Politecnico di Torino

Laboratorio 09

OpenGL

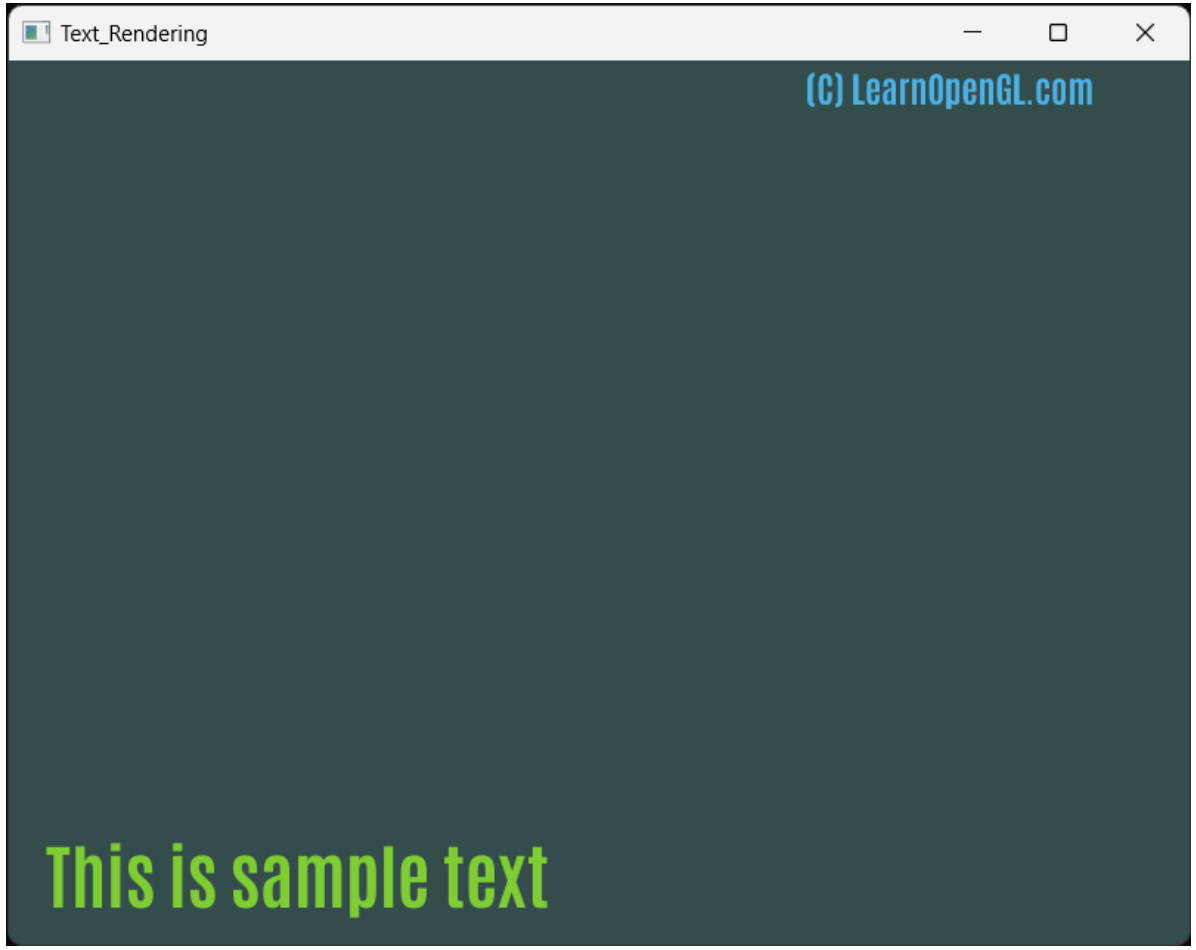
Part IV – Advanced OpenGL

Outline

- Text rendering
- Sounds
- Additional resources
- Breakout game

Text rendering

This example shows how to draw text in OpenGL.



