OpenGL 4.6

Tutorial For Beginners

C/C++ Fundamentals Needed

yunus yıldız

2025

# OpenGL (Open Graphics Library)

* Platform-Independent Renderer API (Application Program Interface)
  + In other words: Specification Standart Library.
* **What is Library?**
  + Library (LIB) contains the definition.
    - Example: "For version n, define function ‘x’"
    - Sometimes LIB used as “Static Link Library”, which contains the implementation too.
  + Dynamic Link Library (DLL) contains the implementation (code).
    - Example: “For version n, implement function ‘x’ code”
  + LIB file gets mixed with a DLL with a help of a Linker.
  + A Linker just links a definition with implementations and gives you an executable at the end of all linking process.
* **What is Specification Standart?**
  + Definition and implementations have rules.
    - Example: "In version n, the result of function ‘x’ should equal to y"
* **If OpenGL is a Library, where is the actual code?**
  + Inside the GPU (Graphic Processing Unit) Driver you just installed.
  + The GPU Driver decides which feature and version to support, data management, so everything about the datas and code.
* **How to access the code?**
  + Every OS (Operating System) has its own unstandardized system for file, window, input, event, etc...
  + So the connection to GPU Driver OpenGL code will differ.
    - Your code <-> OS <-> GPU Driver OpenGL code
  + To keep that connection ease, every OS standardized the connection path by creating their own library so the methods of it will differ.
    - Your code <-> Library (Connection path) <-> OS <-> GPU Driver OpenGL code
  + Because OpenGL is a Library, must be loaded at runtime.
  + If connection path wasnt standardized and if we didnt use any runtime loading, every GPU Driver OpenGL Code could been unflexible because it would have been required to use the LIB & DLL.
* **How to draw stuff?**
  + The GPU Driver manages the VRAM data.
  + You as a developer, will use identifier (handle) to manage or use the VRAM data.
  + Using the the VRAM data, you will draw everything manually.
* **What is VRAM data?**
  + CPU data will stay on RAM (Random Access Memory).
    - Example: Identifiers.
  + GPU data will stay on VRAM (Video Random Access Memory).
    - Example: Identifier data.
* **How to manage VRAM data if i am allowed to use identifiers only?**
  + When you request deletion of an identifier, that means you request the deletion of data inside VRAM.
* **Should OpenGL version be cared?**
  + Yes, use the latest as much as possible.
  + OpenGL <= 1.x The steps to generate Image was using fixed functions (non-changeable) only.
  + OpenGL >= 2.x Render steps started to use programmable functions.
  + OpenGL >= 3.x Render steps slowly getting rid of fixed functions.
* **What is ‘OpenGL ES’?**
  + OpenGL: Windows, Linux, MacOS, ...
  + OpenGL ES (Embedded Systems): Android, IOS, Web (WebGL), ...
* **What is the difference from other API’s like DirectX?**
  + Similiar with other API’s.

# Extensions

* OpenGL functions are called as ‘extensions’.
* The big companies has developed their own extensions for OpenGL too.
* **How to seperate other extensions from standard OpenGL?**
  + All extensions have prefixes.
    - OpenGL
      * GL\_
    - Vendor (Specific)
      * GL\_NV for Nvidia
      * GL\_APPLE for Macintoshe
      * ...
    - Generic (General Purpose)
      * GL\_EXT
    - Architecture Review Board (ARB) (Approved)
      * GL\_ARB
        + Will be there for now and later versions.

# Understand OpenGL

* Oftenly, you will hear "Works with states so its a state machine".
* But lets look imagine how exactly it works.
* **Imaginary workshop**
  + Imagine a workshop (context/container).
  + While working in this workshop, you have tools (objects/datas) that you can use.
  + Each tool has a number (identifier).
  + Before performing an operation, you must grab (bind/use) the tool into your hand (target).
  + When you're done with your operation, you can either put somewhere, (unbind/unuse) trash it, (delete/trash) or use another tool.

# OpenGL Context

* Container of datas for which OpenGL runs.
  + Example: Used by which GPU, using which OpenGL version, decide which window to draw, store object data and manage access, etc...
* Stored at VRAM.
* You will draw everything to OpenGL Context.
* Without it, OpenGL functions wont work.
* **How OpenGL Context works?**
  + Associated with 1 thread.
  + Only 1 can be active per thread in a process/app.
    - Search about thread and processes if you dont know them.
  + You will just select which one will be active at the current thread.
* **How to get the current thread and draw to its OpenGL Context?**
  + Every thread has an entry function (entrypoint).
  + If you use OpenGL functions in the entry function, it will automatically detect the current thread and its OpenGL Context.
* **Why just not use single OpenGL Context** **for everything?**
  + Runing older versions.
  + Multi-threaded drawing.
  + Sharing data between OpenGL Context’s while keeping data private in a process.

