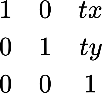
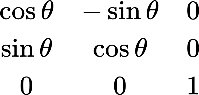
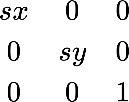
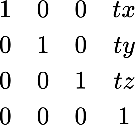
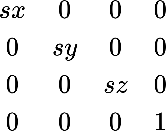
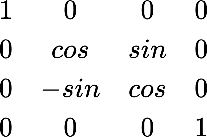
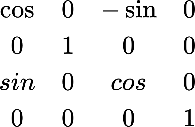
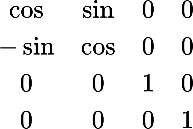
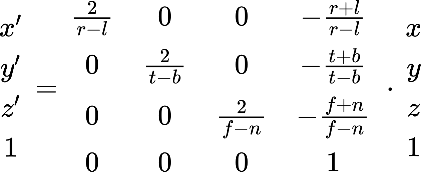
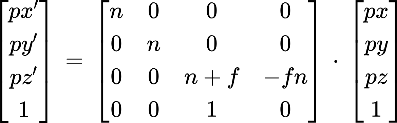
1. Color: RGB values in [0,1] → LDR Storage(8/16bits/color channel), HDR Storage(16/32-bit float)
2. Interleaved storage: RGB… Separate Channels: RRR… GGG… BBB…
   1. RI in memory: scanline order; swizzled order(Z-form)
3. BMP: bitmap image file
4. PPM: portable pixmap image file (uncompressed, LDR, 1/8/16 b per channel)
5. PNG: portable graphics format (lossless compression, LDR, 8/16 b per channel or color table→ just index of the color store in pixel 2 to 256 & Some information may be lost during compression when the color table is used)
6. GIF: graphics interchange format (lossless compression, LDR, color table & Has been widely used on the World Wide Web)
7. JPEG: joint photographic experts group (lossy compression, LDR, equivalent of 8 b per channel & In the worst case, storage and image quality can be lost during compression)
8. EXR: openEXR by ILM (lossy/lossless compression, HDR, 16/32 b per channel)
9. Gamma correction: cout = cinɣ = cdesired1/ɣ → in sRGB: ɣ≈2.2
10. RI in display–digital cameras(Bayer pattern, where each pixel capture only one of the color vale) & accurate for green > red/blue → visual range (**pixel** is not a little square, b/a point sample)
11. Alpha blending: for each pixel:[RGBA] → **c = αfcf+(1-αf)cb** & **α = αf+(1-αf)αb** & c = (**αfcf+(1-αf)αbcb**)/**α →** α = 0 ⇒ αf = 0; αb = 0; c = cb
12. Additive blending: α = clamp(αf+αb) & c = αfcf + cb = (αfcf+αbcb)/α
13. Difference blending: c = | αfcf + cb |
14. Multiply blending: c = αf(cf\*cb)+(1–αf)cb
15. Screen blending: c = 1–(1–cf)(1–cb)
16. Transformations:(translation, rotation, scale, skew→rotation, non-uniform scale, rotation)
    1. 2D: p’ = TSRTSR…p = Mp → T= ; R=; S=
    2. 3D: p’ = MP → T=; S=; Rx=; Ry=; Rz=
17. Viewsing transformations:
    1. Model transformation: model/object space → scene/world space
    2. View transformation: scene/world space → view/camera space
    3. Projection transformation: view/camera space → canonical view volume
    4. Perspective transformation: perspective projection → orthographic projection (eyes and cameras have a natural perspective view, unlike an orthographic projection)
       1.   → orthographic projection
       2.  → perspective transformation
18. GPU pipeline

The Graphics Processing Unit (GPU) pipeline, often referred to as the rendering pipeline, is a series of stages through which data undergoes to be transformed from 3D coordinates to 2D pixels that can be displayed on a screen. Here is a step-by-step breakdown of the main stages:

Vertex Fetch:

Input: Vertex data (like positions, normals, and texture coordinates) is fetched from memory.

Operation: The vertices are prepared for processing and sent to the next stage.

Vertex Shading:

Input: Vertices from the previous stage.

Operation: Custom transformations (often via shaders) are applied to each vertex. Typical operations include transformations from object space to screen space using matrices.

Tessellation (Optional):

Input: Patch data and tessellation levels.

Operation: Breaks higher-level geometric shapes into simpler ones (usually triangles). It allows for finer detail without sending dense mesh data.

Geometry Shading (Optional):

Input: Vertices from the previous stage.

Operation: Generates new shapes or modifies existing ones. One common use is to generate billboards or particle systems.

Clipping:

Input: All the shapes from the previous stages.

Operation: Anything outside the view frustum (the volume of space deemed visible) gets clipped away, ensuring that only visible elements are processed further.

Rasterization:

Input: Shapes (typically triangles) from the clipping stage.

Operation: The shapes are broken down into fragments/pixels. It determines which pixels on the screen the shapes will cover.

Fragment Shading (or Pixel Shading):

Input: Fragments from the rasterization stage.

Operation: Determines the color and depth of each fragment. This is often where lighting, texturing, and other pixel-level operations occur.