Text rendering

FreeType

Since there is no support for text capabilities within OpenGL, it is necessary to define a system for rendering text to the screen. There are no graphical primitives for text characters. Some example techniques are:

- drawing letter shapes via `GL_LINES;`
- create 3D meshes of letters,
- render character textures to 2D quads in a 3D environment.

In this example, the FreeType library is used since it represents a flexible technique for rendering text.

Text rendering

FreeType

FreeType is a software development library that is able to load fonts, render them to bitmaps, and provide support for several font-related operations.

What makes FreeType particularly attractive is that it is able to load **TrueType** fonts. A TrueType font is a collection of character glyphs not defined by pixels or any other non-scalable solution, but by mathematical equations (combinations of splines). Similar to vector images, the rasterized font images can be procedurally generated based on the preferred font height. By using TrueType fonts it is possible to render character glyphs of various sizes without any loss of quality.

FreeType can be downloaded from their website: https://freetype.org/index.html

Text rendering

FreeType

Before starting it is necessary to include the appropriate headers:

```
#include <ft2build.h>
#include FT_FREETYPE_H
```

FreeType loads TrueType fonts and, for each glyph, generates a bitmap image and calculates several metrics. Bitmap images can be extracted for generating textures. Moreover, each character glyph can be positioned by using the loaded metrics.

Text rendering

FreeType

To load a font, first, the FreeType library has to be initialized and the font loaded as FreeType. In the following, the Antonio-Bold.ttf TrueType font file was loaded. Family fonts are available at https://fonts.google.com/:

```
FT_Library ft;
if (FT_Init_FreeType(&ft)){
    std::cout << "ERROR::FREETYPE: Could not init FreeType Library" << std::endl;
    return -1;
}

std::string font_name = "resources/fonts/Antonio/static/Antonio-Bold.ttf";

FT_Face face;
if (FT_New_Face(ft, font_name.c_str(), 0, &face)) {
    std::cout << "ERROR::FREETYPE: Failed to load font" << std::endl;
    return -1;
}
```

Text rendering

FreeType

Once the face is loaded, it is necessary to specify the pixel font size to extract from this face. The function sets the font's width and height parameters. Setting the width to 0 lets the face dynamically calculate the width based on the given height.

```
FT_Set_Pixel_Sizes(face, 0, 48);
```

A FreeType face hosts a collection of glyphs. A glyph can be set as the active glyph by calling FT_Load_Char. For example, to load the char ‘X’:

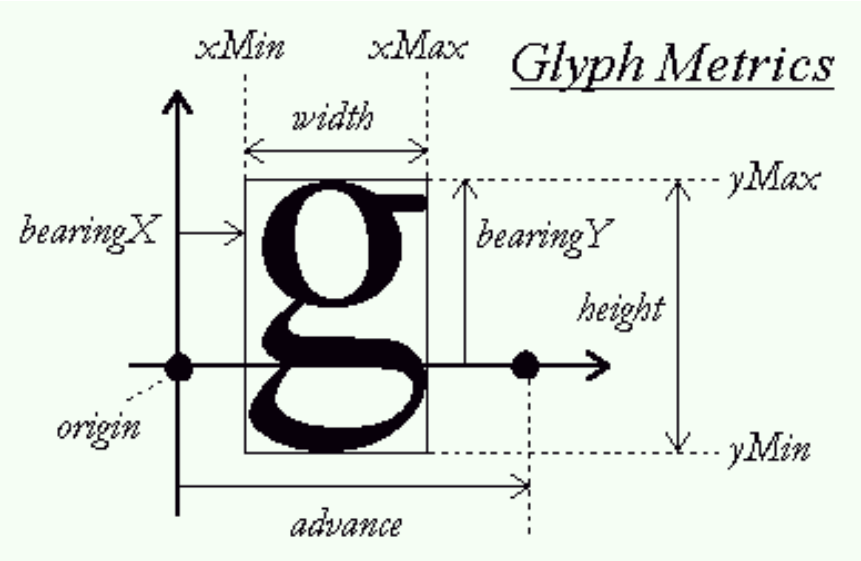
```
if (FT_Load_Char(face, 'X', FT_LOAD_RENDER))
{
    std::cout << "ERROR::FREETYTPE: Failed to load Glyph" << std::endl;
    return -1;
}
```

