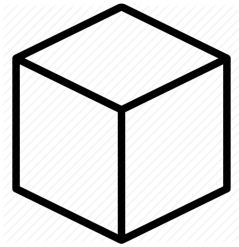


Using the bgfx library with C++ on Ubuntu

THURSDAY. MAY 16, 2019 - 25 MINS

BGFX C++ 3D RENDERER OPENGL

Bgfx - what is it?

> Cross-platform, graphics API agnostic, “Bring Your Own Engine/Framework” style rendering library, licensed under permissive BSD-2 clause open source license.

bgfx is a rendering library. It acts as an abstraction layer over the platform rendering APIs along with some sauce for added efficiency. It also has bindings to most popular languages including C#, F#, D, Go, Haskell, Python, Rust and Swift. I don’t claim to have built anything substantial with

bgfx, but I did manage to figure out how to get a square on screen. The documentation and examples are thorough but they do not hold your hand through the process. So let me.

> Disclaimer: This tutorial is a result of initial exploration with bgfx and getting a square on screen. I do not claim to have the best makefiles, memory safe code or efficient methods. Those topics are for future tutorials.

Prior Knowledge

bgfx is quite close to the metal innards of 3d rendering, being just a thin wrapper. So I would highly recommend going through this OpenGL Tutorial to get comfortable with how modern 3d rendering works. It would really help in making sense of what bgfx is doing at any moment.

Project setup

bgfx only handles the rendering, which means we need to use a windowing library (the brave can use the platform code directly, be my guest). I chose SDL. Ensure your system has SDL dev libraries installed along with bgfx dependencies on Ubuntu.

```
sudo apt-get install libsdl2-2.0 libgl1-mesa-dev x11proto-core-dev libx11-dev
```

Ok, for the sake of this tutorial, lets make a simple C++ project, starting with a Makefile that links up SDL2, GL, X11, DL, pthread and rt libraries.

```
# CC specifies which compiler we're using
CC = g++

# COMPILER_FLAGS specifies the additional compilation options we're using
# -w suppresses all warnings
COMPILER_FLAGS = -w

# LINKER_FLAGS specifies the libraries we're linking against
LINKER_FLAGS = -lSDL2 -lGL -lX11 -ldl -lpthread -lrt

# This is the target that compiles our executable
all : main.cpp
    $(CC) main.cpp -o  main $(COMPILER_FLAGS) $(LINKER_FLAGS)
```

And lets add a simple `main.cpp` file as well.

```
int main ( int argc, char* args[] ) {
  return 0;
}
```

You should try and run make and then run the program to see if everything is setup correctly.

```
make all && ./main
```

Getting a window on screen

As mentioned earlier, we would be using SDL to create a window and poll for events. The code for that is fairly straight forward. If you need to understand more about setting up SDL, I recommend going through LazyFoo's SDL tutorial series.

```
#include <stdio.h>
#include <SDL2/SDL.h>
#include <SDL2/SDL_syswm.h>

SDL_Window* window = NULL;
const int WIDTH = 640;
const int HEIGHT = 480;
int main (int argc, char* args[]) {

  // Initialize SDL systems
  if(SDL_Init( SDL_INIT_VIDEO ) < 0) {
    printf("SDL could not initialize! SDL_Error: %s\n",
           SDL_GetError());
  }
  else {
    //Create a window
     window = SDL_CreateWindow("BGFX Tutorial",
                               SDL_WINDOWPOS_UNDEFINED,
                               SDL_WINDOWPOS_UNDEFINED,
                               WIDTH, HEIGHT,
                               SDL_WINDOW_SHOWN);
    if(window == NULL) {
      printf("Window could not be created! SDL_Error: %s\n",
             SDL_GetError());
    }
  }


  // Poll for events and wait till user closes window
```

```
  bool quit = false;
  SDL_Event currentEvent;
  while(!quit) {
    while(SDL_PollEvent(&currentEvent) != 0) {
      if(currentEvent.type == SDL_QUIT) {
        quit = true;
      }
    }
  }

  // Free up window
  SDL_DestroyWindow(window);
  // Shutdown SDL
  SDL_Quit();

  return 0;
}
```

Compiling and running the program at this point should get you something like the following image.

