CS 351 Computer Graphics, Spring 2017

Dr. Bruce A. Maxwell
Department of Computer Science
Colby College

Course Description

Computer graphics deals with the manipulation and creation of digital imagery. We cover drawing algorithms for two-dimensional graphics primitives, 2D and three-dimensional matrix transformations, projective geometry, 2D and 3D model representations, clipping, hidden surface removal, rendering, hierarchical modeling, shading and lighting models, shadow generation, special effects, fractals and chaotic systems, and animation techniques. Labs will focus on the implementation of a 3D hierarchical modeling system that incorporates realistic lighting models and fast hidden surface removal.

Prerequisites: CS 251 or permission of instructor. Linear algebra recommended.

Desired Course Outcomes

A. Students understand and can implement the fundamental concepts of creating and manipulating images.

B. Students understand and can implement the fundamental concepts of rendering, including matrix transformations, shading, and hidden surface removal.

C. Students understand and can implement the fundamental concepts of modeling objects and scenes and hierarchical modeling systems.

D. Students work in a group to design and develop 3D modeling and rendering software.

E. Students present methods, algorithms, results, and designs in an organized and competently written manner.

F. Students write, organize and manage a large software project.

1 Graphics

If you could make a picture, what would it be?

Graphics is:

- Vector graphics: **rendering** images using points and lines
- 2-D shapes and shape manipulation: presentation software
- Drawing programs: from simple to advanced applications such as Photoshop
- Graphical User Interfaces [GUI] and computer windowing interfaces
- Generating, or **rendering** pictures of 3-D scenes from models
- **Animation** of models
- **Modeling** of physical or virtual phenomena
- Computer-Aided Design [CAD] or Computer-Aided Modeling [CAM]
- Training and simulation, such as flight simulators
- Virtual reality and immersive 3D environments
- Visualization of data, such as medical imaging or web networks
- Games, both 2D and 3D

What is an image?

- Vector representation
- Raster representation
- Spectral representations

What is a color?

- Color is a perception
- The perception is caused by EM radiation hitting sensors in our eyes
- Different spectra generally cause different perceptions of color
- Color is perceived differently depending upon context

How do we perceive color?

- Rods
- Cones
- Context
- Anchoring

How do we generate colors with a computer?

- Cathode Ray Tube (CRT): these use electron guns to activate different kinds of phosphors arranged in an array on a screen. There are generally three different colors of phosphor (red, green, and blue) and they mix together to generate different spectra.
- Liquid Crystal Display (LCD): these use small liquid crystal gates to let through (or reflect) differing amounts and colors of light. Usually, the gates are letting light pass through what are effectively R, G, or B painted glass. LCD displays are currently the most common types of displays in use.
- Plasma Display (PDP): these use very small R, G, B fluorescent lamps at each pixel. The lamps are filled with a noble gas and a trace amount of mercury. When a voltage is applied to the lamp, the mercury emits a UV photon that strikes a phosphorescent material that emits visible light. Plasma colors match CRT colors, because both use the same phosphorescent materials.
- Organic Light Emitting Diodes [OLED]: these are materials that emit photos of varying wavelengths when a voltage is applied in the correct manner. Like plasma displays, OLEDs are an emissive technology that uses collections of small OLEDs to generate a spectrum of colors.

How do we represent colors with a computer?

- The internal representation of colors matches the display (and viewing) mechanisms: mix three intensity values, one each for red, green, and blue.
- The RGB system is additive: with all three colors at maximum intensity you get white.
- There are many other color spaces we can use to represent color
  - HSI: hue, saturation, intensity
  - YIQ: intensity (Y) and two color channels (in-phase and quadrature), the original US broadcast TV standard
  - YUV: intensity (Y) and two color channels (UV), used by PAL, the original European broadcast standard
- We need to use a certain number of bits per color channel to generate different colors. Common formats include: 1, 5/6, 8, and 16 bits/pixel. High end cameras now capture in up to 14 bits/pixel.
- Internally, we will likely be using floating point data to represent pixels.

How do we represent pictures as files?

