OpenGL 3.x Part 2: Textures and Objects

Ingo Radax,
Günther Voglsam

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

- OpenGL 3.x Part 1 - Revisited
- Textures
- Framebuffer Objects
- Vertexbuffer Objects
- Vertex Array Objects
- Uniform Buffer Objects
- Notes on CG2
Set up OpenGL-Project
Setup OpenGL Project

- Set up a MSVC-project as explained in the C++-lecture

- Version 1:
  - Include OpenGL-header:
    ```
    #include <GL/gl.h>  // basic OpenGL
    ```
  - Link OpenGL-library "opengl32.lib"
  - Bind extensions manually
  - Cumbersome!
Better: Version 2:

- Include GLEW-header:
  ```
  #include <GL/glew.h>  // GLEW
  ```

- Link OpenGL-library “opengl32.lib” and “glew32.lib”
- Copy “glew32.dll” to bin folder
- U’r ready to go. 😊
OpenGL-Object life-cycle revisited
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.
OpenGL-Object life-cycle

- 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!
GLSL Shader revisited
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

1. Shader-Source-Code
2. Application
3. OpenGL-API

- **OpenGL-Driver**
  - Compiler
  - Linker
  - Shader-Object
  - Program-Object
  - compiled code
  - executable code

- Graphics-Hardware
OpenGL 3.x Rendering-Pipeline:
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
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.
  
  `glUseProgram(programHandle);`

- Set shader-parameters
- Draw geometry
  
  `glUseProgram(anotherProgramHandle);`

  ...
Textures
Why Texturing?

- Idea: enhance visual appearance of plain surfaces by applying fine structured details
Textures

- First things first:
  - Load image-data from a file or
  - Generate it (i.e. procedurally)

- Use Library to read data from files:
  - GLFW: glfw.sourceforge.net
  - Devil: openil.sourceforge.net

- Enable Texturing in OpenGL:

```c
// enable 2D-texturing
glEnable(GL_TEXTURE_2D);
```
Textures

- As usual in OpenGL:
  - Create texture-handle
  - Bind texture-handle to make it current
  - Pass data to OpenGL (next slide)

```c
GLuint textureHandle; // variable for our texture-handle

// get one texture-handle
glGenTextures(1, &textureHandle);

// bind texture
glBindTexture(GL_TEXTURE_2D, textureHandle); // could also be 1D, 3D, ...
```
Use `glTexImage* (...)` to pass loaded image-data stored in `data` to OpenGL.

If `data` is a null-pointer, the needed memory on the GPU will be allocated.

```c
int mipLevel = 0; int border = 0;
int internalFormat = GL_RGBA,
int width = 800; int height = 600;
int format = GL_RGBA;
int type = GL_UNSIGNED_BYTE;

// pass data for a 2D-texture
glTexImage2D(GL_TEXTURE_2D, mipLevel, internalFormat, width, height, border, format, type, data);
```
As usual in OpenGL:

- After using it, don’t forget to unbind
- Finally, if not needed anymore, delete the texture

```c
// unbind texture
glBindTexture(GL_TEXTURE_2D, 0);

...  

// delete texture
glDeleteTextures(1, &textureHandle);
```
Problem: One pixel in image space covers many texels

Solution: Mipmaps
Mip-Maps

- (Pre-)Calculate different Levels of detail:
  - From original size (level 0) down to size of 1x1 pixel
- After data has been passed to OpenGL:
  - Use `glGenerateMipmap(…)` to generate a set of mipmaps for currently bound texture

```c
// generate mipmaps for current bound 2D-texture
glGenerateMipmap(GL_TEXTURE_2D);
```
Texture Parameters

- **Magnification-Filter:**
  - **Nearest** vs. **Linear**

  // set filter-mode for currently bound 2D-texture
  
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, filter);

  For filter-types see specification!
Texture Parameters

- Minification-Filter:
  - Without Mipmaps:
    - GL_
  - With Mipmaps:
    - GL_*_MIPMAP_*
  - where * = NEAREST || LINEAR
- Recommended:
  - Mipmaps with GL_LINEAR_MIPMAP_LINEAR

```c
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, filter);
```
Texture Parameters

- Wrap and clamp:
  - GL_CLAMP, GL_REPEAT, GL_CLAMP_TO_BORDER, GL_CLAMP_TO_EDGE, GL_MIRRORED_REPEAT

```c
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_* , filter);
// * = S || T || R
```

![Images of different texture wrapping modes](image_url)
Passing Textures to Shader

- Use different texture-units for different textures
- Use `uniform sampler*` variables in shader to access texture-units

```c
// get location of sampler
GLuint texLocation = glGetUniformLocation(programHandle, "colorTexture");

// activate the texture-unit to which the texture should be bound to
glActiveTexture(GL_TEXTURE0 + textureUnit);
glBindTexture(GL_TEXTURE_2D, textureHandle);

// pass the texture unit (i.e., it's id) to the shader
glUniform1i(texLocation, textureUnit);
```
// Textures can be accessed with samplers
uniform sampler2D colorTexture;

// to access textures, coordinates are needed
in vec2 texCoord;

...

void main(void)
{
    ...

