Introdução

Uma das melhores sensações é a de conseguir fazer as coisas por si mesmo, se você pretende fazer as coisas por si mesmo na computação gráfica, esses tutoriais irão a te ajudar a ter uma gama de conhecimento sobre o funcionamento do OpenGL. Mas para isso é necessário que se tenha alguns conhecimentos prévios como requisito.

Pré requisitos

OpenGL pode ser considerada uma api gráfica e não uma plataforma, assim para programa-lo voce precisará ter conhecimento em alguma linguagem de programação. A linguagem usada nesse tutorial é C++ . Portanto para acompanhar com mais facilidade é necessário que se tenha algum conhecimento em C++, durante tutorial vou explicar sobre alguns tópicos de C++ básico até ao avançado, mas mesmo assim voce deve no mínimo saber fazer o famoso "hello world". Se não conhece muito bem o C++ fica alguns links que podem te ajudar a aprender sobre essa poderosa linguagem.

http://www.learncpp.com

https://developers.google.com/edu/c++/

O código de todos os programas estará no github(link) com uma boa documentação nos comentários , em inglês e em português, ao ler o código atente-se aos comentários e informações que o contém, poderá facilitar bastante o seu entendimento. No tutorial também será apresentado trechos de código.

No tutorial sempre que tiver uma colocação em **Azul** significa que é uma maneira vulgar, ou coloquial de se descrever algo, criando uma melhor abstração para facilitar entendimento. Sempre que estiver em **Vermelho**, são dicas em que deve ter atenção e cuidados .

(**TODO**)implementar documentação interativa no site

Sobre OpenGL

Antes de começar a aprender sobre os conceitos do OpenGL é importante você saber o que ele é, e um pouco de sua história

OpenGL pode ser considerada uma API(Application Programming Interface) de baixo nível que nos disponibiliza um gama de funções para manipulação gráfica e de imagens. Quem desenvolve a biblioteca OpenGL atualmente são os próprios desenvolvedores das placas gráficas, assim se você possui um dispositivo Apple, a Apple que da manutenção na biblioteca gráfica do seu dispositivo, e assim por diante. Por isso é sempre importante manter os drivers da sua placa gráfica atualizados, pois podem ter atualizações relativas ao OpenGL que pode corrigir algum bug, ou otimizar alguma função.

Core Profile

Antigamente o OpenGL tinha o modo de desenvolvimento imediato, em que não se era obrigado a usar um pipeline programável(programação com a placa gráfica), esse modo era mais fácil de usar, porém não garantia muitas flexibilizes aos desenvolvedores. Assim desde o OpenGL3.2 o modo imediato se tornou obsoleto, restando somente a pipeline programável.

Quando se trabalha com o a pipeline programável, chamamos de Core Profile mode, esse modo é mais difícil de se compreender e desenvolver, porém garante muito mais eficiência se você souber utilizar.

O OpenGL do 3.2 acima, é chamado de Modern OpenGL , nesse tutorial utilizarmos a versão 3.3, sem utilizar funções obsoletas de OpenGL não moderno que podem te gerar erros no futuro quando o OpenGL atualizar.

Definição Coloquial

O OpenGL por si, pode ser definido como uma grande máquina de estados, com funções que alteram seus estados, em um devido contexto. Então para operar o OpenGL é necessário ter um contexto criado, e operar os estados em cima desse contexto. Sempre que falamos para o OpenGL desenhar linhas ao invés de triângulos, ele muda sua máquina de estado de desenho para linhas, e a partir desse momento tudo no contexto será desenhado com linhas.

Se você pensar no OpenGL com essa definição pode facilitar a abstração, quando for utilizar suas funções, que só alteram como algumas operações serão realizadas no OpenGL, fará mais sentido.

Objetos

O coração da biblioteca do OpenGL é escrita em C, C é uma linguagem baixo nível, e as vezes sua tradução para linguagens de alto nível não é fácil, assim o OpenGL tem várias abstrações para se aproximar dessas linguagens, uma delas são objetos.

Objeto é um conjunto de opções que definem um estado do OpenGL. Por exemplo, se tivermos um objeto que define a nossa Janela onde será desenhado, essa janela possui propriedades como Cor, Tamanho, quantas cores ele suporta, etc. Um objeto em pode ser visualizado como uma struct

```
1  struct object_name {
2    GLfloat option1;
3    GLuint option2;
4    GLchar[] name;
5  };
```

****Tipos primitivos, o OpenGL possui seus próprios tipos primitivos, um GLFloat é similar a um ponto flutuante Float, porém é adaptado para funcionar em múltiplas plataformas, assim se aconselha sempre em programas OpenGL utilizar seus tipos primitivos, permitindo assim que sua aplicação seja portável.**

Assim quando formos utilizar objetos do OpenGL será algo parecido com isso(Um contexto em OpenGL são geralmente structs gigantescas de Objetos):{

```
// O estado do OpenGL
struct OpenGL_Context {
    ...
    object* object_Window_Target;
    ...
};


// Cria um objeto
GLuint objectId = 0;
glGenObject(1, &objectId);
// Liga o objeto ao contexto
glBindObject(GL_WINDOW_TARGET, objectId);
// Altera as opções do seu objeto
glSetObjectOption(objectId, GL_OPTION_WINDOW_WIDTH, 800);
glSetObjectOption(objectId, GL_OPTION_WINDOW_HEIGHT,
600);
// Seta o contexto de volta para o padrão
glBindObject(GL_WINDOW_TARGET, 0);
```

Isso é o que você geralmente vai ver em programas que usam OpenGL, primeiro você cria um objeto, e passa a referência dele como um id, os dados do objeto real são armazenados **por traz das cenas(dentro do OpenGL),** após isso, ligamos o objeto criado ao contexto en questão(A localização do objeto de destino é definido como GL_WINDOW_TARGET), após isso com o nosso objeto faremos a alterações que quisermos no contexto. Após realizar as alterações , desligamos nosso objeto, quebrando o link dele com o contexto, definindo a identificação do (GL_WINDOW_TARGET) para 0.

Esta exemplo de código é só uma leve abstração de como o OpenGL funciona, mais a frente irei mostrar exemplos reais.

A vantagem de se utilizar objetos, é que podemos criar vários estados de operações para um mesmo contexto, por exemplo: Eu quero desenhar dois modelos 3D , um avião, e um meteoro, ambos são desenhados no mesmo contexto, mas de forma diferente, assim nos criamos o objeto1 para o avião, estamos suas propriedades, depois disso criamos o objeto2 para o meteoro e definimos suas propriedades.Agora sempre que formos desenhar algum deles, é só dizer ao linkar o objeto a ser desenhado ao contexto , e após isso realizar as operações de desenho,assim, o OpenGL irá desenhar de acordo propriedades queremos utilizar, as do objeto1 quando for desenhar o avião, e as do objeto2 quando for desenha o meteoro.

Criando uma Janela

A primeira coisa a se fazer, para desenhar coisas mágicas com o OpenGL, é criar um contexto OpenGL e uma Janela de aplicação. Criar uma janela varia de acordo com o sistema operacional, e é uma operação externa ao OpenGL. Assim temos que criar uma janela, aplicar o nosso contexto a ela, e trabalhar com as entradas do usuário tudo por nossa conta.

Para nossa sorte existem diversos Frameworks como SDL, SFML, freeglut e GLFW, que nos auxiliam nesse trabalho. Nesse tutorial iremos utilizar o **GLFW**.

GLFW

GLFW é uma biblioteca escrita em C, focada especificamente para o OpenGL, com funcionalidades básicas do que precisamos para botar algo na tela. Ela nos auxilia na criação de um contexto OpenGL, definir as propriedades e criar uma Janela para aplicar esse contexto, e lidar com as entradas do usuário(mouse , teclado, joystick, etc).

Nesse tutorial, basicamente iremos focar em como instalar essa biblioteca, certificar de que o contexto OpenGL está sendo criado e abrir uma Janela, para desenharmos a magia.

Configurando o GLFW

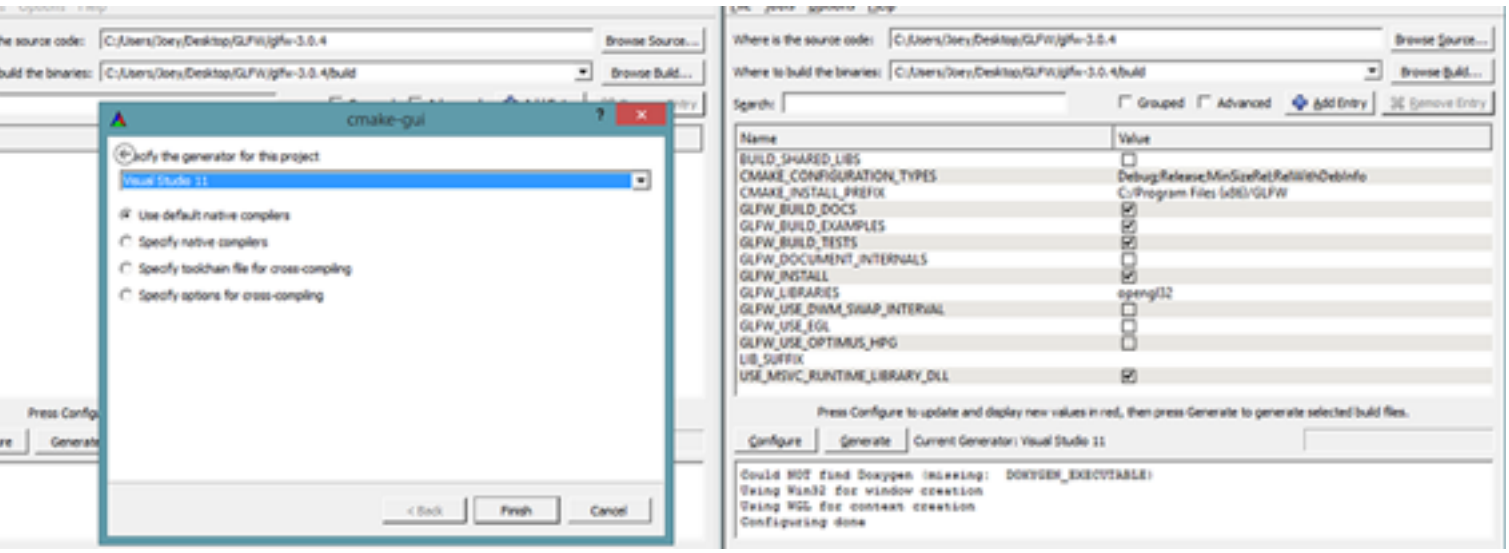
O GLFW pode ser obtido no seu site oficial através do seguinte link http://www.glfw.org/download.html . Como no site recomenda, é preferível que se obtenha a versão SourceCode, até para usuários Windows. Compilar a biblioteca a partir do código fonte(SourceCode) garante que ela será adaptada para seu Sistema Operacional, um problema grande enfrentado em códigos OpenSource é que as pessoas utilizam vários Sistemas Operacionais e IDE`s diferentes, para ajudar solucionar este problema existe um ferramenta chamada CMake.

CMake

O CMake é uma ferramenta que permite carregar arquivos de projeto de acordo com a IDE selecionada pelo usuário(CODE:BLOCKS, XCode, Visual Studio, Eclipse, Unix, etc). Ele gera um arquivo de projeto com as configurações para compilar uma biblioteca(no nosso caso GLFW). O CMake pode ser obtido na seguinte página de Download:

http://www.cmake.org/cmake/resources/software.html

Selecione a versão correspondente ao seu sistema operacional, e faça o download. Após instalar o CMake ele terá dois modos de operações, por linha de comando e visual, iremos utilizar o visual(GUI) , Ele irá pedi o Source Folder(No caso a pasta do código fonte do GLFW), Destionation Folder for the binaries (onde será criado o projeto) .



Após estar as páginas pedidas, ele clique em Configure no canto esquerdo inferior, ele irá abrir uma janela como a da imagem acima, Selecione no primeiro item, o seu IDE desejado, clique em finish. Ele irá mostrar algumas configurações para montagem do projeto, ignore e clique em configure novamente. Se tiver alguma tabela em NAME e VALUE na cor vermelha, clique em Configure até que tudo esteja da cor como na imagem acima. Após isso clique em Generate, seu projeto será criado na pasta selecionada. Nessa pasta você pode visualizar em src/Debug glfw3.lib ou .a (variando de acordo com seu sistema operacional) a biblioteca criada.

Agora que a biblioteca foi criada precisamos fazer as nossa IDE, enxergar os arquivos da biblioteca. Como boa prática para isso iremos criar uma pasta estática no sistema, que irá conter todos os nossos arquivos de bibliotecas, assim sempre que formos iniciar um novo Projeto na IDE teremos que dizer para ela enxergar essa pasta. Eu pessoalmente crio uma pasta chamada Libs, e dentro dessa pasta crio outra pasta com o nome da biblioteca(no nosso caso glfw3), e dentro dessa pasta coloco todos os arquivos de cabeçalho e bibliotecas em questão. Assim minhas bibliotecas de terceiros ficam em um único lugar que pode sempre ser acessada localmente ou remotamente quando precisar.

Após isso, iremos criar nosso primeiro Projeto GLFW

Primeiro Projeto(TODO) Implementar como montar projetos no windows, linux, mac, lindando as bibliotecas, e testando se funcionou.

GLEW

Como havíamos comentado, o OpenGL é de responsabilidade cada fabricante desenvolver. Isso quer dizer que existem várias versões do OpenGL com especificações diferentes de acordo com seus drivers, gera a consequência de não se saber a localização da maior parte das funções e esse locais precisam ser consultados.

