

CS 488/688  
Fall 2023  
Stephen Mann

## Contents

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Administration</b>                              | <b>5</b>  |
| 1.1      | General Information . . . . .                      | 5         |
| 1.2      | Topics Covered . . . . .                           | 6         |
| 1.3      | Assignments . . . . .                              | 6         |
| <b>2</b> | <b>Introduction</b>                                | <b>9</b>  |
| 2.1      | History . . . . .                                  | 9         |
| 2.2      | Pipeline . . . . .                                 | 10        |
| 2.3      | Primitives . . . . .                               | 12        |
| 2.4      | Algorithms . . . . .                               | 12        |
| 2.5      | APIs . . . . .                                     | 12        |
| <b>3</b> | <b>Devices and Device Independence</b>             | <b>13</b> |
| 3.1      | Calligraphic and Raster Devices . . . . .          | 13        |
| 3.2      | How a Monitor Works . . . . .                      | 13        |
| 3.3      | Physical Devices . . . . .                         | 14        |
| <b>4</b> | <b>Device Interfaces</b>                           | <b>17</b> |
| 4.1      | Device Input Modes . . . . .                       | 17        |
| 4.2      | Application Structure . . . . .                    | 17        |
| 4.3      | Polling and Sampling . . . . .                     | 18        |
| 4.4      | Event Queues . . . . .                             | 18        |
| 4.5      | Toolkits and Callbacks . . . . .                   | 19        |
| 4.6      | Example for Discussion . . . . .                   | 20        |
| <b>5</b> | <b>Geometries</b>                                  | <b>23</b> |
| 5.1      | Geometry Overview . . . . .                        | 23        |
| 5.2      | Vector Spaces . . . . .                            | 23        |
| 5.3      | Affine Spaces . . . . .                            | 24        |
| 5.4      | Inner Product Spaces . . . . .                     | 25        |
| 5.5      | Euclidean Spaces . . . . .                         | 25        |
| 5.6      | Cartesian Space . . . . .                          | 26        |
| 5.7      | Why Vector Spaces Inadequate . . . . .             | 26        |
| 5.8      | Summary of Geometric Spaces . . . . .              | 27        |
| <b>6</b> | <b>Affine Geometry and Transformations</b>         | <b>29</b> |
| 6.1      | Linear Combinations . . . . .                      | 29        |
| 6.2      | Affine Combinations . . . . .                      | 29        |
| 6.3      | Affine Transformations . . . . .                   | 31        |
| 6.4      | Matrix Representation of Transformations . . . . . | 33        |
| 6.5      | Geometric Transformations . . . . .                | 34        |
| 6.6      | Compositions of Transformations . . . . .          | 36        |
| 6.7      | Change of Basis . . . . .                          | 36        |
| 6.8      | Ambiguity . . . . .                                | 39        |

|  |           |
|--|-----------|
| <i>CONTENTS</i>  | <b>3</b>  |
| 6.9 3D Transformations . . . . .                       | 40        |
| 6.10 World and Viewing Frames . . . . .                | 43        |
| 6.11 Normals . . . . .                                 | 44        |
| <b>7 Windows, Viewports, NDC</b>                       | <b>47</b> |
| 7.1 Window to Viewport Mapping . . . . .               | 47        |
| 7.2 Normalized Device Coordinates . . . . .            | 49        |
| <b>8 Clipping</b>                                      | <b>51</b> |
| 8.1 Clipping . . . . .                                 | 51        |
| <b>9 Projections and Projective Transformations</b>    | <b>55</b> |
| 9.1 Projections . . . . .                              | 55        |
| 9.2 Why Map Z? . . . . .                               | 60        |
| 9.3 Mapping Z . . . . .                                | 62        |
| 9.4 3D Clipping . . . . .                              | 64        |
| 9.5 Homogeneous Clipping . . . . .                     | 64        |
| 9.6 Pinhole Camera vs. Camera vs. Perception . . . . . | 67        |
| <b>10 Transformation Applications and Extensions</b>   | <b>71</b> |
| 10.1 Rendering Pipeline Revisited . . . . .            | 71        |
| 10.2 A2 World and View Frames . . . . .                | 71        |
| <b>11 Polygons</b>                                     | <b>75</b> |
| 11.1 Polygons – Introduction . . . . .                 | 75        |
| 11.2 Polygon Clipping . . . . .                        | 76        |
| 11.3 Polygon Scan Conversion . . . . .                 | 79        |
| 11.4 Dos and Don'ts . . . . .                          | 80        |
| <b>12 Hidden Surface Removal</b>                       | <b>83</b> |
| 12.1 Hidden Surface Removal . . . . .                  | 83        |
| 12.2 Backface Culling . . . . .                        | 83        |
| 12.3 Painter's Algorithm . . . . .                     | 84        |
| 12.4 Warnock's Algorithm . . . . .                     | 85        |
| 12.5 Z-Buffer Algorithm . . . . .                      | 86        |
| 12.6 Comparison of Algorithms . . . . .                | 87        |
| <b>13 Hierarchical Models and Transformations</b>      | <b>89</b> |
| 13.1 Hierarchical Transformations . . . . .            | 89        |
| 13.2 Hierarchical Data Structures . . . . .            | 91        |
| <b>14 Rotating Around Arbitrary Axes</b>               | <b>97</b> |
| 14.1 Rotation Around Arbitrary Axis . . . . .          | 97        |
| 14.2 3D Rotation User Interfaces . . . . .             | 98        |
| 14.3 The Virtual Sphere . . . . .                      | 99        |

|  |            |
|--|------------|
| <i>CONTENTS</i>  | 4          |
| <b>15 Picking and 3D Selection</b>                                 | <b>101</b> |
| 15.1 Picking and 3D Selection . . . . .                            | 101        |
| <b>16 Colour and the Human Visual System</b>                       | <b>103</b> |
| 16.1 Colour . . . . .  | 103        |
| <b>17 Reflection and Light Source Models</b>                       | <b>109</b> |
| 17.1 Goals . . . . .   | 109        |
| 17.2 Lambertian Reflection . . . . .                               | 109        |
| 17.3 Attenuation . . . . .   | 111        |
| 17.4 Coloured Lights, Multiple Lights, and Ambient Light . . . . . | 113        |
| 17.5 Specular Reflection . . . . .                                 | 114        |
| <b>18 Shading</b>  | <b>117</b> |
| 18.1 Introduction . . . . .  | 117        |
| 18.2 Gouraud Shading . . . . .                                     | 117        |
| 18.3 Phong Shading . . . . .                                       | 121        |
| <b>19 Graphics Hardware</b>  | <b>123</b> |
| 19.1 Graphics Hardware . . . . .                                   | 123        |
| 19.2 Shader Idea . . . . .   | 124        |
| 19.3 High-Level Shading Languages . . . . .                        | 125        |
| <b>20 Ray Tracing</b>  | <b>129</b> |
| 20.1 Fundamentals . . . . .  | 129        |
| 20.2 Intersection Computations . . . . .                           | 130        |
| 20.3 Shading . . . . .   | 133        |
| 20.4 Recursive Ray Tracing . . . . .                               | 135        |
| 20.5 Surface Information . . . . .                                 | 136        |
| 20.6 Modeling and CSG . . . . .                                    | 138        |
| 20.7 Texture Mapping . . . . .                                     | 140        |
| 20.8 Bounding Boxes, Spatial Subdivision . . . . .                 | 145        |
| <b>21 Aliasing</b>   | <b>147</b> |
| 21.1 Aliasing . . . . .  | 147        |
| <b>22 Bidirectional Tracing</b>                                    | <b>153</b> |
| 22.1 Missing Effects . . . . .                                     | 153        |
| 22.2 Distribution Ray Tracing . . . . .                            | 154        |
| 22.3 Bidirectional Path Tracing . . . . .                          | 154        |
| 22.4 Photon Maps . . . . .   | 155        |
| 22.5 Metropolis Algorithm . . . . .                                | 156        |
| 22.6 Path Tracing . . . . .  | 157        |
| <b>23 Rendering Equation</b>                                       | <b>159</b> |
| 23.1 Definitions and Overview . . . . .                            | 159        |



|  |            |
|--|------------|
| <b>24 Radiosity</b>  | <b>163</b> |
| 24.1 Definitions and Overview . . . . .                            | 163        |
| 24.2 Form Factors . . . . .  | 166        |
| 24.3 Progressive Refinement . . . . .                              | 169        |
| 24.4 Meshing in Radiosity . . . . .                                | 171        |
| 24.5 Stochastic Methods — Radiosity Without Form Factors . . . . . | 173        |
| <b>25 Photon Maps</b>  | <b>177</b> |
| 25.1 Overview . . . . .  | 177        |
| <b>26 Shadows</b>  | <b>181</b> |
| 26.1 Overview . . . . .  | 181        |
| 26.2 Projective Shadows . . . . .                                  | 181        |
| 26.3 Shadow Maps . . . . .   | 182        |
| 26.4 Shadow Volumes . . . . .                                      | 185        |
| <b>27 Modeling of Various Things</b>                               | <b>189</b> |
| 27.1 Fractal Mountains . . . . .                                   | 189        |
| 27.2 L-system Plants . . . . .                                     | 190        |
| 27.3 Buildings . . . . .   | 191        |
| 27.4 Particle Systems . . . . .                                    | 198        |
| <b>28 Polyhedral Data Structures</b>                               | <b>201</b> |
| 28.1 Storage Models . . . . .                                      | 201        |
| 28.2 Euler’s Formula . . . . .                                     | 203        |
| 28.3 Winged Edge . . . . .   | 204        |
| 28.4 Mesh Compression . . . . .                                    | 209        |
| <b>29 Splines</b>  | <b>211</b> |
| 29.1 Constructing Curve Segments . . . . .                         | 211        |
| 29.2 Bernstein Polynomials . . . . .                               | 215        |
| 29.3 Bézier Splines . . . . .                                      | 215        |
| 29.4 Spline Continuity . . . . .                                   | 218        |
| 29.5 Tensor Product Patches . . . . .                              | 224        |
| 29.6 Barycentric Coordinates . . . . .                             | 228        |
| 29.7 Triangular Patches . . . . .                                  | 228        |
| 29.8 Subdivision Surfaces . . . . .                                | 231        |
| 29.9 Implicit Surfaces . . . . .                                   | 234        |
| 29.10 Wavelets . . . . .   | 240        |
| <b>30 Non-Photorealistic Rendering</b>                             | <b>245</b> |
| 30.1 2D NPR . . . . .  | 245        |
| 30.2 3D NPR . . . . .  | 251        |
| <b>31 Volume Rendering</b>   | <b>255</b> |
| 31.1 Volume Rendering . . . . .                                    | 255        |

|   |            |
|---|------------|
| <b>32 Animation</b>                                 | <b>261</b> |
| 32.1 Overview . . . . .                             | 261        |
| 32.2 Traditional 2D Cel Animation . . . . .         | 261        |
| 32.3 Automated Keyframing . . . . .                 | 262        |
| 32.4 Functional Animation . . . . .                 | 262        |
| 32.5 Motion Path Animation . . . . .                | 264        |
| 32.6 Orientation and Interpolation . . . . .        | 266        |
| 32.7 Quaternions . . . . .                          | 268        |
| 32.8 Animating Camera Motion . . . . .              | 270        |
| 32.9 Tools for Shape Animation . . . . .            | 272        |
| 32.10 Kinematics and Inverse Kinematics . . . . .   | 274        |
| 32.11 Physically-Based Animation . . . . .          | 275        |
| 32.12 Human Motion . . . . .                        | 277        |
| 32.13 Sensor-Actuator Networks . . . . .            | 278        |
| 32.14 Biologically-Based Simulation . . . . .       | 280        |
| 32.15 Morphing . . . . .                            | 281        |
| 32.16 Motion Capture . . . . .                      | 282        |
| 32.17 Flocking . . . . .                            | 283        |
| <b>33 Computational Geometry</b>                    | <b>287</b> |
| 33.1 Introduction . . . . .                         | 287        |
| 33.2 Data Structures . . . . .                      | 287        |
| 33.3 Convex Hull . . . . .                          | 290        |
| 33.4 Voronoi Diagrams . . . . .                     | 291        |
| 33.5 Delaunay Triangulation . . . . .               | 292        |
| 33.6 More Computational Geometry . . . . .          | 295        |
| <b>34 Assignments</b>                               | <b>297</b> |
| 34.1 Assignment 0: Introduction . . . . .           | 297        |
| 34.2 Assignment 1: Introduction to OpenGL . . . . . | 297        |
| 34.3 Assignment 2: Frames and Perspective . . . . . | 298        |
| 34.4 Assignment 3: Hierarchical Modelling . . . . . | 300        |
| 34.5 Assignment 4: A Raytracer . . . . .            | 301        |
| 34.6 Assignment 5: The Project . . . . .            | 302        |

These notes contain the material that appears on the overhead slides (or on the computer) shown in class. You should, however, read the course text and attend lectures to fill in the missing details. The material in this course is organized around the assignments, with the material at the end of the course being a sequence of topics not covered on the assignments. Some of the images shown on the slides cannot be included in the printed course notes for copyright reasons.

The following former CS 488/688 students have images appearing in these notes:

- Tobias Arrskog
- Franke Belme
- Jerome Carrière
- Bryan Chan
- Stephen Chow
- David Cope
- Eric Hall
- Matthew Hill
- Andrew Lau
- Henry Lau
- Tom Lee
- Bonnie Liu
- Ian McIntyre
- Ryan Meridith-Jones
- Zaid Mian
- Ashraf Michail
- Stephen Sheeler
- Selina Siu



# 1 Administration

## 1.1 General Information

**Instructors:** Stephen Mann

**Electronic Resources:**

Web: <http://www.student.cs.uwaterloo.ca/~cs488/>

Newsgroup: piazza

Assignments submitted in Learn

If moved online:

Crowdmark (midterm, final)

Learn (videos, quizzes)

**Texts:**

- Hearn, Baker, Carithers,  
*Computer Graphics with OpenGL* (**required!**)
- Course Overview and Notes (online)
- *OpenGL Programming Guide* (optional)

**Assignments:**

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

**Marking Algorithm:**

Assignments: 40%; Project proposal: 5%; Project: 20%; Final: 35%

Must obtain 50% in both Programming and Exam portions

Note: if shifted to online, there will be online quizzes and weights will change to reflect this.

**Programming Environment:**

C++, OpenGL, Lua; Linux PCs.

**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).

- Course gallery - student images, with name, term, email
- Freedom of Information and Privacy Protection Act
- If you do not want name/images in gallery, let us know

If your work is in the gallery and you want it removed at a later date, let us know and we will remove it.

## 1.2 Topics Covered

- Graphics Pipeline and Hardware
- Mathematical Foundations:
  - Affine and Projective Geometry, Linear Algebra and Transformations
  - Numerical Analysis and Splines
- 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)

## Course Goals

- Strong understanding of fundamentals of computer graphics
  - Affine and projective transformation, matrix representations
  - Light, shading, and rendering basics
  - Texture mapping
- Exposure to advanced topics in computer graphics
  - Global illumination
  - Graphics card techniques
  - Computer Animation
- Showcase project

## 1.3 Assignments

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

**Environment:** Linux PCs.

You are expected to share machines. Be nice!

Your solution must run on the machines in MC3007

**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.

**No lates!**

**Assignment Format:**

- *Very strict!*
- Read assignment policies in course notes.
- **Do A0 to avoid losing marks later.**
- See (and use) checklist.
- See course notes for assignment specs and objectives.





## 2 Introduction

### 2.1 History

#### A Brief History of Computer Graphics

**Early 60s:** 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 Engelbart 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 digital image processing

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

**1975:** Bui-Tuong 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

**1976:** Future World, first 3D computer graphics in movie

**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 Khan. Particle systems and obvious CG

**1984:** The Last Starfighter, 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

The Abyss: first 3D CG  
movie character

**1993:** Jurassic Park—game changer

**1995:** Toy Story (Pixar and Disney), 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/medical visualization, artistic rendering, image based rendering, photon maps, etc.

**00's:** Real-time photorealistic rendering on consumer hardware.

## Awards

\* are uWaterloo

**1996** , Academy Award, Technical Achievement, Marceli Wein\*, Nestor Burtnick: Computer Assisted Key Framing

**1997** , Academy Award, Rob Krieger\*: Alias

**2002** , Academy Award, Paul Breslin\*: Procedural Modeling

**2018** , Academy Award, Mike Jutan\*: ILM

**2018** , Academy Award, Christin Barghiel, Jeff Lait\*, Mark Tucker\*: Houdini

**2020** , Turing Award: Ed Catmull, Pat Hanrahan:  
Fundamental contributions to 3-D computer 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 Pipeline

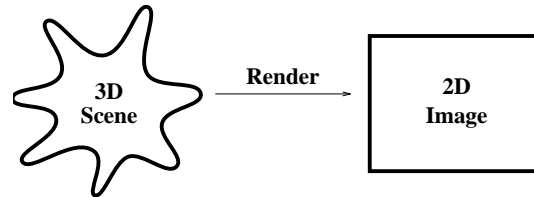
We begin with a description of **forward rendering**, which is the kind of rendering usually supported in hardware and is the model OpenGL uses. In forward rendering, rendering primitives are transformed, usually in a conceptual pipeline, from the model to the device.

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.

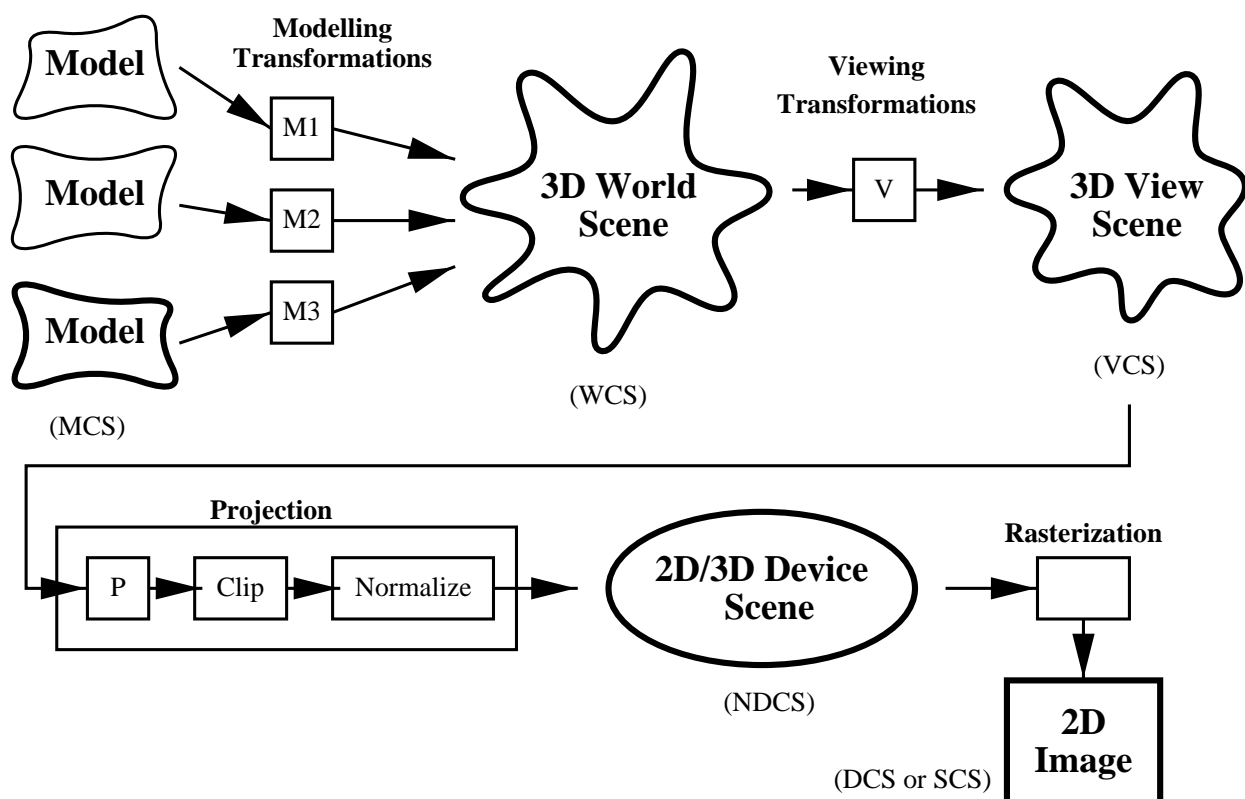
Both approaches have advantages and disadvantages.

## The Graphics 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.
- Classically, “model” to “scene” to “image” conversion broken into finer steps, called the **graphics pipeline**.
- Parts of pipeline implemented in graphics hardware to get interactive speeds.
- The basic **forward projection** pipeline looks like:



(Readings: McConnell, Section 1.2. 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.)

## 2.3 Primitives

- Models are composed of/converted to **geometric primitives**.
- Typical rendering primitives directly supported in hardware:
  - Points (single pixels)
  - Line Segments
  - Polygons (perhaps only convex polygons or triangles).
- Modelling primitives also include
  - Piecewise polynomial (spline) curves
  - Piecewise polynomial (spline) surfaces
  - Implicit surfaces (quadrics, bobbies, etc)
  - Other...

## 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 APIs

**Application Programming Interfaces (APIs):**

**Xlib, GDI:** 2D rasterization.

**PostScript, PDF, SVG:** 2D transformations, 2D rasterization

**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.

## 3 Devices and Device Independence

### 3.1 Calligraphic and Raster Devices

#### Calligraphics and Raster Devices

**Calligraphic Display Devices** draw polygon and line segments directly:

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

represent an image as a regular grid of **samples**.

**Raster Display Devices**

- Each sample is usually called a **pixel** (“picture element”)
- Rendering requires **rasterization algorithms** to quickly determine a sampled representation of geometric primitives.

### 3.2 How a Monitor Works

#### How a CRT Works

**Raster Cathode Ray Tubes (CRTs)** Once was most common display device

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

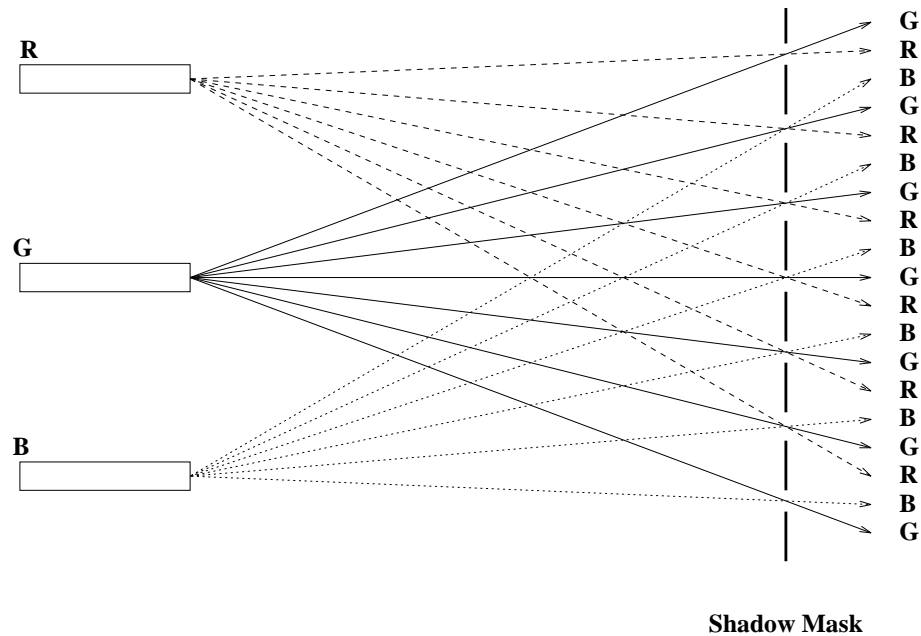
Electron beam scanned in regular pattern of horizontal **scanlines**.

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

Intensity of electron beam modified by the pixel value.

**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.



## How Liquid Crystal Displays Work

### Liquid Crystal Displays (LCDs)

- 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.

(Readings: McConnell, Section 1.6. 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 Physical Devices

### Physical Devices

Physical input devices include

- Dials (Potentiometers)

- Sliders
- 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 Device Interfaces

### 4.1 Device Input Modes

#### 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)/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).

### 4.2 Application Structure

#### 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.

### 4.3 Polling and Sampling

#### Polling

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.

#### Sampling

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)

### 4.4 Event Queues

#### 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
  - First waiting event
  - Highest priority event
  - First event of some category
  - All events
- Application can also
  - Specify which events should be placed in queue
  - Clear and reset the queue
  - Etc.

## Event Loop

- With this model, events processed in *event loop*

```
while ( event = NextEvent() ) {  
    switch (event.type) {  
        case MOUSE_BUTTON:...  
        case MOUSE_MOVE:...  
        ...  
    }  
}
```

For more sophisticated queue management,

- application merely registers event-process pairs
- queue manager does all the rest  
“if event E then invoke process P.”
- 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.
  - 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.

## 4.5 Toolkits and Callbacks

### 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.  
Graphical representation that suggests 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 hierarchy.

Some UI toolkits:

- Tk
- Tkinter
- Motif
- Gtk
- Qt
- WxWindows
- FLTK
- AWT
- Swing
- XUL
- ...

UI toolkits recommended for projects: Gtkmm, GLUT, SDL.

## 4.6 Example for Discussion

### Example for Discussion

```
#include <QApplication>
#include <QDesktopWidget>
#include <QPushButton>

void test() { std::cout << "Hello, World!" << std::endl; }
int main( int argc, char *argv[] ) {
    QApplication app(argc, argv);
```

```
    QMainWindow window;
    window.resize(window.sizeHint());
    QPushButton* pb = new QPushButton("Click me!");
    window.setCentralWidget(pb);
    QObject::connect(pb, SIGNAL(clicked()), &window, SLOT(close()));
    window.show();
    return app.exec();
}
```



## 5 Geometries

### 5.1 Geometry Overview

#### Geometry For Computer Graphics

You probably learned geometry relative to coordinate axes:  
points and vectors 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.
- Glosses over mathematical details...

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

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

**Goal:** Find appropriate space for computer graphics

#### Mathematical Foundation

- Geometric Spaces: Vector, Affine, Euclidean, Cartesian, Projective
- Affine Geometry
- Affine Transformations
- Perspective
- Projective Transformations
- Matrix Representation of Transformations
- Viewing Transformations
- Quaternions and Orientation

### 5.2 Vector Spaces

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

## 5.3 Affine Spaces

### 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**):

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

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



## 5.4 Inner Product Spaces

### Inner Product Spaces

- For any vector space  $V$ , an *inner product* is a binary operator  $\vec{u} \cdot \vec{v} \in \mathbb{R}$  with

$$\begin{aligned}\vec{u} \cdot \vec{v} &= \vec{v} \cdot \vec{u} & (\alpha \vec{u}) \cdot \vec{v} &= \alpha(\vec{u} \cdot \vec{v}) \\ (\vec{u} + \vec{v}) \cdot \vec{w} &= \vec{u} \cdot \vec{w} + \vec{v} \cdot \vec{w} \\ \vec{u} \cdot \vec{u} &\geq 0 & \vec{u} \cdot \vec{u} = 0 &\text{ iff } \vec{u} = 0.\end{aligned}$$

## 5.5 Euclidean Spaces

### Euclidean Spaces

**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:

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

**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.

## 5.6 Cartesian Space

### Cartesian Space

**Cartesian Space:** A Euclidean space with an standard **orthonormal frame**  $(\vec{i}, \vec{j}, \vec{k}, \mathcal{O})$ .

**Orthogonal:**  $\vec{i} \cdot \vec{j} = \vec{j} \cdot \vec{k} = \vec{k} \cdot \vec{i} = 0$ .

**Normal:**  $|\vec{i}| = |\vec{j}| = |\vec{k}| = 1$ .

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

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  $\vec{v} = v_x \vec{i} + v_y \vec{j} + v_z \vec{k}$ .
- 1 for points:  $P = (p_x, p_y, p_z, 1)$  means  $P = p_x \vec{i} + p_y \vec{j} + p_z \vec{k} + \mathcal{O}$ .
- 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.

## 5.7 Why Vector Spaces Inadequate

- Why not use vector spaces instead of affine spaces?

If we trace the tips of the vectors, then we get curves, etc.

- First problem: no metric

Solution: Add a metric to vector space

- Bigger problem:

- Want to represent objects with small number of “representatives”

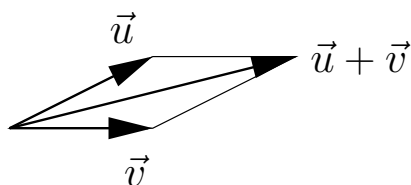
Example: three points to represent a triangle

- Want to translate, rotate, etc., our objects by translating, rotating, etc., the representatives

(vectors don’t translate, so we would have to define translation)

Let  $\vec{u}, \vec{v}$  be representatives

Let  $\vec{u} + \vec{v}$  be on our object



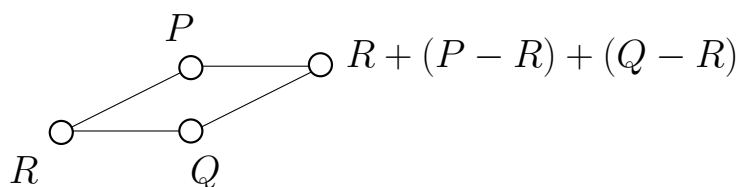
Let  $T(\vec{u})$  be translation by  $\vec{t}$  defined as  $T(\vec{u}) = \vec{u} + \vec{t}$ .  
Then

$$\begin{aligned} T(\vec{u} + \vec{v}) &= \vec{u} + \vec{v} + \vec{t} \\ &\neq T(\vec{u}) + T(\vec{v}) \\ &= \vec{u} + \vec{v} + 2\vec{t} \end{aligned}$$

**Note: this definition of translation only used on this slide! Normally, translation is identity on vectors!**

Let  $P$  and  $Q$  be on our object

Let  $R$  be a third point, and  $R + (P - R) + (Q - R)$  be on our object



Let  $T(\vec{u})$  be translation by  $\vec{t}$  (i.e.,  $T(P) = P + \vec{t}$ ,  $T(\vec{v}) = \vec{v}$ ).  
Then

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

## 5.8 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, Point             | $\vec{u} + \vec{v}$ , $\alpha\vec{v}$ , $P + \vec{v}$ , $\vec{u} \cdot \vec{v}$ , $d(P, Q)$ Question: where does the cross                               |
| Cartesian Space | Vector, Point, O.N. Frame | $\vec{u} + \vec{v}$ , $\alpha\vec{v}$ , $P + \vec{v}$ , $\vec{u} \cdot \vec{v}$ , $d(P, Q)$ , $\vec{i}, \vec{j}, \vec{k}$ , $\mathcal{O}$ product enter? |

(Readings: McConnell: A.7. Watt: 1.1, 1.2. White book: Appendix A. Hearn and Baker: A-2, A-3. )



## 6 Affine Geometry and Transformations

### 6.1 Linear Combinations

#### Linear Combinations

Vectors:

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

By Extension:

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

Linear Transformations:

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

By Extension:

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

Points:

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

By Extension...

### 6.2 Affine Combinations

#### Affine Combinations

Define Point Subtraction:

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

By Extension:

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

Define Point Blending:

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

Alternatively:

$$\text{we may write } Q = a_1 Q_1 + a_2 Q_2 \text{ where } a_1 + a_2 = 1.$$

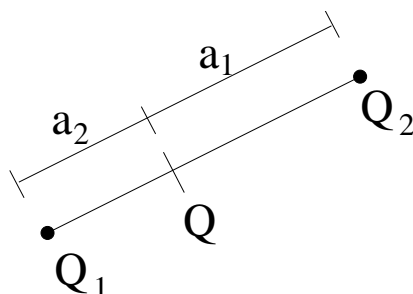
By Extension:

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

Geometrically:

- The following ratio holds for  $Q = a_1Q_1 + a_2Q_2$

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



- 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)

### 6.3 Affine Transformations

#### Affine Transformations

**Let**  $T : \mathcal{A}_1 \mapsto \mathcal{A}_2$ , 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.

#### Affine vs Linear

Which is a larger class of transformations: Affine or Linear?

- All affine transformations are linear transformations on vectors.
- Consider identity transformation on vectors.  
There is one such linear transformation.
- Consider Translations:

- Identity transformation on vectors
- Infinite number different ones (based on effect on points)

Thus, there are an infinite number of affine transformations that are identity transformation on vectors.

- What makes affine transformations a bigger class than linear transformation is their translational behaviour on points.

#### Extending Affine Transformations to Vectors

Suppose we only have  $T$  defined on points.

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

- There exist 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  $P + \vec{v}$ :

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

Can now show that the definition is well defined.

**Theorem:** Affine transformations map parallel lines to parallel lines.

### Mapping Through an Affine Transformation

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  $(p_1, p_2, 1)$ .  
( $P = p_1\vec{v}_1 + p_2\vec{v}_2 + 1\mathcal{O}_{\mathcal{V}}$ )

$\mathcal{O}_{\mathcal{V}}$  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  $F_{\mathcal{B}}$ ?

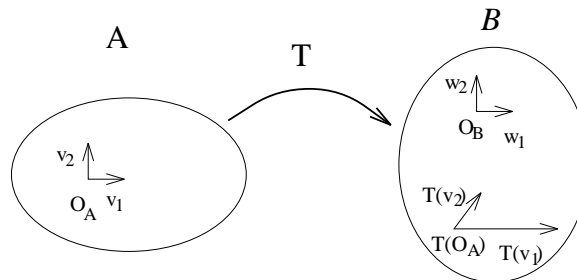
**Fact:** An affine transformation is completely characterized by the image of a frame in the domain:

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

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.



(Readings: White book, Appendix A)



## 6.4 Matrix Representation of Transformations

### 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  $F = (\vec{v}_1, \vec{v}_2, \mathcal{O}_V)$ :

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

**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 ' $\mathbf{p}$ ' denote both

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

**Transformations:**

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

Then ...

$P = \mathbf{p}$  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  $T(\mathcal{O}_V)$  in  $F_B$ .

## 6.5 Geometric Transformations

### 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.
- 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  $\theta$ .

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

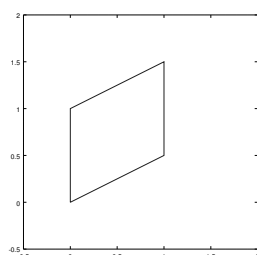
$$\overbrace{\begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}}^{R(\theta)} \begin{bmatrix} x \\ y \\ 0 \text{ or } 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ 0 \text{ or } 1 \end{bmatrix}$$

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

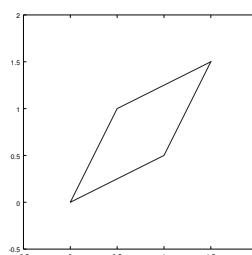
- Applies to points or vectors, is linear.
- Matrix representation:

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

- Easiest to see if we set one of  $\alpha$  or  $\beta$  to zero.



$$\begin{bmatrix} 1 & 0 \\ 0.5 & 1 \end{bmatrix}$$



$$\begin{bmatrix} 1 & 0.5 \\ 0.5 & 1 \end{bmatrix}$$

**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)

## 6.6 Compositions of Transformations

### 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!**

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

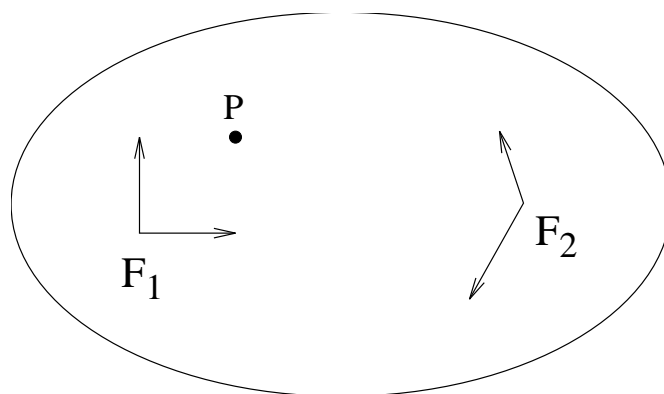
(Readings: McConnell, 3.1.6; Hearn and Baker, Section 5-3; Red book, 5.4; White book, 5.3 )

## 6.7 Change of Basis

### 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  $F_2 = (\vec{v}_1, \vec{v}_2, \mathcal{O}_V)$ .

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

- If  $F_2$  is orthonormal:

$$f_{i,j} = \vec{w}_j \cdot \vec{v}_i,$$

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

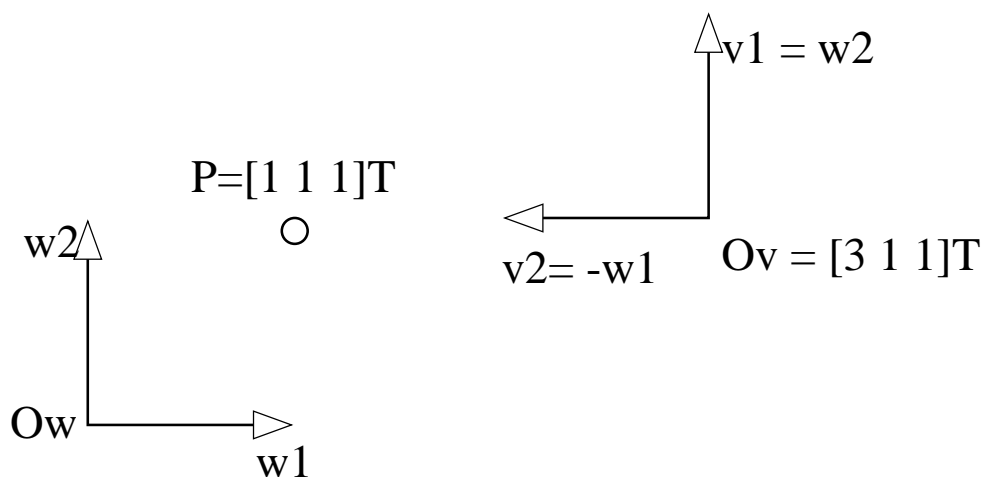
- If  $F_2$  is orthogonal:

$$f_{i,j} = \frac{\vec{w}_j \cdot \vec{v}_i}{\vec{v}_i \cdot \vec{v}_i},$$

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

- 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.

**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.
- In this example,  $F_W$  is the standard frame.
- Note that both  $F_W$  and  $F_V$  are orthonormal.
- The matrix mapping  $F_W$  to  $F_V$  is given by

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

- 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.)

## 6.8 Ambiguity

### 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  $Q \equiv [q_1, q_2, 1]^T$

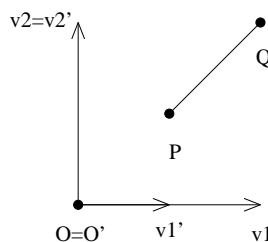
- $P, Q$  relative to  $F = (\vec{v}_1, \vec{v}_2, \mathcal{O})$ ,
- We are given a matrix representation of a transformation  $T$ :

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

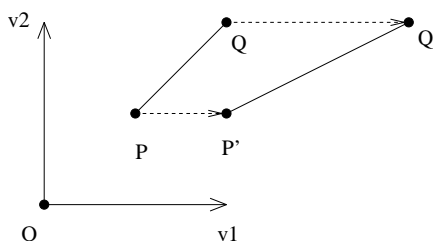
- Consider  $P' = TP$  and  $Q' = TQ$ .
- How do we interpret  $P'$  and  $Q'$ ?

How do we interpret  $P'$  and  $Q'$ ?

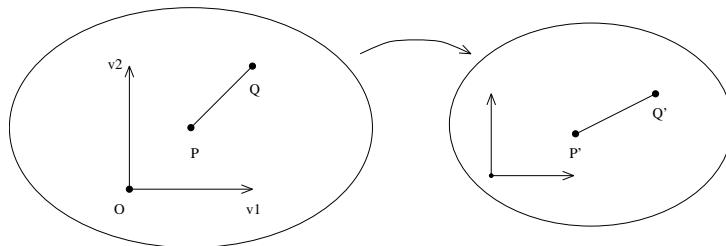
1. Change of Coordinates?



2. Scale?



3. Transformations between spaces?



**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.  $|P' - P| = \sqrt{(2p_1 - p_1)^2 + (p_2 - p_2)^2} = |p_1|$
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

## 6.9 3D Transformations

### 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

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

**Rotation:** About a coordinate axis

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

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

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

**Shear with fixed coordinate direction:**

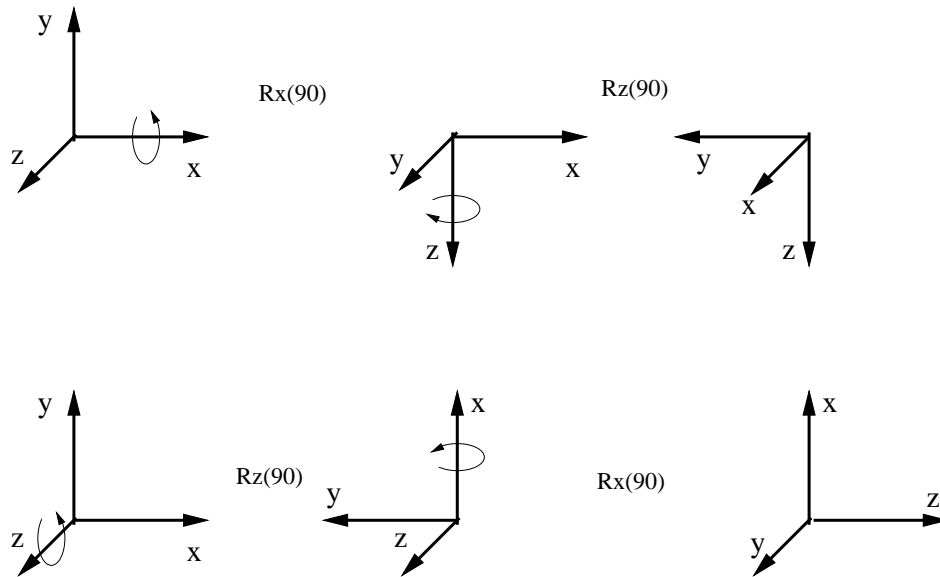
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ \alpha_1 & 1 & 0 & 0 \\ \alpha_2 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & \beta_1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & \beta_2 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & \gamma_1 & 0 \\ 0 & 1 & \gamma_2 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Shear with fixed coordinate plane: one of  $\alpha_i$ s,  $\beta_i$ s,  $\gamma_i$ s = 0

**General Shear:** see book

**Reflection:** see book

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



(Readings: McConnell 3.1; Hearn and Baker, Chapter 11; Red book, 5.7; White book, 5.6 )

- Translation:

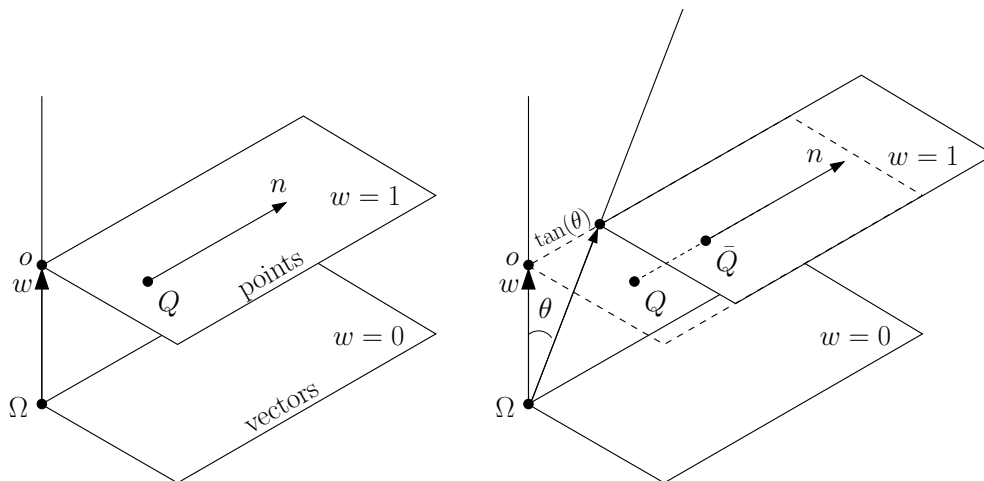
$$T(\Delta x, \Delta y, \Delta z) = \begin{bmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Shear in “ $w$ -subspace” in direction  $n$  by angle  $\theta$ :

$$\begin{bmatrix} I & \tan(\theta)n \\ 0 & 1 \end{bmatrix}$$

where  $I$  is  $3 \times 3$  identity matrix,  $n$  is  $xyz$  vector

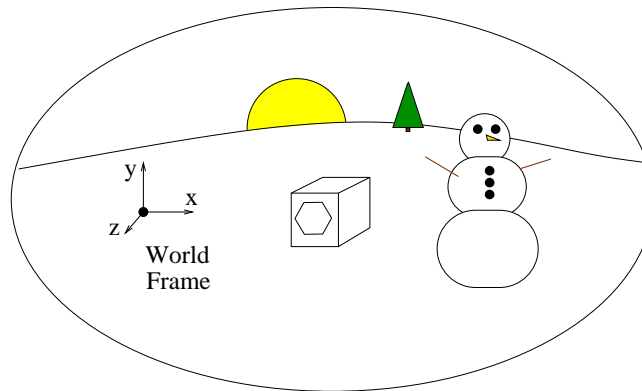
- $\Rightarrow$  Translation in 3D is Shear in 4D



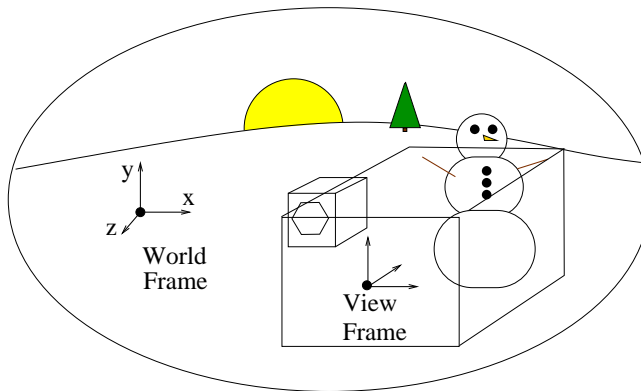
## 6.10 World and Viewing Frames

### 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*.
    - \* This is a **left handed** coordinate system.
  - We could instead specify  $z$  and  $y$  as vectors
    - \*  $z$  is 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

$$F_V = (\vec{i}, \vec{j}, \vec{k}, \mathcal{O})$$

we map  $Q \equiv [q_1, q_2, q_3, 1]^T$  onto

$$F_P = (\vec{u}, \vec{v}, \mathcal{O}')$$

as follows:

$$\text{Ortho}(q_1\vec{v} + q_2\vec{j} + q_3\vec{k} + \mathcal{O}) = q_1\vec{u} + q_2\vec{v} + \mathcal{O}'$$

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?

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

## 6.11 Normals

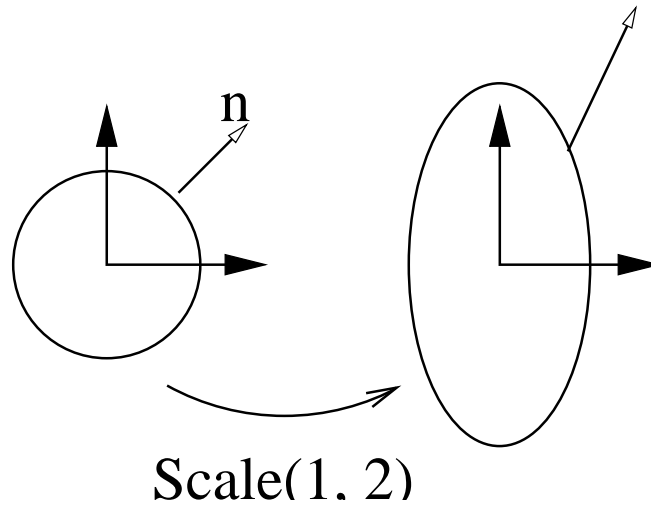
### Transforming Normals

**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:

Consider non-uniform scale of circle, and normal to circle:



Why doesn't normal transform correctly?

- 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 matrix*.
- Suppose we have a transformation  $M$ , a point  $P \equiv \mathbf{p}$ , and a tangent  $\vec{T} \equiv \mathbf{t}$  at  $P$ .
- Let  $M_\ell$  be the “linear part” of  $M$ , i.e., the upper  $3 \times 3$  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. )



## 7 Windows, Viewports, NDC

### 7.1 Window to Viewport Mapping

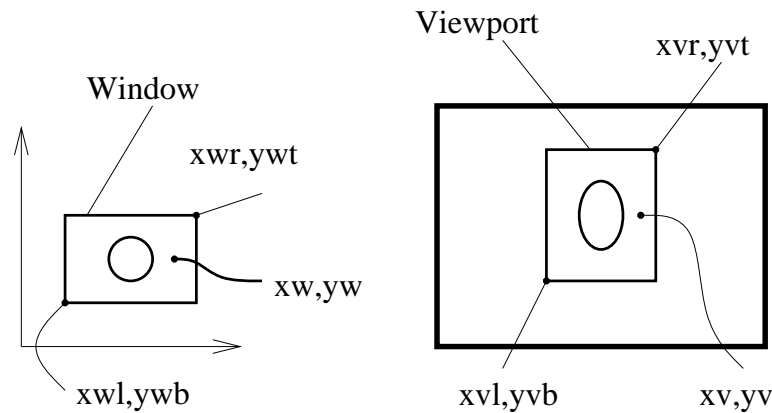
#### Window to Viewport Mapping

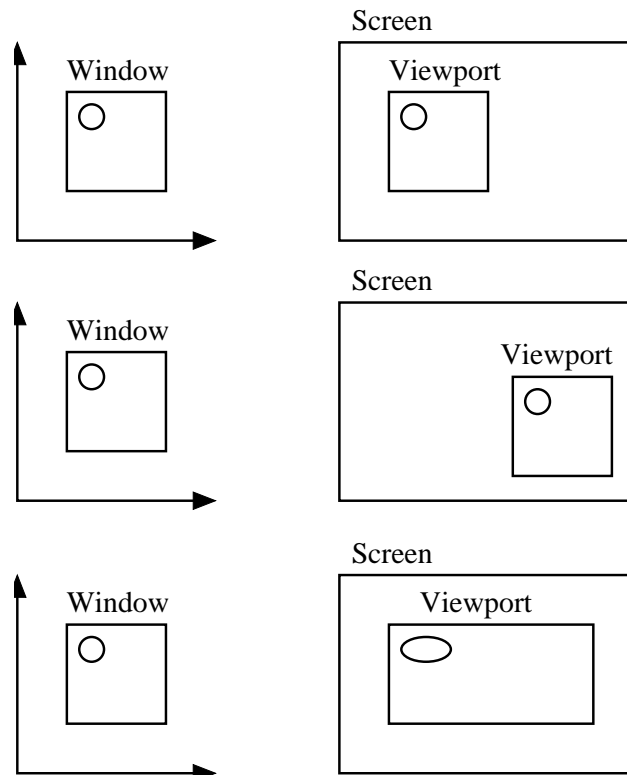
- Start with 3D scene, but eventually project to 2D scene
- 2D scene is infinite plane. Device has a finite visible rectangle. What do we do?
- Answer: map rectangular region of 2D device scene to device.

**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  $(x_v, y_v)$ .

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:

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

- To map  $x_w$  to  $x_v$ :

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

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: McConnell: 1.4. 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.



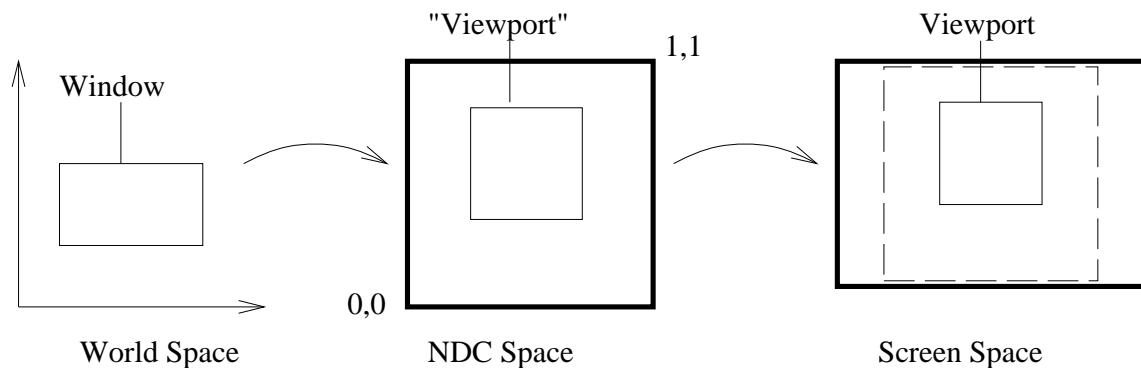
- Scale this  $w$  distance to get a  $v$  distance.
- Add to viewport corner to get  $x_v$ .

)

## 7.2 Normalized Device Coordinates

### Normalized Device Coordinates

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



(Readings: McConnell: 1.4. Watt: none. Hearn and Baker: Sections 2-7, 6-3. Red book: 6.3. White book: 6.5. )



## 8 Clipping

### 8.1 Clipping

#### 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.
- Can use some of the ideas in Liang-Barsky to clip points.

**Parametric representation of line:**

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

or equivalently

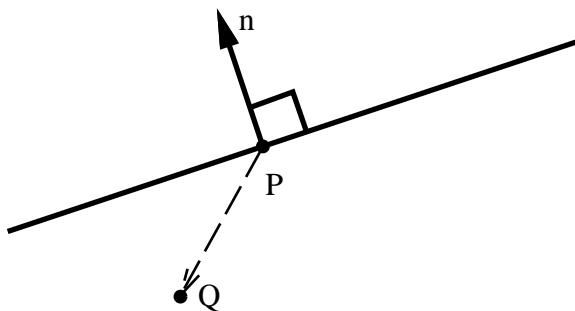
$$L(t) = A + t(B - A)$$

- $A$  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.



