

CSCI 3090
Computer Graphics
and
Visualization

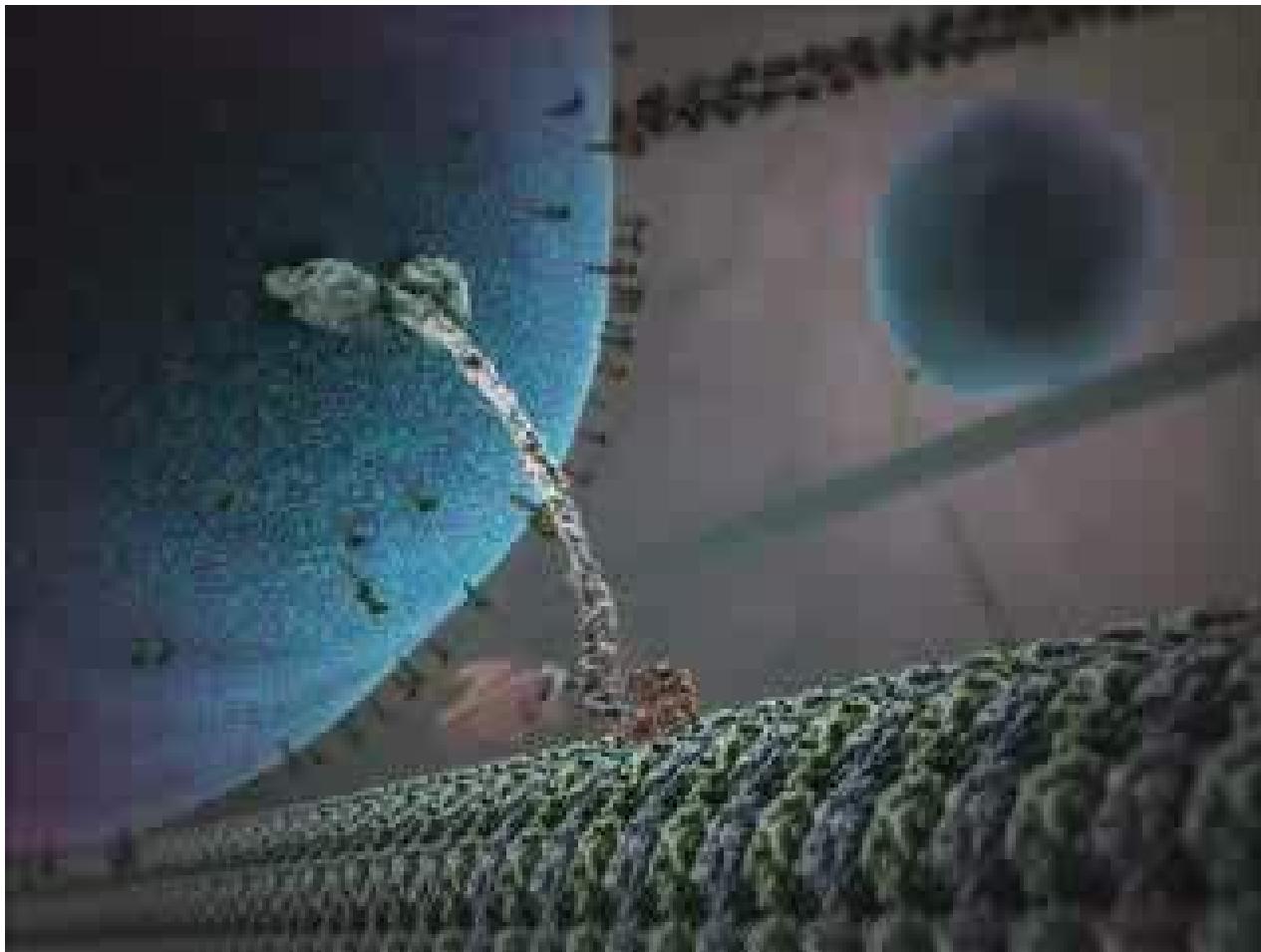
Mark Green
Faculty of Science
Ontario Tech

Note

- This is one of the first CS courses offered at Ontario Tech, it was part of the Computational Science minor
- Over the years it has evolved a lot, I've taught it many times, Dr. Chris Collins taught it several times, along with several other instructors
- Over time it sort of lost direction, which I discovered last year when I taught it again after many years
- This year I'm refocusing the course

What is Computer Graphics?

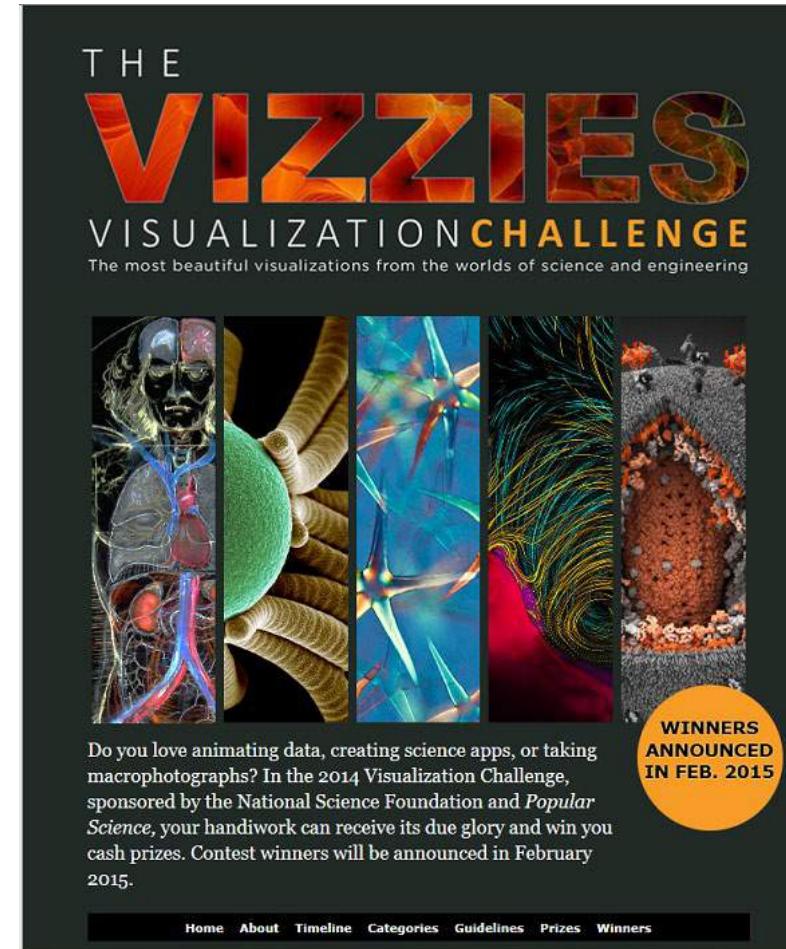
- Visual communications, how we present visual information to human users
- Storage, manipulation and display of geometrical information
- You've all seen examples of computer graphics, it's hard to miss in computer games, film and other media
- But, have you thought about how it's produced?
- This course is an introduction to how it's done



The Inner Life of a Cell, the BioVisions Lab,
Harvard University



NSF Visualization Challenge



Welcome!

- In today's class we will:
 - Get to know each other!
 - Review the course outline
 - Explore the nature of computer graphics

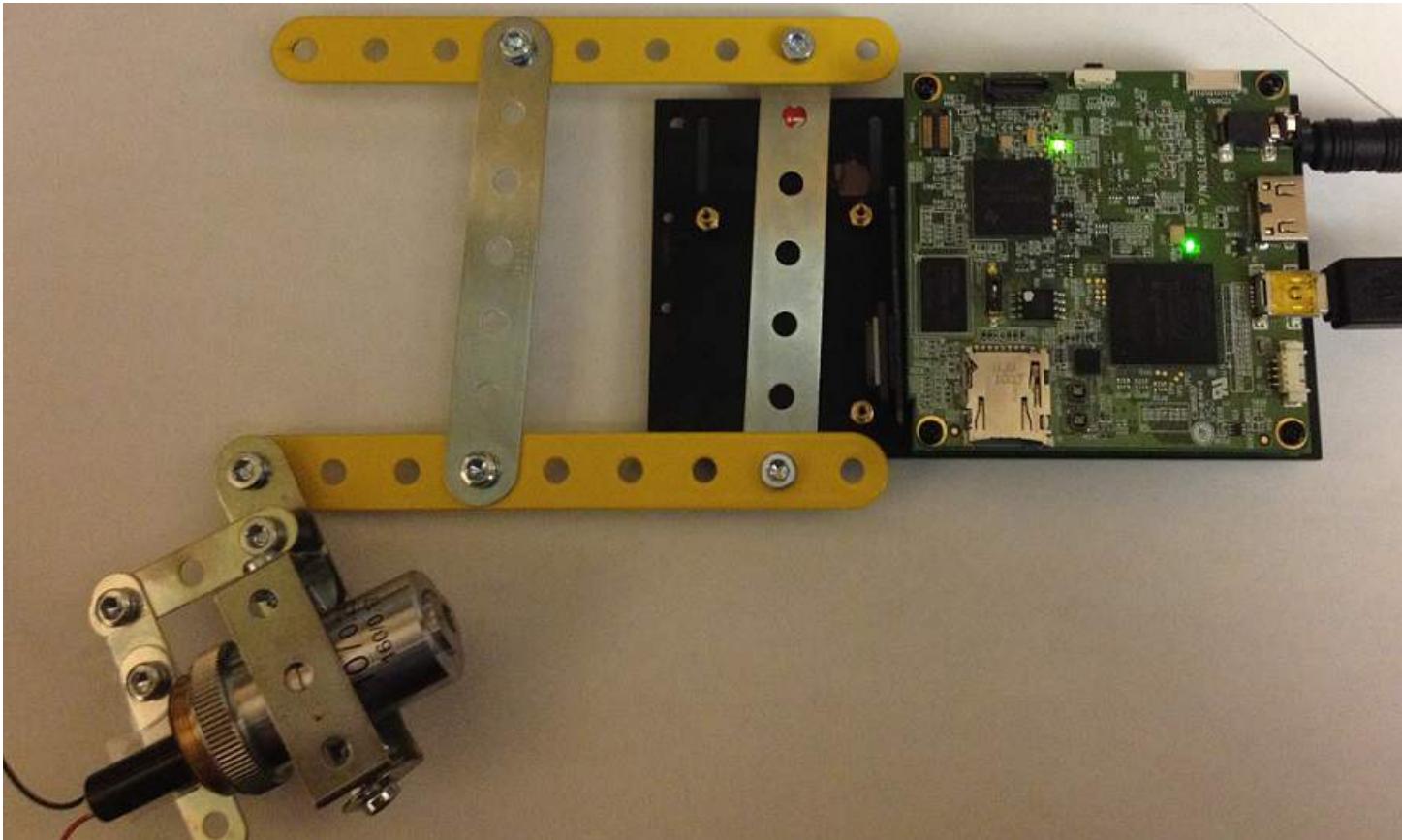
About Me

- First CS faculty member at Ontario Tech
- Previous universities:
 - McMaster University
 - University of Alberta
 - City University of Hong Kong
- Successful startup companies

Research

- Current interests:
 - 3D graphics hardware
 - Computational Holography
 - Quantum Computing
- Previous areas:
 - Graphics software
 - 3D interaction
 - Computer animation
 - Special effects
 - Visualization
 - Parallel and scientific computing
 - Mobile devices

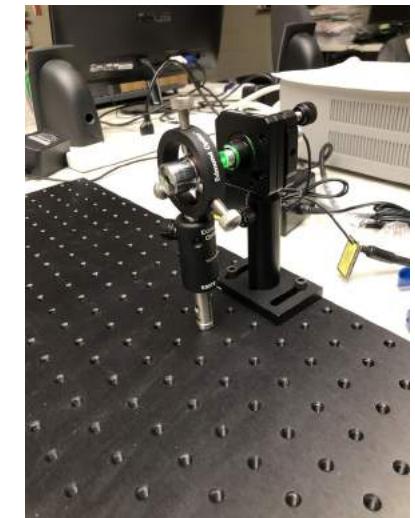
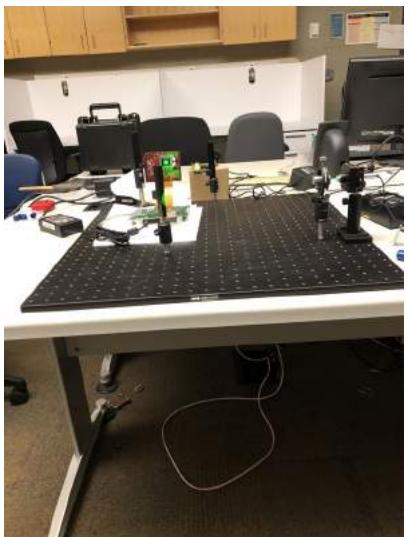
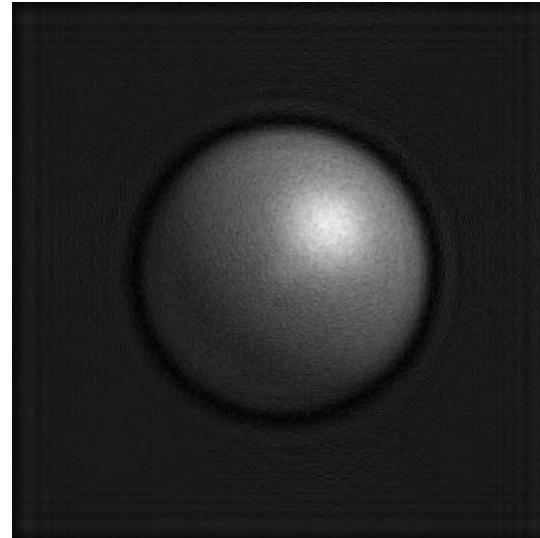
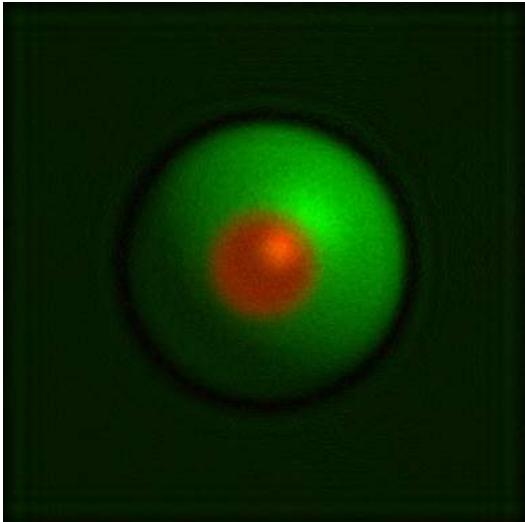
Research - 2016



Research - 2016



Research - 2019



About You!

- In the old days we would go around the class and ask a few simple questions:
 - What is your favourite CS course?
 - What do you expect to learn in this course?
 - What are your hobbies?
- The class is now far too large to do this, and it's hard to do this over Google Meet
- I really miss seeing students in person

About CSCI 3090

Course Philosophy

- There are many ways to teach computer graphics:
 - Theoretical and mathematical
 - Practical, concentrating on writing applications
- We will take a middle ground
- Will do some of the mathematics and theory, stick to basic linear algebra
- Learn how to develop graphics applications using OpenGL

Topics

- Graphics Pipeline
- OpenGL Programming
- Modeling
- Rendering
- Colour
- Ray Tracing
- Graphics Hardware and Performance
- Graphics Application Development

Evaluation

Item	Value
Labs	20%
Individual assignments	30%
Mid-term test	20%
Final Exam	30%
Total	100%

Evaluation

Item	Value
Labs	20%
Individual assignments	30%
Mid-term test	20%
Final Exam	30%
Total	100%

10 lab activities based on readings and lecture materials. Hand in a solution by the start of next week. No make ups, if excused will be marked out of labs completed or may be permitted to complete on your own time.

Evaluation

Item	Value
Labs	20%
Individual assignments	30%
Mid-term test	20%
Final Exam	30%
Total	100%

4 assignments based on readings and lecture materials.

These are individual assignments, with a strong programming component.

Evaluation

Item	Value
Labs	20%
Individual assignments	30%
Mid-term test	20%
Final Exam	30%
Total	100%

Covering material up to the lesson before the test.
Likely will take place mid March.

Evaluation

Item	Value
Labs	20%
Individual assignments	30%
Mid-term test	20%
Final Exam	30%
Total	100%

Cumulative final exam
(closed book)

Assignments

- Due at 11:59pm through Canvas
- Will be posted 2 weeks before they are due

Late Assignments

- Extensions on request with valid reason.
- Without reason:
 - Subtract 10% each day or part day (including weekends)
 - Maximum 3 days late, then not accepted

Exams

- Both the mid-term and final exam will be open book
- I'm not going to play the game of trying to make the exams secure, with multiple cameras, etc.
- This doesn't mean you don't need to study!!!
- I've been doing open book exams for many years, I know about Google
- If you think you can Google all the answers without studying you are in for a big surprise

Remarking

- “Mark, this is totally stupid. You didn’t tell us about X or Y, and anyway, I think I’m right.”
= 0% change
- It is very important that all assessments are fairly graded. If you think there is a problem, please submit an explanation, *by email*, within 7 days of receiving the grade.
- No requests accepted in class or more than 7 days later.
- Don’t come the day before the final exam and say I want everything regraded

Text Books

- There is no text book for the course, they are all too expensive, and a lot of them are out of date
- This is particularly true with OpenGL, where there was been a major change in the standard a little over a decade ago
- There is a lot of old material on the Internet that will lead you astray, so be careful

Laboratories

- There are 10 labs; on weeks where there is no lab activity planned, your TA will be available for a tutorial during the lab time.
- Labs will concentrate on programming techniques and graphics software.
- C and C++ programming Visual Studio (supported) or your preferred IDE (unsupported)
- Will provide some basic instructions for Linux and OSx

Prof Office Hours and Online Contact

- Thursday 12:00-13:00 at <https://meet.google.com/now-nhnw-ttk>
- I'm using Piazza for online discussion of course material
- I will try to respond to email within 24 hours

TA Office Hours

- Watch Canvas for an announcement

What might you come to talk
about?

Email

- Take time when composing an email - think of it as a professional message to a co-worker.
 - There won't be space for SMS-speak in your work life.
- Don't just say my program doesn't work, you need to be more specific
- Email turnaround:
 - Guaranteed: 2 days
 - Average: 1 day
 - Sometimes: 10 seconds
 - ... but don't count on that!

Academic Integrity

- Academic Integrity: All assignments and tests are to be completed independently unless explicitly stated on the assignment.
- You may discuss assignment questions, but each person must hand in their own work.
- Become familiar with:
<https://academicintegrity.ontariotechu.ca/index.php>

Accessibility

- Please speak to me as soon as possible.
- Accommodations can also be arranged through the SAS
- With everything being online, this is a very different world
- With Canvas I can automatically extend the time for exams
- All lecture material will be on Canvas well before the lecture time

What I expect of you...

- Come to class on time and prepared
- Read the assigned readings
- Participate in discussions in class and online
- Ask assignment-related questions early
- Do not spend class time surfing the Web or doing work for other courses
- No texting during class
- ??

You should expect from me...

- Knowledgeable and prepared for class
- Fair grading
- Responsive to comments and suggestions
- Timely return of assignments
- Keep things interesting and relevant
- Ensure a welcome and accessible classroom
- ??

You expect from each other...

- Participate fully in discussions
- Understand everyone has different abilities
- Encourage and organize to use strengths
- Welcome discussions and comments
- Pay attention to class presentations
(laptops closed)
- ??

Laptops In Class

Appropriate Uses

- Note taking
- Researching course topics
- Backchannel chat related to courses
- Participation in class activities

Inappropriate Uses

- Games
- Social networking
- Completing assignments for other courses
- Listening to music
- Watching YouTube

Participation

- Throughout the term we will engage in class discussions, in particular, at the start of class as we watch various short films.
- Please participate in these discussions as we endeavour to apply our graphics knowledge to interpret real-world examples.
- More people engaged = more enjoyable class = more learning for everyone!

OpenGL

- Many students find OpenGL to be difficult, I don't disagree
- OpenGL runs on two separate processors, on the CPU you write a program that prepares program code and data to run on the GPU
- This makes life more complicated
- The first example program is over 170 lines long, and the output isn't very interesting
- I won't be writing code line-by-line in class, it would be incredibly boring

OpenGL

- Instead I give you the code for all of the examples, well before they are discussed in class
- In class I will go over the important parts of the examples, but not line-by-line
- There is a lot of similarity between programs, so once I've described a particular technique, I won't cover it in detail in subsequent examples
- The concentration will be on the new code

OpenGL

- Lecturing will give you the basic ideas and theory, but you need to practice it
- This is what the labs are for
- Read the labs carefully and follow the instructions
- Some students will just cut-and-paste code without thinking about what they are doing
- Try to get the lab done as quickly as possible, this will catch up to you in the first assignment
- The labs are there to help you learn

What is computer graphics?

Computer Graphics

- Visual communications, computer <-> people
- Storage, manipulation and display of visual information
- Interaction with visual information
- Must be keen observers of the world, look at things differently

Challenges

- Photorealistic Rendering:
 - The classic computer graphics problem
 - Can we produce an image that is as good as a camera?
 - Over 4 decades of research, still some open problems
 - Can argue that we can now do better than a camera
 - Test: display an image captured by a camera and an image produced by a computer on the same display, can you tell the difference?

Challenges

- Virtual Reality:
 - Can we produce a computer simulation that is as good as reality?
 - About 3 decades of research, still many open problems
 - In a very small number of cases this is possible, depends upon your metric
 - Test: can you tell the difference between a monitor and a window on the real world

Challenges

- Real-time Rendering:
 - How good of an image can we produce in real time?
 - This is much harder than previous two challenges, since we can't pre-compute
 - Need to produce ~60 frames/second
 - Again 2 to 3 decades of research, but still many open problems

Challenges

- Visualization:
 - Communicate as much information as possible to the human brain
 - We have sophisticated visual processing, our best sense, how can we take advantage of it?
 - Images are not realistic, but informative
 - Again many decades of research

CSCI 3090

The Graphics Pipeline

Mark Green

Faculty of Science

Ontario Tech

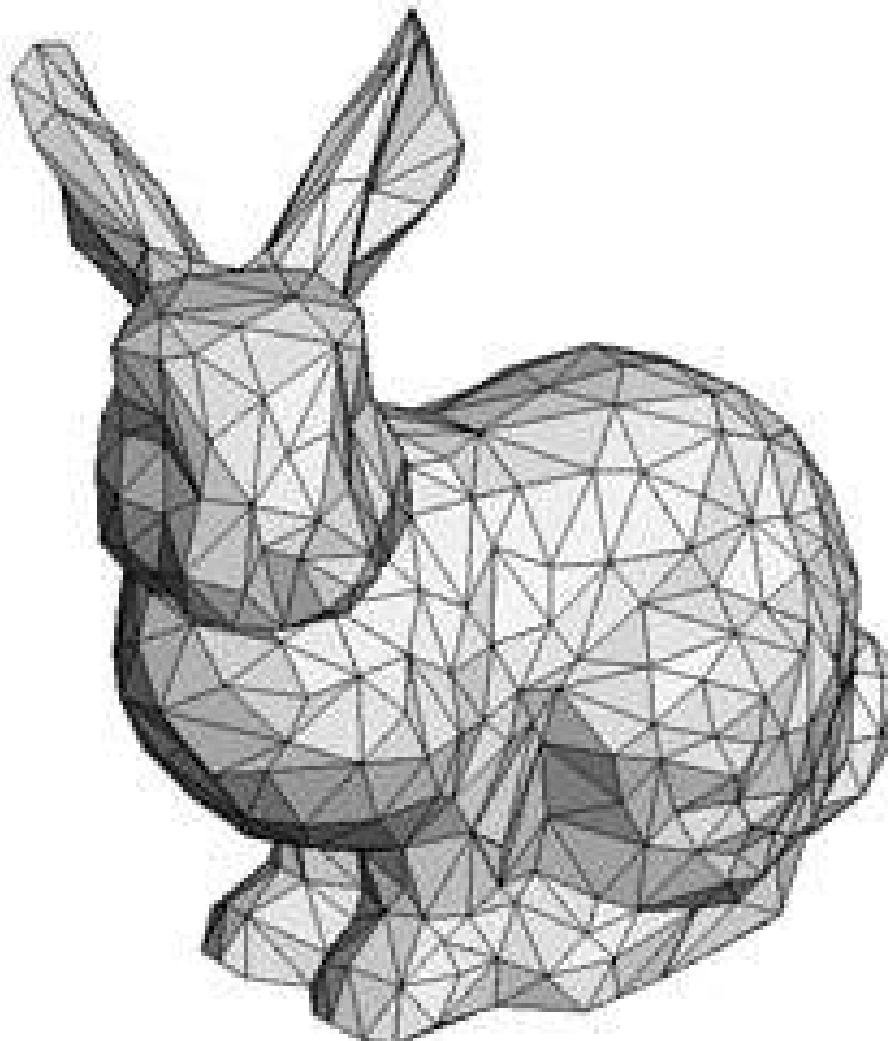
Luxo Jr.



Goals

- By the end of today's lecture, you will:
 - Know the key concepts in basic graphics
 - Understand the traditional graphics pipeline

What's the Share?