Getting and building BGFX

First clone the bgfx repository and its dependencies bx and bimg libraries into the project.

```
git clone git://github.com/bkaradzic/bx.git
git clone git://github.com/bkaradzic/bimg.git
git clone git://github.com/bkaradzic/bgfx.git
```

Then lets build bgfx.

```
cd bgfx
make linux-release64
```

If you following along on anything other than an Ubuntu system, you can find instructions to build bgfx here.

At the end of this process, you'll have a bunch of library files and executables sitting in the `bgfx/.build/linux64_gcc/bin` folder. Let us link them in the makefile.

```
BGFX_HEADERS =  -Ibgfx/include -Ibx/include -Ibimg/include

# Update linker flags to include the shared library that you just built
LINKER_FLAGS = bgfx/.build/linux64_gcc/bin/libbgfx-shared-libRelease.so -lSDL2 -lGL

#This is the target that compiles our executable
all : main.cpp
    $(CC) main.cpp -o  main $(COMPILER_FLAGS) $(LINKER_FLAGS) $(BGFX_HEADERS)
```

I've linked the shared version to make compile faster, but you can also link the static library `libbgfxRelease.a` if you want.

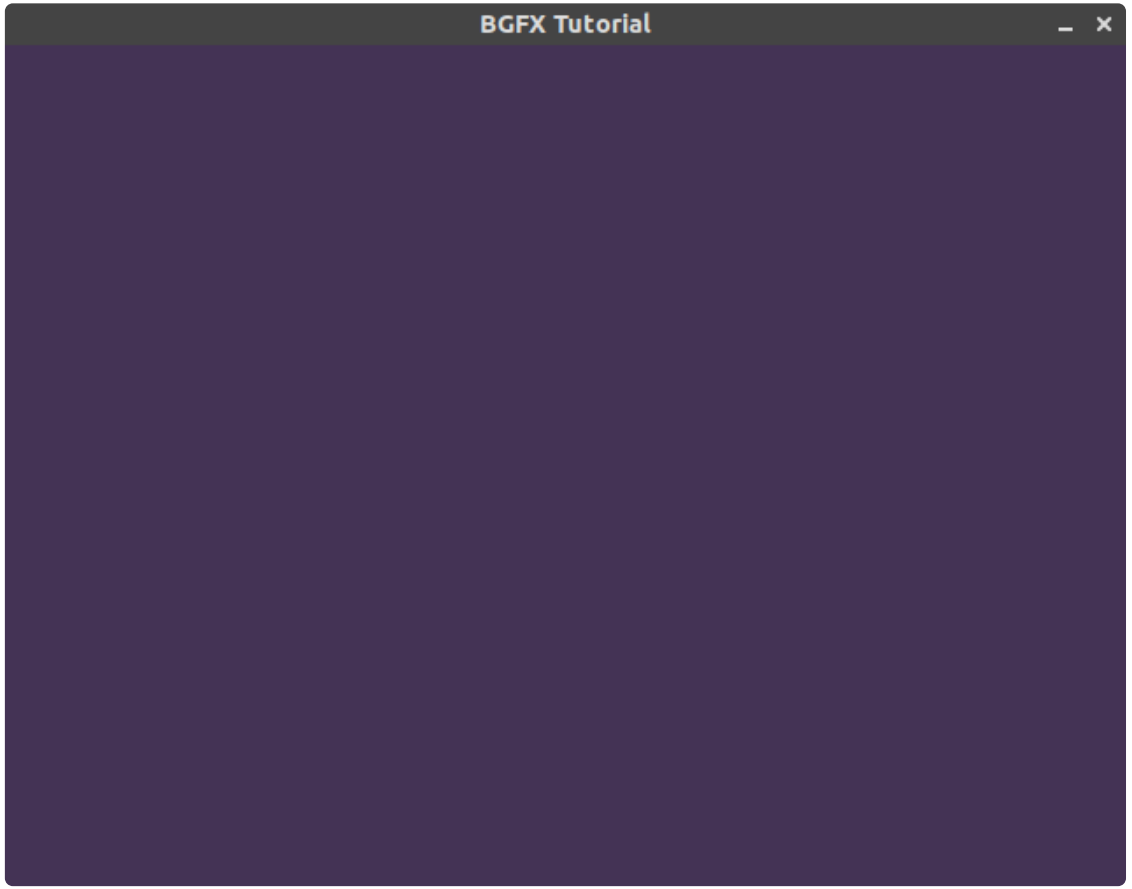
Initializing BGFX

Now that the window is ready and bgfx is linked, lets initialize bgfx in our code.

```
#include <bgfx/bgfx.h>
#include <bgfx/platform.h>
#include <SDL2/SDL_syswm.h>

// Other includes and after opening the window
...

  // Collect information about the window from SDL
  SDL_SysWMinfo wmi;
  SDL_VERSION(&wmi.version);
  if (!SDL_GetWindowWMInfo(window, &wmi)) {
    return 1;
  }

  bgfx::PlatformData pd;
  // and give the pointer to the window to pd
  pd.ndt = wmi.info.x11.display;
  pd.nwh = (void*)(uintptr_t)wmi.info.x11.window;

  // Tell bgfx about the platform and window
  bgfx::setPlatformData(pd);

  // Render an empty frame
  bgfx::renderFrame();

  // Initialize bgfx
  bgfx::init();
p
  // Reset window
  bgfx::reset(WIDTH, HEIGHT, BGFX_RESET_VSYNC);

  // Enable debug text.
  bgfx::setDebug(BGFX_DEBUG_TEXT /*| BGFX_DEBUG_STATS*/);

  // Set view rectangle for 0th view
```

```
