Introduction to Computer Graphics

Core Concepts

The term **computer graphics** describes the use of computers to create and manipulate images. Below are the major areas of computer graphics (Marschner & Shirley, 2021):

- **Modeling** – deals with the mathematical specification of shape and appearance properties in a way that can be stored on the computer.
- **Rendering** – deals with the creation of shaded images from 3D computer models.
- **Animation** – a technique to create an illusion of motion through a sequence of images.

Rendering a scene produces a raster. A **raster** is an array of **pixels** (picture elements) displayed on a screen, arranged in a grid with two (2) dimensions. Pixels specify colors using triples of floating-point numbers between 0 and 1, which represent the amount of red, green, and blue light existing in a color; a value of 0 indicates that no amount of that color exists, while a value of 1 represents that color is displayed at full intensity. The following are various colors and their corresponding RGB values:

Color	R	G	B
Red	1	0	0
Orange	1	0.5	0
Yellow	1	1	0
Green	0	1	0
Blue	0	0	1
Violet	0.5	0	1
Black	0	0	0
White	1	1	1
Gray	0.5	0.5	0.5
Brown	0.5	0.2	0
Pink	1	0.5	0.5
Cyan	0	1	1

The quality of an image depends partly on its resolution and precision. **Resolution** is the number of pixels in the raster while **precision** is the number of bits used for each pixel.

A **buffer** (or data buffer, or buffer memory) is a part of a computer's memory that serves as temporary storage for data while being moved from one location to another. Pixel data is stored in a region of memory called the **framebuffer**. A framebuffer may contain multiple buffers that store different types of data for each pixel. At a minimum, the framebuffer must contain a **color buffer**, which stores RGB values. When rendering a 3D scene, the framebuffer must also contain a depth buffer. A **depth buffer** stores distances from points on scene objects to the virtual camera. Depth values determine whether the object's points are in front of or behind other objects (from the camera's perspective). Thus, these values can also determine whether the points on each object will be visible when the scene is rendered. Finally, framebuffers may contain a buffer called a **stencil buffer**, which may be used to store values used in generating advanced effects, such as shadows, reflections, or portal rendering.

Aside from rendering three-dimensional scenes, another goal in computer graphics is creating animated scenes. Animations consist of a sequence of images quickly displayed. Each of the images that is displayed is called a **frame**. The speed or rate at which these images appear is called the **frame rate** and is measured in **frames per second** (FPS).

The **Graphics Processing Unit** (GPU) features a highly parallel structure that makes it more efficient than CPUs for rendering computer graphics. Programs run by GPUs are called **shaders**. These are used to perform many different computations required in the rendering process. Shader programming languages implement an **application programming interface** (API), which defines a set of commands, functions, and protocols that can be used in interacting with an external system such as the GPU. Below are some APIs and their corresponding shader languages:

- **DirectX API & High-Level Shading Language (HLSL)** – used on Microsoft platforms, including the Xbox game console
- **Metal API & Metal Shading Language** – runs on modern Mac computers, iPhones, and iPads
- **OpenGL (Open Graphics Library) API & OpenGL Shading Language (GLSL)**, a cross-platform library – OpenGL is the most widely adopted graphics API.

The Graphics Pipeline

A **graphics pipeline** is an abstract model used to describe a sequence of steps needed in rendering a three-dimensional scene. Pipelining enables a computational task to be split into subtasks thus increasing overall efficiency.

Graphics pipelines increase the efficiency of the rendering process, enabling images to be displayed at faster rates.