Assim é tarefa do desenvolver obter a localização dessas funções e a armazenar em ponteiros de função para depois serem utilizadas. Isso é um trabalho extremamente complexo e demorado, para auxiliar nesse trabalho existe biblioteca GLEW atualmente a mais popular e atualizada biblioteca do ramo.

GLEW significa OpenGL Extension Wrangler e pode ser obtida no seguinte endereço:

http://glew.sourceforge.net

Para instalar GLEW, deve se fazer o mesmo processo utilizado para instalar o GLFW. Após isso, e adicionado GLEW ao seu projeto estamos prontos para abrir uma janela de mágicas :D .

Primeira Janela

Vamos aprender como abrir uma janela com o GLFW, para isso criaremos uma classe Program, basicamente criaremos dois arquivos Program.h e Program.cpp. Nesse tutorial essa classe irá inicializar o GLFW, inicializar o GLEW, criar o contexto OpenGL e linkar ao GLFW . Na sua classe Program.h inclua as bibliotecas e defina os métodos

```
#include <GL/glew.h>
#include <GL/glfw3.h>
class Program{
public:
    Program();
    ~Program();
    GLFWwindow* initOpenGL();
private:
};
```

****Sempre inclua o GLEW antes do GLFW3. O GLEW contem os cabeçalhos do OpenGL e alguns cabeçalhos do GLFW3 requerem os cabeçalhos do OpenGL.**

Note que o método initOpenGL() retorna uma GLFWwindow, que é uma janela criada pelo GLFW, essa janela será usada no nosso main.cpp(programa principal). Agora inclua o código no Program.cp, atenção para os comentários que estão no código

```
#include "Program.h"

#define SCREEN_SIZE_X 800
#define SCREEN_SIZE_Y 600

//Window utilised by all the program
//Janela Utilizada em todo o programa
GLFWwindow* window;
```

```
Program::Program(){};

Program::~Program(){};

//Inicialisa o GLFW, o GLEW e cria um contexto de OpenGL
GLFWwindow* Program::initOpenGL(){

    //initi GLFW
    //Inicializa o GLEW
    if (!glfwInit())
        exit(EXIT_FAILURE);
```

Inicializamos o GLFW com o glfwInit();

```
    //Force glfw to opengl gl version 3.3
    //Força o GLFW a usar a versao 3.3 do OpenGL
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
    glfwWindowHint(GLFW_OPENGL_PROFILE,
GLFW_OPENGL_CORE_PROFILE);
    glfwWindowHint(GLFW_RESIZABLE, GL_FALSE);
```

glfwWindowHint() : configura o GLFW, o primeiro parâmetro é um enum com o tipo de configuração que deseja realizar, e o segundo é o valor que deseja estar nessa configuração, existem diversas opções de configuração e podem ser encontradas aqui http://www.glfw.org/docs/latest/window.html.

Nesse caso , GLFW_CONTEXT_VERSION major e minor, especifica qual versão exigida do OpenGL para nosso contexto nesse caso (3 ou mais) , quem nao possuir essa versao, nao irá conseguir rodar o programa. Em OPENGL FOWARD COMPAT é uma hint usada em computadores em que o contexto do OpenGL criado poderá ser menor que o exigido, no caso de computadores com OSX,como o meu, se eu nao utilizar essa hint, ele cria um contexto OpenGL 2.1, quando ativo essa hint, ele passa a usar o maior contexto que meu computador suporta , no caso OpenGL 4.2.

Em Core Profile, especifica que queremos trabalhar com o modelo do modern OpenGL , assim sempre que tentarmos usar alguma função obsoleto o programa não irá funcionar.

GLFW_RESIZABLE = FALSE quer dizer que não queremos uma janela redimensionavel pelo usuário.

```
  //create a window in a system
    //Cria uma janela no sistema
    window = glfwCreateWindow(SCREEN_SIZE_X,
SCREEN_SIZE_Y,"OpenGL Tutorial", NULL, NULL);

    if (!window)
    {
        glfwTerminate();
        exit(EXIT_FAILURE);
    }
```

Agora em window, estamos recebendo para nosso objeto de janela uma janela criada pelo GLFW, o tamanho da tela foi definido no cabeçalho do programa, o terceiro argumento é o nome da nossa janela, os últimos parâmetros podem ser ignorados, pois só precisamos dessas 3 configurações para nossa tela.

```
    // create a opengl context in this window, always
before the initialize opengl in  glew, because for glew
initialize, it should have a context  for init.
    //Cria um contexto OpenGL em uma janela, sempre antes
de inicialiar o OpenGL pelo glew, deve se existir um
contexto, para o glew ser criado dentro do mesmo
    glfwMakeContextCurrent(window);



    //Initialize glew
    //Inicializa o GLEW
    glewExperimental = GL_TRUE;
    if(glewInit() != GLEW_OK)
        throw std::runtime_error("glewInit failed");
```

Definimos como glewExperimental = TRUE para que possamos usar funções modernas do glew para usar técnicas modernas no controle do OpenGL , se definirmos FALSE podemos ter problemas ao trabalhar com o Core Profile.

```
    glViewport(0, 0, SCREEN_SIZE_X, SCREEN_SIZE_Y);
```

Essa função diz ao OpenGL as configurações de dimensão da nossa tela de renderização. O primeiro e segundo parâmetro, informa a localização do canto inferior esquerdo da tela. Como vamos usar a tela toda para desenhar, no terceiro e quarto parâmetro, passamos novamente o tamanho da tela(lembre-se do objetos, configuramos a maquina de estado para desenhar num espaço do tamanho da nossa tela, poderíamos definir um espaço menor que a tela, e no espaço restante escrever outras coisas até mesmo não relativas ao OpenGL).

***Por traz das cenas, o glViewport transforma o sistema de coordenas do OpenGL(que varia de 0 a -1 a 0 a 1) para o sistema de coordenadas que pedimos (0 á 800) e (0 a 600), quando desenhamos uma figura nas coordenadas (-0.5,0.5) no final ela será desenhada nas coordenadas (200,450).**

```
//Check API 3.2 is avaible
//Checa se o OpenGL 3.2 existe
    if(!GLEW_VERSION_3_2)
        throw std::runtime_error("OpenGL 3.2 API is not
available.");
    // print out some info about the graphics drivers
    // Imprime no console as informações sobre as drivers
gráficos do contexto criado
    std::cout << "OpenGL version: " <<
glGetString(GL_VERSION) << std::endl;
    std::cout << "GLSL version: " <<
glGetString(GL_SHADING_LANGUAGE_VERSION) << std::endl;
    std::cout << "Vendor: " << glGetString(GL_VENDOR) <<
std::endl;
    std::cout << "Renderer: " << glGetString(GL_RENDERER)
<< std::endl;
    // return a window create, for use in main
    //Retorna um janela criada para ser usada no programa
principal
    return window;
}
```

Loop Principal

Agora vamos criar o nosso loop principal, onde tudo que quisermos desenhar será dentro desse loop, pois se não for em um loop, o desenho irá ser desenhado uma vez em apenas um Frame, e o programa irá fechar, isso não é nem um pouco interessante para nós(não dará nem tempo de ver o desenho na tela). Assim vamos criar esse loop no nosso arquivo de programa principal, o famoso main.cpp. Que irá

chamar receber uma janela pela nossa função recém criada. O nosso loop principal terá o seguinte formato

```
#include "Program.h"
GLFWwindow* mainWindow;
int main(int argc, const char * argv[])
{
Program firstProgram = Program();
    //Window recebe a nossa janela criada pela função que vimos anteriormente.
    mainWindow = firstProgram.initOpenGL();

//Verifica se a janela criada foi fechada a cada iteração.
    while(!glfwWindowShouldClose(mainWindow)){
```

glfwWindowShouldClose(GLFW window), verifica se a nossa janela foi fechada por alguma função ou usuário.**(Exemplo: clicando no famoso botão X vermelho), enquanto ela nao for fechada o nosso loop será executado.)**

```
//Verifica todos os eventos e chama as funções relacionadas a esses eventos
        glfwPollEvents();
```

glfwPollEventos, verifica os eventos definidos, e executa as funções que foram agregadas a eles.**(por exemplo queremos que sempre que apertar a tecla ESC a janela feche, logo mais veremos como isso será feitos. Mas o evento padrão de fechar a janela quando clivamos no X só funcionará se verificarmos ele a cada frame.)**

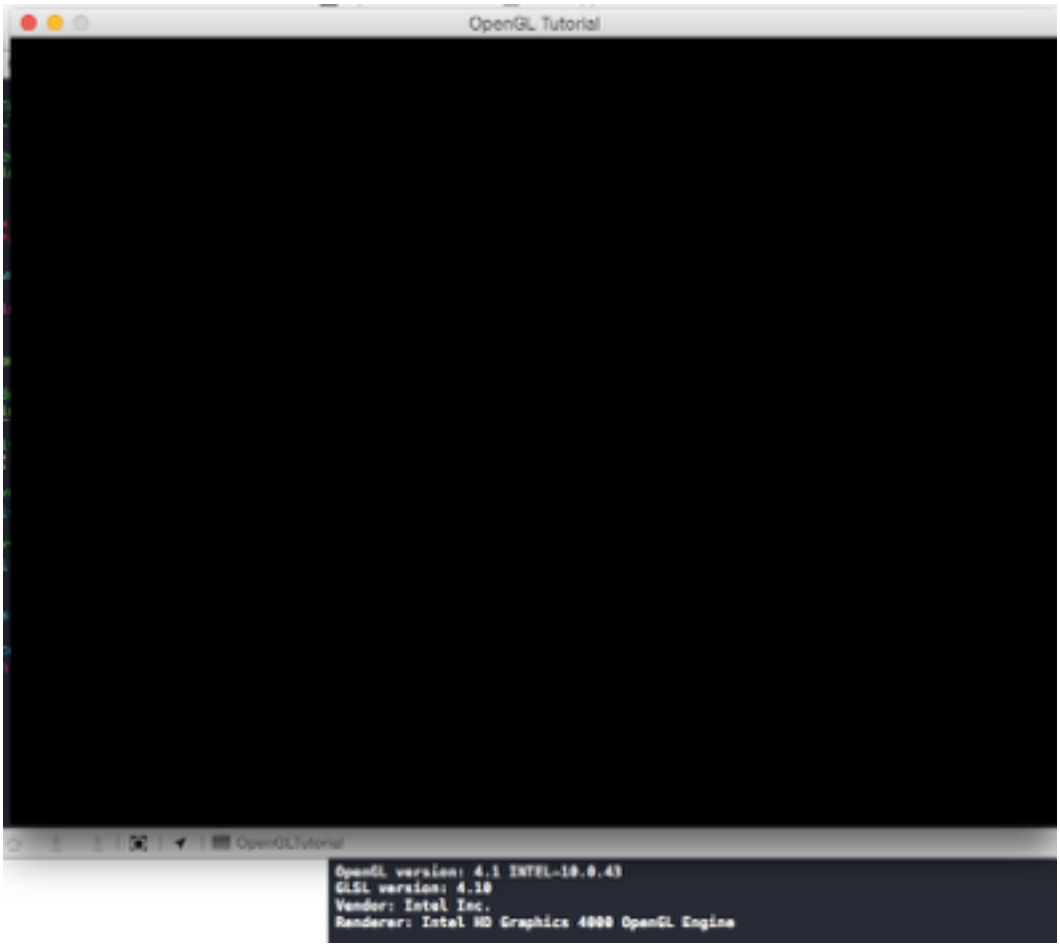
```
        //Troca o buffer de cor a cada iteração pelo definido.
        glfwSwapBuffers(mainWindow);
```

glfwSwappBuffers troca o buffer de cor que foi processado e o faz aparecer na tela. **(DOUBLE BUFFER: quando a aplicação usa um único buffer, pode acontecer alguns bugs e a imagem a ser a apresentada ser deformada, porque a imagem a ser apresentada não é desenhada totalmente em um único instante, ela é desenhada pixel por pixel, geralmente da esquerda pra direita, de cima pra baixo, assim se utilizar um único buffer, pode acontecer de o processamento da imagem a ser apresentada se atrasar ao passo que irá apresentar. Para solucionar isso, se utiliza double buffer,**

enquanto uma imagem está aparecendo na tela (SAÍDA) exit um buffer por trás que está processando a nova imagem a ser apresentada(ENTRADA) assim a imagem pré processada é apresentada em um único instante após terminar de ser renderizada(processada).

```
 }
    glfwTerminate();
    std::cout << "Hello, World!\n";
    return 0;
}
```

Se você estava somente lendo o tutorial, agora é hora de tentar executar, o código fonte estará disponível, mas é importante que você tente executar por si só. Se ja estava executando simultaneamente, quando compilar e executar o programa, você deverá ver uma janela aberta, na cor preta, que é a padrão para o buffer de renderização). como a seguinte:



Note que no console aparecerá as informações sobre seus drivers gráficos como mostrei no código na função initOpenGL(). É importante para se certificar de qual versão do openGL que o contexto esta sendo criado, que não pode ser inferior a 3.3.

