2D game in OpenGL

OpenGL, you may have heard about it, something to do with graphics? If you feel so you are right. It’s a graphics API and as its name says its open and all graphics cards supports it. OpenGL is quite complicated as compared to graphics.h lib that C++ provides, if you have used it. Especially the modern OpenGL which provides tons of features and why it’s so powerful. Today I would try to work with two of my favourite things in programming, C++ and OpenGL and hopefully inspire you to try this awesome API and build something cool.

About OpenGL

OpenGl is by itself a state machine a collection of variables that define how OpenGL should currently operate. the state of OpenGL is commonly referred to as the OpenGL context. we change state by setting some options manipulating some buffers and then using the current context.

OpenGLs core is written in C and is developed with many abstractions in mind. One of them is objects. OpenGL object is a collection of options that represents a subset of OpenGL’s state. It could be visualized like following.

struct object\_name {

float option1;

int option2;

char[] name;

};

The state of OpenGL can be visualized as structure contains lots of object pointers as follows

struct OpenGL\_Context {

...

object\_name\* object\_Window\_Target;

...

};

to create object, we would use object gen function and store the object id.

then we bind the object and set properties

// create object

unsigned int objectId = 0;

glGenObject(1, &objectId);

// bind object to context

glBindObject(GL\_WINDOW\_TARGET, objectId);

// set options of object currently bound to GL\_WINDOW\_TARGET

glSetObjectOption(GL\_WINDOW\_TARGET, GL\_OPTION\_WINDOW\_WIDTH, 800);

glSetObjectOption(GL\_WINDOW\_TARGET, GL\_OPTION\_WINDOW\_HEIGHT, 600);

// set context target back to default

glBindObject(GL\_WINDOW\_TARGET, 0);

Libraries to simplify tasks

To get started we need OpenGL context and an application window. Drawing a window is operating system specific work. We need to some way to handle creating window defining context and handling input. There are many popular libraries like GLUT, SDL, SFML and GLFW. We are going to use GLFW since its newer library.

lib installation

you can download it from their webpage on https://www.glfw.org/download.html there are precompiled binaries and header files for visual studio for windows. but since I am on Linux, I will compile using Cmake.

Once you download source package extract it and open its contents in terminal. create a folder called build and enter the folder using `mkdir build; cd build` using Cmake generate the solution since make file is in parent folder we run `cmake ..` and run `cmake install` to install the files

for more on how to using Cmake GUI or if you are using IDE visit this page https://learnopengl.com/Getting-started/Creating-a-window

linking

we are going to use g++ compiler. to link libraries we add flags to g++ command -l followed by library name like `g++ filename.cpp -lglfw -lGL -lGLEW -lglm` for linking glfw, GL and GLEW

Since there are many different drivers that specific graphic card supports we need we need something that can retrieve the location of functions. we will use glew to do this for us. Download at http://glew.sourceforge.net/

we will later use glm for handling our matrix calculation Download at https://glm.g-truc.net/0.9.9/index.html

window

#include<GL/glew.h>

#include<GLFW/glfw3.h>

#include<iostream>

int main(){

create variable for window and initialize glfw

GLFWwindow\* window;

if(!glfwInit())

return -1;

To draw window we need to are going to use OpenGL core profile by setting

glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MAJOR, 3);

glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MINOR, 3);

glfwWindowHint(GLFW\_OPENGL\_PROFILE, GLFW\_OPENGL\_CORE\_PROFILE);

then we are going to create window and check if its created else will close the application.

window = glfwCreateWindow(640,640, "snake game",NULL, NULL);

if(!window){

glfwTerminate();

return -1;

}

we are going to set window we just created as our context

glfwMakeContextCurrent(window);

//set other window properties

//...

to print opengl version loaded

std::cout<<glGetString(GL\_VERSION)<<std::endl;

if(glewInit() != GLEW\_OK)

std::cout<<"GLEW Error!"<<std::endl;

to keep window alive till closed we run a while loop and in we would run all update code that need to be run every new draw call like

{

while(!glfwWindowShouldClose(window)){

//..Rendering calls

glfwSwapBuffers(window);

glfwPollEvents();

}

}

dont forget to destroy what you created

glfwTerminate();

return 0;

}

Input

to handle input we are going to set callback funciton using

glfwSetKeyCallback(window, key\_callback);

we need to define call back function

void key\_callback(GLFWwindow\* window, int key,int scancode, int action, int mods){

//all input handling following checks if you pressed if you pressed esc key if so it will close window

if(key == GLFW\_KEY\_ESCAPE && action==GLFW\_PRESS)

glfwSetWindowShouldClose(window,true);

