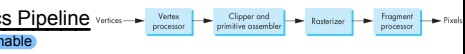
INTRODUCTION

OpenGL graphics API = STATE MACHINE — current state persists until new values set

Graphics Pipeline

programmable



- **Vertex specification** — Sending vertices to pipeline, defines boundary of primitives
- **Vertex shader** — Coordinate transforms & vertex color model-view-projection matrix
- **Tesselation** — Vertices divided into smaller primitives
- **Geometry shader** — Process a set of vertices as a primitive returning one or more primitives
- **Primitive assembly** — Composing primitives from vertices + face culling
- **Vertex post-processing** — Transform feedback: primitives written to buffer + clip = clip space — window (NDC) — viewport transform
- **Rasterization** — Creates fragments for each primitive (position, color, depth
- **Fragment Shader** — Color + depth + stencil, updates fragments generated by rasterizer and updates pixels in framebuffer
- **Per-sample operations** — Stencil & depth test, blending

Primitives

POINTS, TRIANGLES, LINES, LINE_STRIP, POLYGON, TRIANGLE_FAN

- front side of primitive = counter clockwise
- `glCullFace(GL_BACK);`

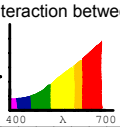
OpenGL converts quads & polygons into triangles

- simple: edges cannot cross
- convex: points on line segment between two points are also in the polygon
- flat: all vertices in the same plane

△ TRIANGLES ARE MOST EFFICIENT

Color

Interaction between physical light & object



Rods: intensity
Cones: color
Fovea: small region with no rods
RGB: 8bits = LDR, 32 bits = HDR

INPUT & INTERACTION

Server Model

Server performs a task for client

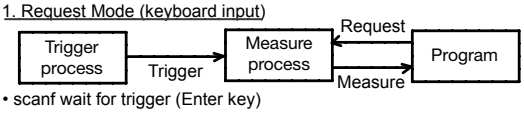
- OpenGL application uses graphic server
- Output services = display Input services = keyboard

Input Modes

- Input devices contain a trigger to send signal to OS (mouse, key)
- When triggered input device return info — a measure

Measure Modes

1. Request Mode (keyboard input)



- scanf wait for trigger (Enter key)

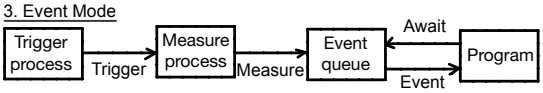
2. Sample Mode

- input is immediate, no trigger, device is active before function call

Issues with 1&2

- need to identify device
- don't know which device user will use
- hard for graphical applications

3. Event Mode



- more than one input device, each trigger generates event and the measure is put into event queue (window, mouse, keyboard)

```
glfwSetCursorPosCallback(GLFWwindow*, callback)
void callback(window, double xpos, double ypos)
glfwSetErrorCallback(callback)
void callback(int error, const char* description)
glfwPollEvents() - process events in the queue
```

Double Buffering

- need to redraw at fast rate, if framebuffer changes = flickering
- use two framebuffers: front (display) & back (draw)
- `glfwSwapBuffers(GLFWwindow*)`
- control speed: computer runs program so fast — blur
  - use a timer (OS timers), lock buffer swap to refresh rate
- `glfwSwapInterval(1);`

Hidden Surface Removal

1. Object Space Approach $O(n^2)$

Use pairwise testing between polygons

Painter's Algorithm

Render polygons back to front so further ones are painted over

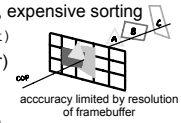
Depth Sort $O(n\log n)$

Requires ordering of polygons

- hard cases: overlap, cyclic overlap penetration
- advantages: simple, easy transparency
- disadvantages: hard for complex geometry, expensive sorting

2. Image Space Approach $O(nmk) \approx O(k)$

Look at each projector (for n x m framebuffer) and find closest k polygons



Z-buffer algorithm

- use z-buffer to store depth of closest objects
- as each polygon is rendered, compare to z-buffer depth
  - if less, update color & z-buffer
  - advantages: no sorting, independent of primitives
  - disadvantages: memory, tough transparency & blend, z-fighting

```
glfwWindowHint(GL_DEPTH_BITS, 24);
glEnable(GL_DEPTH_TEST); glDepthFunc(GL_LESS);
glClear(GL_DEPTH_BUFFER_BIT);
```

GEOMETRY & TRANSFORMATIONS

Scalars

Members of sets which can be combined by two operations

- associative: $2 \cdot (3 \cdot 4) = (2 \cdot 3) \cdot 4$ • inverse
- commutative: $2 \cdot 3 \cdot 4 = 4 \cdot 3 \cdot 2$

Vectors

Two attributes: direction & magnitude

addition: $\vec{u} + \vec{v}$ subtraction: $\vec{u} - \vec{v}$

Coordinate Systems

basis: linearly independent, can't write one vector in terms of others

In 2D, vector is a linear combination of two non-parallel basis vectors: $\vec{V}_{2D} = a\vec{u} + b\vec{v}$

In 3D requires 3 non-parallel, non-coplanar basis vectors $\vec{V}_{3D} = a\vec{u} + b\vec{v} + c\vec{w}$

- coplanar: linearly dependent vectors

Orthonormal Coordinate Systems

- basis vectors that are unit vectors & perpendicular are orthonormal

Dot Product

$$\vec{a} \cdot \vec{b} = \|\vec{a}\|\|\vec{b}\|\cos\theta = \sum_{i=1}^{N} \vec{a}_i \vec{b}_i$$

Commutative & distributive

Projection

$$\vec{x} = \frac{\vec{u} \cdot \vec{v}}{\|\vec{v}\|} = \|\vec{u}\|\cos\theta$$

Cross Product

$$\|\vec{a} \times \vec{b}\| = \|\vec{a}\|\|\vec{b}\|\sin\theta$$

distributive & intransitive

$$s = u \times v = \begin{bmatrix} u_2v_3 - u_3v_2 \\ u_3v_1 - u_1v_3 \\ u_1v_2 - u_2v_1 \end{bmatrix}$$

```
glm::normalize
glm::cross
```

Matrices

OpenGL matrices are column major

$$\begin{bmatrix} m(0) & m(3) & m(6) \\ m(1) & m(4) & m(7) \\ m(2) & m(5) & m(8) \end{bmatrix}$$

If a square matrix M is non-singular (no determinant) there is a unique inverse: M^{-1}: $MM^{-1} = M^{-1}M = I$, $(MPQ)^{-1} = Q^{-1}P^{-1}M^{-1}$