Input

Agora iremos aprender a controlar as entradas do usuário no nosso programa, isso pode ser feito com o GLFW usando as funções GLFW callback. Essas funções são funções que você pode definir usando um ponteiro de função para que o GLFW chame quando você precisar. Um exemplo é ao apertar uma tecla, para isso usaremos as funções de KeyCallBack que processa as entradas de usuário. Iremos sobrescrever essa função do GLFW no nosso arquivo Program.cpp, após a inicialização da classe com o seguinte código.

```
Program::~Program(){};

// Is called whenever a key is pressed/released via GLFW
// É chamado sempre que uma tecla é pressionada ou
solta , através do GLFW
void key_callback(GLFWwindow* window, int key, int
scancode, int action, int mode)
```

O primeiro parametro (window) é a janela sobre a qual ela irá operar, o segundo (key) parâmetro é um inteiro informando qual tecla foi pressionada, o terceiro (scancode) será ignorado por agora, mas está relacionado a definir novas entradas de teclado não especificadas pelo GLFW, o int action indica se a key foi pressionada, solta ou se está sendo pressionada ainda , e o mods indica se ela foi pressionada em conjunto com alguma tecla especial como Shift , Control, etc. mais especificações podem ser encontrada nesse endereço da documentação oficial do GLFW

http://www.glfw.org/docs/latest/group__input.html

{

```
    std::cout << key << std::endl;
    if(key == GLFW_KEY_ESCAPE && action == GLFW_PRESS)
        glfwSetWindowShouldClose(window, GL_TRUE);
}
```

Aqui estamos imprimindo no console a tecla que foi pressionada e verificando se foi pressionada a tecla escape se sim iremos fechar a janela configurando o glfwSetWindowShouldClose(window, GL_TRUE).(se nao entendeu, volte ao tutorial do loop principal e se recorde).

Agora dentro da função initOpenGL iremos definir, antes de retorna a janela criada, para qual função o GLFW KeyCallBacks irá chamar , e em qual janela ela irá trabalhar, faremos isso com o seguinte código:

```
    //Define qual funcao de KeyCallBack será chamada e
para qual janela ela irá funcionar.
    glfwSetKeyCallback(window, key_callback);

    // return a window create, for use in main
    //Retorna um janela criada para ser usada no programa
principal
    return window;
```

Rode o programa e confira, que se ao pressionar a tecla ESC a sua janela vai fechar. Se sim, vamos entender um pouco da renderização no próximo passo :D .

Rendering

Todos os comandos de renderização devem estar dentro do nosso loop principal, anteriormente criar na função main.cpp

Geralmente uma rotina de renderização é estruturada da seguinte maneira:

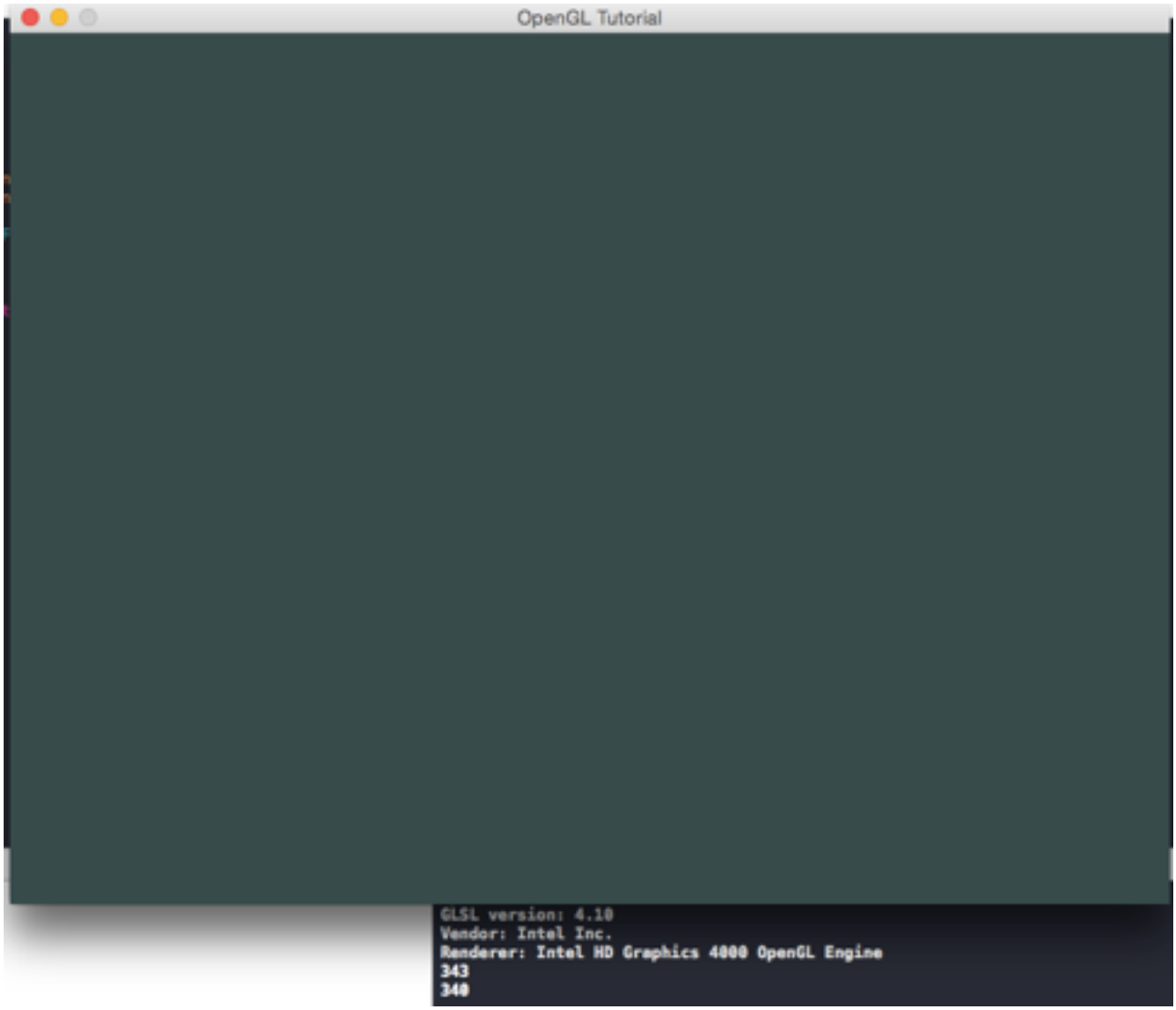
```
// Program loop
while(!glfwWindowShouldClose(window))
{
    // Check and call events
    glfwPollEvents();

    // Rendering commands here
    // Comandos de desenho aqui
    ...
```

```
    // Swap the buffers
    glfwSwapBuffers(window);
}
```

Só para uma introdução a esse funcionamento vamos escolher uma cor na hora de limpar a tela. No início da iteração de renderização vamos sempre limpar a ultima renderização feita, usando o glClear(), se você não usar o glClear, ele não irá limpar a tela antes de renderia uma nova e você poderá ver pedaços da anterior antes da atual estar completamente renderizada. A função glClear pode limpar a nossa tela de várias formas, utilizaremos a função de limpar com cores passando o parâmetro GL_COLOR_BUFFER_BIT . no glClearColor, iremos especificar com qual core queremos que a tela seja limpa.

```
glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT);
```

Se você entendeu como as coisas funcionam, isso deve ser colocado antes do glfwSwapBuffers e após o glfwPollEvents(). Se fez tudo corretamente e com as mesmas cores definidas, deve obter algo do tipo:



Veja que o console imprimiu dois números, que correspondem as teclas command+shift que no meu caso no OSX me auxiliam a tirar um print screen.

Desafio 1.

Você ja conhece a função de entradas do teclado do GLFW3, e também já sabe o básico de como funciona o OpenGL. Então está preparado fazer nossa primeira animação em OpenGL, se você acompanhou atentamente os tutoriais com atenção(principalmente para os conceitos) você vai conseguir completar esse desafio em instantes.

O primeiro desafio consiste em, definir a cor de limpeza de buffer para a cor branca. E ao usuário clicar em alguma tecla(a sua escolha, exceto a tecla ESC que ja está implementada) a tela deverá piscar na cor preta. É basicamente isso, a tela fica branca por default, se você apertar alguma tecla a tela deverá piscar na cor preto. Lembre-se dos conceitos de funcionamento do OpenGL, é um exercício muito simples. É importante que avance para a próxima fase só após concluir o desafio, para garantir que conseguiu fixar bem os conceitos.

Resposta:

alterar:

```
glClear(GL_COLOR_BUFFER_BIT);

glClearColor(1.0f, 1.0f, 1.0f, 1.0f);

glfwPollEvents();
```

Program.cpp

criar:

```
if(key == GLFW_KEY_UP && action == GLFW_REPEAT)

glClearColor(0.0f, 0.0f , 0.0f , 0.0f);
```

Primeiro Triângulo

Se você não conseguiu completar o desafio do módulo anterior, ou se conseguiu mas com muita complicação, vou fazer umas colocações.

1. Lembre-se que o OpenGL funciona como se fosse uma máquina de estados, então se você mudar alguma estado dele, não importa aonde for no código, ele passara a desenha tudo com aquele estado que você configurou.

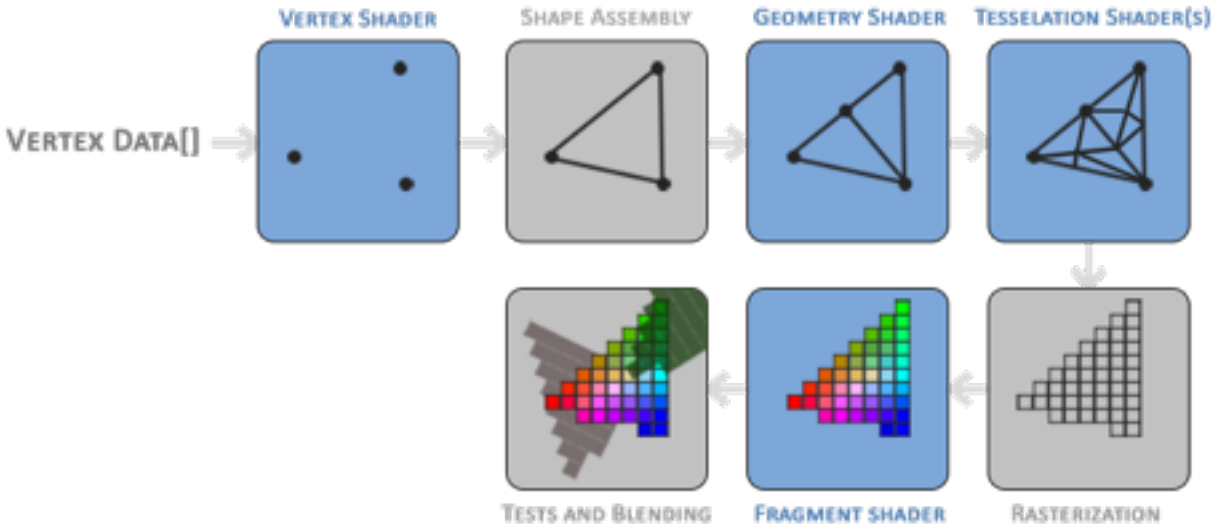
2. Se você conseguiu resolver o desafio, mas usou mais do que 5 linhas de código, volte e faça novamente, otimize seu programa, para fixar mais ainda o conhecimento, eu particularmente venci esse desafio, mudando 1 linha de código de lugar e criando mais 2 linhas de código.

Mais sobre OpenGL moderno

Você nunca se perguntou como conseguimos desenhar coisas em 3 dimensões sendo que possuímos uma tela de duas dimensões ? O trabalho do OpenGL é basicamente este, converter nossas coordenadas 3D para coordenadas 2D de pixels em sua tela, e este processo de transformar os gráficos 3D é gerenciado pela pipeline gráfica que geralmente se divide e duas partes fundamentais: Transformar as coordenadas de 3D para 2D e depois pintar as coordenadas 2D em pixels coloridos. Vamos entender uma pouco mais como funciona o pipeline gráfico nesse tutorial.

Primeiramente, vale ressaltar, que uma coordenada 2D é diferente de um pixel 2D, uma coordenada 2D é uma representação precisa de um ponto em um espaço 2D, um pixel 2D é a aproximação da posição de um ponto em 2D nos limites de resolução da sua tela. O pipeline gráfico consiste em processar uma saída de uma entrada que foi realizada em um passo anterior. Ele é processado pela placa de vídeo, e possui funções específicas para cada efeito que se deseja obter. As placas gráficas contem milhares de processadores que executam essa tarefa do pipeline gráfico, em pequenos programas, que são chamados de **shaders**.

Muitos desses shaders podem ser programados pelo próprio usuário, o que é fascinante, permitindo-nos escrever nossos próprios shaders em cima dos padrões, dando mais flexibilidade nas nossas aplicações e por eles funcionarem na GPU , poupando muito processamento do nosso CPU. Os shaders são escritos na linguagem do OpenGL Shading Language (GLSL). Vai aqui uma representação abstrata do funcionamento de um shader.



Percebe-se que para se ter um triângulo desenhado na tela , há uma série de passos a se fazer. O nosso Vertex Data é uma coleção de Vertex(vértices). Vértices são

um conjunto de dados a serem enviados a um pipeline gráfico onde um conjunto de vértices criam uma forma geométrica(3 vértices formam um triângulo). Vertex Data é um conjunto de vértices que podem conter qualquer tipo de forma que quisermos. No nosso caso vamos considerar que contém posições 2D/3D e uma cor.

Para o OpenGL trabalhar com os nossos dados de vértices, precisamos informar a ele qual tipo de processamento queremos usar,se queremos desenhar pontos, linhas, triângulos. Essas informações são chamadas de primitivas de desenho e podem ser GL_POINTS, GL_LINE, GL_QUADS, GL_TRIANGLE, etc. São passados para o OpenGL antes das etapas de desenho.

A primeira parte do pipeline gráfico é chamado de vertex shader(shader de vértice), que tem como entrada um único vértice, onde realiza principalmente a trasnformação desse vértice para outro sistema de coordenadas(3D para 2D por exemplo) , e também, processamentos com as informações de atributos do vértice.

A fase de montagem(Shape Assembly) da primitiva pega todos os vértices que formam uma primitiva e monta todos os pontos na forma geométrica a se obter(por exemplo triângulo). O retorno dessa fase, é enviado ao Geometry Shader. O geometry shader tem como entrada um conjunto de vértices, e através dele, consegue montar qualquer tipo de forma que desejar, retornando novos vértices para formar uma nova forma primitiva. No exemplo da figura acima, ele gera um segundo triângulo para a forma processada.