    // Access texture at specified coordinates
    vec4 texel = texture2D(colorTexture, texCoord);

    ...
}
Cleaning Up

- If texture is not needed anymore, delete it

```c
glDeleteTextures(1, &texId); // delete texture
```

- References
Framebuffer Objects
FBOs
What are FBOs used for?

- **“Normal” rendering**

```glsl
// GL Program
glBindBuffer(GL_ARRAY_BUFFER, vboHandle);
glVertexAttribPointer(vertexLocation, 4, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(vertexLocation);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vboHandle);
glDrawElements(GL_TRIANGLES, 3, GL_UNSIGNED_INT, 0);
glDisableVertexAttribArray(vertexLocation);
glBindBuffer(GL_ARRAY_BUFFER, 0);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
```

- **With FBO**

```glsl
// GL Program
glBindBuffer(GL_ARRAY_BUFFER, vboHandle);
glVertexAttribPointer(vertexLocation, 4, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(vertexLocation);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vboHandle);
glDrawElements(GL_TRIANGLES, 3, GL_UNSIGNED_INT, 0);
glDisableVertexAttribArray(vertexLocation);
glBindBuffer(GL_ARRAY_BUFFER, 0);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
```
What are FBOs used for?

- Shadow Mapping
- Bloom
- HDR
- Motion Blur
- Depth of Field
- ...

Institute of Computer Graphics and Algorithms
What is an FBO?

- FBO is an encapsulation of attachments
- Attachments can be color- or renderbuffers
- Renderbuffers are objects that support off-screen rendering without an assigned texture
  - Depth- and stencil-buffer
- There can be more then one color attachment
  - Number depends on your HW
  - More than one is advanced stuff
Generating an FBO is done as usual in OpenGL:

- First generate an OpenGL-”name”
- Then bind it to do something with it

```c
GLuint fbo; // this will store our fbo-name

// generate fbo
glGenFramebuffers(1, &fbo);

// bind FBO
glBindFramebuffer(GL_FRAMEBUFFER, fbo);
```
An FBO on its own isn’t much

Therefore: attach renderable objects

So we want to add a depth buffer

Again, create name and bind it:

```c
GLuint depthbuffer; // this will store our db-name

// create a depth-buffer
glGenRenderbuffers(1, &depthbuffer);

// bind our depth-buffer
glBindRenderbuffer(GL_RENDERBUFFER, depthbuffer);
```
Creating storage-space

- We didn’t create any storage for our render-buffer yet, so create it…
- …and attach it to our FBO

```c
// create storage for our renderbuffer
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT, width, height);

// attach renderbuffer to FBO
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_RENDERBUFFER, depthbuffer);
```
Attaching a texture to the FBO

- To render to a texture, we first need one
- We create it as usual
- Note: width and height are the same as those for the FBO and renderbuffers!

```cpp
// create a texture
GLuint img;

glGenTextures(1, &img);
glBindTexture(GL_TEXTURE_2D, img);

glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, width, height, 0, GL_RGBA, GL_UNSIGNED_BYTE, NULL);
```
\[// \text{attach texture to fbo}\]

```c
glfwFramebufferTexture2D(GL_FRAMEBUFFER,
                        GL_COLOR_ATTACHMENT0,
                        GL_TEXTURE_2D, img, 0);
```
Status checking

- Check, if the creation worked out correctly
- See specification for detailed error-codes

```c
// fbo-creation error-checking
GLenum status =
    glCheckFramebufferStatus(GL_FRAMEBUFFER);

if (status != GL_FRAMEBUFFER_COMPLETE) {
    // error
}
```
Rendering to texture

- Bind FBO – render scene – unbind FBO
- Note: need to set viewport for FBO!

```c
// bind fbo
glBindFramebuffer(GL_FRAMEBUFFER, fbo);
glViewport(0, 0, width, height);

// clear our color- and depth-buffer
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

// render something here

// unbind fbo
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```
Using the rendered to texture

- Just bind it like a regular texture
- Note: If you want to create MIP-maps from it, use `glGenerateMipmap()`! (For more see GameDev[1].)

