OpenGL 4.6

Tutorial For Beginners

C/C++ & Math Fundamentals Required

yunus yıldız

2025

# Abbreviations

* You can jump to here by clicking any abbreviation.

|  |  |
| --- | --- |
| API | Application Program Interface |
| CPU | Central Processing Unit |
| DLL | Dynamic Link Library |
| ES | Embedded System |
| GL | Graphics Library |
| GPU | Graphics Processing Unit |
| LIB | Library |
| OOP | Object Oriented Programming |
| OS | Operating System |
| RAM | Random Access Memory |
| VRAM | Video Random Access Memory |
| GLFW | Graphics Library Framework |
| GLUT | OpenGL Utility Toolkit |
| GLEW | OpenGL Extension Wrangler Library |
| GLAD | OpenGL Loader |
| NV | Nvidia |
| EXT | Generic Extension |
| ARB | Architecture Review Board |
| D | Dimension |
| NDC | Normalized Device Coordinate |
| FOV | Field Of View |
| POV | Point Of View |
| GLSL | OpenGL Shader Language |
| VBO | Vertex Buffer Object |
| VAO | Vertex Array Object |
| VEO | Vertex Element Object |

# OpenGL

* Platform-Independent Renderer API or Specification Standart Library.
* **What is a “renderer”?**
  + “Render” is the process of generating an image.
* **What is Specification Standart?**
  + Definition and implementations have rules.
    - Example  
      "In version n, the result of function ‘x’ should equal to y"
* **What is Library?**
  + LIB contains the definition, in this case its OpenGL.
    - For OpenGL, its called “Extensions”.
    - Example  
      "For version n, define function ‘x’"
  + DLL contains the implementation (code), in this case its GPU Driver.
    - The GPU Driver decides which feature and version to support, data management, so everything about the datas and code.
    - Example  
      “For version n, implement function ‘x’ code”
  + Sometimes LIB used as “Static Link Library” which contains the implementation too.
  + LIB file gets mixed with DLLs 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.
  + If LIB and DLL is used, linking process happens at runtime and that is what happens for OpenGL.
* **How Open**GL **works?**
  + Oftenly, you will hear "Works with states so its a state machine".
  + Lets imagine how exactly it works.
    - 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.
  + In this document, we will use some kind of OOP that built on top of OpenGLbut the idea will remain the same.
* **Open**GL **Object**
  + Data inside VRAM.
  + CPU data will stay on RAM.
    - Example  
      Identifier.
  + GPU data will stay on VRAM.
    - Example  
      Identifier data.
  + Always has an identifier.
  + Developer will inject the data using the identifier.
  + When you request deletion of an identifier, that means you request the deletion of data inside VRAM.
  + The suffix is “Object”.
* **How to create an identifier for an Open**GL **Object?**
  + 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 should not access it.
  + So we bind the identifier to a target to inject the data.
    - Imagine your hand. (target)
    - Imagine the tool in your hand. (bind to target)
    - Use your mind to create the tool in your hand. (inject to target)
* **What is binding?**
  + Giving reference to an OpenGL Object by using an identifier.
* **What is target?**
  + Single slot for 1 OpenGL Object.
  + There may be thousands of OpenGL Objects but you are capable of using one at a time.
    - Example  
      GL\_ARRAY\_BUFFER
* **What is buffer?**
  + Temporary data.
* **Open**GL **Context Object**
  + OpenGL Object container associated with a thread.
    - Process is the application which is the container of your data.
    - Thread is a child of process which is the worker.
    - Process is 1, thread can be [0, infinity].
  + You will just attach the OpenGL Context Object to a thread manually.
    - Every thread has an entry function (entrypoint).