O Shader Tesselation pega uma primitiva e divide ela em muitas outras primitivas menores. Isso permite criar um efeito de suavização(smooth) gerando triângulos menores por exemplo. O Shader Tesselation gera um retorno que é recebido pelo Rasterization que mapeia os resultados das primitivas para cada pixel, gerando um monte de fragmentos que é enviado para o Fragment shader.

Antes do fragment shader ser executado é executado o clipping que corta todos os fragmentes que não á sua vista**(um exemplo, voce tem um cubo com uma face frontal e outra oposta, voce vai conseguir ver só uma face por vez, a face que voce nao ver então é cortada pelo clipping para que nao seja renderizada desnecessariamente)** , gerando uma performance maior

Um fragment no OpenGL é contém todo o tipo de dado que o OpenGL precisa para renderizar um pixel. O papel do fragment shader é pegar esse dados e definir o resultado final da cor de um pixel. É onde são processados a maior parte dos efeitos do OpenGL moderno, que geram aqueles gráficos incríveis, são processados ai dados como : Iluminação, sombra, reflexão, cores da luz, etc).

Depois de todos os valores serem definidos no Fragment Shader, no OpenGL ai acontece mais uma etapa que são Alpha Test e Blending Test. Esses testes são

relacionados a profundidade do objeto a ser desenhado e o Stencil Value. Ai ele verifica o valor de profundidade para ver qual objeto será desenhado na frente do outro, assim descartando o de trás que não irá aparecer. Também verifica o valor alpha, que está relacionado a opacidade, assim dependendo da opacidade de um objeto sobre outro a cor resultante pode ser diferente. O que quer dizer que mesmo definindo uma cor no shader a cor resultante pode ser uma cor diferente.

Percebe-se que o pipeline programável é bastante robusto, e complexo também. Uma série de configurações devem ser feitas para se obter um resultado. Mas geralmente só é sobrescrevido o vertex e o fragment shader, deixando o restante como o default do frabicante. Para desenhar um triângulo então precisaremos definir o mínimo que são os vértices e sua cor.

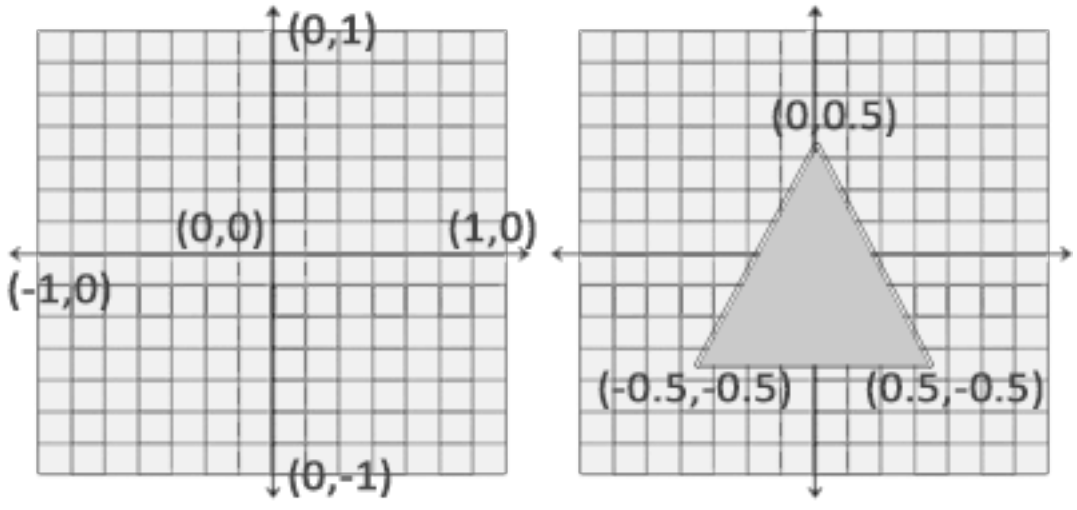
Vertex Input

Para desenhar algo na tela temos que passar então as coordenadas do objeto a ser desenhado, iremos então passar as coordenadas dos três vértices do triângulo a ser desenhado, sendo coordenadas normalizadas para se encaixarem ao limite visível do nosso contexto OpenGL. Definiremos em uma matriz

```
GLfloat vertices[] = {
     -0.5f, -0.5f,
     0.5f, -0.5f,
     0.0f, 0.5f
};
```

Normalized Device Coordinates (NDC)

As coordenadas dos vértices devem estar no sistema NDC de coordenadas que são valores que variam de 1 á -1. Se for passada uma coordenada fora dessa faixa não será mostrada na tela, por exceder os limites da tela.



O sistema de coordenadas tem como o ponto inicial (0,0), o centro da tela. se queremos um desenho mais a esquerda então passamos o ponto x como negativo de 0 á -1, e assim por diante. As coordenadas passadas no sistema NDC são transformadas depois para coordenadas do espaço pelo viewport transformation usando as configurações definidas no glViewPort() anteriormente, esse resultado é então transformado em fragmentos para ser enviado ao fragment shader.

Já definimos nosso Vertex Data, o próximo passo é passar esses dados para Fragment Shader. Isso será feito através da criação de um objeto de buffer, onde serão armazenado os vértices, configurado o layout, e definido como será enviado o buffer para a placa gráfica.

Para enviar dados de vértice a uma placa gráfica, precisaremos de um vertex buffer object (VBO). Esse buffer como qualquer objeto OpenGL tem uma identificação única que pode ser gerado com a função glGenBuffers().

```
GLuint VBO;

glGenBuffers(1, &VBO);
```

O OpenGL tem diversos tipos de objetos de buffer e tipo de buffer de VBO é o GL_ARRAY_BUFFER. O OpenGL nos permite ligar diversos tipos diferentes de buffer se precisarmos. Então vamos ligar nosso buffer criado com a função glBindBuffer:

```
glBindBuffer(GL_ARRAY_BUFFER, VBO);
```

A partir daí toda vez que solicitarmos alguma função de array de buffers será utilizando esse buffer ligado. Após isso, deve-se copiar o dados de vértice para o buffer. Com a função glBufferData(), que é uma função que copia os dados definidos pelo usuário dentro do buffer que está ligado atualmente. Os argumentos dessa função são.

1. o tipo de buffer(nesse exemplo GL_ARRAY_BUFFER).
2. Diz qual é o tamanho dos dados que serão passados para o buffer(o tamanho dos nossos dados de vértice).
3. São os dados que serão passados(os vértices).

4. Ultimo argumento, e define como a placa gráfica irá gerenciar esses dados, possui três tipos diferentes:
    1. GL_STATIC_DRAW: os dados são sempre os mesmos.
    2. GL_DYNAMIC_DRAW: os dados mudam muitas vezes.
    3. GL_STREAM_DRAW: os dados mudam sempre que vão ser desenhados

No nosso exemplo de desenhar um triângulo se configuraria assim:

```
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
```

Vertex Shader

Agora já foram definidos os dados necessários para desenhar algo na forma de VBO, então precisamos criar o nosso programa de shader que irá processar esses dados. No OpenGL moderno, para apresentar algo na tela deve se criar no mínimo o vertex shader e o fragment shader. Então vamos fazer uma leve introdução de como construir esses programas em GLSL. Vamos escrever o nosso Vertex Shader :

```
#version 330 core

layout (location = 0) in vec2 position;

void main()
{
    gl_Position = vec4(position.x, position.y, 0.0,
1.0);
}
```

Esse um código suficientemente simples de um vertex shader, mas para desenhar um triângulo não precisamos mais do que isso. O código GLSL é similar a um código C, o shader criado tem as seguintes características:

1. Sempre no começo deve se declarar a versão do GLSL que está sendo usada do OpenGL3.3 e superior, a versão do GLSL corresponde a versão do OpenGL, nesse caso 3.3 declara-se 330, se for 4.0 declara-se 400. Junto com a versão utilizada está mencionado que vamos utilizar o modo Core Profile.

2. O layout location define especificamente a localização da variável de entrada. Mais a frente a utilidade disso será discutida.
3. Junto com o layout location é declarado todas as variáveis do vértice de entrada do shader. Nesse caso só será usado os dados da posição do objeto a ser desenhado. Será desenhado um objeto 2D no espaço 3D, por isso a entrada é um vec2 que representa um vetor com 2 coordenadas.
4. Em gl_Position processamos a saída do shader, que é um vetor vec4. Ja a nossa saída é um vetor ver4, e a entrada é apenas um vetor vec2, realizamos a transformação invocando um construtor vec4, colocando nele os valores do vec2 e passando o valor W para 1(isso será esclarecido mais á frente).
5. Passamos o valor de z para 0, porque estamos representando uma figura 2d no sistema 3d, assim 0 não será representado.

Assim definimos o shader de vértice mais simples possível, nota-se que nele não opera nenhum calculo. Em aplicações reais geralmente os dados do vértice não são no sistema NDC, assim a conversão dos dados para o sistema NDC é realizada dentro do shader.

Compilando o Shader

Agora com o código fonte do shader escrito, precisamos compilar ele em tempo de execução , para que o OpenGL possa usa-lo.

Então iremos criar um objeto de vertex shader , e guardaremos sua referencia em uma variável do tipo GLint, para isso definiremos uma variável do tipo GLint que irá guardar nosso shader criado, o shader será criado utilizando a função glCreateShader() , que leva como parâmetro o tipo de shader que será criado:

```
GLint vertexShader = glCreateShader(GL_VERTEX_SHADER);
```

Após o shader ser criado, deve-se ligar o código do shader o shader criado, isso será feito usando a função glShaderSource() que recebe os seguintes parâmetros

1. O shader criado.
2. Especifica quantas strings serão passadas relativo ao código fonte.
3. É a string com o código fonte do shader.

4. O tamanho do array de strings que será passado, nesse caso não será passado um array de strings, e sim, somente uma string, então definimos como NULL.

```
glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
```

Com o shader criado e ligado ao código fonte, podemos compilar, usando a seguinte função glCompileShader() que tem como parâmetro o shader criado:

```
glCompileShader(vertexShader);
```

Uma dica que poderá ajudar muito, é criar um log de erro, caso o shader não seja compilado com sucesso, assim ficará mais fácil de encontrar o problema associado ao shader. Isso pode ser feito da seguinte maneira:

```
GLint success;
GLchar infoLog[512];
glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &success);
```

O glGetShaderiv() retorna um parâmetro do Objeto do Shader, os parâmetros são

1. O shader que será solicitado o parametro.
2. Qual parâmetro voce esta solicitando(O estado de compilação).
3. Onde será salvo o parâmetro. Nesse caso ele irá verificar o estado de compilação e salvar na nossa variável criada acima, GLint sucess. Apos isso iremos tomar uma decisão caso ocorra um erro:

```
if(!success)
  {
    glGetShaderInfoLog(vertexShader, 512, NULL, infoLog);
```

```
    std::cout << "ERROR::SHADER::VERTEX::COMPILATION_FAILED\n" << infoLog <<
std::endl;
  }
```

Verificamos nossa variável que guarda o estado de compilação caso o retorno seja falso, não foi compilado com sucesso, ai então iremos imprimir o erro de compilação. Para imprimir esse erro usaremos a função glGetShaderInfoLog(), que tem como parâmetros:

1. O shader a se obter o log.
2. Especifica o número de caracteres que conterá log.
3. O tamanho da string que conterá o log. Será NULL porque não estamos usando uma string.
4. Onde será guardado o log.

Após associar o log a variável, iremos imprimir, usando a função conhecida std::cout, passando a variável infolog.

Com isso o processo de compilar o vertex shader terminou.

Fragment Shader

Cores em computação gráfica são representadas em RGBA que é um vetor com 4 valores, red(vermelho), green(verde), blue(azul) e alpha(opacidade). Para definir uma cor no OpenGL no GLSL define-se a intensidade de cada cor com valores do tipo float entre 0 e 1(como havíamos exercitado no exercício de limpeza de buffer). Quanto definimos a intensidade de duas cores 0.2 red e 0.4 green estamos misturando essas duas cores, assim com 4 valores conseguimos gerar 16 milhões de cores diferentes.

O fragment shader irá computar as cores do triângulo a ser exibido. Então o mesmo irá conter apenas uma saída, que é declarada após a especificação da versão.

```
#version 330 core
  out vec4 color;
  void main()
  {
    color = vec4(1.0f, 0.5f, 0.2f, 1.0f);
  }
```

A saída será um vetor vec4 com o valor das cores no formato RGBA, assim definimos a cor de saída , que receberá um vetor vec4 com a cor que desejarmos.

Para compilar o fragment shader é realizado um processo similar ao do vertex shader, alterando somente o tipo de shader a ser criado.

```
GLint fragmentShader;
fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);
glCompileShader(fragmentShader);
glGetShaderiv(fragmentShader, GL_COMPILE_STATUS, &success);
if(!success)
{
    glGetShaderInfoLog(fragmentShader, 512, NULL, infoLog);
    std::cout << "ERROR::SHADER::VERTEX::COMPILATION_FAILED\n" << infoLog << std::endl;
}
```

Program Shader

Com os shaders compilados, precisamos liga-los a um programa de shader, para assim, realizarmos as operações relacionadas a shaders.