The pipeline model used in OpenGL consists of four (4) stages:

1. **Application** – initializing the window where rendered graphics will be displayed; sending data to the GPU
2. **Geometry Processing** – determining the position of each vertex of the geometric shapes to be rendered, implemented by a program known as **vertex shader**
3. **Rasterization** – determining which pixels correspond to the geometric shapes to be rendered
4. **Pixel Processing** – determining the color of each pixel in the rendered image, involving a program called a fragment shader

Stage 1: Application

The application stage primarily involves processes that run on the CPU. The following are performed during the application stage:

- **Creating a window where the rendered graphics will be displayed**: The window must be initialized to read the graphics from the GPU framebuffer. For animated and interactive applications, the main application contains a loop that repeatedly re-renders the scene, usually aiming for a rate of 60 FPS.
- **Reading data required for the rendering process**: This data may include **vertex attributes**, which describe the appearance of the geometric shapes being rendered. The vertex attribute data is stored in GPU memory buffers called **vertex buffer objects** (VBOs), while images to be used as textures are stored in **texture buffers**. Lastly, source code for the vertex shader and fragment shader programs needs to be sent to the GPU, compiled, and loaded.
- **Sending data to the GPU**: The application needs to specify the associations between attribute data stored in VBOs and attribute variables in the vertex shader program. A single geometric shape may have multiple attributes for each vertex (such as position and color). The corresponding data is streamed from buffers to variables in the shader during rendering. Frequently, it is also necessary to work with many sets of such associations. Multiple geometric shapes may be rendered by the same shader program. Each shape may also be rendered by a different shader program. These sets of associations can be managed using **vertex array objects** (VAOs). VAOs store this information and can be activated and deactivated as needed during the rendering process.

Stage 2: Geometry Processing

The shape of a geometric object is defined by a **mesh**, a collection of points that are grouped into lines or triangles.

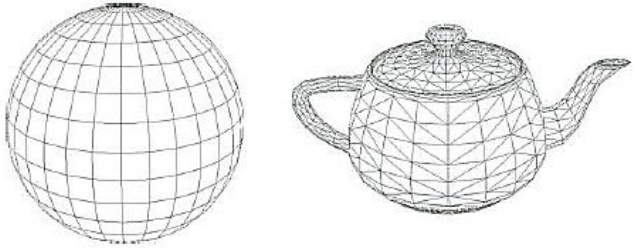

Figure 1. Examples of wireframe meshes

Apart from the object's overall shape, additional information may be required to describe how the object should be rendered. The properties or attributes that are specific to rendering each individual point are grouped together into a data structure called a **vertex**. A vertex should contain the three-dimensional position of the corresponding point. The additional data contained by a vertex often includes the following:

- A color to be used when rendering the point
- **Texture coordinates** (or UV coordinates), which indicate a point in an image that is mapped to the vertex
- A **normal vector**, which indicates the direction perpendicular to a surface and is typically used in lighting calculations

The figure below illustrates different renderings of a sphere that make use of these attributes: wireframe, vertex colors, texture, and with lighting effects

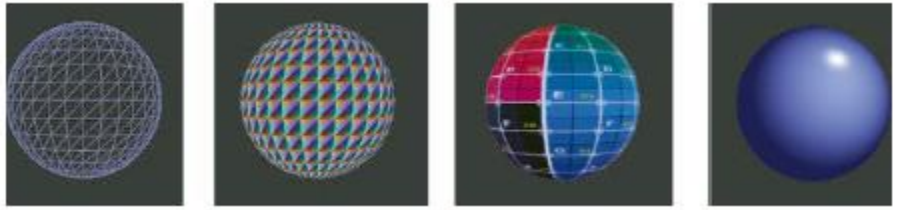

Figure 2. Renderings of a sphere

During this stage, the vertex shader is applied to each of the vertices; each attribute variable in the shader receives data from a buffer according to previously specified associations. The primary purpose of the vertex shader is to determine the final position of each of the points being rendered. This is typically calculated from a series of transformations.