    - If you use OpenGL functions in the entrypoint, it will automatically detect the thread and its OpenGL Context.
  + Without it, OpenGL functions wont work because it contains informations about OpenGL.
    - Example  
      Process is the application.  
      Thread is the window that does some work.  
      Context is inside the window.  
      Context stores; used by which GPU, using which OpenGL version, decide which window to draw, store data and manage access, etc...
* **How to draw stuff?**
  + The GPU Driver manages the OpenGL Objects.
  + You will use identifiers to manage or use the OpenGL Objects.
  + Using the the OpenGL Objects, you will draw everything to an OpenGL Context Object.
* **Open**GL **code access path**
  + Every OS has its own unstandardized system for file, window, input, event, etc...
  + So the connection to OpenGL code wherever it stays, will differ.
    - Application <-> OS <-> GPU Driver <-> OpenGL code.
  + To keep that connection ease, every OS creates a access path by creating their own LIB which you can access in your application.
    - Application <-> Access path LIB <-> OS <-> GPU Driver <-> OpenGL code.
  + To use OpenGL, an application must reference to the OpenGL code.
  + If there was no access path; to support OpenGL, we could have been required to import the LIB & DLL manually.
* **How to reference to the** **Open**GL **code in application?**
  + Firstly, an OpenGL Context Object must be created to gather information about OpenGL.
  + Since the GPU Driver knows everything and you dont, you will request GPU Driver to create an OpenGL Context Object using OpenGL code access path.
  + If we want to simplify the creation for every OS, we use some library.
    - FreeGLUT (USING)
    - GLFW
  + These libraries used for creating and managing windows containing OpenGL Context Objects on a wide range of platforms and also read the mouse, keyboard and joystick functions.
  + Using OpenGL Context Object, we attach a reference manually one by one for all of the needed functions.
  + If we want to simplify the referencing process, we use some library.
    - GLEW (USING)
    - GLAD
    - For fun, the driver can have absurd methods inside it's OpenGL code which you can load!!
  + These libraries used for determining and referencing supported OpenGL extensions on the target platform.
  + Use these libraries for education only because they are covering only fundamental things you need.
* **How Free**GLUT **works?**
  + Assume we created a window.
  + Window creates its own OpenGL Context Object.
  + The window is an identifier too.
  + Be sure you are working with right window.
* **How** GLEW **works?**
  + Uses the current OpenGL Context Object to select which functions to load at once with single function.
  + 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 Open**GL **Context to load functions?**
  + OpenGL Context Object is just a container of datas and for us; its an identifier, not an interface.
  + Even tho GLEW uses the OpenGL Context Object to find which functions to load, technically they are gathering some data only and not using as an interface.
* **Should Open**GL **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 ‘Open**GLES**’?**
  + OpenGL: Windows, Linux, MacOS, ...
  + OpenGL ES: Android, IOS, Web (WebGL), ...
* **What is the difference from other** API**s like DirectX?**
  + Similiar with other APIs.
* **Open**GL **Extensions**
  + The big companies has developed their own extensions for OpenGL too.
* **How to seperate other extensions from standard Open**GL**?**
  + All extensions have prefixes.
    - OpenGL Core
      * GL\_
    - Generic (General Purpose)
      * GL\_EXT
    - Vendor (Specific)
      * GL\_NV for Nvidia
      * GL\_APPLE for Macintoshe
      * ...
    - ARB (Specific)
      * GL\_ARB
        + Vendor specific but capable of being OpenGL Core.

