Advanced Image Synthesis

Los Del DGIIM, losdeldgiim.github.io

Doble Grado en Ingeniería Informática y Matemáticas

Universidad de Granada

Advanced Image Synthesis

Los Del DGIIM, losdeldgiim.github.io

Arturo Olivares Martos

Granada, 2025-2026

Índice general

1. Graphics Pipeline

During this chapter, we will discuss the graphics pipeline, which is a sequence of steps that a computer graphics system uses to render a 3D scene onto a 2D screen. It starts from the initial creation of 3D models and ends with the final display of the rendered image. There are two types of graphics pipelines: the fixed-function pipeline and the programmable pipeline.

1.1. Fixed-Function Pipeline

The fixed-function pipeline is a traditional graphics pipeline that was widely used in older graphics hardware. It consists of a series of predefined stages that perform specific tasks. Each stage has a fixed function, and the data flows through these stages in a specific order. During the following subsections, we will discuss the stages of the fixed-function pipeline in detail.

1.1.1. Geometry

During this stage, the 3D models are created and defined. This includes defining the vertices, normals, texture coordinates, and other attributes of the 3D objects. In order to create the 3D models, the following primitives are used:

- `GL_POINTS`
- `GL_LINES`: A series of disconnected straight lines.
- `GL_LINE_STRIP`: A series of connected lines.
- `GL_LINE_LOOP`: A series of connected lines that form a loop.
- `GL_POLYGON`: A filled polygon defined by a series of vertices.
- `GL_TRIANGLES`: A series of disconnected triangles.
- `GL_TRIANGLE_STRIP`: A series of connected triangles. The first three vertices define the first triangle, and each subsequent vertex forms a new triangle with the previous two vertices.
- `GL_TRIANGLE_FAN`: A series of connected triangles that share a common central vertex. The first vertex is the center of the fan, and each subsequent vertex forms a new triangle with the center vertex and the previous vertex.
- `GL_QUADS`: A series of disconnected quadrilaterals.

- `GL_QUAD_STRIP`: A series of connected quadrilaterals. The first four vertices define the first quadrilateral, and each subsequent pair of vertices forms a new quadrilateral with the previous two vertices.

Even though all of these primitives are supported by OpenGL, it is recommended to use triangles for rendering, as they are the most efficient and widely supported primitive in modern graphics hardware.

In order to specify the vertices of the primitives, a process evolved from immediate mode to vertex arrays and eventually to vertex buffer objects (VBOs).

1. Immediate Mode: In this mode, vertices are specified one at a time. It is simple to use but inefficient, as it requires multiple function calls for each vertex. The geometry is then stored in the code.

2. Vertex Arrays: This mode allows you to specify an array of vertices and their attributes, which can be more efficient than immediate mode. The geometry is then stored in the RAM.

3. Vertex Buffer Objects (VBOs): This mode allows you to store vertex data in the GPU's memory, which can significantly improve performance.

1.1.2. Geometry Processing

During this stage, the vertices defined in the geometry stage are processed to prepare them for rendering. There are different spaces in which the vertices are processed:

- Object Space: The original space in which the vertices are defined.
- World Space: The space in which the vertices are transformed to position them in the scene.
- Camera/Eye Space: The space in which the vertices are transformed to position them relative to the camera (the camera is at the origin).
- Clip Space: The space in which the vertices are transformed to prepare them for clipping (removing parts of the geometry that are outside the view frustum).
- Normalized Device Space: The space in which the vertices are transformed to fit within a standardized coordinate system (usually a cube from -1 to 1 in all axes).
- Screen Space: The final space in which the vertices are transformed to fit the actual screen dimensions.

In order to transform the vertices from one space to another, several transformations are applied.

Model Transformation

Before moving again to computer graphics, we should focus on the transformations that we can apply to the vertices.

Scaling It is a linear application that changes the size of an object. It can be uniform (same scaling factor for all axes) or non-uniform (different scaling factors for each axis). An scaling transformation of values (α, β, γ) can be represented in 3D using the matrix (1.1).

$$\begin{pmatrix} \alpha & 0 & 0 \\ 0 & \beta & 0 \\ 0 & 0 & \gamma \end{pmatrix} \tag{1.1}$$

The inverse of a scaling transformation is also a scaling transformation with the reciprocal of the scaling factors.