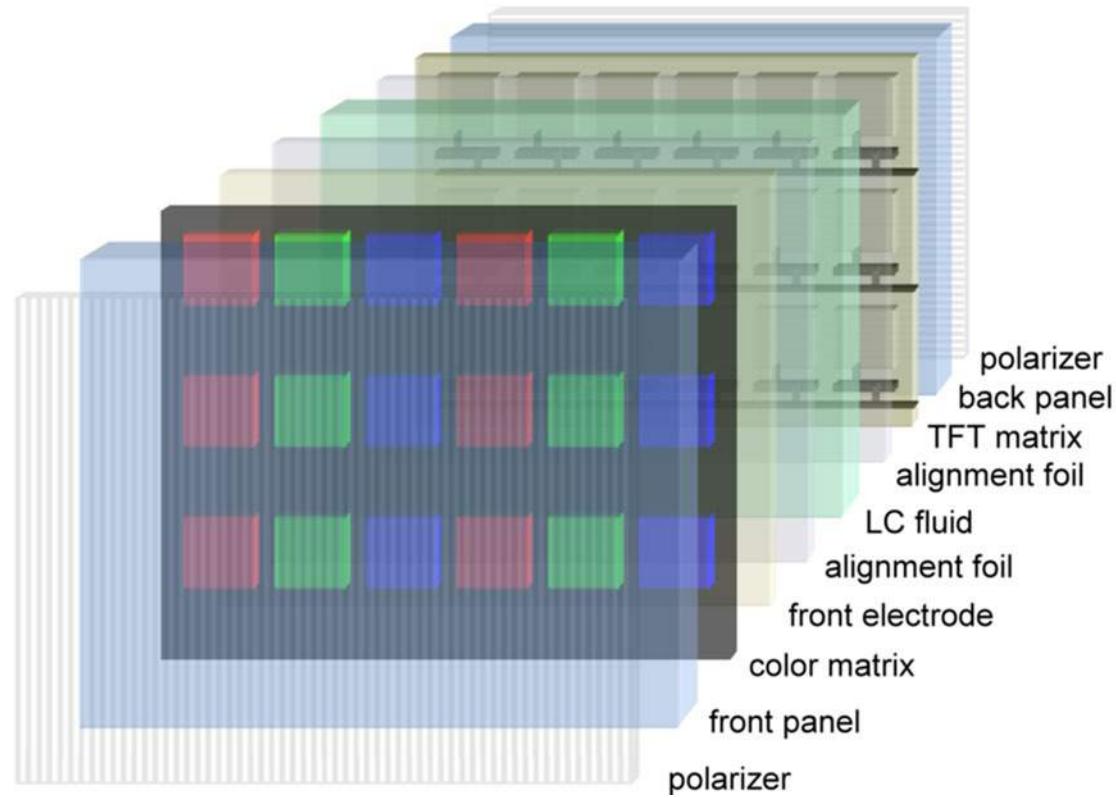
Displays

- We start with geometry, a collection of triangles
- But, we need some way of displaying them
- We need a display device!
- There are many technologies that we can use for display, will come back and examine some of them later
- For the time being, we will examine one, the LCD display, which is used in your laptops and monitors

Liquid Crystal Displays

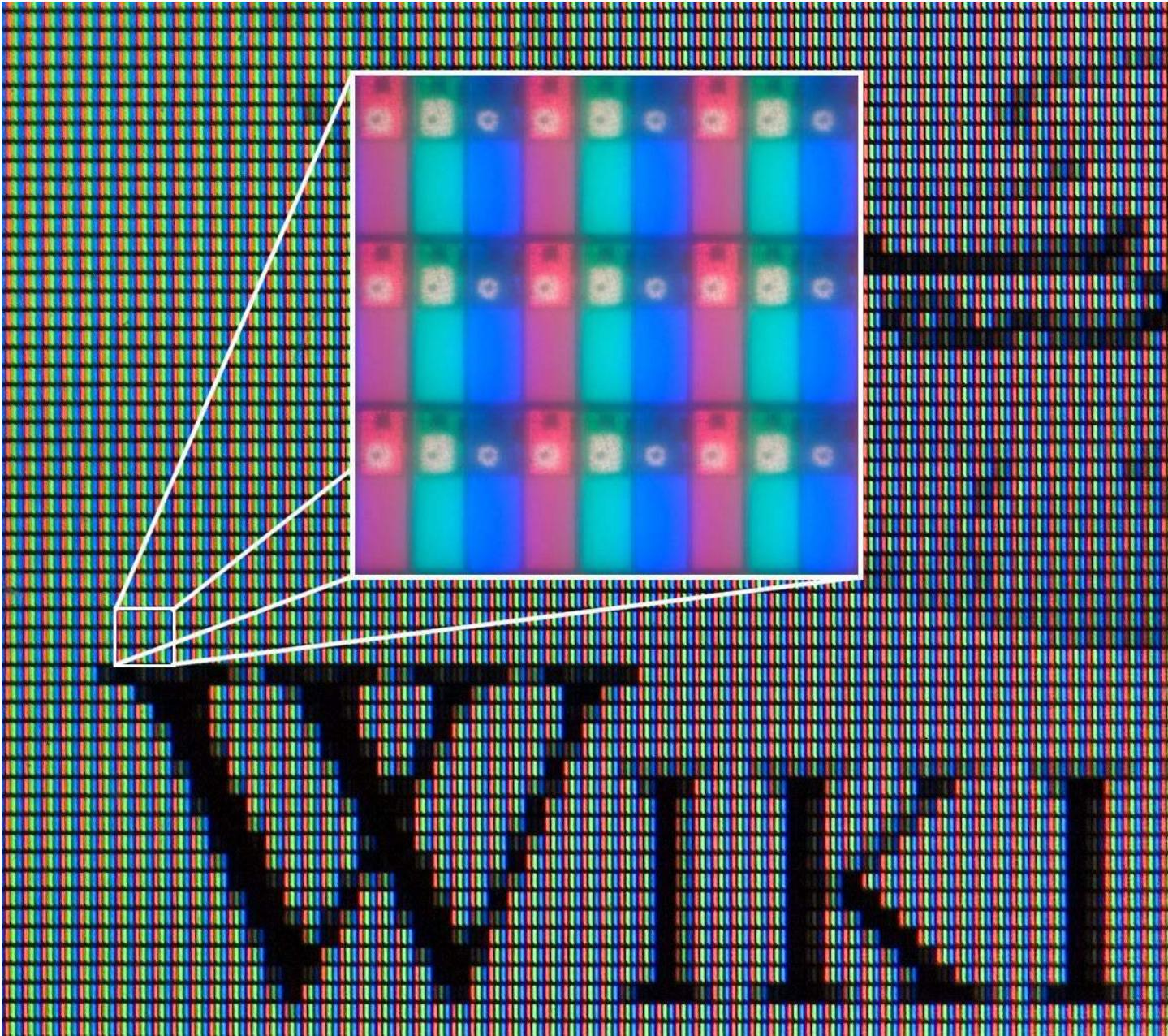
- Liquid crystals are large molecules that change their optical properties when an electrical field is applied to them
- There are many types of liquid crystals that behave in different ways, display manufacturers have labs that investigate them, we won't look at the details
- LCD displays are made up of many small cells containing liquid crystals
- An LCD is a rectangular array of these cells, with small wires running through the cells to apply the electrical field
- The size of this array is called the resolution of the display, and each of these cells is called a pixel, short for picture element

LCD Display

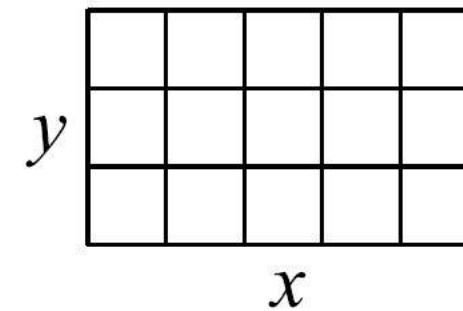


LCD Display

- Each cell or pixel is capable of displaying one colour, so they can be viewed as small rectangles with a solid colour
- We display an image by specifying the colour of each pixel on the screen
- There are approximately 1-2M pixels on the screen, so this is a lot of information to specify



Pixel Raster



Rasters

- View the pixels has a 2D array of cells, we need to provide a value for each of these cells
- The graphics card has a 2D array of memory, with one entry for each of these cells
- The array is called a framebuffer
- The graphics card produces an electrical signal that transfer the pixels from the frame buffer to the display
- Several standards for this: VGA, HDMI, DVI and display port

Liquid Crystal Display Resolution

- Each LCD monitor has a native resolution, the actual number of pixels in the display
- This is the resolution that you should use, you can change the display resolution in the operating system, but this will produce a blurry or hard to read image since the display must attempt to approximate the new resolution
- This was not the case with older CRT monitors which could handle a range of resolutions

Representing Colour

- How do we represent colour?
- Actually this is a hard problem that people are still working on
- Human vision is extremely complicated
- Our response to light intensity is not linear, it is logarithmic, this has an impact on display design
- Also, several different wavelengths of light can produce the same colour!

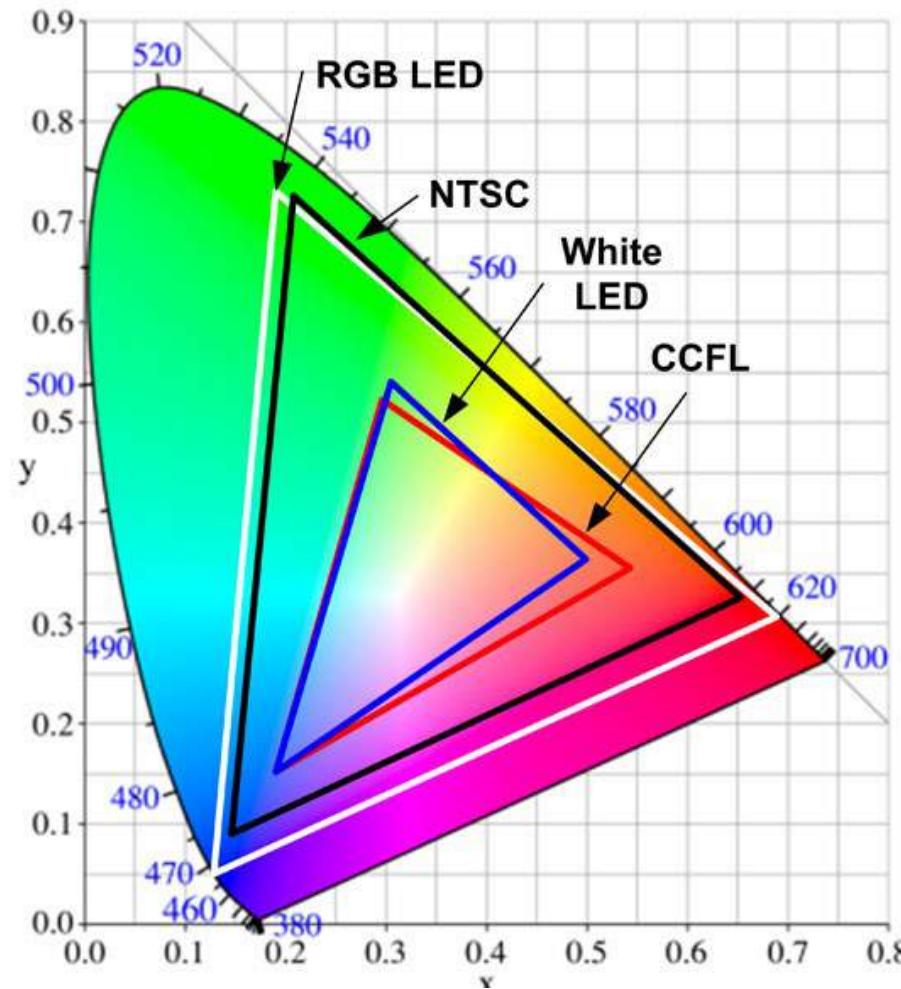
Representing Colour

- We use an approximation to represent colour, it is based on the red, green, blue, or RGB colour space
- A colour is a triple (r,g,b) , where r, g, and b are either integers (usually from 0 to 255) or floating point numbers (from 0 to 1)
- This triple gives the amount of red, green and blue in the colour

Display Gamut

- The total colour space is called a gamut
- Note that the RGB space doesn't include all of the colours, and is a different colour space than used with printers
- It turns out the RGB is a close match to the colours produced by monitors so it is used in computer graphics
- Note: we cannot use a primary based system to get all colours, unless we allow negative colours

Colour Gamut



Sending the Pixels

- Finally we need an organized way of getting the pixels onto the screen, the information that flows down the VGA/DVI/DisplayPort/HDMI cable
- We use a scanning pattern for this, the pixels are sent one at a time in an organized way
- We start with the pixel in the top left corner of the screen and send the first row of pixels, one at a time
- This is repeated row by row until the bottom row of the screen is reached

Sending the Pixels

- Once the bottom row is reached, we go back to the top of the screen and start over again
- This is similar to the pattern used in TV, but not quite the same
- At the end of each line and at the end of each screen there is a small amount of time that isn't used, this is required by CRT monitors to return the beam to the start of the line or the top of the screen, called horizontal and vertical retrace

Refresh Rate

- The number of times, per second, that we send the image to the screen is called the refresh rate
- This is at least 60Hz and is sometimes much higher
- This was very important for CRT monitors, but not much of a concern for LCDs
- The graphics card will generate all of the signals required to drive an LCD or CRT, so we don't need to worry about this

Rendering

- Rendering is the process of converting geometrical information, called the model, into the pixels in the framebuffer
- There are many rendering algorithms that have been developed over the decades
- There are also many ways of classifying rendering algorithms
- One of the important ones is whether the algorithm is real-time
- For games and other interactive applications we want a real-time algorithm, must generate many frame per second

Rendering

- On the other hand, if I'm producing special effects for a movie I really don't care how fast the rendering is
- I want the images to look good
- There is a trade-off between speed and quality, it's very hard to get both
- There are two basic ways in which a rendering algorithm can work, what we call forward rendering and backward rendering

Forward Rendering

- With forward rendering start with a model and then determine the pixels covered by the model:
 - We have hardware that does this very efficiently, real-time
 - It only involves simple mathematics, linear algebra
 - The algorithms are relatively simple
- This is what OpenGL does and we will concentrate on this approach

Backward Rendering

- Start with the individual pixels and determine the objects in the model that contribute illumination to the pixel:
 - Very high quality
 - Very complex mathematics, advanced calculus
 - Complex algorithms
- We will only examine the simplest backward rendering algorithm, ray tracing

The Graphics Pipeline

Graphics Pipeline

- Now concentrate on forward rendering
- Go from a geometrical model to pixels
- This is a complex process, so we will divided it into a number of steps
- These steps form the graphics pipeline
- Each step performs one part of the transformation, can concentrate on a particular type of computation
- Facilitates implementation in hardware

Models

- Need to have some representation inside the computer of the object we are producing an image of, this is called a **model**
- Model is usually mathematical in nature, but in a digital form, something that we can store and manipulate on a computer
- There are many possible representations, we will study some of them later in the course
- In our first assignment, the bunny is a collection of triangles

Models

- Triangles are handy, we can use them to represent almost anything, as long as we use enough of them
- Triangles are also quite simple, they are easy to manipulate and work with, we understand their mathematics
- But, we often need a lot of them, the bunny has 69,451 triangles

Models

- It is not unusual to have models with millions of triangles
- This takes up a lot of memory, and it can be slow to manipulate
- Dealing with large models is an active research area, and an important one as well
- So now we have the model, what's next?

Graphics Pipeline

- So we now have the two ends of the graphics pipeline:
 - Model - triangles
 - Display - pixels
- Now we need to be able to convert the model into pixels, and we want to do this as efficiently as possible
- This is the purpose of the graphics pipeline

Graphics Pipeline

- The triangles are 3D and the image is 2D, this is the first problem we need to solve
- We need to **project** the 3D triangles onto the 2D screen space
- There are a number of projections that we could use for this, all of them work in basically the same way, as we will see later

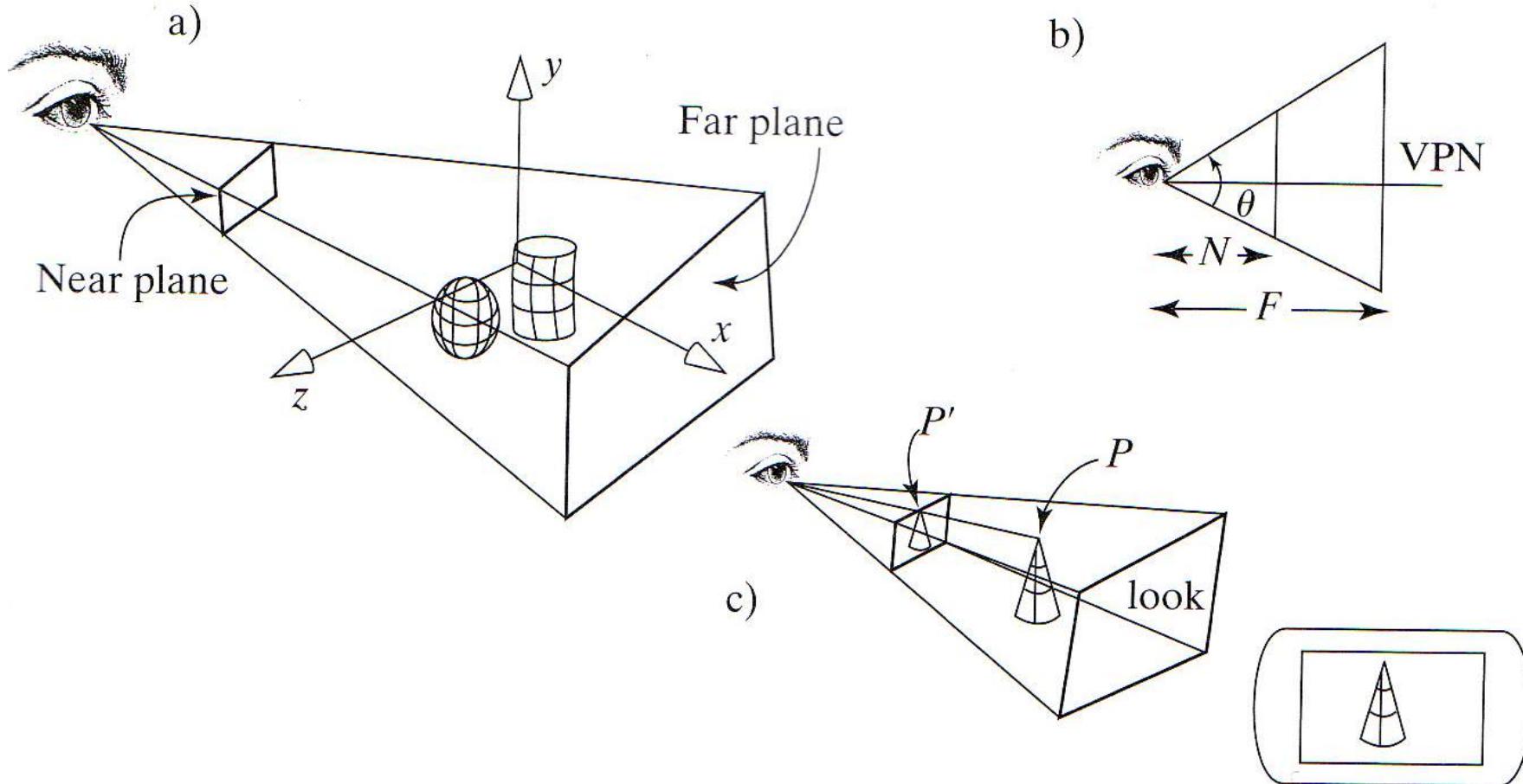
Graphics Pipeline

- We will use a **perspective projection**, since it gives the most realistic appearance
- This is similar to the projection that our eyes use
- When we perform a perspective projection we need to know where we are projecting from, that is where is the eye located
- We also need to know the direction that we are looking

Graphics Pipeline

- The viewing transformation is made up of:
 - Projection, in our case perspective
 - Eye location
 - Viewing direction
 - Up direction
- In the case of a perspective projection this defines a [viewing frustum](#), a pyramid with its apex at the viewer's eye and its axis along the viewing direction
- All the triangles inside the viewing frustum will be visible on the screen, those outside will not

View Frustum

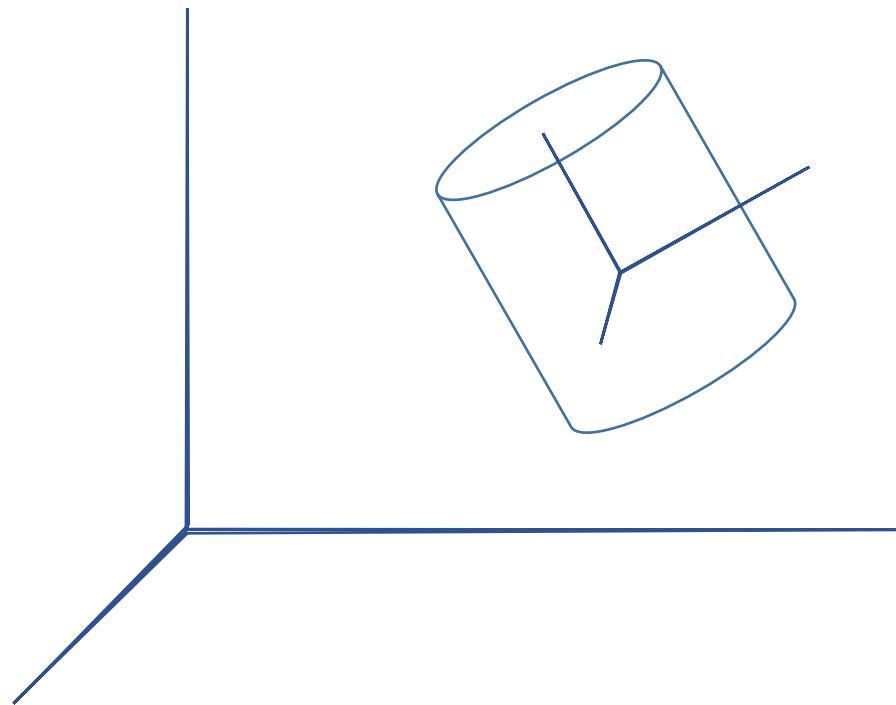


Graphic Pipeline

- We need to be careful to remove the triangles outside of the viewing frustum
- If we don't they will appear on the screen in incorrect positions, basically they will confuse the algorithms and hardware
- The process of eliminating them is called clipping
- Note that some triangles may intersect the viewing frustum, in this case we just process the visible part of the triangle

Local and Global Axes

- Global: “World” Frame of reference
- Local: “Object” Frame of reference



Graphics Pipeline

- We need to have a **colour for our triangles**, otherwise nothing will appear on the screen
- But, mathematically triangles don't have colour, they are just geometry
- We solve this problem by defining a **lighting model and material properties** for our objects

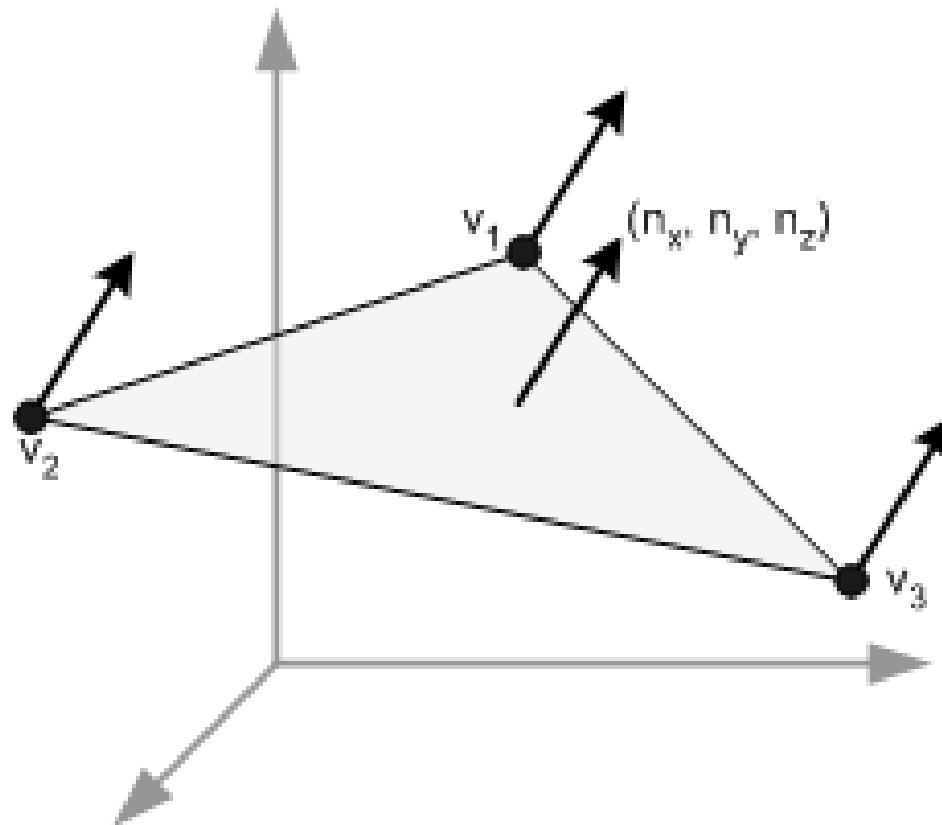
Graphics Pipeline

- In the real world we use lights to illuminate things, we do the same thing in graphics
- We give the **lights a position and/or direction** in space, and we can also assign a **colour** to them (more later)
- We also need to assign material properties to the triangles, how they interact with light, how much of the light gets reflected

Graphics Pipeline

- One of the most important properties that we need are **normal vectors**, they determine how light is reflected off of the triangle surfaces
- Since each triangle is flat its easy to compute a normal vector for it
- We then compute a normal vector for each vertex, by averaging the normal vectors of the triangles it appears in

Normal Vector

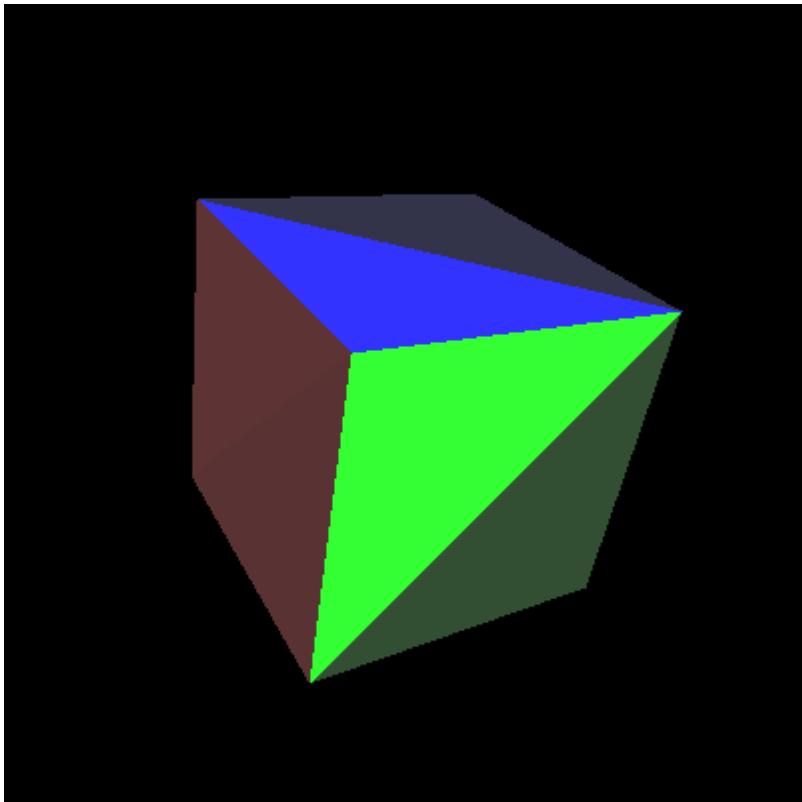


Graphics Pipeline

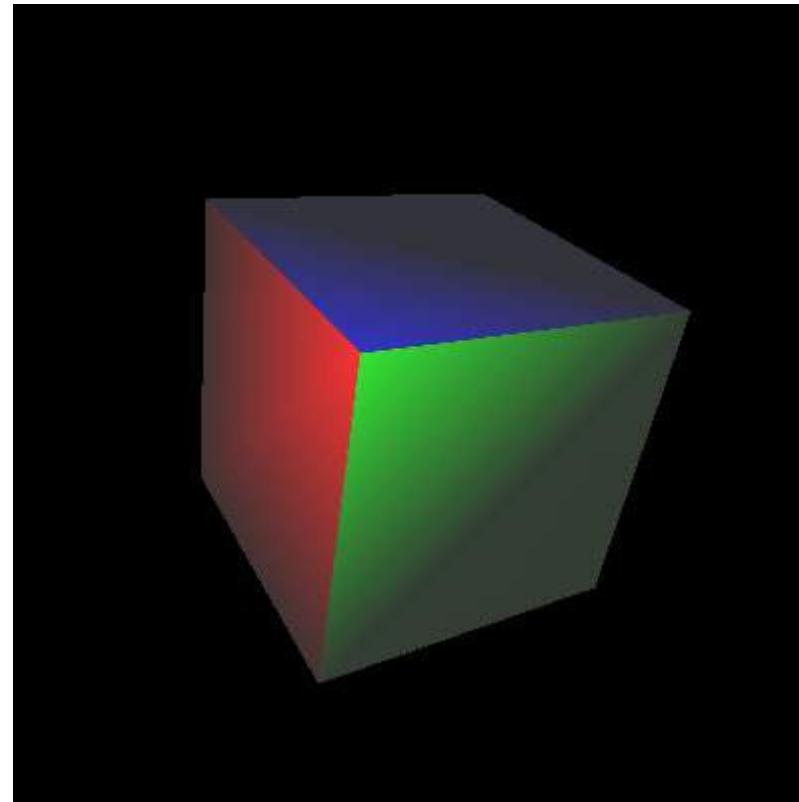
- Thus for each vertex of a triangle we have its position and its normal vector
- Along with the material properties this allow us to compute the colour at each vertex
- We can then **interpolate** the vertex colours across the surface of the triangle to get a colour at every position
- This is called **Gouraud shading**

Shading

Flat



Gouraud



Graphics Pipeline

- We now have the first few stages on the graphics pipeline
- For each triangle we need to do the following things:
 - Project the vertices
 - Clip the triangle to the viewing frustum
 - Compute colour values at each vertex
- Only the second operation needs all the triangle information, otherwise each vertex can be processed independently

Graphics Pipeline

- Note that all of these operations are geometrical in nature, they are independent of the pixels on the display
- Most of the operations just involve the triangle vertices
- We will write GPU programs for doing this processing, they are called vertex programs
- This is viewed as the first stage in our pipeline

Graphics Pipeline

- We now have the projected triangle, all its vertices will lie on the display surface
- But, the display surface is made up of pixels, we need to know which pixels to set for the triangle
- This is called filling, we set all of the pixels inside of the triangle's outline
- There are efficient algorithms for this that are implemented in the GPU hardware

Graphics Pipeline

- As we fill each pixel we compute its colour by interpolating the vertex colours that we computed earlier
- What have we overlooked?
 - We are projecting many triangles onto the same plane
 - In 3D space, some of these triangles are in front of other triangles, so they completely or partially hide the other triangles

Graphics Pipeline

- In the case of the bunny, we only see the triangles that are on the closest side, we don't see the ones on the other side
- The surfaces of solid objects hide the other surfaces that are behind them
- If our image doesn't have this property then it will not look very realistic
- This has been called the **hidden surface problem** and many algorithms have been developed for solving it

Graphics Pipeline

- Current graphics cards solve the hidden surface problem in hardware
- A common technique is based on a **depth buffer or z buffer**, which is an addition to the frame buffer that holds the image
- The depth buffer is usually a 16 or 24 bit number that holds the current depth of the corresponding pixel.
- If a potential pixel is closer to viewer, draw it and update the depth buffer.

