Rendering API coordinate systems

Object space and world space

Modern APIs don't impose a particular coordinate system (CS) for object space or world space (Note: the OpenGL legacy fixed-function pipeline assumes a right-handed world space CS with +Y as the up vector).

Which vector is considered the up vector and whether the CS is left- or right-handed is therefore arbitrary and largely depends on conventions, personal preference, or the modeling tools used.

Different 3D modelling applications and game engines have different conventions. See e.g. Freya Holmér on Twitter: "Here, have a coordinate system chart! http://t.co/riYtt6tLEd" (Note: Vulkan is not included because it has no predefined world CS, as do modern Direct3D (D3D12) and OpenGL (3.0+)?)

Recap linear algebra

The standard basis (canonical basis) of $\mathbb{R}^3$ consists of the unit vectors $e_1 = (1, 0, 0)^T$, $e_2 = (0, 1, 0)^T$, $e_3 = (0, 0, 1)^T$ (orthonormal basis). The names of these vectors are not standardized and there is no distinguished interpretation or handedness. Cartesian coordinates usually use a right-handed CS.

Right-handed CS are defined as such that the rotation from e_1 to e_2 is counter-clockwise when viewed from the point of e_3. Likewise left-handed CS are defined by a clockwise rotation. (Note: also see right-hand rule or cork-screw rule to determine handedness.)

(Note: vector product and triple product give the same result for standard basis vectors, independent of their arrangement in a left-handed or right-handed CS; only if the basis vectors of a CS are given w.r.t. a right-handed basis can the vector product and triple product be used to determine handedness.)

Basis transformation matrices can be used to transform a vector from one basis to another. A basis is given as a matrix consisting of its unit vectors, e.g. $B = [\, e_1\ e_2\ e_3 \,]$. Two bases have the same handedness if the determinant of the transformation matrix is positive.

The matrix that transforms vectors from basis B into basis B' is given by $T^B_{B'} = (B')^{-1} * B$. Both unit vectors of B and B' must be given w.r.t. a canonical basis. If B is the canonical basis this simplifies to $T^B_{B'} = (B')^{-1}$.

Normalized device coordinates and clip space

Modern rendering APIs only define (and know about) clip space, normalized device coordinates (NDC), and coordinates for framebuffers and viewports. These map directly to screen orientation and depth values.

The positive x-axis in NDC always points to the right on the screen. Points further away from the screen have a larger z value (with a default configuration of the depth buffer). For OpenGL the positive y-axis in NDC points up and for Vulkan it points down. This means NDC are left-handed in OpenGL and right-handed in Vulkan.

(Note: see e.g. Coordinate systems · Issue #416 · gpuweb/gpuweb · GitHub for a comparison with D3D12 and Metal.)

The transformation from a right-handed eye space to the left-handed clip space in OpenGL is performed by the (perspective) projection matrix as provided by e.g. gluPerspective or the default glm::perspective.
(Note: see gluPerspective and
https://github.com/g-truc/glm/blob/0.9.5/glm/gtc/matrix_transform.inl#L218).

The transformation assumes that the camera is at the origin of eye space looking along the negative z-axis. This matrix maps points from eye space with z = -near and z = -far to points in clip space which after perspective division have z = -1 and z = 1, respectively. This effectively changes the direction of the z-axis.

When the eye space or clip space are different from the ones in OpenGL the projection matrix has to be adapted accordingly, for example to transform the y-axis. More specifically, if the positive z-axis already points away from the viewer in eye space the projection matrix must not negate the z-axis again. (Note: when glm::perspective is used with GLM_LEFT_HANDED defined, the z-axis is not mirrored by the projection matrix.)

In Vulkan the NDC z values are in the range of [0, 1] instead of [-1, 1] for OpenGL. For this reason the projection matrix looks slightly different. (In glm the z-range of [0, 1] is enabled by defining GLM_DEPTH_ZERO_TO_ONE.)

Framebuffer coordinates

Framebuffer coordinates always align with NDC. Framebuffer coordinates also align with texture coordinates. This is especially convenient when using an image rendered to a framebuffer as a texture.