# Vector

* A structure that contains N number together.
* Every Vector have magnitude and direction on its dimension.
* You can assume a Vector is an array or imaginary MatrixNx1 or Matrix1xN.
* Named as “VectorN”.
  + Example  
    Vector4 = (x, y, z, w) = (1.0, 0.5, 0, 0).
* **What is magnitude?**
  + Length; the value of “How long it is?”.
  + To calculate, imagine a float Vector2
    - Magnitude = SquareRoot(x2 + y2).
    - Square Root = Which number multiplied twice to find the given input?
    - Example  
      (10, 0, 0) = 10  
      (0, 10, 0) = 10  
      (0, 1, 0) = 1.
* **Normalized Vector**
  + Vector that has a magnitude of 1 (The ineffective number in multiplication).
  + Used for representing a direction.
  + To calculate, do easy math
    - Vector / Magnitude = Direction.
    - Try to find number ‘1’.
    - Example  
      (10, 0, 0) = Not normalized  
      (0, 10, 0) = Not normalized  
      (0, 1, 0) = Normalized.
* **What is direction?**
  + Normalized Vector that says “I am pointing to xxxx”.
  + Thinking a direction would be easier if you imagine a direction starts from the center of a circle that is drawn on a paper.
    - Example  
      (1, 0, 0) = Right  
      (0, 1, 0) = Up  
      (0, 0, 1) = Forward.
* **What is dimension?**
  + Number of directions that are perpendicular to eachother.
  + Specifies which directions an object can move.
  + Every direction in a dimension called “Axis”.
    - Example:  
      3D = x, y, z  
      x = Horizontal  
      y = Vertical  
      z = Depth  
      (1, 0, 0) = Right  
      (0, -1, 0) = Bottom  
      (0, 0, 1) = Forward.
* **What is perpendicular?**
  + Directions are facing each other at exactly 90° angle, like an ‘L’.
* **What is angle?**
  + Limited amount of opening between 2 directions that starts from 1 point.
    - In other words: The difference (-) between two directions.
  + Prefix for angle is ‘θ’ .
  + The angle can be showed with 2 techniques
    - Degree (°)
      * Value between [0, 360].
    - Radian (rad)
      * Value between [0, PI \* 2].
* **Matrix**
  + A structure that contains float Vectors arranged in an orderly manner of rows and columns similiar to a table.
  + Named as “MatrixROWxCOLUMN”.
  + Matrixes can be combined together.
  + Commonly used for applying multiple operations (add, subtract, …) to a Vector at once by using 1 multiplication.
    - Example  
      Matrix3x2 =
* **Why do we even use Matrix?**
  + Every GPU optimizes Matrix operations.
    - Example  
      Instead of doing 4 changes with 4 cycle, doing 4 changes with 1 cycle.
* **Dot Product (.)**
  + Multiplication and addition of the components within order between 2 Vector.
  + Take column on the left, row on the right.
    - Example  
       . = (x1 \* y1) + (x2 \* y2) + (x3 \* y3) = Number
  + Has 2 use cases:
    - If used on directions, used for calculating “How much 2 directions point in the same direction?” that actually gives a cosine value on 2D:  
      1 = 0° = same  
      0 = 90° = perpendicular  
      -1 = 180° = opposite.
      * Imagine half Unit Circle.
    - Used for doing operations (add, subtract, …).
* **How does Matrix multiplication works?**
  + You may think “How will we add or subtract numbers from a Vector using Matrix multiplication?”.
  + Matrix multiplication uses Dot Product.
  + Rules
    - Left Matrix COLUMN = Right Matrix ROW.
    - Always applies to its right so you can assume the order of application is right to left.
    - MatrixROWxCOLUMN \* VectorROW multiplication results a Vector.
      * N Matrix can affect only 1 Vector.
      * Example:  
        gl\_Position = subtract \* add \* localPosition;  
        gl\_Position = subtract \* (add \* localPosition);
    - MatrixROWxCOLUMN \* MatrixROWxCOLUMN multiplication results a combined Matrix.
      * N Matrix can be combined together.
      * Example:  
        addSubtract = subtract \* add;  
        addSubtract = subtract \* (add \* addSubtract));
    - Example  
       \* =
* **Sine**
  + Function that takes an angle, gives you a normalized value between [-1, 1].
  + Used for representing vertical movement.
  + 0 is default because Unit Circle pointing to right:  
    1 = top  
    0 = middle  
    -1 =bottom.
* **Cosine**
  + Function that takes an angle, gives you a normalized value between [-1, 1].
  + Used for representing horizontal movement.
  + 1 is default because Unit Circle pointing to right:  
    1 = right  
    0 = middle  
    -1 = left.
* **Unit Circle**
  + Normalized 2D circle that composed of:  
    Center = (0, 0)  
    Magnitude = 1  
    Direction = (1, 0) = (horizontal, vertical) = (cosine, sine) = (x, y)  
    MaxAngle = 360° = PI \* 2  
    DotProduct = Half Unit Circle = (-1 = Left, 0 = Top, 1 = Right).
  + Used for thinking directions and calculating angles on 2D.
    - Example  
      Angle between Unit Circle direction and 2. direction  
      2. direction (right) = 0° = 0 rad  
      2. Direction (up) = 90° = PI / 2 rad  
      2. Direction (left) = 180° = PI rad  
      2. Direction (bottom) = 270° = (PI \* 3) / 2 rad  
      2. Direction (right) = 360° = PI \* 2 rad.
* **How to calculate angle between 2 direction?**
  + DotProduct = Dot Product of 2 direction
  + Angle = Arccos(DotProduct).
  + Arccos takes cosine value and converts to angle between [0, 180] but its complicated so search about it.
* **How to calculate an angle that is between [0, 360]?**
  + You cant directly get by using only Dot Product.
  + However, you can get “If we think two directions create a wall, which side of wall is front?” by doing Cross Product and using the result is up to you.
* **Cross Product (x)**
  + Multiplication, addition and subtraction of the components in Matrix within cross ‘X’ order between 2 Vector.
    - Example  
       x = =  
      ((i \* x2 \* 0) + (x1 \* y2 \* k) + (y1 \* j \* 0)) - ((k \* x2 \* y1) + (0 \* y2 \* i) + (0 \* j \* x1)) =  
      (0 + (x1 \* y2 \* k) + 0) – ((k \* x2 \* y1) + 0 + 0) =  
      x1 \* y2 \* k - k \* x2 \* y1 =  
      Here its 1 value for 2D which gives z-axis for 3D but this results a Vector in 3D
  + If used on directions, used for calculating “New perpendicular direction found using 2 direction” that actually gives sine value on 2D.  
    1 = 90° = forward  
    0 = 0° = undefined  
    -1 = 180° = backward (to you)
    - Imagine half Unit Circle.
* **What are i, j and k?**
  + Unit (Normalized) Vectors for 3D.  
    i = x = (1, 0, 0)  
    j = y = (0, 1, 0)  
    k = z = (0, 0, 1)
    - Example  
      2i + 3j - 5k = (2, 3, -5)