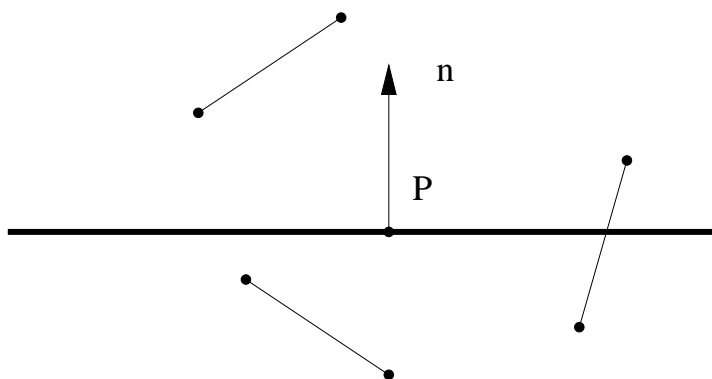
### Clipping a point to a halfspace:

- 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**  $\ell(B) < 0 \implies$  Outside
- $\ell(A) > 0$  **AND**  $\ell(B) > 0 \implies$  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.

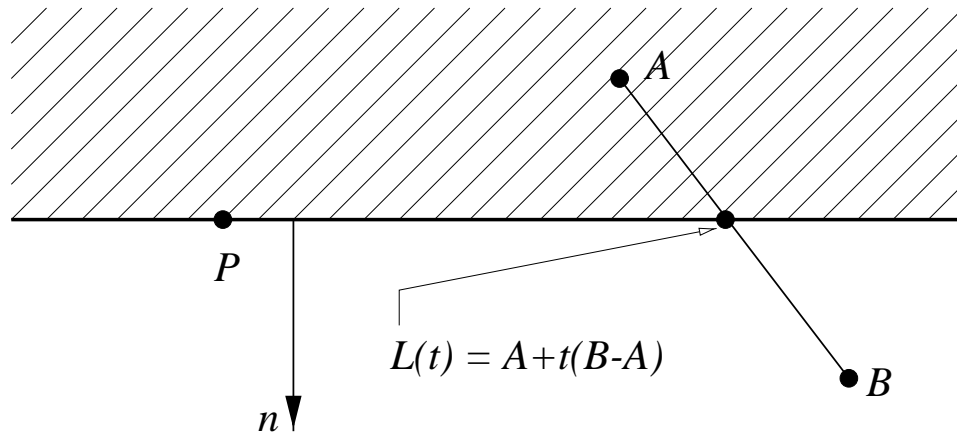
**Do the hard stuff only if we have to:**

If line segment partially in/partially out, 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:



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

- 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 $wec = \textit{window-edge coordinates}$ ):

```

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.

(Readings: McConnell: 5.1. Watt: none. Hearn and Baker: Sections 6-5 through 6-7 (12-5 for 3D clipping). Red book: 3.9. White Book: 3.11. Blinn: 13. )

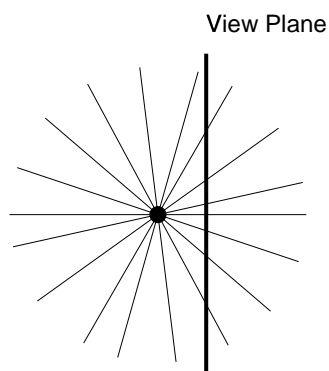
## 9 Projections and Projective Transformations

### 9.1 Projections

#### 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).

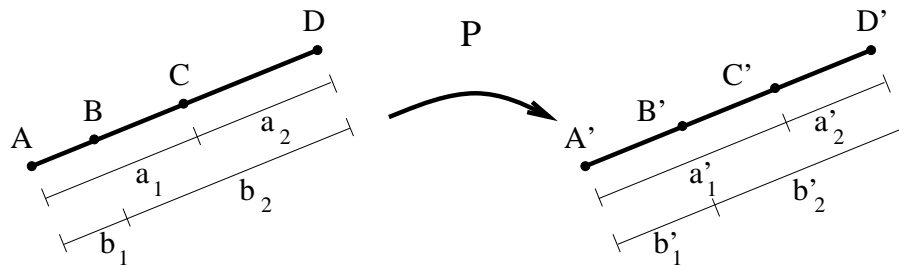
**Comparison:****Affine Transformations**

Image of 2 points on a line  
determine image of line

Image of 3 points on a plane  
determine image of plane

In dimension  $n$  space,  
image of  $n + 1$  points/vectors  
defines affine map.

Vectors map to vectors

$$\vec{v} = Q - R = R - S \Rightarrow$$

$$A(Q) - A(R) = A(R) - A(S)$$

**Projective Transformations**

Image of 3 points on a line  
determine image of line

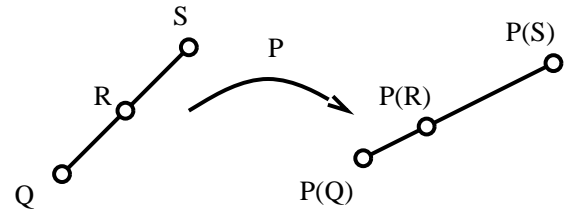
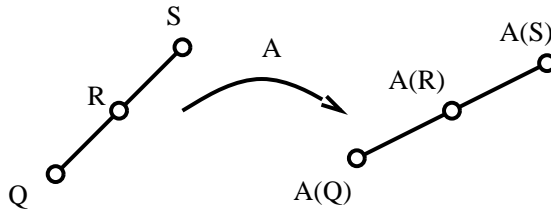
Image of 4 points on a plane  
determine image of plane

In dimension  $n$  space,  
image of  $n + 2$  points/vectors  
defines projective map.

Mapping of vector is ill-defined

$$\vec{v} = Q - R = R - S \text{ but}$$

$$P(Q) - P(R) \neq P(R) - P(S)$$

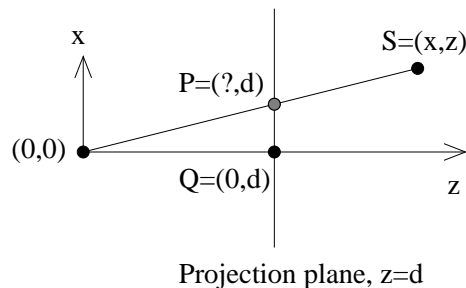


Can represent with matrix multiply  
(sort of)

Can represent with matrix multiply  
and normalization

**Perspective Map:**

- Given a point  $S$ , we want to find its projection  $P$ .



- Similar triangles:  $P = (xd/z, d)$
- In 3D,  $(x, y, z) \mapsto (xd/z, yd/z, d)$
- Have identified all points on a line through the origin with a point in the projection plane.



$$(x, y, z) \equiv (kx, ky, kz), \quad k \neq 0.$$

- 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.

### Matrix form

- Would like matrix form to operate on  $[x, y, z, 1]^t$   
 With homogeneous coordinates, last coordinate after mapping can be non-0,1  
 Will divide by last coordinate after matrix multiply
- The following matrix works (with  $d = 1$ ):

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ z \end{bmatrix} \equiv \begin{bmatrix} x/z \\ y/z \\ 1 \\ 1 \end{bmatrix}$$

but loses depth information after homogenization

- The following matrix

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

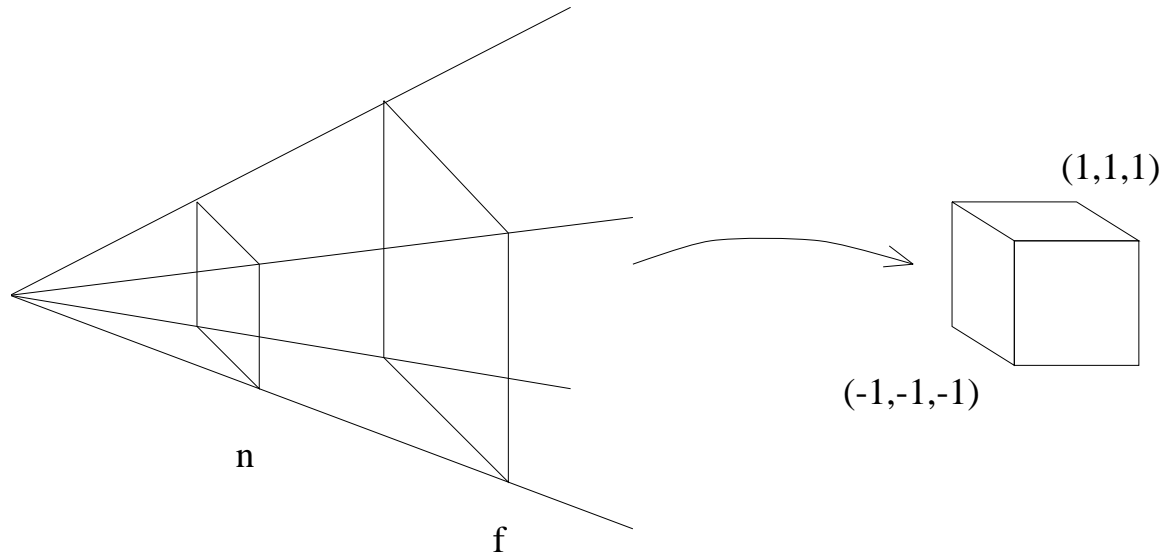
maps  $x$  and  $y$  to  $x/z$ ,  $y/z$ , and maps  $z$  to  $1 + 1/z$

I.e., depth information reversed but retained.

- This is the main idea; variations on entries in this matrix allow us to do other things (near/far clipping, NDC)

### 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”
  - Initially, 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 matrix multiply followed by **normalization**:

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

- Solve for  $a, b, c,$  and  $d$  such that  $z \in [n, f]$  maps to  $z' \in [-1, 1]$ .
- Satisfying our constraints 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 homogenizing we get

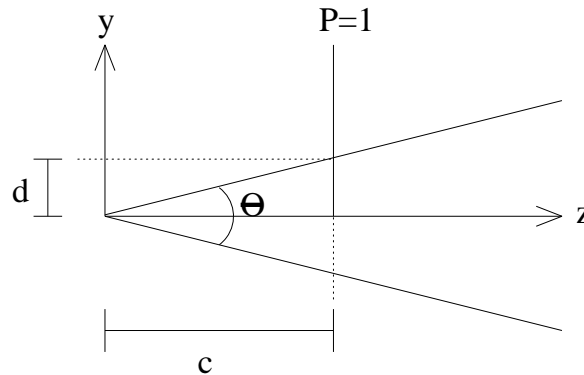
$$\left[ \frac{x}{z}, \frac{y}{z}, \frac{z(f+n)-2fn}{z(f-n)}, 1 \right]^T$$

- 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.

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 map both to  $[-1, 1]$  because we may not have square windows)

OpenGL maps  $y$  to  $[-1, 1]$ :



- Want to map distance  $d$  to 1.
- $y \mapsto y/z$  is the current projection ...

Our final matrix is

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

where the  $\pm$  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.

Projection matrix can be expressed as a shear or rotation in 4D

## 9.2 Why Map Z?

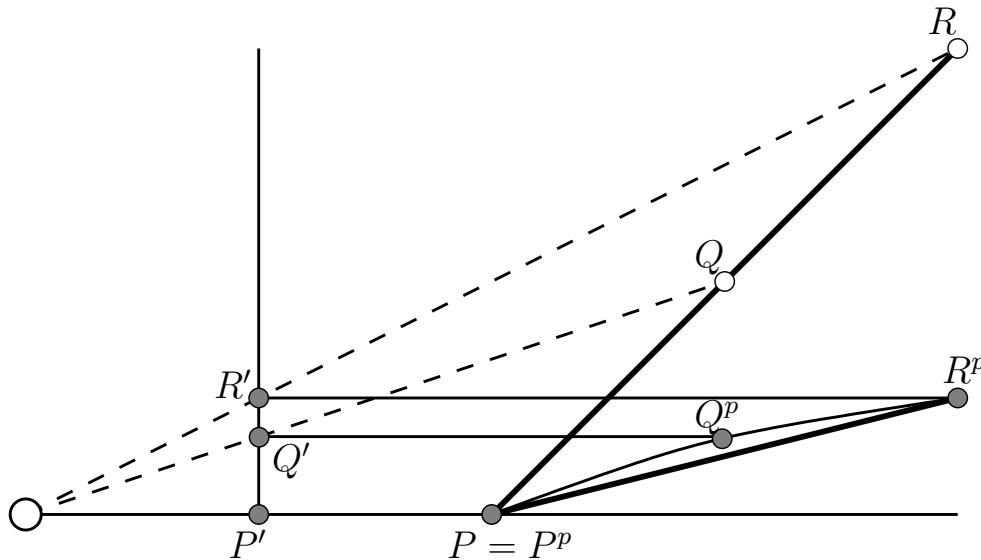
### Why Map Z?

- 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

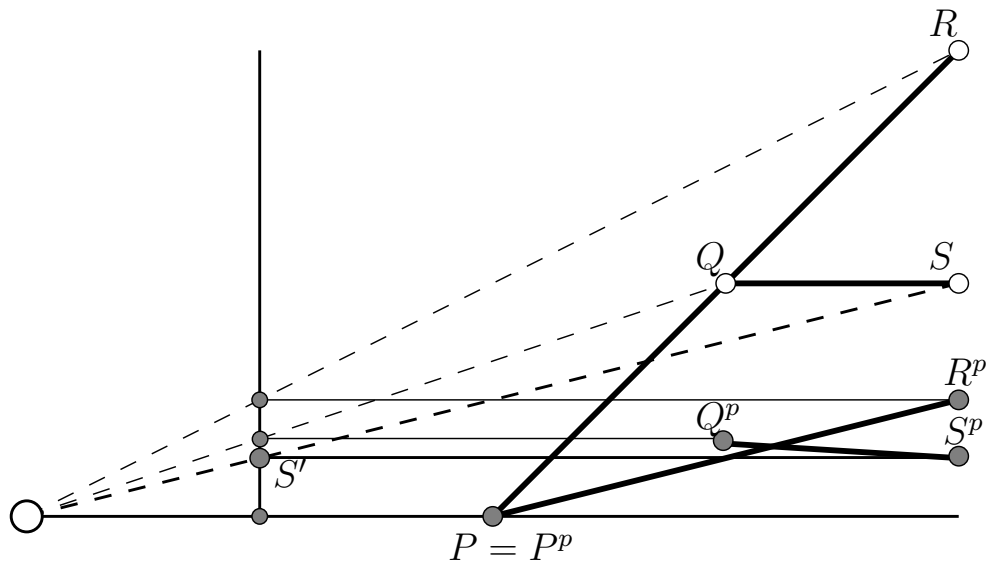
$$(x, y, z, 1) \mapsto \left( \frac{xn}{z}, \frac{yn}{z}, z, 1 \right)$$

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



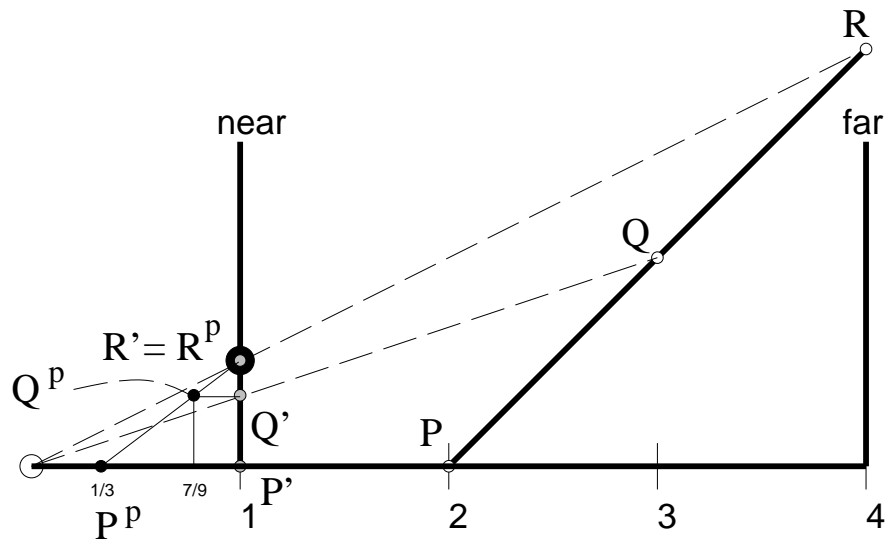
- In this figure,  $P, Q, R$  map to  $P', Q', R'$  under a pure projective transformation.
- With the mapping  $(x, y, z, 1) \mapsto (\frac{xn}{z}, \frac{yn}{z}, z, 1)$   $P, Q, R$  actually map to  $P^p, Q^p, R^p$ , which fail to lie on a straight line.



- Now map  $S$  to  $S'$  to  $S^p$ .
- Line segments  $P^p R^p$  and  $Q^p S^p$  cross!
- The map

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

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



### 9.3 Mapping Z

#### Mapping Z

- It's clear how  $x$  and  $y$  map. How about  $z$ ?
- The  $z$  map affects: clipping, numerics

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

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

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

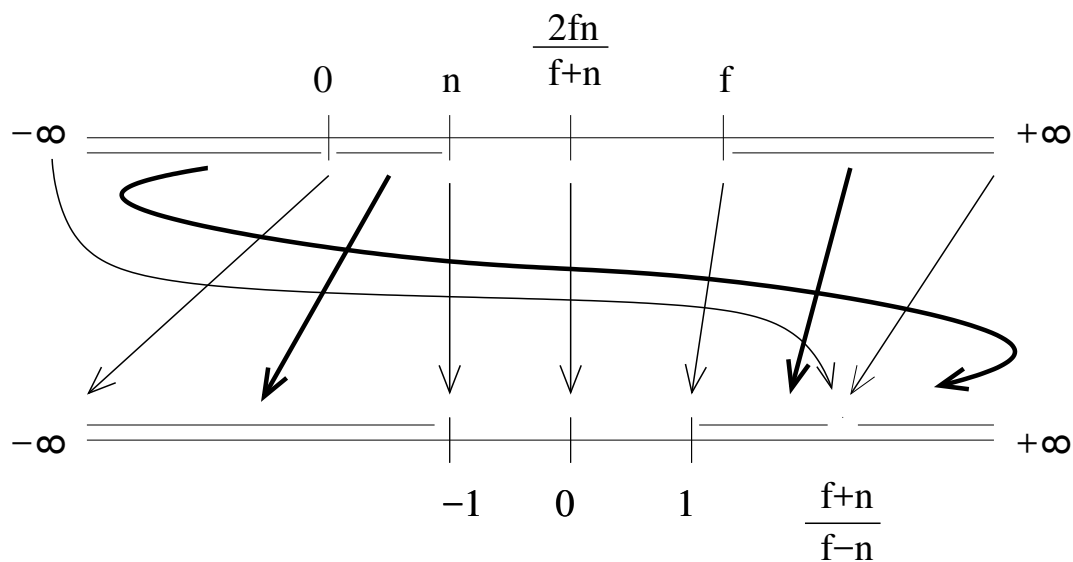
Note that  $f^2 + fn > 2fn > fn + n^2$  so

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

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

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

Pictorially, we have



- What happens if we vary  $f$  and  $n$ ?

—

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

—

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

—

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

- 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.

## 9.4 3D Clipping

### 3D Clipping

- When do we clip in 3D?

We should clip to the near plane *before* we project. Otherwise, we might attempt to project a point with  $z = 0$  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 homogeneous coordinates.

(Readings: Red Book, 6.6.4; White book, 6.5.4. )

## 9.5 Homogeneous Clipping

### Homogeneous Clipping

#### Projection: transform and homogenize

- Linear transformation

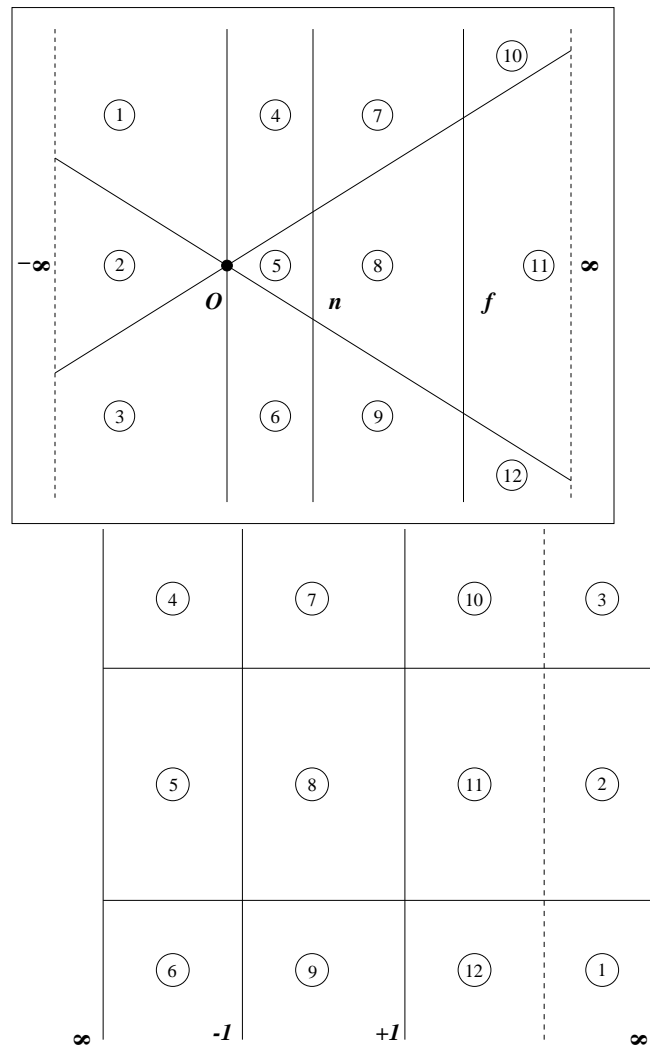
$$\begin{bmatrix} nr & 0 & 0 & 0 \\ 0 & ns & 0 & 0 \\ 0 & 0 & \frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \bar{x} \\ \bar{y} \\ \bar{z} \\ \bar{w} \end{bmatrix}$$

- Homogenization

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

#### Region mapping:



**Clipping not good after normalization:**

- Ambiguity after homogenization

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

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

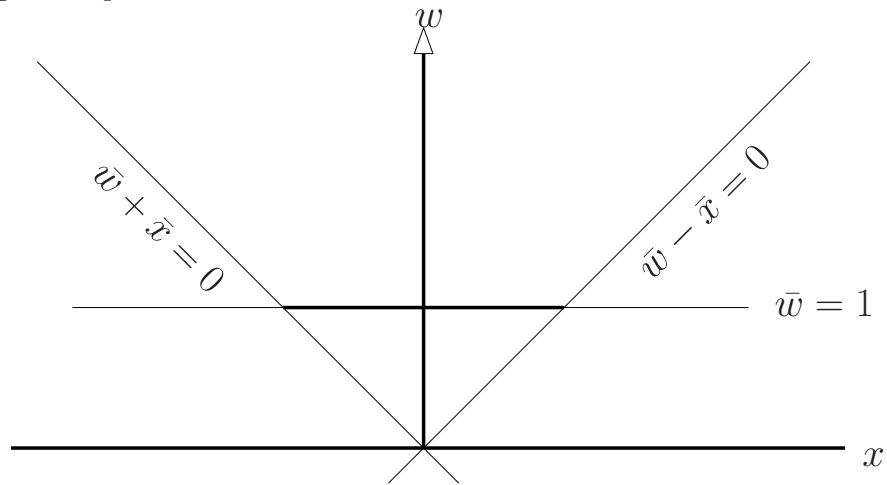
**Clip in homogeneous coordinates:**

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

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

**Clipping Homogeneous Coordinates**

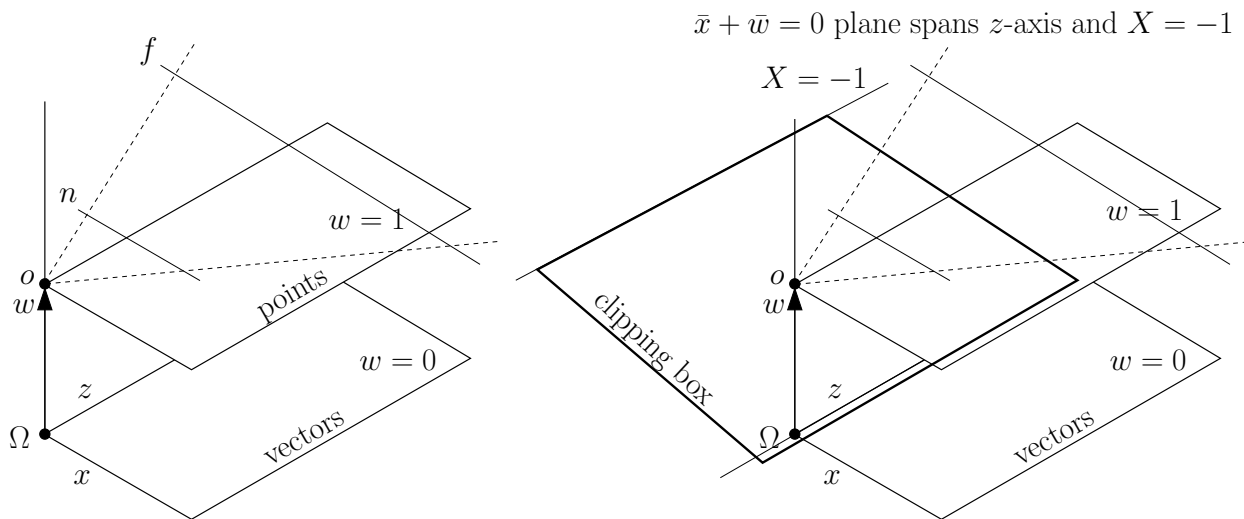
- Assume NDC window of  $[-1, 1] \times [-1, 1]$
- To clip to  $X = -1$  (left):
  - Projected coordinates: Clip to  $X = -1$
  - Homogeneous coordinate: Clip to  $\bar{x}/\bar{w} = -1$
  - Homogeneous plane:  $\bar{w} + \bar{x} = 0$



- Point is visible if  $\bar{w} + \bar{x} > 0$

## Geometry of Homogeneous Clipping

Drop one dimension...



- 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

## 9.6 Pinhole Camera vs. Camera vs. Perception

### Pinhole Camera vs. Camera

- “Lines map to lines – can’t be true because I’ve seen pictures that curve lines”
- Camera is not a pinhole camera
- Wide angle lens suffers from *barrel distortion*

### Convergence

Where do railroad tracks meet?



Where should we place the horizon on the screen?

### Horizon at Eyelevel

With a painting, free to place horizon and hang painting to match

van Ruysdeal painting

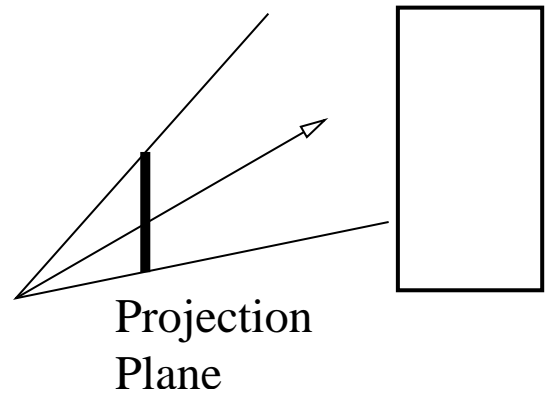
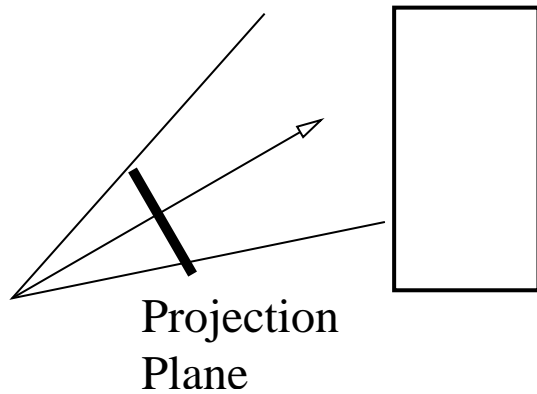
In graphics, don’t know where eye level will be relative to screen.

### Vertical Convergence

- Is vertical convergence a pinhole camera effect or another kind of distortion?

# Slanted apartments

- Vertical convergence is a perspective projection effect.  
Occurs when you tilt the camera/view "up"
- You can get rid of it if view plane parallel to up axis



- Pinhole camera (ray tracer) - Mathematically correct, but looks wrong  
Spheres, cylinders: same size, on a plane parallel to view plane:

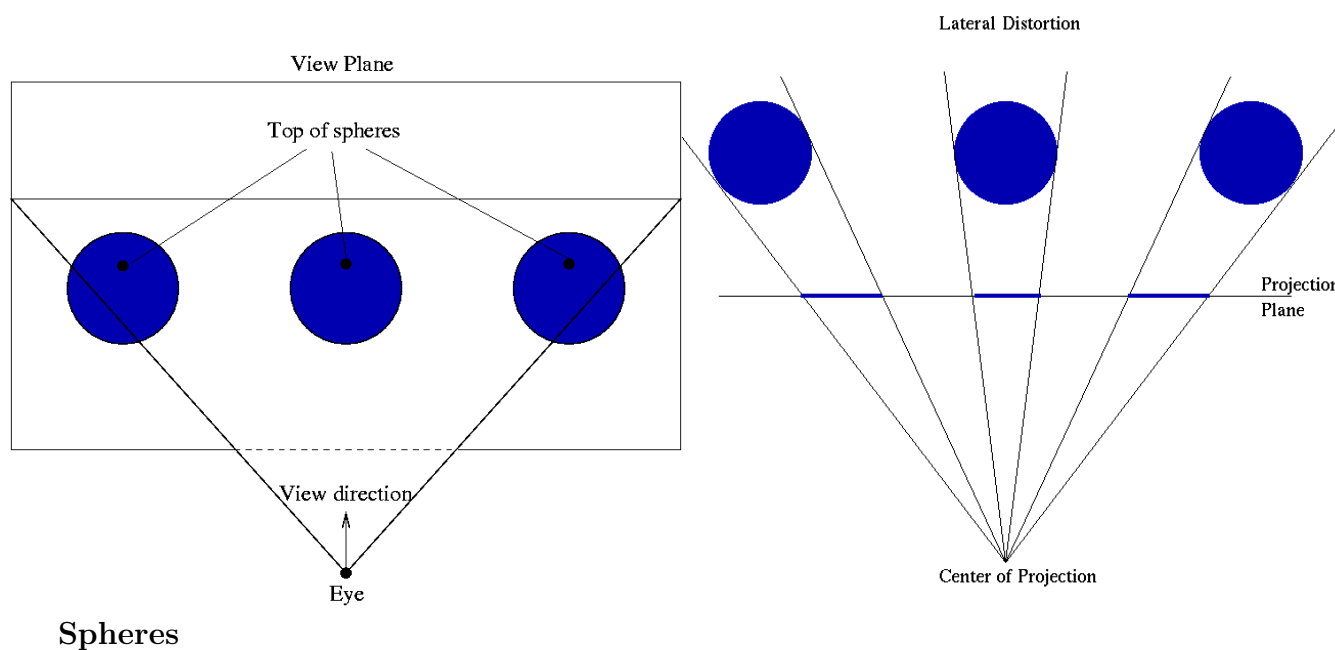
## Spheres on cylinders

### Art

- Painting Sculpture - Mathematically incorrect, but "looks right"

## Skew-eyed David

What's going on?

**Distortion****Spheres**

# Spheres, and lots of them!

Occurs in Computer Graphics Images

## Ben-hur races in front of distorted columns

In Real Life

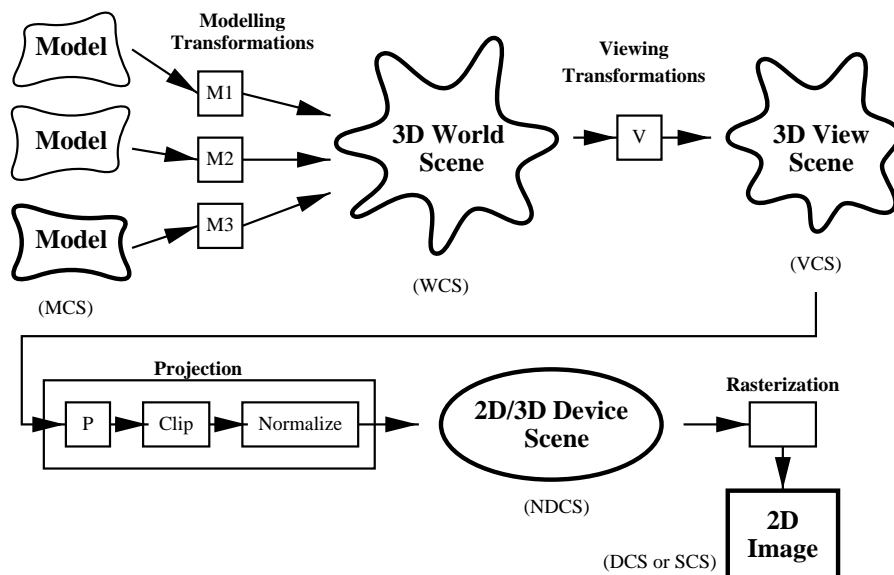
- Eye/attention only on small part of field of view  
Sphere looks circular
- Rest of field of view is “there”  
Peripheral spheres not circular, but not focus of attention and you don’t notice
- When you look at different object, you shift projection plane  
Different sphere looks circular
- In painting, all spheres drawn as circular  
When not looking at them, they are mathematically wrong but since not focus of attention they are “close enough”
- In graphics...



## 10 Transformation Applications and Extensions

### 10.1 Rendering Pipeline Revisited

#### Rendering Pipeline Revisited



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

### 10.2 A2 World and View Frames

#### Viewing-World-Modelling Transformations :

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

$$VM$$

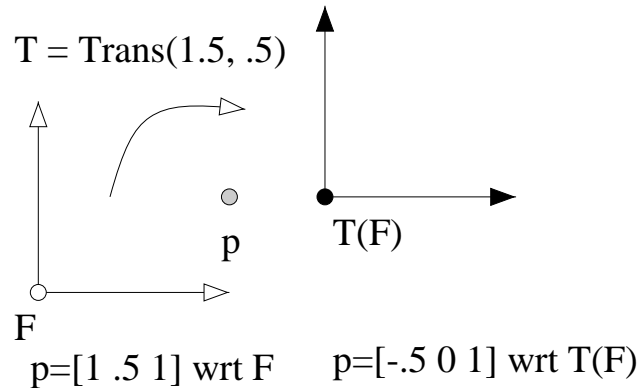
transforms from modelling coordinates to viewing coordinates.

**Note:**  $M$  is performing both modelling 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 modelling frame) how do we adjust  $M$ ?

#### Viewing Transformations:

- Assume all frames are orthonormal
- To transform View Frame by  $T$ , apply  $T^{-1}$  to 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  $T^{-1}$ .
- Old CoB  $V \Rightarrow$  New CoB  $T^{-1}V$

### Modelling Transformations:

- Columns of  $M$  are Model Frame elements expressed relative to World Frame.
- Want to perform modelling transformation relative to modelling coordinates.
- If we have previously transformed Model Frame, then we next transform relative to transformed Model Frame.
- Example: If

$$M = \begin{bmatrix} x_1 & x_2 & x_3 & x_4 \\ y_1 & y_2 & y_3 & y_4 \\ z_1 & z_2 & z_3 & z_4 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

and translate one unit relative to the first Model Frame basis vector, then want to translate by  $(x_1, y_1, z_1, 0)$  relative to the World Frame.

- Could write this as

$$M' = \begin{bmatrix} 1 & 0 & 0 & x_1 \\ 0 & 1 & 0 & y_1 \\ 0 & 0 & 1 & z_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot M = \begin{bmatrix} x_1 & x_2 & x_3 & x_4 + x_1 \\ y_1 & y_2 & y_3 & y_4 + y_1 \\ z_1 & z_2 & z_3 & z_4 + z_1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- But this is also equal to

$$M' = M \cdot \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} x_1 & x_2 & x_3 & x_4 + x_1 \\ y_1 & y_2 & y_3 & y_4 + y_1 \\ z_1 & z_2 & z_3 & z_4 + z_1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



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

$$MT$$

yields that transformation.

- Summary:

Modelling transformations embodied in matrix  $M$

World-to-View change of basis in matrix  $V$

$VM$  transforms from modelling 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 modelling frame, the new modelling transformation  $M'$  is given by

$$M' = MT$$

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

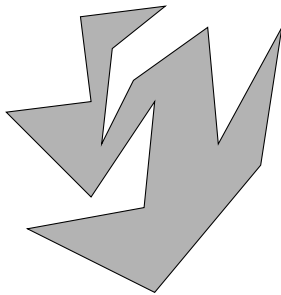


## 11 Polygons

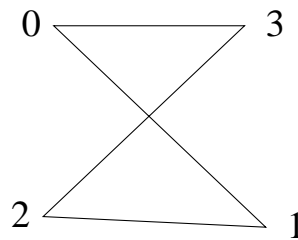
### 11.1 Polygons – Introduction

#### Polygons

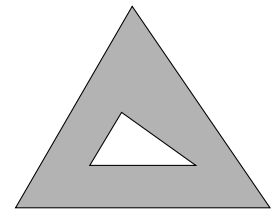
- Need an area primitive
- Simple polygon:
  - Planar set of ordered points,  $v_0, \dots, v_{n-1}$   
(sometimes we repeat  $v_0$  at end of list)
  - No holes
  - No line crossing



OK

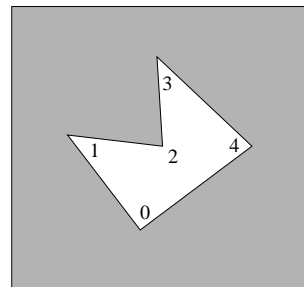
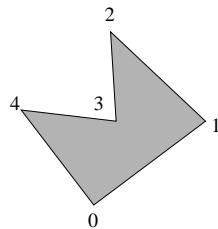


Line Crossing

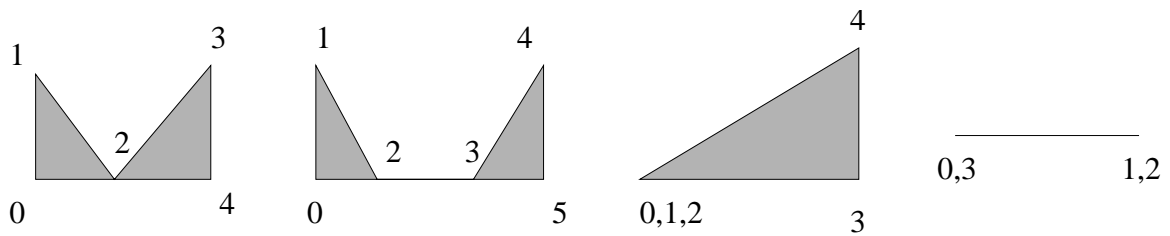


Hole

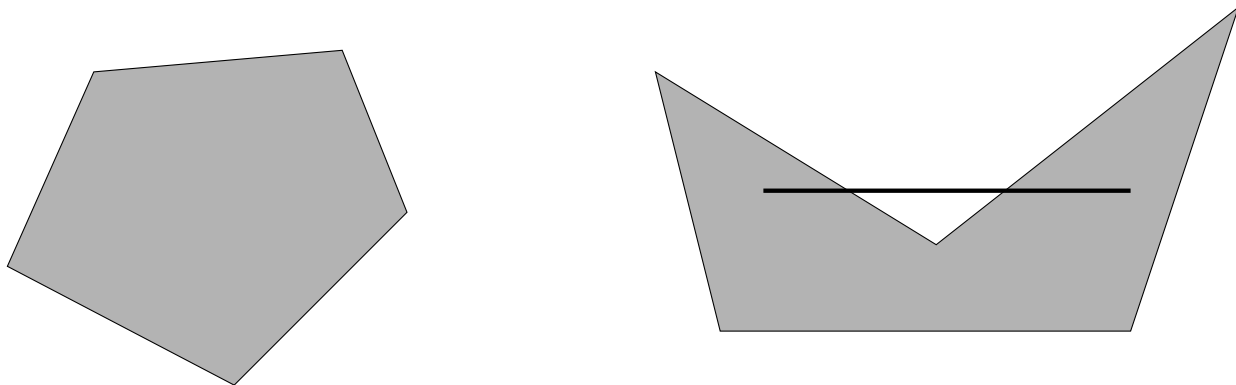
- Normally define an interior and exterior  
Points ordered in counter-clockwise order  
To the “left” as we traverse is inside



- Try to avoid degeneracies, but sometimes unavoidable



- *Convex* and *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 in many operations

- Affine transformations may introduce degeneracies

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

## 11.2 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  $v_1, \dots, v_n$
- Each polygon edge is a pair  $[v_i, v_{i+1}]$   $i = 1, \dots, n$ 
  - 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
  - $v_1, \dots, v_n \rightarrow w_1, \dots, w_m$
- Repeat on resulting polygon with next sequential window edge

**Contrast with Line Clipping:**

- **Line Clipping:**
  - 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

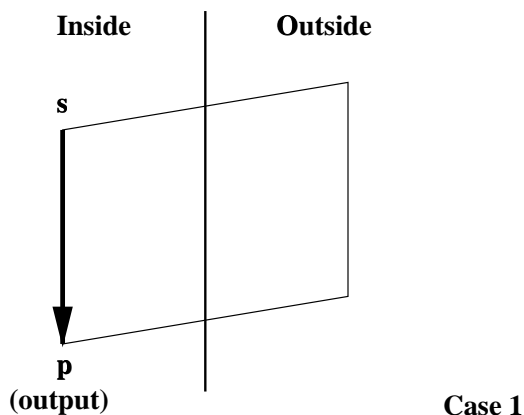
**Notation:**

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

There are four cases to consider

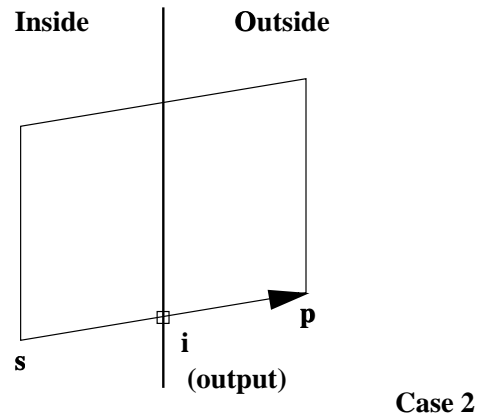
**Case 1:** Polygon edge is entirely inside the window edge

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



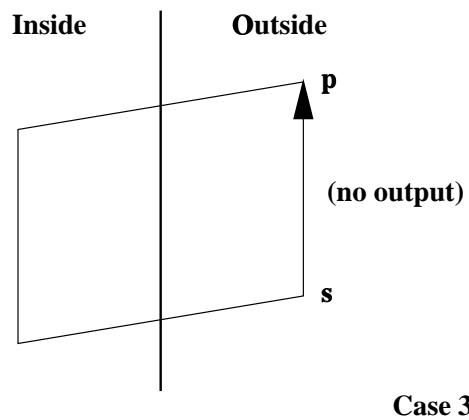
**Case 2:** Polygon edge crosses window edge going out

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



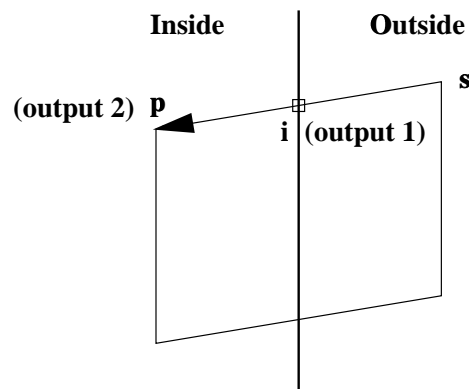
**Case 3:** Polygon edge is entirely outside the window edge

- No output



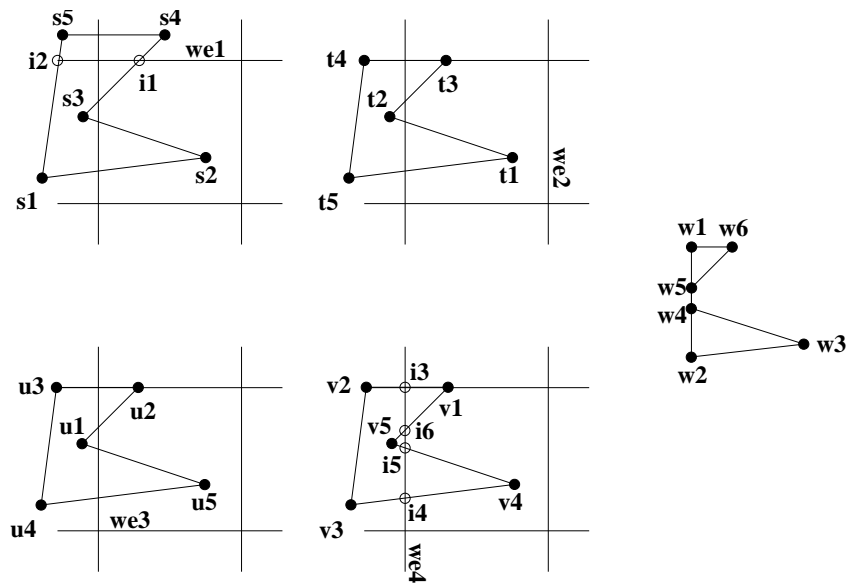
**Case 4:** Polygon edge crosses window edge going in

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



Case 4

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



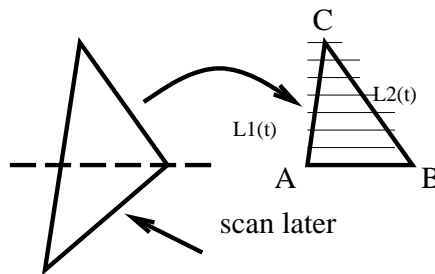
(Readings: Watt: 6.1. Hearn and Baker: Section 6-8. Red book: 3.11. White book: 3.14. )

### 11.3 Polygon Scan Conversion

#### Scan Conversion

- Once mapped to device coordinates, want to scan convert polygon.
- 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  $\overline{AC}$  and from  $\overline{BC}$  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

```

- This is a floating point algorithm

(Readings: McConnell: 3.3. 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). )

#### 11.4 Dos and Don'ts

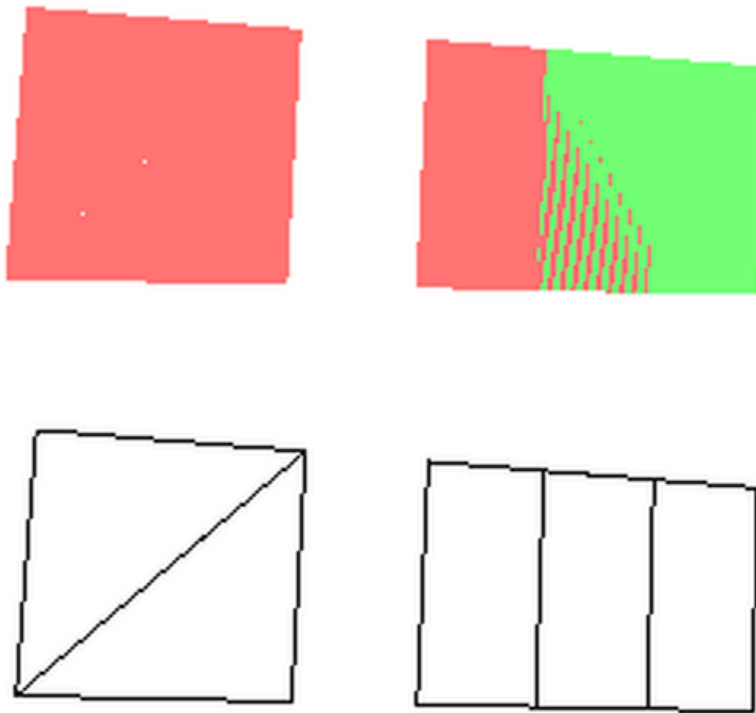
When modelling with polygonal objects, remember the following guidelines:

- Model Solid Objects — No façades  
It's easier to write other software if we can assume polygon faces are oriented
- No T-vertices  
These cause shading anomalies and pixel dropout



- No overlapping co-planar faces  
Violates solid object constraint.  
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

### **Bad Polygon Examples**

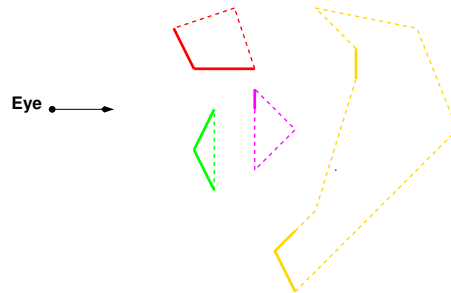




## 12 Hidden Surface Removal

### 12.1 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

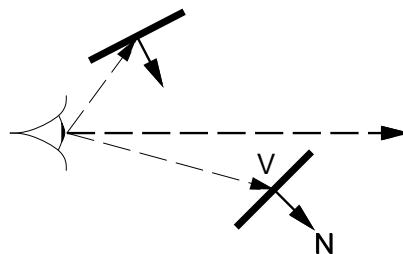
(Readings: McConnell: Chapter 5. Watt: 6.6. Red book: 13. White book: 15. Hearn and Baker: Chapter 13. )

### 12.2 Backface Culling

#### Backface Culling

#### Backface Culling

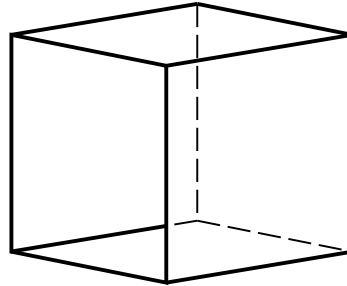
- A simple way to perform hidden surface is to remove all “backfacing” polygons.
- If polygon normal facing away from the viewer then it is “backfacing.”
- For solid objects, this means the polygon will not be seen by the viewer.



- 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.

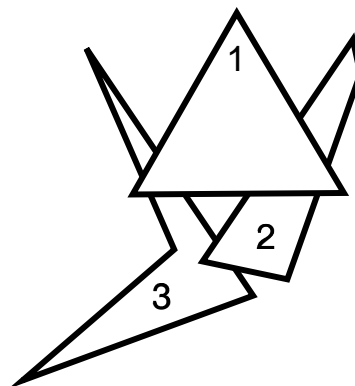


- A HUGE speed advantage if we can use it since the test is cheap and we expect at least half the polygons will be discarded.
- Usually performed in conjunction with a more complete hidden surface algorithm.
- Easy to integrate into hardware (and usually improves performance by a factor of 2).

## 12.3 Painter's Algorithm

### Painter's Algorithm

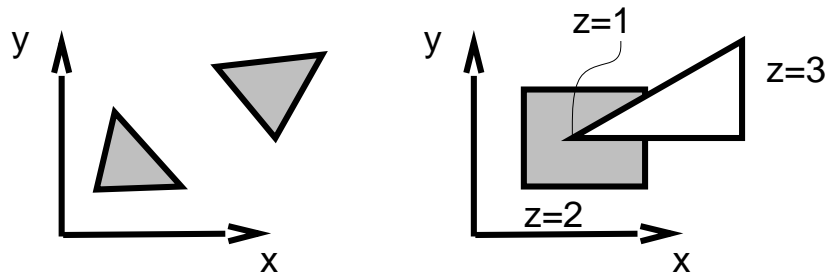
- Idea: Draw polygons as an oil painter might: The farthest one first.
  - Sort polygons on farthest  $z$
  - Resolve ambiguities where  $z$ 's overlap
  - Scan convert from largest  $z$  to smallest  $z$



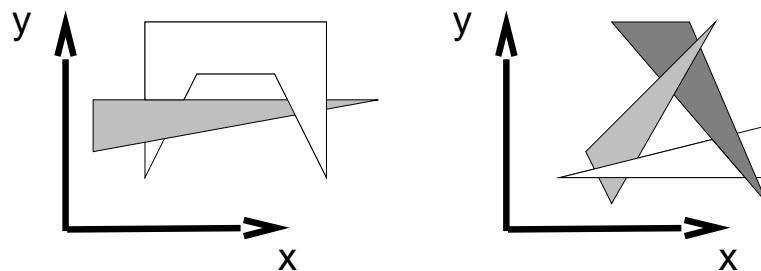
- Since closest drawn last, it will be on top (and therefore it will be seen).
- Need all polygons at once in order to sort.

### 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

## 12.4 Warnock's Algorithm

### Warnock's Algorithm

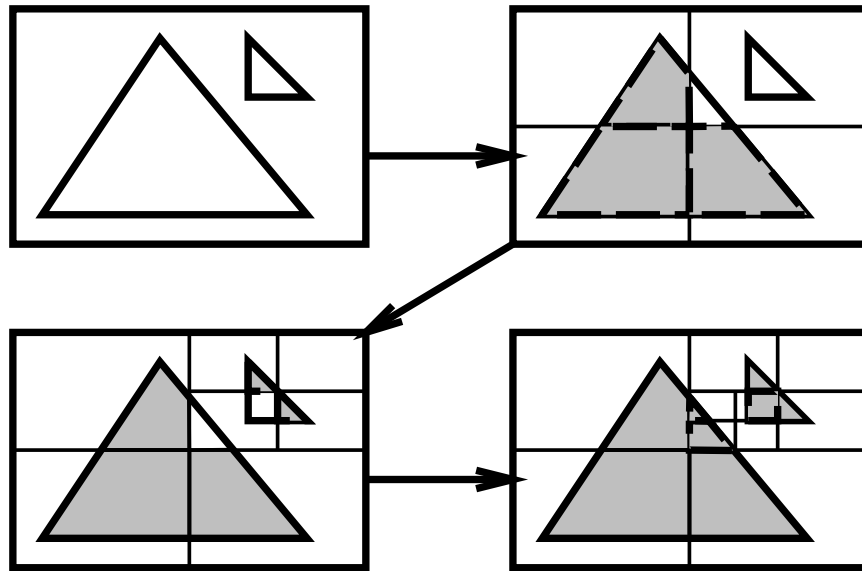
- A divide and conquer algorithm

```

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

## 12.5 Z-Buffer Algorithm

### 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  $\infty$   
Depth of far clipping plane (usually 1) will also suffice

Step in z, both in the while loop and in the for loop.

