Introduction to OpenGL 3.x and Shader-Programming using GLSL
Part 1

Ingo Radax,
Günther Voglsam

Institute of Computer Graphics and Algorithms
Vienna University of Technology
Topics for today

- OpenGL 3.x and OpenGL Evolution
- OpenGL-Program-Skeleton and OpenGL-Extensions, GLEW
- State-machines and OpenGL-objects life-cycle
- Introduction to Shader-Programming using GLSL
OpenGL 3.x
What is OpenGL?

- An open industry-standard API for hardware accelerated graphics drawing
- Implemented by graphics-card vendors
- As of 10\textsuperscript{th} March 2010:
  - Current versions: OpenGL 4.0, GLSL 4.0
- Bindings for lots of programming-languages:
  - C, C++, C#, Java, Fortran, Perl, Python, Delphi, etc.
What is OpenGL?

- Maintained by the Khronos-Group [2]:
  - Members:
What is OpenGL?

Pros & Cons:

- Full specification freely available
- Everyone can use it
- Can use it anywhere (Windows, Linux, Mac, BSD, Mobile phones, Web-pages (soon), …)
- Long-term maintenance for older applications
- New functionality usually earlier available through Extensions
- Inclusion of Extensions to core may take longer
- Game-Industry
Setup OpenGL Project

- Include OpenGL-header:

  ```
  #include <GL/gl.h>  // basic OpenGL
  ```

- Link OpenGL-library “opengl32.lib”

- If needed, also link other libraries (esp. GLEW, see later!).
OpenGL in more detail

- **OpenGL-functions prefixed with “gl”:**

  ```
glFunction{1234}{bsifd...}{v}(T arg1, T arg2, ...);
  ```

  Example: `glDrawArrays(GL_TRIANGLES, 0, vertexCount);`

- **OpenGL-constants prefixed with “GL_”:**

  ```
  GL_SOME_CONSTANT
  ```

  Example: `GL_TRIANGLES`

- **OpenGL-types prefixed with “GL”:**

  ```
  GLtype
  ```

  Example: `GLfloat`
OpenGL in more detail

- OpenGL is a **state-machine**
- Remember state-machines:
  - Once a state is set, it remains active until the state is changed to something else via a transition.
  - A transition in OpenGL equals a function-call.
  - A state in OpenGL is defined by the OpenGL-objects which are current.
OpenGL in more detail

- Set OpenGL-states:

```c
glEnable(...);
glDisable(...);
gl*(...);  // several call depending on purpose
```

- Query OpenGL-states with Get-Methods:

```c
glGet*(...);  // several calls available, depending on what to query
```

- For complete API see [3] and especially the quick-reference [4]!

  ◆ Note: In the references the gl-prefixes are omitted due to readability!
OpenGL 2.1

- Released in August 2006
- Fully supported “fixed function” (FF) *
- GLSL-Shaders supported as well
- Mix of FF and shaders was possible, which could get confusing or clumsy quickly in bigger applications
- Supported by all graphics-drivers

*) See “Introduction to Shader-Programming using GLSL” for more information on FF.
Open GL 3.0

- Released in August 2008
- Introduced a deprecation model:
  - Mainly FF was marked deprecated
  - Use of FF still possible, but not recommend
- Also introduced Contexts:
  - Forward-Compatible Context (FWD-CC) vs. Full Context
- With FWD-CC, no access to FF anymore, i.e. FF-function-calls create error “Invalid Call”.

Institute of Computer Graphics and Algorithms
Furthermore, GLSL 1.3 was introduced
Supported by recent Nvidia and ATI-graphics drivers.
OpenGL 3.1

- Released in March 2009
- Introduced GLSL 1.4
- Removed deprecated features of 3.0, but FF can still be accessed by using the “GL_ARB_compatibility”-extension.
- Supported by recent Nvidia and ATI-graphics drivers.
OpenGL 3.2

- Released in August 2009
- Profiles were introduced:
  - Core-Profile vs.
  - Compatibility-Profile
- With Core-Profile, only access to OpenGL 3.2 core-functions
- With Compatibility-Profile, FF can still be used
- Also introduced GLSL 1.5
- Supported by recent Nvidia and ATI-graphics drivers.
OpenGL 3.3

