Graphics -1

The intro to intro to graphics

New sem, who dis?

- SP Game Dev Alumni
- Current Year 3 RTIS
- Graphics Boi
- OpenGL for GAM150/200
- Vulkan GAM300

Aesir

Minute

Foreword

The thing about graphics...

It's difficult

The thing about graphics...

It's difficult

The thing about graphics...

It's difficult

The thing about graphics...

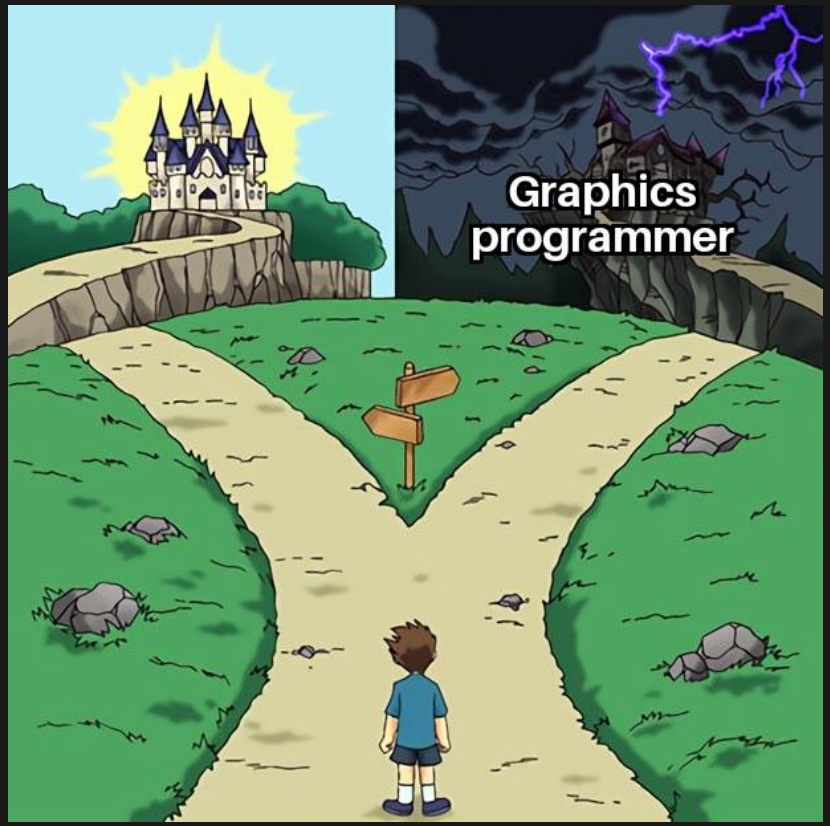
It's difficult

The thing about graphics...

It's difficult

The thing about graphics...

It's difficult

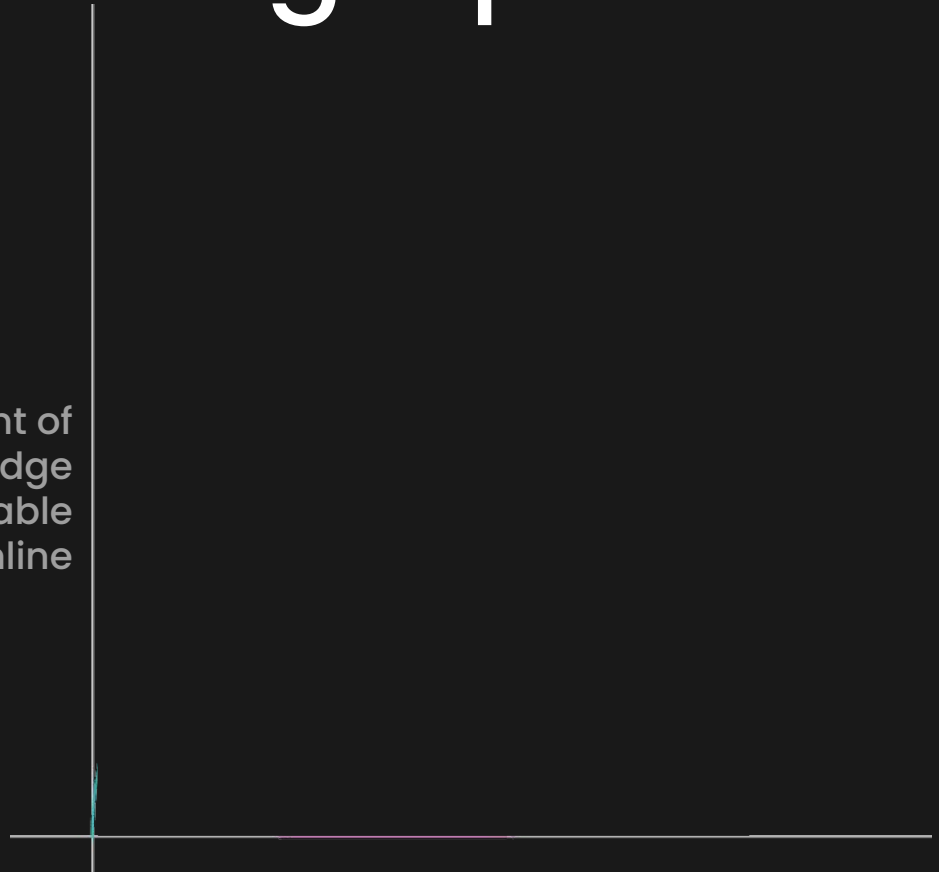


The thing about graphics...

is the knowledge

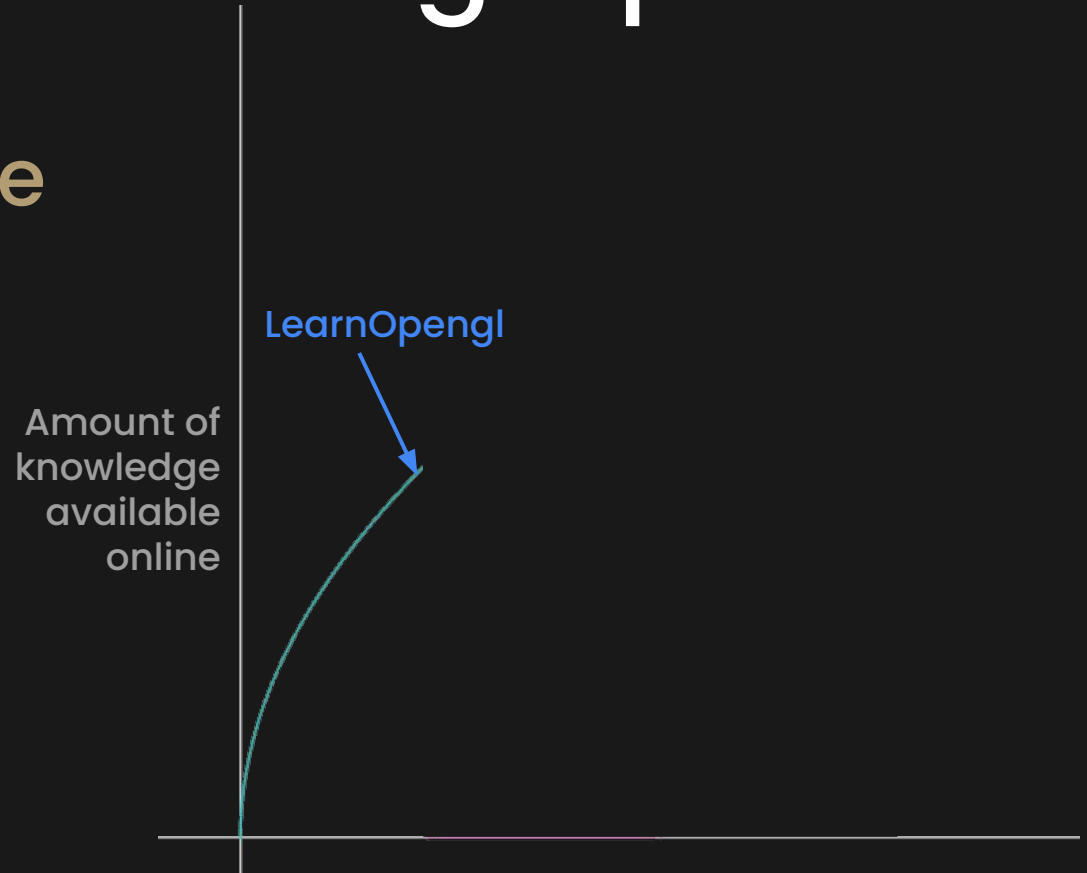
The thing about graphics...

is the knowledge



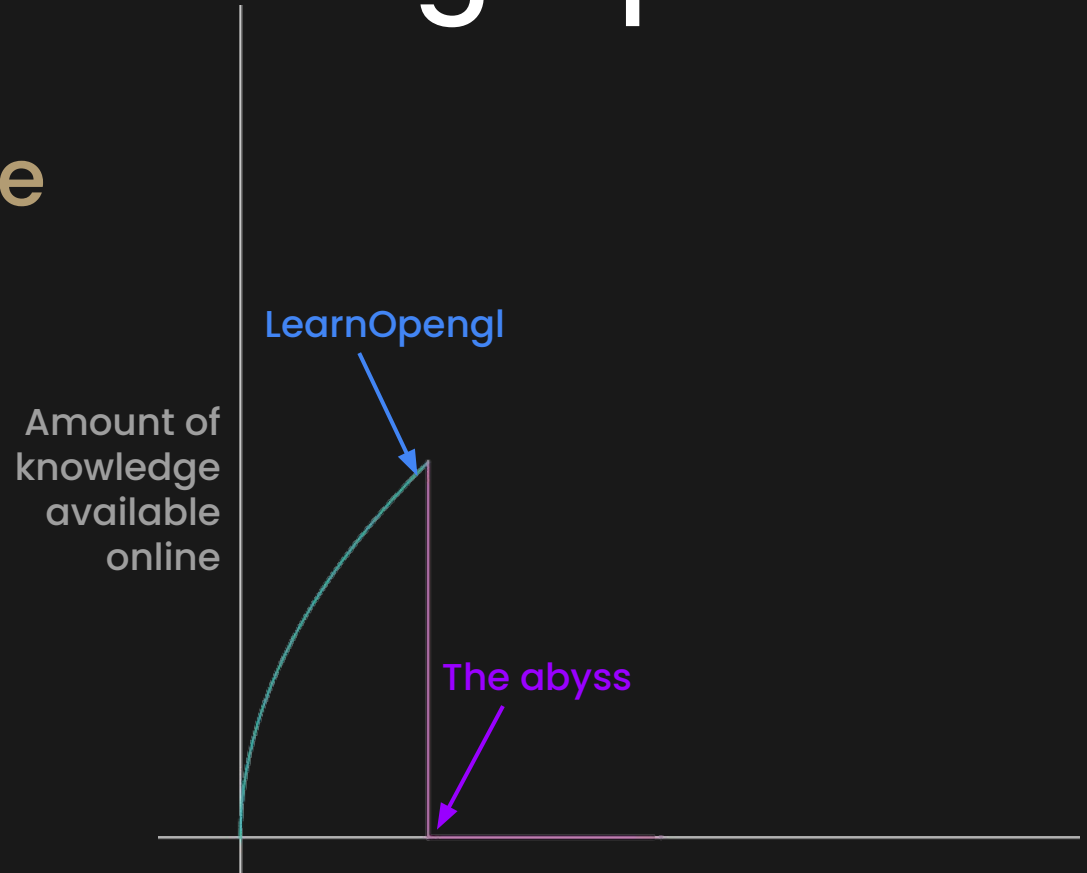
The thing about graphics...

is the knowledge



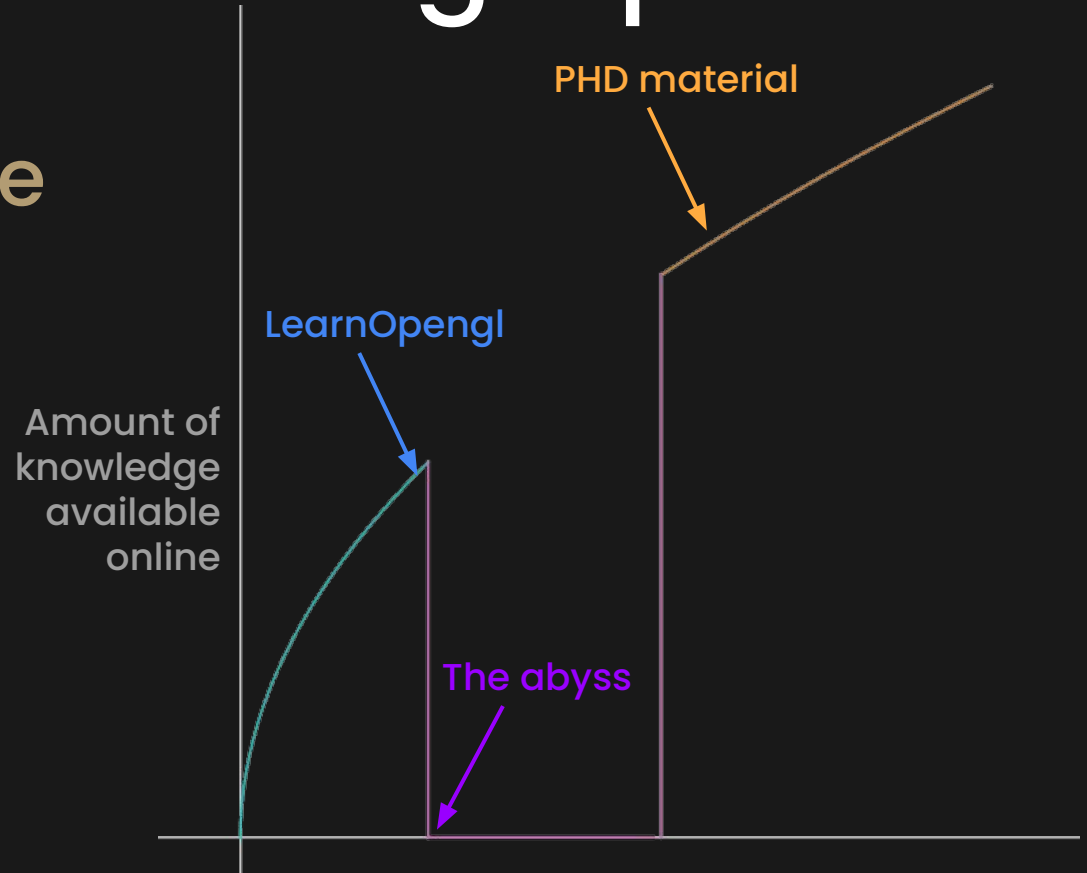
The thing about graphics...

is the knowledge



The thing about graphics...

is the knowledge



The thing about graphics...

is learning a second language

The thing about graphics...

Is the parallelism

The thing about graphics...

is that 99% of the time..

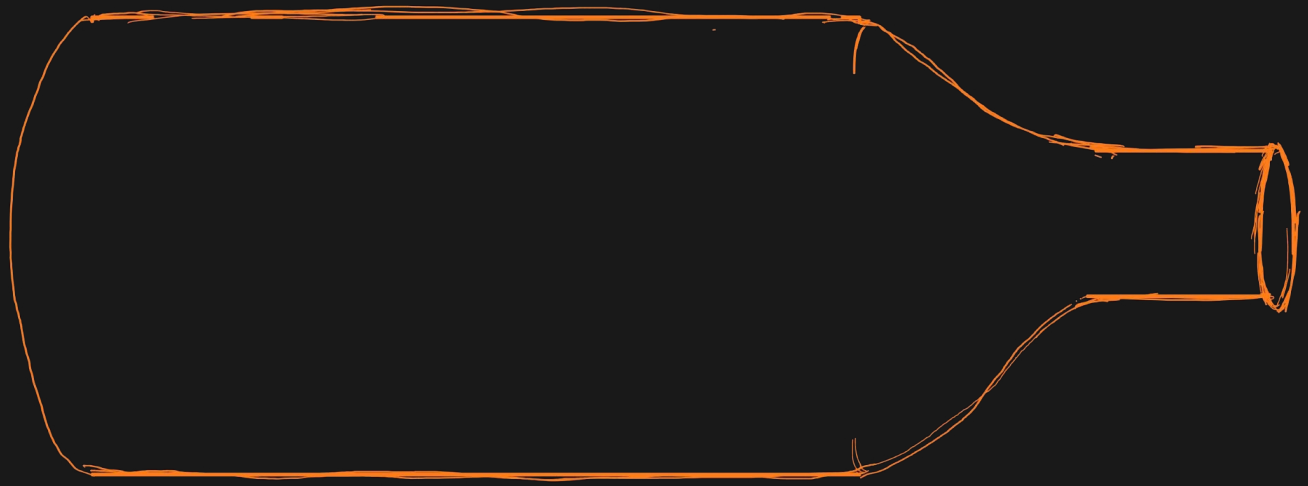
The thing about graphics...

is that 99% of the time..