Linear Transformations

1. Scaling

$$\begin{bmatrix} s & 0 \\ 0 & t \end{bmatrix}\begin{bmatrix} v_x \\ v_y \end{bmatrix} = \begin{bmatrix} sv_x \\ sv_y \end{bmatrix}$$

2. Shearing

$$\begin{bmatrix} 1 & s \\ t & 1 \end{bmatrix}\begin{bmatrix} v_x \\ v_y \end{bmatrix} = \begin{bmatrix} v_x + sv_y \\ tv_x + v_y \end{bmatrix}$$

3. Rotation

$$\begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix}\begin{bmatrix} v_x \\ v_y \end{bmatrix} = \begin{bmatrix} v_x cos\theta - v_y sin\theta \\ v_x sin\theta + v_y cos\theta \end{bmatrix}$$

Homogeneous Coordinates

$p' = Mp + d$ $v' = Mv$ (No d cause can't translate a vector)

- can represent both points & vectors using 4th coordinate

$$p = \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} \quad v = \begin{bmatrix} v_x \\ v_y \\ v_z \\ 0 \end{bmatrix} \rightarrow \begin{bmatrix} m_{xx} & m_{xy} & m_{xz} & d_x \\ m_{yx} & m_{yy} & m_{yz} & d_y \\ m_{zx} & m_{zy} & m_{zz} & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} v_x \\ v_y \\ v_z \\ 0 \end{bmatrix}$$ → displacement becomes 0

Homogeneous rotation: $\vec{v}' = p_2' - p_1' = R \cdot p_2 - R \cdot p_1 = R \cdot (p_2 - p_1) = R \cdot \vec{v}$

Homogeneous scale: $\vec{v}' = p_2' - p_1' = S \cdot p_2 - S \cdot p_1 = S \cdot (p_2 - p_1) = S \cdot \vec{v}$

Homogeneous translation: $\vec{v}' = p_2' - p_1' = (p_2 - d) - (p_1 - d) = (p_2 - p_1) - (d - d) = \vec{v}$

Homogeneous Coordinate Conversion

- divide by homogeneous coordinate w if $w \neq 0$ & drop 4th coordinate

$$P_h = \begin{bmatrix} p_x/w \\ p_y/w \\ p_z/w \\ w/w \end{bmatrix} \rightarrow P_{3D} = \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix}$$

Affine Transformations

$$Translation: \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad Scale: \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad Shear: \begin{bmatrix} 1 & z_1 & z_2 & 0 \\ z_3 & 1 & z_4 & 0 \\ z_5 & z_6 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation $Rz = \begin{bmatrix} cos\theta & -sin\theta & 0 & 0 \\ sin\theta & cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$ $Ry = \begin{bmatrix} cos\theta & 0 & sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -sin\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$ $Rx = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & -sin\theta & 0 \\ 0 & sin\theta & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

VIEWING & PROJECTIONS

To represent points need an origin, with basis vectors = frame

Inside frame: $\vec{v} = a_1\vec{v_1} + \cdots + a_n\vec{v_n}$ $p = p_0 + b_1\vec{v_1} + \cdots + b_n\vec{v_n}$

Change of Coordinate Systems

$$[a_1\ a_2\ a_3][v_1\ v_2\ v_3]^T = [b_1\ b_2\ b_3][u_1\ u_2\ u_3]^T \qquad \vec{a} = [a_1\ a_2\ a_3], \vec{b} = [b_1\ b_2\ b_3]$$

$$u_1 = m_{11}v_1 + m_{12}v_2 + m_{13}v_3$$
$$u_2 = m_{21}v_1 + m_{22}v_2 + m_{23}v_3$$
$$u_3 = m_{31}v_1 + m_{32}v_2 + m_{33}v_3$$

$$M = \begin{bmatrix} m_{xx} & m_{xy} & m_{xz} \\ m_{yx} & m_{yy} & m_{yz} \\ m_{zx} & m_{zy} & m_{zz} \end{bmatrix} \qquad a = M^T \cdot b \qquad u = M \cdot v$$

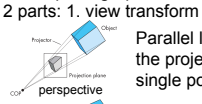
Homogeneous change of base: $M = \begin{bmatrix} m_{xx} & m_{xy} & m_{xz} & 0 \\ m_{yx} & m_{yy} & m_{yz} & 0 \\ m_{zx} & m_{zy} & m_{zz} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$ = affine transform

OpenGL — start in world frame (model), change world frame to camera frame — Model-View Matrix

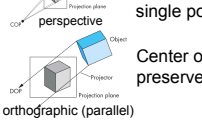
Viewing & Projection

In computer graphics collapse 3D world to 2D = projection

2 parts: 1. view transform (camera) 2. projection.



Parallel lines that are not parallel to the projection plane converge to a single point (vanishing point)



Center of projection at infinity, preserves distances, angles, shapes

Three Point Perspective

- no principal face parallel to projection plane • 3 vanishing points
- non uniform distances • preserved angles parallel to proj. plane
- objects further from viewer will look smaller

OpenGL Viewing Pipeline

- positioning camera • selecting a lens (projection matrix)
- clipping — set view volume (cube of length 2 centered at (0,0,0)

GLM provides function to define camera pose:

```
mat4 camera = lookAt(vec4 eye, vec4 at, vec4 up)
```

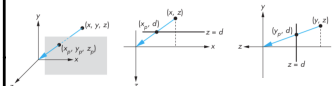
Orthographic Projection

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{2}{n-f} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad N = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{n-f} & \frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Z is zero, no depth

Perspective Projection

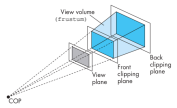
Using similar triangles:



$$x_p = \frac{x}{z/d} \qquad y_p = \frac{y}{y/d} \qquad z_p = d$$

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} \rightarrow p = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \rightarrow q = Mp = \begin{bmatrix} x \\ y \\ z \\ z/d \end{bmatrix} = \begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix}$$

$$N = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$



objects only visible inside frustum

```
glm::ortho(left, right, bottom, top, near, far)
glm::perspective(fovy, aspect, near, far)
```

Pipeline View

4D → 3D (Clip coordinates)

Modelview transformation → Projection transformation → Perspective division → Clipping → Projection → Rasterization

Clipping (NDC) 3D → 2D (orthographic) viewport transform

Normalization (N-matrix)

- convert all projections to orthogonal projections with default view volume = efficient clipping
- delay hidden projection until the end, important for hidden surface removal (retain depth info as long as possible)

