CS488/688
Introduction to Computer Graphics

Lecture Notes

Fall 2003

University of Waterloo
School of Computer Science
Computer Graphics Lab
Instructor(s): Michael McCool

Contents

1 Administration

♠ *Course overview and administration.*

1.1 General Information

Instructor(s): Michael McCool

TA(s): Guillaume Poirier, Yi Lin, Tiberiu Popa

Electronic Resources:
Web: `http://www.student.cs.uwaterloo.ca/~cs488/`
Newsgroup: `uw.cs.cs488`
Email: `cs488@cgl.uwaterloo.ca`

Texts:

- D. Hearn and M.P. Baker,
  Computer Graphics, 2nd ed. (required)
- Course Overview and Notes (online or at Bookstore)
- *OpenGL Programming Guide* (optional; also on `insight`)
- Mark Lutz,
  Programming Python 2nd edition (optional; also man pages)

Assignments:
5 Assignments (A0-A4) plus a Project (A0 is worth 0 marks).

Marking Algorithm:
Assignments: 32%; Project: 24%; Midterm: 14%; Final: 30%.
Must obtain 50% in both Programming and Exam portions to pass.

Programming Environment:
C, UNIX, OpenGL, PyThon; Linux PCs.
Two Windows NT PCs (selected projects only)

General Comments:
A tough course: lots of work! Be committed.
If you don't know C, you will have major problems.
Do **NOT** take graphics and either real-time (CS452) or compilers (CS444).

♠ *The printed course notes contain information on marking, important dates, the cheating policy, and so forth. The course web pages contain all this information plus a list of frequently-asked questions, a gallery, a glossary, and past exam questions. They also contain information that cannot be known until the term starts, such as office hours.*

Assignment specifications are online. Make sure you print out and sign the Objectives sheet for each assignment; include this signed sheet with your hardcopy submission. There is a checklist included with assignment 0 that you should make a copy of for every assignment. Use it to help make sure you get the formalities right.

Several books are listed for the Readings for the course. These books are the current and previous books used for the course. The Red Book and the White Book refer to two editions of books by Foley, van Dam, et al. You are only responsible for readings in the current course textbook; however, use of any of the listed texts should be sufficient for this course.

1.2 Topics Covered

- Graphics Pipeline and Hardware
- Graphical User Interface (GUI) Implementation Techniques
- Mathematical Foundations:
  - Affine and Projective Geometry, Linear Algebra and Transformations
  - Numerical Analysis and Splines
  - Signal Processing
- Modelling and Data Structures
- Hidden Surface Removal
- Colour and the Human Visual System
- Lighting and Shading
- Ray Tracing
- Global Illumination (optional; if time permits)
- Animation (optional; if time permits)

1.3 Assignments

Number: 5 Assignments (A0-A4) plus a Project (A0 worth 0 marks)

Environment: 9 SGI Octanes, and 8 Linux PCs.
You are expected to share machines. Be nice!

Code Credit: ***You must request in `README`.***
For each missed objective, in `README` state what is wrong and how you would attempt to fix it, mark up printout of code.

Hand in: At start of class (first 5 minutes). NO LATES!
Hand in documentation only; code read online.

Documentation Requirements:

- Title page (with name, student ID, userid)
- ***Signed*** *objective list:* 10 points you are graded on; see course notes.
- `README` and Manual
- Annotated hardcopy (only if code broken)
- Checksum (run `/u/gr/cs488/bin/grsubmit` to obtain)

Assignment Format:

- *Very strict!*
- Read assignment policies in course notes.

- **Do A0 to avoid losing marks later.**
- See (and use) checklist.

Code:

- Run `/u/gr/cs488/bin/setup`.
- All files need to be `cs488` group owned and group readable.
- See course notes for assignment specs and objectives.

♠ *Feel free to ask anyone editing on Octanes to log out if you need the console to test a program that requires the graphics hardware. People working on Octanes should occasionally allow classmates to pop up windows for quick testing. Students in FINE 328D may need the graphics console to run Maya or Houdini. Please be gracious and use the X terminals for editing, and show these students how to set up their account to allow testing of your program. See the course notes for details on how to get X11 to allow users logged in remotely to bring up windows.*

You may develop at home on Windows or Linux; the combination of Tcl/Tk, Togl, and OpenGL was in fact chosen to permit this. HOWEVER, we will only be testing your assignment submissions on Octanes. If it doesn't work on an Octane, you will lose marks, so test your assignments thoroughly on the Octanes before you submit them.

2 Introduction

2.1 A Short History

Early 60's: Computer animations for physical simulation;
Edward Zajac displays satellite research using CG in 1961

1963: Sutherland (MIT)
Sketchpad (direct manipulation, CAD)
Calligraphic (vector) display devices
Interactive techniques
Douglas Englebart invents the mouse.

1968: Evans & Sutherland founded

1969: First SIGGRAPH

Late 60's to late 70's: Utah Dynasty

1970: Pierre Bézier develops Bézier curves

1971: Gouraud Shading

1972: Pong developed

1973: Westworld, The first film to use computer animation

1974: Ed Catmull develops z-buffer (Utah)
First Computer Animated Short, *Hunger*:
Keyframe animation and morphing

1975: Bui-Toung Phong creates Phong Shading (Utah)
Martin Newell models a teapot with Bézier patches (Utah)

Mid 70's: Raster graphics (Xerox PARC, Shoup)

1976: Jim Blinn develops texture and bump mapping

1977: Star Wars, CG used for Death Star plans

1979: Turner Whitted develops ray tracing

Mid 70's - 80's: Quest for realism
radiosity; also mainstream real-time applications.

1982: Tron, Wrath of Kahn. Particle systems and obvious CG

1984: The Last Star Fighter, CG replaces physical models. Early attempts at realism using CG

1986: First CG animation nominated for an Academy Award:
Luxo Jr. (Pixar)

1989: Tin Toy (Pixar) wins Academy Award

1995: Toy Story (Pixar and Disney), the first full length fully computer-generated 3D animation
Reboot, the first fully 3D CG Saturday morning cartoon
Babylon 5, the first TV show to routinely use CG models

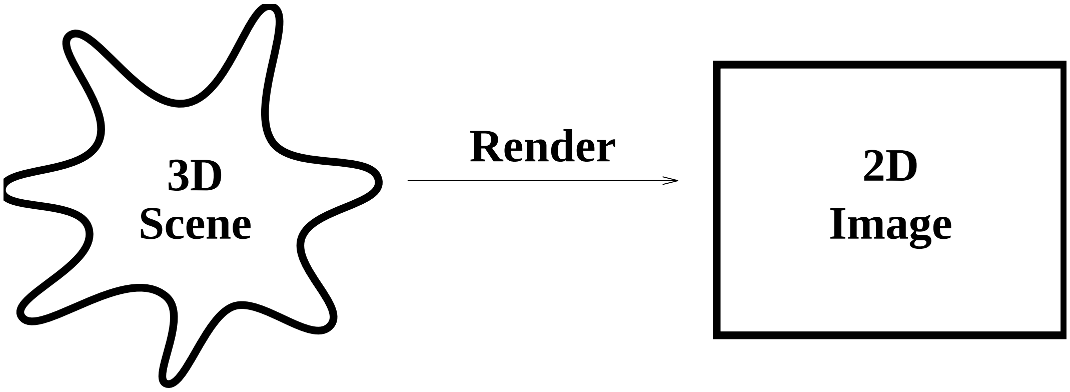
late 90's: Interactive environments, scientific and medical visualization, artistic rendering, image based rendering, path tracing, photon maps, etc.

00's: Real-time photorealistic rendering on consumer hardware? Interactively rendered movies? Ubiquitous computing, vision and graphics?

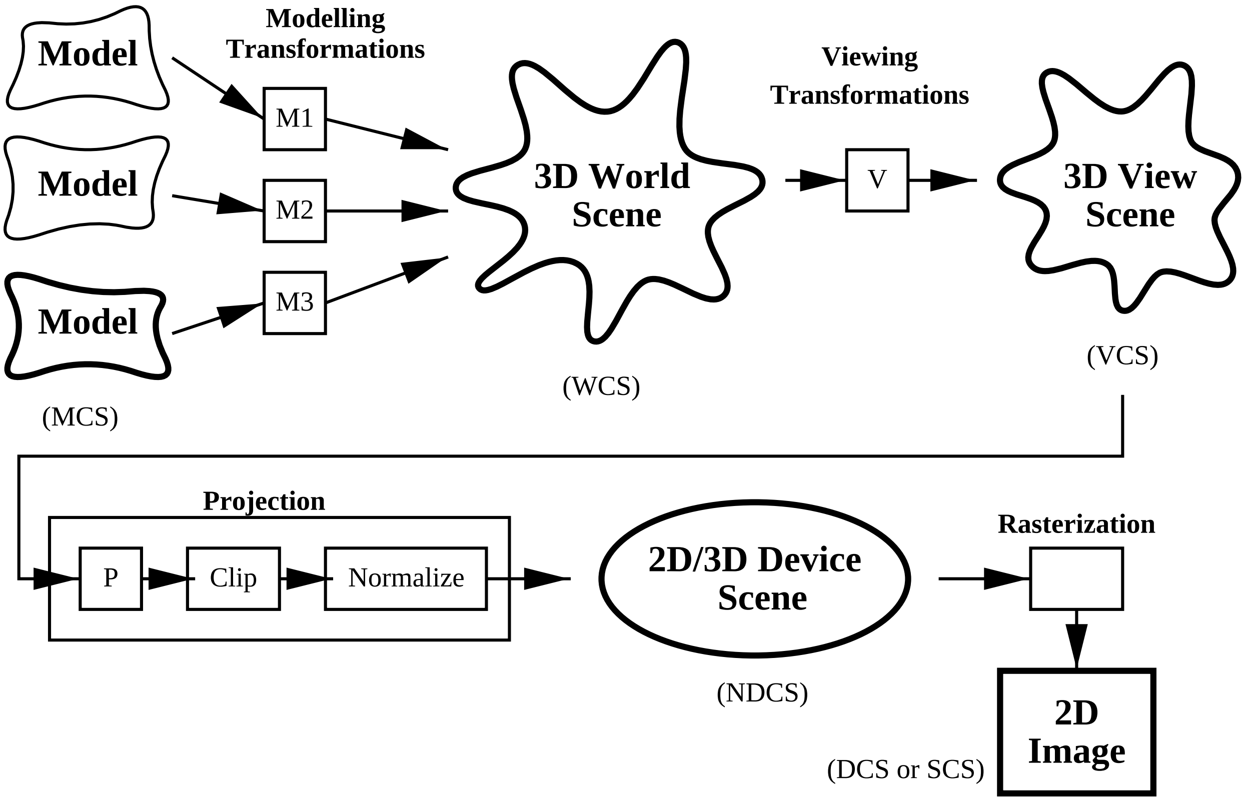
♠ *Readings: Watt, Preface (optional), Hearn and Baker, Chapter 1 (optional), Red book [Foley, van Dam et al: Introduction to Computer Graphics], Chapter 1 (optional). White book [Foley, van Dam et al: Computer Graphics: Principles and Practice, 2nd ed.], Chapter 1 (optional).*

2.2 The Graphics Rendering Pipeline

- **Rendering** is the conversion of a **scene** into an **image**:



- Scenes are composed of **models** in three-dimensional space.
  Models are composed of **primitives** supported by the rendering system.
- Models entered by hand or created by a program.
- The image is drawn on monitor, printed on laser printer, or written to a raster in memory or a file.
  Requires us to consider **device independence**.
- Classically, "model" to "scene" to "image" conversion broken into finer steps, called the **graphics pipeline**.
- Parts of pipeline often implemented in graphics hardware to get interactive speeds.
- Hardware accelerators beginning to support *programmable* pipelines.
- In a programmable pipeline, some of pipeline stages are in fact programmable units…so you can implement your own lighting models or geometric transformations.
- Programmable graphics accelerator: GPU
- You have to know what you're doing (i.e. know the math) to use a GPU!
- The basic **forward projection** pipeline looks like:



- Each stage refines the scene,
  converting primitives in modelling space to primitives in device space,
  where they are converted to pixels (rasterized).
- A number of coordinate systems are used:

  MCS: Modelling Coordinate System.

  WCS: World Coordinate System.

  VCS: Viewer Coordinate System.

  NDCS: Normalized Device Coordinate System.

  DCS or SCS: Device Coordinate System or equivalently the Screen Coordinate System.

  Keeping these straight is the key to understanding a rendering system.
- Transformation between two coordinate systems represented with matrix.
- Derived information may be added (lighting and shading) and primitives may be removed (hidden surface removal) or modified (clipping).

♠ *Readings: Watt, Chapter 5 introduction, 5.1. Hearn and Baker, Section 6-1 (but they give a more detailed version than used here). Red book, 6-1, 6-6 (intro). White book, 8-3, Blinn: 16.*

We begin with a description of **forward rendering***, which is the kind of rendering usually supported in hardware and is the model OpenGL uses. In forward rendering, rendering primitives are transformed, usually in a conceptual pipeline, from the model to the device.*

However, **raytracing***, which we will consider later in the course, is a form of* **backward rendering***. In backward rendering, we start with a point in the image and work out what model primitives project to it.*

Both approaches have advantages and disadvantages.

2.3 Rendering Primitives

- Models are composed of, or can be converted to, a large number of **geometric primitives**.
- Typical rendering primitives directly supported in hardware include:
  - Points (single pixels)
  - Line Segments
  - Polygons (perhaps only convex polygons or triangles).
- Modelling primitives include these, but also
  - Piecewise polynomial (spline) curves
  - Piecewise polynomial (spline) surfaces
  - Implicit surfaces (quadrics, blobbies, etc)
  - Other...
- Software renderer may support modelling primitives directly, or may convert them into polygonal or linear approximations for hardware rendering.

2.4 Algorithms

A number of basic algorithms are needed:

- **Transformation:** Convert representations of models/primitives from one coordinate system to another.
- **Clipping/Hidden Surface Removal:** Remove primitives and parts of primitives that are not visible on the display.
- **Rasterization:** Convert a projected screen-space primitive to a set of pixels.

Later, we will look at some more advanced algorithms:

- **Picking:** Select a 3D object by clicking an input device over a pixel location.
- **Shading and Illumination:** Simulate the interaction of light with a scene.
- **Animation:** Simulate movement by rendering a sequence of frames.

2.5 Application Programming Interfaces

Application Programming Interfaces (APIs):

X11: 2D rasterization.
PostScript, PDF: 2D transformations, 2D rasterization
Phigs+, GL, OpenGL, Direct3D: 3D pipeline

APIs provide access to rendering hardware via a conceptual model.

APIs hide which graphics algorithms are or are not implemented in hardware by simulating missing pieces in software.

For 3D interactive applications, we might modify the scene or a model directly or just the viewing information.

After each modification, usually the image needs to be regenerated.

We need to interface to input devices in an event-driven, asynchronous, and device independent fashion. APIs and toolkits are also defined for this task; we will be using X11 through Tcl/Tk.

3 Devices and Device Independence

♠ *In this module, we*

- *Consider display devices for computer graphics:*
  - *Calligraphic devices*
  - *Raster devices: CRT's, LCD's.*
  - *Direct vs. pseudocolour frame buffers*
- *Discuss the problem of device independence:*
  - *Window-to-viewport mapping*
  - *Normalized device coordinates*

3.1 Calligraphic and Raster Devices

Calligraphic Display Devices draw polygon and line segments directly:

- Plotters
- Direct Beam Control CRTs
- Laser Light Projection Systems

Raster Display Devices represent an image as a regular grid of **samples**.

- Each sample is usually called a **pixel**
- Both are short for *picture element.*
- Rendering requires **rasterization algorithms** to quickly determine a sampled representation of geometric primitives.

3.2 How a Monitor Works

Raster Cathode Ray Tubes (CRTs) most common display device

- Capable of high resolution.
- Good colour fidelity.
- High contrast (100:1).
- High update rates.

Electron beam scanned in regular pattern of horizontal **scanlines**.

Raster images stored in a **frame buffer**.

Frame buffers composed of **VRAM** (video RAM).

VRAM is dual-ported memory capable of

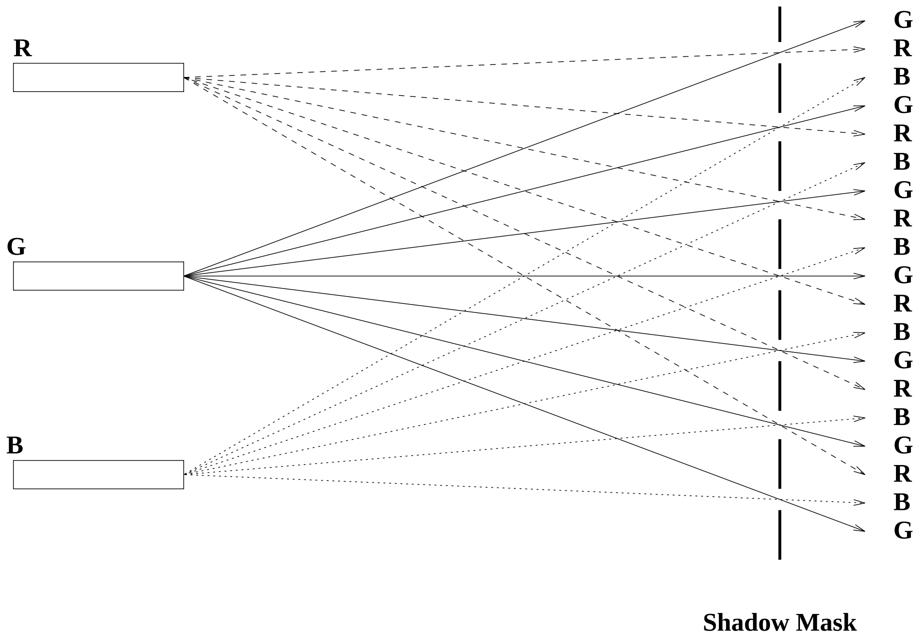
- Random access
- Simultaneous high-speed serial output: built-in **serial shift register** can output entire *scanline* at high rate synchronized to **pixel clock.**

Intensity of electron beam modified by the pixel value.

Burst-mode DRAM replacing VRAM in many systems.

Colour CRTs have three different colours of phosphor and three independent electron guns.

Shadow Masks allow each gun to irradiate only one colour of phosphor.



Colour is specified either

- Directly, using three independent intensity channels, or
- Indirectly, using a **Colour Lookup Table (LUT)**.

In the latter case, a **colour index** is stored in the frame buffer.

Sophisticated frame buffers may allow different colour specifications for different portions of frame buffer. Use a **window identifier** also stored in the frame buffer.

Liquid Crystal Displays (LCDs) becoming more popular and reasonably priced

- Flat panels
- Flicker free
- Decreased viewing angle

Works as follows:

- Random access to cells like memory.
- Cells contain liquid crystal molecules that align when charged.
- Unaligned molecules twist light.
- Polarizing filters allow only light through unaligned molecules.
- Subpixel colour filter masks used for RGB.

♠ *Reading: Watt, none. Hearn and Baker, Chapter 2. Red book, Chapter 4. White book, Chapter 4. LCD reference: http://www.cgl.uwaterloo.ca/˜pogilhul/present/lcd*

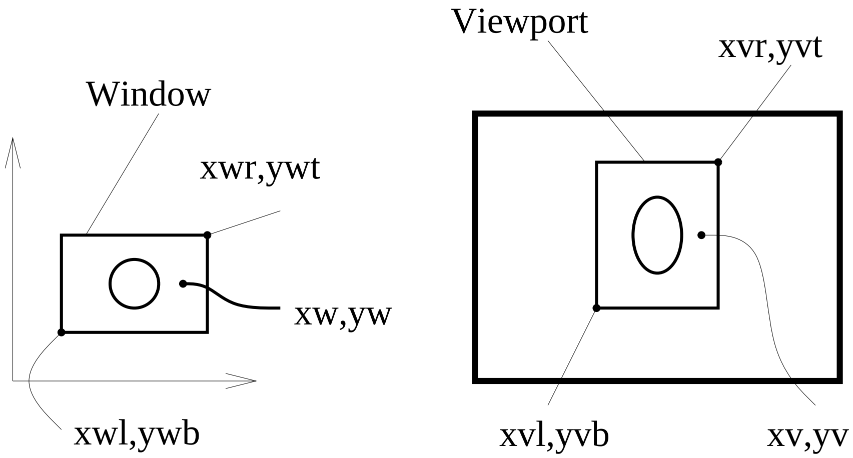
3.3 Window to Viewport Mapping

- Start with 3D scene, but eventually project to 2D scene
- 2D scene is infinite plane. Device has a finite visible rectangle.
  What do we do?
- Answer: map rectangular region of 2D device scene to device.

  Window: rectangular region of interest in scene.

  Viewport: rectangular region on device.

  Usually, both rectangles are aligned with the coordinate axes.



- Window point (x_w, y_w) maps to viewport point (x_v, y_v).

  Window has corners (x_{wl}, y_{wb}) and (x_{wr}, y_{wt});
  Viewport has corners (x_{vl}, y_{vb}) and (x_{vr}, y_{vt}).

  Length and height of the window are L_w and H_w,
  Length and height of the viewport are L_v and H_v.

- Proportionally map each of the coordinates according to:

$$\frac{\Delta x_w}{L_w} = \frac{\Delta x_v}{L_v}, \qquad \frac{\Delta y_w}{H_w} = \frac{\Delta y_v}{H_v}.$$

- To map x_w to x_v:

$$\begin{aligned} \frac{x_w - x_{wl}}{L_w} &= \frac{x_v - x_{vl}}{L_v} \\ \Rightarrow \quad x_v &= \frac{L_v}{L_w}(x_w - x_{wl}) + x_{vl}, \end{aligned}$$

  and similarly for y_v.

- If $H_w/L_w \neq H_v/L_v$ the image will be distorted.
  These quantities are called the **aspect ratios** of the window and viewport.

♠ *Readings: Watt: none. Hearn and Baker: Section 6-3. Red book: 5.5. White book: 5.4, Blinn: 16.*

Intuitively, the window-to-viewport formula can be read as:

- *Convert x_w to a distance from the window corner.*
- *Scale this w distance to get a v distance.*
- *Add to viewport corner to get x_v.*

3.4 Normalized Device Coordinates

- Where do we specify our viewport?
- Could specify it in device coordinates …
  BUT, suppose we want to run program on several hardware platforms or graphic devices.

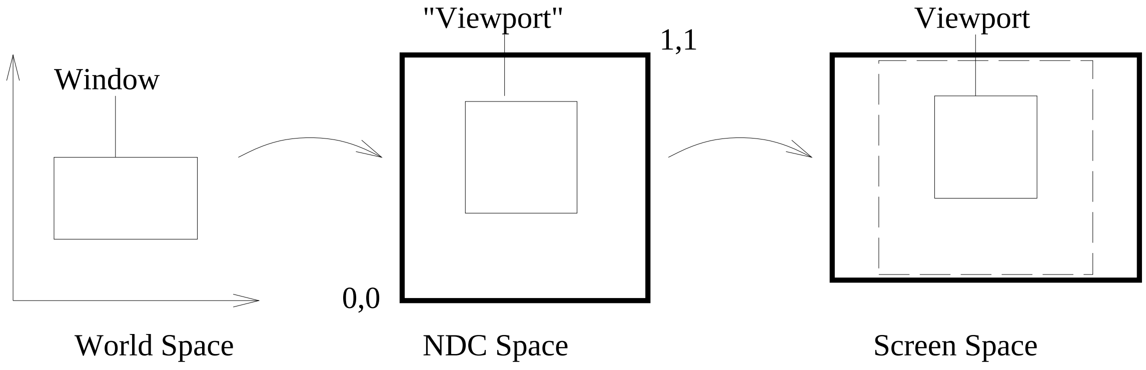
  Two common conventions for DCS:

  - Origin in the lower left corner, with x to the right and y upward.
  - Origin in the top left corner, with x to the right and y downward.

  Many different resolutions for graphics display devices:

  - Workstations commonly have 1280×1024 frame buffers.
  - A PostScript page is 612×792 points, but 2550×3300 pixels at 300dpi.
  - And so on …

  Aspect ratios may vary …
- If we map directly from WCS to a DCS, then changing our device requires rewriting this mapping (among other changes).
- Instead, use **Normalized Device Coordinates (NDC)** as an intermediate coordinate system that gets mapped to the device layer.
- Will consider using only a square portion of the device.
  Windows in WCS will be mapped to viewports that are specified within a unit square in NDC space.
- Map viewports from NDC coordinates to the screen.



♠ *Reading: Watt: none. Hearn and Baker: Sections 2-7, 6-3. Red book: 6.3. White book: 6.5.*

4 Basic User Interface Concepts

♠ *A short outline of input devices and the implementation of a graphical user interface is given:*

- *Physical input devices used in graphics*
- *Virtual devices*
- *Polling is compared to event processing*
- *UI toolkits are introduced by generalizing event processing*

4.1 Physical Devices

Actual, physical input devices include:

- Dials (Potentiometers)
- Selectors
- Pushbuttons
- Switches
- Keyboards (collections of pushbuttons called “keys”)
- Trackballs (relative motion)
- Mice (relative motion)
- Joysticks (relative motion, direction)
- Tablets (absolute position)
- Etc.

Need some abstractions to keep organized…
♠ *Readings: Watt: none. Hearn and Baker, Chapter 8. Red book: 8.1. White book: 8.1.*

4.2 Virtual Devices

Devices can be classified according to the kind of value they return:

Button: Return a Boolean value; can be *depressed* or *released.*

Key: Return a “character”; that is, one of a given set of code values.

String: Return a sequence of characters.

Selector: Return an integral value (in a given range).

Choice: Return an option (menu, callback, ...)

Valuator: Return a real value (in a given range).

Locator: Return a position in (2D/3D) space (eg. ganged valuators).

Stroke: Return a sequence of positions.

Pick: Return a scene component.

Each of the above is called a **virtual device**.

4.3 Device Association

To obtain device independence:

- Design an application in terms of virtual (abstract) devices.
- Implement virtual devices using available physical devices.

There are certain natural associations:

- Valuator ↔ Mouse-X

But if the naturally associated device does not exist on a platform,
one can make do with other possibilities:

- Valuator ↔ number entered on keyboard.

“public interface / private implementation”

4.4 Device Input Modes

Input from devices may be managed in different ways:

Request Mode: Alternating application and device execution

- application requests input and then suspends execution;
- device wakes up, provides input and then suspends execution;
- application resumes execution, processes input.

Sample Mode: Concurrent application and device execution

- device continually updates register(s) or memory location(s);
- application may read at any time.

Event Mode: Concurrent application and device execution together with a concurrent queue management service

- device continually offers input to the queue
- application may request selections and services from the queue
  (or the queue may interrupt the application).

4.5 Application Structure

With respect to device input modes,
applications may be structured to engage in

- **requesting**
- **polling** or **sampling**
- **event processing**

Events may or may not be **interruptive**.
If not interruptive, they may be read in a

- **blocking**
- **non-blocking**

fashion.

4.6 Polling and Sampling

In **polling**,

- Value of input device constantly checked in a tight loop
- Wait for a change in status

Generally, polling is inefficient and should be avoided, particularly in time-sharing systems.
In **sampling**, value of an input device is read and then the program proceeds.

- No tight loop
- Typically used to track sequence of actions (the mouse)

4.7 Event Queues

- Device is monitored by an asynchronous process.
- Upon change in status of device, this process places a record into an **event queue.**
- Application can request read-out of queue:
  - Number of events
  - 1st waiting event
  - Highest priority event
  - 1st event of some category
  - All events
- Application can also
  - Specify which events should be placed in queue
  - Clear and reset the queue

  - Etc.

- Queue reading may be blocking or non-blocking
- Processing may be through callbacks
- Events may be processed interruptively
- Events can be associated with more than physical devices…
  Windowing system can also generate virtual events, like "Expose".

Without interrupts, the application will engage in an *event loop*

- not a tight loop
- a preliminary of *register event* actions followed by a repetition of *test for event* actions.

For more sophisticated queue management,

- application merely registers event-process pairs
- queue manager does all the rest
  "if event E then invoke process P."
- The cursor is usually **bound** to a pair of valuators, typically `MOUSE_X` and `MOUSE_Y`.
- Events can be restricted to particular areas of the screen, based on the cursor position.
- Events can be very general or specific:
  - A mouse button or keyboard key is depressed.
  - A mouse button or keyboard key is released.
  - The cursor enters a window.
  - The cursor has moved more than a certain amount.
  - An `Expose` event is triggered under X when a window becomes visible.
  - A `Configure` event is triggered when a window is resized.
  - A timer event may occur after a certain interval.
- Simple event queues just record a code for event (Iris GL).
- Better event queues record extra information such as time stamps
  (X windows).

4.8 Toolkits and Callbacks

Event-loop processing can be generalized:

- Instead of `switch`, use table lookup.
- Each table entry associates an event with a **callback** function.
- When event occurs, corresponding **callback** is invoked.
- Provide an API to make and delete table entries.
- Divide screen into parcels, and assign different callbacks to different parcels (X Windows does this).
- Event manager does most or all of the administration.

Modular UI functionality is provided through a set of **widgets**:

- **Widgets** are parcels of the screen that can respond to events.
- A widget has a graphical representation that suggests its function.
- Widgets may respond to events with a change in appearance, as well as issuing callbacks.
- Widgets are arranged in a parent/child hierarchy.
  - Event-process definition for parent may apply to child, and child may add additional event-process definitions
  - Event-process definition for parent may be redefined within child
- Widgets may have multiple parts, and in fact may be composed of other widgets in a heirarchy.

Some UI toolkits: Xm, Xt, SUIT, FORMS, Tk, Qt ... UI toolkits recommended for projects: Tk, GLUT, GLUI, SDL.

4.9 Example for Discussion

```
#!/bin/sh
#\
  exec wish "$0" "$@"

button .hello -text "Hello, world" -command {
    puts stdout "Hello, world"; destroy .
}
pack .hello
```

1. Where is the definition of the event?
2. What constitutes the event?
3. What process is associated with the event?
4. Who manages the event loop?

5 Clipping

5.1 Point and Line Clipping

Clipping: Remove points outside a region of interest.

- Discard (parts of) primitives outside our window…

Point clipping: Remove points outside window.

- A point is either entirely inside the region or not.

Line clipping: Remove portion of line segment outside window.

- Line segments can straddle the region boundary.
- Liang-Barsky algorithm efficiently clips line segments to a halfspace.
- Halfspaces can be combined to bound a convex region.
- Use **outcodes** to better organize combinations of halfspaces.
- Can use some of the ideas in Liang-Barsky to clip points.

Parametric representation of line:

$$L(t) = (1-t)A + tB$$

or equivalently

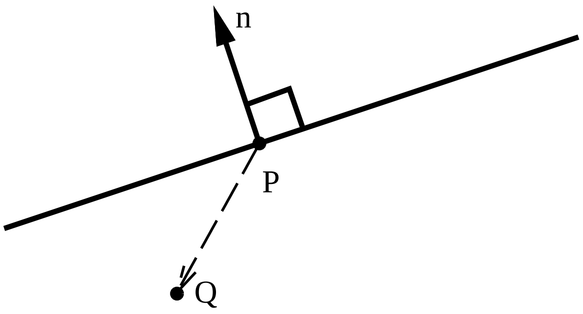
$$L(t) = A + t(B-A)$$

- A and B are non-coincident points.
- For $t \in \mathbb{R}$, $L(t)$ defines an infinite line.
- For $t \in [0,1]$, $L(t)$ defines a line segment from A to B.
- Good for generating points on a line.
- Not so good for testing if a given point is on a line.

Implicit representation of line:

$$\ell(Q) = (Q-P) \cdot \vec{n}$$

- P is a point on the line.
- $\vec{n}$ is a vector perpendicular to the line.
- $\ell(Q)$ gives us the signed distance from any point Q to the line.
- The sign of $\ell(Q)$ tells us if Q is on the left or right of the line, relative to the direction of $\vec{n}$.
- If $\ell(Q)$ is zero, then Q is on the line.
- Use same form for the implicit representation of a halfspace.



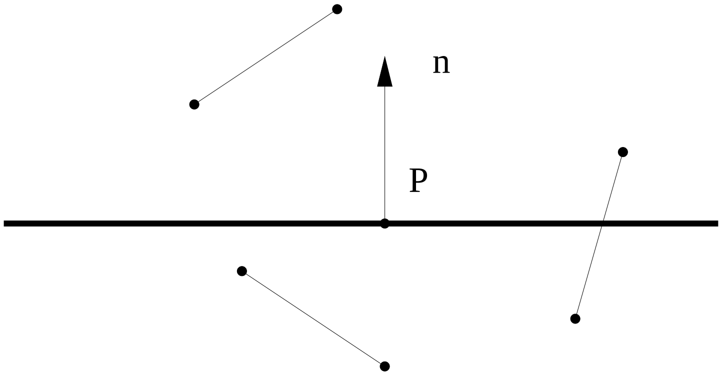
Clipping a point to a halfspace:

- Represent window edge as implicit line/halfspace.
- Use the implicit form of edge to classify a point Q.
- Must choose a convention for the normal:
  points to the *inside.*
- Check the sign of $\ell(Q)$:
  - If $\ell(Q) > 0$, then Q is inside.
  - Otherwise clip (discard) Q:
    It is on the edge or outside.
    May want to keep things on the boundary.

Clipping a line segment to a halfspace:

There are three cases:

1. The line segment is entirely inside:
   Keep it.
2. The line segment is entirely outside:
   Discard it.
3. The line segment is partially inside and partially outside:
   Generate new line to represent part inside.



Input Specification:

- Window edge: implicit, $\ell(Q) = (Q - P) \cdot \vec{n}$
- Line segment: parametric, $L(t) = A + t(B - A)$.

Do the easy stuff first:

We can devise easy (and fast!) tests for the first two cases:

- $\ell(A) < 0$ **AND** $\ell(B) < 0 \Longrightarrow$ Outside
- $\ell(A) > 0$ **AND** $\ell(B) > 0 \Longrightarrow$ Inside

Need to decide: are **boundary points inside** or **outside**?

Trivial tests are important in computer graphics:

- Particularly if the trivial case is the most common one.
- Particularly if we can reuse the computation for the non-trivial case.

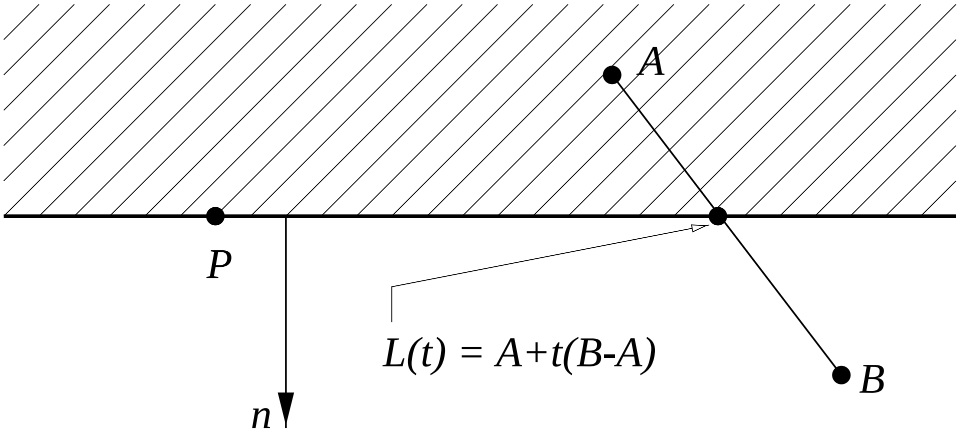
Do the hard stuff only if we have to:

If line segment partially inside and partially outside, need to clip it:

- Line segment from A to B in **parametric** form:

$$L(t) = (1 - t)A + tB = A + t(B - A)$$

- When $t = 0$, $L(t) = A$. When $t = 1$, $L(t) = B$.
- We now have the following:



- We want t such that $\ell(L(t)) = 0$:

$$\begin{aligned}(L(t) - P) \cdot \vec{n} &= (A + t(B - A) - P) \cdot \vec{n} \\ &= (A - P) \cdot \vec{n} + t(B - A) \cdot \vec{n} \\ &= 0\end{aligned}$$

- Solving for t gives us

$$t = \frac{(A - P) \cdot \vec{n}}{(A - B) \cdot \vec{n}}$$

- **NOTE:**
  The values we use for our simple test can be reused to compute t:

$$t = \frac{(A-P)\cdot\vec{n}}{(A-P)\cdot\vec{n}-(B-P)\cdot\vec{n}}$$

Clipping a line segment to a window:
Just clip to each of four halfspaces in turn.

Pseudo-code (here wec = *window-edge coordinates*):

```
Given line segment (A,B), clip in-place:
for each edge (P,n)
    wecA = (A-P) . n
    wecB = (B-P) . n
    if ( wecA < 0 AND wecB < 0 ) then reject
    if ( wecA >= 0 AND wecB >= 0 ) then next
    t = wecA / (wecA - wecB)
    if ( wecA < 0 ) then
        A = A + t*(B-A)
    else
        B = A + t*(B-A)
    endif
endfor
```

Note:

- Liang-Barsky Algorithm can clip lines to any **convex** window.
- Optimizations can be made for the special case of horizontal and vertical window edges.

Question:
Should we clip before or after window-to-viewport mapping?

Line-clip Algorithm generalizes to 3D:

- Half-space now lies on one side of a *plane.*
- Plane also given by normal and point.
- Implicit formula for plane in 3D is same as that for line in 2D.
- Parametric formula for line to be clipped is unchanged.

♠ *Reading: Watt: none. Hearn and Baker: Sections 6-5 through 6-7. Red book: 3.9. White Book: 3.11. Blinn: 13.*

6 Geometries

♠ *Geometry provides a mathematical foundation for much of computer graphics:*

- *Geometric Spaces: Vector, Affine, Euclidean, Cartesian, Projective*

- *Affine Geometry*
- *Affine Transformations*
- *Perspective*
- *Projective Transformations*
- *Matrix Representation of Transformations*
- *Viewing Transformations*
- *Quaternions and Orientation*

You probably learned geometry relative to coordinate axes, with points and vectors being given by (x, y, z) *triples.*

- *You "move that far" in each coordinate (away from the origin) to find a point.*
- *Points are fixed while vectors are free to move anywhere.*
- *This approach glosses over mathematical details...*

Formally, there is a hierarchy of geometric spaces, defined by

- *Sets of objects.*
- *Operations on those objects.*
- *Invariants.*

6.1 Vector Spaces

Definition:

- Set of vectors $\mathcal{V}$.
- Two operations. For $\vec{v}, \vec{u} \in \mathcal{V}$:

  Addition: $\vec{u} + \vec{v} \in \mathcal{V}$.

  Scalar multiplication: $\alpha\vec{u} \in \mathcal{V}$, where α is a member of some field $\mathbb{F}$, (i.e. $\mathbb{R}$).

Axioms:

Addition Commutes: $\vec{u} + \vec{v} = \vec{v} + \vec{u}$.

Addition Associates: $(\vec{u} + \vec{v}) + \vec{w} = \vec{u} + (\vec{v} + \vec{w})$.

Scalar Multiplication Distributes: $\alpha(\vec{u} + \vec{v}) = \alpha\vec{u} + \alpha\vec{v}$.

Unique Zero Element: $\vec{0} + \vec{u} = \vec{u}$.

Field Unit Element: $1\vec{u} = \vec{u}$.

Span:

- Suppose $\mathcal{B} = \{\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_n\}$.
- $\mathcal{B}$ **spans** $\mathcal{V}$ iff any $\vec{v} \in \mathcal{V}$ can be written as $\vec{v} = \sum_{i=1}^{n} \alpha_i \vec{v}_i$.

- This is written $\langle \mathcal{B} \rangle = \mathcal{V}$.
- $\sum_{i=1}^{n} \alpha_i \vec{v}_i$ is a **linear combination** of the vectors in $\mathcal{B}$.

Basis:

- Any minimal spanning set is a basis.
- All bases are the same size.

Dimension:

- The number of vectors in any basis.
- We will work in 2 and 3 dimensional spaces.

Note: In the definition of a Vector Space:

- No notion of distance or size
- No notion of angles
- No "points"

These are Euclidean and Affine space concepts.

6.2 Affine Space

Definition: Set of Vectors $\mathcal{V}$ and a Set of Points $\mathcal{P}$

- Vectors $\mathcal{V}$ form a **vector space**.
- Points can be combined with vectors to make new points:
  $P + \vec{v} \Rightarrow Q$ with $P, Q \in \mathcal{P}$ and $\vec{v} \in \mathcal{V}$.

Frame: An affine extension of a basis:
Requires a vector basis plus a point $\mathcal{O}$ (the **origin**):

$$F = (\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_n, \mathcal{O})$$

Dimension: The dimension of an affine space is the same as that of $\mathcal{V}$.

Note:

- All other point operations are just variations on the $P + \vec{v}$ operation.
- No distinguished origin
- No notion of distances or angles.
- Most of what we do in graphics is affine.

6.3 Euclidean Space

Metric Space: Any space with a **distance metric** $d(P, Q)$ defined on its elements.

Distance Metric:

- Metric $d(P, Q)$ must satisfy the following axioms:
  1. $d(P, Q) \geq 0$
  2. $d(P, Q) = 0$ iff $P = Q$.
  3. $d(P, Q) = d(Q, P)$.
  4. $d(P, Q) \leq d(P, R) + d(R, Q)$.
- Distance is intrinsic to the space, and *not* a property of the frame.

Euclidean Space: Metric is based on a dot (inner) product:

$$d^2(P, Q) = (P - Q) \cdot (P - Q)$$

Note:

- Dot product is defined on *vectors*.
- Distance metric is defined on *points*.

Dot product:

$$\begin{aligned}
(\vec{u} + \vec{v}) \cdot \vec{w} &= \vec{u} \cdot \vec{w} + \vec{v} \cdot \vec{w}, \\
\alpha(\vec{u} \cdot \vec{v}) &= (\alpha\vec{u}) \cdot \vec{v} \\
&= \vec{u} \cdot (\alpha\vec{v}) \\
\vec{u} \cdot \vec{v} &= \vec{v} \cdot \vec{u}.
\end{aligned}$$

Norm:

$$|\vec{u}| = \sqrt{\vec{u} \cdot \vec{u}}.$$

Angles:

$$\cos(\angle \vec{u}\vec{v}) = \frac{\vec{u} \cdot \vec{v}}{|\vec{u}||\vec{v}|}$$

Perpendicularity:

$$\vec{u} \cdot \vec{v} = 0 \;\Rightarrow \vec{u} \perp \vec{v}$$

Perpendicularity is *not* an affine concept! There is no notion of angles in affine space.

6.4 Cartesian Space

Cartesian Space: An Euclidean space with an standard **orthonormal frame** $(\vec{\imath}, \vec{\jmath}, \vec{k}, \mathcal{O})$.

Orthogonal: $\vec{\imath} \cdot \vec{\jmath} = \vec{\jmath} \cdot \vec{k} = \vec{k} \cdot \vec{\imath} = 0$.

Normal: $|\vec{\imath}| = |\vec{\jmath}| = |\vec{k}| = 1$.

Notation: Specify the **Standard Frame** as $F_S = (\vec{\imath}, \vec{\jmath}, \vec{k}, \mathcal{O})$.

As defined previously, points and vectors are

- Different objects.
- Have different operations.
- Behave differently under transformation.

Coordinates: Use an "an extra coordinate":

- 0 for vectors: $\vec{v} = (v_x, v_y, v_z, 0)$ means $\vec{v} = v_x\vec{\imath} + v_y\vec{\jmath} + v_z\vec{k}$.
- 1 for points: $P = (p_x, p_y, p_z, 1)$ means $P = p_x\vec{\imath} + p_y\vec{\jmath} + p_z\vec{k} + \mathcal{O}$.
- Later we'll see other ways to view the fourth coordinate.
- **Coordinates have no meaning without an associated frame!**
  - Sometimes we'll omit the extra coordinate ...
    point or vector by context.
  - Sometimes we may not state the frame ...
    the Standard Frame is assumed.

6.5 Summary of Geometric Spaces

Space	**Objects**	**Operators**
Vector Space	Vector	$\vec{u} + \vec{v}$, $\alpha\vec{v}$
Affine Space	Vector, Point	$\vec{u} + \vec{v}$, $\alpha\vec{v}$, $P + \vec{v}$
Euclidean Space	Vector and Point	$\vec{u} + \vec{v}$, $\alpha\vec{v}$, $P + \vec{v}$, $\vec{u} \cdot \vec{v}$, $d(P, Q)$
Cartesian Space	Vector, Point, O.N. Frame	$\vec{u} + \vec{v}$, $\alpha\vec{v}$, $P + \vec{v}$, $\vec{u} \cdot \vec{v}$, $d(P, Q)$, $\vec{\imath}, \vec{\jmath}, \vec{k}, \mathcal{O}$

Question: where does the cross product enter?

♠ *Readings: Watt: 1.1, 1.2. White book: Appendix A. Hearn and Baker: A-2, A-3.*

7 Affine Geometry and Transformations

♠

Affine Space: $\mathcal{A} = (\mathcal{V}, \mathcal{P})$.

- *Vectors in $\mathcal{V}$ form a vector space.*
- *Points in $\mathcal{P}$ can be added to vectors to generate new points: $P + \vec{v} \in \mathcal{P}$.*

Will Review:

- *Linear Combinations*
- *Linear Transformations*
- *Affine Combinations*
- *Affine Transformations*
- *Matrix Representation of Transformations*

7.1 Linear Combinations

Vectors:

$$\mathcal{V} = \{\vec{u}\}, \quad \vec{u} + \vec{v}, \quad \alpha\vec{u}$$

By Extension:

$$\sum_i \alpha_i \vec{u}_i \ \ \text{(no restriction on } \alpha_i\text{)}$$

Linear Transformations:

$$T(\vec{u} + \vec{v}) = T(\vec{u}) + T(\vec{v}), \quad T(\alpha\vec{u}) = \alpha T(\vec{u})$$

By Extension:

$$T(\sum_i \alpha_i \vec{u}_i) = \sum_i \alpha_i T(\vec{u}_i)$$

Points:

$$\mathcal{P} = \{P\}, \quad P + \vec{u} = Q$$

By Extension...

♠ *Readings: White book, Appendix A; Red book, 5.1*

7.2 Affine Combinations

Define Point Subtraction:

$Q - P$ means $\vec{v} \in \mathcal{V}$ such that $Q = P + \vec{v}$ for $P, Q \in \mathcal{P}$.

By Extension:

$$\sum \alpha_i P_i \text{ is a } \textbf{vector} \text{ iff } \sum \alpha_i = 0$$

Define Point Blending:

$Q = (1 - a)Q_1 + aQ_2$ means $Q = Q_1 + a(Q_2 - Q_1)$ with $Q \in \mathcal{P}$

Alternatively:

we may write $Q = a_1 Q_1 + a_2 Q_2$ where $a_1 + a_2 = 1$.

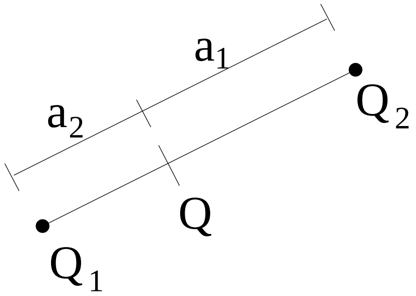
By Extension:

$$\sum a_i P_i \text{ is a } \textbf{point} \text{ iff } \sum a_i = 1$$

Geometrically:

- The following ratio holds for $Q = a_1 Q_1 + a_2 Q_2$

$$\frac{|Q - Q_1|}{|Q - Q_2|} = \frac{a_2}{a_1} \quad (a_1 + a_2 = 1)$$



- If Q breaks the line segment $\overline{Q_1Q_2}$ into the ratio $b_2 : b_1$ then

$$Q = \frac{b_1Q_1 + b_2Q_2}{b_1 + b_2} \quad (b_1 + b_2 \neq 0)$$

Legal vector combinations:
Vectors can be formed into any combinations $\sum_i \alpha_i \vec{u}_i$ (a "linear combination").

Legal point combinations:
Points can be formed into combinations $\sum_i a_i P_i$ iff

- The coefficients sum to 1: The result is a point (an "affine combination").
- The coefficients sum to 0: The result is a vector (a "vector combination").

Parametric Line Equation:
has geometric meaning (in an affine sense):

$$\begin{aligned} L(t) &= A + t(B - A) \\ &= (1 - t)A + tB \end{aligned}$$

The weights t and $(1 - t)$ create an affine combination.
The result is a point (on the line).

Parametric Ray Equation:
Same as above, but $t \geq 0$. Will write as

$$R(t) = A + t\vec{d}$$

Where A is the point of origin and $\vec{d}$ is the direction of the ray. Used in ray-tracing.

♠ *Readings: White book, Appendix A*

7.3 Affine Transformations

Let $T : \mathcal{A}_1 \longmapsto \mathcal{A}_2$, where $\mathcal{A}_1$ and $\mathcal{A}_2$ are affine spaces.

Then T is said to be an affine transformation if:

- T maps vectors to vectors and points to points
- T is a linear transformation on the vectors

- $T(P + \vec{u}) = T(P) + T(\vec{u})$

By Extension:

T preserves affine combinations on the points:

$$T(a_1Q_1 + \cdots + a_nQ_n) = a_1T(Q_1) + \cdots + a_nT(Q_n),$$

If $\sum a_i = 1$, result is a point.
If $\sum a_i = 0$, result is a vector.

Observations:

- Affine transformations map lines to lines:

  $$T((1-t)P_0 + tP_1) = (1-t)T(P_0) + tT(P_1)$$

- Affine transformations map rays to rays:

  $$T(A + t\vec{d}) = T(A) + tT(\vec{d})$$

- Affine transformations preserve ratios of distance along a line (converse is also true: preserves ratios of such distances ⇒ affine).
- Although ratios of distances preserved, no guarantees that absolute distances or angles themselves will be preserved.
  Absolute distances and angles are not affine concepts…

Examples:

- translations
- rotations
- scales
- shears
- reflections

Translations and rotations called *rigid body motions.*

Affine vs Linear

- Which is a larger class of transformations: Affine or Linear?
  - T is linear if $T(aA + bB) = aT(A) + bT(B)$ for all a, b.
  - T is affine if $T(aA + bB) = aT(A) + bT(B)$ if $a + b = 1$.
- If T is linear, then $T(aA + bB) = aT(A) + bT(B)$ when $a + b = 1$ so T is clearly affine.
- But if T is affine and $a + b \neq 1$ then $T(aA + bB)$ might not equal $aT(A) + bT(B)$.
  Thus, it is possible for T to be affine but not linear.
- Therefore, affine transformations are a superclass of linear transformations (surprise!)
- An alternate way to think of this is that the behaviour of linear transformations is restricted for a wider range of values.
  Consider $T(aA + bB)$

  - Linear transformations have their behaviour specified for all values of a, b.
  - Affine transformations have their behaviour specified only when $a + b = 1$. They can behave differently for all other values of $a + b$.
- Most of the transformations we consider will be linear.
- Translation is the only non-linear (but affine) transformation we need to worry about, and we'll "bypass" its non-linear behavior.

Theorem: Affine transformations map parallel lines to parallel lines.

Proof: Let $\ell_1(t) = R + t(Q - R)$ and $\ell_2(t) = U + t(V - U)$.

For parallel lines, $Q - R = \alpha(V - U)$, which implies $T(Q) - T(R) = \alpha(T(V) - T(U))$ by linearity.

Then

$$\begin{aligned} T(\ell_2(t)) &= T(U + t(V - U)) \\ &= T(U) + t(T(V) - T(U)) \end{aligned}$$

and

$$\begin{aligned} T(\ell_1(t)) &= T(R + t(Q - R)) \\ &= T(R) + t(T(Q) - T(R)) \\ &= T(R) + \alpha t(T(V) - T(U)) \end{aligned}$$

and we see that the images of both lines are parallel.

This theorem is also true for rays.

Suppose we only have T defined on points.

Define $T(\vec{v})$ as follows:

- There exists points Q and R such that $\vec{v} = Q - R$.
- Define $T(\vec{v})$ to be $T(Q) - T(R)$.

Note that Q and R are not unique.