You are the bottleneck

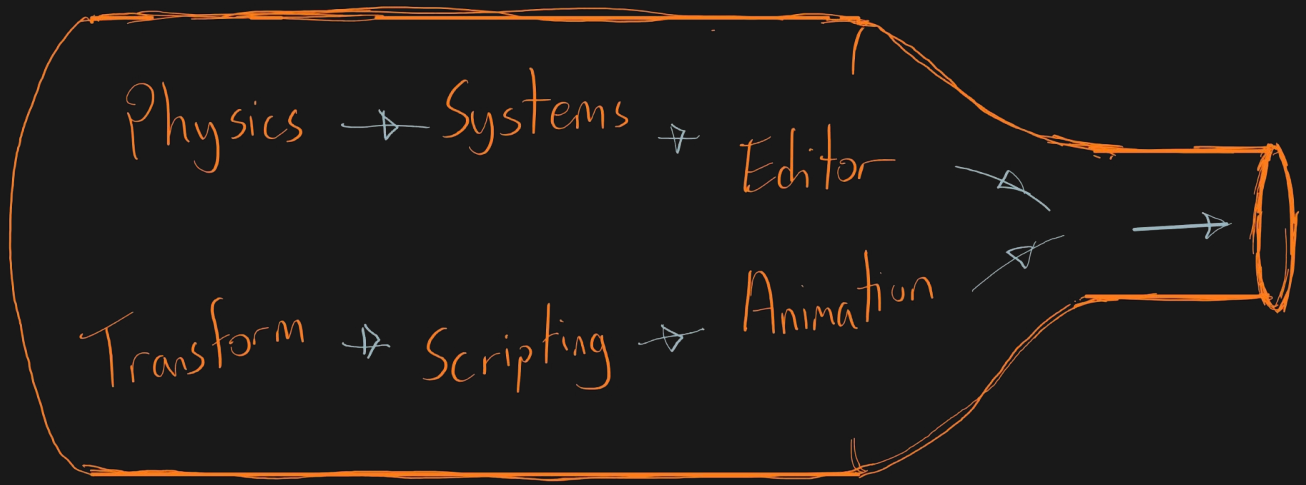
The thing about graphics...

is that 99% of the time you are the bottleneck



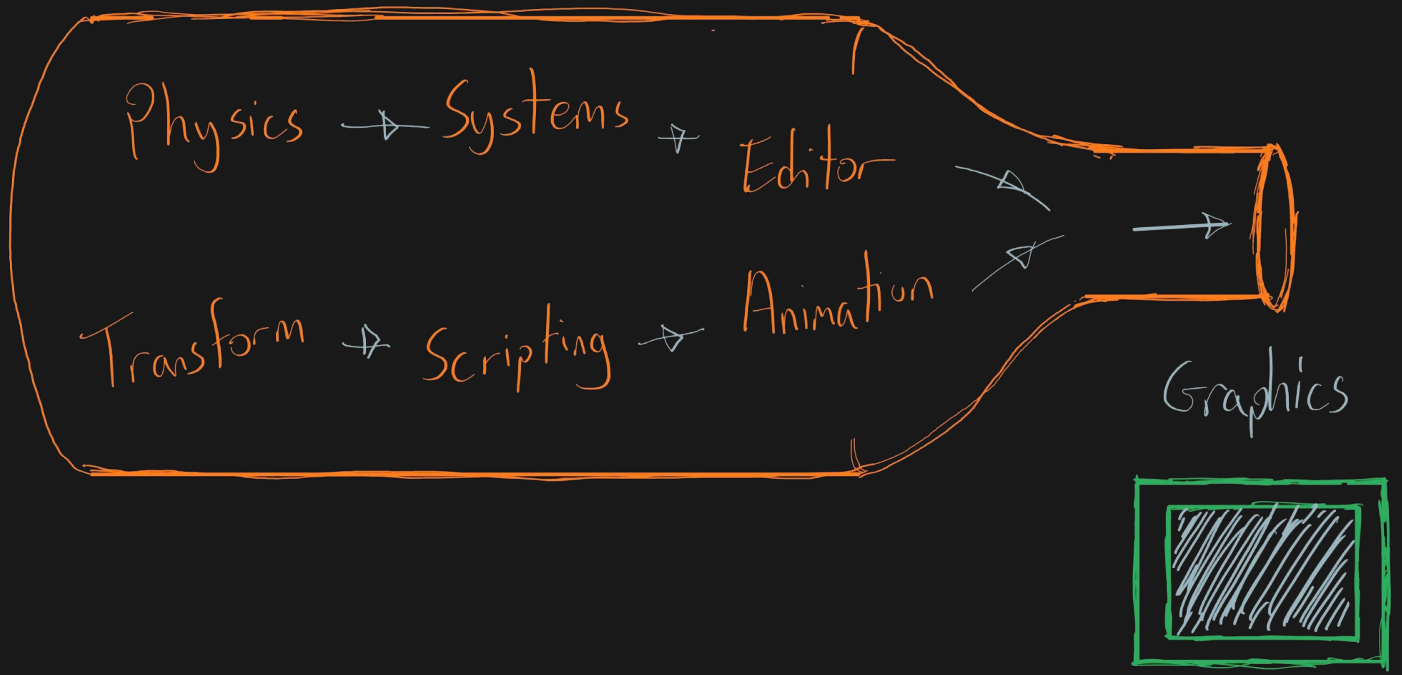
The thing about graphics...

is that 99% of the time you are the bottleneck



The thing about graphics...

is that 99% of the time you are the bottleneck



The thing about graphics...

It's extremely rewarding

The thing about graphics...

It's an exclusive club

What is this talk

Basic
Intermediate
Advanced

What are we going to do

Get you up and sprinting with OpenGL
Learn a bit about the pipeline
Some rendering tips

Rendering API

Direct3D

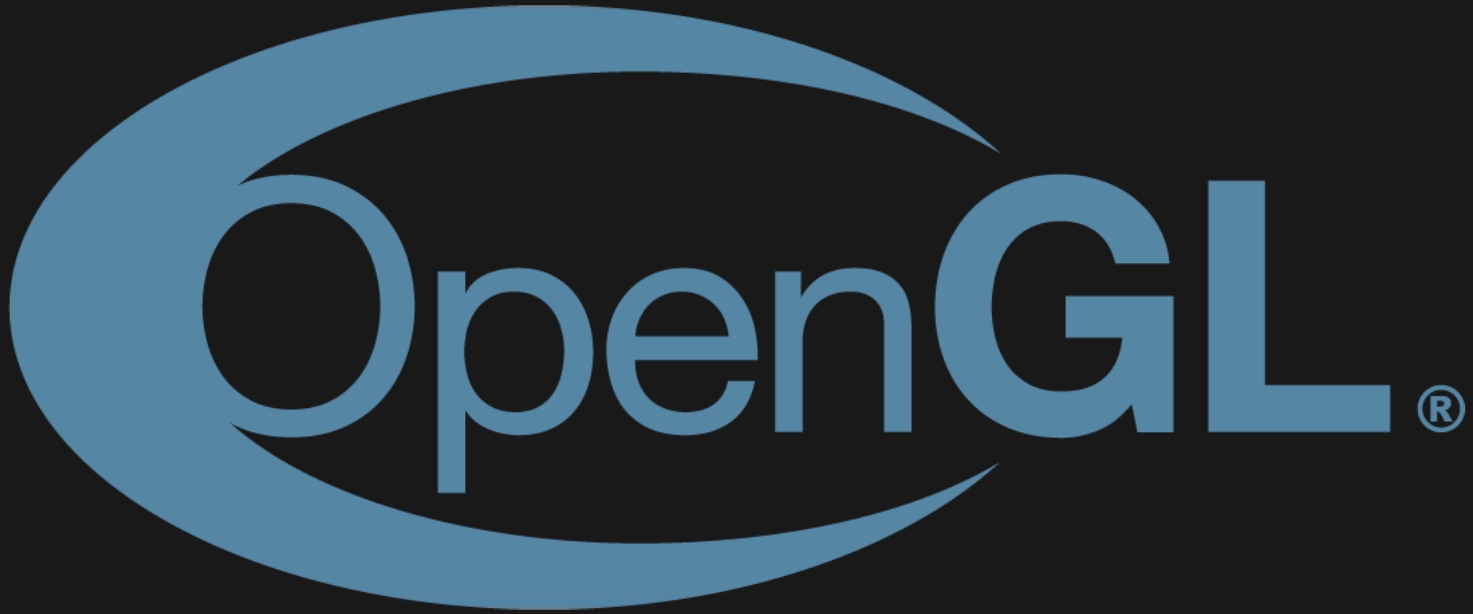
OpenGL

Vulkan

Metal

PS4

OpenGL

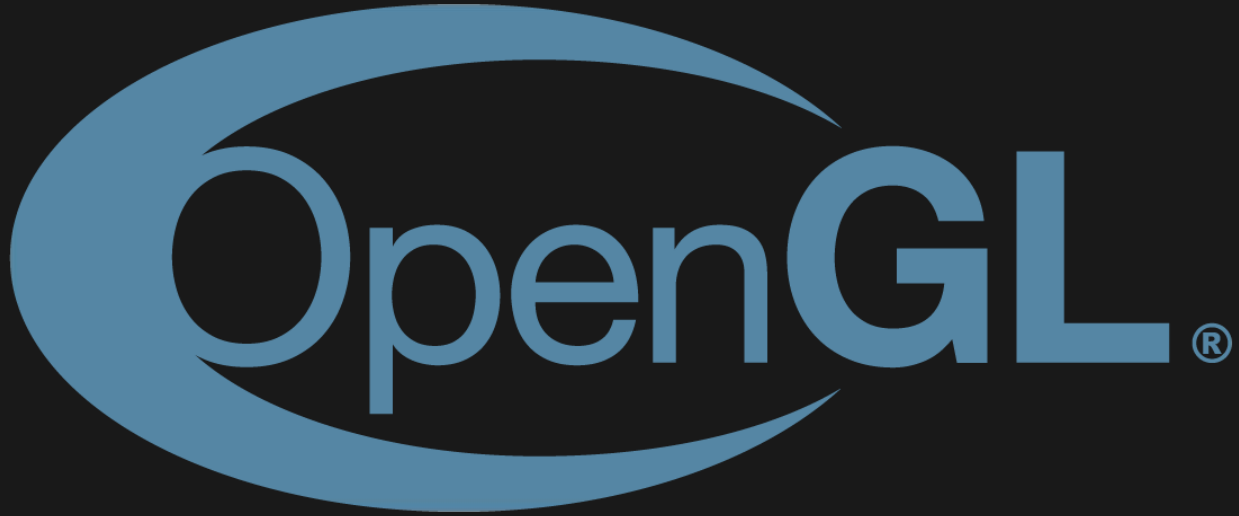


OpenGL

Gigantic state machine

Easy to use

Heavily abstracted



How to start with OpenGL

1. Make a window

2. Create an OpenGL context

3. Start doing cool stuff

Let’s make a window

How about a Win32 window?

```
// Register the window class.
const wchar_t CLASS_NAME[]  = L"Sample Window Class";

WNDCLASS wc = { };

wc.lpfnWndProc   = WindowProc;
wc.hInstance     = hInstance;
wc.lpszClassName = CLASS_NAME;

RegisterClass(&wc);

// Create the window.

HWND hwnd = CreateWindowEx(
    0,                              // Optional window styles.
    CLASS_NAME,                     // Window class
    L"Learn to Program Windows",    // Window text
    WS_OVERLAPPEDWINDOW,            // Window style

    // Size and position
    CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,

    NULL,       // Parent window
    NULL,       // Menu
    hInstance,  // Instance handle
    NULL        // Additional application data
    );

if (hwnd == NULL)
{
    return 0;
}

ShowWindow(hwnd, nCmdShow);
```

a Win32 window?

```
// Register the window class.
const wchar_t CLASS_NAME[]  = L"Sample Window Class";

WNDCLASS wc = { };

wc.lpfnWndProc   = WindowProc;
wc.hInstance     = hInstance;
wc.lpszClassName = CLASS_NAME;

RegisterClass(&wc);

// Create the window.

HWND hwnd = CreateWindowEx(
    0,                              // Optional window styles.
    CLASS_NAME,                     // Window class
    L"Learn to Program Windows",    // Window text
    WS_OVERLAPPEDWINDOW,            // Window style

    // Size and position
    CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,

    NULL,       // Parent window
    NULL,       // Menu
    hInstance,  // Instance handle
    NULL        // Additional application data
    );

if (hwnd == NULL)
{
    return 0;
}

ShowWindow(hwnd, nCmdShow);
```

Processing Keystroke Messages

The window procedure of the window that has the keyboard focus receives keystroke messages when the user type keystroke messages are **WM_KEYDOWN**, **WM_KEYUP**, **WM_SYSKEYDOWN**, and **WM_SYSKEYUP**. A typical window keystroke messages except **WM_KEYDOWN**. The system posts the **WM_KEYDOWN** message when the user presse

When the window procedure receives the **WM_KEYDOWN** message, it should examine the virtual-key code that ac determine how to process the keystroke. The virtual-key code is in the message's *wParam* parameter. Typically, an keystrokes generated by noncharacter keys, including the function keys, the cursor movement keys, and the specia DEL, HOME, and END.

The following example shows the window procedure framework that a typical application uses to receive and proce

```
        case WM_KEYDOWN:
            switch (wParam)
            {
                case VK_LEFT:

                    // Process the LEFT ARROW key.

                    break;

                case VK_RIGHT:

                    // Process the RIGHT ARROW key.

                    break;

                case VK_UP:

                    // Process the UP ARROW key.

                    break;

                case VK_DOWN:

                    // Process the DOWN ARROW key.
```

```
// Register the window class.
```

Client Area Mouse Messages

A window receives a client area mouse message when a mouse event occurs within the window's client area. The system posts the **WM_MOUSEMOVE** message to the window when the user moves the cursor within the client area. It posts one of the following messages when the user presses or releases a mouse button while the cursor is within the client area.

Message	Meaning
WM_LBUTTONDBLCLK	The left mouse button was double-clicked.
WM_LBUTTONDOWN	The left mouse button was pressed.
WM_LBUTTONUP	The left mouse button was released.
WM_MBUTTONDBLCLK	The middle mouse button was double-clicked.
WM_MBUTTONDOWN	The middle mouse button was pressed.
WM_MBUTTONUP	The middle mouse button was released.
WM_RBUTTONDBLCLK	The right mouse button was double-clicked.
WM_RBUTTONDOWN	The right mouse button was pressed.
WM_RBUTTONUP	The right mouse button was released.
WM_XBUTTONDBLCLK	An X mouse button was double-clicked.
WM_XBUTTONDOWN	An X mouse button was pressed.
WM_XBUTTONUP	An X mouse button was released.