Shearing It is a linear application that distorts the shape of an object by shifting its vertices in a specific direction depending on their distance to a reference plane. A shearing transformation of value α in the x direction depending on their distance to the XZ plane can be represented in 3D using the matrix (1.2).

$$\begin{pmatrix} 1 & \alpha & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{1.2}$$

Rotation It is a linear application that rotates an object around a specific axis by a certain angle. A rotation transformation of angle θ around the z axis can be represented in 3D using the matrix (1.3).

$$\begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{1.3}$$

The inverse of a rotation transformation is also a rotation transformation with the opposite angle. Mathematically, given that the rotation matrix is orthogonal, its inverse is equal to its transpose.

Translation This last type of transformation is not a linear application, as it does not preserve the origin. These transformations need to be represented using homogeneous coordinates, which add an extra dimension to the vertices. Therefore:

- Points are represented as $(x, y, z, 1)^t$.
- Vectors are represented as $(x, y, z, 0)^t$.

Every transformation that we have allready seen can be represented in homogeneous coordinates by adding an extra row and column to the transformation matrix, with 0s in the new row and column, except for the last element of the

new column, which is 1. For example, a translation transformation of values (α, β, γ) can be represented in 3D using the matrix (1.4).

$$\begin{pmatrix} 1 & 0 & 0 & \alpha \\ 0 & 1 & 0 & \beta \\ 0 & 0 & 1 & \gamma \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{1.4}$$

The model transformation is the transformation that takes the vertices from object space to world space. It is used to position the objects in the scene. It is usually represented as a combination of different transformations, and mathematically it can be represented as a single matrix M_{model} that is the product of the individual transformation matrices (taking the order of transformations into account).

$$P_{\text{world}} = M_{\text{model}} \cdot P_{\text{object}}$$

When a transformation M is applied to a vertex P, the normal vector N of the vertex is also transformed. Someone might think that the normal vector can be transformed using the same transformation M, but this is not always the case. The normal vector should be transformed using the inverse transpose of the transformation matrix M to ensure that it remains perpendicular to the surface after the transformation. Mathematically, the transformed normal vector N' can be calculated as follows:

$$N' = (M^{-1})^t \cdot N$$

Demostración. To understand why the normal vector should be transformed using the inverse transpose of the transformation matrix, we need to consider how transformations affect the geometry of the surface. Let n be the normal vector of a surface at a point P and t be the tangent vector at the same point. For clarity, let's use T_O to denote the transformation applied to the points (the tangent) and T_N to denote the transformation applied to the normal vector. We want to ensure that after applying the transformations, the normal vector remains perpendicular to the tangent vector, which means:

$$(T_N \cdot n) \perp (T_O \cdot t) \iff (T_N \cdot n) \cdot (T_O \cdot t) = 0 \iff (T_N \cdot n)^t \cdot (T_O \cdot t) = 0 \iff n^t \cdot T_N^t \cdot T_O \cdot t = 0$$

Given that n is perpendicular to t, we have $n^t \cdot t = 0$. Therefore, for the transformed normal vector to remain perpendicular to the transformed tangent vector, we need:

$$T_N^t \cdot T_O = I \iff T_N^t = T_O^{-1} \iff T_N = (T_O^{-1})^t$$

□

This should always be taken into account when applying transforming normals, *the normal vector should be transformed using the inverse transpose of the transformation matrix applied to the points.*

View Transformation

The view transformation is the transformation that takes the vertices from world space to camera/eye space. It is used to position the camera in the scene and to define the view direction. In OpenGL, the camera is always positioned at the origin of the coordinate system, looking in the z direction. As the camera can't move, the view transformation is achieved by applying the inverse of the camera's transformation to the vertices.

Let's say that we want to position the camera at a point C in world space, looking at a direction D. We can define the view transformation using the following steps:

1. First of all, as we want to position the camera at point C, we need to apply a translation transformation that moves the world in the opposite direction of C. This can be represented using a translation matrix T that translates by $-C$.

