Graphics pipeline

A graphics pipeline can be divided into three major steps: application, geometry and rasterization.

Application: The application step is executed in software, so it cannot be divided into individual steps that are executed in the form of a pipeline. However, it is possible to parallelize it on multi-core processors or multiprocessor systems. In the application step, changes are made to the scene as required, for example, due to user interaction with input devices or in the case of an animation. The new scene with all its primitives - mostly triangles, lines and points - is then forwarded to the next step of the pipeline.

Examples of tasks typically performed by the application step include collision detection, animation, morphing, and data management. The latter include, for example, acceleration techniques using spatial subdivision schemes (Quadtree, Octree) that optimize the data currently stored in memory. The "world" and its textures of a modern-day computer game are much bigger than could be loaded into the available RAM or graphics memory.

Geometry: The geometry step, which is responsible for the majority of operations with polygons and their vertices, can be divided into five tasks:

1. Modelling Transformations: In addition to the objects, the scene also defines a virtual camera or viewer that indicates the position and viewing direction from which the scene is to be rendered. In order to simplify later projection and clipping, the scene is transformed so that the camera is at its origin, facing along the Z axis. The resulting coordinate system is called the camera coordinate system and the transformation is called the view transformation.

2. Illumination (Shading): A scene often contains light sources placed at different positions to make the lighting of the objects appear more realistic. In this case, a texture enhancement factor is calculated for each vertex based on the light sources and the material properties associated with the corresponding triangle. In the subsequent screening step, the corner point values of a triangle are interpolated over its surface. General lighting (ambient light) is applied to all surfaces. It is the diffuse and thus direction-independent brightness of the scene. The sun is a directional light source that can be assumed to be infinitely distant. The lighting effect of the sun on a surface is determined by the formation of the scalar product of the directional vector from the sun and the normal vector of the surface. If the value is negative, the surface is facing the sun.

3. Viewing Transformation (Perspective / Orthographic): This step transforms the visible volume into a cube with corner point coordinates (-1, -1, -1, -1) and (1,1,1,1); occasionally other target volumes are also used. This step is called projection, although it transforms a volume into another volume, because the resulting Z coordinates are not stored in the image, but only used for z-buffering in the subsequent rasterizing step. A central projection is used for a perspective image. In order to limit the number of displayed objects, two additional clipping planes are used; the visible volume is a pyramid stump (Frustum). For example, parallel or orthogonal projection is used for technical representations, because it has the advantage that all parallels in object space are also parallel in the image space and surfaces and volumes are the same size regardless of the distance to the viewer. For efficiency reasons, the camera and projection matrix are usually combined in a transformation matrix, so that the camera coordinate system is ignored. The resulting matrix is usually consistent for a single image, while the world matrix looks different for each object. In practice, therefore, view and projection are pre-calculated, so that only the World-Matrix has to be adjusted during display. However, more complex transformations such as vertex blending are possible. Freely programmable geometry shaders that change the geometry can also be executed. In the actual rendering step, the model matrix * camera (view) matrix * projection matrix is then calculated and finally applied to each individual point. The possible combination of matrices is illustrated in Figure 1. This transfers the points of all objects directly into the screen coordinate system (at least nearly, the value ranges of the axes are still -1... 1 for the visible area).

4. Clipping: Only the primitives that are located within the visible volume must actually be rasterized. Primitives that are completely out of sight are discarded; this is called frustum culling. Further culling procedures such as backface pulling, which reduce the number of primitives to be considered, can theoretically be performed in any step of the graphics pipeline. Primitives that are only partially inside the cube must be clipped against the cube. The advantage of the previous projection step is that clipping always takes place against the same cube. Only the - possibly clipped - primitives that are within the visible volume are forwarded to the next step.

5. Projection (to Screen Space): To output the image to any viewport on the screen, a further transformation, the Window Viewport Transformation, must be applied. This is a shift, followed by scaling. The resulting coordinates are the device coordinates of the output device. The viewport contains 6 values: Height and width of the window in pixels, the upper left-hand corner of the window in window coordinates (usually 0.0) and the minimum and maximum values for Z (usually 0 and 1). On modern hardware, most of the geometry calculation steps are performed in the Vertex Shader. This is in principle freely programmable, but as a rule it takes over at least the transformation of the points and the lighting calculation. For the programming interface DirectX from version 10 onwards and OpenGL version 4, a user-defined vertex shader is unavoidable, whereas older versions have provided a standard shader.