Processing Keystroke Messages

sages when the user type
KEYUP. A typical window
age when the user presse

e virtual-key code that ac
parameter. Typically, an
nent keys, and the specia

uses to receive and proce

```
// Register the
Cl
th
case WM_MOUSEWHEEL:
    /*
     * Do not handle zoom and datazo
     */
    if (wParam & (MK_SHIFT | MK_CONTROL)) {
        goto PassToDefaultWindowProc;
    }

    gcWheelDelta -= (short) HIWORD(wParam);
    if (abs(gcWheelDelta) >= WHEEL_DELTA && gucWheelScrollLines > 0)
    {
        int cLineScroll;

        /*
         * Limit a roll of one (1) WHEEL_DELTA to
         * scroll one (1) page.
         */
        cLineScroll = (int) min(
                (UINT) pWndData->ichLinesOnScreen - 1,
                gucWheelScrollLines);

        if (cLineScroll == 0) {
            cLineScroll++;
        }
        ll *= (gcWheelDelta / WHEEL_DELTA);
        oll != 0);
        % WHEEL_DELTA;
        eScroll);
```

WM_
WM_LB
WM_LBU
WM_MBUT
WM_MBUTTO
WM_MBUTTONU
WM_RBUTTONDBL
WM_RBUTTONDOW
WM_RBUTTONUP
WM_XBUTTONDBLCLK
WM_XBUTTONDOWN
WM_XBUTTONUP

sages when the user type
KEYUP. A typical window
age when the user presse
sages when
e virtual-key code that ac
parameter. Typically, an
nent keys, and the specia
uses to receive and proce
An

Yeah... No thanks.

Maybe next time

GLFW

Using GLFW

```
#include <GLFW/glfw3.h>
```

```
// glfw: initialize and configure
// ------------------------------
glfwInit();
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
```

```
GLFWwindow* window = glfwCreateWindow(SCR_WIDTH, SCR_HEIGHT, "LearnOpenGL", NULL, NULL);
if (window == NULL)
{
    std::cout << "Failed to create GLFW window" << std::endl;
    glfwTerminate();
    return -1;
}
glfwMakeContextCurrent(window);
```

```
while (!glfwWindowShouldClose(window))
{
    glfwPollEvents();
}
```

How to start with OpenGL

1. ~~Make a window~~

2. Create an OpenGL context

3. Start doing cool stuff

Manually loading OpenGL functions

```
void *GetAnyGLFuncAddress(const char *name)
{
  void *p = (void *)wglGetProcAddress(name);
  if(p == 0 ||
    (p == (void*)0x1) || (p == (void*)0x2) || (p == (void*)0x3) ||
    (p == (void*)-1) )
  {
    HMODULE module = LoadLibraryA("opengl32.dll");
    p = (void *)GetProcAddress(module, name);
  }

  return p;
}
```

Manually loading OpenGL functions

```
void *GetAnyGLFuncAddress(const char *name)
{
  void *p = (void *)wglGetProcAddress(name);
  if(p == 0 ||
    (p == (void*)0x1) || (p == (void*)0x2) || (p == (void*)0x3) ||
    (p == (void*)-1) )
```

In practice, if two contexts come from the same vendor and refer to the same GPU, then the function pointers pulled from one context will work in the other. This is important when creating an OpenGL context in Windows, as you need to create a "dummy" context to get WGL extension functions to create the *real* one.

```
// Now we can choose a pixel format the modern way, using wglChoosePixelFormatARB.
int pixel_format_attribs[] = {
    WGL_DRAW_TO_WINDOW_ARB,     GL_TRUE,
    WGL_SUPPORT_OPENGL_ARB,     GL_TRUE,
    WGL_DOUBLE_BUFFER_ARB,      GL_TRUE,
    WGL_ACCELERATION_ARB,       WGL_FULL_ACCELERATION_ARB,
    WGL_PIXEL_TYPE_ARB,         WGL_TYPE_RGBA_ARB,
    WGL_COLOR_BITS_ARB,         32,
    WGL_DEPTH_BITS_ARB,         24,
    WGL_STENCIL_BITS_ARB,       8,
    0
};

int pixel_format;
UINT num_formats;
wglChoosePixelFormatARB(real_dc, pixel_format_attribs, 0, 1, &pixel_format, &num_for
if (!num_formats) {
    fatal_error("Failed to set the OpenGL 3.3 pixel format.");
}

PIXELFORMATDESCRIPTOR pfd;
DescribePixelFormat(real_dc, pixel_format, sizeof(pfd), &pfd);
if (!SetPixelFormat(real_dc, pixel_format, &pfd)) {
    fatal_error("Failed to set the OpenGL 3.3 pixel format.");
}
```

Please don’t

Just don’t

Loading OpenGL Functions is an important task for initializing OpenGL after creating an OpenGL context. You are *strongly* advised to use an OpenGL Loading Library instead of a manual process. However, if you want to know how it works manually, read on.

Please don’t

Just don’t

Use a loader library

They do it all for you

Using GLAD

```
#include <glad/glad.h>
// ------------------------------
if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
{
    std::cout << "Failed to initialize GLAD" << std::endl;
    return -1;
}
```

We have an OpenGL app

```
#include <glad/glad.h>
#include <GLFW/glfw3.h>
```
// note that the order matters

```
// glfw: initialize and configure
// ------------------------------
glfwInit();
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
```

```
GLFWwindow* window = glfwCreateWindow(SCR_WIDTH, SCR_HEIGHT, "LearnOpenGL", NULL, NULL);
if (window == NULL)
{
    std::cout << "Failed to create GLFW window" << std::endl;
    glfwTerminate();
    return -1;
}
glfwMakeContextCurrent(window);
```

```
if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
{
    std::cout << "Failed to initialize GLAD" << std::endl;
    return -1;
}
```

```
while (!glfwWindowShouldClose(window))
{
    glfwPollEvents();
}
```

We have an OpenGL app

1. ~~Make a window~~
2. ~~Create an OpenGL context~~
3. Start doing cool stuff

```
#include <glad/glad.h>
#include <GLFW/glfw3.h>
```
// note that the order matters

```
// glfw: initialize and configure
// ------------------------------
glfwInit();
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
```

```
GLFWwindow* window = glfwCreateWindow(SCR_WIDTH, SCR_HEIGHT, "LearnOpenGL", NULL, NULL);
if (window == NULL)
{
    std::cout << "Failed to create GLFW window" << std::endl;
    glfwTerminate();
    return -1;
}
glfwMakeContextCurrent(window);
```

```
if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
{
    std::cout << "Failed to initialize GLAD" << std::endl;
    return -1;
}
```

```
while (!glfwWindowShouldClose(window))
{
    glfwPollEvents();
}
```

How to start with OpenGL

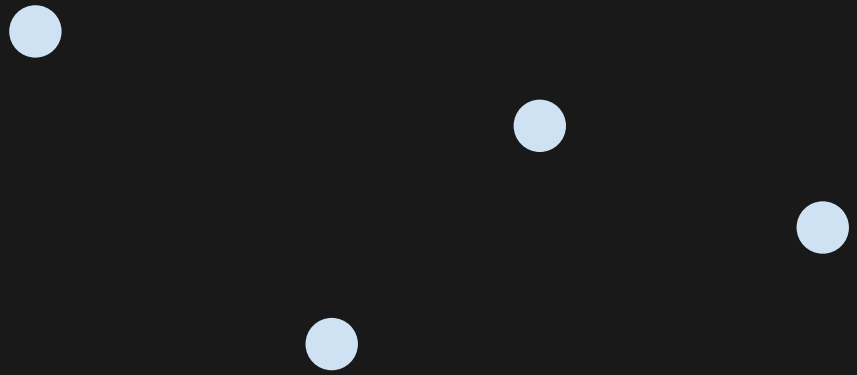
1. ~~Make a window~~

2. ~~Create an OpenGL context~~

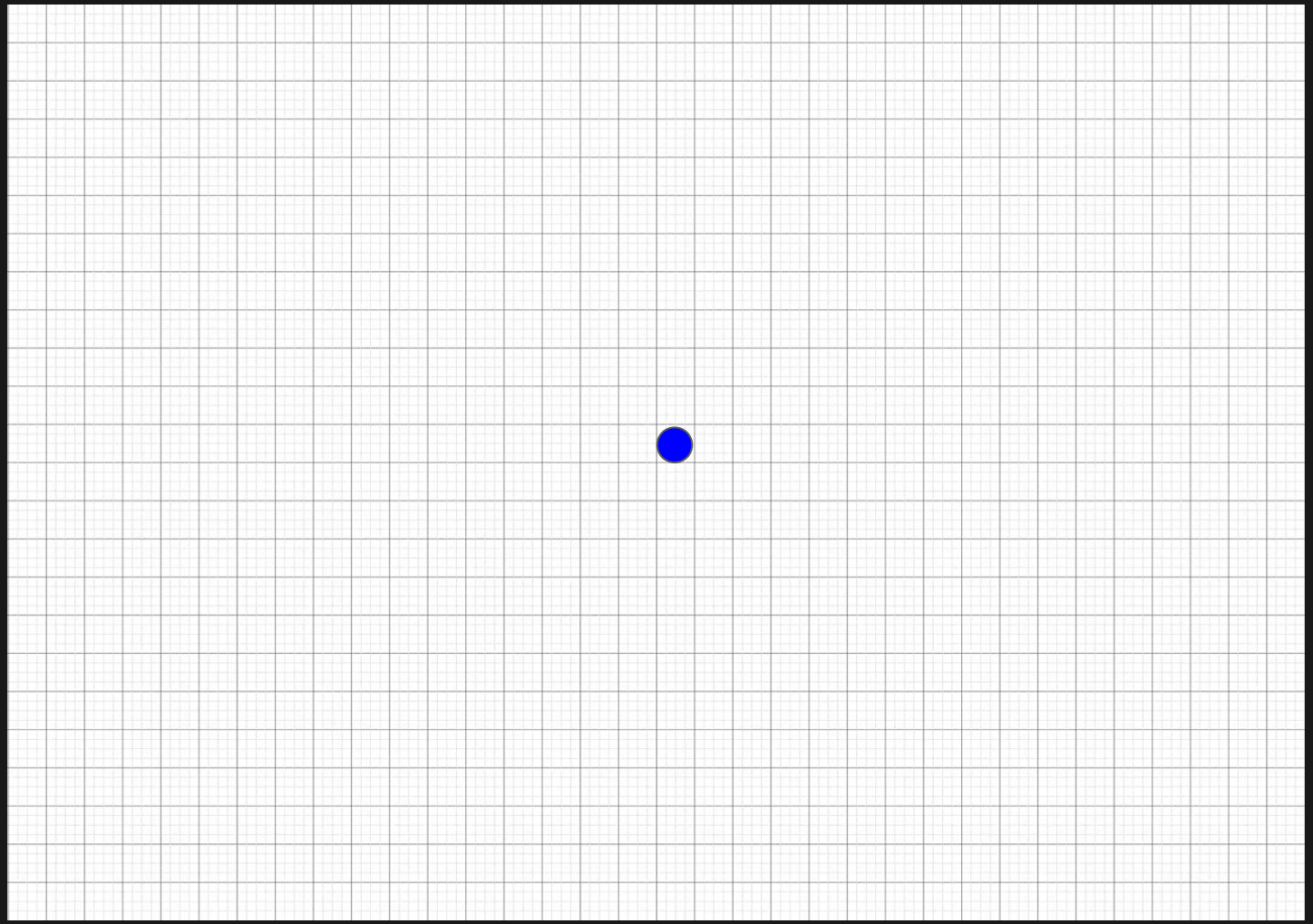
3. Start doing cool stuff

Start doing cool stuff

Drawing points




BIC crystal fine
BIC crystal fine
BIC crystal fine
BIC crystal fine



Drawing points

1. Somewhere to draw
2. Something to draw
3. Something to draw with

Drawing points

1. Somewhere to draw  Window
2. Something to draw
3. Something to draw with

Drawing points

1. Somewhere to draw → Window
2. Something to draw → Draw points
3. Something to draw with

Drawing points

1. Somewhere to draw → Window
2. Something to draw → Draw points
3. Something to draw with → ???

Draw call

glDrawArrays

glDrawArrays — render primitives from array data

OpenGL 4
OpenGL 3
OpenGL 2
OpenGL ES 3
OpenGL ES 2

C Specification

```
void glDrawArrays(GLenum mode,
                  GLint first,
                  GLsizei count);
```

https://docs.gl/gl4/glDrawArrays

Draw call

glDrawArrays

glDrawArrays — render primitives from arra

C Specification

```
void glDrawArrays(GLenum mode,
                  GLint first,
                  GLsizei count);
```

OpenGL 4

Parameters

mode

Specifies what kind of primitives to render. Symbolic constants GL_POINTS, GL_LINE_STRIP, GL_LINE_LOOP, GL_LINES, GL_LINE_STRIP_ADJACENCY, GL_LINES_ADJACENCY, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_TRIANGLES, GL_TRIANGLE_STRIP_ADJACENCY, GL_TRIANGLES_ADJACENCY and GL_PATCHES are accepted.