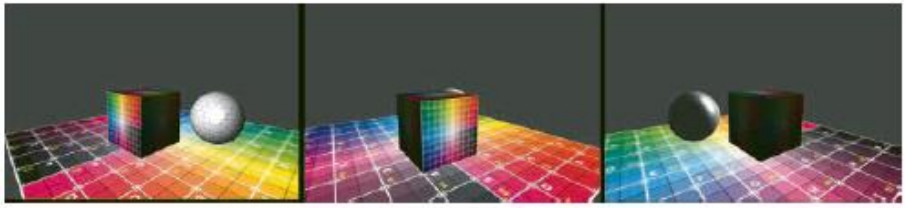

Figure 3. One scene rendered from multiple camera locations and angles

- **Model transformation**: The collection of points defining the intrinsic shape of an object may be translated, rotated, and scaled. Hence, the object appears to have a particular location, orientation, and size with respect to a virtual three-dimensional world. The coordinates expressed from this frame of reference are in world space. In **world space**, the origin is at the center of the scene. (Figure 3)
- **View transformation**: Coordinates in this context are said to be in view space. The **view space** (or camera space, or eye space) is the result when world-space coordinates are transformed to coordinates in front of the user's view.
- **Projection transformation**: Any points outside the specified region are discarded or clipped from the scene; coordinates expressed at this stage are in **clip space**.

Stage 3: Rasterization

Once the vertex shader has specified the final positions of each vertex, the rasterization stage begins. The points themselves must first be grouped into the desired type of geometric primitive: points, lines, or triangles, consisting of sets of 1, 2, or 3 points. For lines and triangles, additional information must be specified. For example, an array of points [A, B, C, D, E, F] is to be grouped into lines. They could be grouped in disjoint pairs, such as (A, B), (C, D), (E, F), producing a set of disconnected line segments. They could also be grouped in overlapping pairs, such as (A, B), (B, C), (C, D), (D, E), (E, F), producing a set of connected line segments (called a **line strip**). The type of geometric primitive and method for grouping points is specified using an OpenGL function parameter when the rendering process begins. The process of grouping points into geometric primitives is termed **primitive assembly**.

The next step is to identify which pixels correspond to the interior of each geometric primitive. A criterion must be specified to clarify which pixels are in the interior. A fragment is created for each pixel corresponding to the interior of a shape. A **fragment** is a collection of data used to determine the color of a single pixel in a rendered image. The data stored in a fragment always includes the **raster position**, also called pixel coordinates.

Stage 4: Pixel Processing

The primary purpose of this stage is to determine the final color of each pixel, storing this data in the color buffer within the framebuffer. During the first part of the pixel processing stage, a program called the **fragment shader** is applied to each of the fragments to calculate their final color. This calculation may involve a variety of data stored in each fragment, in combination with data globally available during rendering, such as the following:

- A base color applied to the entire shape
- Colors stored in each fragment (interpolated from vertex colors)
- Textures (images applied to the surface of the shape), where colors are sampled from locations specified by texture coordinates (Figure 4)
- Light sources, whose relative position and/or orientation may lighten or darken the color, depending on the direction the surface is facing at a point, specified by normal vectors

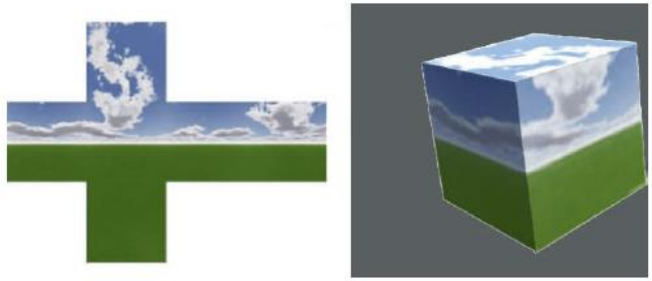

Figure 4. An image file used as a texture for a 3D object

References:

Korites, B. (2018). *Python graphics: A reference for creating 2D and 3D images.* Apress.

Marschner, S. & Shirley, P. (2021). *Fundamentals of computer graphics* (5th ed.). CRC Press.

Stemkoski, L. & Pascale, M. (2021). *Developing graphics frameworks with Python and OpenGL.* CRC Press.