Texture coordinates and image formats

The use of texture and pixel coordinates can also differ between rendering APIs and image file formats. In OpenGL the texture origin is conventionally identified with the lower left corner of a texture, with texture coordinates s and t advancing to the right and upwards, respectively.

File formats such as PNG and JPEG define the image origin in the upper left corner. For this reason textures loaded from files must often be flipped vertically in OpenGL. However, since the direction of texture coordinates is just a convention, it is also possible to define a different origin if both the memory layout of the texture and the texture coordinates in the geometry are adapted accordingly.

An example of a format with the origin in the lower left corner is BMP. The difference between the two types of origins can be described by the orientation of the s and t texture coordinate vectors, which is either clockwise or counter-clockwise. If a texture is mapped to a polygon with a different orientation of the texture coordinates, the image will appear mirrored.

File formats with a configurable origin include TGA and KTX. In TGA files the Image Descriptor field defines how the pixel data is transferred to the screen, i.e. in which corner of the image the first pixel is located
(Note: see TGA specification, Field 5.6 Image Descriptor, Bits 5 & 4. TODO: does this also affect memory layout when reading from TGA files?)

In KTX files textures are arranged so that the first pixel in the data stream is closest to the texture coordinate origin. The actual orientation of the image depends on the content creation tools and the applications used and can be specified in the metadata section. The metadata affects only the logical interpretation of the image, not the mapping of pixels to texture coordinates.
(Note: see https://www.khronos.org/opengles/sdk/tools/KTX/file_format_spec/#3)

The current Vulkan specification is also agnostic to image orientation. It simply defines that the linear image layout starts at texture coordinates (0, 0). In other words, when stored in memory the pixel data closest to the origin has the lowest address.
(Note: see e.g. Spec is becoming orientation agnostic for texture coordinates · Issue #127 · Overv/VulkanTutorial. Earlier versions stated that “The t coordinate goes from 0.0 to 1.0, top to bottom.” Later versions removed the “top to bottom” part.)

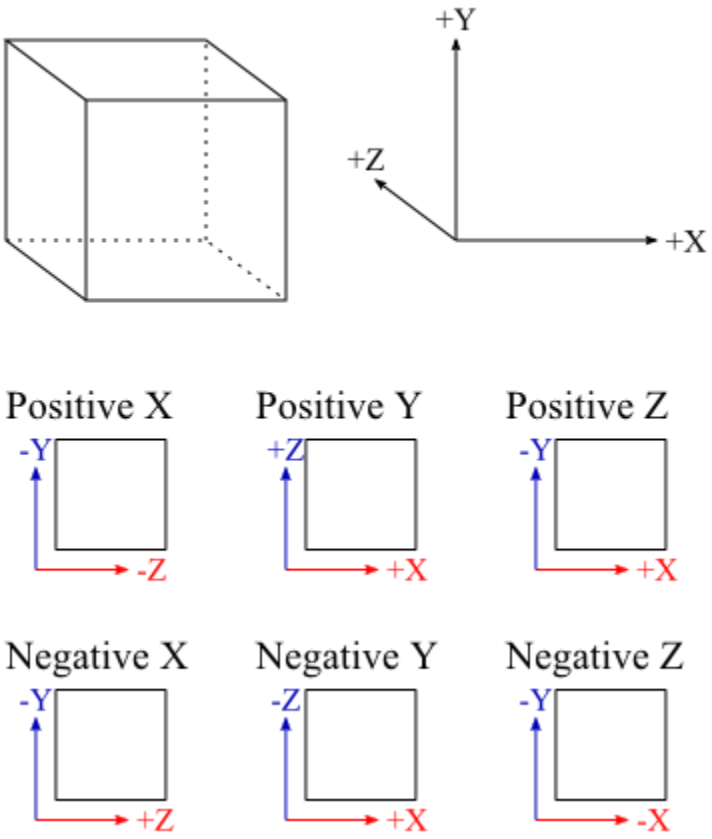
Cube map coordinates

Cube map coordinates in OpenGL and Vulkan are defined in a left-handed coordinate system. (Note: this convention was chosen to maintain compatibility with Renderman which introduced cube maps.) Textures are mapped to and viewed from the inside of the cube. This means that

when looking in the +z direction (at the +Z face of the cube), with the +Y face above, the +X face must be to the right.