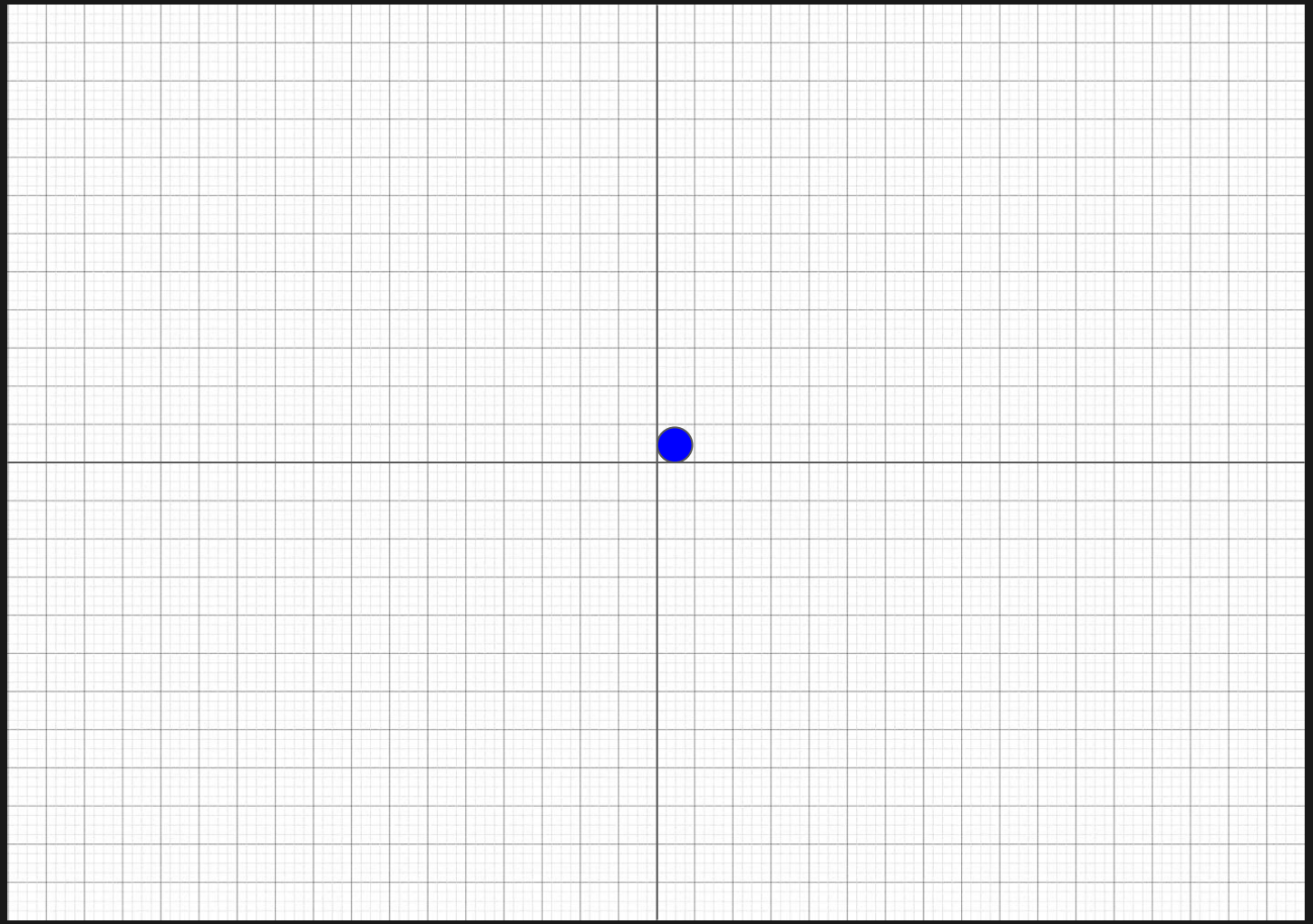
first

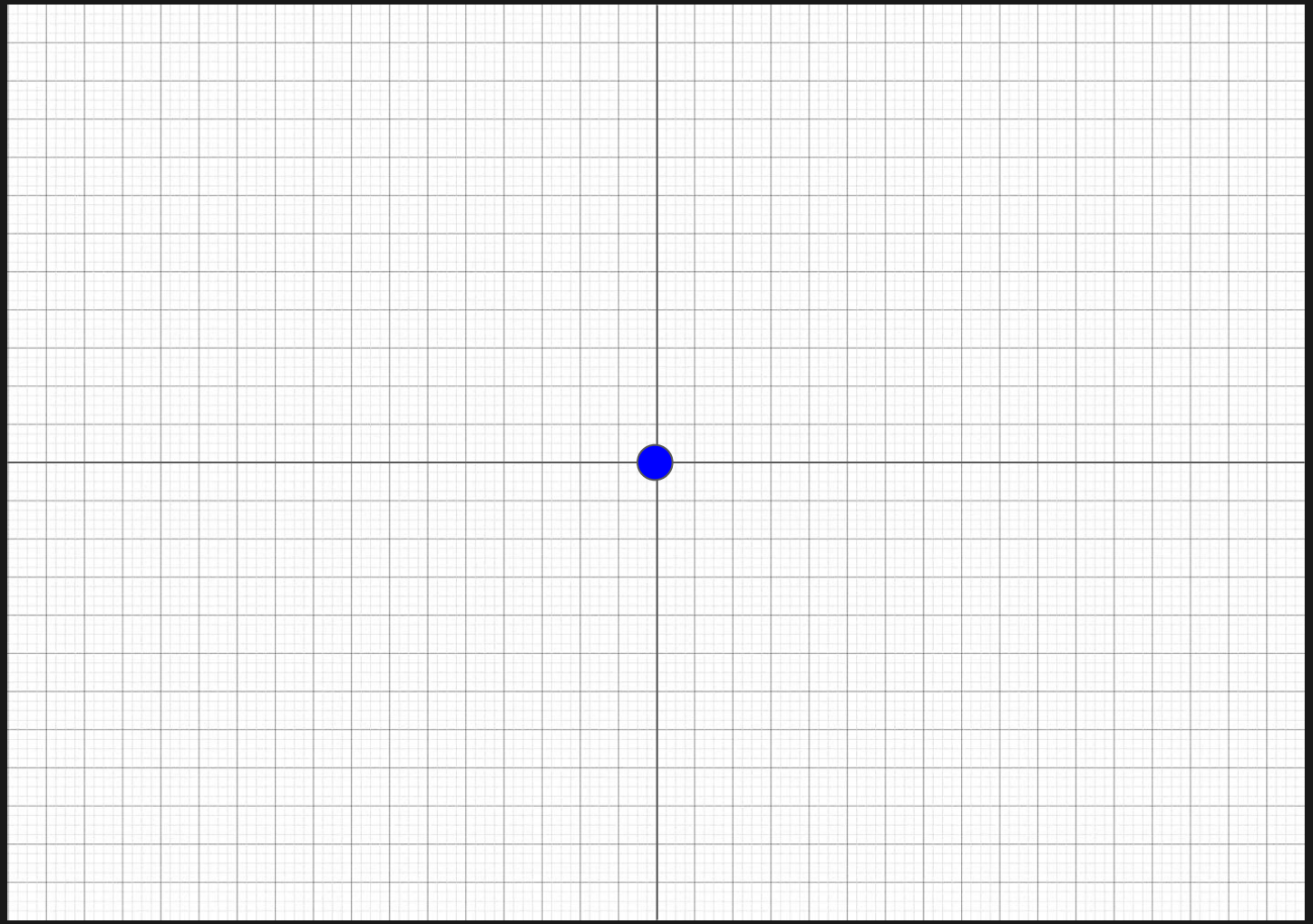
Specifies the starting index in the enabled arrays.

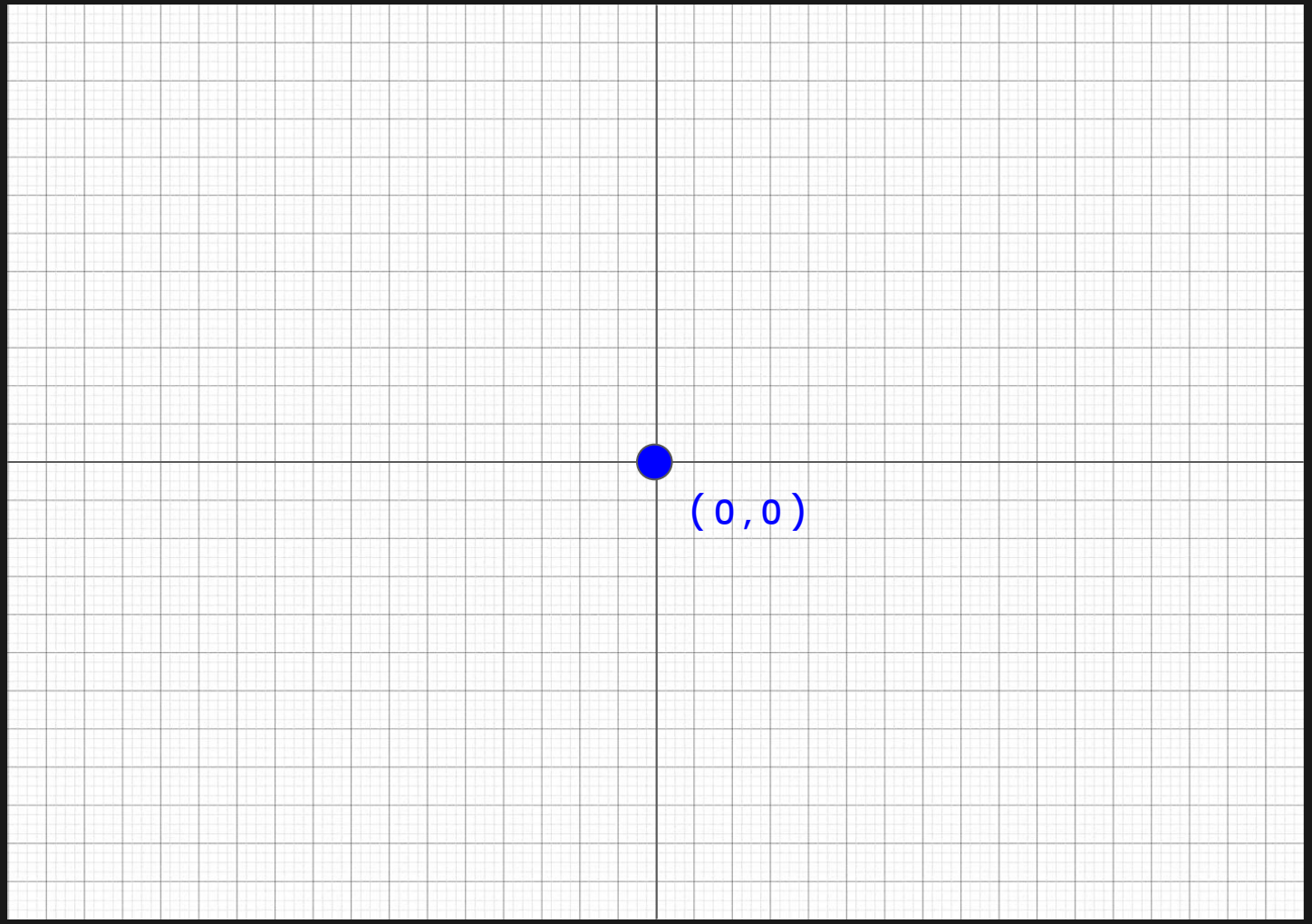
count

Specifies the number of indices to be rendered.

https://docs.gl/gl4/glDrawArrays



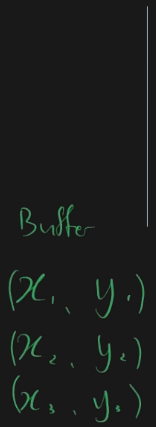



(0,0)