The definition works for $P + \vec{v}$:

$$\begin{aligned} T(P + \vec{v}) &= T(P + Q - R) \\ &= T(P) + T(Q) - T(R) \\ &= T(P) + T(\vec{v}) \end{aligned}$$

This can now be used to show that the definition is well defined.
If $Q - R = B - C$ then

$$\begin{aligned} & T(P + Q - R) &= T(P + B - C) \\ \Rightarrow\ & T(P) + T(Q) - T(R) &= T(P) + T(B) - T(C) \\ \Rightarrow\ & T(Q) - T(R) &= T(B) - T(C) \end{aligned}$$

How to map points/vectors through an affine transformation? How do we actually do
the computation
on a numerical representation of points and vectors
to find a numerical representation of the result?

Need to pick a numerical representation; use *coordinates*:

Let $\mathcal{A}$ and $\mathcal{B}$ be affine spaces.

- Let $T : \mathcal{A} \mapsto \mathcal{B}$ be an affine transformation.
- Let $F_{\mathcal{A}} = (\vec{v}_1, \vec{v}_2, \mathcal{O}_{\mathcal{V}})$ be a frame for $\mathcal{A}$.
- Let $F_{\mathcal{B}} = (\vec{w}_1, \vec{w}_2, \mathcal{O}_{\mathcal{W}})$ be a frame for $\mathcal{B}$.
- Let P be a point in $\mathcal{A}$ whose *coordinates* relative $F_{\mathcal{A}}$ are $(p_1, p_2, 1)$.
  $(P = p_1\vec{v}_1 + p_2\vec{v}_2 + 1\mathcal{O}_{\mathcal{V}})$

The points $\mathcal{O}_{\mathcal{V}}$ and $\mathcal{O}_{\mathcal{W}}$ are called the origins of their respective frames.

Question: What are the coordinates $(p'_1, p'_2, 1)$ of $T(P)$ relative to the frame $F_{\mathcal{B}}$?

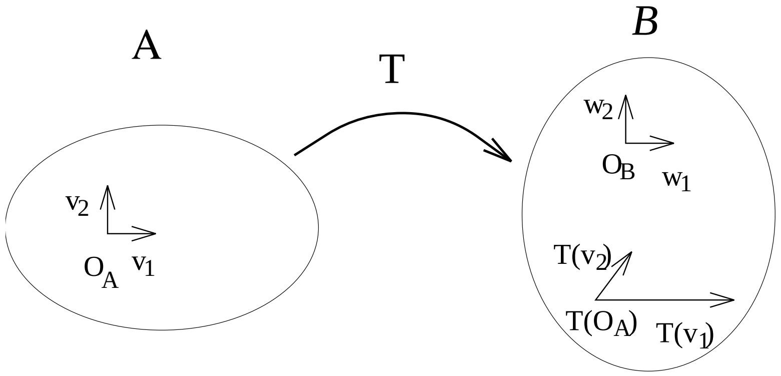
Fact: An affine transformation is completely characterized by the image of a frame in the domain:

$$\begin{aligned} T(P) &= T(p_1\vec{v} + p_2\vec{v} + \mathcal{O}_{\mathcal{V}}) \\ &= p_1T(\vec{v}_1) + p_2T(\vec{v}_2) + T(\mathcal{O}_{\mathcal{V}}). \end{aligned}$$

If

$$\begin{aligned} T(\vec{v}_1) &= t_{1,1}\vec{w}_1 + t_{2,1}\vec{w}_2 \\ T(\vec{v}_2) &= t_{1,2}\vec{w}_1 + t_{2,2}\vec{w}_2 \\ T(\mathcal{O}_{\mathcal{V}}) &= t_{1,3}\vec{w}_1 + t_{2,3}\vec{w}_2 + \mathcal{O}_{\mathcal{W}} \end{aligned}$$

then we can find $(p'_1, p'_2, 1)$ by substitution and gathering like terms.



Substitution:

$$
\begin{aligned}
T(P) &= T(p_1\vec{v} + p_2\vec{v} + \mathcal{O}_{\mathcal{V}}) \\
&= p_1 T(\vec{v}_1) + p_2 T(\vec{v}_2) + T(\mathcal{O}_{\mathcal{V}}) \\
&= p_1(t_{1,1}\vec{w}_1 + t_{2,1}\vec{w}_2) + \\
&\quad p_2(t_{1,2}\vec{w}_1 + t_{2,2}\vec{w}_2) + \\
&\quad t_{1,3}\vec{w}_1 + t_{2,3}\vec{w}_2 + \mathcal{O}_{\mathcal{W}} \\
&= (p_1 t_{1,1} + p_2 t_{1,2} + t_{1,3})\vec{w}_1 + \\
&\quad (p_1 t_{2,1} + p_2 t_{2,2} + t_{2,3})\vec{w}_2 + \\
&\quad \mathcal{O}_{\mathcal{W}} \\
&= p'_1\vec{w}_1 + \\
&\quad p'_2\vec{w}_2 + \\
&\quad \mathcal{O}_{\mathcal{W}}.
\end{aligned}
$$

so

$$
\begin{aligned}
p'_1 &= p_1 t_{1,1} + p_2 t_{1,2} + t_{1,3}, \\
p'_2 &= p_1 t_{2,1} + p_2 t_{2,2} + t_{2,3}.
\end{aligned}
$$

♠ *Readings: White book, Appendix A*

7.4 Matrix Representation of Transformations

Represent points and vectors as $n \times 1$ matrices. In 2D,

$$
\begin{aligned}
P \equiv \mathbf{p} &= \begin{bmatrix} p_1 \\ p_2 \\ 1 \end{bmatrix} \\
\vec{v} \equiv \mathbf{v} &= \begin{bmatrix} v_1 \\ v_2 \\ 0 \end{bmatrix}
\end{aligned}
$$

This is a Shorthand: Coordinates are specified relative to a frame $F = (\vec{v}_1, \vec{v}_2, \mathcal{O}_{\mathcal{V}})$:

$$
\begin{aligned}
P &\equiv [\vec{v}_1, \vec{v}_2, \mathcal{O}_{\mathcal{V}}] \begin{bmatrix} p_1 \\ p_2 \\ 1 \end{bmatrix} \\
&= F\mathbf{p}.
\end{aligned}
$$

Technically,

- Should write the $\mathbf{p}$ as $\mathbf{p}_F$.
- The frame F should note what space it's in.

Usually, we're lazy and let '**p**' denote both

- The point
- Its matrix representation relative to an understood frame.

Transformations:

- $F_A = (\vec{v}_1, \vec{v}_2, \mathcal{O}_\mathcal{V})$ is the frame of the domain,
- $F_B = (\vec{w}_1, \vec{w}_2, \mathcal{O}_\mathcal{W})$ is the frame of the range.

Then …

$$\begin{aligned}
T(P) &= T(F_A\mathbf{p}) \\
&= [T(\vec{v}_1), T(\vec{v}_2), T(\mathcal{O}_\mathcal{V})] \begin{bmatrix} p_1 \\ p_1 \\ 1 \end{bmatrix} \\
&= \left[\ t_{1,1}\vec{w}_1 + t_{2,1}\vec{w}_2; \quad t_{1,2}\vec{w}_1 + t_{2,2}\vec{w}_2; \quad t_{1,3}\vec{w}_1 + t_{2,3}\vec{w}_2 + \mathcal{O}_\mathcal{W}\ \right] \begin{bmatrix} p_1 \\ p_2 \\ 1 \end{bmatrix} \\
&= [\vec{w}_1, \vec{w}_2, \mathcal{O}_\mathcal{W}] \underbrace{\begin{bmatrix} t_{1,1} & t_{1,2} & t_{1,3} \\ t_{2,1} & t_{2,2} & t_{2,3} \\ 0 & 0 & 1 \end{bmatrix}}_{M_T} \begin{bmatrix} p_1 \\ p_2 \\ 1 \end{bmatrix} \\
&= [\vec{w}_1, \vec{w}_2, \mathcal{O}_\mathcal{W}] M_T \begin{bmatrix} p_1 \\ p_2 \\ 1 \end{bmatrix} = F_B M_T \mathbf{p} \\
&= [\vec{w}_1, \vec{w}_2, \mathcal{O}_\mathcal{W}] \begin{bmatrix} p'_1 \\ p'_2 \\ 1 \end{bmatrix} = F_B \mathbf{p}'
\end{aligned}$$

$P = \mathbf{p}$ transforms to $\mathbf{p}' = M_T\mathbf{p}$.

Can also read this as $F_A = F_B M_T$.

M_T is said to be the **matrix representation** of T relative to F_A and F_B.

- First column of M_T is representation of $T(\vec{v}_1)$ in F_B.
- Second column of M_T is representation of $T(\vec{v}_2)$ in F_B.
- Third column of M_T is representation of $T(\mathcal{O}_\mathcal{V})$ in F_B.

7.5 Geometric Transformations

Construct matrices for simple geometric transformations.

Combine simple transformations into more complex ones.

- Assume that the range and domain frames are the Standard Frame.

- Will begin with 2D, generalize later.

Translation: Specified by the vector $[\Delta x, \Delta y, 0]^T$:

- A point $[x, y, 1]^T$ will map to $[x + \Delta x, y + \Delta y, 1]^T$.
- A vector will remain unchanged under translation.
- Translation is **NOT** a linear transformation.
  If it were, then (ignoring the fact that point addition is not allowed) $T(P + Q) = T(P) + T(Q)$ would hold, **BUT**:

$$\begin{aligned} T(P) + T(Q) &= [P_x + \Delta x, P_y + \Delta y, 1]^T + [Q_x + \Delta x, Q_y + \Delta y, 1]^T \\ &= [P_x + Q_x + 2\Delta x, P_y + Q_y + 2\Delta y, 2]^T \\ T(P + Q) &= [P_x + Q_x + \Delta x, P_y + Q_y + \Delta y, 2]^T \end{aligned}$$

- Translation is linear on sums of vectors…

Matrix representation of translation

- We can create a matrix representation of translation:

$$\overbrace{\begin{bmatrix} 1 & 0 & \Delta x \\ 0 & 1 & \Delta y \\ 0 & 0 & 1 \end{bmatrix}}^{T(\Delta x, \Delta y)} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x + \Delta x \\ y + \Delta y \\ 1 \end{bmatrix}$$

- $T(\Delta x, \Delta y)$ will mean the above matrix.
- Note that vectors are unchanged by this matrix.
- Although more expensive to compute than the other version of translation, we prefer this one:
  - Uniform treatment of points and vectors
  - Other transformations will also be in matrix form.
    We can compose transformations by matrix multiply. Thus, the composite operation less expensive if translation composed, too.

Scale about the origin:

Specified by factors $s_x, s_y \in \mathbb{R}$.

- Applies to points or vectors, is linear.
- A point $[x, y, 1]^T$ will map to $[s_x x, s_y y, 1]^T$.
- A vector $[x, y, 0]^T$ will map to $[s_x x, s_y y, 0]^T$.
- Matrix representation:

$$\overbrace{\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}}^{S(s_x, s_y)} \begin{bmatrix} x \\ y \\ 0 \text{ or } 1 \end{bmatrix} = \begin{bmatrix} s_x x \\ s_y y \\ 0 \text{ or } 1 \end{bmatrix}$$

Rotation: Counterclockwise about the origin, by angle θ.

- Applies to points or vectors, is linear.
- Matrix representation:

$$\overbrace{\begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}}^{R(\theta)} \begin{bmatrix} x \\ y \\ 0 \text{ or } 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ 0 \text{ or } 1 \end{bmatrix}$$

Shear: Intermixes coordinates according to $\alpha, \beta \in \mathbb{R}$:

- Applies to points or vectors, is linear.
- Matrix representation:

$$\overbrace{\begin{bmatrix} 1 & \beta & 0 \\ \alpha & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}}^{Sh(\alpha,\beta)} \begin{bmatrix} x \\ y \\ 0 \text{ or } 1 \end{bmatrix} = \begin{bmatrix} x + \beta y \\ \alpha x + y \\ 0 \text{ or } 1 \end{bmatrix}$$

- Easiest to see if we set one of α or β to zero.

Reflection: Through a line.

- Applies to points or vectors, is linear.
- Example: through x-axis, matrix representation is

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 0 \text{ or } 1 \end{bmatrix} = \begin{bmatrix} x \\ -y \\ 0 \text{ or } 1 \end{bmatrix}$$

- See book for other examples

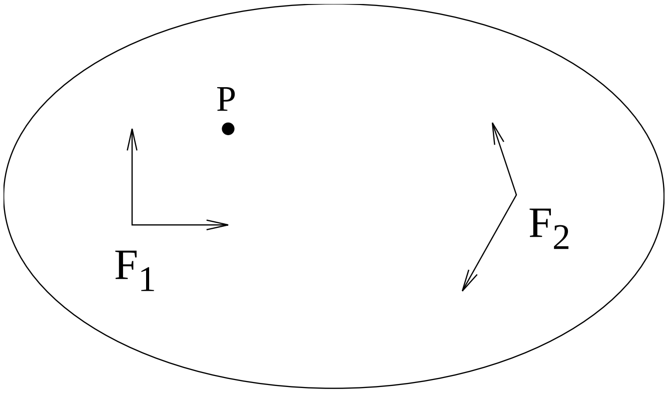
Note: Vectors map through all these transformations as we want them to.

♠ *Readings: Hearn and Baker, 5-1, 5-2, and 5-4; Red book, 5.2, 5.3; White book, 5.1, 5.2*

7.6 Change of Basis

Suppose:

- We have two coordinate frames for a space, F_1 and F_2,
- Want to change from coordinates relative to F_1 to coordinates relative to F_2.



Know $P \equiv \mathbf{p} = [x, y, 1]^T$ relative to $F_1 = (\vec{w}_1, \vec{w}_2, \mathcal{O}_W)$.

Want the coordinates of P relative to $F_2 = (\vec{v}_1, \vec{v}_2, \mathcal{O}_V)$.

- Express each component of F_1 in terms of F_2:

$$\begin{aligned} \vec{w}_1 &= f_{1,1}\vec{v}_1 + f_{2,1}\vec{v}_2 \\ \vec{w}_2 &= f_{1,2}\vec{v}_1 + f_{2,2}\vec{v}_2 \\ \mathcal{O}_W &= f_{1,3}\vec{v}_1 + f_{2,3}\vec{v}_2 + \mathcal{O}_V \end{aligned}$$

- The change of coordinates is given by the matrix

$$M_{1,2} = \begin{bmatrix} f_{1,1} & f_{1,2} & f_{1,3} \\ f_{2,1} & f_{2,2} & f_{2,3} \\ 0 & 0 & 1 \end{bmatrix}$$

  Factoring gives

$$[\vec{w}_1 \ \vec{w}_2 \ \mathcal{O}_W] = F_1 = F_2 M_{1,2} = [\vec{v}_1 \ \vec{v}_2 \ \mathcal{O}_V] M_{1,2}$$

- Consider:

$$\begin{aligned} P &\equiv F_1\mathbf{p} \\ &= F_2 M_{1,2}\mathbf{p} \\ P &\equiv F_2\mathbf{p}' \\ \mathbf{p}' &= M_{1,2}\mathbf{p} \end{aligned}$$

How do we get $f_{i,j}$?

- If F_2 is orthonormal:

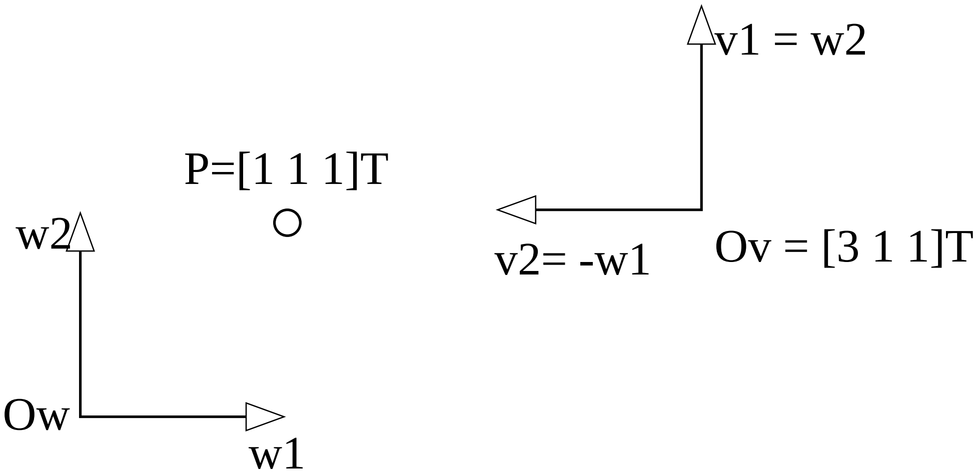
$$f_{i,j} = \vec{w}_j \cdot \vec{v}_i,$$
$$f_{i,3} = (\mathcal{O}_W - \mathcal{O}_V) \cdot \vec{v}_i.$$

- If F_2 is orthogonal:

$$f_{i,j} = \frac{\vec{w}_j \cdot \vec{v}_i}{\vec{v}_i \cdot \vec{v}_i},$$
$$f_{i,3} = \frac{(\mathcal{O}_W - \mathcal{O}_V) \cdot \vec{v}_i}{\vec{v}_i \cdot \vec{v}_i}.$$

- Otherwise, we have to solve a small system of linear equations, using $F_1 = F_2 M_{1,2}$.
- Change of basis from F_1 to Standard Cartesian Frame is trivial (since frame elements normally expressed with respect to Standard Cartesian Frame).

Example:



where F_W is the standard coordinate frame.

- Matrices mapping from/to F_W to/from F_V:

$$M_{WV} = \begin{bmatrix} & & \\ & & \\ 0 & 0 & 1 \end{bmatrix}, \qquad M_{VW} = \begin{bmatrix} & & \\ & & \\ 0 & 0 & 1 \end{bmatrix}$$

- Check
  $F_W[0\ 0\ 1]^T = F_V M_{WV}[0\ 0\ 1]^T =$
  $F_W[1\ 1\ 1]^T =$
  $F_V[0\ 0\ 1]^T =$

Generalization to 3D is straightforward …

Example:

- Define two frames:

$$\begin{aligned}
F_W &= (\vec{w}_1, \vec{w}_2, \vec{w}_3, \mathcal{O}_W) \\
&= \left(\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \right) \\
F_V &= (\vec{v}_1, \vec{v}_2, \vec{v}_3, \mathcal{O}_V) \\
&= \left(\begin{bmatrix} \sqrt{2}/2 \\ \sqrt{2}/2 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} \sqrt{2}/2 \\ -\sqrt{2}/2 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \\ 3 \\ 1 \end{bmatrix} \right)
\end{aligned}$$

- All coordinates are specified relative to the standard frame in a Cartesian 3 space.
- The standard frame happens to be identical to F_W.
- Note that both F_W and F_V are orthonormal.
- The matrix mapping F_W to F_V is given by

$$M = \begin{bmatrix} \sqrt{2}/2 & \sqrt{2}/2 & 0 & -\sqrt{2}/2 \\ 0 & 0 & 1 & -3 \\ \sqrt{2}/2 & -\sqrt{2}/2 & 0 & -\sqrt{2}/2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Check this matrix:
  Map each element of F_V,
  See that we get that element relative to F_W.
  (e.g, $M\vec{v}_1 = [1, 0, 0, 0]^T$)
- Question: What is the matrix mapping from F_V to F_W?

Notes

- On the computer, frame elements usually specified in Standard Frame for space.
  Eg, a frame $F = [\vec{v}_1,\ \vec{v}_2,\ \mathcal{O}_V]$ is given by

$$[\ [v_{1x},\ v_{1y},\ 0]^T,\ \ [v_{2x},\ v_{2y},\ 0]^T,\ \ [v_{3x},\ v_{3y},\ 1]^T\ ]$$

  relative to Standard Frame.

  Question: What are coordinates of these basis elements relative to F?
- Frames are usually orthonormal.
- A point "mapped" by a change of basis does *not change*;
  We have merely expressed its coordinates relative to a different frame.

♠ *Readings: Watt: 1.1.1. Hearn and Baker: Section 5-5 (not as general as here, though). Red book: 5.9. White book: 5.8.*

7.7 Compositions of Transformations

Suppose we want to rotate around an arbitrary point $P \equiv [x, y, 1]^T$.

- Could derive a more general transformation matrix …
- Alternative idea: Compose simple transformations
  1. Translate P to origin
  2. Rotate around origin
  3. Translate origin back to P
- Suppose $P = [x_o, y_o, 1]^T$
- The the desired transformation is

$$\begin{aligned}
&T(x_o, y_o) \circ R(\theta) \circ T(-x_o, -y_o) \\
&= \begin{bmatrix} 1 & 0 & x_o \\ 0 & 1 & y_o \\ 0 & 0 & 1 \end{bmatrix} \circ \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \circ \begin{bmatrix} 1 & 0 & -x_o \\ 0 & 1 & -y_o \\ 0 & 0 & 1 \end{bmatrix} \\
&= \begin{bmatrix} \cos(\theta) & -\sin(\theta) & x_o(1-\cos(\theta)) + y_o\sin(\theta) \\ \sin(\theta) & \cos(\theta) & y_o(1-\cos(\theta)) - x_o\sin(\theta) \\ 0 & 0 & 1 \end{bmatrix}
\end{aligned}$$

- Note where P maps: to P.
- Won't compute these matrices analytically;
  Just use the basic transformations,
  and run the matrix multiply numerically.

Order is important!

$$\begin{aligned}
T(-\Delta x, -\Delta y) \circ T(\Delta x, \Delta y) \circ R(\theta) &= R(\theta) \\
&\neq T(-\Delta x, -\Delta y) \circ R(\theta) \circ T(\Delta x, \Delta y).
\end{aligned}$$

♠ *Readings: Hearn and Baker, Section 5-3; Red book, 5.4; White book, 5.3*

7.8 Ambiguity

Three Types of Transformations:

1. $T : \mathcal{A} \mapsto \mathcal{B}$ (between two spaces)
2. $T : \mathcal{A} \mapsto \mathcal{A}$ ("warp" an object within its own space)
3. T : change of coordinates

Changes of Coordinates:

- Given 2 frames:
  - $F_1 = (\vec{v}_1, \vec{v}_2, \mathcal{O})$, orthonormal,
  - $F_2 = (2\vec{v}_1, 2\vec{v}_2, \mathcal{O})$, orthogonal.
- Suppose $\vec{v} \equiv F_1[1, 1, 0]^T$.

- Then $\vec{v} \equiv F_2[1/2, 1/2, 0]^T$.

Question: What is the length of $\vec{v}$?

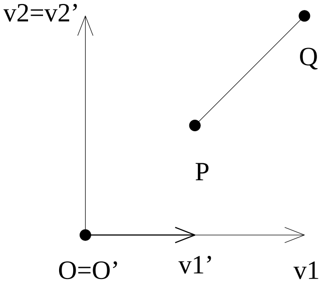
Suppose we have $P \equiv [p_1, p_2, 1]^T$ and $Q \equiv [q_1, q_2, 1]^T$

- P, Q relative to $F = (\vec{v}_1, \vec{v}_2, \mathcal{O})$,
- We are given a matrix representation of a transformation T:

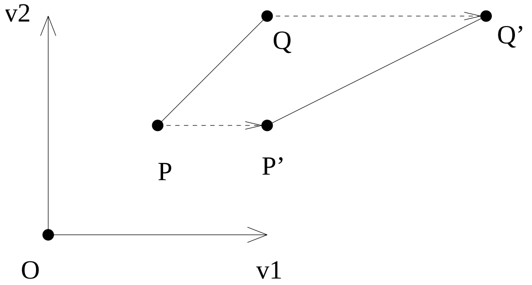
$$M_T = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- Consider $P' = TP$ and $Q' = TQ$.
- How do we interpret P' and Q'?

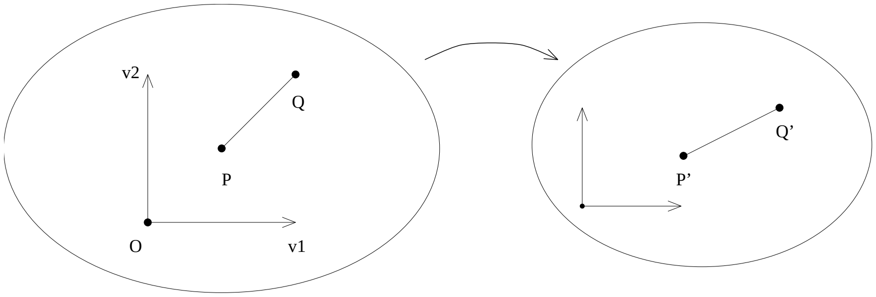
1. Change of Coordinates?



2. Scale?



3. Transformations between spaces?



- With no other information, any of the above is a valid interpretation.

Do we care? YES!

- In (1) nothing changes except the representation.
- In (1) distances are preserved while they change in (2) and the question has no meaning in (3).
- In (3), we've completely changed spaces.

Consider the meaning of $|P' - P|$

1. $|P' - P| = 0$
2. $|P' - P| = \sqrt{(2p_1 - p_1)^2 + (p_2 - p_2)} = |p_1|$
3. $|P' - P|$ has no meaning

To fully specify a transformation, we need

1. A matrix
2. A domain space
3. A range space
4. A coordinate frame in each space

Most of the time not all will be specified!
. . . So be careful out there.

7.9 3D Transformations

Assume a right handed coordinate system

Points $P \equiv [x, y, z, 1]^T$, Vectors $\vec{v} \equiv [x, y, z, 0]^T$

Translation:

$$T(\Delta x, \Delta y, \Delta z) = \begin{bmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scale: About the origin

$$S(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Shear: see book

Reflection: see book

Rotation: About a coordinate axis

$$R_z(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_y(\theta) = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Note the signs in R_y. The convention is that a 90 degree counterclockwise rotation transforms one positive axis into another: R_x takes $+y$ to $+z$; R_y takes $+z$ to $+x$; R_z takes $+x$ to $+y$.

Composition: works same way (but order counts when composing rotations).

♠ *Readings: Hearn and Baker, Chapter 11; Red book, 5.7; White book, 5.6*

7.10 World and Viewing Frames

- Typically, our space S is a Cartesian space.
  - Call the standard frame the **world frame**.
  - The world frame is typically **right handed**.
  - Our scene description is specified in terms of the world frame.
- The viewer may be anywhere and looking in any direction.
  - Often, x to the right, y up, and z straight ahead.
    - z is called the *viewing direction*.
    - Note that this is a **left handed** coordinate system.
  - We could instead specify z and y as vectors
    - z is the the **view direction**.
    - y is the *up vector*.
    - Compute $x = y \times z$
    - Get a **right handed** coordinate system.
  - We can do a change of basis
    - Specify a frame relative to the viewer.
    - Change coordinates to this frame.
- Once in viewing coordinates,
  - Usually place a clipping "box" around the scene.
  - Box oriented relative to the viewing frame.
- An **orthographic projection** is made by "removing the $z-$coordinate."
  - Squashes 3D onto 2D, where we can do the window-to-viewport map.
  - The projection of the clipping box is used as the window.

- Mathematically, relative to
$$F_V = (\vec{\imath}, \vec{\jmath}, \vec{k}, \mathcal{O})$$
we map $Q \equiv [q_1, q_2, q_3, 1]^T$ onto
$$F_P = (\vec{u}, \vec{v}, \mathcal{O}')$$
as follows:
$$\text{Ortho}(q_1\vec{\imath} + q_2\vec{\jmath} + q_3\vec{k} + \mathcal{O}) = q_1\vec{u} + q_2\vec{v} + \mathcal{O}'$$
or if we ignore the frames,
$$[q_1, q_2, q_3, 1]^T \mapsto [q_1, q_2, 1]^T$$
- We can write this in matrix form:
$$\begin{bmatrix} q_1 \\ q_2 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} q_1 \\ q_2 \\ q_3 \\ 1 \end{bmatrix}$$
- Question: why would we want to write this in matrix form?

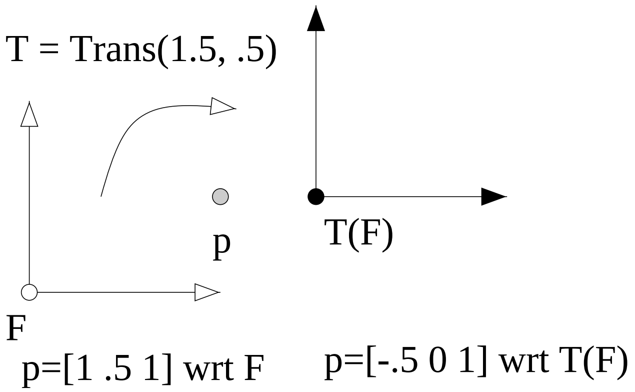
Viewing-World-Modeling Transformations :

- Want to do modeling transformations and viewing transformation (as in Assignment 2).
- If V represents World-View transformation, and M represents modeling transformation, then
$$VM$$
transforms from modeling coordinates to viewing coordinates.

  Note: M is performing both modeling transformation and Model to World change of basis.
- Question: If we transform the viewing frame (relative to viewing frame) how do we adjust V?
- Question: If we transform model (relative to modeling frame) how do we adjust M?

Viewing Transformations:

- Assume all frames are orthonormal
- When we transform the View Frame by T, apply T^{-1} to anything expressed in old view frame coordinates to get new View Frame coordinates



- To compute new World-to-View change of basis, need to express World Frame in new View Frame

  Get this by transforming World Frame elements represented in old View Frame by T^{-1}.
- Recall that the columns of the World-to-View change-of-basis matrix are the basis elements of the World Frame expressed relative to the View Frame.
- If V is old World-to-View change-of-basis matrix, then $T^{-1}V$ will be new World-to-View change-of-basis matrix, since each column of V represents World Frame element, and the corresponding column of $T^{-1}V$ contains T^{-1} of this element.

Modeling Transformations:

- Note that the columns of M are the Model Frame elements expressed relative to the World Frame.
- Want to perform modeling transformation relative to modeling coordinates.
- If we have previously transformed Model Frame, then we next transform relative to transformed Model Frame.
- Example: If

$$M = \begin{bmatrix} x & x' & x'' & x''' \\ y & y' & y'' & y''' \\ z & z' & z'' & z''' \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

  and we translate one unit relative to the first Model Frame basis vector, then we want to translate by $(x, y, z, 0)$ relative to the World Frame.
- Could write this as

$$M' = \begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot M = \begin{bmatrix} x & x' & x'' & x''' + x \\ y & y' & y'' & y''' + y \\ z & z' & z'' & z''' + z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- But this is also equal to

$$M' = M \cdot \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} x & x' & x'' & x''' + x \\ y & y' & y'' & y''' + y \\ z & z' & z'' & z''' + z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- In general, if we want to transform by T our model relative to the current Model Frame, then

$$MT$$

  yields that transformation.
- Translation is easy to do either way, but rotations are much easier the latter way, since rotations around Model Frame basis vectors.
- Summary:
  Modeling transformations embodied in matrix M
  World-to-View change of basis in matrix V
  VM transforms from modeling coordinates to viewing coordinates
  If we further transform the View Frame by T relative to the View Frame, then the new change-of-basis matrix V' is given by

$$V' = T^{-1}V$$

  If we further transform the model by T relative to the modeling frame, the new modeling transformation M' is given by

$$M' = MT$$

- For Assignment 2, need to do further disection of transformations, but this is the basic idea.

♠ *Readings: Hearn and Baker, Section 6-2, first part of Section 12-3; Red book, 6.7; White book, 6.6*

7.11 Transforming Normals

The Truth: Can really only apply affine transforms to points.
Vectors can be transformed correctly iff they are defined by differences of points.

Transforming Normal Vectors:

- Normal vectors **ARE NOT** defined by differences of points
  (formally, they are *covectors*, which are *dual* to vectors).
- Tangent vectors **ARE** defined by differences of points.
- Normals are vectors perpendicular to all tangents at a point:

$$\vec{N} \cdot \vec{T} \equiv \mathbf{n}^T \mathbf{t} = 0.$$

- Note that the natural representation of $\vec{N}$ is as a *row vector*.
- Suppose we have a transformation M, a point $P \equiv \mathbf{p}$, and a tangent $\vec{T} \equiv \mathbf{t}$ at P.

- Let M_ℓ be the "linear part" of M, i.e. the upper 3×3 submatrix.

$$\begin{aligned}
\mathbf{p}' &= M\mathbf{p}, \\
\mathbf{t}' &= M\mathbf{t} \\
&= M_\ell \mathbf{t}. \\
\mathbf{n}^T\mathbf{t} &= \mathbf{n}^T M_\ell^{-1} M_\ell \mathbf{t} \\
&= (M_\ell^{-1T}\mathbf{n})^T (M_\ell \mathbf{t}) \\
&= (\mathbf{n}')^T \mathbf{t}' \\
&\equiv \vec{N}' \cdot \vec{T}'.
\end{aligned}$$

- Transform normals by inverse transpose of linear part of transformation: $\mathbf{n}' = M_\ell^{-1T}\mathbf{n}$.
- If M_T is O.N. (usual case for rigid body transforms), $M_T^{-1T} = M_T$.

Only worry if you have a non-uniform scale or a shear transformation.

Transforming lines: Transform implicit form in a similar way.

Transforming planes: Transform implicit form in a similar way.

♠ *Readings: Red Book, 5.8 (?); White Book, 5.6.*

8 Projections and Projective Transformations

♠

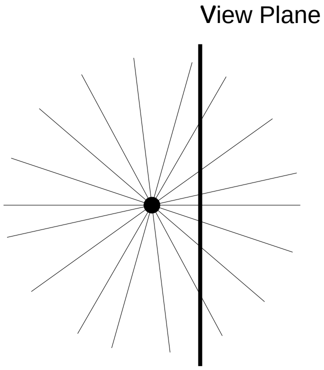
Projective Transformations:

- *Outline*
  - *Properties of projective transformations*
  - *Comparison of projective and affine transformations*
  - *Simple geometry of projection*
  - *Representation as matrix transformation and normalization*
  - *Variations on the matrix*
  - *Mapping z*
  - *Regions and clipping*

8.1 Projections

Perspective Projection

- Identify all points with a line through the eyepoint.
- Slice lines with viewing plane, take intersection point as projection.



- This is **not** an affine transformation, but a **projective transformation**.

Projective Transformations:

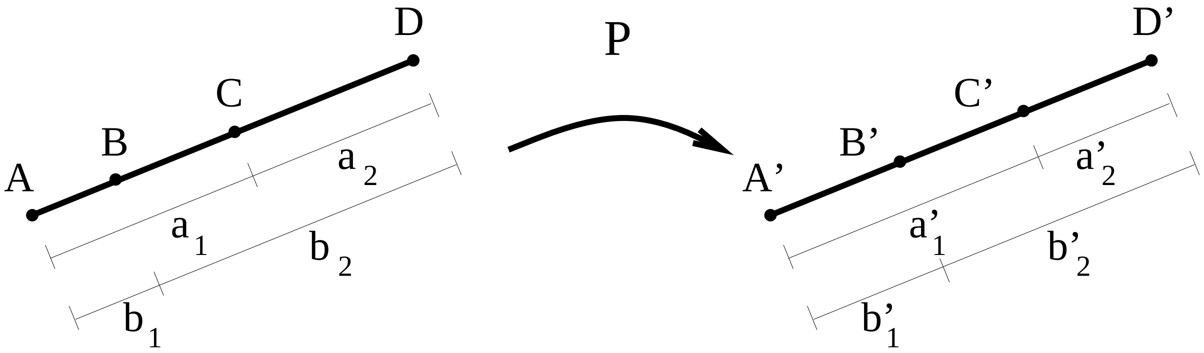
- Angles are not preserved (not preserved under Affine Transformation).
- Distances are not preserved (not preserved under Affine Transformation).
- Ratios of distances are not preserved.
- Affine combinations are not preserved.
- Straight lines are mapped to straight lines.
- *Cross ratios* are preserved.

Cross Ratios

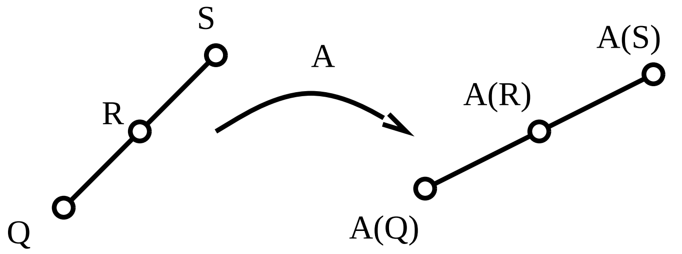
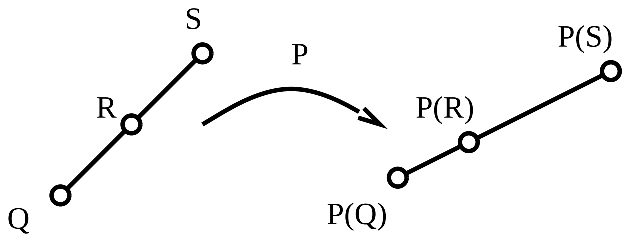
- Cross ratio: $|AC| = a_1$, $|CD| = a_2$, $|AB| = b_1$, $|BD| = b_2$, then

$$\frac{a_1/a_2}{b_1/b_2}\left(=\frac{a_1'/a_2'}{b_1'/b_2'}\right)$$

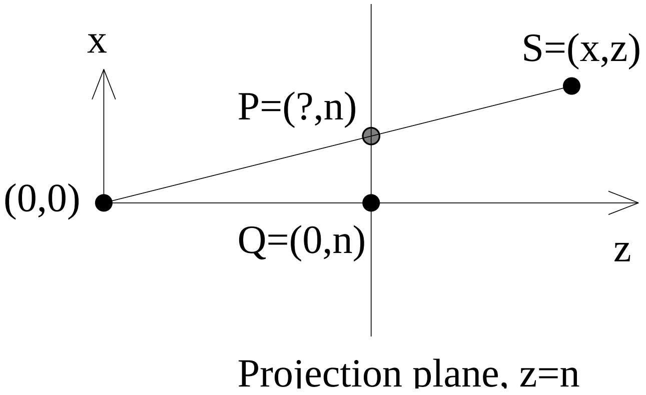
  This can also be used to define a projective transformation (ie, that lines map to lines and cross ratios are preserved).



Comparison:

Affine Transformations	**Projective Transformations**
Image of 2 points on a line determine image of line	Image of 3 points on a line determine image of line
Image of 3 points on a plane determine image of plane	Image of 4 points on a plane determine image of plane
In dimension n space, image of $n+1$ points/vectors defines affine map.	In dimension n space, image of $n+2$ points/vectors defines projective map.
Vectors map to vectors $\vec{v} = Q - R = R - S \Rightarrow$ $A(Q) - A(R) = A(R) - A(S)$ 	Mapping of vector is ill-defined $\vec{v} = Q - R = R - S$ but $P(Q) - P(R) \neq P(R) - P(S)$ 
Can represent with matrix multiply (sort of)	Can represent with matrix multiply and normalization

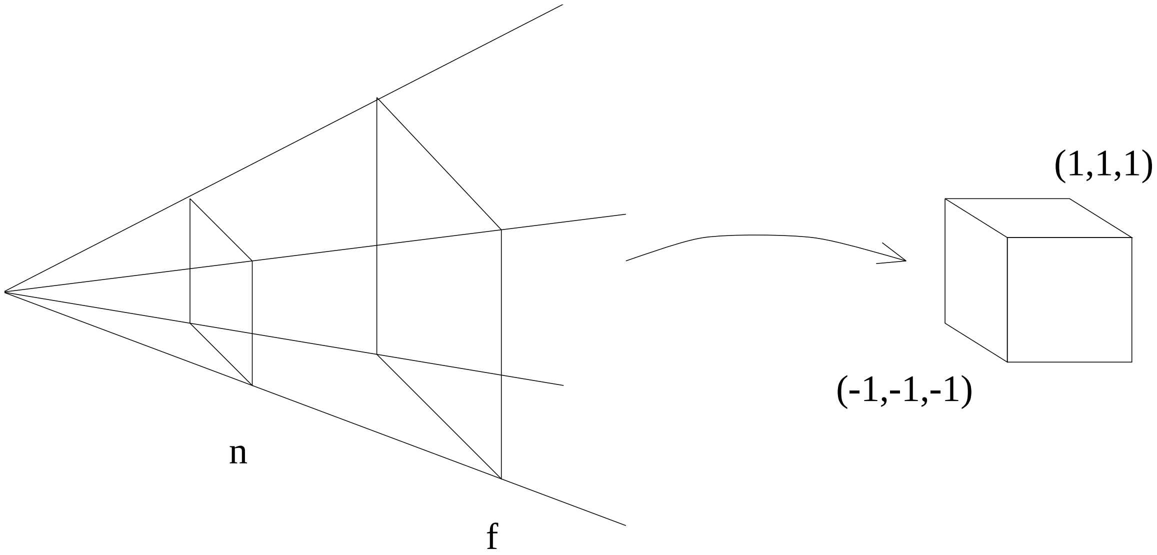
Perspective Map: • Given a point S, we want to find its projection P.



- Similar triangles: $P = (xn/z, n)$
- In 3D, $(x, y, z) \mapsto (xn/z, yn/z, n)$
- Have identified all points on a line through the origin with a point in the projection plane.
- Thus, $(x, y, z) \equiv (kx, ky, kz),\ k \neq 0$.
- These are known as homogeneous coordinates.
- If we have solids or coloured lines,
  then we need to know “which one is in front”.
- This map loses all z information, so it is inadequate.

Pseudo-OpenGL version of the perspective map:

- Maps a near clipping plane $z = n$ to $z' = -1$
- Maps a far clipping plane $z = f$ to $z' = 1$



- The "box" in world space known as "truncated viewing pyramid" or "frustum"
  - Project x, y as before
  - To simplify things, we will project into the $z = 1$ plane.

Derivation:

- Want to map x to x/z (and similarly for y).
- Use a matrix multiply followed by a division (normalization):

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & a & c \\ 0 & 0 & b & d \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ az+c \\ bz+d \end{bmatrix}$$

$$\equiv \begin{bmatrix} \frac{x}{bz+d} \\ \frac{y}{bz+d} \\ \frac{az+c}{bz+d} \\ 1 \end{bmatrix}$$

- Solve for a, b, c, and d such that $z \in [n, f]$ maps to $z' \in [-1, 1]$.
- Know that we want to map x to x/z (assuming our projection plane is at $z = 1$) so

$$\frac{x}{bz+d} = \frac{x}{z} \Rightarrow d = 0 \text{ and } b = 1$$

Thus,

$$\frac{az+c}{bz+d} \text{ becomes } \frac{az+c}{z}$$

- Our constraints on the near and far clipping planes (e.g., that they map to -1 and 1) give us

$$\begin{aligned}
-1 = \frac{an+c}{n} \quad &\Rightarrow \quad c = -n - an \\
1 = \frac{af - n - an}{f} \quad &\Rightarrow \quad f + n = a(f-n) \\
&\Rightarrow \quad a = \frac{f+n}{f-n} \\
&\Rightarrow \quad c = -n - \frac{(f+n)n}{f-n} \\
&\qquad\; = \frac{-n(f-n) - n(f+n)}{f-n} \\
&\qquad\; = \frac{-2fn}{f-n}
\end{aligned}$$

This gives us

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{f+n}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ \frac{-z(f+n)-2fn}{f-n} \\ z \end{bmatrix}$$

- After normalizing we get

$$[\frac{x}{z}, \frac{y}{z}, \frac{z(f+n)-2fn}{z(f-n)}, 1]^T$$

- Could use this formula instead of performing the matrix multiply followed by the division . . .
- If we multiply this matrix in with the geometric transforms, the only additional work is the divide.

Verification:

- If $z = n$, then

$$\begin{bmatrix} x \\ y \\ n \\ 1 \end{bmatrix} \mapsto \begin{bmatrix} x/n \\ y/n \\ -1 \\ 1 \end{bmatrix}$$

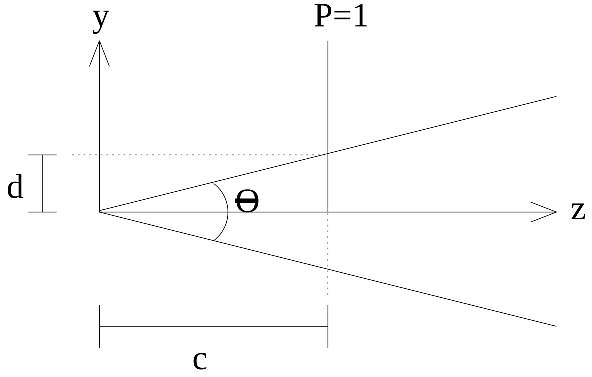
- If $z = f$, then

$$\begin{bmatrix} x \\ y \\ f \\ 1 \end{bmatrix} \mapsto \begin{bmatrix} x/f \\ y/f \\ 1 \\ 1 \end{bmatrix}$$

The OpenGL perspective matrix uses

- $a = -\frac{f+n}{f-n}$ and $b = -1$.

  - OpenGL looks down $z = -1$ rather than $z = 1$.
  - Note that when you specify n and f, they are given as *positive* distances down $z = -1$.
- The upper left entries are very different.
  - OpenGL uses this one matrix to both project and map to NDC.
  - How do we set x or y to map to $[-1, 1]$?
  - We don't want to do both because we may not have square windows.
  - Let's do y:



  - Want to map distance d to 1.
  - $y \mapsto y/z$ is the current projection ...
  - $y \mapsto \frac{y}{zd}$ gives us the scaling.
  - But we have placed our projection plane at $z = 1$, so this is really

$$y \mapsto \frac{yc}{zd}$$

    where $c = 1$.
  - But $c/d = \cot(\frac{\theta}{2})$ so

$$y \mapsto \frac{y}{z}\cot\left(\frac{\theta}{2}\right)$$

- Finally, because the x-y aspect ratio may not be 1, we will scale x to give the desired ratio:

$$\begin{array}{rcl} x & \mapsto & \frac{x}{z} \\ & \mapsto & \frac{x}{z}\cot(\theta/2) \\ & \mapsto & \frac{x}{z}\frac{\cot(\theta/2)}{aspect} \end{array}$$

  where *aspect*$= \Delta x/\Delta y$ of the viewport.

Our final matrix is

$$\begin{bmatrix} \frac{\cot(\theta/2)}{aspect} & 0 & 0 & 0 \\ 0 & \cot(\theta/2) & 0 & 0 \\ 0 & 0 & \pm\frac{f+n}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & \pm 1 & 0 \end{bmatrix}$$

where the ± is 1 if we look down the z axis and -1 if we look down the $-z$ axis.

OpenGL uses a slightly more general form of this matrix that allows skewed viewing pyramids.

♠ *Readings: Watt: 5.2.3, 5.2.4. Hearn and Baker: 12-3. Red book: Chapter 6. White book: Chapter 6. Blinn: Chapter 17, 18.*

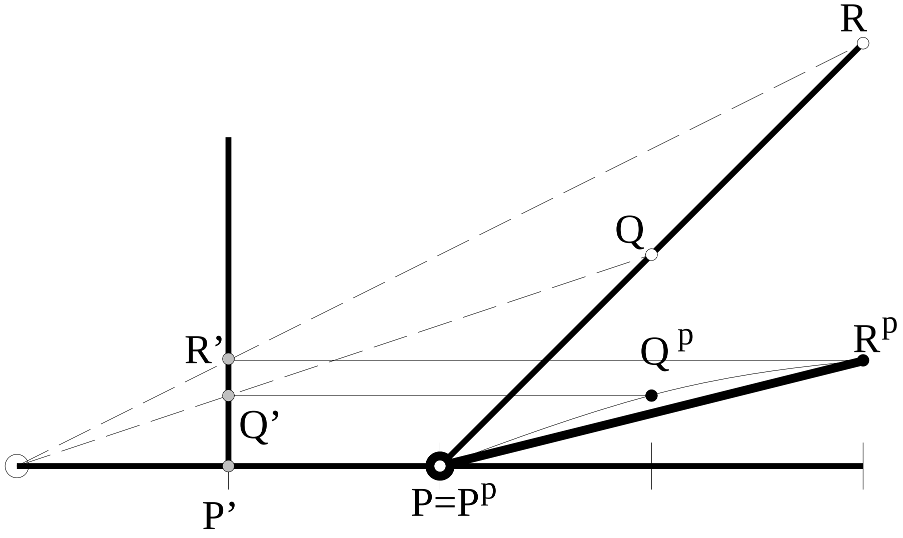
8.2 Why Map Z?

- 3D $\mapsto$ 2D projections map all z to same value.
- Need z to determine occlusion, so a 3D to 2D projective transformation doesn't work.
- Further, we want 3D lines to map to 3D lines (this is useful in hidden surface removal)
- The mapping $(x, y, z, 1) \mapsto (xn/z, yn/z, n, 1)$ maps lines to lines, but loses all depth information.
- We could use

  $$(x, y, z, 1) \quad \mapsto \quad (\frac{xn}{z}, \frac{yn}{z}, z, 1)$$

  Thus, if we map the endpoints of a line segment, these end points will have the same relative depths after this mapping.

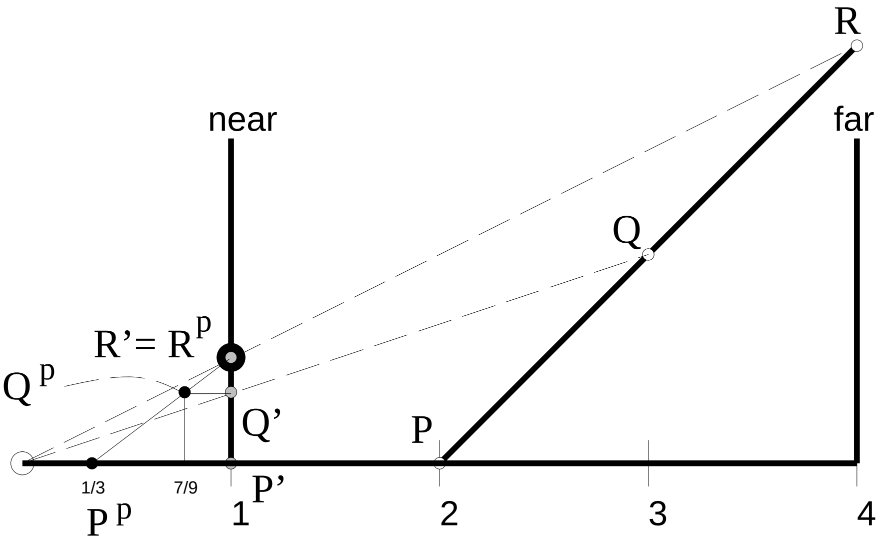
  BUT: It fails to map lines to lines



- In this figure, P, Q, R map to P', Q', R' under a pure projective transformation.
- With the mapping $(x, y, z, 1) \mapsto (\frac{xn}{z}, \frac{yn}{z}, z, 1)$ P, Q, R actually map to P^p, Q^p, R^p, which fail to lie on a straight line.
- The map

  $$(x, y, z, 1) \quad \mapsto \quad (\frac{xn}{z}, \frac{yn}{z}, \frac{zf + zn - 2fn}{z(f - n)}, 1)$$

  does map lines to lines, **and** it preserves depth information.



8.3 Mapping Z

- It’s clear how x and y map. How about z?
- The z map effects: clipping, numerics

$$z \quad \mapsto \quad \frac{zf + zn - 2fn}{z(f-n)} = P(z)$$

- We know $P(f) = 1$ and $P(n) = -1$. What maps to 0?