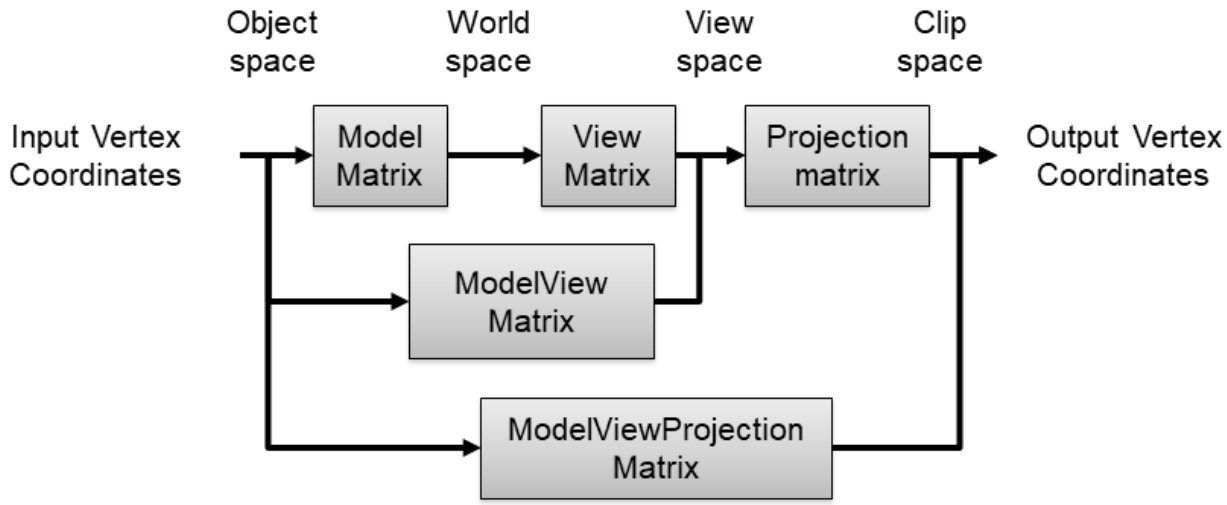

Figure 1: Possible pre-combinations of transformation matrices and their common names.

It depends on the respective implementation how these tasks are organized as actual pipeline steps that are executed in parallel.

Rasterization: The geometry steps are followed by rasterization, i.e. the sampling of primitives into pixels on screen. In this step, all primitives are rasterized, i.e. discrete fragments are created from continuous surfaces.

In this stage of the graphic pipeline, the raster points are also called fragments, i.e. each fragment corresponds to one pixel in the frame buffer and this corresponds to one pixel of the screen.

These can then be coloured (illuminated if necessary). Furthermore, it is necessary to determine the visible, i.e. closer to the viewer, of overlapping polygons. A z-buffer is usually used for this so-called masking calculation. The colour of a fragment depends on the illumination, texture, and other material properties of the visible primitive and is often interpolated using the triangular corner points. Where available, a fragment shader is executed after the rasterization step for each fragment of the object. If a fragment is visible, it can be mixed with existing color values in the image if transparency is simulated or multi-sampling is used. In this step, one or more fragments become a pixel.

To prevent the user from seeing the gradual screening of the primitives, double buffering is used. The

rasterization takes place in a special memory area. As soon as the image has been completely rastered, it is copied into the visible area of the image memory (frame buffer).

An overview over the currently used graphics pipeline for modern OpenGL can be found at `https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview`.

OpenGL

OpenGL (Open Graphics Library) is a specification of a cross-platform and cross-programming language programming interface for the development of 2D and 3D computer graphics applications. The OpenGL standard describes about 250 commands that allow the representation of complex 3D scenes in real time. In addition, other organizations (mostly manufacturers of graphics cards) can define proprietary extensions.

The OpenGL API is usually implemented by system libraries, on some operating systems also as part of the graphics card drivers. These execute commands according to the commands of the graphics card, in particular, functions that are not available on the graphics card must be emulated by the CPU.

Many parameters can influence the appearance of rendered objects, for example, they can be textured and illuminated, stretched, shifted, transparent or opaque, they can have a rough or smooth surface and so on.

OpenGL was designed as a state machine, which does not receive all the required parameters with every function call, but uses the same values until the corresponding states are changed. In this way, for example, one does not have to specify the desired colour for each vertex, but set a colour once and all following vertices will be displayed in this colour. In the same way, one can switch light sources on and off globally and set many other states.

The reason for this design is that almost every change in the drawing mode causes extensive reorganization of the graphics pipeline, so it is better to avoid it as long as it is reasonable. It would also be tedious for the programmer to enter dozens of parameters over and over again. Often, many thousands of vertices can be edited before a state needs to be changed, while some states are never changed. For example, the light sources usually remain the same for all objects in a scene. Many states are maintained at least for the duration of the rendering of a complete object, for example, a car as a whole is moved by a specific vector and not broken down into its individual parts and moved one by one. This condition-based concept is also pursued in Direct3D.