```

y = A.y;
d0 = (C.x-A.x)/(C.y-A.y); d0z = (C.Z-A.Z)/(C.y-A.y);
d1 = (C.x-B.x)/(C.y-B.y); d1z = (C.Z-B.Z)/(C.y-B.y);
x0 = A.x;  z0 = A.z;
x1 = B.x;  z1 = B.z;
while ( y <= C.y ) do
  z = z0;  d2z = (z1-z0)/(x1-x0);
  for ( x = x0 to x1 ) do
    WritePixel(x,y,z);  z += d2z;
  end
  x0 += d0;  z0 += d0z;
  x1 += d1;  z1 += d1z;
  y++;
end

```

- Scan convert using the following WritePixel:

```

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

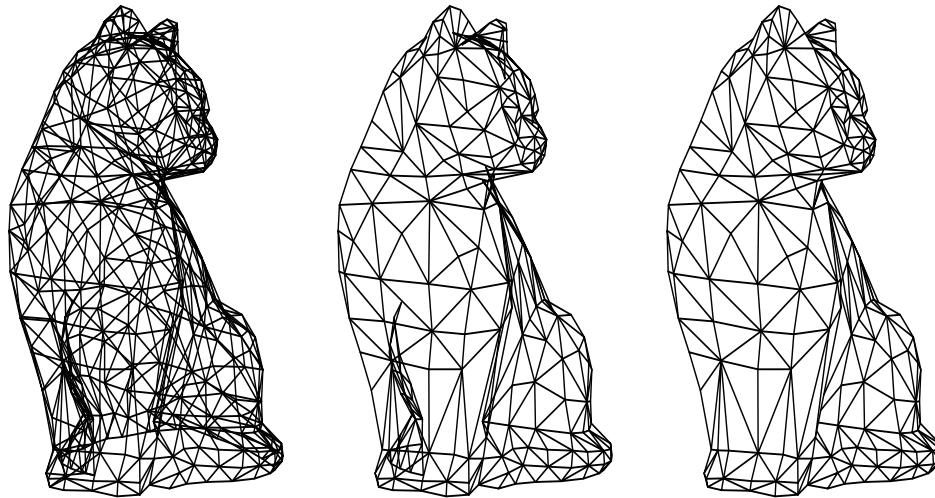
## 12.6 Comparison of Algorithms

### 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:





## 13 Hierarchical Models and Transformations

### 13.1 Hierarchical Transformations

#### Hierarchical Transformations

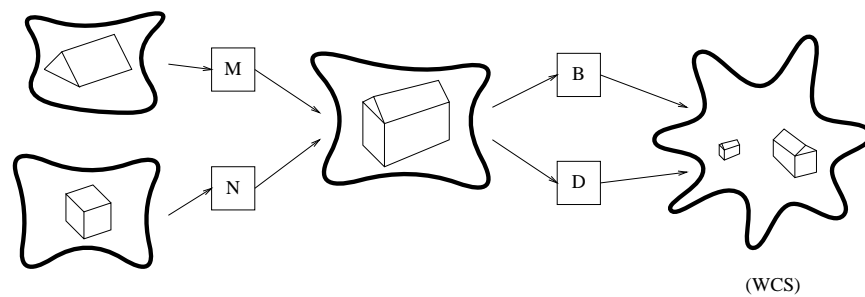
How do we model complex objects and scenes?

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

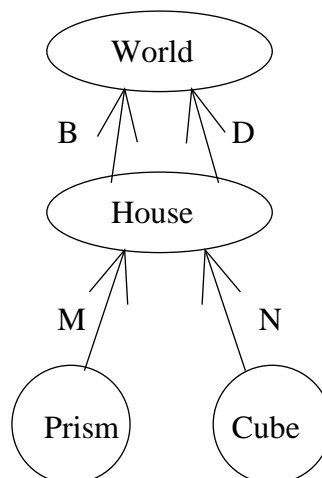
Start with basic set of 3D primitives: Cubes, spheres, prisms...

- Each is defined in a “nice” way in its own space.
- To put primitives together, use transformations.
- Use hierarchy of spaces to build complex models

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

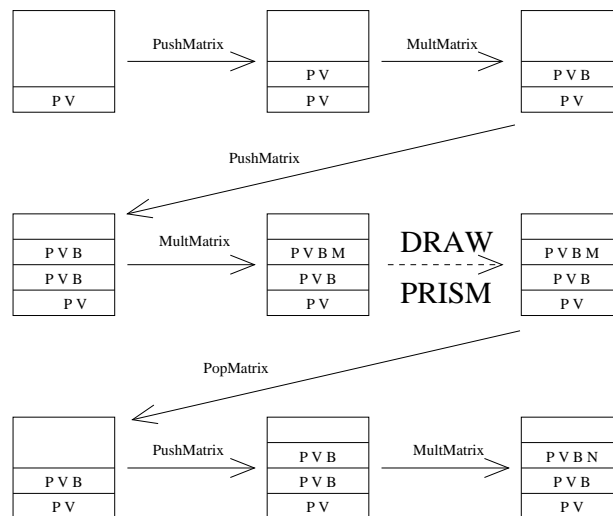
Procedure Prism(F)
    . . .
end

```

### 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.
- Use `std::stack` for a stack
 

Add `MultMatrix` that multiplies matrix with top element of stack and replaces top of stack with result.
- Sometimes put perspective and world-to-view matrices on stack.



• • •

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

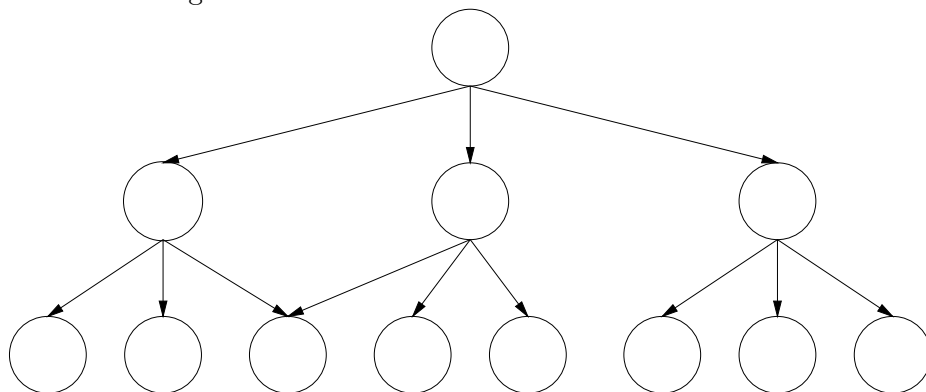
Procedure Cube()      Procedure Prism()
  . . .                . . .
end                    end

```

## 13.2 Hierarchical Data Structures

### Hierarchical Data Structures

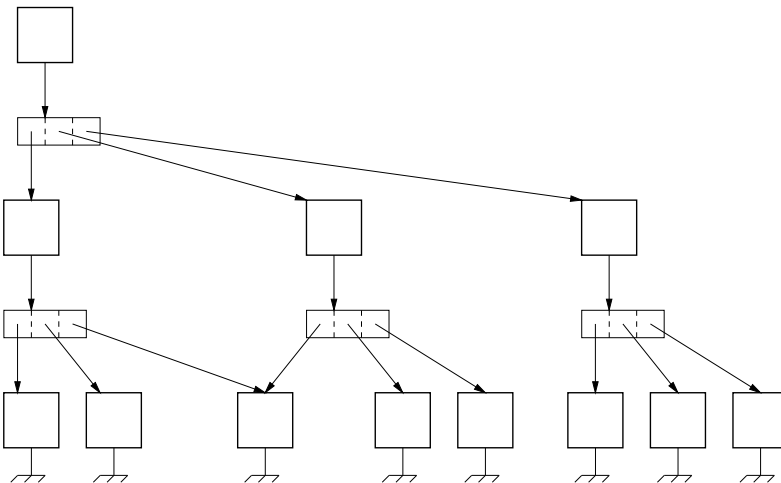
- Don't want to write a program for each scene  
Instead, develop a data structure and traverse it.
- Consider the following DAG:



- Use one of a variety of data structures to allow an arbitrary number of links from each node.

Example:

```
class Node {
    ...
private:
    Primitive *prim;
    vector<Node*> children;
};
```



- We walk through the hierarchy with a preorder traversal
- Traversal pseudocode:

```
void Node::traverse() {
    PushMatrix(); MultMatrix(transform);
    if (prim) {
        prim->draw();
    } else {
        for (vector<Node*> iterator i=children.begin();
            i != children.end(); ++i) {
            (*i)->traverse();
        }
    }
    PopMatrix();
}
```

### A3/A4 House Code, Lua

```
scene = gr.transform( 'scene' )
```

```
# Assume we have predefined materials 'red', 'green' and 'blue'
```

```
house = gr.node( 'house' )
prism = gr.polyhedron( 'prism', ... )
```

```
roof = gr.node( 'roof' )
roof:add_child( prism )
roof:translate( 0,1,0 )
house:add_child( roof )
frame = gr.cube( 'frame' )
house:add_child( frame )
```

```
farmhouse = gr.node( 'farmhouse' )
farmhouse:add_child( house )
farmhouse:set_material( green )
```

(continued on next slide)

(continuation from previous slide)

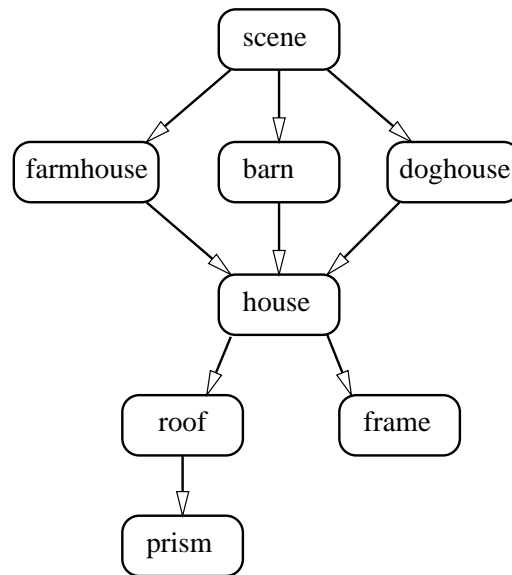
```
barn = gr.node( 'barn' )
barn:add_child( house )
barn:set_material( red )
barn:translate( 2,0,3 ) ; barn:scale( 2,2,2 ) ; barn:rotate( 'Y', 30 )
```

```
doghouse = gr.node( 'doghouse' )
doghouse:add_child( house )
doghouse:set_material( blue )
doghouse:translate(1.5, 0, -0.2 ) ; doghouse:scale( 0.2,0.2,0.2 )
    doghouse:rotate( 'Y', -15 ) )
```

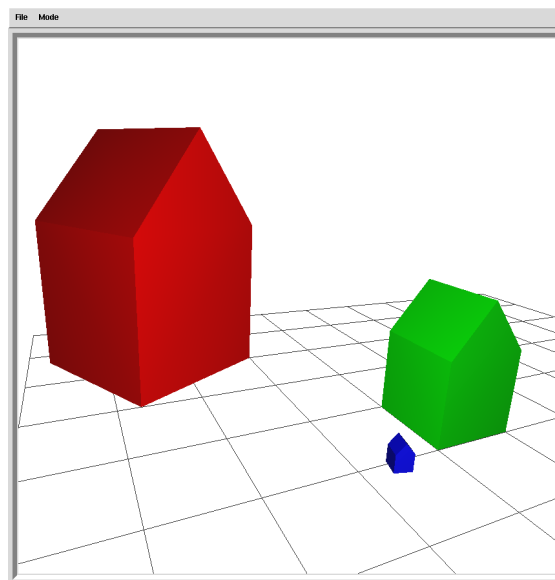
```
scene:add_child( farmhouse )
scene:add_child( barn )
scene:add_child( doghouse )
```

```
# Later in the code...
... gr.render( scene ) ...
```

- Conceptually, DAG is



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



- How to associate materials with objects?
- For assignments, just associate with primitives.
- More complex:
  - Material assigned to inner nodes
  - Who has precedence, lower or higher in hierarchy?

- Parameterized materials
  - \* Subtree with materials  
(house with white paint, green roof)
  - \* Might want instances of house to have different materials  
(cream paint, black roof)
  - \* Have to parameterize materials on creation
  - \* Have to have method for setting materials in subtree





## 14 Rotating Around Arbitrary Axes

### 14.1 Rotation Around Arbitrary Axis

#### Rotation Around Arbitrary Axis

Derivation by Composition

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

$$\vec{b} = (x, 0, z).$$

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

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

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  $\theta$ :  $R_x(\theta)$ .
10. Reverse  $z$ -axis rotation:  $R_z(\psi)$ .
11. Reverse  $y$ -axis rotation:  $R_y(\phi)$ .

The overall transformation is

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

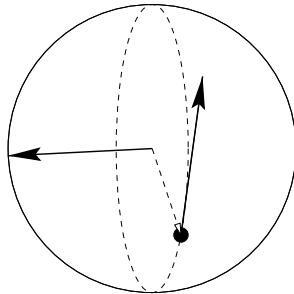
## 14.2 3D Rotation User Interfaces

### 3D Rotation User Interfaces

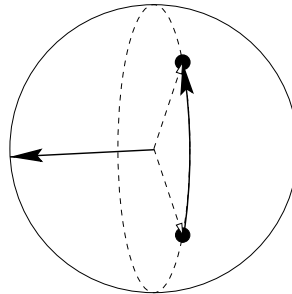
**Goal:** Want to specify angle-axis rotation “directly”.

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

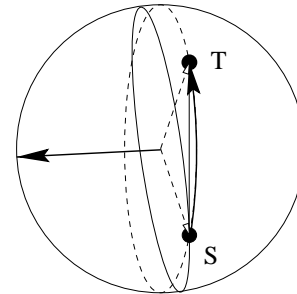
**Solutions:** Virtual Sphere, Arcball.



**Virtual Sphere**



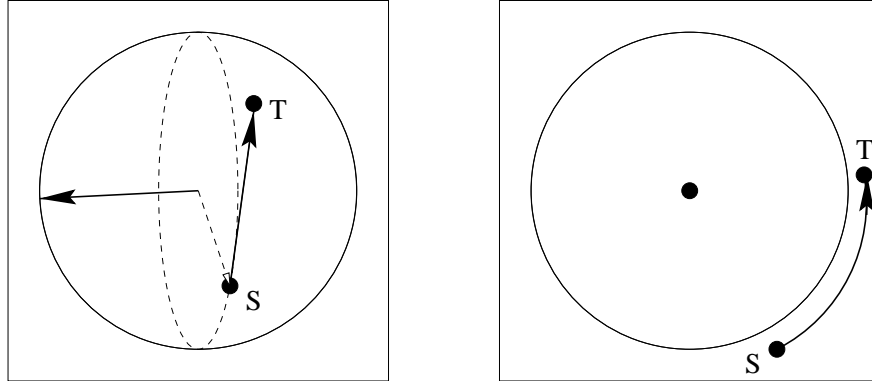
**Arcball**



**Comparison**

### 14.3 The Virtual Sphere

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



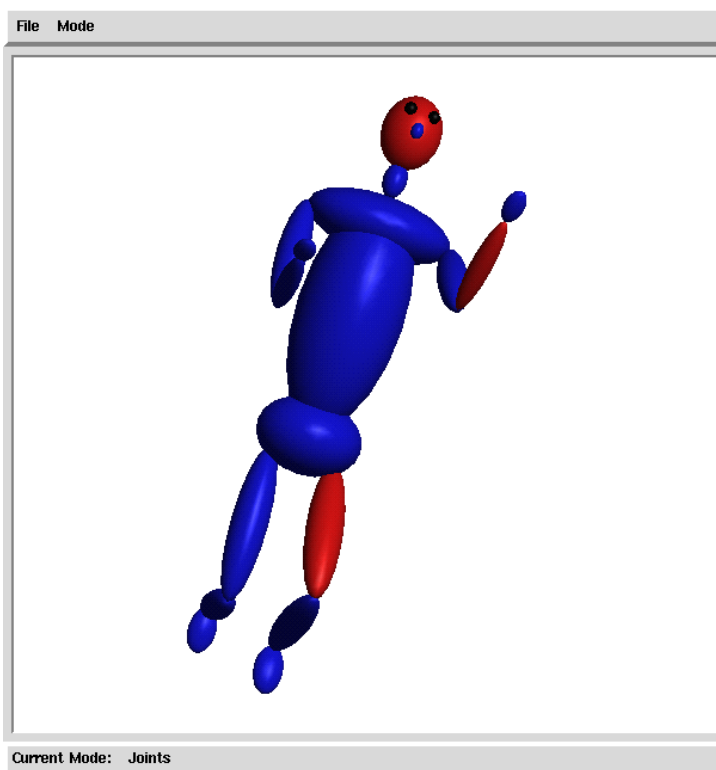
## 15 Picking and 3D Selection

### 15.1 Picking and 3D Selection

#### 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.
  - Give each object a unique colour
  - Draw with shading turned off
  - When done drawing, read back a small (1 pixel) portion of screen.
  - Look up colour to figure out which object.

Example





## 16 Colour and the Human Visual System

### 16.1 Colour

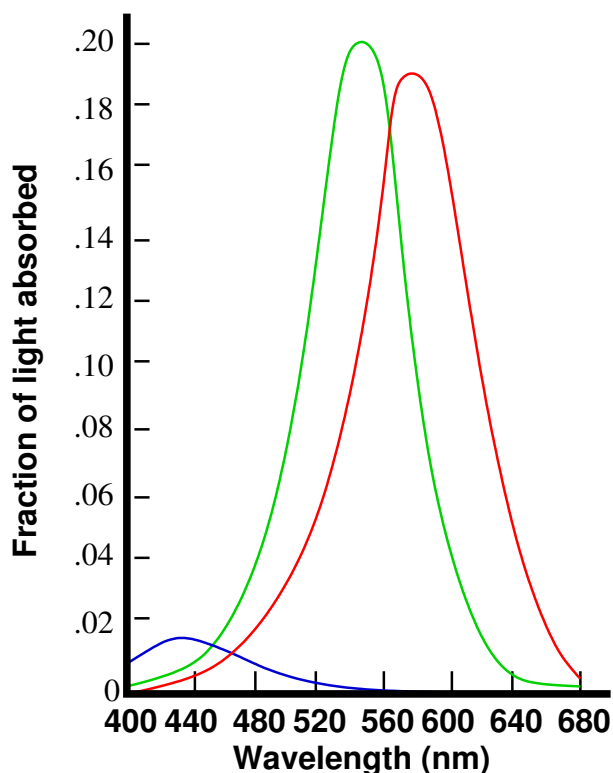
#### Introduction to Colour

- 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.
- Experiments show that there are distinct  $I$  perceived as the same colour (metamers)
- Normal human retina have three types of colour receptors which respond most strongly to short, medium, or long wavelengths.



- 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

- Mantis shrimp: 12
- Common bluebottle butterfly: 15
- 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  $\Lambda$  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

$$\begin{aligned} \ell &= \int_0^\infty L(\lambda) I(\lambda) d\lambda \\ m &= \int_0^\infty M(\lambda) I(\lambda) d\lambda \\ s &= \int_0^\infty S(\lambda) I(\lambda) d\lambda \end{aligned}$$

where  $L$ ,  $M$ , and  $S$  are weight functions.

### Colour Systems

- RGB (Red, Green, Blue) Additive

Prism to RGB/CMY wheel

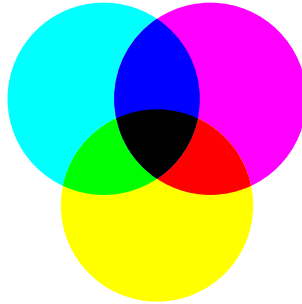


## Colour Systems

- RGB (Red, Green, Blue) Additive

### Prism to RGB/CMY wheel

- CMY (Cyan, Magenta, Yellow) Subtractive (complement of RGB)

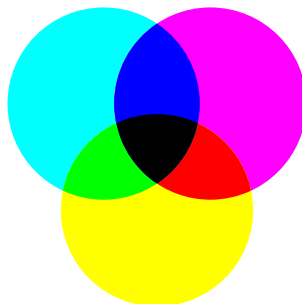


## Colour Systems

- RGB (Red, Green, Blue) Additive

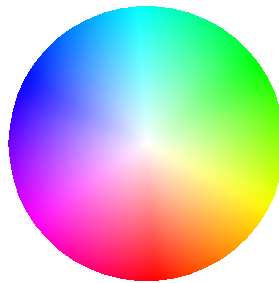
### Prism to RGB/CMY wheel

- CMY (Cyan, Magenta, Yellow) Subtractive (complement of RGB)



Often add K to get better black

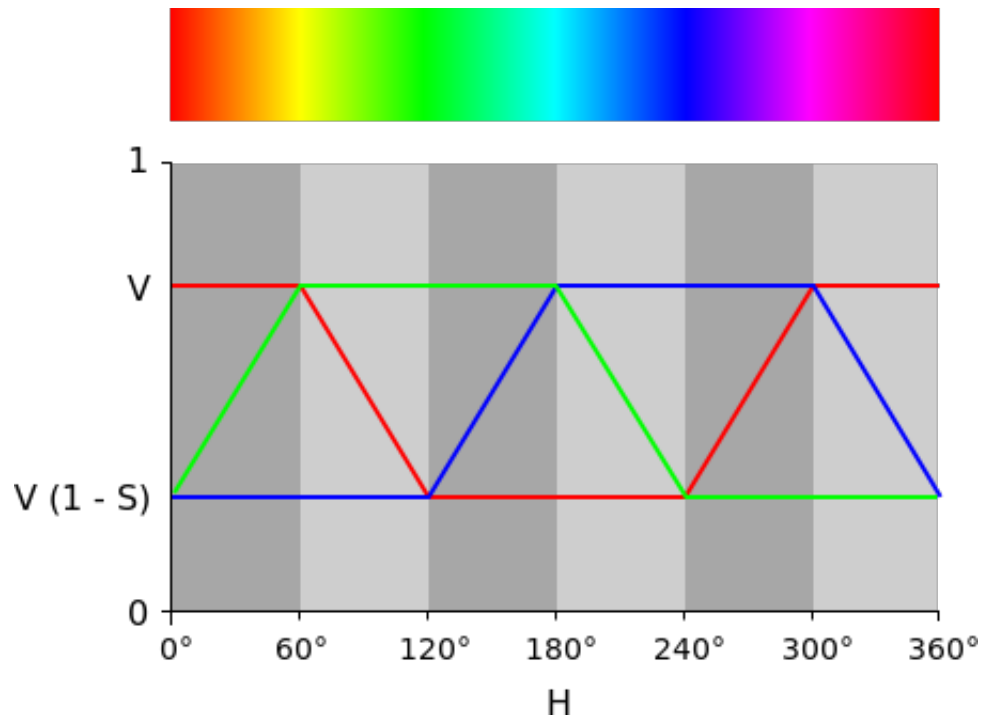
- HSV (Hue, Saturation, Value) Cone shaped colour space  
Hue-Saturation wheel:



Hue (circular), saturation (radial)

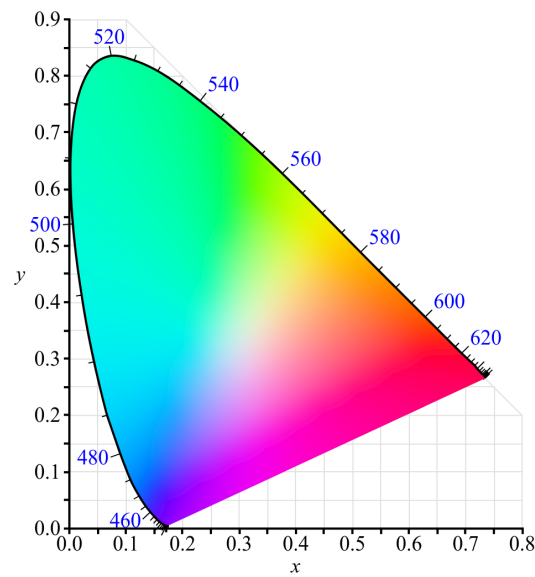
Also HSL (double cone)

- Conversion:



By en:user:Goffrie - en:Image:HSV\_RGB-Comparison.svg, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index>

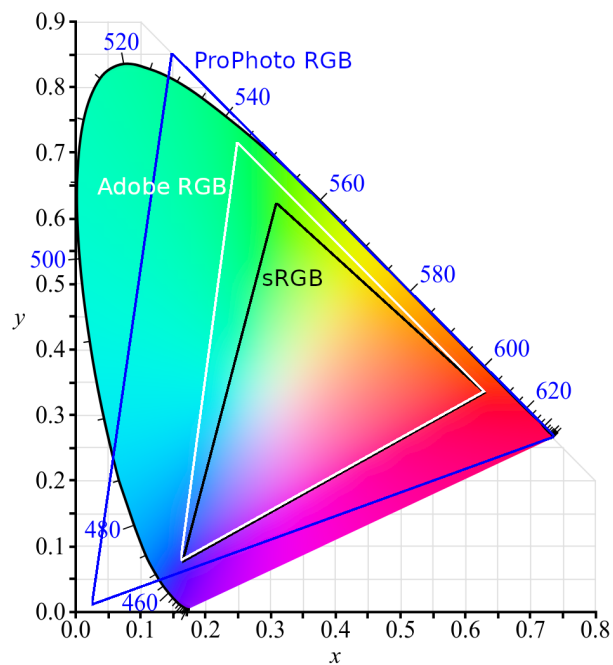
- CIE XYZ (Colour by committee) More complete colour space  
Also  $L^*u^*v$  and  $L^*a^*b$



- YIQ (Y == Luminance == CIE Y; IQ encode colour)  
Backwards compatible with black-and-white TV (only show Y)

## RGB Colour Spaces

- RGB is a subspace of CIE diagram  
Choice of where to place R,G,B



sRGB (standard RGB) used by many devices



## 17 Reflection and Light Source Models

### 17.1 Goals

#### Lighting

- What we want:
 

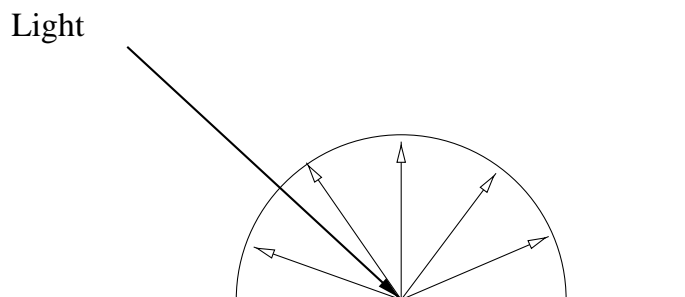
Given a point on a surface visible to the viewer through a pixel, what colour should we assign to the pixel?
- Goals:
  - Smoothly shade objects in scene
  - Want shading done quickly
  - Interactive speeds
- Begin with a simple lighting model at a single point on a surface
- Not going to worry *too* 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: McConnell: 2.3.2. Watt: 6.2. Red book: 14.1. White book: 16.1. )

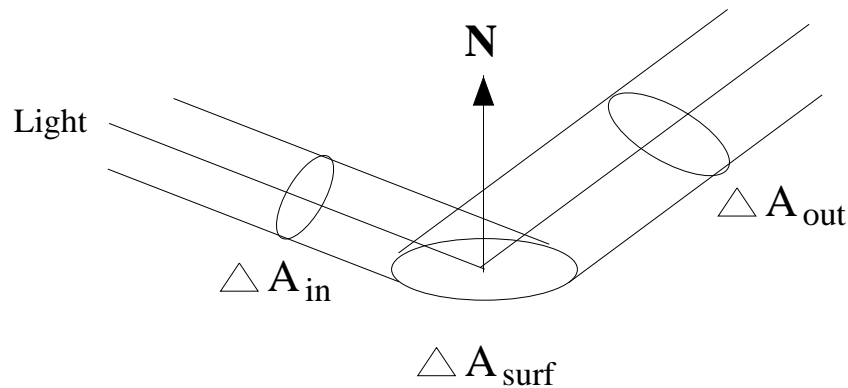
### 17.2 Lambertian Reflection

#### Lambertian Reflection

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



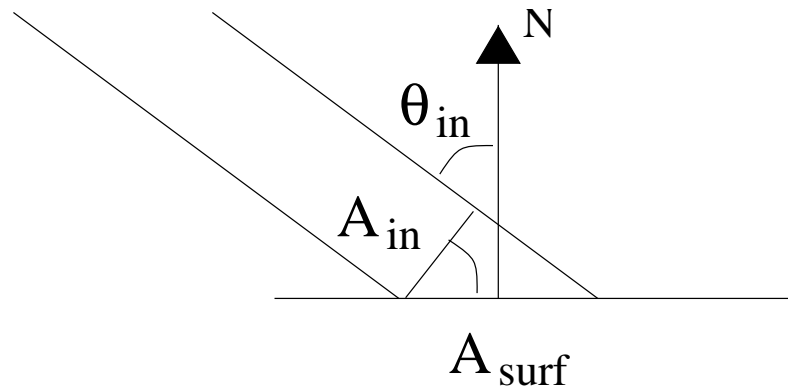
- Approximates the behavior of matte materials.
- Want an expression for  $L_{\text{out}}(\vec{v})$ , the radiance (light energy transported along a ray) reflected in some direction  $\vec{v}$ , given
  - the incoming direction  $\vec{\ell}$
  - incoming radiance  $L_{\text{in}}(\vec{\ell})$ ,
  - and the material properties (reflectance).
- Given radiance  $L_{\text{in}}(\vec{\ell})$  striking surface from direction  $\vec{\ell}$
- Want to determine  $L_{\text{out}}(\vec{v})$  reflected in direction  $\vec{v}$ .



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

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

- 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 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  $\vec{\ell}$ :

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

- Have assumed outgoing radiance is equal in all directions so  $\rho$  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  $\Omega$  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  $\pi$  is necessary to ensure conservation of energy.
- Often convert integral to a sum over light sources

## 17.3 Attenuation

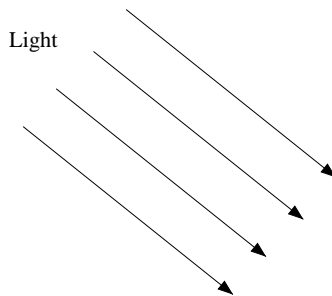
### Attenuation

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:

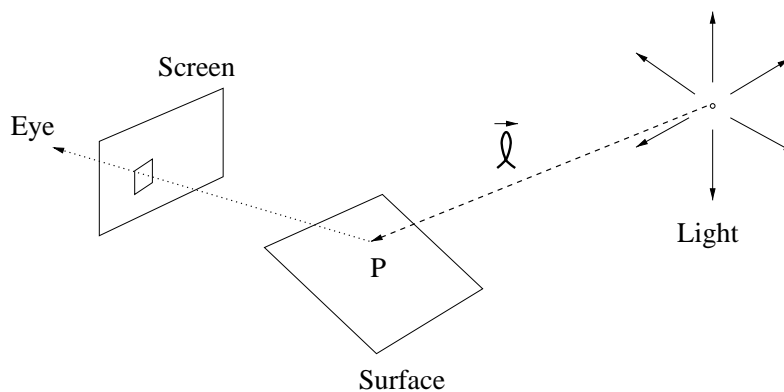


Most appropriate for distant light sources (the sun)

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.

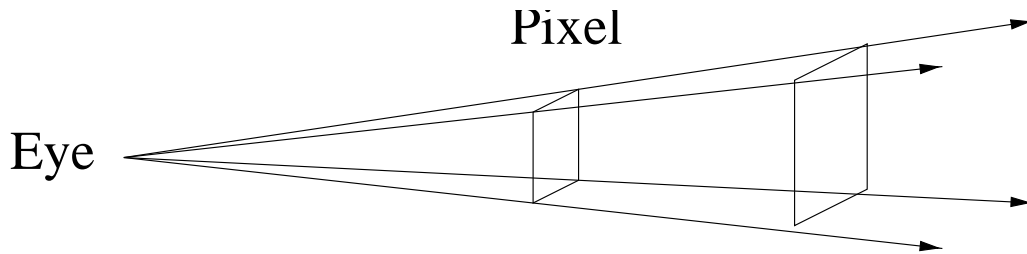
- For “empirical graphics”, usually ignore  $\pi$  factors...
- However,  $r^2$  attenuation looks too harsh.  
Harshness because real lighting is from area sources, multiple reflections in environment. True point sources impossible.

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.

## 17.4 Coloured Lights, Multiple Lights, and Ambient Light

### 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

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

## 17.5 Specular Reflection

### 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:

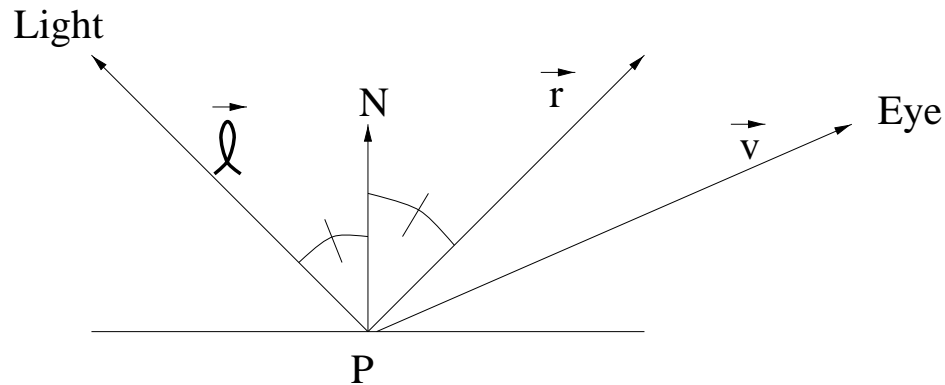
$$L_{\text{out}}(\vec{v}) = k_a I_a + k_d (\vec{\ell} \cdot \vec{n}) I_d + k_s (\vec{r} \cdot \vec{v})^p I_s$$

- Using our previous notation:

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

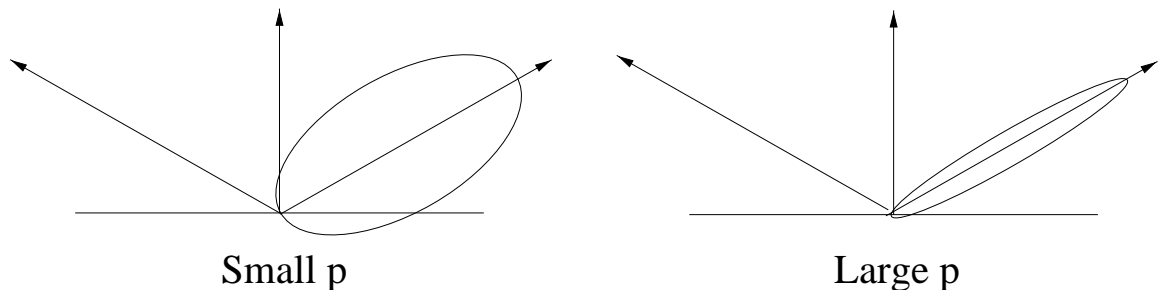
- The vector  $\vec{r}$  is  $\vec{\ell}$  reflected by the surface:

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

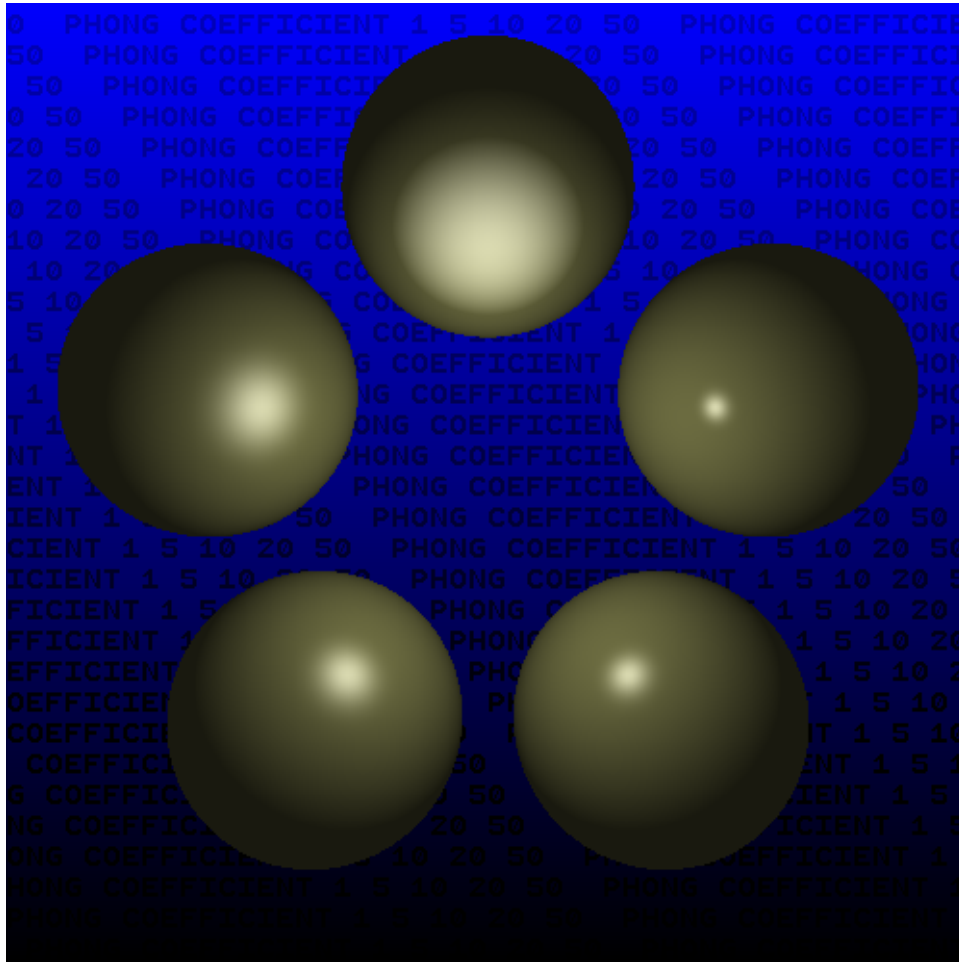


$$k_s (\vec{r} \cdot \vec{v})^p I_s$$

- 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



Caution: gives non-zero values for some directions *below* the surface.



- 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 similarly to Phong model.
- OpenGL uses Blinn-Phong model.



## 18 Shading

### 18.1 Introduction

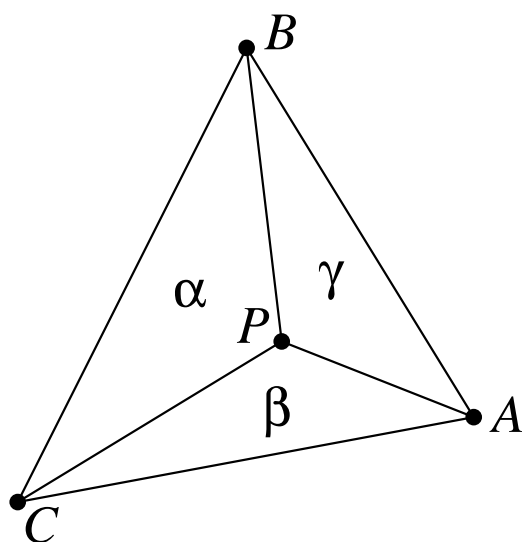
- Want to shade surfaces
- Lighting calculation for a point
  - Given:**  $L_{\text{in}}$ ,  $\vec{\ell}$ , and surface properties (including surface normal)
  - Compute:**  $L_{\text{out}}$  in direction  $\vec{v}$
- Need surface normals at every point
- Commonly, surface is polygonal
  - True polygonal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

### 18.2 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.



$$\alpha = \triangle PBC / \triangle ABC$$

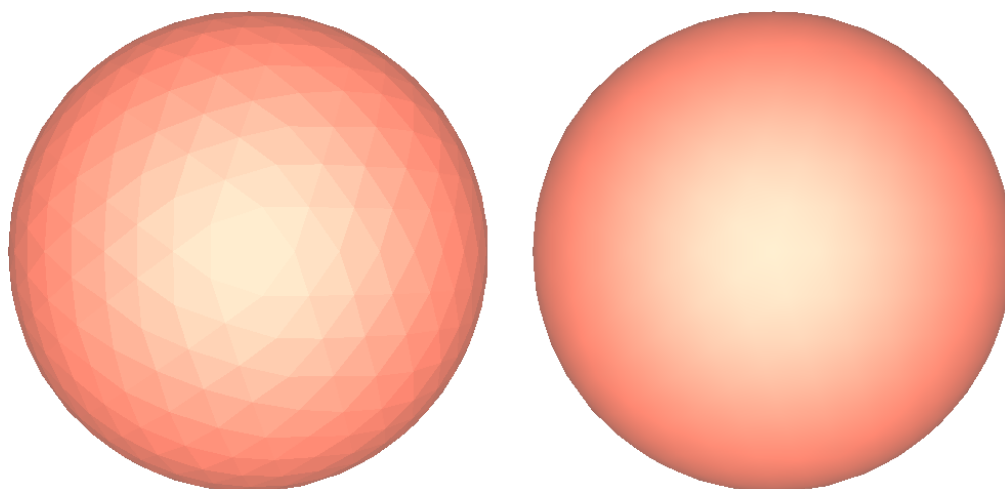
$$\beta = \triangle APC / \triangle ABC$$

$$\gamma = \triangle ABP / \triangle ABC$$

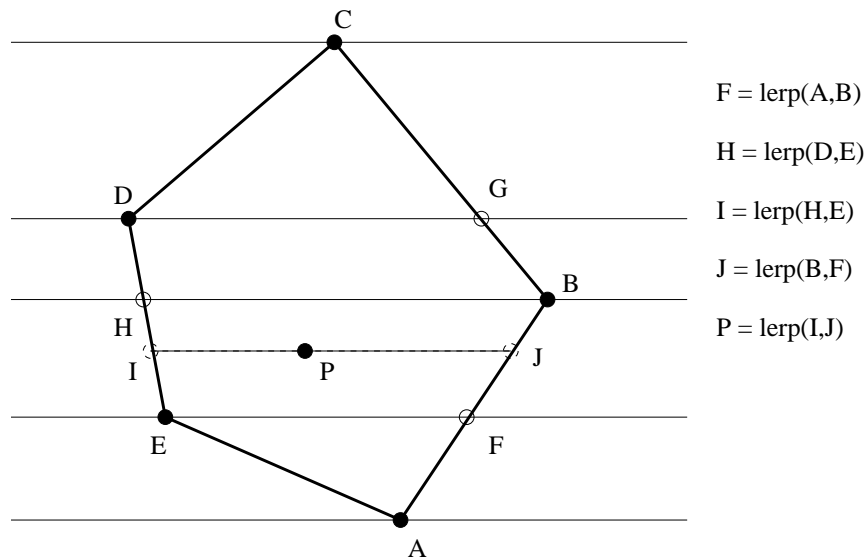
$$\alpha + \beta + \gamma = 1$$

$$P = \alpha A + \beta B + \gamma C$$

- To implement, can use repeated affine combination along edges, across spans, during rasterization.

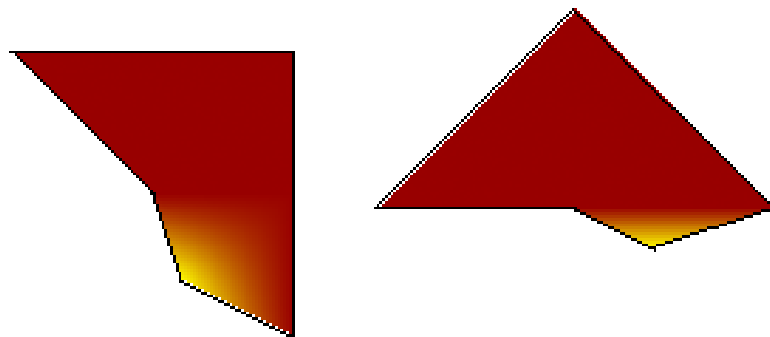


- Gouraud shading is well-defined only for triangles
- For polygons with more than three vertices:
  - Sort the vertices by  $y$  coordinate.
  - Slice the polygon into trapezoids with parallel top and bottom.
  - Interpolate colours along each edge of the trapezoid. . .
  - Interpolate colours along each scanline.



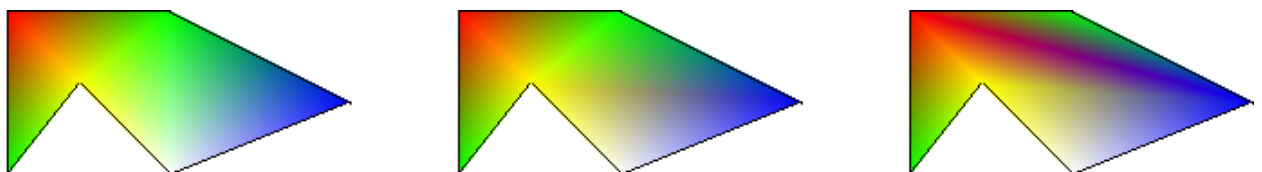
### Problems with Shading by Slicing

- Shading by slicing gives shading that is non-invariant under rotation.



### Triangulate and Shade

- Another idea: triangulate and shade triangles
- Problems: expensive, and can see triangulation



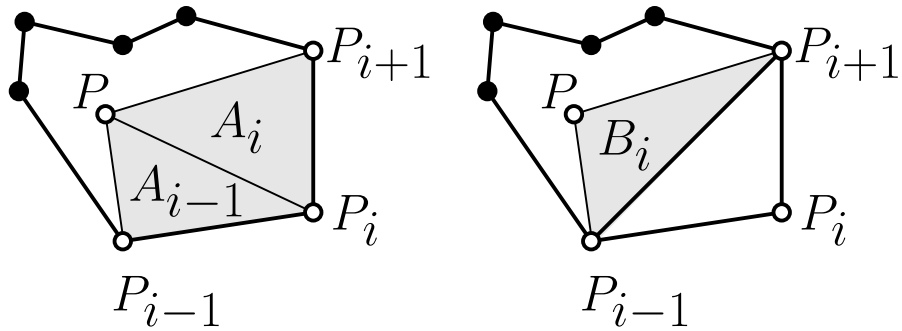
### Mean Value Coordinates

Mean value coordinates provide a generalization of Barycentric coordinates

- Unnormalized:

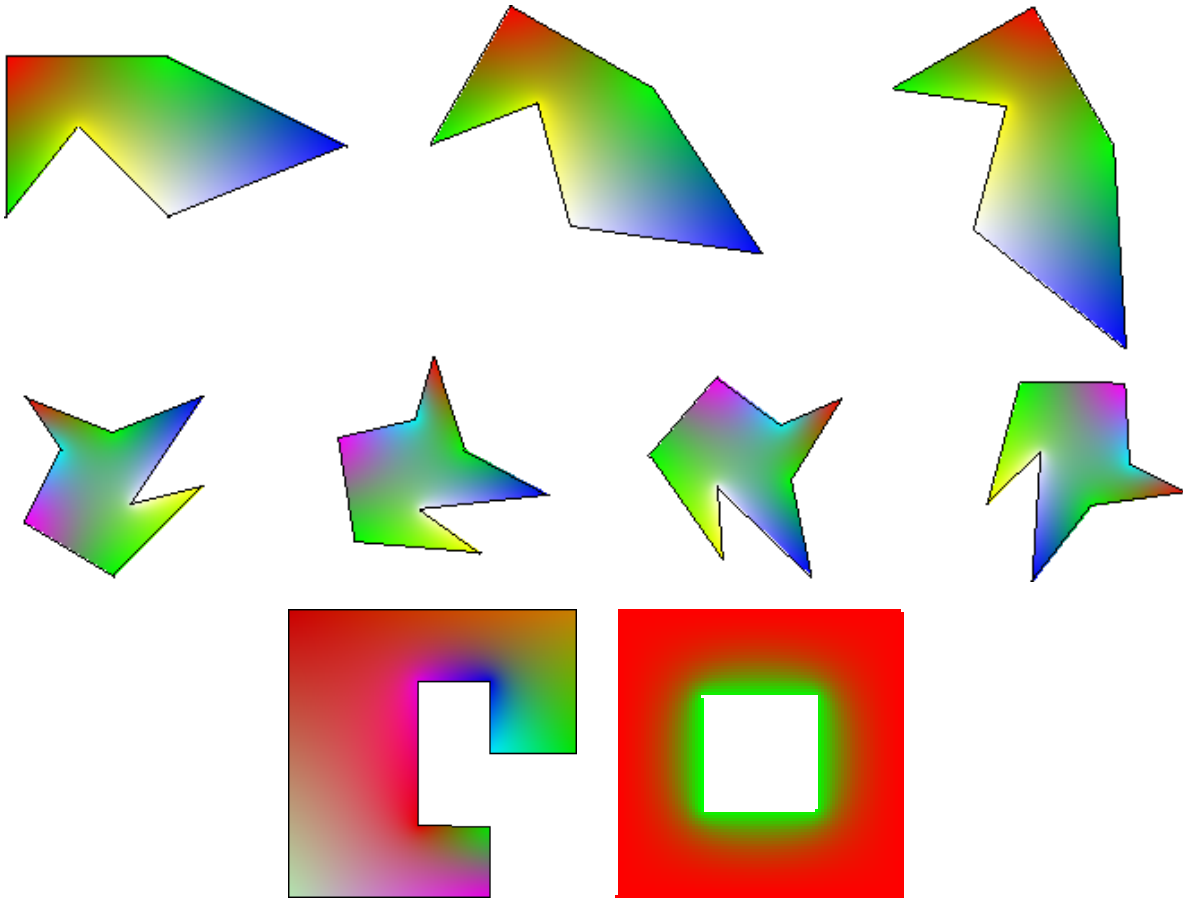
$$w_i = \frac{r_{i+1}A_{i-1} - r_iB_i + r_{i-1}A_i}{A_{i-1}A_i}$$

where  $r_i = \|P - P_i\|$ ,



- To normalize, divide by sum of  $w_i$

Example of shading with MVC:



(Readings: Kai Hormann and Michael S. Floater, Mean Value Coordinates for Arbitrary Planar Polygons, ACM TOG, Vol 25, No 4, 2006. Pg 1424–1441. )



### 18.3 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.
- Phong 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
  - Interpolate normals, view vectors, and light vectors using generic interpolation hardware
  - Usually also interpolate diffuse term after computing it in vertex shader.
  - Other kinds of advanced shading possible

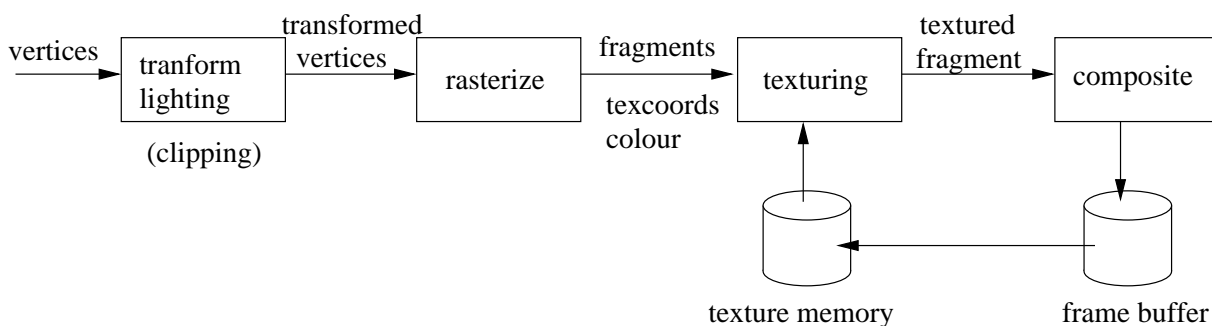


## 19 Graphics Hardware

### 19.1 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 (later Direct3D) is MicroSoft's challenge to OpenGL
- In the 00's, PC graphics hardware had completely displaced workstation graphics  
Excellent inexpensive cards for PCs
- 2014: Apple's Metal
- 2016: Vulkan: low-level OpenGL
- Intrinsically parallel GPUs getting faster faster than single-core CPUs
- Many GPUs are significantly more powerful than CPUs in floating-point
- Can be used for general purpose computation too! PCI-express makes graphics accelerators like coprocessors

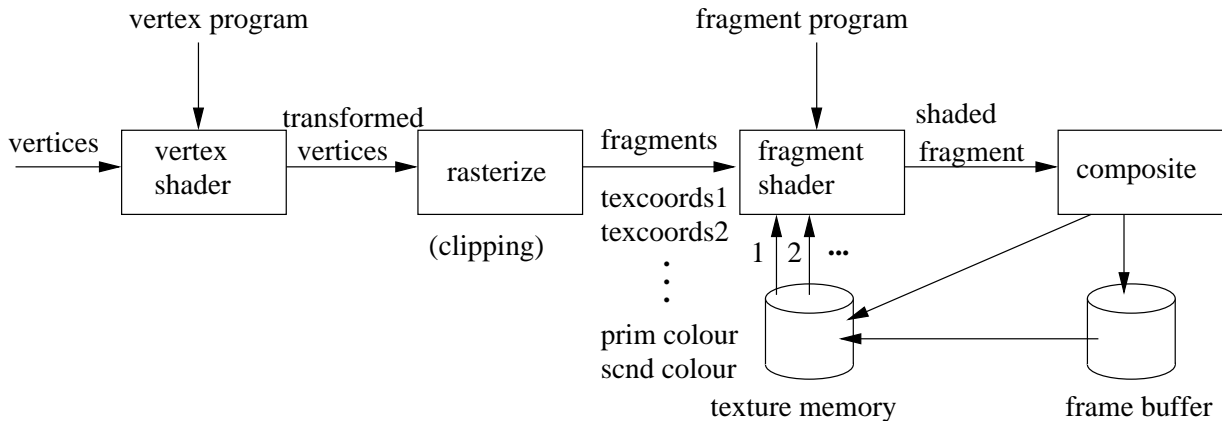
Traditional OpenGL (1.2)



- A fragment is a sort of virtual pixel
- fragment carries colour that is composited onto pixel in frame buffer if it passes depth, stencil, and alpha tests.

