CG Q&A

Aditya Raj

February, 2026

Contents

1 Background of Computer Graphics

1.1 Purpose of Computer Graphics

Computer graphics serves several key purposes:

- **Visualization:**
  Makes complex data and concepts easier to understand (charts, diagrams, simulations, medical imaging).
- **Communication:** Conveys ideas more effectively than text alone (presentations, user interfaces, instructional materials).
- **Entertainment:** Produces content for movies, video games, animation, and virtual experiences.
- **Design and modeling:** Supports CAD and 3D modeling to design, test, and refine products before physical production.
- **Simulation and training:** Creates realistic environments for training (flight simulators, surgical practice).
- **User interaction:** Enables intuitive interfaces for interacting with digital systems.

In short, it *bridges digital information and human perception*, making technology more accessible, useful, and engaging.

1.2 First Computer Graphics System

The first computer graphics on a digital computer system were created on the **Whirlwind computer (MIT, 1951)**. It displayed *vector graphics* and used an early input device similar to a *light pen*, allowing direct interaction with the screen.

Whirlwind is therefore one of the earliest examples of **interactive computer graphics** on a true digital computing system.

1.3 First Fully Computer-Animated Film

The first fully computer-animated feature film is ***Toy Story* (1995)**, released by Pixar and Disney. While there were earlier uses of **computer-generated imagery (CGI)** in films (e.g., *Tron* (1982) and *Young Sherlock Holmes* (1985)), *Toy Story* was the first feature-length film created *entirely* using computer animation. It marked a major milestone in animation history and helped pave the way for modern CGI films.

1.4 Uncanny Valley

The **uncanny valley** is the unsettling feeling people experience when a character looks *almost—but not quite—human*. In computer graphics, this happens when a digital character is highly realistic (face, skin, movement) but still has subtle imperfections that feel unnatural. Small discrepancies (slightly off expressions, unnatural eye motion, mismatched proportions) can trigger discomfort or unease.

The term was introduced by Japanese roboticist **Masahiro Mori (1970)**. He observed that as robots or avatars become more human-like, emotional affinity increases—until a point where they look nearly human but still fall short. At that point, the observer's response drops into discomfort, forming a *"valley"* in a graph of human likeness vs. emotional response.

This is especially relevant in **film**, **games**, and **virtual environments**. For example, *The Polar Express* (2004), *Final Fantasy: The Spirits Within* (2001), and the 2019 *Cats* adaptation were criticized for lifelike yet unsettling characters. Game designers often avoid hyper-realistic humans and use stylized designs to maintain immersion.

To reduce the uncanny valley, designers focus on **consistency** across facial expressions, motion, voice, and proportions so that all cues align with realistic human behavior. Others deliberately choose *stylization* instead of photorealism to create more relatable characters.

2 Main Steps in Computer Graphics

2.1 Abstract Steps to Create Graphics

The main abstract steps in creating a computer graphic are:

- **Modeling:** Defining the objects and components of a scene in terms of shape, size, color, texture, and other parameters. This involves creating a digital representation of the 3D or 2D elements using primitives or detailed geometry.
- **Scene setup:** Arranging virtual objects, lights, cameras, and other entities in a virtual environment. This includes positioning and orienting elements to create a coherent composition (and keyframing for animation when needed).
- **Rendering:** Generating the final 2D image (or animation) from the prepared scene by applying lighting, shading, textures, and effects using algorithms such as ray tracing or scanline rendering.

These stages form the core of the graphics pipeline, which transforms abstract scene data into a visual image displayed on a screen.

2.2 Graphics Pipeline Stages

The **graphics pipeline** is a sequence of stages that transforms **3D scene data** into a **2D image** for display. It operates on primitives (vertices, triangles) using **GPUs** and graphics APIs (e.g., *OpenGL*, *Vulkan*) for real-time rendering.