Um programa de shader, é uma versão que contem os seus shaders compilados, lembra no diagrama os shaders se relacionavam um com o outro? Por enquanto nós so compilamos os shaders, nesse programa nos iremos ligar a saída de um shader com a a entrada de outro shader. Aqui erros irão acontecer se a saída de um shader for de um tipo diferente da entrada do próximo shader.

```
GLint shaderProgram = glCreateProgram();
```

glCreateProgram() cria um programa de shader, você ja deve estar se familiarizando com objetos OpenGL então já sabe que estamos guardando o programa criado em uma variável. Agora precisamos colocar os shaders compilados nesse programa criado. para isso é usado a função glAttachShader:

```
glAttachShader(shaderProgram, vertexShader);
glAttachShader(shaderProgram, fragmentShader);
```

Com os shaders no nosso programa, agora iremos liga-lo, isso é um processo que pode gerar erros, então iremos também usar um log semelhante ao visto anteriormente na criação do vertex shader. Para ligar o programa de shaders usa-se a função glLinkProgram():

```
glLinkProgram(shaderProgram);
glGetProgramiv(shaderProgram, GL_LINK_STATUS, &success);
if(!success){

    glGetProgramInfoLog(shaderProgram,512, NULL, infoLog);
    std::cout << "ERROR::SHADER::VERTEX::COMPILATION_FAILED\n" << infoLog << std::endl;

}
```

glGetProgramiv e glGetProgramInfoLog sao similar as funções de log vistas anteriormentes só que essas são relacionadas a um programa de shader. Uma vez que o programa foi ligado, os shaders já estão dentro do mesmo, não precisamos mais do shaders criados, então vamos deleta-los para economizar memoria. Isso pode ser feito da seguinte maneira

```
glDeleteShader(vertexShader);
```

```
glDeleteShader(fragmentShader);
```

Quando o programa é criado com sucesso, ele ainda não está sendo usado, assim para ele começar a ser usado pelo OpenGL, usa-se a função glUseProgram():

```
glUseProgram(shaderProgram);
```

A partir daí, cada vez que a função de desenhar for solicitada,essa função que informa o programa a ser usado deverá ser usada, assim sempre irá desenhar com esse programa.

Linking Vertex Atributes

Estamos usando osos programa de shaders, mas o OpenGL ainda não sabe como interpretar os dados de vértice e como será essa conexão com os atributos do vertex shader. Para informar ao OpenGL como interpretar os dados, pode-se usar a função glVertexAttribPointer() , da seguinte maneira:

```
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0, (GLvoid*)0);
glEnableVertexAttribArray(0);
```

glVertexAttribPointer() possui os seguintes parâmetros:

1. Informa para qual atributo queremos passar os dados,lembre-se que no shader foi definido o location=0, então os dados serão passados par ao atributo position que possui location=0.
2. Informa quantos valores que compõe o atributo, o atributo position é um vec2 então é composto por 2 valores.
3. Informa qual o tipo de valor que compõe esse atributo(vec2 está recebendo floats).
4. Informa se é para normalizar os valores antes do atributo, como os valores definidos ja estão normalizados então está configurado GL_FALSE.

5. Especifica quanto de espaço vai ter entre cada atributo, se tiver mais de um vertex attribute , deve se informar o espaço entre cada um. Nesse caso só a um vertex attribute definido, assim é 0.
6. Define o deslocamento de onde os dados de posição irão começar, a posição inicial é o início vetor esta começando de 0. Por esse atributo ser do tipo void, foi realizado um cast.

Em glEnableVertexAttribArray(), como nome diz, estamos ativando os atributos de vértice passando como parâmetro a localização(location).

Já está tudo pronto, mas sempre que formos desenhar algo deveremos fazer esse processo todo, a rotina de desenho iria ficar algo parecido com isso:

```
// 0.Liga o array de vértices a um objeto de buffer a ser usado pelo OpenGL.
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
// 1. Configura a interpretação dos atributos
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0, (GLvoid*)0);
glEnableVertexAttribArray(0);
// 2. Informa o programa de shader que será usado para desenhar
glUseProgram(shaderProgram);
// 3. Desenha o triangulo
funcaoDeDesenharUmTrianguloOpenGL();
// 4.Desliga o objeto de buffer
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

Você terá que repetir esse processo todas as vezes que for desenhar algo, imagine então se tivéssemos 5 atributos de vértice diferentes, em uma cena em que você desenha mais de 50 objetos que são diferentes ? Pensando nisso o OpenGL criou o Vertex Array Objects(VAO).

Vertex Array Objects

Um vertex array object, se liga dados configurados de um VBO , e a partir dai todas as vezes que ele um VAO é ligado ele traz as configurações que foram feitas pelo VBO e atributos. Ja deve estar imaginando então como vamos simplificar aquela rotina criada acima. A regra é sempre ligar o VAO antes de configurar os buffers. Criar um VAO é semelhante a criar um VBO :

```
GLuint VAO;
glGenVertexArrays(1, &VAO);
```

Após criar o VAO é só configurar o VBO e todas as suas configurados estarão armazenadas no VAO assim a rotina completa ficaria assim

```
// 1. Liga o VAO
glBindVertexArray(VAO);
// 2. Copia o array de vertices na variavel VBO para o OpenGL usar.
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
// 3. Configura a OpenGL para interpretar os atributos de vertice
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 2, (GLvoid*)0);
glEnableVertexAttribArray(0);
//4. Desliga o VAO
glBindVertexArray(0);

… // Em outra parte do Código

// 5. Desenha o Objeto
glUseProgram(shaderProgram);
glBindVertexArray(VAO);
```

```
funcaoParaDesenhaUmTrianguloNoOpenGL();
glBindVertexArray(0);
```

Nota-se que a rotina dentro do loop de renderizacão o ficou bem mais simples, baste sempre ligar o VAO antes de desenhar e após desenhar desliga-lo.

É importante sempre desligar o VAO e ligar somente no instante antes de desenhar, pois se ele estiver ligado em algum outro lugar do programa pode gerar algum bug, lembre-se que ele salva todas as configurações do buffer, durante o momento que ele estiver ligado pode ser salva alguma configuração indesejada.

Enfim todas as configurações de desenho que é preciso no momento estão definidas, normalmente se fosse desenhar vários objetos, bastava criar um VAO para cada objeto, e na hora de desenhar esse objeto ligar seu VAO e após desenhar desligar.

Organizando

Bem, foi especificado um monte de funções, agora vamos organiza-las em nas classes já criadas, primeiramente na Program.h, serão definidas dois métodos públicos logo abaixo do já criado initOpenGL:

```
GLFWwindow* initOpenGL();
void setupGLSL();
void render();
```

setupGLSL() irá fazer todas as configurações e ligações de buffer que vimos acima.

render() será composto com as funções de desenho. No Program.cpp serão definidas algumas variáveis e constantes globais abaixo da primeira variável já definida:

```
GLFWwindow* window;

GLuint VAO;

GLuint shaderProgram;



//String com o codigo do vertex shader

const GLchar* vertexShaderSource = "#version 330 core\n" //Informa versao do shader e que é do tipo core profile

"layout (location = 0) in vec2 position;\n" //define a localizacao do atributo de entrada de vertice e declara o atributo

"void main()\n"

"{\n"

"gl_Position = vec4(position.x, position.y, 0.0, 1.0);\n" //retorna a posicao 2D transformada para o plano 3D

"}\0";


//String com o codigo do fragment shader

const GLchar* fragmentShaderSource = "#version 330 core\n"

"out vec4 color;\n" //define o atributo de saida

"void main()\n"

"{\n"

"color = vec4(1.0f, 0.5f, 0.2f, 1.0f);\n" //retorna uma cor RGBA como saida de cor

"}\n\0";
```

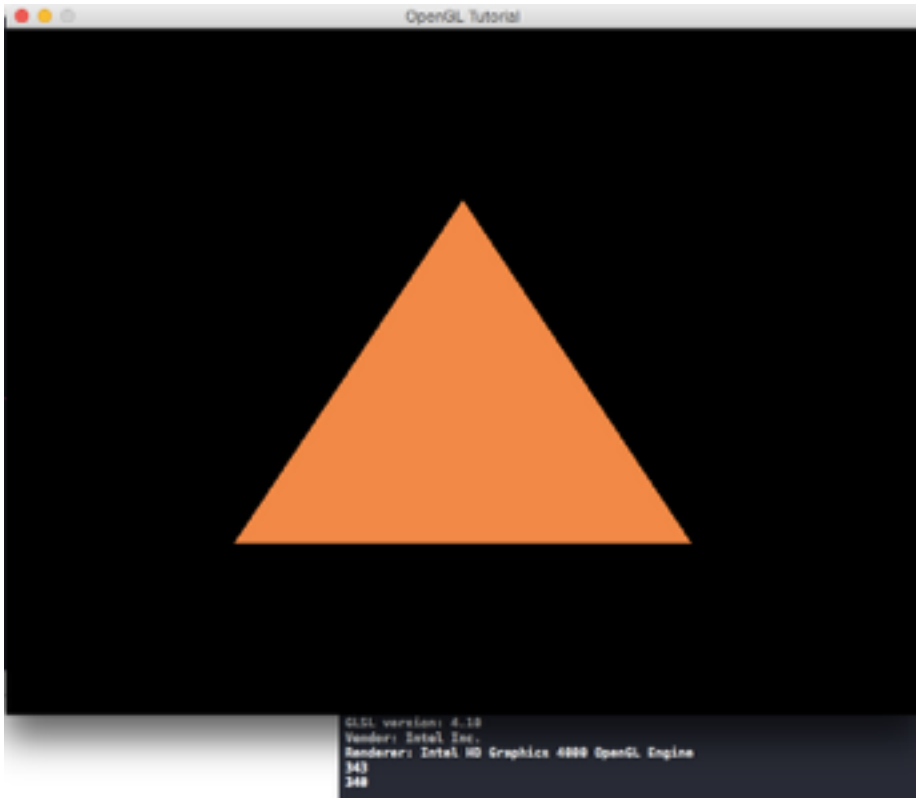
Os comentários já estão especificando bem, foi definido o VAO e shaderProgram que será usado em vários locais da classe. E foram definidas duas strings com o código dos shaders a serem compilados

Após isso iremos implementar as funções definidas no .h, começando pela função setupGLSL:

```
void Program::setupGLSL(){
//* configuracoes de buffer
  //Define vertices que serao desenhados ( triangulo)
   GLfloat vertices[] = {
      -0.5f, -0.5f,
      0.5f, -0.5f,
      0.0f, 0.5f
   };
   //Cria um objeto de array de vertices
   glGenVertexArrays(1,&VAO);
   GLuint VBO;
   //Cria um objeto de buffer de vértices
        glGenBuffers(1, &VBO);

   //Liga o array de objetos de vertice
   glBindVertexArray(VAO);

   //Copia o buffer para o objeto de buffer a ser usado pelo OpenGL
   glBindBuffer(GL_ARRAY_BUFFER, VBO);
```

```
    //Copia os dados de vertices definidos para o objeto de buffer, e configura como será desenhado
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

    //Envia informacoes ao OpenGL de como será os atributos dos vertices passados ao shader
    glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0, (GLvoid*)0);
    //Ativa os atributos configurados
    glEnableVertexAttribArray(0);

    //Desliga array de objetos de vertice
    glBindVertexArray(0);

//

// Configura os shaders

    GLint success;
    GLchar infoLog[512];

    //Cria um shader de vertice e guarda sua localizacao em uma variavel
    GLint vertexShader = glCreateShader(GL_VERTEX_SHADER);
    //Copia o código do shader para o shader criado
    glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
    //Compila o shader
    glCompileShader(vertexShader);
    //Passa a informacao sobre o estado de compilacao para uma variavel
```

```
glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &success);
//Verifica se o shader foi compilado com sucesso
if(!success)
{
    glGetShaderInfoLog(vertexShader, 512, NULL, infoLog);
    std::cout << "ERROR::SHADER::VERTEX::COMPILATION_FAILED\n" << infoLog << std::endl;
}
```




```
//Rotina do fragment shader é similar a do Vertex Shader acima
GLint fragmentShader;
fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);
```

```
    glCompileShader(fragmentShader);
    glGetShaderiv(fragmentShader, GL_COMPILE_STATUS, &success);
    if(!success)
    {
        glGetShaderInfoLog(fragmentShader, 512, NULL, infoLog);
        std::cout << "ERROR::SHADER::VERTEX::COMPILATION_FAILED\n" << infoLog <<
std::endl;
    }

    //Cria um programa de shaders e liga a uma variavel
    shaderProgram = glCreateProgram();

    //Liga os shaders criados ao programa de shaders
    glAttachShader(shaderProgram, vertexShader);
    glAttachShader(shaderProgram, fragmentShader);

    //Liga o programa de shader no OpenGL
    glLinkProgram(shaderProgram);

    glGetProgramiv(shaderProgram, GL_LINK_STATUS, &success);
    if(!success){

        glGetProgramInfoLog(shaderProgram,512, NULL, infoLog);
        std::cout << "ERROR::SHADER::VERTEX::COMPILATION_FAILED\n" << infoLog <<
std::endl;

    }
```

```
    //Deleta os shaders criados pois ja estao dentro do programa(trash)
    glDeleteShader(vertexShader);
    glDeleteShader(fragmentShader);

//



}
```

Os comentários estão explicativos, mas nao é nada diferente do que foi passado no tutorial, e agora vem a função de desenho :

```
void Program::render(){
    //Define a cor de limpeza do buffer
    glClearColor(1.0f, 1.0f, 1.f, 1.0f);
    //Limpa o bufer com o tipo de limpeza por cor
    glClear(GL_COLOR_BUFFER_BIT);

    //Informa ao OpenGL que vai usar o programa de shader respsctivo para desenhar
    glUseProgram(shaderProgram);
  //DESENHA TRIANGULO
}
```

Agora sim estamos preparados para desenhar o tão esperado Triângulo. O mesmo será desenhado colocando as funções no espaço comentado no código cima.

O grande início

Agora será gerado o primeiro objeto no OpenGL, com uma arquitetura bem definida, é um processo um tanto quanto simples e rápido. O triângulo ansiosamente desejado, será desenhado usando a função glDrawArrays(), que desenha formas primitivas que foram incluídas no VBO(Indiretamente usando o VAO). (Lembrando que as funções de desenho devem ficar após enviar a informação ao OpenGL sobre qual programa de shaders será usado(glUseProgram())).