- Clipping done in transform/lighting portion

OpenGL (2.0)



- Vertex Shader and Fragment Shader are programmable by application

## 19.2 Shader Idea

Consider z-buffer code:

```

y = A.y;
d0x = (C.x-A.x)/(C.y-A.y); d0z = (C.Z-A.Z)/(C.y-A.y);
d1x = (C.x-B.x)/(C.y-B.y); d1z = (C.Z-B.Z)/(C.y-B.y);
x0 = A.x; x1 = B.x; z0 = A.z; z1 = B.z;
while ( y <= C.y ) do
    z = z0; d2z = (z1-z0)/(x1-x0);
    for ( x = x0 to x1 ) do
        WritePixel(x,y,z); z += d2z;
    end
    x0 += d0x; x1 += d1x; z0 += d0z; z1 += d1z;
    y++;
end

```

Want to add Gouraud shading, given RGB values at vertices

Do the same thing in RGB that we did in z:

```

y = A.y;
d0x = ...; d1x = ...; d0z = ...; d1z = ...;
d0R = (C.R-A.R)/(C.y-A.y); d0G = ...; d0B = ...;
d1R = (C.R-B.R)/(C.y-B.y); d1G = ...; d1B = ...;
x0 = ; x1 = ; z0 = ; z1 = ; R0 = A.R; R1 = B.R; G1 = ...
while ( y <= C.y ) do
    z = z0; d2z = (z1-z0)/(x1-x0);

```

```

    r = r0; d2r = (R1-R0)/(x1-x0); ...
    for ( x = x0 to x1 ) do
        WritePixel(x,y,z);    z += d2z; r += d2r; ...
    end
    x0 += d0x; x1 += d1x;  z0 += d0z; z1 += d1z;
    R0 += d0R; R1 += d1R;  G0 += ...          y++;
end

```

Idea: compute data per vertex; have generic slope computations

```

y = A.y;
A.D = VertexInfo(A); B.D = VertexInfo(B);
C.D = VertexInfo(C);
d0 = SlopeInfo(A.D,C.D); d1 = SlopeInfo(B.D,C.D);
E0 = A.D; E1 = B.D;
while ( y <= C.y ) do
    [SL,dSL] = ScanLine(E0,E1);
    for ( x = E0.x to E1.x ) do
        RGB = ComputeColour(SL);    SL += dSL;
        WritePixel(x,y,z,RGB);
    end
    E0 += d0; E1 += d1;
    y++;
end

```

### Shader idea

- User (programmer) writes `VertexInfo` and `ComputeColour` routines, hardware does the rest.
- Data computed in `VertexInfo` put in special variable type so that hardware knows to compute slopes, etc.
- `ComputeColour` has access to these special variables
- Have “access to” transformation matrices, perspective matrix  
Map between spaces
- Additional idea: give shader programs access to “images”  
Read and write (well...)

## 19.3 High-Level Shading Languages

- OpenGL, DirectX APIs support assembly interface for vertex and fragment programs
- Hard to program (maintain, read) assembly
- Several high level shading languages now available: Cg, GLSL, Microsoft HLSL, Sh

## OpenGL Shading Language (GLSL)

- C-based language; has similar syntax and flow control a little more restrictive in some areas
- Functions: return by value, pass by value
- Has special built-in variables for input/output.  
Their names always begin with `gl_`
- Has built-in datatypes for matrices and vectors
- Useful built-in functions: `dot`, `cos`, `sin`, `mix`, ...

## Swizzling

- Structure-member selector (`.`) also used to SWIZZLE components of a vector:  
select or rearrange components by listing their names after the swizzle operator (`.`)

```
vec4 v4;
v4.rgba; // is a vec4 and the same as using v4,
v4.rgb;  // is a vec3,
v4.b;    // is a float,
v4.xy;   // is a vec2,
v4.xgba; // is illegal - the component names do
// not come from the same set.
```

- Component names can be out of order to rearrange the components, `v4.rbag`  
or they can be replicated to duplicate the components
- Uniform variables: can be changed once per primitive  
`uniform bool bumpon;`
- Attribute variables: can be changed anytime, represent attributes associated with vertices  
`attribute vec3 tangentVec;`
- Varying variables (in/out): used to communicate between vertex shader and fragment shader.  
They are automatically interpolated across the polygon.  
`out vec3 lightVec;`  
`in vec3 lightVec;`

## Diffuse Gouraud Shader

```
// diffuse.vert
in vec3 position; in vec3 normal;
...
uniform mat4 ModelView; uniform mat4 Perspective;
out vec4 colour;
```

```

vec3 diffuseLighting(vec3 vertPosition, vec3 vertNormal) {
    vec3 l = normalize(light.position - vertPosition);
    float n_dot_l = max(dot(vertNormal, l), 0.0);
    vec3 diffuse = material.kd * n_dot_l;
    return ambientIntensity*material.kd + light.rgbIntensity*diffuse;
}

void main( ) {
    vec4 pos4 = vec4(position, 1.0);
    colour = diffuseLighting((ModelView * pos4).xyz, NormalMatrix * normal);
    gl_Position = Perspective * ModelView * pos4;
}

// diffuse.frag
in vec4 colour; out vec4 fragColour;
void main() { fragColor = colour; }

```

## Normal Map

- Will perturb normal based on *normal map*  
Assume we perturb normal (0,0,1)
- An image where each pixel represents a normal:

$$R \rightarrow x, \quad G \rightarrow y, \quad B \rightarrow z$$

- Normal is perturbed a small amount, so normal maps tend to be bluish in colour
- R, G, and B each range from 0.0 to 1.0  
 $N.x = (R - 0.5) * 2.0, \dots$

```

1. uniform bool bumpon;
2. attribute vec3 tangentVec;
3. out vec3 lightVec;
4. out vec3 eyeVec;
5.
6. void main (void)
7. {
8.     vec4 ecPosition = gl_ModelViewMatrix * gl_Vertex;
9.
10.    gl_Position = ftransform();
11.    gl_TexCoord[0] = gl_MultiTexCoord0; //texture mapping stuff
12.
13.    vec3 orgLightVec = ( gl_LightSource[0].position.xyz - ecPosition.xyz);
14.
15.    vec3 n = normalize(gl_NormalMatrix * gl_Normal);
16.    vec3 t = normalize(gl_NormalMatrix * tangentVec);
17.    vec3 b = cross(n, t);
18.
19.    lightVec.x = dot( orgLightVec, t);
20.    lightVec.y = dot( orgLightVec, b);

```

```
21.     lightVec.z = dot( orgLightVec, n);
22.
23.     vec3 position = -ecPosition.xyz;
24.     eyeVec.x = dot( position, t);
25.     eyeVec.y = dot( position, b);
26.     eyeVec.z = dot( position, n);
27. }
```

```
1. uniform bool bumpon;
2. uniform sampler2D normalMap;//must initialize with texture unit integer
3. in vec3 lightVec;
4. in vec3 eyeVec;
5.
6. void main (void)
7. {
8.     vec3 N = vec3(0.0, 0.0, 1.0);
9.     if( bumpon ) N = normalize( ( texture2D(normalMap, gl_TexCoord[0].xy).xyz) - 0.5) * 2.0 );
10.
11.     vec3 L = normalize(lightVec);
12.     vec3 V = normalize(eyeVec);
13.     vec3 R = reflect(L, N);
14.
15.     float pf = pow( dot(R, V), gl_FrontMaterial.shininess );
16.
17.     vec4 GlobalAmbient = gl_LightModel.ambient;
18.     vec4 Ambient = gl_LightSource[0].ambient * gl_FrontMaterial.ambient;
19.     vec4 Diffuse = gl_LightSource[0].diffuse * gl_FrontMaterial.diffuse * dot(N, L);
20.     vec4 Specular = gl_LightSource[0].specular * gl_FrontMaterial.specular * pf;
21.
22.     vec4 color1 = GlobalAmbient + Ambient + Diffuse;
23.     vec4 color2 = Specular;
24.
25.     gl_FragColor = clamp( color1 + color2, 0.0, 1.0 );
26. }
```

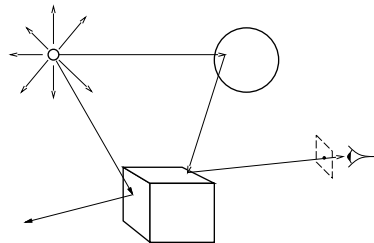


## 20 Ray Tracing

### 20.1 Fundamentals

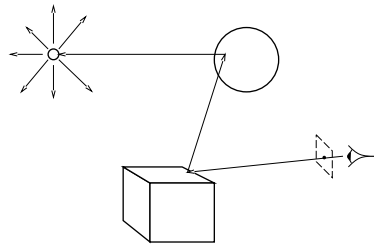
**Ray Tracing:** Photorealism – The Holy Grail

- 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

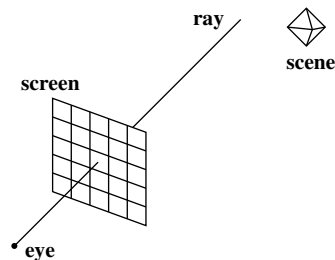
```
foreach pixel
  ray = (eye, pixel-eye);
  Intersect(Scene, ray);
end
```

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



(Readings: McConnell: Chapter 8. Watt: 12.1. Red book: 14.7. White book: 16.12. )

## 20.2 Intersection Computations

### Intersection Computations

#### General Issues:

- Ray: express in parametric form

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

where  $E$  is eyepoint and  $P$  is pixel point

- Direct implicit form
  - express object 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  $f(E + t(P - E)) = 0$

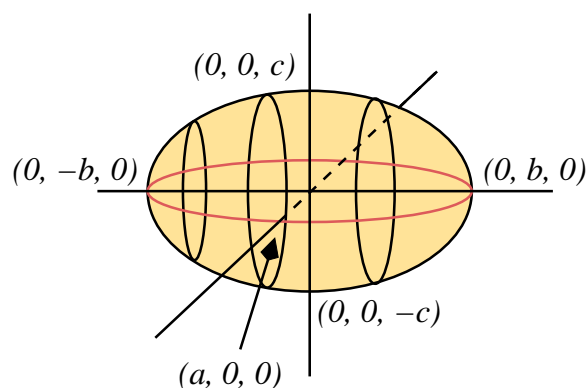
- Procedural implicit form
  - $f$  is not a given formula
  - $f$  is only defined procedurally
  - $\mathcal{A}(f(E + t(P - E)) = 0)$  yields  $t$ , where  $\mathcal{A}$  is a root finding method (secant, Newton, bisection, ...)

**Quadric Surfaces**

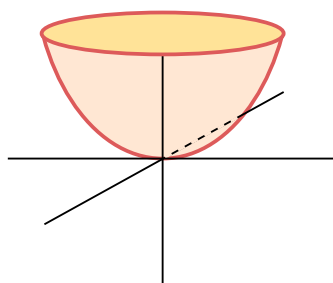
Surface given by

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

- Ellipsoid:  $\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} = 1$



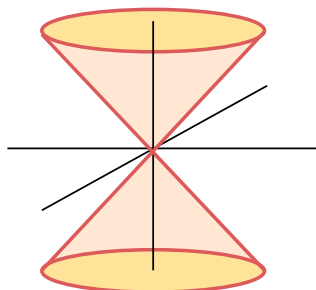
- Elliptic paraboloid:  $\frac{x^2}{p^2} + \frac{y^2}{q^2} = 1$



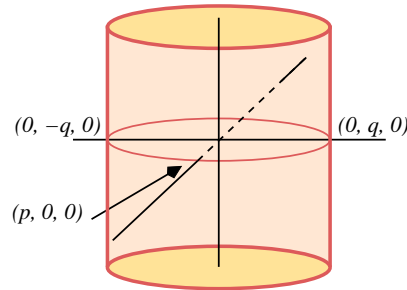
- Hyperboloid of one sheet:  $\frac{x^2}{a^2} + \frac{y^2}{b^2} - \frac{z^2}{c^2} = 1$

**Hyperboloid of one sheet**

- Elliptic Cone:  $\frac{x^2}{p^2} + \frac{y^2}{q^2} - \frac{z^2}{r^2} = 0$



- Elliptic cylinder:  $\frac{x^2}{p^2} + \frac{y^2}{q^2} = 1$



- 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 successive 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  $\alpha$  coordinates

**Alternatives:** Must this all be done in world coordinates?

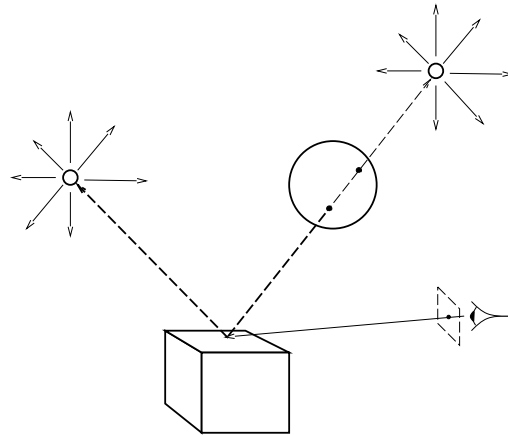
- Alternative: Intersections in model space
  - form normals and planar coordinates in model space and store with model
  - backtransform the ray into model space using inverse modeling transformations
  - perform the intersection and illumination calculations
- Alternative: Intersections in world space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

(Readings: McConnell: 8.1.)

## 20.3 Shading

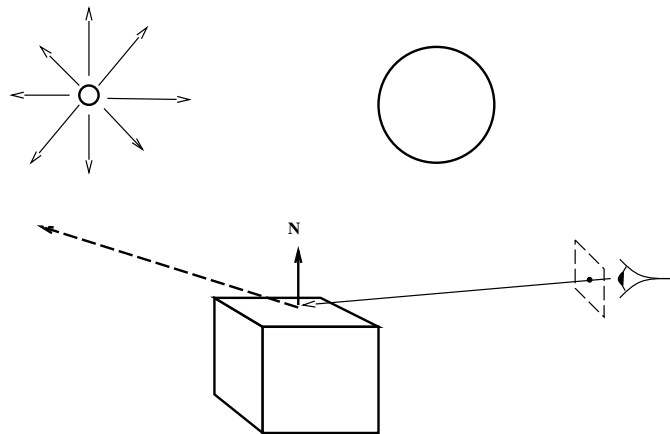
Shading, shadows

- How do we use ray to determine shade of pixel?
- At closest intersection point, perform Phong shading.
  - Gives same images as polygon renderer, but more primitives.
- Idea: Before adding contribution from a light, cast ray to light.
  - If ray hits object before light, then don't shade.
  - This gives shadows.

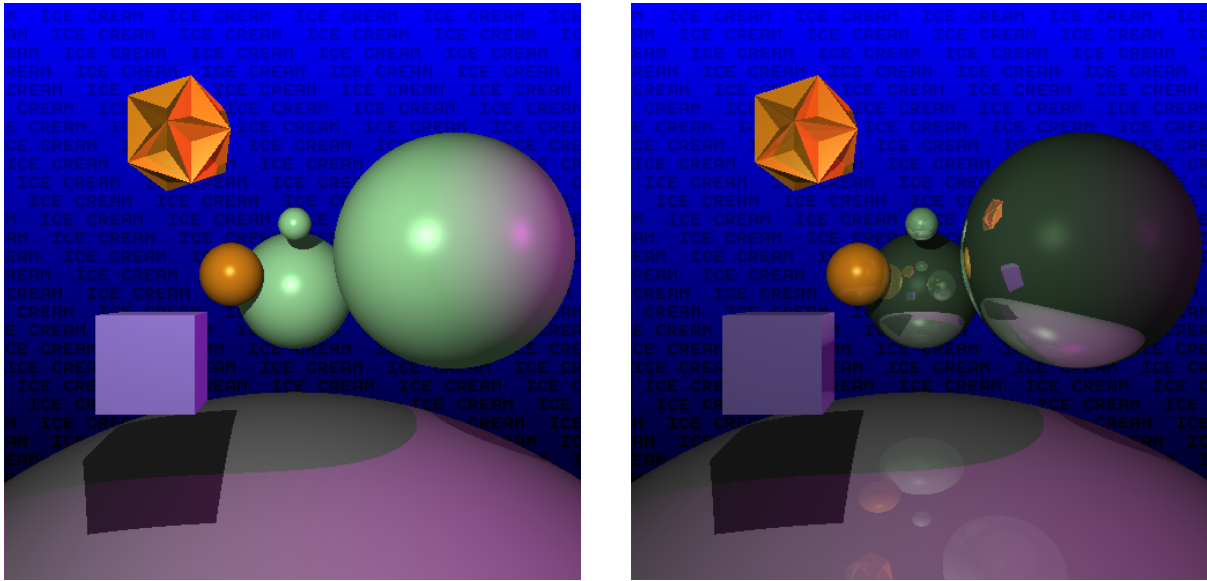


### Reflections

- 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.



(Readings: McConnell: 8.2, 8.3. Watt: 12.1.)

## 20.4 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: McConnell: 8.4. Watt: 12.2.)

## 20.5 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:

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

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

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

Normal:

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

- Normal to a sphere, cylinder, cone...

**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  $\mathbf{N}$ :

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

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:

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

$\implies$  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:

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

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

$\Rightarrow$  reciprocal scale applied to normal

### Normal transformation details

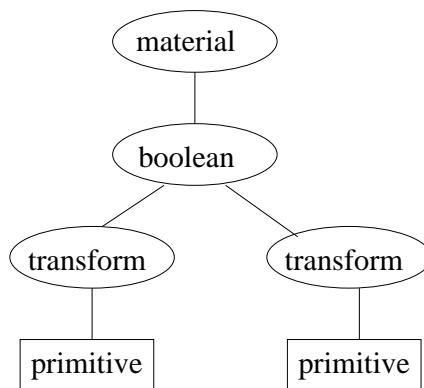
For affine transformation matrix  $M$  (upper 3x3 portion only),  
row matrix  $n$

- Apply  $(M^{-1})^T n$  if treating normal as column matrix
- Apply  $n^T M^{-1}$  if treating normal as row matrix

## 20.6 Modeling and CSG

**Modeling:** Constructive Solid Geometry

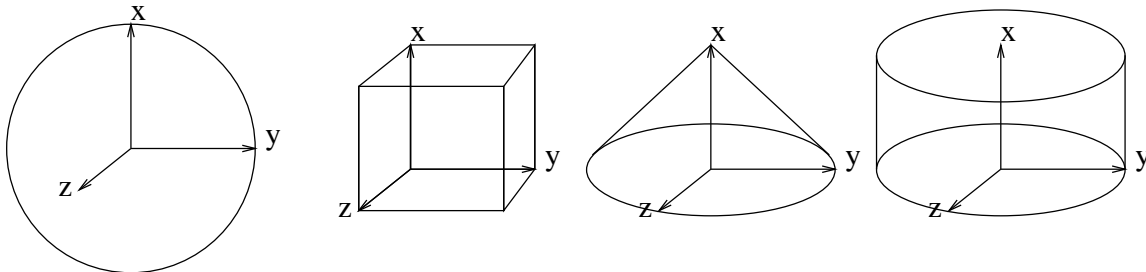
- How do we model for a Ray Tracer?
- Hierarchical modeling works well, but we can do more.
- In *Constructive Solid Geometry* all primitives are solids.
- New type of internal node: Boolean operation.  
Intersection, Union, Difference
- Thus, our model is a DAG with
  - Leaf nodes representing primitives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



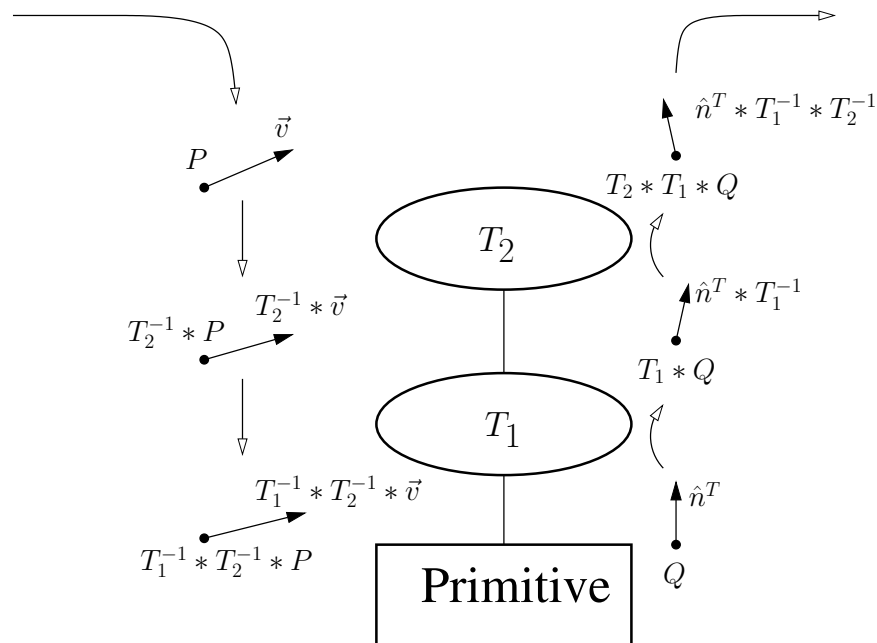
`Sphere()*Scale(2,2,2)*Translate(x,y,z)`

- 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);`

**Traversing the transformation tree**

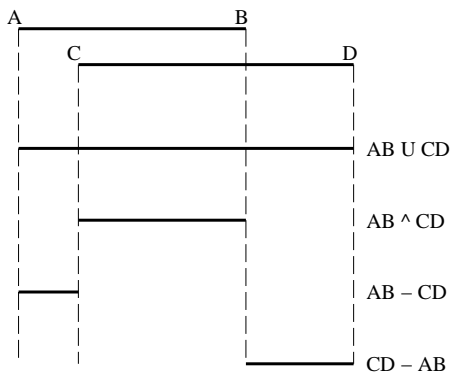
- After transformation applied, primitives will be warped.
- Rather than intersect ray with warped primitive, we transform the ray.
- On the way down, apply *inverse transformation* to ray.
- On the way back, apply transformation to the point, normal

**Caution:** see notes on transforming normals

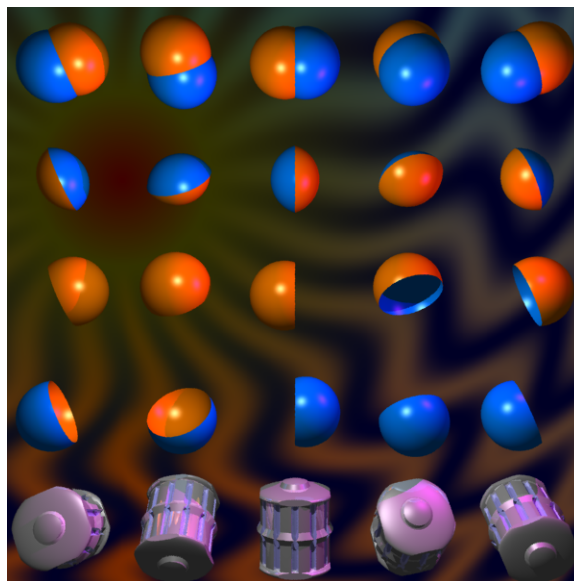


**CSG:** Boolean operations

- Could apply boolean operations to transformed primitives  
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.

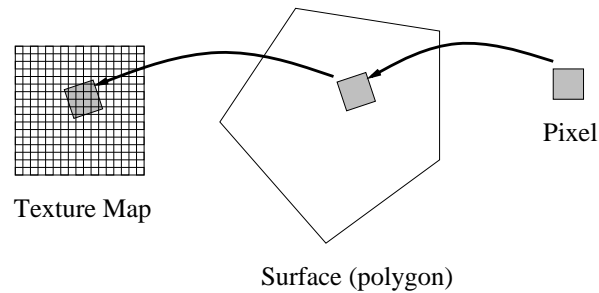


Examples: Union, Intersection, Difference, Difference, Complex Model

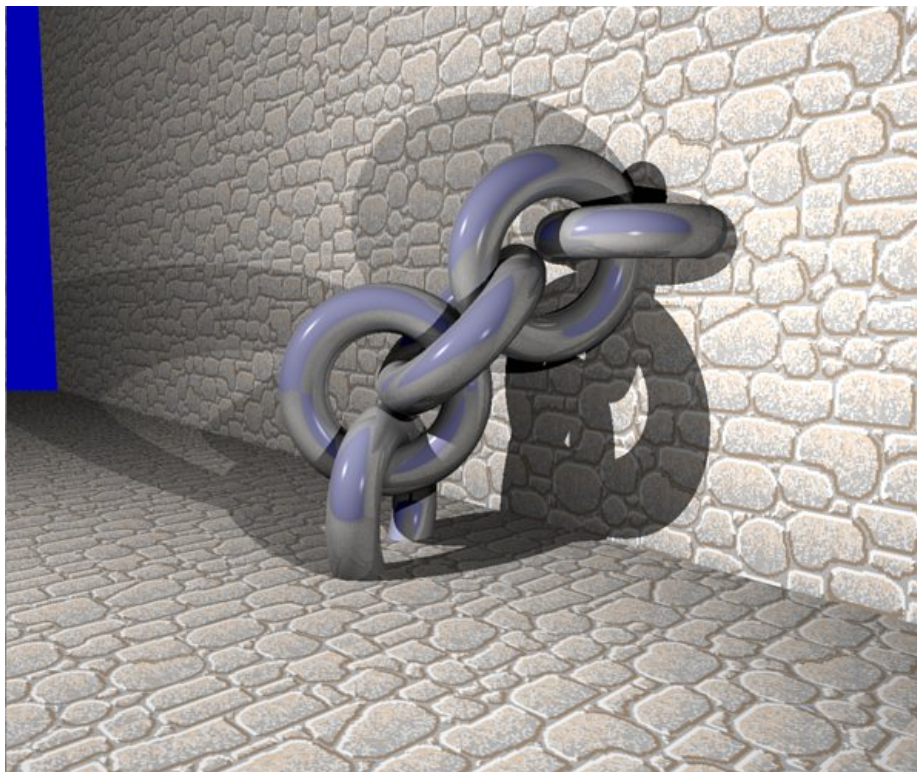
**20.7 Texture Mapping**

- 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.

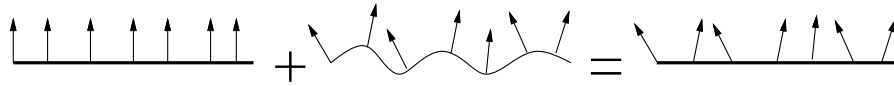


- Tile polygon with texture if needed
  - For some textures, tiling introduces unwanted pattern
  - Various method to extend textures
- Greatly improves images!
- Not just ray traced image

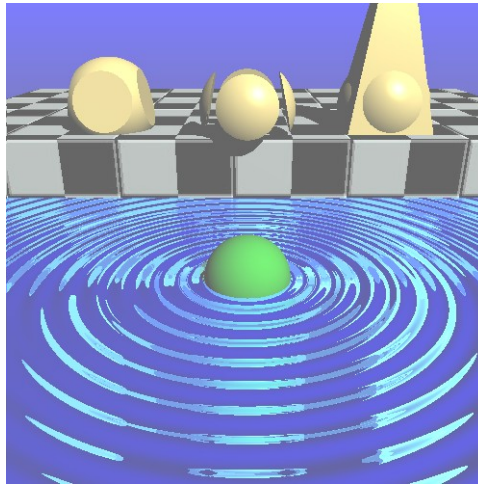


## Bump (Normal) Mapping

- Textures will still appear “smooth” (i.e., no shadows)
- Bump mapping is similar to texture mapping except that we perturb the normal rather than the colour.



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



- Large geometric features require geometry  
Window facades
- Texture mapping adds flat detail  
Soup can label
- Normal mapping appropriate when angle of light affects shading in obvious ways  
Mortar in brick/cylinder block walls
- Normal mapping okay for certain types of rough texture  
(but texture mapping probably just as good)
- Mis-used normal maps draw unwanted attention  
Flickering—avoid normal maps on highly specular surfaces!

## Examples

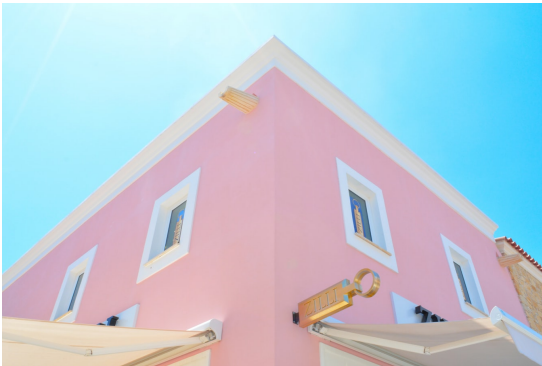
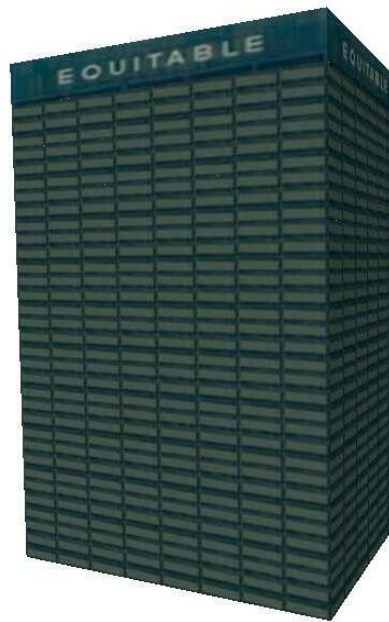


Photo by Mike Yukhtenko on Unsplash

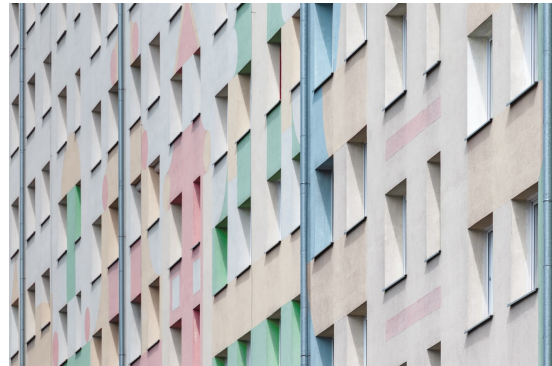
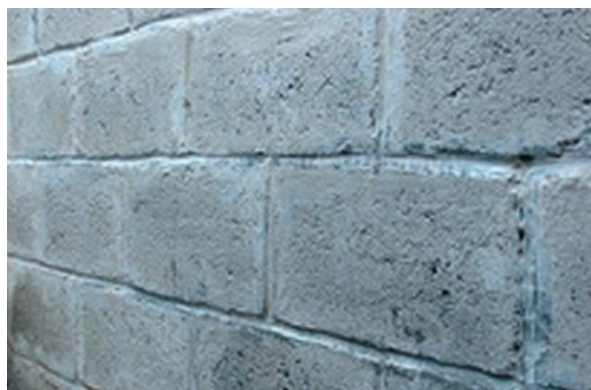


Photo by Pawel Czerwinski on Unsplash





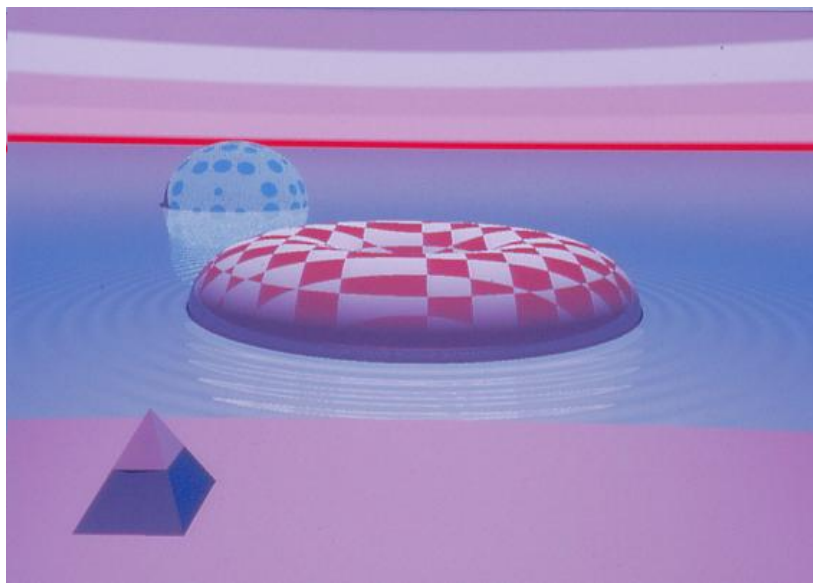
## Solid Textures

- 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) Examples:



(Readings: Watt: 8.)



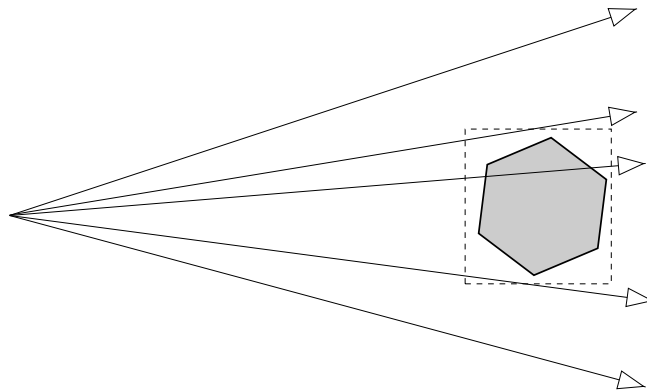
## 20.8 Bounding Boxes, Spatial Subdivision

### 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

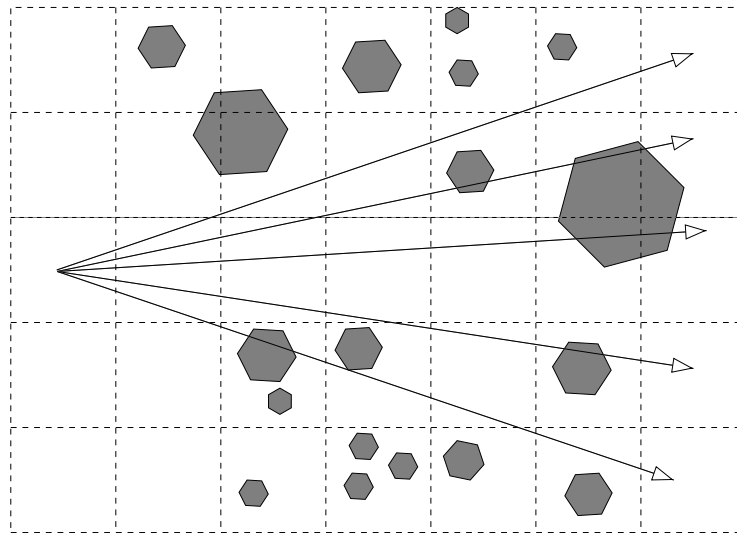
- Idea: Place a bounding box around each object
- Only compute ray intersect object if the ray intersects the bounding box



- If box aligned with coordinate axes, ray intersect box is very cheap
- 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
- Be careful with CSG/Hierarchical

### 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



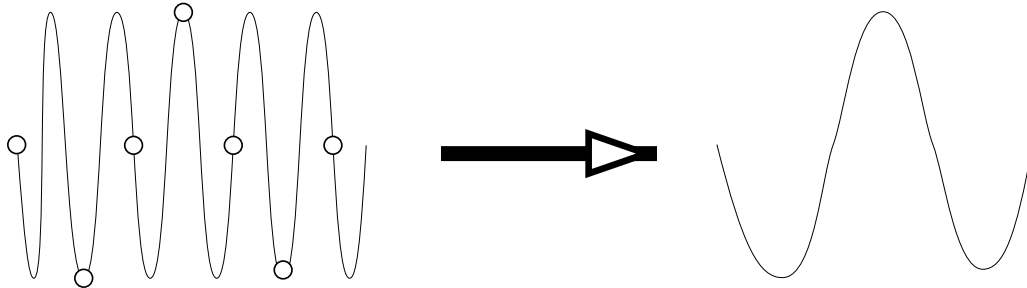
- Useful when lots of small objects in scene
- Determining subdivision is difficult

## 21 Aliasing

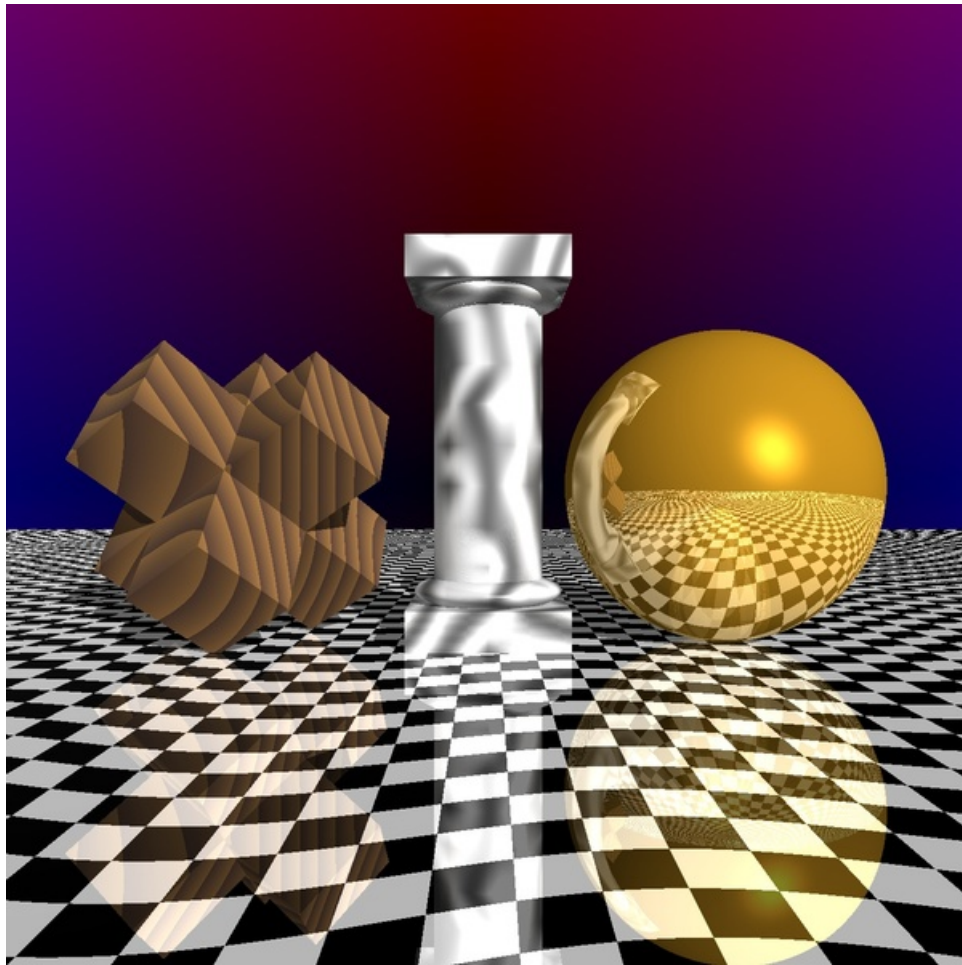
### 21.1 Aliasing

#### Aliasing and Anti-Aliasing

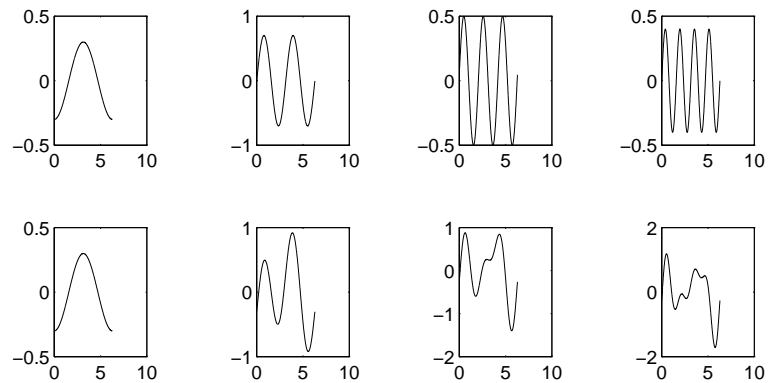
- Raster Image is a sampling of a continuous function
- If samples spaced too far apart, then don't get true representation of scene:



- In graphics, a variety of bad things happen:
  - Stairstep or “jaggies”  
Shaders: texture maps
  - Moire Patterns
  - Loss of small objects
  - Temporal
    - \* Sampled too far apart in time  
Backwards rotating wheels  
Stationary helicopter blades
    - \* Crawling jaggies
    - \* Appearing/dissappearing objects, flashing objects

**Example Moire pattern****Image as a signal**

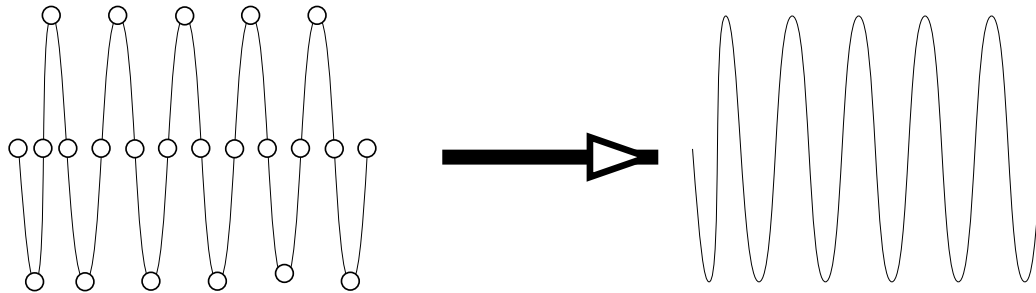
- Signal processing signal is function of time
- Scene is a function of space  
Spatial domain:  $f(u)$
- Raster image is a sampling of this signal
- Can represent signal as sum of sine waves  
Frequency domain:  $F(u)$



- 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?

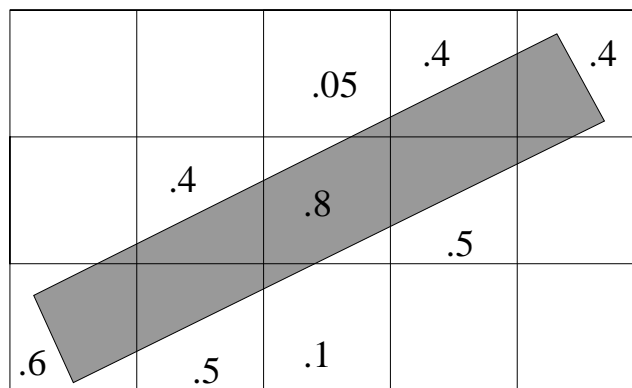
- After we get image, run through low pass filter  
This helps, but not a full solution
- Get a Higher Resolution Monitor  
This helps, but...
  - Not usually feasible

- 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



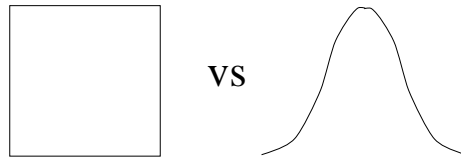
- Colour shades of gray based on fraction of pixel covered
- Gives better looking images *at a sufficiently far distance*  
Look close and it looks blurry

## 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

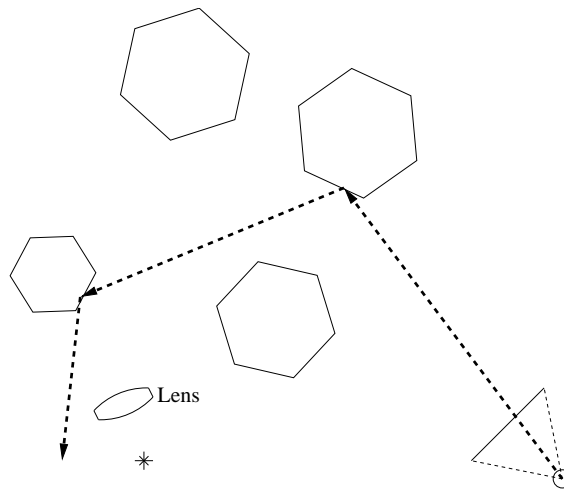




## 22 Bidirectional Tracing

### 22.1 Missing Effects

- Problem: Raytracing can't model all effects  
Caustics, indirect illumination



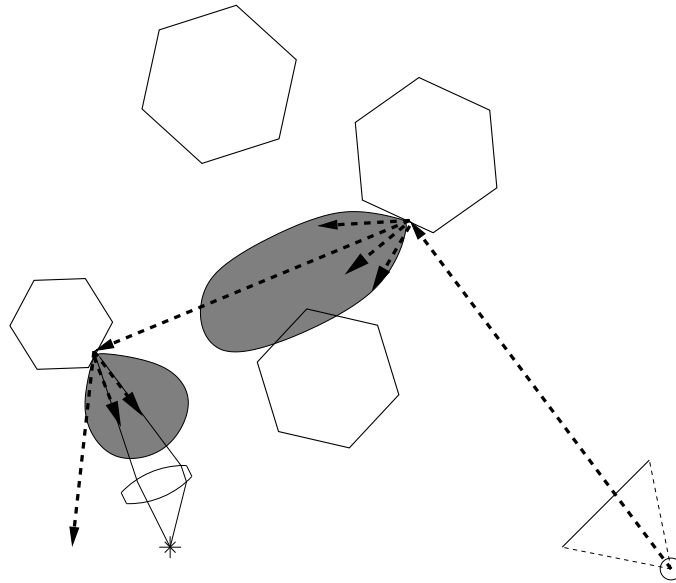
- Some effects requiring tracing light rays from light back to eye
- Too many to trace them all. How do we select?

### Radiosity

- Model diffuse interaction of light only in steady state
  - Discretize scene into polygons  
Assume each polygon “emits” constant light over its surface
  - Determine light interaction between each pair of polygons  
(Including light sources)  
Fraction of light leaving one polygon that reaches the other
  - Solve large linear system  
(iterative solution)
- Expensive, but compute once and view anywhere
- Occlusion/shadows are hard

## 22.2 Distribution Ray Tracing

- Ray trace, but at each reflection cast a lot of rays
- Cast with distribution function to represent reflection properties



- Distributing rays over reflected direction gives soft reflections (similar effect if distributed over transmission direction)
- Distributing rays over light area gives soft shadows
- Distributing rays over aperture gives depth of field
- Distributing rays over time gives motion blur
- Can use to model caustics, but need *lots* of rays (20+ rays per reflection)
- Can model many effects simply, but often very expensive

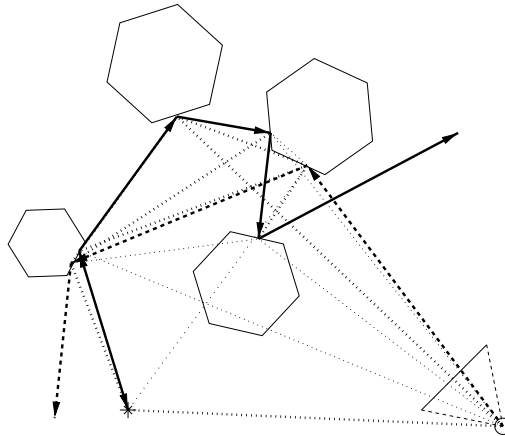
Example: Distributed reflection

Example: Distributed aperture

## 22.3 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

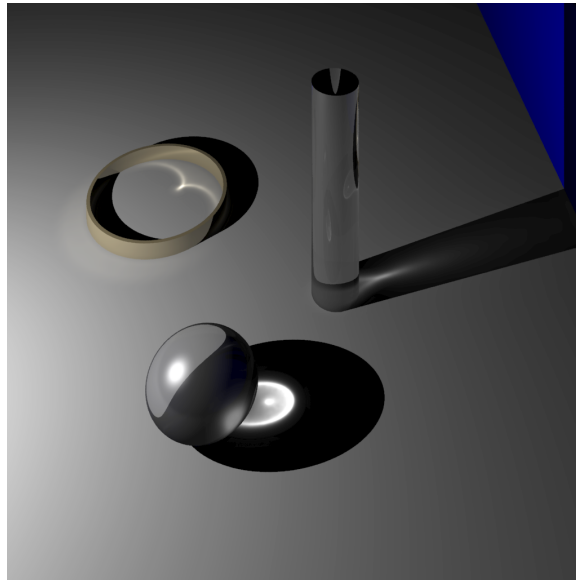


- Low error, high noise
- Less expensive than distribution ray tracing

## 22.4 Photon Maps

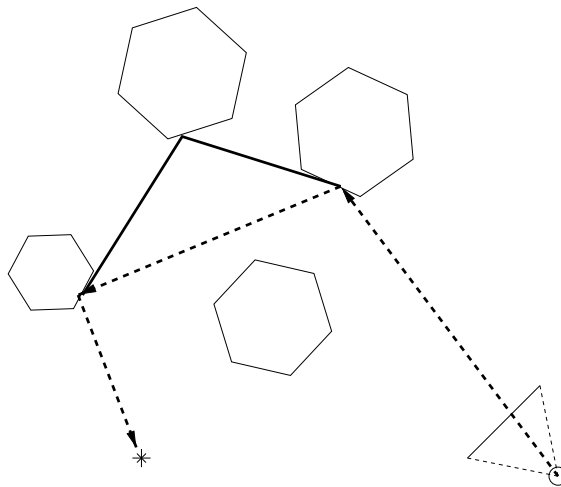
- Cast rays from light  
Global illumination (view independent)
- Create a “photon map” wherever this light hits
- Then use standard ray *casting*,  
no secondary rays  
look at photon map(s) to determine special lighting
- Use density estimation to convert photon map to intensity
- Fast, easy to program, high noise
- Errors

Example: caustics



## 22.5 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

## 22.6 Path Tracing

- Idea: Cast multiple rays from eye  
Do better simulation for each ray
- BRDF material functions
- Use area lights; don't cast light, shadow, mirror rays  
Pick single random direction to cast a refelected ray
- Distribution comes from multiple rays per pixel  
and random ray on bounce
- In the limit, converges to true image
- Problem: Need a lot of rays per pixel  
Convergence may be slow
- Stopping early gives noisy images  
Filter to remove noise



## 23 Rendering Equation

### 23.1 Definitions and Overview

#### The Rendering Equation

Want a mathematical model of light

- Should model most effects of light
- Possibly too complex to solve, but look at approximations  
Look at error, see how to reduce
- Will create model for  
light coming into/leaving a point on a surface

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

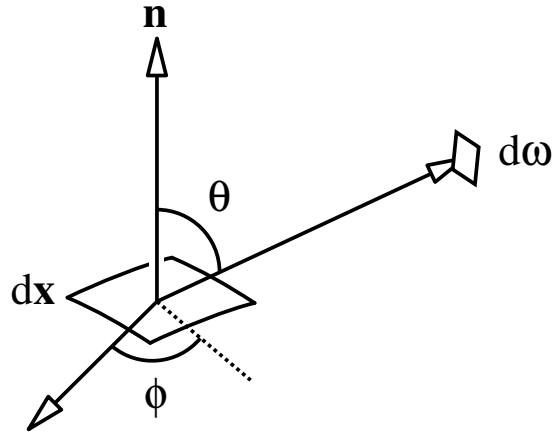
- at some point  $x$
- in a specified direction  $\theta, \phi$
- per unit time
- per unit area perpendicular to the direction
- per unit solid angle
- for a specified wavelength  $\lambda$
- denoted by  $L(x, \theta, \phi, \lambda)$

**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)

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

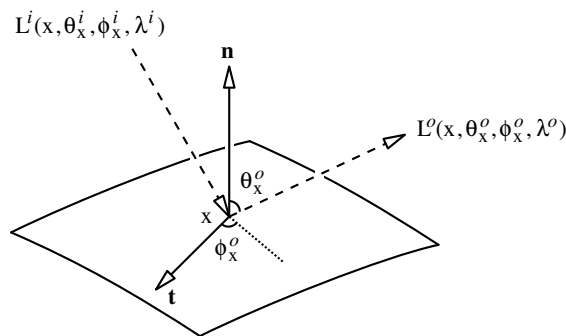
**Radiosity:** Total power leaving a surface point per unit area

- $\int_{\Omega} L(x, \theta, \phi) \cos \theta d\omega = \int_0^{\frac{\pi}{2}} \int_0^{2\pi} L(x, \theta, \phi) \cos \theta \sin \theta d\phi d\theta$   
(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

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





**Energy Balance Equation:**

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

- $L^o(x, \theta_x^o, \phi_x^o, \lambda^o)$  is the radiance
  - at wavelength  $\lambda^o$
  - leaving point  $x$
  - in direction  $\theta_x^o, \phi_x^o$
- $L^e(x, \theta_x^o, \phi_x^o, \lambda^o)$ : radiance emitted by surface from point
- $L^i(x, \theta_x^i, \phi_x^i, \lambda^i)$ : 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

**Assumptions, limitations**

Assumptions:

- Wavelengths independent (removes one integral)
  - But: can't model fluorescence
- Often, additional assumptions made to simplify
  - Example: radiosity assumes piecewise constant

Limitations:

- Doesn't model:
  - Transmission, subsurface scattering, polarization, phosphorescence, other effects



## 24 Radiosity

### 24.1 Definitions and Overview

#### Radiosity

**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

#### Energy Balance Equation:

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

- $L^o(x, \theta_x^o, \phi_x^o, \lambda^o)$  is the radiance
  - at wavelength  $\lambda^o$
  - leaving point  $x$
  - in direction  $\theta_x^o, \phi_x^o$
- $L^e(x, \theta_x^o, \phi_x^o, \lambda^o)$ : radiance emitted by surface from point
- $L^i(x, \theta_x^i, \phi_x^i, \lambda^i)$ : 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
  - $L(x, \theta_x, \phi_x) \equiv L(x)$
  - $B(x) \text{ def. } = \int_{\Omega} L(x) \cos(\theta_x) d\omega = \pi L(x)$

**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  $\pi$  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

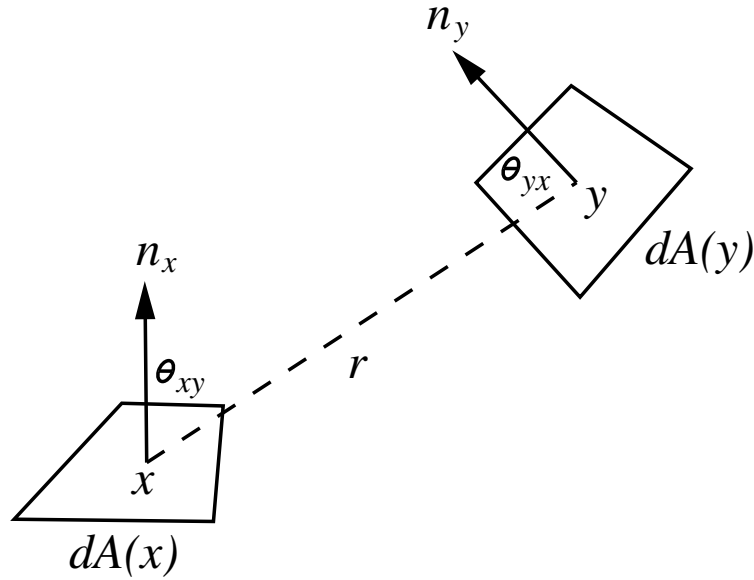
$$L^i(x, \theta_x^i, \phi_x^i) = L^o(y, \theta_y^o, \phi_y^o) \text{ for some } y$$

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):**

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

**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 Symmetry)

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

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

**Radiosity:** (Summary)

- Discretize scene into  $n$  patches (polygons) each of which emits and reflects light uniformly under its entire area.
- Radiosity emitted by patch  $i$  is given by

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

- $B_i, B_j$ : radiosity in energy/unit-time/unit-area
- $E_i$ : light emitted from patch  $i$
- $\rho_i$ : 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

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

or

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

- In matrix form

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

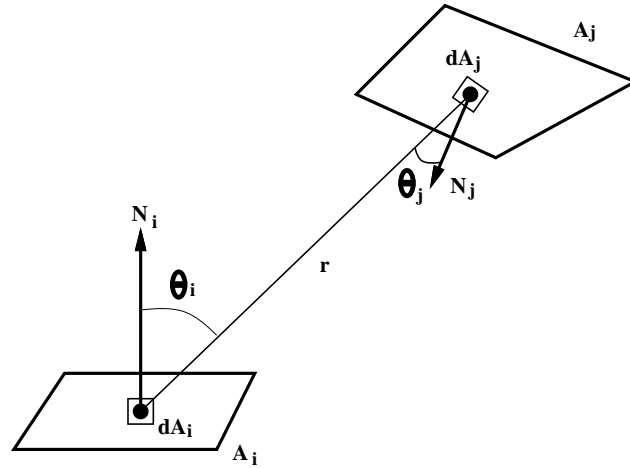
- $B_i$ 's are only unknowns
- If  $F_{i,i} = 0$  (true for polygonal patches) then diagonal is 1
- Solve 3 times to get RGB values of  $B_i$

(Readings: McConnell: Chapter 9. Hearn and Baker in OpenGL: 10-12. Watt: Chapter 11.)