- Released on 10th March 2010
- Introduces GLSL 3.3
- Includes some new Extensions
- Maintains compatibility with older hardware
- Currently no drivers available
- Will be supported by Nvidia’s Fermi architecture immediately when Fermi will be released (scheduled: March 29th 2010).
OpenGL 4.0

- Released on 10\textsuperscript{th} March 2010
- Introduces GLSL 4.0
- Introduces new shader-stages for hardware-tesselation.
- Adoption of new Extensions to Core.
- Currently no drivers available
- Will be supported by Nvidia’s Fermi architecture immediately when Fermi will be released (scheduled: March 29\textsuperscript{th} 2010).
Overview of the evolution:

◆ FF equals roughly in other versions:

<table>
<thead>
<tr>
<th>2.1</th>
<th>3.0</th>
<th>3.1</th>
<th>3.2/3.3/4.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>FF</td>
<td>Deprecated Features and Non-FWD-CC</td>
<td>&quot;GL_ARB_compatibility&quot; extension</td>
<td>Compatibility-Profile</td>
</tr>
</tbody>
</table>

Important!

See the Quick-Reference Guide [4] for the “current” (=3.2) OpenGL-API!
OpenGL Evolution

- Note that from OpenGL 3.x (FWD-CC || Core) onwards there is no more built-in:
  - Immediate-Mode
  - Matrix-Stacks and Transformations
  - Lighting and Materials

- You have to do “missing” stuff by yourself!

- That’s why there are shader. (More on shader later on.)
OpenGL Extensions

- Extensions are additional and newer functions which are not supported by the core of the current OpenGL-version.

- Collected and registered in the OpenGL Extension Registry [5].

- Extensions may eventually be adopted into the OpenGL core at the next version.
On Windows only OpenGL 1.1 supported natively.

To use newer OpenGL versions, each additional function, i.e. ALL extensions (currently ~1900), must be loaded manually!

→ Lots of work!

Therefore:
GLEW

Include it in your program and initialize it:

```c
#include <GL/glew.h>  // include before other GL headers!
#include <GL/gl.h>    // included with GLEW already

void initGLEW()
{
    GLenum err = glewInit();  // initialize GLEW

    if (err != GLEW_OK)  // check for error
    {
        cout << "GLEW Error: " << glewGetErrorString(err);
        exit(1);
    }
}
```
GLEW

- Check for supported OpenGL version:

```c
if (glewIsSupported("GL_VERSION_3_2"))
{
    // OpenGL 3.2 supported on this system
}
```

- To check for a specific extension:

```c
if (GLEW_ARB_geometry_shader4)
{
    // Geometry-Shader supported on this system
}
```
If OpenGL 3.x context can not be created on your hardware one can use 2.1 without the „fixed function“-pipeline:

- Be sure to use the latest drivers, libs et al and test if our OpenGL 3.x demo is running!
- If it doesn‘t work out, you can use OpenGL 2.1 w/o FF.
- This means…
Do **NOT** use the following in OpenGL 2.1:

- **Built-In matrix-functions/stacks:**
  - `glMatrixMode`, `glMult/LoadMatrix`, `glRotate/Translate/Scale`, `glPush/PopMatrix`...

- **Immediate Mode:**
  - `glBegin/End`, `glVertex`, `glTexCoords`...

- **Material and Lighting:**
  - `glLight`, `glMaterial`, ...

- **Attribute-Stack:**
  - `glPush/PopAttrib`, ...
No-FF in OpenGL 2.1

- some Primitive Modes:
  - GL_QUAD*, GL_POLYGON

- Do NOT use the following in GLSL 1.1/1.2:
  - ftransform()
  - All built-in gl_*-variables, except:
    - gl_Position in vertex-shader
    - gl_FragColor, gl_FragData[] in fragment-shader
The list may not be complete!

To see what can be used and what not, see the quick-reference guide [4]!
Everything written in **black** is allowed; **blue** is not allowed. (But we will not be too strict about that in CG2LU.)

If you are not sure what you can use, do it the way it works for you and ASK US in the forum or by PM.
Be sure to use the most recent version working on your hardware (and use: no FF || no deprecation || Full-Context || Core-Profile)!