```
//Liga o array de objetos de vertice
glBindVertexArray(VAO);
//Desenha as primitivas passadas para o VBO
glDrawArrays(GL_TRIANGLES, 0, 3);
//Desliga o array de objetos de vertice
glBindVertexArray(0);
```

A função de desenho usada glDrawArrays() deve vir sempre depois de ligar o VAO e antes de desligar o VAO. Os parâmetros da função são:

1. Especifica o tipo de primitiva a ser desenhado(no caso um triângulo).
2. Informa a partir de qual ponto será desenhado(no caso a partir do primeiro, 0) .
3. Informa o número de vértices a serem desenhados(no caso 3 vértices).

Agora que a função render está pronta, deve-se se alterar o main.cpp para chamar esse método no loop principal. Primeiramente, abaixo da criação da janela, chame o método setupGLSL():

```
//Window recebe a nossa janela criada pela função que vimos anteriormente.
mainWindow = firstProgram.initOpenGL();

//Configura os shaders e ligacoes com OpenGL
```

```
firstProgram.setupGLSL();
```

Após isso é só modificar o loop principal assim:

```
//Verifica se a janela criada foi fechada a cada iteração.
  while(!glfwWindowShouldClose(mainWindow)){

    //Verifica todos os eventos e chama as funções relacionadas a esses eventos
    glfwPollEvents();
    //Executa funções de desenhar.
    firstProgram.render();

    //Troca o buffer de cor a cada iteração pelo definido.
    glfwSwapBuffers(mainWindow);
  }
```

E VIVA!!! Compile, execute e o resultado deverá ser semelhante a este:

Um triângulo foi impresso na tela, mas ainda há algumas considerações a serem feitas.

Element Buffers Objects

Agora que uma triângulo foi desenhado é possível desenhar um retângulo formado por dois triângulos, utilizando só as configurações que foram feitas anteriormente o array de vértices para desenhar um retângulo se definiria da seguinte maneira:

```
GLfloat vertices[] = {
```

```
    // Primeiro Triângulo
    0.5f, 0.5f,             // Superior Direito
    0.5f, -0.5f,  // Inferior direito
    -0.5f, 0.5f,  // Superior Esquerdo
    // Segundo triangulo
    0.5f, -0.5f,  // Inferior direito
    -0.5f, -0.5f,           //Inferior Esquerdo
    -0.5f, 0.5f   // Superior Esquerdo
};
```

Percebe-se que 2 vértices estão sendo definidos duas vezes, gerando uma sobrecarga de 50%, o que pode ser um grande problema, pois geralmente objetos 3D conté mais de 1000 triangulos, seria uma sobrecarga absurda.Vértices que não se repetem nesse retângulo foram definidos 4.

O Element Buffer Objects(EBO) foi criado para otimizar essa tarefa, ele é um vertex buffer object que armazena os índices dos vértices que formam cada triângulo, podendo ser definido da seguinte maneira:

```
GLfloat vertices[] = {
    0.5f, 0.5f,      // superior direito
    0.5f, -0.5f,   // inferior direito
    -0.5f, -0.5f,  // inferior esquerdo
    -0.5f, 0.5f      // superior esquerdo
};
GLuint indices[] = {  // índices começam com 0
    0, 1, 3,        // Primeiro Triângulo
    1, 2, 3         // Segundo Triângulo
};
```

Nota-se que quando se usa EBO só é preciso definir 4 vértices ao invés de 6. O EBO precisa ser criado para poder usar, o processo é similar ao do VBO, inclusive no mesmo local de código, abaixo da definição do VBO:

```
GLuint EBO;
glGenBuffers(1, &EBO);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices,
GL_STATIC_DRAW);
```

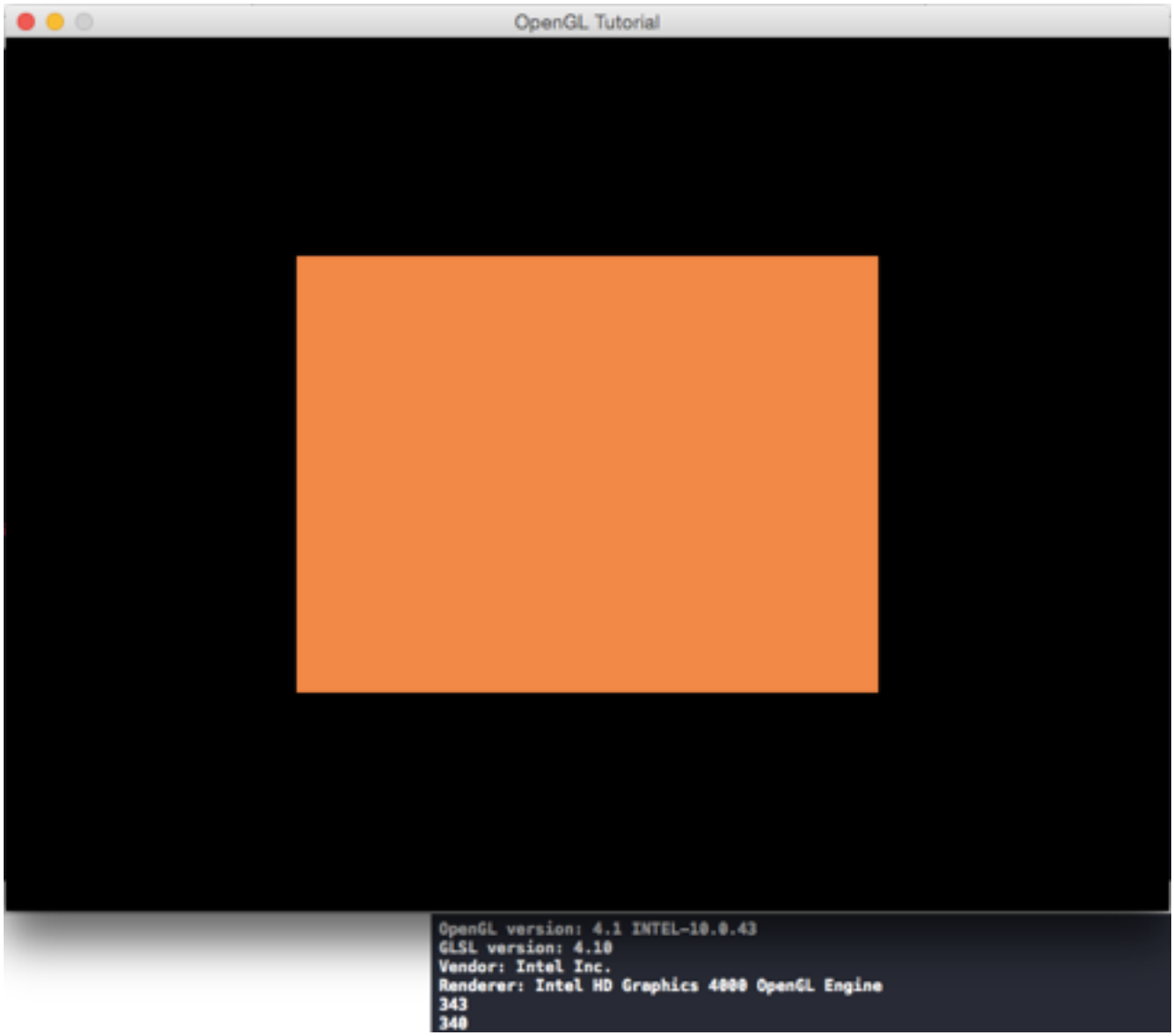
Deve se manter a atenção ao tipo de buffer que está sendo criado, nesse caso GL_ELEMENT_ARRAY_BUFFER. Uma vez definido o EBO, deve-se mudar o modo de desenho de glDrawArrays() para glDrawElements();

```
//Desenha as primitivas passadas para o VBO
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
```

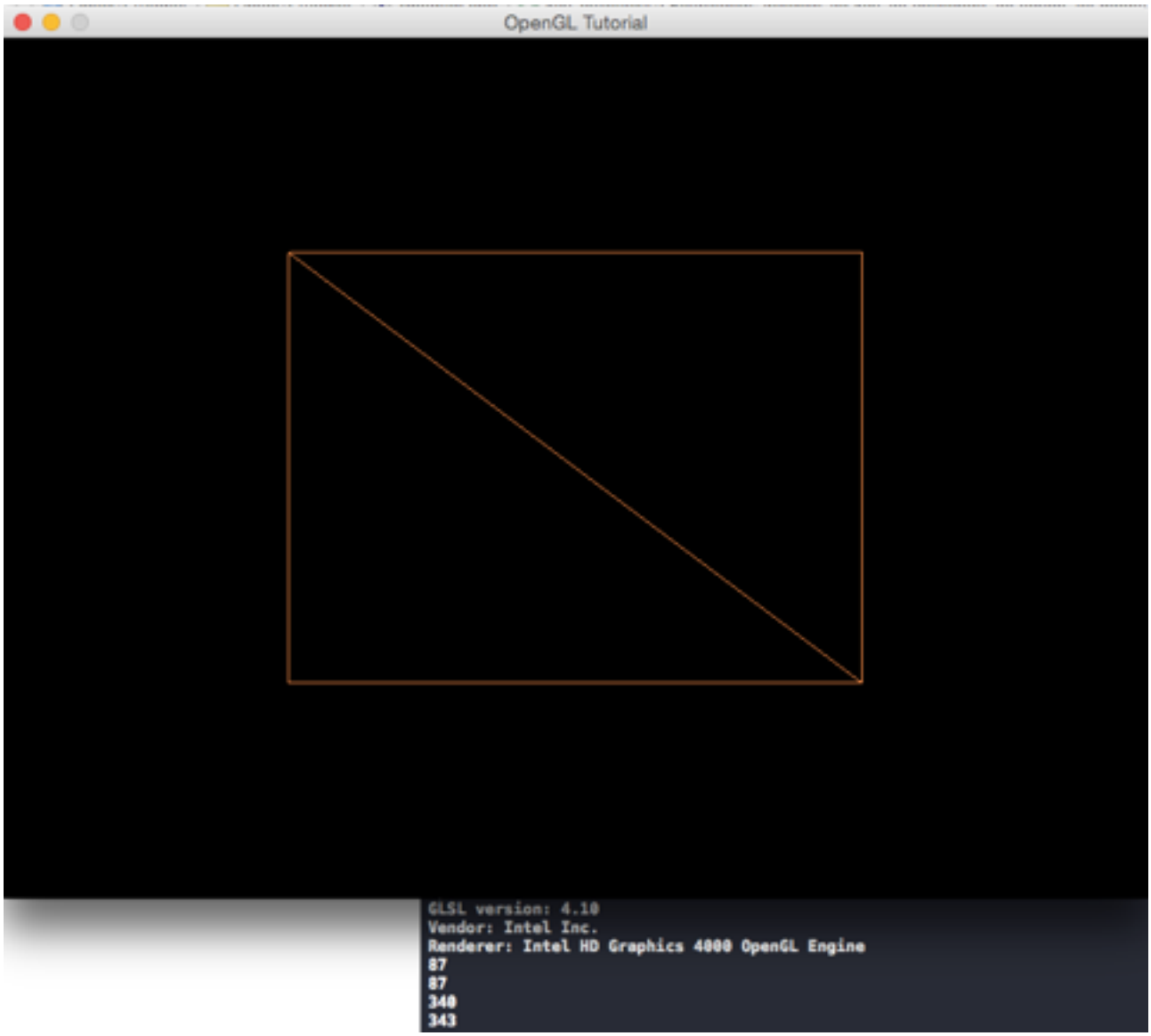
glDrawElements() tem como parâmetros:

1. O tipo de primitiva a ser desenhado (ainda está se desenhando triangulos).
2. A quantidade de índices que serão desenhados.
3. O tipo de valor dos indices.
4. O ultimo elemento deve ser zero. No momento não detalharei sua função.

Substitua o e inclua código nos locais indicados, compile, execute o programa, o resultado deverá ser algo semelhante a isso:



Modo WireFrame , para ativar o modo WireFrame, é só dizer ao OpenGL antes de desenhar algo, que usará o seguinte estado: glPolygonMode (GL_FRONT_AND_BACK, GL_LINE). O primeiro argumento diz que será aplicado parte da frente e verso do



retângulo(é 2D, mas é como uma folha, possui frente e verso, mesmo que com uma expesura de 00000,1mm), o segundo argumento diz para desenhar usando linhas. Para voltar ao normal é só configurar glPolygonMode(GL_FRONT_AND_BACK,GL_FILL), que ele voltará a preencher o triângulo. O modo wireframe gera o seguinte desenho:

Desafio 2

2.1 Desenhe o retângulo usando o método de desenho glDrawArrays. E ao apertar a tecla W ele deve ficar em modo wireframe, ao apertar S voltar ao modo normal.

2.2 Agora crie Dois VBO(e VAO também) e desenhe o primeiro triângulo usando um, e o segundo triângulo usando outro.

2.3 Agora crie um novo programa de shader, que irá imprimir a cor do segundo triângulo em uma cor diferente a sua escolha.

Solucoes no github(TODO)

Mais sobre Shaders

Revendo alguns conceitos de shaders, eles são simplesmente programas isolados que transformam entradas em saídas, ou seja, a única forma de se comunicar com um programa de shader é através de suas entradas e saídas. Shaders são escritos em linguagem GLSL que é bastante parecida com C. Eles são escritos para serem executados na placa gráfica, e conta com recursos principalmente de operações com vetores e matrizes.

O código de um shader, sempre começa com a definição da sua versao, seguido pela declaração dos atributos de entrada e de saída, logo após vem as variáveis uniformes e a função main():

```
#version version_number

in tipo nome_variavel_entrada;

in tipo nome_variavel_entrada;

out tipo nome_variavel_saída;

uniform type nome_variavel_uniforme;

int main()

{

   // Processamento dos dados de entrada

   ...