  bgfx::setViewRect(0, 0, 0, uint16_t(WIDTH), uint16_t(HEIGHT));

  // Clear the view rect
  bgfx::setViewClear(0,
                     BGFX_CLEAR_COLOR | BGFX_CLEAR_DEPTH,
                     0x443355FF, 1.0f, 0);

  // Set empty primitive on screen
  bgfx::touch(0);


  /*
  Put this inside the event loop of SDL, to render bgfx output
  */
  bgfx::frame();


  /*
  And put this just before SDL_Quit()
  */
  bgfx::shutdown();
...
```

You should see a pleasant purple being rendered in the window now.



Vertices and Indices

If you went through the OpenGL tutorial linked above, you would know that getting a triangle on screen isn't as simple anymore. We need to define a few things before we can see any action on screen. These include:

1. The vertex buffer: a list of 3d vertices that define what your 3d shape looks like in normalized space.

2. The index buffer: a list of vertex indices that define which of the vertices in the vertex buffer form a triangle.
3. A vertex and a fragment shader which compute how each of the vertices are renderered and colored on the screen.
4. A vertex attribute declaration which translate what your vertex buffer means for the shaders.

We know each of the vertices of the square would have a 3 points that denote where the vertex is in space, and lets also have a 4th variable that represents the color of the vertex. We hold these values in a struct and use `bgfx::VertexDecl` to define the vertex attribute declaration.

```
struct PosColorVertex {
    // 3d space position
    float m_x;
    float m_y;
    float m_z;
    // Color value
    uint32_t m_abgr;

    static void init() {
        // start the attribute declaration
        ms_decl
            .begin()
            // Has three float values that denote position
            .add(bgfx::Attrib::Position, 3, bgfx::AttribType::Float)
            // and a uint8 color value that denotes color
            .add(bgfx::Attrib::Color0,   4, bgfx::AttribType::Uint8, true)
            .end();
    };

    static bgfx::VertexDecl ms_decl;
};
```

```
bgfx::VertexDecl PosColorVertex::ms_decl;
```

