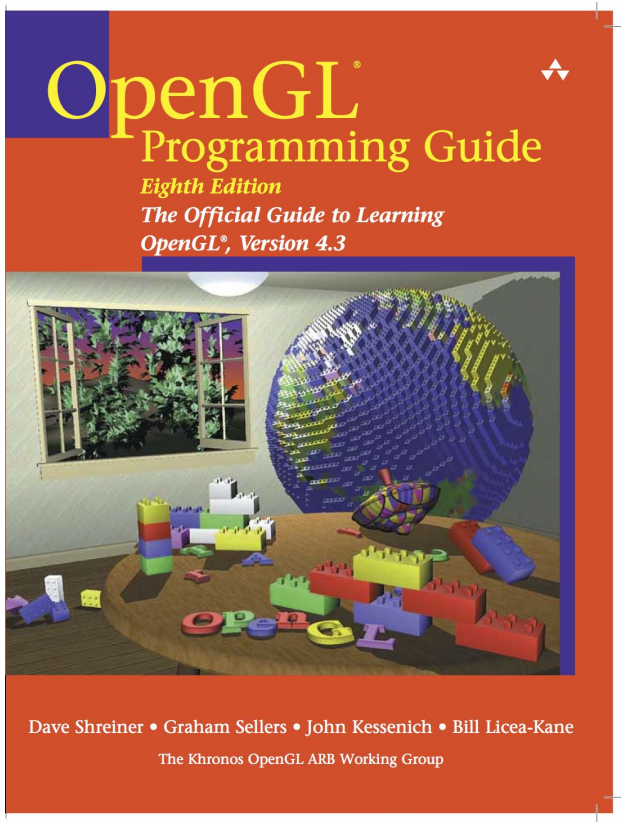
- all planets are simple spheres -> can use the same geometry
- difference between planets: size, rotation speed and distance to origin (all model transformations)

  -> draw same geometry for each planet, with a modified Model Matrix:

  1. implement planet struct which holds values for the properties above
  2. store planet structs in a container as member of the applicationSolar class
  3. remove Model- and Normal Matrix calculation from `render()` function
  4. add new function `upload_planet_transforms()` taking a planet struct as argument, which calculates and uploads the Model- and Normal Matrix
  5. render method, iterate over the planet container and call `upload_planet_transforms()` and `render()` for each planet

Books


Real-Time Rendering
Third Edition
Tomas Akenine-Möller
Eric Haines
Naty Hoffman


OpenGL®
Programming Guide
Eighth Edition
The Official Guide to Learning
OpenGL®, Version 4.3
Dave Shreiner • Graham Sellers • John Kessenich • Bill Licea-Kane
The Khronos OpenGL ARB Working Group

Websites

- OpenGL Wiki explains all objects and operations in detail
- OpenGL reference pages explain all functions with arguments and possible errors
- OpenGL specification is the official reference on how the API works
- GLSL specification is the official reference on the shading language