Graphics Pipeline

- When a vertex is projected onto the display screen, the screen coordinates are called the x and y coordinates of the vertex, they form the 2D coordinates of the vertex
- We can also produce a third coordinate, called the depth of the vertex, the distance from the viewer to the vertex
- This information is interpolated across the triangle in the same way as the vertex colour

Graphics Pipeline

- At the start of computing each image the z buffer is initialized to the largest possible value
- Before we set the value of a pixel we first check the z buffer
 - If pixel depth is less than the current contents of the z buffer, draw the pixel and update the z buffer
 - Otherwise the pixel isn't drawn
- In other words, if the new pixel is closer to the viewer we draw it, otherwise we skip it

Graphics Pipeline

- We are performing a simple comparison that enables or blocks the writing of a pixel
- This is very efficient and more or less comes for free
- This is now the standard way of doing hidden surface removal
- At this point we have our image on the screen

Graphics Pipeline

- The second set of steps in our pipeline are:
 - Compute the pixels covered by the triangle
 - Interpolate the colour of the pixel
 - Interpolate the depth value of the pixel
 - Perform hidden surface using z buffer
- Note that all of the pixels in the triangle can be processed in parallel, since they don't depend on each other

Graphics Pipeline

- Note that all of these operations involve pixels, we call them fragments
- The fragments contribute to the pixels on the display, for what we are doing fragments and pixels are the same thing
- In more advanced applications, they are different
- We call this part of the pipeline fragment processing
- Again, we will write fragment programs that run on the GPU

Graphics Pipeline

- Note that we have divided the graphics pipeline into two parts:
 - The first part deals with geometrical information, the information associated with the triangle vertices
 - The second part deals with the individual pixels covered by the triangle
- This is a common division that we will see throughout the course

Graphics Pipeline

- The graphics pipeline is important for two reasons:
 - It provides an outline for most of the topics that will be covered in this course
 - It provides a model for graphics programming, which we explore in the next part of the course

Action Items

- Get your programming environment set up
- You should already have a version of Visual Studio installed on your laptop, if not get one of the free ones
- From the resources module on Canvas you will find the OpenGL.zip file that includes the glfw, glew and glm packages
- Download them and install them in the C:\CSCI 3090
- This will makes life simpler when we get to OpenGL examples in the next lecture

Summary

- Examined the use of LCDs as a display device, the basic ideas behind display
- Introduced different types of rendering algorithms
- Introduced the graphics pipeline
- In the next lecture we will start OpenGL and you will see the graphics pipeline in action

CSCI 3090

OpenGL Programming

Mark Green

Faculty of Science

Ontario Tech

Goals

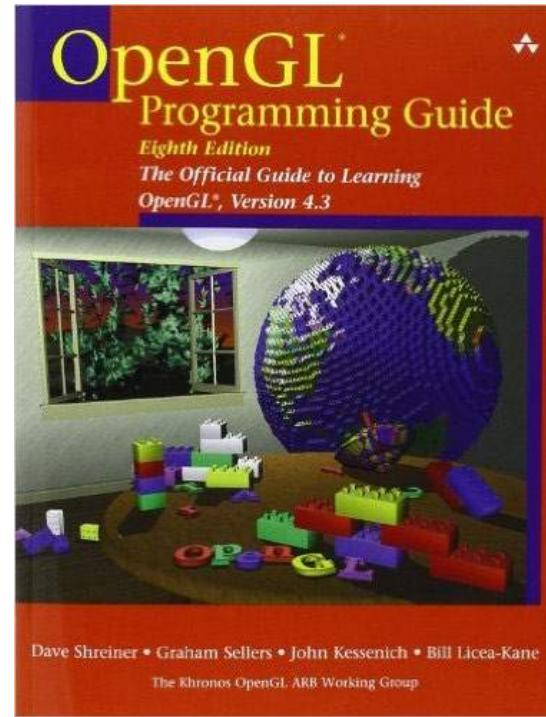
- At the end of this part of the course you will be able to:
 - Implement simple graphics applications using OpenGL
 - Utilize support libraries that simplify the development of graphics applications
 - Understand the structure of modern graphics applications

Why OpenGL?

- Widest range of platforms:
 - High end graphics workstations to cell phones and cars to web browsers
- Open standard, can be implemented anywhere
- By far the most popular 3D graphics package

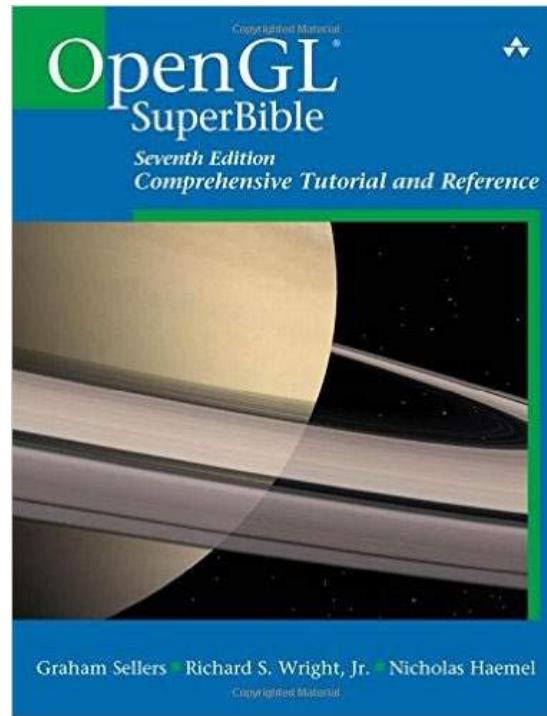
Red Book

- The OpenGL Programming Guide has been viewed as the standard reference on OpenGL programming
- Original versions written by standard authors, so very reliable
- Not always the case now
- It is call the “Red Book” due to the cover colour, this is how people refer to it



OpenGL SuperBible

- Over the years this has developed into quite a good book
- I now think it is better than the “Red Book”, though it may not be quite as up to date
- It is a better book to learn from, and all of their examples work!



Introduction

- Graphics hardware has changed significantly over the past decade
- Graphics software has changed in response
- We can write more powerful and efficient programs, but it is a steeper learning curve
- The first OpenGL program used to be fairly simple, few new concepts
- Now we need a lot of new concepts, even for the most basic program

Introduction

- To make life easier we will be using three libraries that simplify application development:
 - glfw: simple application framework for OpenGL programs
 - glew: configure the OpenGL library for our program
 - glm: matrix package

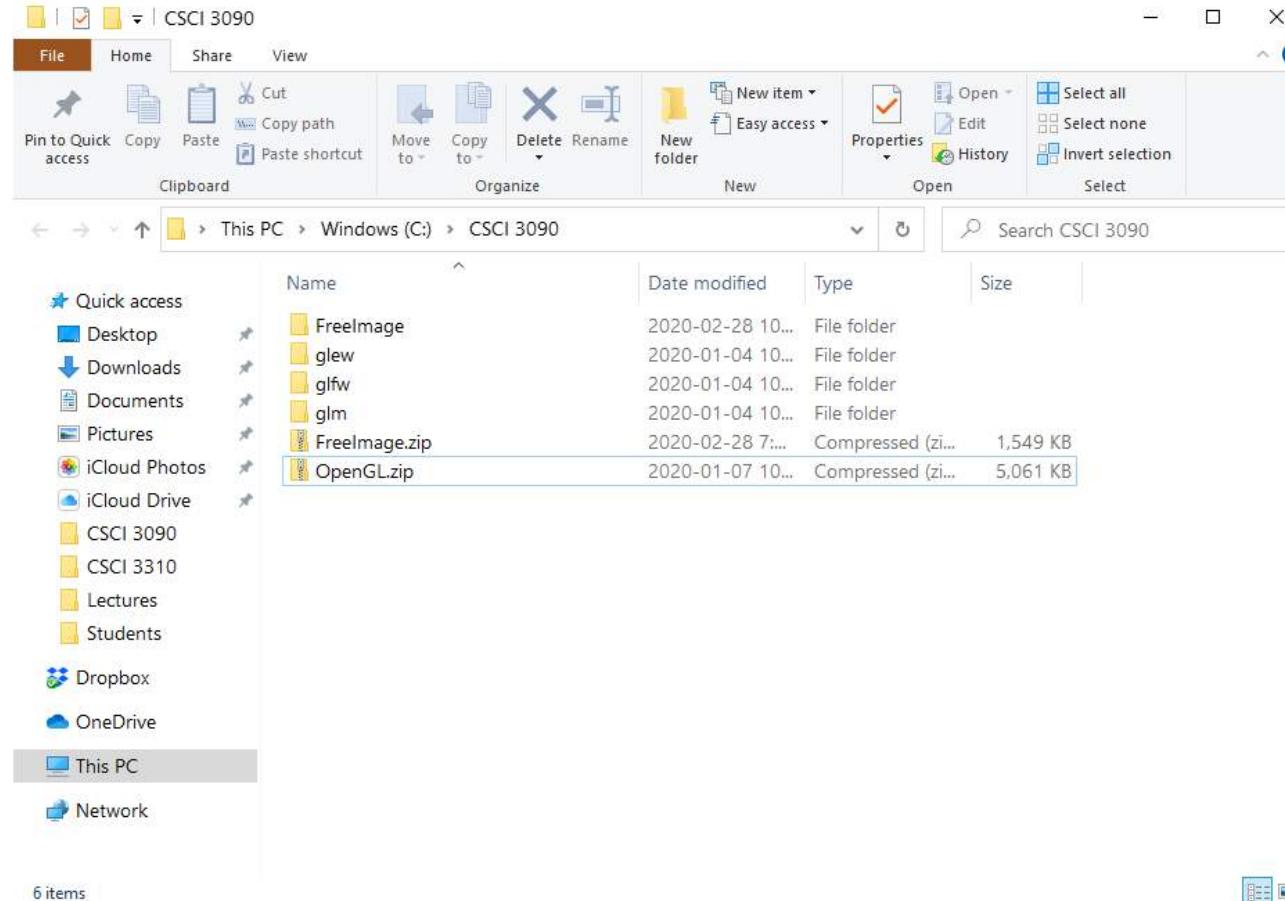
Introduction

- All the examples are on Canvas so you can run them yourself
- These examples are Visual Studio 2017 projects, but they will also work with Visual Studio 2019
- These projects expect a certain directory structure, which you should mirror on your laptop

Introduction

- Create a CSCI 3090 folder on the C drive
- Download the glfw, glew and glm libraries from the resources module
- There is one zip file that needs to be expanded, it should be expanded directly into the C:\CSCI 3090 folder
- The following slide shows the directory structure, you can add FreeImage later, we won't need it for several weeks
- You can create an Examples folder any place you like, this is where you will download the examples

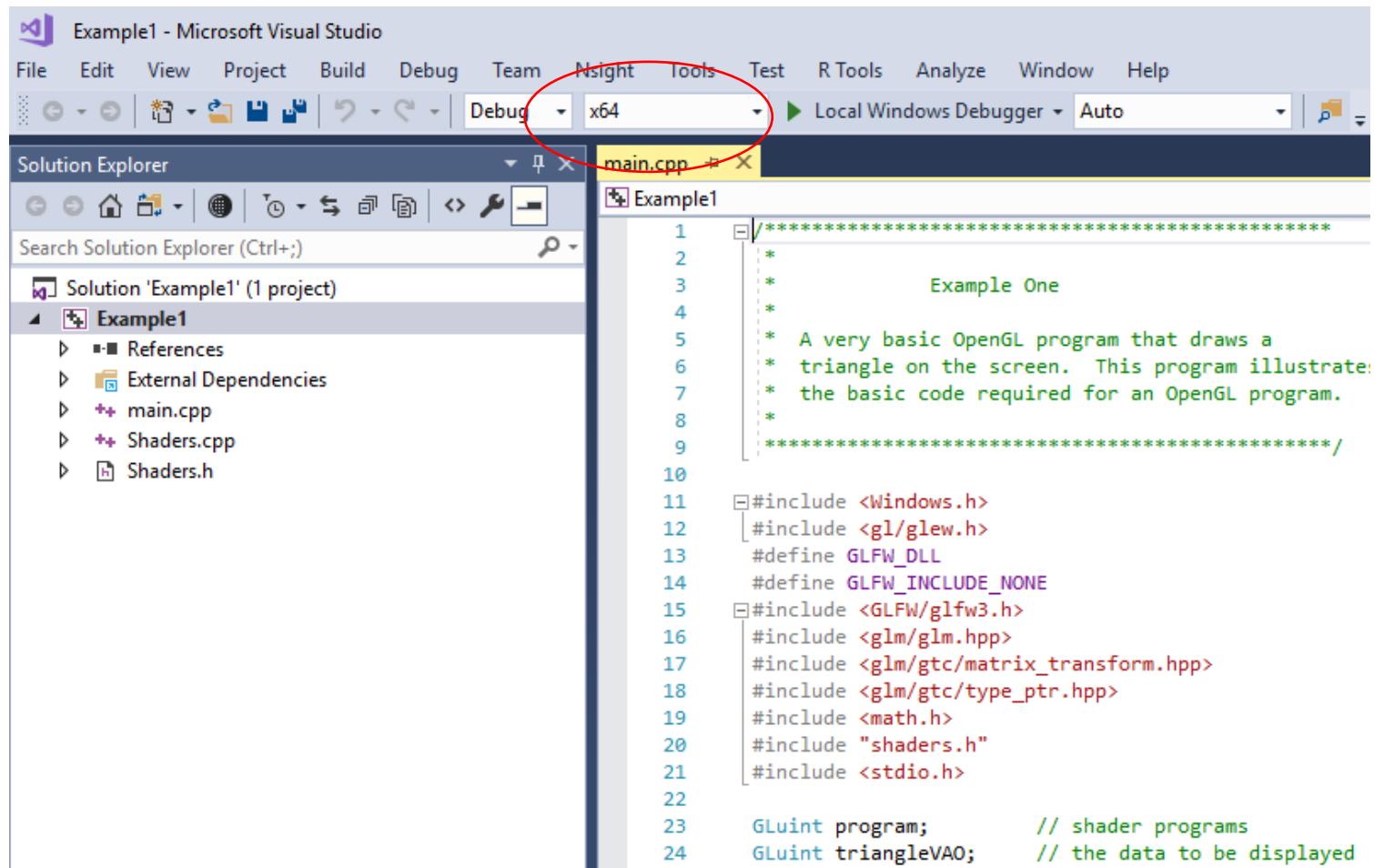
Introduction



Introduction

- Under both glfw and glew you will find include and lib
- We will be using the x64 versions of these libraries, note that this is not the default for Visual Studio, you will need to check this before you build
- Both of these libraries have dll's, which you will need to copy to the folder containing the executables for your programs

Introduction



The screenshot shows the Microsoft Visual Studio interface for a project named 'Example1'. The top menu bar includes File, Edit, View, Project, Build, Debug, Team, Nsight, Tools, Test, R Tools, Analyze, Window, and Help. The 'Debug' menu is highlighted with a red circle. Below the menu bar is the toolbar with various icons. The Solution Explorer on the left shows the project structure with files: Solution 'Example1' (1 project), Example1, References, External Dependencies, main.cpp, Shaders.cpp, and Shaders.h. The main code editor window displays the 'main.cpp' file. The code is a basic OpenGL program that draws a triangle. It includes includes for Windows.h, gl/glew.h, GLFW.h, glm/glm.hpp, glm/gtc/matrix_transform.hpp, glm/gtc/type_ptr.hpp, math.h, shaders.h, and stdio.h. It also defines GLFW_DLL and GLFW_INCLUDE_NONE. The code uses GLuint variables for shader programs and triangleVAO.

```
*****  
*          Example One  
*  
*  A very basic OpenGL program that draws a  
*  triangle on the screen. This program illustrate:  
*  the basic code required for an OpenGL program.  
*  
*****  
#include <Windows.h>  
#include <gl/glew.h>  
#define GLFW_DLL  
#define GLFW_INCLUDE_NONE  
#include <GLFW/glfw3.h>  
#include <glm/glm.hpp>  
#include <glm/gtc/matrix_transform.hpp>  
#include <glm/gtc/type_ptr.hpp>  
#include <math.h>  
#include "shaders.h"  
#include <stdio.h>  
  
GLuint program;      // shader programs  
GLuint triangleVAO; // the data to be displayed
```

Introduction

- What if I am not using Windows??
- All three libraries are available under Linux
- First check for pre-compiled packages
- If not available you will need to build them
- Glm is include files only, so nothing needs to be done for it
- There are instructions for doing this in the Resources module

Hardware Evolution

- Originally each stage of the graphics pipeline was implemented by a separate hardware unit
- Programmer had little control over hardware, optimized for average application
- Started introducing small amount of programming for pixel operations, called fragment programs
- Then added vertex programs for processing individual vertices

Hardware Evolution

- Programs were short assembly language, only used for specialized applications
- Soon evolved to larger programs and programming languages
- Having separate vertex and fragment processors caused performance problems
- Now have just one type of processor, can handle all types of programs

Hardware Evolution

- Have added other types of programs, but won't discuss them now
- OpenGL has its own programming language for vertex and fragment programs, GLSL, similar to C
- Compile GLSL programs within the OpenGL program and download them to the GPU (Graphics Processing Unit)

Hardware Evolution

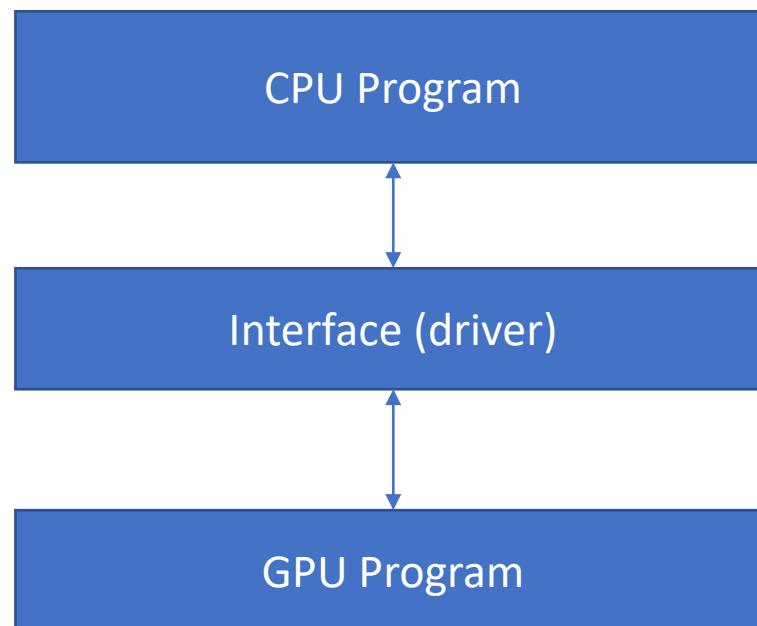
- Most GPUs have hundreds of processors
- Process many vertices and fragments in parallel, this is where performance comes from
- We need to prepare the data to be sent to the GPU and the programs that will process it
- This requires a bit of code

Multiple Processors

- Key thing to remember we are running programs on two processors:
 - OpenGL on the CPU
 - GLSL on the GPU
- Both processors execute at their own rate with little interaction
- We set up the GPU program through our CPU program and then let it run
- This makes debugging difficult, have no way of knowing what's gone wrong on the GPU, more on this later

Multiple Processors

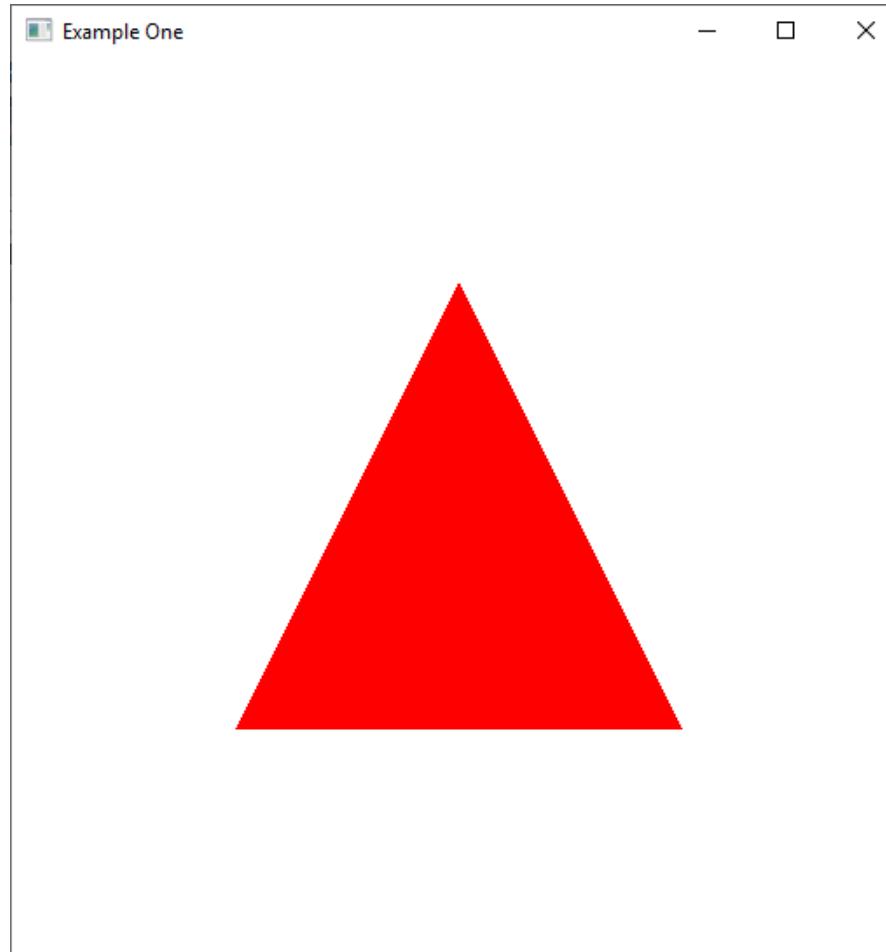
- Really dealing with two programs, one on CPU and one on GPU



Example One

- Our first example displays a single 2D triangle on the screen
- This isn't very interesting, but it illustrates the basic structure of an OpenGL program
- The main things that we need to do to get something displayed on the screen
- Later we will look at 3D

Example One



Example One

- Our triangle has 3 vertices, we need to have their coordinates and pass them to the GPU
- We pass this data to the GPU in a buffer, basically an array of vertices
- We also need to tell the GPU the vertices that are in each triangle
- We use another buffer for this
- We package this as an array buffer object

Example One

- This is done in our init() procedure, called once at the start of our program
- The first few lines of this procedure is shown on the next slide
- Start with some declarations, note that OpenGL defines some of its own types
- We then create our vertex array object
- We first create an integer identifier for this object and then bind it

Example One

```
GLuint vbuffer;
GLint vPosition;
int vs;
int fs;

glGenVertexArrays(1, &triangleVAO);
 glBindVertexArray(triangleVAO);

GLfloat vertices[3][2] = { // coordinates of triangle vertices
    { -0.5, -0.5 },
    { 0.0, 0.5},
    { 0.5, -0.5}
};

GLushort indexes[3] = { 0, 1, 2 }; // indexes of triangle vertices
```

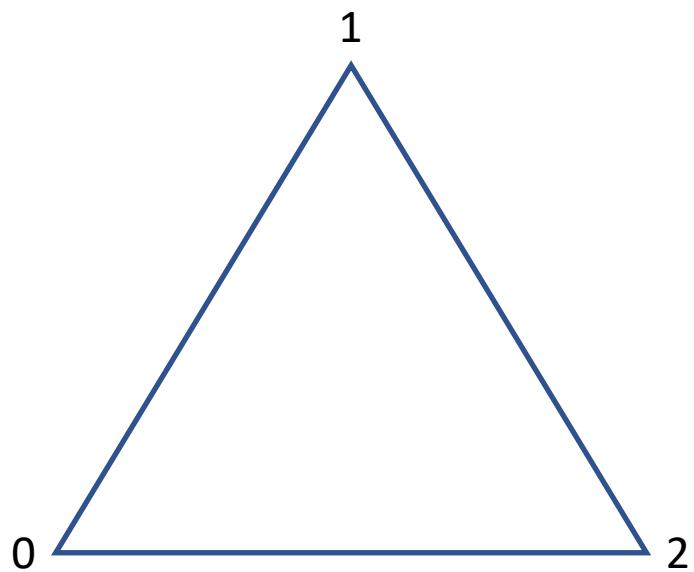
Example One

- We have no direct access to any of the OpenGL data structures
- Most of them are kept internal in the driver, or even on the GPU
- We reference these resources using an integer ID, we will use this for all the resources in OpenGL
- The call to `glGenVertexArrays` has two parameters, the first is the number of IDs to be generated, and the second is where these IDs will be stored
- The `glBindVertexArray` call specifies the vertex array that we will be manipulating