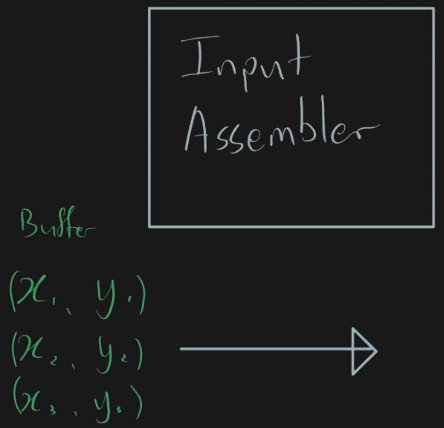
Let’s take a break from coding

The graphics pipeline

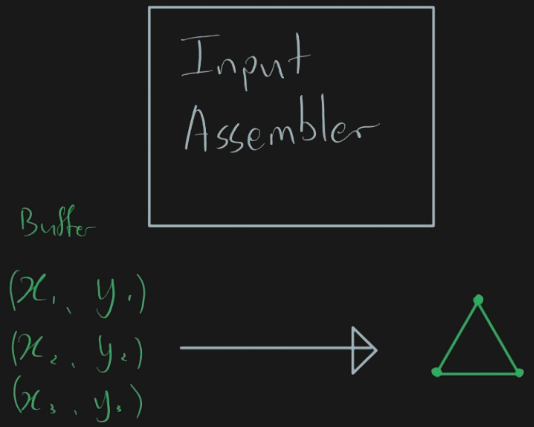
The graphics pipeline



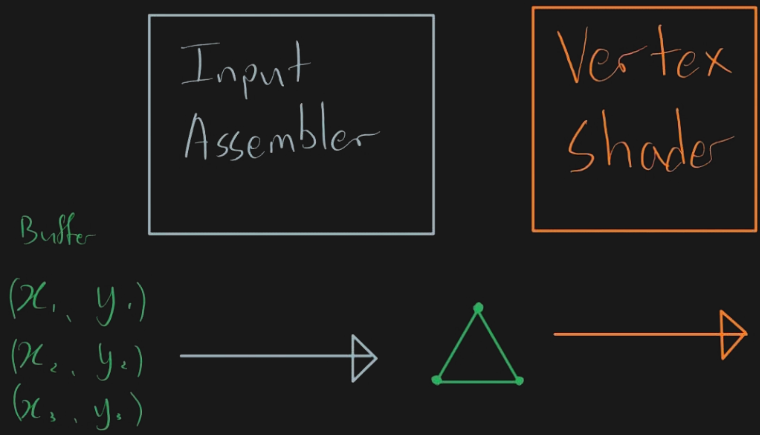
The graphics pipeline



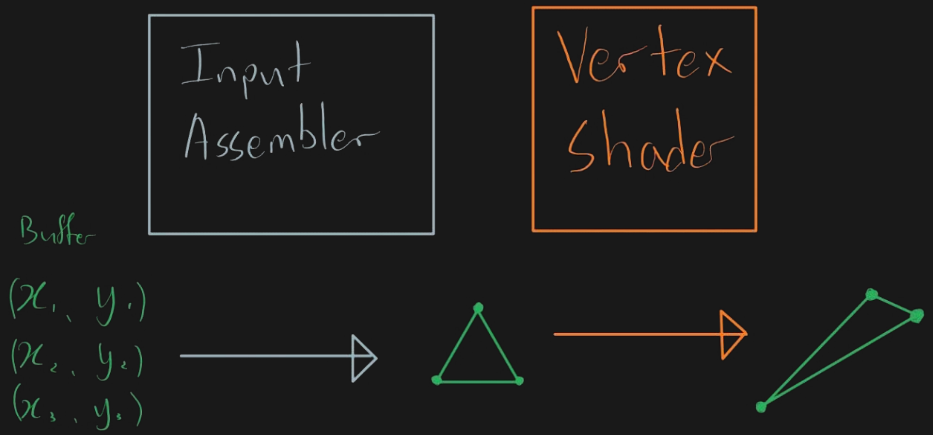
The graphics pipeline



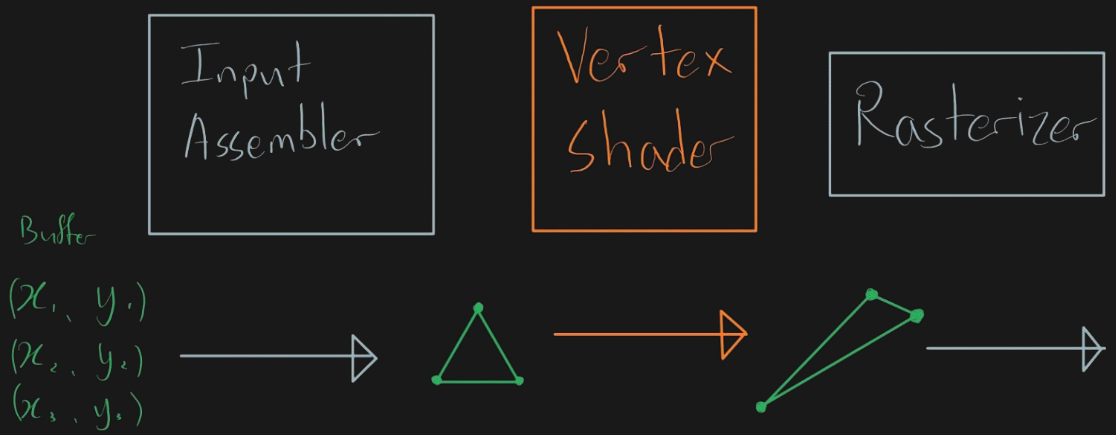
The graphics pipeline



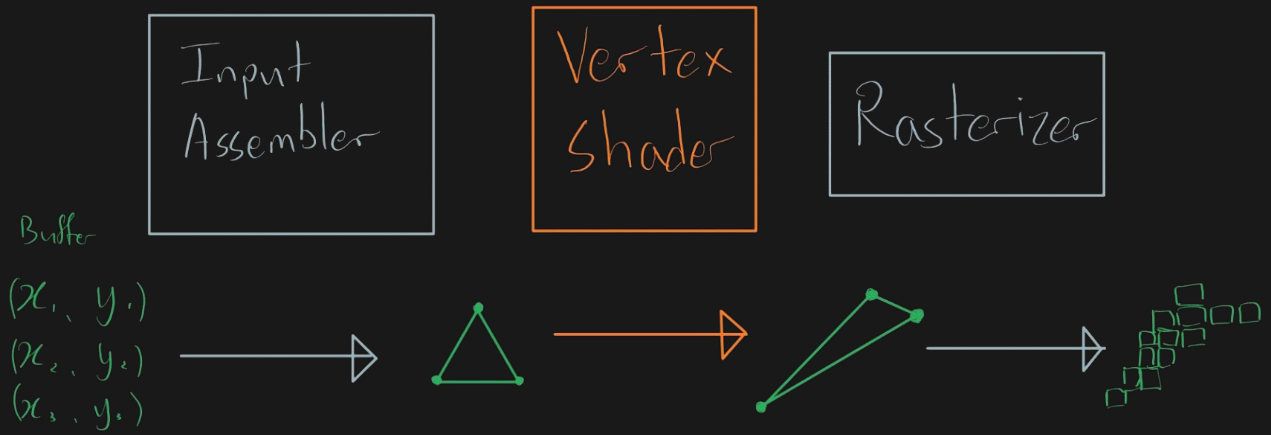
The graphics pipeline



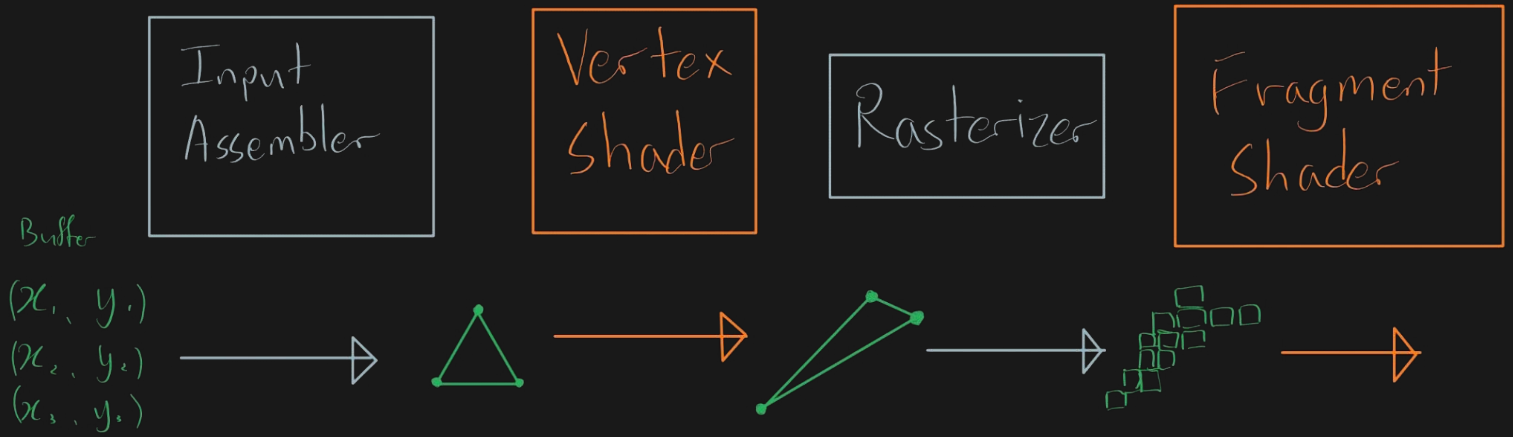
The graphics pipeline



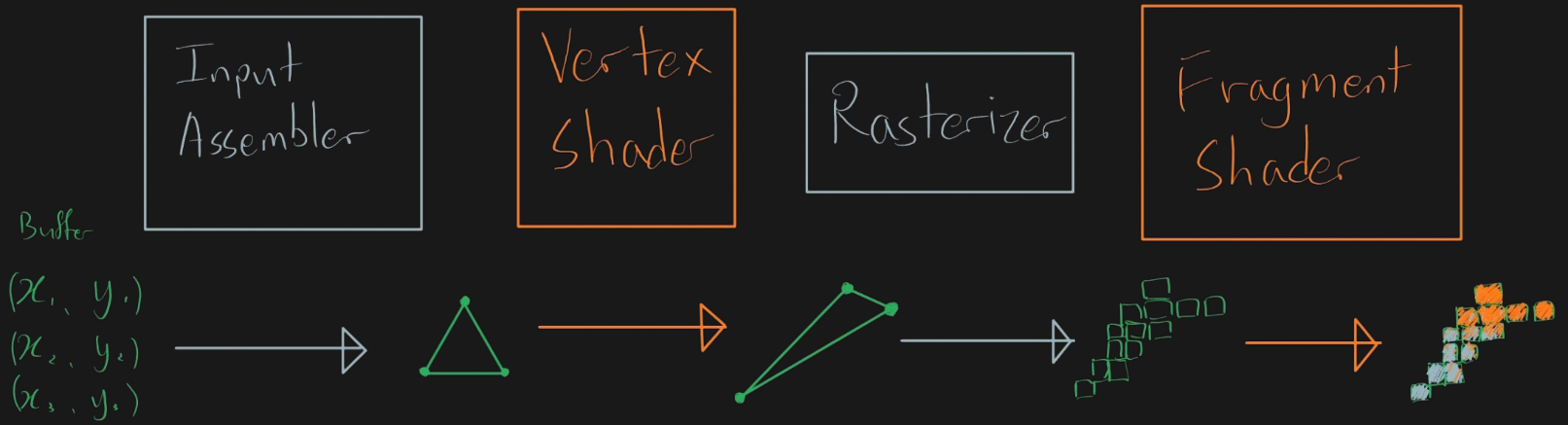
The graphics pipeline



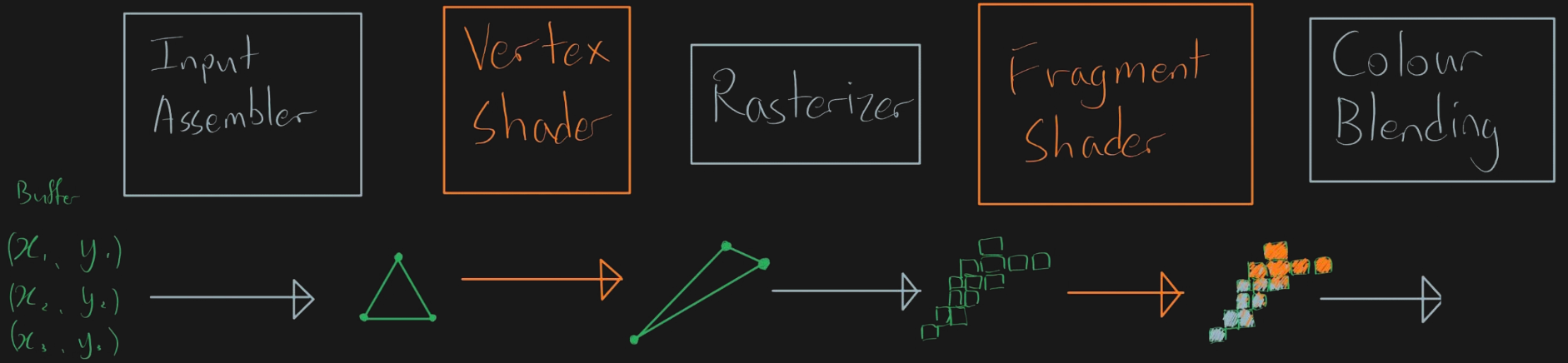
The graphics pipeline



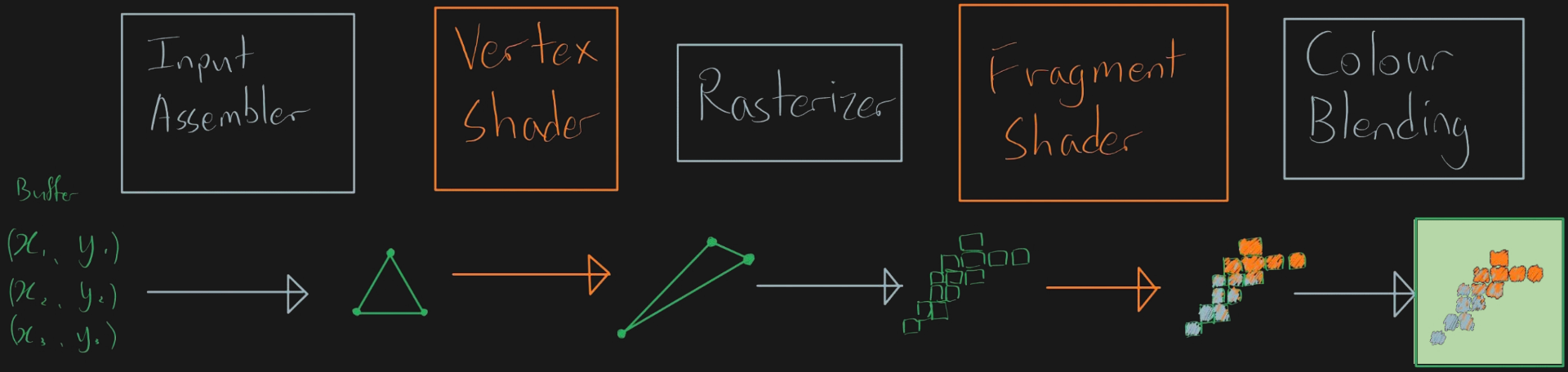
The graphics pipeline



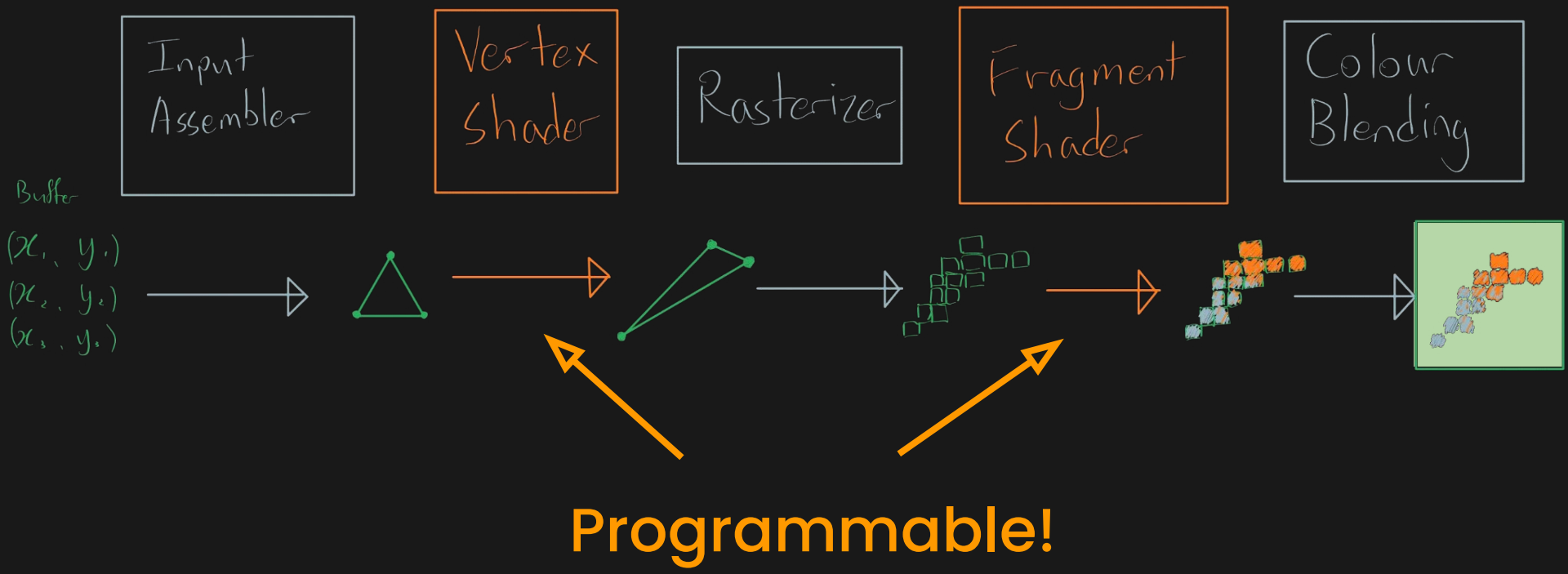
The graphics pipeline



The graphics pipeline



The graphics pipeline



Shaders

Shaders

GLSL

HLSL

CG

Shaders

GLSL

HLSL

CG

Shaders

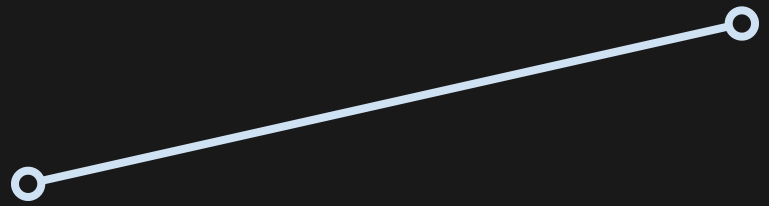
- Vertex Shader

Executed once per vertex

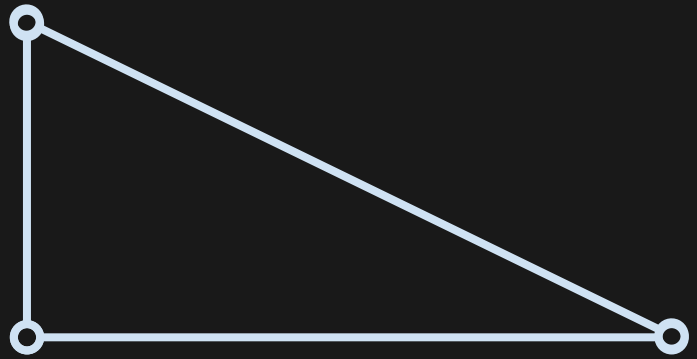
- Fragment Shader

Executed one per fragment

Drawing lines



Drawing triangles



Buffers

Buffers

Buffer Objects

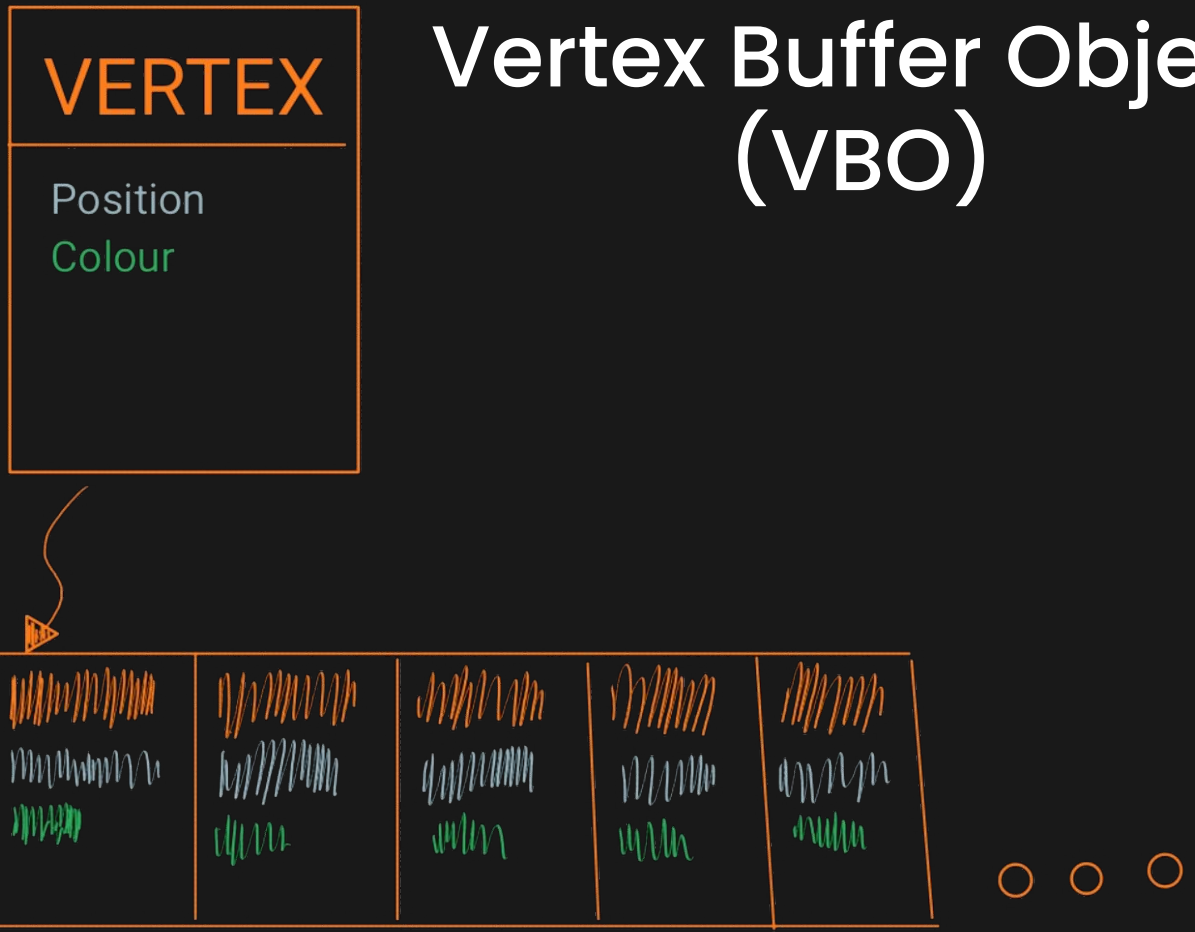
Buffer Objects

- Vertex Buffer object
- Pixel Buffer object
- Element Buffer object
- Transform feedback buffer ...
- . . .

