**Computer Graphics Basics**

**Definitions**

Clipping: Removing what isn’t in the scene

Raster Display: Display of pixels instead of vectors

Rasterization: Converting images into colored pixels

Ray Tracing: Way of rendering, math heavy

Orthographic Projection: Represents 3D objects in 2D

Perspective View: 3D image that portrays height, width, and depth

Transformation: Scale, rotate, and position objects

Vertex Shader: Figures where each vertex appears on the screen

What is an Image / Video?

Objects in the world (static or dynamic)

Illumination (light sources)

Imaging device (eye, camera)

Modeling

How do we model (mathematically represent) objects?

How do we construct models of specific objects?

Use Primitives, 3D points, 3D lines and curves, Volumetric representations, Image-based representations

Attributes include color, texture maps, and lighting properties

Animation

How do we represent the motions of objects?

How do we give animators control of this motion?

Rendering

Drawing visible surfaces onto the display, in terms of triangles

Simulating light propagation: Reflection, Absorption, Scattering, Emission, Interference

Subsurface Scattering: Light penetration into objects glow

How do we simulate the real-world behavior of light?

How do we simulate the formation of images?

Interaction

How do we enable humans and computers to interact?

How do we design human-computer interfaces

OpenGL Pipelines: Vertices => Geometry Shader => Vertex Shader => Fragment Shader => Framebuffer => Pixels on Screen

* New method allows programming 3-4th steps
* Geometric Primitives (triangles) => Vertex Shader (location of each vertex on display) => Fragment Shader (Colors the object, called once per pixel) => Framebuffer (array of pixels)

Graphics Pipeline: Modeling => Illumination => Viewing (Projection) => Clipping => Visibility => Rasterization

**Linear Algebra**

**Definitions**

Affine Combination: linear combination where a1 + a2 + … + an= 1

Affine Transformation: Has a constant in the matrix, add a row of 0,0,1 to allow matrix multiplication [Qx Qy 1] = M[Px Py 1]

Aperture: Hole allows light in, but unideal aperture creates blur

Basis: A set of vectors that can generate all of Rn

Convex Combination: Affine combination where ai ≥ 0

Coordinate Systems: An origin and some basis vectors

Cross Product: Use determinate, |a × b| = a b sinθ

Depth of Field: Lens tend to lose depth of field, good cameras do

Dot Product: a · b = a b cosθ, 90° = 0

Euler Angle: Rotate about all 3 axis in order

Gimble Lock: Euler rotation problem when axis align, bad moving

Inverse: AA-1 = I. (AB)-1 = B-1A-1.

Lens: Bends the light ray to create correct image, undo aperture blur

Linear Combination: w = a1v1 + a2v2 + … + anvn

Linear Independence: linear combination is 0 only if all ai = 0

Matrix: A rectangular grid of scalars

Orthonormal: Inverse is transpose, right angles, columns unit length

Pinhole: Images formed through pinholes (small ideal) are reversed

Projection: Transformation from higher to lower dimension

Ray Tracing: Computer intersections of rays and objects, slower

Rigid Body Transformation: Translations and rotations

Shearing: Qx = Px + aPy; Qy = Py; Sheers wrt. x

Standard Unit Vectors: (1,0,0) (0,1,0) etc. Linear combination

Symmetric Matrix: (Aij) = (Aji)

Transformation: Applying a matrix to a vector to get new vectors

Translation: Q = P + t, Not linear transformation because not mult.

Transpose: AT = (aij)T = (aji)