By setting `FT_LOAD_RENDER` as one of the loading flags, FreeType is instructed to create an 8-bit grayscale bitmap image that can be accessed via `face->glyph->bitmap`

Text rendering

FreeType

Each of the glyphs load with FreeType however, do not have the same size. The bitmap image generated by FreeType is just large enough to contain the visible part of a character. For example, the bitmap image of the dot character '.' is much smaller in dimensions than the bitmap image of the character ‘X’. For this reason, FreeType also loads several metrics that specify how large each character should be and how to properly position them. All the metrics are shown in the image.

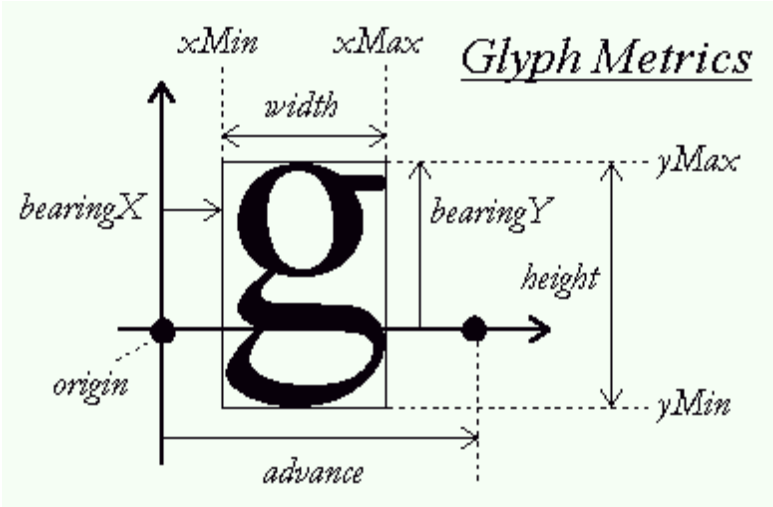


Text rendering

FreeType

Each of the glyphs reside on a horizontal baseline where some glyphs sit exactly on top of this baseline (like 'X') or some slightly below the baseline (like 'g' or 'p'). These metrics define the exact offsets to properly position each glyph on the baseline, how large each glyph should be, and how many pixels are needed to advance for rendering the next glyph.

- `width`: the width (in pixels) of the bitmap accessed via `face->glyph->bitmap.width`.
- `height`: the height (in pixels) of the bitmap accessed via `face->glyph->bitmap.rows`.
- `bearingX`: the horizontal bearing e.g. the horizontal position (in pixels) of the bitmap relative to the origin accessed via `face->glyph->bitmap_left`.
- `bearingY`: the vertical bearing e.g. the vertical position (in pixels) of the bitmap relative to the baseline accessed via `face->glyph->bitmap_top`.
- `advance`: the horizontal advance e.g. the horizontal distance (in 1/64th pixels) from the origin to the origin of the next glyph. Accessed via `face->glyph->advance.x`.



Text rendering

FreeType

To render a character to the screen it would be possible to load a character glyph, retrieve its metrics, and generate a texture each time. However, this solution would be inefficient to do each frame. For this reason, it is better to store the generated data somewhere in the application and query it whenever a character has to be rendered.

