Contents

[primitives 2](#_Toc138176696)

[OpenGL rendering pipeline 3](#_Toc138176697)

# primitives

In computer graphics, primitives refer to basic geometric shapes or elements used to construct more complex objects or scenes. Primitives serve as the building blocks of a 3D model or a 2D image. They define the fundamental shape and structure of objects within a computer graphics system.

In the context of the OpenGL rendering pipeline, the most commonly used primitives are points, lines, and triangles. These primitives are defined by a set of vertices (points) and are assembled together to form more complex geometry.

Here's a brief explanation of each primitive type:

1. Points:
   1. The simplest primitive consisting of a single vertex.
   2. Represented as a single point in space.
   3. Usually used for representing particles, individual vertices, or simple effects.
2. Lines:
   1. A primitive consisting of two vertices connected by a straight-line segment.
   2. Represented as a line segment in space.
   3. Lines can be used to represent wireframes, outlines, or simple shapes.
3. Triangles:
   1. The most common primitive used in 3D graphics.
   2. Consists of three vertices forming a planar surface.
   3. Represented as a flat polygon with three sides.
   4. Triangles are versatile and can represent any 3D surface when assembled together.

By combining these basic primitives, complex shapes and objects can be constructed. More advanced graphics techniques, such as tessellation, subdivision, or procedural generation, can further refine and manipulate these primitives to achieve more intricate and detailed geometry.

It's important to note that while points, lines, and triangles are the most commonly used primitives, there are other types of primitives available in some graphics APIs or extensions, such as quads or polygons with more than three sides. However, triangles remain the standard choice due to their simplicity and efficiency in rendering.

# OpenGL rendering pipeline

The OpenGL rendering pipeline, also known as the OpenGL graphics pipeline, is a series of stages through which geometry and other data pass to generate a final rendered image on the screen. The pipeline consists of several stages, each responsible for different tasks. Here's an overview of the OpenGL rendering pipeline:

1. Vertex Specification:
   1. Application code provides vertex data, such as position, color, texture coordinates, etc.
   2. Vertices are typically defined in object space.
   3. Vertex data is stored in vertex buffers, which are arrays of data in GPU memory.
2. Vertex Shader:
   1. Executes on each vertex of the input geometry.
   2. Transforms vertices from object space to clip space.
   3. Performs operations like transformations, lighting calculations, and other per-vertex computations.
   4. Outputs the transformed vertices along with any additional data (e.g., colors, texture coordinates) to the next stage.
3. Primitive Assembly:
   1. Takes the transformed vertices and assembles them into geometric primitives, such as points, lines, or triangles.
   2. Primitives are defined in clip space, which means their coordinates are specified within a normalized coordinate system (-1 to 1 in all dimensions).
4. Tessellation (Optional):
   1. Subdivides geometric primitives into smaller primitives.
   2. Used for more detailed geometry representation, such as curved surfaces or displacement mapping.
   3. Tessellation control and evaluation shaders are used to control the subdivision and generate new vertices.
5. Geometry Shader (Optional):
   1. Operates on entire primitives (e.g., a whole triangle).
   2. Generates or modifies primitives, allowing for tasks like generating new geometry, creating additional primitives, or performing procedural effects.
   3. Allows for operations such as particle systems, wireframe rendering, or shadow volume generation.
6. Compute shader (Optional):

A compute shader is a type of shader in modern graphics APIs, such as OpenGL and DirectX, that allows for general-purpose computation on the GPU (Graphics Processing Unit). Unlike other shader stages in the rendering pipeline (e.g., vertex shader, fragment shader), which are primarily used for processing graphics-related tasks, the compute shader is designed for performing highly parallel computations on the GPU.

Here are some key points about compute shaders:

1. Purpose:

- The main purpose of a compute shader is to perform non-graphical calculations and data processing on the GPU.

- It is not directly involved in the rendering of images or geometry.

2. Execution Model:

- Compute shaders execute in parallel on the GPU across many threads, allowing for massive parallelism.

- Each compute shader thread is relatively independent and can perform computations on its own portion of the data.

3. Data Processing:

- Compute shaders operate on data stored in buffers or images.

- Buffers are typically used for general-purpose data, while images are often used for more specialized image processing tasks.

- Compute shaders can read from and write to these data structures.

4. Applications:

- Compute shaders have a wide range of applications, including physics simulations, data processing, computational photography, artificial intelligence, and more.