}

you can also run it in while loop by as

if(glfwGetKey(window, GLFW\_KEY\_ESCAPE) == GLFW\_PRESS)

glfwSetWindowShouldClose(window, true);

Debugging

before we move to rendering, we are going to set way to debug our code

#define ASSERT(x) if(!(x)) std::cerr<<"Error ";

#define GLCall(x) GLClearError(); x;ASSERT(GLLogCall(#x, \_\_FILE\_\_, \_\_LINE\_\_))

void GLClearError(){

while(glGetError() != GL\_NO\_ERROR);

}

bool GLLogCall(const char\* function, const char\* file, int line){

while(GLenum error = glGetError()){

std::cout<<"[OpenGL Error] (0x"<<std::hex <<error<<"): "

<<function<<" "<<file<<":"<< line <<std::endl;

return false;

}

return true;

}

I learned from that simple debugging setup can reduce lot of hea

encapsulate all gl call with GLCall to get error messages

this checks for runtime errors after each gl command and prints line number for more refer YouTube video Dealing with Errors in OpenGL from OpenGL playlist by TheChernoProject at https://youtu.be/FBbPWSOQ0-w

Shader

In OpenGL everything is in 3D space. but on screen and window its 2D. OpenGL’s graphics pipeline hence takes 3D coordinates as input, and transforms these into 2D pixels. This task is divided into several steps. each step is highly specialized. can be executed in parallel. this is done using small programs

called shaders. shaders are written in OpenGL Shading Language (GLSL).

vertex data is collection of 3D point data, represented using vertex attributes that can contain any data like position or colour value.

vertex shader takes input as 3D coordinates and transforms into different 3D coordinates. vertex shaders allow us to do some basic processing on vertex attributes.

geometry shader takes collection of vertices and generates shapes as primitives.

fragment shader calculates the final colour of a pixel and this is usually the advanced OpenGL effects occur. it can calculate final pixel value. In most cases we need to build only vertex and fragment shader. For others let other shaders default implementations will be used.

for a vertex we need x, y, z coordinates. but since we are treating z as 0. we need to send this data to vertex shader. to do that we need to create memory on GPU to store data. And configure how OpenGL should interpreted the data. we manage the data using vertex buffer objects that can store large number of vertices in GPUs memory. sending data to GPU is slow hence we try to bundle as much data as possible and send it all at once.

float positions[] ={

//positions

1.0f, 1.0f, //top right

1.0f, -1.0f, //bottom right

-1.0f, -1.0f, //bottom left

-1.0f, 1.0f //top left

};

we create vertex buffer bind it and assign data.

unsigned int buffer;

GLCall(glGenBuffers(1, &buffer));

GLCall(glBindBuffer(GL\_ARRAY\_BUFFER, buffer));

glBufferData takes in mode, size in bytes since here we are storing 4 vertices each with 2 coordinate values of size of float. pointer to data and mode to tell usage pattern.

GL\_STATIC\_DRAW: the data will most likely not change at all or very rarely.

GL\_DYNAMIC\_DRAW: the data is likely to change a lot.

GL\_STREAM\_DRAW: the data will change every time it is drawn.

GLCall(glBufferData(GL\_ARRAY\_BUFFER, 4\*2\*sizeof(float), positions, GL\_STATIC\_DRAW));

to draw a square, we need to draw 2 triangles. since there will be repetition of vertices, we use indexing. to save some memory.

Element Buffer Objects

unsigned int indices[]={

0, 1, 2,

2, 3, 0

};

unsigned int ibo;

GLCall(glGenBuffers(1, &ibo));

GLCall(glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, ibo));

mode we set here is (GL\_ELEMENT\_ARRAY\_BUFFER since its for indexing

GLCall(glBufferData(GL\_ELEMENT\_ARRAY\_BUFFER, 6\*sizeof(unsigned int), indices,

GL\_STATIC\_DRAW));

modern OpenGL requires that we at least set up a vertex and fragment shader. we need to write this shader in GLSL (OpenGL Shading Language). GLSL looks very similar to c.

vertex shader

#version 330 core

layout(location = 0) in vec4 position;

uniform mat4 u\_MVP;

void main(){

gl\_Position = u\_MVP \* position;

};

shader begins with declaration of its version. we declare all the input vertex attributes in vertex shader with in keyword and output is specified using out keyword. GLSL has vector datatype vector of 4 values for position. vector of 3 values can represent position/direction in any space as x, y, z, 4th value w is used for perspective division. since we are developing for 2d we will let z and w initialize to 0.

we are specifically setting the location of input variable via location=0. when input and output of two shaders match, they are passed along. vertex shader gets its input directly from vertex data hence we need to specify extra location metadata as layout (location = 0). we can also skip this and query for its location using glGetAttribLocation but we will stick to setting location metadata.

uniforms are way of passing data from our application on the CPU to the shader on GPU. Uniforms are global that is they are unique per shader program object and can be accessed from any shader at any stage in shader program. After setting value it keeps their value until they're either reset or updated.