To this aim a convenient struct can be used and stored into a map.

```
struct Character {
    unsigned int TextureID; // ID handle of the glyph texture
    glm::ivec2   Size;      // Size of glyph
    glm::ivec2   Bearing;   // Offset from baseline to left/top of glyph
    unsigned int Advance;   // Horizontal offset to advance to next glyph
};
```

Text rendering

FreeType

In this example the first 128 characters of the ASCII character set are considered. For each character, a texture is generated, and its relevant data are stored into a `Character struct` and added to the `Characters map`.

```
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

for (unsigned char c = 0; c < 128; c++)
{
    // load character glyph
    if (FT_Load_Char(face, c, FT_LOAD_RENDER))
    {
        std::cout << "ERROR::FREETYTPE: Failed to load Glyph" << std::endl;
        continue;
    }
    // generate texture
    unsigned int texture;
    glGenTextures(1, &texture);
    glBindTexture(GL_TEXTURE_2D, texture);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RED, face->glyph->bitmap.width,
face->glyph->bitmap.rows, 0, GL_RED, GL_UNSIGNED_BYTE, face->glyph-
>bitmap.buffer);
    // set texture options
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    // now store character for later use
    Character character = {
        texture,
        glm::ivec2(face->glyph->bitmap.width, face->glyph->bitmap.rows),
        glm::ivec2(face->glyph->bitmap_left, face->glyph->bitmap_top),
        static_cast<unsigned int>(face->glyph->advance.x)
    };
    Characters.insert(std::pair<char, Character>(c, character));
}
```

Text rendering

FreeType

Once the glyphs has been processed, FreeType's resource can be cleared using:

```
FT_Done_Face(face);
FT_Done_FreeType(ft);
```

Text rendering

Shaders

To render the glyphs, the following vertex shader is needed:

```
#version 330 core
layout (location = 0) in vec4 vertex; // <vec2 pos, vec2 tex>
out vec2 TexCoords;

uniform mat4 projection;

void main()
{
    gl_Position = projection * vec4(vertex.xy, 0.0, 1.0);
    TexCoords = vertex.zw;
}
```

Text rendering

Shaders

The following fragment shader is used:

```
#version 330 core
in vec2 TexCoords;
out vec4 color;

uniform sampler2D text;
uniform vec3 textColor;

void main()
{
    vec4 sampled = vec4(1.0, 1.0, 1.0, texture(text, TexCoords).r);
    color = vec4(textColor, 1.0) * sampled;
}
```

Text rendering

Shaders

In this example, it is also needed to enable blending thought:

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

For the projection matrix an orthographic projection matrix is used. By using an orthographic projection matrix it is possible to specify all vertex coordinates in screen coordinates if the matrix is configured as follow:

```
glm::mat4 projection = glm::ortho(0.0f, static_cast<float>(SCR_WIDTH), 0.0f,
static_cast<float>(SCR_HEIGHT));
```

Text rendering

Shaders

Last up is creating a VBO and VAO for rendering the quads.

```
unsigned int VAO, VBO;
glGenVertexArrays(1, &VAO);
glGenBuffers(1, &VBO);
glBindVertexArray(VAO);
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(float) * 6 * 4, NULL, GL_DYNAMIC_DRAW);
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 4 * sizeof(float), 0);
glBindBuffer(GL_ARRAY_BUFFER, 0);
glBindVertexArray(0);
```

