2D game in OpenGL

OpenGL, is a graphics API. It is like cpp standards. All graphic card vendors have their own implementation. Though the name says open its source is not freely available. In this article I will try to give you glimpse of how to build 2D game using OpenGL in cpp.

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. To create object, we would use object generation function and store the object id. Then we bind the object and set properties.

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.

library 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. On Linux, compile and install using Cmake. 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

Since there are many different drivers that specific graphic card supports, 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/

Also download glm for handling our matrix calculation Download at

https://glm.g-truc.net/0.9.9/index.html

Window

Create variable for window and initialize glfw

```
GLFWwindow* window;
if(!glfwInit())
      return -1;
```

set it up to use OpenGL core profile

```
glfwWindowHint(GLFW_CONTEXT_VER
SION_MAJOR, 3);
glfwWindowHint(GLFW_CONTEXT_VER
SION_MINOR, 3);
glfwWindowHint(GLFW_OPENGL_PROF
ILE, 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;
}
```

Set the window as current context

```
glfwMakeContextCurrent(window);
//set other window properties
//...
```

to print loaded OpenGL version

```
std::cout<<glGetString(GL_VERSI
ON)<<std::endl;
```

initialize glew

```
if(glewInit() != GLEW_OK)
std::cout<<"GLEW
error!"<<std::endl;
```

to keep window alive till terminate flag is turned on we run a while loop and in which would run all update code that need to be run for every draw call

```
while(!glfwWindowShouldClose(wi
ndow)){
```

```
        //..Rendering calls
        glfwSwapBuffers(window);
        glfwPollEvents();
}
```

don't forget to destroy glfw context

```
glfwTerminate();
```

Input

to handle input, we are going to set call-back function using

```
glfwSetKeyCallback(window,
key_callback);
```

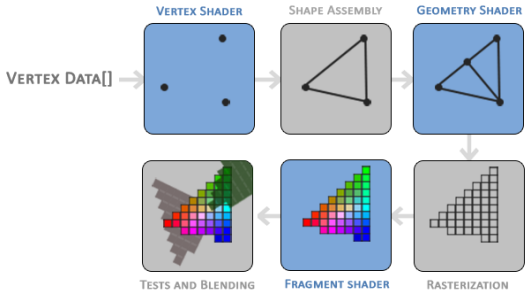
call back function looks something like this

```
void key_callback(GLFWwindow*
window, int key,int scancode,
int action, int mods){
// if you pressed if you pressed
// esc key this will set close
// window flag
if(key == GLFW_KEY_ESCAPE &&
            action==GLFW_PRESS)
glfwSetWindowShouldClose(window
,true);
}
```

you can also handle input in while loop

```
if(glfwGetKey(window,
GLFW_KEY_ESCAPE) == GLFW_PRESS)
glfwSetWindowShouldClose(window
, true);
```

Shader



In OpenGL everything is in 3D space, but it is displayed on window as 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 shader. 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. This is input to vertex Shader. Vertex shader takes input this 3D coordinates and transforms into different 3D coordinates. Fragment shader calculates the final colour of a pixel. Usually the advanced OpenGL effects occur here. Modern OpenGL requires that we at least set up a vertex and fragment shader.

Vertex Buffer Objects

For a vertex we need to send x, y coordinates. to vertex shader. To do that we need to create memory on GPU to store data. And configure how OpenGL should interpret 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;
glGenBuffers(1, &buffer);
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.

```
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;
glGenBuffers(1, &ibo);
glBindBuffer(
GL_ELEMENT_ARRAY_BUFFER, ibo);
```

mode we set here is `GL_ELEMENT_ARRAY_BUFFER` since its for indexing

```
glBufferData(
GL_ELEMENT_ARRAY_BUFFER,
6 *sizeof(unsigned int),
indices, GL_STATIC_DRAW);
```

Vertex Shader

```
#version 330 core
layout(location = 0) in vec4
position;
uniform mat4 u_MVP;
void main(){
      gl_Position = u_MVP *
position;
};
```

Fragment Shader

```
#version 330 core
uniform vec4 u_color;
out vec4 color;
void main(){
           color = u_color;
};
```

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.

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 as its easier.

Uniforms

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. You do that by querying for uniform name.

```
u_mvp_location =
glGetUniformLocation(shader,
"u_MVP");
```

After then you can set values as follows

```
glUniformMatrix4fv(u_mvp_locati
on, 1, GL_FALSE, &mvp[0][0]);
```

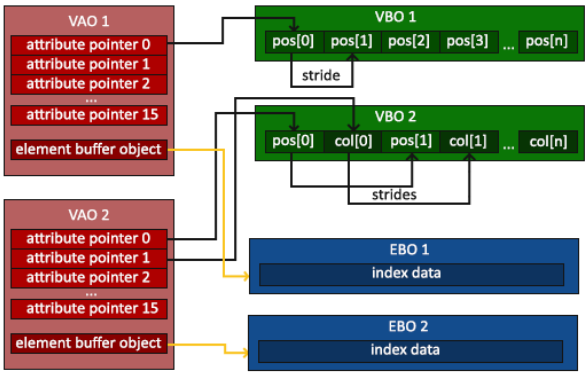
There are different functions to pass different types of values. `glUniformMatrix4fv` is used to send matrix of 4 by 4 while `glUniform4f` is used to send vector of 4 value.
gl_position pre-set variable which is input for next shader in line in graphics pipeline of OpenGL. Similarly, it applies for fragment shader colour needs to be outputted for next shader in line. color 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 shaderId =
glCreateShader(shaderType);
glShaderSource(shaderId, 1,
&src, nullptr);
glCompileShader(shaderId);
int result;
```

```
glGetShaderiv(shaderId,
GL_COMPILE_STATUS, &result);
if(!result){
      std::cout<<"Failed to
Compile"<<std::endl;
}
glAttachShader(shader,shaderId)
;
```

shader type is which shader you want to compile `GL_VERTEX_SHADER` or `GL_FRAGMENT_SHADER` in our case.

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;
glGenVertexArrays(1, &vao);
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);
```

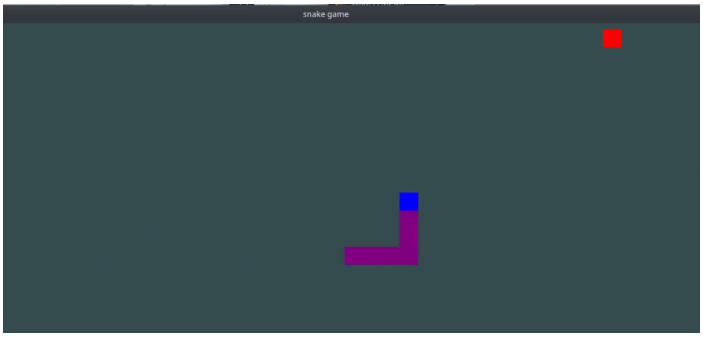
Math

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. Model represents local coordinates, after multiplying with view matrix we get world coordinates and finally projection matrix gives us screen coordinates

Snake Game

Using all of the above knowledge I build my very own snake game. I wrote a function to generate random coordinates for 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. Block drawing function then draws fruit and snake on screen. 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 a snake game using OpenGL.

Find full source code in my GitHub 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