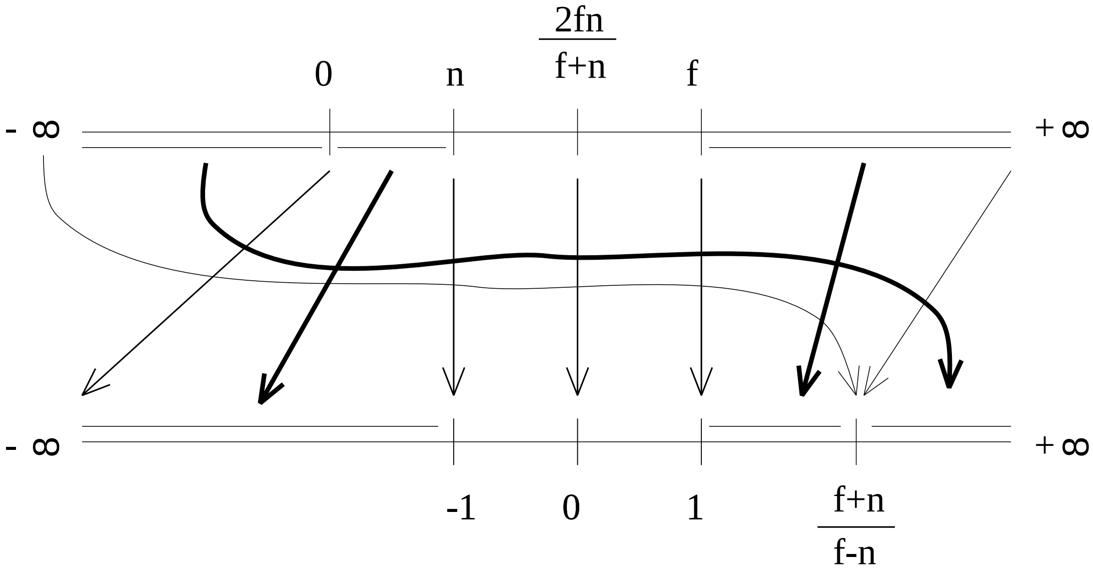
$$\begin{array}{ccl} & P(z) & = 0 \\ \Rightarrow & \frac{zf+zn-2fn}{z(f-n)} & = 0 \\ \Rightarrow & z & = \dfrac{2fn}{f+n} \end{array}$$

Note that $f^2 + fn > 2fn > fn + n^2$ so

$$f > \frac{2fn}{f+n} > n$$

- What happens as z goes to 0 or to infinity?

$$
\begin{aligned}
\lim_{z\to 0^+} P(z) &= \frac{-2fn}{z(f-n)} \\
&= -\infty \\
\lim_{z\to 0^-} P(z) &= \frac{-2fn}{z(f-n)} \\
&= +\infty \\
\lim_{z\to +\infty} P(z) &= \frac{z(f+n)}{z(f-n)} \\
&= \frac{f+n}{f-n} \\
\lim_{z\to -\infty} P(z) &= \frac{z(f+n)}{z(f-n)} \\
&= \frac{f+n}{f-n}
\end{aligned}
$$



- What happens if we vary f and n?

  –

$$
\begin{aligned}
\lim_{f\to n} P(z) &= \frac{z(f+n)-2fn}{z(f-n)} \\
&= \frac{(2zn-2n^2)}{z\cdot 0}
\end{aligned}
$$

  Not surprising, since we're trying to map a single point to a line segment.

  –

$$
\begin{aligned}
\lim_{f\to \infty} P(z) &= \frac{zf-2fn}{zf} \\
&= \frac{z-2n}{z}
\end{aligned}
$$

This means we are mapping an infinite region to $[0, 1]$ and we will effectively get a far plane due to floating point precision.

–

$$\begin{aligned}\lim_{n->0} P(z) &= \frac{zf}{zf} \\ &= 1\end{aligned}$$

I.e., the entire map becomes constant (again, we are mapping a point to an interval).

- What happens as f and n move away from each other.

  Look at size of the regions $[n, 2fn/(f + n)]$ and $[2fn/(f + n), f]$.

  When f is large compared to n, we have

  $$\frac{2fn}{f+n} \doteq 2n$$

  So

  $$\frac{2fn}{f+n} - n \doteq n$$

  and

  $$f - \frac{2fn}{f+n} \doteq f - 2n.$$

  But both intervals are mapped to a region of size 1.

  Thus, as we move the clipping planes away from one another, the far interval is compressed more than the near one. With floating point arithmetic, this means we'll lose precision.

  In the extreme case, think about what happens as we move f to inifinity: we compress an infinite region to a finite one.

  Therefore, we try to place our clipping planes as close to one another as we can.

8.4 3D Clipping

- When do we clip in 3D? We should clip to the near plane *before* we project. Otherwise, we might map z to 0 and then x/z and y/z are undefined.
- We could clip to all 6 sides of the truncated viewing pyramid. But the plane equations are simpler if we clip after projection, because all sides of volume are parallel to coordinate plane.
- Clipping to a plane in 3D is identical to clipping to a line in 2D.
- We can also clip in homogenous coordinates.

♠ *Readings: Red Book, 6.6.4; White book, 6.5.4.*

8.5 Homogeneous Clipping

Projection: linear transformations then normalize

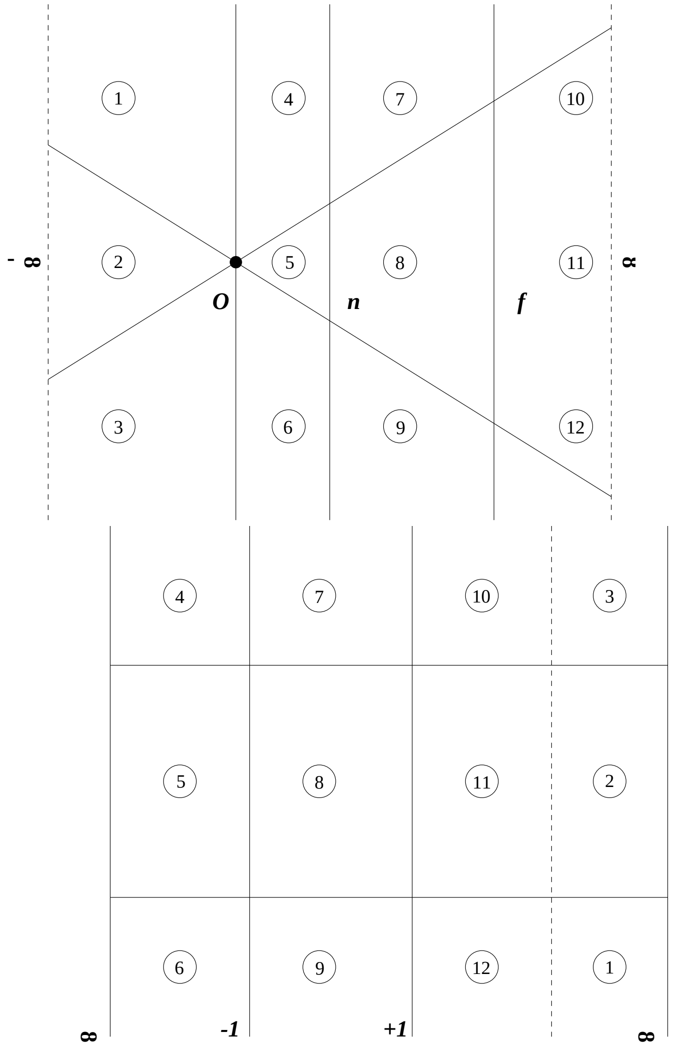
- Linear transformation

$$\begin{bmatrix} nr & 0 & 0 & 0 \\ 0 & ns & 0 & 0 \\ 0 & 0 & \frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \bar{x} \\ \bar{y} \\ \bar{z} \\ \bar{w} \end{bmatrix}$$

- Normalization

$$\begin{bmatrix} \bar{x} \\ \bar{y} \\ \bar{z} \\ \bar{w} \end{bmatrix} = \begin{bmatrix} \bar{x}/\bar{w} \\ \bar{y}/\bar{w} \\ \bar{z}/\bar{w} \\ 1 \end{bmatrix} = \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Region mapping:



Clipping not good after normalization:

- Ambiguity after normalization

$$-1 \leq \frac{\bar{x}, \bar{y}, \bar{z}}{\bar{w}} \leq +1$$

  - Numerator can be positive or negative
  - Denominator can be positive or negative
- Normalization expended on points that are subsequently clipped

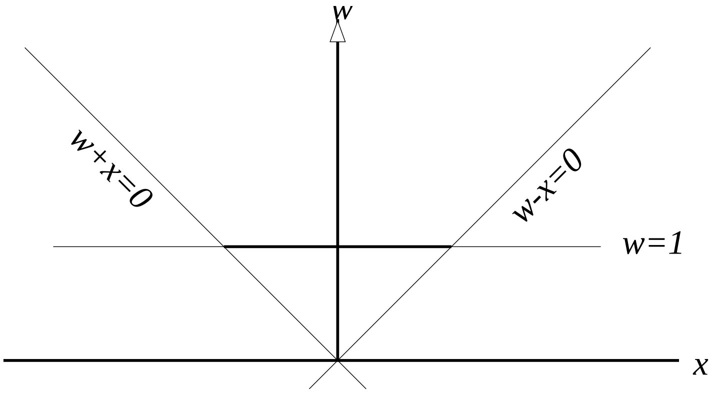
Clip in homogeneous coordinates:

- Compare unnormalized coordinate against $\bar{w}$

$$-|\bar{w}| \leq \bar{x}, \bar{y}, \bar{z} \leq +|\bar{w}|$$

Clipping Homogeneous Coordinates

- Assume NDC window of $[-1, 1] \times [-1, 1]$
- To clip to $X = -1$ (left):
  - Projected coordinates: Clip to $X = -1$
  - Homogeneous coordinate: Clip to $\bar{x}/\bar{w} = -1$
  - Homogeneous plane: $\bar{w} + \bar{x} = 0$



  - Point is visible if $\bar{w} + \bar{x} > 0$
  - For line segment $\overline{P_1 P_2}$ want a such that

$$P = (1 - a)P_1 + aP_2$$

    lies on the plane $w + x = 0$
  - Solving for a

$$[(1 - a)w_1 + aw_2] + [(1 - a)x_1 + ax_2] = 0$$

    gives us

$$a = \frac{w_1 + x_1}{(w_1 + x_1) - (w_2 + x_2)}$$

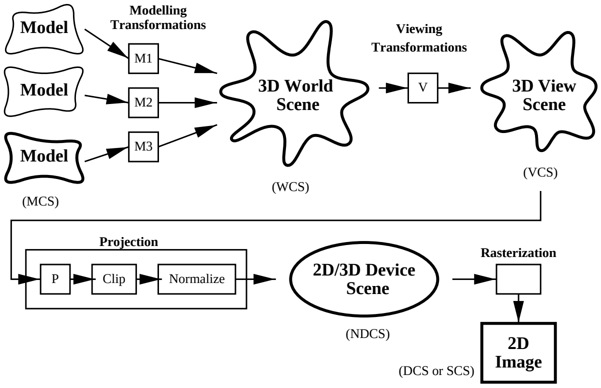
- Repeat for remaining boundaries:
  - $X = \bar{x}/\bar{w} = 1$

- $Y = \bar{y}/\bar{w} = -1$
- $Y = \bar{y}/\bar{w} = 1$
- Near and far clipping planes

♠ *Readings: Jim Blinn's Corner, Chapters 13, 18*

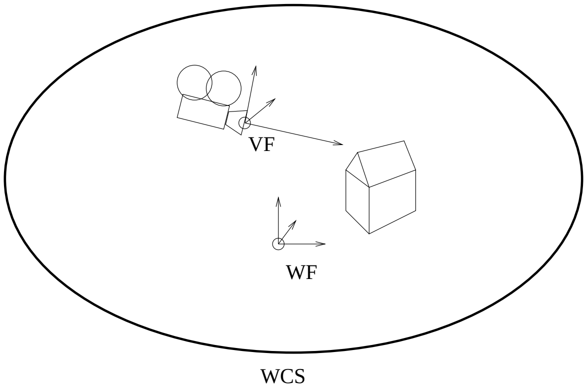
9 Transformation Applications and Extensions

9.1 Rendering Pipeline Revisited



Composition of Transforms: $\mathbf{p}' \equiv PVM_i\mathbf{p}$.

9.2 Viewing Transformations



- WCS is supported by $WF = (\mathcal{O}_W, \vec{w}_x, \vec{w}_y, \vec{w}_z)$, the **World Frame**.
- WF is usually orthonormal, i.e. $(\mathcal{O}, \vec{e}_x, \vec{e}_y, \vec{e}_z)$, the Cartesian Standard Frame.
- VCS is supported by $VF = (\mathcal{O}_V, \vec{v}_x, \vec{v}_y, \vec{v}_z)$. the **Viewing Frame**.
- Express VF relative to the WCS: $VF_{WCS} = (O_V, \mathbf{v}_x, \mathbf{v}_y, \mathbf{v}_z)$.
- Use Change of Basis to derive V, the **Viewing Transformation**.

- If VF was an orthonormal frame in WCS, then

$$V = \begin{bmatrix} v_{1x} & v_{2x} & v_{3x} & 0 \\ v_{1y} & v_{2y} & v_{3y} & 0 \\ v_{1z} & v_{2z} & v_{3z} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \circ Tr(-O_V).$$

- Note:

$$\begin{aligned} V\mathbf{v}_x &= \mathbf{e}_x \\ V\mathbf{v}_y &= \mathbf{e}_y \\ V\mathbf{v}_z &= \mathbf{e}_z \\ VO_V &= 0 \end{aligned}$$

- Not usually given an O.N. frame for VF; might have
  - Position of camera C.
  - Direction of view of camera $\vec{v}$ (*or* lookat point L, $\vec{v} = L - C$).
  - Up vector $\vec{u}$.
- Convert C, $\vec{v}$, and $\vec{u}$ to left-handed O.N. frame:

$$\begin{aligned} \mathcal{O}_V &= C, \\ \vec{V} &= \frac{\vec{v}}{|\vec{v}|}, \\ \vec{u}' &= \vec{u} - (\vec{u} \cdot \vec{V})\vec{V}, \\ \vec{U} &= \frac{\vec{u}'}{|\vec{u}'|}, \\ \vec{W} &= \vec{u} \times \vec{v}. \end{aligned}$$

- For a right-handed frame, use $\vec{W} = \vec{v} \times \vec{u}$.
- Slight problem if $\vec{u}$ is parallel to $\vec{v}$ ...

9.3 3D Rotation Specification

9.3.1 Derivation by Composition

- Can derive the matrix for angle-axis rotation by composing basic transformations.
- Rotation given by $\vec{a} = (x, y, z)$ and θ.
- Assume that $|\vec{a}| = 1$.
- General idea: Map $\vec{a}$ onto one of the canonical axes, rotate by θ, map back.

1. Pick the closest axis to $\vec{a}$ using $\max_i \vec{e}_i \cdot \vec{a} = \max(x, y, z)$.
   (Assume we chose the x-axis in the following).

2. Project $\vec{a}$ onto $\vec{b}$ in the xz plane:

$$\begin{aligned} \vec{b} &= (x, 0, z). \end{aligned}$$

3. Compute $\cos(\phi)$ and $\sin(\phi)$, where ϕ is the angle of $\vec{b}$ with the x-axis.

$$\begin{aligned} \cos(\phi) &= \frac{x}{\sqrt{x^2+z^2}}, \\ \sin(\phi) &= \frac{z}{\sqrt{x^2+z^2}}. \end{aligned}$$

4. Use $\cos(\phi)$ and $\sin(\phi)$ to create $R_y(-\phi)$:

$$R_y(-\phi) = \begin{bmatrix} \cos(-\phi) & 0 & -\sin(-\phi) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(-\phi) & 0 & \cos(-\phi) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

5. Rotate $\vec{a}$ onto the xy plane using $R_y(-\phi)$:

$$\begin{aligned} \vec{c} &= R_y(-\phi)\vec{a} \\ &= \left(\sqrt{x^2+z^2}, y, 0\right). \end{aligned}$$

6. Compute $\cos(\psi)$ and $\sin(\psi)$, where ψ is the angle of $\vec{c}$ with the x-axis.

$$\begin{aligned} \cos(\psi) &= \frac{\sqrt{x^2+z^2}}{\sqrt{x^2+y^2+z^2}} \\ &= \sqrt{x^2+z^2}, \\ \sin(\psi) &= \frac{y}{\sqrt{x^2+y^2+z^2}} \\ &= y. \end{aligned}$$

7. Use $\cos(\psi)$ and $\sin(\psi)$ to create $R_z(-\psi)$:

$$R_z(-\psi) = \begin{bmatrix} \cos(-\psi) & -\sin(-\psi) & 0 & 0 \\ \sin(-\psi) & \cos(-\psi) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

8. Rotate $\vec{c}$ onto the x axis using $R_z(-\psi)$.

9. Rotate about the x-axis by θ: $R_x(-\theta)$.

10. Reverse z-axis rotation: $R_z(\psi)$.

11. Reverse y-axis rotation: $R_y(\phi)$.

The overall transformation is

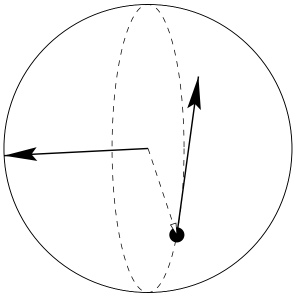
$$R(\theta, \vec{a}) = R_y(\phi) \circ R_z(\psi) \circ R_x(\theta) \circ R_z(-\psi) \circ R_y(-\phi).$$

9.3.2 3D Rotation User Interfaces

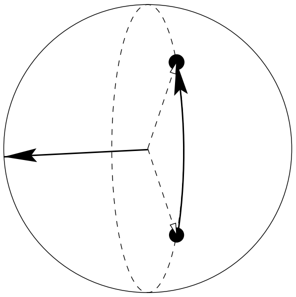
Goal: Want to specify angle-axis rotation "directly".

Problem: May only have mouse, which only has two degrees of freedom.

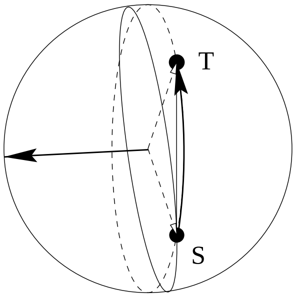
Solutions: Virtual Sphere, Arcball.



Virtual Sphere

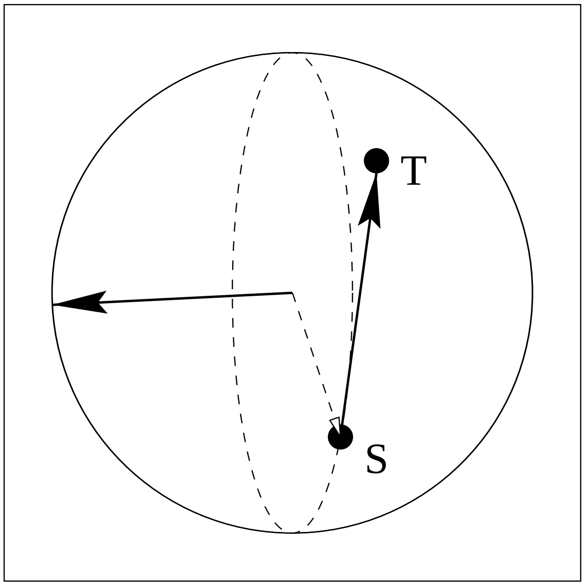


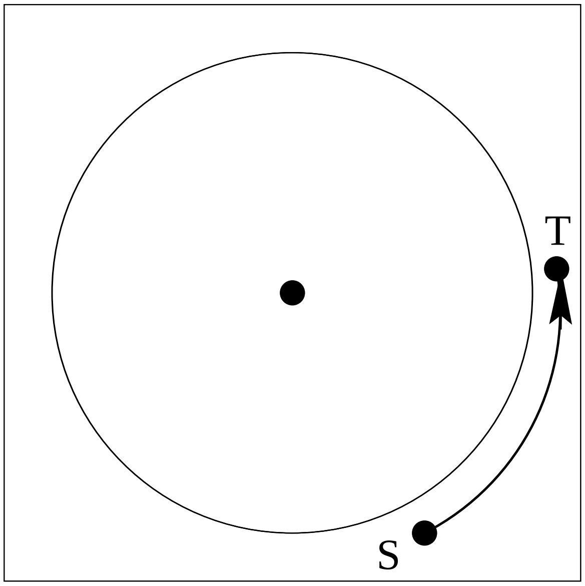
Arcball



Comparison

9.3.3 The Virtual Sphere





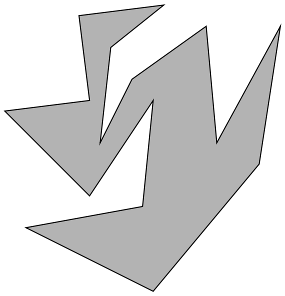
1. Define portion of screen to be projection of virtual sphere.
2. Get two sequential samples of mouse position, S and T.
3. Map 2D point S to 3D unit vector $\vec{p}$ on sphere.
4. Map 2D vector $\vec{ST}$ to 3D tangental velocity $\vec{d}$.
5. Normalize $\vec{d}$.

6. Axis: $\vec{a} = \vec{p} \times \vec{d}$.
7. Angle: $\theta = \alpha|\vec{ST}|$.
   (Choose α so a 180° rotation can be obtained.)
8. Save T to use as S for next time.

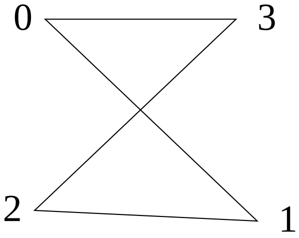
10 Polygons

10.1 Polygons

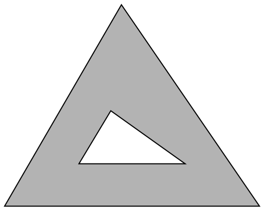
- Need an area primitive
- Simple polygon:
  - Planar set of ordered points, $v_0, \ldots, v_{n-1}$
    (sometimes we repeat v_0 at end of list)
  - No holes
  - No line crossing



OK



Line Crossing

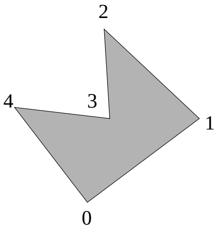


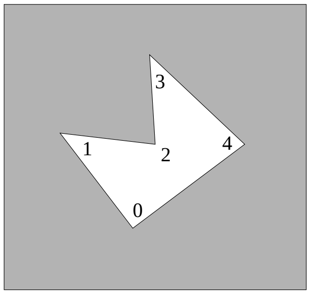
Hole

- Normally define an interior and exterior

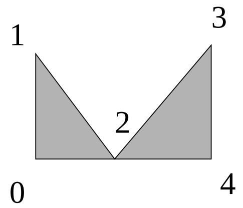
  Points ordered in counter-clockwise order

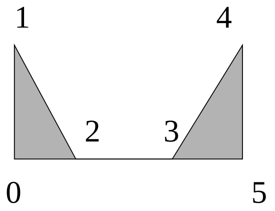
  To the "left" as we traverse is inside

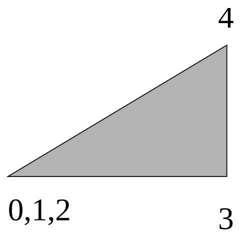


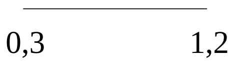


- Try to avoid degeneracies, but sometimes unavoidable

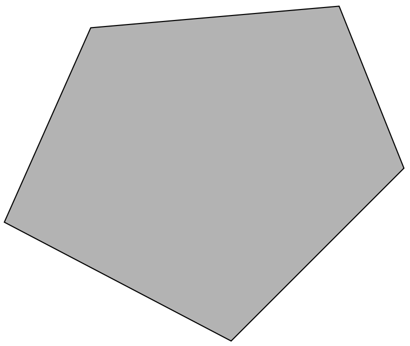
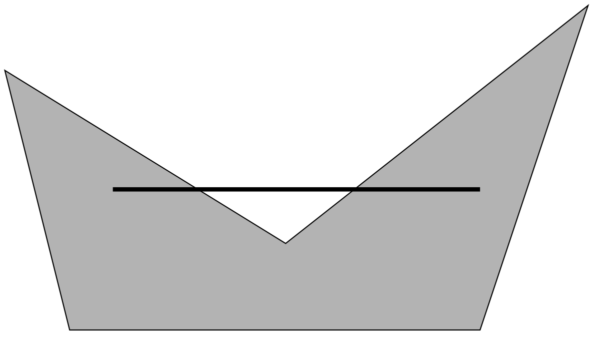








- We prefer *Convex* polygons to *Concave* polygons

  Polygon is convex if for any two points inside polygon, the line segment joining these two points is also inside.

  Convex polygons "behave" better when shading, etc

- Although convex polygons normally "well behaved" under affine transformation, affine transformations may introduce degeneracies

  Example: Orthographic projection may project entire polygon to a line segment.

10.2 Polygon Clipping

Polygon Clipping (Sutherland-Hodgman):

- Window must be a convex polygon
- Polygon to be clipped can be convex or not

Approach:

- Polygon to be clipped is given as $v_1, \ldots, v_n$
- Each polygon edge is a pair $[v_i, v_{i+1}]$ $i = 1, \ldots, n$
  - Don't forget wraparound; $[v_n, v_1]$ is also an edge
- Process all polygon edges in succession against a window edge
  - Polygon in – polygon out

  - $v_1, \ldots, v_n \rightarrow w_1, \ldots, w_m$
- Repeat on resulting polygon with next sequential window edge

Contrast with Line Clipping:

- **Line Clipping:**
  - Use outcodes to check all window edges before any clip
  - Clip only against possibly intersecting window edges
  - Deal with window edges in any order
  - Deal with line segment endpoints in either order
- **Polygon Clipping:**
  - Each window edge must be used
  - Polygon edges must be handled in sequence
  - Polygon edge endpoints have a given order
  - Stripped-down line-segment/window-edge clip is a subtask

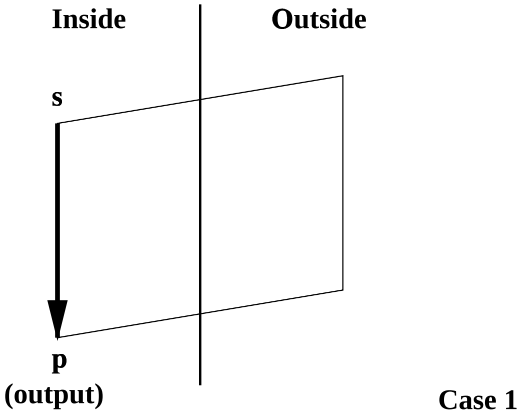
There are four cases to consider

Four Cases:

- $\mathbf{s} = v_i$ is the polygon edge starting vertex
- $\mathbf{p} = v_{i+1}$ is the polygon edge ending vertex
- $\mathbf{i}$ is a polygon-edge/window-edge intersection point
- w_j is the next polygon vertex to be output

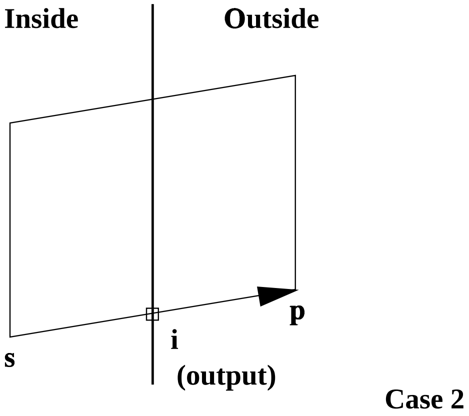
Case 1: Polygon edge is entirely inside the window edge

- $\mathbf{p}$ is next vertex of resulting polygon
- $\mathbf{p} \rightarrow w_j$ and $j + 1 \rightarrow j$



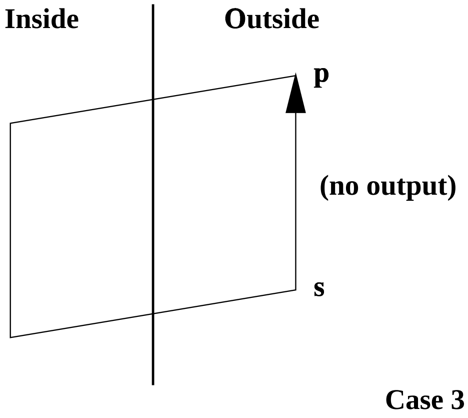
Case 2: Polygon edge crosses window edge going out

- Intersection point **i** is next vertex of resulting polygon
- **i** $\rightarrow w_j$ and $j+1 \rightarrow j$



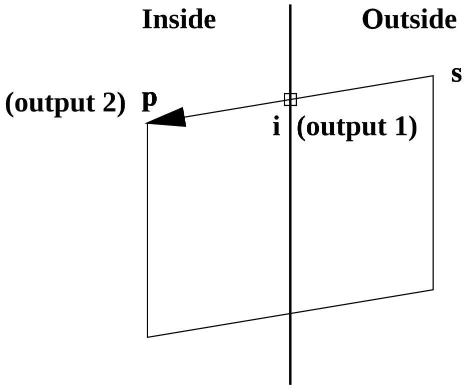
Case 3: Polygon edge is entirely outside the window edge

- No output



Case 4: Polygon edge crosses window edge going in

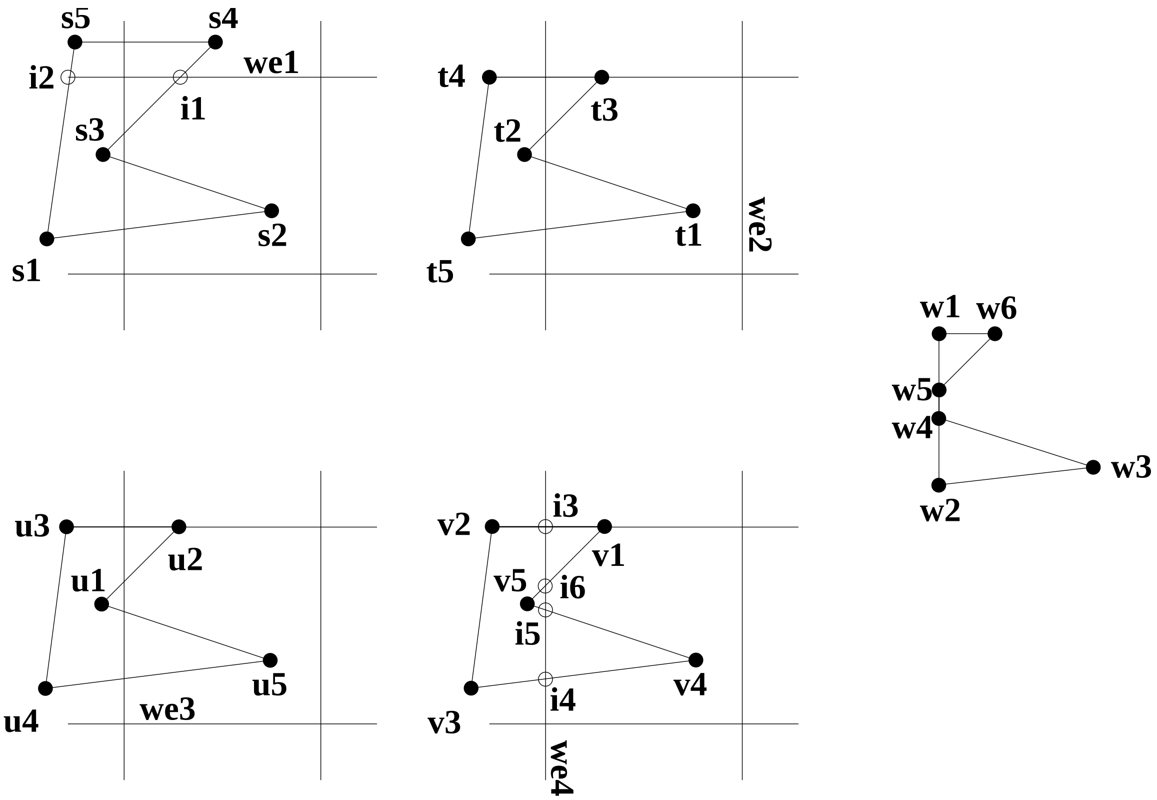
- Intersection point **i** and **p** are next two vertices of resulting polygon
- **i** $\rightarrow w_j$ and **p** $\rightarrow w_{j+1}$ and $j+2 \rightarrow j$



Case 4

Simple Examples: ...

An Example: With a non-convex polygon...

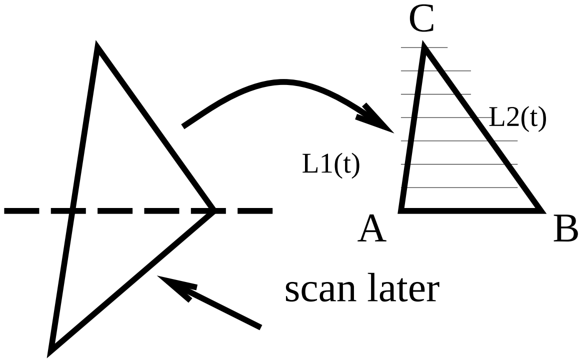


♠ *Readings: Watt: 6.1. Hearn and Baker: Section 6-8. Red book: 3.11. White book: 3.14.*

10.3 Polygon Scan Conversion

Scan Conversion

- Once we have mapped the vertices of polygon to device coordinates, we want to scan convert it.
- Scan converting a general polygon is complicated.
- Here we will look at scan conversion of a triangle.
- Look at y value of vertices. Split triangle along horizontal line at middle y value.



- Step along $L1$ and $L2$ together along the scan lines from A to C and from B to C respectively.

- Scan convert each horizontal line.

Code for triangle scan conversion

- Assume that triangle has been split and that A, B, C are in device coordinates, that $A.x < B.x$, and that $A.y = B.y \neq C.y$.

```
y = A.y;
d0 = (C.x-A.x)/(C.y-A.y);
d1 = (C.x-B.x)/(C.y-B.y);
x0 = A.x;
x1 = B.x;
while ( y <= C.y ) do
   for ( x = x0 to x1 ) do
       WritePixel(x,y);
   end
   x0 += d0;
   x1 += d1;
   y++;
end
```

- Note: This is a floating point algorithm

♠ *Readings: Watt: 6.4. Hearn and Baker: Chapter 3-11. Red book: 3.5 (more general than above). White book: 3.6 (more general than above).*

10.4 Do's and Don'ts

When modelling with polygonal objects, remember the following guidelines:

- Model Solid Objects

  No facades

  It's easier to write other software if we can assume polygon faces are oriented

- No T-vertices

  These cause shading anomolies and pixel dropout

- No overlapping co-planar faces

  Violates solid object contraint.

  If different colours, then psychedelic effect due to floating point roundoff

- Well shaped polygon

  Equilateral triangle is best.

  Long, skinny triangles pose problems for numerical algorithms

11 Hidden Surface Removal

11.1 Hidden Surface Removal

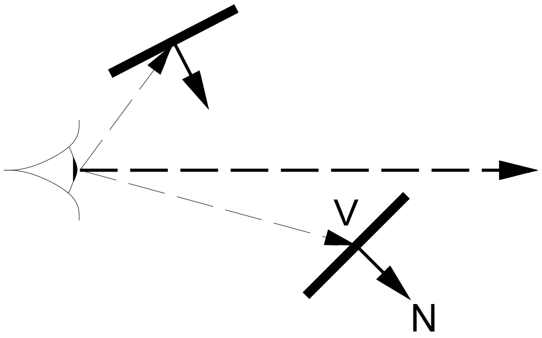
- When drawing lots of polygons, we want to draw only those "visible" to viewer.
- There are a variety of algorithms with different strong points.
- Issues:
  - Online
  - Device independent
  - Fast
  - Memory requirements
  - Easy to implement in hardware

♠ *Readings: Watt: 6.6. Red book: 13. White book: 15. Hearn and Baker: Chapter 13.*

11.2 Backface Culling

Backface Culling

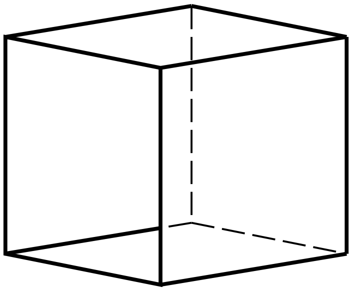
- A simple way to perform hidden surface is to remove all "backfacing" polygons.
- The observation is that if polygon normal is facing away from the viewer then it is "backfacing."
- For solid objects, this means the polygon will not be seen by the viewer.



- Thus, if $N \cdot V > 0$, then cull polygon.
- Note that V is vector from eye to point on polygon
  You cannot use the view direction for this.

Backface Culling Not a complete solution

- If objects not convex, need to do more work.
- If polygons two sided (i.e., they do not enclose a volume) then we can't use it.

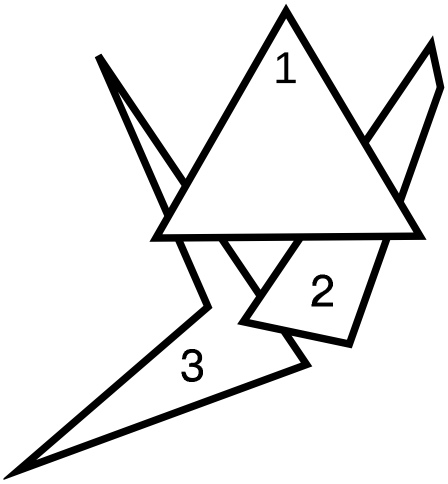


- A HUGE speed advantage if we can use it since the test is cheap and we expect at least half the polygons will be discarded.
- Usually performed in conjunction with a more complete hidden surface algorithm.
- Easy to integrate into hardware (and usually improves performance by a factor of 2).

11.3 Painter's Algorithm

Painter's Algorithm

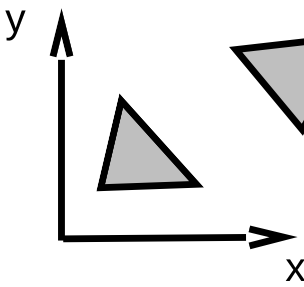
- Idea: Draw polygons as an oil painter might: The farthest one first.
  - Sort polygons on farthest z
  - Resolve ambiguities where z's overlap
  - Scan convert from largest z to smallest z

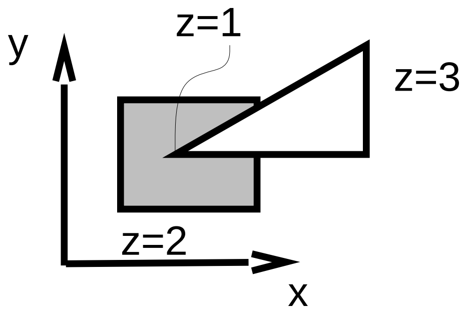


- Since closest drawn last, it will be on top (and therefore it will be seen).
- Need all polygons at once in order to sort.

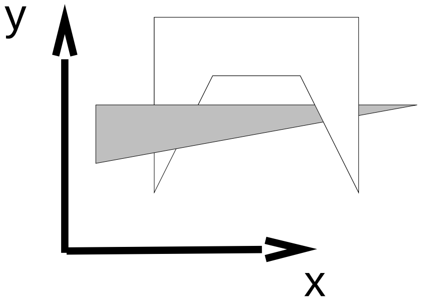
Painter's Algorithm Z overlap

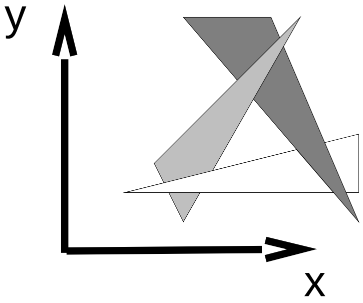
- Some cases are easy:





- But other cases are nasty!





  (have to split polygons)
- $\Omega(n^2)$ algorithm
- Lots of subtle detail

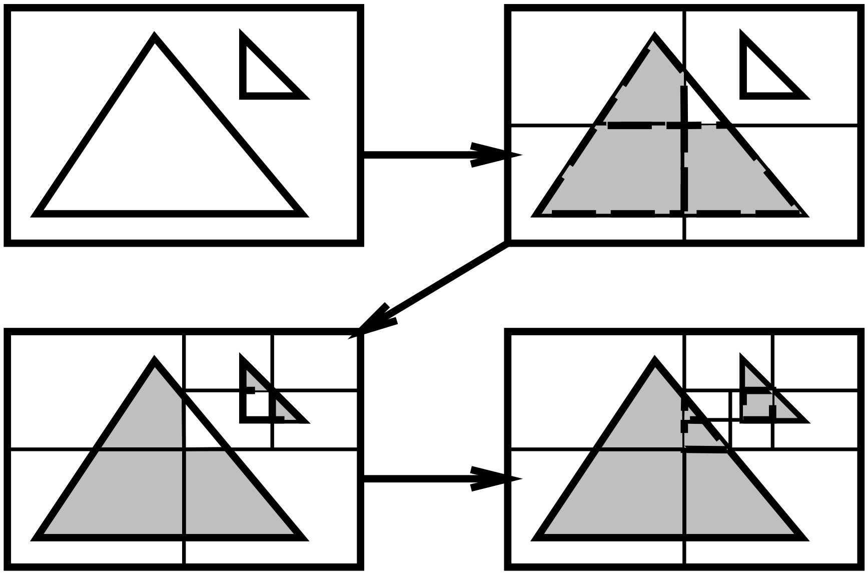
11.4 Warnock's Algorithm

- A divide and conquer algorithm

```
Warnock(PolyList PL, ViewPort VP)
If ( PL simple in VP) then
    Draw PL in VP
else
    Split VP vertically and horizontally into VP1,VP2,VP3,VP4
    Warnock(PL in VP1, VP1)
    Warnock(PL in VP2, VP2)
    Warnock(PL in VP3, VP3)
    Warnock(PL in VP4, VP4)
end
```

- What does "simple" mean?
  - No more than one polygon in viewport
    Scan convert polygon clipped to viewport
  - Viewport only 1 pixel in size
    Shade pixel based on closest polygon in the pixel

Warnock's Algorithm



- Runtime: $O(p \times n)$
  - p: number of pixels
  - n: number of polygons

11.5 Z-Buffer Algorithm

- Perspective transformation maps viewing pyramid to viewing box in a manner that maps lines to lines
- This transformation also maps polygons to polygons
- Idea: When we scan convert, step in z as well as x and y.
- In addition to framebuffer, we'll have a depth buffer (or z buffer) where we write z values
- Initially, z buffer values set to ∞

  Depth of far clipping plane (usually 1) will also suffice
- Scan convert using the following WritePixel:

```
WritePixel(int x, int y, float z, colour)
if ( z < zbuf[x][y] ) then
    zbuf[x][y] = z;
    frambuffer[x][y] = colour;
end
```

- Runtime: $O(p_c + n)$
  - p_c: number of scan converted pixels

  - n: number of polygons

Z-buffer in Hardware

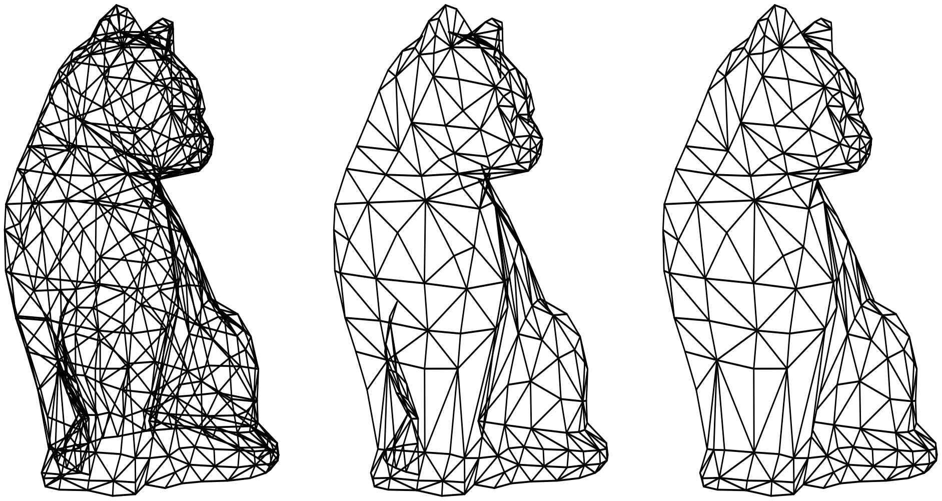
Z-buffer is the algorithm of choice for hardware implementation

+ Easy to implement
+ Simple hardware implementation
+ Online algorithm (i.e., we don't need to load all polygons at once in order to run algorithm)
- Doubles memory requirements (at least)
  But memory is cheap!
- Scale/device dependent

11.6 Comparison of Algorithms

- Backface culling fast, but insufficient by itself
- Painter's algorithm device independent, but details tough, and algorithm is slow
- Warnock's algorithm easy to implement, but not very fast, and is semi-device dependent.
- Z-buffer online, fast, and easy to implement, but device dependent and memory intensive. Algorithm of choice for hardware

Comparison of no hidden surface removal, backface culling, and hidden surface removal:



12 Hierarchical Models and Transformations

12.1 Hierarchical Transformations

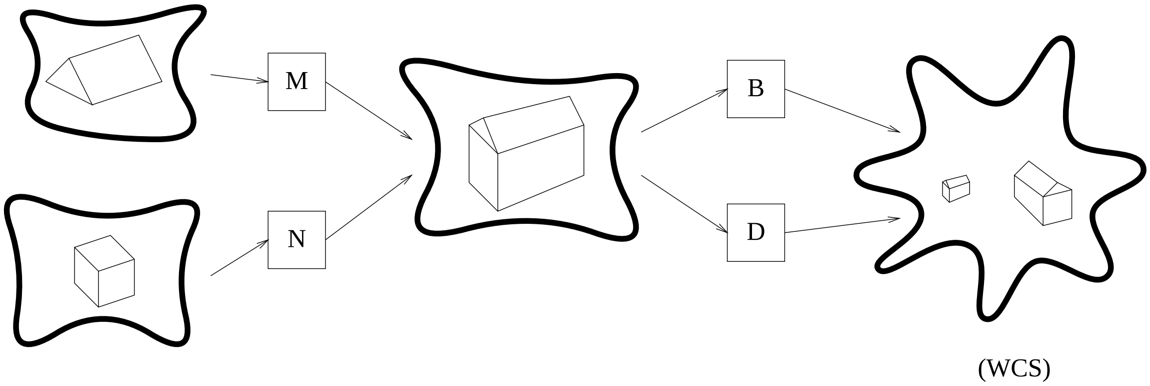
How do we model complex objects and scenes?

- Describing everything as a single complex model is a Bad Idea.
- Use hierarchical modeling instead...

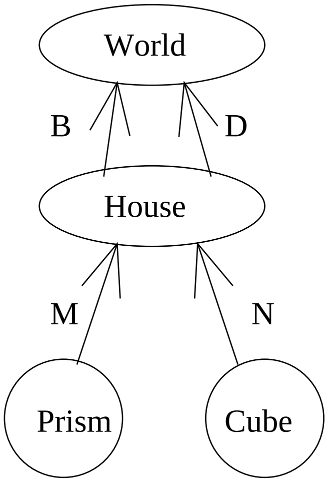
Start with basic set of 3D primitives: Cubes, spheres, prisms ...

- Each is defined in a “nice” way in its own space.
- To put primitives together, use transformations.
- Each transformation embeds one space in another.
- Use a whole hierarchy of spaces to build up complex models. . .
- Not just one “Model space”, but one for each model and part of a model.

Suppose we want two houses.



Pictorially we have a DAG —a directed acyclic graph.

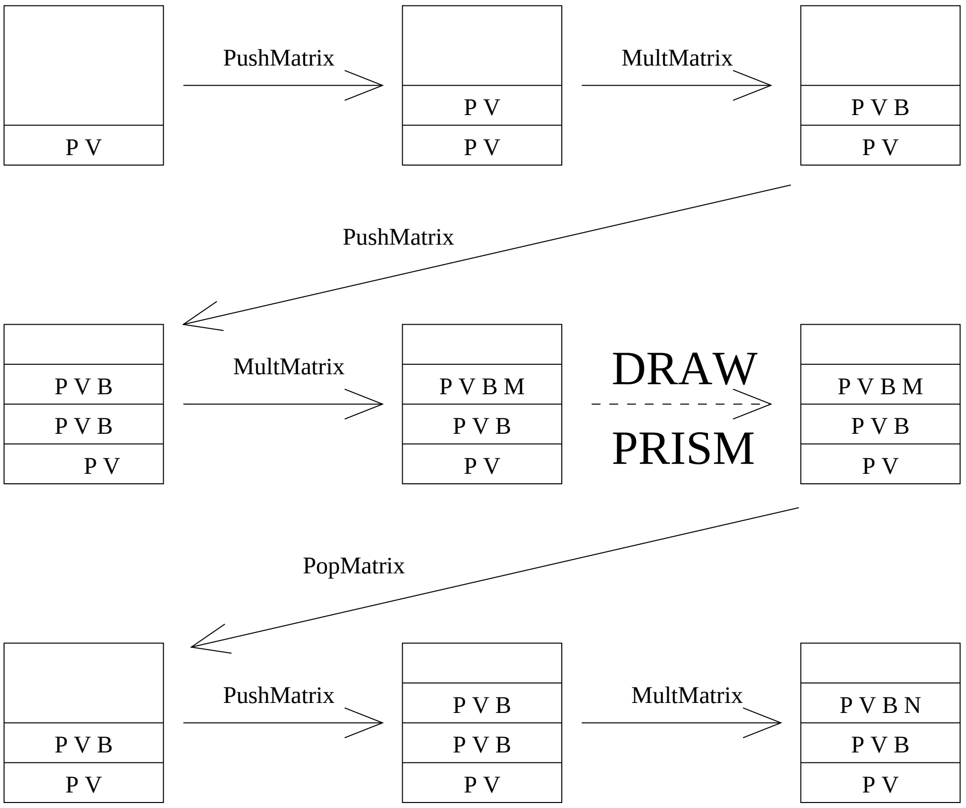


We can model this procedurally:

```
Procedure Scene()
    House(B);
    House(D);
end

Procedure House(E)
    Prism(E o M);
    Cube(E o N);
end

Procedure Cube(F)
    . . .
end

Procedure Prism(F)
    . . .
end
```

Implementation:

- Procedure calls are making depth first traversal of tree/DAG.
- Use a **matrix stack**—sometimes maintained in H/W.
- Each time we make an embedding,
  1. "Push" the new transformation onto the matrix stack
  2. "Pop" it on return.
- OpenGL's `glPushMatrix` call duplicates the top of the stack.
- OpenGL's `glMultMatrix` multiplies a matrix with the top of the stack, replacing the top of the stack with the result.
- OpenGL's `glPopMatrix` pops the stack.
- Stack starts with identity matrix on bottom.
- OpenGL also has transformation calls, such as `glRotate`, `glScale`, etc that perform basic transformations.
- These transformations operate on the top of the stack.
- Put perspective and world-to-view matrices into the stack.
- These are pushed on first, giving PV on the bottom.
- Might have more than one stack, i.e. MODEL, VIEW, and PROJECTION stacks.