Text rendering

Render line of text

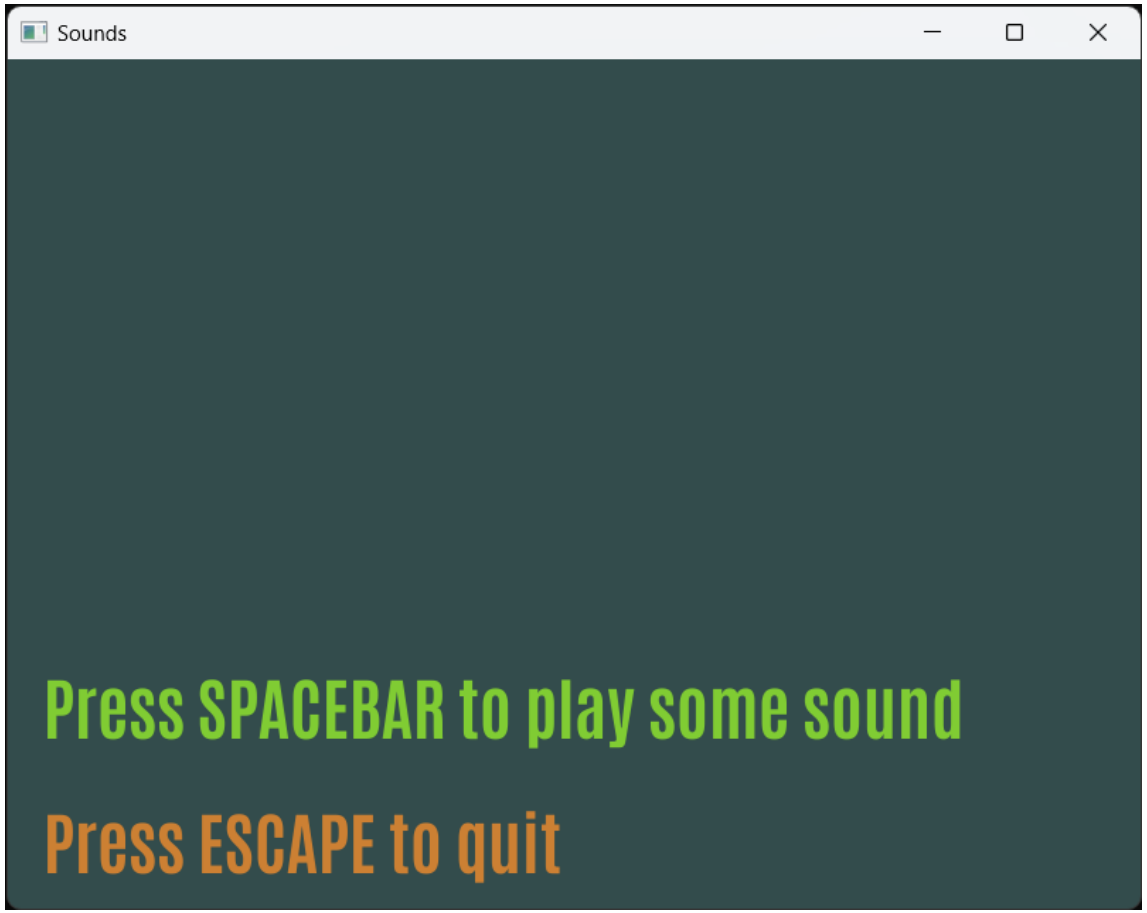
To render a character, it is necessary to extract the corresponding Character struct from the map and calculate the quad's dimensions using the character's metrics. With the quad's calculated dimensions is it possible to update the content of the memory managed by the VBO using `glBufferSubData`.

The function `RenderText` is in charge of rendering a string of characters. The function first calculates the origin position of the quad (as `xpos` and `ypos`) and the quad's size (as `w` and `h`) and generates a set of 6 vertices to form the 2D quad. Each metric has been multiplied by scale to redefine the size. Then the content of the VBO is updated and quads rendered.

Sounds

This example shows how to reproduce sounds in OpenGL using the irrKlang Library.

More tutorials on this library can be found at: https://www.ambiera.com/irrklang/tutorials.html



Sounds

In order to reproduce sounds, it is possible to look at the benefits of the **irrKlang** library. irrKlang is a cross platform sound library that support both 2D and 3D (spatialized) audio. The sound engine supports several file formats, e.g., WAV, MP3, OGG, FLAC, MOD, XM, IT, S3M, and is usable in C++ and all .NET languages.