# OpenGL Object

* Data inside OpenGL Context.
* If there is an OpenGL Object, there is an identifier.
* We will inject the data using the identifier.
* **How to create an identifier?**
  + Using loaded functions of OpenGL, we request an identifier for somewhere in VRAM.
  + Even tho we have the identifier, there is no data inside of it yet.
* **How to inject data to the identifier?**
  + The VRAM is protected, you shouldnt access it.
  + So we bind the identifier to a Target to inject the data.
    - Imagine a tool in your hand. (bind to target)
    - Use your mind to create the tool. (inject to target)
* **What does ‘binding’ means?**
  + Giving reference to an OpenGL Object by using an identifier.
* **What is Target?**
  + Single slot for 1 OpenGL Object.
    - Example: GL\_ARRAY\_BUFFER
  + There may be thousands of OpenGL Objects but you are capable of using one at a time.
* **What is Buffer?**
  + Temporary data.

# Prepare OpenGL Context

* To draw something, we need an OpenGL Context.
* Since the GPU Driver knows everything and you dont, you will request Driver’s code to create an OpenGL Context.
* **How can i access Driver code and create an OpenGL Context?**
  + By using a function from the OS’s library.
  + If we want to abstract the creation, we use some kind of window and input management library that works with OpenGL.
    - You must use these libraries for education only, because they are covering only fundamental things you need.
      * FreeGlut (USING)
      * GLFW
* **How FreeGlut works?**
  + Assume we created a window.
  + Window creates its own context and stores an identifier.
  + The window is an identifier too.
  + Now, the events connected to OpenGL Context Object.
    - Context <-> OS<-> GPU
* **Loading Extensions (Functions of OpenGL)**
  + To draw to OpenGL Context, we must load the function pointers from the GPU Driver into our code using the OS’s library.
  + We can load every function manually but its a headache.
  + So if we want to abstract the loading, we use a DLL called 'Extension Loader Library' and blame the driver developers if anything goes wrong.
    - GLEW (USING)
    - GLAD
    - For fun, the driver can have absurd methods inside it's OpenGL code which you can load!!
* **How GLEW works?**
  + Uses the current OpenGL Context to select which functions to load.
  + You will be able to call every OpenGL function but becareful, some of them may not be compitable with your version.
* **Why dont we use OpenGL Context to load functions?**
  + OpenGL Context is just a container of datas and for us; its an identifier, not an interface.
  + Even tho GLEW uses the OpenGL Context (stores OpenGL version) to find which functions to load, technically they are gathering some data only and not using as an interface (connection path).

# Vertex

* A point composed of arbitrary Attributes.
  + Example
    - (1 Vertex)  
      (position, normal, color, dooDOOOO)  
      [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
* **What is Vertex Attribute?**
  + Data.
    - Example: position.
  + There are built-in Vertex Attributes at Shader Object.
  + But for you, completely arbitrary.

# Primitive

* Unconnected raw array of Vertex.
* Has specific count of Vertex.
  + Example: Dot (1), line (2), triangle (3), …
* Only 1 Primitive type can be used to send data for drawing on the screen (OpenGL Context).
* **Why Primitive used to draw on screen while Vertex exists?**
  + Drawing Vertexes on the screen requires a connection between these points.
* **Primitive Struct (Base Primitive)**
  + Data that stores the Vertexes as connected.
    - Example: Calculation of an edge, interior, area or bounds, just one function away.
* **Primitive Identifier**
  + Index of Primitive, NOT Primitive Struct.
    - Example: If there is a Vertex array of triangles: [V0, V0, V0, V1, V1, V1]
      * Primitive Identifier 0 = V0, V0, V0
      * Primitive Identifier 1 = V1, V1, V1
  + You cannot access the Primitive Struct.
* **Types**
  + Triangle
    - 3 Vertex.
    - Fastest because optimized for GPU especially.
      * In the ancient history, was used for rendering stuff.
  + Patch
    - User-defined count of vertex.
    - There is no ‘Patch’ for GPU, it must be divided into other types.