Vertex Buffer Object (VBO)

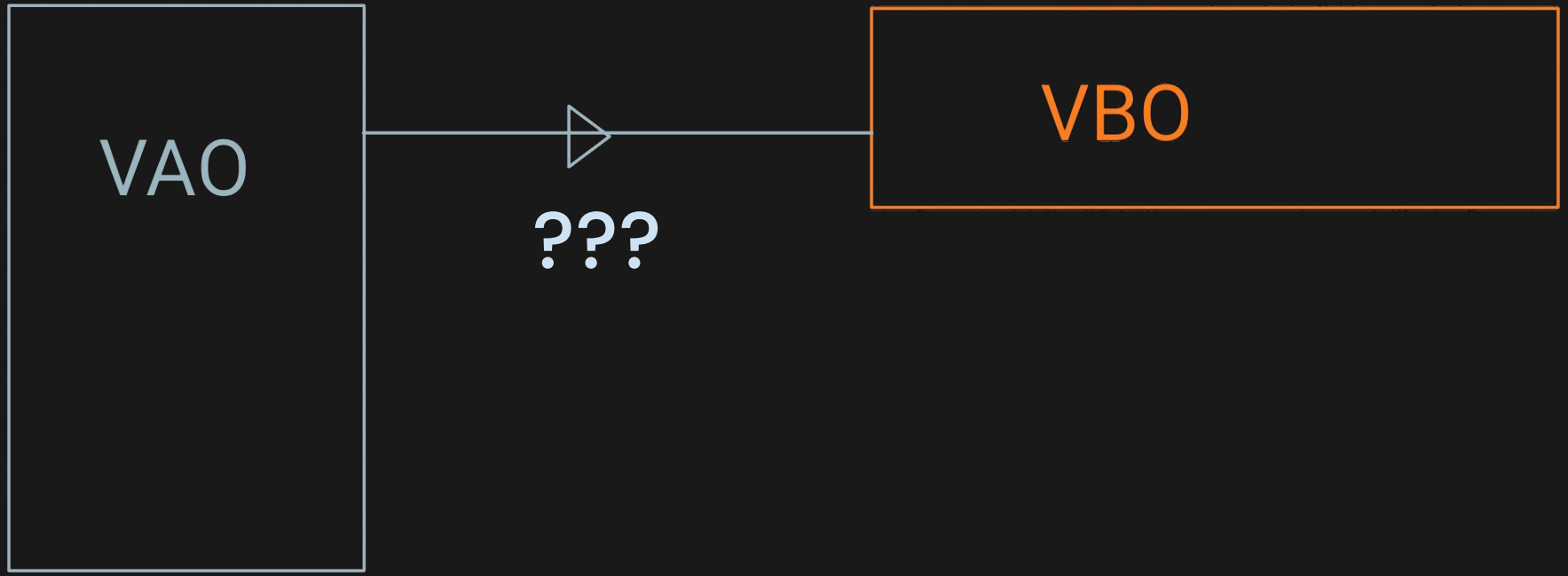
Vertex Buffer Object (VBO)

Vertex Buffer Object (VBO)

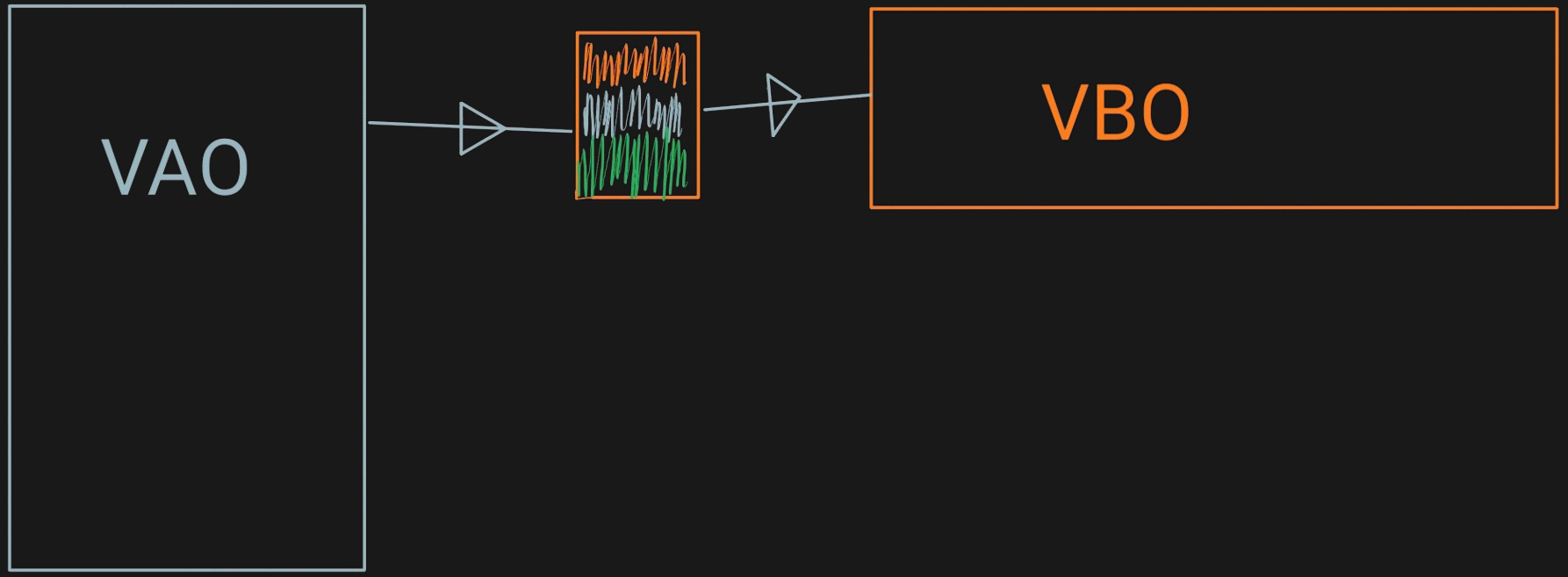


Vertex Array Object (VAO)

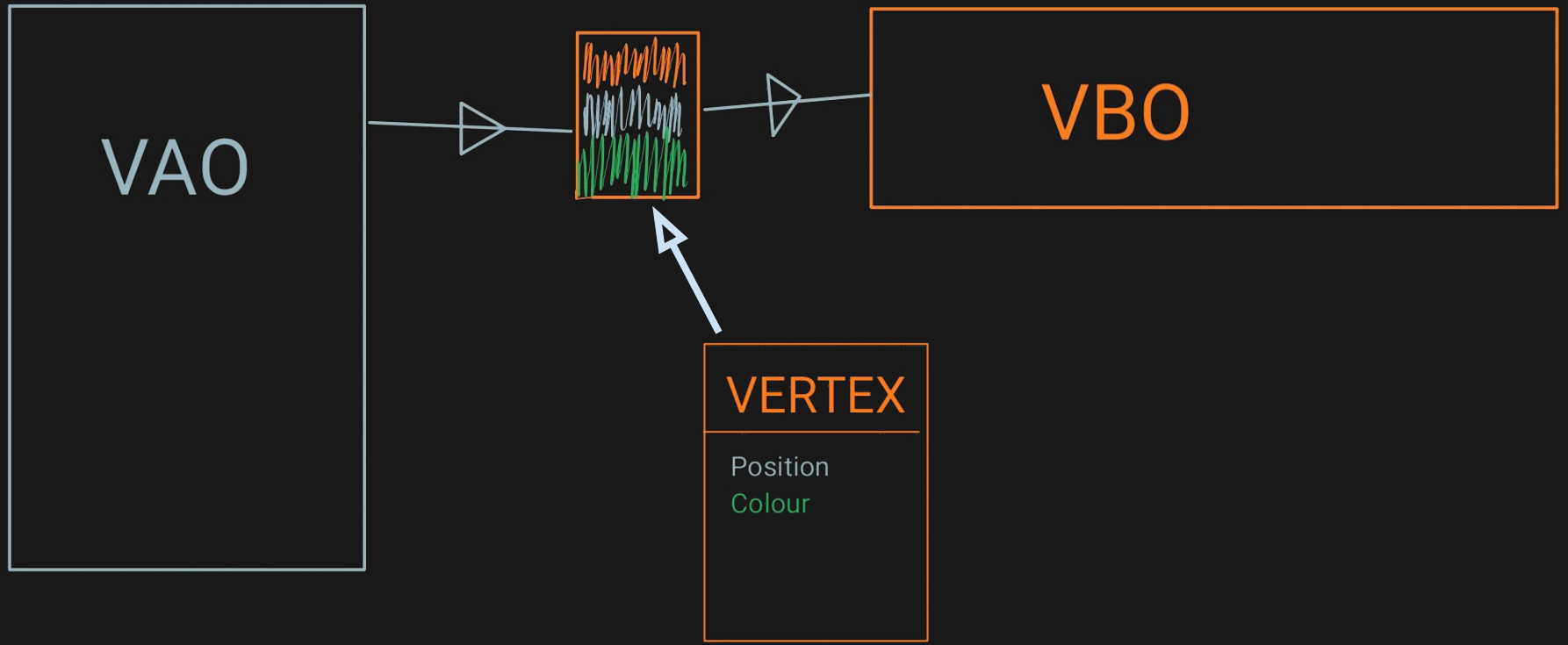
Vertex Array Object (VAO)



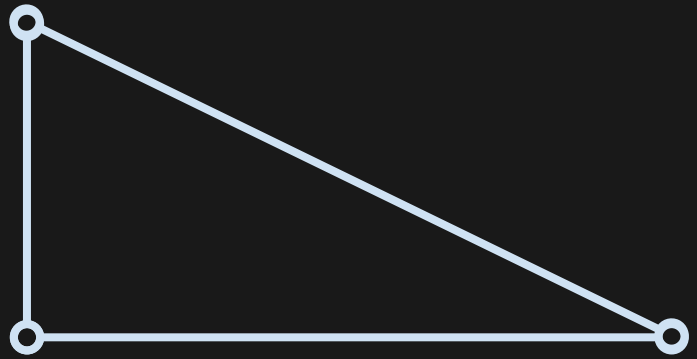
Vertex Array Object (VAO)



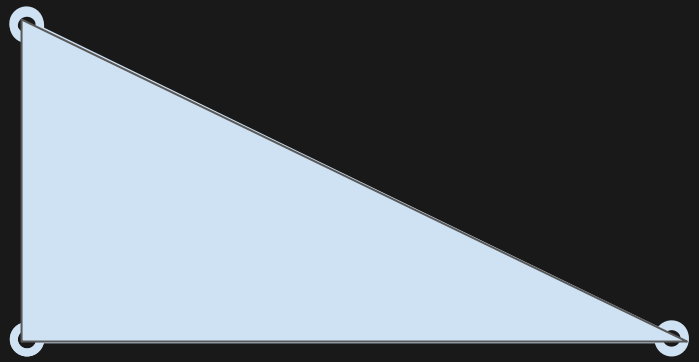
Vertex Array Object (VAO)