• • •

Code now looks like

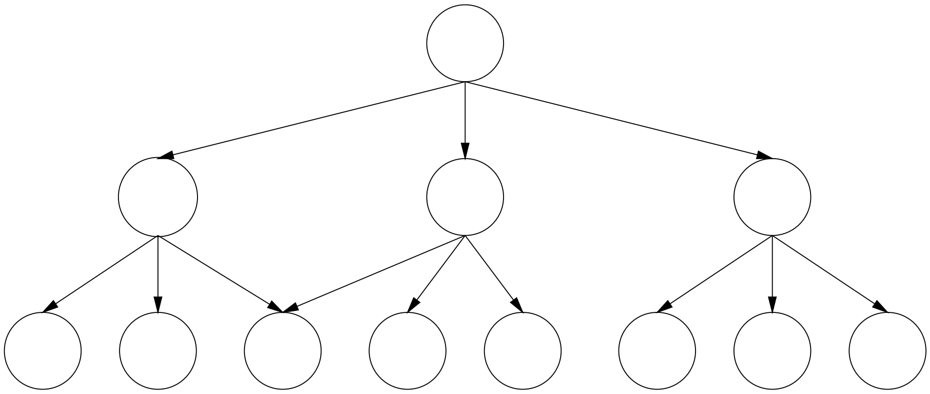
```
Procedure Scene()
    MultMatrix(P);
    MultMatrix(V);
    PushMatrix();
        MultMatrix(B);
        House();
    PopMatrix();
    PushMatrix();
        MultMatrix(D);
        House();
    PopMatrix();
end

Procedure House()
    PushMatrix();
        MultMatrix(M);
        Prism();
    Popmatrix();
    PushMatrix();
```

```
        MultMatrix(N);
        Cube();
    Popmatrix();
end

Procedure Cube()
    . . .
end

Procedure Prism()
    . . .
end
```

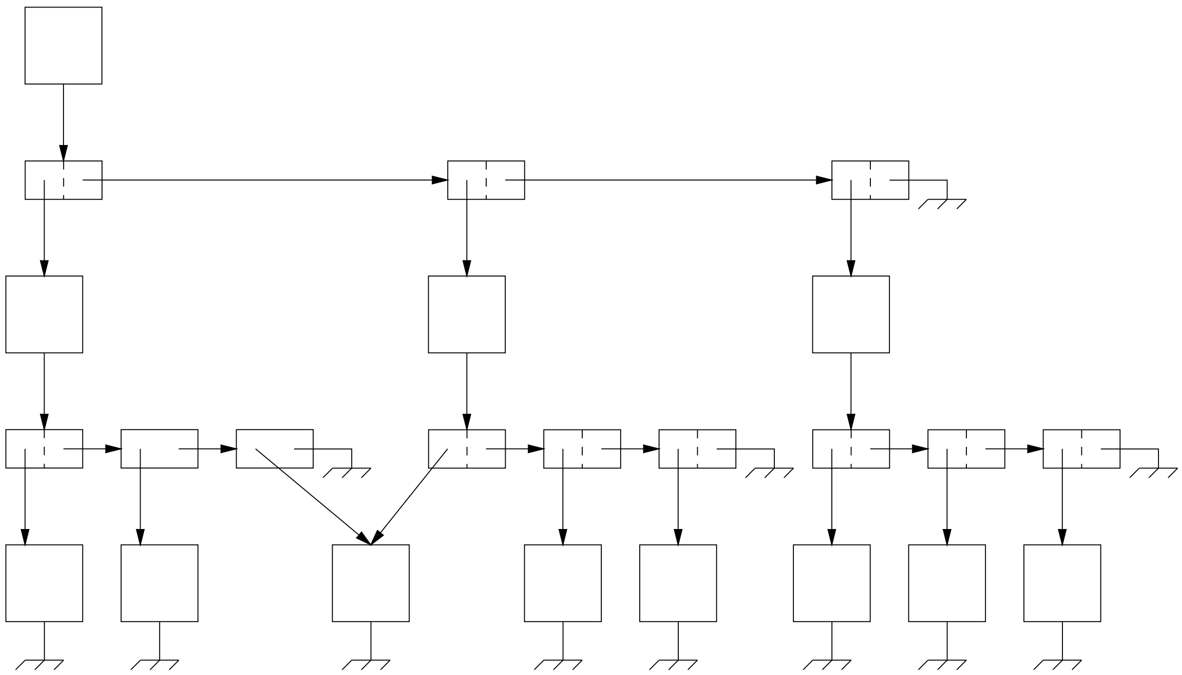
♠ *Readings: Hearn and Baker, Chapter 7; IRIS GL programing guide, Chapter 7; OpenGL Programming Guide, Chapter 3, Manipulating Matrix Stacks; Red book, Chapter 7; White book, Chapter 7.*

12.2 Hierarchical Data Structures

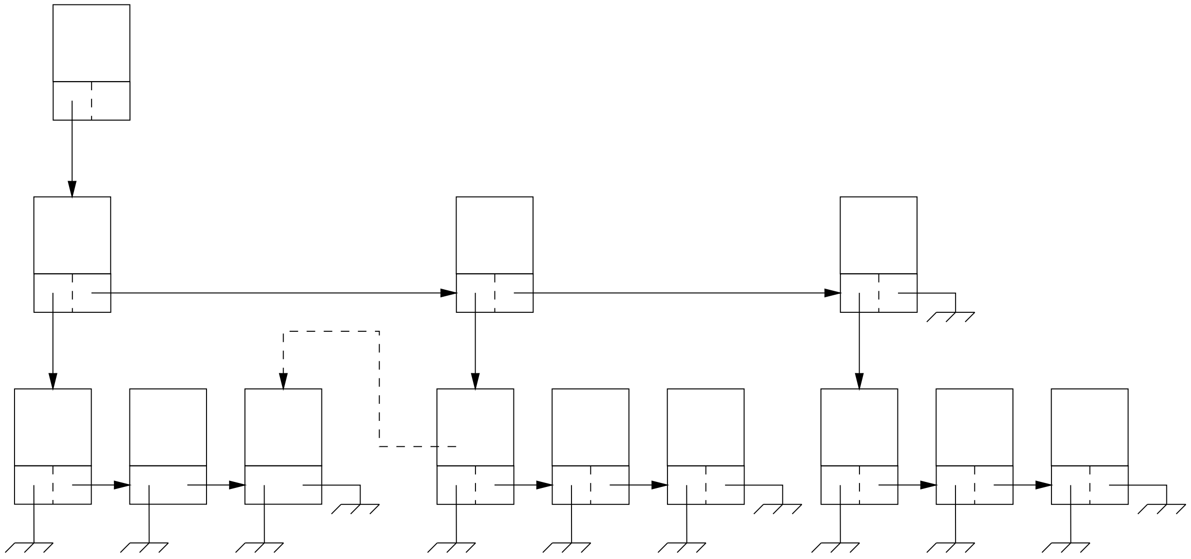
- Don't want to write a program for each scene…
- Instead, develop a data structure and traverse it.
- Use an external data format to describe model hierarchies.
- Process many different scenes with the same program.
- DAG data structure: lots of variations on the example given here.
- Consider the following DAG:



- Use a linked list to allow an arbitrary number of links from each node.



- Two node types: `Data` and `Link`.
  - `Data`: Transformation, Model Geometry.
  - `Link`: First child pointer, sibling pointer.
- We walk through the hierarchy with a preorder traversal…
- If multi-parented nodes uncommon, treat as special case…
- Combine `Link` and `Data` nodes.



  - One node type.
  - Contains transformation, model geometry, first child pointer, sibling pointer.

  - Define an "instance" geometry type which is a link elsewhere in the tree.

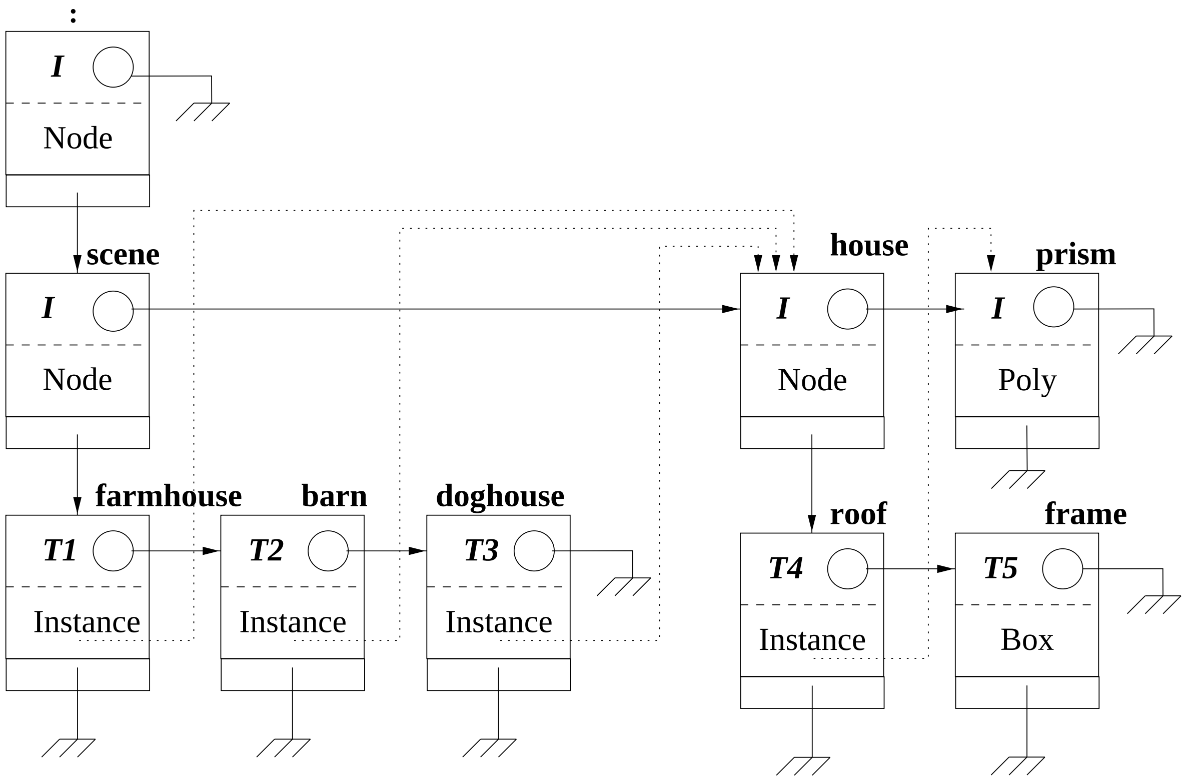
- Traversal pseudocode:

```
Traverse (Node* ptr) {
    Node *child, *instance;
    PushMatrix();
        MultMatrix(ptr->transform);
        if (ptr->geom_type == INSTANCE_GEOM) {
            instance = (Node *) ptr->geom_data;
            Traverse(instance);
        } else if (ptr->geom_type == PRIMITIVE_GEOM) {
            DrawPrimitive(ptr->geom_type, ptr->geom_data);
        }
        child = ptr->first_child;
        while (child != NULL) {
            Traverse(child);
            child = child->sibling;
        }
    PopMatrix();
}
```

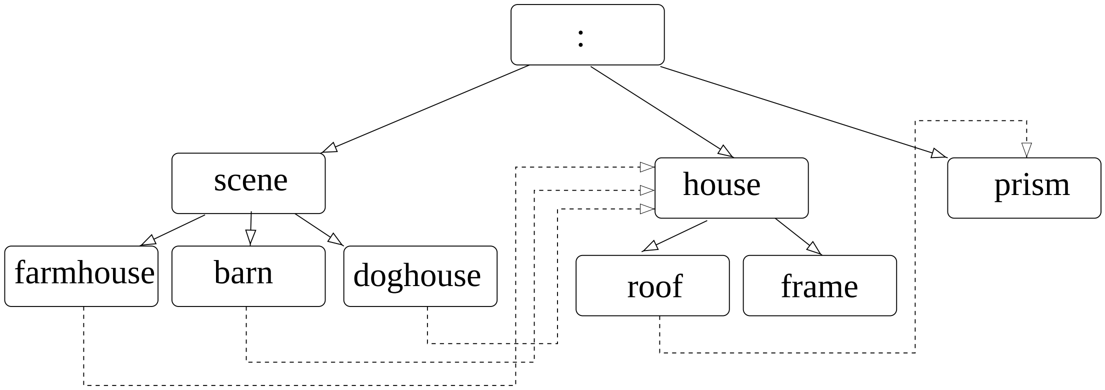
A3/A4 House Code

```
gr_transform : scene

gr_transform : house
gr_polyhedron : prism {...} {...}
gr_instance :house roof :prism
gr_translate :house:roof {0 0 1}
gr_cube :house frame

gr_instance :scene farmhouse :house
    gr_surfaceproperty :scene:farmhouse farmhouse_green
gr_instance :scene barn :house
    gr_surfaceproperty :scene:barn barn_red
    gr_translate :scene:barn {-2 0 3}
    gr_rotate :scene:barn rotate "y" 30
    gr_scale :scene:barn 2
gr_instance :scene doghouse :house
    gr_surfaceproperty :scene:doghouse doghouse_blue
    gr_translate :scene:doghouse {-1.5 0 -0.2}
    gr_rotate :scene:doghouse rotate "y" -15
    gr_scale :scene:doghouse 0.2

    gr_render :scene
```

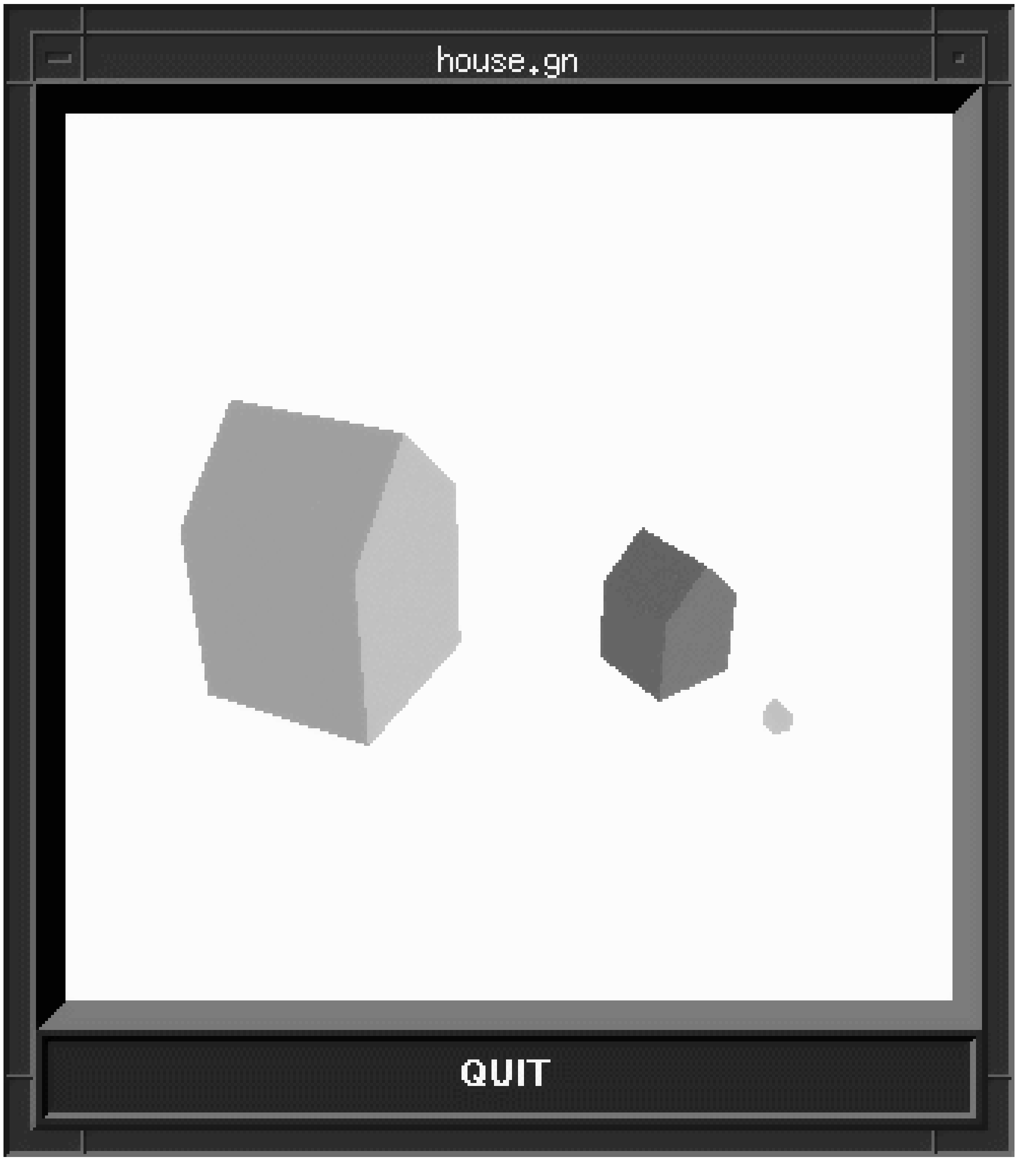
- House example:



- Note: Conceptually, DAG is



- Primitives only occur at leaf nodes.
- Transformations only occur at internal nodes.
- Rendered House:



13 Picking and 3D Selection

13.1 Picking and 3D Selection

- **Pick:** Select an object by positioning mouse over it and clicking.
- **Question:** How do we decide what was picked?
  - We could do the work ourselves:
    - Map selection point to a ray;
    - Intersect with all objects in scene.
  - Let OpenGL/graphics hardware do the work.
- **Idea:** Draw entire scene, and "pick" anything drawn near the cursor.

  - Only “draw” in a small viewport near the cursor.
  - Just do clipping, no shading or rasterization.
  - Need a method of identifying “hits”.
  - OpenGL uses a **name stack** managed by `glInitNames()`, `glLoadName()`, `glPushName()`, and `glPopName()`.
  - “Names” are unsigned integers
  - When hit occurs, copy entire contents of stack to output buffer.

- Process:
  - Set up pick buffer
  - Initialize name stack
    May need `glPushName(-1);`
  - Save old projective transformation and set up a new one
  - Draw your scene with appropriate use of name stack
  - Restore projective transformation
  - Query for number of hits and process them

- A hit occurs if a draw occurs in the (small) viewport

- All information (except ’hits’) is stored in buffer given with `glSelectBuffer`

- **Example:**
  - Two objects
  - Might pick none
  - Might pick either one
  - Might pick both

```
glSelectBuffer(size, buffer);      /* initialize */
glRenderMode(GL_SELECT);
glInitNames();
glPushName(-1);
glGetIntegerv(GL_VIEWPORT,viewport);    /* set up pick view */
glMatrixMode(GL_PROJECTION);
glPushMatrix();
glLoadIdentity();
gluPickMatrix(x,y,w,h,viewport);
glMatrixMode(GL_MODELVIEW);

ViewMatrix();
glLoadName(1);
Draw1();
glLoadName(2);
Draw2();
```

```
glMatrixMode(GL_PROJECTION);
glPopMatrix();
glMatrixMode(GL_MODELVIEW);
hits = glRenderMode(GL_RENDER);
```

- Hits stored consequetively in array
- In general, if h is the number of hits, the following is returned.
  - `hits =` h.
  - h **hit records**, each with four parts:
    1. The number of items q on the name stack at the time of the hit (1 int).
    2. The minimum z value amoung the primitives hit (1 int).
    3. The maximum z value amoung the primitives hit (1 int).
    4. The contents of the hit stack, deepest element first (q ints).
- At most one hit between each call `glRenderMode` or a change to name stack
- In our example, you get back the following:
  - If you click on Item 1 only:
    `hits = 1`,
    `buffer = 1, min(z1), max(z1), 1`.
  - If you click on Item 2 only:
    `hits = 1`,
    `buffer = 1, min(z2), max(z2), 2`.
  - If you click over both Item 1 and Item 2:
    `hits = 2`,
    `buffer = 1, min(z1), max(z1), 1, 1, min(z2), max(z2), 2`.
- **More complex example:**

```
/* initialization stuff goes here */
glPushName(1);
    Draw1();        /* stack: 1 */
    glPushName(1);
        Draw1_1();      /* stack: 1 1 */
        glPushName(1);
            Draw1_1_1();    /* stack: 1 1 1 */
        glPopName();
        glPushName(2);
            Draw1_1_2();    /* stack: 1 1 2 */
        glPopName();
    glPopName();
    glPushName(2);  /* stack: 1 2 */
        Draw1_2();
    glPopName();
glPopName();
glPushName(2);
    Draw2();        /* stack: 2 */
glPopName();
/* wrap-up stuff here */
```

- What you get back:
  - If you click on Item 1:
    `hits = 1`,
    `buffer = 1, min(z1), max(z1), 1`.
  - If you click on Items 1:1:1 and 1:2:
    `hits = 2`,
    `buffer = 3, min(z111), max(z111), 1, 1, 1, 2, min(z12), max(z12), 1, 2`.
  - If you click on Items 1:1:2, 1:2, and 2:
    `hits = 3`,
    `buffer = 3, min(z112), max(z112), 1, 1, 2, 2, min(z12), max(z12), 1, 2, 1, min(z2), max(z2), 2`.
- Important Details:
  - Make sure that projection matrix is saved with a `glPushMatrix()` and restored with a `glPopMatrix()`.
  - `glRenderMode(GL_RENDER)` returns negative if buffer not big enough.
  - When a hit occurs, a flag is set.
  - Entry to name stack only made at next `gl*Name(s)` or `glRenderMode` call. So, each draw block can only generate at most one hit.

♠ *Readings: GL Programming Manual, Chapter 12*

14 Colour and the Human Visual System

14.1 Introduction to Colour

Colour

- Light sources emit intensity:
  $$I(\lambda)$$
  assigns intensity to each wavelength of light
- Humans perceive $I(\lambda)$ as a colour – navy blue, light green, etc.
- Exeriments show that there are distinct I with are perceived as the same colour (metamers)
- Normal human retina have three types of colour receptor which respond most strongly to short, medium, or long wavelengths.
  (picture)
- Note the low response to blue.
  One theory is that sensitivity to blue is recently evolved.

- Different animals have different number of wavelengths that they are sensitive to:
  - Dogs: 1
  - Primates: 2 or 3
  - Pigeon: 4
  - Birds: up to 18 (hummingbird?)
- Different Regions of the eye are "designed" for different purposes:
  - Center - fine grain colour vision
  - Sides - night vision and motion detection

Tri-Stimulus Colour Theory

- Tri-stimulus Colour Theory models visual system as a linear map:

  $$V : \Lambda \rightarrow C$$

  where Λ is a continuous function of wavelength
  C is a three dimensional vector space (colour space)

  $$V(I(\lambda)) = \vec{C} = \ell\vec{\ell} + m\vec{m} + s\vec{s}$$

  where

  $$\begin{aligned} \ell &= \int_{-\infty}^{\infty} L(\lambda)I(\lambda)d\lambda \\ m &= \int_{-\infty}^{\infty} M(\lambda)I(\lambda)d\lambda \\ s &= \int_{-\infty}^{\infty} S(\lambda)I(\lambda)d\lambda \end{aligned}$$

  where L, M, and S are weight functions.
- Monitor has Red, Green, Blue additive colour system
- Most colour creation is guess work.

Colour Systems

- RGB (Red, Green, Blue) Additive
- CMY (Cyan, Magenta, Yellow) Subtractive (complement of RGB)
  Often add K (blacK) to get better black
- HSV (Hue, Saturation, Value) Cone shaped colour space
  Also HSL (double cone)
- CIE XYZ (Colour by committee) More complete colour space
  Also L*u*v and L*a*b
- YIQ (Y == Luminence == CIE Y; IQ encode colour)
  Backwards compatible with black-and-white TV (only show Y)

♠ *Readings: Watt: Chapter 15. Hearn and Baker: Chapter 15, 14-5. Glassner: Volume 1, Chapter 1, 2.*

15 Reflection and Light Source Models

15.1 Introduction

- What we want:

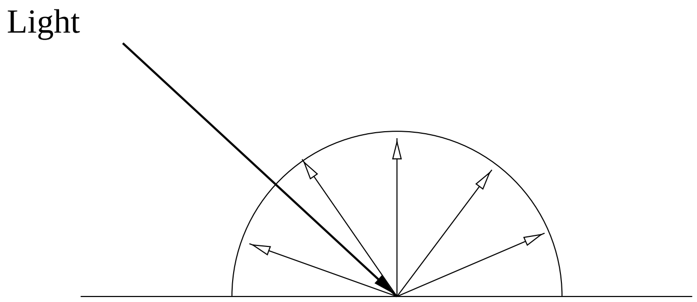
  Given a point on a surface visible to the viewer through a pixel, what colour should we assign to the pixel?

- We want to smoothly shade objects in scene
- We want shading to be done quickly so that we can achieve real time
- Begin with creating a simple lighting model at a single point on a surface
- Not going to worry *too* much about the physics...
- Later, we will improve lighting model
- Initial Assumptions:
  - Linearity of reflection: outgoing energy proportional to incoming
  - Energy conservation: no more outgoing energy than incoming
  - Full spectrum of light can be represented by three floats (Red,Green,Blue)

♠ *Readings: Watt: 6.2. Red book: 14.1. White book: 16.1.*

15.2 Lambertian Reflection

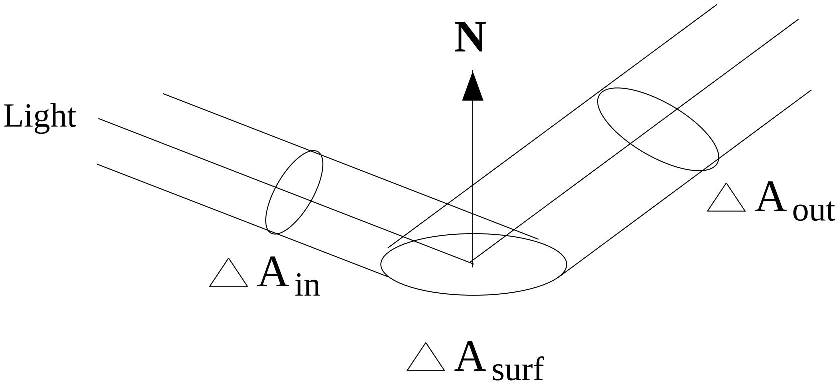
- Assume: Incoming light is partially absorbed, then remainder of energy propagated **equally** in all directions



- Approximates the behavior of matte materials.
- Want an expression for $L_{\text{out}}(\vec{v})$, the radiance (light energy transported along a ray) reflected in some direction $\vec{v}$, given
  - the incoming direction $\vec{\ell}$
  - incoming radiance $L_{\text{in}}(\vec{\ell})$,

  - and the material properties (reflectance).

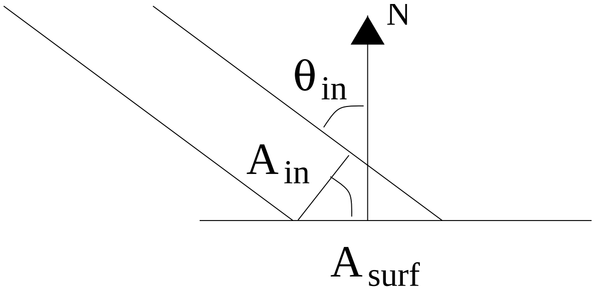
- Given radiance $L_{\text{in}}(\vec{\ell})$ striking the surface from a direction $\vec{\ell}$...
- Want to determine $L_{\text{out}}(\vec{v})$ reflected towards a viewer in direction $\vec{v}$.



- Energy received at surface (irradiance E) depends on projection of incoming beam orientation onto surface's orientation:

$$E_{\text{in}}(\vec{\ell}) \quad = \quad L_{\text{in}}(\vec{\ell})\left(\frac{\Delta A_{\text{in}}}{\Delta A_{\text{surf}}}\right).$$

Want to compute projected area:



$$\begin{aligned}\frac{\Delta A_{\text{in}}}{\Delta A_{\text{surf}}} &= \cos(\theta_{\text{in}}) \\ &= \vec{\ell}\cdot\vec{n}\end{aligned}$$

where $\vec{n}$ is the surface normal and $\vec{\ell}$ points *towards* the light.

Our "view" of the surface will also include a corresponding factor for $\cos(\theta_{\text{out}})$.

Outgoing radiance proportional to incoming irradiance, but proportionality may depend on view vector $\vec{v}$ and light vector $\vec{\ell}$:

$$L_{\text{out}}(\vec{v}) \quad = \quad \rho(\vec{v},\vec{\ell})E_{\text{in}}(\vec{\ell}).$$

Have assumed outgoing radiance is equal in all directions so ρ must be a constant.

The *Lambertian lighting model* is therefore

$$\begin{aligned} L_{\text{out}}(\vec{v}) &= k_d E_{\text{in}}(\vec{\ell}) \\ &= k_d L_{\text{in}}(\vec{\ell})\, \vec{\ell} \cdot \vec{n}. \end{aligned}$$

For complete environment, Lambertian lighting model is

$$L_{\text{out}}(\vec{v}) = \int_{\Omega} (k_d/\pi) L_{\text{in}}(\vec{\ell})\; (\vec{\ell} \cdot \vec{n})\; d\sigma(\vec{\ell})$$

where Ω is the hemisphere of all possible incoming directions and $d\sigma$ is the solid angle measure.

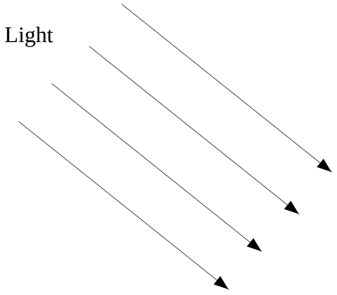
If $k_d \in [0, 1]$, then factor of π is necessary to ensure conservation of energy.

15.3 Attenuation

Attenuation

- We will model two types of lights:
  - Directional
  - Point

  Want to determine $L_{\text{in}}(\vec{\ell})$ at surface for each
- Directional light source has parallel rays:

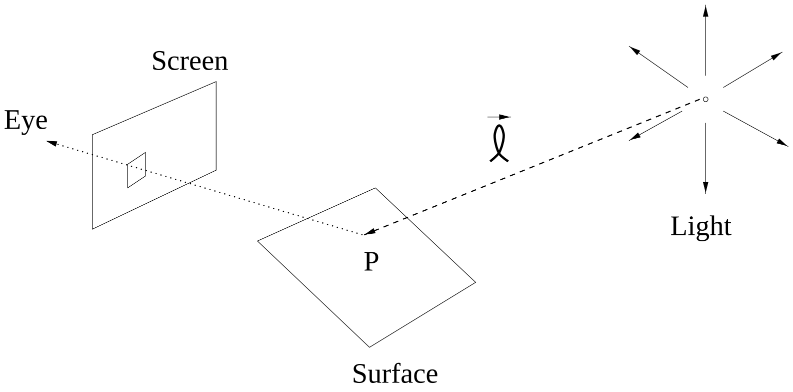


  Most appropriate for distant light sources (the sun)

  No attenuation, since energy does not "spread out".

Point light sources:

- Light emitted from a point equally in all directions:



- Conservation of energy tells us

$$L_{\text{in}}(\vec{\ell}) \propto \frac{I}{r^2}$$

  where r is the distance from light to P (energy is spread over surface of a sphere), and I is light source intensity.

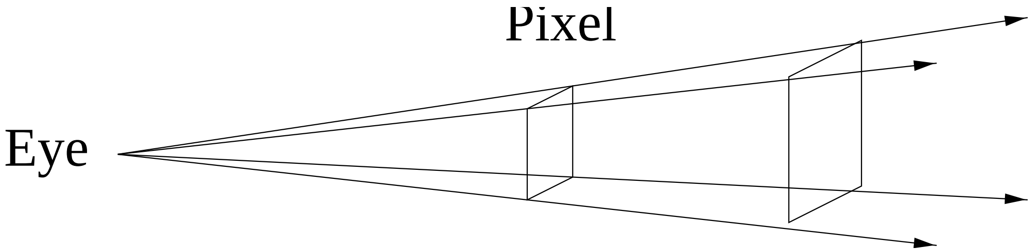
- For "empirical graphics", usually ignore π factors...
- However, r^2 attenuation looks too harsh.

  Harshness because real lighting is from area sources, multiple reflections in environment. True point sources impossible.

  Commonly, we will use

$$L_{\text{in}}(\vec{\ell}) = \frac{I}{c_1 + c_2 r + c_3 r^2}$$

- Note that we do NOT attenuate light from P to screen. Surface foreshortening when farther away takes care of that:



The pixel represents an area that increases as the square of the distance.

15.4 Coloured Lights, Multiple Lights, and Ambient Light

- To get coloured lights, we perform lighting calculation three times to get an RGB triple.
- More correct to use spectra, and better approximations to spectra exist, but RGB sufficient for now
- To get multiple lights, compute contribution independently and sum:

$$L_{\text{out}}(\vec{v}) = \sum_i \rho(\vec{v}, \vec{\ell}_i) I_i \frac{\vec{\ell}_i \cdot \vec{n}}{c_1 + c_2 r_i + c_3 r_i^2}$$

- Question: what do pictures with this illumination look like?

Ambient Light:

- Lighting model so far is still too harsh
- Problem is that only direct illumination is modeled
- Global illumination techniques address this—but are expensive
- Ambient illumination is a simple approximation to global illumination
- Assume everything gets uniform illumination in addition to direct illumination

$$L_{\text{out}}(\vec{v}) = k_a I_a + \sum_i \rho(\vec{v}, \vec{\ell}_i) I_i \frac{\vec{\ell}_i \cdot \vec{n}}{c_1 + c_2 r_i + c_3 r_i^2}$$

15.5 Specular Reflection

- Lambertian term models matte surface but not shiny ones
- Shiny surfaces have "highlights" because energy reflected depends on viewer's position
- Phong Bui-Tuong developed an empirical model:

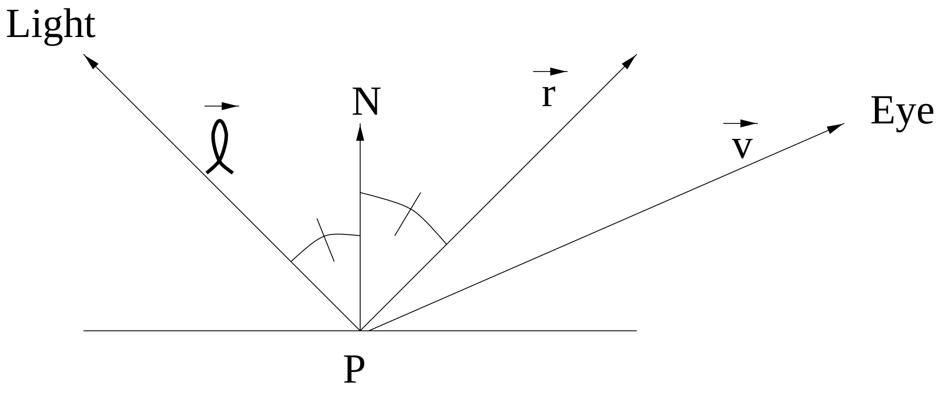
$$L_{\text{out}}(\vec{v}) = k_a I_a + k_d(\vec{\ell} \cdot \vec{n}) I_d + k_s (\vec{r} \cdot \vec{v})^p I_s$$

- Using our previous notation:

$$\rho(\vec{v}, \vec{\ell}) = k_d + k_s \frac{(\vec{r} \cdot \vec{v})^p}{\vec{n} \cdot \vec{\ell}}$$

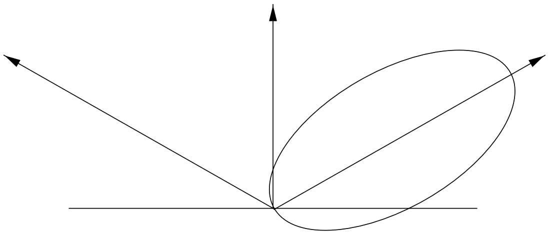
- The vector $\vec{r}$ is $\vec{\ell}$ reflected by the surface:

$$\vec{r} = -\vec{\ell} + 2(\vec{\ell} \cdot \vec{n})\vec{n}$$

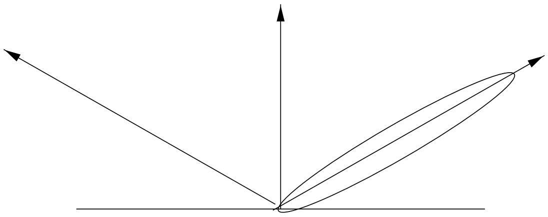


- This is the classic *Phong lighting model*
- Specular term at a maximum (k_s) when $\vec{v} = \vec{r}$

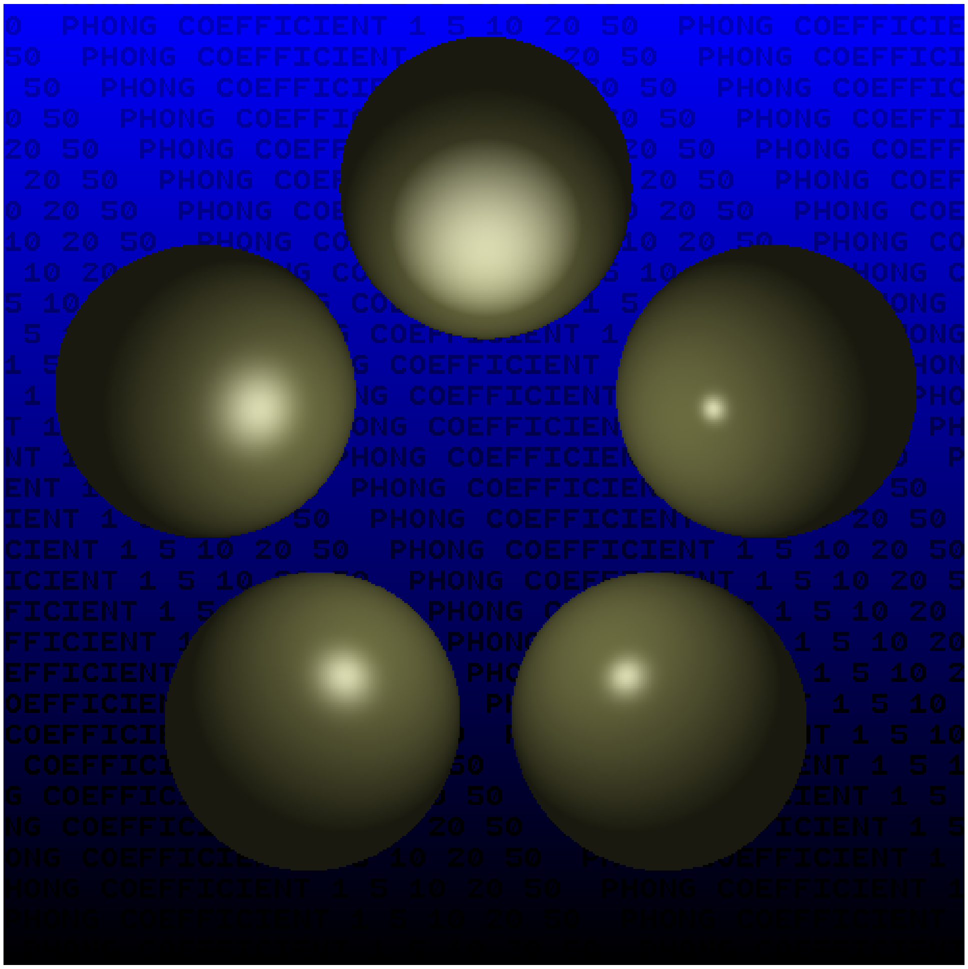
- The exponent p controls sharpness of highlight:
  - Small p gives wide highlight
  - Large p gives narrow highlight



Small p



Large p



- Blinn introduced a variation, the *Blinn-Phong lighting model*:

$$L_{\text{out}}(\vec{v}) = k_a I_a + k_d(\vec{\ell} \cdot \vec{n}) I_d + k_s(\vec{h} \cdot \vec{n})^p I_s$$

  Compare to Phong model of

$$L_{\text{out}}(\vec{v}) = k_a I_a + k_d(\vec{\ell} \cdot \vec{n}) I_d + k_s(\vec{r} \cdot \vec{v})^p I_s$$

- The halfway vector $\vec{h}$ is given by
$$\vec{h} = \frac{\vec{v} + \vec{\ell}}{|\vec{v} + \vec{\ell}|}$$
- Value $(\vec{h} \cdot \vec{n})$ measures deviation from ideal mirror configuration of $\vec{v}$ and $\vec{\ell}$
- Exponent works similarily to Phong model.
- OpenGL uses Blinn-Phong model.
- Blinn-Phong better motivated physically …
- **BUT** both models break several laws of physics
(conservation of energy, for instance).
- Better physically-based and empirical models exist:
Cook-Torrance, He, Ward, Lafortune's modified Phong, etc.
- Good empirical model for project:
Ashikhmin and Shirley,
An Anisotropic Phong BRDF Model,
Journal of Graphics Tools, AK Peters/ACM,
Volume 5, Number 2, 2000.

16 Shading

♠ *Shading algorithms apply lighting models to polygons, through interpolation from the vertices.*

Flat Shading: *Perform lighting calculation once, and shade entire polygon one colour.*

Gouraud Shading: *Lighting is only computed at the vertices, and the colours are interpolated across the (convex) polygon.*

Phong Shading: *A normal is specified at each vertex, and this normal is interpolated across the polygon. At each pixel, a lighting model is calculated.*

Note distinction between lighting model and shading algorithm…

16.1 Introduction

Shading

- Want to shade surfaces
- Lighting calculation for a point
Given: L_{in}, $\vec{\ell}$, and surface properties (including surface normal)
Compute: L_{out} in direction $\vec{v}$
- Need surface normals at every point to be lighted
- Commonly, surface is polygonal
  - True polygonal surface: use polygon normal

  - Sampled surface: sample position and normal, create polygonal approximation
- Want colour for each pixel in rasterized surface

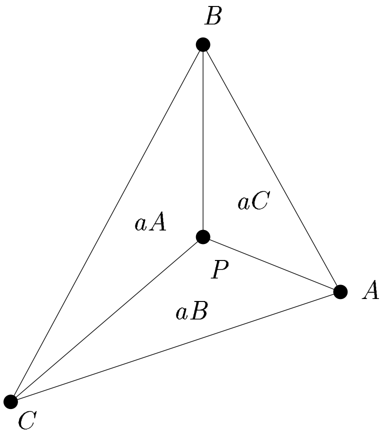
Flat Shading :

- Shade entire polygon one colour
- Perform lighting calculation at:
  - One polygon vertex
  - Center of polygon
    What normal do we use?
  - All polygon vertices and average colours
- Problem: Surface looks faceted
- OK if really is a polygonal model, not good if a sampled approximation to a curved surface.

♠ *Readings: Watt: 6.3. Red book: 14.2. White book: 16.2.4, 16.2.5. Hearn and Baker: 14.5.*

16.2 Gouraud Shading

- **Gouraud shading** interpolates colours across a polygon from the vertices.
- Lighting calculations are only performed at the vertices.
- Interpolation well-defined for triangles.
- Extensions to convex polygons . . . but not a good idea, convert to triangles.
- Barycentric combinations are also **affine combinations**. . .
  Triangular Gouraud shading is **invariant** under affine transformations.



$aA = \triangle PBC/\triangle ABC$

$aB = \triangle APC/\triangle ABC$

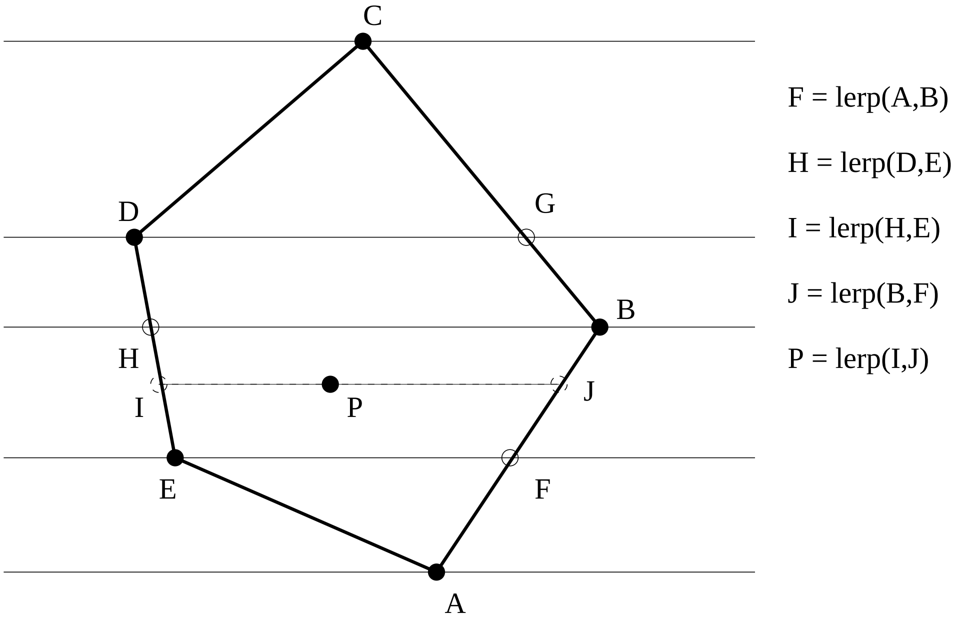
$aC = \triangle ABP/\triangle ABC$

$aA + aB + aC = 1$

$P = aAA + aBB + aCC$

- To implement, can use repeated affine combination along edges, across spans, during rasterization.
- Gouraud shading is well-defined only for triangles

- For polygons with more than three vertices:
  - Sort the vertices by y coordinate.
  - Slice the polygon into trapezoids with parallel top and bottom.
  - Interpolate colours along each edge of the trapezoid...
  - Interpolate colours along each scanline.



- Gouraud shading gives **bilinear** interpolation within each trapezoid.
- Since rotating the polygon can result in a different trapezoidal decomposition, n-sided Gouraud interpolation is **not affine invariant.**
- Aliasing also a problem: highlights can be missed or blurred.

  Not good for shiny surfaces unless fine polygons are used.
- Exercise: Provide an example of the above effect.
- Exercise: Prove repeated affine combinations (the above algorithm) is equivalent to barycentric combinations on triangles.
- Exercise: Prove that the extension of the repeated affine combination algorithm to arbitrary polygons is not invariant under affine transformations.
- Common in hardware renderers; the model that (classic) OpenGL supports.
- Linear interpolation in device space not consistent with linear interpolation in world space. Modern implementations of OpenGL actually implement rational linear interpolation, which takes perspective into account.

16.3 Phong Shading

- **Phong Shading** interpolates lighting model parameters, **not** colours.
- Much better rendition of highlights.
- A **normal** is specified at each vertex of a polygon.
- Vertex normals are independent of the polygon normal.
- Vertex normals should relate to the surface being approximated by the polygonal mesh.
- The normal is interpolated across the polygon (using Gouraud techniques).
- At each pixel,
  - Interpolate the normal…
  - Interpolate other shading parameters…
  - Compute the view and light vectors…
  - Evaluate the lighting model.
- The lighting model does not have to be the Phong lighting model!
- Normal interpolation is nominally done by vector addition and renormalization.
- Several “fast” approximations are possible.
- The view and light vectors may also be interpolated or approximated.
- Problems with Phong shading:
  - Distances change under perspective transformation
  - Where do we do interpolation?
  - Normals don’t map through perspective transformation
  - Can’t perform lighting calculation or linear interpolation in device space
  - Have to perform lighting calculation in *world space* or *view space*, assuming model-view transformation is affine.
  - Have to perform linear interpolation in world or view space, project into device space
  - Results in rational-linear interpolation in device space!
  - Interpolate homogenous coordinates, do per-pixel divide.
  - Can be organized so only need *one* division per pixel, regardless of the number of parameters to be interpolated.
- Phong shading, and many other kinds of advanced shading, can be simulated with programmable vertex and fragment shaders on modern graphics hardware:
  - Classic Gouraud shading is linear in device space.
  - Modern graphics hardware performs rational linear interpolation — looks like interpolation happens in world space.

- Interpolate normals, view vectors, and light vectors using generic interpolation hardware
- Usually also interpolate diffuse term after computing it in vertex shader.
- Write fragment shader to renormalize normals to unit length, compute lighting model, and assign colour to pixel.
- Can also use texture maps as lookup tables…

17 Graphics Hardware

17.1 Graphics Hardware

- Graphics hardware has been evolving over the past several decades
- In the 60's, 70's, vector displays costing millions of dollars

  US military was main (only) customer
- Raster displays started appearing in the 70's
- In the 80's, raster displays and graphics hardware cost 10's to 100's of thousands of dollars.

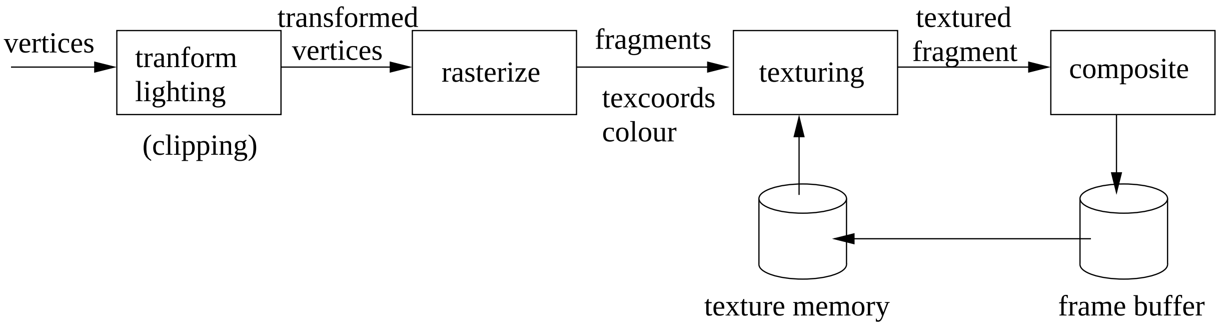
  Cost low enough that companies could afford the hardware.

  SGI won the battle of work station graphics, OpenGL became standard
- In the 90's, graphics hardware started moving to PC's

  DirectX is MicroSoft's challenge to OpenGL
- In the 00's, PC graphics hardware has displaced work station graphics

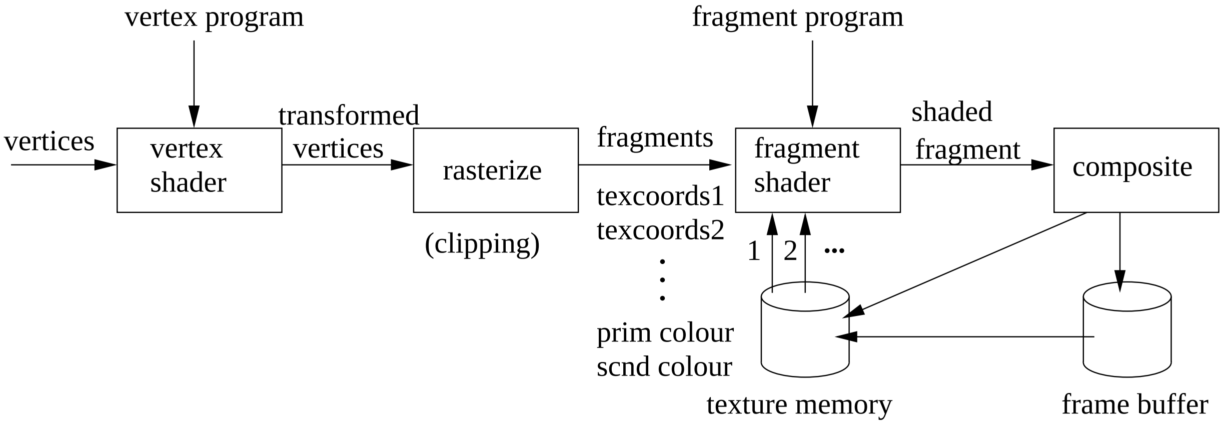
  Good, inexpensive cards for PCs
- Currently getting faster in excess of Moore's law
- Many graphics cards are "more powerful" than CPU

Traditional OpenGL (1.2)



- A fragment is a sort of virtual pixel
- Clipping done in texture lighting portion

OpenGL (2.0)



- Vertex Shader and Fragment Shader are programmable by application

GeForce 3/4:

- 100's of lines of assembly for vertex shader (floating point 4-tuple)
- Configurable pipeline for fragment shader [SPEC ONLY]
    - 4 texture shader stages (floating point)
    - 8 register combiner stages (10-bit signed int)

ATI 9700

- 1000's of lines of assembly for both vertex and fragment shader
- Single precision float for vertex shader
- 24-bit float in fragment shader
  15-bit significand

OpenGL 2.0

- full floating point in both shaders
- 1000's of assembly instructions, loops
- high level shading language

17.2 GeForce 3/4 Vertex Shader

- Sets of four word registers
- Data passed in through v, c registers
  Results passed to next stage through o registers

  - 16 vertex attribute registers
    v[0],...,v[15]
    Mnemonic names: v[OPOS], v[WGHT], V[NML],...
    position, weight, normal, primary color,...
    Programmer responsible for loading data into them
    Vertex attribute registers are read only
    Only one v register per instruction
  - 15 vertex result registers
    o[HPOS] – Homogeneous clip space position
    o[COL0] – Primary colour
    o[TEX0] – Texture coordinate set 0
    ...
    Vertex result registers are write only
    Vertex program *must* write to o[HPOS]
  - 12 temporary registers, R0,...,R11
    Read/write
  - A0.x is the "address register"
    Write only
    Used to access constant registers
  - 96 constant registers, c[0],...,c[95]
    May also address as c[A0.x]
    Read-only to normal program
    Read/write to vertex state program
    Only one constant register per instruction

- 17 assembly instructions
  - Operate on scalar(s) or 4-component register
  - Result is for single register
    Either 4-component or replicated scalar
  - Examples:

Opcode	Inputs	Output	Operations
MOV	v	v	move
ADD	v,v	v	add
RSQ	s	ssss	reciprocal square root
DP4	v,v	ssss	4-component dot product

  - Scalar input must have modifier .x, .y, .z, or .w
  - Swizzle: .xxyy, .wzyx, etc.