2. Next, we need to align the camera's view direction with the z axis. To do this, we only know the direction D that the camera is looking at.

   a) We can calculate the right vector R of the camera using the cross product of the view direction D and the up vector U' (which is usually defined as $(0, 1, 0)$). This can be represented as $R = D \times U$.

   b) Then, we can calculate the up vector U of the camera using the cross product of the right vector R and the view direction D. This can be represented as $U = R \times D$.

   This way, the matrix $[R, U, D]$ maps this orthogonal basis to the standard basis of the camera space. Therefore, the inverse of this transformation is:

$$[R, U, D]^{-1}$$

Therefore, the view transformation can be represented as the product of the translation and the inverse of the rotation:

$$M_{\text{view}} = [R, U, D]^{-1} \cdot T$$

Lighting and Shading

Before moving to the next stage, lighting and shading should be considered, because they are the processes that determine the color and intensity of the pixels in the final rendered image. Lighting is the process of calculating how light interacts with the surfaces of the objects in the scene, while shading is the process of determining the color of each pixel based on the lighting calculations. The Phong Lighting Model is a widely used model for calculating the lighting in computer graphics. It uses the following notation.

Notación.

- X is the point on the surface being shaded.

- N is the normal vector at point X.
- L is the vector from point X to the light source.
- V is the vector from point X to the viewer (camera).
- R is the reflection of the light vector L around the normal vector N, calculated as $R = 2(N \cdot L)N - L$.

The Phong Lighting Model calculates the color of a pixel using three components: ambient, diffuse, and specular.

- Ambient Component: This component represents the constant background light in the scene. It is calculated as:

  $$\text{Ambient}(X) = C_{X,a} \cdot C_{L,a}$$

  where $C_{X,a}$ is the ambient color of the surface at point X and $C_{L,a}$ is the ambient color of the light source.

- Diffuse Component: This component represents the light that is scattered in all directions when it hits a surface. It is calculated as:

  $$\text{Diffuse}(L, X) = C_{X,d} \cdot C_{L,d} \cdot \text{máx}(0, \cos(\theta)) = C_{X,d} \cdot C_{L,d} \cdot \text{máx}(0, N \cdot L)$$

  where $C_{X,d}$ is the diffuse color of the surface at point X, $C_{L,d}$ is the diffuse color of the light source, and θ is the angle between the normal vector N and the light vector L.

- Specular Component: This component represents the light that is reflected in a specific direction when it hits a surface. It is calculated as:

  $$\text{Specular}(L, X, V) = C_{X,s} \cdot C_{L,s} \cdot \text{máx}(0, \cos(\alpha))^n = C_{X,s} \cdot C_{L,s} \cdot \text{máx}(0, R \cdot V)^n$$

  where $C_{X,s}$ is the specular color of the surface at point X, $C_{L,s}$ is the specular color of the light source, α is the angle between the reflection vector R and the view vector V, and n is the shininess coefficient that controls the size of the specular highlight.

The final color of the pixel is then calculated as the sum of these three components:

$$\text{Color}(L, X, V) = \text{Ambient}(X) + \text{Diffuse}(L, X) + \text{Specular}(L, X, V)$$

Perspective Transform

In this stage, the vertices are transformed from camera/eye space to clip space. The next definition is important.

Definición 1.1 (Frustrum)**.** The view frustum is a geometric shape that represents the volume of space that is visible to the camera. It can be defined in several ways (using the field of view, just the planes...). Two important planes are:

- Near Plane: The plane that is closest to the camera. Objects that are closer than this plane will not be rendered.
- Far Plane: The plane that is farthest from the camera. Objects that are farther than this plane will not be rendered.

Two type of frustrums can be defined:

- Perspective Frustum: A frustum that represents a perspective projection, where objects that are farther from the camera appear smaller. This type of frustum looks like a truncated pyramid, with the near plane being smaller than the far plane.
- Orthographic Frustum: A frustum that represents an orthographic projection, where objects that are farther from the camera appear the same size as objects that are closer to the camera. This type of frustum looks like a rectangular box, with the near and far planes being the same size. This type of projection is often used for 2D rendering or for technical drawings where accurate measurements are important. It represents that the viewer is at the infinity.