- They are particularly useful for tasks that can benefit from parallelism and harnessing the computational power of modern GPUs.

5. Interaction with the Rendering Pipeline:

- Compute shaders can be used independently of the rendering pipeline.

- However, they can also interact with other shader stages or graphics resources.

- For example, a compute shader can perform pre-processing tasks, generate or modify data used by other shaders, or update textures or buffers during the rendering process.

Compute shaders offer a flexible and powerful tool for performing computation on the GPU. They allow developers to offload complex and parallelizable tasks to the graphics hardware, resulting in significant performance improvements for certain types of applications.

1. Clipping:
   1. Discards primitives that fall outside the view frustum (the visible portion of the scene).
   2. Primitives are defined in clip space.
   3. Clipping ensures that only the visible parts of the scene are further processed, improving efficiency.
2. Rasterization:
   1. Converts the clipped primitives into fragments (pixels) that will be rendered on the screen.
   2. Determines which fragments are covered by the primitives.
   3. Interpolates the attributes (e.g., colors, texture coordinates) of the vertices across the fragments to be processed in the fragment shader.
3. Fragment Shader:
   1. Executes on each fragment produced by rasterization.
   2. Performs per-fragment operations, such as texture mapping, lighting calculations, and color interpolation.
4. Per-Sample Operations:
   1. Performs operations like depth testing, stencil testing, and alpha blending.
   2. Determines the final color and depth values for each fragment.
5. Framebuffer Operations:
   1. Fragments are written to the framebuffer (the image displayed on the screen).
   2. Operations like double buffering and swapping the front and back buffers occur here.

Throughout the pipeline, various data can be passed between stages using variables and buffers. The pipeline is highly parallel and allows for efficient utilization of modern GPUs. It enables the transformation, shading, and rendering of 3D objects and scenes.

It's worth noting that with the introduction of modern graphics APIs like Vulkan and DirectX 12, the concept of a fixed-function pipeline has been replaced with more programmable and flexible approaches, allowing developers to have more control over the rendering process.

# VAO & VBO

Vertex Array Objects (VAOs) and Vertex Buffer Objects (VBOs) are in the context of OpenGL:

1. Vertex Array Objects (VAOs):

- A VAO is an OpenGL object that encapsulates the configuration of vertex attributes for rendering.

- It acts as a container or a binding point for all the necessary state related to vertex data.

- VAOs store references to Vertex Buffer Objects (VBOs) that hold the actual vertex data.

- VAOs allow for efficient switching between different sets of vertex data and attribute configurations.

2. Vertex Buffer Objects (VBOs):

- A VBO is an OpenGL object used to store vertex data, such as positions, colors, normals, or texture coordinates.

- VBOs serve as a high-performance buffer to hold large amounts of vertex data on the GPU.

- The data stored in VBOs can be accessed and processed by the GPU during rendering operations.

- VBOs can be used for both static and dynamic vertex data, depending on how frequently the data needs to be modified.

In summary, VAOs and VBOs are crucial components in modern OpenGL rendering pipelines for organizing and managing vertex data. Here's a breakdown of their roles and interactions:

1. Setting up VBOs:

- Create a VBO using `glGenBuffers()`.

- Bind the VBO using `glBindBuffer()` to specify the target buffer type (e.g., `GL\_ARRAY\_BUFFER`).

- Upload vertex data to the VBO using `glBufferData()` or `glBufferSubData()`.

2. Setting up VAOs:

- Create a VAO using `glGenVertexArrays()`.

- Bind the VAO using `glBindVertexArray()`.

- Enable and configure vertex attribute pointers using `glVertexAttribPointer()` and `glEnableVertexAttribArray()`.

- Associate VBOs with attribute indices within the VAO.

3. Rendering with VAOs and VBOs:

- Bind the VAO using `glBindVertexArray()` to activate the attribute configurations and associated VBOs.

- Issue draw calls (`glDrawArrays()` or `glDrawElements()`) to render the geometry.

The separation of VAOs and VBOs allows for efficient reuse of vertex data and attribute configurations. Once the VAO and VBO configurations are set up, you can bind the VAO and issue draw calls to render the geometry without having to repeatedly specify the vertex attributes.

VAOs and VBOs are essential tools for managing vertex data and rendering efficiently in modern OpenGL applications. They enable you to organize and access vertex data on the GPU, improving performance by reducing redundant data transfers and attribute configurations.