Main processing steps and computations:

1. **Vertex Processing**
   - **Computations:** Transform vertices from local object space to screen space using 4×4 transformation matrices (modeling, viewing, projection).
   - **Key operations:** Per-vertex calculations such as position, normal, and color transformations.
   - **Programmable:** Vertex shaders allow custom logic (e.g., procedural deformation, animation).

2. **Tessellation** *(optional)*
   - **Computations:** Subdivide coarse geometry into finer meshes using Tessellation Control Shaders (TCS) and Tessellation Evaluation Shaders (TES).
   - **Purpose:** Dynamic level-of-detail and smooth surfaces (e.g., subdivision surfaces).

3. **Geometry Processing** *(optional)*
   - **Computations:** Modify or generate new primitives (e.g., convert points to triangles, create particles).
   - **Programmable:** Geometry shaders process whole primitives and can output new ones.

4. **Primitive Assembly & Clipping**
   - **Computations:**
     Group vertices into primitives (triangles, lines, points), then clip primitives outside the view frustum.
   - **Perspective division:** Maps coordinates to normalized device coordinates (NDC).

5. **Rasterization**
   - **Computations:** Convert primitives into fragments (potential pixels).
   - **Key tasks:** Determine pixel coverage and interpolate vertex attributes (color, texture coordinates, depth) across the primitive.

6. **Fragment Processing**

- **Computations:** Compute final pixel color per fragment using fragment shaders.
- **Operations:** Apply lighting, texturing, shadows, reflections, and material effects.
- **Highly parallel:** Each fragment is processed independently.

7. **Per-Sample Operations**
   - **Computations:** Final decisions before writing to the framebuffer.
   - **Key operations:**
     - Depth (Z) testing (via Z-buffering) to resolve visibility.
     - Stencil testing for masking.
     - Blending for transparency.
     - Logical operations and write masks.

This pipeline evolved from *fixed-function* hardware (1990s) to *programmable* stages (post-2001), enabling complex visual effects. Modern GPUs extend this with **ray tracing** (e.g., NVIDIA RTX) and **AI-accelerated denoising**, blending rasterization with newer rendering paradigms.

3 Requirements in Computer Graphics

3.1 Interactive vs. Non-Interactive

Interactive computer graphics involve **real-time** user engagement and manipulation of visual content. Users interact through input devices (keyboard, mouse, touchscreen, VR controllers), and the system responds dynamically. Examples include video games, VR simulations, flight simulators, CAD systems, and interactive data visualizations.

Non-interactive computer graphics are **pre-generated** and static, with no real-time user control. The content is fixed and displayed as-is, and the user can only view the output. Examples include digital images, movies, slide shows, and static website graphics.

Key difference: *user interaction*. Interactive graphics respond immediately to user input, while non-interactive graphics are fixed once rendered.

3.2 Requirements for Realistic Rendering (Interactive vs. Offline)

Realism in non-interactive (offline) computer graphics

For non-interactive computer graphics—such as pre-rendered movie scenes or high-quality visualizations—**image quality** is prioritized over speed. Key requirements include high computational power, advanced rendering algorithms, and accurate physical modeling. This enables techniques like **ray tracing**, **global illumination**, **physically based rendering (PBR)**, subsurface scattering, and photon mapping. The **rendering equation** (Kajiya, 1986) is foundational for modeling real-world lighting. Offline pipelines also rely on shaders (HLSL/GLSL) for effects like bump mapping, normal mapping, and reflections.

Realism in interactive (real-time) computer graphics

In interactive systems (games, simulations, VR), **real-time performance** is the priority. The main challenge is balancing realism with fast rendering and low latency. Requirements include an efficient graphics pipeline, GPU hardware acceleration, and optimized algorithms. Most real-time systems rely on **rasterization** for speed, supported by APIs such as OpenGL, DirectX, and Vulkan. Techniques like depth buffering, level-of-detail (LOD), normal mapping, and shadow mapping help maintain realism with smooth frame rates.