BUILDING MODELS

Mesh

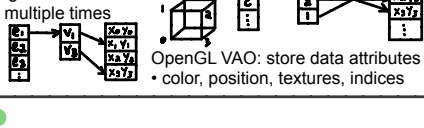
Set of polygons that share vertices & edges to make an object

- geometry: location of vertices
- topology: organization of vertices & edges

Vertex List

data for location of vertices, if drawn by edges — shared edges drawn multiple times

Edge List



OpenGL VAO: store data attributes

- color, position, textures, indices

SHADOWS

Light Sources

1. Point Light

Isotropic: light rays are emitted in all directions

- single point, easy to compute, not realistic (hard shadows)

2. Spotlight

Anisotropic: light rays emitted in one direction

- emits light in a cone, easy to compute, not realistic (hard shadows)

3. Directional Light

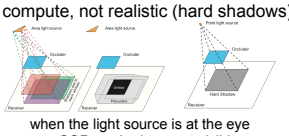
All light rays are parallel, light is located at ∞

- results in hard shadows

4. Area Light

Light emitted from a surface

- computationally expensive, realistic (produces soft shadows)



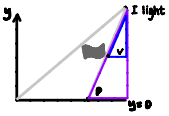
when the light source is at the eye or COP — shadows not visible

Shadow Algorithms

1. World level global visibility tests (accurate yet expensive)
• ray casting, ray tracing, radiosity, monte carlo simulation
2. Image level or local visibility tests (approximate)
• inexpensive, projective shadows, shadow map & shadow volume

Projective Shadows

• projection of an occluder polygon is a shadow polygon
• given a point light source & polygon, vertices of shadow are the projections of the original polygon onto receiver surface



Similar triangles:

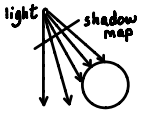
$$\frac{P_x - I_x}{V_x - I_x} = \frac{I_y}{I_y - V_y} \qquad M = \begin{bmatrix} I_y & -I_x & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & -I_z & I_y & 0 \\ 0 & -1 & 0 & I_y \end{bmatrix}$$

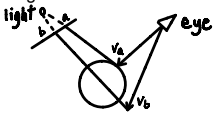
projection matrix for y=0

Limitations
• z-fighting — add offset `glPolygonOffset()`
  • draw receivers first, turn z-depth test off, then draw shadow polygons then draw rest of the scene
• shadow polygons fall outside receiver — use stencil buffer
• shadows have to be rendered each frame (render to texture)
• restrictive to planar objects
• can't cast shadows on self or other objects

Shadow Maps

• render scene from a light source, depth buffer will contain distances to each fragment
1st pass: store depth in a texture (depth map/shadow map)
2nd pass: compare the distance from the fragment to the light source with distance in shadow map
  • if depth in shadow map is less than the distance from the fragment to the light source, the fragment is in shadow



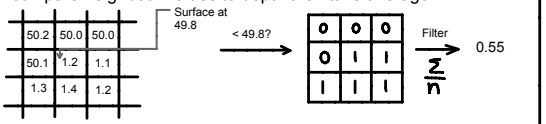


Depth From Shadow Map

Use a 4x4 perspective projection matrix from light source
• if shadowMap(x, y) < z — in shadow
• if shadowMap(x, y) > z — in light
Limitations
• shadow acne (fragments sample same value) — use bias
• peter panning — bias causes shadow to be too offset
• field of view
• aliasing: under sampling shadow map
  • reprojection aliasing (difference between projected point & measured point (bad when camera & light are opposite)

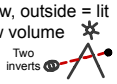
Percentage Closer Filtering (PCF)

Average results of depth test
• compare neighbour values to depth then take average



Shadow Volumes

Shadow cast by an object blocking light is a volume
1. Compute shadow volume formed by a light source & set of shadow objects
2. Check if point is in shadow volume, inside = shadow, outside = lit
• get polygon boundary representation for the shadow volume
• render scene with lights (depth map)
• clear stencil buffer & render shadow volume
• when rendered fragment of shadow volume is closer than depth of other objects, invert bit in stencil value
• after render, if bit is on, fragment is in shadow volume



Advantages:
• general purpose graphics hardware
• only use stencil buffer
• finer shadow resolution (no aliasing)
Disadvantages:
• bottle neck at rasterizer
• many shadow volumes cover pixels

PROGRAMMABLE SHADERS

Vertex Processor

• vertex, normal transformations, texture coordinates, lighting, color

Fragment Processor

Fragment operations after interpolation & rasterization of vertex data
• phong lighting, textures, fog, color sum

Buffer Objects

```
uint vao; glGenVertexArrays(1, &vao);
glBindVertexArray(vao);
uint vbo; glGenBuffers(1, &vbo);
glBufferData(GL_ARRAY_BUFFER, vbo);
glBufferData(...);
glEnableVertexAttribArray(0);
glVertexAttribPointer(location, size, GL_FLOAT,
GL_FALSE, stride, offset);
```

Shaders

```
glCreateShader, glCompileShader, glAttachShader,
glLinkProgram
```

Drawing

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
glUseProgram(shaderID);
uint m = glGetUniformLocation(shaderID, "model")
glUniformMatrix4fv(m, 1, false, &model_matrix[0])
glBindVertexArray(vao);
glDrawArrays(GL_TRIANGLES, 0, num_vertices)
```

Instancing

```
glDrawArraysInstanced()
glDrawElementsInstanced()
```
Method 1: `uniform offset[] with gl_InstanceID`
Method 2: `glVertexAttribDivisor(index, divisor)`
• pass in offsets via vertex attributes

Cleanup