Example One

- The vertices array contains the 2D coordinates of the three triangle vertices
- The indexes array contains one entry for each triangle vertex, it gives the index in the vertices array for that vertex
- This makes more sense when we have larger models where a vertex can appear in more than one triangle

Example One



Example One

- Now we need to package this data so it can be sent to the GPU
- This is sent in buffers, and each buffer must have an integer name
- The glGenBuffers() procedure does this for us, the first parameter is the number of names that we need
- The second parameter is an array that will hold the names
- This is the same as we saw with glGenVertexArrays, this pattern will repeat itself through all our programs

Example One

```
glGenBuffers(1, &vbuffer);
 glBindBuffer(GL_ARRAY_BUFFER, vbuffer);
 glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), NULL, GL_STATIC_DRAW);
 glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(vertices), vertices);

glGenBuffers(1, &ibuffer);
 glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibuffer);
 glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indexes), indexes, GL_STATIC_DRAW);
```

Example One

- Once we have done this we bind the buffer name
- The glBindBuffer() procedure is used for this
- The first parameter is the binding point, and the second parameter is the buffer name
- Now we are ready to load the data into the buffer
- There are two ways of doing this

Example One

- In the case of the indexes we do this with one call to `glBufferData`
- This call allocates space for the data and copies the data into the buffer
- The parameters to this procedure are the binding point, the amount of space to allocate, the data to be copied into the buffer, and finally a flag to indicate where the data should be allocated

Example One

- In the second approach the space is allocated by `glBufferData` and the data is copied in separately by `glBufferSubData`
- We pass `NULL` as the third parameter to `glBufferData` to indicate that it only allocates space
- The `glBufferSubData` parameters are the binding point, the location where the copy starts, the number of bytes of copy and the data to be copied

Example One

- Why would we want to do this??
- A vertex can have more than coordinate data, for example normals, we can only do this using the second approach
- Note: in this case we have stored the vertex and index data in arrays
- We can use `sizeof()` on these arrays to compute the number of bytes required to store the data, and the number of bytes to be copied
- Later we will dynamically create the data, in that case you will need to compute the number of bytes

Example One

- Now we are ready to compile the GPU programs
- The following three lines does this for us:

```
vs = buildShader(GL_VERTEX_SHADER, "example1.vs");
fs = buildShader(GL_FRAGMENT_SHADER, "example1.fs");
program = buildProgram(vs,fs,0);
```
- Compiling shader programs is a complex task, I've provided some procedures to simplify the process
- Later in the course we will examine the process

Example One

- The procedures for compiling the shader programs are in shaders.h and shaders.cpp
- The buildShader procedure compiles a shader, the first parameter is the type of shader and the second parameter is the file where the shader is stored
- The buildProgram procedure takes a zero terminated list of shaders and produces a complete shader program
- A complete shader program needs both a vertex and fragment shader
- Now we need to link the vertex data to the shader program, the code on the next slide shows how this is done

Example One

```
glUseProgram(program);
vPosition = glGetUniformLocation(program,"vPosition");
 glVertexAttribPointer(vPosition, 2, GL_FLOAT, GL_FALSE, 0, 0);
 glEnableVertexAttribArray(vPosition);
```

- The glUseProgram procedure states that we are going to use the program we just produced
- A OpenGL program can have multiple GLSL programs and this allows us to switch between them
- Our vertex program has a variable called vPosition, this is the variable that will get the vertex data

Example One

- The `glGetAttribLocation()` procedure finds the location of this variable in the GLSL program so we can reference it
- This will return a positive integer
- We could compute this location by hand, some examples on the Internet do this, but this is a good source of bugs
- If you change the shader program these locations could change, there is no check that you are setting the right variable
- The `glVertexAttribPointer` procedure then performs the link between the variable and the data

Example One

- The first parameter to this procedure is the variable pointer, in this case vPosition
- The second parameter is the size of the data, in its data units
- In our case we have 2 coordinates
- The third parameter is the type of data, in this case it is floating point data
- The fourth parameter is whether the data needs to be normalized, this is mainly used when we are converting one type of data into another, which we won't be doing

Example One

- The fifth parameter is the stride, the distance in bytes between data values
- Our vertices occur one after the other in the buffer so we use a value of zero
- The final parameter is the location in bytes where the data starts in the buffer
- The glEnableVertexAttribArray procedure enables the use of the vertex data that we have just set up

Example One

- Now that we have set all this up we need to display it
- The function on the next slide is called each time we need to update the screen
- We start by clearing the screen
- Then we tell OpenGL which GLSL program to use
- We then specify the buffer array object that contains our data
- We then specify the buffer that has the vertex indices

Example One

```
void displayFunc() {  
  
    glClear(GL_COLOR_BUFFER_BIT);  
    glUseProgram(program);  
  
    glBindVertexArray(triangleVAO);  
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,ibuffer);  
    glDrawElements(GL_TRIANGLES, 3, GL_UNSIGNED_SHORT, NULL);  
  
}
```

Example One

- Finally we use `glDrawElements` to draw the triangle
- The data we want to draw is triangles, so that is the first parameter (we can also draw points and lines)
- The second parameter is the number of vertices and the third parameter is the type of vertex data
- The fourth parameter is an offset within the indices array where the drawing starts

Example One

- The last thing we need for this part of the example is the main procedure
- The first set of statements initializes glfw and creates the window that we will use for our example
- The second set of statements initializes glew and reports any errors
- glfw provides the interface between OpenGL and the operating system
- glew provides the include files for OpenGL, plus any extensions

Example One

```
GLFWwindow *window;  
  
// start by setting error callback in case something goes wrong  
glfwSetErrorCallback(error_callback);  
  
// initialize glfw  
  
if (!glfwInit()) {  
    fprintf(stderr, "can't initialize GLFW\n");  
}  
  
// create the window used by our application  
  
window = glfwCreateWindow(512, 512, "Example One", NULL,  
NULL);
```

```
if (!window) {  
    glfwTerminate();  
    exit(EXIT_FAILURE);  
}  
  
glfwMakeContextCurrent(window);  
  
GLenum error = glewInit();  
  
if(error != GLEW_OK){  
    printf("Error starting GLEW: %s\n",glewGetString(error));  
    exit(0);  
}
```

Example One

- The second half of the main procedure is shown on the following slide
- first we call our init() procedure
- We then set the clear colour to white, to give us a white background
- Finally we have a loop that displays the triangle while waiting for input

Example One

```
init();
glClearColor(1.0,1.0,1.0,1.0);
glfwSwapInterval(1);
// GLFW main loop, display model, swapbuffer and check for input
while (!glfwWindowShouldClose(window)) {
    display();
    glfwSwapBuffers(window);
    glfwPollEvents();
}

glfwTerminate();
```

Example One

- All we have left is our vertex and fragment programs that are stored on separate files
- The vertex shader program is shown on the next slide
- It starts with a `#version`, which tells OpenGL the version of GLSL that we are using, 330 stands for version 3.3
- The first thing we do is declare `vPosition`, this is an input to our program so the declaration starts with `in`
- The declarations for variables that are outputs start with `out`

Example One

```
#version 330 core
/*
 * Simple vertex shader for example one
 */
in vec4 vPosition;
void main() {
    gl_Position = vPosition;
}
```

Example One

- This is followed by the type, which is a vector of 4 floating point values
- How does this work, we are only passing in two floats for each vertex?
- In the GPU vertex coordinates are 4 vectors, so our vertex coordinates are padded to 4 values
- The main procedure is executed once for each vertex in our program, all it does is copy the vertex coordinates to the next stage

Example One

- The fragment program is shown on the next slide
- This is even simpler, it just sets the fragment colour to red
- The main procedure is called once for each fragment, think pixel for now
- A colour value has four components: red, green, blue and alpha

Example One

```
#version 330 core
/*
 * Simple fragment shader for example one
 */
void main() {
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

Example One

- Shader programs should be stored in pure text files, i.e. straight ASCII characters
- On Windows use something like notepad to edit them
- For program development we usually store the shader programs in external files, this make it easier to debug them during program development
- For distribution they can be placed in the C++ code as character string constants

Example One

- This is fairly long, but we have covered most of the process
- This gives you a template for building your own applications
- The other examples are incremental and build on what we have done so far

Example One

- To build the example code, download Example1.zip from Canvas and unzip it into your Examples folder
- Next build the program
- Before you run the program copy the fragment and vertex shaders into the debug folder
- Also copy the dll's into this folder

Example Two

- Example two is a bit more exciting, we have a rotating triangle
- This example illustrates the use of transformations and the glm library
- It also shows how simple animations can be performed in OpenGL programs
- This example starts with the code for example one and adds to it

Example Two

- To make our triangle rotate we need to use a rotation transformation
- All transformations in OpenGL are represented by matrices, either 4x4 matrices or 3x3 matrices
- To combine transformations we multiply their matrices
- To transform a point we multiply the point by the transformation matrix

Example Two

- For this example we will rotate the triangle about the z axis, the axis coming out of the screen
- Since we want to animate the rotation we will have a variable, angle, that changes each time we draw the triangle
- Now we need to know how to construct the matrix that will perform the rotation

Example Two

- glm has a function called rotate that will construct the transformation matrix
- The first parameter to this function is the previous transformation matrix, in our case the identity matrix
- The second parameter is the rotation angle, in radians
- The third parameter is the rotation axis, the axis we are rotating about

Example Two

- The next slide shows our new display() procedure
- It starts with two declarations, the first one is for the transformation matrix
- Our vertex shader program will use this matrix, the location of this matrix in the shader program is modelLoc
- Note that we use the `glm::` scope for all of the glm data types and procedures

Example Two

```
void display () {  
    glm::mat4 model;  
    int modelLoc;  
  
    model = glm::rotate(glm::mat4(1.0), angle, glm::vec3(0.0, 0.0, 1.0));  
  
    glClear(GL_COLOR_BUFFER_BIT);  
    glUseProgram(program);  
    modelLoc = glGetUniformLocation(program,"model");  
    glUniformMatrix4fv(modelLoc, 1, 0, glm::value_ptr(model));  
  
    glBindVertexArray(triangleVAO);  
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibuffer);  
    glDrawElements(GL_TRIANGLES, 3, GL_UNSIGNED_SHORT, NULL);
```

Note that model is a C++ class, we can't directly pass model as a parameter. Instead we need to get a pointer to its data.

Example Two

- Next we have the statement that constructs the matrix
- The function `glm::mat4(1.0)` creates a 4x4 identity matrix
- The next new thing is the call to `glGetUniformLocation`, to find the shader variable where we will store the matrix
- This is a variable that is stored on the GPU, it is part of the vertex shader program
- This is part of the communications between the CPU and the GPU

Example Two

- There are two common types of shader variables
- We have already seen attribute variables, they get a new value for each vertex or fragment
- Our transformation matrix will be a uniform variable, a variable that has the same value for many vertices or fragments
- We find the location of these variables in essentially the same way as attribute variable, except we use a different procedure

Example Two

- The `glUniformMatrix4fv` procedure is used to transfer the matrix to the GPU
- The “4fv” in the name specifies a 4x4 matrix of float value passed by a pointer
- The first parameter is the shader variable to receive the matrix
- The second parameter is the number of matrices, we could have an array of matrices in the shader program

Example Two

- The third parameter is a flag to indicate whether the matrix should be transposed, we don't want this
- The final parameter is a pointer to the matrix
- Now lets examine the vertex shader program, shown on the next slide
- Note that we have a uniform variable called model which matches the transformation matrix in our OpenGL program

Example Two

```
#version 330 core
/*
 * Simple vertex shader for example two
 */

in vec4 vPosition;
uniform mat4 model;

void main() {
    gl_Position = model * vPosition;
}
```

Example Two

- The vertex program multiplies the current vertex by this matrix and passes it down the pipeline
- How is the angle variable updated so we have a different value each time we draw the triangle?
- How do we end up drawing the triangle more than once?

Example Two

- We can update the angle in the loop we have at the end of the main() procedure
- We have a global variable angle that is initialized to zero
- This variable is referenced in the display() procedure
- The following slides shows how it is updated in the display loop

Example Two

```
while (!glfwWindowShouldClose(window)) {  
    display();  
    glfwSwapBuffers(window);  
    glfwPollEvents();  
    angle = angle + 0.1;  
}
```

Example Two

- We add 0.1 radians to angle each time we go through the loop
- Increasing this value makes the triangle spin faster
- Decreasing this value makes the triangle spin slower
- The glm library has a number of useful functions
- It has functions to construct scale and translation matrices
- It also has functions for viewing transformation, which we will use in the fourth example

Example Three

- Our triangle still looks kind of boring, we can improve this by adding some simulated lighting
- We haven't discussed light models yet, it's part of rendering, so we will just use a very simple one
- To do lighting we need to have a normal vector at each vertex, in our case the normal is just $(0,0,1)$

Example Three

- To add normal vectors we need to make some changes to our init() procedure
- First we construct an array of normal vectors, one for each vertex, the code for doing this is:

```
GLfloat normals[3][3] = {  
    {0.0, 0.0, 1.0},  
    {0.0, 0.0, 1.0},  
    {0.0, 0.0, 1.0}  
};
```

Example Three

- Next we need to add the normals to the same buffer that contains the vertices
- The code for doing this is shown on the next slide
- The call to `glBufferData` now allocates enough memory for both the vertex coordinates and the normal vectors
- We add a second call to `glBufferSubData` to copy the normal vectors into the buffer
- The normal vectors will appear after the vertex coordinates in the buffer

Example Three

```
glGenBuffers(1, &vbuffer);
 glBindBuffer(GL_ARRAY_BUFFER, vbuffer);
 glBufferData(GL_ARRAY_BUFFER, sizeof(vertices)+sizeof(normals), NULL, GL_STATIC_DRAW);
 glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(vertices), vertices);
 glBufferSubData(GL_ARRAY_BUFFER, sizeof(vertices), sizeof(normals), normals);
```

Example Three

- In the vertex shader the vNormal variable will receive the normal vectors, so we need to specify this in the OpenGL program:

```
vNormal = glGetUniformLocation(program, "vNormal");
glVertexAttribPointer(vNormal, 3, GL_FLOAT, GL_FALSE, 0, (void*)
sizeof(vertices));
 glEnableVertexAttribArray(vNormal);
```

- The last parameter to glVertexAttribPointer is the location in the buffer where the normal vectors are stored

Example Three

- The only other change to the OpenGL programs is to rotate about the y axis instead of the z axis
- The vertex program for this example is shown on the next slide
- There are several new things here
- First we have an out variable, this is the transformed normal vector that will be passed to the fragment program

Example Three

```
#version 330 core
/*
 * Simple vertex shader for example three
 */

in vec4 vPosition;
in vec3 vNormal;
uniform mat4 model;
out vec3 normal;

void main() {

    gl_Position = model * vPosition;
    normal = (model * vec4(vNormal,1.0)).xyz;

}
```

Example Three

- Next we have the line that transforms the normal vector
- Our transformation matrix is 4x4 but the normal vector has only three components
- We need to add the fourth component, multiply by the matrix and then extract just the first three components

Example Three

- In general, the way we are transforming the normal vector is incorrect
- We should use the inverse transpose of the transformation matrix
- But since our matrix is a rotation matrix, its inverse transpose is the original matrix
- We will see the correct way of doing this in the next example

Example Three

- Now we turn to the fragment program, shown on the next slide
- This program has an input variable, the normal vector computed by the vertex program
- We normalize this to get the variable N , L is the direction to the light source
- We take the dot product of these two vectors to get the diffuse light component

Example Three

```
#version 330 core
in vec3 normal;
void main() {
    vec3 N;
    vec3 L = vec3(0.0, 0.0, 1.0);
    vec4 colour = vec4(1.0, 0.0, 0.0, 1.0);
    float diffuse;

    N = normalize(normal);
    diffuse = dot(N,L);
    if(diffuse < 0.0) {
        diffuse = 0.0;
    }
}
```

```
    gl_FragColor = min(0.3*colour + 0.7*diffuse*colour,
vec4(1.0));
    gl_FragColor.a = colour.a;
}
```

Example Three

- If the light source is behind the triangle the dot produce will be negative, in this case we set the diffuse light to zero
- The variable colour contains the colour of the triangle
- The final colour is 30% ambient (a constant term) and 70% diffuse
- Watch the triangle as it rotates and see how the colour changes

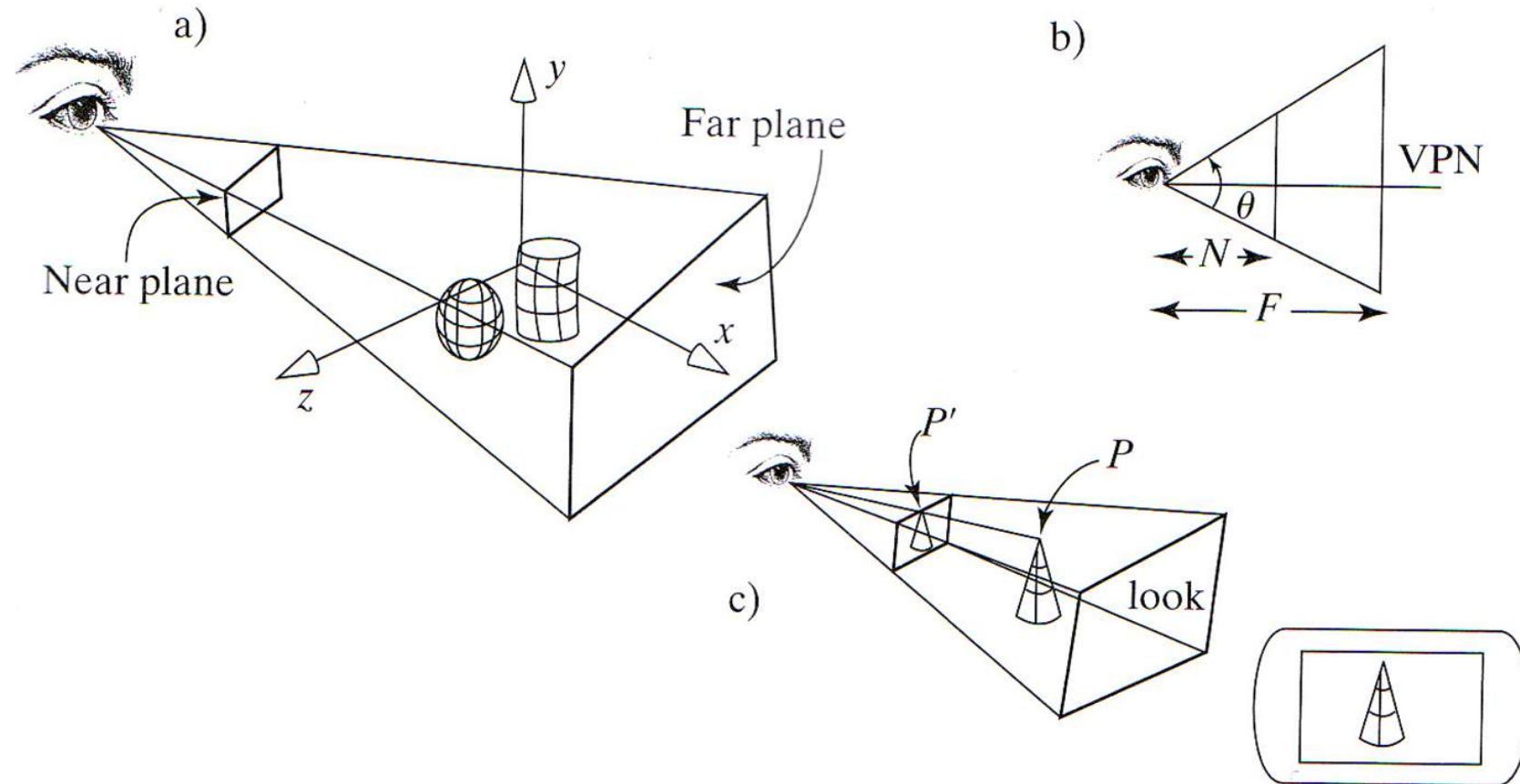
Example Four

- Our final example introduces 3D viewing
- So far we've used a fixed view point where we were looking down the z axis, this was essentially a 2D view
- Now we will introduce a perspective projection and the ability to move the eye position

Example Four

- Recall the viewing frustum from the previous lecture, see the next slide
- There are two parts to viewing: one is the shape of the frustum and the other is the position of the frustum
- The perspective projection determines the shape of the viewing frustum, and this is usually constant

Example Four



Example Four

- The position of the frustum can change each time we draw the window
- This is called the viewing transformation
- So it makes sense to construct two matrices, one for the perspective transformation and the other for the viewing transformation
- Our program will follow this approach

Example Four

- For the perspective transformation the framebufferSizeCallback() function is called each time the size of the window changes
- This function constructs the perspective matrix based on the aspect ratio of the window
- This function is registered in the main function using the following statement:
`glfwSetFramebufferSizeCallback(window, framebufferSizeCallback);`

Example Four

```
void framebufferSizeCallback(GLFWwindow *window, int w, int h) {  
  
    // Prevent a divide by zero, when window is too short  
    // (you cant make a window of zero width).  
  
    if (h == 0)  
        h = 1;  
  
    float ratio = 1.0f * w / h;  
  
    glfwMakeContextCurrent(window);  
    glViewport(0, 0, w, h);  
    projection = glm::perspective(0.7f, ratio, 1.0f, 100.0f);  
  
}
```

Example Four

- The `glm::perspective` function is used to construct the perspective matrix
- The first parameter is the field of view of the frustum
- The second parameter is the aspect ratio of the window
- The third and fourth parameters are the near and far clipping planes
- This is a pretty standard set up that we will use in most of our programs

Example Four

- The viewing transformation matrix is constructed in the `display()` function
- The first part of this function is shown on the next slide
- The `glm::lookat` function constructs the viewing matrix
- The first parameter to this function is the eye position, this is computed in a separate function

Example Four

```
void display () {  
    glm::mat4 model;  
    glm::mat4 view;  
    glm::mat4 modelViewPerspective;  
    int modelLoc;  
    int normalLoc;  
  
    model = glm::rotate(glm::mat4(1.0), angle, glm::vec3(0.0, 1.0, 0.0));  
  
    view = glm::lookAt(glm::vec3(eyex, eyey, eyez),  
                      glm::vec3(0.0f, 0.0f, 0.0f),  
                      glm::vec3(0.0f, 1.0f, 0.0f));  
  
    glm::mat3 normal = glm::transpose(glm::inverse(glm::mat3(view*model)));  
  
    modelViewPerspective = projection * view * model;
```

Example Four

- The second parameter is the position the eye is looking at
- The third parameter is the up vector, this is the direction that is up in the view
- Note: the eye positon and look at position define a line, the direction the viewer is looking
- The frustum can rotate about this line, the up vector fixes this rotation

Example Four

- The vertices will be transformed by the combination of the perspective, viewing and model transformations
- This is the `modelViewPerspective` matrix
- The normal vectors are only transformed by the view and model transformation
- The matrices are combined, the inverse and transpose are taken, this is assigned to the `normal` variable

Example Four

- The following statement are then used to send the matrices to the GPU:

```
modelLoc = glGetUniformLocation(program,"model");
glUniformMatrix4fv(modelLoc, 1, 0, glm::value_ptr(modelViewPerspective));
normalLoc = glGetUniformLocation(program,"normalMat");
glUniformMatrix3fv(normalLoc, 1, 0, glm::value_ptr(normal));
```

Example Four

- This example uses keystrokes to control the eye position
- The `key_callback()` procedure shown on the next slide does this
- This function is called each time a key is pressed on the keyboard
- A sequence of if statements is used to determine the key that was pressed and perform the appropriate action

Example Four

```
static void key_callback(GLFWwindow* window, int key, int  
scancode, int action, int mods)  
{  
if (key == GLFW_KEY_ESCAPE && action == GLFW_PRESS)  
    glfwSetWindowShouldClose(window, GLFW_TRUE);  
  
if (key == GLFW_KEY_A && action == GLFW_PRESS)  
    phi -= 0.1;  
if (key == GLFW_KEY_D && action == GLFW_PRESS)  
    phi += 0.1;  
if (key == GLFW_KEY_W && action == GLFW_PRESS)  
    theta += 0.1;  
if (key == GLFW_KEY_S && action == GLFW_PRESS)  
    theta -= 0.1;
```