Shared requirements

Both domains rely on geometric primitives (lines, polygons, curves), transformations (translation, rotation, scaling), clipping, texture mapping, and correct color handling. The key difference is the trade-off: offline systems maximize realism; interactive systems maximize responsiveness.

3.3 Real-Time vs. Offline Rendering

Real-time computer graphics generates and displays frames fast enough to create the illusion of motion and support immediate interaction. Typical targets are 30–60 FPS (or higher). Real-time rendering uses GPU-accelerated rasterization and performance-friendly techniques, sometimes sacrificing photorealism to maintain speed.

Offline (non-real-time) rendering prioritizes maximum quality and realism and may take minutes to hours per frame. It can use computationally expensive techniques such as global illumination, ray tracing, and path tracing, and is common in feature films and high-end visualization.

Summary:

- **Real-time:** Speed first; interactivity second $\rightarrow$ used in games and VR.
- **Offline:** Quality first; speed second $\rightarrow$ used in movies and high-end VFX.

3.4 Hard vs. Soft Real-Time

Hard real-time in computer graphics means missing a deadline (e.g., failing to render a frame within a strict time limit) is unacceptable and can cause unsafe or catastrophic outcomes. Examples include safety-critical simulators or medical/aerospace systems.

Soft real-time means missing a deadline degrades quality (e.g., dropped frames, stutter) but the system continues operating. This is typical in games and multimedia playback.

Summary:

- **Hard real-time:** Deadline misses are unacceptable (e.g., safety-critical medical/aerospace systems).
- **Soft real-time:** Deadline misses degrade quality but the system continues (e.g., games, video playback).

4 General Information about OpenGL

4.1 What is the difference between the OpenGL programming interface specification and an OpenGL implementation?

The **OpenGL specification** is a technical document developed and maintained by the Khronos Group that defines the exact behavior, functions, and expected outputs of the OpenGL API. It is not a software product but a standard that outlines what OpenGL should do, without specifying how it should be implemented. The specification describes the API's functions, constants (like `GL_TEXTURE_2D`), and the state-machine model, focusing on results rather than implementation details.

An **OpenGL implementation** is the actual software that brings the specification to life. It is developed by hardware vendors (NVIDIA, AMD, Intel) and/or operating-system vendors, and it translates OpenGL API calls into GPU/driver-specific commands.

Implementations vary by platform:

- **Windows:** Usually provided by graphics card vendors (IHVs) via the driver.
- **macOS:** Controlled by Apple.
- **Linux:** Provided by vendor drivers and open-source stacks such as Mesa.

The key difference is that the specification is the *rulebook*, while the implementation is the *software* that follows those rules. As long as an implementation adheres to the specification, OpenGL programs behave consistently across diverse hardware and operating systems.

4.2 What are the benefits of an open and standardised interface specification?

- **Interoperability:** Different systems and vendors can work together using the same contract.
- **Lower cost and faster integration:** Standard interfaces reduce custom glue code, testing effort, and maintenance.
- **Competition and choice:** Open standards lower entry barriers and prevent lock-in to a single vendor.
- **Longevity and sustainability:** Standard ports and formats make products easier to repair, reuse, and upgrade.

- **Quality and security:** Clear, stable specifications reduce ambiguity and integration errors and make auditing/testing easier.

4.3 What makes an OpenGL implementation work on a computer?

An OpenGL implementation works on a computer through a combination of drivers, GPU acceleration, and an OS graphics stack.

- **Graphics drivers:** Hardware vendors provide drivers that implement the OpenGL specification and translate API calls into GPU commands.
- **Hardware acceleration:** Most rendering work runs on the GPU (highly parallel), which enables real-time performance.
- **OS integration:** The window system and OS graphics stack provide context creation and presentation (showing the framebuffer on screen).
- **Software fallback (rare):** If no suitable GPU/driver is available, a software renderer may run on the CPU, but performance is typically poor.