- Somehow we have to store the information in the image, which may include colors, but may also include shapes, text, or other geometric entities.
- Raster image format: each pixel gets a value
  - TIFF: Tagged Image File Format, non-lossy, any data depth is possible
  - GIF: Graphics Interchange Format, lossy, at most 256 colors, supports animation
  - JPG: long name, lossy method (can't recreate the original data), compacts images well
  - PNG: Portable Network Graphics, non-lossy compression, retains original data
  - PPM: Portable Pixel Map, very simple image representation with no compression
  - RLE: Run-Length Encoded, simple, non-lossy (but not very good) compression
- Vector image representations: images are collections of objects with geometric definitions
  - SVG: Scalable Vector Graphics, created for the web, based on XML
  - GXL: Graphics eXchange Language, also created for the web and based XML
- Element list representations: images are collections of objects and pictures
  - PICT
  - PDF
- Graphical language representations: Postscript

2 Coding

Code Organization

- source files: code, global variables, local macros, includes
- header files: prototypes, typdefs, externs, general includes, global macros
- libraries: a set of precompiled functions
- object files: intermediate compilation products, enable partial recompilation
- executables: executable code

Data Types

- char/unsigned char
- short / unsigned short
- int / unsigned int
- long / unsigned long
- float
- double

Arrays and strings: strcpy, strcat, strncpy, strncat, memcpy, memset

Typedefs: creating new types, e.g. an Image struct

Header Files

- typedef statements
- #define statements
- function prototypes
- extern statements
- useful include files: stdio, stdlib, math, string

3 Graphics as Canvas / Paints / Tools

Memory

- Pointers are variables that hold typed addresses
- C does not initialize memory
- Allocating and freeing a simple array
- Allocating a 1-D image
- Accessing a 1-D image
- Allocating a 2-D images
- Accessing a 2-D image

Makefiles: remind them about the makefile tutorial

4 Image API

What does an Image API look like in C?

- Passing around data structures: use pointers
- Using both structures and pointers
- Macros v. functions v. inline

Differences between image_create / image_free and image_init / image_alloc / image_dealloc

Accessing pixels: safe v. unsafe, macros, and inline

5 The Art of Graphics

Canvas/Paint/Brushes

- Canvas: display v. reflection
- Canvas: raster v. vector
- Canvas: aliasing, resolution, and refresh
- Paint: Color spaces, RGB, HSV/HSI, CIE-Lab, CIE-Luv
- Paint: spectral aliasing
- Brushes: input devices, paint tools, pixels, primitives
- Brushes: 3D models, procedural models
- Brushes: animation, keypoints, joints, procedures, planning
- Brushes: NPR

6 The Graphics API / Lines

Graphics primitives

- what is a pixel?
- what information do we need at a pixel level?

State v. parameter systems

- state based: most drawing attributes are stored in a global data structure (e.g. turtle, OpenGL)
- parameter based: most drawing attributes are passed as arguments
- mixed system: drawing attributes are stored in a data structure passed as a parameter

C does not allow polymorphic functions; every function must have a unique name.

Lines

How would you draw a line?

How would you represent a line?

Bresenham's line algorithm, presented as a test point, error term, and error update

7 Lines, Circles, Ellipses, Polygons, Polylines

Moving Bresenham's Line algorithm to lower left corner screen coordinates

- Change the initial test to $3\Delta y - 2\Delta x$
- Write out the algorithm for the first octant: need reflections around each octant
- How do we draw horizontal and vertical lines?
- Making Bresenham's fast: from either end, move in steps of 2

Bresenham's Circle algorithm

- Note the process of creating a fast algorithm
- Figure out where you are testing and the crux of the test
- Figure out the incremental change in the error term
- Use symmetry wherever possible to speed up the algorithm and make it simpler
- Identify how to fill a circle by rows

Coordinate system issues: just start the algorithm in the 3rd quadrant and use the updated mirroring equations

8 Polygons / Polygon Fill / Textures

Comment: pow, sqrt, and other habits die hard, kill them

Bresenham's Ellipse algorithm

- Similar to circle algorithm, but you need to switch to moving in Y when the slope hits -1
- Two loops in the ellipse algorithm
- Only 4-way symmetry
- Fill the same way

How would you fill with a pattern?

How would you fill with an image?

How would you deal with oriented ellipses?

- Use the radial version of the ellipse equation and draw short lines
- Circle: $x = x_c + r\cos(\theta) \quad y = y_c + r\sin(\theta)$
- Treat it as a polygon, but adapt the number of vertices to the size of the ellipse
- Ellipse: $x = x_c + a\cos(t) \quad y = y_c + b\sin(t)$, where t is the eccentric anomaly
- Step in uniform steps by t to get vertex points which will be denser where it changes faster

Polyline data structure and API

- Note the similarities with the Image API
- Note the dynamic nature of polylines and polygons

What have we learned by drawing simple shapes?