Be sure to see the 8-page Quick-Reference Guide [4] for the current OpenGL-API!

Use the (complete) specification [3] for detailed information on a particular OpenGL-method!
References


OpenGL Program-Skeleton
Typical OpenGL-program runs in a window (maybe fullscreen)
Therefore: window-loop-based applications
Independent of window-manager!
- Can use: GLFW, SDL, WinAPI, (GLUT), Qt, ...
- Choose the one you like most.
- We recommend using GLFW [1]. For more information about GLFW check the LU-HP [2]!
Typical OpenGL-Application:

1. Start Application
   - `main()`

2. Main-loop
   - `init()`
   - `update()`
   - `render()`
   - `cleanup()`

3. Exit application
OpenGL Program Skeleton

- main():
  - Program-Entry
  - Create window
  - Call init()
  - Start main window-loop
  - Call cleanup()
  - Exit application

- init():
  - Initialize libraries, load config-files, …
  - Allocate resources, preprocessing, …
OpenGL Program Skeleton

- **update()**:  
  - Handle user-input, update game-logic, …

- **render()**:  
  - Do actual rendering of graphics here!  
  - Note: Calling render() twice without calling update() in between should result in the same rendered image!

- **cleanup()**:  
  - Free all resources
Example `init()`-function:

```c
void init() {
    Create and initialize a window with depth-buffer and double-buffering. See your window-managers documentation.

    // enable the depth-buffer in OpenGL
    glEnable(GL_DEPTH_TEST);

    // enable back-face culling in OpenGL
    glEnable(GL_CULL_FACE);

    // define a clear color
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);

    // set the OpenGL-viewport
    glViewport(0, 0, windowWidth, windowHeight);

    Do other useful things
}
```
OpenGL Program Skeleton

- The geometry of a 3D-object is stored in an array of vertices called Vertex-Array.
- Each vertex can have so called Attributes, like a Normal Vector and Texture-Coordinates.
- OpenGL also treats vertices as attributes!
- To render geometry in OpenGL, vertex-(attribute)-arrays are passed to OpenGL and then rendered.
OpenGL Program Skeleton

To do so:

- Query the attribute-location in the shader: *

  GLint vertexLocation = glGetAttribLocation(
    myShaderProgram, "in_Position");

- Enable an array for the vertex-attribute:

  glEnableVertexAttribArray(vertexLocation);

- Then tell OpenGL which data to use:

  glVertexAttribPointer(vertexLocation, 3, GL_FLOAT,
    GL_FALSE, 0, myVertexArray);

*) See “Introduction to Shader-Programming using GLSL” for more information on shader attribute-variables.
OpenGL Program Skeleton

- Draw ("render") the arrays:

```c
glDrawArrays(GL_TRIANGLES, 0, 3); // this does the actual drawing!
```

- Finally disable the attribute-array:

```c
glDisableVertexAttribArray(vertexLocation);
```

- See the demo on the LU-HP for full program and code!
OpenGL Program Skeleton

Example render()-function:

```
// triangle data
static GLfloat vertices[] = {-0.5, -0.333333, 0, // x1, y1, z1
  +0.5, -0.333333, 0, // x2, y2, z2
  +0.0, +0.666666, 0}; // x3, y3, z3

void render() {
  // clear the color-buffer and the depth-buffer
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  // activate a shader program
  glUseProgram(myShaderProgram);

  // Find the attributes
  GLint vertexLocation = glGetAttribLocation(
    myShaderProgram, "in_Position");
```

Institute of Computer Graphics and Algorithms
// enable vertex attribute array for this attribute
glEnableVertexAttribArray(vertexLocation);

// set attribute pointer
glVertexAttribPointer(vertexLocation, 3, GL_FLOAT, GL_FALSE, 0, vertices);

// Draw ("render") the triangle
glDrawArrays(GL_TRIANGLES, 0, 3);

// Done with rendering. Disable vertex attribute array
glDisableVertexAttribArray(vertexLocation);

// disable shader program
glUseProgram(0);

Swap buffers

}
In OpenGL, all objects, like buffers and textures, are somehow treated the same way.