Vectors: Tuples with length √Σxi2

Z-Buffer: Polygon Based and Fast method of drawing objects

* Scalar (a), Vector (**a**), Matrix (**A**)
* Matrix Multiplication: A(B + C) = AB + AC, (AB)T = BTAT, MM-1 = I, (AB)-1 = B-1A-1
* Represents vectors and points, [x1, x2, x3, 0] vector, [p1, p2, p3, 1] point
* Point - Point = Vector, Point = Vector + Point
* Affine combination of points, average of points is a point because the bottom component of the vector averages to 1
* Ex: (1 - s)P1 + sP2 // s determines how much of P1 and P2 you use, like a slider on the line connecting two points
* Ex: (1 - s - t)P1 + sP2 + tP3 // covers an entire plane
* Lines: Explicit form y = mx + b, Implicit form f(x, y) = (x - x0)dy - (y - y0)dx = 0 (on the line), Parametric x = x0 + t(x1 - x0) and y = y + t(y1 - y0)
* Line Vector Notation: P(t) = P0 + t**v**
* Planes: Explicit z = ax + by + c, Implicit F(x,y,z) = ax + by + cz + d = n · P + d, Parametric P(s,t) = P0 + s(P1 - P0) + t(P1 - P0)
* Plane Vector: P(s,t) = P0 + sP1 + tP2
* Raytracing: Pass ray from eye to objects geometrically and cast rays recursively (reflected rays, refracted rays, shadow rays)
* Raytracing: Trace light rays and calculate colors
* Z-Buffer: For each polygon, project it onto the plane you see, calculate pixel colors and z-values (dist) based on projection
* If z value is less than previous z value (object is closer), override the pixel color value
* Z-Buffer: Modeling -> View Selection -> Perspective -> …
* Coordinate Systems: Object -> World -> View -> … -> Display
* Rotation, Scale, Translation, Sheer make all affine combination
* Inverse translation is just translate the negative, inverse of scale is just scale 1/s, inverse of shear is shearing by negative shear constant, inverse rotation negative upper diagonal
* Rotate Arbitrary Axis: Roll to align axis with x-axis, then rotate and un-align back to the original axis
* Affine Transformations: Composed of elementary transformations, preservation of point combinations, lines, planes, parallelism, and relative ratios
* Transform coordinate systems is also transformations
* Each transformation happens with respect to the previous coord
* Coordinate System: Modeled by an affine matrix with all 0 columns being basis vector and 1 column being the origin

**Objects**

**Definitions**

Camera: An object that has the transformations you see in the world

Far Plane: End of view volume, don’t process objects past far plane

Homogeneous Matrix: Add fourth dimension, 0 vector 1 point

Indexed Face Set: Define vertices then map sets of vertices to a face

Lookat Matrix: Transformation of camera to control what it looks at

Near Plane: The plane which perspective projects on

Non-Coplanar: polygons that are not flat

Normal: Direction of shapes, normal to surface of plane

Normalized Display CS: Make the view volume the canonical coordinate system, left hand coordinate system 2 \* 2 \* 2 cube

N-Gon: A polygon made of more than 4 edges, made of triangles

Orthographic: Parallel projection where lines 90 degree from plane

Parallel Projection: Orthographic and Oblique projections, center of projection is at infinity

Perspective Foreshortening: See difference in size depending on dist

Perspective Projections: 1-3 points act as anchors for the artwork, there are centers of projection where lines converge

Planar Projection: Projecting objects onto a plane

Polygons: Collections of points connected with lines

Primitives: Torus, Sphere, Cylinder, etc. Use to create more objects

Subdivision Surface: Add more polygons to increase smoothness

Triangle: Most simple polygon is simple, planar, etc.

Vanishing Point: Objects decrease in size and speed until vanish

View Volume: Only display geometry in a certain cuboid

* Polygon: Simpleness crossing edges, convex and concave,
* Plane Equation: ax+by+cz = n · p = points on a plane
* Vertices have color and texture coordinates
* Per vertex or per face, has color and normals
* Objects are polygons that share a scale, rotation, and translation
* Parenting allows objects to inherit scale, rot, trans information
* Parent all parts under a null object to keep object uniform
* Opposite of rotation matrix is transpose of rotation part
* Modeling and camera setup is all an Affine transformation
* Camera is just a change of basis
* Model View transformation preserves lines, parallelism, ratios
* Camera starts out with world coordinates, x, y, z out of screen
* Lookat Matrix [i, j, k, Peye = [Px, Py, Pk, 1]]
* Perspective projections are the most realistic, converge at eyes
* Wider the lense, the more perspective things look -> fisheye
* Orthographic projections are straight simple translation
* Perspective P’ on z = -N plane. Py’ = PyN / -Pz | “x” | Pz = -N
* Perspective projections are not linear transformations
* Perspective Matrix [[1,0,0,0],[0,1,0,0],[0,0,1,-1/N],[0,0,0,0]] \* [Px,Py,Pz,1] / (-Pz / N) = [P’x, P’y, P’z, 1]
* Linear transformation but not affine (bottom row 0,0,0,1)
* Hardware made to multiply matrices so matrix form still useful
* Undefined for Pz = 0, if P behind eye Pz sign flips, N scales img
* Lines remain lines but as line extends to infinity, speed -> 0
* Anything not parallel to image plane vanishes at infinity
* View volume goes from near plane (view plane) to far plane
* Perspective NDCS preserves depth but converts x-y to ortho