```
glDeleteProgram, glDeleteShader,
glDeleteBuffers, glDeleteVertexArrays
```

GLSL (Similar to C)

Uniform can read but not write in shaders `(uniform mat4 model)`
in - datatype — read only variables (variable names must match)

LIGHTING & SHADING

Global Illumination

ray tracing, radiosity, photon mapping
• follow light through scene, direct + indirect lighting, accurate/ realistic but expensive (usually offline)

Local Illumination

Local interaction between light, surface and viewer
• color determined by surface normals, relative camera position & light position

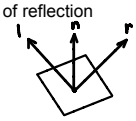
Normal Vector

Method 1: (a, b, c) — plane equation: f(p) = ax + by + cz + d = 0
$n_0 = [a\ b\ c]^T \longrightarrow$ normalize $n = \frac{n_0}{\|n_0\|}$
Method 2: plane given by p_0, p_1, p_2
• points must not be co-linear — cross product defined as 0
• $n = (p_1 - p_0) \times (p_2 - p_0)$ — normalize $n = \frac{n_0}{\|n_0\|}$

Reflected Vector

• perfect reflection: angle of incidence equals angle of reflection
• l, n, r are on the same plane

$r = al + bn \quad l \cdot n = \cos\theta = n \cdot r \quad a = -1$
$n \cdot r = al \cdot n + b = l \cdot n \quad b = 2(l - n)$
$1 = r \cdot r = a^2 + 2abl \cdot n + b^2 \quad r = 2(l \cdot n) \cdot n - l$



Light Sources & Material Properties

Appearance depends on: light, material, viewer position

1. Point Light

• shading & shadows inaccurate
• similar highlight problem
• better with ray tracing

Attenuation

$$\frac{1}{a + bq + cq^2}, q = |p - p_0|$$

2. Directional Light

Direction source: $[x\ y\ z]^T$ (vector only, no position)

3. Spotlight

Cutoff by a cone determined by θ

4. Ambient Light

Independent of light source, lights entire scene, inexpensive

Phong Illumination

Calculate color for arbitrary point on surface
1. Start with ambient light (rgb)
2. Add contributions from each light source
3. Clamp result [0, 1]

Light Source Contributions

1. Ambient Reflection

$I_a = k_a L_a$ ambient material coefficient: $0 \le k_a \le 1$
ambient light color: L_a (vec3)

2. Diffuse Reflection

Diffuse reflector scatters light equally

$I_d = k_d L_d (l \cdot n)$ diffuse material coefficient: $0 \le k_d \le 1$
diffuse light color: L_d (vec3) $(l \cdot n) = \cos\theta$

3. Specular Reflection

Shiny surfaces have high specular coefficients

$I_s = k_s L_s (v \cdot r)^\alpha$ specular material coefficient: $0 \le k_s \le 1$
specular light component: L_s (vec3)
shininess coefficient: α $(v \cdot r) = \cos\phi$

$$I = \frac{1}{a + bq + cq^2}[k_d L_d (l \cdot n) + k_s L_s (r \cdot v)^\alpha] + k_a L_a$$

Gouraud Shading

Phong shading at vertex level, interpolated for whole triangle, need many vertices

Phong in GLSL

• light & material — uniforms
• light computed in view space (normals, eye, light, vectors)
• vertex shader: vertices to clip space, lighting data to view space
• fragment shader: ambient, diffuse & specular

CULLING & CLIPPING

Back Face Culling

Back face: surface of object that faces away from the eye
• In a closed polygonal surface, back faces have normals pointing away from viewer

$V_{view} \cdot N > 0$

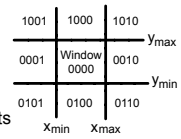
```
glCullFace(GL_BACK)
glEnable(GL_CULL_FACE) // hide cw faces
glCullFace(GL_FRONT)
glEnable(GL_CULL_FACE) // hide ccw faces
glCullFace(GL_FRONT_AND_BACK) // hide polygon
```

Clipping (line and polygon clipping)

Occurs after CTM and before rasterization. Any geometry outside canonical view volume is clipped

Line Clipping — Cohen Sutherland

• Extend edges of clip rectangle into 9 regions
• Assign a 4-bit code to each region (TBRL)
• Reject lines when: the logical AND of the endpoints is non-zero



6th bit = 1 if $z < z_{min}$
5th bit = 1 if $z > z_{max}$
4th bit = 1 if $y > y_{max}$
3rd bit = 1 if $y < y_{min}$
2nd bit = 1 if $x > x_{max}$
1st bit = 1 if $x < x_{min}$

1. Find the endpoints that lie outside
2. Compute the 4 bit codes for each endpoint $\text{if } p_1 \mid p_2 = 0$ Completely inside
3. If p1 AND p2 ≠ 0 determine the intersection point (LRBT)
3. Replace the end point by intersection point
4. If the logical AND of the bit codes of the new and old end points is non-zero then reject.

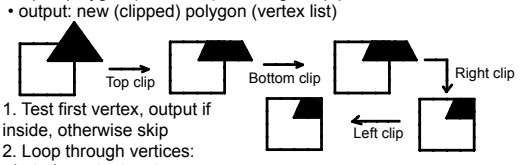
Polygon Clipping

Clip polygons against other polygons eg. hidden surface removal

1st approach: clip and then join pieces to form a single polygon (difficult to manage)
2nd approach: tesselate and clip triangles (common solution)

Sutherland-Hodgeman

Clipping polygons against rectangular polygons (apply Cohen-Sutherland line clipping on and edge-by-edge basis)
• At each clip plane (4 in 2D, 6 in 3D):
  • input: polygon (vertex list) and single clip plane
  • output: new (clipped) polygon (vertex list)



1. Test first vertex, output if inside, otherwise skip
2. Loop through vertices:
• in-to-in: output vertex
• in-to-out: output intersection
• out-to-in: output intersection and vertex
• out-to-out: no output

View Frustum Culling

• optimization done in software to minimize the amount of draw calls
• do not draw if outside the view frustum
• scene can be structured as a hierarchy of bounding primitives
  • Axis Aligned Bounding Boxes (AABB)
  • Oriented Bounding Boxes (OBB), Bounding Spheres
  • before drawing an object we test if the bounding primitive is inside the view frustum, if not don't draw primitive

RASTERIZATION

Convert screen coordinates (float) to pixels (int). Write pixels into framebuffer:
• depth (z-buffer)
• shadows (stencil buffer)
• display (frame buffer)
• blending (accumulation buffer)

Digital Differential Analyzer (DDA)

Given a line with endpoints (x_1, y_1) & (x_2, y_2), it is represented as y = ax + b.

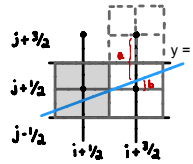
$$a = \frac{y_2 - y_1}{x_2 - x_1} = \frac{\Delta y}{\Delta x} \quad \text{if } \Delta x = 1 \text{ pixel } \Delta y = a\Delta x = a$$