**24.2 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:

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

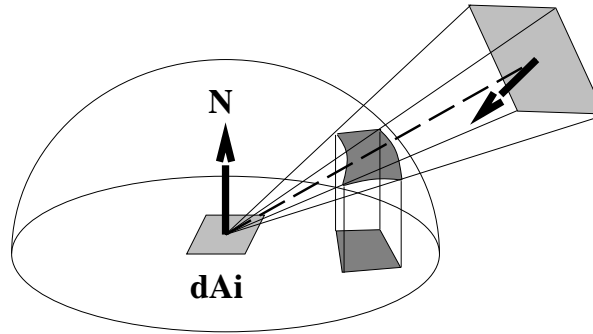
- The form factor from  $A_i$  to  $A_j$  is an area average of the integral over  $A_i$ :

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

- 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

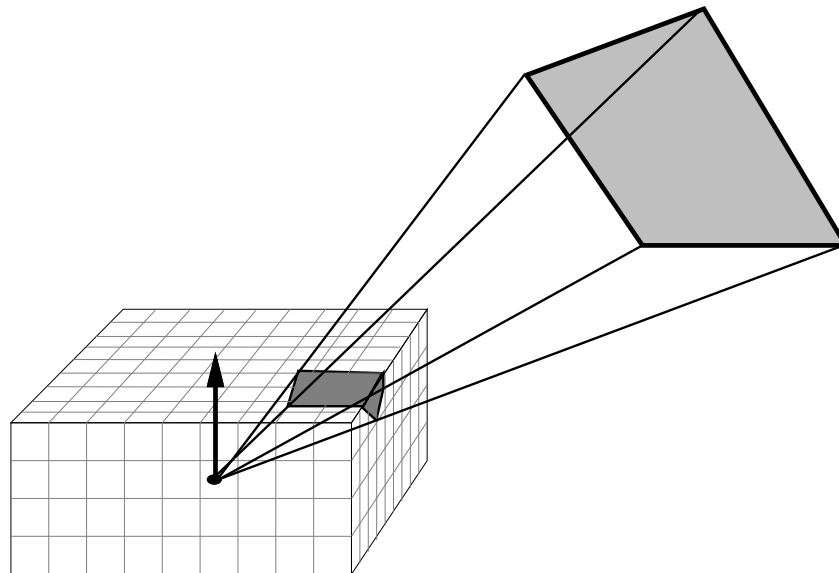
- Place a hemisphere around patch  $i$  and project patch  $j$  onto it



- 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.
- 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



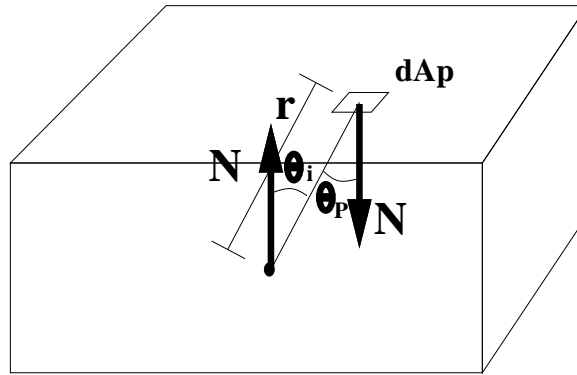
- Now determine which squares patch  $j$  projects onto

#### Form Factors: Delta Form Factors

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

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

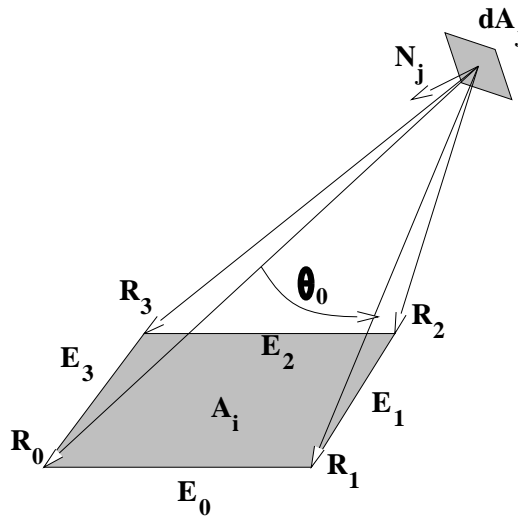




- Approximate  $dF_{dj,i}$  by summing delta form factors
- If distance from  $i$  to  $j$  is large, this is good approximation for  $F_{j,i}$ .

#### 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  $j$
- $\Gamma_i$  is the vector in direction  $R_{i+1} \times R_i$  with length equal to  $\theta_i$  (in radians).
- Doesn't account for occlusion

### 24.3 Progressive Refinement

- Recall our equation says how much radiosity 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, but...
- Form factors 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

## 24.4 Meshing in Radiosity

### Issues

- Initial Mesh Size?
- T-vertices, etc?
- Shadow Boundaries
- Soft Shadows

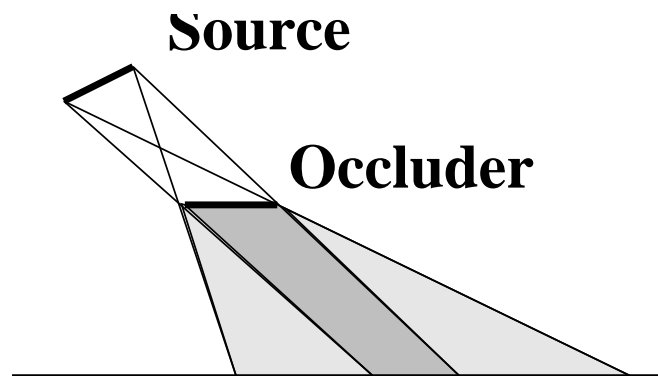
### Two Techniques:

- Split along shadow boundaries
- Adaptive Subdivision

### Split Along Shadow Boundaries – 2D

3 regions:

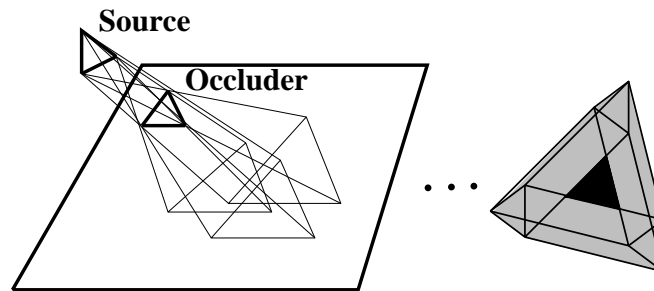
- Lit
- Unlit
- Linear



## Split Along Shadow Boundaries – 3D

Lots of regions:

- Lit
- unlit
- Linear
- Quadratic



## Adaptive Subdivision

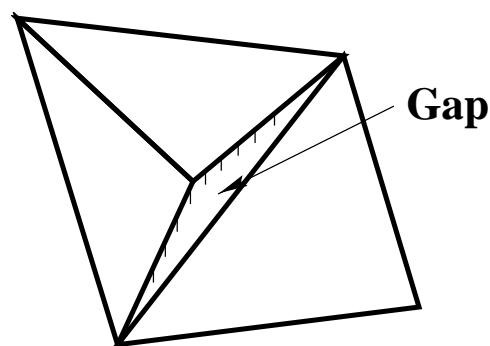
- Make initial Meshing Guess
- Point sample a patch
- Average and check variance
  - Low  $\Rightarrow$  Good estimate
  - High  $\Rightarrow$  Poor estimate
- Subdivide and repeat

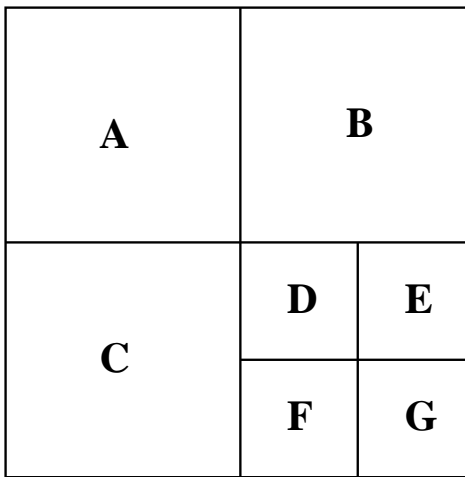
## Problems with Adaptive Subdivision

- Radiosity has  $O(n^2)$  run time  
Subdivision increases  $n$
- Ideas:
  - Shoot from big patches
  - Receive at small patches

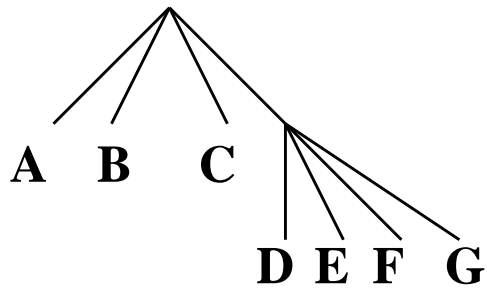
Sub problems:

- sub-patch to big-patch association
- T-vertices



**Quad Tree**

- Store as



- Traverse up to find parent
- Traverse up-down to find neighbor
- Anchoring

**24.5 Stochastic Methods — Radiosity Without Form Factors****Form factor sampling**

- Rather than compute form factor, we can estimate it by *sampling*
- Idea: cast  $N_i$  particles/rays from patch  $i$   
Let  $N_{ij}$  be the number of particles that reach patch  $j$ . Then

$$\frac{N_{ij}}{N_i} \approx F_{ij}$$

- Particles/rays should be cosine-distributed over patch  $i$ .

## Stochastic Radiosity

Consider an iterative approach

- Let  $\Delta P_i^{(k)}$  be unshot radiosity at step  $k$
- Select a source patch  $i$  with probability

$$p_i = \frac{\Delta P_i^{(k)}}{\Delta P_T^{(k)}}$$

where  $\Delta P_T^{(k)} = \sum_i \Delta P_i^{(k)}$

- Select destination patch  $j$  with probability  $F_{ij}$  by “casting stochastic ray”
- Probability of picking patch pair  $(i, j)$  is

$$p_{ij} = p_i F_{ij} = \Delta P_i^{(k)} F_{ij} / \Delta P_T^{(k)}$$

- Estimate for  $\Delta P_i^{(k+1)}$  becomes

$$\begin{aligned} \Delta P_i^{(k+1)} &\approx \frac{1}{N} \sum_{s=1}^N \frac{\Delta P_{j_s}^{(k)} F_{j_s, l_s} \rho_{l_s} \delta_{l_s, i}}{\Delta P_{j_s}^{(k)} F_{j_s, l_s} / \Delta P_T^{(k)}} \\ &= \rho_i \Delta P_T^{(k)} \frac{N_i}{N} \end{aligned}$$

## Stochastic Radiosity: Code

```

Procedure Shoot(i)
For Ni times
  Sample random point x on Si
  Sample cosine-distribution direction T at x
  Determine nearest patch j in direction T from x
  dPj += 1/N * rho j * DPT
endfor

Call from

while (unshot > eps) do
  For each patch i
    Shoot(i)
  Update total power, set DPi to dPi, clear dPi
  Compute new DPT
end

```

(Readings: Based on exposition in Advanced Global Illumination, Dutre, Bala, and Bekaert.)

## **Evaluation**

- Much faster than form factor methods
- Noise, noise, noise  
Need to filter

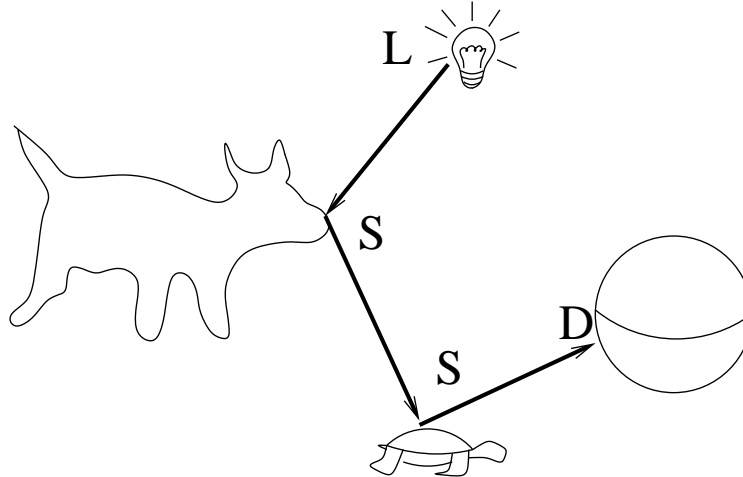




## 25 Photon Maps

### 25.1 Overview

- Radiosity too expensive, doesn't do caustics, etc.
- Photon mapping idea:
  - trace rays from lights, bounce off shiny surfaces,
  - store light information where light hits diffuse surfaces.
  - Then ray trace, and use stored light to achieve global illumination effects.
- General steps of photon mapping:
  - Emit photons  
Mostly distribution of  $n$  photons around light source
  - Trace photons  
Bounce off shiny surfaces, store at diffuse surfaces
  - Estimate irradiance
  - Ray trace, using estimated irradiance as needed
- LS+D: Idea based on bouncing light off shiny surfaces until it hits diffuse surface

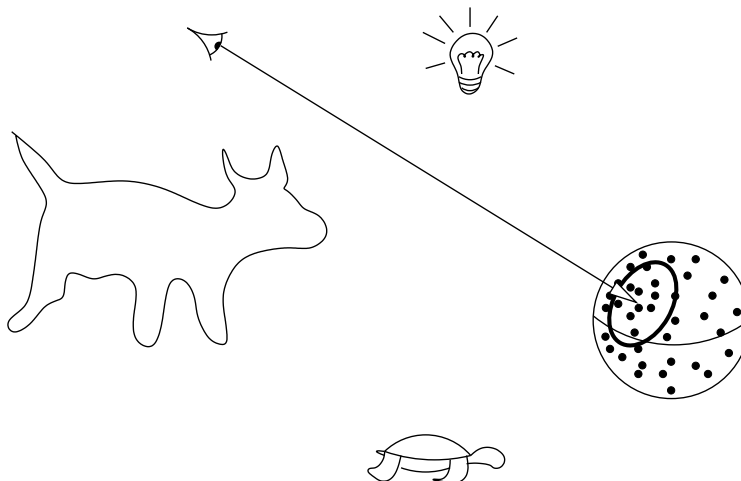


- $L(S|D)*D$ : bounce off both shiny and diffuse surfaces and store information at diffuse surfaces.
- “Shiny” surface: light leaves in a fixed direction (relative to incoming direction)  
Includes transparent (refractive) surfaces
- Similar to ray tracing, but cast rays from lights
- When light hits shiny surface, bounce to next surface  
Use Russian Roulette to send ray to next surface based on BRDF.

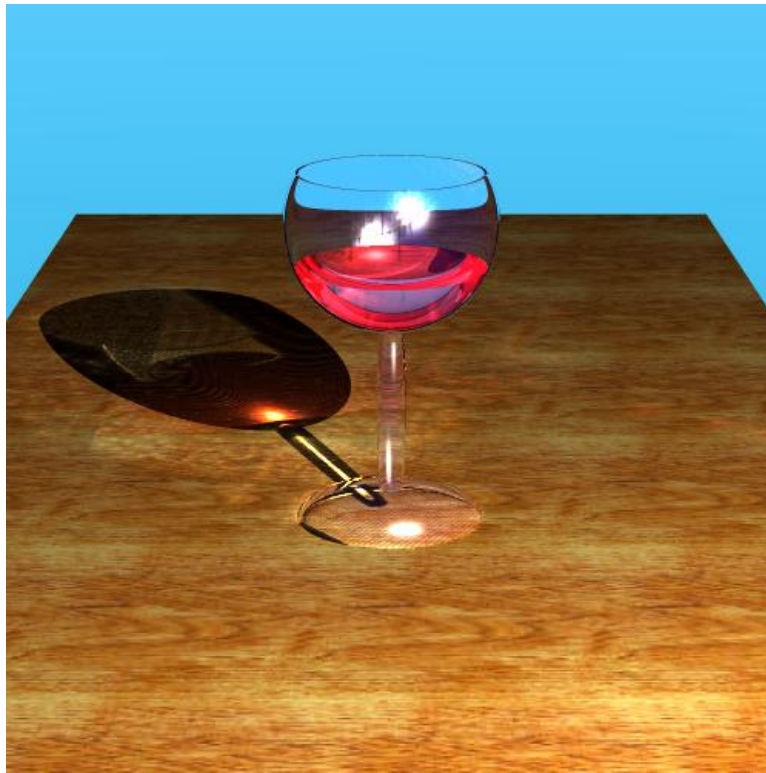
- When light hits diffuse surface, store surface point in data structure (kD tree, etc).  
Also store surface normal, intensity/power of light.

To estimate irradiance at a point,

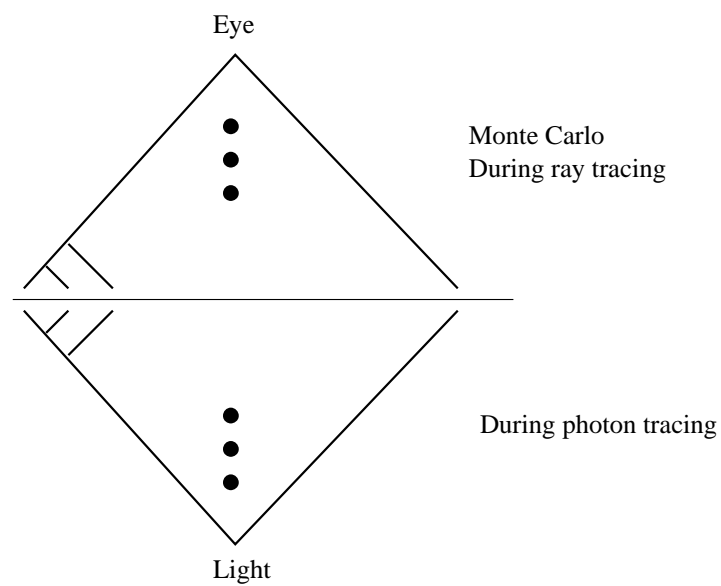
- Do a  $k$  nearest neighbour search to find closest stored photon information  
Usually try to limit search to a local region (sphere)

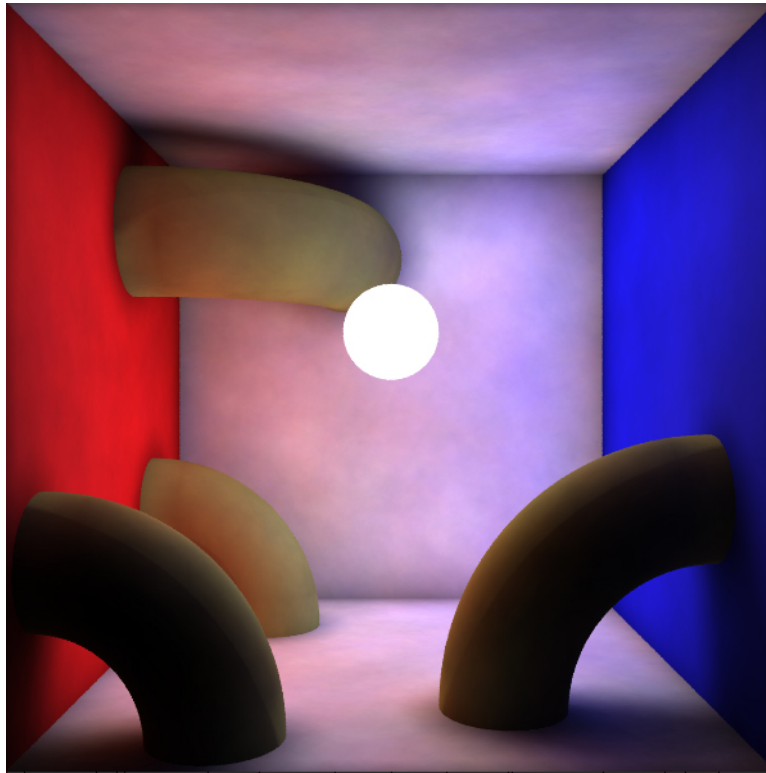


- Average to get irradiance for the point
- Since kD-tree is space (not surface) based, may pick up photons on nearby surface  
May cause noise in image
- When ray tracing, use radiance information in lighting equation
- To get caustics, use  
Direct light + mirror + refraction + ambient + irradiance est
- To use for global illumination, use  
Mirror + refraction + irradiance est  
but photon tracing is performed differently



- Need to work harder or smarter to get global illumination
- Harder: trace a LOT more rays from light
- Smarter: trace rays from light (Russian Roulette)  
and trace rays from eye (Monte Carlo)  
and meet in the middle



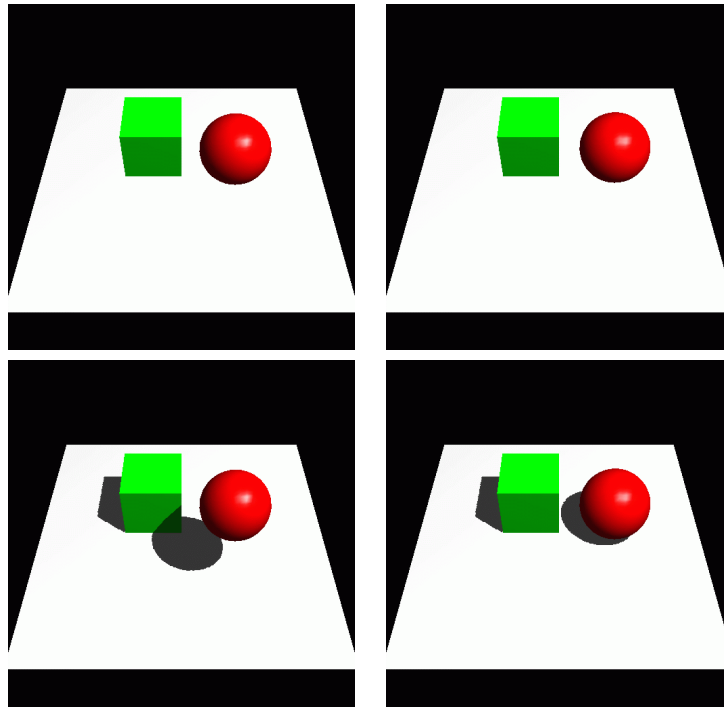


- Number of photons to cast from light
- Parameters in irradiance estimate  
value of  $k$ , radius of limiting volumes, etc.
- Monte Carlo vs Russian Roulette
- Noise, noise, noise

## 26 Shadows

### 26.1 Overview

- Shadows help viewer to locate objects



- Already seen shadows in ray tracer: how about interactive?
- Will consider three methods:  
Projective shadows, shadow maps, volume shadows

### 26.2 Projective Shadows

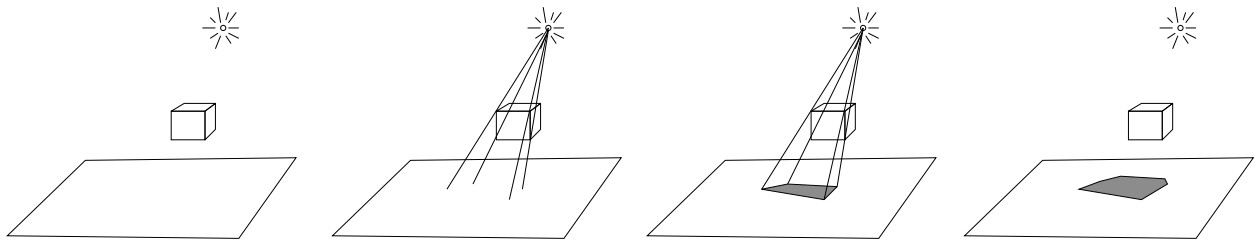
For drawing shadows on a plane.

After drawing scene,

- Draw ray from light through corners of polygon and intersect with plane.
- Draw projected polygon as a dark colour with alpha blending  
Depth test strictly less than!

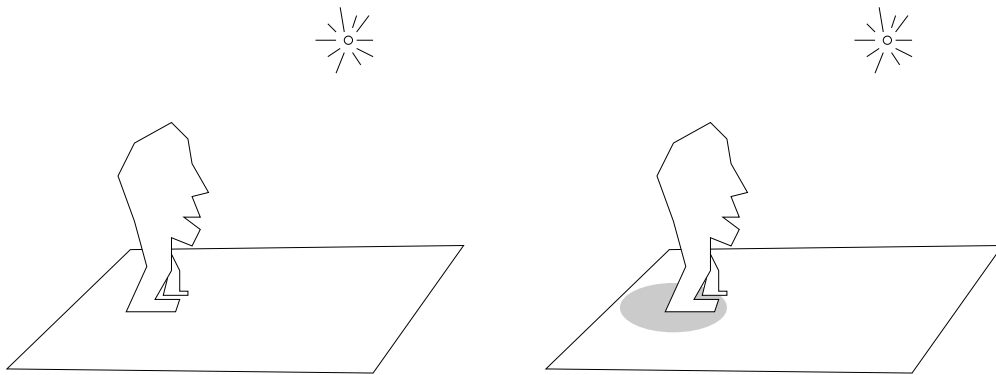
Advantages: fast, simple

Disadvantages: shadows only on a plane



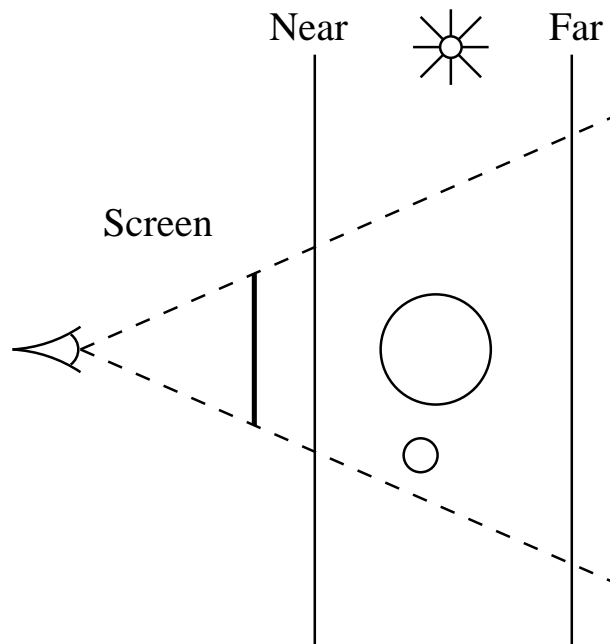
## Approximation to Projective Shadows

- Often simple ground plane, light
  - Idea: just draw an alpha-blended polygon under object as shadow
- Wrong, but...

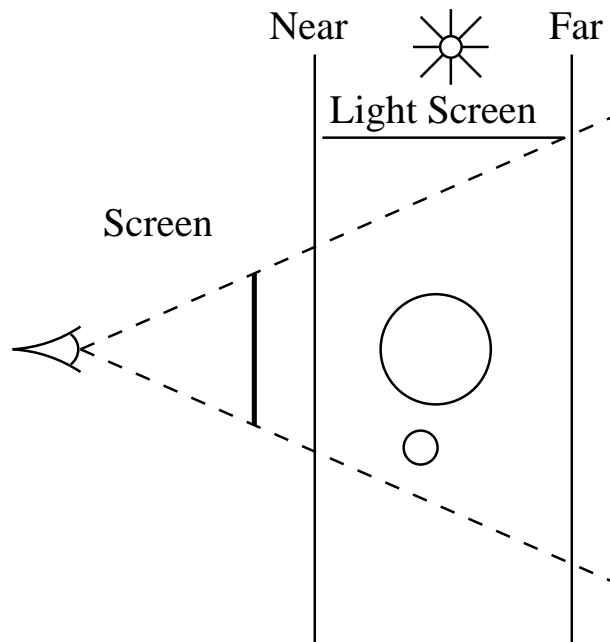


## 26.3 Shadow Maps

Given a scene...

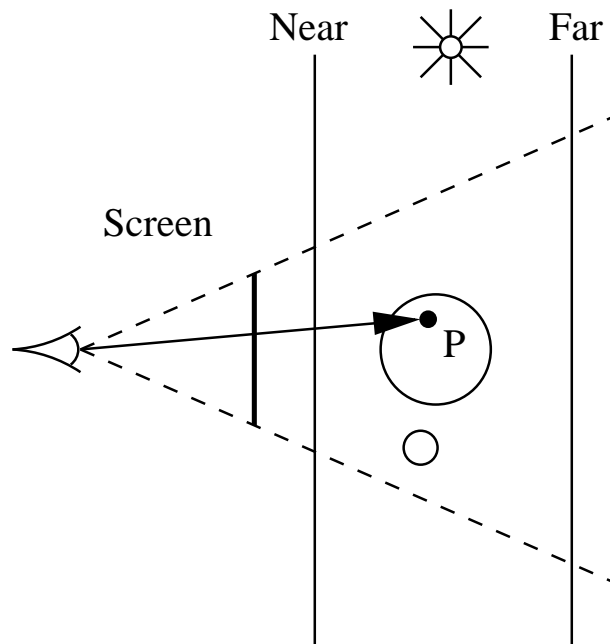


Render scene from *light's* viewpoint (disable colour write)



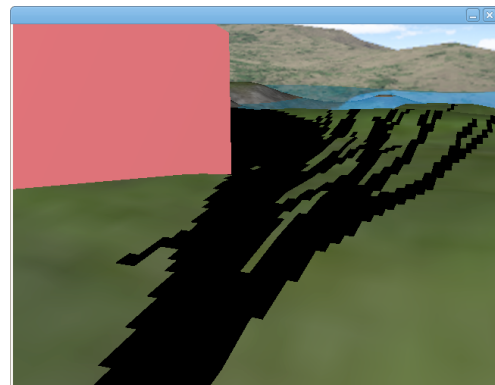
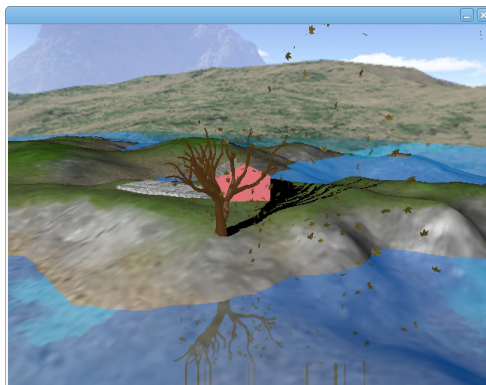
- Store the  $z$ -map (this is the shadow map)
- Possibly render “deeper”

Now render from eye's viewpoint



At point  $P$  that eye sees on surface,

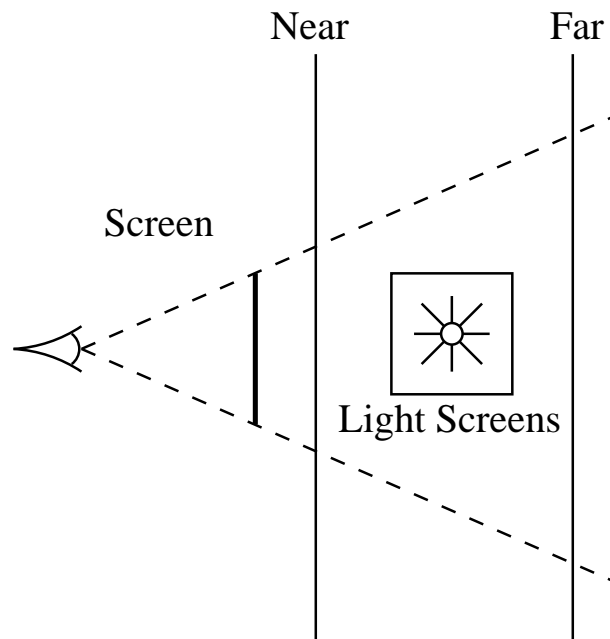
- Convert  $P$ 's coordinates to light's frame
- Look up distance from light using shadow map
- If distance in shadow map "equal" to distance from light to  $P$ , do lighting calculation. Otherwise in shadow.
- One (or more!) shadow map(s) per light
- Aliasing (lots of work on this)



- Depth test (lots of work on this)
- Needs hardware support



- Special purpose hardware (obsolete)
- Use fragment shader to do look up
- If you don't move lights or objects (just view), then you don't need to rerender shadow maps each frame.
- Works best with directional and spot lights...
- With point light source in viewing volume, have to put cube around light and render 6 shadow maps

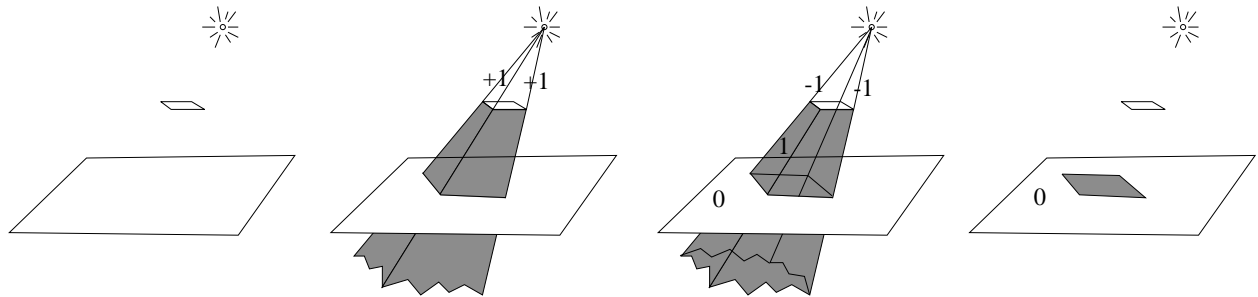


## 26.4 Shadow Volumes

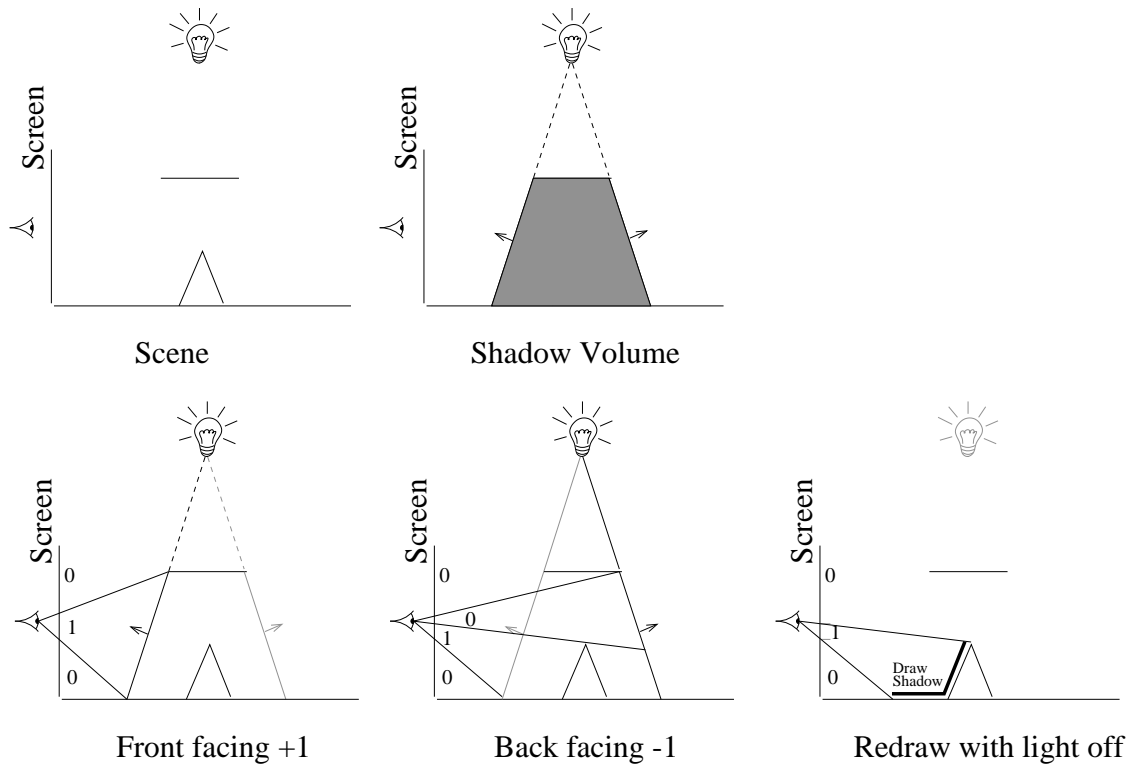
- Multiple drawing passes to render shadows
- Uses stencil buffer to determine where shadows are
 

Stencil buffer: store per pixel data, do operations and tests on stencil data, and use to affect drawing to frame buffer
- Idea (assume one light source):
  - For each point  $P$  on surface, consider shadow planes cast by silhouette edges
  - Count each front facing shadow plane as +1 and each rear facing plane as -1
  - If sum is  $> 0$  then  $P$  in shadow, otherwise light reaches  $P$ .

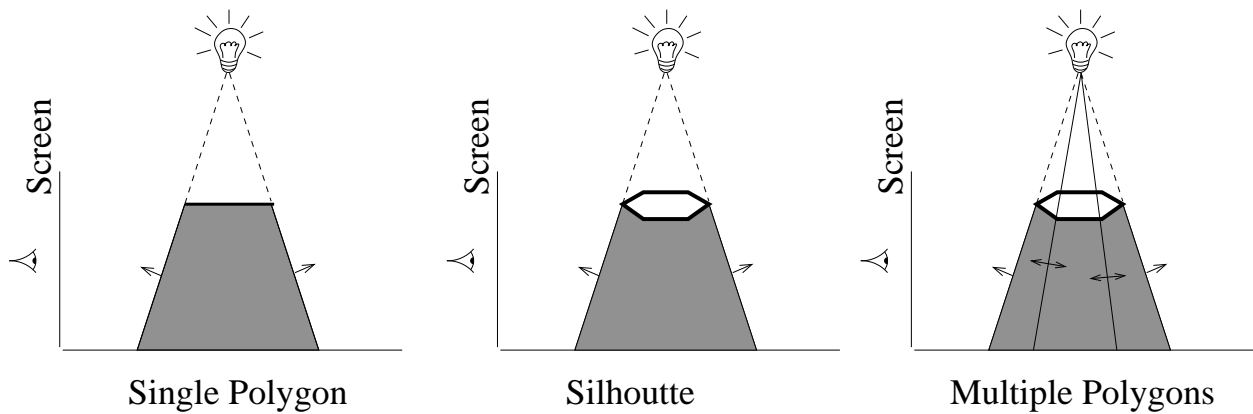
- We can use graphics hardware for shadow volumes as described on next slide.



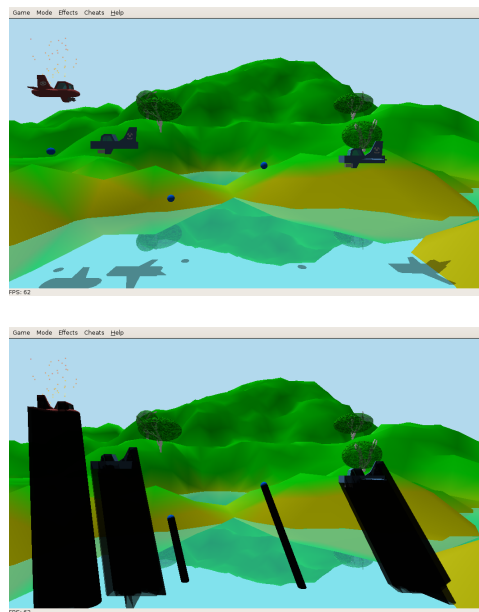
1. Draw scene normally
2. Draw in stencil buffer, front facing shadow polygons using prev depth buffer (do not update depth buffer)  
Increment stencil buffer on draws
3. Draw in stencil buffer, backfacing polygons  
Decrement stencil buffer on draws
4. Redraw scene with light off, but only update pixels that
  - have exact depth match
  - have non-zero stencil value



When more than one polygon...



- Draw shadow faces only for silhouette edges of polyhedra  
Problem: have to detect silhouette edges
- Draw shadow volume for each polygon facing away from light  
Shadow face for two polygons sharing an edge cancel  
Only silhouette edges have only one of front/back pair  
Downside: a lot more drawing than needed



- Lots of drawing modes to get right
- Infinite polygons?  
OpenGL tweaks can give you this
- Light map may help

- Problems if viewpoint inside shadow volume
- Need to repeat for each light

(Readings: McConnell, Chapter 7.)

## 27 Modeling of Various Things

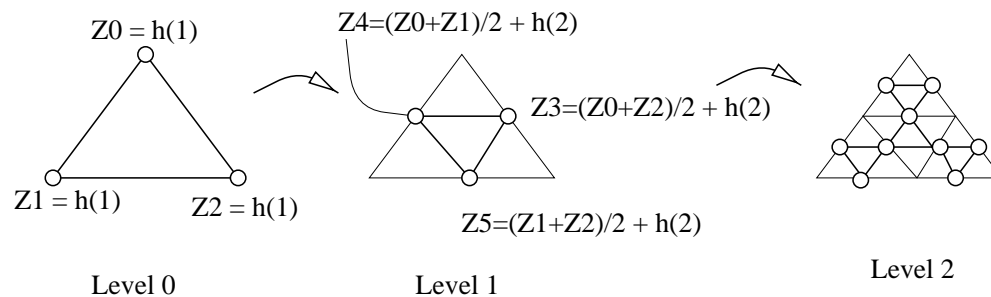
### 27.1 Fractal Mountains

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



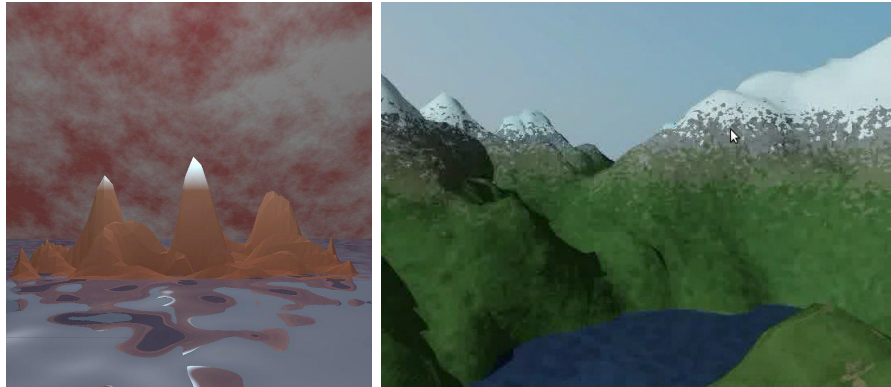
3. Details

- 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: McConnell, 11.3; Red book, 9.5; White book, 20.3;  
References; Fournier, Fussell, Carpenter, Computer Rendering of Stochastic Models, GIP, 1982  
)

## 27.2 L-system Plants

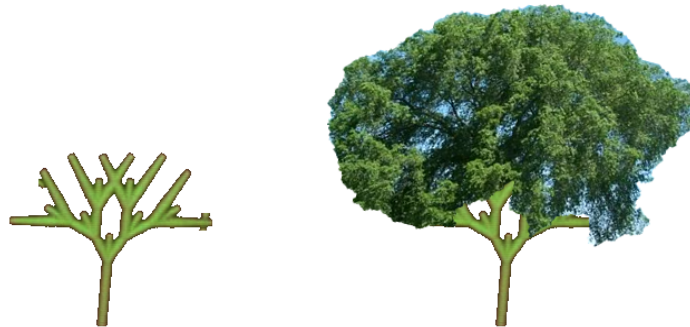
1. L-systems: Context free plants
2. Idea: Model growth/structure of plant as CFG
3. CFG: Symbols, rules
  - $A, B, C, D$
  - $A \rightarrow CBA, B \rightarrow BD$
4. Start symbol  $A$

$$A \rightarrow CBA \rightarrow CBDCBA \rightarrow CBDDCBDCBA$$

5. Plants need more complex L-systems/geometry
  - Thickness
  - Semi-random, 3D branching
  - Leaf model
6. Top three things to make it look good:
  - Texture mapping,
  - Texture mapping,
  - Texture mapping
7. Plants don't grow like this
  - Grow from top up, outside out
  - Respond to environment (context sensitive)
  - Interact with each other

## Trees

- L-systems good for trunk branching structure, harder to use for leaf ball.
- Idea: model simple leaf ball and put on top of L-system trunk

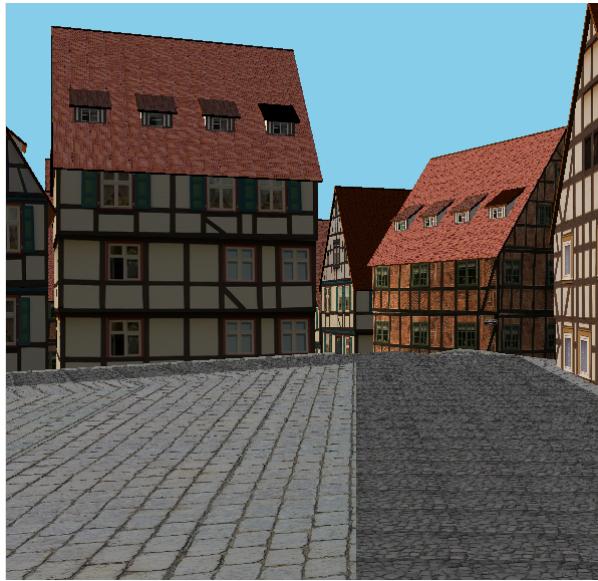


(Readings: McConnell, 11.2)

## 27.3 Buildings

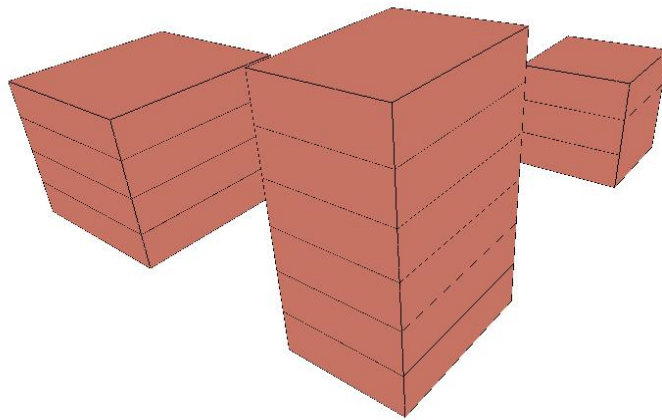
### Context “Free” Buildings

- In games, want to model cities
- Could use texture mapped facades of streetscape
- Idea: Each building is a texture mapped box
- Want a simple method to generate varied geometry cheaply  
Idea: Context free grammars



- Grammar:

1.  $B \rightarrow GIT$
2.  $I \rightarrow FI$
3.  $I \rightarrow F$



Each symbol keeps track of "box"

Each production has minimum width/height/depth

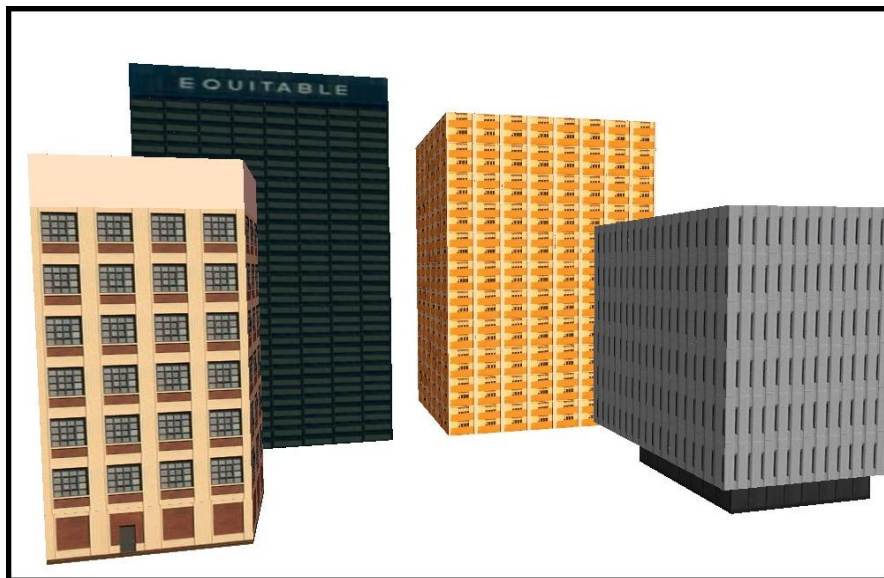
- Need to control growth of buildings  
Neighboring buildings better not intersect!
- Split Grammars:  
A symbol represents a volume  
A production splits this volume into non-overlapping subvolumes
- Texture map terminal symbols



- Productions may need conditions

1.  $B \rightarrow GIT$
2.  $I \rightarrow FI$
3.  $I \rightarrow F$
4.  $F \rightarrow CWCWCWCW$
5.  $W \rightarrow wW$  (in  $x$ )
6.  $W \rightarrow wW$  (in  $y$ )
7.  $W \rightarrow w$
8.  $G \rightarrow cScScScS$
9.  $S \rightarrow DU$  (in  $x$ ),  $S \rightarrow DU$  (in  $y$ )
10.  $U \rightarrow sU$  (in  $x$ ),  $U \rightarrow sU$  (in  $y$ )
11.  $U \rightarrow s$ ,  $S \rightarrow s$

### Examples



- Each symbol represents a box
  - In split grammars, a production subdivides a box into smaller boxes that fill original box
- Each symbol needs to keep track of
  - Min corner  $(x,y,z)$  and size  $(w,d,h)$
  - Facing (NSEW)

- Texture map
- Each production needs to examine symbol and set attributes in production
  - Minimum width/depth/height to apply
  - Symbols on right need to have x,y,z and w,d,h etc set

In previous example, the production

$$\mathbf{B} \rightarrow \mathbf{GIT}$$

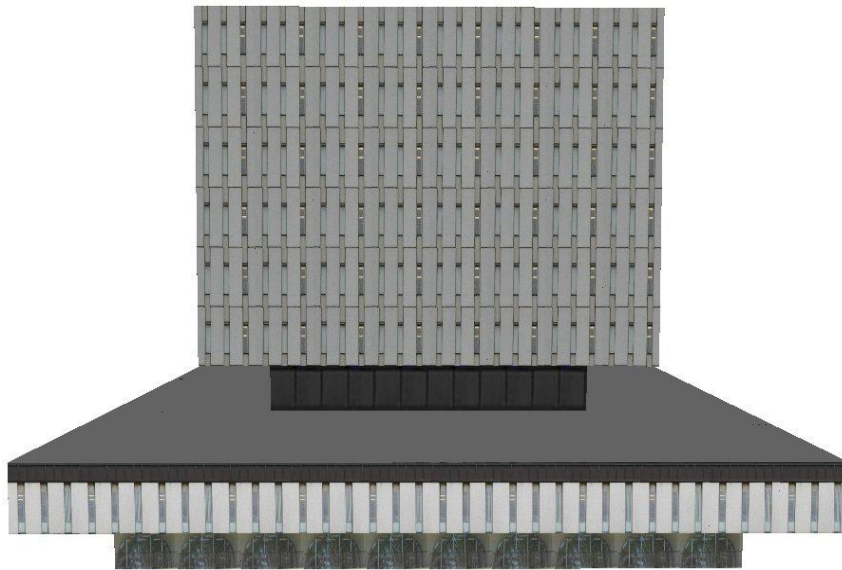
is really

$${}^{w \geq 3} {}^{d \geq 3} {}^{h \geq 3} \mathbf{B}_{xyz}^{wdh} \rightarrow \mathbf{G}_{x,y,z}^{w,d,1} \mathbf{I}_{x,y,z+1}^{w,d,h-2} \mathbf{T}_{x,y,z+h-1}^{w,d,1}$$

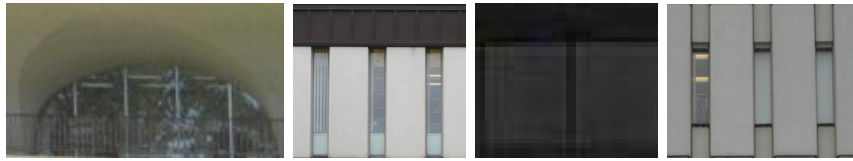
where

- Condition on left must be met to apply production
- Superscript/subscript on right indicates how xyz,wdh on left are modified to locate each subbox

**Dana Porter Example**

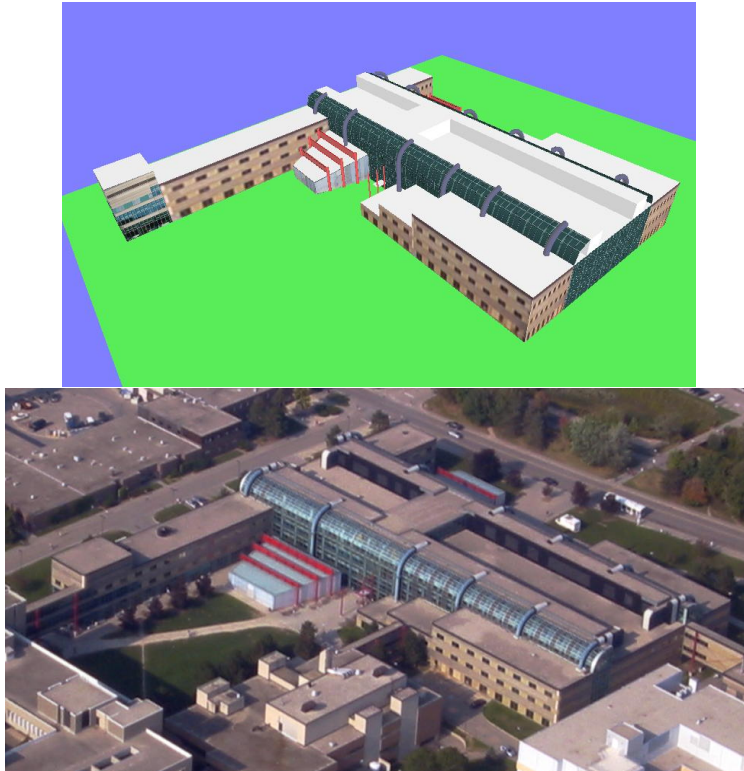


**B → F F G F I**  
**I → F I**  
**I → F**  
**F → C W C W C W C W**  
**W → w W**  
**W → w W**  
**W → w**



- Grammar is simple: Different textures on different floors handled on first production.