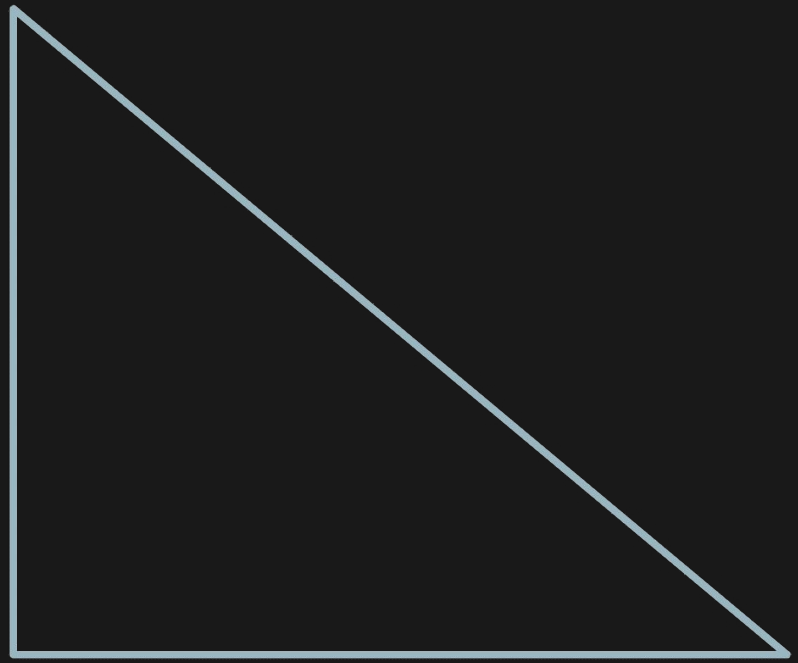
Drawing triangles



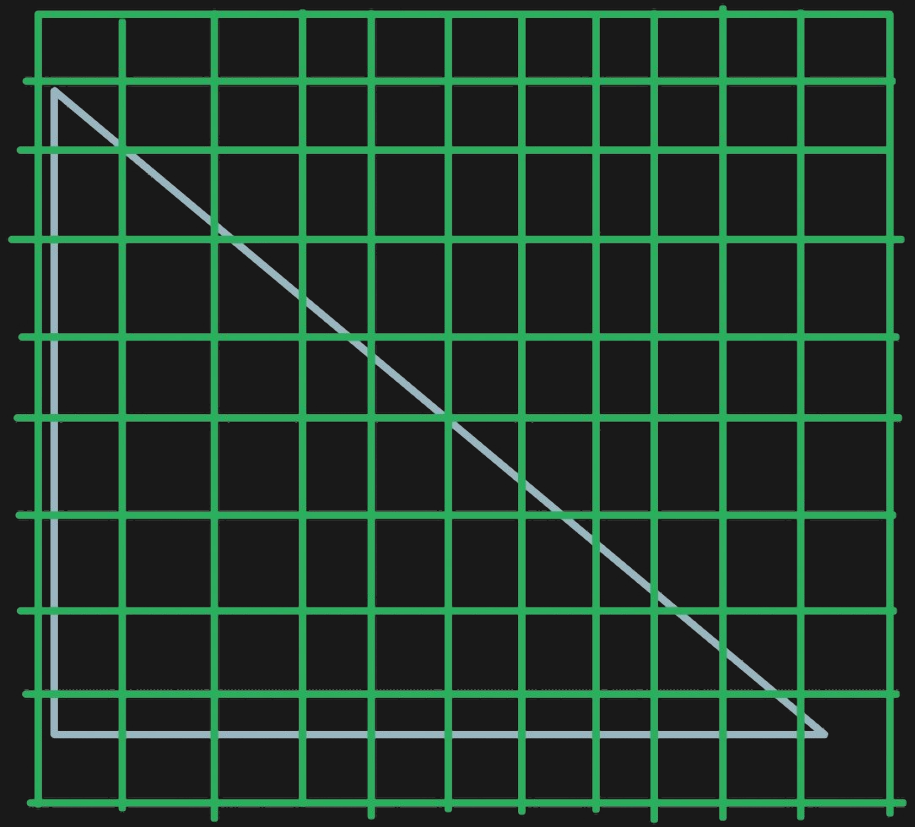
Drawing triangles



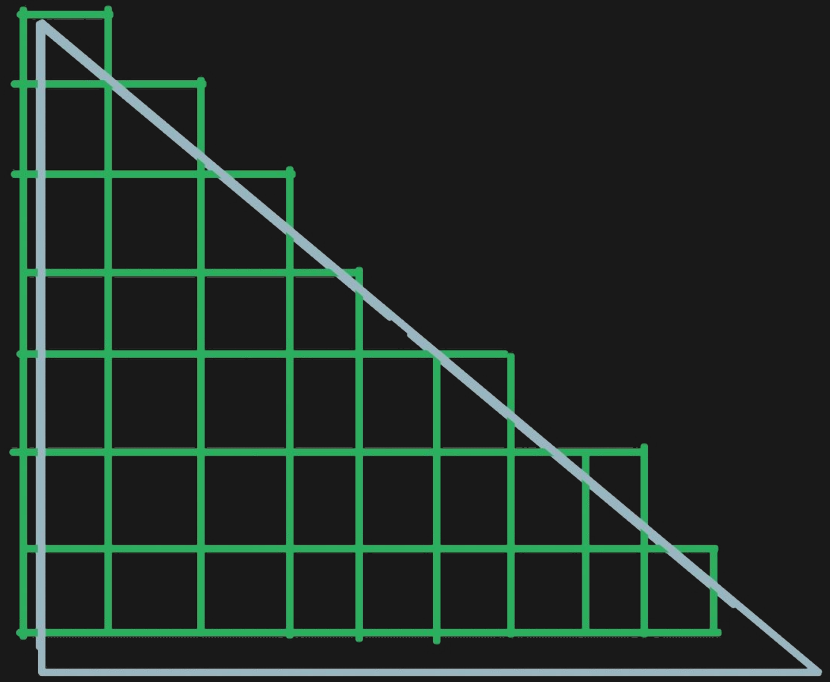
Rasterizer



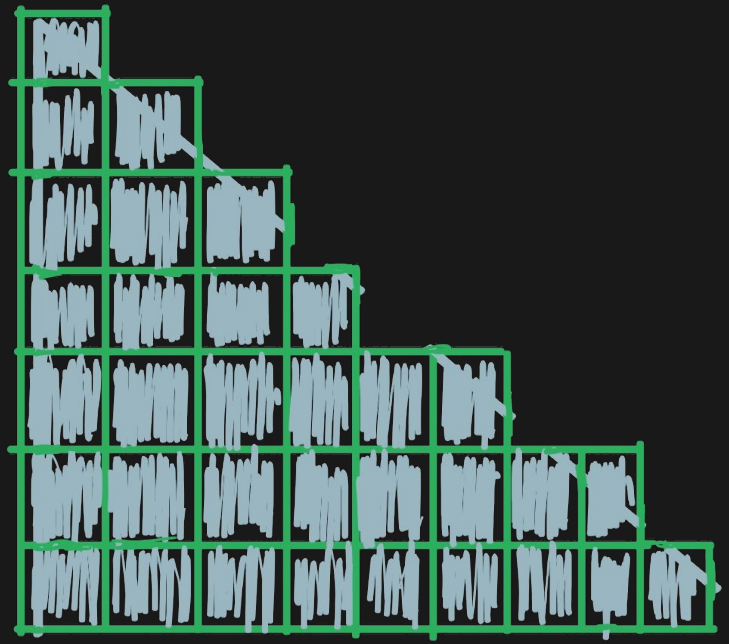
Rasterizer



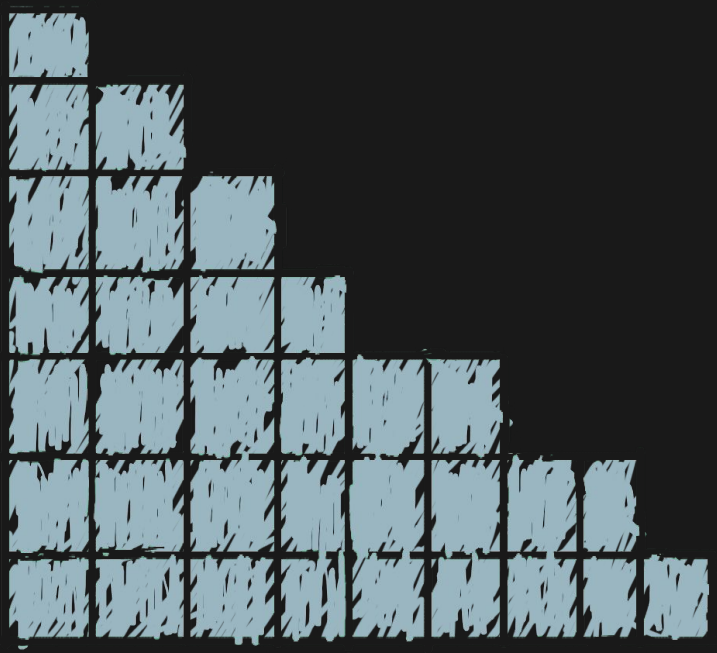
Rasterizer



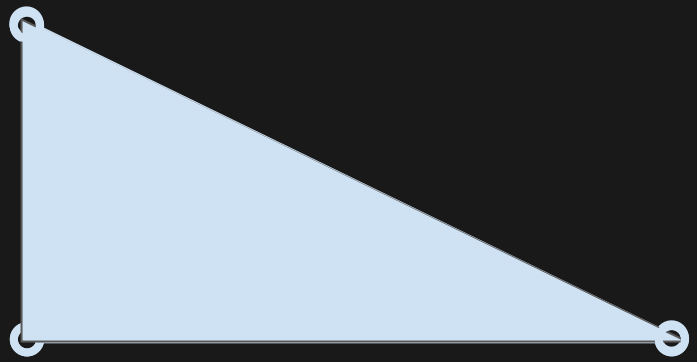
Rasterizer



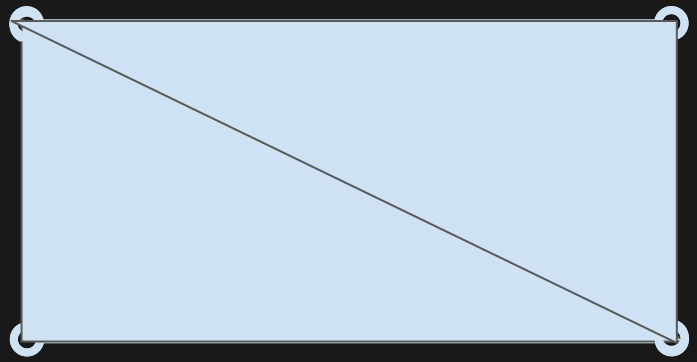
Rasterizer



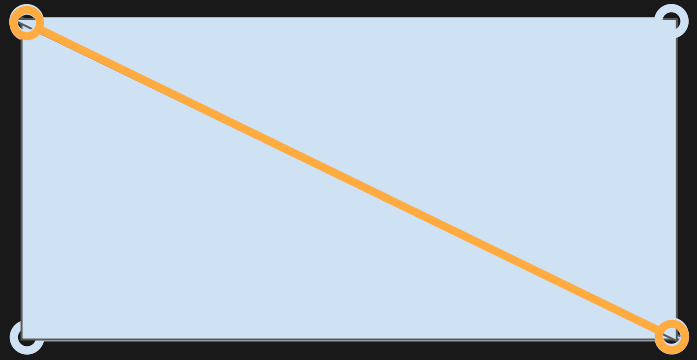
Drawing quad



Drawing quad

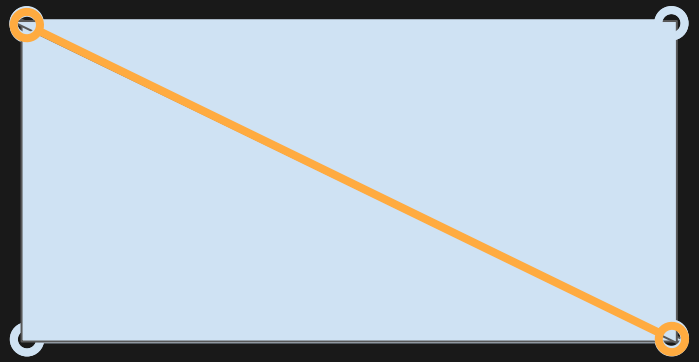


Drawing quad



Drawing quad

Shared!



Element Buffer Object (EBO)