```
Data: P1(x1,y1) P2(x2,y2) vec3 color
Result: draws a line
Function: writePixel
m = (y2 - y1) / (x2 - x1)
y = y1
for i = x1 to x2 do
    y += m
    writePixel(i, round(y), color)
```

Problems
• requires floating point addition
• missing pixels for steep slopes
• assume $0 \le m \le 1$

Bresenham's Algorithm

Eliminate floating point addition from DDA. (Assume $0 \le m \le 1$)
• assume pixel centres are halfway between integers



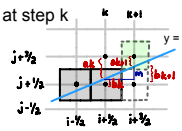
Decision variable — a - b
if a - b > 0 — lower pixel
if a - b ≤ 0 — higher pixel

Goal: avoid computing a - b

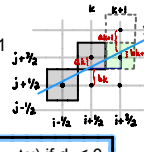
Step 1: Re-scale

$d = (x_2 - x_1)(a - b) = \Delta x(a - b)$ d is always an integer (round it)

Step 2: Compute d at step k + 1 from d at step k
• Case 1: j did not change $d_k > 0$
a decreases by m, b increases by m
(a - b) decreases by $2m = 2(\Delta y/\Delta x)$
$\Delta x(a - b)$ decreases by $2\Delta y$



• Case 2: j did change $d_k \le 0$
a decreases by m - 1, b increases by m - 1
(a - b) decreases by 2m - 2 = 2(Δy/Δx - 1)
$\Delta x(a - b)$ decreases by $2(\Delta y - \Delta x)$



$d_{k+1} = d_k - 2\Delta y$ if $d_k > 0$ $\quad$ $d_{k+1} = d_k - 2(\Delta y - \Delta x)$ if $d_k \le 0$