# Color Channel

* Represents a color and precision of that color.
* **Why not called as “Color”?**
  + Channels have bit depth to ensure how precise the color is.
* **What is Bit Depth (Bit Count)?**
  + The Bit Count (value range) of a color channel.
  + Determines the memory usage and precision.
  + Dont forget:
    - 8bit = 1 byte (0-255)
    - 16bit = 2 byte (Who cares?)
    - ...
  + Mostly they use 8 bit for each channel.
  + Sometimes you will see that they sum all of the color channel's bits, that is wrong.
* **Types**
  + The main ones are: (because we can create almost any color we want)
    - 8 Bit Red
    - 8 Bit Green
    - 8 Bit Blue

# Pixel

* A container consisting of a combination of Subpixels.
* **What is Subpixel?**
  + A color of a Color Channel.
  + A combination of Subpixels will give the illusion of the actual color.

# Fragment Struct

* A set of values that is created from a Primitive Struct.
* Candidate to be 1 pixel but if wanted, can be more.

# Coordinates

* Represents a location.
* **Types**
  + Local Space
    - Origin (0,0,0) is (Object center).
  + World Space
    - Origin (0,0,0) is (0, 0, 0).
  + View Space
    - Origin (0, 0, 0) is (Point of view).
  + Clip Space
    - Origin (0, 0, 0, 1) is (Point of view)
    - Perspective applied with 4th dimension. (xyzw)
  + Fractional Coordinate
    - Percentage: Normalized [0,1] floating-point number.
      * Consists from an array of coordinate’s distances.
    - Coordinate: Adding the Percentage to every coordinate in an array, will give a valid coordinate.
  + Normalized Device Coordinate (NDC)
    - Floating-point number.
    - -1 left, 0 center, 1 right.
  + Screen Space Coordinate
    - [X, Y] Integer value.
    - For OpenGL: Origin (0, 0) is (Left-Bottom side of your screen).
* **Conversion**
  + There is a simple multiplication to convert a coordinate to new coordinate while physically keeping in the same coordinate with previous coordinate.
  + **Model Matrix (Local -> World)**
    - Model Matrix \* Local Position
  + **View Matrix (World -> View)**
    - View Matrix \* World Position
  + **Custom Projection Matrix (View -> Clip)**
    - Projection Matrix \* View Position
    - Applies a perspective by creating a 4th dimension. (xyzw)
    - Created via OpenGL function.
    - Projection is calculation of 3D view to be seen like a 3D view in 2D view using 4th dimension.
      * 3D to 2D but without losing 3D view.
    - The user can decide if distant objects should appear smaller (Perspective) or normal (Orthographic) by switching OpenGL functions.

# Shader Object

* User-defined/fixed function of a Shader Program Object.
* Used for temporarily manipulating the Vertex, Primitive or Fragment Struct.
* Written in GLSL (openGL Shader Language).
* **How to know which one is fixed and user-definable?**
  + I lied to you.
  + Not everything is Shader Object in a Shader Program Object.
  + But it will help you to understand most of the things.
  + Know that whenever you see “Shader” on titles, they are user-definable.
* **What is** **Shader Program Object?**
  + Container that stores Shader Objects sequentially.
  + Syncs data between Shader Objects.
  + When a draw call received, runs all contained Shader Objects at GPU with specific data and order.
  + Actually, its a Rendering Pipeline.

