Text Rendering

-그래픽을 만들다 보면 글자 text가 필요할 때가 생길 것임

-화면에 간단한 문자열을 얻는 방법은 OpenGL같은 low-level API를 사용하면 쉬움

-단, 서로 다른 크기의 문자를 렌더링 하는데 여백이 달라지는 순간 제작이 어려움

-ex) 수식을 추가하거나 대문자, 소문자, 등등

-OpenGL에서는 텍스트 기능을 지원하지 않기 때문에 텍스트 렌더링을 위한 시스템은 직접 정의해야함

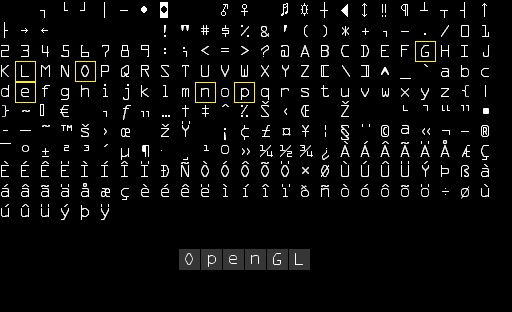
-문자를 만들기 위해서 여러 방법이 있는데, GL\_LINES을 이용해서 직접 문자를 그리거나, 2D쿼드에 텍스트를 입혀서 사용해도 됨

-대부분은 쿼드에 텍스처를 입혀서 렌더링 함, FreeType library를 사용해서 구현할 예정

Classical Text Rendering: bitmap fonts

-초기에는 텍스트를 렌더링 하는 것은 직접 폰트를 선택해서 모든 글자를 추출해 하나의 큰 텍스처 내에 배치하였음

-bitmap fonts라고 부르는 이런 텍스처에는 미리 정의된 영역에서 사용하려는 글자가 들어있었음, 이러한 문자 기호는 Glyphs라고 부름

-큰 텍스처에서 특정 위치에 있는 글자들을 사용해서 OpenGL을 만든 모습임

-이러한 비트맵 fonts를 만드는 방법은 Codehead's의 bitmap font generator을 사용하였음

<http://www.codehead.co.uk/cbfg/>

Modern text rendering : FreeType

-FreeType은 fonts를 읽고, 비트맵으로 렌더링하는 여러 글꼴 관련 작업을 하는 라이브러리

-맥os, Java, Playstation, Linux, Android에서 사용함

-TrueType fonts를 읽어올 수 있음

|  |
| --- |
| TrueType Fonts // .ttf  -일반 문서 작업 사용자, 한글, 일반 프로그램 등에 쓰임    OpenType Fonts // .otf  -고해상도 출력 작업 사용자, QuarkXprees 및 포토샵 등에 쓰임 |

-TTF는 픽셀같은 확대가 불가능한 것이 아닌 수학적 방정식으로 정의되는 문자

-TTF를 사용하면 품질 저하 없이 다양한 크기의 문자를 렌더링 할 수 있음

-FreeType 라이브러리 include

<https://freetype.org/>

-FreeType는 TTF를 로드하고, 각 glyph에 대해서 비트맵 이미지를 생성, 여러 행렬을 계산함

-texture을 생성하기 위해서 비트맵 이미지를 추출하고, 로그된 행렬을 이용해서 각 문자

glyph를 적절하게 배치가 가능함

-라이브러리 초기화, 글꼴 로드

|  |
| --- |
| FT\_Library ft;  if (FT\_Init\_FreeType(&ft))  {  std::cout << "ERROR::FREETYPE: Could not init FreeType Library" << std::endl;  return -1;  }  FT\_Face face;  if (FT\_New\_Face(ft, "fonts/arial.ttf", 0, &face))  {  std::cout << "ERROR::FREETYPE: Failed to load font" << std::endl;  return -1;  } |

-추출할 픽셀 글꼴 크기를 정의

|  |
| --- |
| FT\_Set\_Pixel\_Sizes(face, 0, 48); |

-글꼴의 너비 빛 높이 파라미터를 설정

-너비가 0이면, 주어진 높이를 기준에서 너비를 동적으로 계산함

-FT\_Load\_Char로 글리프 중 하나를 활성 글리프로 설정, 여기서는 'X'를 사용

|  |
| --- |
| if (FT\_Load\_Char(face, 'X', FT\_LOAD\_RENDER))  {  std::cout << "ERROR::FREETYTPE: Failed to load Glyph" << std::endl;  return -1;  } |

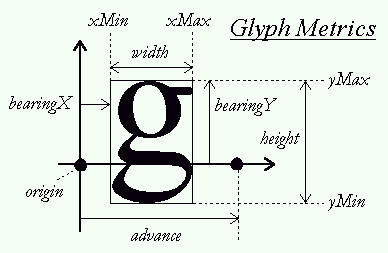
-FT\_LOAD\_RENDER로 로딩 플래그로 설정하면 FreeType face -> 글리프-> 비트맵으로 엑세스 할 수 있는 8비트 grayscale 비트 맵 이미지를 만들 수 있음

-각 글리프에는 글꼴의 크기가 다양함

-예를 들어서 '.' 은 'X'보다 작은 크기의 이미지를 가짐

-각 문자와 크기를 올바르게 지정하는 방법을 지정하는 행렬을 로드

-지정하는 방법

-일부 문자는 origin의 위치보다 위에 있거나, 걸쳐있기도 함 g, X 차이를 보면 알 수 있음

-각 글자 하나를 렌더링할 때 마다 글리프를 로드하고, 메트릭스를 검색해서 텍스처를 생성하는 방식은 비효율적임

-생성된 데이터를 미리 저장하여 렌더링할 때 불러오는 것이 좋음

-map에 저장

|  |
| --- |
| 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; // Offset to advance to next glyph  };  std::map<char, Character> Characters; |