```
Data: P1(x1,y1) P2(x2,y2) vec3 color
Result: draws line
Function: drawLine
int x, y = y1;
int dx = 2(x1 - x2),dy = 2(y2 -y1)
int dydx = dy - dx
D = (dy - dx) / 2
For x = x1 to x2 do
    writePixel(x, y, color)
    if D > 0 then
        D -= dy
    else
        y++; D -= dydx
```

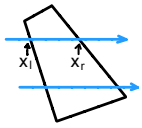
Details
Need different cases to handle m > 1
• highly efficient
• easy to implement in software and hardware
• widely used

Scan Conversion of Polygons

filling polygon (inside/outside), pixel shading (color interpolation), blending (accumulation, not just writing), depth values (z-buffer, hidden-surface removal), texture coordinate interpolation (texture mapping)

Filling convex polygons

1. Find top and bottom vertices
2. List edges along left and right sides
3. For each scan line from bottom to top
   1. find left and right end points (x_l & x_r)
   2. fill pixels between x_l & x_r
   3. Bresenham's algorithm to update x_l & x_r

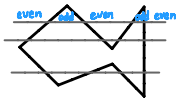


Concave polygons

Test every pixel in the raster to see if it lies inside the polygon

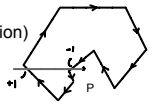
Even-odd Parity Test

• parity is initially even — each intersection inverts parity bit
• if parity is odd — draw (interior point)
• if parity is even — don't draw (exterior point)



Non-zero Winding Number Test

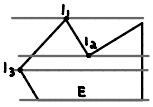
Use direction of the line as well as intersections
• draw a line from a point P to infinity (any direction)
• initially set winding number W = 0 for point P
• intersection from right-to-left — W++
• intersection from left-to-right — W--
• if W = 0 — P is outside, else inside



Scan-line Method

• proceeding from left to right the intersections are paired and intervening pixels are set to the specified intensity
1. Find intersections of scan line with all the edges in the polygon
2. Sort the intersections by increasing x-coordinates
3. Fill the pixels between pair of intersections

• intersections like I_1 & I_2 should be considered as two intersections
• intersections like I_3 should be considered as one intersection
• horizontal edges like E need not be considered



Aliasing

Artifacts produced during scan conversion (inevitable cts to discrete)
• aliasing caused by sampling a cts image at grid points
  • causes jagged edges and moire pattern

Solutions
• use area averaging at boundaries
• supersampling (mostly for offline)
  • render 3x3 grid of mini pixels for each pixel, take average
  • can be done adaptively, stop if colours are similar
• Stochastic sampling: avoid sample position repetitions
• Stratified sampling (jittering): perturb a regular grid of samples

Temporal Aliasing: fast moving objects are blurred
Solution: supersample in time and average

RAY TRACING

Local Illumination (OpenGL)

object lighting is independent, no light scattering between objects, no real shadows, no reflection or transmission

Global Illumination

ray tracing (highlight, reflection, transmission), photon mapping, radiosity (interreflections), precomputed radiance transfer (PRT)

Object Space vs Image Space

Object Space
• graphics pipeline for each object render
• efficient pipeline architecture, real-time
• difficulty: object interactions (shadows, reflections)

Image Space
• ray tracing: for each pixel determine color
• pixel-level parallelism
• difficulty: very intensive computation (offline)

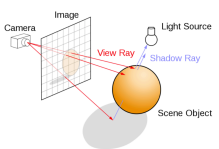
Forward Ray Tracing

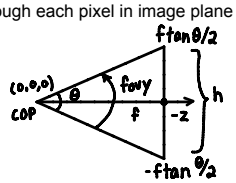
• shoot light rays from each light source
• rays bounce off the objects
• simulates paths of photons

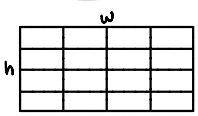
Problem: rays will miss camera and not contribute to image

Backward Ray Tracing

Shoot one ray from camera through each pixel in image plane







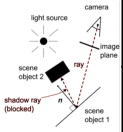
$aspect\ ratio\ (\alpha) = {}^{w}/_{h}$
$h = 2f\tan{}^{\theta}/_{2}$
$w = \alpha h = 2\alpha f\tan{}^{\theta}/_{2}$

Calculating Pixel's Color

1. Phong model (local)
2. Shadow rays
3. Specular reflection (recursively)
4. Specular transmission (recursively)

Shadow Rays

Determine if light really hits surface point
• cast shadow ray from intersection point to each light
• if shadow ray hits opaque object, no contribution from light
• if shadow ray can reach light, apply Phong model



Reflection Rays

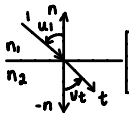
$r = 2(l \cdot n) \cdot n - l$

For specular component of illumination, compute reflection ray

Transmission Rays

Light transmitted through surfaces, compute transmission ray

Index of refraction is speed of light, relative to speed of light in vacuum (1.0), air (1.000277), water (1.33), glass (1.49)



$$\frac{\sin(u_i)}{\sin(u_t)} = \frac{\eta_t}{\eta_i} = \eta$$

Translucency

Most real objects are not transparent but blur the background image
• scatter light on other side of surface (stochastic sampling = distributed ray tracing)

Ray Tracing Algorithm

1. For each pixel (x, y) fire a ray from COP through (x,y)
2. For each ray and object, calculate closest intersection
3. For closest intersection point p
   • calculate surface normal
   • for each light source, fire shadow ray
   • for each unblocked shadow ray evaluate Phong

Advantages: relatively accurate shadows, reflections refractions
Disadvantages: slow (only pixel parallelism, not pipeline parallelism), aliasing, inter-object diffuse reflections require many bounces

RAY TRACING - GEOMETRIC QUERIES

Intersection of Rays and Parametric Surfaces

Ray in parametric form:

origin: $p_0 = [x_0 \quad y_0 \quad z_0]^T$ $\quad d = [x_d \quad y_d \quad z_d]^T$

assume d is normalized: $x_d^2 + y_d^2 + z_d^2 = 1$

$$p(t) = p_0 + dt$$

Surface in parametric form: $q = g(u, v)$

Solve: $p_0 + dt = g(u, v)$ — Three equations three unknowns (t, u, v)

Surface in implicit form: any point q lying on the surface satisfies the equation f(q) = 0 — substitute ray equation for q:

$f(p_0 + dt) = 0$ Solve for t (univariate root finding), or numerical approximation

Ray-Sphere

A sphere has center: $c = [x_c \quad y_c \quad z_c]^T$ & radius r

$$f(q) = (x - x_c)^2 + (y - y_c)^2 + (z - z_c)^2 - r^2 = 0$$

Plug in: $x = x_0 + x_d t \quad y = y_0 + y_d t \quad z = z_0 + z_d t$ → simplify $at^2 + bt + c$

Solve: $t_0, t_1: \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$ $\quad if\ t_0, t_1 > 0\ return \min(t_0, t_1)$

For lighting: $unit\ normal = \frac{1}{r}[(x_i - x_c) \quad (y_i - y_c) \quad (z_i - z_c)]^T$

Ray-quadric

Quadric: $f(p) = f(x, y, z) = 0$

Ray-polygon

Assume planar polygon in 3D, intersect ray with plane containing polygon

Plane implicit form: $ax + by + cz + d = 0$ with: $n = [a \quad b \quad c]^T$, $a^2 + b^2 + c^2 = 1$

Plug in: $x = x_0 + x_d t \quad y = y_0 + y_d t \quad z = z_0 + z_d t$

Solve: $t = -\frac{ax_0 + by_0 + cz_0 + d}{ax_d + by_d + cz_d} = \frac{-(n \cdot p_0 + d)}{n \cdot d}$

If n•d = 0, no intersection (ray parallel to plane)
If t ≤ 0, the intersection is behind ray origin

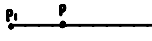
Test for Point Inside Polygon

• use even-odd rule or winding rule
• easier if polygon is in 2D
• easier for triangles (tesselate polygons)

1. Project the triangle and plane intersection point onto one of the planes x = 0, y = 0, z = 0 (pick plane not perpendicular to triangle
2. Do the 2D test in the plane by computing barycentric coordinates

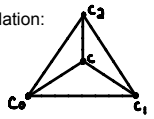
Barycentric Coordinates

Linear Interpolation: (1D)



$p(t) = (1 - t)p_1 + tp_2, \quad 0 \le t \le 1$
$p(t) = ap_1 + bp_2, \quad a + b = 1$

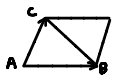
Linear Interpolation: (2D)



$$\alpha = \frac{Area(CC_1C_2)}{Area(C_0C_1C_2)}$$
$$\beta = \frac{Area(CC_0C_2)}{Area(C_0C_1C_2)}$$
$$\gamma = \frac{Area(CC_0C_1)}{Area(C_0C_1C_2)} = 1 - \alpha - \beta$$

Computing Triangle Area in 3D

cross product: $area(ABC) = \frac{1}{2}|(B - A) \times (C - A)|$



How to get correct sign: compare directions of vectors $(C_2 - C_0) \times (C_1 - C_0)$ or project to 2D

For triangles: $C_0C_1C_2$ vs $C_0C_1C_2$ either 0° (+) or 180° (−)

Project the triangle to XY plane:

$area(xy - projection(ABC)) = \frac{1}{2}((bx - ax)(cy - ay) - (cx - ax)(by - ay))$

SPATIAL DATA STRUCTURES

Ray Tracing Accelerations

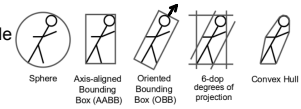
• faster intersections — bounding volumes
• fewer ray object intersections — hierarchical bounding volumes, spatial data structures, directional techniques
• fewer rays — adaptive tree depth control, stochastic sampling
• generalized rays — beams, cones

Spatial Data Structures

collisions, location queries, simulations, rendering, raytracing

Bounding Volumes

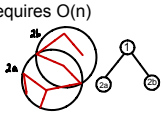
Wrap complex objects in simple ones
• effectiveness depends on:
• probability that ray hits bounding volume
• cost of calculating intersections with bounding volume and enclosed objects



Hierarchical Bounding Volumes

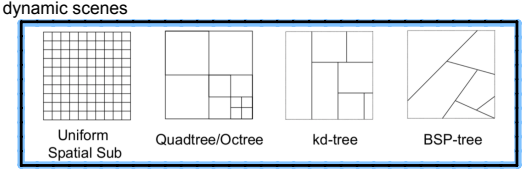
With simple bounding volumes ray casting still requires O(n) intersection tests
Use a tree data structure
• larger bounding volumes contain smaller ones
• difficult to compute but O(logn) runtime
1. recursive descend down the tree
2. if ray misses bounding volume no intersection, else recurse with the enclosed volumes and objects

Maintain near and far bounds to prune further



Spatial Subdivision

Bounding volumes enclose objects recursively
• expensive for animations
• hierarchical bounding volumes better for hierarchical objects and dynamic scenes



Grids
• 3D array of cells that tile space
• Each cell keeps a list of all intersecting surfaces
• only calculate intersections for cells with objects

Need to cache intersections for cells with multiple objects to intersect with closest object
Assesment:
• Poor when world is non-homogenous (most objects in 1 place)
• grid resolution: too large = too many surfaces per cell, too small = too many empty cells to traverse
• use Bresenham's algorithm for efficient traversal

Quadtrees

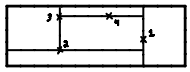
generalization of binary trees in 2D
• node (cell) is square, recursively split into 4 equal sub-squares
• stop division if square has 2 or less objects

Hard to traverse quadtree (going from big cell to small cell)

Octrees

generalization of quadtree in 3D (8 qual sub-cells)
Assement:
• better for most scenes (more adaptive)

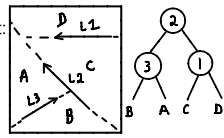
Kd Trees
• split at arbitrary interior point, split one dimension at a time



BSP Trees

Split space with any line (2D) or plane (3D)
• used for painters algorithm, ray casting

Inherent spatial ordering given viewpoint:
left subtree = front,
right subtree = behind
If line 3 chosen first, need to split line 2 into two halves



Building a Good Tree
• native partioning of n polygons yields O(n³) polygons (3D)
• algorithms with O(n²): try all polygons with fewest splits, do not need to split exactly along polygon planes

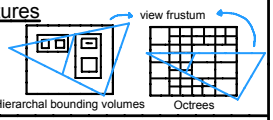
Painter's Algorithm

Each plane has the form Ax + Bx + Cz + D = 0
• plug coordinates: positive = front, zero = on plane, negative = back

Back-to-front: inorder traversal, far child first
Front-to-back; inorder traversal near child first
Do back face culling with same sign test

Culling with Spatial Data Structures

Accelerate culling
• accept or reject whole sets of objects (use spatial data structures)
• scene is mostly fixed



TEXTURE MAPPING

Color in OpenGL

glm::vec3 color(r, g, b, a)

Transparency

low alpha: transparent, high alpha: opaque
blending: combine new color (source) with existing (destination color)