# Rendering Pipeline

* Sequence of Shader Objects (Shader Program Object) for converting all of the sent Primitives to be 1 Image on your screen.
* Couldnt be %100 accurate but at least the Sequence will be same so i think its okay for beginners like me.
* For this document, assume GPU Driver uses some kind of Stream Processing model.
* **What is Stream Processing?**
  + Continuous usage of datas.
    - Example
      * NO: An array of 123 Primitive goes somewhere.
      * YES: 1 Primitive will take 123 Primitive one by one and goes somewhere.
* **Input/Output**
  + Every Shader Object has input (in) and output (out) variables that matches 1:1 with previous and next Shader Object.
  + A Shader Object will copy the specified data that comes from previous step to inside of the ‘in’ variables you will define, and copies its result to next step which is ‘out’ variables.
* **Primitive Vertex Buffer Object**
  + Primitive array.
    - Example
      * Primitive Triangle Vertex Buffer Object ‘A’  
        Assume float3 is 3 floating-point number; x, y, z.  
        1 Vertex = (float3 position, float3 normal)  
        2 Triangle = [V0, V0, V0, V1, V1, V1]  
        [V0, V0, V0, V1, V1, V1] = [6 x (float3 position, float3 normal)]
  + Problem is, Shader Object cant know every Vertex Attribute (position, normal) is a Vertex.
  + To let Shader Object know, we use Primitive Vertex Array Object.
* **Primitive Vertex Array Object**
  + Repeated connection of index to Primitive Vertex Buffer Object’s 1 Vertex Attribute.
    - Example
      * Primitive Triangle Vertex Buffer Object ‘A’  
        1 Vertex = (float3 position, float3 normal)  
        2 Triangle = [V0, V0, V0, V1, V1, V1]  
        [V0, V0, V0, V1, V1, V1] = [6 x (float3 position, float3 normal)]
      * Primitive Triangle Vertex Array Object ‘B’  
        Assume float3 occupying 12 byte.  
        1 position = index 0, starting from 0. byte, float3 \* 3, connected to ‘A’  
        1 normal = index 1, starting from 12. byte, float3 \* 3, connected to ‘A’
* **What is the purpose of using Primitive Vertex Array Object?**
  + You will use indexes in Shader Objects.
  + Furthermore, you are capable of taking a different Vertex Attribute from different Primitive Vertex Buffer Object.
* **If data is copied, what if i need to update my data?**
  + You may use those:
    - Transform Feedback (old)
      * Exactly does what you need.
    - Shader Storage Buffer Object (SSBO) (idk)
      * A special type of memory that can be written to or read from within a shader.
    - Compute Shader + glGetBufferSubData (new)
      * Compute shader is just literally computer. (lol i did a joke, see?)
* **What is Transform Feedback?**
  + Feature for copying specific a type of Primitive(s) to a specific Primitive Vertex Buffer Object.
  + To get results from multiple draw calls, you enable/disable from your code by calling some functions.
  + While enabling, you will specify what type of Primitive should be sent to you by default but its not guaranteed.
* **Tesellation**
  + Creation of more Primitives for Primitive Patch to obtain more geometry.
* **Draw call**
  + Function for starting the Rendering Pipeline using datas of current bound Primitive Vertex Array Object.
  + You will specify the wanted Primitive type here.
    - Because Primitive Buffer&Array Object does not knows the Primitive type of itself.
* **Preview of Pipeline**
  + **Prepare and invoke draw call**
  + **Vertex Shader**
  + **(If Primitive Type is Patch)**
    - **(Optional) Tesellation Control Shader**
    - **Tesellation Primitive Generator**
    - **Tesellation Evaluation Shader**
  + **(Optional) Geometry Shader**
  + **Primitive Assembly**
  + **Vertex Post-Process**
  + **Rasterizer**
  + **(Optional) Fragment Shader**
  + **Per-Sample Operations**