On object creation and initialization:

- First, create a *handle* to the object (in OpenGL often called a *name*). Do this ONCE for each object.
- Then, *bind* the object to make it current.
- *Pass data* to OpenGL. As long as the data does not change, you only have to do this ONCE.
- *Unbind* the object if not used.
On rendering, or whenever the object is used:

- **Bind** it to make it current.
- **Use** it.
- **Unbind** it.

Finally, when object is not needed anymore:

- **Delete** object.
  Note that in some cases you manually have to delete attached resources!

NOTE: OpenGL-objects are **NOT** objects in an OOP-sense!
References


Introduction to Shader Programming using GLSL
What shaders are

- Small C-like programs executed on the graphics-hardware
- Replace fixed function pipeline with shaders
- Shader-Types
  - Vertex Shader (VS): per vertex operations
  - Geometry Shader (GS): per primitive operations
  - Fragment shader (FS): per fragment operations
- Used e.g. for transformations and lighting
Shader-Execution model

Shader-Source-Code → Application

OpenGL-API

OpenGL-Driver

Compiler → Shader-Object

Linker → Program-Object

compiled code

executable code

Graphics-Hardware

Institute of Computer Graphics and Algorithms
OpenGL 3.x Rendering-Pipeline:

1. Geometry → Vertex-Shader
2. Vertex-Shader → Primitive Assembly
3. Primitive Assembly → Clip
4. Clip → Project
5. Project → Viewport
6. Viewport → Cull
7. Programmable!
8. Rasterize → Fragment-Shader
9. Fragment-Shader → Per Fragment Operations
10. Per Fragment Operations → Framebuffer Operations
11. Framebuffer Operations → Framebuffer

Hardware (GPU)
Remember:

- The *Vertex-Shader* is executed ONCE per each vertex!
- The *Fragment-Shader* is executed ONCE per rasterized fragment (~ a pixel)!

A *Shader-Program* consists of both,

- One VS
- One FS
Setting up shaders and programs

- **Compile shaders:**

  ```c
  char* shaderSource;  // contains shadersource
  int shaderHandle = glCreateShader(GL_SHADER_TYPE);
      // shader-types: vertex || geometry || fragment
  glShaderSource(shaderHandle, 1, shaderSource, NULL);
  glCompileShader(shaderHandle);
  ``

- **Create program and attach shaders to it:**

  ```c
  int programHandle = glCreateProgram();
  glAttachShader(programHandle, shaderHandle);  // do this
  for vertex AND fragment-shader (AND geometry if needed)!
  ``

- **Finally link program:**

  ```c
  glLinkProgram(programHandle);
  ```

Institute of Computer Graphics and Algorithms
Enabling shaders

Enable a GLSL program:

```c
glUseProgram(programHandle); // shader-program now active
```

- The active shader-program will be used until `glUseProgram()` is called again with another program-handle.
- Call of `glUseProgram(0)` sets no program active (undefined state!).
Do this for each shader to check for error:

```c++
bool succeeded = false;
glGetShader(shaderHandle, GL_COMPILE_STATUS, &succeeded);

if (!succeeded) // check if something went wrong while compiling
{
    // get log-length
    int logLength = 0;
glGetShader(shaderHandle, GL_INFO_LOG_LENGTH, &logLength);

    // get info-log
    std::string infoLog(logLength, '');
glGetStringInfoLog(shaderHandle, logLength, NULL, &infoLog[0]);

    // print info-log
    std::cout << "Shader compile error:
    " << infoLog << std::endl;
}
```
**Program Error checking**

Do this for each program to check for error:

```cpp
bool succeeded = false;
glGetProgram(programHandle, GL_LINK_STATUS, &succeeded);

if (!succeeded) // check if something went wrong while compiling
{
  // get log-length
  int logLength = 0;
glGetProgram(programHandle, GL_INFO_LOG_LENGTH, &logLength);

  // get info-log
  std::string infoLog(logLength, '');
glGetProgramInfoLog(programHandle, logLength, NULL,
                      &infoLog[0]);

  // print info-log
  std::cout << "Program linking error:\n\n" << infoLog << std::endl;
}
```
Basic shader layout

- Shader-Programs must have a `main()`-method
- Vertex-Shader outputs to at least `gl_Position`
- Fragment-Shader to custom defined output

```cpp
//preprocessor directives like:
#define version 150

variable declarations

void main()
{
    do something and write into output variables
}
```
Shader Parameter