```
glEnableFunc(source_factor, destination_factor)
glEnable(GL_BLEND)
```

Color Blending in Framebuffer

Select source (S_r, S_g, S_b, S_a) & destination (D_r, D_g, D_b, D_a) factor
`GL_ONE` (default source blend factor S_i= 1)
`GL_ZERO` (default destination factor D_i= 0)
`GL_SRC_COLOR, GL_SRC_ALPHA, GL_DST_ALPHA, GL_ONE_MINUS_ALPHA`

New color is: $(S_rR_s + D_rR_d \quad S_gG_s + D_gG_d \quad S_bB_s + D_bB_d \quad S_aA_s + D_aA_d)$

Good Choice $\alpha C_s + (1-\alpha)C_D$

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)
```

Drawing Order
1. Disable blending
2. Draw all opaque models
3. Enable blending
3. draw transparent object sorted back-front

Light Extinction

Probability of photon absorption or out-scattering is proportional to distance light travels through a medium
Monochromatic extinction: fog makes distant objects dimmer

Fog

$$C = f(z)C_i + \big(1 - f(z)\big)C_{fog}$$

C_i is incoming color (vertex or fragment)
C_{fog} is the fog color
f is the fog function (from [0, 1])
z is the distance of the point
• usually implemented in fragment shader

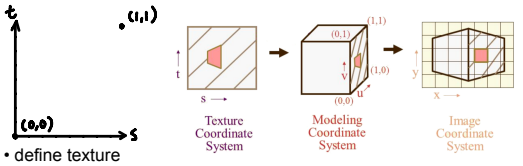
Linear:
$$f(z) = \frac{end - z}{(end - start)}$$
Inverse exponential:
$$f(z) = e^{-density \cdot z}$$
$$f(z) = e^{-(density \cdot z)^2}$$

Texture Mapping (OpenGL uses interpolation)

attach an image to an object, supporting transformations
• texture is a 2D array of data containing color values
• individual values in texture are texels (texture elements)

Texture Coordinates $s,t = u,v$

texture is addressed parametrically by coordinates s, t between 0,1



• define texture
• specify mapping from texture to surface
• lookup texture values during scan conversion
• during scan conversion:
  • map image coordinate system (x,y) to
  • modelling coordinate system (u,v) to
  • texture image (s,t)

OpenGL Texture Boundaries

Behaviour for texture coordinates outside 0,1
• repeat: assume texture is tiled

```
glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT)
```

• clamp to edge: texture coordinates are truncated to valid values

```
glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP)
```

• border: specify a border color

```
glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, R,G,B,A)
```

repeat  clamp  border 

Magnification & Minification

minification: more than one texel covers a pixel

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST)
```

magnification: more than one pixel covers a texel (requires extra border texel for filtering edges)

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR)
```

Mipmaps

keep textures prefiltered at multiple resolutions
• for each pixel linearly interpolate between closest levels (trilinear filtering)
• fast, easy for hardware

Texture Parameters

• pixels can be rbga, grayscale, alpha only
• replace: replace the polygon color with the texture color
• modulate: multiply polygon color with texture color or intensity
• compositing: composite texture with base color using an operator

Texture & Lighting (fragment shader)

Texture brightness
1. make the polygon white and light it normally
2. modulate the texture color with fragment color
3. if texture has opacity multiply texture color by opacity value

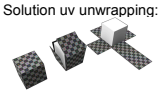
Render to Texture

Render to texture instead of framebuffer
1. create texture.
2. render scene in texture
3. bind generated texture while drawing on the framebuffer

Texture Issues

• must be in fast memory, if exceeded performance degrades
• applying multiple textures to the same point = multitexturing
• 1D textures: save memory if texture is 1D, like stripes
• 3D textures: memory intensive

Solution uv unwrapping:



Texture Atlas

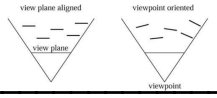
• avoid context switches for textures, place many assets in one file

Rendering Decals

polygons slightly above geometry
• constant z offset leads to z-fighting since z-buffer precision degrades with distance
• need to increase offset with distance (use projection matrix)
• or `glPolygonOffset` (slow)

Billboards

• geometry aligns with the camera (UI elements, particles)



SPLINES

Curve Specifications

• local control of shape: moving a point on the curve only affects the immediate neighbours
• stability: not subject to any conditions
• smoothness & continuity: being able to evaluate derivatives at any point along the curve
• ease of rendering: able to evaluate points along the curve

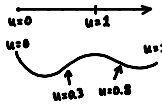
Curve Representations

Explicit form: $y = f(x)$ Implicit form: $f(x,y) = 0$

Advantages: simple
Disadvantages:
• single values: for given x there is only one y
• vertical lines: cant be modeled
•. Circle: need two equations subject to a condition

Advantages: y can be a multiple values function of x
Disadvantages:
Hard to specify, modify and control

Parametric form: $(x,y) = (f(u), g(u))$



Advantages: easy to specify, modify, control
Disadvantages: introduces an extra variable

Interpolation vs Approximation

Given a set of control points along a curve, connect them through
• interpolation: polynomial sections connect each control point
• approximation: polynomial curve plotted so some or all of control points are not on the curve path

Polynomial Interpolation

Lagrange interpolation: nth degree polynomial fits n+1 points curve
• curve becomes wiggly (non local) — bad method (need smooth)

Piecewise Polynomials (Splines)

Spline: curve broken into consecutive segments of low degree polynomials interpolating through control points