An important feature of OpenGL is its extensibility. Individual vendors (typically graphics card manufacturers) can add more states to the state machine of OpenGL. A four-step procedure is followed:

1. If a manufacturer wants to implement an extension, they deliver a C header file, in which they define the extension with the required constants and any function prototypes. The function names and constants are given a manufacturer-specific suffix (e. g. NV for Nvidia or ATI for ATI).
2. If several manufacturers agree to offer the same extension, the function names and constants get the suffix EXT.
3. Finally, if the ARB (Architecture Review Board) agrees to standardize the extension, all names will receive the suffix ARB.
4. Most of the extensions standardized by ARB become core in the following OpenGL specification. This means that they become part of OpenGL itself and no longer have a suffix.

Advantages and disadvantages of OpenGL compared to Direct3D

Advantages:

1. client server model

2. draw calls are under certain circumstances more powerful than in Direct3D
3. cross-platform
4. expandable by manufacturers themselves
5. there are a number of extensions for new functions not yet supported by the standard.
6. the available features depend on the GPU or its drivers and not on the operating system.
7. a comprehensive documentation of the API and the standard can be found online at `https://www.opengl.org/`.

Disadvantages:

1. OpenGL still has a partially obsolete and complex application programming interface (API) that some developers consider to be cumbersome.

OpenGL Shading Language: GLSL

Shading in computer graphics refers to the modification of individual vertices or fragments within the graphics pipeline. Shaders calculate the appearance of an object or create special effects. Typical tasks include texturing and illumination. In the classical (so-called Fixed Function) OpenGL pipeline, the individual calculation steps of the shader cannot be changed and only individual parameters can be configured. In order to overcome this limitation, the OpenGL version 1.4 introduced GLSL as an extension. GLSL allows to freely define parts of the pipeline using own programs. For example, a special illumination model or a texture effect such as bump mapping can be implemented.

With OpenGL version 2.0, the language became an official component of the OpenGL specification, which defines the functional scope of OpenGL. The initial GLSL version offered only a vertex and fragment shaders. With OpenGL version 3.2, it was supplemented by the Geometry Shader, with version 4.0 by the Tessellation Control and Tessellation Evaluation Shader and with version 4.3 by the Compute Shader. [2, 1]

With today's GLSL-based pipelines, all processing steps of the graphics card can be programmed directly, with the exception of rasterization.

GLSL competes with the High-level shading language (HLSL), which provides the equivalent functionality for Direct3D.

GLSL is a C-like programming language that has been specially adapted to the needs of shaders. There are built-in types for vectors, matrices and a variety of math and graphics functions. Many of the operations offered can work on several data elements at the same time (SIMD). Unlike C, however, there are no pointers.

There are five different GLSL shader types; vertex, tessellation, geometry and fragment shaders as part of the rendering pipeline and their independent compute shaders. The compute shader type is the only type that can process data independently of the graphics pipeline and thus perform GPGPU calculations in the OpenGL context. Each shader type has characteristic input and output parameters. The application developer passes the shader source code and all additional variables and constants for each shader type to the OpenGL driver. The driver compiles and links the shaders to a shader program. It is not mandatory to use all shader types.

Each primitive sent by an application will pass through the shaders contained in the shader program in the following order:

1. **Vertex Shader**: The vertex shader is executed once for each vertex. The shader only has access to the vertex (including its texture coordinates, normal and other transferred data), but not to neighbouring vertices, the topology or similar.

2. **Tessellation Shader**: In the tessellation shader, an area (triangle or square) can be divided into smaller areas. During implementation, a distinction is made between tessellation control shaders and tessellation evaluation shaders.

3. **Geometry Shader**: In the geometry shader, new primitives can be created from an existing primitive (point, line, triangle).

4. **Fragment Shader**: The fragment shader is executed once for every fragment (pixels before they are displayed on the display device). The colour for the corresponding fragment is calculated here. Fragment shaders are the equivalent of Direct3D's pixel shader.

A complete documentation of the GLSL API and syntax definition is provided by the Khronos group at `https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.40.pdf`.

References

[1] khronos group. *OpenGL 4.4 reference https://www.khronos.org/registry/OpenGL/specs/gl/glspec44.core.pdf*, 2017.

[2] Dave Shreiner and The Khronos OpenGL ARB Working Group. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1*. Addison-Wesley Professional, 7th edition, 2009.