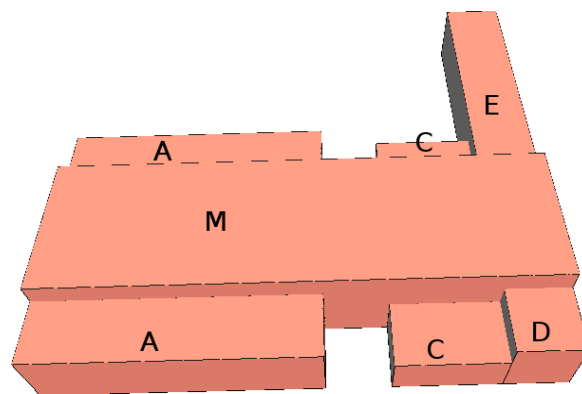
### Davis Centre Example



Aerial photo of Davis Centre Copyright Jeff Orchard. Reprinted here with his kind permission.

**B** → **M A A C C D E p p p p p q q q q q d d**  
**A** → **a b c**  
**E** → **e f**  
**M** → **I T**  
**I** → **F I**  
**I** → **F**  
**F** → **H W H W H W H W**  
**W** → **w W**  
**W** → **w W**  
**W** → **w**  
**T** → **s t u v x y m m n m**  
**t** → **t t, x** → **x x**

- First production separates into main building (M), wings (ACDE)



Also pipes (pq) and doors (D), but not shown in figure

- Lecture halls (C) refined as geometry, not grammar
- Most of grammar dedicated to M
- Texture maps were so-so  
Need good textures for good results

- To generate a cityscape, need
  - Several types of buildings  
Different geometries, choices of textures
  - Layout generation

- Randomness in production selection
- To render modern skyscrapers (e.g., glass buildings),  
renderer needs to model reflections
- To generate very modern buildings (Gehry, curved surfaces) need...

(Readings:

- Peter Wonka’s 2008 SIGGRAPH Course provides an overview of the work.  
<http://portal.acm.org/citation.cfm?id=1185713>
- Instant Architecture. Peter Wonka, Michael Wimmer, Francois Sillion, and William Ribarsky ACM Transactions on Graphics. volume 22. number 3. pages 669 - 677. July 2003. Proceedings of SIGGRAPH 2003.
- Procedural Modeling of Buildings. Pascal Mueller, Peter Wonka, Simon Haegler, Andreas Ulmer, Luc Van Gool. ACM Transactions on Graphics. volume 25. number 3. pages 614-623. 2006. Proceedings of SIGGRAPH 2006.

This one has the best description of their split grammars.

- Image-based Procedural Modeling of Facades. Pascal Mueller, Gang Zeng, Peter Wonka, Luc Van Gool ACM Transactions on Graphics, volume 26, number 3, article number 85. pages 1-9. 2007. Proceedings of SIGGRAPH 2007.

This paper describes how to get better textures from a picture of the side of a building, and how to automatically extrude it to get a 3D face.

)

## 27.4 Particle Systems

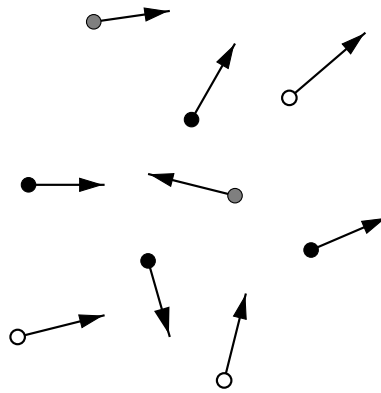
- How do we model “fuzzy” objects? Objects with
  - “soft” boundary
  - changing boundary
  - chaotic behavior

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 collection of particles  
Used to model “Genesis” effect in The Wrath of Kahn

- 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



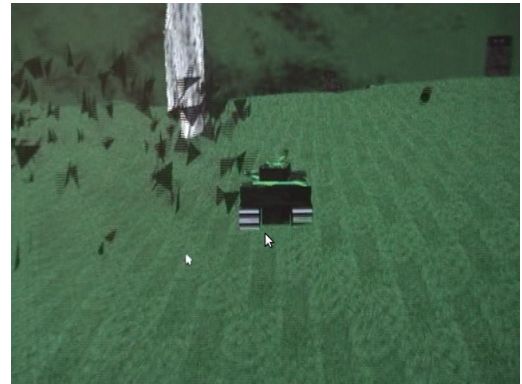
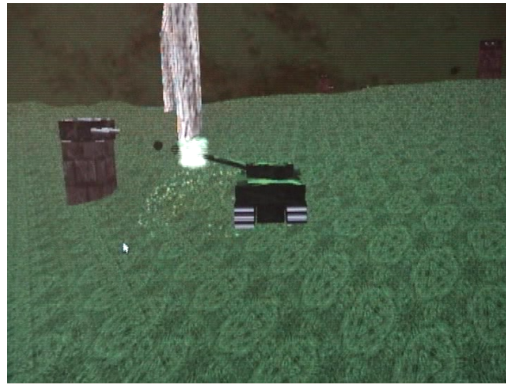
### Animating Particles Systems

- 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

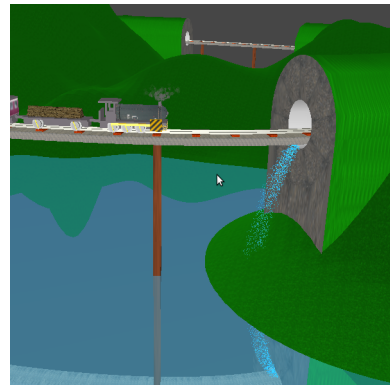
### Particle System Details

- 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



Examples





## 28 Polyhedral Data Structures

### 28.1 Storage Models

#### Polyhedra

- How do we store a polyhedron in memory?

Must consider the following:

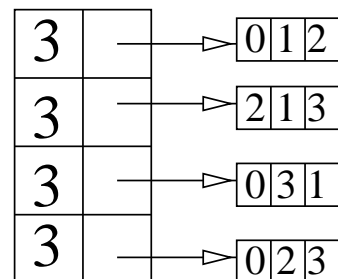
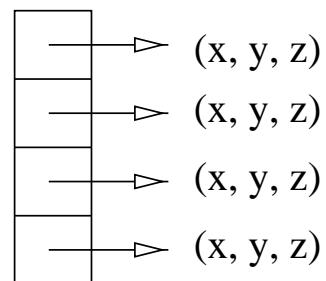
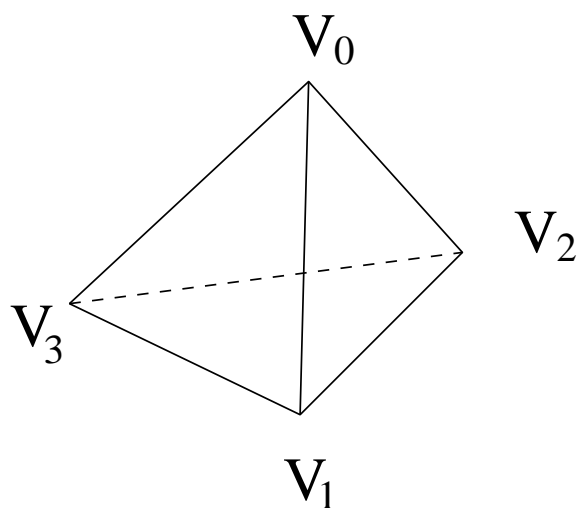
- Memory
- Efficiency
- Operations

- 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



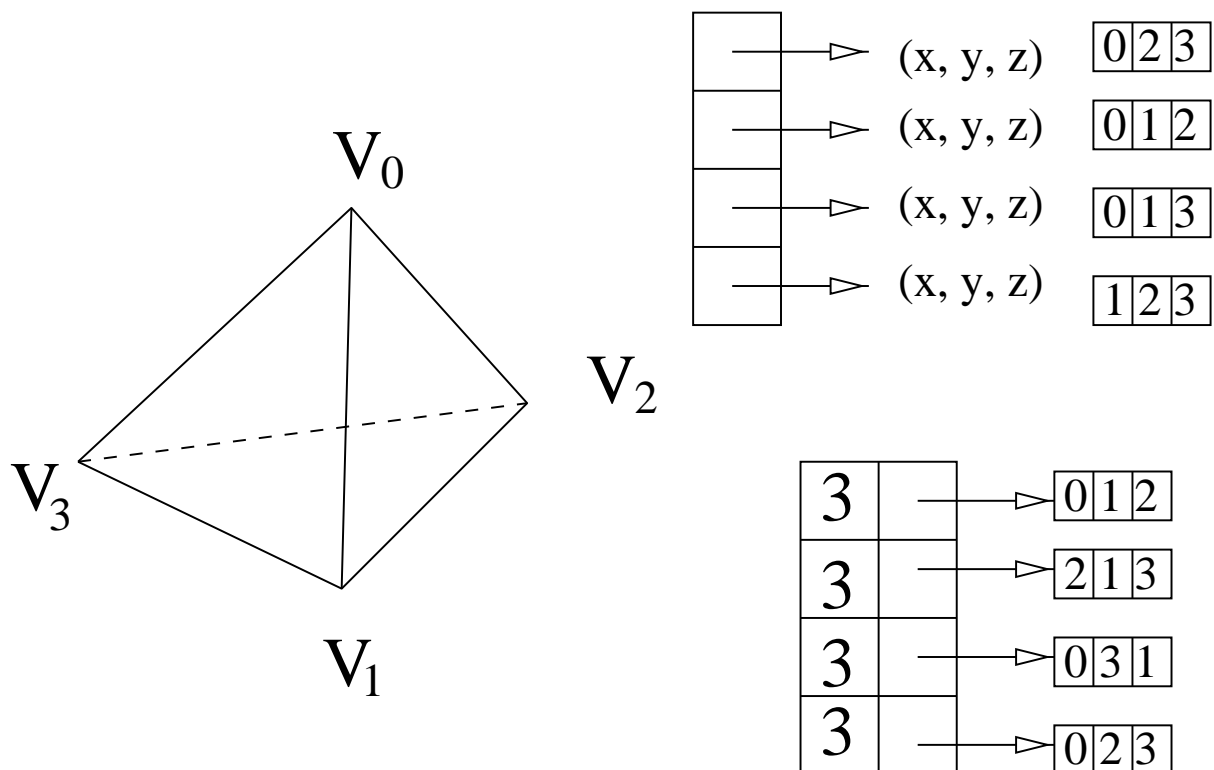
Compact, simple to get face information

- Advantages
  - Space efficient
  - Easy to generate, parse, build data structure
- Disadvantages
  - Vertex neighbours?
  - Modify the data structure?

Example: What if we want to know faces surround a vertex?

Radiosity might want this

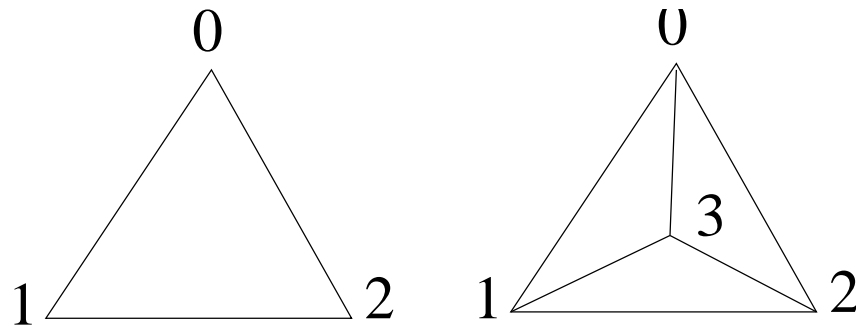
- Solution: For each point, have a list of pointers to surrounding faces.
  - Can use same ASCII input format
  - Build lists when reading data



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



$$[0 \ 1 \ 2] \rightarrow \begin{matrix} [0 \ 1 \ 3] \\ [1 \ 2 \ 3] \\ [2 \ 0 \ 3] \end{matrix}$$

- Delete is hard
- $O(n)$  or leaves holes in our list

(Readings: Hearn and Baker in OpenGL: 8-1 (not very useful). Watt: 2.1.)

## 28.2 Euler's Formula

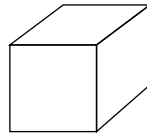
- Closed polyhedron:

$$V - E + F - H = 2(C - G)$$

- 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)

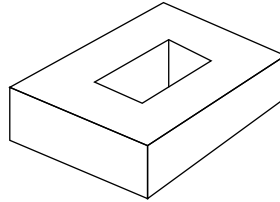
- Examples:

Cube



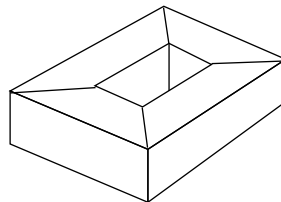
V=    E=    F=    G=    C=    H=

Torus 1



V=    E=    F=    G=    C=    H=

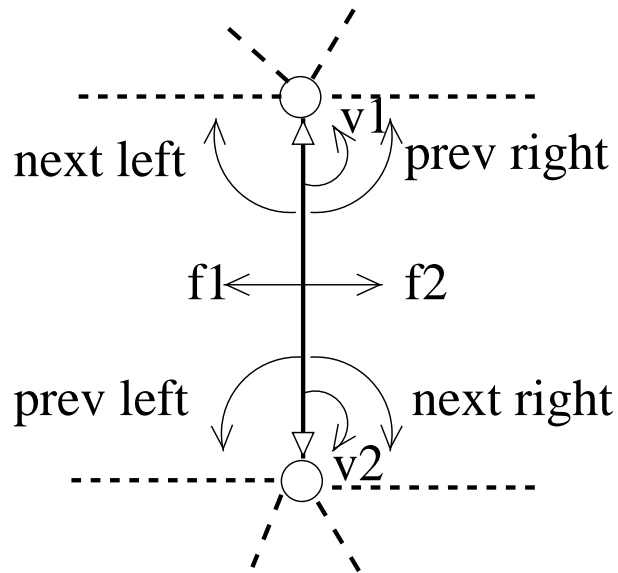
Torus 2



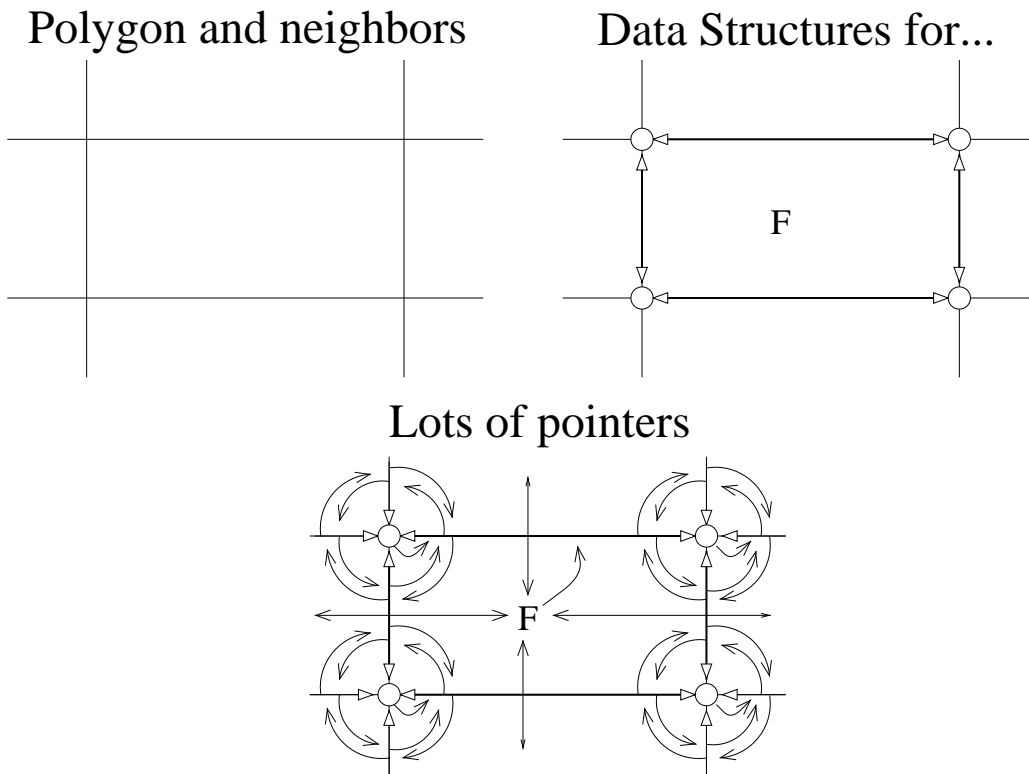
V=    E=    F=    G=    C=    H=

### 28.3 Winged Edge

- Operations:
  - Want to traverse
    - \* Neighbouring faces of a face/vertex
    - \* Neighbouring 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 geometry
- Edge is the primary data structure

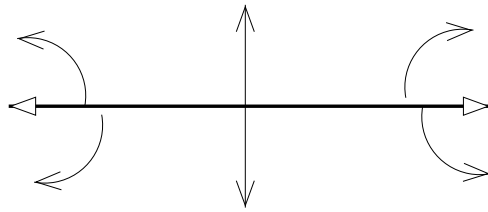


- Pictorially:

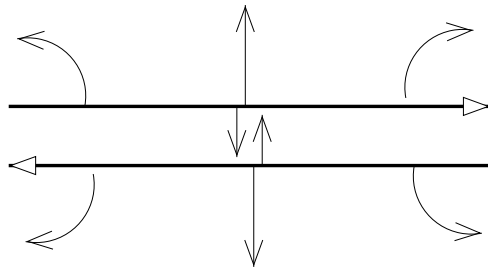


### Variations

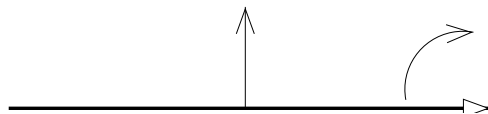
- Some operations simplified by splitting edge into half-edges
- Uses a lot of space. Time-space trade-off.



Winged Edge



Half-edge



Minimal Edge

### 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: Neighbouring faces of f

```

struct face* f;
struct he* e;
struct he* s;
struct face* 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);
}

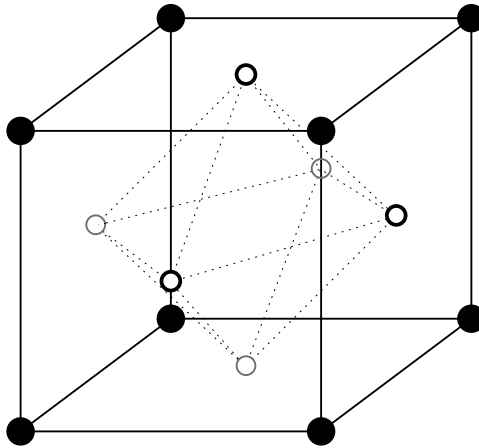
```

- Manifolds with Boundary

- Half-edge allows representation of manifold w/boundary
- Set sym pointer to NULL
- Allows for lone face
- Makes some constructions easier

- Duality

- Faces and vertices are dual
- Representations and rôles 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 neighbours
- + 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?

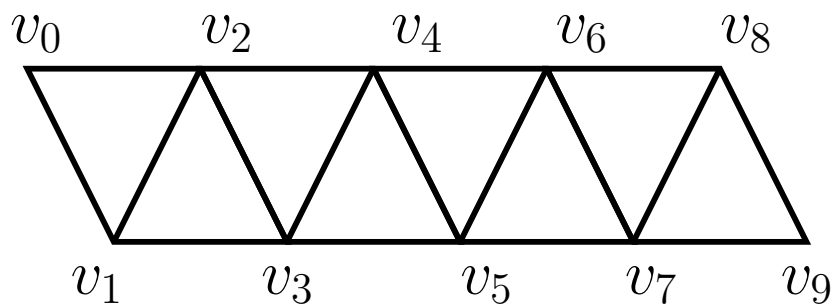


## 28.4 Mesh Compression

- How to store large polyhedron?  
Digital Michael Angelo: Statue of David: 32GB
- Have to store geometry (vertices) and connectivity (faces)
- Base-line for connectivity: ray tracer format
  - $v$  vertices
  - $\log v$  to store index
  - $3 \log v$  to store triangle
  - Twice as many triangles as vertices (roughly)
  - $6 \log v$  bits per vertex (bpv)

Triangle strips

- Use index ordering to reduce repetitions of indexes



List vertices in strip as 0, 1, 2, 3, 4, ...

- Most vertices listed once per three triangles
- Also Fans
- Variations can reduce storage to 11 bpv
- Better schemes can reduce this to around 2 bpv

Geometry Compression

- Geometry data is much larger than connectivity data  
3 floats for position, 3 floats for normals
- Even single precision floats more than adequate for viewing
- Simplest variation: reduce bits of precision in position, normals

- Neighboring vertices have many bits of coordinates the same  
Use small precision deltas to represent one vertex as offset of another.

### Progressive Compression

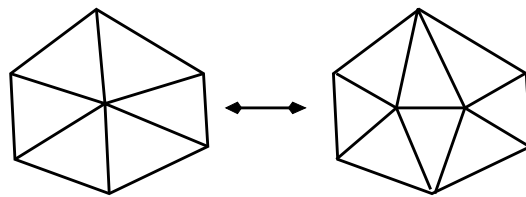
- Some applications benefit from progressive storage:

$$M_0 \subset M_1 \subset M_2 \dots$$

- Web transmission (transmit from coarser to finer; allows aborting transmission based on coarse version)
- Multiple viewing resolutions

### Edge Collapse/Vertex Split

- One approach: start with fine mesh, do edge collapses



This gives sequence of meshes based on collapsed edges

When going from coarse to fine, can animate vertex split to reduce artifacts

- Not really a compression scheme

(Readings:

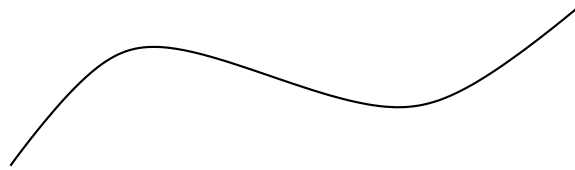
- Technologies for 3D mesh compression: A survey, Jinliang Peng, Chang-Su Kim, C.-C. Jay Kuo, J Vis Commun Image R, 16(2005) 688–733
- Progressive Meshes, Hugues Hoppe, SIGGRAPH 1996, 99–108.

)

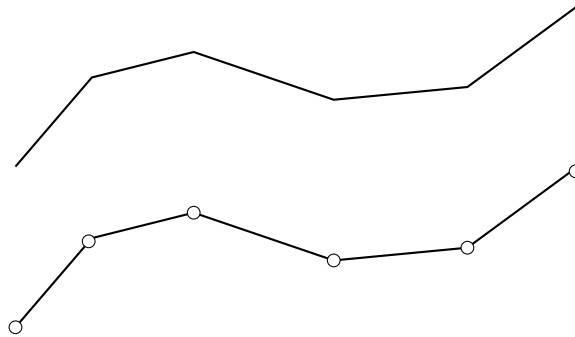
## 29 Splines

### 29.1 Constructing Curve Segments

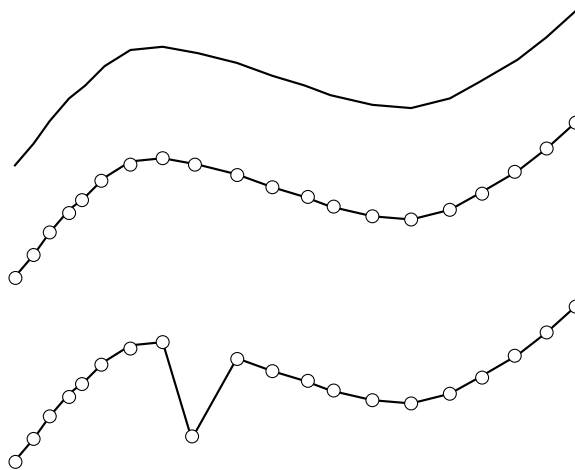
(Readings: McConnell: Chapter 4.)



Could sample and represent as polyline, but...  
Looks bad if undersampled



Dense sampling looks fine



Looks wrong if we edit it.

Want: Editable representation with “nice” properties.

Polynomials

**Monomials**

- Familiar form

$$f(x) = \sum_{i=0}^n c_i x^i$$

- Parametric

$$P_0 + \sum_{i=1}^n \vec{v}_i t^i$$

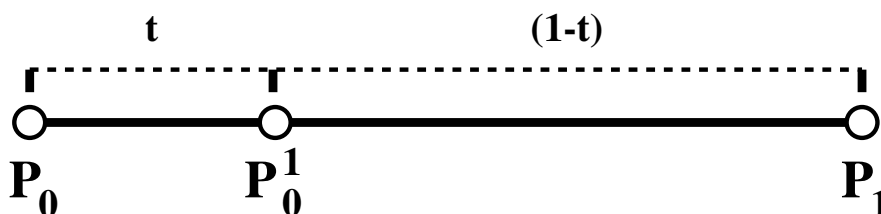
$\vec{v}_i$  proportional to  $f^{(i)}(0)$

- Not useful as shape handles  
Prefer to manipulate points

**Linear Curve Segments****Linear blend:**

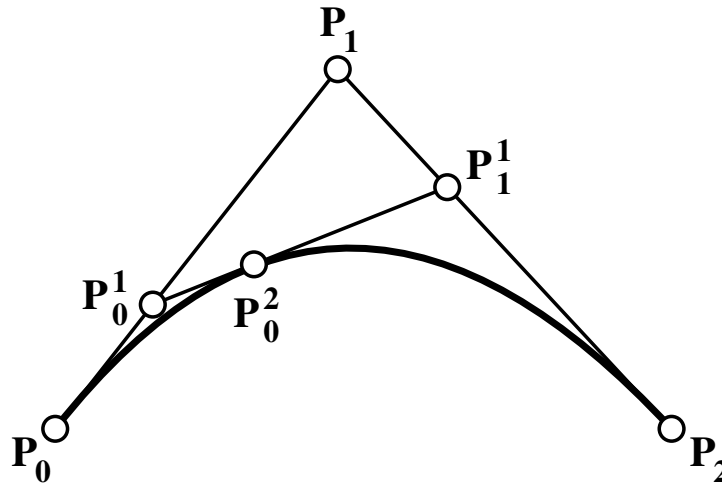
- Line segment from an affine combination of points

$$P_0^1(t) = (1-t)P_0 + tP_1$$

**Quadratic blend:**

- Quadratic segment from an affine combination of line segments

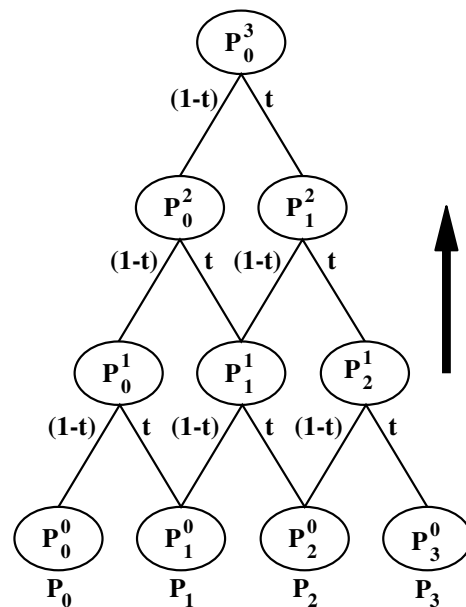
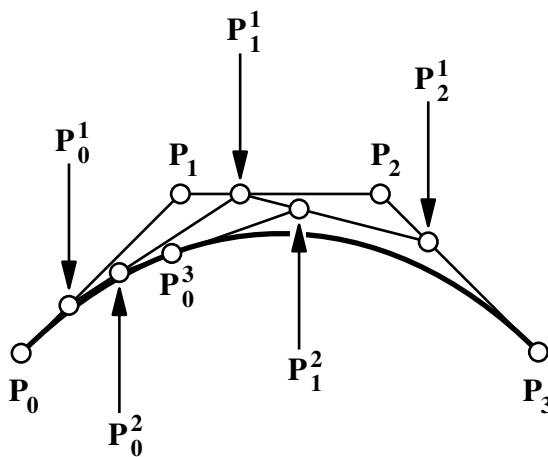
$$\begin{aligned} P_0^1(t) &= (1-t)P_0 + tP_1 \\ P_1^1(t) &= (1-t)P_1 + tP_2 \\ P_0^2(t) &= (1-t)P_0^1(t) + tP_1^1(t) \end{aligned}$$



## Cubic blend:

- Cubic segment from an affine combination of quadratic segments

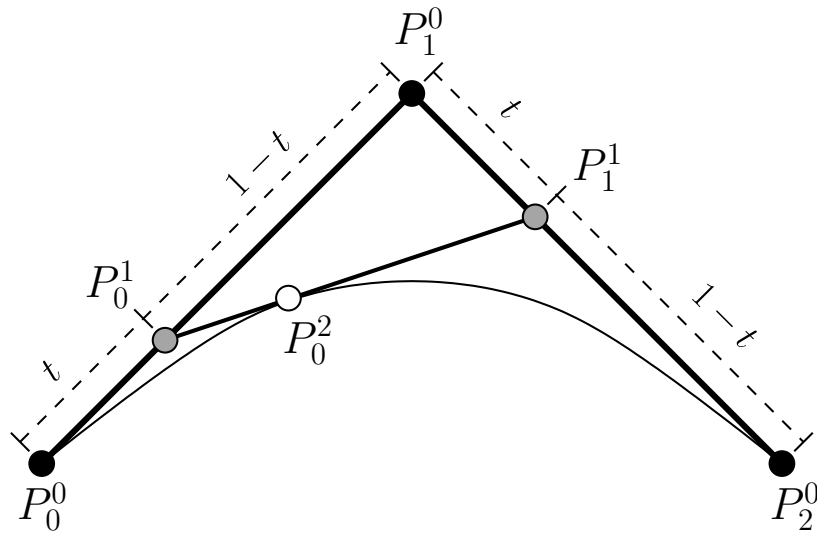
$$\begin{aligned}
 P_0^1(t) &= (1-t)P_0 + tP_1 \\
 P_1^1(t) &= (1-t)P_1 + tP_2 \\
 P_0^2(t) &= (1-t)P_0^1(t) + tP_1^1(t) \\
 P_1^2(t) &= (1-t)P_1^1(t) + tP_2^1(t) \\
 P_0^3(t) &= (1-t)P_0^2(t) + tP_1^2(t)
 \end{aligned}$$



- The pattern should be evident for higher degrees.

### Geometric view (de Casteljau 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*

$$\begin{aligned}
 P_0^0(t) &= 1 \cdot P_0 \\
 P_0^1(t) &= (1-t)P_0 + tP_1 \\
 P_0^2(t) &= (1-t)^2P_0 + 2(1-t)tP_1 + t^2P_2 \\
 P_0^3(t) &= (1-t)^3P_0 + 3(1-t)^2tP_1 + 3(1-t)t^2P_2 + t^3P_3
 \end{aligned}$$

$$P_0^n(t) = \sum_{i=0}^n P_i B_i^n(t)$$

$$\text{where } B_i^n(t) = \frac{n!}{(n-i)!i!} (1-t)^{n-i} t^i = \binom{n}{i} (1-t)^{n-i} t^i$$

- The Bernstein polynomials of degree  $n$  form a basis for the space of all degree- $n$  polynomials

## 29.2 Bernstein Polynomials

### Bernstein Polynomial Properties

**Partition of Unity:**  $\sum_{i=0}^n B_i^n(t) = 1$

Proof:

$$\begin{aligned} 1 &= (t + (1 - t))^n \\ &= \sum_{i=0}^n \binom{n}{i} (1 - t)^{n-i} t^i \\ &= \sum_{i=0}^n B_i^n(t). \end{aligned}$$

**Non-negativity:**  $B_i^n(t) \geq 0$ , for  $t \in [0, 1]$ .

Proof:

$$\begin{aligned} \binom{n}{i} &> 0 \\ t &\geq 0 \text{ for } 0 \leq t \leq 1 \\ (1 - t) &\geq 0 \text{ for } 0 \leq t \leq 1 \end{aligned}$$

Many other nice properties...

## 29.3 Bézier Splines

### Bezier Curves

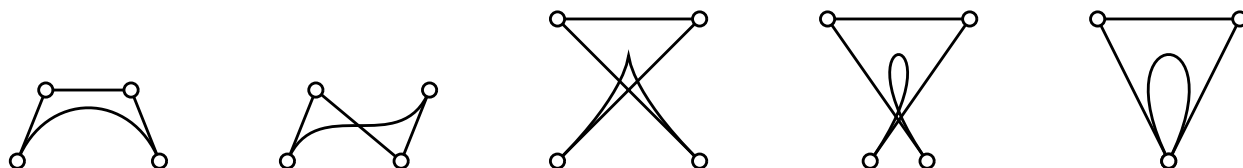
**Definition:**

A degree  $n$  (order  $n + 1$ ) **Bézier curve segment** is

$$P(t) = \sum_{i=0}^n P_i B_i^n(t)$$

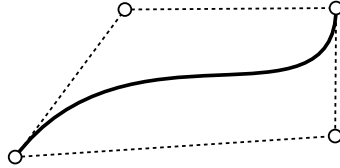
where the  $P_i$  are  $k$ -dimensional **control points**.

**Examples:**



**Convex Hull:**

$\sum_{i=0}^n B_i^n(t) = 1, B_i^n(t) \geq 0$  for  $t \in [0, 1]$   
 $\implies P(t)$  is a convex combination of the  $P_i$  for  $t \in [0, 1]$   
 $\implies P(t)$  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.

$$T \left( \sum_{i=0}^n P_i B_i^n(t) \right) = \sum_{i=0}^n T(P_i) B_i^n(t)$$

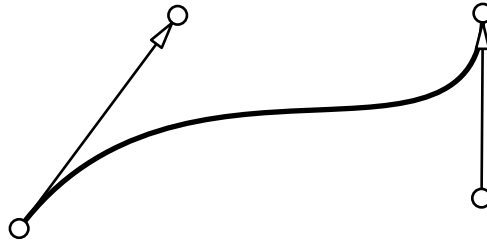
- This property does not hold for projective transformations!

**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  $t \in [0, 1]$   
 $\implies B_i^n(0) = 0$  if  $i \neq 0, B_i^n(1) = 0$  if  $i \neq n$   
 $\implies P(0) = P_0, P(1) = P_n$ .

**Derivatives:**

$\frac{d}{dt} B_i^n(t) = n(B_{i-1}^{n-1}(t) - B_i^{n-1}(t))$   
 $\implies P'(0) = n(P_1 - P_0), P'(1) = n(P_n - P_{n-1})$ .

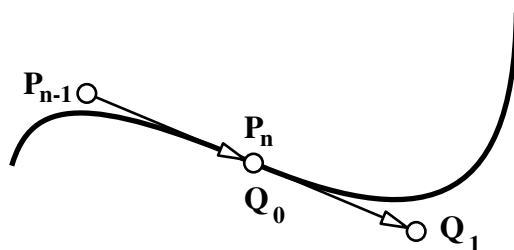


**Smoothly Joined Segments (C<sup>1</sup>): :**

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.

$$\begin{aligned} P_n &= Q_0 \\ (P_n - P_{n-1}) &= (Q_1 - Q_0) \end{aligned}$$

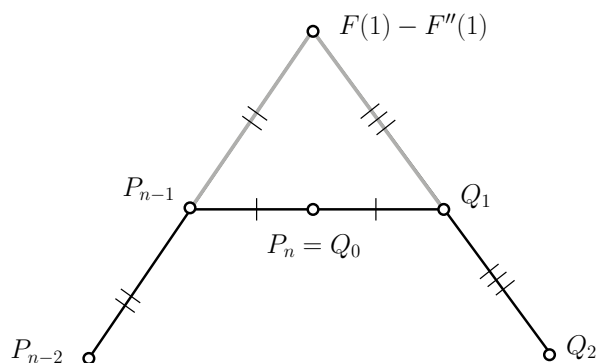




**Smoothly Joined Segments ( $G^1$ ):**

$$\begin{aligned} P_n &= Q_0 \\ (P_n - P_{n-1}) &= \beta(Q_1 - Q_0) \text{ for some } \beta > 0 \end{aligned}$$

$C^2$  continuity: **A-frame**



- $C^0$  :  $P_n = Q_0$
- $C^1$  :  $P_n - P_{n-1} = Q_1 - Q_0$
- $C^2$  :  $(P_n - P_{n-1}) - (P_{n-1} - P_{n-2}) = (Q_2 - Q_1) - (Q_1 - Q_0)$

**Recurrence, Subdivision:**

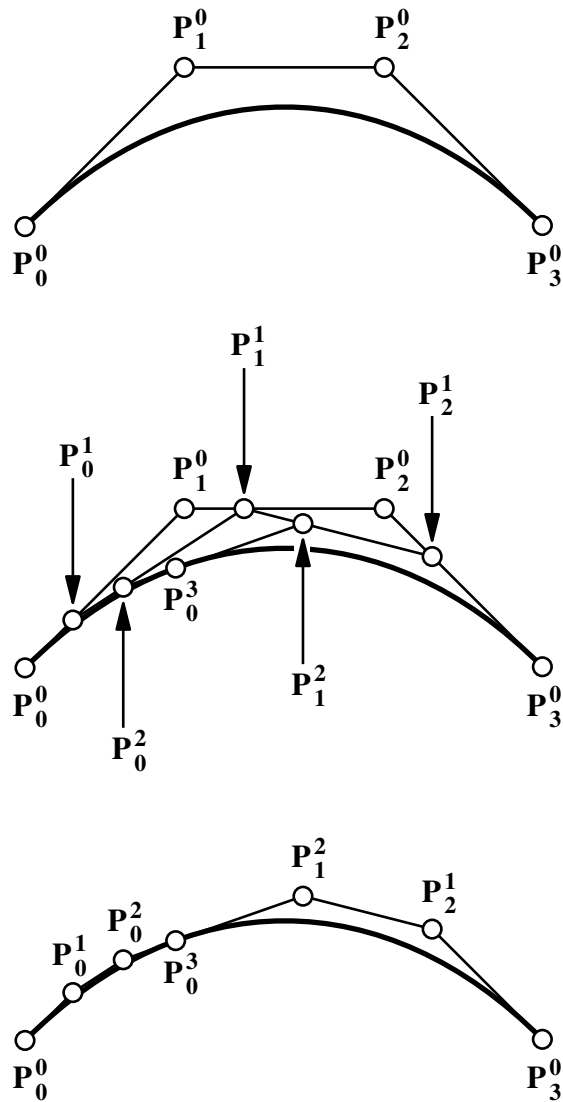
$$\begin{aligned} B_i^n(t) &= (1-t)B_i^{n-1} + tB_{i-1}^{n-1}(t) \\ \implies &\text{de Casteljau's algorithm:} \end{aligned}$$

$$\begin{aligned} P(t) &= P_0^n(t) \\ P_i^k(t) &= (1-t)P_i^{k-1}(t) + tP_{i+1}^{k-1} \\ P_i^0(t) &= P_i \end{aligned}$$

Use to evaluate point at  $t$ , or subdivide into two new curves:

- $P_0^0, P_0^1, \dots, P_0^n$  are the control points for the left half;

- $P_n^0, P_{n-1}^1, \dots, P_0^n$  are the control points for the right half



## 29.4 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

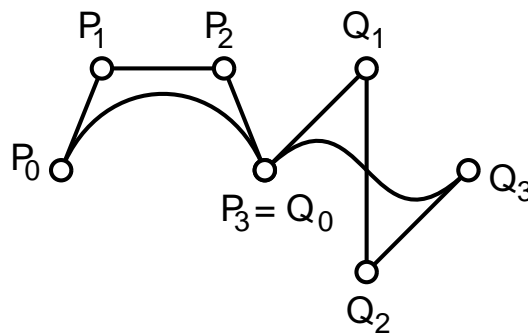
- Expensive to evaluate
- Slow convergence as we increase degree
- Control points need to be off-screen to get fine detail.

### Piecewise Polynomials

- Idea: Instead of one high degree polynomial, piece together lots of low degree polynomials
- Benefits:
  - Fast to evaluate
  - Good convergence properties
  - Potentially local
- Problem:
  - Continuity between pieces

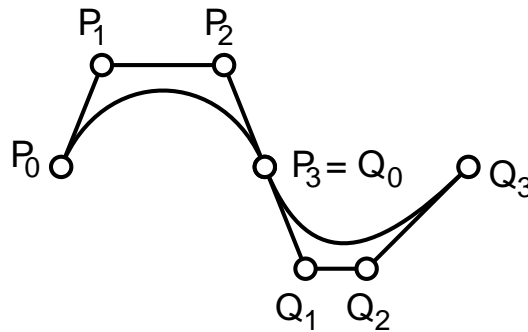
### $C^0$ Piecewise Cubic Bézier

- 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  
 $[0, 1], [1, 2], [2, 3], \dots$
- 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  $P_0, \dots, P_N$  and vectors  $\vec{v}_0, \dots, \vec{v}_N$   
Find: Piecewise  $C^1$  cubic  $P$  s.t.

$$P(i) = P_i$$

$$P'(i) = \vec{v}_i$$

for  $i = 0, \dots, N$

- Solution: Use one cubic Bézier segment per interval  
Then

$$P_{i,0} = P_i$$

$$P_{i,1} = P_i + \frac{\vec{v}_i}{3}$$

$$P_{i,2} = P_{i+1} - \frac{\vec{v}_{i+1}}{3}$$

$$P_{i,3} = P_{i+1}$$

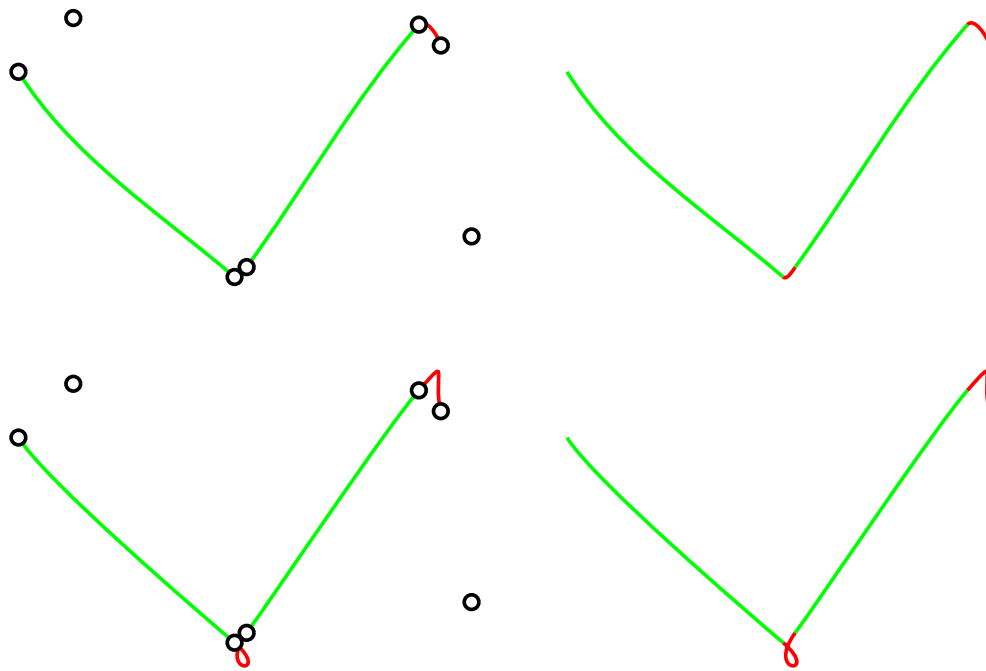
### Catmull-Rom Splines

- When modelling, 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  $i = 1, \dots, N-1$  let

$$\vec{v}_i = (P_{i+1} - P_{i-1})/2$$

- Need derivatives for  $i = 0, N$ .
  - Discard data
  - Set to 0
  - Set to  $P_1 - P_0$  (perhaps scaled)
- Parameterization: uniform, chord length, centripetal?

## Parameterization of Catmull-Rom Splines

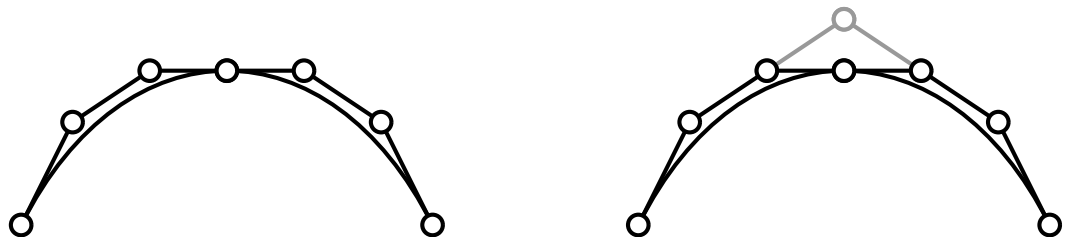


Top: Chord length

Bottom: Uniform

 $C^2$  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

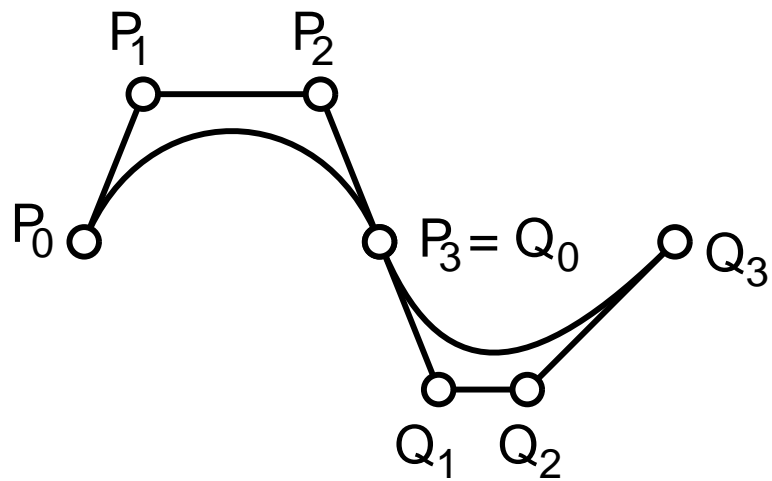


- Too hard to make user place points in this manner, so...
- Modelling: User places four control points, program places next three, user places one more, program places next three, ...

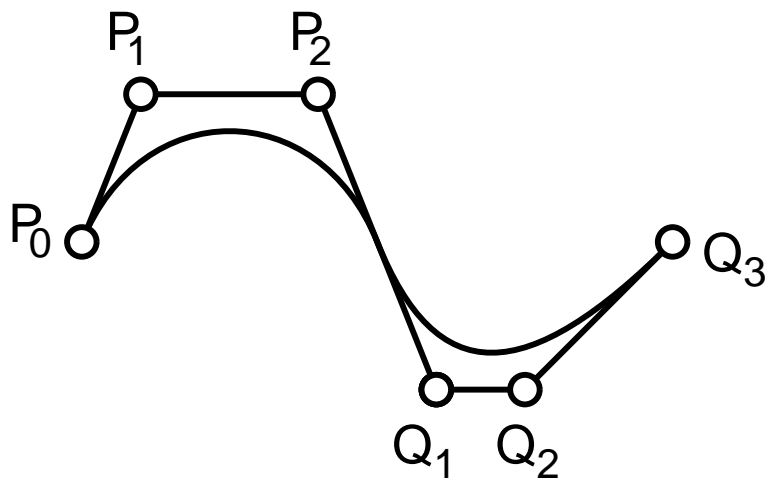
- Could hide program placed control points
- Redundant representation

### $C^1$ Bézier Spline

- Suppose we have a  $C^1$  Bézier spline

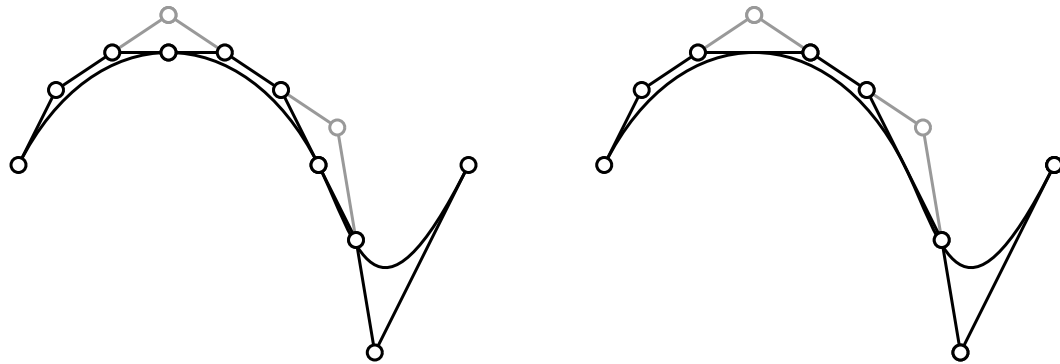


- For  $C^1$ , cubic Bézier spline, need only two control points per segment, plus one more at start and one more at end.

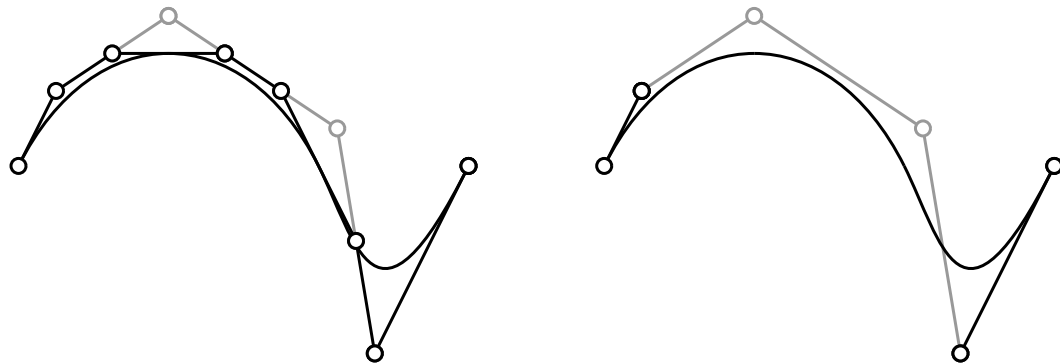


### $C^2$ Bézier Spline

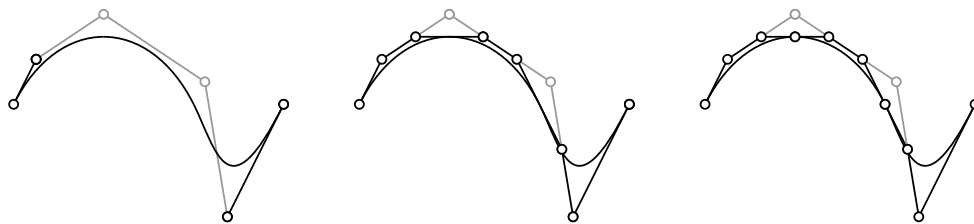
- Suppose we have a  $C^2$  Bézier spline  
Joints can be computed as described for  $C^1$



If we keep the gray points, we can compute the black points



- Thus, for  $C^2$ , cubic B-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.

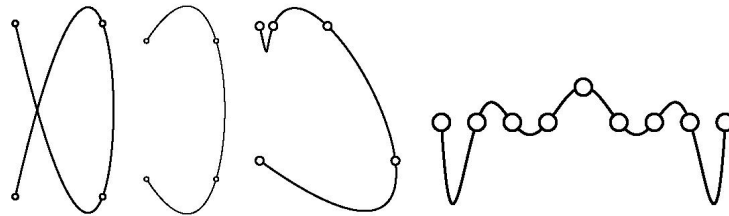
### Evaluating B-splines

- 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  $[0, 1], [1, 2], \dots$  (i.e., integers)  
One polynomial over each interval
- In general, B-splines can go over arbitrary intervals  $[t_0, t_1], [t_1, t_2], \dots$  where  $t_0 < t_1 < t_2 < \dots$   
 $t_i$  are called *knots*
- B-splines have different basis functions

Why not use interpolatory curves?