Graphics Pipeline: Model all the objects -> Set up a virtual camera for viewing -> Projection to canonical coordinate system

PView = MCam-1MModPObj // Transforming between coordinate systems

PWorld = MCamPView -> PView = MCam-1PWorld // PWorld = MModPObj

Go from view coords to world coords, first rotate then translate

Default Camera Matrix -> Peye where camera should be, Pref where camera is looking, Vup up direction: Direction look Pref - Peye

Projection: F: Rn -> Rm where n > m, flatten 3D to 2D

ZBuffer Graphics Pipeline: [Obj Coords] -> Modeling -> [World coords] -> Viewing -> [View coords] -> Projection -> [Cannon coords] -> Clipping -> Perspective division -> [Normalized display coords] -> Viewport -> [Display coords] -> Rasterization -> Image

Perspective NDCS: [[n000][0n00][00,n+f,-1][00,nf,0]]

MProj: VCS -> NDCS: MOMP

MViewport: Undo distortion from projection, map to pixel coordinates

MVP: Translate by the midpoint and scale by the midpoint

Z coord unchanged, x & y scaled

**Rasterization**

Bresenham Midpoint Algorithm: Incremental rasterization method

Flood Fill: Expand every direction from each colored pixel

Rasterization: Produces lines and triangles on image, fill in pixels based on overlap with ideal line based on x y calculations

Scanline fill: Iterative flood fill, more efficient than recursive way

Wireframe: Lines and vectors of model

Wireframe Rendering Algorithm: Rasterize all the vectors

* Lines designed to look as straight as possible
* Optimize algorithm efficiency with midpoint algorithm
* Bresenham’s algorithm also works for more complex primitives such as circles and polynomials
* Flood fill converts the color of the pixel if the pixel is the color of the old color (stops upon reaching a different color)
* 4 neighbor flood fill stops at diagonal bounds, 8 neighbor leaks
* Polygon rasterization: Use scan lines to determine what’s in/out of polygon and fill everything inside of the polygon

Bresenham algorithm: Uses implicit form of line

Given 0 < slope < 45, Current pixel P(x,y), next pixel P’(x+1,y’)

If F(x+1,y+0.5) > 0, choose NE pixel, else choose E pixel

Bring out all constants for full optimization

**Texture Mapping**

Bump Mapping: Texture that has shadows, looks 3D but not

Displacement Mapping: The geometry of object actually changed

Interpolation: Inject view coords onto the ST/UV coordinates

ST/UV Coords: 2D Coordinate system of the texture image

Texture Mapping: Putting texture images onto a 3d object

Unwrapping: Flatten the object on top of the UV canvas and project

UV Canvas: Displays U and V respective to X and Y, texture map

* Map the image to the vertices on the object, draw picture on polygon instead of just filling in a color
* Easier to check rendered image and see where that pixel comes from on the texture image, have to invert the matrix
* Ex: Map from square to cylinder, use cylindrical coordinates, x = rcos(θ), y = rsin(θ), z = z
* Cylinders would be wrapped using vertical rectangular strips
* Spheres are very difficult to unwrap, difficult to make into 2D
* Interpolation is not linear, it’s inverse of perspective mapping
* Displacement mapping needs more geometry, both disp and bump use grayscale mapping (black less obv white more obv)

Texture Coordinates -> 3D World -> Rendered image space

Only the vertices matter in terms of the graphics pipeline

**Visibility**

Clipping: Cut a polygon that intersects with the view volume

Culling: Eliminate entire polygon from pipeline if outside view

Object occlusion: Another object is blocking some image

Scanline: Scans across depth, Z-Buffer uses this technique

Z-Buffer: Keeps depth information about each pixel, most common

* Invisible to outside of view volume
* Invisible due to self/object occlusion (blocked from view)
* Backface culling: normals facing away from view culled
* If projection orthographic, use Z component to cull faces
* Perspective: plug in camera point into infinite plane of polygon implicit form, if on normal side is visible