# Position

* Arbitrary Vector that represents a coordinate.
* Position is the main key of OpenGL.
* OpenGL uses 3D Homogeneous Coordinate to represent a position.
  + OpenGL assumes that you will use Matrix4x4 to apply a change to a position.
* Can be represented with different techniques.
* **Homogeneous Coordinate**
  + General purpose mathematical tool that adds extra arbitrary component named ‘w’ to a Coordinate.
  + Usually used for perspective where a rule says “two parallel lines will meet at an origin.”.
  + We can convert between actual coordinate and homogeneous coordinate by doing simple division.
    - Example:  
      2D: (x, y)  
      2D Homogeneous: (x, y, w)  
      3D: (x, y ,z)  
      3D Homogeneous: (x, y, z, w)
* **Barycentric Coordinate**
  + Offset value as percentage that is between [0, 1] which is calculated using an array of values.  
    n = 1 = equal to value  
    n < 1 = not equal to value.
    - Example  
      For 3 vertexes on a triangle may be:  
      barycentric at vertex 1 = (1, 0, 0)  
      barycentric at vertex 2 = (0, 1, 0)  
      barycentric at vertex 3 = (0, 0, 1).
  + For a mathematical reason, the sum of the percentages must equal to 1.
  + Muliplying Barycentric Coordinate with every value in array will give a new Coordinate.
    - Example  
      barycentric = (x, y, z)  
      actual = (barycentric \* vertex1.position) + (barycentric \* vertex2.position) + (barycentric \* vertex3.position)
* NDC
  + Normalized floating-point number.
  + Represents perspective applied Homogeneous Coordinate.
  + -1 left, 0 center, 1 right.
    - Example  
      (0.5, 0.5, 0.5)
* **Screen Coordinate**
  + [X, Y] Integer value.
  + NDC is converted to pixel coordinates on the screen by applying viewport size/transform.
  + For OpenGL
    - Origin (0, 0) is (Left-bottom side of your screen).
      * Example  
        (100, 200)

# Rotation

* Arbitrary Vector that represents an orientation.
* OpenGL does not gives any support for rotation so we will store Rotation as float Matrix4x4 to apply a Rotation to a Position.
* Can be represented with different techniques.
* **Euler Angle**
  + Rotation amount around an axis.
  + Axises are parented.
    - So the rotation will be processed for each axis seperately.
    - You will determine the parent axis of the other axises by sorting them.
  + Every Euler Angle in 3D axis has special name and all of them called “Euler Angles”  
    X (Look Left/Right) = Yaw  
    Y (Look Top/Down) = Pitch  
    Z (Roll Left/Right) = Roll.
    - Example:  
      Yaw: 360° rotation around the X-axis (nodding your head up and down)  
       Pitch: 360° rotation around the Y-axis (looking left and right)  
       Roll: 360° rotation around the Z-axis (opening a lid that is facing to you)  
      (x, y, z) = (360, 360, 360)
  + Problems
    - Gimbal Lock (Loss of one axis) can occur.
      * People usually change the parenting order of Rotation to minimalize this issue.
    - Hard to do smooth transformation.
* **Quaternion**
  + Normalized float Vector4 that creates rotation amount around a direction in 3D without any problem:  
    xyz: Direction  
    w: Angle.
  + Unlike others, you can combine Quaternions with multiplication.
    - Different from Matrix multiplication because needs to optimize calculation while preventing Gimbal Lock.
    - However, the order of application is same as Matrix multiplication. (Right to left)  
      a \* b =  
      x = (aw \* bx) + (ax \* bw) + (ay \* bz) – (az \* by)  
      y = (aw \* by) – (ax \* bz) + (ay \* bw) + (az \* bx)  
      z = (aw \* bz) + (ax \* by) – (ay \* bx) + (az \* bw)  
      w = (aw \* bw) – (ax \* bx) – (ay \* by) – (az \* bz)
  + To create, you do simple math
    - Quaternion = Vector4(Direction Vector3 \* sin(Angle / 2), cos(Angle / 2))
    - Example  
      90° rotation around y-axis.  
      Y-axis: (0, 1, 0)  
      Angle: 90°.
  + Problems
    - Hard to manually create so you will use a simple technique and convert to Quaternion.
* **Euler Angles to Quaternion**
  + Create a Quaternion for each Euler Angle.
  + Combine created Quaternions by multiplying them.
* **How to rotate around a specific position?**
  + Rotations cant hold such position information.
  + So you will do that using a position movement (translation) trick.
    - Move pivot to the origin (inversed translation).
    - Apply the rotation.
    - Move the pivot to its place (translation).