The perspective transformation is then used to transform the vertices from camera/eye space to clip space. It is represented as a matrix that prepares the vertices for perspective division. The exact form of the perspective transformation matrix depends on the parameters of the view frustum (how was it given), but it generally includes terms that account for the field of view, aspect ratio, and the near and far planes. We will simply consider the $M_{\text{projection}}$ matrix as the one that transforms the vertices from camera/eye space to clip space, without going into the details of its construction. Therefore, what the programmer should implement in OpenGL is:

$$P_{\text{clip}} = M_{\text{projection}} \cdot M_{\text{view}} \cdot M_{\text{model}} \cdot P_{\text{object}}$$

Clipping

The clipping stage removes any vertices that are outside the view frustum. Here, the vertices are transformed from clip space to normalized device space by performing perspective division. Given that the vertices are in homogeneous coordinates and we need the last w coordinate to be 1, the other coordinates are divided by w to achieve this. After perspective division, and given that the correct perspective transformation was applied, the vertices will be in normalized device space, and only the vertices that are within the range of $[-1, 1]$ in all axes will be kept for further processing. The vertices that are outside this range will be clipped, meaning that they will be discarded and not rendered.

Viewport Transformation

In this last stage, the vertices are transformed from normalized device space to screen space. This transformation maps the normalized device coordinates to the actual pixel coordinates on the screen. The viewport transformation is defined by the viewport parameters, which specify the position and size of the viewport on the screen. The viewport transformation can be represented as a matrix that scales and translates the vertices to fit within the specified viewport. After this transformation, the vertices are ready to be rasterized and rendered on the screen.

1.1.3. Rasterization

The rasterization stage is the process of converting the vertices and primitives into pixels on the screen. During this stage, the graphics pipeline determines which pixels on the screen correspond to each primitive, and a fragment is generated for each of these pixels. The data for each vertex (color, texture coordinates, etc.) is interpolated across the primitive to determine the attributes of each fragment.

Data Interpolation

Given $n+1$ points $\overrightarrow{x}_i \in \mathbb{R}^k$ with their corresponding values $f_i \in \mathbb{R}$, $i \in \{0, \ldots, n\}$, interpolating is finding a function $f : \mathbb{R}^k \to \mathbb{R}$ such that $f(\overrightarrow{x}_i) = f_i$ for all $i \in \{0, \ldots, n\}$. When achieved, the function f can be used to estimate the value of f at any point $\overrightarrow{x} \in \mathbb{R}^k$ by evaluating $f(\overrightarrow{x})$.

Linear interpolation is a simple method of interpolation that only considers the linear functions f, which can be represented as $f(\overrightarrow{x}) = \overrightarrow{a} \cdot \overrightarrow{x} + b$ for some $\overrightarrow{a} \in \mathbb{R}^k$ and $b \in \mathbb{R}$. Given that we have $n+1$ constraints (one for each point) and $k+1$ unknowns (the components of $\overrightarrow{a}$ and b), we can solve for the coefficients of the linear function f if $n+1 \leqslant k+1$. In our case, we are interested in interpolating the attributes of the vertices across the primitives, which means that we have 3 vertices (for triangles) and two dimensions (the screen coordinates), so $k = 2$. Therefore, our aim is to find $a, b, c \in \mathbb{R}$ such that $f(x, y) = ax + by + c$ and $f(\overrightarrow{x}_i) = f_i$ for $i \in \{0, 1, 2\}$. This can be achieved by solving the system of equations of Equation (1.5).

$$\begin{cases} ax_0 + by_0 + c = f_0 \\ ax_1 + by_1 + c = f_1 \\ ax_2 + by_2 + c = f_2 \end{cases} \implies \begin{pmatrix} x_0 & y_0 & 1 \\ x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \end{pmatrix} = \begin{pmatrix} f_0 \\ f_1 \\ f_2 \end{pmatrix} \tag{1.5}$$

However, this does not take the perspective into account. To achieve perspective-correct interpolation, we need to consider the depth of each vertex.