In main code to set uniform value you need to get the location of uniform

u\_mvp\_location = glGetUniformLocation(shader, "u\_MVP");

After then you can set values as follows

glUniformMatrix4fv(u\_mvp\_location, 1, GL\_FALSE, &mvp[0][0]);

There are different functions to pass different types of values. glUniformMatrix4fv is used to send matrix of 4 by 4 while glUniform4f is used to send vector of 4 value.

gl\_position is internal variable that is input for next shader in line in graphics pipeline of OpenGL.

and fragment shader

#version 330 core

uniform vec4 u\_color;

out vec4 color;

void main(){

//color = vec4(0.0, 0.0, 1.0, 1.0);

color = u\_color;

};

similarly, it applies for fragment shader colour needs to be outputted for next shader in line. since colour is output of fragment shader, we specify it using out keyword. output of fragment shader is sent to next step in graphics pipeline and finally it would be displayed on the screen.

we then create a shader object and compile the source. And then attach and direct the program to use it. We can delete shaders after linking them.

unsigned int compileShader(unsigned int shader,const char\* src,int shaderType){

unsigned int shaderId = glCreateShader(shaderType);

GLCall(glShaderSource(shaderId, 1, &src, nullptr));

GLCall(glCompileShader(shaderId));

int result;

glGetShaderiv(shaderId, GL\_COMPILE\_STATUS, &result);

if(!result){

int length;

glGetShaderiv(shaderId, GL\_INFO\_LOG\_LENGTH,&length);

char\* message = (char\*)alloca(length\* sizeof(char));

glGetShaderInfoLog(shaderId, length, &length, message);

std::cout<<"Failed to Compile";

if(shaderType==GL\_VERTEX\_SHADER) std::cout<<" GL\_VERTEX\_SHADER ";

else if(shaderType==GL\_FRAGMENT\_SHADER) std::cout<<" GL\_FRAGMENT\_SHADER ";

std::cout<<std::endl;

std::cout<<message<<std::endl;

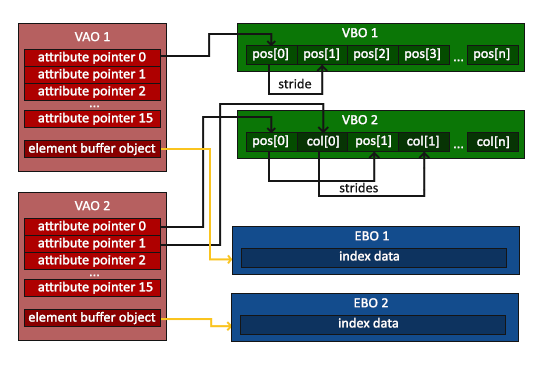
}

glAttachShader(shader,shaderId);

return shaderId;

}

Vertex array object



Vertex array object (VAO) van be bound just like a vertex buffer. This way we can store many pointers inside a VAO. We then have to configure vertex attribute pointers only once and then when we need to draw, we just bind the corresponding VAO. This makes switching between different vertex data and attribute configurations very easy.

unsigned int vao;

GLCall(glGenVertexArrays(1, &vao));

GLCall(glBindVertexArray(vao));

After all configuration only what remains is making draw call

glDrawElements takes mode of drawing, count of indices, type of input and indices if ebo is not used

glDrawElements(GL\_TRIANGLES, 6, GL\_UNSIGNED\_INT, nullptr);

Math

To draw blocks at different location we need to do some math. I previously mentioned about glm library that handles all matrix calculations of my project. to calculate the position of a model we do model view projection. vertex we use Model, View and Projection matrices and multiply them to get final vertex.

Using all of the above knowledge I build my very own snake game. I give random coordinates to fruit. for snake I start at centre. I used queue to keep track of snake’s body. I pop the queue every time it exceeds the tail length. wrote a keyboard call-back function to handle all input. Next step is to add texture instead of plain colours and loading text on screen. For loading text, you basically have to load character bitmaps of each letter as a texture. And later refactoring code to nicely organised object-oriented code. And there you have it how I made game using OpenGL.

find my source code in my git hub repository at https://github.com/intelligentchild/2d-snake-game-opengl-cpp

References

In-depth basics, intermediate and all the advanced knowledge using modern (core-profile) OpenGL tutorials https://learnopengl.com/

OpenGL and Game Engine Development Tutorials https://www.youtube.com/user/TheChernoProject