- Generate id, load/compile vertex program, etc.,...
  glGenProgramsNV, glLoadProgramNV,...

- Web pages:

```
http://developer.nvidia.com/docs/IO/1239/ATT/VertexPrograms.pdf
http://www.nvidia.com/dev_content/nvopenglspecs/GL_NV_vertex_program.txt
http://www.nvidia.com/dev_content/nvopenglspecs/GL_NV_vertex_program1_1.txt
```

Example: Phong Lighting

```
// X = Vertex position
// LP = Light position
// EP = Eye position
// L = Normalized light vector
// E = Normalized eye vector
// R = Reflection vector
// n = Specular exponent

// Parameters:
//   c[0-3] = Modelview times projection matrices
//   c[10] = Diffuse color
//   c[11] = Specular color
//   c[12] = Light position
//   c[13] = Eye position
//   c[14] = (n, 0.0, 0.0, 0.0)
//   c[16] = (2.0, 2.0, 2.0, 2.0)

GLubyte vpPhong_str[] =
"!!VP1.0\n"

"DP4 o[HPOS].x, c[0], v[OPOS];"  // Apply modelview and
"DP4 o[HPOS].y, c[1], v[OPOS];"  // projection matrices
"DP4 o[HPOS].z, c[2], v[OPOS];"
"DP4 o[HPOS].w, c[3], v[OPOS];"

"ADD R0, c[12], -v[OPOS];"    // R0 = LP - X
"DP3 R1, R0, R0;"             // R1 = |LP - X|^2
"RSQ R2, R1.w;"               // R2 = 1/|LP - X|
"MUL R0, R0, R2;"             // R0 = L

"DP3 R3, R0, v[NRML];"        // R3 = N.L

"ADD R4, c[13], -v[OPOS];"    // R4 = EP - X
"DP3 R5, R4, R4;"             // R5 = |EP - X|^2
"RSQ R6, R5.w;"               // R6 = 1/|EP - X|
"MUL R4, R4, R6;"             // R4 = E

"DP3 R7, R4, v[NRML];"        // R7 = E.N
"MUL R7, R7, c[16];"          // R7 = 2*(E.N)
"MAD R8, R7, v[NRML], -R4;"   // R8 = 2*(E.N)N-E = R

"DP3 R9, R8, R0;"             // R9 = R.L

"LOG R10, R9.x;"              // R10.z = LOG(R.L)
"MUL R9, c[14].x, R10.z;"     // R9 = n*LOG(R.L)
```

```
"EXP R11, R9.z;"              // R11 = EXP(n*LOG(R.L)) = (R.L)^n

"MUL R10, R3, c[10];"               // R10 = Cd*(N.L)
"MAD o[COL0], c[11], R11.z, R10;"   // o[COL0] = Cd*(N.L) + Cs*(R.L)^n

"END";
```

17.3 Cg: C for Graphics

- Hard to program (maintain, read) assembly
- Several high level shading languages being developed

  Cg, GL2, HLSL
- Cg is an example of a C-like language to create shaders
- Compile at runtime
- Web page for docs

  http://developer.nvidia.com/Cg

Cg example (from nvidia Cg Toolkit User's Manual)
Have to define inputs and outputs:

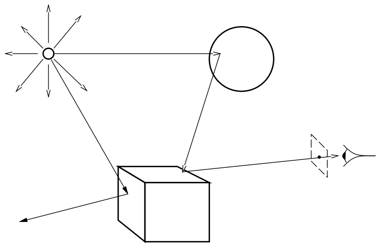
```
// define inputs from application
struct appin {
float4 Position : POSITION;
float4 Normal : NORMAL;
};

// define outputs from vertex shader
struct vertout {
float4 Hposition : POSITION;
float4 Color0 : COLOR0;
};

vertout main(appin In,
     uniform float4x4 ModelViewProj : C0,
     uniform float4x4 ModelViewIT : C4,
     uniform float4 LightVec)
{
vertout Out;
Out.Hposition = mul(ModelViewProj, In.Position);
// transform normal from model space to view space
float4 normal = normalize(mul(ModelViewIT, In.Normal).xyzz);

// store normalized light vector
float4 light = normalize(LightVec);

// calculate half angle vector
float4 eye = float4(0.0, 0.0, 1.0, 1.0);
float4 half = normalize(light + eye);

// calculate diffuse component
float diffuse = dot(normal, light);

// calculate specular component
float specular = dot(normal, half);
specular = pow(specular, 32);

// blue diffuse material
float4 diffuseMaterial = float4(0.0, 0.0, 1.0, 1.0);

// white specular material
float4 specularMaterial = float4(1.0, 1.0, 1.0, 1.0);

// combine diffuse and specular contributions
// and output final vertex color
Out.Color0 = diffuse * diffuseMaterial + specular * specularMaterial;
return Out;
}
```

18 Ray Tracing

18.1 Basics

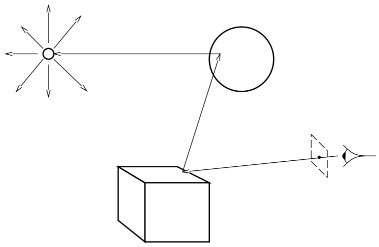
Ray Tracing: Photorealism – The Holy Grail

- Want to make more realistic images
  - Shadows
  - Reflections
- What happens in the real world?



  Problem: Many rays never reach the eye.

- Idea: Trace rays backwards



Ray Tracing: Basic Code

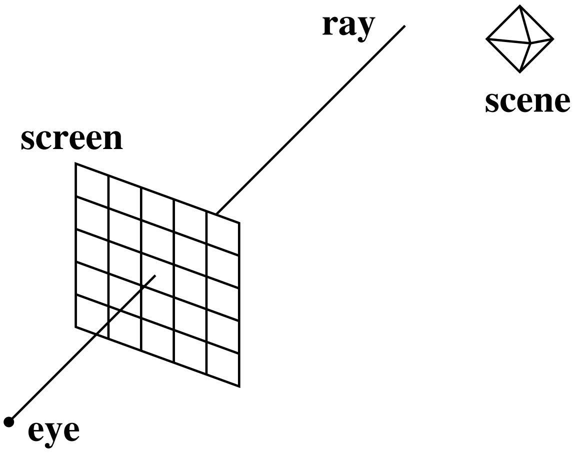
```
foreach pixel
    ray = (eye, pixel-eye);
    Intersect(Scene, ray);
end
```

Issues:

- Which pixels?
- How do we perform intersections?
- How do we perform shading?
- How can we speed things up?
- How can we make things look better?

Ray Tracing/Casting: Selecting the initial ray

- Setting: *eyepoint*, *virtual screen* (an array of *virtual pixels*, and *scene* are organized in convenient coordinate frames (e.g. all in view or world frames)
- Ray: a half line determined by the eyepoint and a point associated with a chosen pixel
- Interpretations:
    - Ray is the path of photons that succesfully reach the eye (we simulate selected photon transport through the scene)
    - Ray is a sampling probe that gathers color/visibility information



♠ *Readings: Watt: 12.1. Red book: 14.7. White book: 16.12.*

18.2 Intersection Computations

General Issues:

- Ray: express in parametric form

  $$E + t(P - E)$$

  where E is the eyepoint and P is the pixel point
- Scene Object: direct implicit form
  - express as $f(Q) = 0$ when Q is a surface point, where f is a given formula
  - intersection computation is an equation to solve:
    find t such that $f(E + t(P - E)) = 0$
- Scene Object: procedural implicit form
  - f is not a given formula
  - f is only defined procedurally
  - $\mathcal{A}(f(E + t(P - E)) = 0)$ yields t, where A is a root finding method (secant, Newton, bisection, ...)

Quadric Surfaces:

- Surface given by

  $$Ax^2 + Bxy + Cxz + Dy^2 + Eyz + Fz^2 + Gx + Hy + Jz + K = 0$$

  - Ellipsoid
    $$\frac{x^2}{p^2} + \frac{y^2}{q^2} + \frac{z^2}{r^2} = 1$$
  - Hyperboloid of one sheet
    $$\frac{x^2}{p^2} + \frac{y^2}{q^2} - \frac{z^2}{r^2} = 1$$
  - Cone
    $$\frac{x^2}{p^2} + \frac{y^2}{q^2} - \frac{z^2}{r^2} = 0$$
  - Elliptic cylinder
    $$\frac{x^2}{p^2} + \frac{y^2}{q^2} = 1$$
  - Others...
- Ray given by

  $$\begin{aligned} x &= x_E + t(x_P - x_E) \\ y &= y_E + t(y_P - y_E) \\ z &= z_E + t(z_P - z_E) \end{aligned}$$

- Substitute ray x,y,z into surface formula
  - quadratic equation results for t
  - organize expression terms for numerical accuracy; ie. to avoid

    - cancellation
    - combinations of numbers with widely different magnitudes

Polygons:

- The plane of the polygon should be known

$$Ax + By + Cz + D = 0$$

  - $(A, B, C, 0)$ is the normal vector
  - pick three succesive vertices

$$\begin{aligned} v_{i-1} &= (x_{i-1}, y_{i-1}, z_{i-1}) \\ v_i &= (x_i, y_i, z_i) \\ v_{i+1} &= (x_{i+1}, y_{i+1}, z_{i+1}) \end{aligned}$$

  - should subtend a "reasonable" angle (bounded away from 0 or 180 degrees)
  - normal vector is the cross product $(v_{i_1} - v_i) \times (v_{i-1} - v_i)$
  - $D = -(Ax + By + Cz)$ for any vertex $(x, y, z, 1)$ of the polygon
- Substitute ray x,y,z into surface formula
  - linear equation results for t
- Solution provides planar point $(\bar{x}, \bar{y}, \bar{z})$
  - is this inside or outside the polygon?

Planar Coordinates:

- Take origin point and two independent vectors on the plane

$$\begin{aligned} \mathcal{O} &= (x_{\mathcal{O}}, y_{\mathcal{O}}, z_{\mathcal{O}}, 1) \\ \vec{b_0} &= (u_0, v_0, w_0, 0) \\ \vec{b_1} &= (u_1, v_1, w_1, 0) \end{aligned}$$

- Express any point in the plane as

$$P - \mathcal{O} = \alpha_0 \vec{b}_0 + \alpha_1 \vec{b}_1$$

  - intersection point is $(\bar{\alpha}_0, \bar{\alpha}_1, 1)$
  - clipping algorithm against polygon edges in these α coordinates

Alternatives: Must this all be done in world coordinates?

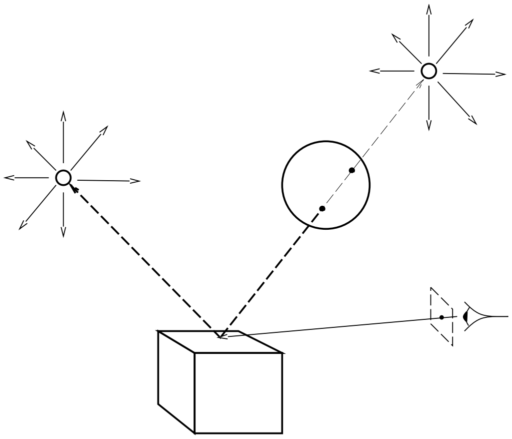
- Alternative: Intersections in model space
  - form normals and planar coordinates in model space and store with model
  - backtransform the ray into model space using inverse modeling transformations
  - perform the intersection and illumination calculations
- Alternative: Intersections in world space
  - form normals and planar coordinates in model space and store
  - forward transform using modeling transformations

18.3 Shading

Shading: Shading, shadows

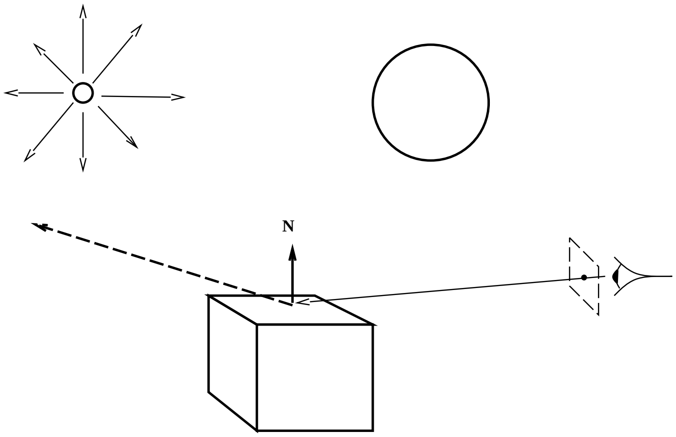
- How do we use ray to determine shade of pixel?
- At closest intersection point, perform Phong shading.

  Gives essentially the same images as standard polygon renderer, but with more primitives.
- Idea: Before adding contribution from a light, cast ray to light.
  - If ray hits another object before hitting light, then don't perform shading calculation.
  - This gives shadows.



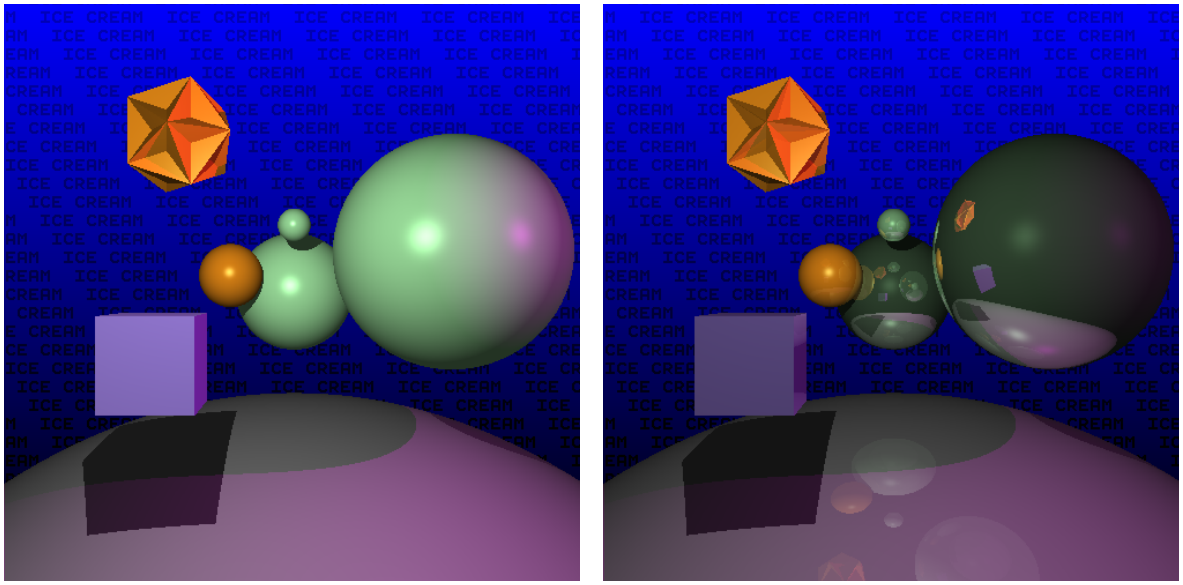
Shading: Reflections

- Idea: Cast a ray in the mirror direction.
- Add the colour coming from this ray to the shade of the initial ray.
- This gives mirror reflections



- We perform a full lighting calculation at the first intersection of this reflected ray.
- If we send another reflected ray from here, when do we stop?
  - Answer 1: Stop at a fixed depth.

  - Answer 2: Accumulate product of reflection coefficients and stop when this product is too small.
- Refraction similar



♠ *Readings: Watt: 12.1.*

18.4 Recursive Ray Tracing

Ray Tracing: Recursive

- Eye-screen ray is the *primary* ray
- Backward tracking of photons that could have arrived along primary
- Intersect with all objects in scene
- Determine nearest object
- Generate secondary rays
  - to light sources
  - in reflection-associated directions
  - in refraction-associated directions
- Continue recursively for each secondary ray
- Terminate after suitably many levels
- Accumulate suitably averaged information for primary ray
- Deposit information in pixel

Ray Casting: Non-recursive

- As above for ray tracing, but stop before generating secondary rays
- Apply illumination model at nearest object intersection with no regard to light occlusion
- Ray becomes a sampling probe that just gathers information on

  - visibility
  - color

♠ *Readings: Watt: 12.2.*

18.5 Surface Information

Surface Normals:

- Illumination models require:
  - surface normal vectors at intersection points
  - ray-surface intersection computation must also yield a normal
  - light-source directions must be established at intersection
  - shadow information determined by light-ray intersections with other objects
- Normals to polygons:
  - provided by planar normal
  - provided by cross product of adjacent edges
  - Or use Phong normal interpolation if normals specified at vertices
- Normals to any implicit surface (eg. quadrics)
  - move from (x, y, z) to $(x + \Delta x, y + \Delta y, z + \Delta z)$ which is *maximally* far from the surface
  - direction of greatest increase to $f(x, y, z)$
- Taylor series:

$$f(x + \Delta x, y + \Delta y, z + \Delta z) = f(x, y, z) + [\Delta x, \Delta y, \Delta z] \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \\ \frac{\partial f}{\partial z} \end{bmatrix} + \cdots$$

  - maximality for the *gradient vector* of f
  - not normalized
- Normal to a quadric surface

$$\begin{bmatrix} 2Ax + By + Cz + G \\ Bx + 2Dy + Ez + H \\ Cx + Ey + 2Fz + J \\ 0 \end{bmatrix}$$

Normal Transformations: How do affine transformations affect surface normals?

- Let P_a and P_b be any two points on object
  - arbitrarily close
  - $\vec{n} \cdot (P_b - P_a) = 0$
  - using transpose notation: $(\vec{n})^T (P_b - P_a) = 0$

- After an affine transformation on P_a,P_b:

$$\mathbf{M}(P_b - P_a) = \mathbf{M}P_b - \mathbf{M}P_a$$

we want $(\mathbf{N}\vec{n})$ to be a normal for some transformation $\mathbf{N}$:

$$\begin{aligned} & (\mathbf{N}\vec{n})^T\mathbf{M}(P_b - P_a) = 0 \\ \implies & \vec{n}^T\mathbf{N}^T\mathbf{M}(P_b - P_a) \end{aligned}$$

and this certainly holds if $\mathbf{N} = (\mathbf{M}^{-1})^T$

- Only the upper 3-by-3 portion of $\mathbf{M}$ is pertinent for vectors
- Translation:

$$(\mathbf{M}^{-1})^T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -\Delta x & -\Delta y & -\Delta z & 1 \end{bmatrix} \longrightarrow \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} n_x \\ n_y \\ n_z \\ 0 \end{bmatrix} = \begin{bmatrix} n_x \\ n_y \\ n_z \\ 0 \end{bmatrix}$$

$\implies$ no change to the normal

- Rotation (example):

$$\begin{aligned} (\mathbf{M}^{-1})^T &= \begin{bmatrix} \cos(-\theta) & \sin(-\theta) & 0 & 0 \\ -\sin(-\theta) & \cos(-\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}^T = \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 & 0 \\ -\sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &\longrightarrow \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{aligned}$$

$\implies$ rotation applied unchanged to normal

- Scale:

$$(\mathbf{M}^{-1})^T = \begin{bmatrix} \frac{1}{s_x} & 0 & 0 & 0 \\ 0 & \frac{1}{s_y} & 0 & 0 \\ 0 & 0 & \frac{1}{s_z} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \longrightarrow \begin{bmatrix} \frac{1}{s_x} & 0 & 0 \\ 0 & \frac{1}{s_y} & 0 \\ 0 & 0 & \frac{1}{s_z} \end{bmatrix}$$

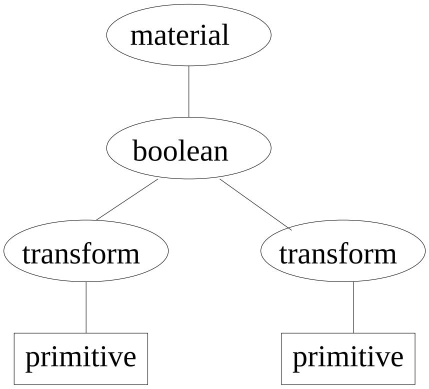
$$\begin{bmatrix} \frac{1}{s_x} & 0 & 0 \\ 0 & \frac{1}{s_y} & 0 \\ 0 & 0 & \frac{1}{s_z} \end{bmatrix} \begin{bmatrix} n_x \\ n_y \\ n_z \\ 0 \end{bmatrix} = \begin{bmatrix} \frac{n_x}{s_x} \\ \frac{n_y}{s_y} \\ \frac{n_z}{s_z} \\ 0 \end{bmatrix}$$

$\implies$ reciprocal scale applied to normal

18.6 Modeling and CSG

Modeling: Constructive Solid Geometry

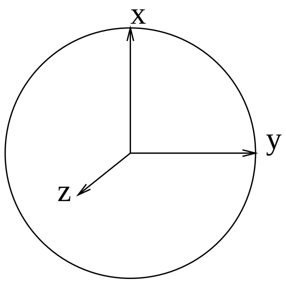
- How do we model for a Ray Tracer?
- Hierarchical modeling works well, but we easily can do more.
- In *Constructive Solid Geometry* all primitives are solids.
- New type of internal node: Boolean operation.
  Intersection, Union, Difference
- Thus, our model is a DAG with
  - Leaf nodes representing primitives
  - Interal nodes are transformations, materials, or boolean operations.

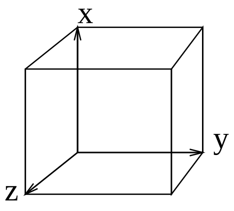


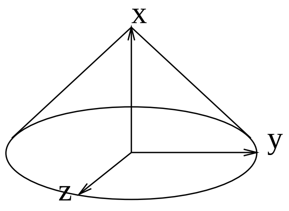
CSG: The primitives

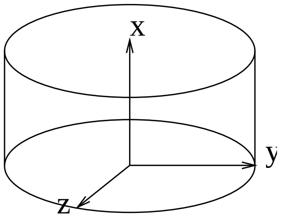
We will want a rich set of primitives. How do we specify them?

- As a "canonical" primitive









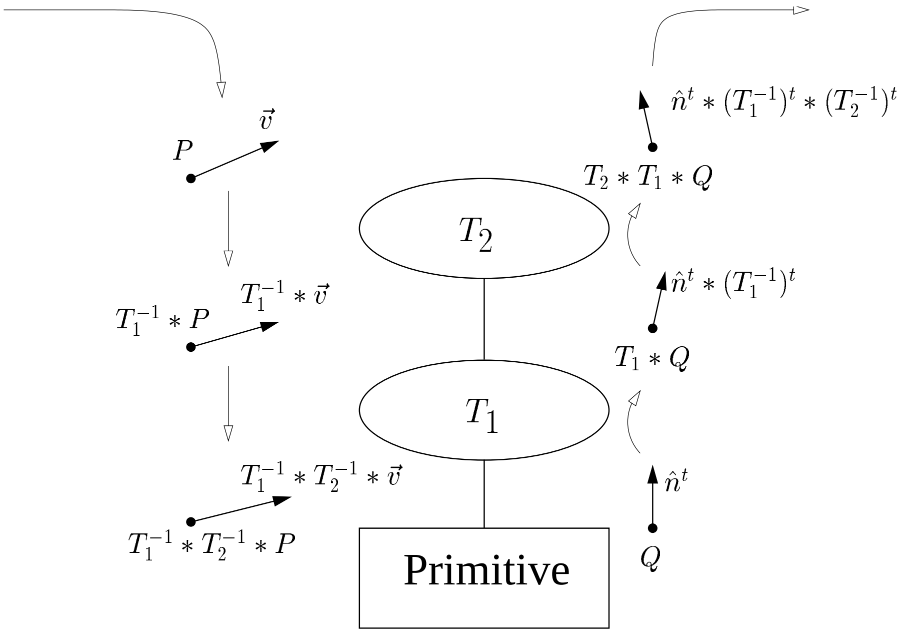
Sphere()*Scale(2,2,2)*Translate(x,y,z)

- As a transformed canonical primitive
  - Sphere(r,x,y,z);
  - Cube(x,y,z,dx,dy,dz);
  - Cone(width,height,x,y,z,dx,dy,dz);

CSG: Traversing the transformation tree

- After transformation are applied, our primitives will be warped.
- Rather than intersect ray with warped primitive (which is worse once we add boolean operations), we will transform the ray.
- On the way down, apply *inverse transformation* to ray.
- On the way back, apply the transformation to the point and to the normal

  Caution: see notes on transforming normals

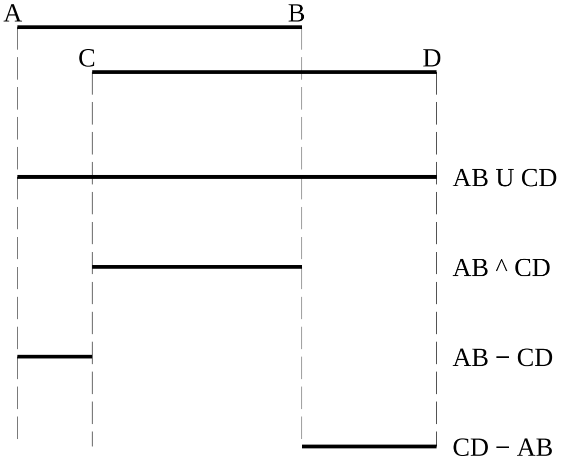

P
$\vec{v}$
$T_1^{-1} * P$
$T_1^{-1} * \vec{v}$
$T_1^{-1} * T_2^{-1} * \vec{v}$
$T_1^{-1} * T_2^{-1} * P$
T_2
T_1
Primitive
$\hat{n}^t * (T_1^{-1})^t * (T_2^{-1})^t$
$T_2 * T_1 * Q$
$\hat{n}^t * (T_1^{-1})^t$
$T_1 * Q$
$\hat{n}^t$
Q

CSG: Boolean operations

- Could apply boolean operations to transformed primitives and construct a surface for that object.

  Problem: representation too complex
- Idea: perform a complete ray intersect object with each primitive. This gives us a (set of) line segment(s).

  Next, perform boolean operations on line segments.



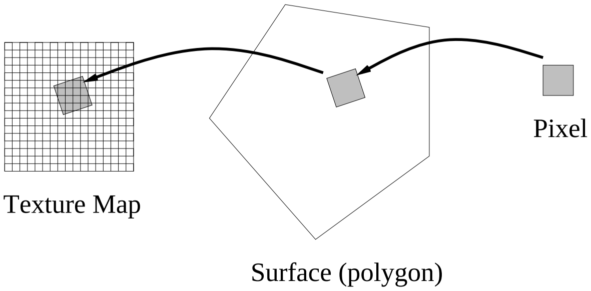
- Note that if we transform the ray on the way down, then we must transform the entire segment on the way back
- Also, we must be careful with normals: they flip for difference.

♠ *Readings: Watt: 2.2.*

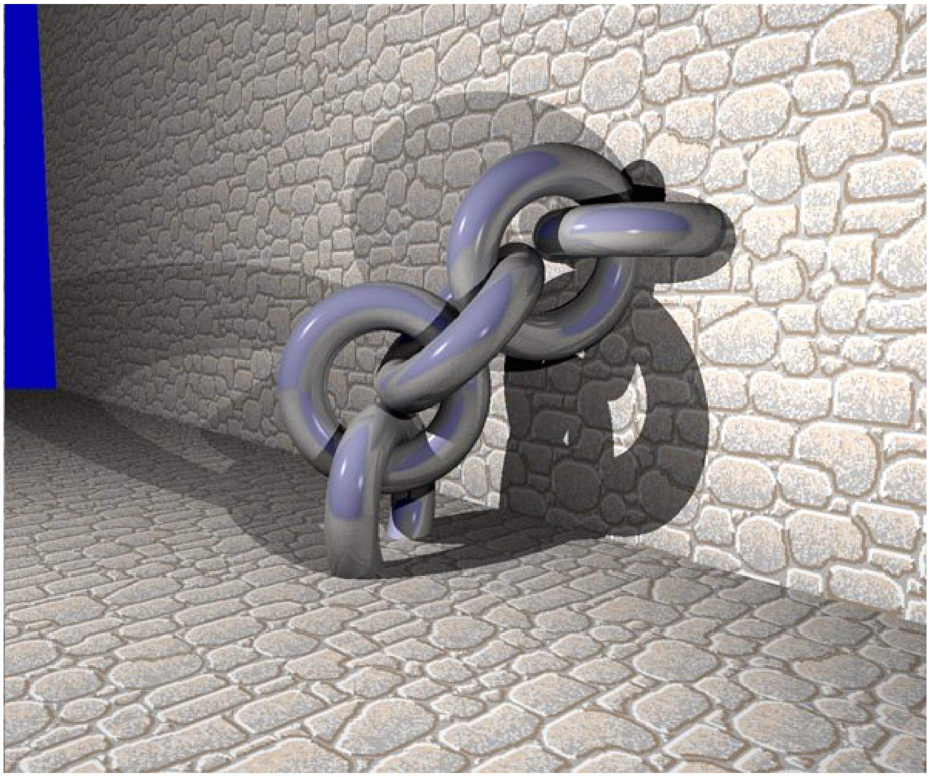
18.7 Texture Mapping

Texture Mapping:

- If we add detail by increasing model complexity, then computation cost increases.
- If detail is surface detail, then we can use texture mapping.
- Idea: scan a photo of the detail and paste it on objects.
    - Associate texture with polygon
    - Map pixel onto polygon and then into texture map
    - Use weighted average of covered texture to compute colour.

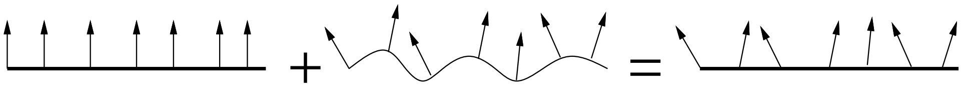


- Tile polygon with texture if needed
    - For some textures, tiling introduces unwanted patern
    - See SIGGRAPH 97 for method to extend textures (expensive)
- Greatly improves images!
- Not just ray traced image

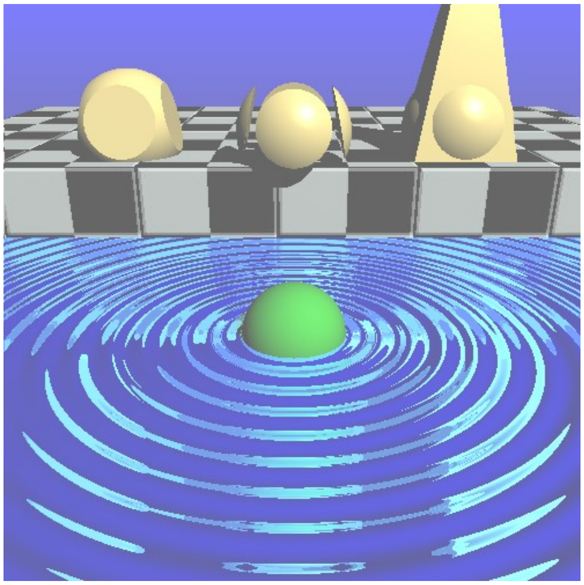


Bump Mapping:

- Textures will still appear "smooth" (i.e., no shadows)
- Bump mapping is similiar to texture mapping except that we peturb the normal rather than the colour.



- Perturbed normal is used for lighting calculation.
- Convincing – usually fail to notice silhouette is wrong.



Solid Textures:

- 2D textures can betray 2D nature in images
- Hard to texture map onto curved surfaces
- Idea: Use a 3D texture instead
- Usually procedural

  Example:

  ```
  if ( (floor(x)+floor(y)+floor(z))%2 == 0 ) then
      return RED;
  else
      return SILVER:
  end
  ```

  Gives a "3D checker board"
- Turbulence can also be simulated (marble)

♠ *Readings: Watt: 8.*

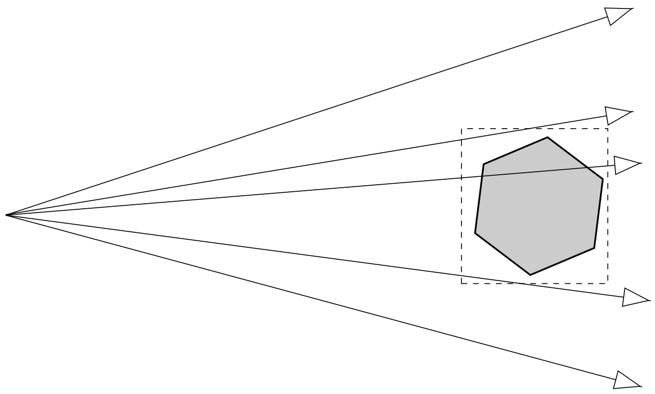
18.8 Bounding Boxes, Spatial Subdivision

Speed-ups

- Ray tracing is slow
- Ray intersect object is often expensive
- Improve speed two ways
  - Reduce the cost of ray intersect object
  - Intersect ray with fewer objects

Bounding Boxes :

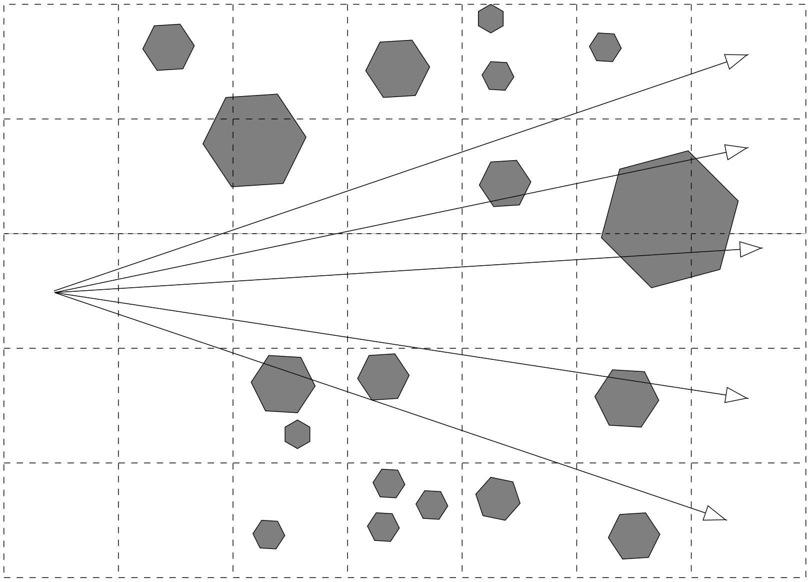
- Idea: Place a bounding box around each object
- Only compute ray intersect object if the ray intersects the bounding box



- If box aligned with coordinate axes, ray intersect box is very cheap
- Be careful with CSG/Hierarchical
- Can also use bounding spheres
- Most useful when ray intersect object is VERY expensive
  Example: polygonal object with lots of facets
- Construction of good bounding box can be difficult

Spatial Subdivision :

- Idea: Divide space in to subregions
- Place objects from scene in to appropriate subregions
- When tracing ray, only intersect with objects in sub-regions through which the ray passes



- Useful when lots of small objects in scene
- Determining subdivision is difficult

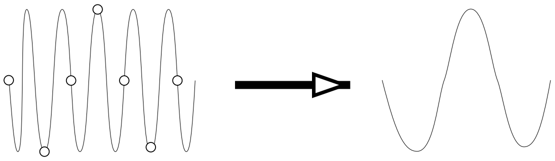
♠ *Readings: Watt: 12.5.*

19 Aliasing and Anti-Aliasing

19.1 Introduction

Signal Processing

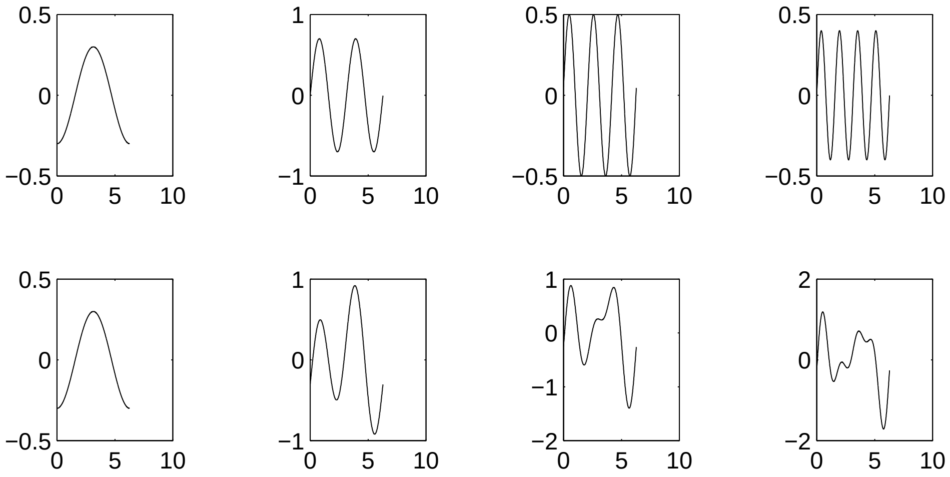
- Raster Image is a sampling of a continuous function
- If samples spaced too far apart, then don't get true representation of scene:



- In graphics, a variety of bad things happen:
  - Stairstep or "jaggies"
  - Moire Patterns
  - Loss of small objects
  - Temporal
    - Sampled too far apart in time
      Backwards rotating wheels
    - Crawling jaggies
    - Appearing/dissappearing objects, flashing objects

Image as a signal :

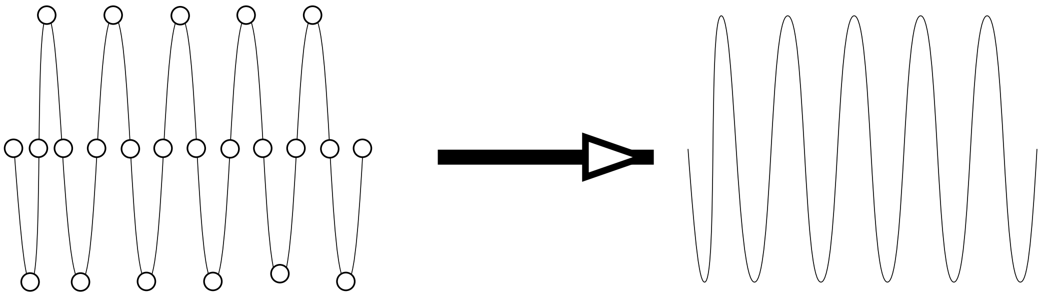
- Signal processing signal is function of time
- Scene is a function of space
  Spatial domain: $f(u)$
- Raster image is a sampling of this signal
- Can represent signal as sum of sine waves
  Frequency domain: $F(u)$



- Regular sampling restricts frequencies at which we sample

Nyquist Limit

- Theory tells us that we must sample at twice the highest frequency in the image to avoid aliasing.
- This sampling rate is known as the *Nyquist Limit*



- Problem: Man made objects have distinct edges
  Distinct edges have infinite frequency

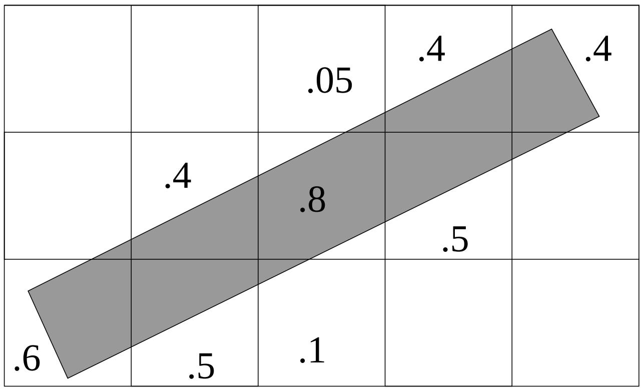
What to do?

- After we get image, run through low pass filter
  This helps, but not a full solution
- Get a Higher Resolution Monitor
  This helps, but...
  - Not usually feasible
  - Alleviates jaggies but
    - Moire patterns merely shifted
    - Small objects still vanish
  - Increased CPU spent on extra pixels can be put to better use.
- Smarter sampling

Area Sampling

Rather than sample, we could integrate.

For lines, this means treating them as boxes



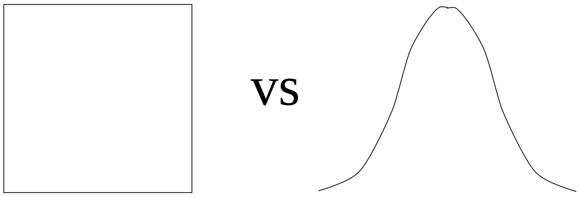
- Colour shades of gray based on fraction of pixel covered
- Gives better looking images *at a sufficiently far distance*
  Look close and it looks blurry

Weighted Sampling

- Unweighted sampling is a box filter

$$I = \int_{x \in \text{Box}} I_x dI$$

  - No contribution outside of pixel
  - All contributions are equal
- Weighted sampling
  - Give different weights depending on position in pixel



  - Filter may extend outside of pixel
    Avoids certain temporal aliasing problems
    "Correct" filter is infinite

Anti-aliasing in Ray Tracing

- Can't always integrate area under pixel
- For ray tracing, we want to point sample the image
- **Super Sampling**
  Take more samples and weight with filter
  If sampling pattern regular, we still get aliasing

- **Stochastic Sampling**
  - Idea: Eye easily detects coherent errors (aliasing)
    Eye is poor at detecting incoherent errors (noise)
  - Rather than regular supersampling, we "jitter" the samples in one of several ways:
    - Choose random location within pixel
    - Displace small, random distances from regular grid

♠ *Readings: Watt: 8 (Introduction), 8.8.*

20 Radiosity-based Global Illumination

20.1 Definitions and Review

Radiosity: Diffuse interaction of light

- Ambient term is an approximation to diffuse interaction of light
- Would like a better model of this ambient light
- Idea: "Discretize" environment and determine interaction between each pair of pieces.
- Models ambient light under following assumptions:
  - Conservation of energy in closed environment
  - Only diffuse reflection of light
- All energy emitted or reflected accounted for by its reflection or absorption elsewhere
- All light interactions computed once, in a view independent way
  - Multiple views can then be rendered just using hidden surface removal
  - No mirror/specular reflections
  - Ideal for architectural walkthroughs

Radiance: Electromagnetic *energy flux*, the amount of energy traveling

- at some point x
- in a specified direction θ,ϕ
- per unit time
- per unit area perpendicular to the direction
- per unit solid angle
- for a specified wavelength λ
- denoted by $L(x, \theta, \phi, \lambda)$

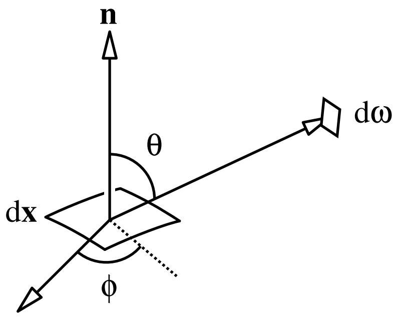
Spectral Properties: Total energy flux comes from flux at each wavelength

- $L(x, \theta, \phi) = \int_{\lambda_{\min}}^{\lambda_{\max}} L(x, \theta, \phi, \lambda) d\lambda$

Picture: For the indicated situation $L(x, \theta, \phi) dx \cos\theta d\omega dt$ is

- energy radiated through differential solid angle $d\omega = \sin\theta d\theta d\phi$
- through/from differential area dx

- not perpendicular to direction (projected area is $dx \cos \theta$)
- during differential unit time dt



Power: Energy per unit time (as in the picture)

- $L(x, \theta, \phi) dx \cos \theta d\omega$

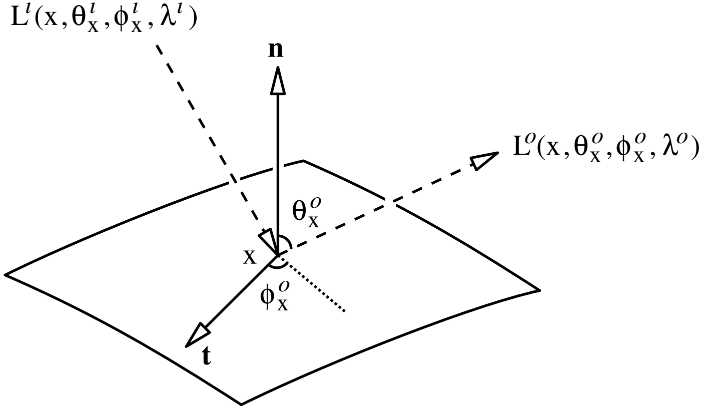
Radiosity: Total power leaving a surface point per unit area

- $\int_\Omega L(x, \theta, \phi) \cos \theta d\omega = \int_0^{\frac{\pi}{2}} \int_0^{2\pi} L(x, \theta, \phi) \cos \theta \sin \theta d\phi d\theta$
(integral is over the hemisphere above the surface point)

Bidirectional Reflectance Distribution Function:

- is a surface property at a point
- relates energy in to energy out
- depends on incoming and outgoing directions
- varies from wavelength to wavelength
- Definition: Ratio
  - of radiance in the outgoing direction
  - to radiant flux density for the incoming direction

$$\rho_{bd}(x, \theta_i, \phi_i, \lambda_i, \theta_o, \phi_o, \lambda_o) = \frac{L^o(x, \theta_x^o, \phi_x^o, \lambda^o)}{L^i(x, \theta_x^i, \phi_x^i, \lambda^i) \cos \theta_x^i d\omega_x^i}$$


$L^i(x, \theta_x^i, \phi_x^i, \lambda^i)$
n
$L^o(x, \theta_x^o, \phi_x^o, \lambda^o)$
θ_x^o
x
ϕ_x^o
t

Energy Balance Equation:

$$\begin{aligned} L^o(x, \theta_x^o, \phi_x^o, \lambda^o) &= L^e(x, \theta_x^o, \phi_x^o, \lambda^o) + \\ & \int_0^{\frac{\pi}{2}} \int_0^{2\pi} \int_{\lambda_{min}}^{\lambda_{max}} \rho_{bd}(x, \theta_x^i, \phi_x^i, \lambda^i, \theta_x^o, \phi_x^o, \lambda^o) \\ & \qquad \cos(\theta_x^i) L^i(x, \theta_x^i, \phi_x^i, \lambda^i) d\lambda^i \sin(\theta_x^i) d\phi_x^i d\theta_x^i \end{aligned}$$

- $L^o(x, \theta_x^o, \phi_x^o, \lambda^o)$ is the radiance
  - at wavelength λ^o
  - leaving point x
  - in direction θ_x^o, ϕ_x^o
- $L^e(x, \theta_x^o, \phi_x^o, \lambda^o)$ is the radiance emitted by the surface from the point
- $L^i(x, \theta_x^i, \phi_x^i, \lambda^i)$ is the incident radiance impinging on the point
- $\rho_{bd}(x, \theta_x^i, \phi_x^i, \lambda^i, \theta_x^o, \phi_x^o, \lambda^o)$ is the BRDF at the point
  - describes the surface's interaction with light at the point
- the integration is over the hemisphere above the point

Radiosity Approach to Global Illumination:

- Assume that all wavelengths act independently
  - $\rho_{bd}(x, \theta_x^i, \phi_x^i, \lambda^i, \theta_x^o, \phi_x^o, \lambda^o) \equiv \rho_{bd}(x, \theta_x^i, \phi_x^i, \theta_x^o, \phi_x^o)$
  - $L(x, \theta_x, \phi_x, \lambda) \equiv L(x, \theta_x, \phi_x)$
- Assume that all surfaces are purely Lambertian
  - $\rho_{bd}(x, \theta_x^i, \phi_x^i, \theta_x^o, \phi_x^o) \equiv \frac{\rho_d(x)}{\pi}$
- As a result of the Lambertian assumption
  - $L(x, \theta_x, \phi_x) \equiv L(x)$
  - $B(x)$ def. $= \int_\Omega L(x) \cos(\theta_x) d\omega = \pi L(x)$

Simple Energy Balance (Hemisphere Based):

$$L^o(x) = L^e(x) + \int_\Omega \frac{\rho_d(x)}{\pi} L^i(x) \cos(\theta_x^i) d\omega$$

Multiplying by π and letting $E(x) = \pi L^e(x)$:

$$B(x) = E(x) + \rho_d(x) \int_\Omega L^i(x) \cos(\theta_x^i) d\omega$$

But, in general

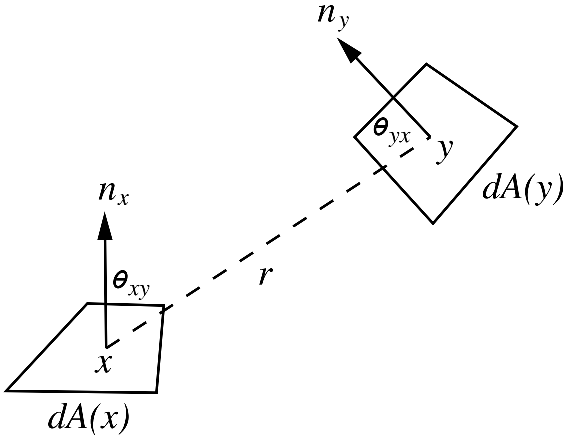
$$L^i(x, \theta_x^i, \phi_x^i) = L^o(y, \theta_y^o, \phi_y^o) \text{ for some } y$$

So we let

$$\theta_x^i = \theta_{xy}, \theta_y^o = \theta_{yx}, \text{ and } L^i(x) = \frac{B(y)}{\pi}$$

for the appropriate point y.

Picture (Scene Based Energy Balance):



Visibility:

We use a term to pick out the special y

$$H(x,y) = \begin{cases} 1 & \text{if } y \text{ is visible from } x \\ 0 & \text{otherwise} \end{cases}$$

Area:

We convert $d\omega$ to surface area

$$d\omega = \frac{\cos\theta_{yx}}{\|x-y\|^2} dA(y)$$

Simple Energy Balance (Scene Based):

$$B(x) = E(x) + \rho_d(x) \int_{y\in\mathcal{S}} B(y) \frac{\cos\theta_{xy}\cos\theta_{yx}}{\pi\|x-y\|^2} H(x,y) dA(y)$$

Piecewise Constant Approximation:

- Approximate the integral by breaking it into a summation over patches
- Assume a constant (average) radiosity on each patch

$$B(y) = B_j \text{ for } y \in P_j$$

$$B(x) \approx E(x) + \rho_d(x) \sum_j B_j \int_{y\in P_j} \frac{\cos\theta_{xy}\cos\theta_{yx}}{\pi\|x-y\|^2} H(x,y) dA(y)$$

- Solve only for a per-patch average density
  - $\rho_d(x) \approx \rho_i$ for $x \in P_i$
  - $B_i \approx \frac{1}{A_i} \int_{x\in P_i} B(x) dA(x)$
  - $E_i \approx \frac{1}{A_i} \int_{x\in P_i} E(x) dA(x)$

Piecewise Constant Radiosity Approximation:

$$B_i = E_i + \rho_i \sum_j B_j \frac{1}{A_i} \int_{x\in P_i} \int_{y\in P_j} \frac{\cos\theta_i \cos\theta j}{\pi\|x-y\|^2} H_{ij} dA_j dA_i$$

Form Factor:

$$F_{ij} = \frac{1}{A_i} \int_{x\in P_i} \int_{y\in P_j} \frac{\cos\theta_i \cos\theta j}{\pi\|x-y\|^2} H_{ij} dA_j dA_i$$

Note, by symmetry, that we have

$$A_i F_{ij} = A_j F_{ji}$$

Linear Equations: (by Summetry)

$$B_i = E_i + \rho_i \sum_j F_{ij} B_j$$

$$B_i = E_i + \rho_i \sum_j F_{ji} \frac{A_j}{A_i} B_j$$

Radiosity: (Summary)

- Idea is to discretize scene into n patches (polygons) each of which emits and reflects light uniformly under its entire area.
- Then radiosity emitted by patch i is given by

$$\begin{aligned} B_i &= E_i + \rho_i \sum_j F_{ij} B_j \\ &= E_i + \rho_i \sum_j F_{ji} \frac{A_j}{A_i} B_j \end{aligned}$$

  where
  - $B_i,\ B_j$: radiosity in energy/unit-time/unit-area
  - E_i: light emitted from patch i
  - ρ_i: patch i's reflectivity
  - $F_{j,i}$: Form factor specifying fraction of energy leaving j that reaches i (accounts for shape, orientation, occulsion)
  - $A_i,\ A_j$: Area of patches

Radiosity: Full Matrix Solution

- The equations are

$$B_i = E_i + \rho_i \sum_{1\le j\le n} B_j F_{ij}$$

  or

$$B_i - \rho_i \sum_{1\le j\le n} B_j F_{ij} = E_i$$

- In matrix form

$$\begin{bmatrix} 1-\rho_1 F_{1,1} & -\rho_1 F_{1,2} & \dots & -\rho_1 F_{1,n} \\ -\rho_2 F_{2,1} & 1-\rho_2 F_{2,2} & \dots & -\rho_2 F_{2,n} \\ \vdots & & \ddots & \vdots \\ -\rho_n F_{n,1} & -\rho_n F_{n,2} & \dots & 1-\rho_n F_{n,n} \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{bmatrix} = \begin{bmatrix} E_1 \\ E_2 \\ \vdots \\ E_n \end{bmatrix}$$

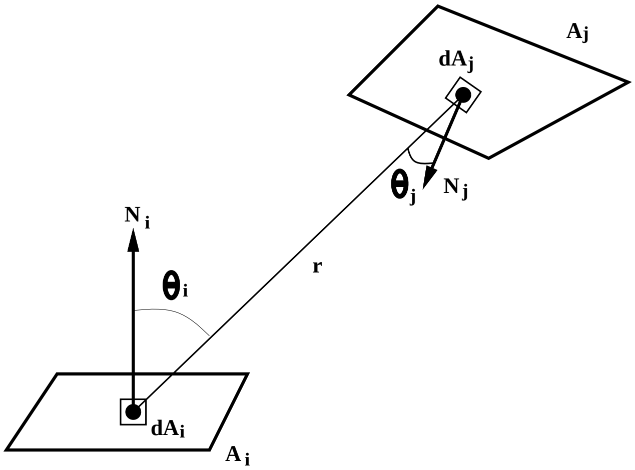
- B_i's are only unknowns
- If $F_{i,i} = 0$ (true for polygonal patches) then diagonal is 1
- Solve 3 times to get RBG values of B_i

♠ *Readings: Watt: Chapter 11.*

20.2 Form Factors

Form Factors: Calculation

- Form factor specifies fraction of energy leaving one patch that (directly) arrives at another patch.