```
eyex = (float)(r*sin(theta)*cos(phi));  
eyey = (float)(r*sin(theta)*sin(phi));  
eyez = (float)(r*cos(theta));  
}
```

Example Four

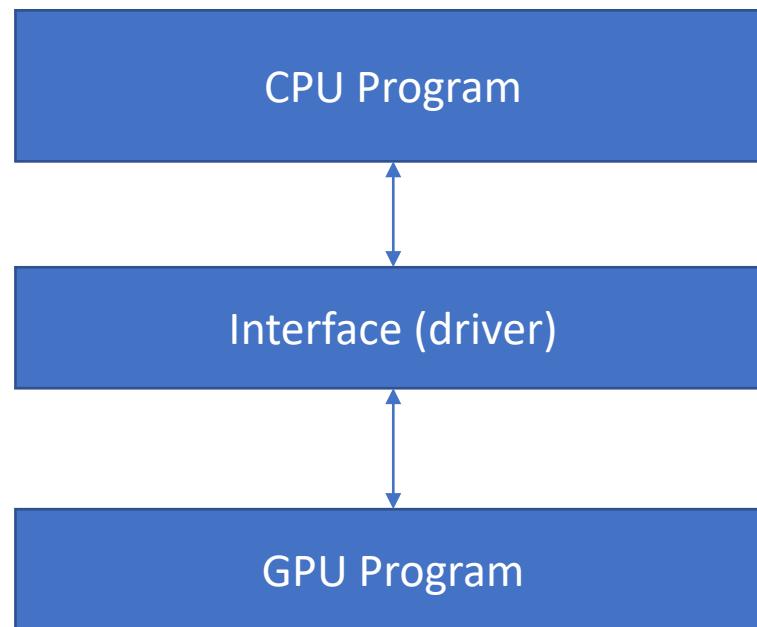
- The variables eyex, eyey and eyez are initialized in the main function
- The vertex shader is shown on the next slide
- This is similar to the previous example, except in this case we have a separate matrix for the normal vectors
- There is no change in the fragment shader

Example Four

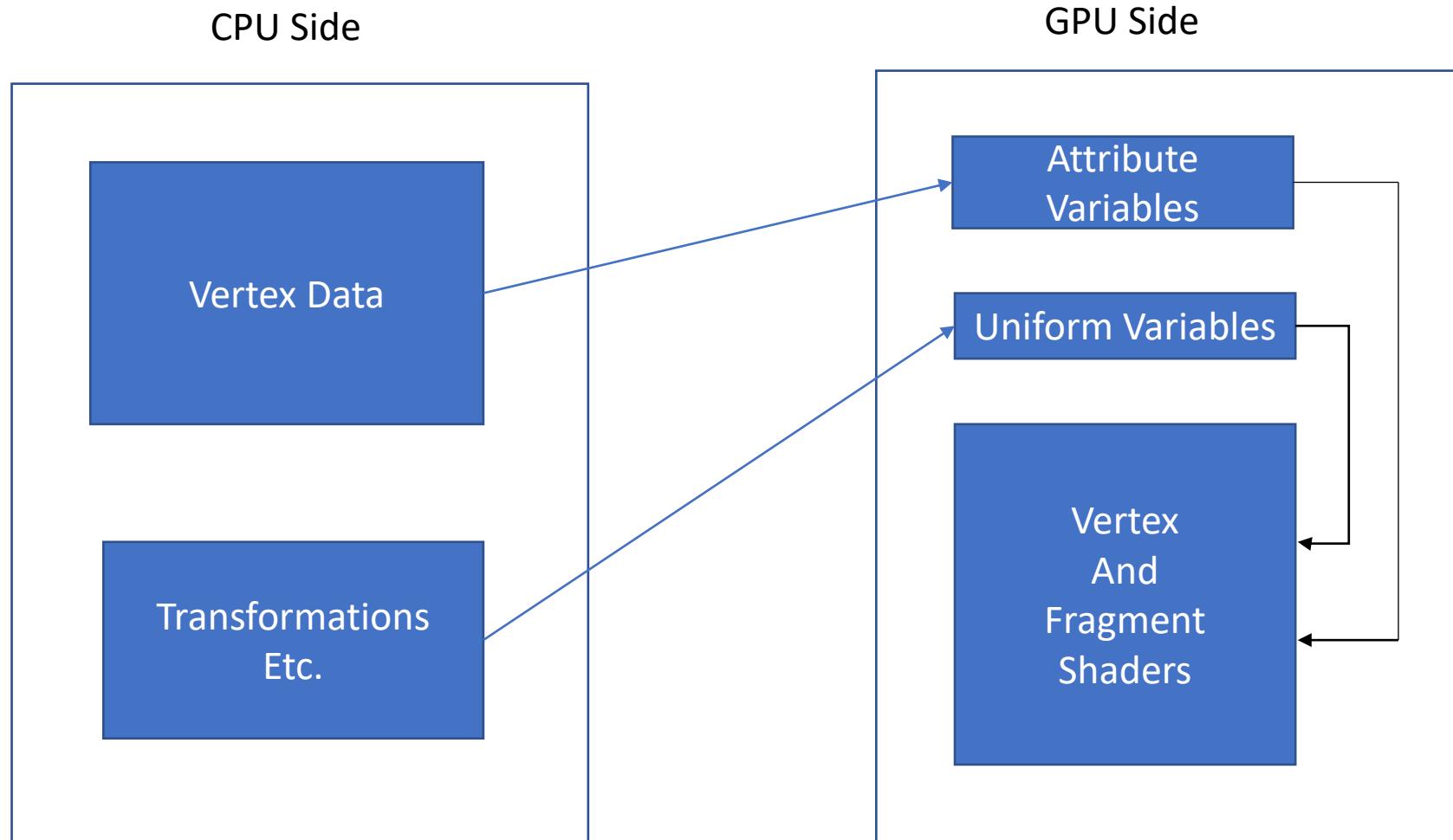
```
in vec4 vPosition;  
in vec3 vNormal;  
uniform mat4 model;  
uniform mat3 normalMat;  
out vec3 normal;  
  
void main() {  
    gl_Position = model * vPosition;  
    normal = normalMat* vNormal;  
}
```

Recap

- Really dealing with two programs, one on CPU and one on GPU



Recap



Summary

- Examined the basics of OpenGL application development
- You can now produce simple OpenGL applications with limited visual effects
- More example programs will be examined as we cover more graphics theory

CSCI 3090

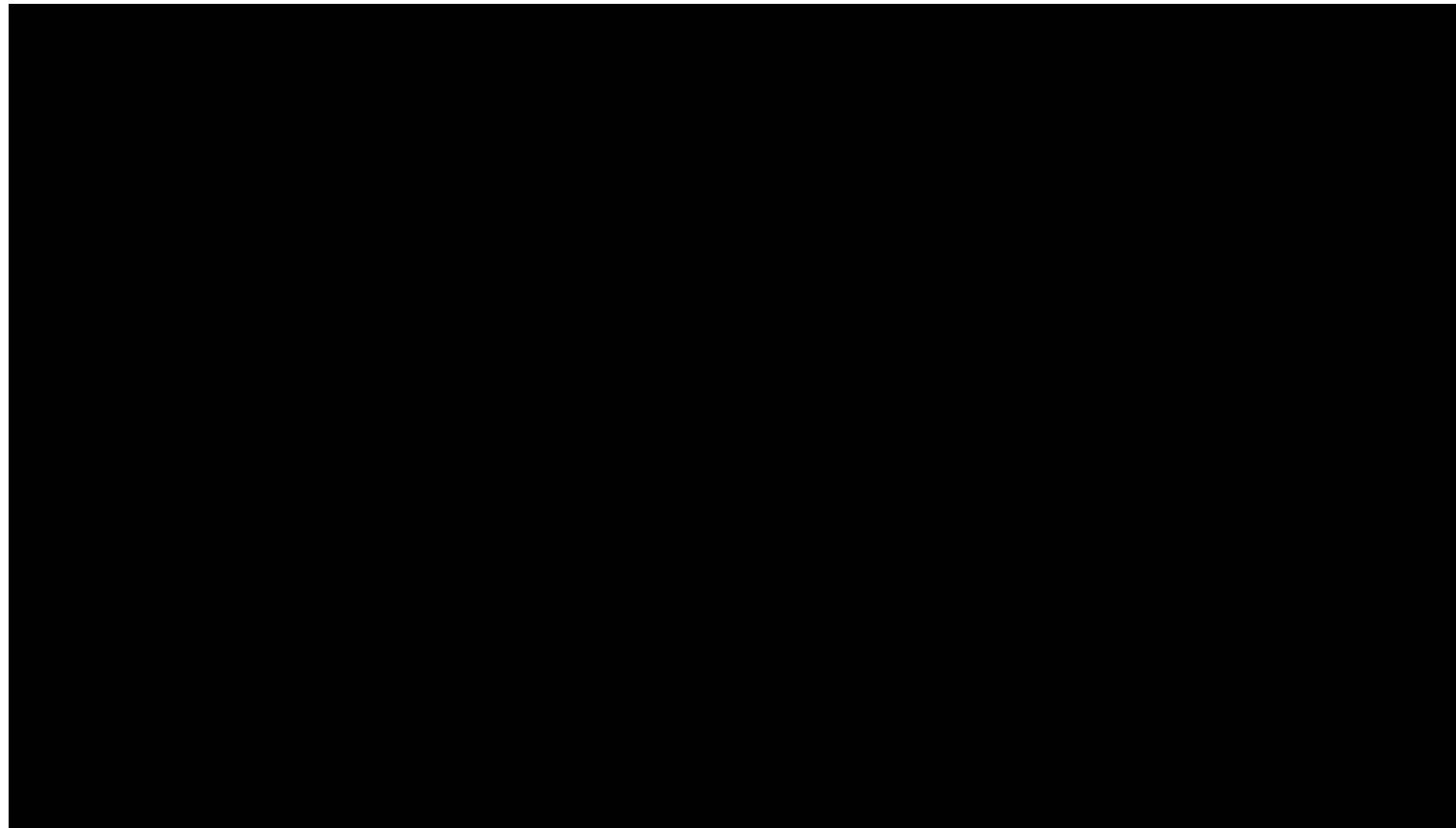
Introduction to Modeling

Mark Green

Faculty of Science

Ontario Tech

Tin Toy (1988)



Goals

- By the end of today's class, you will:
 - Be able to represent points and vectors in matrix form
 - Understand the basic properties of polygons and polygonal meshes
 - Compute normal vectors for vertices in a polygonal mesh

Introduction

- Modeling is about representing graphical information in a computer
- Mostly interested in **shape** information, or geometrical information
- This is the common usage of the term, but could also deal with **material or motion**
- There are many ways of representing geometrical information in a computer, we will look at the basics

The Basics

- Before we start examining data structure, we need to review basic geometry
- From linear algebra we know the difference between scalars, vectors and matrices
- In graphics vectors usually have 3 or 4 components, but can be used to represent two different things
- The first is a position in 3D space, and the second is a direction in 3D space
- The first we can view a point and the second as an arrow

The Basics

- While we can represent both by a 3 component vector, they behave differently
- A position can be translated, it makes sense to move a chair from one side of the room to another
- A direction can't be translated, west is always west, no matter where it is, the arrow always points in the same direction
- This is important when we start doing transformations
- There is also a difference when we come to coordinate systems

The Basics

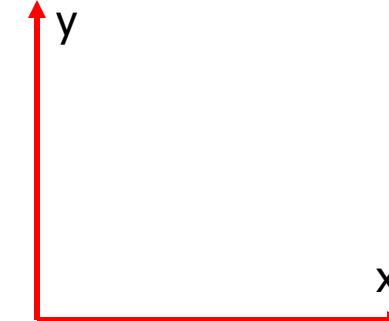
- For both points and directions we need to have 3 orthogonal axis
- The length along these axis become the three components of the vector
- We will often call these axis x, y, and z
- Note: there is no preferred direction for these axis, they just need to be orthogonal
- For points we also need a reference point, this is the origin of the coordinate system

The Basics

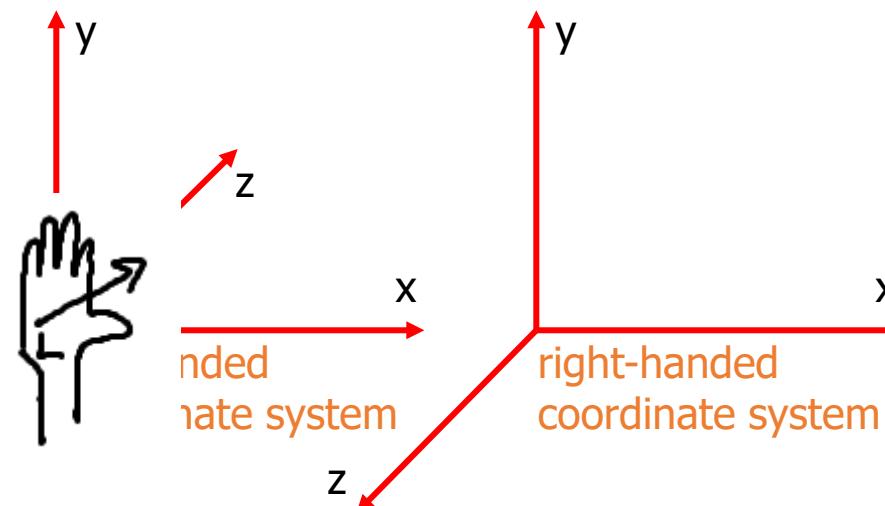
- Now is a good time to review operations on vectors, we will make extensive use of the dot and cross products
- We will also use 3x3 and 4x4 matrices
- They are mainly used to represent transformations, but we will also use them in the representation of curves and surfaces
- Again review matrix multiplication, multiplying a vector by a matrix

Coordinate Systems in CG: 2D & 3D

- two-dimensional

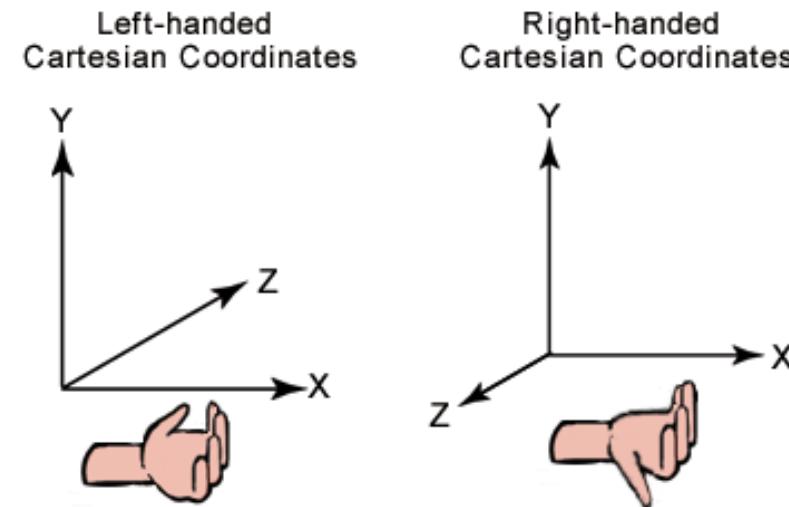


- three-dimensional
 - 2 mirrored systems
 - cannot be matched by rotation
 - OpenGL uses right-handed



LHS / RHS

- To determine direction of z when given LHS or RHS: point the fingers of the selected hand in the positive x-direction; curl fingers towards positive y. Direction of thumb is positive z!

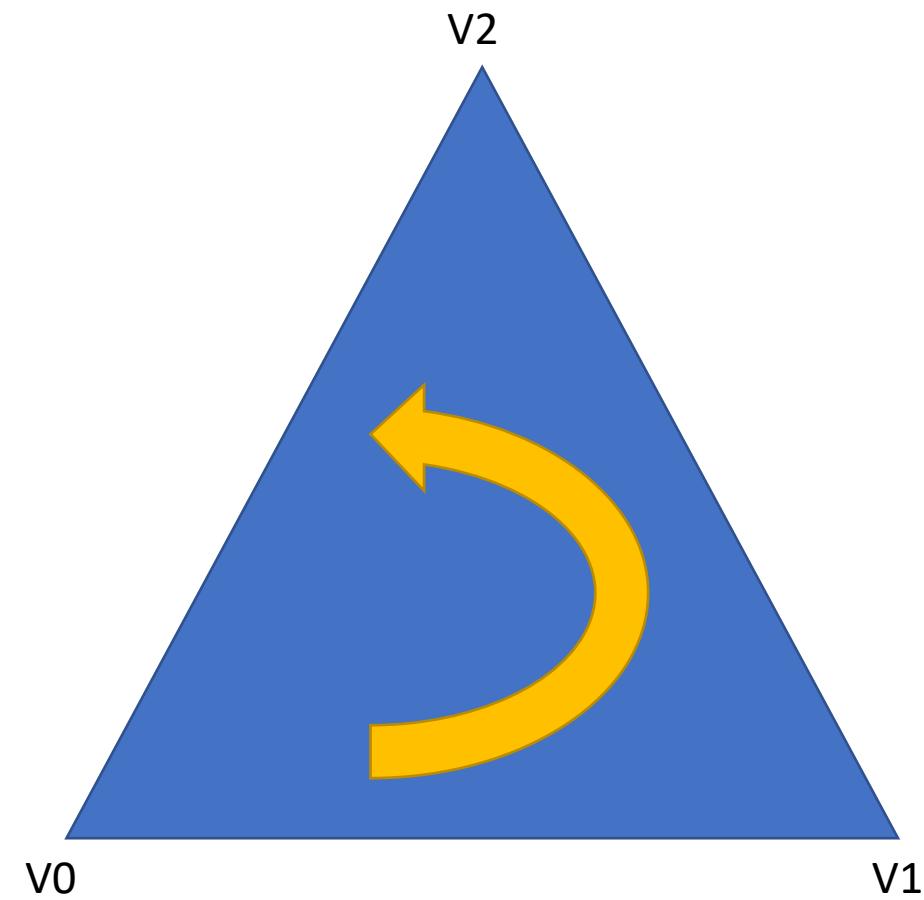


Polygons

- In many ways the most basic representation, this is what the hardware deals with, in fact it only deals with triangles
- We keep things simple to increase performance
- Other representations are usually converted to polygons for display
- Problem is that we often need a lot of them for an accurate representation

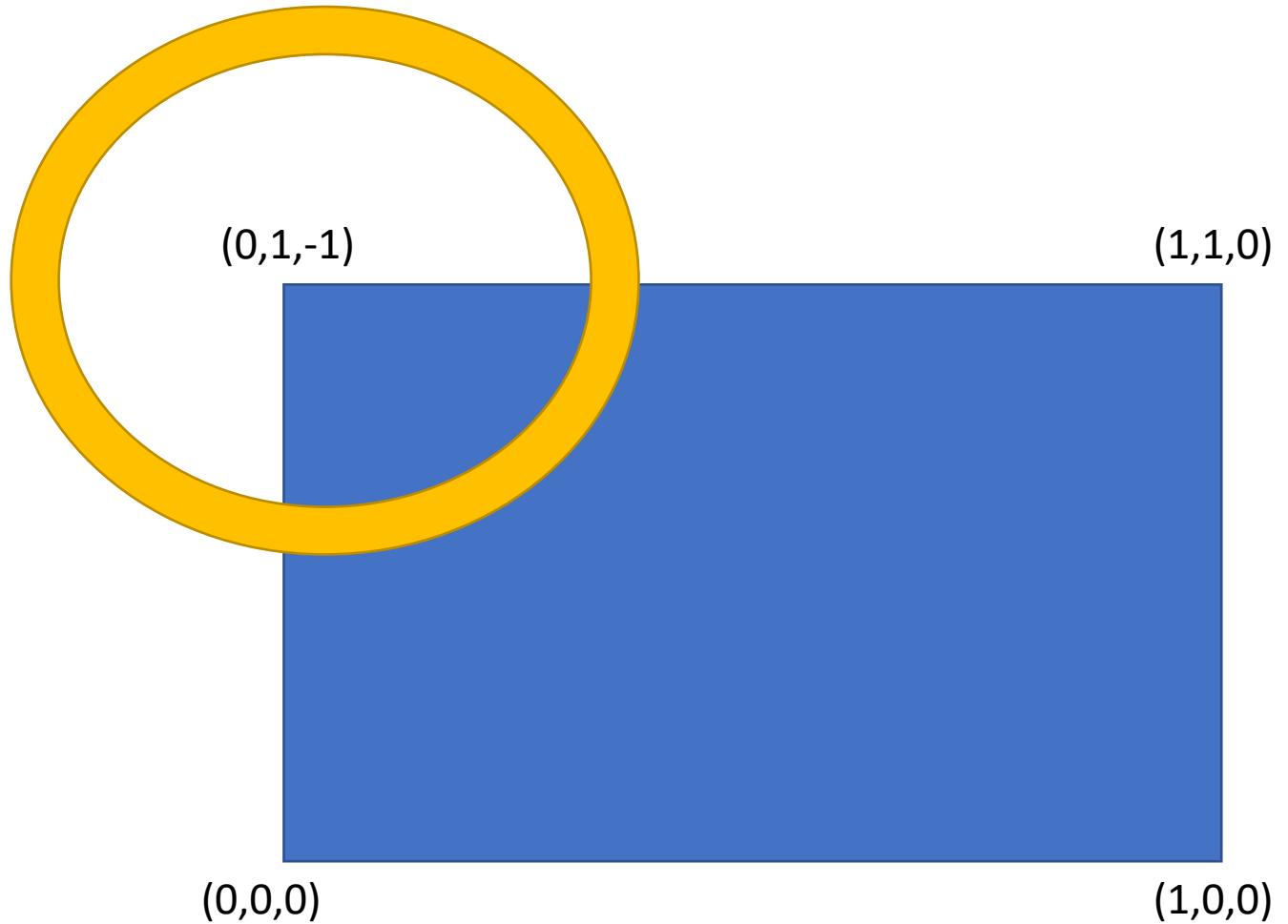
Polygons

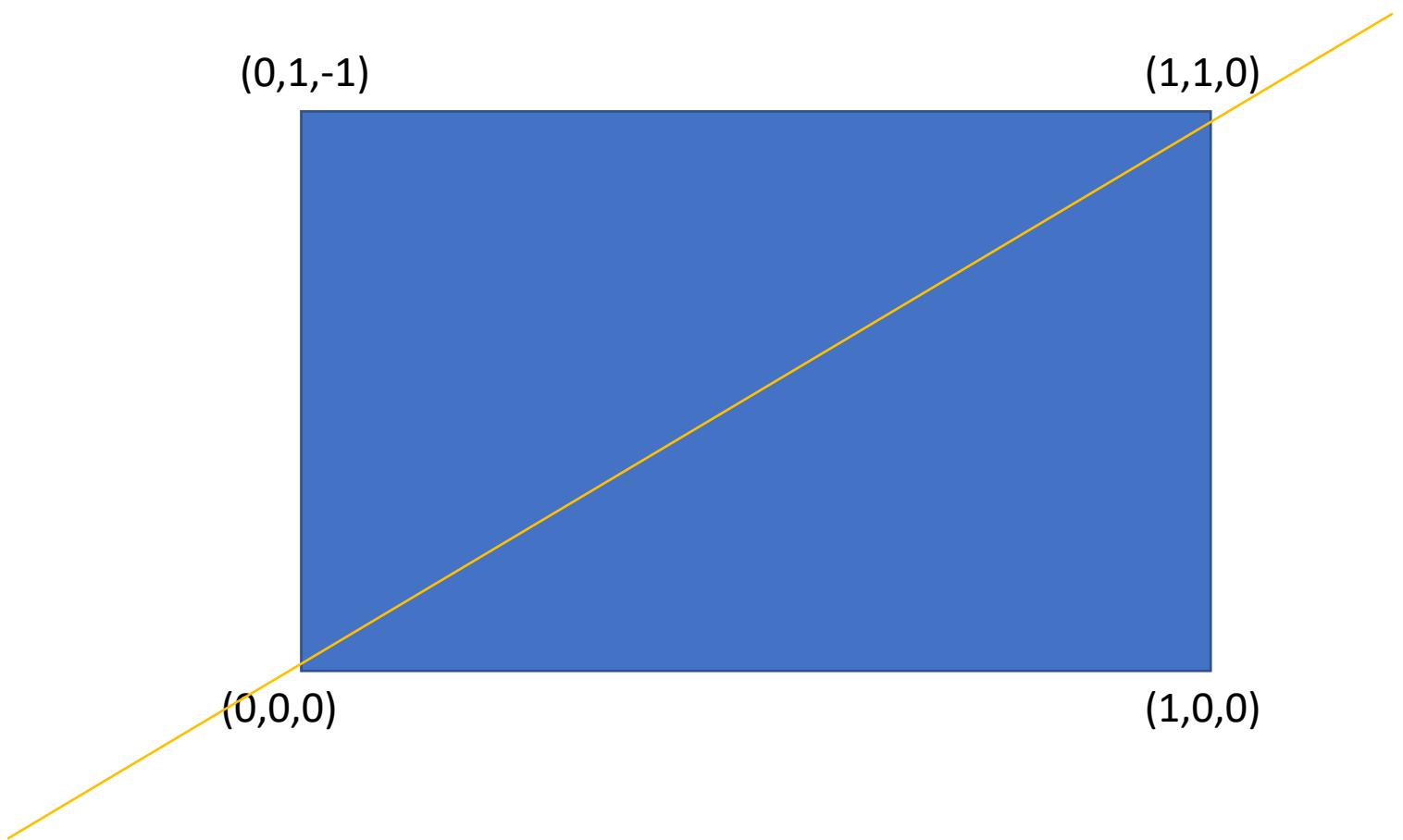
- Polygons are flat, a segment of a plane outlined by straight lines
- Polygons are usually defined by a list of vertices, **the order of the vertices is important, determines inside and outside**
- Usually list the vertices in **counter clockwise** direction (**the *winding order***)
- Must use a consistent ordering for all of the polygons in the model



Polygons

- List of vertices doesn't guarantee that the polygon is planar (why?)
- If it isn't planar many algorithms will fail
- Triangles are always planar, but it's a good idea to test other types of polygons
- This is also a reason for favoring triangles





Polygons

- If it isn't planar many algorithms will fail
- Triangles are always planar, but it's a good idea to test other types of polygons
- This is also a reason for favoring triangles
- We could build our model from independent polygons, which is okay for small models, but this leads to problems – polygon soup

Plane Equation

- A point $p = (x, y, z)$ is on a plane if it satisfies the plane equation:

$$ax+by+cz+d = 0$$

- Where (a, b, c) is the normal to the plane
- If the value of the plane equation is non-zero, this value gives us the distance from the plane to the point
- The sign of this value tells which side of the plane the point lies on

Meshes

- If we represent a surface by a collection of polygons we want the surface to be continuous
- There should be no gaps between the polygons, they should meet at their edges and vertices
- One way of doing this is to use a mesh, where the shared vertices are stored only once
- This also reduces the amount of memory required to store the model

Meshes

- By storing each vertex once, no matter how many polygons it appears in, we guarantee that there are no gaps
- Where could the gaps come from? Shouldn't all the common vertices have the same value?
- We are dealing with floating point, so there are several place things could go wrong:
 - We could have rounding errors in computations
 - Some of the file formats are text based, when the models are read and written we loose precision

Meshes

- We want a simple, memory efficient way of storing meshes
- We also want a data structure that is compatible with OpenGL, make programming easier
- The model is usually divided into two tables or arrays
- The first table, called the vertex table, contains the information on all of the vertices, including their coordinates, normal vectors, and other information
- This is similar to the vertex arrays in OpenGL

Vertex Table

Index	Location	Normal	Color
0	X, Y, Z	Nx, Ny, Nz	R, G, B
1	X, Y, Z	Nx, Ny, Nz	R, G, B
...

Meshes

- Each vertex is identified by an index into this table
- The face table has one entry for each polygon
- This entry contains the list of vertices, indices into the vertex table, in the polygon
- It could also contain the plane equation or normal for the polygon
- Note that this is similar to the element array buffer in OpenGL
- There is a clear mapping between this data structure and OpenGL

Face Table

Face Index	Vertices
0	0, 2, 3
1	1, 3, 4
...	...

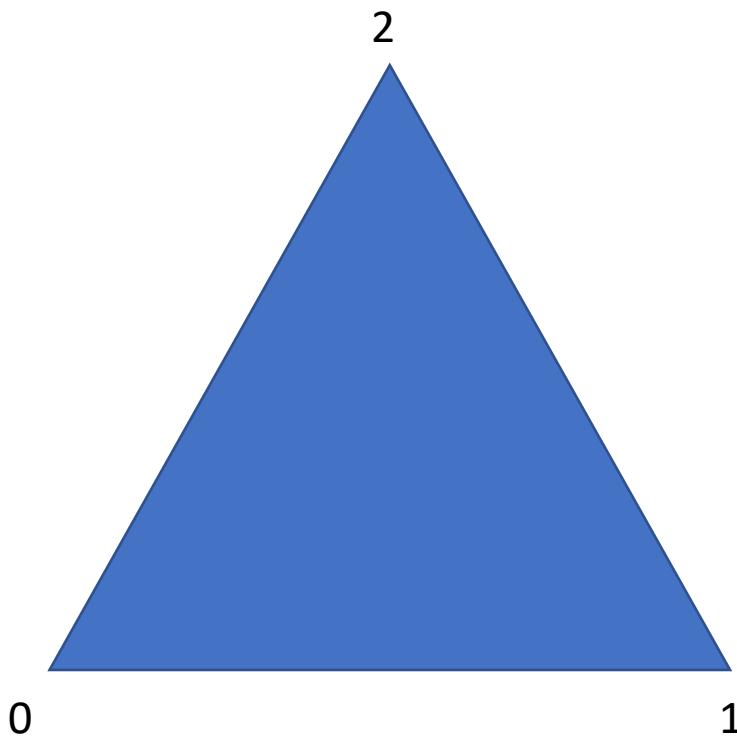
OpenGL

- This is essentially what we have been doing in our OpenGL programs
- We had one buffer that contained the vertex information: coordinates and normal vectors
- A second buffer contains the indices into this table
- A close match between the modeling technique and its implementation in OpenGL

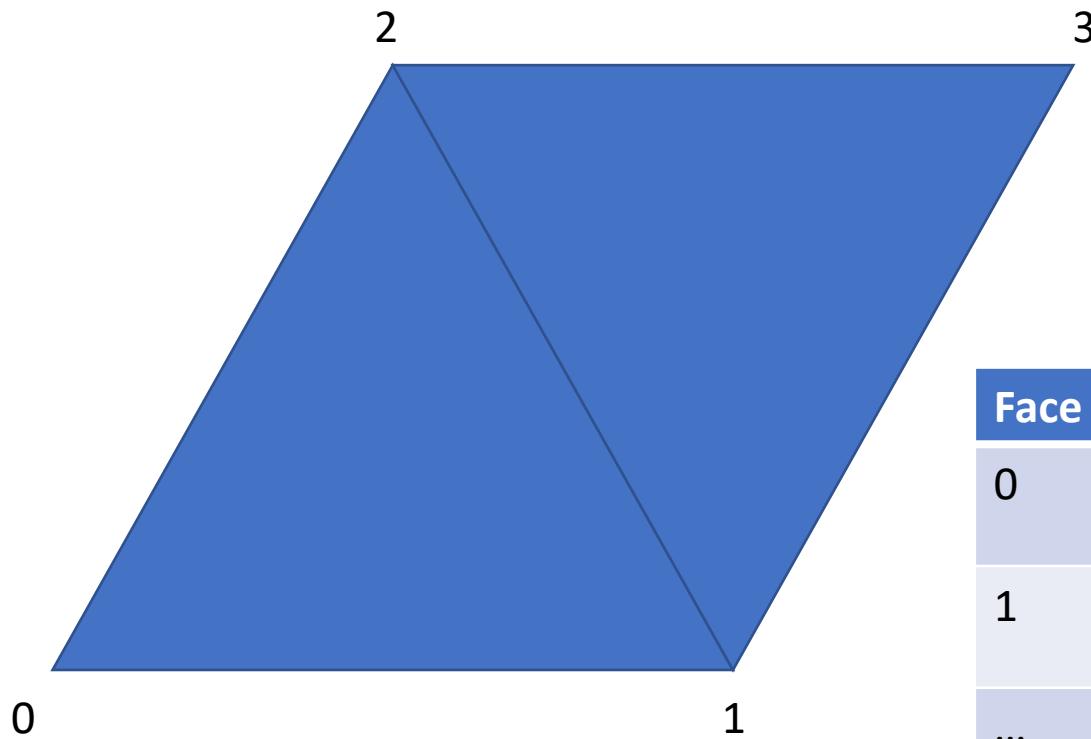
Variations on the Theme

- In OpenGL we have two other ways of representing triangle data
- They are based on two different ways of looking at meshes
- The first is a triangle strip
- This occurs when we can view the triangles as forming a line
- In this case the last two vertices of one triangle can be the first two vertices of the next
- After the first triangle, each subsequent triangle only needs one vertex

Variations on the Theme



Variations on the Theme

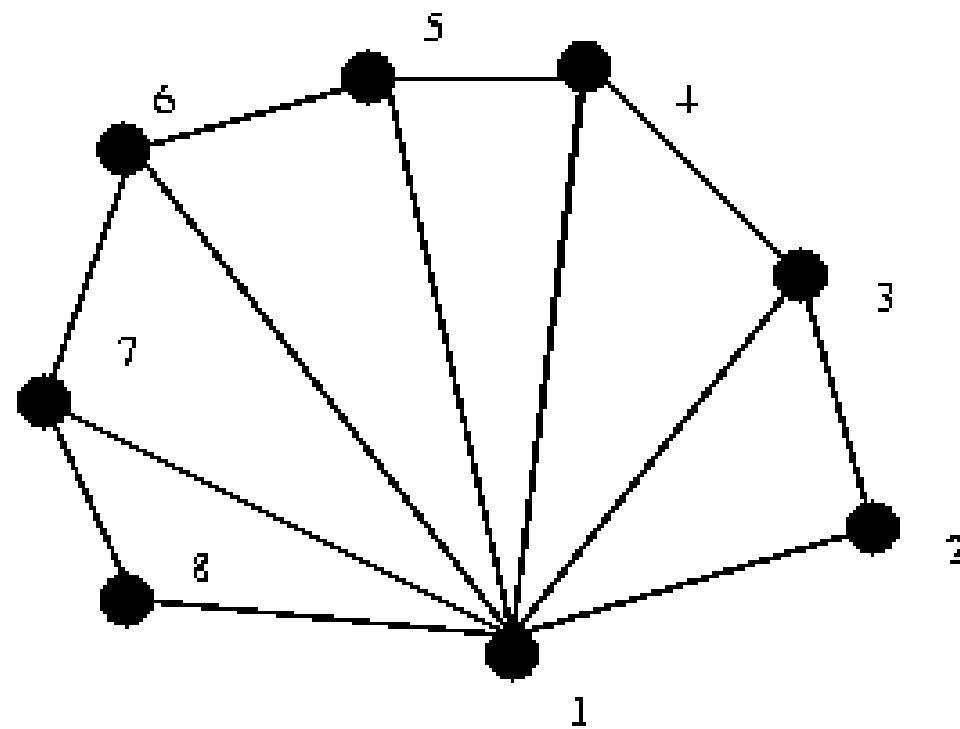


Face Index	Vertices
0	0,1,2
1	2,1,3
...	...

Variations on the Theme

- A triangle fan is a similar idea
- In this case all the triangles have a common vertex, this is the first vertex of the fan
- Subsequent vertices are in counter clockwise order around the first vertex
- Consider the top and bottom of a cylinder, this can easily be done with a fan
- The first vertex is the center of the top, the rest are around the outside of the cylinder

Variations on the Theme



OpenGL

- In `glDrawElements` we used `GL_TRIANGLES` as the first parameter
- We can use `GL_TRIANGLE_STRIP` to draw a triangle strip
- We can use `GL_TRIANGLE_FAN` to draw a triangle fan
- These are special cases that occur in some models that can be useful
- We will see an example of a triangle fan later in the course

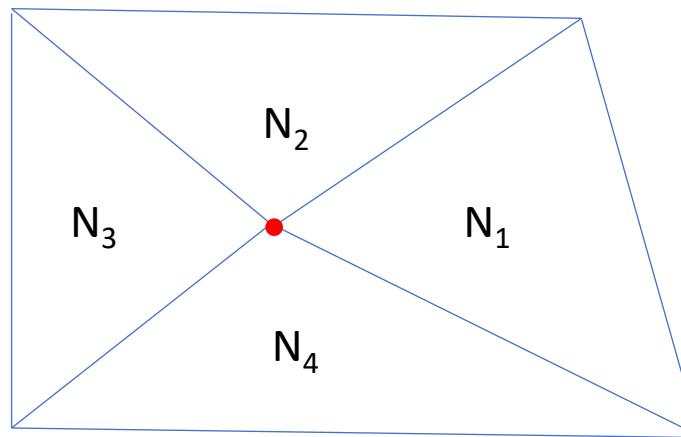
Meshes

- Vertex and face tables are a standard way of representing polygons, some graphics packages even have optimizations for displaying them
- We need normal vectors when displaying polygons, but sometimes they aren't supplied, but they are quite easy to compute
- Well sort of, there are a few things that you need to be careful of when you are doing it
- The first time is a bit challenging, but after that it isn't hard

Mesh Normals

- In order to have a smooth looking mesh each vertex must have its own normal vector
- We want these vectors to vary smoothly over the surface of the mesh
- In most cases we can't directly compute the normal vector at a vertex
- We need to use a two step approach:
 - Start by computing normal vectors for the triangles or polygons
 - Then for each vertex, average the normal vectors for the polygons it belongs to

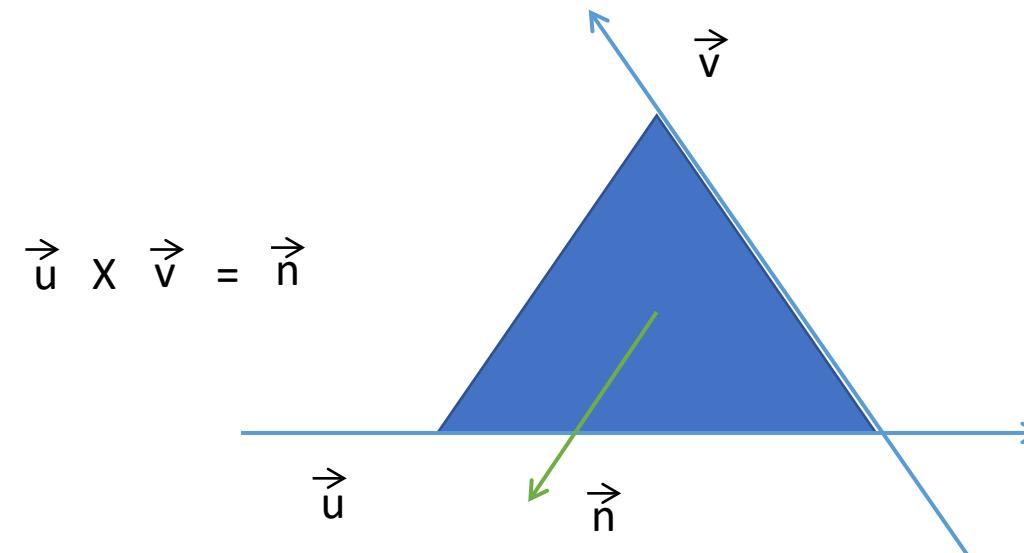
Mesh Normals



$$N_v = (N_1 + N_2 + N_3 + N_4)/4$$

Polygon Normals

- One way of computing a normal vector is to take the cross product of two vectors that lie in the polygon's plane
- We can construct these vectors from the vertices, we can construct two vectors from the first 3 vertices and take their cross product



Polygon Normals

- If the three vertices for the polygon are v_1 , v_2 , and v_3 in counter clockwise order, construct the two vectors:

$$v_2 - v_1$$

$$v_3 - v_2$$

- Take the cross product of these two vectors
- Must do all the polygons in counter clockwise order, cannot just randomly choose two vectors
- Otherwise, normal vectors will have random orientations

Polygon Normals

- For general polygons there can be a problem with this: the three vertices could be close to co-linear
- Only in pathological cases is this a problem with triangles, another reason for using triangles
- If we only use polygon normals our models will look faceted, the polygons are obvious

Normals

- A better approach is to compute a normal vector for each vertex, produces a smoother looking shape
- Once we have per polygon normals we can easily get per vertex normals
- Each vertex appears in multiple polygons, each with a different normal
- Compute the average of the polygon normals to get the vertex normals

Computing Normals

- There is an efficient way of doing this
- Start by constructing a normal table, one entry per vertex, four columns: three for normal vector (x , y and z), one for count of polygons
- Initialize this table to zero

Computing Normals

- Now loop through all the polygons:
 - Compute the polygon normal
 - For each vertex add the polygon normal to the vertex's entry in the normal table, increment the polygon count
- After all the polygons have been processed, loop through the normal table, divide normal components by polygon count
- You will then need to normalize the normal vectors

Algorithm – Part One

Set all entries in normal table to zero

for(all polygons) {

 compute polygon normal

 for(all vertices in polygon) {

 add normal to vertex entry in normal table

 increment count by 1

}

}

Algorithm – Part Two