Shading

Shading is the process of determining the color of each pixel based on the lighting calculations. During rasterization, the attributes of the vertices are interpolated across the primitive to determine the attributes of each fragment. There are different ways to apply the Phong Lighting Mode, depending on what is interpolated and when is the color calculated:

- Flat Shading: The color is calculated for each primitive (e.g., triangle) using a single value of each attribute for the whole primitive. This results in a faceted appearance, as each primitive has a single color.
- Gouraud Shading: The color is calculated at each vertex , and then interpolated across the primitive. This results in a smoother appearance than flat shading, but it can produce artifacts such as Mach bands.
- Phong Shading: The normal vector is interpolated across the primitive, and the color is calculated for each fragment using the interpolated normal vector. This results in a smoother appearance than Gouraud shading and can produce more accurate lighting effects.

1.1.4. Fragment Processing

The fragment processing stage is almost the final stage of the graphics pipeline, where the fragments generated during rasterization are processed to determine their final color and whether they should be rendered on the screen. During this stage, several operations can be performed on the fragments, such as depth testing, stencil testing, blending, and more. The specific operations performed during fragment processing depend on the settings and configurations of the graphics pipeline.

Textures

A texture map is a discretely sampled multi-dimensional function containing material properties, like color or normals. Texture mapping is the process of applying a texture map to a 3D model to add detail and realism to the rendered image. During fragment processing, the texture coordinates of each fragment are used to sample the corresponding attribute from the texture map, which can then be used. Some concepts about textures are:

- Wrapping: Texture coordinates usually are planned to be in the range of $[0, 1]$, but sometimes they can be outside this range. Wrapping is the process of determining how to handle texture coordinates that are outside the $[0, 1]$ range. Common wrapping modes include:
  - `GL_REPEAT`: The texture is repeated when the texture coordinates are outside the $[0, 1]$ range.
  - `GL_MIRRORED_REPEAT`: The texture is repeated and mirrored when the texture coordinates are outside the $[0, 1]$ range.
  - `GL_CLAMP_TO_EDGE`: The texture coordinates are clamped to the edge of the texture when they are outside the $[0, 1]$ range. This means that the texture will be stretched to fill the area outside the $[0, 1]$ range.

  One mode should be selected for each texture coordinate (e.g., s and t).
- Filtering: Usually, the texture maps are of a different resolution than the rendered image, which means that the texture needs to be filtered to determine the color of each pixel. There are two types of filtering:
  - `Minification`: The process of determining the color of a fragment when the texels (texture pixels) are smaller than the screen pixels.
  - `Magnification`: The process of determining the color of a fragment when the texels are larger than the screen pixels.

  Common filtering modes include:
  - `GL_NEAREST`: The color of the fragment is determined by the color of the nearest texel. In minification, this can lead to aliasing artifacts, while in magnification, it can lead to a blocky appearance.

  - `GL_LINEAR`: The color of the fragment is determined by the bilinear interpolation of the colors of the 4 nearest texels. In minification it does not make a big difference (considering 4 texels when the pixel may be covered by 100 texels does not improve the quality), but in magnification it can improve the appearance of the texture, making it appear smoother.

    In order to achieve bilinear interpolation, it is simply interpolated in both dimensions.

- MIP Mapping: MIP mapping is a technique used to improve the quality of textures when they are minificated. It involves creating multiple levels of detail for a texture, where each level is a downscaled version of the original texture. During rendering, the appropriate level of detail is selected based on the distance and angle of the fragment being rendered. This can help to reduce aliasing artifacts and improve the overall appearance of the texture, as no minification is applied when the appropriate level of detail is selected.

Fragment Tests

After the fragment processing stage, several tests can be performed on the fragments to determine whether they should be rendered on the screen or discarded. Before performing these tests, the content of each pixel should be explained. Each pixel (not fragment) on the screen has a buffer that stores:

- Color: The color of the pixel, usually represented as RGBA (red, green, blue, alpha).