Lets define the vertices that form our square.

```
static PosColorVertex s_cubeVertices[] =
{
    {  0.5f,  0.5f, 0.0f, 0xff0000ff },
    {  0.5f, -0.5f, 0.0f, 0xff0000ff },
    { -0.5f, -0.5f, 0.0f, 0xff00ff00 },
    { -0.5f,  0.5f, 0.0f, 0xff00ff00 }
};
```

And the indexes of the vertices that form two triangles to make a square.

```
static const uint16_t s_cubeTriList[] =
{
    0,1,3,
    1,2,3
};
```

We also need a `bgfx::VertexBufferHandle` variable to hold the actual vertex buffer and a `bgfx::IndexBufferHandle` variable to hold the index buffer.

```
bgfx::VertexBufferHandle m_vbh;
bgfx::IndexBufferHandle m_ibh;
```

Now we need to initialize the vertex buffer handle and the index buffer handle with the vertices and triangle we have defined above. Put this after `bgfx::init()`.

```
PosColorVertex::init();
m_vbh = bgfx::createVertexBuffer(
            // Static data can be passed with bgfx::makeRef
            bgfx::makeRef(s_cubeVertices, sizeof(s_cubeVertices)),
            PosColorVertex::ms_decl
        );

m_ibh = bgfx::createIndexBuffer(
            // Static data can be passed with bgfx::makeRef
            bgfx::makeRef(s_cubeTriList, sizeof(s_cubeTriList))
        );
```

Shaders

bgfx has a shader language very close to glsl with a few caveats. You can read about it on this page. We need a vertex shader and a fragment shader to render our square on the screen. Lets start with the vertex shader in a new file and name it `v_simple.sc`.

```
$input a_position, a_color0
$output v_color0

#include <bgfx_shader.sh>

void main()
{
    gl_Position = mul(u_modelViewProj, vec4(a_position, 1.0) );
    v_color0 = a_color0;
}
```

`bgfx_shader.sh` has many useful macros that we can use in our shaders. We use the `u_modelViewProj` variable from the file to get a projection of our position into the screen and we pass on the color as it is.

Let's create a fragment shader and name it `f_simple.sc`.

```
$input v_color0

void main()
{
    gl_FragColor = v_color0;
}
```

The fragment shader just returns the color as it is.

bgfx requires another file apart from just the vertex and fragment shaders called `varying.def.sc` which defines the input and output variables being used in these shaders. Lets create that as well.

```
// outputs
vec4 v_color0 : COLOR0;

// inputs
vec3 a_position : POSITION;
vec4 a_color0 : COLOR0;
```

Now we need to compile these shaders before we can use them with bgfx. The tool for that was built along with the bgfx library. You can find the tool `shadercRelease` in `bgfx/.build/linux64_gcc/bin`.

Lets run the `shaderc` tool to create our compiled shader files.

```
./bgfx/.build/linux64_gcc/bin/shadercRelease \
    -f v_simple.sc \
    -o v_simple.bin \
    --platform linux \
    --type vertex \
    --verbose \
    -i bgfx/src
./bgfx/.build/linux64_gcc/bin/shadercRelease \
    -f f_simple.sc \
    -o f_simple.bin \
    --platform linux \
    --type fragment \
    --verbose \
    -i bgfx/src
```

We can add these lines to our make file as well so that the shaders are recompiled when we build.

```
#This is the target that compiles our executable
all : main.cpp
    ./bgfx/.build/linux64_gcc/bin/shadercRelease \
    -f v_simple.sc \
    -o v_simple.bin \
    --platform linux \
    --type vertex \
    --verbose \
    -i bgfx/src
    ./bgfx/.build/linux64_gcc/bin/shadercRelease \
    -f f_simple.sc \
    -o f_simple.bin \
    --platform linux \
```

```
    --type fragment \
    --verbose \
    -i bgfx/src
    $(CC) main.cpp -o  main $(COMPILER_FLAGS) $(LINKER_FLAGS) $(BGFX_HEADERS)
```