```
for(each entry in normal table) {  
    divide normal by count  
    normalize the resulting vector  
}
```

Summary

- In today's lecture you learned:
 - Representation of points and vectors
 - Coordinate systems
 - Polygonal models and meshes
 - How to compute normal vectors

Next Classes

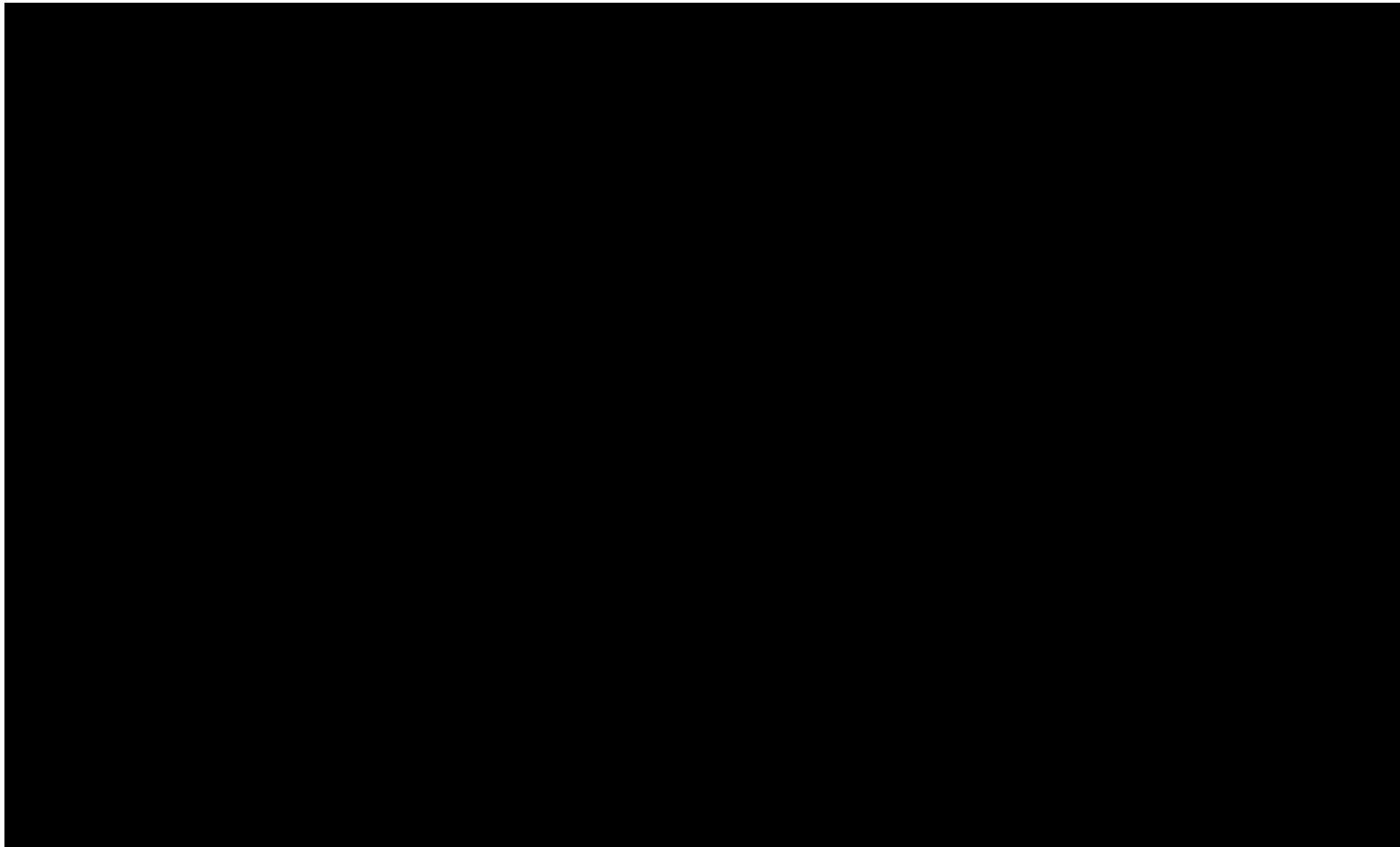
- Transformations in 2D and 3D

CSCI 3090

Transformations

Mark Green
Faculty of Science
Ontario Tech

Knick Knack (1989)



Goals

- By the end of today's class, you will:
 - Use transformation matrices to calculate changes to points and vectors
 - Explain the concept of homogeneous coordinates
 - Be able to program rotations around an arbitrary axis in 3D

Transformations

- Transformations are an important part of computer graphics, and are key to a number of ideas in modeling
- We have mentioned three standard transformations: translation, scale and rotate, and we will examine a few more
- First, we will look at the three standard transformations, and then show how all transformations can be represented

Transformations in 2D

- goal: represent changes and movements of objects in the vector space
- common transformations:
 - **translation**
 - **rotation**
 - **scaling**
 - mirroring
 - shearing
 - combinations of these

Translation

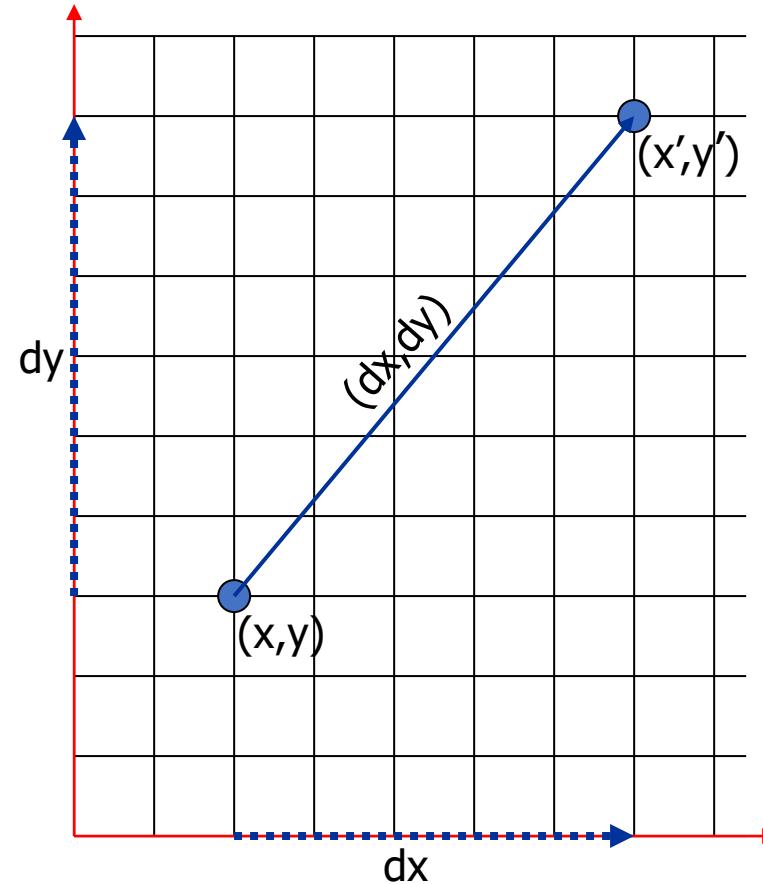
- Translation moves a point from one location to another, it is a displacement applied to the point's coordinates
- This is basically addition, if we apply a translation of (dx, dy, dz) to the point (x, y, z) we end up with the point $(x+dx, y+dy, z+dz)$
- Note: we don't apply translations to vectors, such as a normal vector

2D Translation

- move point $(x, y)^T$ on a straight line to $(x', y')^T$
- represent translation by a translation vector that is added

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} dx \\ dy \end{pmatrix}$$

- vector: movement from one point to another



Scale

- Scale is used to change the size of an object, to make it bigger or smaller
- If the object is centered at the origin scale is just a multiplication
- For point (x, y, z) and scale transformation (sx, sy, sz) the new coordinates are $(sx*x, sy*y, sz*z)$
- Scale makes sense for both points and vectors

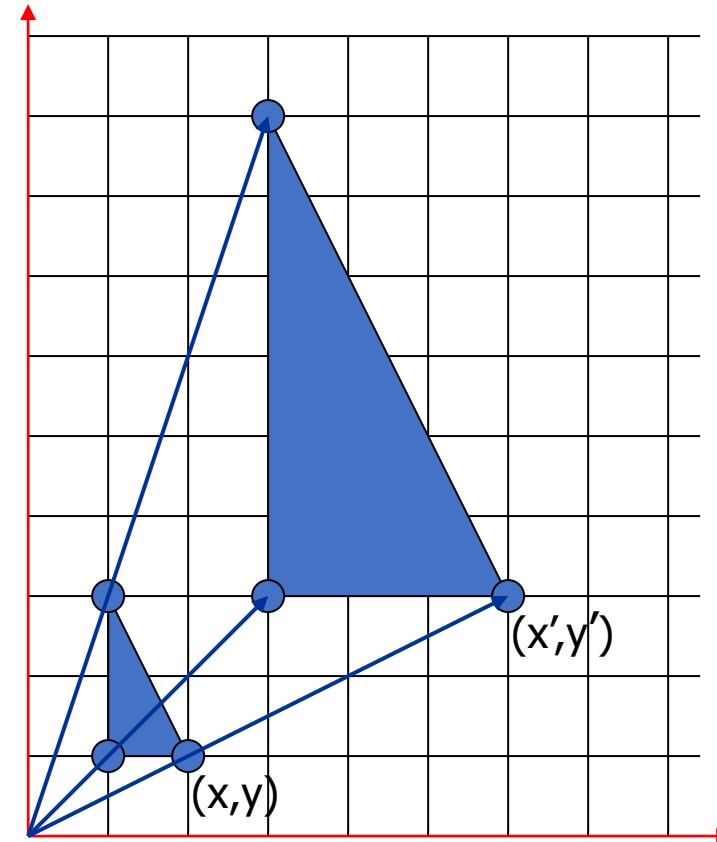
Scale

- Problem: the object must be centered at the origin, what happens if its center is somewhere else?
- We solve this by combining scale with translation
- We translate the object's center to the origin, perform the scale, and then translate back to the original position
- Thus for a scale transformation we have both the scale factors and the center

2D Uniform Scaling

- center of scaling is o
- scaling uniformly in all directions
- stretching of $(x, y)^T$'s position vector by scalar factor α to get $(x', y')^T$
- mathematically: multiplication with α

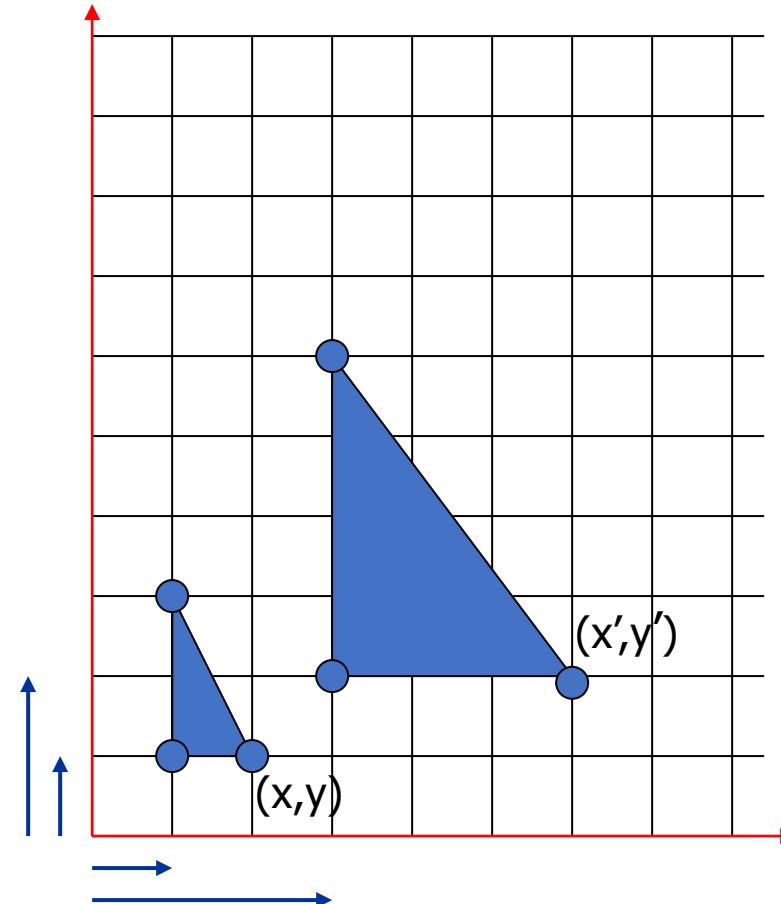
$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \alpha \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \alpha x \\ \alpha y \end{pmatrix}$$



2D Non-Uniform Scaling

- center of scaling is o
- scaling in x-direction by α and in y-direction by β (scaling vector $(\alpha, \beta)^T$)
- mathematically: multiplication with α and β according to axis

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \alpha x \\ \beta y \end{pmatrix}$$



Note

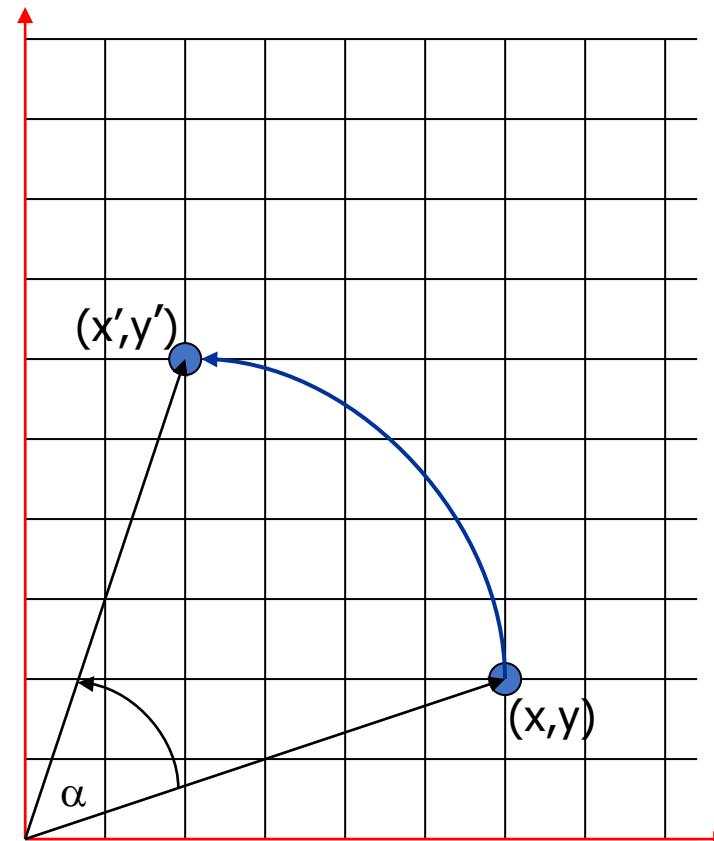
- Some algorithms and graphics systems don't support non-uniform scaling
- It is best to stick with uniform scaling
- If you want a non-uniform scale build it into the model
- Example: don't scale a square to construct a rectangle, use the vertex coordinates for the rectangle you want

2D Rotation

- Rotation is harder, so we will start with 2D
- Rotation is about the origin, it is the center of the rotation, and we can view the z axis as the axis of rotation
- This is what we are used to in trig, the rotation is counter-clockwise about the origin and we can use sine and cosine to compute the coordinates

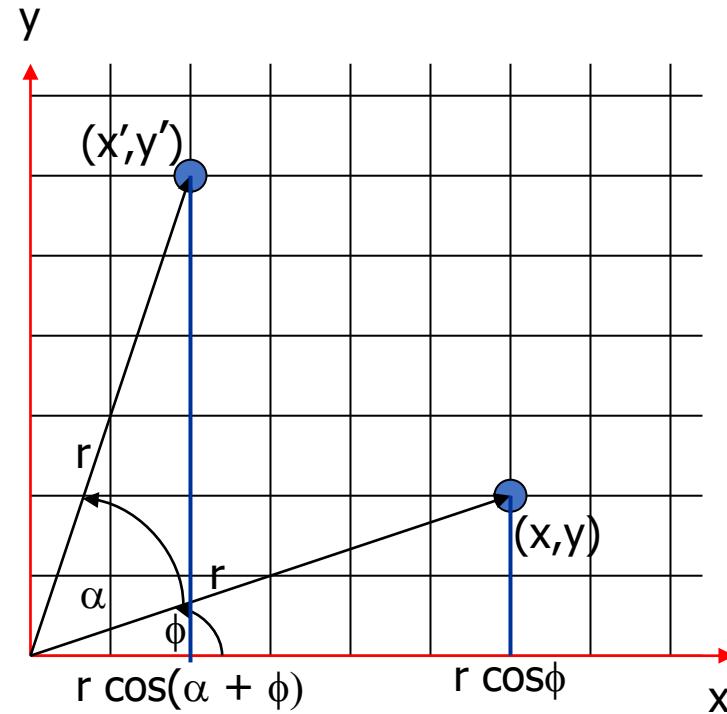
2D Rotation

- center of rotation is o
- point $(x, y)^T$ is rotated by an angle α around o to obtain $(x', y')^T$
- positive angles α mean counter-clockwise rotation



2D Rotation

- distances $(x,y)^T - o$ & $(x',y')^T - o$ are both r

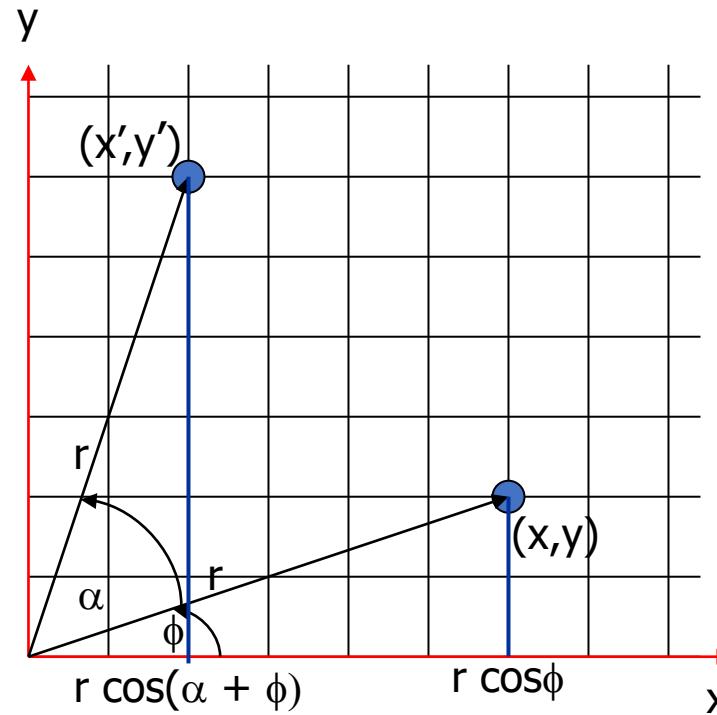


2D Rotation

- distances $(x,y)^T - o$ & $(x',y')^T - o$ are both r

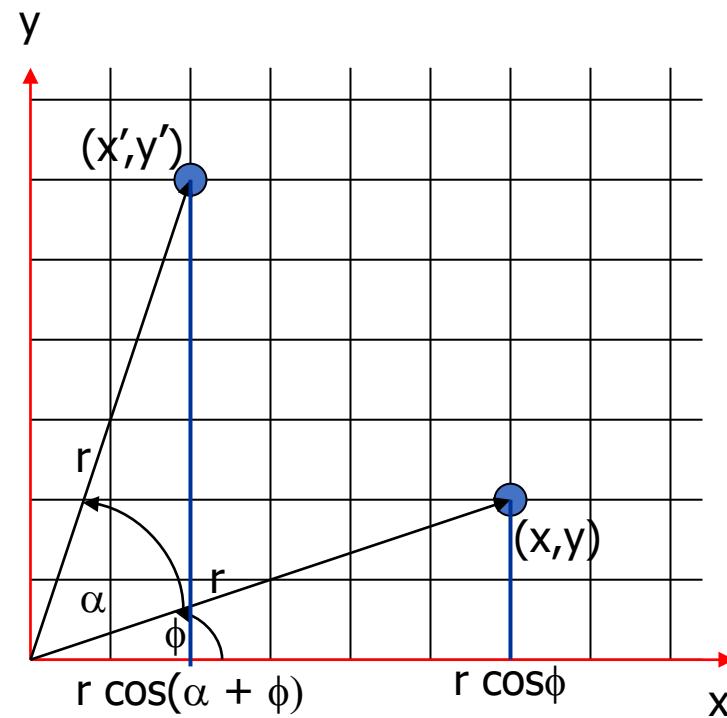
$$\cos\phi = x/r$$

$$\sin\phi = y/r$$



2D Rotation

- distances $(x,y)^T - o$ & $(x',y')^T - o$ are both r
- $x = r \cos\phi$ and $y = r \sin\phi$



2D Rotation

- distances $(x,y)^T - o$ & $(x',y')^T - o$ are both r
- $x = r \cos\phi$ and $y = r \sin\phi$
- $x' = r \cos(\alpha+\phi)$ and $y' = r \sin(\alpha+\phi)$

Recall:

$$\cos(\alpha + \phi) = \cos(\alpha)\cos(\phi) - \sin(\alpha)\sin(\phi)$$

$$\sin(\alpha + \phi) = \sin(\alpha)\cos(\phi) + \cos(\alpha)\sin(\phi)$$

2D Rotation

- distances $(x,y)^T - o$ & $(x',y')^T - o$ are both r
- $x = r \cos\phi$ and $y = r \sin\phi$
- $x' = r \cos(\alpha+\phi)$ and $y' = r \sin(\alpha+\phi)$
- $x' = r \cos\alpha \cos\phi - r \sin\alpha \sin\phi$
 $= x \cos\alpha - y \sin\alpha$

2D Rotation

- distances $(x,y)^T - o$ & $(x',y')^T - o$ are both r
 - $x = r \cos\phi$ and $y = r \sin\phi$
 - $x' = r \cos(\alpha+\phi)$ and $y' = r \sin(\alpha+\phi)$
-
- $x' = r \cos\alpha \cos\phi - r \sin\alpha \sin\phi$
 $= x \cos\alpha - y \sin\alpha$
 - $y' = r \sin\alpha \cos\phi + r \cos\alpha \sin\phi$
 $= x \sin\alpha + y \cos\alpha$

Two Views

- For both scale and rotation, the transformation is about the origin
- This can be viewed as a transformation of the entire world, the entire scene we are constructing
- If we only want to transform one object we need to translate it to the origin first
- Apply the scale or rotations
- Then translate it back to its original location
- Thus, we need to be able to combine transformations

Unifying The Transformations

Transformations

- We would like to have a single representation for all transformations
- This makes them easier to work with mathematically, and it simplifies the programming
- Luckily there is such a representation
- We will start by examining the 2D scale and rotation transformations

Transformations

- In the case of scale we multiply x and y by the scale factors
- In the case of rotating (x, y) by an angle of α we have
 - $x' = x \cos\alpha - y \sin\alpha$
 - $y' = x \sin\alpha + y \cos\alpha$
- Do you see anything in this form?

Transformations

- We can represent both of these transformation by 2x2 matrices, and we can represent points by column vectors
- Then for scale we get the following transformation matrix:

$$\begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}$$

Transformations

- Given the point (x,y) , we can apply the transformation in the following way:

$$\begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} s_x * x + 0 * y \\ 0 * x + s_y * y \end{bmatrix} = \begin{bmatrix} s_x * x \\ s_y * y \end{bmatrix}$$

- We can do the same thing for rotation, to give the following matrix:

$$\begin{bmatrix} \cos \alpha & -\sin \alpha \\ \sin \alpha & \cos \alpha \end{bmatrix}$$

2D Rotation

- $x' = x \cos\alpha - y \sin\alpha$
- $y' = x \sin\alpha + y \cos\alpha$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Combining Transformations

- Consider applying two transformations M and N to the point x
- The first transformation gives $x' = Mx$, and the second transformation gives $x'' = Nx'$
- But we can combine these transformations
 - $x'' = N(Mx)$
 - $x'' = (NM)x$
- We can combine transformations by multiplying their matrices

Combining Transformations

- We can represent any sequence of scale and rotate transformations by a single matrix
- This can save us a lot of time, when transforming all of the vertices in a mesh we only need one matrix multiply per vertex
- This is a big savings if there are 2 or more transformations

Concatenating Transformations

- efficiency example
 - model with 1,000,000 vertices
 - operations: scaling, rotation, translation
 - individual matrix multiplications:
$$3 \times (9M + 6A) \times 1,000,000 = \begin{array}{l} 27,000,000M \\ 18,000,000A \end{array}$$
 - one concatenated matrix :
$$(9M + 6A) \times 1,000,000 = \begin{array}{l} 9,000,000M \\ 6,000,000A \end{array}$$

What about Translation?

- There is no 2×2 matrix that will perform a 2D translation, since translation involves adding a constant, not multiplying
- But, there is a trick that we can use, we multiply a row of the matrix by the column vector, we add the results of the products
- **What if one component of the column vector was 1?**
- We could multiply it by the constant translation and it would be part of the sum

Translation Transformation

- To do this we need to add an extra component to each vector, that is (x, y) becomes $(x, y, 1)$
- We also need to use a 3×3 matrix instead of a 2×2 matrix, in this case:

$$\begin{bmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{bmatrix}$$

Translation Transformation

- Translation works in the following way:

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

Wait... we can translate
points, but not vectors!

Translating Vectors?

- We said that we can translate points, but not vectors!
- Something interesting happens here, If we use $(x, y, 0)^T$ for vectors, the translate part of the transformation will not be applied, so we can use the same matrices for both points and vectors:

$$\begin{bmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 0 \end{bmatrix} = \begin{bmatrix} x \\ y \\ 0 \end{bmatrix}$$

- Thus, we have a uniform representation for all transformations

Homogeneous Coordinates

- Adding the third component to a 2D point or vector is called **homogenous coordinates**
- We will do the same thing in 3D, use a 4 vector for its coordinates: (x, y, z, w) -> w is usually 1
- Homogenous coordinates are widely used in graphics, they not only help with translation, we will also use them for perspective projections