   // Saída com os dados processados

   nome_variavel_saída = algum_dado_processado;
```

}

Como descrito a função main() , processa os dados de entrada, e a saída são esses dados processados. Não há nenhuma regra para quantidade de variáveis de entrada e saída, podem ser declaradas quantas precisar, e ainda é permitido definir uma localização que identifica essas variáveis.

Em GLSL assim como qualquer outra linguagem de programação tem os tipos básicos como: int, float, uint, bool e double. Porém ele também conta com dois tipos especiais que são : vectors

e matrices.

Vectors

Um vetor em GLSL pode ter de 1 a 4 componentes que podem ser dos tipos descritos acima. Eles são declarados da segunda forma, de acordo com o tipo do conteúdo(N varia de 1 a 4):

bool: bvecN (contém N componentes do tipo booleano).

uint: uvecN (contém N componentes do tipo inteiro sem sinal).

int: ivecN (contem N componentes do tipo inteiro).

float: vecN (contém N componentes do tipo ponto flutuante).

double: dvecN (contém N componentes do tipo double).

Um componente de um vetor pode ser acessado facilmente com um "." um vector vec4 pode ser representado da seguinte maneira: vec4(1.0f, 2.0f, 3.0f, 4.0f), se quiser acessar a primeira componente bastar dizer vec.x , a ultima vec.w e assim por diante.

Outro ponto interessante desse tipo especial do GLSL é que permite agrupar variáveis, por exemplo:

Existe um vec2(com duas componentes float) , para declarar um vec4 com o conteúdo desse vec2, basta dizer vec4(vec2,z,w), assim as componentes x e y do vec2 serão as componentes x e y do vec4.

Também é permitida a seguinte operação com esses vetores:

vec2 meuVetor; //declarando um vec2

vec2 outroVetor = meuVetor.yy // outroVetor vai receber um vetor vec2 com ambas as componentes igual a componente y do meuVetor

O mesmo vale para vec1, vec3 e vec4. Assim nota-se a flexibilidade desse tipo de dados, e como pode ser útil e facilitar muitas operações que serão tratadas mais abaixo.

Entradas e saídas(in, out)

Como foi relatado, shaders são pequenos programas que funcionam por contra própria, então o único meio de se relacionar com o resto da arquitetura, é através de atributos de entrada e saída. Para a declaração desse tipo de variável o GLSL conta com palavras reservadas. Algumas delas são in(para atributos de entrada) e out(para atributos de saída). Apesar de todo shader conter atributos de entrada e saída, a declaração é um pouco diferente entre o vertex shader e o fragment shader.

Um vertex shader, deve ter no mínimo um atributo de entrada para que tenha sentido em usa-lo, um especialidade na declaração desse atributo de entrada no vertex shader é que o mesmo deve possuir uma especificação de layout que configura a localização desse atributo, para que o mesmo possa ser vinculado ao vertex data(os dados de vértice a serem processados). Como ja foi visto anteriormente as palavras reservadas para esse tipo de atribuição é layout(location = N) em que N é um numero inteiro relacionado a localização.

O fragment shader, deve conter no mínimo um atributo de saída de cor do tipo vec4(r,g,b,a) , pois o mesmo retorna a cord que será pintada na tela, se nao tiver uma variável do tipo na saída, irá pintar tudo preto.

Assim para os shaders comunicarem entre si, as variáveis de saída de um devem ser do mesmo tipo de entrada do outro, no caso o fluxo é vertexShader -> fragmentShader, então a saída do vertexShader deve ser do mesmo tipo da entrada do fragmentShader. Para testar isso praticando, o shader do tutorial anterior deve ser alterado para:

Vertex Shader :

```
#version 330 core


layout (location = 0) in vec2 position; // Our position variable has attribute position 0
```

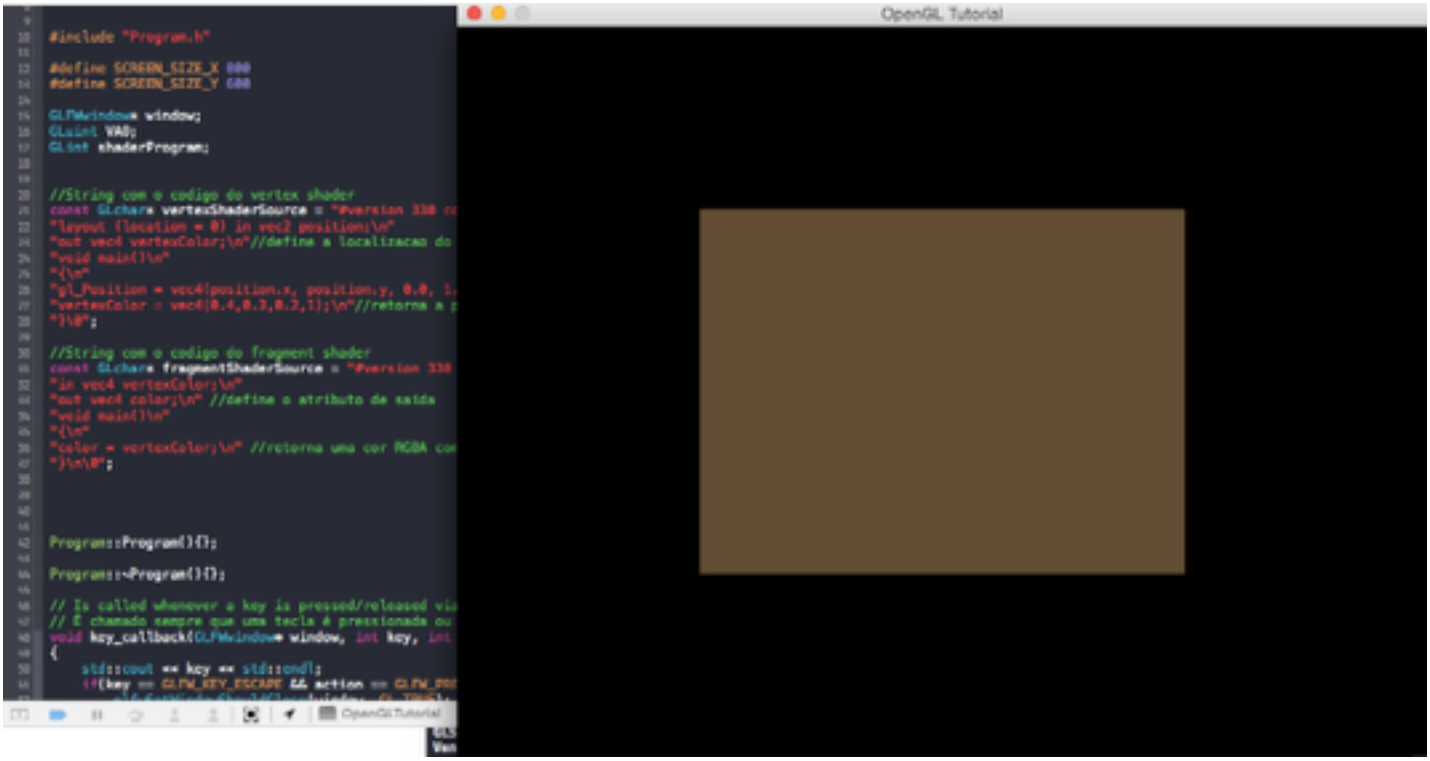
```
out vec4 vertexColor; // Output a color to the fragment shader

void main()
{
   gl_Position = vec4(position.x, position.y, 0.0, 1.0);
   vertexColor = vec4(0.3f, 0.3f, 0.2f, 1.0f); // Set our output variable to a red color
}
```

Fragment Shader:

```
#version 330 core

in vec4 vertexColor; // our input variable from the vertex shader

out vec4 color;

void main()
{
   color = vertexColor;
}
```

Lembre-se de colocar o código na string do seu shader, declarado como constante no cabeçalho.

Percebe-se que foi declarada uma saída do vertex shader, e uma entrada para o fragment shader , com o mesmo tipo e nome, quando isso acontece, as variáveis de ambos os shaders estão ligadas. Compile, execute e terá um resultado similar a este:



Conseguimos enviar uma cor do shader de vértice para o de fragmento, e para enviar uma cor do programa para o shader. Atributos uniform(mais uma palavra reservada GLSL) estão ai casos do tipo.

Uniform

Uniform é uma maneira diferente dos vértices de passar uma variável do programa para os shaders. É diferente porque uma variável do tipo Uniform, é declarada como global, pode ser acessada a qualquer momento pelo shader, e terá sempre o mesmo valor, ao menos que o mesmo seja alterado(como uma máquina de estado). Para declarar um atributo Uniform, basta colocar uniform anteriormente a declaração.

Fragment Shader:

#version 330 core

out vec4 color;

```
uniform vec4 algumaCor; // We set this variable in our OpenGL code.
void main()
{
   color = algumaCor;
}
```

Foi declarado que a cor resultante do fragment shader será igual ao atributo uniform algumaCor, então não precisa mais receber a cor que vem do vertex shader. O fragment shader está usando o atributo algumaCor, mas em nenhum estante foi passado algum valor para o mesmo. Então para ter algum contato com algumas funções novas do GLFW e vamos passar para nosso atributo algumaCor uma cor que vai ser definida com base no tempo que se passou dentro da aplicação.

```
GLfloat timeValue = glfwGetTime();
GLfloat greenValue = (sin(timeValue) / 2) + 0.5;
GLint vertexColorLocation = glGetUniformLocation(shaderProgram, "ourColor");
glUniform4f(vertexColorLocation, 0.0f, greenValue, 0.0f, 1.0f);
```

1. glfwGetTime() - pega o tempo que foi passado desde que o GLFW foi inicializado. Esse tempo será guardado na variável timeValue
2. greenValue está recebendo o seno do tempo passado, dividindo por 2 e somando mais 0.5, isso porque o seno é uma função que varia de -1 a 1(como foi dividido por 2 , variará de -0.5 a 0.5) . Somando com 0.5 assim o nosso valor será de 0 a 1. Este será nosso valor G no vetor RGBA que define a cor de saída. Para usar a função seno deve se incluir a biblioteca <math.h> no cabeçalho.
3. vertexColorLocation, usa a função glGetUniformLocation(Programa de shader onde esta o atributo, “nome do atributo”); que retorna a posição do atributo uniforme presente no programa de shader. Se ele não encontrar irá retornar -1. E para definir a variável uniforme o programa de shader deve estar ativo ou seja, deve ser depois de se usar o glUseProgram().

4. Define o valor do atributo uniforme com a função glUniform4f(localizacao do atributo, valor x, valor y, valor z, valor w). Nota-se que se fosse um vec3 seria glUniform3f(), e assim por diante.

Como citado no início do livro, o OpenGL tem seu coração escrito em Linguagem C, linguagem C nao permite sobrescrever operadores e tipos. Para isso o OpenGL define um função para cada tipo de atributo. A função glUniform é um exemplo disso, a ultima letra da função é relacionado ao tipo de atributos que serão passados

glUniform4f(valores float).

glUniformt4i(valores int).

glUniform4ui(valores unsigned int).

glUniform4fv(valores de vetor de float).

Entao se no exemplo acima poderia ter sido usado também o glUniform4fv() e passado um vetor.

Se substitui o seu fragment shader pelo mencionado com a variável uniforme, escreveu o código que define a variável uniforme no lugar correto, ao complicar o programa, você deverá a sua cor , variando de verde a preto, é uma animação bem interessante.

Vertex Data - Nova definição

Foi demonstrado o quanto é interessante trabalhar com os atributos uniformes, pois é uma ponte para passarmos informações entre os shaders e o nosso programa. Mas e se precisasse definir uma cor para cada vértice? Seria definido uma variável uniforme para cada vértice, isso é de certa forma, bastante trabalhoso, mas com os conceitos estudados até agora sobre shaders existe uma maneira mais prática para isso. Isso será solucionado, definido atributos de cores para cada vértice dentro do array de vértices.

```
//Define vertices que serao desenhados
GLfloat vertices[] = {
```

```
    // Posicoes        // Cores
    0.5f, 0.5f,     1.0f, 0.0f, 0.0f, //vermelho
    0.5f, -0.5f,          0.0f, 1.0f, 0.0f, //verde
   -0.5f, -0.5f,    0.0f, 0.0f, 1.0f, //azul
    -0.5, 0.5,      1.0f, 1.0f, 1.0f //branco
};
```

Mas como os dados do array de vértices mudaram, é necessário configurar o vertex shader para receber esses vértices.