1. **Prepare Primitive Vertex Array&Buffer Object and start**
   * Main Purpose: User creates Primitive Vertex Buffer&Array Object and using a draw call.
   * Process
     1. Create Primitive Vertex Buffer&Array Object.
     2. At Shader Objects, define ‘in’ and ‘out’ variables.
     3. Invoke a draw call.
        + (Imaginary) Loop over Primitive Vertex Array Object
          1. Output.
   * Output
     1. 1 Vertex
2. **Vertex Shader**
   * Main Purpose: Invidually calculate position of 1 Vertex in Clip Space.
   * Input
     1. 1 Vertex.
   * Process For: 1 Vertex.
     1. Set Clip Space position.
   * Output
     1. 1 Vertex.
3. **(Imaginary) Output Router Of Vertex Shader**
   * Buffer += Output.
   * If (Buffer’s Vertex count == Draw Call Primitive Type’s Vertex count)
     1. If last
        + Invoke Transform Feedback (Draw Call’s Primitive Type) with copy of Buffer.
        + Run Primitive Assembly (Buffer).
     2. Else if (Draw Call’s Primitive Type == Patch)
        + Run Tesellation Control Shader (Buffer).
     3. Else
        + Run Geometry Shader (Buffer).
     4. Reset Buffer.
4. **(Optional, default used) Tessellation Control Shader**
   * Main Purpose: Set tesellation level (creation density).
   * Input
     1. 1 Primitive Patch.
   * Process For: 1 Vertex
     1. Optionally, manipulate Vertex.
     2. Calculate tesellation level for whole patch.
        + Increase of level means more new vertex.
   * Output
     1. 1 Primitive Patch.
     2. Tesellation levels.
5. **Tessellation Primitive Generator**
   * Main Purpose: Calculate Fractional Coordinate using all of the positions in a Primitive Patch.
   * Input
     1. 1 Primitive Patch.
     2. Tesellation levels.
   * Process For: 1 Primitive Patch
     1. Create Primitives that has a type defined in Tesellation Evaluation Shader.
     2. Loop over created Primitives
        + Loop over created Primitive
          1. Output.
   * Output
     1. 1 Primitive Patch.
     2. Tesellation levels.
     3. Primitive type and format defined in Tesellation Evaluation Shader.
     4. 1 Vertex Fractional Coordinate.
6. **Tessellation Evaluation Shader**
   * You must define an ‘in’ variable of the type of Primitive that Tesellation Primitive Generator should create.
   * You must define an ‘out’ variable of what type of Primitive will be sent.
   * Main Purpose: Calculate 1 Vertex Clip Space position of newly created Primitive that has a defined type.
   * Input
     1. 1 Primitive Patch.
     2. Tesellation levels.
     3. Defined Primitive type and format.
     4. 1 Vertex Fractional Coordinate.
   * Process For: 1 Newly Created Primitive’s 1 Vertex
     1. Calculate Vertex Clip Space position using Fractional Coordinate and every Vertex in Primitive Patch.
   * Output
     1. 1 Vertex.
7. **(Imaginary) Output Router Of Tesellation Evaluation Shader**
   * Buffer += Output.
   * If (Used Geometry Shader)
     1. If (Buffer’s Vertex count == Geometry Shader’s Defined Primitive Type’s Vertex count)
        + Run Geometry Shader (Buffer).
        + Reset Buffer.
   * Else
     1. If (Buffer’s Vertex count == Tesellation Evaluation Shader’s Defined Primitive Type’s Vertex count)
        + Invoke Transform Feedback (Tesellation Evaluation Shader’s defined ‘out’ Primitive Type) with copy of Buffer.
        + Run Primitive Assembly (Tesellation Evaluation Shader’s defined ‘out’ Primitive Type) with copy of Buffer.
        + Reset Buffer.
8. **(Optional) Geometry Shader**
   * If Tesellation used: You must specify an ‘in’ variable for the type of Primitive that Tesellation Evaluation Shader defined.
   * You must define an ‘out’ variable of what type of Primitive(s) will be created at this step.
   * Main Purpose: Create new Primitive(s) using a Primitive.
   * Input
     1. 1 Primitive.
   * Process For: 1 Primitive.
     1. Create Vertex by using function ‘EmitVertex’.
     2. Mark end of the Primitive by using function ‘EndPrimitive’.
   * Output
     1. 1 new Primitive.
9. **(Imaginary) Output Router Of Geometry Shader**
   * Invoke Transform Feedback (Geometry Shader’s defined ‘out’ Primitive Type) with copy of Output.
   * Run Primitive Assembly (Geometry Shader’s defined ‘out’ Primitive Type) with copy of Output.
10. **Primitive Assembly**
    * Main Purpose: Create Primitive Struct using a Primitive.
    * Input
      1. 1 Primitive.
    * Process For: 1 Primitive
      1. Create 1 Primitive Struct.
      2. Output.
    * Output
      1. 1 Primitive Struct.
11. **Vertex Post-Process: Clipping, Face Culling, Perspective Divide, Viewport Transform**
    * Main Purpose: Convert seen Primitive Struct’s Clip Space coordinates to Normalized Device Coordinates.
    * Input
      1. 1 Primitive Struct.
    * Process For: 1 Primitive Struct.
      1. Do Clipping in Clip Space
         + Discard Primitive Struct if not seen in the camera's field of view.
      2. If Face Culling is enabled: Do Face Culling.
      3. Do Perspective Divide
         + Convert Clip Space Vertex positions of Primitive Struct to Normalized Device Coordinate.
      4. Do Viewport Transform
         + Convert Normalized Device Coordinate Vertex positions of Primitive Struct to Screen Space.
    * Output
      1. 1 Primitive Struct.
12. **Rasterizer**
    * Main Purpose: Create Fragments in Screen Space from a Primitive Struct.
    * Input
      1. 1 Primitive Struct.
    * Process For: 1 Primitive Struct
      1. Create Fragments in Screen Space.
      2. Loop over created Fragments
         + Output.
    * Output
      1. 1 Fragment Struct.
13. **(Optional) Fragment Shader**
    * Main Purpose: Manipulate fragment.
    * Input
      1. 1 Fragment Struct.
    * Process For: 1 Fragment
      1. Manipulate Fragment.
      2. Output.
    * Output
      1. 1 Fragment Struct.
14. **Per-Sample Operations**
    * Main Purpose: Test and write Fragment to Frame Buffer as pixels.
    * Input
      1. 1 Fragment Struct.
    * Process For: 1 Fragment Struct:
      1. Pixel ownership test: Fails if the Fragment Struct's Pixel is not "owned" by OpenGL
         + Example: A window is overlapping with the GL window.
         + Always passes when using a Framebuffer Object.
         + Failure means that the Pixel contains undefined values.
      2. Scissor Test: When enabled, the test fails if the Fragment Struct's pixel lies outside of a specified rectangle of the screen.
      3. Stencil Test: When enabled, the test fails if the stencil value provided by the test doesnt meet the instructions given by the user.
      4. Depth Test: When enabled, the test fails if the Fragment Struct’s depth doesnt meet the instructions given by the user.
      5. Blending: Blend Fragment Struct color using specific blending operation and the color already in the framebuffer at that location.
      6. Write With Mask: Write Fragment Struct to the Framebuffer using Masking Operations that allows users to prevent writes to certain values.