- Aliasing
- Coordinate system issues
- Integerizing algorithms
- Developing incremental algorithms that take advantage of coherence and symmetry

Generic Fill Algorithms

- Flood fill
- Scanline flood fill

Concept of connectedness

Generic Fill Algorithms

- Flood fill: recursive, sort of
- Scanline flood fill: takes advantage of coherence

9 Polygon Scanline Fill

A final note on generic fills: connectedness

Filing with various inputs

- Pattern fill: review
- Gradient fill: blend between two colors given some metric associated with the shape
- Transparency: alpha blending, can use functions to control transparency
- Texture fill: mapping from (u, v) to (x, y)
- Texture fill: interpolation when expanding
- Texture fill: box filtering when shrinking

Scanline fill: Overview of the algorithm

1. Generate the edge list
2. Loop over each scanline
   (a) Insert/delete edges from the active edge list
   (b) Fill the scanline left-to-right by edge pairs
   (c) Update the intersection points for each active edge

   Go over the simple polygon example.

   Go over the rules for integer

   Go over the rules for continuous vertices

   - Scanlines go through the middle of each pixel
   - Show the diagram and an edge meeting up with the next scanline
   - The starting row is the pixel within which the edge first crosses a scanline

     How do you round numbers in C? `x = (int)(v + 0.5);`
   - The ending row is the last pixel within which the edge crosses a scanline
   - The starting location of the edge is the first intersection of the edge with a scanline

10 Scanline Fill Details / Barycentric Fill

Scanlne Fill: setting up the edges

- All edges go from the lower edge to the top
- Calculating dxPerScan: calculate it exactly given the endpoints
- Calculating ystart: first scanline the edge crosses, so round
- Calculating yend: last scanline the edge crosses, so round - 1 to stop early, or rows-1
- Calculating xIntersect: adjust to the first scanline it crosses
- Adjust xIntersect if it starts below the image (-edge-¿y0 * dxPerScan)
- Adjust xIntersect if it is beyond the end of the edge: set to the end of the edge

Barycentric fill algorithm

- basic concept: set up a half-plane and coordinate system for each side of the triangle
- Test if the point is on the correct side of each plane and the distance is between 0, 1

Dealing with the case where the pixel is exactly on the border.

- Drawing all border pixels: overlapping polygons
- Drawing no border pixels: gaps
- Rounding to the left (or right) in screen space

11 Anti-aliasing

What is aliasing?

What kinds of aliasing exist?

Nyquist criterion

Anti-aliasing

Supersampling: render on a big image and then scale it down.

Area sampling: render thick lines on a big image and scale it down

Using error terms for anti-aliasing.

Post-filtering: in particular, how do you deal with non integer scaling?

12 Transformations / Viewing

Matrix representations: 2D transforms as 4x4 matrices

- Think about your rotation matrix functions: take in cos and sin, not theta

2-D Viewing pipeline

- Model coordinates
- World coordinates
- Viewing coordinates
- Normalized viewing coordinates
- Screen coordinates

$VTM = T(\frac{cols}{2}, \frac{rows}{2})S(\frac{1}{du}, \frac{1}{dv})R_z(n_x, -n_y)T(-c_x, -c_y)$

Defining the orientation of the viewing rectangle with a normal vector

Orienting coordinate systems with orthonormal bases in 2D

3D transformations

Matrix-Matrix and Matrix-Vector multiplication

3D Viewing Projections

- Parallel v. Perspective
- Orthographic
- Axonometric: orient the object so a certain number of faces are visible
- Oblique: modify the projection vector to be oblique to the viewing plane
- Arbitrary parallel project: done in CS 251

Perspective Projection

Definition of the Perspective Viewing setup

13 View Transformations

Parallel Viewing

- VRP
- VPN, VUP, U
- DOP: Direction of projection, defined in VRC
- du, dv: window could be off-center, in which case $(u_0, v_0), (u_1, v_1)$ defines the window
- F, B (front, back clip plane distances)

$$\text{VTM}_{\text{World}\rightarrow\text{CVV}} = S(\frac{2}{du}, \frac{2}{dv}, \frac{1}{B'})T(-\frac{u_0+u_1}{2}, -\frac{v_0+v_1}{2}, -F)Sh_z(\frac{D_x}{D_z}, \frac{D_y}{D_z})R_{xyz}(\hat{U}, \hat{\text{VUP}}, \hat{\text{VPN}})T(-\text{VRP}) \quad (1)$$