- The differential form factor is given by

$$ddF_{di,dj} = \frac{\cos\theta_i \cos\theta_j}{\pi r^2} H_{ij}\ dA_j\ dA_i$$

where

$$H_{ij} = \begin{cases} 0 & \text{if } dA_i \text{ occluded from } dA_j \\ 1 & \text{if } dA_i \text{ visible from } dA_j \end{cases}$$

Form Factors: Calculation

- To see how much of dA_i illuminates patch j, we integrate:

$$dF_{di,j} = \int_{A_j} \frac{\cos\theta_i \cos\theta_j}{\pi r^2} H_{ij}\ dA_j\ dA_i$$

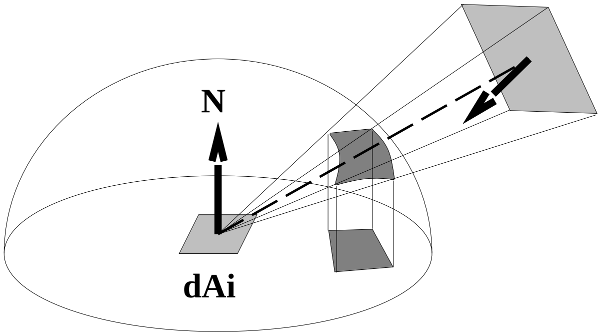
- The form factor from A_i to A_j is an area average of the integral over A_i:

$$F_{i,j} = \frac{1}{A_i} \int_{A_i} \int_{A_j} \frac{\cos\theta_i \cos\theta_j}{\pi r^2} H_{ij}\ dA_j\ dA_i$$

- Typically we approximate this integral:
  - Ray tracing
  - Hemi-sphere
  - Hemi-cube
  - Compute $dF_{di,j}$
- Lots of $F_{i,j}$'s may be zero leading to a sparse matrix

Form Factors: Hemi-sphere Form Factors

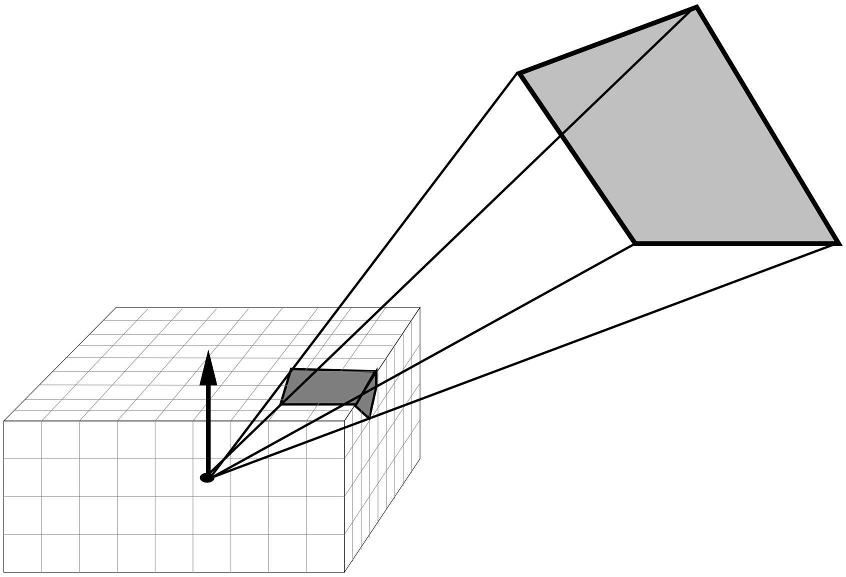
- Place a hemisphere around patch i and project patch j onto it



- Select a point on each patch and look at angles
- Project patch j onto hemisphere, and project projection into circle: $F_{di,j} = \frac{P_A}{H_A}$ where H_A is the area of the circle.
- In the integral for form factors
  - projecting onto the hemisphere accounts for $\cos(\theta_j)/r^2$
  - projecting onto circle accounts for $\cos(\theta_i)$
  - divide by area of circle accounts for π
- Note that any polygon with same "cross section" has same form factor
- Projecting onto hemisphere is still hard

Form Factors: Hemi-cube Form Factors

- Project onto a hemi-cube
- Subdivide faces into small squares

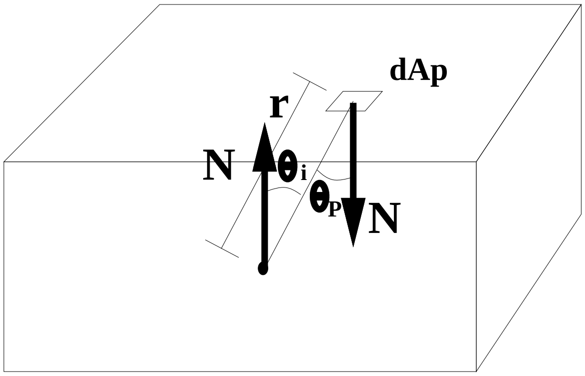


- Now determine which squares patch j projects onto

Form Factors: Delta Form Factors

- Each hemi-cube cell P has pre-computed delta form factor

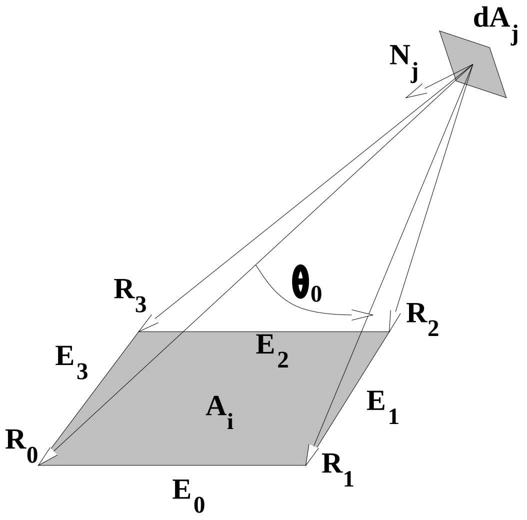
$$\Delta F_P = \frac{\cos(\theta_i)cos(\theta_P)}{\pi r^2}\Delta A_P$$



- Approximate $dF_{dj,i}$ by summing delta form factors
- If distance from i to j is large, this is good approximation for $F_{j,i}$.
- Plus! Use z-buffer and do all patches at once to get shadows.
  Have to render all patches 5 times (once for each side) using center of patch i as viewpoint.
  Hack!! When scan converting a patch, use the pointer to that patch for its "colour"

Form Factors: Exact Form Factors

- We can compute the exact form factor for point to polygon



The differential form factor is given by $F_{dA_j A_i} = \frac{1}{2\pi} \sum_i N_j \cdot \Gamma_i$ where

- N_j is the normal to patch j
- Γ_i is the vector in direction $R_{i+1} \times R_i$ with length equal to θ_i (in radians).

- Doesn't account for occlusion
- Also an exact polygon to polygon form factor but...

20.3 Progressive Refinement

Progressive Refinement: Basics

- Equivalent to Southwell's relaxation method
- Recall our equation says how much radiosity is comes from patch j to patch i:

$$B_i \text{ due to } B_j = \rho_i B_j F_{ji}$$

Light is being *gathered* at patch i.

- Instead we can ask "how much radiosity" is *shot* from patch i to patch j?

$$\begin{aligned} B_j \text{ due to } B_i &= \rho_j B_i F_{ij} \\ &= \rho_j B_i F_{ji} \frac{A_j}{A_i} \end{aligned}$$

- Idea: Choose a patch, shoot its radiosity to all other patches, repeat.

Progressive Refinement: Code

Keep track of total radiosity (B) and unshot radiosity (dB)

```
Procedure Shoot(i)
For all j, calculate all Fji
Foreach j
    drad = pj*dBi*Fji*Ai/Aj
    dBj += drad
    Bj += drad
endfor
dBi = 0
```

Call from

```
while (unshot > eps) do
    Choose patch i with highest dB
    Shoot(i)
end
```

Progressive Refinement: Analysis

- Shoot lights first
- Can stop and look at partial results
- Can be formulated as a matrix equation
- *Lots* of cute tricks:
  - Add initial ambient to all patches so that initial results more viewable
  - Can move objects and lights after calculation has started
- Technically it is slower than full matrix methods, specially for environments with high average reflectances and high level of occlusion
- Form factors are usually not stored since they are computed on the fly
  - Advantage: save storage space
  - Disadvantage: some form factors may need to be computed multiple times

Radiosity Issues

Questions:

- How big should we make initial patches?
- When do we stop Progressive-Refinement?

Problems

- T-vertices and edges cause light/shadow leakage
- Occlusions and shadows hard to model
- Form factor estimations often cause aliasing

Extensions

- Methods for adding specular (but *expensive*)
- Substructuring improves quality without increasing big-O
- Smarter substructuring reduces big-O

20.4 Meshing in Radiosity

Issues

- Initial Mesh Size?
- T-vertices, etc?
- Shadow Boundaries
- Soft Shadows

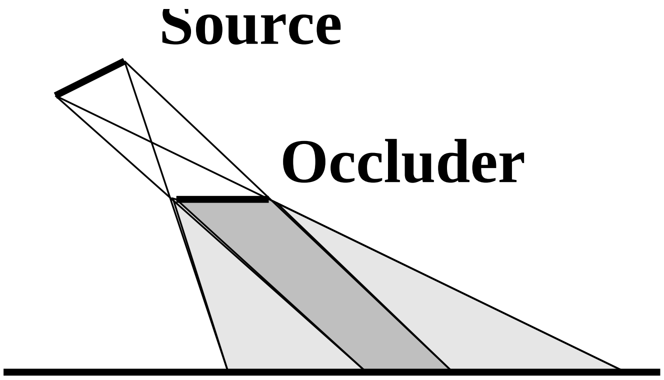
Two Techniques:

- Split along shadow boundaries
- Adaptive Subdivision

Split Along Shadow Boundaries – 2D

3 regions:

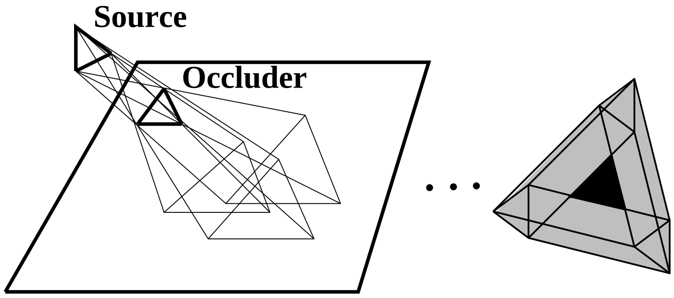
- Lit
- Unlit
- Linear



Split Along Shadow Boundaries – 3D

Lots of regions:

- Lit
- unlit
- Linear
- Quadratic



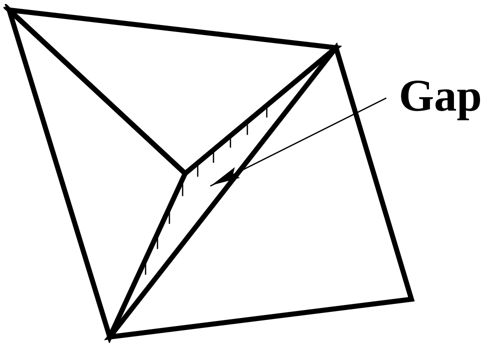
Adaptive Subdivision

- Make initial Meshing Guess
- Point sample a patch
- Average and check variance
  - Low ⇒ Good estimate
  - High ⇒ Poor estimate
    Subdivide and repeat

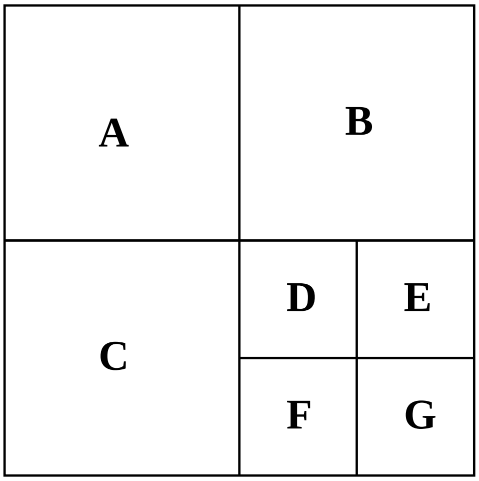
Problems with Adaptive Subdivision

- Radiosity has $O(n^2)$ run time
  Subdivision increases n
- Ideas:
  - Shoot from big patches
  - Receive at small patches

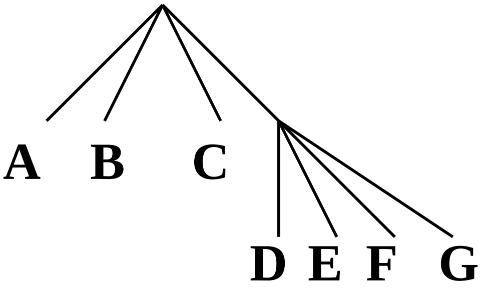
  Sub problems:
  - sub-patch to big-patch association
  - T-vertices



Quad Tree



- Store as



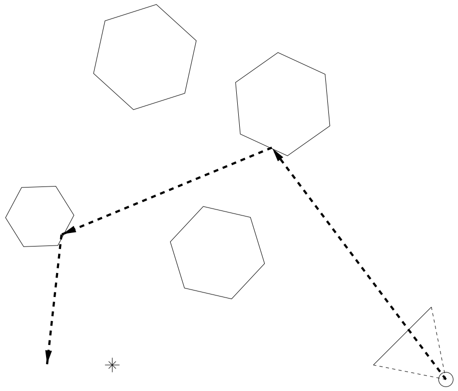
- Traverse up to find parent
- Traverse up-down to find neighbor
- Anchoring

21 Bidirectional Tracing

21.1 Missing Effects

- Problem: Raytracing and radiosity can't model all effects

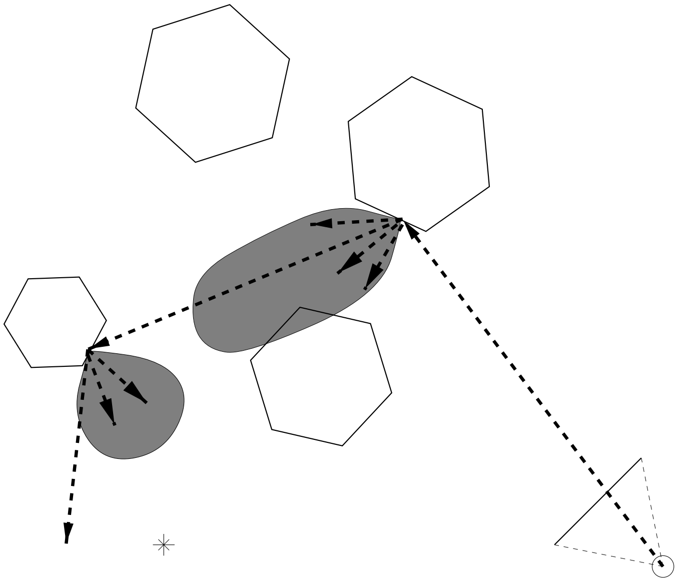
  Caustics



- Some effects requiring tracing light rays from light back to eye
- Too many to trace them all. How do we select?
- We'll look at four techniques:
  - Distribution ray tracing
  - Photon maps
  - Bidirectional path tracing
  - Metropolis Light Transportation Algorithm

21.2 Distribution Ray Tracing

- Ray trace, but at each reflection cast a lot of rays
- Cast with distribution function to represent reflection properties
- Will model caustics, etc, but need *lots* of rays
  (10+ rays per reflection)



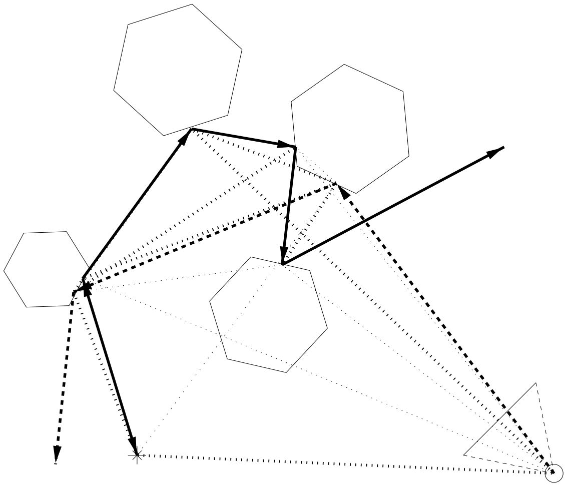
21.3 Photon Maps

- Cast rays from light
  Global illumination (view independent)
- Create a "photon map" wherever this light hits
- The use standard ray *casting*,
  no secondary rays
  look at photon map(s) to determine special lighting
- Use density estimation to convert photon map to intensity
- Fast, easy to program, low noise
- Errors

21.4 Bidirectional Path Tracing

- Idea: trace paths from eye and from light and connect them up
  Trace paths (single reflected ray)
- Reflect with distribution function

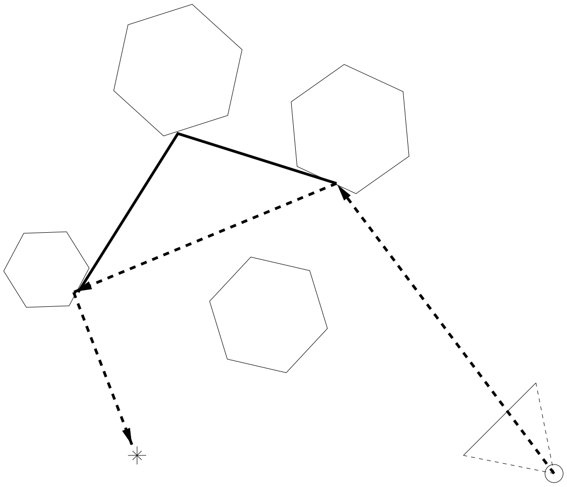
- When far enough, connect them up
- Problem: A whole lot of rays



- Low error, high noise
- Less expensive than distribution ray tracing

21.5 Metropolis Algorithm

- Problem with Monte Carlo bidirection path tracing:
  Need lots of ray (Very expensive, Noisy results)
- Metropolis Light Transport Algorithm reduces the cost
- Idea: Take one path from light to eye
  Perturb this *path* and see if it still reaches the eye.



- Low error, low noise
- New, untested

22 Polyhedral Data Structures

22.1 Ray Tracer formats

- How do we store a polyhedron in memory?

  Must consider the following:

  - Memory
  - Efficiency

    Operations

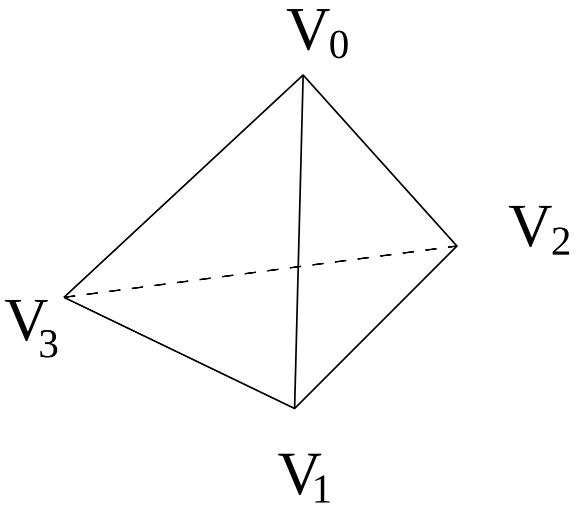
- Will look at three formats

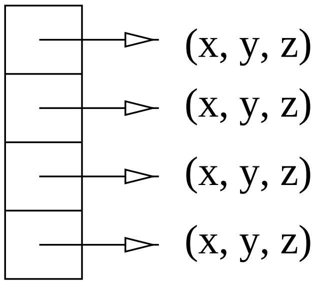
  - Ray Tracer
  - Generalized Ray Tracer
  - Winged-edge variations

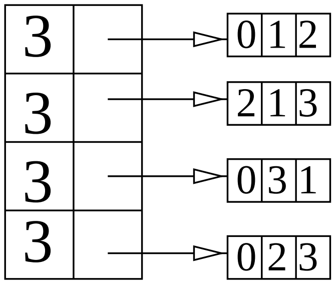
Ray Tracer Format

- We already saw one ASCII format (for the ray tracer)
- Data structure for this is simple:

  - Array of points (triples of floats)
  - Array of faces, with pointers to points







Compact, simple to get face information

- Advantages

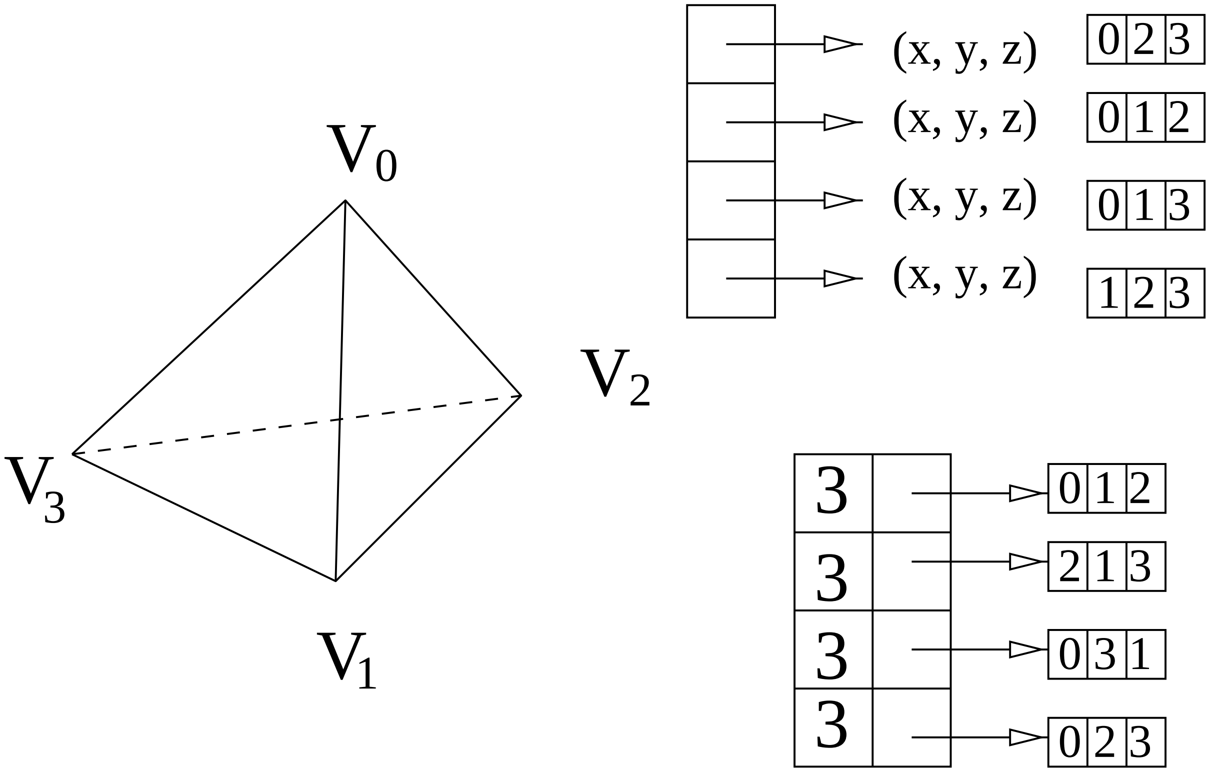
  - Space efficient
  - Easy to generate, parse, build data structure

- Disadvantages
  - Vertex neighbors?
  - Modify the data structure?

  Example: What if we want to know faces surround a vertex?

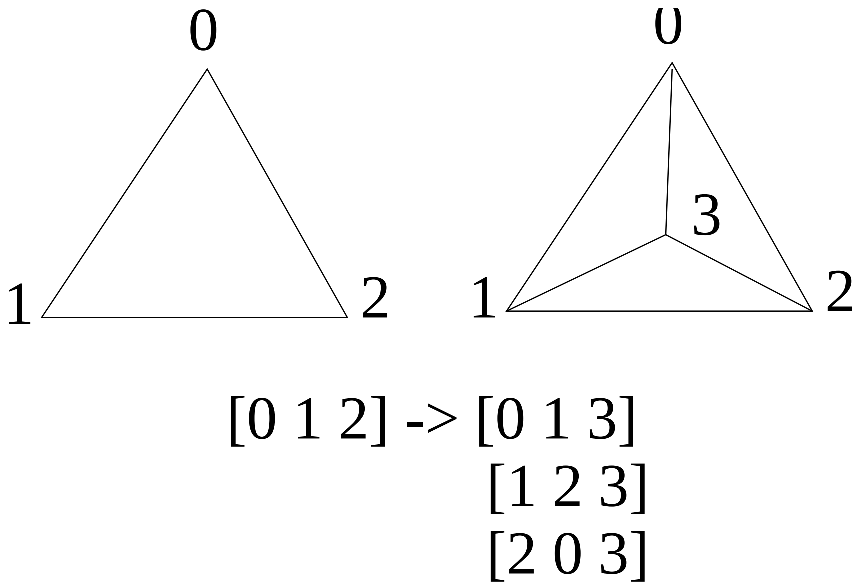
  Radiosity might want this

- Solution: For each point, have a list of pointers to surrounding faces.
  - Can use same ASCII input format
  - Build lists when reading data



Question: Where do these lists come from?

- Problems still remain:
  - How do we modify a polyhedron?
  - Easy to move vertex
  - Add face or vertex is easy
  - Split face not too hard



- Delete is hard
  $O(n)$ or leaves holes in our list

♠ *Readings: Watt: 2.1.*

22.2 Euler's Formula

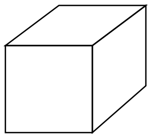
Euler's Formula

- Closed polyhedron:

$$V - E + F - H = 2(C - G)$$

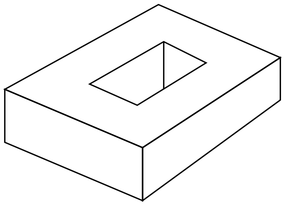
  - V: Number of vertices
  - E: Number of edges
  - F: Number of faces
  - H: Number of holes
  - C: Number of connected components
  - G: Number of genus (sphere = 0, torus = 1)
- Examples:

Cube



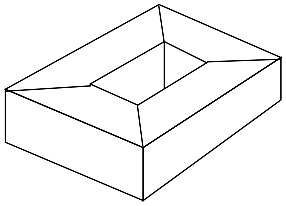
V= E= F= G= C= H=

Torus 1



V= E= F= G= C= H=

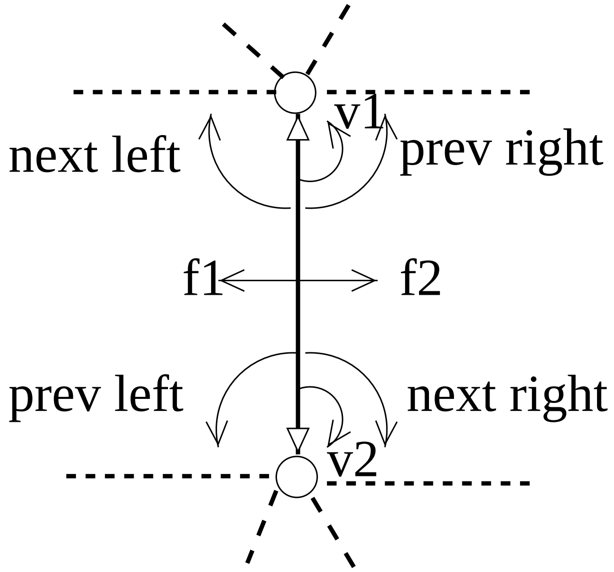
Torus 2



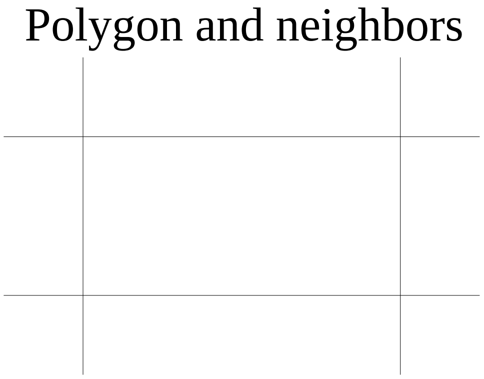
V= E= F= G= C= H=

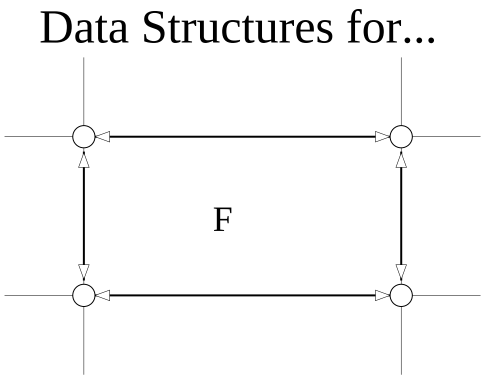
22.3 Winged Edge

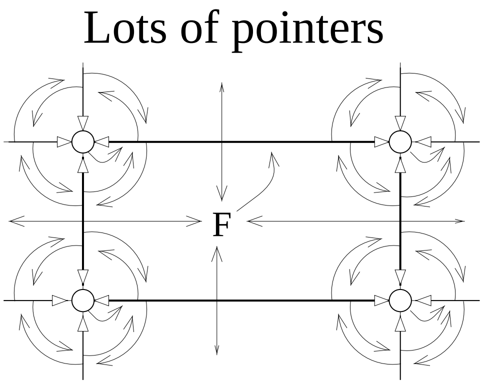
- Operations:
  - Want to traverse
    - Neighboring faces of a face/vertex
    - Neighboring vertices of a face/vertex
    - Edges of a face
  - Want to modify (add/delete) the polyhedron
- Key ideas of winged-edge:
  - Edge is important topological data structure
  - Also, separate topology from geoemtry
- Edge is the primary data structure



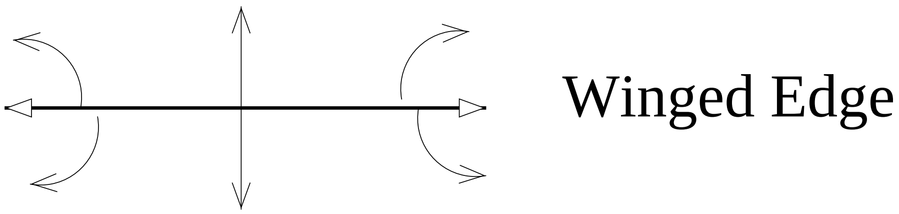
- Pictorially:

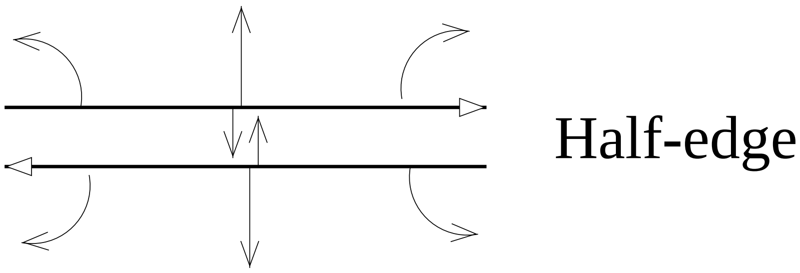

Polygon and neighbors

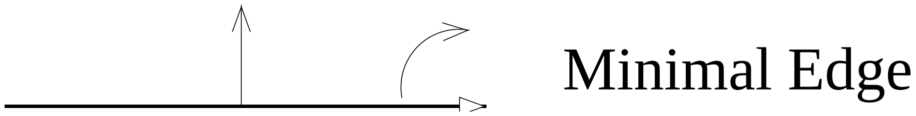

Data Structures for...


Lots of pointers

- Variations:
  - Some operations simplified by splitting edge into half-edges
  - Uses a lot of space. Time-space trade-off.







- Data types:

  ```
  struct he {
    struct he* sym;
    struct he* next;
    struct he* prev;
    struct vert* v;
    struct face* f;
    void* data;
  }
  struct vert {
    struct he* rep;
    void* data;
  }
  struct face {
    struct he* rep;
    void* data;
  }
  ```

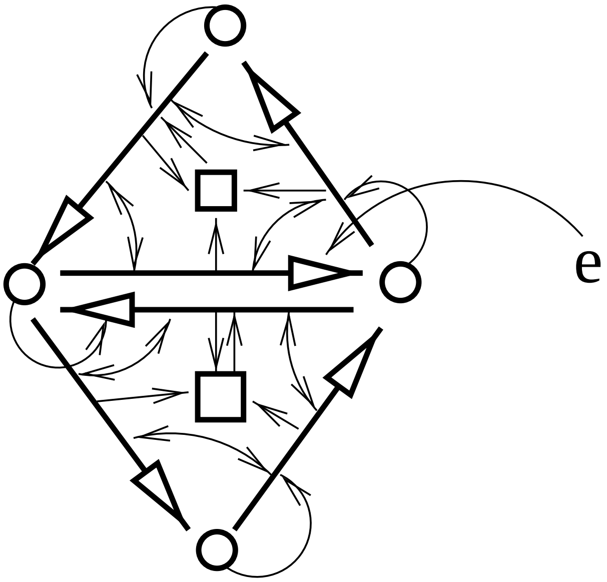
- Example: Neighboring faces of `f`

  (picture)

  ```
  struct face* f;
  struct he* e;
  ```

```
struct he* s;
struct he* fl[LOTS];
int i;

s = e = f->rep;
i = 0;
do {
  fl[i++] = e->sym->f;
  e = e->next;
} while (e != s);
```

- Example: Removing an edge



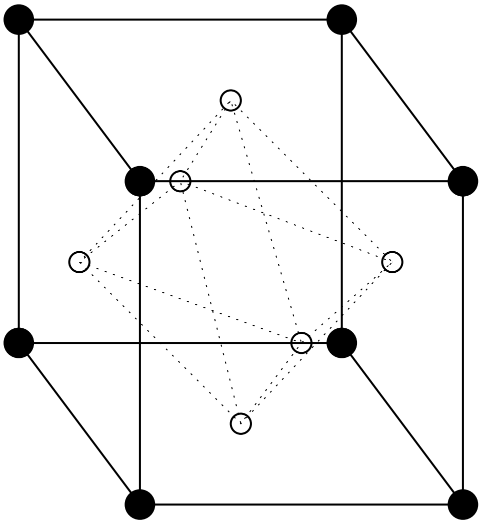
```
RemoveEdge(struct he* e) {
  /* fix edge->edge pointers */
  e->prev->next = e->sym->next;
  e->next->prev = e->sym->prev;
  e->sym->next->prev = e->prev;
  e->sym->prev->next = e->next;

  /* fix edge->face pointers */
  ee = e->sym->next;
  while (ee != e->sym) {
    ee->f = e->f;
    ee = ee->next;
  }

  /* fix face->edge pointers */
  e->f->rep = e->prev;
```

```
    /* fix vertex->edge pointers */
    e->v->rep = e->sym->prev;
    e->sym->v->rep = e->prev;

    DeleteFace(e->sym->f);
    DeleteEdge(e);
  }
```

- Can you spot the bug in the above code?
- Manifolds with Boundary
  - Half-edge allows representation of manifold w/boundary
    Set sym pointer to NULL
  - Allows for lone face
  - Makes some constructions easier
  - New Euler formula

    $$V - E + F - H - B = 2(C - G)$$

    where B is number of boundaries
- Duality
  - Faces and vertices are dual
    Representations and rolls are identical
  - Can replace each face with a vertex and each vertex with a face



  - If boundaries, then dual may not be manifold
- Evaluation

  + Faces with arbitrary number of sides
    Vertices with arbitrary number of neighbors

+ Iterate over everything

– Not space efficient

– Painful to write/debug code

** Simpler data structure better unless you need to modify, iterate

- API
  - Discussions in papers use low level operators
  - Better if user of package does NOT manipulate pointers
  - Higher level “safe” operators, iterators:
    - `ForeachMeshVertex`
    - `ForeachFaceVertex`
    - `Split 3-1`
    - `RemoveEdge`
  - Example: Split all faces 3-1

```
ForeachMeshFace(m,f) {
  p = SplitFace3to1(f);
  SetPosition(P,x,y,z);
}
```

  - Which operations should we provide?

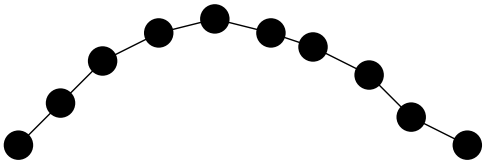
23 Splines

♠

Motivation

- *Could represent curves as piecewise line segments*
- *Problem: Too little data*

- *Problem: Too much data*



- *Want compact representation of curve that can be drawn at different resolutions as needed*
- *Want representation to have geometric meaning for modeling purposes*

- *Polynomials are curve of choice*
    - *PostScript (Adobe)*
    - *TrueFont (MicroSoft)*

Spline Curves

- *Succesive linear blends*
- *Basis polynomials*
- *Recursive evaluation*
- *Properties*
- *Joining segments*

Tensor-product-patch Spline Surfaces

- *Tensor product patches*
- *Evaluation*
- *Properties*
- *Joining patches*

Triangular-patch Spline Surfaces

- *Coordinate frames and barycentric frames*
- *Triangular patches*

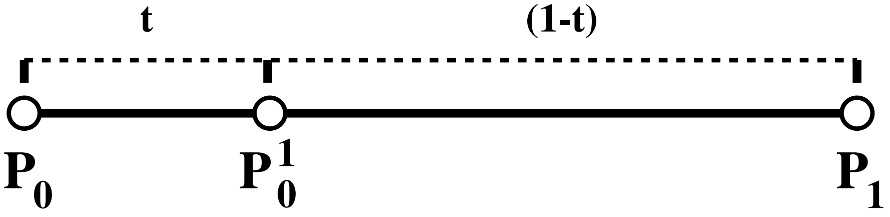
Subdivision Surfaces

23.1 Constructing Curve Segments

Linear blend:

- Line segment from an affine combination of points

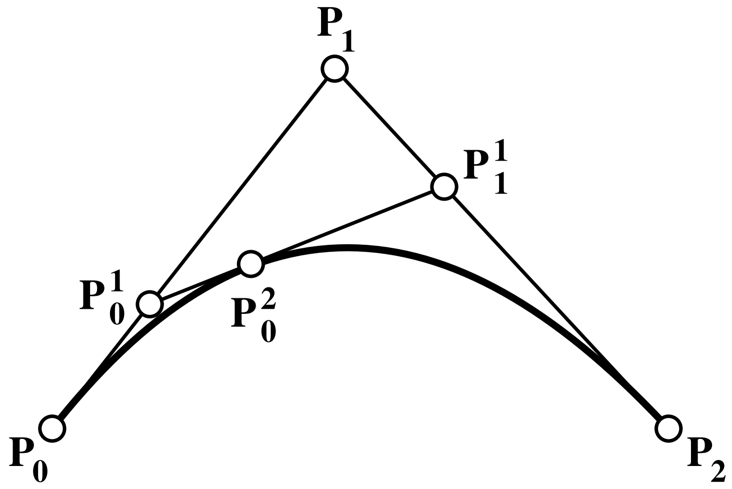
$$P_0^1(t) = (1-t)P_0 + tP_1$$



Quadratic blend:

- Quadratic segment from an affine combination of line segments

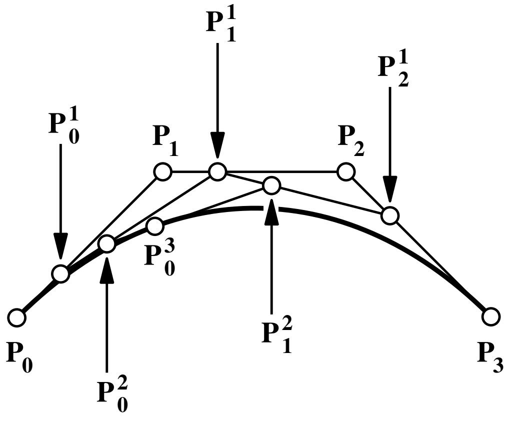
$$\begin{aligned}
P_0^1(t) &= (1-t)P_0 + tP_1\\
P_1^1(t) &= (1-t)P_1 + tP_2\\
P_0^2(t) &= (1-t)P_0^1(t) + tP_1^1(t)
\end{aligned}$$

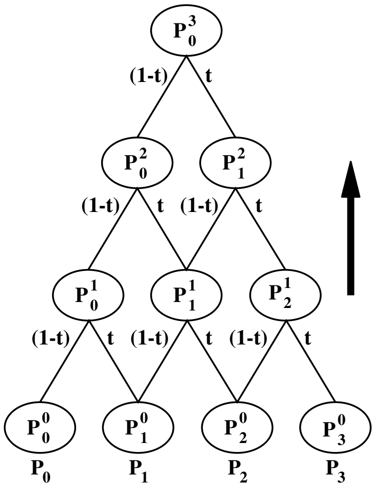


Cubic blend:

- Cubic segment from an affine combination of quadratic segments

$$\begin{aligned}
P_0^1(t) &= (1-t)P_0 + tP_1\\
P_1^1(t) &= (1-t)P_1 + tP_2\\
P_0^2(t) &= (1-t)P_0^1(t) + tP_1^1(t)\\
P_1^1(t) &= (1-t)P_1 + tP_2\\
P_2^1(t) &= (1-t)P_2 + tP_3\\
P_1^2(t) &= (1-t)P_1^1(t) + tP_2^1(t)\\
P_0^3(t) &= (1-t)P_0^2(t) + tP_1^2(t)
\end{aligned}$$

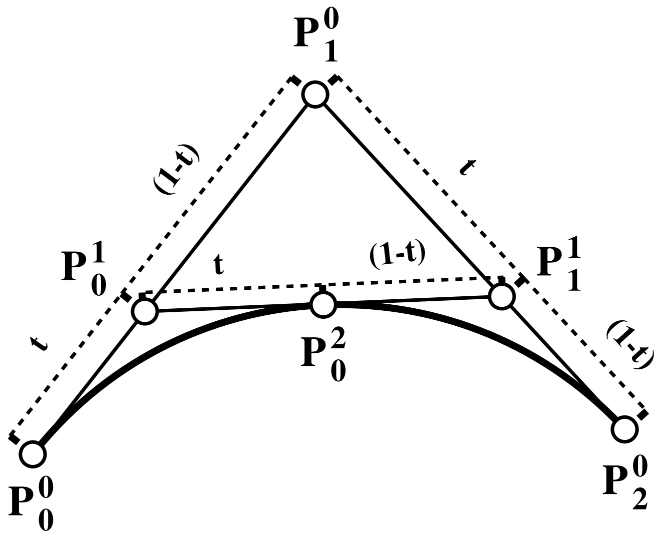




- The pattern should be evident for higher degrees.

Geometric view (deCasteljau Algorithm):

- Join the points P_i by line segments
- Join the $t : (1-t)$ points of those line segments by line segments
- Repeat as necessary
- The $t : (1-t)$ point on the final line segment is a point on the curve
- The final line segment is tangent to the curve at t



Expanding Terms (Basis Polynomials):

- The original points appear as coefficients of *Bernstein polynomials*

$$\begin{aligned}
P_0^0(t) &= 1 \cdot P_0 \\
P_0^1(t) &= (1-t)P_0 + tP_1 \\
P_0^2(t) &= (1-t)^2 P_0 + 2(1-t)tP_1 + t^2 P_2 \\
P_0^3(t) &= (1-t)^3 P_0 + 3(1-t)^2 tP_1 + 3(1-t)t^2 P_2 + t^3 P_3 \\
\\
P_0^n(t) &= \textstyle\sum_{i=0}^n P_i B_i^n(t)
\end{aligned}$$

$$\text{where} \qquad B_i^n(t) = \tfrac{n!}{(n-i)!i!}(1-t)^{n-i}t^i = \begin{pmatrix} n \\ i \end{pmatrix} (1-t)^{n-i}t^i$$

- The Bernstein polynomials of degree n form a basis for the space of all degree-n polynomials

♠ *Readings: Watt: Chapter 3.*

23.2 Bézier Splines

Bezier Curve Segments and their Properties

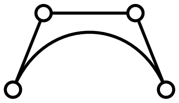
Definition:

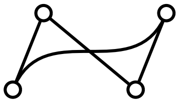
A degree n (order $n+1$) **Bézier curve segment** is

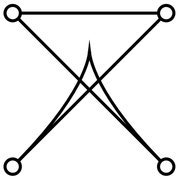
$$P(t) = \sum_{i=0}^{n} P_i B_i^n(t)$$

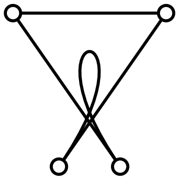
where the P_i are k-dimensional **control points**.

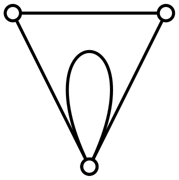
Examples:





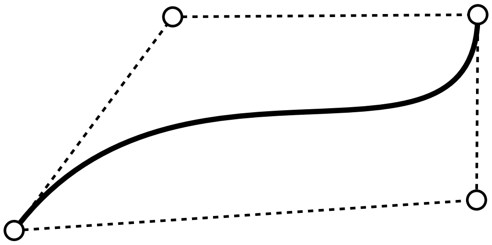






Convex Hull:

$\sum_{i=0}^{n} B_i^n(t) = 1$, $B_i^n(t) \geq 0$ for $t \in [0,1]$
$\Longrightarrow P(t)$ is a convex combination of the P_i for $t \in [0,1]$
$\Longrightarrow P(t)$ lies within convex hull of P_i for $t \in [0,1]$.



Affine Invariance:

A Bézier curve is an affine combination of its control points.
Any affine transformation of a curve is the curve of the transformed control points.

$$T\left(\sum_{i=0}^{n} P_i B_i^n(t)\right) = \sum_{i=0}^{n} T(P_i) B_i^n(t)$$

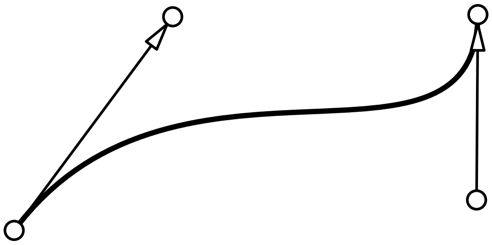
This property does not hold for projective transformations!

Interpolation:

$B_0^n(0) = 1$, $B_n^n(1) = 1$, $\sum_{i=0}^{n} B_i^n(t) = 1$, $B_i^n(t) \geq 0$ for $t \in [0,1]$
$\Longrightarrow B_i^n(0) = 0$ if $i \neq 0$, $B_i^n(1) = 0$ if $i \neq n$
$\Longrightarrow P(0) = P_0$, $P(1) = P_n$.

Derivatives:

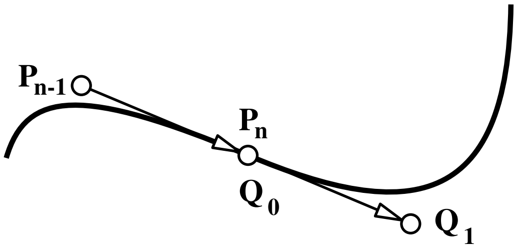
$\frac{d}{dt}B_i^n(t) = n(B_{i-1}^{n-1}(t) - B_i^{n-1}(t))$
$\Longrightarrow P'(0) = n(P_1 - P_0)$, $P'(1) = n(P_n - P_{n-1})$.