```
#version 330 core

layout (location = 0) in vec2 position; // A variable de posicao fica no l
layout (location = 1) in vec3 color;      // Our color variable has attribute position 1

out vec4 ourColor; // Output a color to the fragment shader

void main()
{
    gl_Position = vec4(position.x, position.y, 0.0, 1.0);
    ourColor = color; // Set ourColor to the input color we got from the vertex data
}
```

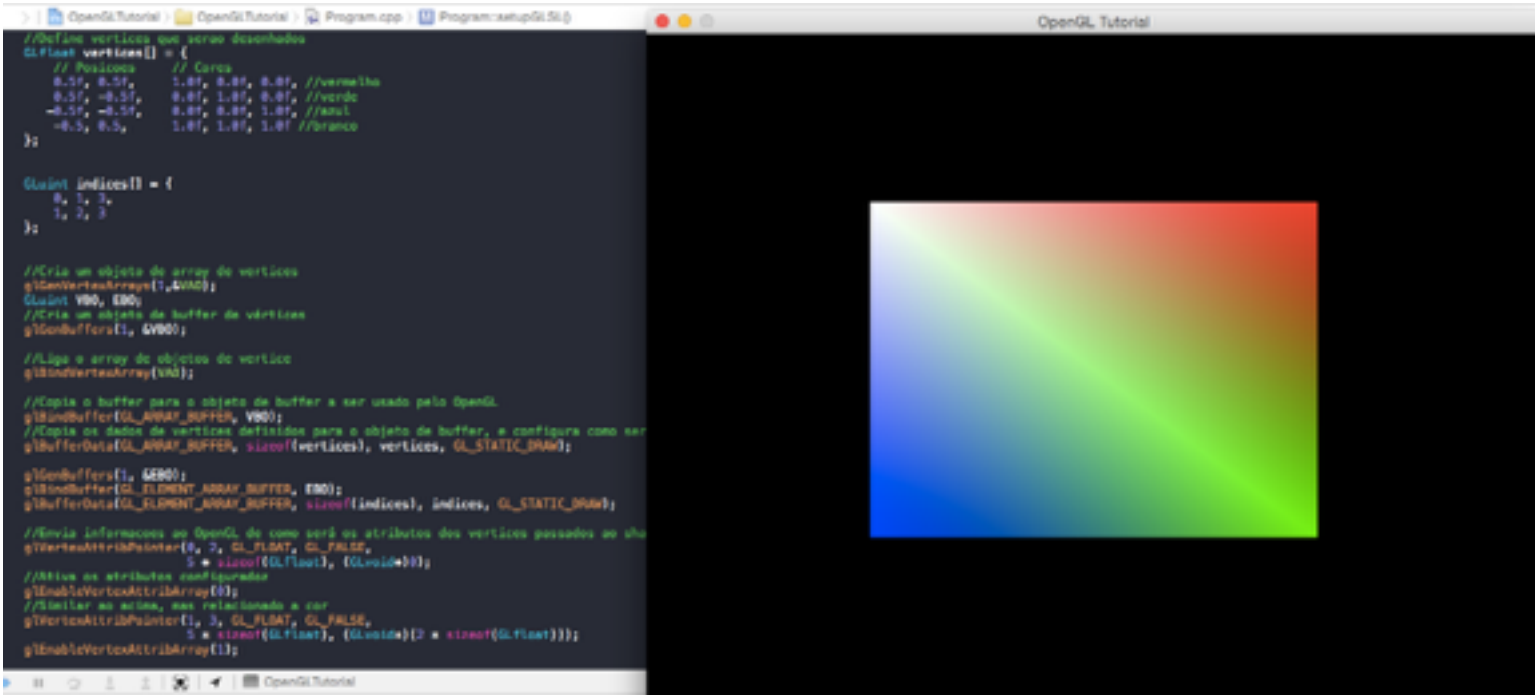
Como os dados do vértice mudaram deve também se alterar o vertexAttribPointer() que referencia os dados de vértice ao shader. Nele que será usado o conceito de localização.

```
//Envia informacoes ao OpenGL de como será os atributos dos vertices passados ao shader
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE,
                5 * sizeof(GLfloat), (GLvoid*)0);
//Ativa os atributos configuradoz
glEnableVertexAttribArray(0);
//Similar ao acima, mas relacionado a cor
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE,
                5 * sizeof(GLfloat), (GLvoid*)(2 * sizeof(GLfloat)));
glEnableVertexAttribArray(1);
```

mais detalhes para esclarecer o glVertexAttribPointer(), os parâmetros são:

1. A posição do attributo, definido no layout(location) dentro do shader.
2. A quantidade de valores que compõem o atributo, no caso 3 , vec3.
3. O tipo de valor que compõe o atributo, GL_FLOAT.
4. Informa se é para os valores serem normalizados, se não GL_FALSE.
5. Informa quantos passos faltam até o próximo atributo, no caso O próximo componente X do vetor está a 5 valores após. Então é para dar um passo de 5 vezes o valor de um float.
6. Informa onde está o primeiro atributo no array de vertices, no caso da posição o primeiro atributo é sempre na posição 0, no caso das cores, começam no terceiro valor, que ficam depois de 2 valores floats(da posição), assim se deito 2*sizeof(float).

Lembrar sempre de ativar os atributos que foram configurados com o glEnableVertexAttribArray(location). Se já tiver alterado os shaders no código antigo, e criado os atributos de cor no array de vértice como visto acima, pode complicar e executar o programa, para obter um excepcional triângulo como este:



Mas fica a dúvida, porque tem uma mistura de cores, se foram definidos apenas 4 cores ? (vermelho, verde, azul e branco). Isso acontece porque o fragmente shader possui uma propriedade chamada Fragment Interpolatiaon, voce nunca se perguntou como nós passamos openas os vertices e o fragment shader preenche todos os pixels no interior da forma geometrica ? Isso que o fragment interpolation faz, na figura acima temos 4 vértices, em pixels mais ou menos 75000 pixels, o fragment shader decide qual será a cor de cada um desses pixels, como temos mais de uma cor de vértice definida, ele mistura as duas cores dividindo a tela quando está chegando na extremidade da cor. Observe a linha do meio que divide o retângulo(se colocar em modo wire frame fica mais fácil) o vertice superior dessa linha, possui a cor branca e a outra extremidade a cor verde, quando a poiscao do fragment se afasta do branco vai ficando cada vez mais verde, e o contrario também.

Prática com orientação a objetos

Daqui pra frente os códigos de shader se tornarão cada vez maiores, assim é interessante salvar esse código em um arquivo separado, ao invés de atribuir ele á um char diretamente, para isso cria dois arquivos fragment.glsl e vertex.glsl (os arquivos podem ter qualquer nome e qualquer extensão após o “.”). Nele coloque o código limpo dos shaders respectivos, sem aquelas aspas e \n.

Bem, para utilizar boas práticas e conceitos de orientação a objetos, vamos criar uma nova classe que ficará responsável em gerar shaders. Essa classe também irá ler o arquivo de shaders e transformar em um array de char que será compilado.
Pessoalmente nomeei essa classe de Shader, no arquivo Shader.h declarei os seguintes métodos e propriedades:

```
#pragma once //Garante que a classe vai ser compilada apenas uma vez, para evitar conflitos se for declarada em mais de um lugar

#include <iostream> //Fluxo de entrada e saída padrao do sistema
#include <fstream> //Manipulador de fluxo arquivos
#include <sstream> //Manipulador de fluxo strings
#include <string> //Suporte a funcoes especiais de string
#include <GL/glew.h>

class Shader{

public:
    Shader();
    ~Shader();

    //Inicializa o shader que foi passado, e cria um associa ao programa da classe
    void initShader(const char* vertexPath, const char* fragmentPath);

    //Programa de shader da classe
    GLuint shaderProgram;

    //Usa o programa de shader da classe
```

```
    void use();
private:



};
```

Agora vamos escrever os métodos dessa classe no Shader.cpp, esta tudo bem explicado no comentário.

```
#include "Shader.h"

Shader::Shader(){};

Shader::~Shader(){};

void Shader::initShader(const char* vertexPath, const char* fragmentPath){

    // Irá guarda o código do shader de um arquivo
    std::string vertexShaderSource;
    std::string fragmentShaderSource;
    try
    {
        // Abre os arquivos
        std::ifstream vShaderFile(vertexPath);
        //verifica se foi aberto
```

```
if(!vShaderFile.is_open()){
   throw std::runtime_error("");
}



std::ifstream fShaderFile(fragmentPath);

if(!fShaderFile.is_open()){
   throw std::runtime_error("");
}

std::stringstream vShaderStream, fShaderStream;

// Le o conteudo do arquivo e salva nas variáveis string
vShaderStream << vShaderFile.rdbuf();
fShaderStream << fShaderFile.rdbuf();



// fecha o arquivo
vShaderFile.close();
fShaderFile.close();

// Converte a strint para o tipo const char, para ser lido pelo shader.
vertexShaderSource = vShaderStream.str();
fragmentShaderSource = fShaderStream.str();
```

```
}
catch(std::exception e)
{
    std::cout << "ERROR::SHADER::FILE_NOT_SUCCESFULLY_READ" << std::endl;
}

//guarda a string em uma array de chars do OpenGL
const GLchar* vertexShaderCode = vertexShaderSource.c_str();
const GLchar* fragmentShaderCode = fragmentShaderSource.c_str();

// Configura os shaders
GLint success;
GLchar infoLog[512];

//Cria um shader de vertice e guarda sua localizacao em uma variavel
GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
//Copia o código do shader para o shader criado
glShaderSource(vertexShader, 1, &vertexShaderCode, NULL);
//Compila o shader
glCompileShader(vertexShader);
//Passa a informacao sobre o estado de compilacao para uma variavel
glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &success);
//Verifica se o shader foi compilado com sucesso
if(!success)
{
    glGetShaderInfoLog(vertexShader, 512, NULL, infoLog);
```

```
    std::cout << "ERROR::SHADER::VERTEX::COMPILATION_FAILED\n" << infoLog <<
std::endl;
  }



  //Rotina do fragment shader é similar a do Vertex Shader acima
  GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
  glShaderSource(fragmentShader, 1, &fragmentShaderCode, NULL);
  glCompileShader(fragmentShader);
  glGetShaderiv(fragmentShader, GL_COMPILE_STATUS, &success);
  if(!success)
  {
    glGetShaderInfoLog(fragmentShader, 512, NULL, infoLog);
    std::cout << "ERROR::SHADER::VERTEX::COMPILATION_FAILED\n" << infoLog <<
std::endl;
  }



  //Cria um programa de shaders e liga a uma variavel
  this->shaderProgram = glCreateProgram();
  //Liga os shaders criados ao programa de shaders
  glAttachShader(this->shaderProgram, vertexShader);
  glAttachShader(this->shaderProgram, fragmentShader);
  //Liga o programa de shader no OpenGL
  glLinkProgram(this->shaderProgram);
  glGetProgramiv(this->shaderProgram, GL_LINK_STATUS, &success);
  if(!success){
```

```
        glGetProgramInfoLog(this->shaderProgram,512, NULL, infoLog);
        std::cout << "ERROR::SHADER::VERTEX::COMPILATION_FAILED\n" << infoLog << std::endl;

    }

    //Deleta os shaders criados pois ja estao dentro do programa(trash)
    glDeleteShader(vertexShader);
    glDeleteShader(fragmentShader);

    //
}

void Shader::use(){

    //Configura o opengl para usar esse programa de shaders.
    glUseProgram(this->shaderProgram);

}
```

No método void Shader::initShader(const char* vertexPath, const char* fragmentPath), ele recebe o diretório de onde se encontra o arquivo de shader criado, como está comentado no código, logo no início do método ele realiza as operações com o arquivo e passa seu conteúdo para um array de chars.

Após escrever esses métodos, é só apagar no Program.cpp o código respectivo que esses métodos implementam, que é de, abrir o arquivo de shader, compilar, criar

um programa de shader, e também, podem ser deletados os chars que foram declarados no cabeçalho com o código do shader, pois o mesmo já está em um arquivo externo. Após isso, importe a classe no seu Program.h.

```
#include "Shader.h"
```

E no Program.cpp, declare no cabeçalho uma variável do tipo Shader instancie uma classe de shader, e crie os shaders que foram usados no tutorial anterior. Isso ficaria após efetuar as configurações com o VAO, no setupGLSL.

```
//Cabecalho

Shader newShader;

//…..

//Antes do fim do método setupGLSL

const char* vertexPath = "/Users/thiagoTMB/Documents/Repositories/Tutorials/
OpenGLTutorial/OpenGLTutorial/OpenGLTutorial/vertex.glsl";

   const char* fragmentPath = "/Users/thiagoTMB/Documents/Repositories/Tutorials/
OpenGLTutorial/OpenGLTutorial/OpenGLTutorial/fragment.glsl";

   newShader = Shader();

   newShader.initShader(vertexPath, fragmentPath);

//…
```

Agora quando for renderizar no render(). Basta informar ese programa de shader criado.

```
//dentrodo método render()

GLint vertexColorLocation = glGetUniformLocation(newShader.shaderProgram,
"xOffset");

// …
```

E antes de desenhar lige o programa de shaders.

```
//Informa ao OpenGL que vai usar o programa de shader respsctivo para desenhar
    newShader.use();

    //Liga o array de objetos de vertice
    glBindVertexArray(VAO);
    //Desenha as primitivas passadas para o VBO
    glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);

    //Desliga o array de objetos de vertice
    glBindVertexArray(0);
```

Se tudo estiver certo, irá obter o mesmo resultado anterior, só que agora o código está com um formato bem mais limpo e prático, e sempre que quisermos criar um novo programa de shader, basta criar os novos shaders, e fazer uma nova instancia da classe shader , passando esses shaders para o método initShaders().

Desafio 3

1. Modifique o vertex shader para que o retangulo seja apresentado de cabeça para baixo(o retângulo esta com o vértice superior direito na cor vermelha no canto superior direito, se ele ficar de cabeça para baixo esse vértice devera ficar no canto inferior direito.) é muito simples.
2. Crie uma variável que vai especificar um deslocamento em X do retangulo, esse deslocamento será passado como um uniform, e o retângulo deverá ser desenhado com esse deslocamento que declarado.
3. Crie um animação gameificada com o retângulo. Ao usuário apertar a tecla direita o triângulo deverá andar para direita, ao apertar a tecla esquerda ele deverá andar para esquerda.

4. Agora é uma animação com algumas condições, o triângulo deverá se mover constantemente sozinho a cada iteração do programa, mas tem uma regra, ele não pode sair da tela, ao chegar no final da tela deverá começar a andar para o lado contrário.