# Face Culling

* Every Primitive Struct got 2 face.
  + Example: Imagine a filled triangle; sometimes we want to see (render) just one face because %50 ez performance gain.
* This technique just takes the part of choosen visible face using Cross Product vector math.
* To put that in words, they just thought of a logic of "Clockwise/Counter Clockwise order”.
  + Begins from first, an array of Vertex (Primitive) is visually not inverted or inverted on screen.
  + Back Face (GL\_CW) = Clockwise order of primitive Vertex array.
  + Front Face (GL\_CCW) = Counter clockwise order of Vertex array.

# Image

* Okay, lets get rid of a confusion first:
  + Image = Raw Pixel array and its attributes.
  + .png, .jpg, ... = Compressed Image data.
* **What is raw pixel array and its attributes?**
  + Raw Data (Color): 1 pixel array.
  + Attributes of raw data.
    - Dimension
      * Structure of the pixel array.
      * 1D: Width (x)
      * 2D: Width (x), Height (y)
      * 3D: Width (x), Height (y), Depth (z)
    - Size
      * Count of the pixels for every dimension.
      * 1D: Width Count, 0, 0
      * 2D: Width Count, Height Count, 0
      * 3D: Width Count, Height Count, Depth Count
    - Format
      * Color Channels.
      * Determines the Subpixel count.
    - Data Type
      * Each Color Channel’s Bit Count.
    - Alignment, Row Pitch
      * Optimization.
* **Lowscale/Upscale**
  + The Format and Data Type can be lowscaled or upscaled when passing around.
  + You will usually see those words so:
    - External
      * Actual format and data type.
    - Internal
      * Convert External to allowed Format and Data Type.
* **How to Lowscale or Upscale an Image?**
  + The OpenGL has an enum for safety.
    - GL\_[Color Channel][Bit Count][Value Type of Bit Count]
  + Lets talk about Value Type of Bit Count:
    - none: [0, 1] Unsigned normalized floating-point number.
    - "\_SNORM": [-1, 1] Signed normalized floating-point number.
    - "F": Floatint-point number.
    - "I": Signed integer.
    - "UI": Unsigned integer.
    - Examples:
      * GL\_RGBA8I: each channel has 8bit signed integer.
      * GL\_RGBA32F: each channel has 32bit float value.
      * GL\_R4: not allowed.
      * GL\_RGBA10: not allowed.

# Texture Object

* Container of 1 Image.
  + Type (Image Dimension)
  + Size (Image Size)
  + Other Attributes (Other Image Attributes)
* **Why used?**
  + Image cannot be stored in VRAM because its not an OpenGL Object.
  + We can get 1 Texel using Sampling and Filtering.
* **Interpolation**
  + Estimated value based on an array of values.
    - Example: Mixing 3 color gives 1 color.
* **What is Texel?**
  + Interpolated Pixel.
* **What is Sampling?**
  + Process of getting Texel from a specific coordinate.
  + The coordinates are typically [x, y] with normalized floating-point number [0, 1].
    - Example: [x, y] percentage each between [0, 1].
* **What is Filtering?**
  + Filtering is applied when Sampling do not directly correspond a Texel.
    - Example: Get a Texel between 2 Texel.