Smoothly Joined Segments ($\mathbf{C^1}$): :

Let P_{n-1}, P_n be the last two control points of one segment.
Let Q_0, Q_1 be the first two control points of the next segment.

$$\begin{aligned} P_n &= Q_0 \\ (P_n - P_{n-1}) &= (Q_1 - Q_0) \end{aligned}$$



Smoothly Joined Segments ($\mathbf{G^1}$):

$$\begin{aligned} P_n &= Q_0 \\ (P_n - P_{n-1}) &= \beta(Q_1 - Q_0) \text{ for some } \beta > 0 \end{aligned}$$

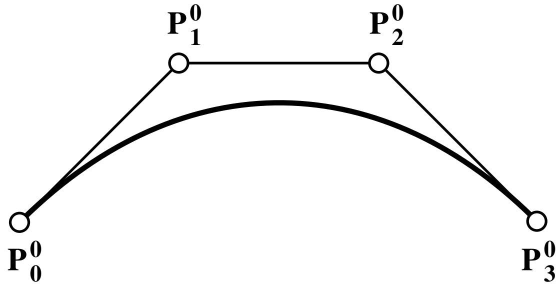
Recurrence, Subdivision:

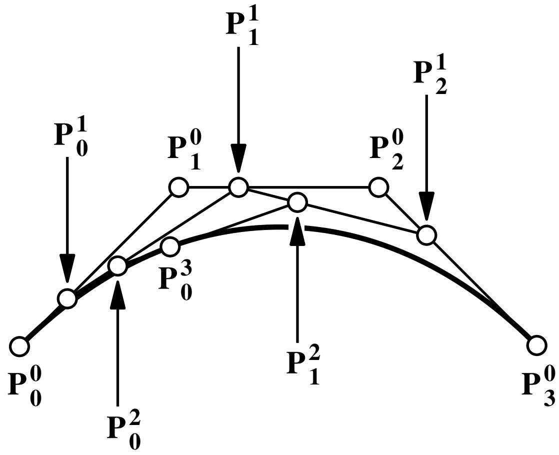
$B_i^n(t) = (1-t)B_i^{n-1} + tB_{i-1}^{n-1}(t)$
$\Longrightarrow$ de Casteljau's algorithm:

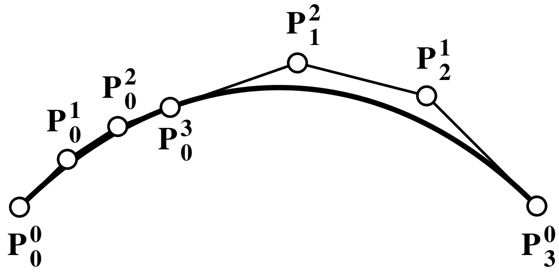
$$\begin{aligned} P(t) &= P_0^n(t) \\ P_i^k(t) &= (1-t)P_i^{k-1}(t) + tP_{i+1}^{k-1} \\ P_i^0(t) &= P_i \end{aligned}$$

Use to evaluate point at t, or subdivide into two new curves:

- P_0^0, P_0^1, $\ldots P_0^n$ are the control points for the left half;
- P_n^0, P_{n-1}^1, $\ldots P_0^n$ are the control points for the right half







23.3 Spline Continuity

Polynomials Inadequate

- Weierstrass Approximation Theorem: Can approximate any C^0 curve to any tolerance with polynomials

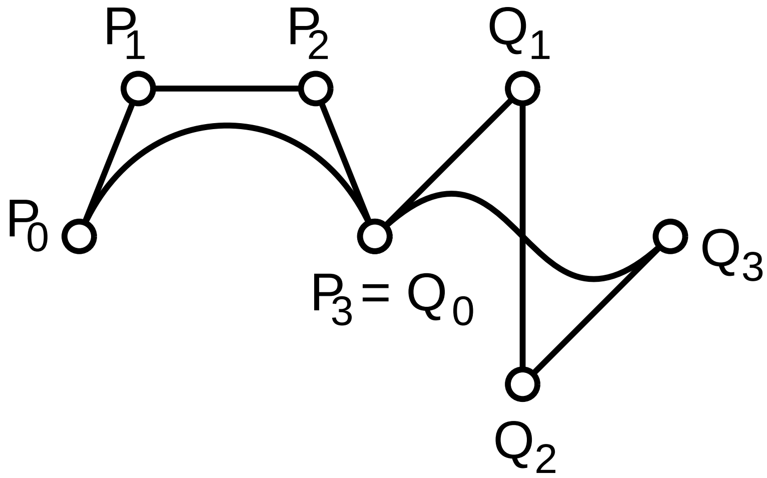
  But may require high degree
- To model complex curves, will need high degree
- High degree:
  - Non-local
  - Expensive to evaluate
  - Slow convergence as we increase degree

Piecewise Polynomials

- Idea: Instead of 1 high degree polynomial, piece together lots of low degree polynomials
- Benefits:
  - Fast to evaluate
  - Good convergence properties
  - Potentially local
- Problem:
  - Continuity between pieces

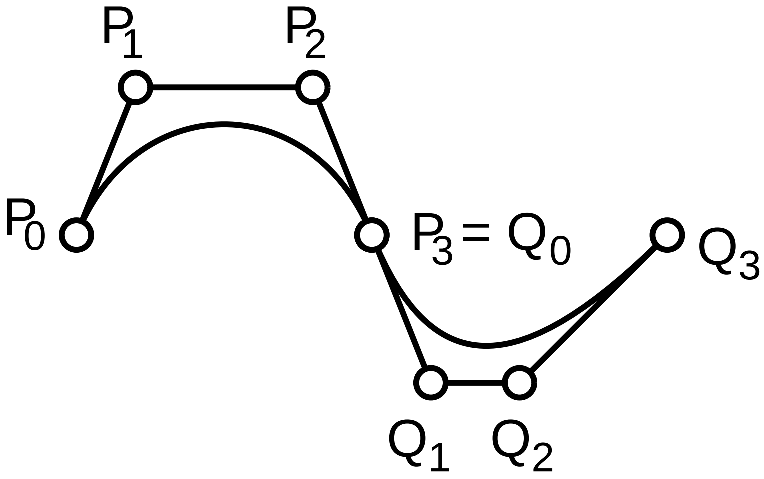
C^0 Piecewise Cubic Bézier

- Piecewise cubic
- Bézier form
- Bézier Property: curve interpolates first control point
- Bézier Property: curve interpolates last control point
- C^0: Set last control point of one segment to be first control point of next segment



C^1 Piecewise Cubic Bézier

- Need parameterizations of intervals to discuss C^1 continuity
- Assume uniform parameterization
  $[0, 1]$, $[1, 2]$, $[2, 3]$, ...
- Need C^0 to have C^1
  $P_3 = Q_0$
- C^1 is $P_3 - P_2 = Q_1 - Q_0$



Cubic Hermite Interpolation

- Problem: Given points $P_0, \ldots, P_N$ and vectors $\vec{v}_0, \ldots, \vec{v}_N$
  Find: Piecewise C^1 cubic P s.t.

$$\begin{aligned} P(i) &= P_i \\ P'(i) &= \vec{v}_i \end{aligned}$$

  for $i = 0, \ldots, N$
- Solution: Use one cubic Bézier segment per interval
  Then

$$\begin{aligned} P_{i,0} &= P_i \\ P_{i,1} &= P_i + \frac{\vec{v}_i}{3} \\ P_{i,2} &= P_{i+1} - \frac{\vec{v}_{i+1}}{3} \\ P_{i,3} &= P_{i+1} \end{aligned}$$

- (Cubic Hermite Basis functions)

Catmull-Rom Splines

- When modeling, specifying derivatives can be either good or bad, but more commonly bad
- Want to specify just points
- Idea: make up derivatives from points, and use Cubic Hermite
- For $i = 1, \ldots, N-1$ let

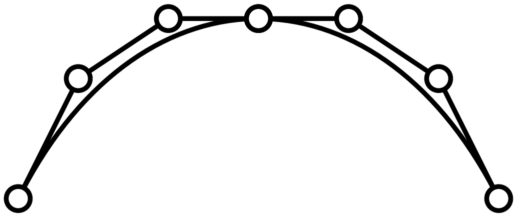
$$\vec{v}_i = P_{i+1} - P_{i-1}$$

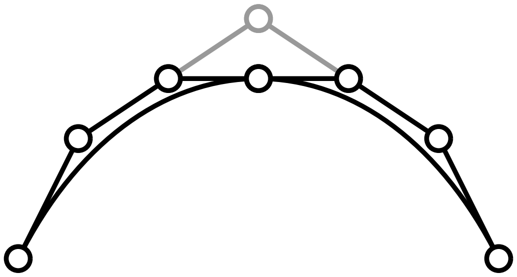
- Need derivatives for $i = 0, N$.
  - Discard data
  - Set to 0

    - Set to $P_1 - P_0$ (perhaps scaled)

C^2 Piecewise Cubic Bézier

- With cubics, we can have C^2 continuity (why not C^3?)
- A-frame construction gives C^2 constraints on Bézier segments

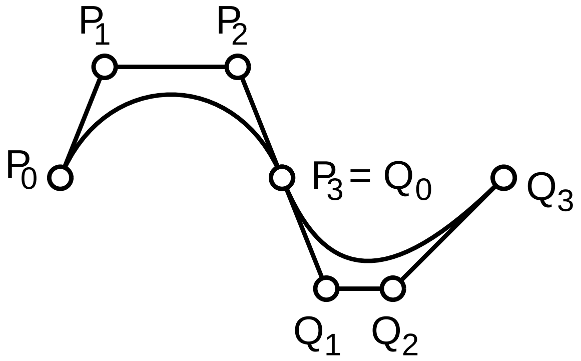




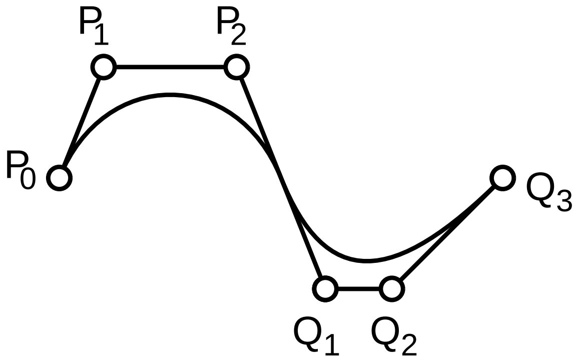
- Too hard to make user place points in this manner, so...
- Modeling: User places 4 control points, program places next 3, user places 1 more, program places next 3, ...
- Could hide program placed control points
- Redundant representation

C^1 Bézier Spline

- Suppose we have a C^1 Bézier spline



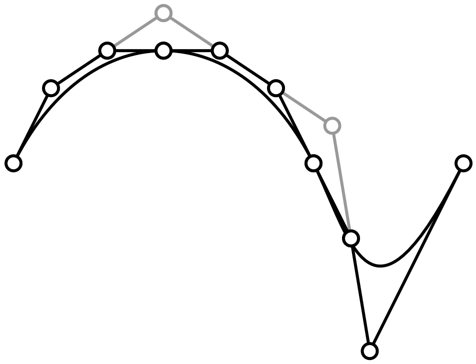
Control point Q_0 is not needed (same as P_3)
Control point P_3 can be computed from P_2 and Q_1

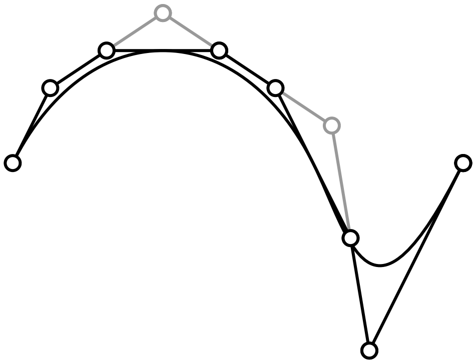


- Thus, for C^1, cubic Bézier spline, need only two control points per segment, plus one more at start and one more at end.

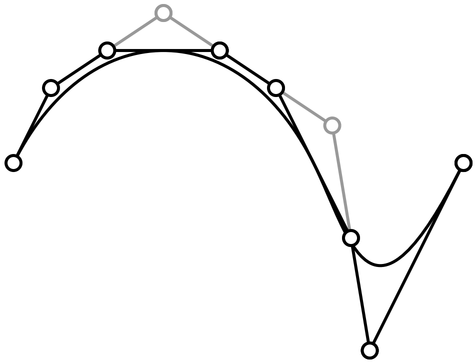
C^2 Bézier Spline

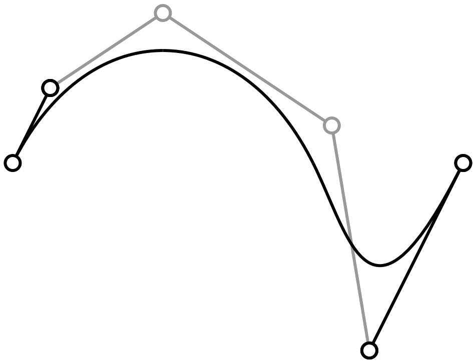
- Suppose we have a C^2 Bézier spline
  Joints can be computed as described for C^1



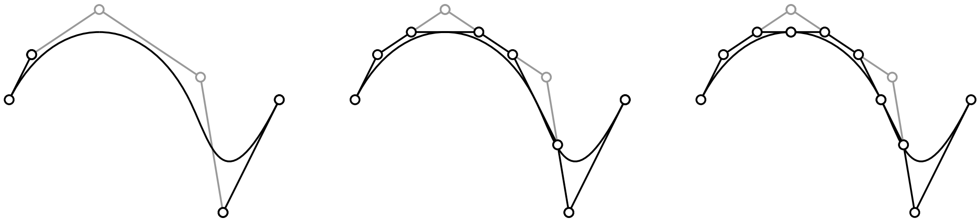


If we keep the gray points, we can compute the black points





- Thus, for C^2, cubic Bézier spline, need only one control point per segment, plus two more at start and end.
- From gray points, we can recover Bézier control points



- The gray points are **B-spline** control points.

Evaluating B-splines

- Cubic B-splines give us C^2 continuity for little effort.
- To evaluate a B-spline, we could convert to Bézier and evaluate the Bézier curves.
- There is also a de Casteljau style evaluation
  The de Boor algorithm.
- Thus far, have considered the case for intervals $[0,1],[0,2],\ldots$
  (i.e., integers)
  One polynomial over each interval
- In general, B-splines can go over arbitrary intervals $[t_0,t_1],[t_1,t_2],\ldots$
  where $t_0 < t_1 < t_2 < \ldots$
  t_i are called *knots*
- Repeating a knot reduces continuity between segments
- B-splines have different basis functions
  No closed form formula (only recurrence relation)

23.4 Tensor Product Patches

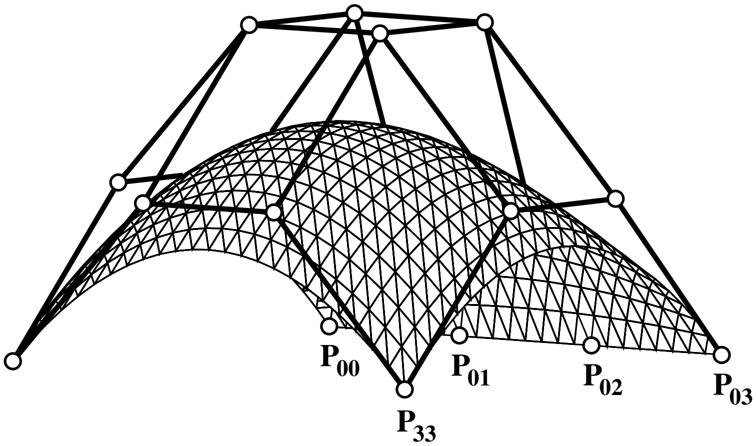
Tensor Product Patches:

- The **control polygon** is the polygonal mesh with vertices $P_{i,j}$
- The **patch basis functions** are products of curve basis functions

$$P(s,t) = \sum_{i=0}^{n} \sum_{j=0}^{n} P_{i,j} B_{i,j}^{n}(s,t)$$

where

$$B_{i,j}^{n}(s,t) = B_i^n(s) B_j^n(t)$$



Properties:

- Patch basis functions **sum to one**

$$\sum_{i=0}^{n}\sum_{j=0}^{n} B_i^n(s)B_j^n(t) = 1$$

- Patch basis functions are **nonnegative** on $[0,1] \times [0,1]$

$$B_i^n(s)B_j^n(t) \geq 0 \text{ for } 0 \leq s, t \leq 1$$

$\Longrightarrow$ Surface patch is in the **convex hull** of the control points

$\Longrightarrow$ Surface patch is **affinely invariant**
(Transform the patch by transforming the control points)

Subdivision, Recursion, Evaluation:

- As for curves in each variable separately and independently
- **Tangent plane is not produced!**
  - Normals must be computed from partial derivatives.

Partial Derivatives:

- Ordinary derivative in each variable separately:

$$\frac{\partial}{\partial s}P(s,t) = \sum_{i=0}^{n}\sum_{j=0}^{n} P_{i,j}\left[\frac{d}{ds}B_i^n(s)\right]B_j^n(t)$$

$$\frac{\partial}{\partial t}P(s,t) = \sum_{i=0}^{n}\sum_{j=0}^{n} P_{i,j}B_i^n(s)\left[\frac{d}{dt}B_j^n(t)\right]$$

- Each of the above is a **tangent vector** in a parametric direction
- Surface is **regular** at each (s,t) where these two vectors are linearly independent

- The (unnormalized) **surface normal** is given at any regular point by

$$\pm\left[\frac{\partial}{\partial s}P(s,t)\times\frac{\partial}{\partial t}P(s,t)\right]$$

  (the sign dictates what is the **outward pointing normal**)

- In particular, the **cross-boundary tangent** is given by
  (e.g. for the $s = 0$ boundary):

$$n\sum_{i=0}^{n}\sum_{j=0}^{n}(P_{1,j}-P_{0,j})\,B_j^n(t)$$

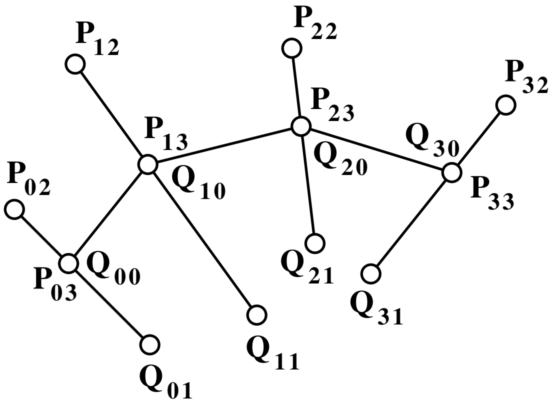
  (and similarly for the other boundaries)

Smoothly Joined Patches:

- Can be achieved by ensuring that

$$(P_{i,n}-P_{i,n-1})=\beta(Q_{i,1}-Q_{i,0})\ \text{ for }\ \beta>0$$

  (and correspondingly for the other boundaries)



Rendering:

- Divide up into polygons:
  A. By stepping

$$\begin{aligned} s &= 0,\delta,2\delta,\ldots,1 \\ t &= 0,\gamma,2\gamma,\ldots,1 \end{aligned}$$

  and joining up sides and diagonals to produce a triangular mesh

  B. By subdividing and rendering the control polygon

Tensor Product B-splines

- Could use B-splines instead of Bézier
  Automatic continuity between patches

Problem with tensor product patches

- Work well for rectilinear data
- Problems arise when filling non-rectangular holes (suitcase corner)

23.5 Barycentric Coordinates

Coordinate Frames:

- Vector oriented; Derived from linear space basis
- One point and n vectors in space of dimension n: $D_n, \vec{v}_0, \ldots, \vec{v}_{n-1}$
  - Vectors $\vec{v}_i$ are *linearly independent*

Barycentric Frames

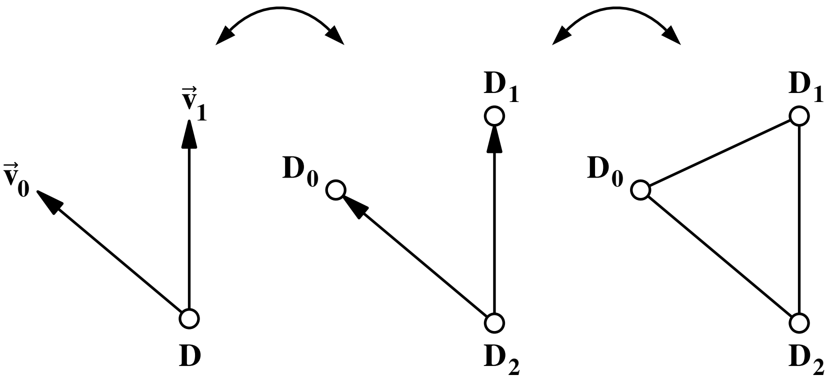
- Point oriented
- *n+1* points in space of dimension n: $D_0, \ldots, D_n$
  - Points are *in general position*

Frames of Both Types Are Equivalent

- Express each $\vec{v}_i$ as $D_i - D_n$ for $D_i = D_n + v_i$

$$\begin{aligned} P &= D_n + \sum_{i=0}^{n-1} p_i \vec{v}_i \\ &= D_n + \sum_{i=0}^{n-1} p_i (D_i - D_n) \\ &= (1 - \sum_{i=0}^{n-1} p_i) D_n + \sum_{i=0}^{n-1} p_i D_i \\ &= \sum_{i=0}^{n} w_i D_i \text{ where } \sum_{i=0}^{n} w_i = 1 \end{aligned}$$

- And, of course, conversely



23.6 Triangular Patches

deCasteljau Revisited Barycentrically:

- Linear blend expressed in barycentric terms

$$(1-t)P_0 + tP_1 = rP_0 + tP_1 \quad \text{where} \quad r+t=1$$

- Higher powers and a symmetric form of the Bernstein polynomials:

$$\begin{aligned} P(t) &= \sum_{i=0}^{n} P_i \left(\frac{n!}{i!(n-i)!}\right)(1-t)^{n-i}t^i \\ &= \sum_{\substack{i+j=n \\ i\geq 0, j\geq 0}} P_i \left(\frac{n!}{i!j!}\right) t^i r^j \quad \text{where} \quad r+t=1 \\ &\Longrightarrow \sum_{\substack{i+j=n \\ i\geq 0, j\geq 0}} P_{ij} B_{ij}^n(r,t) \end{aligned}$$

- Examples

$$\begin{aligned} \{B_{00}^0(r,t)\} &= \{1\} \\ \{B_{01}^1(r,t), B_{10}^1(r,t)\} &= \{r, t\} \\ \{B_{02}^2(r,t), B_{11}^2(r,t), B_{20}^2(r,t)\} &= \{r^2, 2rt, t^2\} \\ \{B_{03}^3(r,t), B_{12}^3(r,t), B_{21}^3(r,t), B_{30}^3(r,t)\} &= \{r^3, 3r^2t, 3rt^2, t^3\} \end{aligned}$$

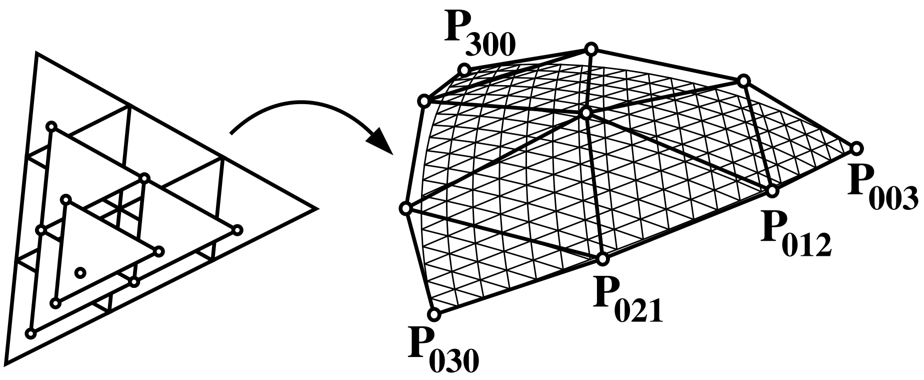
Surfaces – Barycentric Blends on Triangles:

- Formulas:

$$\begin{aligned} P(r,s,t) &= \sum_{\substack{i+j+k=n \\ i\ge 0, j\ge 0, k\ge 0}} P_{ijk} B^n_{ijk}(r,s,t) \\ B^n_{ijk}(r,s,t) &= \frac{n!}{i!j!k!} r^i s^j t^k \end{aligned}$$

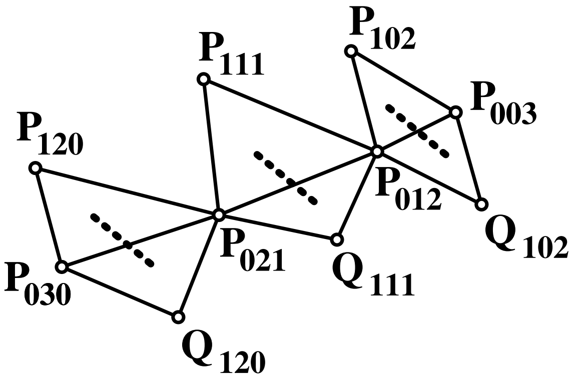
Triangular deCasteljau:

- Join adjacently indexed P_{ijk} by triangles
- Find $r:s:t$ barycentric point in each triangle
- Join adjacent points by triangles
- Repeat
  - Final point is the surface point $P(r,s,t)$
  - Final triangle is tangent to the surface at $P(r,s,t)$
- Triangle up/down schemes become tretrahedral up/down schemes



Properties:

- Each boundary curve is a Bézier curve
- Patches will be joined smoothly if pairs of boundary triangles are affine images of domain triangles



Problems:

- Continuity between two patches is easy, but joining n patches at a corner is hard
- In general, joining things together with either tensor product or triangular patches is hard

23.7 Subdivision Surfaces

Introduction

- Hard to piece spline surfaces together
- Hard to fit splines to arbitrary mesh
  n-sides faces, n-vertex neighbors
- Subdivision is an alternative surfacing scheme
  Fits surface to arbitrary mesh
  Only a few "gotcha's"
- Will look at two simple subdivision schemes
  Both schemes are simple to implement
  Neither scheme is one of "choice"
- (This material is from Chapter 7 of Warren, Weimer, Subdivision Methods for Geometric Design)

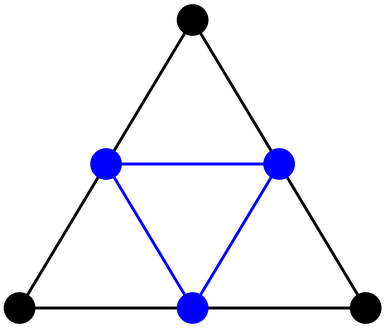
Polyhedron

- Start with a description of a polyhedron
  List of vertices, list of faces
- Example: A cube
  Vertices:
  {{0,0,0}, {1,0,0}, {1,1,0}, {0,1,0}, {0,0,1}, {1,0,1}, {1,1,1}, {0,1,1}}
  Faces:
  {{1,4,3,2}, {1,2,6,5}, {2,3,7,6}, {3,4,8,7}, {4,1,5,8}, {5,6,7,8}}

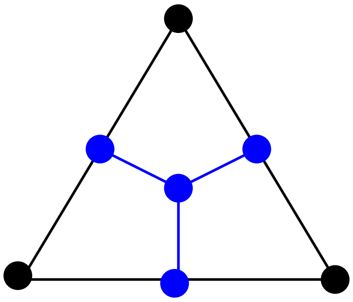
- Valence(v): number of faces containing v
- Triangle mesh: all faces are triangles
- Quad mesh: all faces are quads

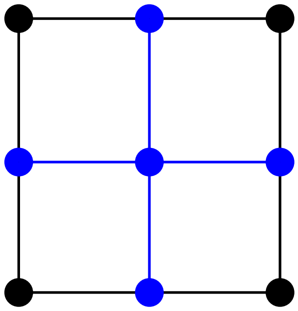
Topological Subdivision

- First step in subdivision is to split each face into new set of faces
  Will consider two types of splits (others exist)
- Triangular (linear) splits



- Quad (bilinear) subdivision



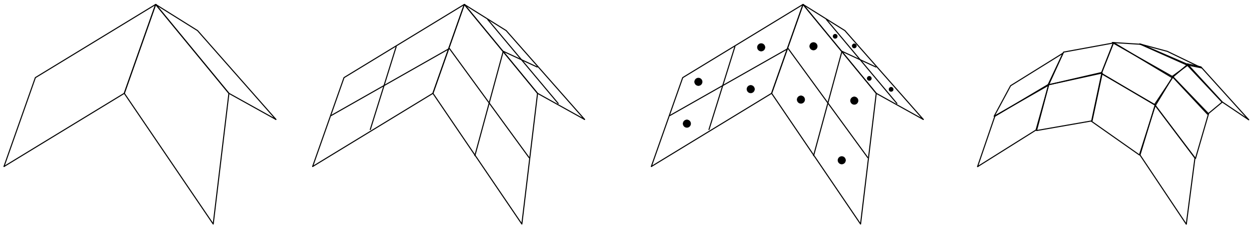


Bilinear Subdivision Plus Averaging

1. Peform bilinear subdivision of mesh
2. Compute centroid c_f of each quad face f
3. For each vertex v of subdivided mesh, set v to

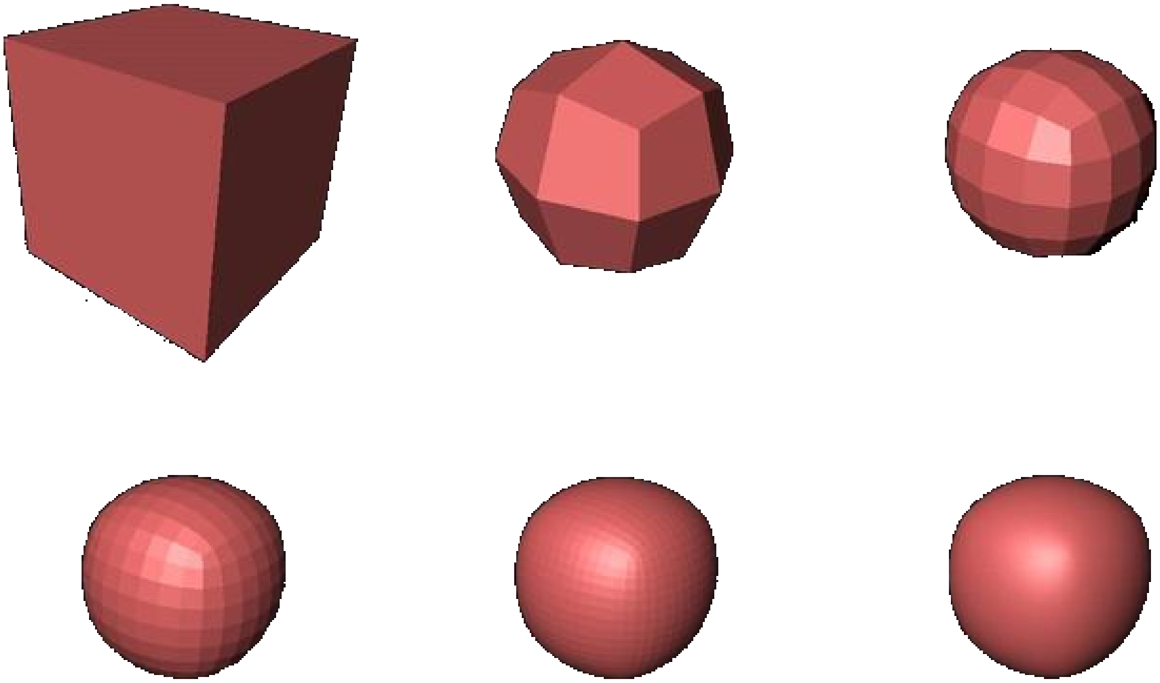
$$\frac{\sum_{f\in n(v)} c_f}{\#n(v)}$$

where $n(v)$ is the set of faces neighboring v and
$\#n(v)$ is the number of faces neighboring v



(different rules used for mesh boundaries)

Catmull-Clark Example



- Start with a cube
- Repeatedly subdivide
- Do "something" to get normals

Liner Subdivision Plus Triangle Averaging

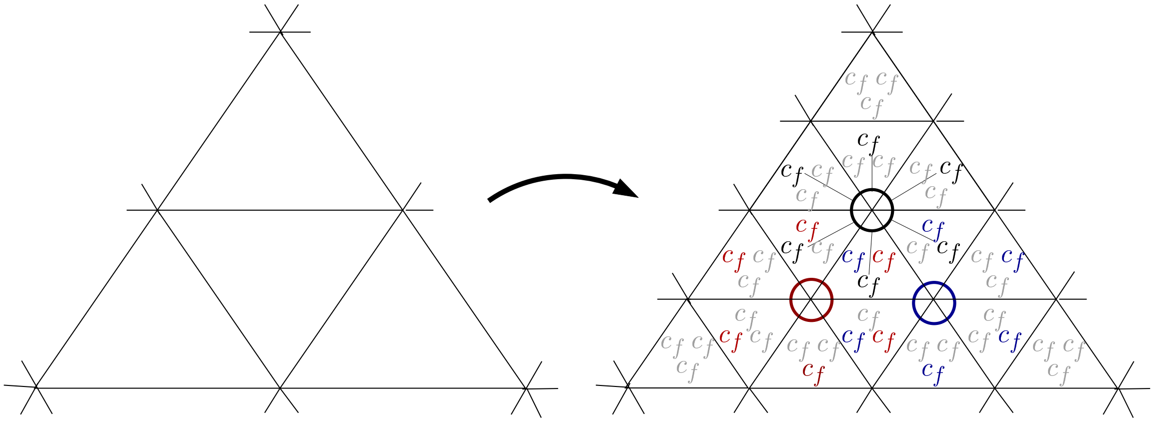
1. Perform linear subdivision of each triangle in mesh
2. For each vertex v of subdivided mesh,
   (a) For each triangle f neighboring v compute

   $$c_f = v/4 + 3v^+/8 + v^-/8$$

   where v^+ and v^- are the other two vertices of f

   (b) Set v to

   $$\frac{\sum_{f \in n(v)} c_f}{\# n(v)}$$


c_f c_f
c_f
c_f
c_f c_f c_f c_f c_f c_f
c_f c_f
c_f c_f
c_f c_f c_f c_f c_f c_f c_f c_f c_f c_f
c_f c_f
c_f c_f c_f
c_f c_f c_f c_f c_f c_f c_f c_f c_f c_f c_f c_f c_f c_f c_f c_f
c_f c_f c_f c_f c_f

Details, Issues

- The two schemes shown here are simple, but give poor surfaces
  Catmull-Clark and Loop are examples of better schemes
- Boundaries are handled using different rules
- Special rules to add creases to surface
- Major implementation issue is data structure
- Extraordinary vertices pose difficulties
  Valence other than 4 in quad mesh
  Valence other than 6 in tri mesh
- Mathematics behind the schemes is far more elaborate than the schemes themselves
- Parameterizations needed for texture mapping
- Normals?

23.8 Wavelets

Wavelets are a discrete form of Fourier series

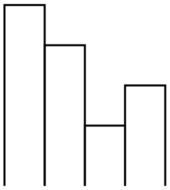
- Take a discrete signal (image).
- Decompose it into a sequence of frequencies.
- Use compact support basis functions rather than infinite support sin and cos.
- Construct a vector space using “unit pixels” as basis elements (piecewise constant functions).
- Average to get low frequencies.
- Construct “detail functions” to recover detail.
- Unit pixel bases form nested sequence of vector spaces, with the detail functions (wavelets) being the difference between these spaces.

1-D Haar

- Suppose we have the coefficients

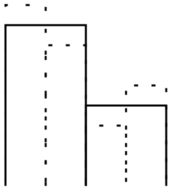
  [9 7 3 5]

  where we think of these coefficients as 1-D pixel values



  The simplest wavelet transform averages adjacent pixels, leaving

  [8 4]



- Clearly, we have lost information.

  Note that 9 and 7 are 8 plus or minus 1, and 3 and 5 are 4 minus or plus one.

  These are our detail coefficients:

  $$[1 \ \ -1]$$

- We can repeat this process, giving us the sequence

Resolution	Averages	Details
4	[9 7 3 5]	
2	[8 4]	[1 -1]
1	[6]	[2]

- The Wavelet Transform (wavelet decomposition) maps from [9 7 3 5] to [6 2 1 –1] (i.e., the final scale and all the details).

  The process is called a Filter Bank.

  No data is gained or loss; we just have a different representation.

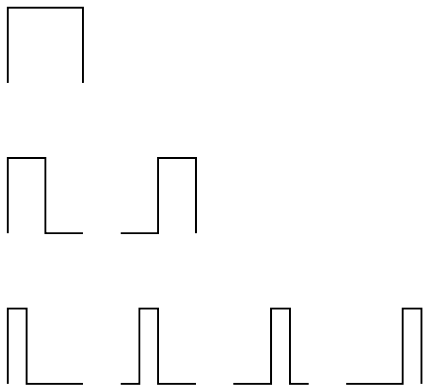
  In general, we expect many detail coefficients to be small. Truncating these to 0 gives us a lossy compression technique.

- Basis function for wavelets are called scaling functions.

  For Haar, can use

  $$\phi_i^j(x) = \phi(2^j x - i)$$

  where $\phi(x) = 1$ for $0 \leq x < 1$ and 0 otherwise.



Support of a function refers to the region over which the function is non-zero.

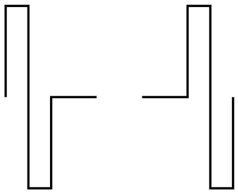
Note that the above basis functions have *compact support*, meaning they are non-zero over a finite region.

- The Haar Wavelets are

$$\psi_i^j(x) = \psi(2^j - i)$$

where

$$\psi(x) = \begin{cases} 1 & \text{for } 0 \leq x < 1/2 \\ -1 & \text{for } 1/2 \leq x < 1 \\ 0 & \text{otherwise} \end{cases}$$



- Example again.

Originally, w.r.t. V^2,

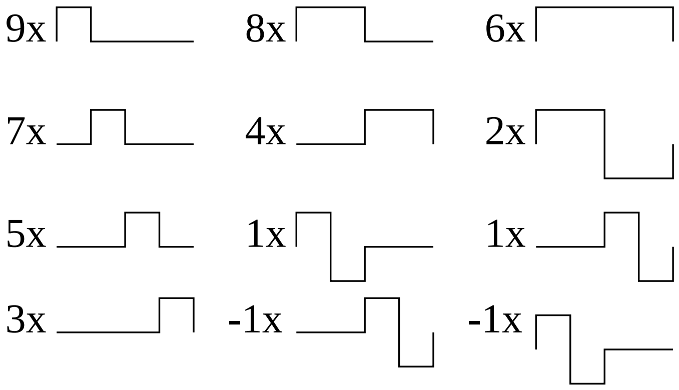
$$I(x) = 9\phi_0^2(x) + 7\phi_1^2(x) + 3\phi_2^2(x) + 5\phi_3^2(x)$$

Rewriting w.r.t. V^1 and W^1,

$$I(x) = 8\phi_0^1(x) + 4\phi_1^1(x) + 1\psi_0^1(x) + (-1)\phi_1^1(x)$$

Rewriting the ϕs in terms of V^0, W^0,

$$I(x) = 6\phi_0^0(x) + 2\psi_0^0(x) + 1\psi_0^1(x) + (-1)\phi_1^1(x)$$



Wavelet Compression

- Lossless image compression – how do we represent an image using as few bits as possible?

  Pigeon hole principle tells us we've lost before we start. Have to have expectations about image before you can achieve any compression.

  Lossy compression involves two main steps: Losing part of the data, and then performing lossless compression on what's left.

- Can use wavelets to perform simple lossy part of compression.

  If we order our coefficients in decreasing magnitude, then error is minimized.

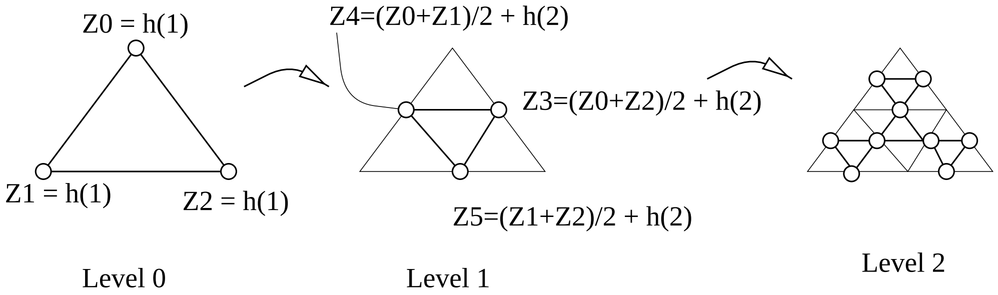
2D Haar and Image Compression

- Standard decomposition: Apply 1-D Haar to each row. Then apply 1-D Haar to each column.
- Nonstandard decomposition: Apply one step of 1-D Haar to each row. Then apply one step of 1-D Haar to each column. Repeat on quadrant containing averages in both directions.
- Image compression in similar fashion, except it's expensive to sort coefficients.

24 Modeling Natural Phenomenon

24.1 Fractal Mountains

1. How do we model mountains?
    - Lots of semi-random detail
    - Difficult to model by hand
    - Idea: Let computer generate random effect in controlled manner
2. Start with triangle, refine and adjust vertices
    - At each step of refinement, adjust height of vertices
    - Use scaled random numbers to adjust height



3. Details
   - What to use for h?
     Random number, Gaussian distribution, scaled based on level
     I.e., for deeper level, use smaller scale

   - May want to do first step manually
     Triangular grid, place peaks and valleys

   - Could use different h functions to simulate different effects

   - Automatic colour generation
     high == white
     medium == brown
     low == green
4. Results:
   - Not physically based, but...
   - Look good for little work
   - Don't look at from directly overhead

♠ *Readings: Red book, 9.5; White book, 20.3;*
References; Fornier, Fussell, Carpenter, Computer Rendering of Stochastic Models, GIP, 1982

24.2 L-system Plants

1. L-systems: Context free plants

2. Idea: Model growth/structure of plant as CFG

3. CFG: Symbols, rules
   - A, B, C, D
   - $A \rightarrow CBA$, $B \rightarrow BD$

4. Start symbol A

$$A \rightarrow CBA \rightarrow CBDCBA \rightarrow CBBDCBDCBA$$

1. Plants need more complex L-systems/geometry
    - Thickness
    - Semi-random, 3D branching
    - Leaf model
2. Top three things to make it look good:

   Texture mapping, Texture mapping, Texture mapping
3. Plants don't grow like this
    - Grow from top up, outside out
    - Respond to environment (context sensitive)
    - Interact with each other

25 Volume Rendering

25.1 Volume Rendering

Volume Data

- scalar field
    - like temperature in room at each (x, y, z)
- many applications
    - medical (MRI, CT, PET)
    - geophysical
    - fluid dynamics

Volume Data
Visible Human Project
Data Format

- 3D array of scalars (a volume)
    - 256^3 or 5123 common
    - 32-128 Mb, pretty big
    - but each slice a low-res image
- each volume element is a *voxel*
    - often 16 bits per voxel
    - several bits for segmentation (bone,muscle,fluid,...)
    - most bits for intensity

Volume Slicing

- combine slices of volume
  - use 2D texture-mapping for each slice
  - use alpha buffer to blend slices
- can use 3D textures if enough memory
- quick method
- rough results

Volume Slicing
Volume Slicing
Rendering Pipeline

- segmentation
- gradient computation
- resampling
- classification
- shading
- compositing

Segmentation

- marking voxels according to type
- medical volumes: skin, bone, muscle, air, fluids, ...
- manual procedure, need good software
- problems:
  - noise
  - overlapping intensity ranges

Segmentation
Gradient Computation

- gradient shows density changes
- useful for illumination model
- gradient magnitude like edge detection
  - can build isosurfaces
  - can colour according to magnitude

Gradient Computation
Resampling

- forward method
  - splat voxels onto screen
  - must interpolate to get pixel values
- reverse method
  - is ray-casting through volume
  - resample along ray
  - collect intensity/gradient info for pixel

Resampling
Classification

- must map intensity data to RGB
- 16-bit (monochrome) data ¿¿ 8-bit RGB
- look-up table (LUT) for range of interest
- user sets LUTs for
  - R, G, B
  - gradient magnitude
  - opacity/transparency

Classification
Classification
Shading

- use illumination model (like ray-tracing)
- gradient like surface normal
- know viewing direction from ray
- secondary rays
  - to lights
  - in reflection direction
- add shading contribution to pixel

Shading
Compositing

- combining data from samples along ray
- maximum intensity projection (MIP)
  - take biggest sample
- X-ray

  - sum or average all samples
- evaluation of ray-casting integral
  - opacity/transparency methods

Compositing
Gradients

- for $y = f(x)$
  - grad $f = df/dx$, along real line
  - sign gives direction of greatest increase in f
- for $z = f(x, y)$
  - $gradf = (df/dx, df/dy)$
- paraboloid: $z = x^2 + y^2$
  - $gradf = (2x, 2y)$, in xy-plane
  - radial direction gives greatest increase in z

Gradient Estimation

- need discrete approximation
- for $y = f(x)$
$$gradf(x_i) \doteq (f(x_i + h) - f(x_i))/h = f(x_i + 1) - f(x_i)$$
- Intermediate difference grad estimator
$$gradf(x_i) \doteq f(x_i + 1) - f(x_i)$$
- Central difference grad estimator
$$gradf(x_i) \doteq f(x_i + 1) - f(x_i - 1)$$

Gradient Estimation

- for $z = f(x, y)$
  - intermediate difference
$$gradf(x_i, y_k) \doteq (f(x_{i+1}, y_k) - f(x_i, y_k), f(x_i, y_{k+1}) - f(x_i, y_k))$$
  - central difference
$$gradf(x_i, y_k) \doteq (f(x_{i+1}, y_k) - f(x_{i-1}, y_k), f(x_i, y_{k+1}) - f(x_i, y_{k-1}))$$
- for $w = f(x, y, z)$
  - 6 neighbours: intermediate/central difference

    - 26 neighbours: Sobel operator

Gradient Estimation

- central difference smooths image

Resampling

- intersect volume
- then sample or traverse voxels

Resampling

- sampling rate affects
    - image quality
    - speed

Interpolation

- for each sample along ray
- interpolate intensity and gradient

Interpolation: Nearest Neighbour

- fast method
- but gives aliasing

Interpolation: Trilinear

- good basic method
- higher-degree splines are slower and better

Interpolation: Volume Example

- trilinear interpolation smoothes image

Interpolating Gradient?

- Gouraud
    - lighting calc at each vertex to give RGB
    - interpolate RGB to get sample on ray
- Phong
    - interpolate gradients to get gradient on ray
    - lighting calc on ray gives RGB

Interpolating Gradient / Shading

- Gouraud vs Phong

Classification

- map quantities through user-defined LUTs
- R = RLUT[I]
- G = GLUT[I]
- B = BLUT[I]
- opacity: α = ALUT[I] * NLUT[gradient mag]
- user defines LUTs through graphs

Classification

- layers of earth shown better with gradient classification

Compositing

- MIP: largest sample on ray
- X-ray: sum or average samples
- evaluation of ray-casting integral:

$$l(a,b) = \int_a^b g(s) e^{-\int_a^s \tau(x)dx} ds$$

Approximating the Integral

$$l(a,b) = \int_a^b g(s) e^{-\int_a^s \tau(x)dx} ds$$

- integral → sum
- exp(sum) → product
- voxel: transparency = 1 - opacity

$$T_i = 1 - \alpha_i$$

- approximation:

$$l(a,b) = \sum_{i=0}^{n} I_i \pi_{j=0}^{i-1} (1 - \alpha_i)$$

Compositing

- can accumulate samples along ray
  - front-to-back
  - back-to-front
- back-to-front can be very efficient
- rays can also be split for parallel processing

26 Animation

26.1 Overview

Animation: Rapid display of slightly different images create the illusion of motion.

Convential approach: Keyframed animation

Keyframes, inbetween frames

Computer Assisted:

- Human key frames, computer inbetweening
- Need to consider arc-length parameterization, interpolation of orientation, camera animation.
- Supporting techniques: Squish-box deformations and skeletons, motion capture, kinematics and inverse kinematics.
- Other animation approaches: Physically-based dynamics, constraint-based animation, procedural animation, behavioral animation.

♠ *Readings: Watt: Chapter 17.*

26.2 Traditional 2D Cel Animation

♠ *An overview of 2D cel animation, introducing the concept of keyframing. Some discussion of the "art" of animation, á la Disney.*

26.2.1 2D Cel Animation

- Traditional animation is labor intensive.
  - Start with story board.
  - Master artist draw keyframes to define action.
  - Sweat-shop artists draw inbetween frames to complete animation.
- A few traditional animiation rules:
  - Stretch and squash (anti-aliasing), timing, secondary actions
  - Exaggeration, appeal, follow through
  - Anticipation, staging.
  - Be careful with camera
    - Smooth changes
    - Few changes
- Disney was the most successful, but it was high risk.

26.3 Issues in Automated Keyframed Animation

♠ *A general overview of keyframed animation to ferret out the issues: we end with a discussion of transformation matrix animation and show why and how it fails. This motivates the rest of the module.*

26.3.1 Automated Keyframing

- Replace the human inbetweener with interpolation algorithms.
- Keyframes correspond to settings of parameters at different points in time.

+ The computer provides repeatability and automatic management of keyframes.

– Interpolation is not as smart as a human inbetweener. The animator may have to provide more key frames and/or additional information to get a good result.

26.3.2 Utility of Key Parameters

A good keyframe animation computer system limits the number of key parameters and maximizes the utility of each.

- Parameters should have immediate geometric or visible significance. Entries in a transformation matrix are *not* good parameters.
- Properties of objects that are static should be maintained automatically.
- Relationships between parts of articulated objects should be maintained automatically.
- Interpolation routines should be optimized to particular parameter types. Motion paths, orientation, camera attitude, and surface properties all require different types of control.
- A real-time interface should be provided to interactively set parameters and iteratively improve the animation.

26.3.3 Functional Animation

- An independent scalar function is specified for each "keyframed" parameter.
- Functional animation is a basic capability that supports all others.
- Most useful for truly scalar quantities: brightness of a light source, for example.
- Splines are useful for representing functions.
- Continuity control is a must: both its automatic maintenance and selective breaking.
- The functions can be edited both *explicitly*, as graphs, or *implicitly*, through a direct manipulation interface.

26.3.4 Linear Interpolation

- The most basic interpolation technique for supporting animation is linear interpolation, or the *lerp*.
- Given two parameters p_0 and p_1 at times t_0 and t_1, an intermediate value is given by

$$p(t) = \frac{t_1 - t}{t_1 - t_0} p_0 + \frac{t - t_0}{t_1 - t_0} p_1.$$

- Advantages and Disadvantages:
  - – Discontinuities in derivative exist at all key frame points.
  - + The rate of change within a segment is constant (and so can easily be controlled).

26.3.5 Velocity Control

Given a constant rate of change, we can create a specific *variable* rate of change. Given any function $\tau = f(t)$, reparameterize with τ.

For the lerp,

$$p(\tau) = p(f(t)) = \frac{f(t_1) - f(t)}{f(t_1) - f(t_0)} p_0 + \frac{f(t) - f(t_0)}{f(t_1) - f(t_0)} p_1.$$

26.3.6 Spline Interpolation

Instead of linear interpolation, spline interpolation can be used.

+ Continuity control can be obtained.

+ Fewer keyframes may be required for a given level of “quality”.

– Control points for the splines may have to be computed based on interpolation constraints at the control points.

– Extra information may be required at keyframes, such as tangent vectors.

– Splines are more expensive to evaluate.

– Splines are more difficult to implement.

26.3.7 Spline Types and Animation

Different spline types will have different tradeoffs and capabilities:

B-splines: Give automatic continuity control, but only approximate their control points.

Non-uniform B-splines: Are needed to break continuity.

Non-uniform Rational B-splines (NURBS): Provide additional local control over the shape of a curve.

Bézier splines: Require additional information for each segment and do not automatically maintain continuity. However, they do interpolate the ends of segments.

Hermite splines: Require tangent vectors at each keyframe.

Catmull-Rom splines: Provide only first derivative continuity.

Beta-splines: Provide automatic continuity, control over the tangent and approximate arc length of each segment.

26.3.8 Transformation Matrix Animation

- One way to support an animation capability: interpolate a transformation matrix.
- Keyframes would be "poses" of objects given by the animator.
- Functional animation would be applied independently to all entries of the transformation matrix.

Unfortunately, does not support animation of rigid bodies.

- Given two poses of an object that differ by a 180° rotation.
- Under linear interpolation, object will not rotate but will turn inside out
  collapses to a plane in the middle of the interpolation.

26.3.9 Rigid Body Animation

- Rigid body transformations only have 6 degrees of freedom
  General affine transformations have 12.
- To obtain good interpolation we have to consider separately
  - three degrees of freedom from translation
  - three that result from orientation.

26.4 Motion Path Animation

♠ *Motion path animation in the context of spline interpolation of position. Deals mostly with problems of arc-length-reparameterization and velocity control.*

26.4.1 Motion Path Overview

As a basic capability, we would like to translate a point through space along a given path. We would like:

- Independent control of velocity along path.
- Continuity control.

While splines can easily support continuity control, *velocity* control is more difficult.

26.4.2 Spline Interpolation Problems

Spline position interpolation will give us continuity control over changes position, but:

- An equal increment in the spline parameter does not usually correspond to an equal increment in distance along the spline.
- Different segments of the spline with the same parametric length can have different physical lengths.
- If we parameterize the spline directly with time objects will move at a non-uniform speed.

26.4.3 Arc Length Parameterization

1. Given a spline path $P(u) = [x(u), y(u), z(u)]$, compute the arclength of the spline as a function of u: $s = A(u)$.
2. Find the inverse of $A(u)$: $u = A^{-1}(s)$.
3. Substitute $u = A^{-1}(s)$ into $P(u)$ to find a motion path parameterized by arclength, s: $P(s) = P(A^{-1}(s))$.

Note that u (and thus s) should be global parameters, extending across all segments of the original spline.

26.4.4 Velocity Control

To control velocity along a spline motion path,

- Let $s = f(t)$ specify distance along the spline as a function of time t.
- The function $f(t)$, being just a scalar value, can be supported with a functional animation technique (i.e. another spline).
- The function $f(t)$ may be specifed as the integral of yet another function, $v(t) = df(t)/dt$, the *velocity*. Note that the integral of a spline function can be found analytically, through a manipulation of the control points.
- The motion path as a function of time is thus given by $P(t) = P(f(t)) = P(A^{-1}(f(t)))$.