Cubic piecewise: $ax^3 + bx^2 + cx + d = 0$ (most common)

• Lowest order polynomias that interpolate two points and allow the gradient at each point to be defined (C^1 continuity)
• local control, but higher or lower degrees are possible

C^{-1}: curve discontinuous
C^0: curve continuous
C^1: 1st derivative continuous
C^2: 1st & 2nd derivatives continuous
C^n: nth derivatives continuous

Cubic Curves in 3D

$$p(u) = au^3 + bu^2 + cu + d = [u^3 \quad u^2 \quad u \quad 1][a \quad b \quad c \quad d]^T$$

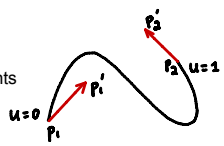
a, b, c, d = vec3, u = scalar

$$[x(u) \quad y(u) \quad z(u)] = [u^3 \quad u^2 \quad u \quad 1]\begin{bmatrix} a_x & a_y & a_z \\ b_x & b_y & b_z \\ c_x & c_y & c_z \\ d_x & d_y & d_z \end{bmatrix} \qquad p = [u^3 \; u^2 \; u \; 1]A$$

• magnitude of tangents determine how fast curve changes

Hermite Splines $u \in [0, 1]$

• interpolating piecewise cubic polynomial, pass through every point
• specify the start/end points & tangents



$$p(u) = au^3 + bu^2 + cu + d$$
$$p'(u) = 3au^2 + bu + c$$

$p(0) = p_1 = (x_1, y_1, z_1) = d$
$p(1) = p_2 = (x_2, y_2, z_2) = a + b + c + d$
$p'(0) = p'_1 = (x'_1, y'_1, z'_1) = c$
$p'(1) = p'_2 = (x'_2, y'_2, z'_2) = 3a + 2b + c$

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{bmatrix}\begin{bmatrix} a_x & a_y & a_z \\ b_x & b_y & b_z \\ c_x & c_y & c_z \\ d_x & d_y & d_z \end{bmatrix} = \begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x'_1 & y'_1 & z'_1 \\ x'_2 & y'_2 & z'_2 \end{bmatrix}$$

$$[x \quad y \quad z] = [u^3 \quad u^2 \quad u \quad 1]\begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}\begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x'_1 & y'_1 & z'_1 \\ x'_2 & y'_2 & z'_2 \end{bmatrix}$$

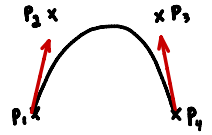
point = parametric vector x basis matrix x control matrix

$$p(u) = \begin{bmatrix} 2u^3 - 3u^2 + 1 \\ -2u^3 + 3u^2 \\ u^3 - 2u^2 + u \\ u^3 - u^2 \end{bmatrix}^T \begin{bmatrix} p_1 \\ p_2 \\ p'_1 \\ p'_2 \end{bmatrix}$$

• multiply the parameter vector and basis matrix results in four basis functions
• every cubic hermite spline is a linear combination of these 4 functions

Bezier Splines

• Specify four control points, end points are interpolated
• tagents are computed from the control points
Convex hull property: curve contained inside convex hull of control points



• points P_1, P_4 are endpoints — p(0) = P_1 & p(1) = P_4
• points P_2 , P_3 are off the curve
• p(0) = 3(P_2- P_1)
• p(1) = 3(P_4- P_3)

hermite basis | bezier to hermite | bezier control matrix

$$[x \; y \; z] = [u^3 \; u^2 \; u \; 1]\begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ -3 & 3 & 0 & 0 \\ 0 & 0 & -3 & 3 \end{bmatrix}\begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \\ x_4 & y_4 & z_4 \end{bmatrix}$$

$$= [u^3 \; u^2 \; u \; 1]\begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}\begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \\ x_4 & y_4 & z_4 \end{bmatrix}$$

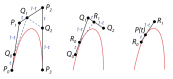
bezier basis

$$p(u) = \begin{bmatrix} (1-u)^3 \\ 3u(1-u)^2 \\ 3u^2(1-u) \\ u^3 \end{bmatrix}^T \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{bmatrix}$$

• scale factor 3 is chosen to make velocity approximately constant

DeCasteljau Construction

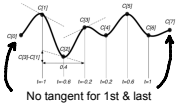
Efficient algorithm to evaluate Bezier splines



$$p(x) = a_0 + x(a_1 + x(a_2 + \ldots + x(a_{n-1} + a_n x)))$$

Catmull-Rom Splines

Hermite, Bezier: designer must arrange for tangents to be co-linear to get C continuity



For n control points in 3D, the tangent p_i is set to: $s \cdot (p_{i+1} - p_{i-1})$ *for* $i = 2 \ldots n-1$
s is the tension parameter: determines the magnitude of tangent
• use extra control points for endpoint tangents
• After computing positions & tangents = Hermite

Spline with More Continuity

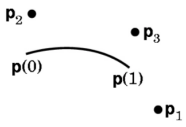
• use higher degree polynomial: more expensive, wiggly
• give up local control: natural cubic splines
• give up interpolation: cubic b-splines (use approximation)

Natural Cubic Splines

• for 2nd derivatives to match, find weight (tridiagonal linear system)

B-splines

• curve passes near the control points
• best with interactive placement



$$[x \; y \; z] = [u^3 \; u^2 \; u \; 1]\frac{1}{6}\begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}\begin{bmatrix} P_{i-3} \\ P_{i-2} \\ P_{i-1} \\ P_i \end{bmatrix}$$

How to Draw Splines

Method 1: brute force

• calculate the coefficients
• for each cubic segment, vary u from 0 to 1 (fixed step size)
• plug in u value, matrix multiply to compute position on curve
• draw line segment from last position to current position

Problems:
• draws in even steps of u — even steps of u is not even steps of x
• line length will vary over the curve, want to bound line length
  too long — curve looks jagged, too short — curve is slow to draw

Method 2: recursive subdivision

```
Data: u0, u1, maxLineLength
Result: calculates points on spline and draws
line between consecutive pairs
umid = (u0 + u1) / 2
x0 = f(u0)
x1 = f(u1)
if |x0 - x1| > maxLineLength then
  Subdivide(u0, umid, maxLineLength)
  Subdivide(umid, u1, maxLineLength)
else
  drawLine(x0, x1)
end
```

type	local control	continuity	interpolation
Hermite	✓	C^1	✓
Bezier	✓	C^1	✓
Catmull-Rom	✓	C^1	✓
Natural	×	C^2	✓
B-splines	✓	C^2	×