# Spaces

* Represents a Position relative to an origin.
* To apply any change to a position or change the space type of a position, we use Matrix4x4-Vector4 multiplication because OpenGL uses 3D Homogeneous Coordinate for position.
* We will create these Matrixes at application.
* **3**D **Local Space**
  + Origin (0, 0, 0) is (Object).
    - Example  
      Origin in World Position (Object) = 5, 0, 0  
      Local Position = 10, 0, 0
* **Model Matrix**
  + Combination of 4x4 Matrixes in order.
  + Used for applying arbitrary transformation or conversion to Homogeneous 3D World Space.  
    Translation \* Rotation \* Scale.
    - Scale Matrix
    - Quaternion Matrix
    - Position (Translation) Matrix
  + What Homogeneous Coordinate (w) means here?  
    w = 1 = Position  
    w = n = Position with perspective.
    - Example  
      Origin in World Position (Object) = 5, 0, 0, 1  
      Local Position = 10, 0, 0, 1  
      Translation Matrix = Origin  
      Translation Matrix \* Local Position =  
       \* = = (15, 0, 0, 1) = World Position
* **Homogeneous** 3D **World Space**
  + Origin (0, 0, 0, w) is (0, 0, 0, w).
    - Example:  
      Origin in World Position (Object) = 5, 0, 0  
      Local Position = 10, 0, 0  
      World Position = 15, 0, 0, 1.
* **View (Camera) Matrix**
  + Inverted and reduced version of Model Matrix.
  + Difference is, its especially created using point of view (camera) transform.
  + Used for applying camera transform, so conversion to Homogeneous 3D View Space.  
    Quaternion Inversed \* Translation Inversed
    - Inversed Position (Translation) Matrix
    - Inversed Quaternion Matrix
  + What Homogeneous Coordinate (w) means here?  
    w = 1 = Position  
    w = n = Position with perspective.
* **Homogeneous 3**D **View (Camera) Space**
  + Homogoeneus 3D World Space but camera transform applied.
  + Origin (0, 0, 0, w) is (0, 0, 0, w).
    - Example:  
      Origin in World Position (Object) = 5, 0, 0, 1  
      Local Position = 10, 0, 0, 1  
      View World Position = 15, 0, 0, 1  
      World Position = 0, 0, 0, 1.
* **Projection Matrix**
  + It doesn't have as much to do with "position and rotation" as Model and View, but it ultimately allows us to draw the 3D scene on the screen in 2D.
  + Represented as a single matrix in two categories:  
    Perspective = Distant objects appear small, while nearby objects appear large.  
    Orthographic = The size of objects does not change with distance.
  + Used for embedding distance between view point and thing into Homogeneous Coordinate while applying perspective, so conversion to Homogeneous 3D Clip Space.
    - Perspective Projection Matrix  
      Used: Near plane, far plane, FOV,aspect ratio
    - Orthographic Projection Matrix  
      Used: Left, right, bottom, top, near, far
* **Homogeneous 3**D **Clip Space**
  + Origin (0, 0, 0, w) is (POV).
  + What Homogeneous Coordinate (w) means here?
    - Can be seen.
      * Example  
        -w <= x <= +w
    - Distance between view point and object.

# 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.
* **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.
* **Primitive Triangle**
  + 3 Vertex.
  + Fastest because optimized for GPU especially.
    - In the ancient history, was used for rendering stuff.
* **Primitive 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?**
  + 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

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

# Shader Object

* User-defined/fixed function of a Shader Program Object.
* Used for temporarily manipulating the Vertex, Primitive or Fragment Struct.
* Written in GLSL.
* **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.
* **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 Rendering Pipeline.