# IBO

An Index Buffer Object (IBO) is an OpenGL object used to store indices that define the order in which vertices are rendered. It is also known as an element array buffer. When rendering complex 3D models or scenes, vertices are often reused to form triangles or other primitive shapes. Instead of duplicating vertex data for shared vertices, an index buffer is used to reference the vertices in a specific order, reducing memory usage and improving rendering performance.

Here's how an IBO works in conjunction with VAOs and VBOs:

1. Vertex Array Object (VAO):
   * A VAO encapsulates the configuration of vertex attributes for rendering.
   * It can store references to one or more VBOs, including an IBO.
2. Vertex Buffer Object (VBO):
   * A VBO holds vertex data such as positions, colors, normals, or texture coordinates.
   * It can be associated with a VAO and used to provide vertex attribute data.
3. Index Buffer Object (IBO):
   * An IBO stores indices that represent the order of vertices to form triangles or other primitives.
   * It can be associated with a VAO and used to provide the index data for rendering.

The typical process of using an IBO involves the following steps:

1. Create and bind a VAO using glGenVertexArrays() and glBindVertexArray().
2. Create and bind a VBO using glGenBuffers() and glBindBuffer().
3. Upload vertex data to the VBO using glBufferData() or glBufferSubData().
4. Create and bind an IBO using glGenBuffers() and glBindBuffer().
5. Upload index data to the IBO using glBufferData() or glBufferSubData().
6. Configure the vertex attribute pointers using glVertexAttribPointer() and glEnableVertexAttribArray().
7. Bind the VAO and issue draw calls (glDrawArrays() or glDrawElements()) to render the geometry.

When rendering with an IBO, the indices stored in the IBO determine the order in which the vertices are rendered. This allows for reusing vertices and referencing them by their index, resulting in more efficient memory usage and rendering performance.

In summary, an IBO (Index Buffer Object) is an OpenGL object used in conjunction with VAOs and VBOs to store indices that define the order of vertices during rendering. It helps reduce memory usage and optimize rendering performance by allowing efficient sharing and referencing of vertices.

# فارسیش

## مقدمه :

ابتدا از نام OpenGLشروع می کنیم که نشان دهنده مفهوم آن نیز می باشد ، OpenGL مخفف کلمات Open source Graphic Library می باشد، خوب ابتدا مختصری درباره نرم افزار های open sourceیا همان متن باز توضیح می دهیم :

حتما با نرم افزار های متن باز یا به اصطلاح اصلی open sourceآشنایی دارید و می دانید که این نرم افزار ها به صورت رایگان دراختیار عموم قرار می گیرند و تهیه کننده گان آنها سورس اصلی برنامه را بیشتر به منظور عیب یابی و کاهش خطا ها و bugهای احتمالی برنامه در اختیار برنامه نویسان قرار می دهند تا برنامه نویسان و علاقمندان بتوانند به راحتی کد آن برنامه را ویرایش کرده و تغییرات دلخواه را روی آن انجام دهند. (یکی از معروفترین برنامه های open sourceسیستم عامل لینوکس است و البته پیشرفت ها و موفقیت های جدید آن را، که مهمترین عامل این پیشرفت ها همین متن باز بودن لینوس بوده و است.)

openGLهم همانطور که از نام آن مشخص است جزه نرم افزار متن باز به شمار می رود که یک از مزیت ها و امتیازات آن نیز به شمار می رود.

حال نوبت به graphic libraryمی رسد:

graphic libraryیا همان کتابخانه گرافیکی به مجموعه کنابخانه ها یا کامپوننت های گفته می شود که برای استفاده از دستورا مخصوص گرافیکی تهیه شده است و با الصاق includeآن ها به برنامه اصلی ( هنگام برنامه نویسی ) می توانیم ، از دستورات گرافیکی مجاز برای ایجاد رابطه با بخش گرافیک سیستم استفاده نماییم ( از جمله کار هایی که با گرافیک سیستم سر و کار دارد نوشتن بازی ها و اشکال سه بعدی و دو بعدی و کلا برنامه هایی که با مانیتور در ارتباط باشند.)