Requires scaling and translation to screen coordinates: $\text{VTM}_{\text{CVV}\rightarrow\text{Screen}} = T(\frac{C}{2}, \frac{R}{2})S(-\frac{C}{2}, -\frac{R}{2})P_{\text{orth}}$

Review of perspective view setup

- VRP
- VPN
- VUP
- du, dv: assume the view window is centered
- d: projection distance, defines center of projection (COP) on negative VPN
- B, F: back/Front clip plane distance

Perspective Projection Matrix and perspective math

Perspective Viewing Pipeline

1. Build a coordinate system from VPN, VUP.
2. Translate VRP to the origin. $T(-vpn_x, -vpn_y, -vpn_z)$
3. Align the axes. $R_{xyz}(\hat{vpn}, \hat{vup}, \hat{u})$
4. Translate the COP to the origin: COP is a distance $-d$ from the VRP along the negative VPN. $T(0, 0, d)$
5. Update the distance to the back plane. $B' = d + B$
6. Scale the view volume to the canonical pyramid. $S(\frac{2d}{B'du}, \frac{2d}{B'dv}, \frac{1}{B'})$
7. Calculate the new focal length/projection distance. $d' = d/B'$
8. Project the scene onto the view plane using the projective matrix. $P(d')$
9. Scale the image to the screen and translate the origin to the upper left. The size of the view plane is $2d \times 2d$, and the origin is in the center of the view window. $T(C/2, R/2)S(-C/2d, -R/2d)$

14 No Class

15 Exam

16 Hierarchical Modeling

Review setting up a view

Modeling concepts

- Coordinate systems: model, world
- Want to be able to create objects out of primitives
- Want to be able to create objects out of other objects

Objects as lists of primitives and transformations

Scenes as directed acyclic graphs

Go over the robot arm graph

- One way to design is from the leaf to the root: start with the hand in model coordinates
- Another way to design is from the top down: logical modular subdivision of the scene
- The code has to be written bottom-up because modules must be defined before they can be referenced

What capabilities are necessary to build a scene graph?

- Begin and end modules
- Add graphics primitives to modules
- Add transformations to modules
- Add drawing attribute state changes to modules

Design considerations

- Data structure: linked list is efficient: only adding to the end, only traversing the graph
- Handling the module data structures: accessible or not (e.g. OpenGL)

C details: union data structure

17 Hierarchical Modeling / Line Clipping

Scene graph traversal

Interpreter to parse a scene graph: module_draw

- Parameters: module, VTM, GTM, DrawState, Lighting, Image
- Interpreter has to maintain local state: LTM, local DS
- Interpreter needs to be able to recursively parse models
- Interpreter needs to maintain the current transformation LTM
- As it descends into sub-modules, the current transformation gets stored

Modeling: enterprise example

Lines and Line clipping

- Subdivision algorithm
- Check if the line is trivially visible/invisible
- If not, subdivide in half and recurse on each half

18 Clipping, Polygons, and other Shapes

Lines and Line clipping

- Parametric line representation: $P = L_0 + t(L_1 - L_0) = A + t\vec{V}$, $t \in [0, 1]$
- Liang-Barsky / Cyrus-Beck clipping: find the values of t such that the line is in the window
  - Compute p_i, q_i values based on V and comparing the boundaries to A
  - If any p_i is 0, the line is parallel to a boundary, so q_i determines if it is trivially invisible
  - The sign of p determines if the line is heading outside the boundary or inside
  - See notes for the full algorithm

19 Polygons, Parametric Surfaces, and Splines

Polygons

- Normal to the plane, Plane equation
- Dot products to VRP to test visibility of one-sided polygons

Polygon Lists

- Polygon fans, strips
- Polygon collections: representing polygon sets as polygons, edges, and vertices

Algebraic Surfaces: $0 = f(\vec{x})$

- Sphere, elllipsoid, torus, super-ellispoid
- Rendering: generate a triangle mesh that represents the surface, render the triangles