Homogeneous Coordinates

- What are we doing here??
- In the case of 2D we are projecting our points to 3D space: $(x, y) \rightarrow (xw, yw, w)$
- We can then get back to 2D by dividing by w : $(xw, yw, w) \rightarrow (x, y)$
- For points we have $w=1$, so this is all trivial
- For vectors we have $w=0$, this results in divide by zero
- Vectors are points at infinity!

Transformations

- We now need to convert our matrices for scale and rotate to 3x3 matrices. Just need to add a column and row:

$$\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} \cos \phi & -\sin \phi & 0 \\ \sin \phi & \cos \phi & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Transformations

- To see how this all works, consider a scale of (s_x, s_y) about the point (x, y)
- We need to translate (x, y) to the origin, apply the scale and translate back
- We can use the following matrix:

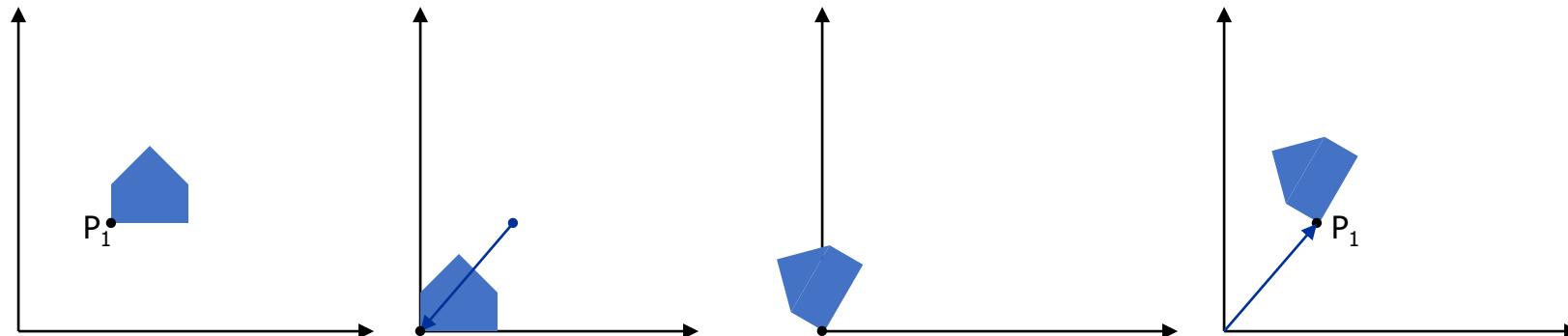
$$\begin{bmatrix} 1 & 0 & x \\ 0 & 1 & y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -x \\ 0 & 1 & -y \\ 0 & 0 & 1 \end{bmatrix}$$

Transformations

- But, isn't this backwards???
- Remember that the vector representing the point is on the right side of the matrices
- This means that we read the transformation matrices from right to left
- The same thing happens in our program code, we need to read the transformations from the bottom up
- This can be a source of bugs in your programs

Using Concatenated Transformations

- example: rotation around arbitrary point P_1
- three steps:
 - Translation of P_1 to the origin
 - rotation
 - inverse translation to bring back to P_1
- $P' = T_+ \cdot R \cdot T_- \cdot P$



Homogeneous Coordinates in 2D

- general transformation matrix

$$\begin{pmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{pmatrix}$$

scaling

rotation

translation

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

- what about normalizing w' when done?

- in all basic transformations we get $w' = 1$
- no normalization necessary

Transformations in 3D

Transformations

- In 3D we use 4x4 transformation matrices
- The matrices for scale and translation are straight forward generalizations of the 2D ones:

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Transformations

- Rotation is the problem, since we can rotate about any 3D axis
- In 2D there was only one axis, so we really didn't need to worry about it
- In 3D we really can't construct the transformation matrix until we know what the **axis of rotation** is
- Historically this has caused a lot of problems, *and incorrect code*

Specifying the Axis of Rotation

- In 3D we can have any axis of rotation, it can be any line or vector in 3D space
- When we specify a 3D rotation we need to specify:
 - rotation angle
 - center of rotation
 - axis of rotation
- To get the correct center of rotation, just use translation (as we saw with scaling)

3D Rotation Matrices

- If we restrict our rotations to be about the x, y or z axis, then its fairly easy to construct the rotation matrices:

$$\begin{array}{c} \textcolor{orange}{X} \\ \left[\begin{array}{cccc} 1 & 0 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi & 0 \\ 0 & \sin\phi & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{array} \right] \end{array} \quad \begin{array}{c} \textcolor{orange}{Y} \\ \left[\begin{array}{cccc} \cos\phi & 0 & \sin\phi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\phi & 0 & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{array} \right] \end{array} \quad \begin{array}{c} \textcolor{orange}{Z} \\ \left[\begin{array}{cccc} \cos\phi & -\sin\phi & 0 & 0 \\ \sin\phi & \cos\phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{array} \right] \end{array}$$

Euler Angles

- Since these matrices are easy, Euler angles became popular
 - Euler angles: a series of three rotations about the 3 coordinate axis
- Many versions of Euler angles, depend upon the order of the axis
- None of these orders work in general
- Euler proved that you cannot use just 3 numbers to specify an arbitrary 3D rotation

Euler Angles

- Why don't Euler angles work?
- A 90° rotation about the X will rotate the Y axis onto the Z axis
- We can no longer rotate about the original Y axis, we have lost one degree of freedom
- In mechanical engineering this is called gimbal lock
- This is also the reason why we lost a number of satellites in the early years of the space race

Moving to Arbitrary Rotation Vectors

- To get around this problem we need to take a closer look at rotation matrices
- Lets take the matrix for z axis rotation and use it to rotate its first row

$$\begin{bmatrix} \cos \phi & -\sin \phi & 0 & 0 \\ \sin \phi & \cos \phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos \phi \\ -\sin \phi \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$



- Notice that this vector rotates onto the **x axis**, the same thing happens to the other two rows

Rotation Matrix Identity

- That is, a rotation matrix rotates its own rows onto the x, y and z axis, **this is true for all rotation matrices**
- Something else interesting comes out of this, the rows of a rotation matrix R form its transpose R^T so we have the following relationship:
$$R R^T = I$$
- Thus, the **inverse of a rotation matrix is just its transpose** (recall $RR^{-1}=I$) which is easy to compute

Transformations

- So R^T will rotate the x, y and z axis onto the rows of R
- This gives us an easy way of constructing a rotation matrix if we have three orthogonal vectors that we want to rotate onto the x, y and z axis, or vice versa

Rotating around \vec{u}

- Given three orthogonal vectors $\vec{u}, \vec{v}, \vec{w}$
- Want to rotate by ϕ around \vec{u}

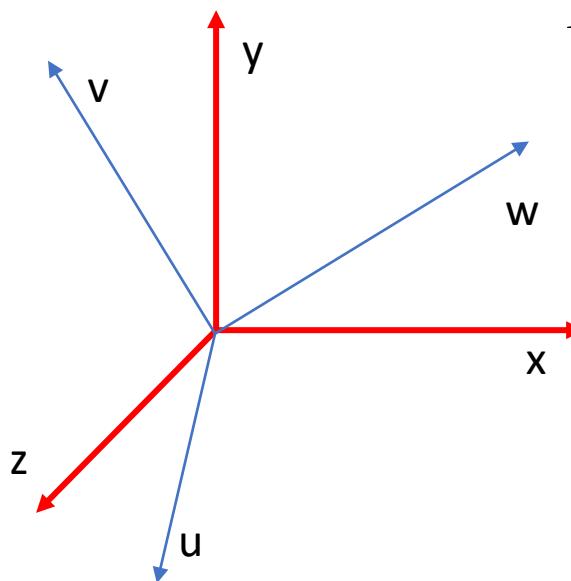
Steps:

- Rotate onto $\vec{x}, \vec{y}, \vec{z}$ using R
- Use x rotation matrix to rotate by ϕ
- Rotate axes back to $\vec{u}, \vec{v}, \vec{w}$ using R^T

Rotating around \vec{u}

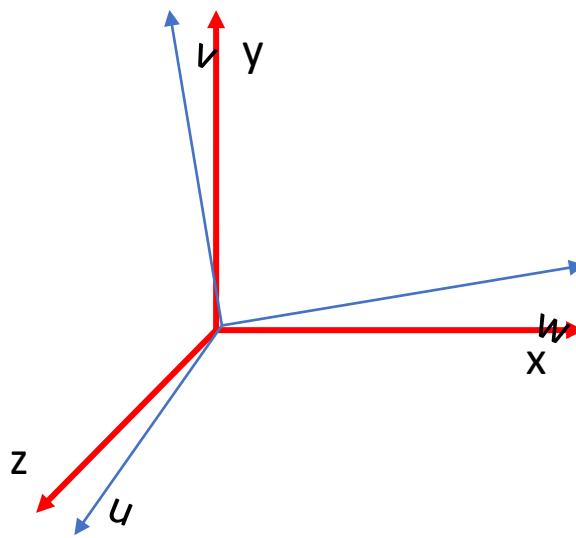
- R is:
- $$\begin{bmatrix} u1 & u2 & u3 & 0 \\ v1 & v2 & v3 & 0 \\ w1 & w2 & w3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotate arbitrary orthogonal vectors to axes

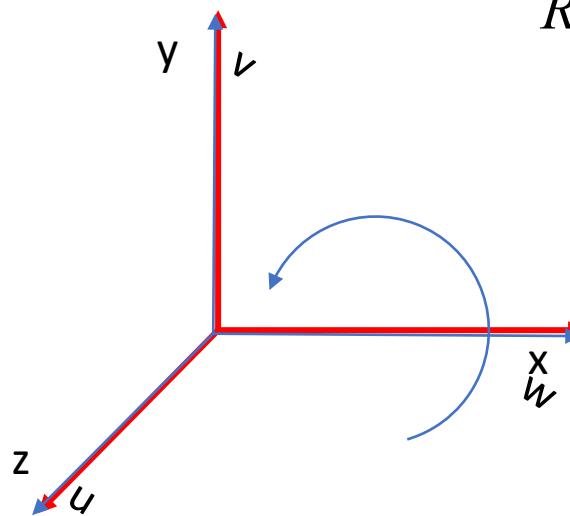


$$R = \begin{bmatrix} u_1 & u_2 & u_3 & 0 \\ v_1 & v_2 & v_3 & 0 \\ w_1 & w_2 & w_3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotate arbitrary orthogonal vectors to axes

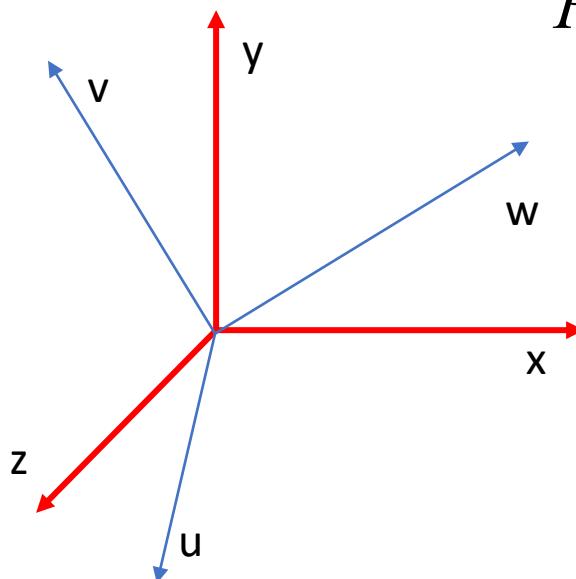


Rotate arbitrary orthogonal vectors to axes



$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \phi & -\sin \phi & 0 \\ 0 & \sin \phi & \cos \phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotate arbitrary orthogonal vectors to axes



$$R^T = \begin{bmatrix} u_1 & v_1 & w_1 & 0 \\ u_2 & v_2 & w_2 & 0 \\ u_3 & v_3 & w_3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Arbitrary Axis?

- Now what about rotating about an arbitrary axis \vec{a} , how are we going to do that?

Rotating Around an Arbitrary Vector

- This is easy, we rotate \vec{a} onto the z axis, perform a rotation about the z axis, and then use the inverse of the first rotation

Transformations

- We have one vector \vec{a} , we can **normalize it** and **make it the third row of the matrix**, but we need two more orthogonal vectors, \vec{u} and \vec{v} to complete R
- These vectors will not be unique, since we can rotate them about the a axis and they will still work

Transformations

- To start with we need a vector \vec{t} that is not collinear with \vec{a}
- We can do this by setting \vec{t} to \vec{a} , and then changing the smallest magnitude component to 1
- We now construct \vec{u} in the following way

$$\vec{u} = \frac{\vec{t} \times \vec{a}}{\|\vec{t} \times \vec{a}\|}$$

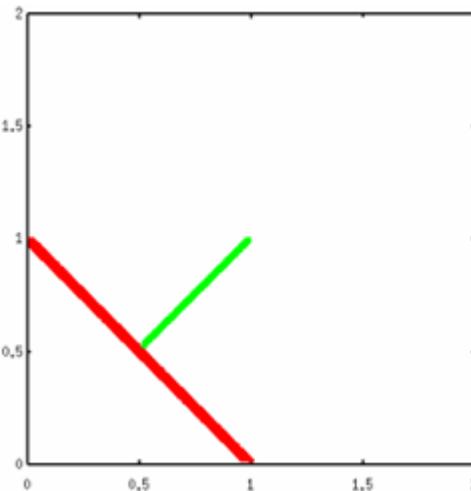
Transformations

- Now \vec{u} is a **unit vector** orthogonal to \vec{a}
- We can get one more orthogonal vector by doing another cross product, $\vec{v} = \vec{a} \times \vec{u}$
- Our rotation matrix R has \vec{u} as the first row, \vec{v} as the second row and \vec{a} as the third row
- If $R_z(\phi)$ rotates by ϕ about the z axis our matrix is:
$$R^T R_z(\phi) R$$

Transforming Normal Vectors

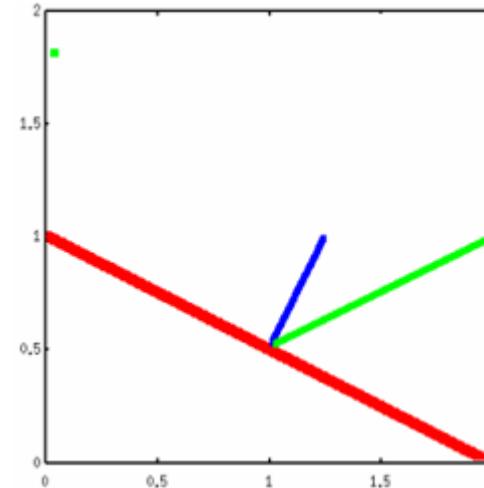
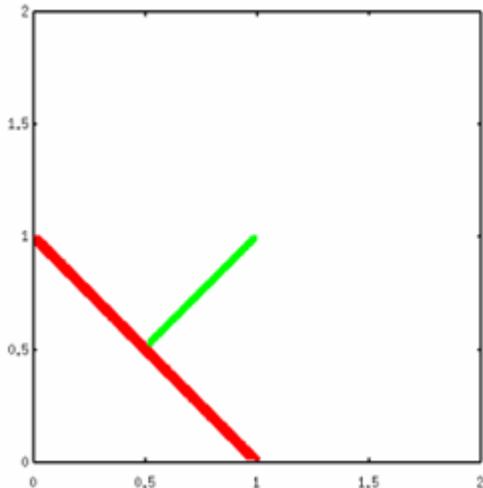
- There is one detail that we need to deal with here, and that's how **normal vectors transform**
- Consider a vector \vec{n} that is normal to a surface, if we transform \vec{n} and the surface by the same matrix M , the transformed \vec{n}' may or may not be normal to the transformed surface
- **Affine transformations which include scale do not preserve angles between lines!**

Transforming Normal Vectors



red line \rightarrow line of intersection of the plane $x=y$ with the XY plane
green line \rightarrow the normal to this plane at the point [0.5 0.5 0]
(direction of this line is [0.707 0.707 0])

Transforming Normal Vectors



apply a non-uniform scaling matrix that scales by 2 in the X-axis
green line -> is the 'unit' plane normal with the non-uniform scaling matrix applied (direction has become [1.4 0.707 0])

Green line no longer normal to the transformed plane (should be [0.48 0.89 0] which is drawn in blue)

Transforming the Normal

- Considering a tangent vector \vec{t} to the surface
- We have the following relationship between \vec{n} and \vec{t} :
$$\vec{n}^T \vec{t} = 0$$
- If M is the transformation matrix, then the transformed tangent vector is $M\vec{t}$
- The transformed normal \vec{n}' must satisfy $\vec{n}'^T (M\vec{t}) = 0$
- What is \vec{n}' ?

Recall

- Perpendicular vectors have dot product of 0!
- $(AB)^T = B^T A^T$

Transforming the Normal

- Let $\vec{n}' = L\vec{n}$ for some L
- $\vec{n}'^T (M\vec{t}) = (L\vec{n})^T (M\vec{t})$
- $= \vec{n}^T L^T (M\vec{t})$
- $= \vec{n}^T (L^T M)\vec{t} = 0 = \vec{n}^T \vec{t}$

$$(AB)^T = B^T A^T$$

Transforming the Normal

- Let $\vec{n}' = L\vec{n}$ for some L
- $\vec{n}'^T (M\vec{t}) = (L\vec{n})^T (M\vec{t})$
- $= \vec{n}^T L^T (M\vec{t})$
- $= \vec{n}^T (L^T M)\vec{t} = 0 = \vec{n}^T \vec{t}$
- So, $(L^T M) = I$