4.4 In which programming language is OpenGL specified?

OpenGL is not a programming language; it is a cross-language, cross-platform graphics API. The specification is written in technical prose and defines a C-based API surface.

In practice:

- **API:** Defined as a C API and commonly used from C/C++.
- **Bindings:** Available for many languages (Java, Python, JavaScript, etc.).
- **Shaders:** Written in GLSL (OpenGL Shading Language), which has a C-like syntax.

4.5 Please explain what is JOGL and what is LWJGL.

JOGL (Java Binding for OpenGL) is an open-source library that provides Java bindings for the OpenGL graphics API. It allows Java applications to access OpenGL's full range of 2D and 3D rendering capabilities, including support for OpenGL versions 1.0 through 4.3, as well as OpenGL ES 1, 2, and 3. JOGL is maintained by JOGAMP and integrates well with Java's AWT and Swing GUI frameworks, enabling OpenGL rendering within standard Java components like JFrame or GLCanvas. It also includes a Native Windowing Toolkit (NEWT) and supports features like Java2D/OpenGL interoperability and debugging tools such as DebugGL, which throws exceptions at the point of OpenGL error.

LWJGL (Lightweight Java Game Library) is a Java library that provides low-level access to native APIs such as OpenGL, OpenAL, and OpenCL, along with input and windowing support via GLFW. Designed primarily for game development, LWJGL offers a thin, efficient wrapper around native code, making it ideal for performance-critical applications. It supports full-screen mode natively, offers better cross-platform reliability, and includes built-in support for game controllers (via JInput), audio (OpenAL), and modern OpenGL features. Unlike JOGL, LWJGL does not integrate with AWT/Swing and uses a callback-based event model, which gives developers more control but requires more manual setup.

In summary:

- **JOGL:** Best when you want OpenGL integrated into AWT/Swing style GUIs.
- **LWJGL:** Best when you want a thin, game-focused wrapper (often paired with GLFW) and broader native API access.

4.6 Compatibility profile vs. core profile

Explain the difference between the compatibility profile and the core profile of the OpenGL. How are the fixed-function pipeline and the programmable pipeline related to these profiles?

The **core profile** and **compatibility profile** are two modes of modern OpenGL, introduced to manage the transition from legacy features to a cleaner, shader-based API.

- **Core profile:** Removes deprecated legacy functionality (including the fixed-function pipeline). Rendering is shader-based, so you typically use at least a vertex shader and a fragment shader.
- **Compatibility profile:** Retains legacy APIs (e.g., immediate mode and fixed-function lighting/texturing) so older code can still run.

extbfRelationship to pipelines:

- The **fixed-function pipeline** exists only in the compatibility profile.
- The **programmable pipeline** (GLSL shaders) is required in the core profile and is also available in the compatibility profile.

5 OpenGL Pipeline

5.1 What is the purpose of vertex processing?

Vertex processing transforms per-vertex data into a form suitable for rasterization and display.

- **Transforms:** Applies model, view, and projection transforms to bring positions into clip space.

- **Per-vertex work:** Processes attributes (normals, colors, texture coordinates) and prepares values for interpolation.
- **Programmability:** Implemented via vertex shaders (e.g., skinning, deformation, custom lighting inputs).
- **Data flow:** Outputs vertices to primitive assembly and clipping.

Vertex processing operates on individual vertices (not whole primitives), so connectivity-dependent work happens later.

5.2 What is the purpose of rasterization?

Rasterization converts geometric primitives (triangles, lines, points) into **fragments** on a pixel grid.

- **Pixel-grid conversion:** Determines which pixels are covered by a primitive.
- **Interpolation:** Interpolates attributes (depth, color, texture coordinates) across the primitive.
- **Real-time efficiency:** GPUs are heavily optimized for rasterization, making it the default approach for interactive graphics.

Rasterization is fast, but it often uses approximations for lighting; modern engines sometimes combine rasterization with ray tracing for better realism.