Subdivision Surfaces

- A wide variety of subdivision surfaces, concept is to have a goal shape and subdivide the triangles using a rule that approaches the goal shape.
- Sphere, ellipsoid, super-ellipsoide: start with eight triangles and subdivide each triangle into four
- Torus: make a square donut out of triangles and subdivide

Parametric Models

- Anything for the form $X = f(\vec{t})$. X is a point on the surface and $\vec{t}$ defines a coordinate system
- Line: $X = A + t\vec{V}$
- Circle: $P(\theta) = \left[\begin{array}{cc} \cos\theta & \sin\theta \end{array} \right]^T$
- Circle: $P(t) = \left[\begin{array}{cc} (1-t^2)/(1+t^2) & 2t/(1+t^2) \end{array} \right]^T$
- Rendering: generate a triangle mesh by calculating points in the parameter space for vertices

Splines

- parametric curves defined by control points
- interpolation v. approximation, convex hull
- possible constraints: control points, boundary conditions, tangent vectors

20 Splines

Example splines:

- Cubic splines: show as a set of equations, then as a matrix equation
- Natural cubic splines, each segment is a new equation set up by constraints
- Hermite splines, show how you set up the constraints and the matrix equation
- Cardinal splines, controlled by the control points plus some parameters

21 Bezier Curves and Surfaces

Bezier curves

- Each point on a Bezier curve is a weighted sum of the control points
- The Bernstein polynomials provide the basis functions for an Nth degree curve
- Approximating curves
- The curve is guaranteed to be inside the convex hull of the control points
- The curve goes through the first and last control points
- The tangent at the first and last control points is defined by the first two/last two control points
- Show the characteristic matrix for a Bezier curve

de Casteljau Algorithm

- Recursively subdivide each segment at the desired parameter
- Connect the new vertices and repeat the subdivision

Bezier surfaces

- Perpendicular sets of Bezier curves
- The (u, v) coordinate system defines a coordinate system on the surface
- The de Casteljau algorithm provides a simple method of dividing the surface into triangles
- The surface normal for a Bezier surface is the cross product of the tangent vectors in u and v

Potato chips

- Start with a triangle and subdivide, using random perturbations that get smaller

Hidden Surface Removal

Painter's algorithm

22 Hidden Surface Removal, Illumination

BSP Trees

- Pick a polygon, defines a boundary that splits the world of polygons in half
- Split any polygons bisected by the plane
- Repeat recursively for each side
- When building a BSP tree, it can be helpful to roughly sort polygons before choosing
- Usually build many BSP trees and minimize the number of split polygons/optimize balance

Traverse the tree from the point of view of the eye.

Z-buffer algorithm

- Basic concept: interpolate z in scanline fill
- Handling perspective issues: interpolate 1/z
- Testing v. 1/z
- Back plane / front plane

Create a BSP tree, then render back to front, no need to split the polygons since Z-buffer handles it

23 Illumination and Reflection

A-buffer algorithm

- Store linked lists of polygons at each pixel, not colors and z-values
- Enables transparency

Light Sources

- ambient light (review)
- directional light (review)
- point light sources (review)
- spot light sources
- area light sources

Modeling lights

- Make them part of a module
- Run a light pass through the hierarchy and calculate all light sources in CVV space
- Modify your drawing cases to calculate shading in module_draw after GTM and before the VTM

Overview of Modeling reflection

- ambient reflection
- body reflection
- specular reflection
- falloff term

24 Illumination and Reflection II

Key Reflection Values

- $\vec{N}$ Surface normal, local orientation of the surface
- $\vec{L}$ Light vector, points from the surface point towards the light source
- $\vec{V}$ View vector, points from the surface point towards the viewer (COP)
- $\vec{R}$ Reflection vector, view vector reflected around the surface normal: $\vec{R} = (2\vec{N} \cdot \vec{L})\vec{N} - \vec{L}$
- $\vec{H}$ Halfway vector, average of the view vector and the light vector: $\vec{H} = \frac{(\vec{L}+\vec{V})}{||\vec{L}+\vec{V}||} \approx (\vec{L} + \vec{V})/2$
- L_a Ambient light color
- L_d Direct light color
- C_b Body reflection color of the material
- C_s Surface reflection color of the material
- n roughness coefficient (bigger is shinier)