Clip to remove objects outside of view volume, VCS, CCS, NDCS

VCS: Find intersections with 6 planes that make up view volume

CCS: Simpler and still efficient due to orthogonality of CCS

Plug in parametric line eq into implicit plane eq, find intersection

NDCS: Easy to do backface culling because everything orthogonal

Image-space alg: Z-buffer, Watkin’s algorithm, ray-tracing

Object-space alg: Binary-space partitions, Painter’s algorithm, O(N2) worst case

Z-Buffer: keeps RGB info, Depth info, and blending info, it’s fast but memory intensive, handles polygon intersect but has jaggies

A-Buffer: Able to use transparency effects, linked list of surfaces

BSP: Process all objects into a tree, build image by traversing tree

Object-space alg, must find ways to organize depth of polygons

Ray Tracing: Project line from camera through pixel in plane, cast off rays recursively (need really good intersection algorithms)

**Light and Shade**

Area Light (light source that’s not a point, line a plane, or a window): Light comes in an area, all in one direction

Blinn-Torrance Model: Uses halfway vector between light & view

Diffuse: Reflection is distributed in all directions, bad as mirror

Gouraud Shading: Associate colours with vertices and then mix

Lambertian: Diffuse reflection, matte surfaces such as chalk

Phong Model: I = ILKd(n · L) + ILKd (r · v)n

Point Light: Light is emitted equally in every direction (isotropic)

Reflection: Angle in equals angle out

Refraction: Snell's law, nasin(θ1) equals nbsin(θ2)

Specular: Perfect mirror, light is all reflected in the correct angle

Specularities: Glossy highlights of light on specular surfaces

* Ray tracing takes care of both reflection and refraction
* Calculate each color (RGB) separate and add together at end
* Phong Model combines diffuse and specular equations
* L is light out, R = 2N(N · L) -L
* Phong and Blinn-Torrance missing ambient glow
* Add constant light source for ambient glow, additive light
* Shiny-ness / Mirrors are White: Reflect all of the color
* Gouraud shading might miss some details not on vertices
* Problems: Orientation dependence, Silhouettes showing up, Handling T-Vertices, generating the vertex normals

Light bounces off surface and the bouncing determines look

Light hits a point, bounces in many directions, some enters eye/cam

I = ILKdcos(θ) I = ILKd(n-L) IL = incoming intensity Kd = surface reflectance coefficient

Ray Tracing

Shadow Ray (if you have a ray from the object to the light, there’s nothing between, then there’s no shadow): Ray from object to light source, see if shadow exists

* Allows reflections and transparent object, bending light rays
* Best for specular / transparent objects, based on physics optics
* More efficient to trace from eye to light sources
* More sophisticated systems trace eye to light and light to eye
* More recursion looks better but expensive
* Use parametric form for lines for ray tracing, most convenient

Find light from light source, find reflected light, find refracted light

Add together using phong model, Ambient + Diffuse + Specular

Set Camera / Image Plane ⇒ Compute Ray per Pixel ⇒

Shadow, reflected, and refracted arrays that come from the camera.

Computer-Generated Animation

Behavioral Animation: How animals or humans behave in reality

Dynamics (when have masses in time): Factors in movement, gravity, mass, inertia, etc.

Inverse Kinematics (finding equations from start and finish of what wanting to do): Calculate angle from position, non-linear

Key Framing (sequence of how you define what’s happening, need to just write the most important parts and computer works rest): Specify parameters at specific times, important parts

Kinematics (physical kinematic equations): Constrained movements regardless of math, force, etc

Motion Capture: Strap sensors onto actors, capture the movement

Physical Simulation: Useful for clothes, simulate physics for objects

Procedural Animation (to draw a person, draw parts of body and reuse it): The animator is a programmer, give params

Spline: Smooth a line, useful for smooth motion between frames

Linear subdivision: add more control points/vertices, makes it smoother because there are more

Linear division move the vertices around.

Linear sub-division of curves: splitting each curve in the middle with another vertices. Making it smoother.

Computer may interpolate between key frames, make it continuous

3D Model ⇒ Vary parameters (vary how it moves) ⇒ Render 130K frames (how many frames a movie needs)

Inverse dynamics: You want a certain result such as a hole-in-one or 3 pointer in basketball, and calculate the initial hit afterwards