Next up we want to load our shaders into memory and into `bgfx::ProgramHandle` so that we can use them to render our square.

```
#include <fstream>
bgfx::ShaderHandle loadShader(const char* _name) {
    char* data = new char[2048];
    std::ifstream file;
    size_t fileSize
    file.open(_name);
    if(file.is_open()) {
        file.seekg(0, std::ios::end);
        fileSize = file.tellg();
        file.seekg(0, std::ios::beg);
        file.read(data, fileSize);
        file.close();
    }
    const bgfx::Memory* mem = bgfx::copy(data,fileSize+1);
    mem->data[mem->size-1] = '\0';
    bgfx::ShaderHandle handle = bgfx::createShader(mem);
    bgfx::setName(handle, _name);
    return handle;
}

bgfx::VertexBufferHandle m_vbh;
bgfx::IndexBufferHandle m_ibh;
bgfx::ProgramHandle m_program; // we create a program handle

...
    // after creating the vertex and index buffers
```

```
    // we initialize our programhandle
    bgfx::ShaderHandle vsh = loadShader("v_simple.bin");
    bgfx::ShaderHandle fsh = loadShader("f_simple.bin");
    m_program = bgfx::createProgram(vsh,fsh,  true);
...
```

We're so close. We have everything we need to start rendering a square on the screen. We need to define where the camera is and where it is looking at.

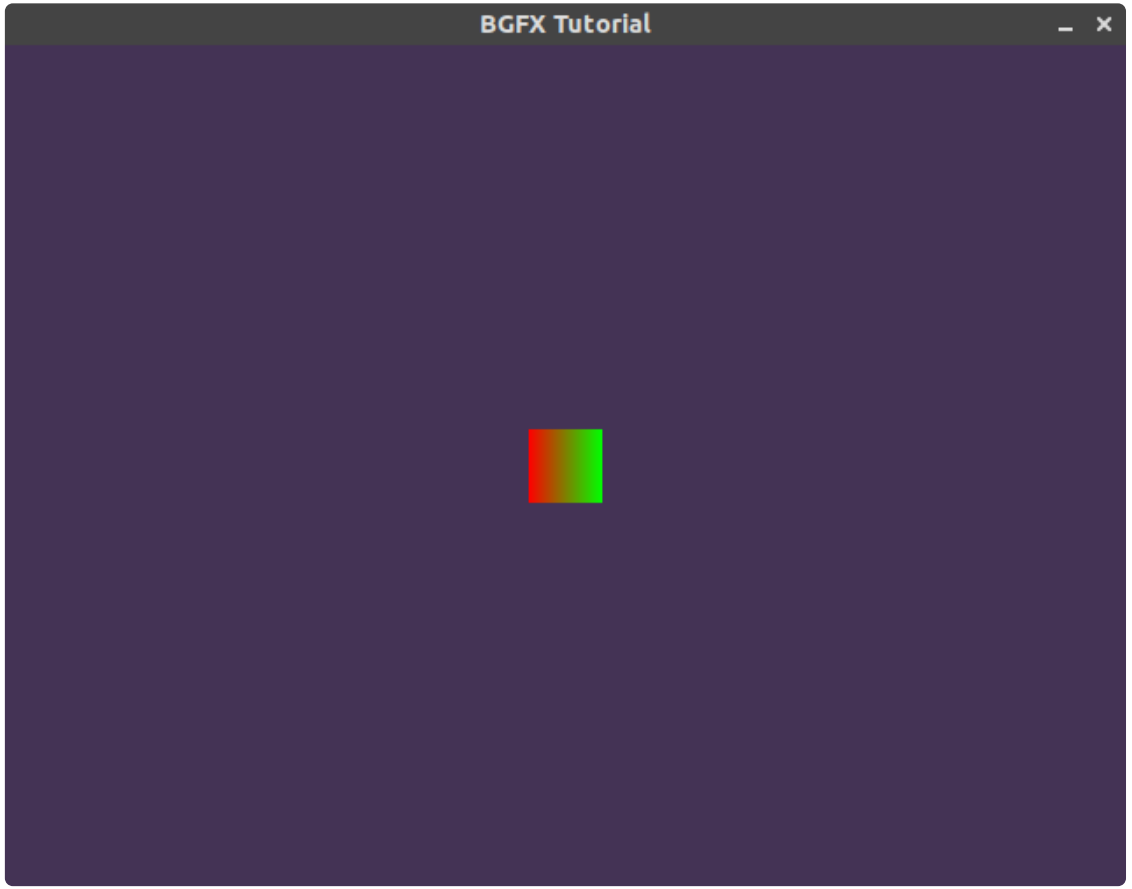
```
#include <bx/math.h>

// inside the while loop

const bx::Vec3 at  = { 0.0f, 0.0f,   0.0f };
const bx::Vec3 eye = { 0.0f, 0.0f, 10.0f };

// Set view and projection matrix for view 0.
float view[16];
bx::mtxLookAt(view, eye, at);

float proj[16];
bx::mtxProj(proj,
            60.0f,
            float(WIDTH)/float(HEIGHT),
            0.1f, 100.0f,
            bgfx::getCaps()->homogeneousDepth);

bgfx::setViewTransform(0, view, proj);

// Set view 0 default viewport.
bgfx::setViewRect(0, 0, 0,
                  WIDTH,
                  HEIGHT);
```

```
bgfx::touch(0);
```