# GLSL

* Shader Object Language inspired by C/C++.
* Every shader must have entrypoint function "void main()".
* GLSL version must be defined by using preprocessor "#version" at first line.
* You will frequently send datas from application to GLSL by using OpenGL functions.
* **Preprocessor Directives**
  + #versioninteger\_version\_number support\_type
    - integer\_version\_number
      * Can be found in their documentation.
    - 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.
    - uniform
      * const value that stays for a draw call sent from your application.
    - in
      * The input value sent from your application or previous step in Rendering Pipeline.
        + Example: Vertex Attribute in Vertex Shader.
    - out
      * The output value.
        + Example: Vertex Attribute in Vertex Shader.
    - inout
      * Combination of in and out.
  + 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 (Number)
    - 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
    - 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
    - Two methods can be used for storing multidimensional arrays in memory
      * Example:

|  |  |  |
| --- | --- | --- |
| Address | Row-Major | Column-Major |
| 0 | x11 | x11 |
| 1 | x12 | x21 |
| 2 | x13 | x12 |
| 3 | x21 | x22 |
| 4 | x22 | x13 |
| 5 | x23 | x23 |

* + - GLSL uses Column-Major.
      * I will use Row-Major so we will tell GLSL to transpose (convert) while sending data from application to GLSL.
    - In math, you define as ROWxCOLUMN.
    - In GLSL, you define as COLUMNxROW.
      * 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**
  + Happens once when Shader Program Object is compiled/used.
    - 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.

# Draw Call

* Function for drawing Primitives by starting the Rendering Pipeline using Vertex datas you prepared.
* You will specify the wanted Primitive type here.
* VBO
  + Vertex array for Primitive(s).
    - Example
      * Triangle VBO ‘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 VAO.
* VAO
  + Indexed format of Vertex and connection to VBO data.
  + In a Shader Object, you will connect to VBO using VAO indexes.
  + You are capable of taking a different Vertex attribute from different VBO.
    - Example
      * Triangle VBO ‘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)]
      * Triangle VAO ‘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’ .
* VEO
  + Vertex index array that creates a connection to VBO in specified order.
  + VAO uses that.
  + Used for sharing a Vertex between Primitives.
    - Example  
      Imagine a flat surface that consists of 2 triangles.  
      Normal: 2 Triangle = 6 Vertex  
      Indexed: 2 Triangle = 4 Vertex.
* **Ordered Draw**
  + Using VBO and VAO to draw.
* **Indexed Draw**
  + In addition to Ordered Draw, use of VEO to draw.
  + More performant because Vertexes are shared.

# Rendering Pipeline

* Sequence of Shader Objects 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 999 Primitive goes somewhere.  
      YES: 1 Primitive will take 999 Primitive one by one and goes somewhere.
* **How pipeline works?**
  + Every Shader Object has input and output 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.
* **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) into a specific VBO.
  + 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.
* **Preview of Pipeline**
  + **Prepare and invoke draw call**
  + **Vertex Shader**
  + **(If Primitive Type is Patch)**
    - **(Optional, default used) 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 and invoke draw call** 
   * Main Purpose: User calls draw call after preparing Vertexes.
   * Process
     1. Prepare Vertexes and call draw call.
     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.
    * Process For: 1 Fragment
      1. Manipulate Fragment.
      2. Output.
    * Output
      1. 1 Fragment.
14. **Per-Sample Operations**
    * Main Purpose: Test and write Fragment to Frame Buffer as pixels.
    * Input
      1. 1 Fragment.
    * Process For: 1 Fragment:
      1. Pixel ownership test: Fails if the Fragment'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’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’s Vertexes.
  + Front Face (GL\_CCW) = Counter clockwise order of Primitive’s Vertexes.

# 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.

# Other

* Colors in OpenGL are RGBA.
* Diffuse color
  + Fragment 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.
* Mesh/Model
  + A list of Vertex that creates triangles.

# 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.
  + Compiled source swaps with new source every time linked/compiled.
  + 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.
* glGetUniformLocation
  + Gets GLSL uniform variable layout location index by name from a linked Shader Program Object.
* glGetAttribLocation
  + Gets GLSL in variable layout location index by name from a linked Shader Program Object.
* glUniform\*
  + Sets uniform value.
  + If there is a “v”, it means “vector” and you can set more than 1 uniform value.
* glVertexArrayElementBuffer
  + Binds veo to vao.