# GLSL (OpenGL Shader Language)

* Shader Object Language inspired by C/C++.
* Every shader has entrypoint "void main()".
* You must define the GLSL version with preprocessor "#version" at first line.
* **Preprocessor Directives**
  + #versioninteger\_version\_number support\_type
    - integer\_version\_number
      * Can be found in their documentation but usually they are really straightforward with OpenGL version.
        + Example: OpenGL 4.6 = 460
    - support\_type
      * compatibility
        + Support deprecated features.
      * core
        + Support new stuff.
      * es
        + Support embedded systems.
* **Macros**
  + \_\_FILE\_\_
    - Not a filename; a decimal integer representing which string in the list of strings given to the shader.
  + \_\_LINE\_\_
    - The line number.
  + \_\_VERSION\_\_
    - A decimal integer representing the GLSL version being compiled.
      * Example: OpenGL 4.6 = 460
* **Functions**
  + Recursive Functions not allowed.
  + Uses a calling convention called "value-return" by using Type Qualifiers.
* **Keywords**
  + goto
    - Not allowed.
  + discard
    - Fragment Shader only.
* **Type Qualifiers**
  + Determines where the value of variable will come from and how the variable will be used.
  + Storage
    - (Default) none
      * As normal variable.
    - const
      * Non-changing readonly value.
    - in
      * The input value coming from VRAM.
        + Example: Vertex Attribute in Vertex Shader.
    - out
      * The output value going to VRAM.
        + Example: Vertex Attribute in Vertex Shader.
    - inout
      * Combination of in and out.
    - uniform
      * Readonly value sent from your application.
  + Function
    - (Default) in
      * On get call, copies from input parameter.
      * Initial value: Copied value.
    - out
      * On self return, writes (copies) to output parameter.
      * Initial value: undefined.
    - inout
      * Combination of in and out.
      * Initial value: Copied value.
* **Basic Types**
  + If uniform: Copy of your data.
  + Scalar
    - bool
      * true (1 or anything) or false (0).
    - int
      * 32-bit two's complement signed integer number.
    - uint
      * 32-bit unsigned integer number.
      * Suffix "uU".
        + Example: 3u.
    - float
      * Single-precision (32-bit) floating-point number.
      * Suffix "fF".
        + Example: 3f.
    - double
      * Double-precision (64-bit) floating-point number.
      * Suffix "lfLF".
        + Example: 2lf.

* + Vector
    - Struct that contains N Scalar variable.
    - N can be 2, 3 or 4.
      * bvecN: bools
      * ivecN: ints
      * uvecN: uints
      * vecN: floats
      * dvecN: doubles

* + - Swizzling
      * Constructor/initializator syntactic sugar for Vectors.
        + Becareful, dont get diabetes HA AH A HAHAH HA HAH!
      * It swaps your code with constructors/initializators, super useful.
      * Consider this valid code:

vec2 someVec;

vec4 otherVec = someVec.xyxx; // otherVec.xyzw = someVec.xyxx; So they match.

vec3 thirdVec = otherVec.zy; // l-value takes more than right. 'otherVec.zy' repearts the latest: 'otherVec.zyy'

someVec.xy = vec2(3.0, 5.0); // "someVec.xx" is NOT VALID

* + - * Any combination of up to these 4 letters can bring meaning to any vector but cannot be mixed:
        + xyzw (General Purpose)
        + rgba (Colors)
        + stpq (Textures)
  + Matrix
    - N (Column) and M (Row) can be 2, 3 or 4.
      * matNxM: Flipped float Matrix
      * dmatNxM: Flipped double Matrix

* **Opaque Types**
  + Always 'uniform': Identifier of your data that is bound in your application.
  + Restrictions
    - Can only be declared at
      * At global scope, with a type of 'uniform'.
      * Member of a struct, but if so, then the struct can only be used to declare with a type of 'uniform'.
    - Can only be used as
      * A parameter to Function Type Qualifier ‘in’.
  + Sampler (Texture Reader)
    - Identifier of a texture.
      * sampler2D: 2D Texture
      * sampler3D: 3D Texture
      * ...
  + Texture Image
    - Identifier of an Image that is bound to a Texture.
    - Sampling and Filtering not allowed.
      * You will directly access the pixel with exact coordinates.
      * image2D: 2D Image
      * image3D: 3D Image
      * ...
  + Atomic Counter (Unique uint)
    - Identifier of an uint.
    - Used for threading shaders.
    - You could have asked "bro why?"
      * Shared index for shared data.
      * How many pixels met a certain condition in the shader?
* **Type Qualifiers Continued**
  + Layout
    - Used for gathering data using a format.
    - Can be extracted in your application.
    - Used with suffix "layout(...)".
    - location
      * Input/Output index for Basic Type.
        + Example: Vertex Shader's Vertex position input index that bound in your application.
        + Example: Input/Output index between 2 Shader Object.
    - binding
      * Input index for Opaque Type.
        + Example: Input index that bound to an identifier in your application.
        + Example: Input/Output index between 2 Shader Object.
    - index
      * Output index of Fragment Shader.
* **Struct**
  + If you use Opaque Type in a struct, then you must use entire struct like an Opaque Type.
    - Example: Used a sampler.
* **Interface Block (Not a struct)**
  + Group of Basic Types that all share a Type Qualifier.
  + Consider the definition rule:

storage\_type\_qualifier block\_name

{

<define basic type members here>

} optional\_instance\_or\_instance\_array;