Next we define a 4x4 matrix that defines where our square is going to be and what orientation it is at. We also set the index and vertex buffer handles and then submit the drawing with the program.

```
float mtx[16];
bx::mtxRotateY(mtx, 0.0f);

// position x,y,z
mtx[12] = 0.0f;
mtx[13] = 0.0f;
mtx[14] = 0.0f;

// Set model matrix for rendering.
bgfx::setTransform(mtx);

// Set vertex and index buffer.
bgfx::setVertexBuffer(0, m_vbh);
bgfx::setIndexBuffer(m_ibh);

// Set render states.
bgfx::setState(BGFX_STATE_DEFAULT);

// Submit primitive for rendering to view 0.
bgfx::submit(0, m_program);

bgfx::frame();
```

And thats it! Make and run and you would see a square with a horizontal gradient of red to green.



This is the whole `main.cpp` program.

```
#include <stdio.h>
#include <SDL2/SDL.h>
#include <bgfx/bgfx.h>
#include <bgfx/platform.h>
#include <SDL2/SDL_syswm.h>
#include <fstream>
#include <bx/math.h>
```

```
SDL_Window* window = NULL;
const int WIDTH = 640;
const int HEIGHT = 480;

bgfx::ShaderHandle loadShader(const char* _name) {
    char* data = new char[2048];
    std::ifstream file;
    size_t fileSize;
    file.open(_name);
    if(file.is_open()) {
        file.seekg(0, std::ios::end);
        fileSize = file.tellg();
        file.seekg(0, std::ios::beg);
        file.read(data, fileSize);
        file.close();
    }
    const bgfx::Memory* mem = bgfx::copy(data,fileSize+1);
    mem->data[mem->size-1] = '\0';
    bgfx::ShaderHandle handle = bgfx::createShader(mem);
    bgfx::setName(handle, _name);
    return handle;
}

struct PosColorVertex {
  float m_x;
  float m_y;
  float m_z;
  uint32_t m_abgr;

  static void init() {
    ms_decl
      .begin()
      .add(bgfx::Attrib::Position, 3, bgfx::AttribType::Float)
      .add(bgfx::Attrib::Color0,   4, bgfx::AttribType::Uint8, true)
      .end();
```

```
  };

  static bgfx::VertexDecl ms_decl;
};


bgfx::VertexDecl PosColorVertex::ms_decl;

static PosColorVertex s_cubeVertices[] =
  {
   {  0.5f,  0.5f, 0.0f, 0xff0000ff },
   {  0.5f, -0.5f, 0.0f, 0xff0000ff },
   { -0.5f, -0.5f, 0.0f, 0xff00ff00 },
   { -0.5f,  0.5f, 0.0f, 0xff00ff00 }
  };


static const uint16_t s_cubeTriList[] =
  {
   0,1,3,
   1,2,3
  };

bgfx::VertexBufferHandle m_vbh;
bgfx::IndexBufferHandle m_ibh;
bgfx::ProgramHandle m_program;

int main (int argc, char* args[]) {

  // Initialize SDL systems
  if(SDL_Init( SDL_INIT_VIDEO ) < 0) {
    printf("SDL could not initialize! SDL_Error: %s\n",
           SDL_GetError());
  }
  else {
```

```
  //Create a window
  window = SDL_CreateWindow("BGFX Tutorial",
                            SDL_WINDOWPOS_UNDEFINED,
                            SDL_WINDOWPOS_UNDEFINED,
                            WIDTH, HEIGHT, SDL_WINDOW_SHOWN);
  if( window == NULL ) {
    printf("Window could not be created! SDL_Error: %s\n",
           SDL_GetError());
  }