Link: https://www.ambiera.com/irrklang/index.html

Sounds

irrKlang

Before starting it is necessary to include the appropriate headers:

```
#include <iostream>
#include <irrKlang.h>
```

To use the library it is first necessary to start the irrKlang sound engine:

```
irrklang::ISoundEngine* engine = irrklang::createIrrKlangDevice();

if (!engine)
    return 0; // error starting up the engine
```

To play some sound stream, looped

```
engine->play2D("resources/media/getout.ogg", true);
```

Sounds

irrKlang

The second parameter of the play2D function tells the engine to play it looped. Therefore, this parameter can be ignored to play the sound once:

```
engine->play2D("resources/media/bell.wav");
```

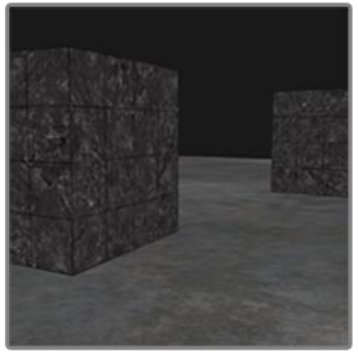
When the application is terminated, it is necessary to delete the irrKlang Device created earlier with `createIrrKlangDevice()` function:

```
engine->drop();
```

Additional resources

More resources on Advanced OpenGL features are available at:

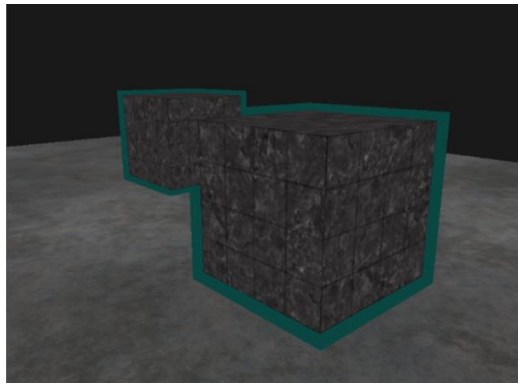
- Stencil testing and Object outlining: https://learnopengl.com/Advanced-OpenGL/Stencil-testing
- Blending: https://learnopengl.com/Advanced-OpenGL/Blending
- Cubemaps: https://learnopengl.com/Advanced-OpenGL/Cubemaps
- Advanced lighting: https://learnopengl.com/Advanced-Lighting/Advanced-Lighting



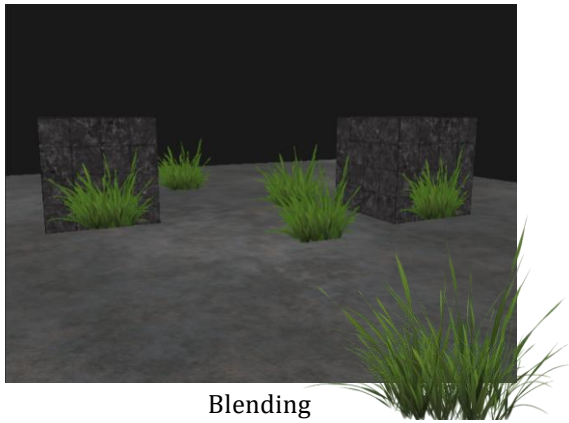
Color buffer

Stencil buffer

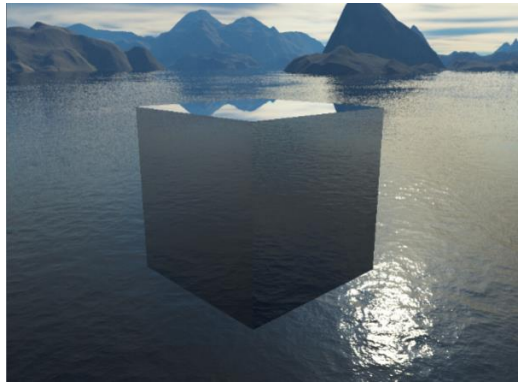
After stencil test



Outlined objects



Blending



Cube maps

Blinn-Phong lighting model

Breakout game

Tutorial for creating a practical application with OpenGL. The tutorial presents the basic steps to create a relatively simple 2D game: Breakout

https://learnopengl.com/In-Practice/2D-Game/Breakout