-ASCII 문자의 128가지로 제한하여 생성할 예정

|  |
| --- |
| glPixelStorei(GL\_UNPACK\_ALIGNMENT, 1); // disable byte-alignment restriction    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),  face->glyph->advance.x  };  Characters.insert(std::pair<char, Character>(c, character));  } |

-for 루프에서 128개의 문자를 검색

-텍스처 내부 형식을 GL\_RED로 했는데, 글리프에서 생성되는 비트 맵은 단일 바이트로 표시되어지기 때문에 단일 색상 값으로 저장한 것임

-단일 바이트를 사용해서 텍스처의 색상을 나타내는 경우, OpenGL에 알려줘야 함

|  |
| --- |
| glPixelStorei(GL\_UNPACK\_ALIGNMENT, 1); |

-텍스처는 모두 4바이트를 필요로 하기 때문에, 항상 크기는 4의 배수임

-대부분의 텍스처는 4바이트이기 때문에 문제가 되지 않지만, 우리는 픽셀당 단일 바이트를 사용하기 때문에 텍스처는 어떤 너비도 가질 수 있음

-언팩 정렬을 1로 설정해서 정렬 문제가 있는지 없는지 확인함

-글리프 처리가 끝나면 FreeType의 리소스를 제거

|  |
| --- |
| FT\_Done\_Face(face);  FT\_Done\_FreeType(ft); |

Shader

-font vertex shader

|  |
| --- |
| #version 460 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;  } |

-font fragment shader

|  |
| --- |
| #version 460 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;  } |

-텍스처r 구성요소를 알파 값으로 샘플링함

-출력 색상의 알파 값을 변경하면, 모든 글리프의 배경 색상은 투명하게 되고, 글자 부분은 투명하지 않게 됨

-RGB 색상에 textColor uniform을 곱해서 색상을 변경함

-이후 블랜딩함

|  |
| --- |
| glEnable(GL\_BLEND);  glBlendFunc(GL\_SRC\_ALPHA, GL\_ONE\_MINUS\_SRC\_ALPHA); |

-projection 행렬엔 직교 projection 행렬을 사용함

-text는 일반적으로 투시법이 필요하지 않음

|  |
| --- |
| glm::mat4 projection = glm::ortho(0.0f, 800.0f, 0.0f, 600.0f); |

-ortho(왼쪽, 오른쪽, 아래, 위), 즉 0, 화면너비, 0, 화면높이

-쿼드 렌더링용 VBO, VAO 만들기

|  |
| --- |
| 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); |

-쿼드에 6개의 vertices에 4개의 float가 필요함, 6 \* 4

-VBO 메모리는 자주 업데이트 되기 때문에 GL\_DYNAMIC\_DRAW로 할당함

Render line of text

-text 렌더링을 위해 Character map의 문자 구조를 추출, 행렬을 가져와서 쿼드 치수를 계산함

-계산된 치수를 사용해서 glBufferSubData를 사용, VBO에 관리되는 메모리의 내용을 업데이트하는데 사용하는 6개의 vertex를 동적으로 생성함

|  |
| --- |
| void RenderText(Shader &s, std::string text, float x, float y, float scale, glm::vec3 color)  {  // activate corresponding render state  s.Use();  glUniform3f(glGetUniformLocation(s.Program, "textColor"), color.x, color.y, color.z);  glActiveTexture(GL\_TEXTURE0);  glBindVertexArray(VAO);  // iterate through all characters  std::string::const\_iterator c;  for (c = text.begin(); c != text.end(); c++)  {  Character ch = Characters[\*c];  float xpos = x + ch.Bearing.x \* scale;  float ypos = y - (ch.Size.y - ch.Bearing.y) \* scale;  float w = ch.Size.x \* scale;  float h = ch.Size.y \* scale;  // update VBO for each character  float vertices[6][4] = {  { xpos, ypos + h, 0.0f, 0.0f },  { xpos, ypos, 0.0f, 1.0f },  { xpos + w, ypos, 1.0f, 1.0f },  { xpos, ypos + h, 0.0f, 0.0f },  { xpos + w, ypos, 1.0f, 1.0f },  { xpos + w, ypos + h, 1.0f, 0.0f }  };  // render glyph texture over quad  glBindTexture(GL\_TEXTURE\_2D, ch.textureID);  // update content of VBO memory  glBindBuffer(GL\_ARRAY\_BUFFER, VBO);  glBufferSubData(GL\_ARRAY\_BUFFER, 0, sizeof(vertices), vertices);  glBindBuffer(GL\_ARRAY\_BUFFER, 0);  // render quad  glDrawArrays(GL\_TRIANGLES, 0, 6);  // now advance cursors for next glyph (note that advance is number of 1/64 pixels)  x += (ch.Advance >> 6) \* scale; // bitshift by 6 to get value in pixels (2^6 = 64)  }  glBindVertexArray(0);  glBindTexture(GL\_TEXTURE\_2D, 0);  } |

-텍스트 문자열 렌더링 하는 법

|  |
| --- |
| RenderText(shader, "github.com/kkr970/Study\_OpenGL",  25.0f, 25.0f, 0.8f, glm::vec3(0.5, 0.8f, 0.2f));  RenderText(shader, "(C) LearnOpenGL.com", 540.0f, 570.0f, 0.5f, glm::vec3(0.3, 0.7f, 0.9f)); |

-font는 메이플스토리 서체를 사용 // 폰트가 예뻐서 사용하였음