Modeling reflection

- ambient reflection: $I_a = L_a C_b$
- body reflection: $I_b = L_d C_b \cos\theta = L_d C_b (\vec{L} \cdot \vec{N})$
- specular reflection: $I_s = L_d C_s \cos^n \phi = L_d C_s (\vec{V} \cdot \vec{R})^n \approx L_d C_s (\vec{H} \cdot \vec{N})^n$
- falloff term

Inhomogeneous Dielectrics v. Metals

The modeling pipeline

- Make lights part of a module
- Run a separate light pass through the hierarchy and calculate all light sources in CVV space
- In module_draw, modify your drawing cases to calculate shading after $GTM \cdot LTM$ and before VTM
- Transform surface normals into CVV space, do shading calculations there, COP is at (0, 0, 0)

Shading

- Flat: where to compute the color and how to assign it, use average location of vertices, average surface normal, copy the color to all vertices, or just draw as a single-color polygon (no color interpolation)
- Gouraud: compute at vertices, interpolate colors
- Phong: compute at each pixel (or some variation), interpolate position and surface normal

Interpolating colors in perspective: represent what you want to interpolate in homogeneous coordinates, and interpolate those. So interpolate 1/R, 1/G, 1/B.

Interpolating surface normals and locations in perspective: same process, use homogeneous coordinates

Making Phong shading faster

25 Shadows

Review of rendering pipeline

- In module_draw transform polygon points and surface normals to CVV space (GTM * LTM)
- Calculate color at each vertex given light sources, $\vec{N}$, and $\vec{P}$
- Transform vertices to view space (VTM)
- Pass polygon and colors into scanline fill and interpolate colors (Gouraud Shading)
- Pass polygon and surface normals into scanline fill and interpolate positions and normals (Phong Shading)

Shadows

- Shadow polygons/volumes
- Ray casting

Go over uniform sampling / random sampling / Poisson sampling / jitter sampling

26 Texture

Texture concepts

- texture coordinates
- mapping vertices to textures using linear equations: two anchor points is enough to define the mapping

Z-buffer texture mapping

- Have to do Phong shading, since you need a color from the texture map at each pixel
- Go over mods to scanline fill
- Calculate texture value at each corner of the pixel

Exam Monday

27 Texture, Mip-Mapping

Mip-mapping

- Pre-generate texture maps at scales of 2
- Use the largest side of a bounding box $\Delta = \max(ds, dt)$ to determine scale $d = \log_2(\Delta)$
- Use a weighted average of the two closest levels

Bump mapping

- The surface normal is the result of a cross product of tangent vectors
- Perturb the surface in the direction of the surface normal
- Calculate the new tangent vectors of the perturbed point
- The end result is a function of the partial derivatives of the perturbation function

Mapping 2-D textures onto 3-D objects

- a common approach is to unwrap the object onto the texture map (e.g. cube example)

But how do you map a bumpy asteroid onto a plane?

Texture mapping 3D objects using 2-stage texture mapping

- Given a 3-D surface, pick a topologically similar surface with a simple mapping to a plane
- Map the target surface to the intermediate surface
- Use the mapping from the intermediate surface to the plane to identify the texture

Methods of mapping the target to the intermediate surface

1. Ray casting from the surface normal of the target surface to the intermediate surface
2. Ray cast from the intermediate surface onto the target surface
3. Ray cast from the center of the target surface through the target surface point
4. Ray cast from the reflected view direction off the target surface to the intermediate surface

Environment mapping (method 4)

- Identify the location of the object to be rendered in the scene
- Construct the intermediate surface (cube) around the object
- Render the scene onto the sides of the cube from the point of view of the object center
- Use the six rendered views as the texture map images for method 4

28 Particle Systems, Animation

The final exam period

- final exam will be a timed take-home
- final presentations during final exam time: 6pm Wed
- Last Starfighter after presentations, bring friends, popcorn

Solid texture mapping

- Generate a 3D texture: 3D array of voxels or a procedural texture
- Anchor the 3D texture to the shape
- Determine the volume of a pixel during rendering time to identify the voxels to be averaged

Particle systems are the main method of creating fire, explosions, smoke, and swarms

Basic concept

- emitter
- particles
- rules

A particle can be rendered any way you want and can be a substantial object or an ephemeral modifer

What can the rules control?