```c
// bind texture
glBindTexture(GL_TEXTURE_2D, img);
```
Cleaning up

- If FBO is not needed anymore, delete it
- Delete also all with the FBO associated renderbuffers and textures!

```c
// delete fbo
glDeleteFramebuffers(1, &fbo);

// delete renderbuffer
glDeleteRenderbuffers(1, &depthbuffer);

// delete texture
glDeleteTextures(1, &img);
```
That’s all?

- With an FBO, you can render into more than one texture simultaneously.
- For more check the tutorials at www.gamedev.net[1] about DrawBuffers.

References:
- [1] Gamedev.net
  - http://www.gamedev.net/reference/programming/features/fbo1/
Vertexbuffer Objects
VBOs
Why use VBOs?

- **Without VBOs**
  - _Init()_: Load model data from file
  - _Render()_: Send model data to GPU
    - _Send model data to GPU_
    - _Render model_

- **With VBOs**
  - _Init()_: Load model data from file
    - _Send model data to GPU and store it in VBOs_
  - _Render()_: Enable VBOs
    - _Render model_

- **Slow**: Send model data often to GPU
- **Fast**: Send model data once to GPU
- **Conclusion**: Use VBOs
Create VBOs

Generate VBO

```c
glGenBuffers(1, &vboHandle)

glBindBuffer(target, vboHandle);

glBufferData(target, size, data, usage)
```

- **target**
  - **GL_ARRAY_BUFFER**
    - for vertex data: vertex position, normals, tex coords, tangent vector, ...
  - **GL_ELEMENT_ARRAY_BUFFER**
    - For index data
Create VBOs

Generate VBO

```c
glGenBuffers(1, &vboHandle)

glBindBuffer(target, vboHandle);

glBufferData(target, size, data, usage)
```

- **size**
  - used memory of data array
  - e.g. array_length * sizeof(float)

- **data**
  - Array containing vertex data
Create VBOs

Generate VBO

```c
glGenBuffers(1, &vboHandle);

glBindBuffer(target, vboHandle);

glBufferData(target, size, data, usage);
```

**usage**

- GL_STREAM_DRAW, GL_STREAM_READ,
- GL_STREAM_COPY, GL_STATIC_DRAW,
- GL_STATIC_READ, GL_STATIC_COPY,
- GL_DYNAMIC_DRAW,
- GL_DYNAMIC_READ, GL_DYNAMIC_COPY
Create VBOs

usage

GL_STREAM_... You will modify the data once, then use it once, and repeat this process many times.

GL_STATIC_... You will specify the data only once, then use it many times without modifying it.

GL_DYNAMIC_... You will specify or modify the data repeatedly, and use it repeatedly after each time you do this.

..._DRAW The data is generated by the application and passed to GL for rendering.

..._READ The data is generated by GL, and copied into the VBO to be used for rendering.

..._COPY The data is generated by GL, and read back by the application. It is not used by GL.

GL_STATIC_DRAW should be the most useful for CG2
Enable VBO and connect to Shader

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

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

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

// finally enable attribute-array
glEnableVertexAttribArray(vertexLocation);
Using VBOs

- Render triangles with DrawArrays or with DrawElements (if you have indices)

```c
glDrawArrays(GL_TRIANGLES, 0, 3);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vboHandle);

glDrawElements(GL_TRIANGLES, 3, GL_UNSIGNED_INT, 0);
```

- Disable VBOs

```c
glDisableVertexAttribArray(vertexLocation);

glBindBuffer(GL_ARRAY_BUFFER, 0);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
```
Cleaning up

- If VBO is not needed anymore, delete it

```c
glDeleteBuffers(1, &vboHandle)
```

References

- OpenGL,
  http://www.opengl.org/wiki/Vertex_Buffer_Objects
- DGL Wiki,
Vertex Array Objects

VAOs
Why use VAOs?

Without VAOs

Render()

Enable vertex attribute 1
Enable vertex attribute 2
...
Enable vertex attribute n

Render model

Disable vertex attribute 1
Disable vertex attribute 2
...
Disable vertex attribute n

With VAOs

Render()

Enable VAO

Render model

Disable VAO

VAOs are a collection of VBOs and attribute pointers
// Create and Bind VAO
glGenVertexArrays(1, &vaoId);
glBindVertexArray(vaoId);

// Bind VBO
glBindBuffer(GL_ARRAY_BUFFER, vbo1Id);

// Set Attribute Pointer
GLint loc = glGetAttribLocation(programHandle, "attrib1");
glEnableVertexAttribArray(loc);
glVertexAttribPointer(loc, 3, GL_FLOAT, GL_FALSE, 0, 0);

// Continue with other VBOs/AttribPointers
...

// Unbind VAO
glBindVertexArray(0);
// Enable Shader
glUseProgram(programHandle);

// Bind VAO
glBindVertexArray(vaoId);

// Set Render Calls
glDrawElements(GL_TRIANGLES, 3, GL_UNSIGNED_INT, 0);

// Unbind VAO
glBindVertexArray(0);

// Disable Shader
glUseProgram(0);
Per combination of Shader and Model (VBOs) one VAO is needed

Don't call