}

SDL_SysWMinfo wmi;
SDL_VERSION(&wmi.version);
if (!SDL_GetWindowWMInfo(window, &wmi)) {
  return 1;
}

bgfx::PlatformData pd;
// and give the pointer to the window to pd
pd.ndt = wmi.info.x11.display;
pd.nwh = (void*)(uintptr_t)wmi.info.x11.window;

// Tell bgfx about the platform and window
bgfx::setPlatformData(pd);

// Render an empty frame
bgfx::renderFrame();

// Initialize bgfx
bgfx::init();

PosColorVertex::init();
m_vbh = bgfx::createVertexBuffer(
            // Static data can be passed with bgfx::makeRef
            bgfx::makeRef(s_cubeVertices, sizeof(s_cubeVertices)),
```

```
            PosColorVertex::ms_decl
        );

m_ibh = bgfx::createIndexBuffer(
            // Static data can be passed with bgfx::makeRef
            bgfx::makeRef(s_cubeTriList, sizeof(s_cubeTriList))
        );

bgfx::ShaderHandle vsh = loadShader("v_simple.bin");
bgfx::ShaderHandle fsh = loadShader("f_simple.bin");

m_program = bgfx::createProgram(vsh,fsh,  true);



// Reset window
bgfx::reset(WIDTH, HEIGHT, BGFX_RESET_VSYNC);

// Enable debug text.
bgfx::setDebug(BGFX_DEBUG_TEXT /*| BGFX_DEBUG_STATS*/);

// Set view rectangle for 0th view
bgfx::setViewRect(0, 0, 0, uint16_t(WIDTH), uint16_t(HEIGHT));

// Clear the view rect
bgfx::setViewClear(0,
                   BGFX_CLEAR_COLOR | BGFX_CLEAR_DEPTH,
                   0x443355FF, 1.0f, 0);


// Set empty primitive on screen
bgfx::touch(0);

// Poll for events and wait till user closes window
bool quit = false;
```

```
SDL_Event currentEvent;
while(!quit) {
  while(SDL_PollEvent(&currentEvent) != 0) {
    if(currentEvent.type == SDL_QUIT) {
      quit = true;
    }

    const bx::Vec3 at  = { 0.0f, 0.0f,   0.0f };
    const bx::Vec3 eye = { 0.0f, 0.0f, 10.0f };

    // Set view and projection matrix for view 0.
    float view[16];
    bx::mtxLookAt(view, eye, at);

    float proj[16];
    bx::mtxProj(proj,
                60.0f,
                float(WIDTH)/float(HEIGHT),
                0.1f, 100.0f,
                bgfx::getCaps()->homogeneousDepth);

    bgfx::setViewTransform(0, view, proj);

    // Set view 0 default viewport.
    bgfx::setViewRect(0, 0, 0,
                      WIDTH,
                      HEIGHT);

    bgfx::touch(0);


    float mtx[16];
    bx::mtxRotateY(mtx, 0.0f);

    // position x,y,z
```

```
      mtx[12] = 0.0f;
      mtx[13] = 0.0f;
      mtx[14] = 0.0f;

      // Set model matrix for rendering.
      bgfx::setTransform(mtx);

      // Set vertex and index buffer.
      bgfx::setVertexBuffer(0, m_vbh);
      bgfx::setIndexBuffer(m_ibh);

      // Set render states.
      bgfx::setState(BGFX_STATE_DEFAULT);

      // Submit primitive for rendering to view 0.
      bgfx::submit(0, m_program);

      bgfx::frame();
    }
  }

  bgfx::shutdown();
  // Free up window
  SDL_DestroyWindow(window);
  // Shutdown SDL
  SDL_Quit();

  return 0;
}
```