کتابخانه ها یا همان فایل های با پسوند libکه مخفف شده libraryمی باشد ، به منظور آسان تر شدن کار برنامه نویسان در موقع برنامه نویسی ایجاد و تهیه می شوند، به این صور که مثلا برنامه نویسی داخل برنامه های خود از تابع خاصی چندین بار شاید هم صد ها و هزاران بار) به صورت متوالی استفاده می کند ، خوب نوشتن این تابع خاص به این تعداد در کد اصلی برنامه حجم فایل نهایی را بالا برده و البته وقت برنامه نویس را هم خواهد گرفت ، اینجاست که کتابخانه ها یا libها وارد عمل می شوند و تابع خاص را داخل خود قرار می دهند تا موقع نیاز به تابع از آن استفاده شود .

Open source Graphic یا همانOpenGL ، حال شما با مفهوم اسمgraphic library این هم از مفهوم Libraryآشنایی کامل را دارید . پس دراینجا نتیجه می گیریم که OpenGLیک زبان برنامه نویسی نیست ، یک سری کتابخانه از پیش تعریف شده می باشد که کار این کتابخانه ارتباط راحت تر با بخش سخت افزاری گرافیک سیستم می باشد ، برای درک بیشتر این مطلب باید توضیحی هم درباره رابط های نرم افزاری یا همان APIها بدهیم :

برای اینکه بتوانیم با سخت افزار های موجود در سیستم خود مثل کارت شبکه ، کارت گرافیک ، ... به صورت درست و صحیح ارتباط برقرار کنیم (به زبان خود ماشین )احتیاج به رابط هایی داریم که بتوانند این کار را به درستی انجام دهند ، فرض کنید یک فارسی زبان ( که زبان دیگری هم نمی داند) با یک انگلیسی زبان ( که او هم زبان فارسی را نمی داند ) برخورد می کنند و می خواهند با هم صحت کنند ، به نظر شما چاره کار چیست ؟ خوب اولین و تنها گزینه یک مترجم می باشد که هم زبان فارسی بداند و هم زبان انگلیسی ، حال آن دو به راحتی می توانند با هم مکاتبه داشته باشند . در مورد کامپیوتر هم دقیقا به همین صورت است ، نرم افزار و سخت افزار کامپیوتر زبان هم دیگر را نمی دانند و احتیاج به یک سری مترجم هایی به نام APIدارند .

APIها رابط های نرم افزاری هستند که زبان سیستم ( زبان ماشین ) و چگونگی ارتباط با سخت افزار را می دانند . به عنوان مثال من برنامه نویس اگر بخواهم یک پیکسل از مانیتور را روشن کنم به APIمربوطه فرمان می دهم که این کار را ( با زبانی که بلد هستی ) برای من انجام بده و مستقیم با سخت افزار صحبت نخواهم کرد، APIمربوطه دستور را گرفته و به سخت افزار می رساند ، به همین راحتی ...

APIهای مختلفی برای بخش های مختلف سخت افزار وجود دارد، اما دراینجا و این مقاله APIهای گرافیکی مد نظر ما می باشد ، که معروفترین آنها OpenGlو DirectXمی باشند . البته سخت افزار های موجود نیز باید از APIها پشتیبانی کنند، که بعضی شرکت های سازنده سخت افزار و کارت های گرافیکی از APIهای OpenGLپشتیبانی و همایت می کنند و بعضی هم از DirectXکه محصول شرکت ماکروسافت است . هر یک از این واسط های نرم افزاری OpenGlو DirectXمعایب و مزایای خواص خود را دارند که دراینجا قصد مقایسه این دو را ندارم .

تاریخچه و نحوه تکامل : OpenGl

سازنده OpenGLیا بهتر بگوییم توسعه دهنده و بهینه کننده آن شرکت Silicon Graphicsمی باشد (که به اختصار SGI نیز نامیده می شود و یکی از بزرگان صنعت گرافیک کامپیوتری می باشد )، خود OpenGLتوسعه یافته کتابخانه دوبعدی IRIS GLکه آن هم محصول شرکت SGLبود، می باشد . شرکت های دیگری نیز به توسعه این محصول متن باز کمک کرده اند از جمله شرکت D Labs3که تکنیک و قابلیت سایه زنی را به OpenGLاخافه نمود . در سالهای اخیر کنابخانه های OpenALو OpenILنیز به موازات OpenGLدر حال پیشروی و کسترش هستند که اولی برای کنترل و مدیریت سخت افزار های صوتی ( )Audioو ایجاد صدا های سه بعدی و دومی برای مدیریت و کنترل دستگاه های ورودی ( ) Inputها ایجاد شده اند .