- How the particles leave the emitter
- Particle states at the time of creation
- Rate / one time burst / continuous stream / repeated bursts
- Motion rules
- Interaction with the environment
- Shape and appearance
- Color
- Lifetime
- Curves/functions for each control parameter

The Disney animation process

- Storyboard showing sketches of the key actors
- Master animators generate key actors and expressions
- Journeyman animators generate in-between images
- Other cels showing the background

29 Animation

Lasseter, 1987 SIGGRAPH paper on animation principles

Keyframe animation: how do you interpolate position/orientation?

Interpolating orientation using quaternions and spherical interpolation (SLERP)

- Introduced by Ken Shoemake

$$q = a + bi + cj + dk \tag{2}$$

- A quaternion is a 4-dimensional group, i, j, k are orthonormal unit vectors following the RH rule.
- Any arbitrary rotation by angle α around vector $\vec{u}$ can be expressed as a unit quaternion

$$R(\alpha, \vec{u}) = q = \cos\frac{\alpha}{2} + \vec{u}\sin\frac{\alpha}{2} \tag{3}$$

- Executing a rotation using a quaternion is: $\vec{v}_r = q\vec{v}_o q^{-1}$, where $q^{-1} = \cos\frac{\alpha}{2} - \vec{u}\sin\frac{\alpha}{2}$
- Rotations can be combined using the quaternion product: $q_{12} = q_1 q_2$, which is not commutative
- Spherical interpolation creates smooth motions between orientations

$$SLERP(p_0, p_1, t) = \frac{\sin((1-t)\Omega)}{\sin\Omega}p_0 + \frac{\sin(t\Omega)}{\sin\Omega}p_1 \tag{4}$$

Procedural animation

- physics engines: gravity, force
- differential equations for spring/mass/damper systems
- control theory for managing robots and other actors
- collisions
- bounding surfaces (spheres, boxes, cylinders with hemispherical ends, convex polygons in 2D)

Representational animation: e.g. cloth, trees, other objects with changeable internal states

Behavioral animation

- particle systems but with more intelligence
- state machines are the most common mechanism of control
- subsumption architectures
- multi-layer architectures with a planning system
- A* search for path planning
- LOTR battle scenes, Lion King stampede sequence

30 Ray Tracing

Main concept

- Send rays from your eye into the world and follow them as they bounce around the scene
- Primary task of a ray tracer is to do ray-object intersections
- At an intersection:
  - Transmission rays, if a semi-transparent material
  - Reflection rays, if the surface has any surface reflection
  - Shadow rays, to determine whether the point is in shadow from each light source
- When a ray hits nothing or its coefficient is small enough, traverse the ray tree to calculate shading
- Can easily use mathematical objects instead of polygons
- In general, need to sample a pixel with multiple rays: can be adaptive

Speedups

- Bounding surfaces for objects: e.g. spheres
- Hierarchical bounding surfaces for objects: hierarchies of spheres
- BSP tree for the polygons so you can search them in order
- Octree for polygons for the same reason
- Subdivide the polygons in ray space (originating position plus orientation)

Enhancements

- Depth of field: model lenses
- Motion blur: send rays into the DB at different times
- Body reflection rays: sample the reflection function with many rays
  - Don't calculate the diffuse signal at every location, store it when you do
  - Use an octree to store the diffuse values
  - Use nearby diffuse calculations to get the diffuse value
  - Don't use values that are in front, potentially on blocking surfaces
- Distributed ray tracing: go forwards from light sources, not backwards (crazy)
- Monte-Carlo Markov Chains: go from both directions and the modulate

31 Caustics

Global surface photon map (caustics and indirect lighting)

- Shoot photons from the light source into the scene, use projection maps
- When a photon hits a non-specular surface it is stored (and possibly transmitted)
- Store position, incident angle, color
- Collect photons *after* one or more reflections or transmissions in caustics map LS^+D
- Collect photons after a diffuse reflection in the indirect map $L(S|D)^+D$
- Use the photon maps to estimate radiance for non-direct illumination
- Estimate the size of the circle required to hold K photons
- Use a balanced k-d tree to hold the photons

Rendering

- At each pixel, calculate the direct illumination effects, then add the caustic and indirect effects
- Treat all polygons as blockers when computing shadows