5.3 What is the purpose of fragment processing?

Fragment processing determines the final output for each fragment produced by rasterization (which may or may not become a visible pixel).

- **Shading:** Fragment shaders compute final color using lighting and textures.
- **Visibility/tests:** Depth and stencil tests decide whether a fragment is kept.
- **Compositing:** Blending combines fragment output with existing framebuffer values (e.g., transparency).

Many fragments are discarded by tests, so fragment processing strongly affects both performance and visual quality.

6 Programmable OpenGL Pipeline

6.1 Which parts of the fixed-function pipeline have become programmable in the programmable pipeline?

- **Vertex processing:** Vertex transforms and per-vertex lighting/attribute generation moved into **vertex shaders**.

- **Fragment (pixel) processing:** Per-pixel shading moved into **fragment shaders**.
- **Additional stages:** Modern pipelines can also use geometry shaders, tessellation shaders, and compute shaders for specialized workloads.

Rasterization and many per-sample output operations remain fixed-function for performance, but most of the rendering logic is now controlled by shaders.

6.2 What exactly is a shader?

In computer graphics, a shader is a small program that runs on the GPU (Graphics Processing Unit) to control how pixels, vertices, or geometry are rendered in a 3D scene or 2D image. Shaders are essential for creating realistic lighting, textures, shadows, and visual effects.

Common shader stages include:

- **Vertex shaders:** Process vertices and output clip-space positions plus per-vertex attributes.
- **Fragment shaders:** Compute per-fragment color (and other outputs) using lighting and textures.
- **Geometry shaders:** Optionally generate/modify primitives.
- **Tessellation shaders:** Optionally subdivide patches into finer geometry.
- **Compute shaders:** Run general-purpose GPU computations (not necessarily tied to drawing).

Shaders are written in specialized languages like GLSL (OpenGL Shading Language) or HLSL (High-Level Shader Language) and are executed in parallel across thousands of GPU cores, enabling high-performance rendering for video games, movies, and real-time visualizations.

6.3 Which shaders do you know and which functions can they perform?

- **Vertex shaders:** Transform vertices, compute per-vertex outputs for later interpolation.
- **Fragment shaders:** Compute final per-fragment color and other outputs (e.g., normals for deferred rendering).
- **Geometry shaders (optional):** Amplify or modify primitives.
- **Tessellation shaders (optional):** Adaptive subdivision for level of detail.
- **Compute shaders (optional):** General GPU compute (physics, culling, post-processing, etc.).

6.4 Which shaders must always be present in the compatibility profile?

In the OpenGL Compatibility Profile, there are no shaders that must always be present. Unlike the Core Profile or OpenGL ES, which require a linked program with at least a vertex and fragment shader to render anything, the compatibility profile maintains the Fixed-Function Pipeline (FFP).

6.5 Which shaders must always be present in the core profile?

In an OpenGL core profile, you generally use a shader program for drawing.

- **Vertex shader:** Required for processing vertices.
- **Fragment shader:** Required for normal color rendering to a framebuffer.
- **Optional stages:** Tessellation control/evaluation, geometry, and compute shaders are used only when needed.

7 Explain the difference between a vertex, a fragment and a pixel

Term	Definition	Role in Graphics Pipeline
Vertex	A data structure that defines a point or corner in 3D space, characterized by attributes like position, color, and texture coordinates.	The initial input data (part of geometry processing) that forms the basic building blocks (vertices) of a 3D model.
Fragment	A potential pixel, essentially a data packet generated during rasterization that is located at a specific screen coordinate, containing information such as color, depth (Z-buffer value), and other attributes interpolated from the vertices of the primitive it belongs to.	An intermediate stage after geometry is converted into screen-space data (rasterization), but before final display (fragment processing).
Pixel	Short for "picture element," it is the smallest physical unit of a digital image or display device (screen).	The final, displayed element on a screen after all rendering calculations are complete.

Table 1: Graphics Pipeline Terms