The cube map layout was also developed with the texture coordinate origin in the upper left corner. So for example a direction vector of (1,1,1) corresponds to the top right corner of the +Z face but will map to the texture coordinate (1, 0) according to the default rules in OpenGL and Vulkan.

For this reason cube map textures may have to be flipped vertically or the texture coordinate t mirrored in OpenGL. In particular, this applies to image loaders that automatically align the image to the default OpenGL texture orientation, such that the texture origin maps to the lower left corner of the image. Since Vulkan does not define any texture orientation, and with file formats such as JPEG where the image orientation aligns with the orientation of cubemap textures, no adjustment is needed in this case.



Cube map coordinate system and texture coordinates for cube map faces in OpenGL as viewed from within the cube. Red and blue arrows denote the direction of texture coordinates s and t, respectively. Note how t coordinates are flipped due to cube map faces and OpenGL textures having different coordinate origins. From:

https://community.khronos.org/t/image-orientation-for-cubemaps-actually-a-very-old-topic/105338/4

The rules for mapping the direction vector to faces and (s, t) coordinates are described in https://www.khronos.org/registry/vulkan/specs/1.1-extensions/html/chap16.html#_cube_map_face_selection. Table 26, cube map face and coordinate selection, reads as follows for e.g. the direction vector (r_x, r_y, r_z) = (-5, -5, -5). The major axis direction is $-r_z$, because $-r_z = -(-5) = 5$ has a larger or equal magnitude than the others, and r_z wins over r_x and r_y if equal. The values (s_c, t_c, r_c) in the table are then used to compute the texture coordinates (s_{face}, t_{face}).The cube map coordinate transformation maps points on a cube of arbitrary side length to values in the range of [0, 1].

Note that the direction vector must already be given w.r.t. to the cube map CS. When using the coordinates of a skybox. defined in a right-handed CS where y is also up, as the direction vector, either the x- or z-coordinate must be flipped. It does not matter which one as both transformations will properly align the cube map with the skybox.

Instead of transforming the direction vector in the shader it is also possible (and cleaner) to define the skybox in the same CS as the cubemap and use the model matrix to transform it into world space. This way the shader code becomes agnostic to the world space CS.

Q & A

When does face culling happen? How is the winding order of polygons defined?

Face culling is done after the viewport transformation but before rasterization. This means that the winding order is always based on how polygons appear on screen. The winding is clockwise if the points of the polygon are listed in clockwise order on screen.

There is one caveat in Vulkan: when the KHR_VK_maintainance1 extension is used to flip the y-axis, the winding order of polygons rendered to the screen will be different to polygons rendered to an offscreen framebuffer.
(Source: Q & A section of Coordinate systems · Issue #416 · gpuweb/gpuweb · GitHub)

Should I flip the y-axis of the viewport in Vulkan using the KHR_VK_maintainance1 extension?

Problem: rendering an OpenGL scene with Vulkan results in the geometry being upside-down and front faces culled

Reason: Vulkan NDC has +y pointing downwards in contrast to upwards as in OpenGL, resulting in vertically mirroring the geometry and changing the winding order of polygons.

Solution: transform the y coordinates of the geometry. This can be done on the host (in model, view, or projection matrix), in shaders (when outputting gl_position), or in the viewport transformation (if the KHR_VK_maintainance1 extension is available).
(Source: Flipping the Vulkan viewport)

The extension allows setting a negative viewport height, resulting in vertical mirroring of the image during viewport transformation. You have to adjust the viewport origin as well or the viewport will be outside of the window. The extension is part of the core specification in Vulkan 1.1 and above.

Note that the viewport transformation does not apply when rendering to offscreen framebuffers so you still need to adjust the culling mode when using geometry where y means up (see e.g. http://anki3d.org/vulkan-coordinate-system/).

In short: you only need to use the extension if you cannot do the transformation to Vulkan NDC on the host. Adapting shaders is probably not a good solution as you have to take the transformation into account in every shader, which is error-prone and makes shaders less readable. It also makes it more difficult to write shaders that target different rendering APIs.