Depth and Stencil Testing:

Input: Fragments and depth/stencil buffers.

Operation: Checks each fragment against the depth and stencil buffers to decide if it's visible or if it should be discarded.

Blending:

Input: Fragments and the current content of the framebuffer.

Operation: Combines the fragments with existing data in the framebuffer, which might involve operations like transparency or additive blending.

Framebuffer Operations:

The final pixel values are written to the framebuffer, which is then displayed or used in further operations.

1. WebGL
2. Initialize

<canvas id=”mycanvas”></canvas>

window.onload = function() –will run after have canvas container

{

canvas = document.getElementById(“mycanvas”);

gl = canvas.getContext(“webgl”);

const pixelRatio = window.devicePixelRatio || 1;

canvas.width = pixelRatio \* canvas.clientWidth;

canvas.height = pixelRatio \* canvas.clientHeight;

gl.viewport(0, 0, canvas.width, canvas.height);

gl.clearColor(1, 1, 1, 0); —RGBA (background color–useful only when clearing the scren)

gl.lineWidth(1.0); —thickness of line

}

1. Blind buffer → Vertex attributes

var color/position\_buffer = gl.createBuffer();

gl.bindBuffer(

gl.ARRAY\_BUFFER,

position\_buffer);

gl.bufferData(

gl.ARRAY\_BUFFER,

new Float32Array(positions),

gl.STATIC\_DRAW);

1. Vertex shader

attribute vec3 pos;

attribute vec4 clr;

uniform mat4 trans;

varying vec4 vcolor;

void main()

{

gl\_Position = trans \* vec4(pos,1); —vertex shader to rasterizer

vcolor = clr;  —vertex shader to fragment shader

}

1. Fragment Shader

precision mediump float;

varying vec4 vcolor;

void main()

{

gl\_FragColor = vcolor; –output color set to the vcolor

}

1. Render → clear the screen, set the program, do the rendering

var p = gl.getAttribLocation(prog, ‘pos’);

gl.bindBuffer(gl.ARRAY\_BUFFER, position\_buffer);

gl.vertexAttribPointer(p, 3, gl.FLOAT, false, 0, 0); –p is a 3d vector of float xyz value and will start from the beginning

fl.enableVertexAttribArray(p); –enable attribute so that gpu can get the data

–color are do the same

1. Curve: f(t) = (1 - t)p0 + tp1; f(t) = (p0 - 2p1 + p2)t2 + 2(p1 - p0)t + p0; f(t) = (1 - t)3p0 + 3(1 - t)2tp1 + 3(1 - t)t2p2 + t3p3 → Bezier curves
   1. Are affine invariant → When get transform the points, it can transform the curve
   2. Derivatives → f’(t) give the direction of the tangent line
   3. C0 continuity: they touch each other
   4. C1 continuity: their derivative direction & magnitude are same at the point they join
   5. G1 continuity: their derivative direction is same at the point they join
   6. C2 continuity: the change of the direction is goint to be continuous
   7. G2 continuity: same as C2 but the magnitude can be different.
   8. De Casteljau’s algrithm:  to calculate points along a Bézier curve by recursively subdividing it into smaller segments
   9. B-Splines → f(t) = ∑Ni,n(t)pi, which ∑Ni,n(t) = 1
   10. Rational curves: f(t) = (∑ai(t)pi)/(∑aj(t)) = ∑aj\*(t)pi; rational bezier curves: f(t) = (∑bi,n(t)wipi)/(∑bi,n(t)wi); non-uniform rational B-Splines(NURBS): (∑Ni,n(t)wipi)/(∑Ni,n(t)wi)
   11. NURBS curves and surfaces provide a flexible and precise way to describe complex shapes in a way that allows for both local and global control over the geometry.
   12. Catmull-Rom curves(uniform, chordal, centripetal): curve passes through the given control points (except the very first and last)
2. Surfaces: **Implicit surfaces**(方程); **Bezier patches**(a set of control points); **NURBS surfaces**(their ability to represent complex 3D shapes with precision while offering control over the smoothness and continuity of the surfaces); **polygonal meshes**(It consists of a collection of vertices, edges, and faces, where faces are typically polygons (usually triangles or quadrilaterals) that connect a set of vertices to form the surface of a 3D object) → winged edge: 2 vertex indices & 1 or 2 face indices; half edge: 1 vertex index & 1 face index;
3. **Triangular meshes**(the surface is approximated using a collection of triangles)
   1. Barycentric coordinates: p = αp0+ꞵp1+𐔢p2 → α+ꞵ+𐔢=1 ⇒ c = αc0+ꞵc1+𐔢c2 ⇒ α=a0/a; ꞵ=a1/a; 𐔢=a2/a, for a0+a1+a2=a (The GPU computes Barycentric coordinates during rendering.)
   2. List of vertices: x,y,z position per vertex & list of triangles: vertex indices
4. Subdivision: Catmull-Clark Subdivision (widely used in 3D modeling software, animation, and computer graphics to generate high-quality surfaces from low-resolution base meshes)