Shader variable examples:

uniform mat4 projMatrix;  // uniform input
in vec4 vertex;           // attribut-input
out vec3 fragColor;      // shader output

Three types:

- **uniform**: does not change per primitive; read-only in shaders
- **in**: VS: input changes per vertex, read-only; FS: interpolated input; read-only
- **out**: shader-output; VS to FS; FS output.
Set uniform parameters in an application:

```c
// first get location
projMtxLoc = glGetUniformLocation(programHandle, "projMatrix");

// then set current value
glUniformMatrix4fv(projMtxLoc, 1, GL_FALSE, currentProjectionMatrix);
```

- First get the „location“ of the uniform-variable
- Then set the current value
- Can pass values to vertex- and fragment-shader
A vertex can have attributes like a normal-vector or texture-coordinates

OpenGL also treats the vertex itself as an attribute

We want to access our current vertex within our vertex-shader (as we used to do with gl_Vertex in former GLSL-versions):

- Therefore, we declare in our vertex-shader:

```glsl
in vec4 vertex; // vertex attribute
```
■ Now, there are two ways to pass data to this shader attribute-variable, depending on:
  ◆ if you just have an array of vertices (*Vertex Array*),
  ◆ or an VBO (*Vertex Buffer Object*, more about that next week!).

■ To do so: Query shader-variable location
  ◆ Enable vertex-attribute array
  ◆ Set pointer to array
  ◆ Draw and disable array
For a Vertex-Array, pass data like this:

```c
// first get the attribute-location
vertexLocation = glGetAttribLocation(programHandle, "vertex");

// enable an array for the attribute
glEnableVertexAttribArray(vertexLocation);

// set attribute pointer
glVertexAttribPointer(vertexLocation, 3, GL_FLOAT, GL_FALSE, 0, myVertexArray);

// Draw ("render") the triangle
glDrawArrays(GL_TRIANGLES, 0, 3);

// Done with rendering. Disable vertex attribute array
glDisableVertexAttribArray(vertexLocation);
```
Setting attribute parameters with VBOs:

```cpp
// first get location
vertexLocation = glGetAttribLocation(programHandle, "vertex");

// activate desired VBO
glBindBuffer(GL_ARRAY_BUFFER, vertexBuffer);

// set attribute-pointer
glVertexAttribPointer(vertexLocation, 4, GL_FLOAT, GL_FALSE, 0, 0);

// finally enable attribute-array
glEnableVertexAttribArray(vertexLocation);
...
```
Since GLSL 1.3, `gl_FragColor` is deprecated. Therefore, need to define output on our own.

Declare output variable in FS:

```cpp
out vec4 fragColor; // fragment color output
```

In the application, **before** linking the shader-program with `glLinkProgram()`, bind the FS-output:

```cpp
glBindFragDataLocation(programHandle, 0, "fragColor");
```

Finally assign a value to `fragColor` in the FS.
Example usage

An application using shaders could basically look like this:

Load shader and initialize parameter-handles

Do some useful stuff like binding texture, activate texture-units, calculate and update matrices, etc.

```gl
glUseProgram(programHandle);
```

Set shader-parameters

Draw geometry

```gl
glUseProgram(anotherProgramHandle);
```

...
Conclusion and Tips

- Setup is more complicated nowadays, but more flexible.
- Use the info-log to debug!
- Use tools like gDebugger (see some LU-HP and forum!) for better debugging!
- See the specifications [1] for exact information on methods!
- Look at useful examples at [2]!
- Have fun with OpenGL! 😊
Resources

- OpenGL „Red Book“
- OpenGL „Orange Book“
- Norbert Nopper, [http://nopper.tv/opengl_3_2.html](http://nopper.tv/opengl_3_2.html)
- NeHe, [http://nehe.gamedev.net](http://nehe.gamedev.net)
- GameDev, [http://www.gamedev.net](http://www.gamedev.net)
- Graphic Remedy‘s gDEBugger, [http://www.gremedy.com](http://www.gremedy.com)

We have a academic license for it, so USE it!!
Questions?

Thanks for your time!