This program is a direct subversion of the cubes example from the examples list in the bgfx docs. I had to dig in to really make sense of how the shaders needed to be and how to set up the camera. I hope that integrating bgfx on Ubuntu would be an easier task for you now. Between

the examples and the docs, it is fairly easy to figure out the corresponding steps for other platforms. You can find the full repository of this tutorial at Github.

Happy Coding!

« Emacs setup for C++

An update - Leaving Livelike »

Related Posts

- Emacs setup for C++

Sandeep Nambiar
Measuring the world

9 Comments Sandeep Nambiar Privacy Policy

Aleksey Komarov

Favorite Tweet Share Sort by Best

Join the discussion…

Aleksey Komarov • a few seconds ago

I updated bgfx, enable SPIRV shader for Vulkan and add github CI to make sure, that it's working on Ubuntu 20.04:
https://github.com/q4a/bgfx...

• Edit • Reply • Share ›

dimsh • 3 years ago

comments in varying.def.sc lead to shader compilation fail on Windows (didn't check it on other systems).
So, if anyone sees errors like:

Error: (393,43): error: a_position' undeclared (393,38): error: cannot constructvec4' from a non-numeric data type

when compile simplest shaders and all of $input, $output are placed correctly, then try to remove comments in varying.def.sc and leave just:

vec4 v_color0 : COLOR0;

vec3 a_position : POSITION;
vec4 a_color0 : COLOR0;

To get shaderc on windows don't forget to build with --with-tools to make correct solution.

Thank you for that article! I just want to make it more clearer for others.

2 • Reply • Share ›

dbotha ➚ dimsh • a year ago

Same on OSX, remove the comments in varying.def.sc and it will compile.

• Reply • Share ›

Theogen R. ➚ dimsh • 2 years ago

I can confirm that this is the case on Linux as well.

• Reply • Share ›

Haiek Nagasaki ➚ dimsh • 3 years ago

I wish I had seen this earlier, I spent way too long figuring that out :(

• Reply • Share ›

Theogen R. • 2 years ago

`VertexDecl` has been renamed to `VertexLayout`.

1 • Reply • Share ›

Michael Katz • 2 years ago

I might be misunderstanding, but I *believe* this whole example is backward in terms of the z axis. If you look at the definition of the triangles in s_cubeVertices and s_cubeTriList, you see they are clockwise. Those triangles would be culled (invisible) if the camera were looking at their front side (because BGFX_STATE_DEFAULT culls clockwise triangles). However, this is counter-balanced by the fact that the Z value of the eye vector is 10, and because a left handed coordinate system is used by default in mtxLookAt() and mtxProj(), that means 10 units *into* the screen, "behind" the square. So we are looking at the square from behind, but it shows because the triangles are clockwise, so from behind they are counter-clockwise. You can see the result in the screenshot: the square goes from red on the left to green on the right. But if you look at the first

goes from red on the left to green on the right. But if you look at the first entry in s_cubeVertices, it's { 0.5f, 0.5f, 0.0f, 0xff0000ff }. That last value is abgr, so it's red, meaning that the top-right corner of the cube should be red, but in the screenshot it's green.

Reply • Share ›

Michael Katz • 2 years ago

I believe the data array is leaked in the function loadShader().

Reply • Share ›

dimsh • 3 years ago

And it's fine in Git repo.

Reply • Share ›