- Depth: The depth value of the current pixel, which is the depth value of the fragment that is currently rendered at that pixel (the closest fragment to the camera).

- Stencil: The stencil value of the pixel, which is used for stencil testing. The programmer can define the stencil value for each pixel and store it in the stencil buffer.

Once this has been explained, some common fragment tests include:

- Alpha Test: This test discards fragments based on their alpha value. It consists of a comparison between the alpha value of the fragment and a reference value, using a specified comparison function (e.g., less than, greater than, equal to). If the comparison fails, the fragment is discarded and not rendered on the screen.

- Stencil Test: This test discards fragments based on their stencil value. It compares the stencil value of the pixel where the fragment would be rendered with a reference value, using a specified comparison function. If the comparison fails, the fragment is discarded and not rendered on the screen. The stencil test can be used for various effects, such as creating shadows, outlining objects, or implementing complex masking operations.

- Depth Test: This test discards fragments based on their depth value. It compares the depth value of the fragment with the depth value of the corresponding pixel in the depth buffer. If the fragment is closer to the camera than the existing pixel, it passes the depth test and is rendered on the screen, and the depth buffer is updated with the new depth value. If the fragment is farther from the camera than the existing pixel, it fails the depth test and is discarded.

Blending

Blending is the process of combining the color of a fragment with the color of the corresponding pixel in the framebuffer to produce a final color that is rendered on the screen. Blending is typically (but not only) used to achieve transparency effects, where the color of the fragment is blended with the color of the background to create a semi-transparent appearance. The blending operation is defined by:

$$C_{\text{final}} = C_{\text{source}} \cdot F_{\text{source}} \odot C_{\text{destination}} \cdot F_{\text{destination}}$$

where:

- C_{final} is the final color that is rendered on the screen.
- C_{source} is the color of the fragment being processed.
- $C_{\text{destination}}$ is the color of the corresponding pixel in the framebuffer.
- F_{source} and $F_{\text{destination}}$ are the blending factors for the source and destination colors, respectively. These factors determine how much of each color contributes to the final color. Common blending factors include:
  - `GL_ZERO`: The factor is 0, meaning that the corresponding color does not contribute to the final color.
  - `GL_ONE`: The factor is 1, meaning that the corresponding color fully contributes to the final color.
  - `GL_SRC_ALPHA`: The factor is equal to the alpha value of the source color, meaning that the contribution of the source color is proportional to its transparency.
  - `GL_ONE_MINUS_SRC_ALPHA`: The factor is equal to 1 minus the alpha value of the source color, meaning that the contribution of the destination color is proportional to the transparency of the source color.

  The specific blending factors used can be configured based on the desired effect.
- $\odot$ represents the operator used to combine the source and destination colors. Common operators include addition, subtraction, minimum, and maximum.

1.1.5. Framebuffer

The framebuffer is the final destination for the rendered image. It is a memory buffer that stores the color, depth, and stencil information for each pixel on the screen. After all the fragment processing and tests have been performed, the final color of each pixel is written to the framebuffer, which is then displayed on the screen. The framebuffer can also be used for off-screen rendering, where the rendered image is stored in a texture or a renderbuffer instead of being displayed on the screen. This allows for various effects and techniques, such as post-processing, shadow mapping, and more.

1.2. Programmable Pipeline

The programmable pipeline is a modern graphics pipeline that allows developers to write custom shaders to control the behavior of each stage of the pipeline. This provides greater flexibility and control over the rendering process, allowing for more complex and realistic graphics. The programmable pipeline consists of several stages, including vertex shading, tessellation, geometry shading, fragment shading, and more. Each stage can be programmed using a shading language such as GLSL (OpenGL Shading Language) or HLSL (High-Level Shading Language). The programmable pipeline has largely replaced the fixed-function pipeline in modern graphics hardware, as it allows for more advanced rendering techniques and greater performance.

2. Reflection

2.1. Planar Reflection

2.1.1. Recursive Reflections

Alternative Solution

2.2. Non-Planar Reflection

2.3. Refraction

2.3.1. Fresnel Equations

...
Aproximations