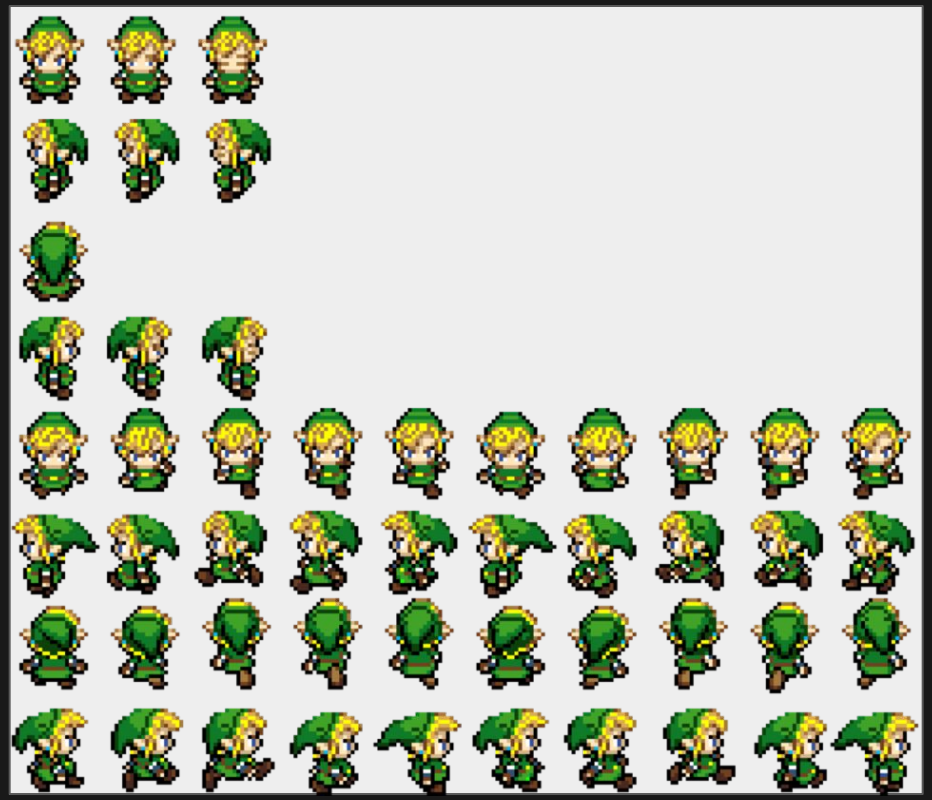
Images

Images

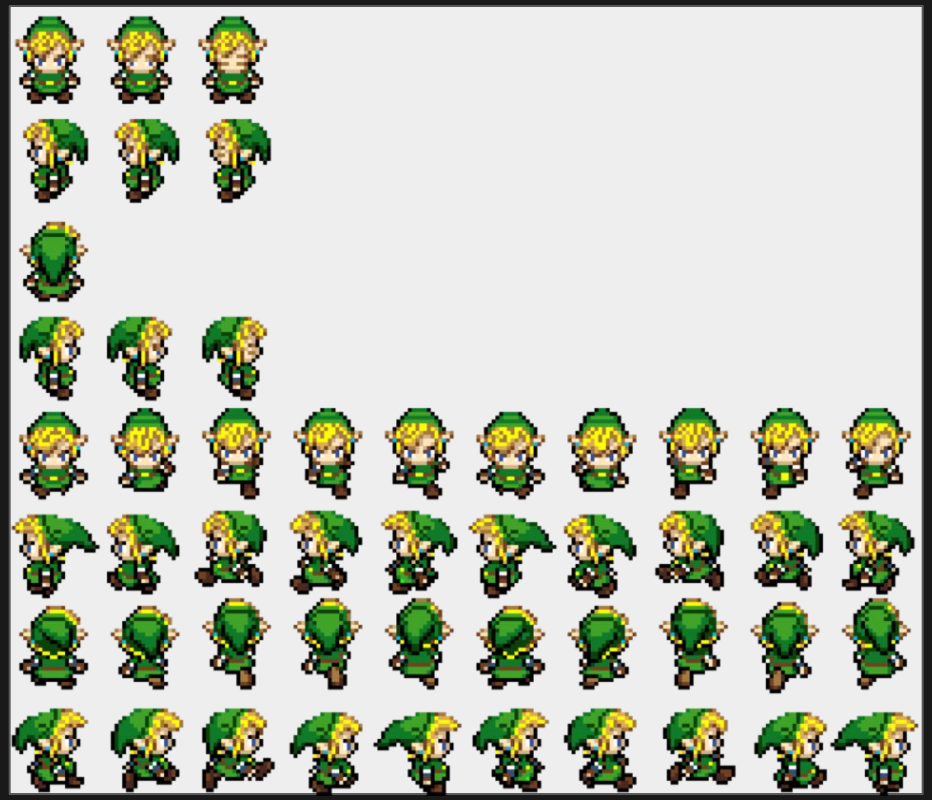
Images

Spritesheets

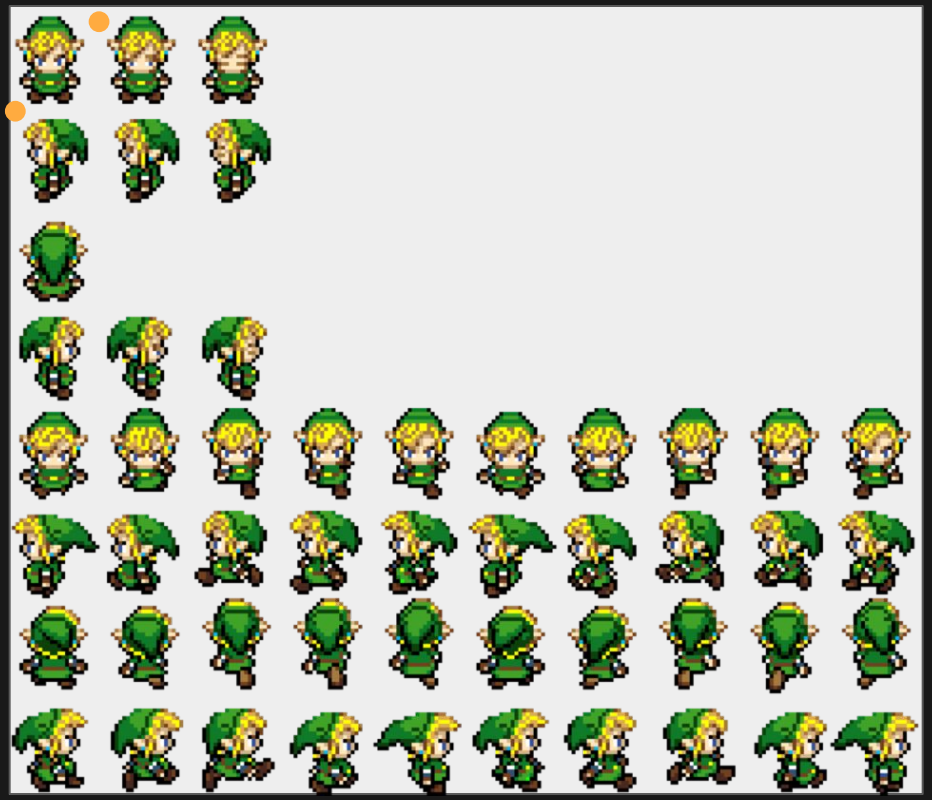
Spritesheets



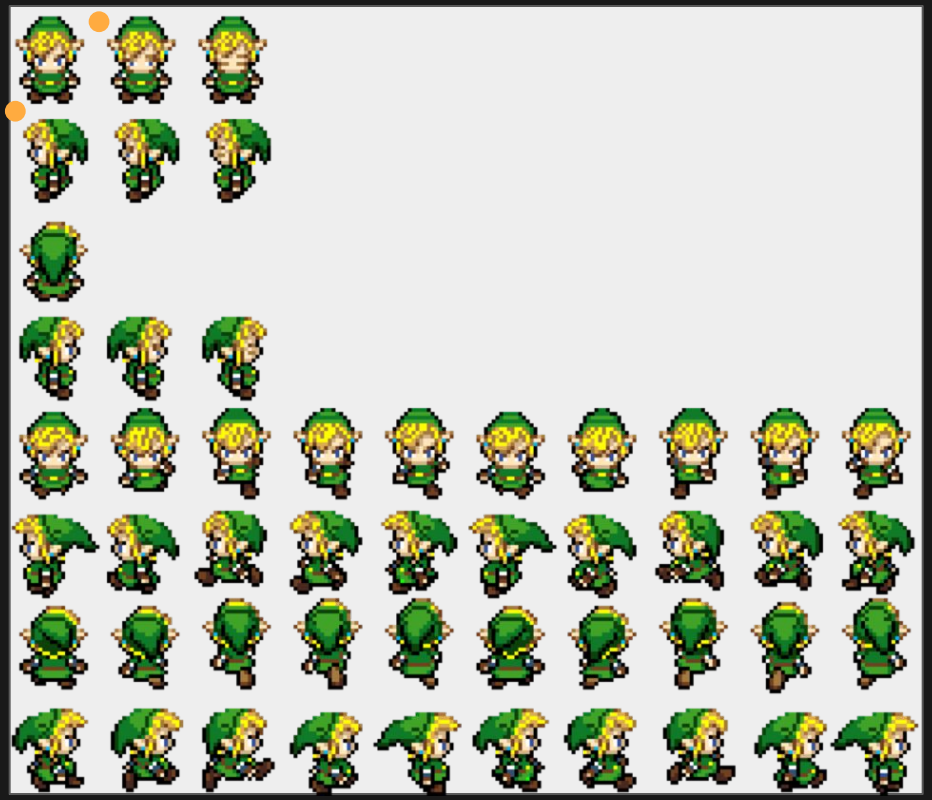
Spritesheets



Spritesheets



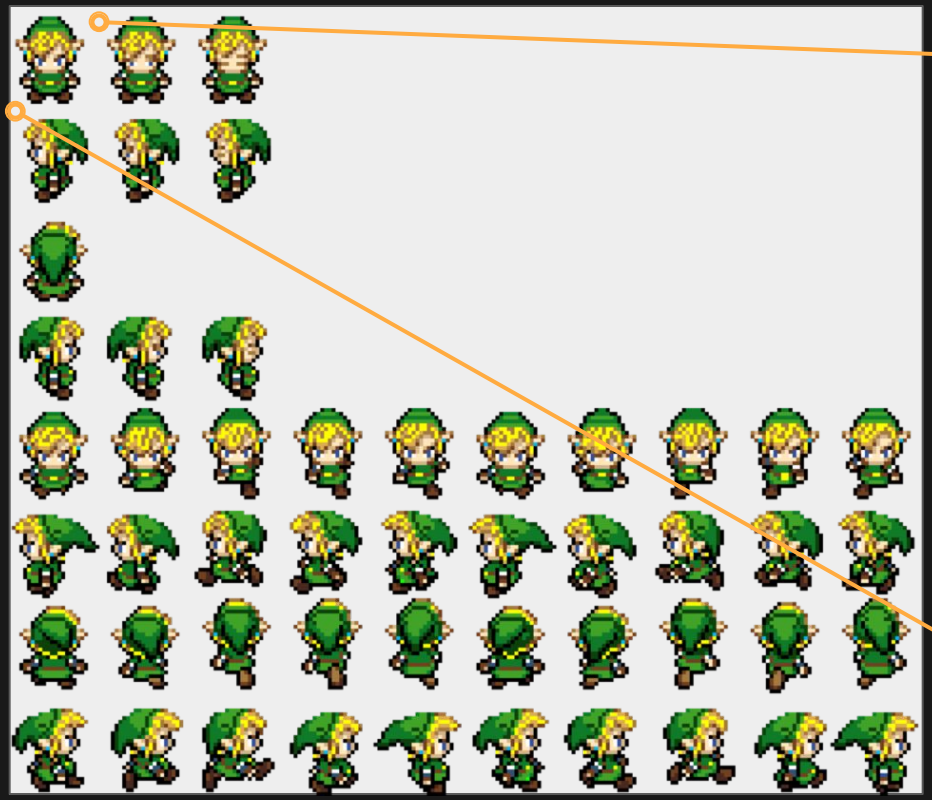
Spritesheets



(?? , ??)

(?? , ??)

Spritesheets



(0.1 , 1.0)

(0.0 , 0.875)

Uniforms

Let’s talk about cameras

Let’s draw a scene

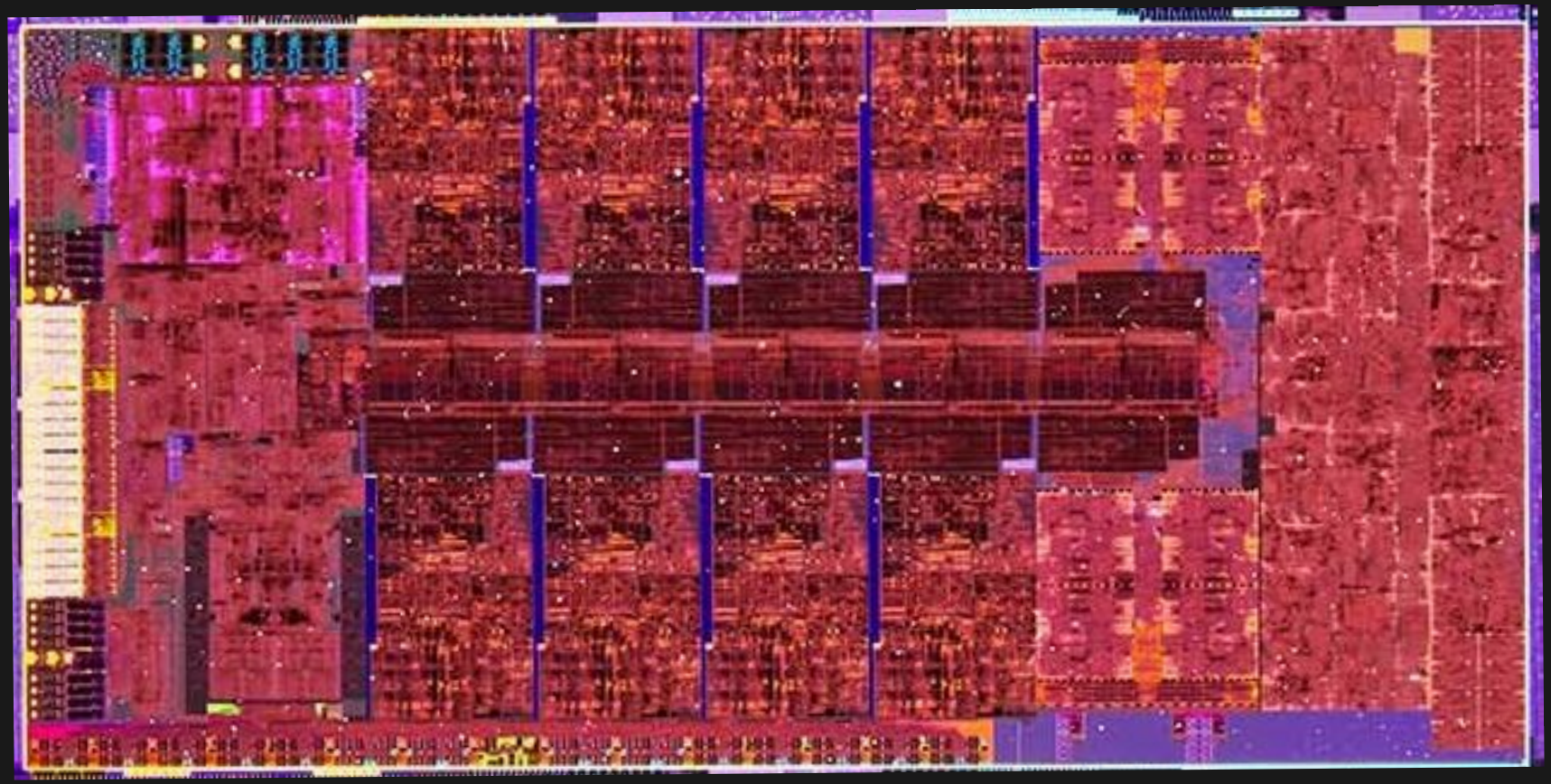
Driver Overhead

We’re done

Batch rendering

Instancing

Indirect Rendering

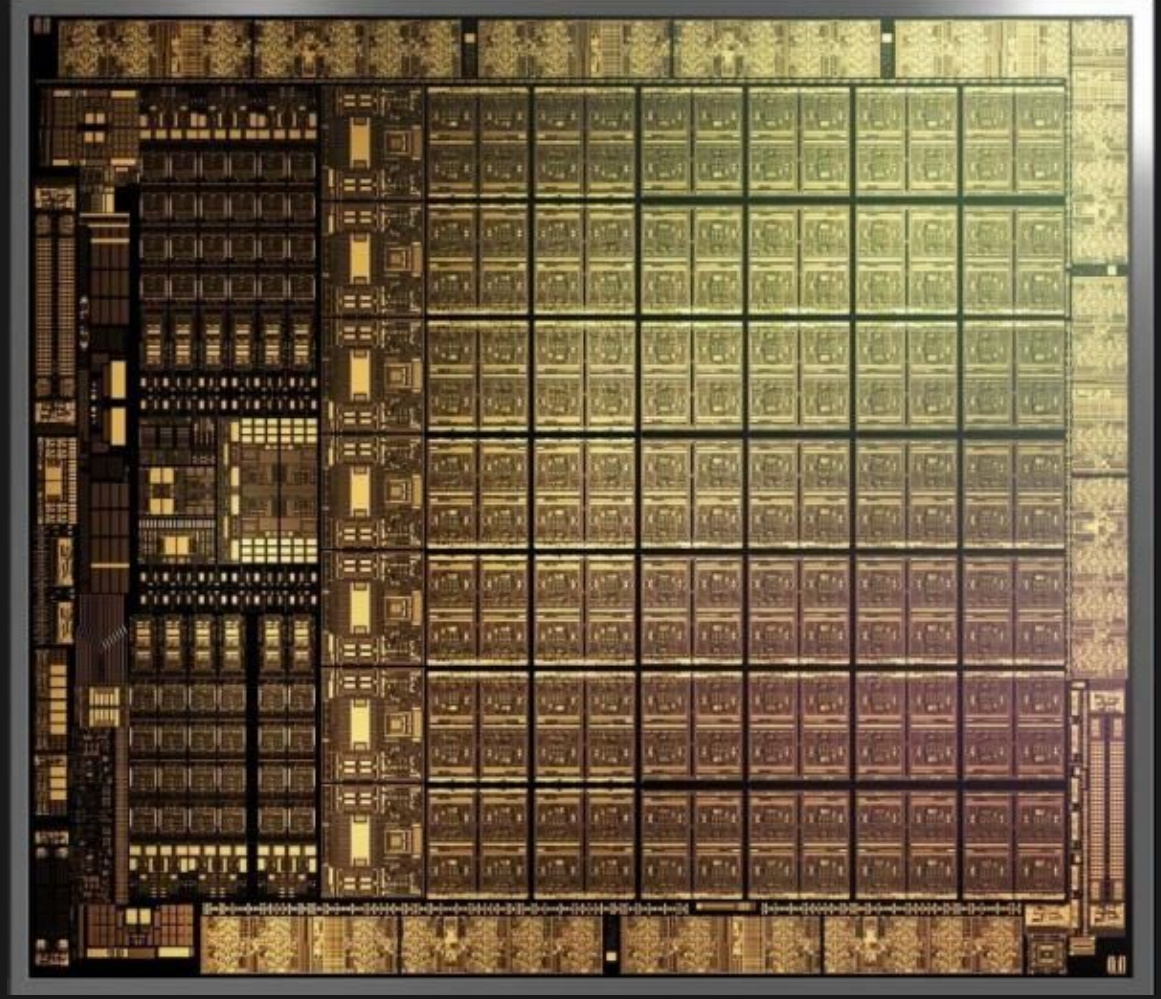


10nm ESF=Intel 7 Alder Lake die shot (~209mm²) from Intel: https://www.intel.com/content/www/us/en/newsroom/news/12th-gen-core-processors.html

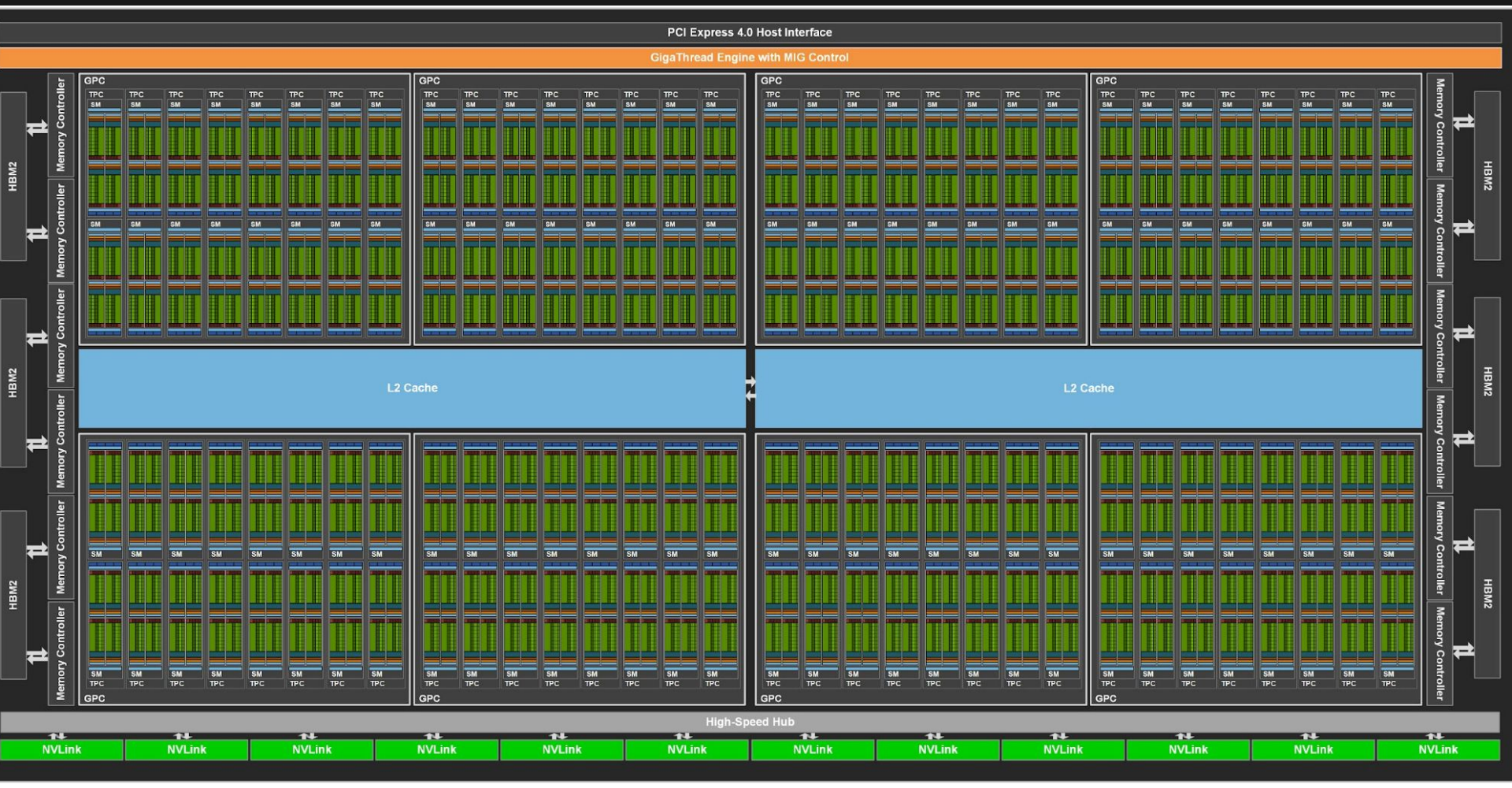
Die shot interpretation by Locuza, October 2021

Credit : https://twitter.com/Locuza_/status/1454152714930331652

SIMD



Source: https://www.pctestbench.com/zotac-rtx-3090-trinity-24gb-gpu-review/2/



Source: nextplatform.com/2020/05/28/diving-deep-into-the-nvidia-ampere-gpu-architecture/

If you forget everything Remember this

Game engines are hard