رقیب اصلی OpenGLهمان DirectXمحصول شرکت Microsoftمی باشد که در سال 1995وارد عرصه رقایت API ها شد، دراوایل رقابت بین این دو این جی ال کاملا یک سر و گردن از حریف خود جلو بود، اما کار به همین روال پیش نرفت و شرکت ماکروسافت یا استفاده از قدرت تجاری خود توانست از رقیب خود جلو بزند، البته متن باز بودن OpenGLهم بی تقصیر نبود به گونه ای که ماکروسافت ازاین ویزگی استفاده می کرد و محصولات خود را تقریبا مشابه با محصولات OpenGLارائه می داد تا اینکه در تاریخ 2000میلادی DirectX 8منتشر شد که امکانات بیشتر و بهتری نسبت به OpenGLداشت ، دراین برهه زمانی بود که OpenGLاز رقیب خود عقب افتاد ولی همچنان رقابت بین این دو ادامه دارد.

## انواع گرافیک رایانه ای :

دو نوع گرافیک رایانه ای وجود دارد: نوع اول گرافیک رستری Raster که به گرافیک Bitmapنیز شهر دارد. گاهی نیز به گرافیک رستری گرافیک پیکسلی هم می گویند. نوع دوم آن گرافیک وکتورVectorمی باشد که به نوع برداری" نیز شهرت دارد.

### گرافیک پیکسلی یا رستری

این نوع گرافیک تشکیل شده است از خانه های پیکسلی که دارای درجه مشخصی از یک رنگ می باشند و وقتی این پیکسل ها در کنار یکدیگر با درجه رنگهایی مختلف از هم، قرار می گیرند، تصاویر را تشکیل می دهند. در گرافیک پیکسلی به دلیل اینکه شما با رنگ پیکسل ها سرو کار دارید، قادر به ویرایش، حذف یا ایجاد تصویر یا عکس نیستید. تنها می توانید عکس یا تصاویر دیجیتال را توسط دسترسی به پیکسل‌های رنگی آن، ویرایش نمایید. آنچه در گرافیک رستری اهمیت دارد، تعداد پیکسل های رنگی در اینچ می باشد، زیرا این تعداد، کیفیت رنگی و وضوع تصویری عکس دیجیتال را مشخص می کند. به تعداد پیکسل های رنگی در اینچ، رزولوشن می گویند. و مقدار رزولوشن، همان مقدار پیکسل ها را در واحد اینچ مشخص می کند که اصطلاحا به آن واحد dpiمی گویند (یعنی Dot Per Inchیا تعداد نقاط دراینچ.)همین واحد dpiاست که باعث می شود وقتی شما یک تصویر رستری را بزرگ نمایی می کنید، کیفیت تصویر پایین تر می آید. زیرا هرچه هم که آنرا بزرگ تر کنید نمی توانید رزولوشن یا تعداد پیکسل های آن را در واحد اینچ افزایش دهید. کیفیت این تصاویر با سایز ارتباط مستقیم دارد.

### گرافیک وکتور یا برداری

این نوع گرافیک، گرافیکی رایانه ای است که با فرمولهای ریاضی سر و کار دارد. از خط ها، منحنی ها و اشکالی که دارای بعد و مختصات ریاضی می باشند و مکان آنها با Xو yتعریف می گردد. دارای طول و عرض ریاضی می باشند و رزولوشن آنها با تغییر سایز، تغییر نمی یابد. بنابراین کیفیت تصاویر در هر سایزی یکسان می ماند و این کار از طریق محاسبات ریاضی انجام می شود.این گرافیک یکی از بهترین و پرکاربردترین انواع گرافیک رایانه است، برای کار تصویر سازی استفاده می شود و طراحی آرم ها و گاهی تصویر سازی کتاب های کودک و ... توسط برنامه های وکتوری انجام می شود

### قوانین نوشتاری در openGL

* دستورات openGLبا پیشوند glشروع میشود و با یک حرف بزرگ ادامه می یابند مانند دستورهای glutDisplayFunc( ) و یاglBegin()
* ثوابت با پیشوند GL\_ شروع میشود و تمام حروف بعد از آن حروف بزرگ است مانند: GL\_COLOR\_BUFFER\_BIT
* بعضی از دستورات نیز با یک عدد وچندین حرف درانتها برای یاداوری تعداد و نوع آرگومانهای ورودی خاتمه می یابند . برای مثال در دستورglVertex3fv به معنای وجود 3 آرگومان ورودی و f به معنای اعشاری بودن ورودی ها و V به معنای برداری بودن ورودی ها است .