// storage\_type\_qualifier = uniform, in/out, buffer

* + Its members are globally defined directly or using instance.
  + Used to abstract the input/output data from your application to Shader Object or between the Shader Objects.
  + Can be used in 2 ways:
    - Access it directly to a member.
      * Example: member.
    - Using optional instance in the code.
      * Example: optional\_instance\_or\_instance\_array.member.
* **Initializers**
  + Uniform Basic Types can be initialized once when Shader Program Object is compiled.
    - Example: uniform int aValue = 5;
* **Constructors**
  + Matrix

mat4(

vec4, // first column

vec4, // second column

vec4, // third column

vec4, // fourth column

);

* **Predefined Variables**
  + Different Shader Objects have their own predefined variables.
  + All start with prefix "gl\_".
    - You cannot use that prefix for your variables.
* **Useful Functions**
  + imageLoad
    - Loads Texture Image.
  + imageStore
    - Writes to Texture Image.
  + length
    - Compile-time constant array length.

# Other

* Colors in OpenGL are RGBA.
* Diffuse color
  + Fragment Struct Color resulting from the direct, matte (non-glowing) reflection of light coming to a surface.
  + In other words: The light from a light source that scatters when it hits a surface.
  + In other words: Primary color of a surface.
  + In other words: True underlying color of a surface under light. It doesn't shine; it simply appears brighter if light hits that area, and darker if it doesn't.
* Ambient color
  + The general light of the environment is distributed equally.
* Specular (reflected)
  + Shine, the effect of reflection
    - Example: Points of light shining on surfaces such as metal or glass.

# Glossary

* glutSetWindow
  + Set the active context for current thread.
* GL\_COLOR\_BUFFER\_BIT
  + Mask that tells "its a FREAKING COLOR BUUUUUUUUUUfFER".
* glClear
  + Clear the active context.
* glClearColor
  + Set the color of cleared context.
* glGenXXXX
  + Create identifier for specific target.
* glBindXXX
  + Use identifier data in specific target.
* glBufferData
  + Set whole data for the buffer identifier.
* glBufferSubData
  + Update part of the data for the buffer identifier.
* glGenVertexArrays
  + Create identifier for vertex array(s).
* glBindVertexArray
  + Use identifier data as vertex array.
* glBindVertexBuffer
  + Use Vertex Buffer Object data when THAT binding-index is used. (binding-index = vertex buffer object)
* glVertexAttribBinding
  + Use binding-index data when THAT attribute (position, normal etc...) is used. (attribute USES binding-index)
  + If not used, (attribute = binding-index)
* glVertexAttribFormat
  + For THAT attribute, tell where is the data and how data is stored at the Vertex Buffer Object.
* glEnableVertexAttribArray
  + Make attribute sendable to the shader.
* GL\_ARRAY\_BUFFER
  + Target for Primtive Vertex Array Object.
* glDrawArrays
  + Reads from Vertex Array Object and passes to pipeline.
  + The last two parameters are selection of part of your data.
  + From the first parameter, it understands how many vertex a primitive needs.
* glEnable
  + Sets the settings of active context.
* glFrontFace
  + Sets which face is considered as the front face.
* glCullFace
  + Sets which face will be culled.
  + Default is Back Face.
* glCreateProgram
  + Creates a shader program identifier.
* glCreateShader
  + Creates a shader identifier.
* glShaderSource
  + Set whole data of a shader identifier.
* glCompileShader
  + Turns shader into a binary-readable format (resource/source file).
* glAttachShader
  + Use identifier data in Shader Program's Shader Target (hand. example: if the Shader Type is Vertex Shader, then Target is Vertex Shader).
* glDetachShader
  + Stop using identifier data in Shader Program's Shader Target (hand. example: if the Shader Type is Vertex Shader, then Target is Vertex Shader).
* glLinkProgram
  + Compiles the shader program USING the attached shaders. (REFRESHES EVERY TIME U LINK/COMPILE. OLD COMPILED STUFF WONT STAY THERE LOL)
  + So turns into a binary-readable format (resource/source file).
* glValidateProgram
  + Checks if something could possibly mess up the Shader Object Program.
* glUseProgram
  + Uses the program in Rendering Pipeline.
  + I AM SHOCKED THAT THEY DIDNT USE SOME KIND OF NAME "GLBINDPROGRAM" omg