Answer 1: high degree interpolatory has oscillation



Answer 2: don't confuse modeling with representation

- Sketch with mouse and least squares fit a B-spline
- Direct manipulation of B-splines avoids exposing control points

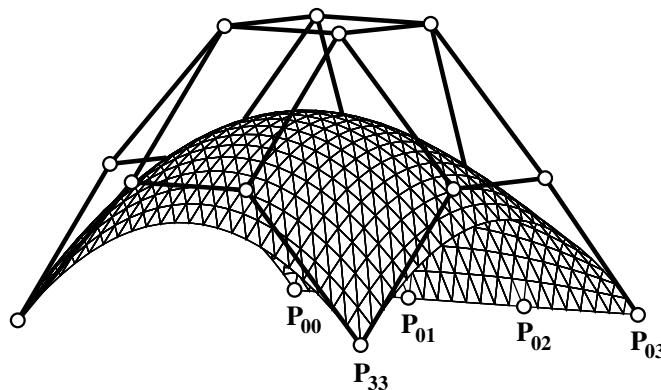
## 29.5 Tensor Product Patches

- **Control polygon** is polygonal mesh with vertices  $P_{i,j}$
- **Patch blending 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

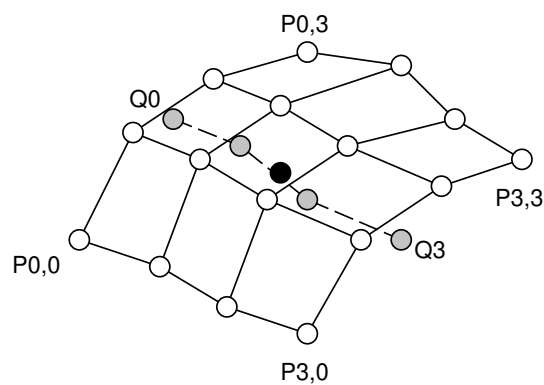
$$B_{i,j}^n(s, t) = B_i^n(s) B_j^n(t)$$



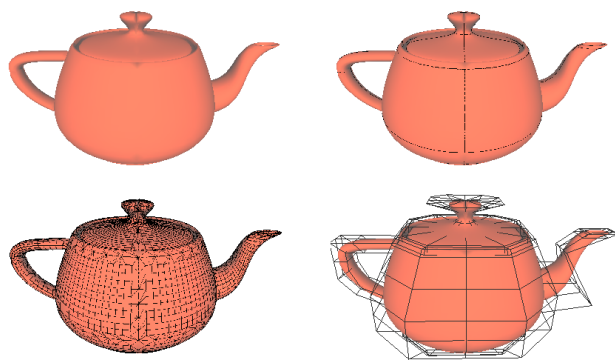


## Repeated Curve Evaluation

$$\begin{aligned}
 P(s, t) &= \sum_{i=0}^n \sum_{j=0}^n P_{i,j} B_i^n(s) B_j^n(t) \\
 &= \sum_{i=0}^n \left( \underbrace{\sum_{j=0}^n P_{i,j} B_j^n(t)}_{Q_i} \right) B_i^n(s) = \sum_{i=0}^n Q_i B_i^n(s)
 \end{aligned}$$



## The Teapot



## Properties:

- Patch blending functions **sum to one**

$$\sum_{i=0}^n \sum_{j=0}^n B_i^n(s) B_j^n(t) = 1$$

- Patch blending functions are **nonnegative** on  $[0, 1] \times [0, 1]$

$$B_i^n(s) B_j^n(t) \geq 0 \text{ for } 0 \leq s, t \leq 1$$

$\Rightarrow$  Surface patch is in the **convex hull** of the control points

$\Rightarrow$  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 is a **tangent vector** in a parametric direction
- The (unnormalized) **surface normal** is given at any regular point by

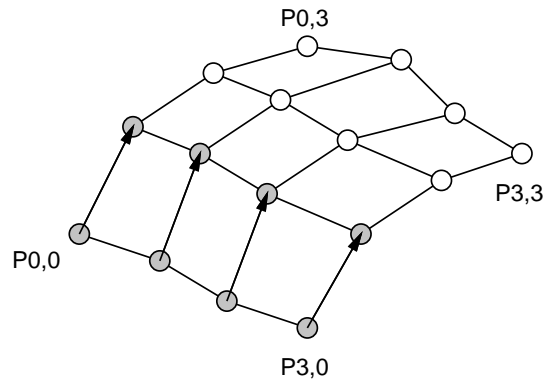
$$\pm \left[ \frac{\partial}{\partial s} P(s, t) \times \frac{\partial}{\partial t} P(s, t) \right]$$

(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):

$$n \sum_{i=0}^n (P_{i,1} - P_{i,0}) B_i^n(t)$$

(and similarly for the other boundaries)

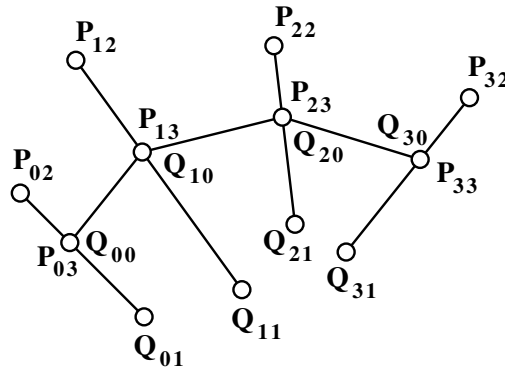


**Smoothly Joined Patches:**

- Can be achieved by ensuring that

$$(P_{i,n} - P_{i,n-1}) = \beta(Q_{i,1} - Q_{i,0}) \quad \text{for } \beta > 0$$

(and correspondingly for the other boundaries)

**Rendering:**

- Divide up into polygons:

A. By stepping

$$\begin{aligned} s &= 0, \delta, 2\delta, \dots, 1 \\ t &= 0, \gamma, 2\gamma, \dots, 1 \end{aligned}$$

and joining up sides and diagonals to produce a triangular mesh

B. By subdividing and rendering the control polygon

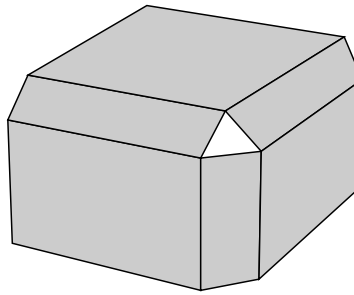
**Tensor Product B-splines**

- Could use B-splines instead of Bézier
- Automatic continuity between patches

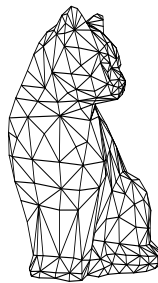
$$P(s, t) = \sum_{i=0}^S \sum_{j=0}^T P_{i,j} N_{i,j}^n(s) N_{i,j}^n(t)$$

**Problem with tensor product patches**

- Work well for rectilinear data
  - Problems arise when filling non-rectangular holes
- Suitcase Corner



- A lot of data is non-rectangular  
Triangular



## 29.6 Barycentric Coordinates

- $n+1$  points in space of dimension  $n$ :

$$D_0, \dots, D_n$$

- Points are *in general position*
- For any point  $P$  in space,

$$P = \sum a_i D_i,$$

Coordinates are weights of points, with coordinates summing to 1

## 29.7 Triangular Patches

deCasteljau Revisited Barycentrically:

- Linear blend expressed in barycentric terms

$$(1-t)P_0 + tP_1 = rP_0 + tP_1 \quad \text{where } r+t=1$$

- Higher powers:

$$P(t) = \sum_{i=0}^n P_i \left( \frac{n!}{i!(n-i)!} \right) (1-t)^{n-i} t^i$$

**Symmetric Form of Bernstein Polynomials**

$$\begin{aligned}
P((r,t)) &= \sum_{\substack{i+j=n \\ i \geq 0, j \geq 0}} P_i \left( \frac{n!}{i!j!} \right) t^i r^j \quad \text{where } r+t=1 \\
&\Rightarrow \sum_{\substack{i+j=n \\ i \geq 0, j \geq 0}} P_{ij} B_{ij}^n(r,t)
\end{aligned}$$

Examples

$$\begin{aligned}
\{B_{00}^0(r,t)\} &= \{1\} \\
\{B_{01}^1(r,t), B_{10}^1(r,t)\} &= \{r, t\} \\
\{B_{02}^2(r,t), B_{11}^2(r,t), B_{20}^2(r,t)\} &= \{r^2, 2rt, t^2\} \\
\{B_{03}^3(r,t), B_{12}^3(r,t), B_{21}^3(r,t), B_{30}^3(r,t)\} &= \{r^3, 3r^2t, 3rt^2, t^3\}
\end{aligned}$$

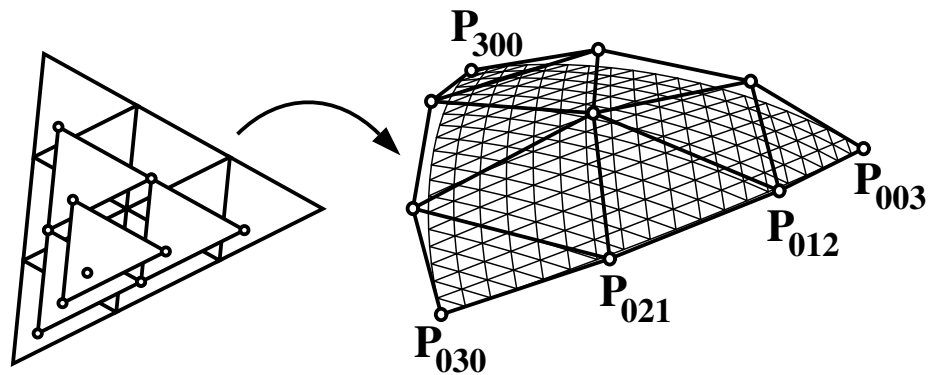
**Surfaces – Barycentric Blends on Triangles:**

- Formulas:

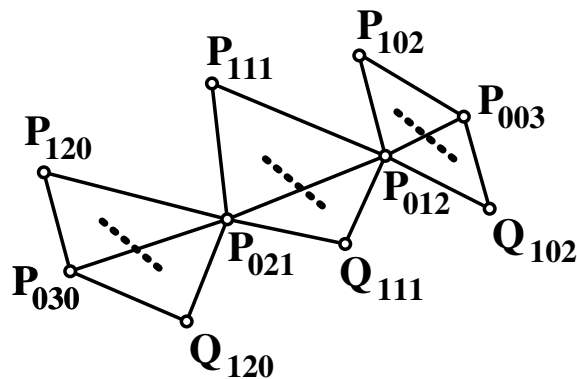
$$\begin{aligned}
P(r,s,t) &= \sum_{\substack{i+j+k=n \\ i \geq 0, j \geq 0, k \geq 0}} P_{ijk} B_{ijk}^n(r,s,t) \\
B_{ijk}^n(r,s,t) &= \frac{n!}{i!j!k!} r^i s^j t^k
\end{aligned}$$

**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  $P(r,s,t)$

**Properties:**

- Patch interpolates corner control points
- Each boundary curve is a Bézier curve  
Boundary control points define this boundary 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

## 29.8 Subdivision Surfaces

### Introduction

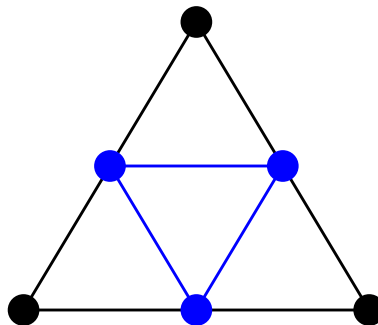
- Hard to piece spline surfaces together
- Hard to fit splines to arbitrary mesh  
 $n$ -sided faces,  $n$ -vertex neighbours
- Subdivision is an alternative surfacing scheme  
Fits surface to arbitrary mesh  
Only a few “gotcha’s”
- Will look at two simple subdivision schemes  
Both schemes are simple to implement  
Neither scheme is one of “choice”

### Polyhedron

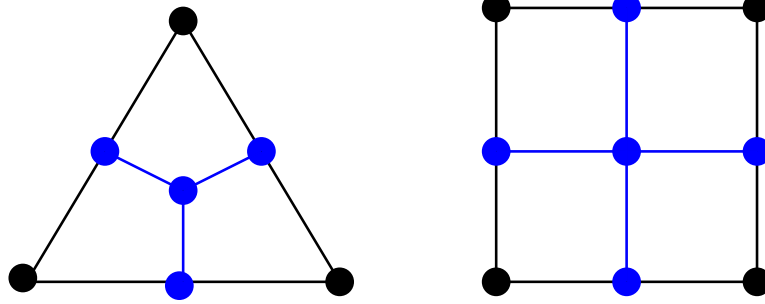
- Start with a description of a polyhedron  
List of vertices, list of faces
- Example: A cube  
Vertices:  
 $\{\{0,0,0\}, \{1,0,0\}, \{1,1,0\}, \{0,1,0\}, \{0,0,1\}, \{1,0,1\}, \{1,1,1\}, \{0,1,1\}\}$   
Faces:  
 $\{\{1,4,3,2\}, \{1,2,6,5\}, \{2,3,7,6\}, \{3,4,8,7\}, \{4,1,5,8\}, \{5,6,7,8\}\}$
- 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

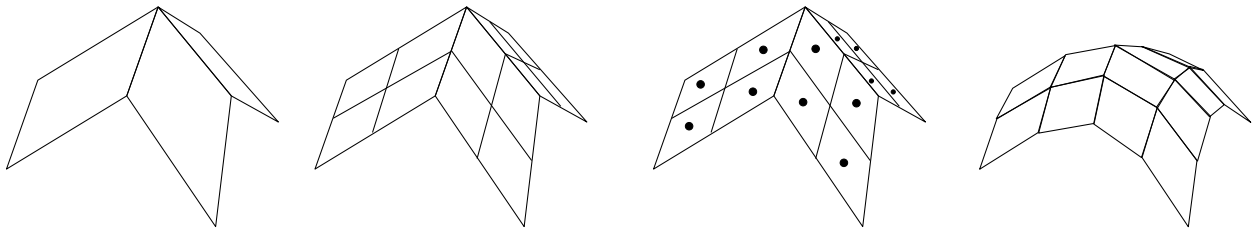


### Bilinear Subdivision Plus Averaging

1. Perform 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

$$\frac{\sum_{f \in n(v)} c_f}{\#n(v)}$$

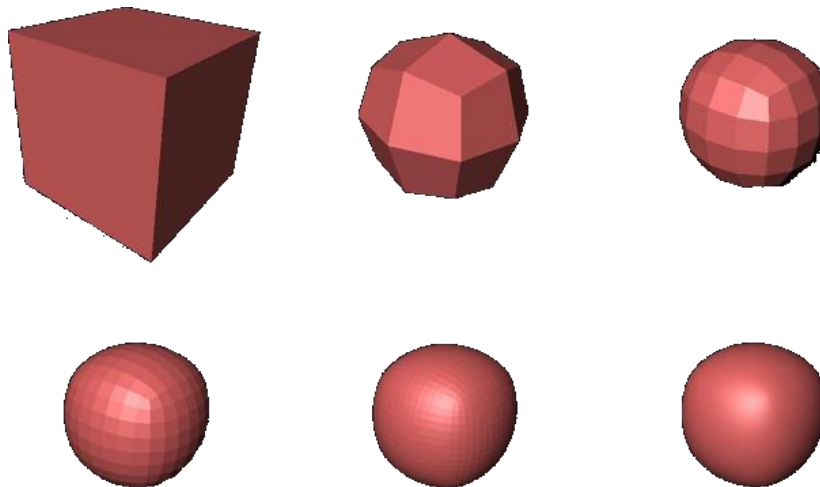
where  $n(v)$  is the set of faces neighbouring  $v$  and  $\#n(v)$  is the number of faces neighbouring  $v$



(different rules used for mesh boundaries)

In the limit,  $G^1$  surface (locally tensor product Bézier patches)

### Catmull-Clark Example



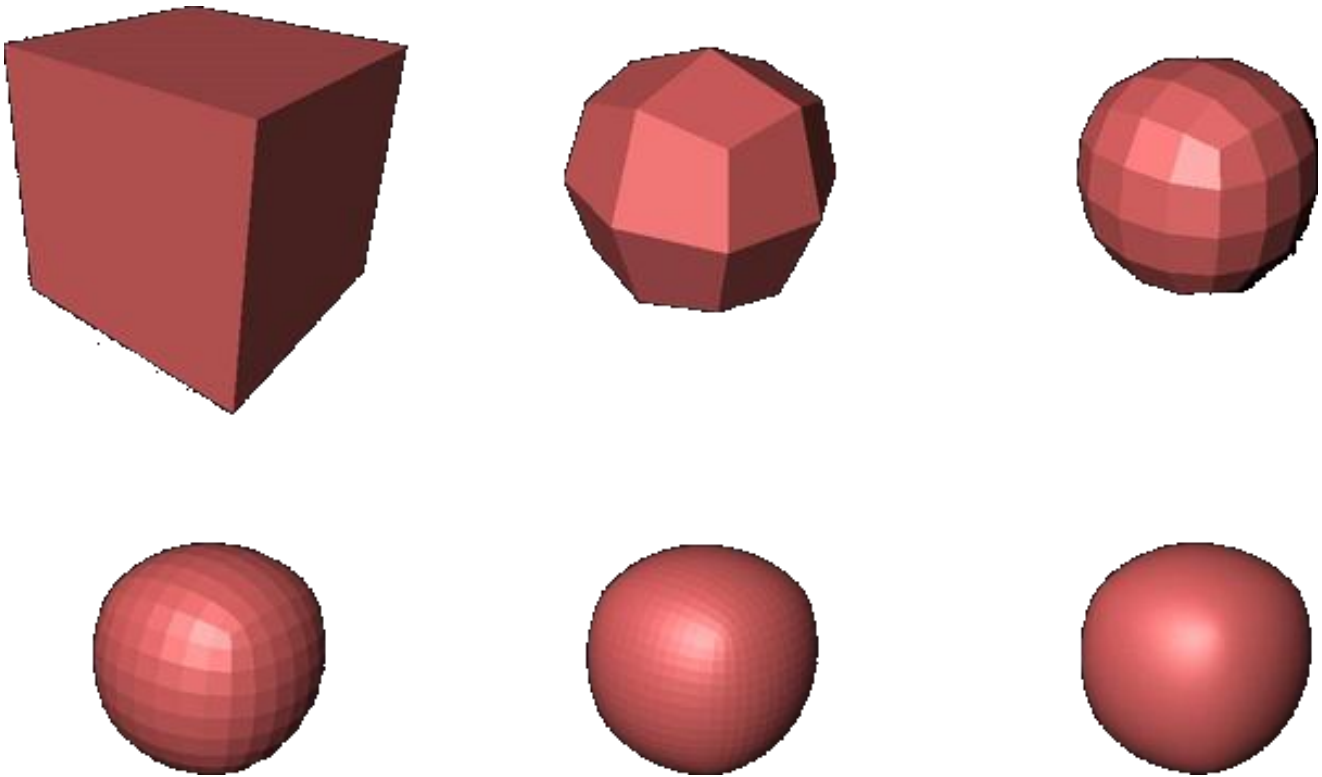


- ## Linear Subdivision Plus Triangle Averaging

- $$c_f = v/4 + 3v^+/8 + 3v^-/8$$

(b) Set  $v$  to

### Example: Loop subdivision of Icosahedron



### 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?

(Readings: This material is from Chapter 7 of Warren, Weimer, Subdivision Methods for Geometric Design. )

## 29.9 Implicit Surfaces

### Introduction

- Saw simple implicit surfaces for ray tracing:  
set of  $P$  such that

$$F(P) = 0$$

- More sophisticated methods:
  - High degree algebraics
  - CSG/Hierarchical skeletal modeling
  - A-Patches
  - Interpolatory implicit surfaces
- Issues
  - How to model with them?
  - How to render them?

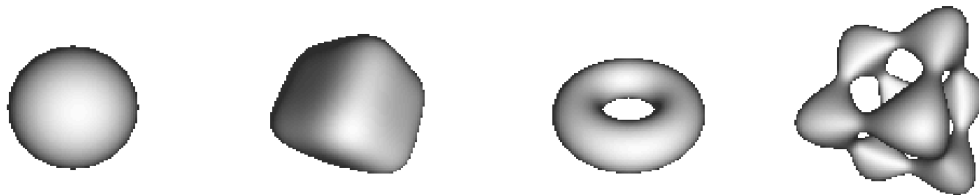
### Algebraic Surfaces

- Implicit surface in polynomial form:

$$F(P) = 0$$

where  $F$  is a polynomial

- Simple to complex models



- Root finding is “easy” (Sturm’s sequences, bisection)
- Modeling is hard  
Modeling issue tends to limit them to simple (sphere, torus) surfaces

### CSG/Hierarchical Modeling

- Model “primitives” as distance function  $R(P)$  from point, line
- Apply drop-off function to distance function:  $F(R(P))$   
Drop-off function monotonically decreasing and goes to 0 above certain distance (limits radius of influence)
- Build hierarchical CSG model from primitives

## CSG/Hierarchical Modeling

- Many nice properties
  - Fairly standard modeling paradigm
  - Surface lies in known bounded region
- Issues:
  - Blends often “bulge”
  - Modeling is as complex as using standard CSG modeling

## Interpolatory Implicit Surfaces

- Given: a polyhedron with vertices  $V$   
Find: an implicit surface  $F$  such that  $F(v_i) = 0$  for  $v_i \in V$ .
- One solutions involves using radial basis function

$$r_i(v) = |v - v_i|^3$$

We then construct

$$F(v) = \left( \sum_{v_i \in V} \phi_i r_i(v) \right) + P(v)$$

where  $P$  is a linear polynomial.

- The conditions  $F(v_i) = 0$  give a system of linear equations to solve for  $\phi$  (need 4 additional equations for  $P$ ).

## Interpolatory Implicit Surfaces—Non-zero constraints

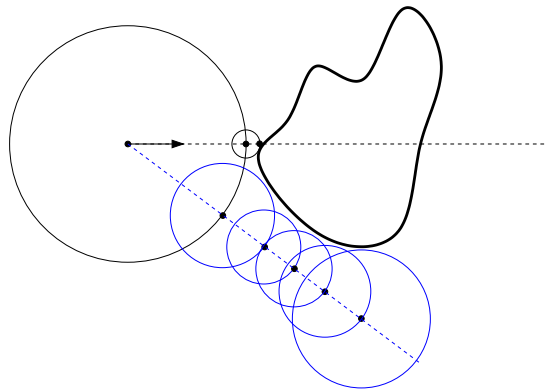
- We need at least one constraint NOT on the implicit surface, or will obtain trivial solution of  $\phi_i = 0$ .
- Interior and exterior constraints  
( $v$  s.t.  $F(v) > 0$  or  $F(v) < 0$  respectively)
- Interior/exterior constraints also specify topology
- One approach is “normal constraints”
  - At each  $v_i \in V$ , associate normal  $n_i$ .
  - Constraint is  $F(v_i + \epsilon n_i) = 1$  (with  $\epsilon = 0.01$  for example).
  - Solve with other constraints
  - Solution will interpolate positional constraints ( $F(v_i) = 0$ ) but only approximate normal constraints

Example: 800 vertices and normals

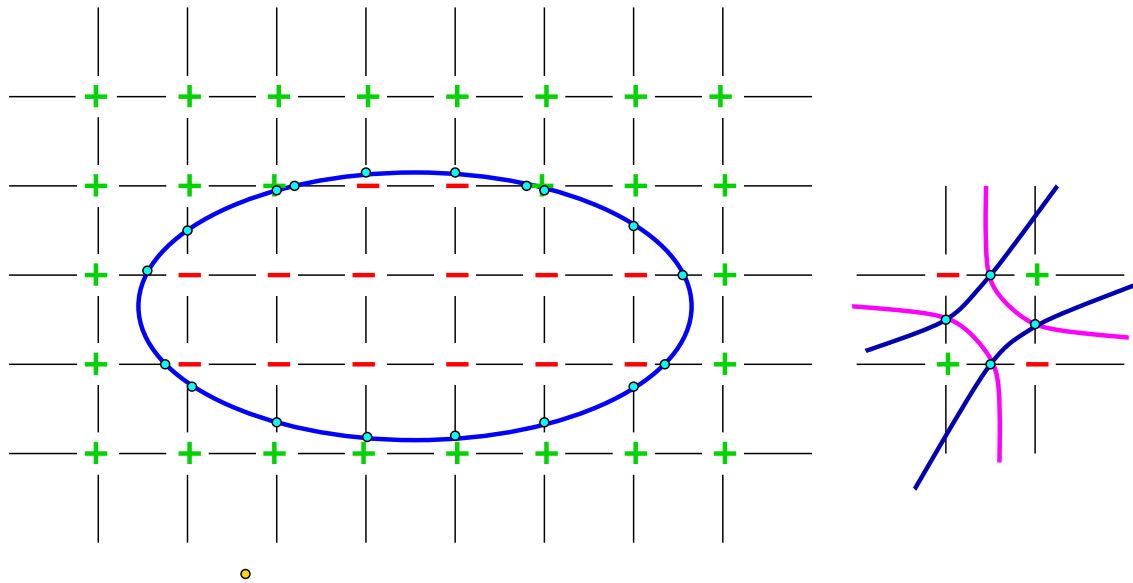
# Bunny

Rendering Implicit Surfaces

- Ray Tracing
  - Root finding
    - \* Reasonable for algebraics (Sturm sequences)
    - \* Need numerical methods for general implicits
  - Sphere tracing—variation on ray tracing
    - \* If bound of distance to surface is cheap to compute, then advance along ray by that distance and repeat.



- Marching Cubes
  - Sample function on uniform grid.
  - If adjacent samples have opposite sign, then surface passes between samples
  - Used to create triangulation of surface



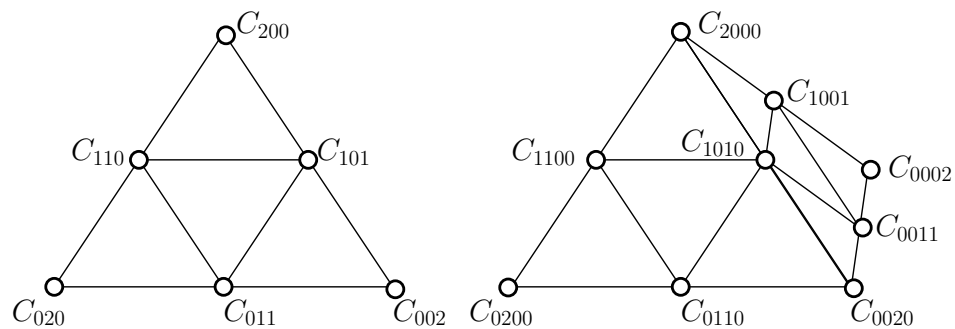
### A-Patches

Algebraic patches—easy to tessellate algebraics

- Use multivariate Bernstein basis rather than monomial basis
  - Defined over tetrahedron

$$B_i^n(P) = \binom{n}{\vec{i}} p_0^{i_0} p_1^{i_1} p_2^{i_2} p_3^{i_3},$$

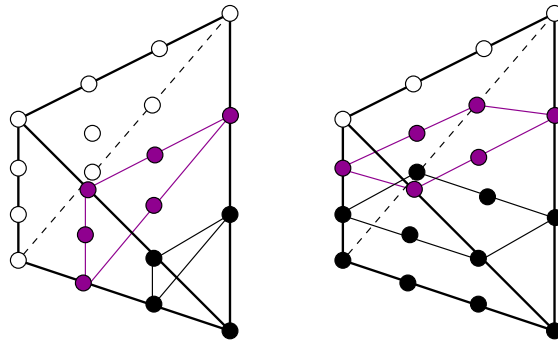
$$F(P) = \sum_{\vec{i}} C_i B_i^n(P).$$



- If all coefficients positive, then surface doesn't pass through

### A-Patches

- A-patch restriction: positive coefficients on one side, negative on the other side, with mixed sign separating layer



- Two types of A-patches (3-sided, 4-sided)
- Benefits: patch is single sheeted within tetrahedron

#### Modeling with A-Patches

- Model in a similar manner to triangular Bézier patches
  - $C^0$ : Equal coefficients on faces of neighboring tetrahedron
  - $C^1$ : Second layer coefficients have affine relationship

- Lots of control points!

(Readings:

- Gomes, Voiculescu, Joge, Wyvill, Galbraith, “Implicit Curves and Surfaces: Mathematics, Data Structures and Algorithms,” Springer, 2009.
- Greg Turk and James F. O’Brien. Modelling with implicit surfaces that interpolate. *ACM Trans. Graph.*, 21(4):855–873, 2002.
- John C. Hart. Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer*, 12:527–545, 1996.
- William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph.*, 21(4):163–169, August 1987.
- Chandrajit L. Bajaj, Jindon Chen, and Guoliang Xu. Modeling with cubic A-patches. *ACM Trans. Graph.*, 14(2):103–133, April 1995.
- Brian Wyvill, Andrew Guy and Eric Galin. Extending the CSG Tree - Warping, Blending and Boolean Operations in an Implicit Surface Modeling System, *Comput. Graph. Forum*, 18(2):149–158, 1999.

)

## 29.10 Wavelets

Wavelets are a discrete form of Fourier series

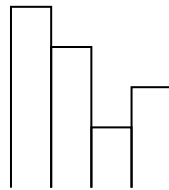
- Take a discrete signal (image).
- Decompose it into a sequence of frequencies.
- Use compact support basis functions rather than infinite support sin and cos.
- Construct a vector space using “unit pixels” as basis elements (piecewise constant functions).
- Average to get low frequencies.
- Construct “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

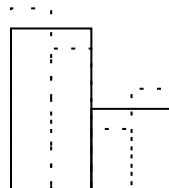
$$[9 \ 7 \ 3 \ 5]$$

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

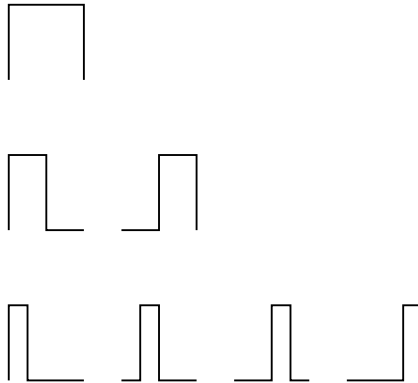
| Resolution | Averages  | Details |
|------------|-----------|---------|
| 4          | [9 7 3 5] |         |
| 2          | [8 4]     | [1 -1]  |
| 1          | [6]       | [2]     |

- 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.

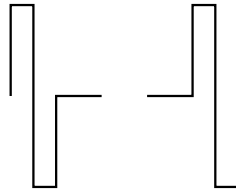
Note that the above basis functions have *compact support*, meaning they are non-zero over a finite region.

- The Haar Wavelets are

$$\psi_i^j(x) = \psi(2^j x - i)$$

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.  $V^2$ ,

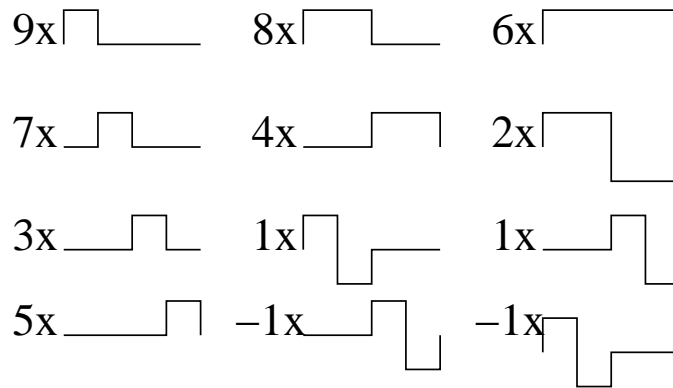
$$I(x) = 9\phi_0^2(x) + 7\phi_1^2(x) + 3\phi_2^2(x) + 5\phi_3^2(x)$$

Rewriting w.r.t.  $V^1$  and  $W^1$ ,

$$I(x) = 8\phi_0^1(x) + 4\phi_1^1(x) + 1\psi_0^1(x) + (-1)\psi_1^1(x)$$

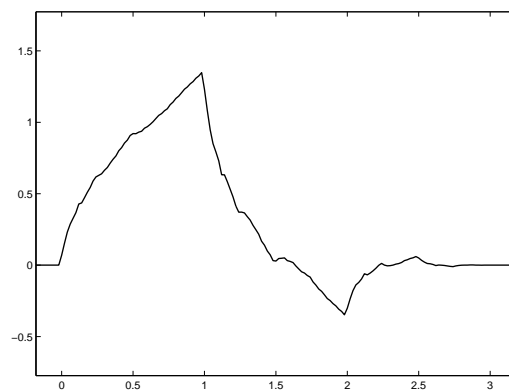
Rewriting the  $\phi$ s in terms of  $V^0$ ,  $W^0$ ,

$$I(x) = 6\phi_0^0(x) + 2\psi_0^0(x) + 1\psi_0^1(x) + (-1)\psi_1^1(x)$$



## Non-Haar Wavelets

- Haar is simple but doesn't have certain properties
- Non-Haar wavelets possible
- Example: Daubechies wavelet starts from



The function is not as smooth as it looks!

- Haar often used in practice because it's simple, cheap

## 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 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.



## 30 Non-Photorealistic Rendering

### 30.1 2D NPR

- Traditional goal of computer graphics is photorealism
- Accurate depiction of reality isn't always the real goal
- How can computer graphics help us visualize, communicate, create art?
- We'll look at some applications of computer graphics in the creation of art
- NPR issues
  - What is the goal?
  - How can we define success?

#### Painterly rendering

- Idea: create a painterly interpretation of a photograph
- Starting with a source image, apply a model of strokes to approximate it
- Possible goals:
  - Closeness of approximation
  - Use as little paint (number of strokes, total stroke length) as possible
  - Coverage of canvas

#### Paint by Relaxation

Could try to optimize directly, but it's easier to have a kind of greedy approach, painting a sequence of strokes

4 photographer images

#### Paint by numbers

Haeberli, SIGGRAPH 1990

- Interactive stroke placement (random displacement from pointer position)
- Stroke colours sampled from source image

3 face images

#### Extensions

Stroke orientation

## Extensions

### Painterly rendering with curved brushstrokes of multiple sizes

Hertzmann, SIGGRAPH 1998

- Create painting in layers: coarse to fine strokes (fill in detail where it's needed)
- Pick a set of stroke locations; draw strokes where painting is unlike source image
- Long, curved strokes flow along contours of constant colour (perpendicular to image gradient)
- Large set of intuitive customizable parameters to create painting styles (e.g., max stroke length)



### Fast paint texture

Hertzmann, NPAR 2002

- Can also render a secondary bump map by drawing stroke texture instead of solid colour
- Bump map colour image using stroke texture image

4 stroke images  
4 palm tree images

## 6 skeleton images

### Stylization and abstraction of photographs

DeCarlo and Santella, SIGGRAPH 2002

- Stroke models follow image features, but they don't understand the image. How can we apply an NPR filter in a meaningful way?
- Let a person determine importance by having them look at the source image and tracking their eye movements
- Not strokes, but large regions of constant colour
- Edges drawn as well

Woman with guitar photo  
Woman with guitar NPR image

### Simulating natural media

- Implement a physical simulation to approximate the behaviour and appearance of a traditional artistic medium

### Computer-Generated Watercolor

Curtis, Anderson, Seims, Fleischer and Salesin, SIGGRAPH 1997

- Realistic simulation of the appearance of watercolour paints
- Treat the painting as a big fluid flow simulation
- Navier-Stokes equations to push fluid around on top of the paper
- Fluid and paper exchange pigment
- Paper propagates water via capillary action

Photo of watercolour brush strokes  
Real paint

Simulation of watercolour brush strokes  
Simulated

Samples of watercolour paint simulations  
Fruit, and lots of it!

## Pencil Rendering

Sousa and Buchanan, GI 1999, CGF 2000

- Simulation of pencil lead and paper, derived from close observation of real materials
- Model the geometry of the pencil tip, pressure on page
- Graphite particles can move to and from paper, so system can model pencils, erasers, and blenders
- Pressure affects geometry of paper, and therefore appearance of future strokes

Real and simulated pencil  
Crumpled paper  
photo and 2 NPR pencil sketches of girl

## Batik

## Moose

## Embroidery

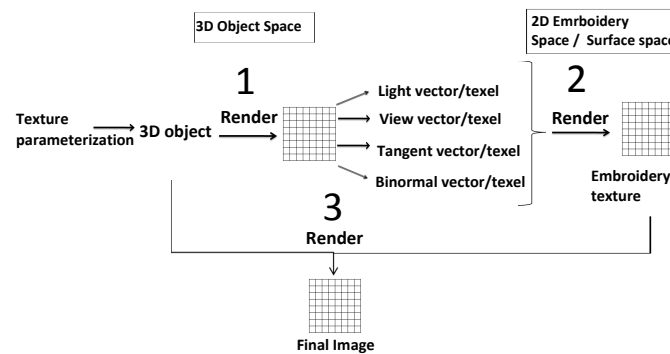
Chen, McCool, Kitamoto, Mann, GI 2012







- How to render embroidery?  
Stitching pattern: very sensitive to light direction  
Threads potentially have aliasing, geometry issues
- Stitch model
- Stitch pattern
- GPU rendering for speed



## Pen and Ink Illustration

- Problem: each stroke must simultaneously convey texture and tone

2D NPR

30.0 254

Stroke textures

Example textures

Wall images

Approximately constant  
space

View-dependent rendering of surface texture

Wall images

Wall images

Lighting-dependent rendering for shadows and accenting

House

Two more houses

Indication

Interactive pen-and-ink

Examples

Examples

Examples

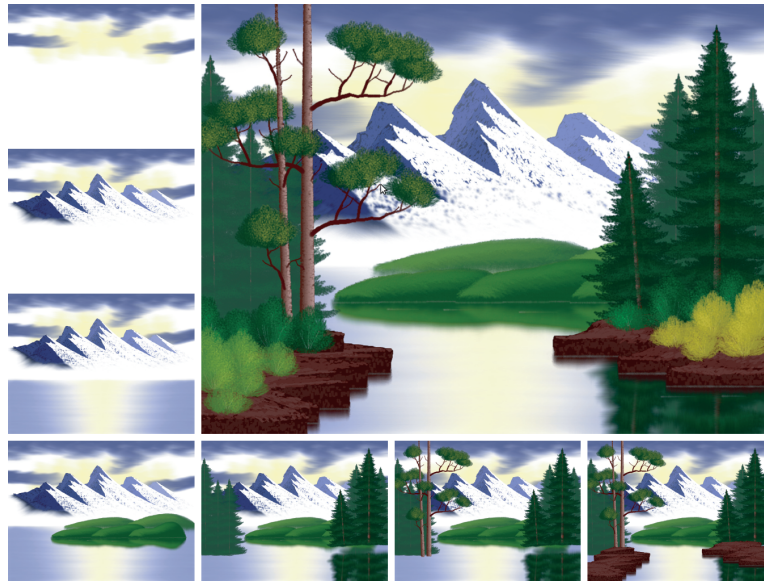
ExamplesExamples

Orientable textures

Paint bushes

Racoon

## Simulating Painting



## 30.2 3D NPR

- Support real-time non-photorealistic rendering of 3D scenes via hardware acceleration

## HijackGL and NPRQuake

NPR Quake

NPR Quake

NPR Quake

## Real-time hatching

Six examples

## Image-space silhouettes

- Silhouettes are discontinuities in the depth image
- Creases are second order discontinuities in the depth image
- The right image on which to do edge detection

Hex nut example

Funky box examples

- Creases can also be seen as discontinuities in the normal image

Bust

## Object-space silhouettes

- Given a polyhedral mesh and a camera position, find and draw all visible silhouettes
- Two subproblems:
  - Find all possible silhouettes relative to the current viewpoint
  - Process these silhouettes to determine visibility

## Silhouettes

- What is a silhouette? For a smooth object, it's a point where the vector from the camera just grazes the point.
- In other words:

$$(p - c) \cdot n = 0$$

Where  $p$  is the point on the surface,  
 $c$  is the location of the camera,  
 and  $n$  is the normal at point  $p$ .

## Silhouettes on meshes

- A mesh with face normals is not smooth
- Approximate silhouettes by finding mesh edges that separate a front face from a back face
- Slow to check all edges every frame, so use some kind of acceleration (edge buffer, probabilistic checking, ...)
- Can also identify silhouettes on smooth mesh (one with vertex normals)

## Silhouette visibility

- Just because an edge is a silhouette doesn't mean it's visible:
  - Occlusion by another object
  - Occlusion by another part of the same object
  - Partially visible
- Need to decide, for every silhouette edge, which parts are visible

## Image-space visibility processing

- Turn on depth buffering
- Draw all mesh faces in background colour
- Draw all silhouette edges, give every edge a unique colour that identifies it
- Frame buffer now holds pixel-accurate image of which pieces of silhouettes are visible

## Stylized silhouettes

- Use visibility to guide placement of stylized strokes along paths
- Parameterization and temporal coherence are still problems

## Toon shading

- Cartoons typically use a small range of colours for each object in a scene
  - Shadow, body, and sometimes highlight
- Easy to simulate in ray tracer or real-time
  - Quantize by diffuse attenuation to two or three colours

Elf girl

Roses

## Toon shading in OpenGL

- Perform diffuse lighting computation at every vertex in software
- Use diffuse brightness as parameter to a one-dimensional texture map
  - Map has only two or three colours
- Sample the texture with nearest-neighbour interpolation: deliberately not blended
- Can be on the GPU as a fragment program
- Usually draw silhouettes, etc., on top of toon shading

Cell shaded bear. SIGGRAPH 2002: WYSIWYG NPR



Saxophone Hero



## 31 Volume Rendering

### 31.1 Volume Rendering

#### Volume Data

- scalar field
  - like temperature in room at each  $(x, y, z)$
- many applications
  - medical (MRI, CT, PET)
  - geophysical
  - fluid dynamics
- The images in this section of the notes came from "Introduction to Volume Rendering", Lichtenbelt, Crane, and Naqvi, HP Professional Book.

#### Volume Data

VR Data - engine, guts

#### Visible Human Project

Visible Human slices

#### Seismic Data

Seismic Data

#### Data Format

- 3D array of scalars (a volume)
  - $256^3$  or  $512^3$  common
  - 32-128 MB
  - Visible human project:
    - Male: 15 GB (1mm slices)
    - Female: 40 GB (0.33mm slices)
  - Seismic data (for oil exploration) will be 100's of GB or more 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

## Displaying Volume Data

- How do we display volume data?  
Data on “outside” hides data on “inside”
  - Transparency
  - Slicing
  - Segmentation and selective display
- How do we render volume data?  
Sampling issues

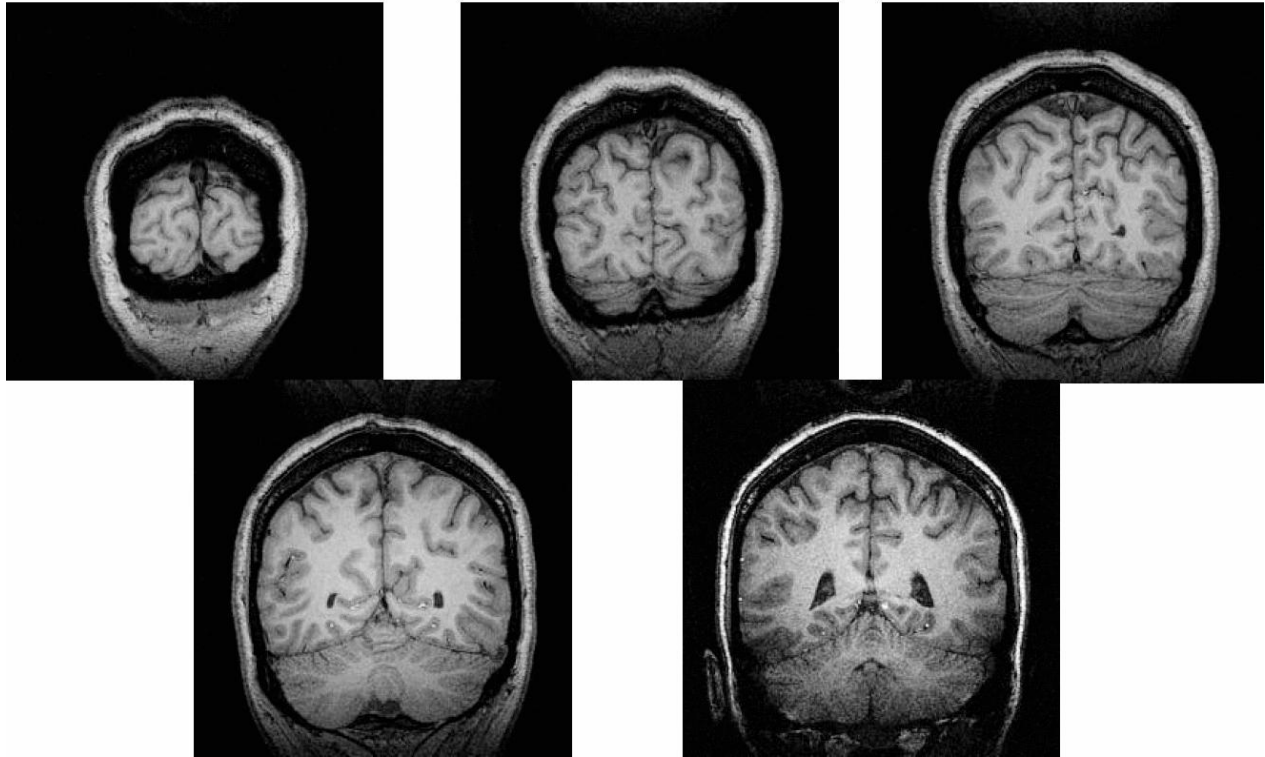
## Volume Slicing

- 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, ...
- geological volumes: sand, slate, oil, ...
- manual procedure, need good software

- problems:
  - noise
  - overlapping intensity ranges

## Segmentation

### Labled guts

#### Gradient Computation

- gradient shows density changes
- useful for illumination model
- gradient magnitude like edge detection
  - can build isosurfaces
  - can colour according to magnitude

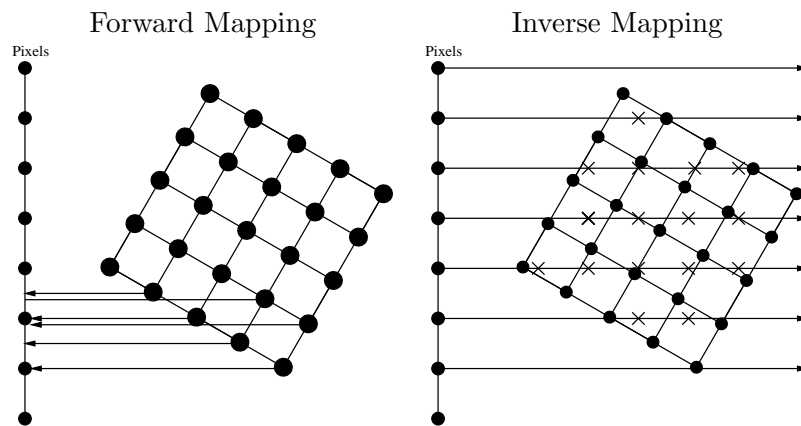
#### Gradient Computation

### Xray of foot x 3

#### 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

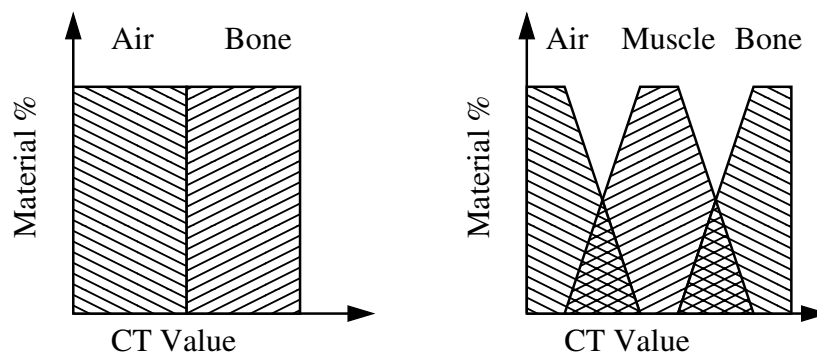
## Resampling



## Classification

- must map intensity data to RGB
- 16-bit (monochrome) data  $\Rightarrow$  8-bit RGB
- look-up table (LUT) for range of interest
- user sets LUTs for
  - R, G, B
  - gradient magnitude
  - opacity/transparency

## Classification



## Classification

Classified engine



## 32 Animation

### 32.1 Overview

#### Computer Animation

**Animation:** Rapid display of slightly different images create the illusion of motion.

**Conventional approach:** Keyframed animation

Keyframes, inbetween frames

**Computer Assisted:**

- Human key frames, computer inbetweening
- 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: McConnell, Chapter 10.)

### 32.2 Traditional 2D Cel Animation

#### Traditional 2D Cel Animation

- Traditional animation is labor intensive.
  - Start with story board.
  - Master artist draws keyframes to define action.
  - Sweat-shop artists draw inbetween frames to complete animation.
- Disney was the most successful, but it was high risk.

A few traditional animation rules:

- Stretch and squash (anti-aliasing), timing, secondary actions
- Exaggeration, appeal, follow through
- Anticipation, staging.
- Be careful with camera
  - Smooth changes
  - Few changes

### 32.3 Automated Keyframing

- Replace the human inbetweener with interpolation algorithms.
  - Keyframes: settings of parameters at different points in time.
- + The computer provides
- repeatability
  - automatic management of keyframes.
- Interpolation is not as smart as a human inbetweener.
- Requires more key frames and/or additional information

#### Utility

Limit the number and maximizes the utility of key parameters.

- Immediate geometric or visible significance.
- Static properties maintained automatically.
- Relationships between parts of articulated objects maintained automatically.
- Interpolation routines geared towards parameter types.
- Real-time interface to interactively set parameters and iteratively improve the animation.

### 32.4 Functional Animation

- Independent scalar function specified for each “keyframed” parameter.
- Basic capability that supports others.
- Most useful for truly scalar quantities:  
brightness of a light source
- Splines useful for representing functions.
- Continuity control a must: both automatic maintenance and selective breaking.
- The functions can be edited both *explicitly*, as graphs, or *implicitly*, through a direct manipulation interface.

## Linear Interpolation

- 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

$$p(t) = \frac{t_1 - t}{t_1 - t_0}p_0 + \frac{t - t_0}{t_1 - t_0}p_1.$$

- Advantages and Disadvantages:
  - Discontinuities in derivative exist at all key frame points.
  - + The rate of change within a segment is constant  
(can easily be controlled).

## Controlling Velocity

Given a constant rate of change, can create *variable* rate of change.

Given any function  $\tau = f(t)$ , reparameterize with  $\tau$ .

For the lerp,

$$p(\tau) = p(f(t)) = \frac{f(t_1) - f(t)}{f(t_1) - f(t_0)}p_0 + \frac{f(t) - f(t_0)}{f(t_1) - f(t_0)}p_1.$$

## 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. (non-issue)
- Splines are more difficult to implement. (non-issue)

## Transformation Matrix Animation

- One way to support an animation capability:  
interpolate a transformation matrix.
- Keyframes are “poses” of objects given by the animator.
- Functional animation applied independently to all entries of the transformation matrix.

Does not support animation of rigid bodies!

- Given two poses of an object that differ by a  $180^\circ$  rotation.
- Under linear interpolation, object will turn inside out, collapse to a plane in the middle of the interpolation.

## 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.

## 32.5 Motion Path Animation

Want 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.

## Motion Path Spline Interpolation Problems

Spline position interpolation gives continuity control over changes position, but:

- Visually notice discontinuities in 2nd derivative  
Need  $C^3$  splines (degree 4).
- Equal increment in spline parameter does not correspond to equal increment in distance along the spline.
- If we parameterize the spline directly with time objects will move at a non-uniform speed.

## Arc Length Parameterization

1. Given spline path  $P(u) = [x(u), y(u), z(u)]$ , compute arclength of 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 motion path parameterized by arclength,  $s$ :  $P(s) = P(A^{-1}(s))$ .



4. Let  $s = f(t)$  specify distance along the spline as a function of time  $t$ .

The motion path as a function of time  $t$  is thus given by

$$P(A^{-1}(f(t)))$$

### Problems with Arc-Length Parameterization

1. The arc-length  $s = A(u)$  is given by the integral

$$s = A(u) = \int_0^u \sqrt{\left(\frac{dx(v)}{dv}\right)^2 + \left(\frac{dy(v)}{dv}\right)^2 + \left(\frac{dz(v)}{dv}\right)^2} dv$$

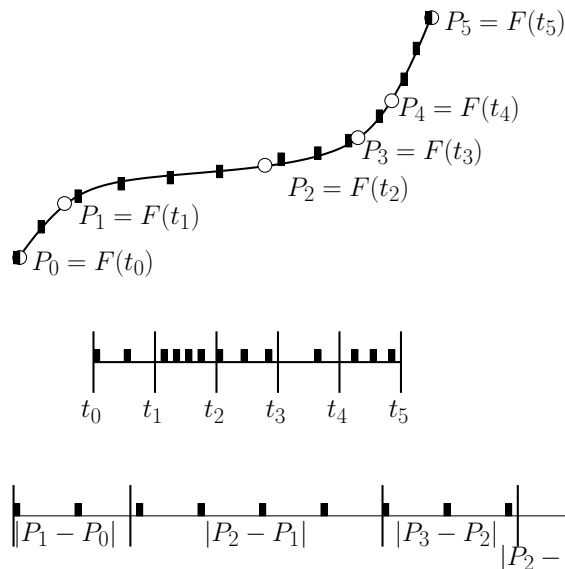
No analytic solution if the motion path is a cubic spline.