Volume photon map

- Shoot photons from the light source into the medium
- A photon can pass unaffected or interact with the medium
- Transmittance is computed using ray marching (adaptive, jittered)
- A photon can be absorbed or scattered, depending on albedo
- An absorbed photon is stored in the volume photon map
- A scattered photon shoots off in a new direction according to a distribution

Rendering

- Same as above, but use calculate the smallest sphere containing K photons
- For participating media, use ray marching and jittered/adaptive sampling
- At each step, compute the emission and in-scattering contributions of the step

32 Radiosity

The radiosity of a surface consists of two parts: emitted and reflected energy

Given radiosity B_i, emittance E_i, albedo ρ_i, irradiance H_i: $B_i = E_i + \rho_i H_i$

- The above holds for all surfaces in the scene
- We want to discover the B_i values, which are constant for a steady state scene
- The irradiance H_i is the light reaching patch i from all directions

Given the areas A_i and A_j and form factor F_{ji}, the irradiance from patch j is $H_i = \sum_{j=1}^{N} B_j \frac{A_j F_{ji}}{A_i}$

Form factors F_{ij} and F_{ji} are related as $A_i F_{ij} = A_j F_{ji}$ or $F_{ij} = F_{ji} A_j / A_i$

Radiance Equation defines a set of linear equations: $E_i = B_i - \rho_i \sum_{j=1}^{N} B_j F_{ij}$

- F_{ii} form factors are zero, E_i are zero except for light sources, sum of the form factors in a row is ≤ 1.

$$\begin{bmatrix} 1 & -\rho_1 F_{12} & \dots & -\rho_1 F_{1N} \\ -\rho_2 F_{21} & 1 & \dots & -\rho_2 F_{2N} \\ \dots & \dots & \dots & \dots \\ -\rho_N F_{N1} & -\rho_N F_{N2} & \dots & 1 \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ \dots \\ B_N \end{bmatrix} = \begin{bmatrix} E_1 \\ E_2 \\ \dots \\ E_N \end{bmatrix} \tag{5}$$

Above matrix can be solved using Gauss-Seidel iteration. The matrix will be large.

Radiosity algorithm

- Divide the world into small patches, possibly using adaptive subdivision
- Calculate the form factors between all patches ($N^2/2$ form factors) or use lazy evaluation
- Solve for the B_i radiosity values using Gauss-Seidel iteration
- Use the radiosities in a standard z-buffer rendering pipeline using Gouraud shading
- Can add surface reflection with Phong shading
- Body reflection vertex color is the average of the adjoining patches

Calculating form factors: hemi-cube algorithm, use z-buffer rendering to estimate F values

Re-ordering the radiosity solution: go over progressive radiosity

- Shooting light from patches instead of gathering light from patches
- Allows faster viewing of the rough solution

33 OpenGL Pipeline

- Vertex specification
  - Set up the vertices and what kind of data they need
  - The attributes of a vertex are arbitrary, meaning is assigned by what is done with them
  - Almost all stages are programmable, some are optional
- Vertex Shader (not optional)
  - Vertex shaders do whatever needs doing at the vertex level
  - For simple shading, the vertex should end with a position in clip space (e.g. CVV space)
- Tessellation (optional)
  - Determines the amount of tessellation to apply to a primitive
  - Three stages: how much tessellation, tessellation, interpolating vertex values to the new vertices
- Geometry shader (optional)
  - Process primitives
  - Have access to vertices adjacent to the primitive being processed
  - Can produce more primitives as outputs
- Vertex post-processing
  - Transform feedback: allows transformations from earlier layers to be recalled
  - Clipping to the view volume, primitives are subdivided as necessary
- Primitive Assembly
  - Collect vertex data and make simple primitives (points, lines, triangles)
  - Face culling: remove one-sided polygons facing the wrong way
- Rasterization
  - Rasterize the primitives
  - The result of rasterization is a set of fragments
  - A fragment is a set of state used to compute the final data for a pixel (or sub-pixel if multi-sampling)
  - State includes screen space position or sample coverage and arbitrary data
- Fragment shader (optional)
  - Determines the color of the pixel, depth value, stencil value
- Per-Sample Operations
  - Scissor test: is it in a specified rectangle of the screen?
  - Stencil test: is it in a user provided stencil

- Depth test: is it visible given the depth buffer
- Blending: any blending?
- Logical operation: any bitwise ops?
- Write mask: a final output mask to determine what kinds of values can be written