```c
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
```

when a VAO is bound, or the VAO will lose the current set index vbo

References:

- OpenGL,
  
Uniform Buffer Objects
UBOs
Why use UBOs?

- Without UBOs

  Render()
  
  Set uniform parameter 1
  Set uniform parameter 2
  ...
  Set uniform parameter n
  Render model

- With UBOs

  Render():
  
  Enable UBO
  Pass uniform parameters at once
  Render model
  Disable UBO

ONCE

MANY
Uniform Blocks

- In shaders: uniforms are grouped into blocks
- Blocks can have scope names
  - Access to uniform only via scope name

```cpp
uniform MaterialBlock {
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
    float shininess;
};

void main(void)
{
    out_Color = ambient;
}
```

```cpp
uniform MaterialBlock {
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
    float shininess;
} material;

void main(void)
{
    out_Color = material.ambient;
}
```
Data layout should be specified

- 3 layouts available: packed, shared, std140
- Use std140

```cpp
layout(std140) uniform MaterialBlock {
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
    float shininess;
};
```

It is possible to choose between row-major and column-major for matrices

```cpp
layout(row_major) uniform;
// Row major is now the default for matrices.
```
Data structure

- The same data structure is needed in both, the shader and the program

In the shader:

```cpp
uniform MaterialBlock {
  vec3 ambient;
  vec3 diffuse;
  vec3 specular;
  float shininess;
};
```

In the program:

```cpp
GLfloat material[] = {
  0.3f, 0.3f, 0.3f, // ambient
  0.6f, 0.6f, 0.6f, // diffuse
  0.1f, 0.1f, 0.1f, // specular
  50 // shininess
};
```
Creating UBOs

- Start with getting an id for the UBO
  
  ```c
  GLuint uboId;
  glGenBuffers(1, &uboId);
  ```

- Then get the index of the uniform block
  - This index helps us to identify a block
  
  ```c
  GLuint blockIdx;
  blockIdx = glGetUniformBlockIndex(programHandle, "MaterialBlock");
  ```
Creating UBOs (cont.)

- You might wanna ask OpenGL for the size of the block.
- The block size should be the same as the size of the data structure in the program.

```glsl
Glint blockSize;
glGetActiveUniformBlockiv(programHandle, blockIdx,
GL_UNIFORM_BLOCK_DATA_SIZE, &blockSize);

// Test if both data structures have the same size
if( sizeof(material) != blockSize )
    ERROR!
```
Creating UBOs (cont.)

- Create the buffer
- Choose DYNAMIC_DRAW since uniforms might be changed

```c
glBindBuffer(GL_UNIFORM_BUFFER, uboId);
glBufferData(GL_UNIFORM_BUFFER, blockSize, NULL, GL_DYNAMIC_DRAW);
```
For rendering, just pass the data to the UBO

The uniform blocks will automatically get the data since they are connected with the UBO

---

Enable Shader

Connect block/buffer to binding point (see next slide)

```c
// Bind Buffer
glBindBuffer(GL_UNIFORM_BUFFER, uboId);
// And pass data to UBO
glBufferData(GL_UNIFORM_BUFFER, blockSize, material, GL_DYNAMIC_DRAW);
```

Render Calls

Disable Shader
Connect to binding point

- At last, connect the uniform block and the uniform buffer to a binding point
- Binding points connect uniform blocks to uniform buffers
- Use different binding points for different blocks/buffers
  - Like you should use different texture units for different textures/samplers

```c
GLuint bindingPoint = 0;
glBindBufferBase(GL_UNIFORM_BUFFER, bindingPoint, uboId);
glUniformBlockBinding(programHandle, blockIdx, bindingPoint);
```
Cleaning up

- If UBO is not needed anymore, delete it

```
// GLFW 3.1 example app 2.7.6
int main(void)
{
    printf(“Hello World!\n”);
    return 0;
}
```

References

- OpenGL,
  http://www.opengl.org/registry/specs/ARB/uniform_buffer_object.txt
Notes on CG2
Textures/VBOs are mandatory

You also have to implement (at least) one of the following:

- FBOs: USEFUL FOR A LOT OF EFFECTS
- VAOs: JUST 5 LINES OF CODE
- UBOs