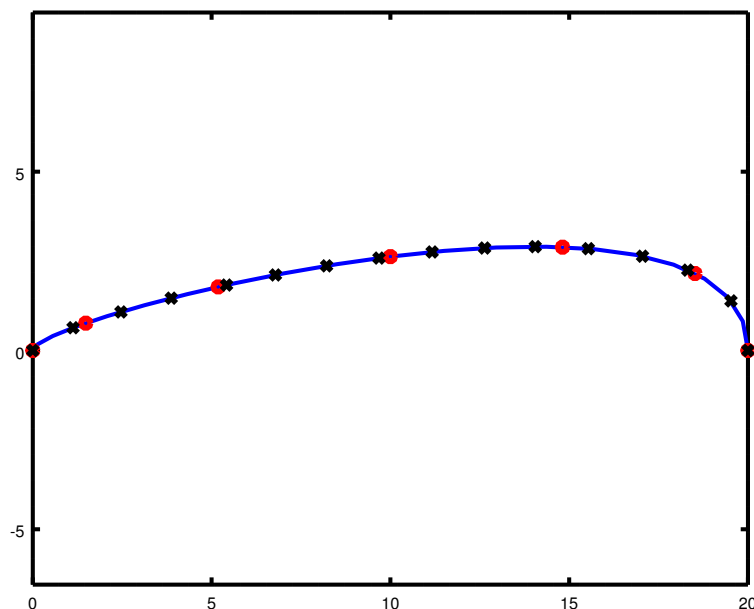
2. Since  $A(u)$  has no analytic form,  $A^{-1}(s)$  has no analytic form.

### Issues in Real-Time Animation

- Exact arc-length parameterization not feasible.
- Alternative: compute points on the spline at equally-spaced parametric values, use linear interpolation along these chords.
- The linear interpolation should consider the distance between samples to maintain constant velocity.

### Approximate Arc Length Parameterization



**Example**

**Blue:** curve; **Red:** 7 regular samples in domain;

**Black:** 16 approx arc length samples

**32.6 Orientation and Interpolation**

Interpolate angles rather than transformation matrices.

- In two dimensions, only one angle is involved and animation is straightforward.
- In three dimensions, orientation requires three degrees of freedom.

Interpolation is much nastier and harder to visualize.

**Approaches**

Two standard approaches to the orientation interpolation.

Both related to 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 mathematically.
- Interpolation can be expensive in practice.

## Euler Angles

- 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  $\theta_x$ ,  $\theta_y$ , and  $\theta_z$ .

Problems 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 (not unique).

**Coordinate system dependence:** The orientation of the coordinate axes is important.

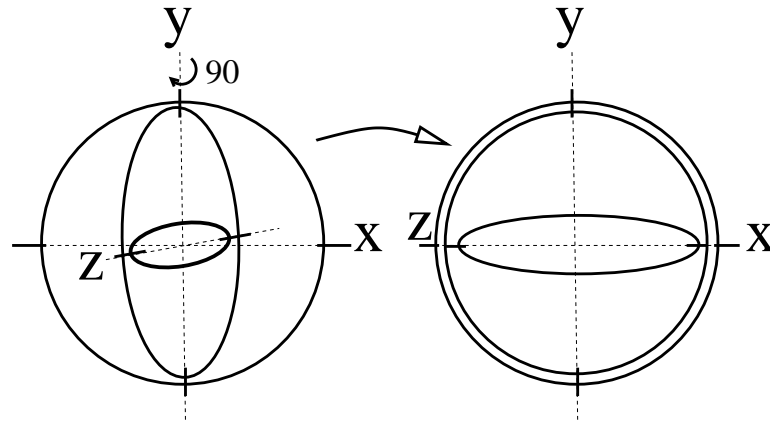
## Gimbal Lock: An Example

- *Gimbal lock* is an example of a **parametric singularity**.
- Gimbal lock is a mechanical problem that arises in gyroscopes as well as Euler angles.

- Set  $\theta_y = \pi/2 = 90^\circ$ , and set  $\theta_x$  and  $\theta_z$  arbitrarily. Then  $c_y = 0$ ,  $s_y = 1$  and the matrix  $R(\theta_x, \pi/2, \theta_z)$  can be reduced to

$$\begin{aligned}
 R(\theta_x, \theta_y, \theta_z) &= \begin{bmatrix} 0 & 0 & -1 & 0 \\ s_x c_z - c_x s_z & s_x s_z + c_x c_z & 0 & 0 \\ c_x c_z + s_x s_z & c_x s_z - s_x c_z & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
 &= \begin{bmatrix} 0 & 0 & -1 & 0 \\ \sin(\theta_x - \theta_z) & \cos(\theta_x - \theta_z) & 0 & 0 \\ \cos(\theta_x - \theta_z) & \sin(\theta_x - \theta_z) & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.
 \end{aligned}$$

- A  $y$ -roll by  $\pi/2$  rotates the  $x$ -axis onto the negative  $z$  axis, and so a  $x$ -roll by  $\theta$  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.

## 32.7 Quaternions

- Four-dimensional analogs of complex numbers.
- Can be used to represent orientation without a directional selectivity.
- **Recall:** multiplication of complex numbers represents orientation and rotation in the plane.
- The rectangular and polar form of a complex number  $p$  are

$$p = a + bi = Ae^{i\theta}.$$

- Multiplication is a rotation about the origin:

$$p_1 p_2 = A_1 A_2 e^{i(\theta_1 + \theta_2)}$$

## Quaternions — Definitions

- Quaternions defined using three imaginary quantities:  $i$ ,  $j$ , and  $k$ :

$$q = a + bi + cj + dk.$$

- Rules for combining these imaginary quantities:

$$\begin{aligned} i^2 = j^2 = k^2 &= -1, \\ ij = -ji &= k, \\ jk = -kj &= i, \\ ki = -ik &= j. \end{aligned}$$

- Rules sufficient to define multiplication for quaternions.
- Quaternion multiplication *does not commute*.
- A quaternion can be broken into a scalar and a vector part:

$$q = (s, \vec{v}) = s + v_1i + v_2j + v_3k.$$

- In vector notation, product of two quaternions is

$$q_1q_2 = (s_1s_2 - \vec{v}_1 \cdot \vec{v}_2, s_1\vec{v}_2 + s_2\vec{v}_1 + \vec{v}_1 \times \vec{v}_2).$$

- Quaternions also have conjugates:  $\bar{q} = (s, -\vec{v})$ .
- The norm of a quaternion is defined by

$$|(s, \vec{v})| = \sqrt{s^2 + v_1^2 + v_2^2 + v_3^2}.$$

## Unit Quaternions

- Unit quaternions have unit norms
- Isomorphic to orientations
- The unit quaternion given by

$$q_r = (\cos(\theta), \sin(\theta)\vec{v})$$

is equivalent to a rotation by an angle of  $2\theta$  around the unit  $\vec{v}$ .

Note that  $q_r$  is equivalent to  $-q_r$  when interpreted as an orientation.

## Rotations and Quaternions

- Points in space can be represented by quaternions with a zero scalar part.
- Point  $P$  is represented by the quaternion

$$q_P = (0, P).$$

- The rotation of a point  $P$  by  $q_r$  is

$$q_{P'} = q_r q_P q_r^{-1}.$$

- For unit quaternions, the inverse is equivalent to the conjugate.
- **Idea:** Use unit quaternions to represent *orientation*

## Interpolating Unit Quaternions

- Linear interpolation on all the entries of the quaternion does not have unit norm. Angular velocity is not constant if we renormalize.
- Spherical linear interpolation (*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* is

$$q(u) = q_1 \frac{\sin((1-u)\omega)}{\sin(\omega)} + q_2 \frac{\sin(u\omega)}{\sin(\omega)}.$$

## 32.8 Animating Camera Motion

- Requirements for camera motion are different from object motion:
  - Specification of camera motion has a long cinematic tradition that should be respected.
  - The camera should always be level, unless we specify otherwise.
  - The image of objects of interest should be stable on the film plane.

### Camera Cinematic Terminology

**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.

### Camera Cinematics

- An animated zoom changes perspective, which can be disturbing.  
Use *dolly* 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.
- Use smooth spline animation 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.

### Camera Animation Specification

- 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.

## 32.9 Tools for Shape Animation

### 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.

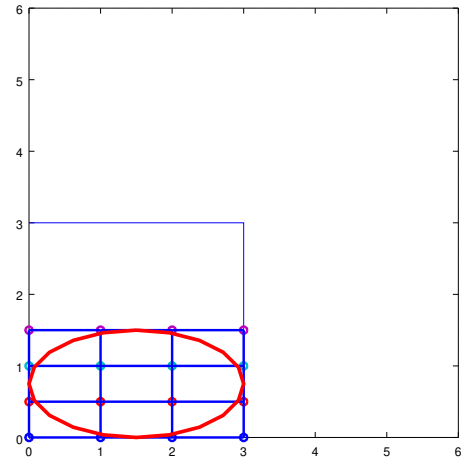
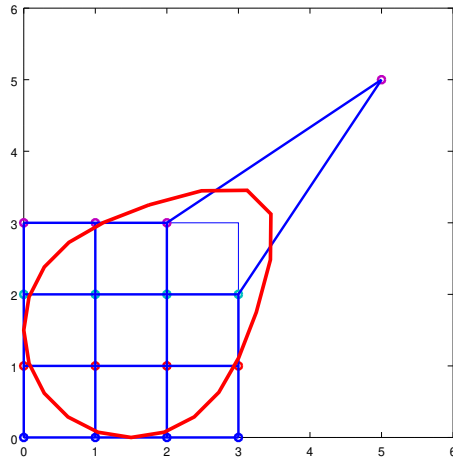
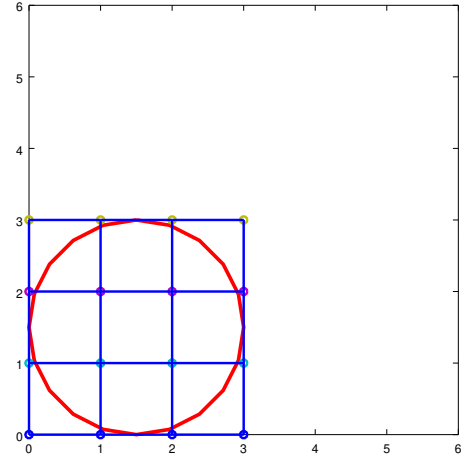
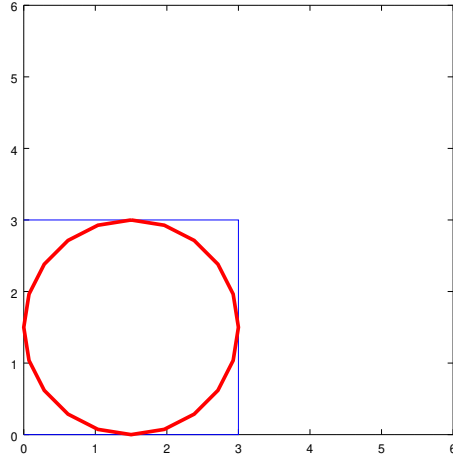
Example: a kneecap and its skin surface affected by both the femur and shinbone, and assume a position halfway between either.

## Skeleton/skinning example

### Free-Form Deformations

- Sometimes, skeleton cannot capture desired shape change of an object.
- Example: The “squash 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.



**FFD Process****Process**

1. Place rectangular “squish box” object 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 the control points are initially set to these values and are moved to effect a deformation.

### FFD Issues

- 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.

- The object should be finely tessellated or radical deformations will not work properly.
- Not volume preserving.

For realistic animation, typically only small deformations are used.

- Modeling UI non-trivial

Manipulating control points inadequate

### Skeletons and FFDs

- 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.

## 32.10 Kinematics and Inverse Kinematics

### 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.

### 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.

### 32.11 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 secondary 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!

## 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.

## Spring-Mass Models

Create a model:

- Composed of a mesh of point masses  $m_i$  connected by springs with spring constants  $k_{ij}$  and rest lengths  $\ell_{ij}$ .
- 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 all 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:

$$\begin{aligned}\vec{f}_{ij}^s(t) &= -k_{ij}(\ell_{ij} - |P_i(t) - P_j(t)|) \frac{P_i(t) - P_j(t)}{|P_i(t) - P_j(t)|}, \\ \vec{f}_i^s(t) &= \sum_{j \in N_i} \vec{f}_{ij}^s(t).\end{aligned}$$

- 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  $t$ .

- 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

$$\begin{aligned}\vec{a} &= \left[ \frac{d^2 x_i(t)}{dt^2}, \frac{d^2 y_i(t)}{dt^2}, \frac{d^2 z_i(t)}{dt^2} \right], \\ \vec{v} &= \left[ \frac{dx_i(t)}{dt}, \frac{dy_i(t)}{dt}, \frac{dz_i(t)}{dt} \right]\end{aligned}$$

- Initial conditions: user supplies initial positions of all masses and velocities.
- The user supplies external forces: gravity, collision forces, keyframed forces, wind, hydrodynamic resistance, etc. as a function of time.
- Simulation.
  - Factor second-order ODE into two coupled first-order ODE's:

$$\begin{aligned}\vec{v} &= [v_{xi}(t), v_{yi}(t), v_{zi}(t)] \\ &= \left[ \frac{dx_i(t)}{dt}, \frac{dy_i(t)}{dt}, \frac{dz_i(t)}{dt} \right], \\ \vec{a} &= \left[ \frac{dv_{xi}(t)}{dt}, \frac{dv_{yi}(t)}{dt}, \frac{dv_{zi}(t)}{dt} \right], \\ &= \frac{1}{m_i} \left( -\rho_i \vec{v}(t) + \vec{f}_i^s(t) + \vec{f}_i^e(t) \right).\end{aligned}$$

- Solve using your favourite ODE solver.

The simplest technique is the Euler step:

- \* Pick a  $\Delta t$ .
- \* Then compute the values of all positions at  $t + \Delta t$  from the positions at  $t$  by discretizing the differential equations:

$$\begin{aligned}\vec{a}_i(t + \Delta t) &\leftarrow \frac{\Delta t}{m_i} \left( -\rho_i \vec{v}(t) + \vec{f}_i^s(t) + \vec{f}_i^e(t) \right), \\ \vec{v}_i(t + \Delta t) &\leftarrow \vec{v}_i(t) + \vec{a}_i(t) \Delta t, \\ P_i(t + \Delta t) &\leftarrow P_i(t) + \vec{v}_i(t) \Delta t.\end{aligned}$$

## 32.12 Human Motion

### Walking

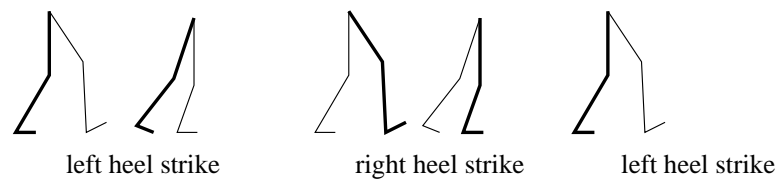
Modeling walking is hard

- 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

## 32.13 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**

- Rigid
- Have mass
- May restrict to planar for simplicity

**Sensors : 4 types**

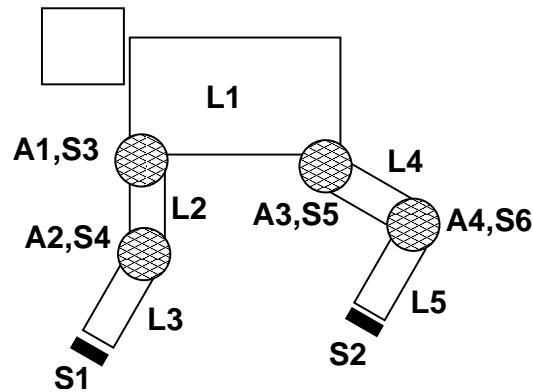
- Touch (picture)
- Angle (picture)
- Eye - used to find target
- Length (picture)

**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****32.14 Biologically-Based Simulation****Biologically-Based Simulation**

- To improve walking simulations, match model to body:
  - Model of muscle contractions to get motions
  - Minimize metabolic energy expended to walk
  - Use modern CPU power to solve

**Actuators**

Muscle actuators

**Controller**

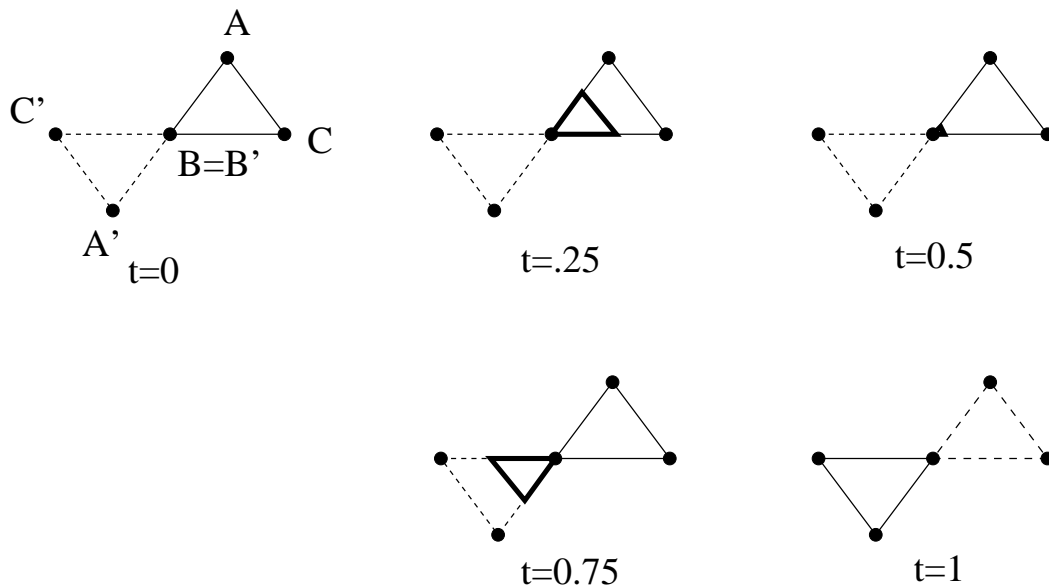
Controller

(Readings: Optimizing Locomotion Controllers Using Biologically-Based Actuators and Objectives Jack M. Wang, Samuel R. Hamner, Scott L. Delp, and Vladlen Koltun ACM Transactions on Graphics 31(4), 2012)



### 32.15 Morphing

- Morphing is lerp'ing between images.
- Nice effects at low cost.
- Key issues:
  - Partitioning images into pieces that correspond  
E.g., eyes to eyes, etc.
  - Lerp'ing pieces (geometry)
  - Lerp'ing 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 properties not violated

# Morphing Example — associating key points

## 32.16 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.

## Body suit motion capture

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.
- Motion capture tends to require more-or-less exotic technology.
  - Even puppetry requires at least real-time graphics.
  - Technical 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?
  - Perceptual problem:
 

Motion capture animations often look too stiff

**32.17 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 rigid
 

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



## 33 Computational Geometry

### 33.1 Introduction

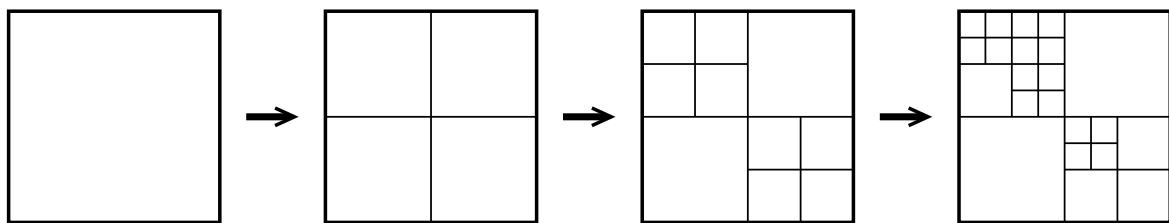
#### Computational Geometry

- Study of algorithms that can be stated in terms of geometry
  - Data structures
    - Spatial partitioning, locality
  - Algorithms
- This is meant to be a brief introduction to a few common things that you see in computational geometry

### 33.2 Data Structures

#### Quad Trees

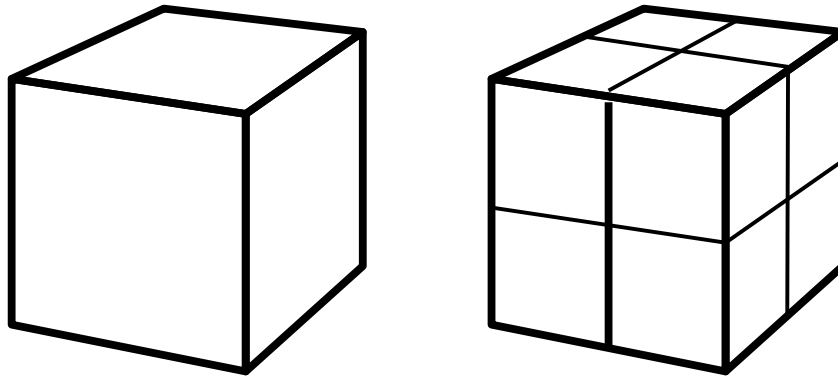
- Adaptively 4:1 split



- Simple data structure (4 or 5 ptrs + data)
- Simple creation
- Neighbors can be non-trivial
- Triangular variant
- Anchoring to avoid T-vertices
  - Easy if neighbors no more than 1 depth different
- Example use: subdivide height field terrain

#### OctTrees

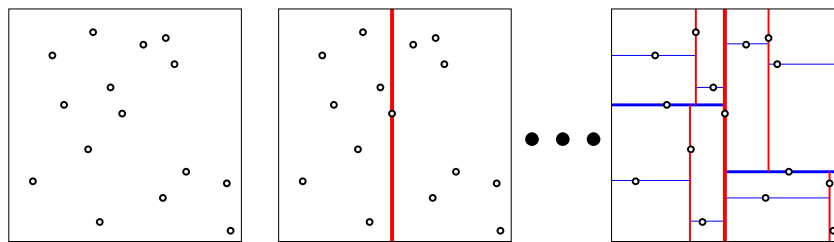
- Similar to quadtree, but 8:1 split



- Example use: spatial subdivision to speed ray tracing

### $k$ D-Trees

- $k$ D-Trees  
Separate data based on  $x, y$  alternately



- Example use: photon mapping

### Nearest Neighbors in $k$ D-tree

Given point  $p$

- Search tree as if inserting  $p$  in tree.  
Save node as tentative solution
- Unwind recursion, determining at each step of element of neighboring node could be closer  
If possible for point in neighbor cell to be closer, search it.

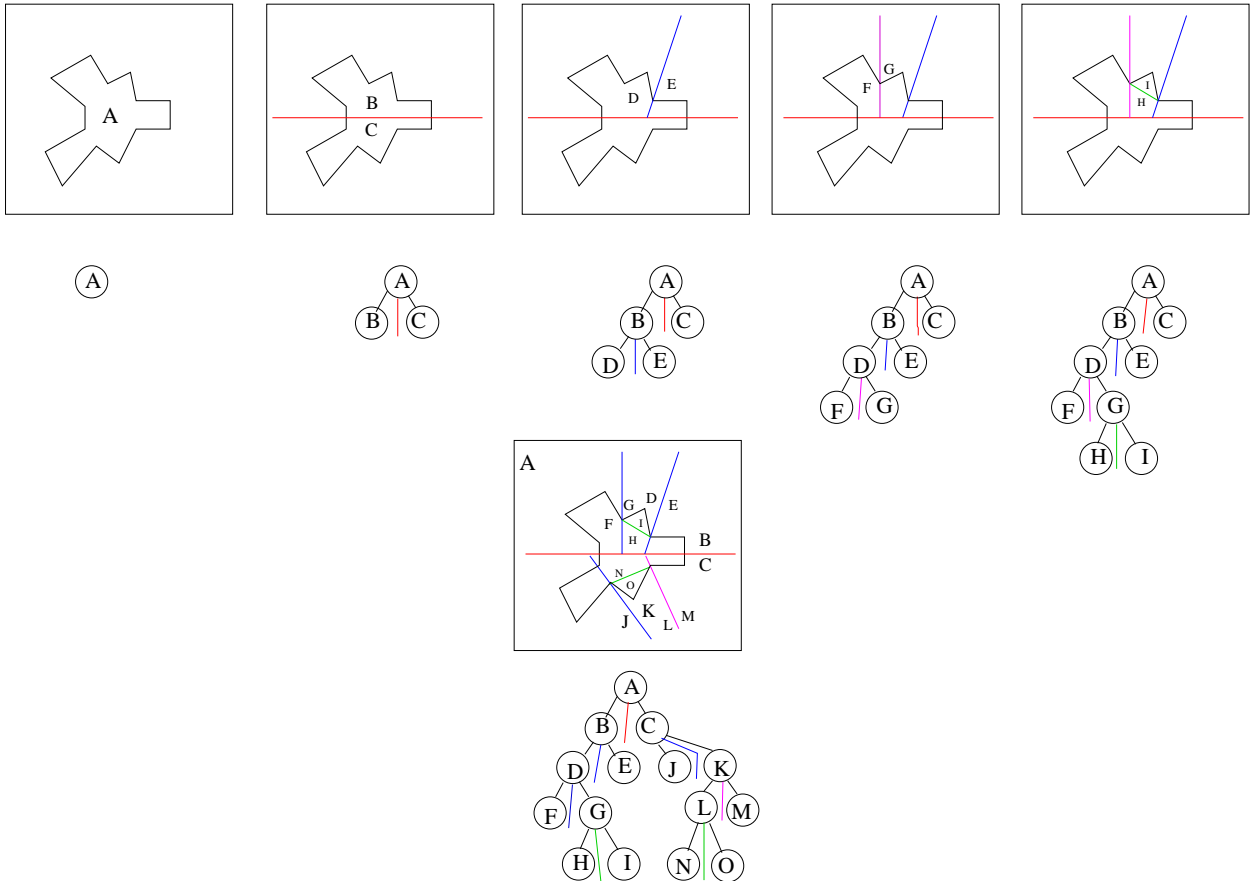
$k$ -nearest neighbors:

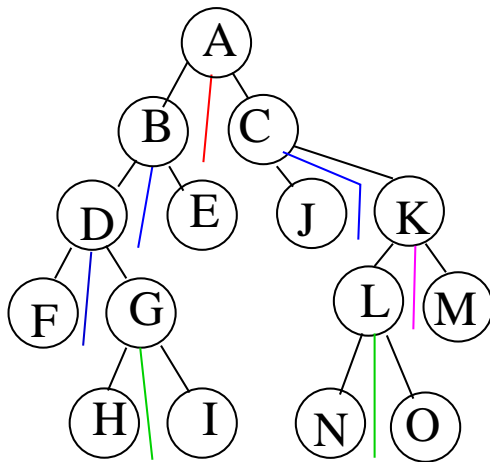
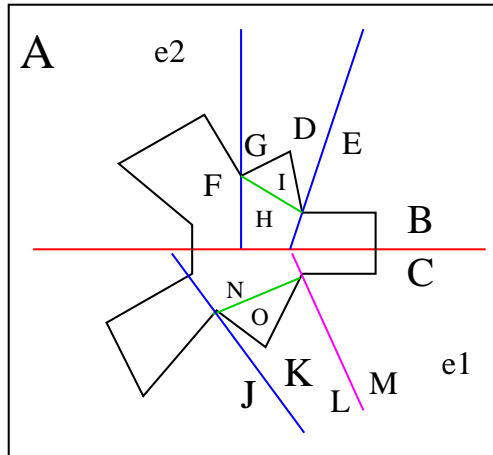
- Simple idea: Run nearest neighbors search  $k$  times



**BSP Trees**

- Binary Space Partitioning (BSP) Trees
- Subdivide space along carefully chosen planes  
Planes stored in tree are oriented
- Example use: Painter's algorithm

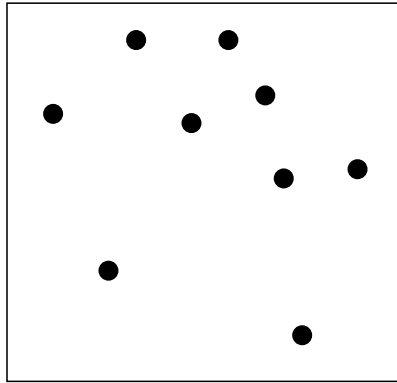


**BSP assisted Painter's Algorithm**

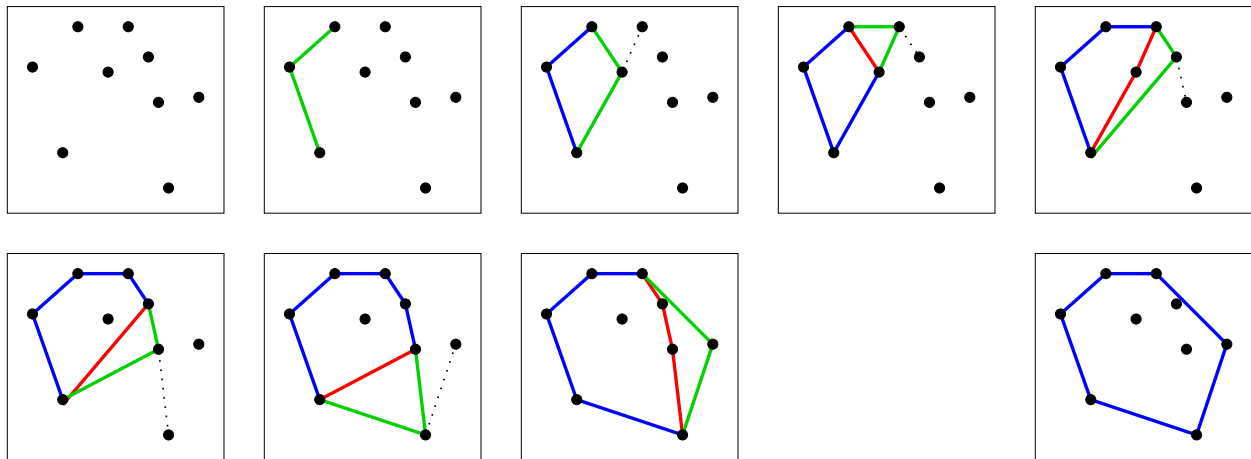
- Traverse tree
- At each node, locate eye relative to oriented splitting plane
- Draw the side the eye isn't on first, then draw the side the eye is on
- May need to resolve depth order within node

**33.3 Convex Hull****Convex Hull**

- Given a set of points, find the small polygon enclosing the set of points
- Physical algorithm: stretch rubber band around all points and allow it to tighten



### One convex hull algorithm

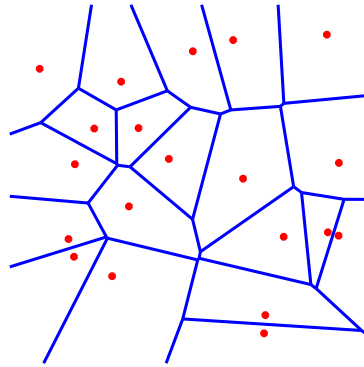


- Sort points on  $x$ -coordinate
- Create top and bottom chains, processing left to right  
Add next point to both chain;  
delete “interior” edges

## 33.4 Voronoi Diagrams

### Voronoi Diagram

- Given a set of points in the plane  
Partition the plane into regions closest to each point



- Often computed as dual of Delaunay triangulation
- Example uses: game AI, ghost maps, and many more

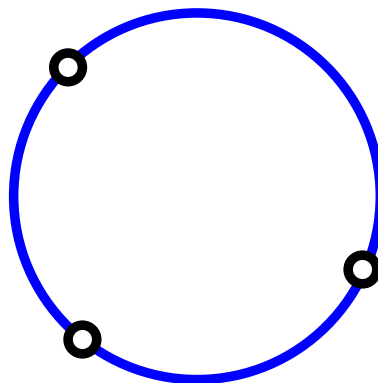
## Ghost Map

- Each bar represents one cholera death

## 33.5 Delaunay Triangulation

### Delaunay Triangulation

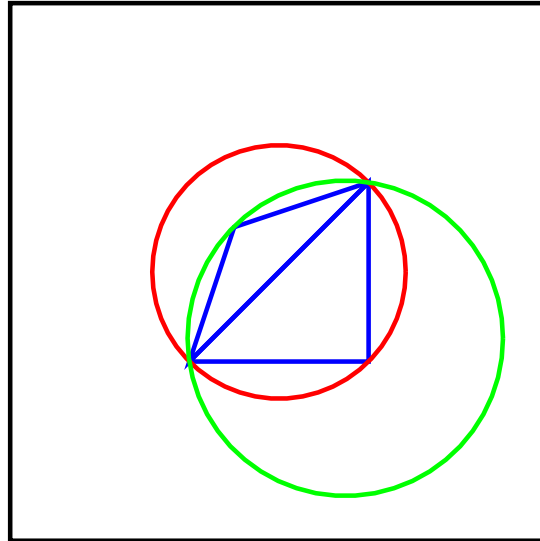
- Given a set of points and a triangulation of these points  
Delaunay if for any triangle, no other point lies inside the circumcircle of the triangle.



- Key operation: flip

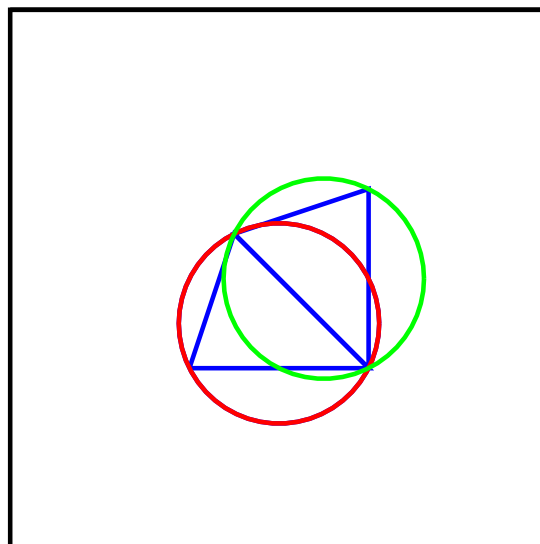
## Flip

For a pair of adjacent triangles, compute circumcircle(s):



If fourth point inside circle, then flip...

## Flop

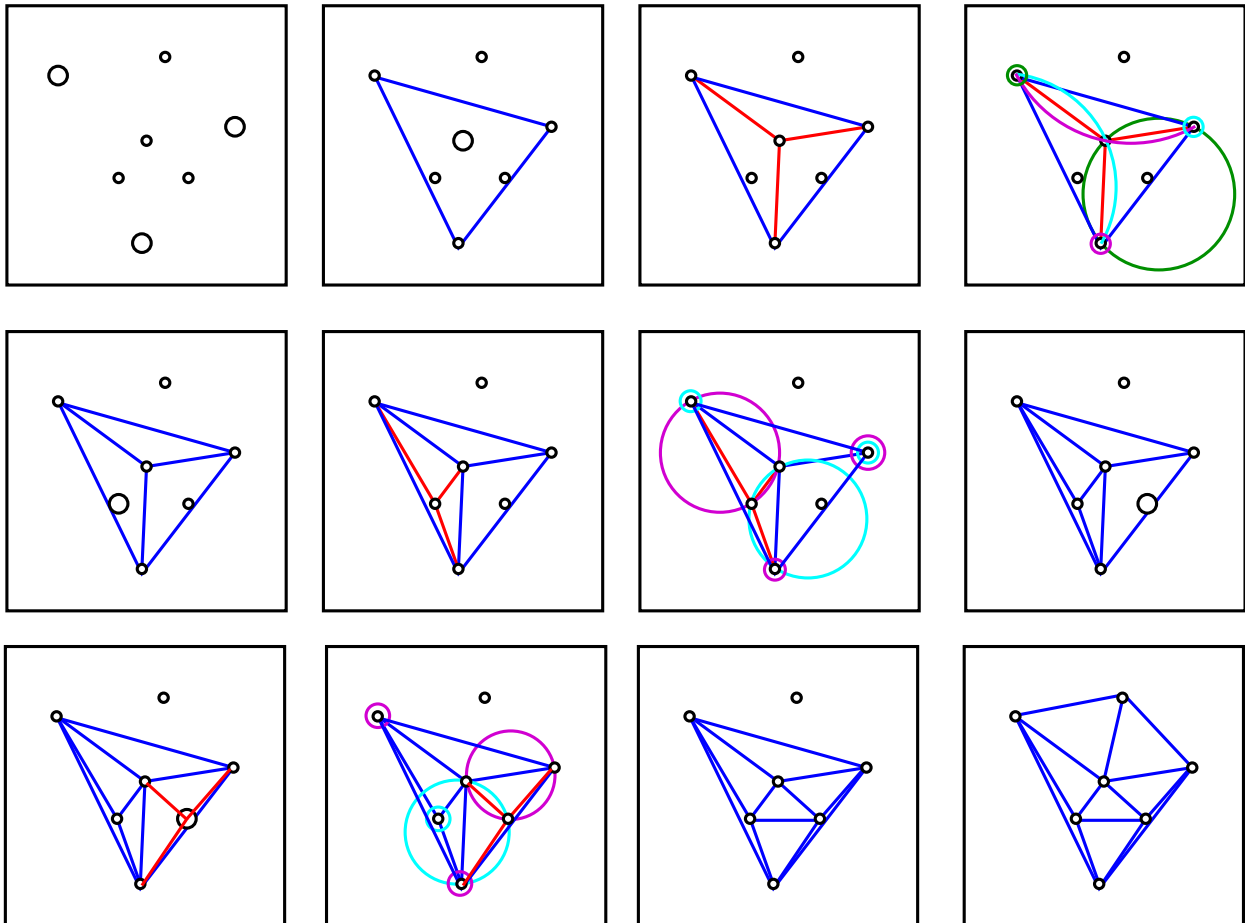


### Algorithm 1

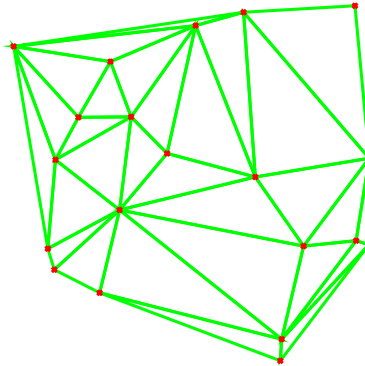
- Start with any triangulation of points
- Look at each pair of adjacent triangles and flip as needed

### Algorithm 2

- Start with any 3 points; form triangle.
- For each remaining point:  
 Insert new point into triangulation  
 3:1 split if inside a triangle, grow new triangle if not  
 Flip if needed

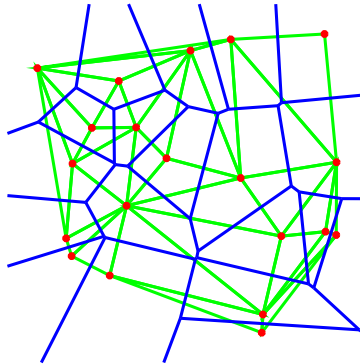


## Delaunay Triangulation



- Maximizes minimum angle
- “Avoids” long, skinny triangles

## Voronoi and Delaunay are Duals



## 33.6 More Computational Geometry

### Issues

- Algorithms assume points in “general position.”  
In particular, no four points cocircular
- In practice, people put points on grids, which are naturally cocircular.
- For Delaunay triangulation, this means “triangles” may be squares.

## More Computational Geometry

- Other computational geometry problems
- 3D and higher dimensions





## 34 Assignments

### 34.1 Assignment 0: Introduction

The goals of this assignment include:

- Familiarization with the course computing environment.
- Setting up your account and directories.
- Modifying a UI.
- A trial assignment submission.

This assignment is not worth any marks. However, we *strongly suggest you do it*:

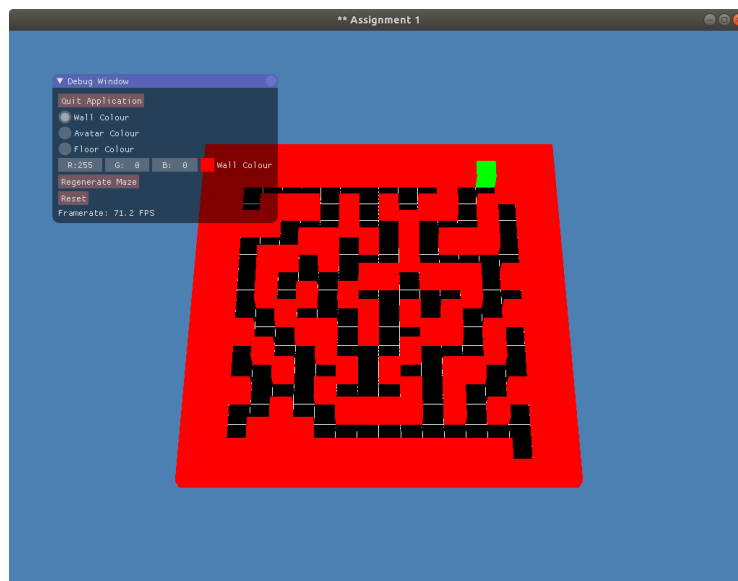
- It's easy.
- You have to learn most of it eventually anyways.
- It'll save you from losing marks over dumb mistakes in protocol.

### 34.2 Assignment 1: Introduction to OpenGL

In this assignment you will learn how to use OpenGL to render a polygonal model.

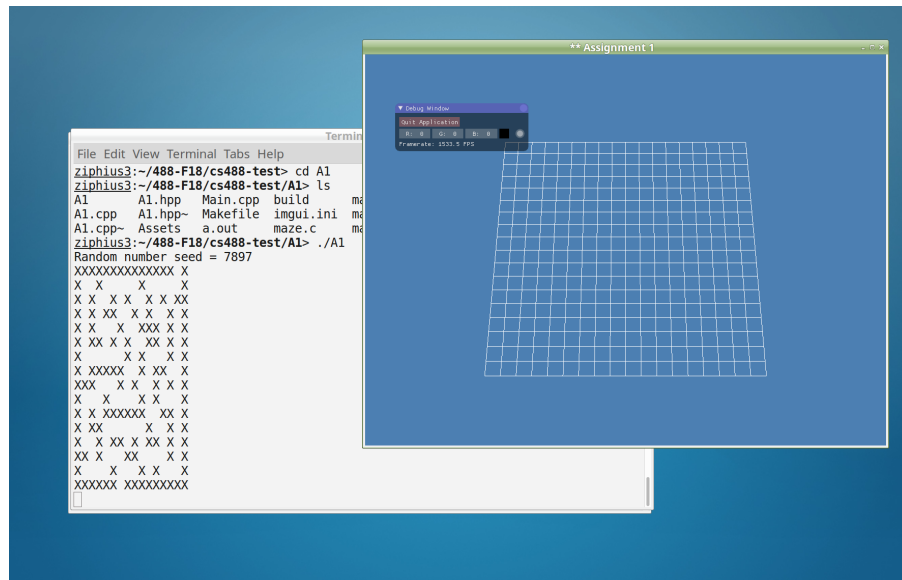
- Draw polygons
- Transformations
- UI

#### Screen Shots



Sample screen shots. Some details may vary term to term.

## ASCII Maze



ASCII maze

### 34.3 Assignment 2: Frames and Perspective

This assignment includes:

- Modeling, Viewing, and Perspective Transformations
- 3D Line/Plane Clipping
- Menus and Valuator

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.

#### Frames

There are six entities you need to consider:

**The Cube Model:** Defined on the unit points  $[\pm 1, \pm 1, \pm 1]$ .

**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.

## Features

Your program will support the following features:

**Menus:** Selection of transformation mode and coordinate system.

**Valuators:** Mouse movement to specify scalar parameters.

**Modelling Transformations:** Rotation, translation, scale.

**Viewing Transformations:** Camera rotation and translation, perspective.

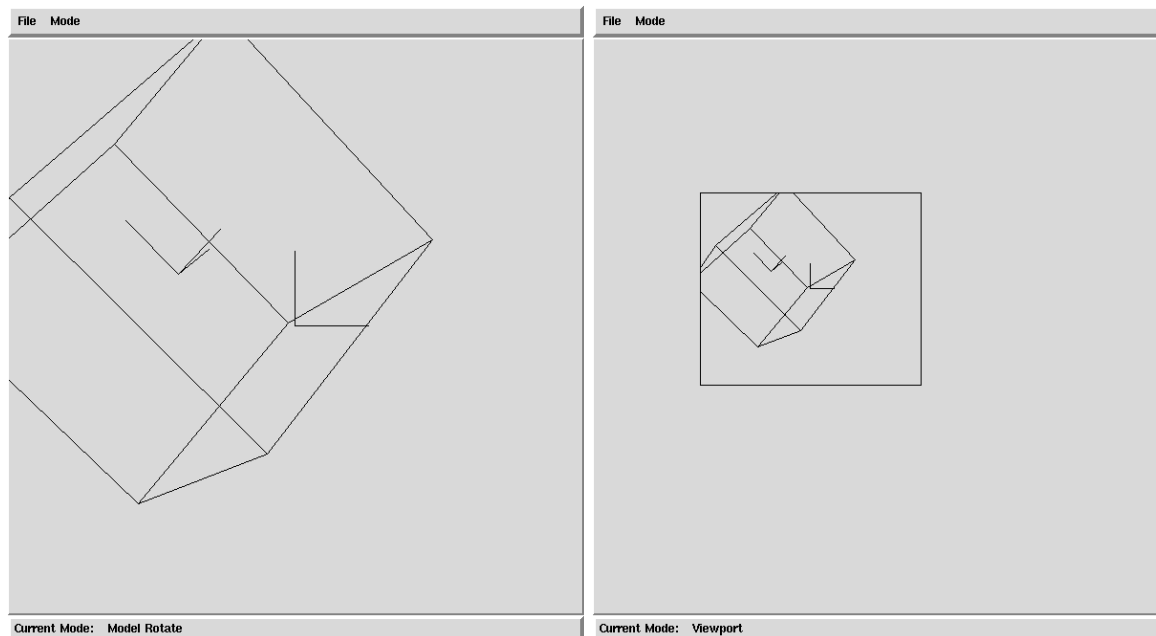
**3D Line Clipping:** To avoid divide by zero, you need to clip all lines to the near plane.

**Viewport:** Clip the drawing to a 2D viewport.

OpenGL may be used for only 2D line drawing.

You must implement all transformations and clipping yourselves.

## Screen Shots



Sample screen shots. Some details may vary term to term.

## Tips

Suggestions, not requirements

- Near and far planes: put about 2 units on each side of cube center.
- Leave perspective to the end, using orthographic projection to complete all other objectives.
- Perspective projection: don't need to use the matrix; just divide x,y by z. I.e., you don't use the depth information.
- Clipping: choices, choices, choices.
  - Clip to 6 sides of truncated viewing pyramid
  - Clip to near (and possibly far) clipping plane, project, then clip to sides (and possibly far)
  - Homogeneous clipping (clip in 4-space after multiplying by projective matrix but before normalizing by w).

## 34.4 Assignment 3: Hierarchical Modelling

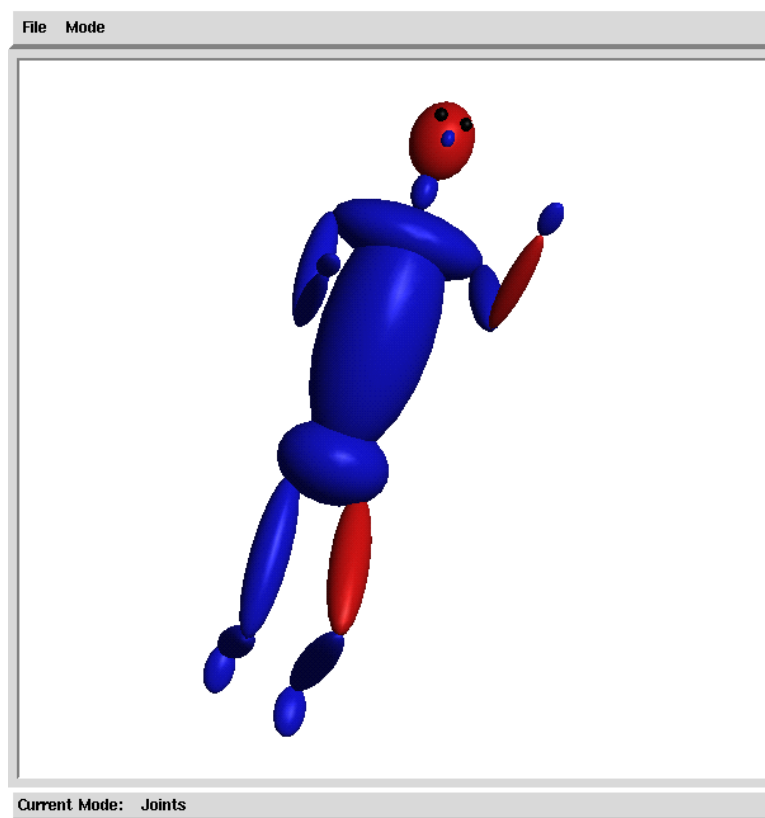
This assignment includes:

- Hierarchical Models (and data structures).
- Matrix Stacks
- 3D Picking
- Z-Buffer Hidden Surface Removal
- Lighting Models and Shaded Polygons
- Virtual Sphere or Arcball

You may use any feature of OpenGL you desire, including transformations and lighting models, to implement this assignment.

Lua should be used as the modelling language.

## Screen Shots



Sample screen shots. Some details may vary term to term.

### 34.5 Assignment 4: A Raytracer

- 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.

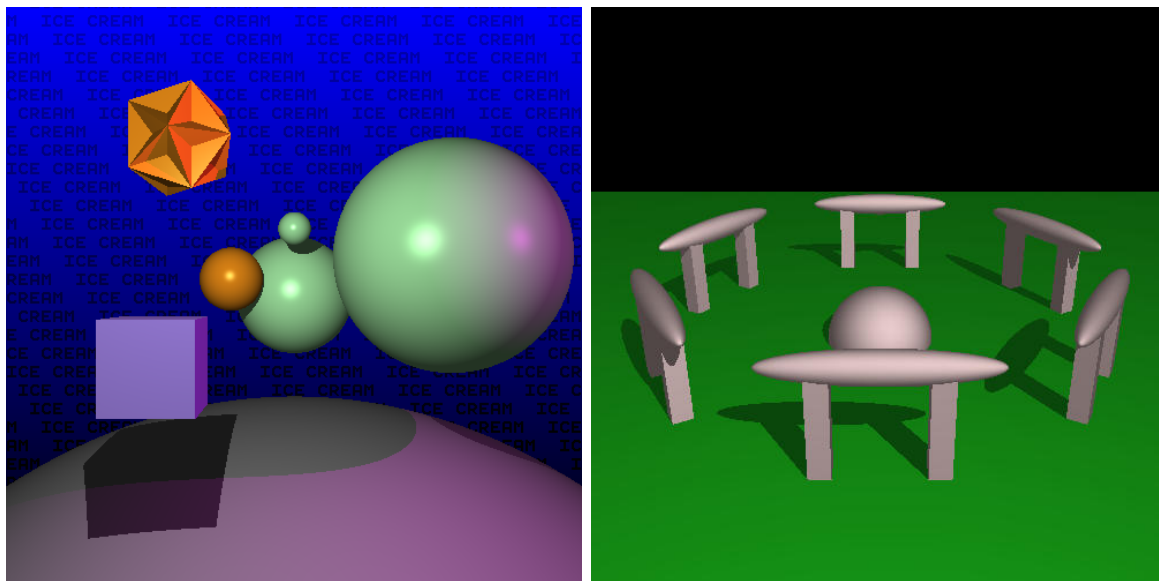
### Additional Features

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

## Screen Shots



Sample screen shots. Some details may vary term to term.

## 34.6 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.
- A project does not have to have an interactive user interface, but if not you will have to work harder on the algorithms
- 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.

Proposal Structure:

- Purpose — elevator pitch
- Statement — sit in my office description
- Technical Outline — details on objectives
- Bibliography
- Objective list

## Objective List

- Your proposal needs to include an Objectives list, which we will use to mark you.
- No objectives from assignments!
- Roughly...
  - 1/3 to 1/2 easy objectives  
Modelling, texture mapping, alpha blending
  - A few medium ones  
L-systems, reflection, 2D collisions
  - 1 or 2 hard ones  
3D collisions, hardware tricks, stuff from papers
- Try to break hard objectives into multiple pieces

## Common Objectives

- Modelling/scene
- Animation
- User interface (more complex interaction)
- Texture mapping
- Artificial Intelligence, sound  
If your other objectives had enough graphics content
- Ray tracer:
  - Must tell us your A4 extra feature
  - CSG
  - Reflection, refraction, anti-aliasing
  - Extra primitives (torus)

## Warm/cold Fuzzies About Objectives

- Think about what subjective mark you could get  
Technical, artistic, documentation, difficulty
- Negative subjective marks for not using what you've learned  
eg, flat shading in OpenGL projects
- Get advice from TAs and profs
- Beware of too easy/too hard projects  
You can always do extra stuff
- Make objectives, not subjective  
“Good UI” is subjective...
- When wondering if something makes a good project, try making an objective list for it.

## General Project Categories

- Virtual World Exploration  
Includes most games
- 2D Games (chess, risk, etc.)  
Usually hard to get 10 objectives
- Simulation (physics, cloth, etc.)  
Usually focussed on one harder technique/paper
- Ray Tracing
- Animated Movie  
Simple key framing, ray trace the frames

## OpenGL vs Ray Tracing

### OpenGL Projects

- Interactive
- Avoid UI intensive projects
- Try to have a purpose (game, etc)

### Ray Tracing

- Must get 9/10 on A4 to do ray tracing project
- Easier to come up with objective list



- **Design objectives around a scene**

Photograph or real object you want to ray trace

- Nice scene must be one objective!

Describe it

## Other Stuff

- Technical outline should clarify objectives

- You can start before you get proposal back

- Avoid dynamic memory

- See TAs if you have problems

Don't spend 2 days hacking at it

- If running out of time, prof or TA can give advice on what objectives to tackle

- **Project is marked on objective list of revised proposal.**