26.4.5 Some Nasty Problems

There are some nasty problems with the arc-length parameterization approach:

1. The arc-length $s = A(u)$ is given by the integral

   $$s = A(u) = \int_0^u \sqrt{\left(\frac{dx(v)}{dv}\right)^2 + \left(\frac{dy(v)}{dv}\right)^2 + \left(\frac{dz(v)}{dv}\right)^2}\, dv$$

   which has no analytic solution if the motion path is a cubic spline.
2. Since $A(u)$ has no analytic form, $A^{-1}(s)$ has no analytic form.

We have to use numerical techniques:

1. Find $A(u)$ with numerical quadrature. Precompute some values of s, at least at the boundaries of segments, to make this easier.
2. Since s is a monotonic function of u, for a given s^* we can find the solution u^* to $s^* = A(u^*)$ by bisection (binary search).

26.4.6 Real Time Issues

- Exact arc-length parameterization may not be feasible.
- An alternative: compute points on the spline at equally-spaced parametric values, and use linear interpolation along these chords.
- The linear interpolation should consider the distance between samples to maintain constant velocity.

26.5 Orientation and Interpolation

♠ *The problems of orientation representation and interpolation are discussed. Both Euler angles and quaternions are introduced and compared. Spherical linear interpolation is introduced.*

26.5.1 Orientation Interpolation

Interpolation of orientation should proceed indirectly by interpolation of angles rather than transformation matrices.

- In two dimensions, only one angle is involved and animation is straightforward.
- In three dimensions, *three* angles are involved and the topology is that of a sphere in four dimensions. Interpolation is much nastier and harder to visualize.

26.5.2 Approaches

Two standard approaches to the orientation interpolation problem. Both related to problem of specifying orientation in the first place:

Euler angles. Three angles: x-roll followed by y-roll followed by z-roll.

- − Has defects: parameterization singularities, anisotropy, “gimbal lock”, unpredictable interpolation.
- − Hard to solve inverse problem: given orientation, what are the angles?
- + Widely used in practice.
- + Easy to implement.
- + Inexpensive computationally.

Quaternions. Four-dimensional analogs of complex numbers.

- + Isotropic: avoid the problems of Euler angles.
- + Inverse problem easy to solve.
- + The user interface tends to be simpler.
- – More involved mathmatically.
- – Interpolation can be expensive in practice.

26.5.3 Euler Angles

Euler angles parameterize orientation by defining an affine transformation which is a concatenation of three primitive rotations. With $c_a = \cos(\theta_a)$ and $s_a = \sin(\theta_a)$,

$$\begin{aligned} R(\theta_x, \theta_y, \theta_z) &= \begin{bmatrix} c_y c_z & c_y s_z & -s_y & 0 \\ s_x s_y c_z - c_x s_z & s_x s_y s_z + c_x c_z & s_x c_y & 0 \\ c_x s_y c_z + s_x s_z & c_x s_y s_z - s_x c_z & c_x c_y & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &= R_x(\theta_x) R_y(\theta_y) R_z(\theta_z), \end{aligned}$$

where $R_x(\theta_x)$, $R_y(\theta_y)$ and $R_z(\theta_z)$ are the standard rotation matrices.

Given a point P represented as a homogeneous row vector, the rotation of P is given by $P' = PR(\theta_x, \theta_y, \theta_z)$. Animation between two rotations involves simply interpolating independently the three angles θ_x, θ_y, and θ_z.

♠ *The standard rotation matrices are given by*

$$\begin{aligned} R_x(\theta_x) &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & c_x & s_x & 0 \\ 0 & -s_x & c_x & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \\ R_y(\theta_y) &= \begin{bmatrix} c_y & 0 & -s_y & 0 \\ 0 & 1 & 0 & 0 \\ s_y & 0 & c_y & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \\ R_z(\theta_z) &= \begin{bmatrix} c_z & s_z & 0 & 0 \\ -s_z & c_z & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \end{aligned}$$

26.5.4 Euler Angle Problems

Problems[1] with the Euler angle approach include:

Parametric Singularity: A degree of freedom can suddenly vanish.

Anisotropy: The order of the axes is important.

Nonobviousness: The parameters lack useful geometric significance.

Inversion: Finding the Euler angles for a given orientation is difficult.

Coordinate system dependence: The orientation of the coordinate axes is important.

---

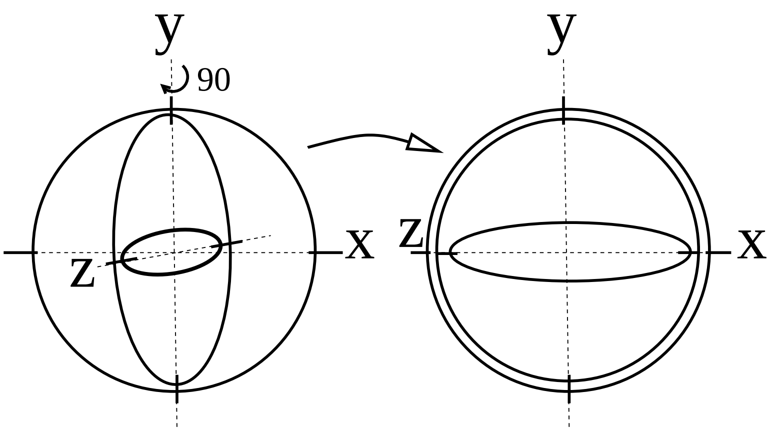
[1] note that the initial letters here spell PANIC.

26.5.5 Gimbal Lock

- *Gimbal lock* is an example of a **parametric singularity**.
- Gimbal lock is a mechanical problem that arises in gyroscopes as well as quaternions.
- Set $\theta_y = \pi/2 = 90°$, and set θ_x and θ_z arbitrarily. Then $c_y = 0$, $s_y = 1$ and the matrix $R(\theta_x, \pi/2, \theta_z)$ can be reduced to

$$
\begin{aligned}
R(\theta_x, \theta_y, \theta_z) &= \begin{bmatrix} 0 & 0 & -1 & 0 \\ s_x c_z - c_x s_z & s_x s_z + c_x c_z & 0 & 0 \\ c_x c_z + s_x s_z & c_x s_z - s_x c_z & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
&= \begin{bmatrix} 0 & 0 & -1 & 0 \\ \sin(\theta_x - \theta_z) & \cos(\theta_x - \theta_z) & 0 & 0 \\ \cos(\theta_x - \theta_z) & \sin(\theta_x - \theta_y) & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.
\end{aligned}
$$

- The transformation only depends on the difference $\theta_x - \theta_z$, and hence only has one degree of freedom when it should have two.
- The problem occurs because a y-roll by $\pi/2$ rotates the x-axis onto the negative z axis, and so a x-roll by θ has the same effect as a z-roll by $-\theta$.



- Gimbal lock can be very frustrating in practice:
  - During interactive manipulation the object will seem to "stick";
  - Certain orientations can be hard to obtain if approached from the wrong direction;
  - Interpolation through these parametric singularities will behave strangely.

26.5.6 Quaternions

- Four-dimensional analogs of complex numbers.
- Can be used to represent orientation without a directional selectivity.

Consider: multiplication of complex numbers can be used to represent orientation and rotation in the plane. The rectangular and polar form of a complex number p are

$$p = a + bi = Ae^{i\theta}.$$

Multiplication of two complex numbers is equivalent to a rotation about the origin:

$$p_1 p_2 = A_1 A_2 e^{i(\theta_1 + \theta_2)}$$

26.5.7 Definition of Quaternions

Quaternions are defined using three imaginary quantities: i, j, and k:

$$q = a + bi + cj + dk.$$

The rules for combining these imaginary quatities are as follows:

$$\begin{aligned} i^2 = j^2 = k^2 &= -1, \\ ij = -ji &= k, \\ jk = -kj &= i, \\ ki = -ik &= j. \end{aligned}$$

These rules are sufficient to define the operation of multiplication for quaternions. Note that quaternion multiplication *does not commute.*

26.5.8 Properties of Quaternions

A quaternion can be broken into a scalar and a vector part:

$$q = (s, \vec{v}) = s + v_1 i + v_2 j + v_3 k.$$

In vector notation, we can write the product of two quaternions as

$$q_1 q_2 = (s_1 s_2 - \vec{v}_1 \cdot \vec{v}_2, s_1 \vec{v}_2 + s_2 \vec{v}_1 + \vec{v}_1 \times \vec{v}_2).$$

Quaternions also have conjugates: $\bar{q} = (s, -\vec{v})$. The norm of a quaternion is defined by

$$|(s, \vec{v})| = \sqrt{s^2 + v_1^2 + v_2^2 + v_3^2}.$$

26.5.9 Unit Quaternions

Unit quaternions have unit norms, and can be shown to be isomorphic to orientations. The unit quaternion given by

$$q_r = (\cos(\theta), \sin(\theta)\vec{v})$$

for unit vector $\vec{v}$ is equivalent to a rotation by an angle of 2θ around the axis defined by $\vec{v}$. Note that q_r is equivalent to $-q_r$ when interpreted as an orientation.

26.5.10 Rotations and Quaternions

Points in space can be represented by quaternions with a zero scalar part. Point P is represented by the quaternion

$$q_P = (0, P).$$

Finally, the rotation of a point P by the incremental rotation defined by q_r can be found by evaluating the product

$$q_{P'} = q_r q_P q_r^{-1}.$$

and interpreting the result as a point. For unit quaternions, the inverse is equivalent to the conjugate.

26.5.11 Advantages of Quaternions

Doing rotation this way is fairly perverse, except for three points:

1. For many applications, the independent definition of an axis of rotation and an angle makes sense.
2. The definition of an orientation by a quaternion is coordinate system independent and isotropic.
3. Spherical interpolation of quaternions gives better results than interpolation of Euler angles.

26.5.12 Interpolating Unit Quaternions

To interpolate two orientations, we could use linear interpolation on all the entries of the quaternion (i.e. treat the quaternion as a four-dimensional vector). Unfortunately, intermediate quaternions would not have unit norm, and the angular velocity would not be constant even if we renormalized.

The solution: spherical linear interpolation, or the *slerp*. Consider the quaternions as vectors, and find the angle between them:

$$\omega = \cos^{-1}(q_1 \cdot q_2).$$

Given a parameter $u \in [0, 1]$, the slerp interpolated value is defined as

$$q(u) = q_1 \frac{\sin((1-u)\omega)}{\sin(\omega)} + q_2 \frac{\sin(u\omega)}{\sin(\omega)}.$$

The slerp will have numerical difficulties when $\omega \approx 0$. In such as case, it is wise to replace the slerp with a lerp. There will also be a problem with $\omega \approx n\pi/2$; this should probably be flagged with an error and/or a request for more keyframes.

26.5.13 Great Circles

There are two possible great circles joining any two orientations. Normally, we want to choose the shortest one. This can be accomplished by testing the condition $(q_1 - q_2) \cdot (q_1 - q_2) > (q_1 + q_2) \cdot (q_1 + q_2)$. If true, q_2 should be replaced with $-q_2$. Recall that two quaternions that differ only in sign represent the same orientation.

26.5.14 Quaternions to Rotation Matrices

A unit quaternion $q = (W, (X, Y, Z))$ is equivalent to the matrix

$$\begin{bmatrix} 1-2Y^2-2Z^2 & 2XY-2WZ & 2XZ+2WY & 0 \\ 2XY+2WZ & 1-2X^2-2Z^2 & 2YZ-2WX & 0 \\ 2XZ-2WY & 2YZ+2WX & 1-2X^2-2Y^2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

This matrix should be compared with the one given for Euler angles; note the increased symmetry of the quaternion matrix, which reflects the isotropy of the representation.

26.5.15 Rotation Matrices to Quaternions

To convert from a orthonormal rotation matrix to a unit quaternion, we observe that if $M = [m_{i,j}]$ is the affine transformation in homogeneous form,

$$\text{trace}(M) = 4 - 4(X^2 + Y^2 + Z^2) = 4W^2.$$

Once we have W, we can find the other quantities by cancelling symmetric terms:

$$\begin{aligned} X &= \frac{m_{32} - m_{23}}{4W}, \\ Y &= \frac{m_{13} - m_{31}}{4W}, \\ Z &= \frac{m_{21} - m_{12}}{4W}. \end{aligned}$$

26.6 Animating Camera Motion

♠ *Animation issues specifically dealing with camera motion. Some cinematic perspectives are developed through a look at the terminology used to describe camera motion.*

26.6.1 Overview of Camera Animation

While cameras could be animated using a combination of motion path and orientation interpolation tools, the requirements for camera motion are somewhat different:

- The camera should always be level, unless we specify otherwise.
- The image of objects of interest should be stable on the film plane.
- Specification of camera motion has a long cinematic tradition that should be respected.

26.6.2 Cinematics of Camera Motion

- An animated zoom changes perspective, which can be disturbing and/or distracting. A dolly should be used instead to enlarge the image of an object of interest.
- The camera should almost never be rotated about its view direction—unless a seasick audience is the objective.
- Changing the focal depth can be used to track objects of interest, but a sophisticated renderer is required to simulate focus.

- Depth of field is rarely changed in real cinematography due to technical limitations, but might be useful in computer animation.
- Smooth spline animation should almost always be used for camera animation. Quick camera moves should normally be replaced by cuts (instantaneous changes of scene) unless a special effect is desired.
- Animation of spotlight sources is similar to the animation of a camera. Support of a "light's-eye-view", therefore, is often useful.

26.6.3 Terminology of Camera Motion

Dolly: Move forward, along the line of sight of the camera (towards the object of interest).

Track: Move horizontally, perpendicular to the line of sight of the camera. More generally, move in a horizontal plane.

Crane: Raise the camera vertically.

Tilt (Bank): Rotate about the horizontal axis perpendicular to the line of sight of the camera.

Pan (Yaw): Rotate about the vertical axis of the camera (after tilt).

In addition, cameras have other parameters that can be animated:

Zoom (in, out): Change the angular field of view of the camera.

Focus: Change the focal depth, i.e. the distance at which objects are in focus.

Depth of Field: Changing the aperture size has the effect of changing the depth of field, i.e. the range of depths over which objects can be considered to be in focus.

26.6.4 Specification of Camera Motion

- With a camera, usually more interested in what it's looking at than its exact orientation
- Camera animation may be specified by
  - A motion path for the camera itself.
  - A motion path for a lookat point
    (possibly derived from the motion of an object of interest).
- Camera moved along the motion path and the orientation of camera determined by the lookat point.
- Ideally, "focus would be pulled" to match the distance to object of interest.
- Tilt might be constrained to lie within certain bounds.

26.7 Tools for Shape Animation

26.7.1 Skeletons

- Given a model with large number of vertices, vertices can be grouped and manipulated as a unit.

  Example: in an arm, all points describing the forearm move more-or-less as a rigid body.

- Connectivity between groups can be represented by tying each group to a "bone" and arranging bones in an articulated "skeleton".
- The bone and all its associated vertices are treated as a single rigid object for the purposes of transformation.
- Movement of the bones is constrained by the joints in the skeleton.
- Different kinds of joints (revolute, hinge, ball) can support specific types of motion.
- An animation can be previewed using only the bones, speeding rendering time.
- In the final rendering, the model is shown without bones.
- Enhancement: vertices may be influenced by more than one bone.

  Results in more flexible and realistic surface near joints.

  Example: a kneecap and its skin surface affected by both the femur and shinbone, and assume a position halfway between either.

26.7.2 Free-Form Deformations

Sometimes, skelton cannot capture desired shape change of an object.

Example: The "sqaush and stretch" of a bouncing ball as it hits the ground.

Such global changes in shape can be expressed using a *deformation.*

A deformation

- Changes the *space* around an object
- Nonlinear transformation
- New coordinates for every point in space are determined as functions of the old coordinates.

Process:

1. Rectangular "squish box" is placed around part of the model to be animated.
2. The coordinates of all points within the box are determined relative to the frame given by the box. Suppose a vertex P can be expressed as $[u, v, w]$ relative to the coordinates of the box.
3. The new coordinates of P are given by a tensor product Bézier spline $B^{n_1,n_2,n_3}(u, v, w)$ with control points $P_{i,j,k}$.

4. If the $P_{i,j,k}$ have coordinates

   $$P_{i,j,k} = [i/(n_1+1), j/(n_2+1), k/(n_3+1)],$$

   the transformation is given by the identity:

   $$[u, v, w] = B^{n_1,n_2,n_3}(u, v, w).$$

   Normally, then, the control points are initially set to these values and are moved to effect a deformation.

5. Continuity conditions can be enforced.

   Example: A hand is being deformed but the arm is not.

   The control points along the edge of the box that cuts the wrist should not be moved, nor the next layer.

   This maintains both position and derivative continuity across the wrist.

6. The object should be finely tesselated or radical deformations will not work properly.

   For realistic animation, typically only small deformations are used.

26.7.3 Using FFDs and Skeletons Together

- Skeletons and free-form deformations can be used simultaneously.
- The number of control points in a free-form deformation can be large
- If moved in groups relative to skeleton, excellent animation control can be obtained for arbitrary models.

26.8 Kinematics and Inverse Kinematics

26.8.1 Kinematics and Inverse Kinematics

Kinematics: The study of motion independent of the forces that cause the motion. Includes position, velocity, acceleration.

Forward kinematics: The determination of the

  - positions,
  - velocities,
  - accelerations

of all the links in an articulated model given the

  - position,
  - velocity,
  - acceleration

of the root of the model and all the transformations between links.

  - Forward kinematics is a necessity for skeletal keyframed animation

  - Easy to implement.

Inverse kinematics: The derivation of the motion of intermediate links in an articulated body given the motion of some key links.

26.8.2 Inverse Kinematics

- Often nonlinear, underdetermined or overdetermined, possibly ill-conditioned.
- Complexity of determining solution proportional to number of free links.
- One free joint between two fixed ones can be solved fairly efficiently, for example, with only one spare degree of freedom.
- Extra constraints may be needed to obtain a unique and stable solution.
  - Example: Requiring a joint to point downwards as a gross approximation to gravity
  - Joint motion constraints (hinge vs. revolute vs. ball) are also useful.
- Additional optimization objectives
  - Resulting optimization problem solved iteratively as an animation proceeds.
  - Example optimization objectives:
    - minimize the kinetic energy of structure;
    - minimize the total elevation of the structure;
    - minimize the maximum (or average) angular torque.

26.9 Physically-Based Animation

♠ *Use of physical simulation for animation. Comparison with keyframed animation, derivation of an example using a spring-mass system.*

26.9.1 Physically-Based Animation

Idea: to obtain a physically plausible animation, *simulate* the laws of Newtonian physics.
In contrast to kinematics, this is called a *dynamics* approach.

- Have to control objects by manipulating forces and torques, either directly or indirectly.
- Animated objects may be passive: bouncing balls, jello, wall of bricks falling down, etc.

Inverse dynamics is difficult

- Finding forces to move a physical object along a desired path
- Most current applications are "passive" variety
- Useful for seconary effects in keyframe animation:
  - a mouse character's ears should flap when it shakes its head,
  - a T. Rex's gut should sway while it runs,
  - a feather in a cap should wave when wind blows on it,

  - cloth should drape around an actor and respond to his/her/its movements,
  - a wall of bricks should fall down when a superhero crashes into it.

Caution: simulated secondary effects can make bad keyframed animation look even worse!

26.9.2 Simulation

Setting up and solving a physically-based animation proceeds as follows:

1. Create a model with physical attributes: mass, moment of inertia, elasticity, etc.
2. Derive differential equations by applying the laws of Newtonian physics.
3. Define initial conditions, i.e., initial velocities and positions.
4. Supply functions for external forces (possibly via keyframing).
5. Solve the differential equations to derive animation, i.e., motion of all objects in scene as a function of time.

- During solution of the differential equations we have to be prepared to deal with discontinuities due to collisions.
- We will discuss the simulation of spring-mass and point mass models.
- Both have useful applications and are relatively simple to simulate.

26.9.3 Spring-Mass Models

1. Create a model:
   - Composed of a mesh of point masses m_i connected by springs with spring constants k_{ij} and rest lengths ℓ_{ij}.
   - Let $P_i(t) = [x_i(t), y_i(t), z_i(t)]$ be the position of mass m_i at time t.
   - Let N_i be the set of all indices of masses connected to mass m_i.
     If $j \in N_i$, then m_i and m_j are connected by a spring.
   - Springs exert force proportional to displacement from rest length, in a direction that tends to restore the rest length:

     $$\begin{aligned} \vec{f}^s_{ij}(t) &= -k_{ij}(\ell_{ij} - |P_i(t) - P_j(t)|)\frac{P_i(t) - P_j(t)}{|P_i(t) - P_j(t)|}, \\ \vec{f}^s_i(t) &= \sum_{j \in N_i} \vec{f}^s_{ij}(t). \end{aligned}$$

   - Masses assumed embedded in a medium that provides a damping force of

     $$\vec{f}_d = -\rho_i \vec{v}_i(t),$$

     $\vec{v}_i(t)$ is velocity of m_i at time t.
     (Damping could also be provided on the derivative of the change in rest length of the springs, which would be coordinate-system independent.)

2. Motion of each mass governed by second order ordinary differential equation:

   $$m_i\vec{a}(t) = -\rho_i\vec{v}(t) + \vec{f}_i^s(t) + \vec{f}_i^e(t).$$

   $\vec{f}_i^e(t)$ is the sum of external forces on node i and

   $$\begin{aligned} \vec{a} &= \left[\frac{d^2x_i(t)}{dt^2}, \frac{d^2y_i(t)}{dt^2}, \frac{d^2z_i(t)}{dt^2}\right], \\ \vec{v} &= \left[\frac{dx_i(t)}{dt}, \frac{dy_i(t)}{dt}, \frac{dz_i(t)}{dt}\right] \end{aligned}$$

3. Initial conditions: user supplies initial positions of all masses and velocities.

4. The user supplies external forces: gravity, collision forces, keyframed forces, wind, hydrodynamic resistance, etc. as a function of time.

5. Simulation.

   - Factor second-order ODE into two coupled first-order ODE's:

     $$\begin{aligned} \vec{v} &= [v_{xi}(t), v_{yi}(t), v_{zi}(t)] \\ &= \left[\frac{dx_i(t)}{dt}, \frac{dy_i(t)}{dt}, \frac{dz_i(t)}{dt}\right], \\ \vec{a} &= \left[\frac{dv_{xi}(t)}{dt}, \frac{dv_{yi}(t)}{dt^2}, \frac{dv_{zi}(t)}{dt}\right], \\ &= \frac{1}{m_i}\left(-\rho_i\vec{v}(t) + \vec{f}_i^s(t) + \vec{f}_i^e(t)\right). \end{aligned}$$

   - Solve using your favourite ODE solver. The simplest technique is the Euler step: pick a Δt. Then compute the values of all positions at $t + \Delta t$ from the positions at t by discretizing the differential equations:

     $$\begin{aligned} \vec{a}_i(t+\Delta t) &\leftarrow \frac{\Delta t}{m_i}\left(-\rho_i\vec{v}(t) + \vec{f}_i^s(t) + \vec{f}_i^e(t)\right), \\ \vec{v}_i(t+\Delta t) &\leftarrow v_i(t) + \vec{a}(t)\Delta t, \\ P_i(t+\Delta t) &\leftarrow P_i(t) + \vec{v}(t)\Delta t. \end{aligned}$$

26.9.4 Collisions

- Simulation may be interrupted by a collision.
- A collision force should be generated to reflect the momentum of the colliding masses.
- Using fixed time step, collision may be detected via interpenetration.
- May be necessary to use a Δt shorter than the frame time and/or back up the simulation after interpenetration to handle collisions reasonably robustly.
- $O(n^2)$ collision checks between $O(n)$ objects

  Partition space, hierarchies, etc.
- Elastic vs inelastic

$$F = ma$$

26.10 Human Motion

26.10.1 Walking

Modeling walking is hard

- Human motion (walking) is extremely complex

  Fine grained balancing act
- Human walking involves more than just the feet and legs

  Hips, body, shoulder, head
- Human perception of walking is very precise

  Poor walking immediately spotted
- Task is still an area of active research

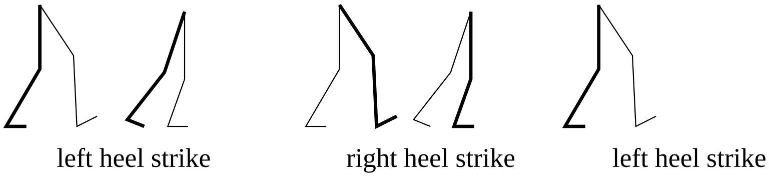
We will look at some basics

Walk Cycle

- Walking is mostly a cycle of motions

  Have to adapt each time through cycle for terrain, etc.
- In walking, at least one foot on ground
- Left stance, right swing, right stance, left swing

  Repeat



- Hip movement: rotation
- Hip rotation requires knee flexion
- Ankle, toe joints
- Can animate leg motion by specifying angles of all joints as function of time

26.11 Sensor-Actuator Networks

26.11.1 Sensor-Actuator Networks

Sensor-Actuator-Networks

- van de Panne
- Model creature as set of
  - Links

    - Sensors
    - Actuators
- SAN relates sensors and actuators
- Control uses sensor data to apply forces thru actuators

Links

- Ridgid
- Have mass
- May restrict to planar for simplicity

Sensors : 4 types

- Touch (picture)
- Angle (picture)
- Eye - used to find target
- Length (picture)

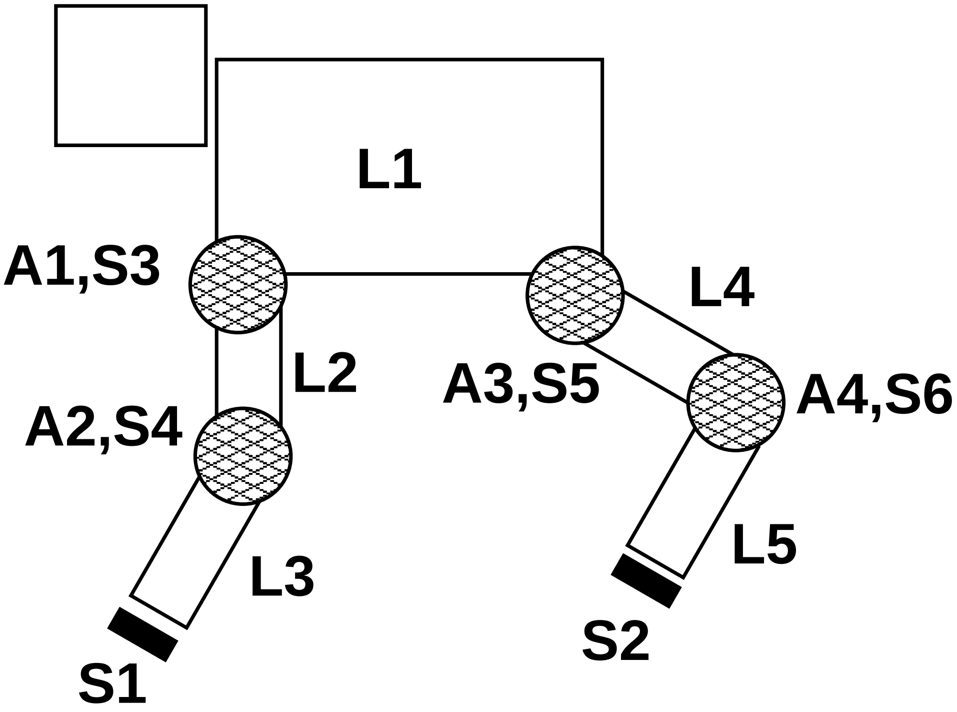
Actuators

- Length/linear
    - push/pull points
    - max/min, strength
- Angle
    - create torques
    - max/min angle, strength

Motion

- Evaluation metric
    - Distance traveled
    - Don't fall over
- Generate random controllers and evaluate with metric
- Try lots, and fine tune the best

SAN example:



26.12 Alternative

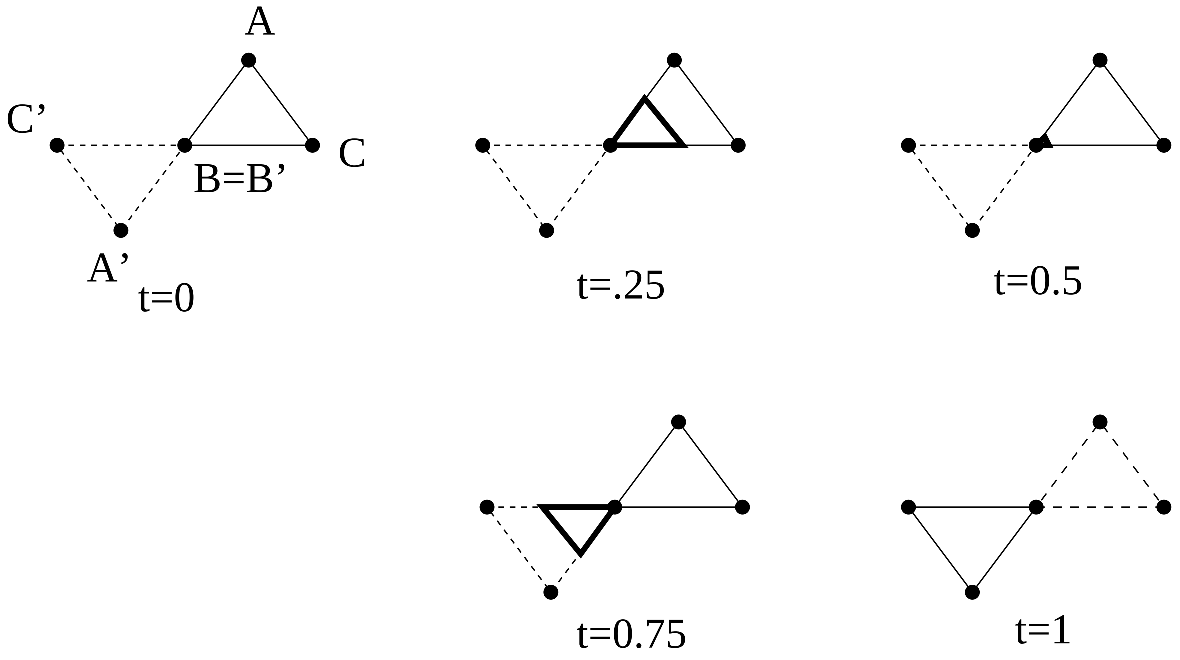
♠ *A collection of alternative animation techniques and technologies.*

- *Morphing*
- *Motion Capture*
- *Particle systems*
- *Flocking*

26.12.1 Morphing

- Morphing is lerping between images.
- Nice effects at low cost.
- Key issues:
  - Partitioning images into pieces that correspond
    E.g., eyes to eyes, etc.
  - Lerping pieces (geometry)
  - Lerping pieces (pixels, colours)
  - Filtering
- Developed at NRC in the 1970's (line drawings).
- Make it quick, don't look too closely at intermediate images.
- Simplest morph linearly interpolates each point

  Mostly okay, but can cause problems:



  Often tweak by hand to get rid of such problems

- More sophisticated morphs introduce other problems
- Desired properties:

  Vertex trajectories smooth and continuous

  Triangles don't cross each other

  Minimize triangle size variations

  Linear when other properites not violated

26.12.2 Motion Capture

- Keyframing complex actions (walking, etc) requires a lot of key frames.
- Instead, capture motion from real objects.
  Creatures and/or actors performing the desired motion.
- The positions of key points on the actor are tracked
  these motions are mapped to corresponding points in the model.
- May be necessary to generate secondary actions for the computer model.
  (tracking is expensive)
- Tracking technologies include
  1. Electromagnetic position and orientation sensors.
     - Requires a wire to each sensor.
     - Readings can be distorted by metal and magnetic fields.
     - Limited working volume.
  2. Ultrasonic rangefinder triangulation.
     - Cheaper but less accurate than magnetic sensors.
  3. Optical triangulation.
     - Reflective or emissive markers are attached to objects
     - Two cameras triangulate position
     - Can track more points cheaply, without wires
     - Points can be occluded.
  4. Body suit.
     - Fibers in instrumented clothing detect angles of all joints.
  5. Rotoscoping.
     - Arbitrary video from two cameras converted to motion paths
       - manual selection of key points
       - computer vision
     - Prone to error and occlusion.
     - Tedious if done manually.
  6. Puppetry.
     - Real-time data collected from an arbitrary input device
     - Mapped to an computer model, which is displayed in real time during capture.
     - A special-purpose input device in the shape of the object being animated may be built.

26.12.3 Problems with Motion Capture

- Motion capture tends to require more-or-less exotic technology.
- Even puppetry requires at least real-time graphics.
- Problems with motion capture include:
  - How do we map motion paths to objects of a different shape and scale (the ostrich-to-T. Rex problem)?
  - How do we deal with noise and missing data?
  - How can we reduce the number of samples?
  - How can we generalize motion? I.e. we have a walk and a turn. How do we get an actor to follow a path?

26.12.4 Particle Systems

- How do we model "fuzzy" objects? Objects with
  - "soft" boundary
  - changing boundary
  - chaotic behavior

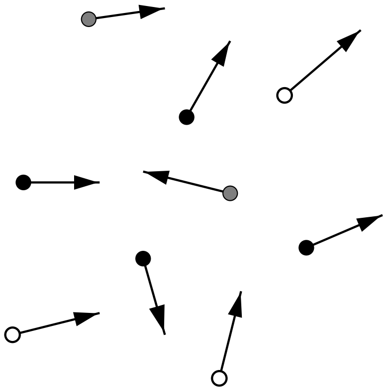
  Examples: Clouds, fire, water, grass, fur
- Could texture map "fuzziness," but changing shape is hard
- Another way is with Particle Systems

  Basic idea: model fuzzy objects as large colletion of particles

  Used to model "Genesis" effect in The Wrath of Kahn

Particle Systems :

- Animate fuzzy or gaseous objects as a changing cloud of "particles"
- Aim for overall effects by having many, simple particles
- Each particle has properties:
  - Geometry (point, sphere, line segment)
  - Position
  - Velocity vector
  - Size
  - Colour
  - Transparency
  - State
  - Lifetime



Animating Particles Systems :

- New particles are born, old ones die
- State information describes type of particle
- At each time step, based on state
  - Update attributes of all particles
  - Delete old particles
  - Create new particles
  - Display current state of all particles
- To anti-alias, draw line segment from old position to new position
- To draw grass or fur, draw entire trajectory of particle

Particle System Details :

- Easy to implement state machine
- Hard part is determining how to make particles act
- Particles usually independent
  Don't know about one another
- Often will model gravity when updating particles
- Render fire etc using transparency
  Brighter where particle system denser

26.12.5 Flocking

- Want to animate groups of animals
  Birds, fish, herds
- Motion of individuals is semi-independent
  Too many individuals to key frame each one
- Fewer elements than particle systems, but more interelement interaction

- Relative placing not ridgid
  Flock of birds vs airplanes in formation
  (migrating geese, etc., being more like the latter)

Two main forces at work

- Collision avoidance
  - Avoid hitting objects (trees, buildings, etc.)
  - Avoid hitting one another
- Flock centering
  - Each member trying to remain a member of flock
  - Global flock centering too restricting
    Does not allow flock splitting to pass around objects

Local Control

- Controlling each flock member with local behaviour rules
  Computationally desirable
  Appears to be how flocks work in real world
- No reference to global conditions of flock, environment
- Three things to model:
  - Physics
    Gravity, collisions
  - Perception
    Information flock member has with which to make decisions
  - Reasoning/reaction
    Rules by which flock member decides where to go

Perception

- Aware of itself and two or three neighbors
- What's in front of it
- Doesn't follow a designated leader
- No knowledge of global center

Reasoning/reaction

- Collision avoidance
- Flock centering

- Velocity matching

  Try to match velocity of neighbors

  Helps with flock centering and helps avoid collisions

Additional details

- Global control

  Animator needs to direct flock.

- Flock leader

  Usually one member whose path is scripted (global control)

  Not realistic (leader changes in real flocks),
  but usually not noticed and easier to deal with

- Splitting and Rejoining

  Flocks pass around objects, splitting them

  Difficult to balance rejoining with collision avoidance

27 Assignments

♠ *These are short descriptions of each of the assignments, meant to be presented and discussed in class. Longer descriptions and formal requirements are available on the course web.*

27.1 Assignment 0: Introduction

♠ *This assignment is meant to walk you through most of the tools that will be used in the course, and get you used to the environment. The TA's will assume you are familiar with the environment when marking later assignments.*

27.1.1 Assignment 0

The goals of this assignment include:

- Familiarization with the course computing environment.
- Setting up your account and directories.
- Modifying a UI built with Tcl/Tk.
- A trial assignment submission.

This assignment is not worth any marks. However, we *strongly suggest you do it*:

- It's easy.
- You have to learn most of it eventually anyways.
- It'll save you from losing marks over dumb mistakes in protocol.

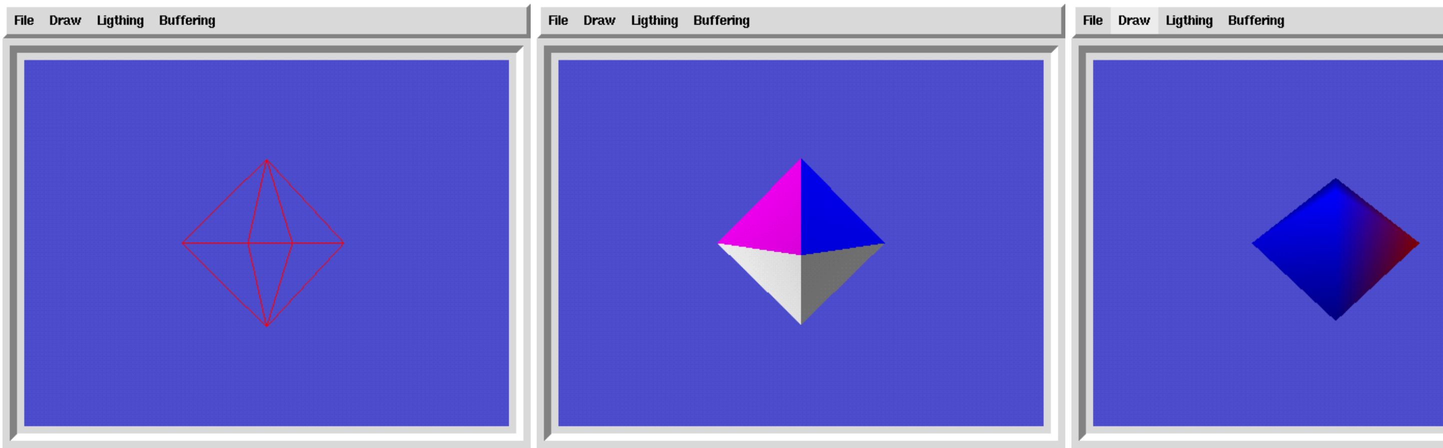
27.2 Assignment 1: Introduction to OpenGL

27.2.1 Assignment 1

In this assignment you will learn how to use OpenGL to render a polygonal model.

- Draw polygons
- Lighting
- Transformations
- Tcl/Tk UI

27.2.2 Screen Shots



Sample screen shots. Some details may vary term to term.

27.3 Assignment 2: Frames and Perspective

27.3.1 Assignment 2

This assignment includes:

- Modeling, Viewing, and Perspective Transformations
- 3D Line/Plane Clipping
- Menus and Valuators
- Double Buffering

Your program will manipulate a cube and 3 sets of coordinate frames.

The cube is in a modelling frame, is transformed to a world frame, and then into a viewing frame which is projected onto a window.

You will draw gnomons for the modelling and world frames.

27.3.2 Models and Coordinate Frames

There are six entities you need to consider:

The Cube Model: Defined on the unit points $[\pm1, \pm1, \pm1]$.

The Cube Model Frame: For entering the initial coordinates of the Cube Model.

The Cube Model Gnomon: A *graphic representation* of the Cube Model Frame.

The World Frame: A coordinate system for describing the position and orientation of objects in the scene.

The World Frame Gnomon: A *graphic representation* of the World Frame.

The Viewing Frame: A *coordinate system* for representing a view of the scene relative to the eyepoint and window.

27.3.3 Features

Your program will support the following features:

Menus: Selection of transformation mode and coordinate system.

Valuators: Mouse movement will be used to specify scalar parameters.

Modelling Transformations: Rotation, translation, and scale.

Viewing Transformations: Camera rotation and translation, perspective.

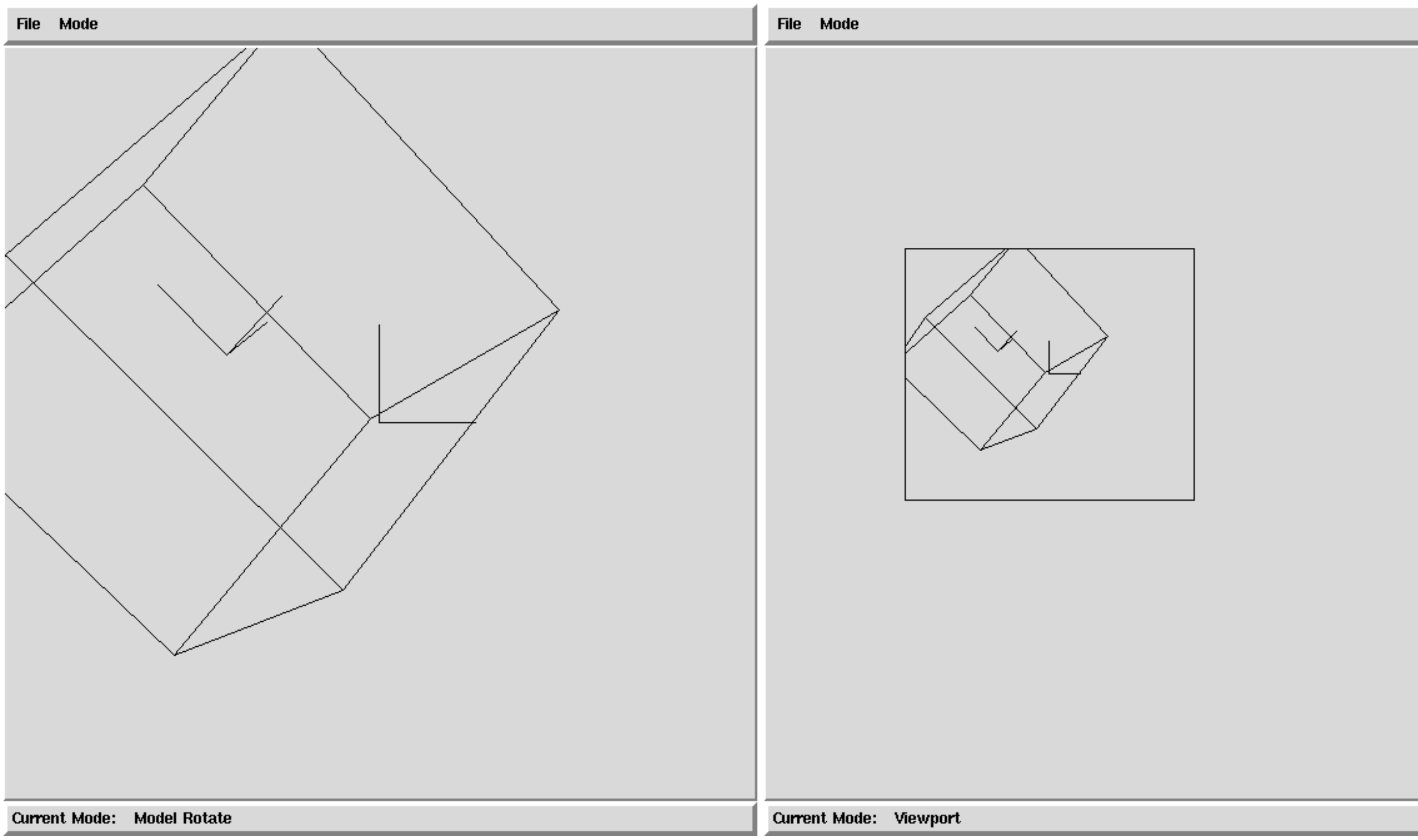
3D Line Clipping: To avoid divide by zero, you need to clip all lines to the near plane. Clipping to other planes is optional.

Viewport: Clip the drawing to a 2D viewport.

Double Buffering: To avoid redraw flicker.

Tcl/Tk will be used for user interface support.
OpenGL may only be used for 2D line drawing in a [-1,1]x[-1,1] NDCS.
You must implement all transformations and clipping yourselves.

27.3.4 Screen Shots



Current Mode: Model Rotate

Current Mode: Viewport

Sample screen shots. Some details may vary term to term.

27.4 Assignment 3: Hierarchical Modelling

27.4.1 Assignment 3

This assignment includes:

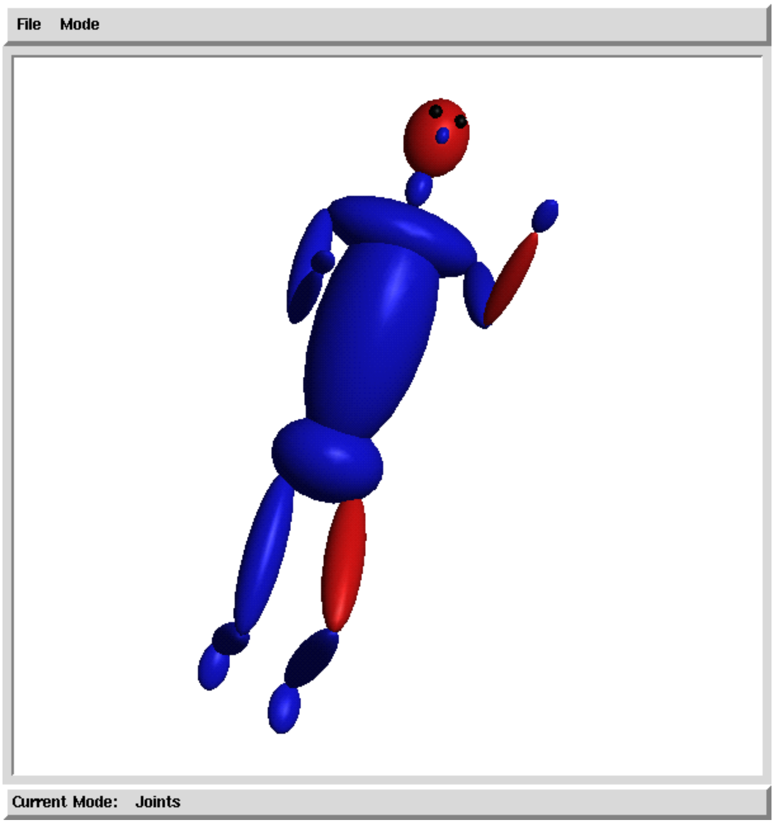
- Hierarchical Models (and data structures).
- Matrix Stacks
- 3D Picking
- Z-Buffer Hidden Surface Removal
- Lighting Models and Shaded Polygons
- Display Lists
- Virtual Sphere or Arcball

You may use any feature of OpenGL you desire, including transformations and lighting models, to implement this assignment.

Tcl/Tk should be used to implement the user interface.

♠ *Readings: Blinn, Chapter 3.*

27.4.2 Screen Shots



Sample screen shots. Some details may vary term to term.

27.5 Assignment 4: A Raytracer

27.5.1 Assignment 4

In this assignment you will investigate the implementation of

- Ray Tracing
- Colour and Shading

Some sample code will be provided to get you started.
Your raytracer will implement at least spheres, cubes, polygons, and the Phong lighting model.
Only primary and shadow rays are required in the basic implementation.

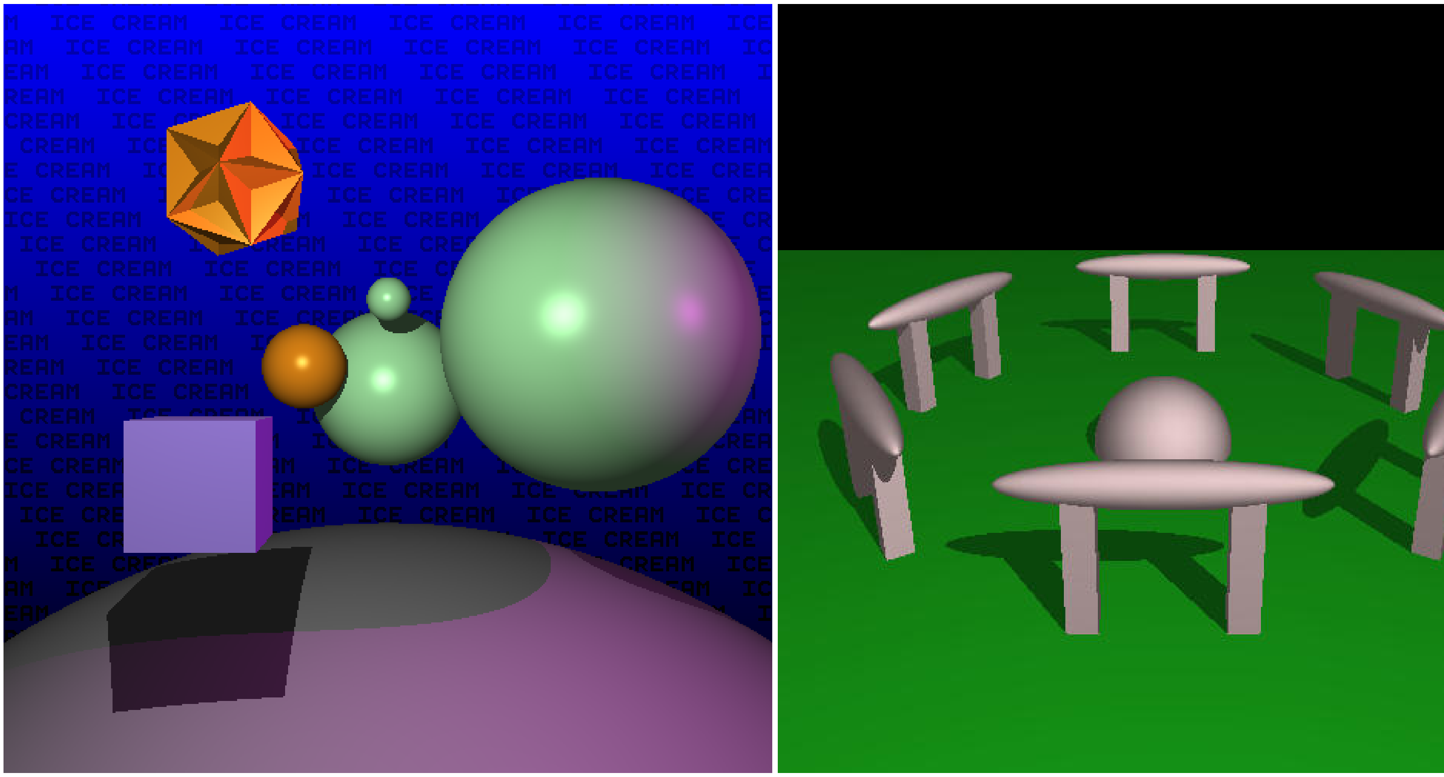
27.5.2 Additional Feature Requirement

You will also be asked to implement one additional feature of your choice. For example, you could implement:

- Secondary rays for reflective surfaces
- Depth of field using (jittered) aperture supersampling
- Antialiasing using (jittered) pixel supersampling

- A fisheye lens
- Texture or bump mapping
- CSG operations
- Additional object types

27.5.3 Screen Shots



Sample screen shots. Some details may vary term to term.

27.6 Assignment 5: The Project

27.6.1 Assignment 5: The Project

- The project submission is distributed over three phases:
  - A proposal
  - A revised proposal, addressing our comments.
  - A final submission and demonstration.
- Your project should have a significant computer graphics content, but is otherwise unconstrained.
- Your proposal needs to include an Objectives list, which we will use to mark you. Choose your topic and Objectives carefully!

- A project does not have to have an interactive user interface, but if it does not you will have to work harder on the algorithms to make up for it.
- You will have to prepare documentation for your project, and the quality of this documentation will strongly influence your mark.
- Ask us for project suggestions; also, it would be wise to run your idea past us *before* submitting your proposal.

♠ *Readings for project ideas: Blinn, Chapters 4, 6, 8, 9, 10.*