- $L^T M M^{-1} = M^{-1}$
- $L^T = M^{-1}$
- $L = (M^{-1})^T$

$$(AB)^T = B^T A^T$$

The matrix to transform a normal is the inverse transpose of the original transformation matrix!

Transformations

- From this we can conclude that the transformed normal vector $\vec{n}' = (M^{-1})^T \vec{n}$
- That is we apply the transpose of the inverse transformation matrix to the normal vector
- Note that the resulting vector may no longer be unit length, so normalization may be required
- This also gives us some insight into when M can be used to transform n

Transformations

- If M is a rotation matrix its inverse is just its transpose, so we are essentially transposing M twice
- So if M contains only rotations and translations we can apply it directly to n , otherwise we must use the inverse transpose of M
- Note that rotation and translation are rigid body transformations, so we would expect them to preserve the normal vector

Additional Transformations

Reflection Transformation

- There are two other transformations that we haven't looked at yet
- The reflection transformation reflects an object about a line in 2D and a line or a plane in 3D
- In 2D, a reflection in the y axis is given by:

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

Reflection Transformation

- We can get a reflection in x in the same way
- If we want to reflect about $x=5$ instead of $x=0$, we need to combine the reflection with two transformation matrices to translate to $x=0$ and back again
- If we want to reflect through an arbitrary axis, we need to translate to the origin, rotate the axis onto the x or y axis, perform the reflection and then undo all the transformations

Reflection Transformation

- The reflection transformations generalize to 3D in the obvious way with reflections about a line becoming reflections about a plane
- In the case of reflections about a line we have two entries with negative signs, giving the following matrix for the z axis:

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

Note

- Be careful with reflections in 3D!
- They can change a right handed coordinate system into a left handed one
- If there is an odd number of -1's the handedness changes
- Why do I care?
- Things will not move the way you expect them to, they will move in the opposite direction!

Shear Transformation

- The other transformation is the shear transformation, which basically slides one end of the object over, like pushing on a deck of cards
- In 2D we can shear in the x direction or the y direction using:

$$\text{shear - x} = \begin{bmatrix} 1 & s \\ 0 & 1 \end{bmatrix} \quad \text{shear - y} = \begin{bmatrix} 1 & 0 \\ s & 1 \end{bmatrix}$$

Shear and Reflection Transformations

- There are no `glm` calls for reflection transformation, there is a `glm` extension library that constructs matrices for the shear transformations
- For the reflection transformation you need to build the transformation yourself by setting the individual matrix entries

Decomposing Transformations

Advanced

Decomposing Transformations

- So far we have started with transformations and then constructed the transformation matrix
- This is the normal process, but sometimes you need to go the other way
- You are given a transformation matrix and you want to determine the transformations it performs
- We cannot determine the exact process used to construct the matrix, but we can decompose it into simpler transformations

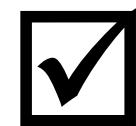
Decomposing Transformations

- Given any transformation matrix we can decompose it into a translation, two rotations and a scale, that is a total of four matrices
- As we have seen the translation is always in the last column of the matrix, so we can easily extract that part of the transformation
- We can then construct a pure translation matrix

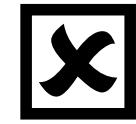
Decomposing Transformations

- The problem is where does this transformation appear in the sequence of transformations?
- There are two choices, either left or the right side:

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



$$\begin{bmatrix} a & b & 0 \\ c & d & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & x \\ 0 & 1 & y \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} a & b & ax+by \\ c & d & cx+dy \\ 0 & 0 & 1 \end{bmatrix}$$



Decomposing Transformations

- Now all we need to worry about is the upper left 2×2 or 3×3 sub-matrix
- If this sub-matrix is symmetric we can perform Eigen value decomposition on it to get a rotation matrix and a scale matrix, that is RSR^T
- S is a diagonal matrix, with its entries being the Eigen values of the sub-matrix, and the columns of R are the Eigen vectors of the sub-matrix
- It is fairly easy to interpret both of these matrices as simple transformations

Decomposing Transformations

- If the sub-matrix is not symmetric we need to do an SVD (Singular Value Decomposition) to decompose it
- In this case we get a scale matrix and two different rotation matrices
- That is, the sub-matrix is given by USV , where S is the scale matrix and U and V are the rotation matrices
- You can find techniques for performing these operations in books on matrix computations, they also available in many software libraries

Decomposing Transformations

- Since the sub-matrices are small, 2x2 or 3x3 we can use analytical methods to find the Eigen values and Eigen vectors and the SVD as well
- The decomposition will tell us if we have pure rotation and translation, or whether a scale is involved
- It will also allow us to use the standard glm transformation procedures

Summary

- In today's lecture you learned:
 - Standard transformations
 - Homogeneous coordinates
 - Representations of transformation as matrices
 - Mathematics of matrix composition

Next Classes

- Hierarchical modelling
- Introduction to Implicit and Parametric Geometry

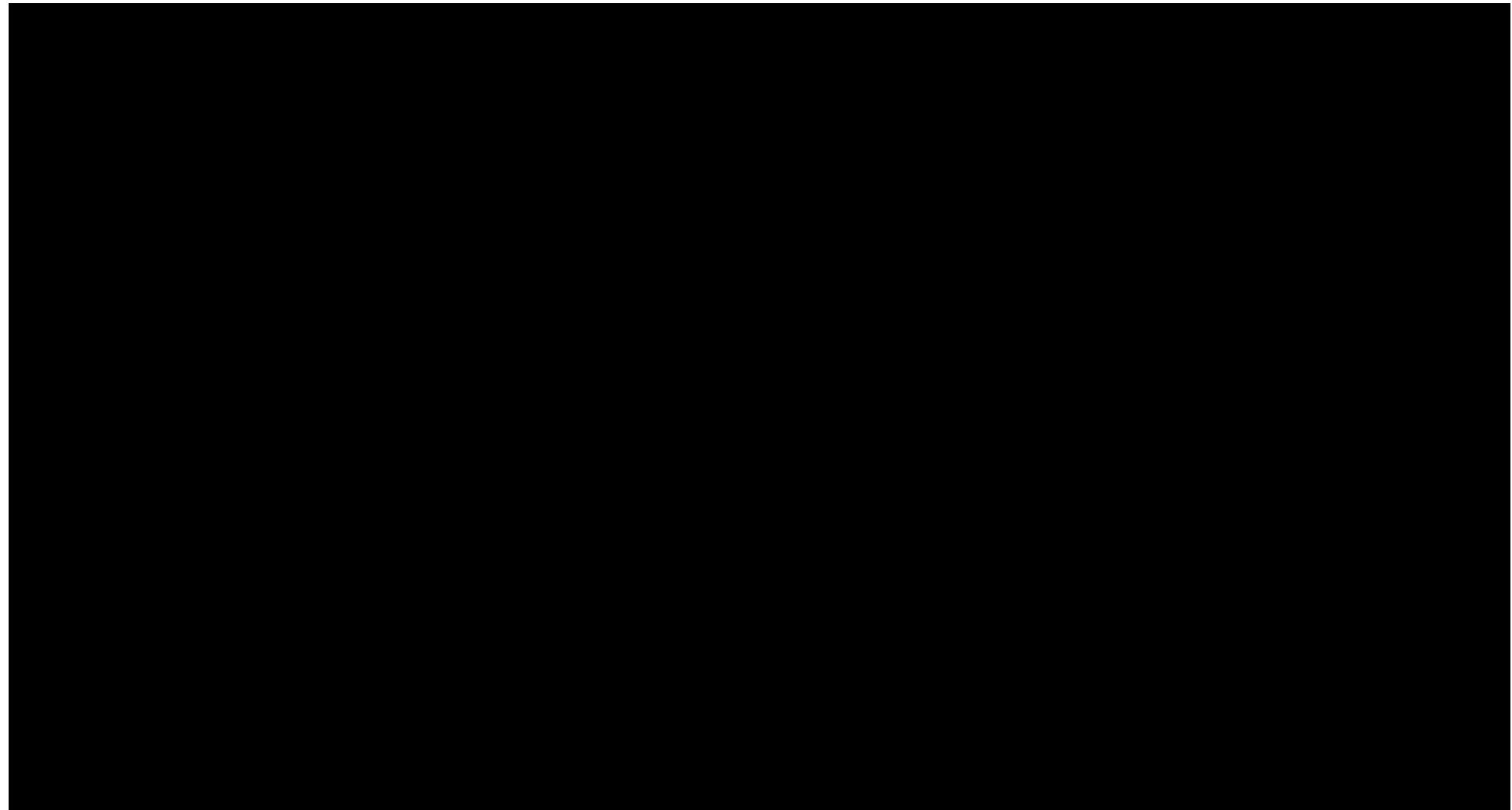
CSCI 3090

Hierarchical Modeling

Mark Green

Faculty of Science

Ontario Tech



Goals

- By the end of today's class, you will:
 - Create hierarchical models describing the relative geometry of complex objects and scenes
 - Be able to implement hierarchical models in OpenGL
 - Introduced to scene graphs

Where do Models Come From?

- We need a model in order to display something, but where do these models come from
- For very simple objects we can construct them by hand, enter the list of vertices and the triangle indices
- We could load a model from a file, for example vase.obj
- But, where did that come from?
- In most cases they are produced by modeling programs, such as Blender or Maya

Where do Models Come From?

- Another possibility is to write program code to create the model
- This is more than just listing the vertices and indexes
- It involves writing procedures that calculate the vertices and indexes
- Once we have the procedures, producing the model is quite easy
- The trick is writing the procedures
- This lecture presents an example of this process

Hierarchical Modeling

- Real objects are usually made up of multiple parts, quite often they can move where these parts come together
- We would like to have a modeling scheme that reflects this, such a modeling scheme is called **hierarchical modeling**

Hierarchical Modeling

- Hierarchical modeling is based on using a set of simple shapes or objects to build a more complex object
- As the name implies a **tree structure** is used to combine these simple objects
- The parent node is composed of the objects or shapes in its child nodes
- But, how do we do this combination?

Hierarchical Modeling

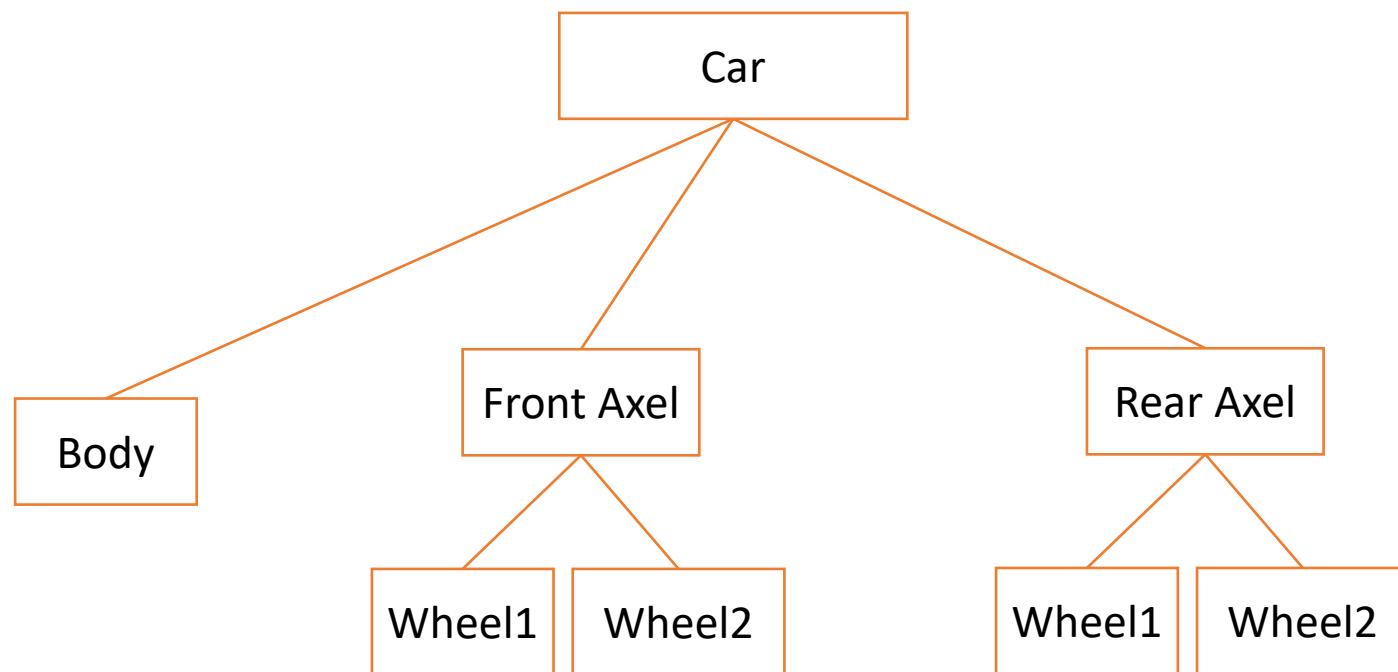
- Each edge from the parent node to a child node has a transformation attached to it
- This transformation is used to convert from the child coordinate space to the parent coordinate space
- It places the child at the correct position and orientation **within the parent's coordinate space**

Hierarchical Modeling

- The root of the tree represents the entire object, this is sometimes called the world or global coordinate system
- As we go down the tree we multiply the transformation matrices to get the transformation from the child node to the root node
- That is we multiply the matrices on the path from the root to the child to get the matrix that transforms the child to the global coordinate system

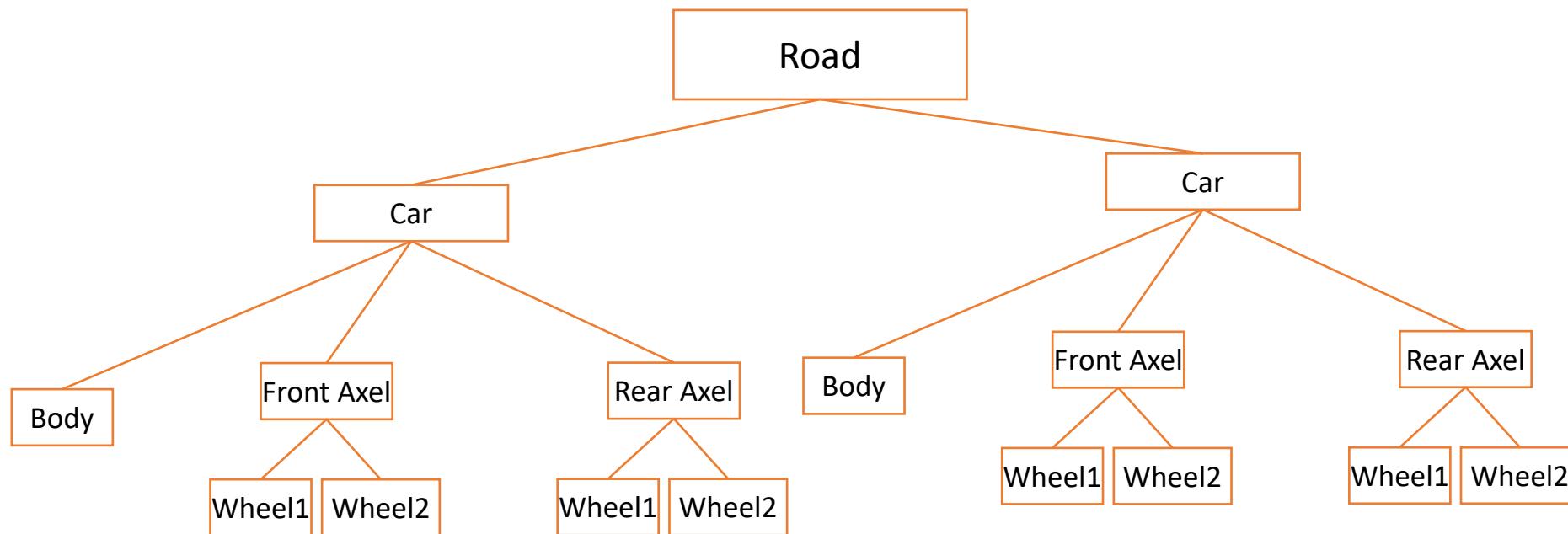
Hierarchical Modeling

- As an example consider the following very simplistic model of a car



Scene Graph

- Placing this model into a larger scene creates a **scene graph**, used to keep track of complex geometries



Hierarchical Transformations

- We use transformations to position the body, axels, and four wheels
- To move the car **we only need to transform the root node**, this transformation is inherited by all of the other nodes and they will move as well
- To rotate the wheels as the car moves we need to **add a rotation to each of the edges leading to an axel**

Repetition in the Hierarchy

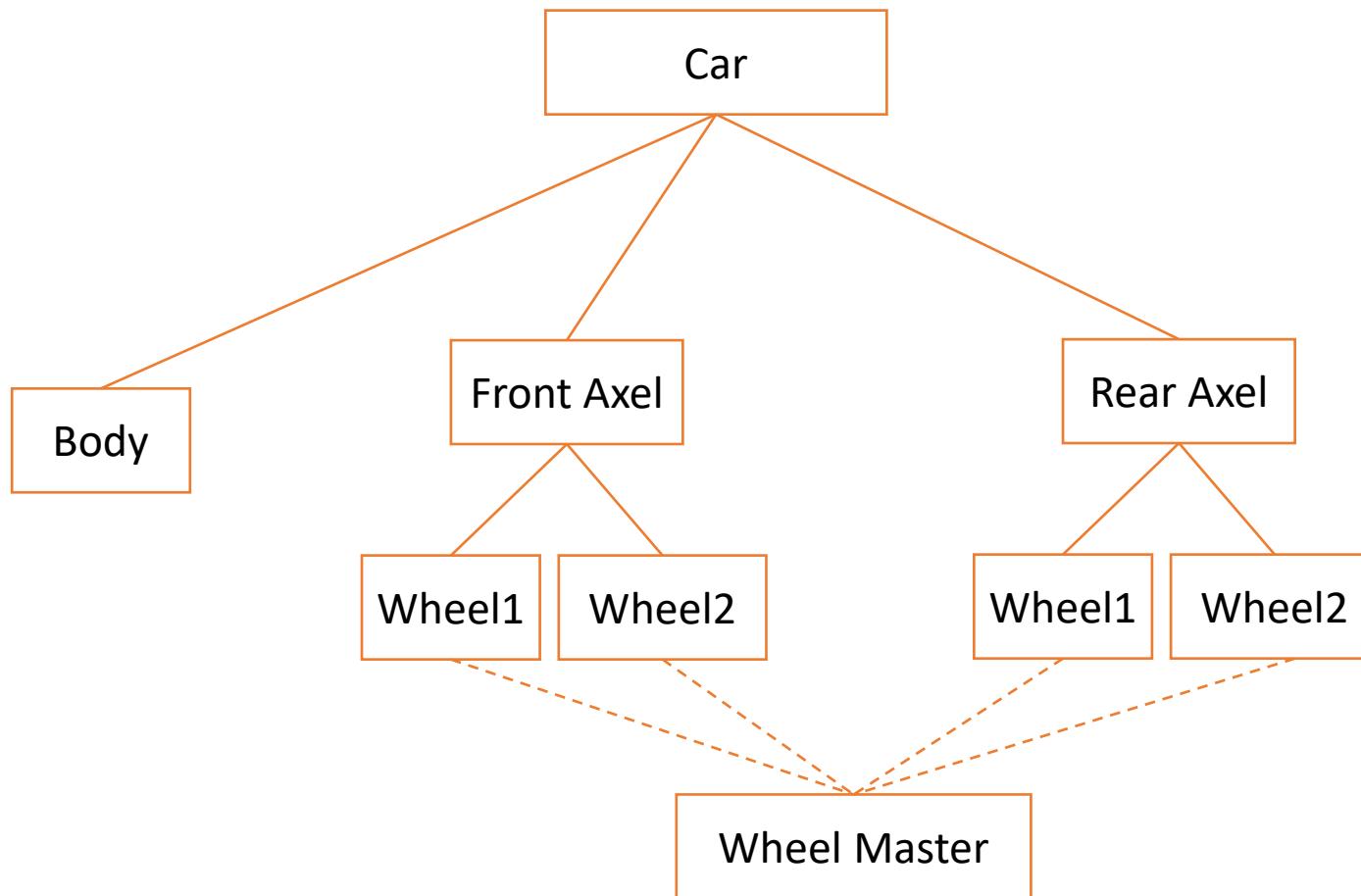
- All four wheels look the same, so there is no reason to repeat the geometry
- The only difference is the position and orientation of the wheel
- This can easily be handled by a transformation

Master / Instance Structure

- This is one of the powers of hierarchical modeling, we can define the geometry once, **called a master**
- We can then make **copies**, called **instances**, and place them using transformations
- If we need to change the geometry we only need to change the master, all of the instances will be updated automatically

Hierarchical Modeling

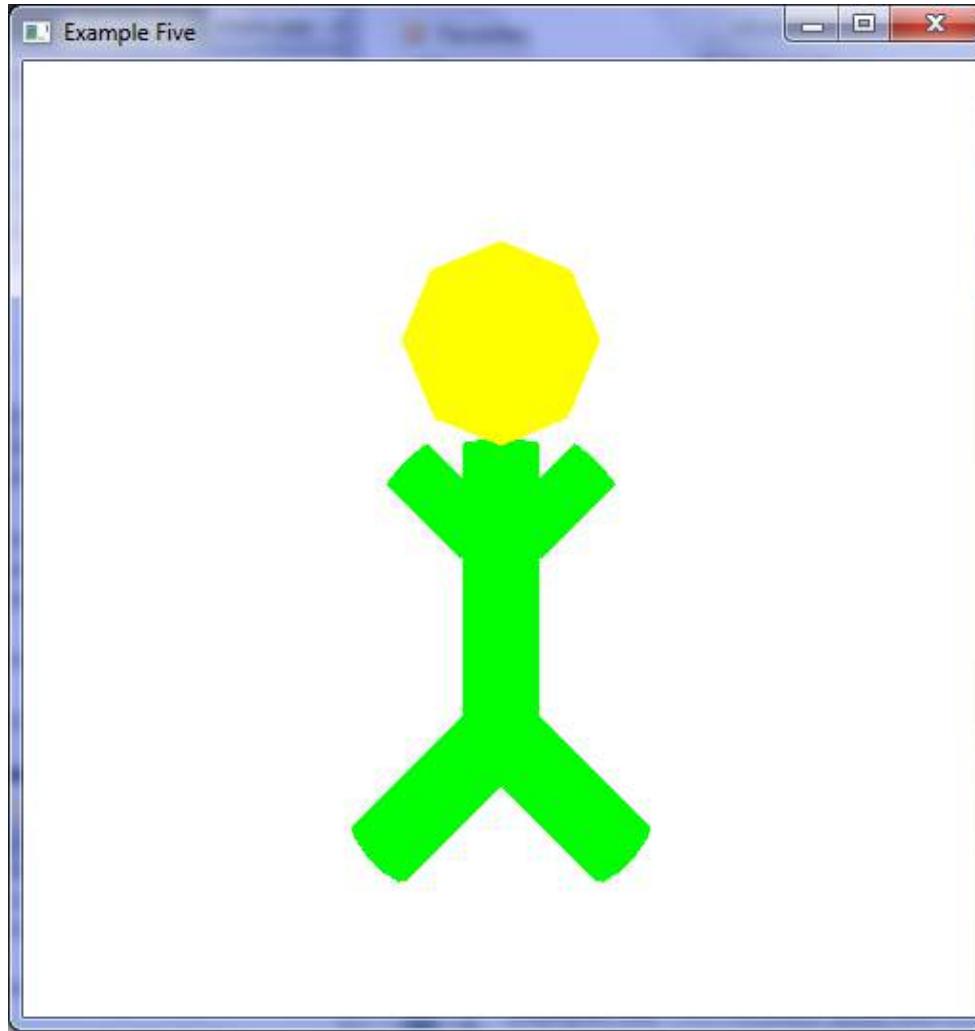
- So our revised model becomes:



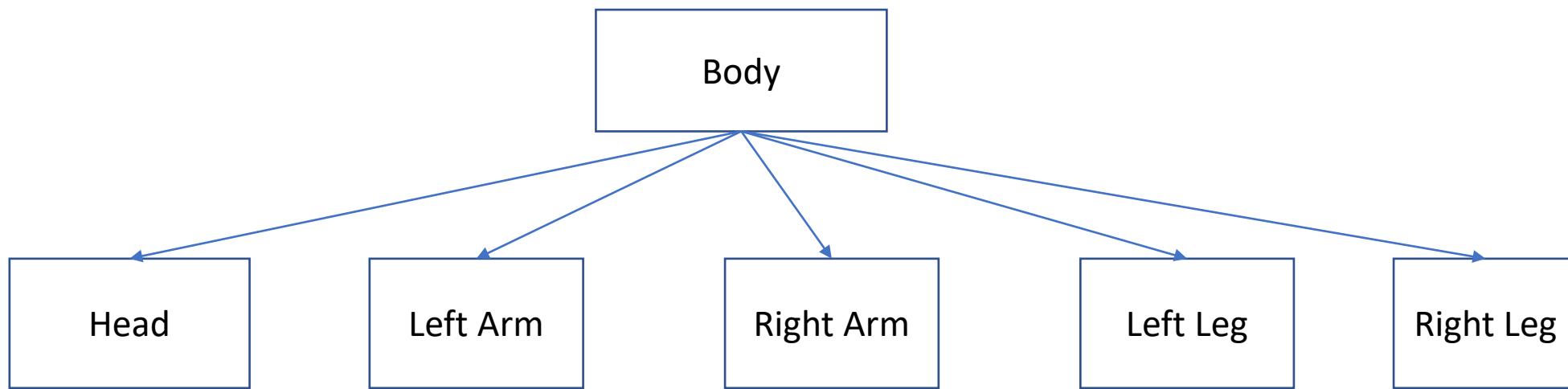
Stick Man Example

- To show how this all works in OpenGL we will produce a stick man
- This stickman is made up of cylinders and not just lines
- The stickman is shown on the next slide, notice that **every part of the body is a cylinder**

Hierarchical Modeling



Hierarchical Modeling



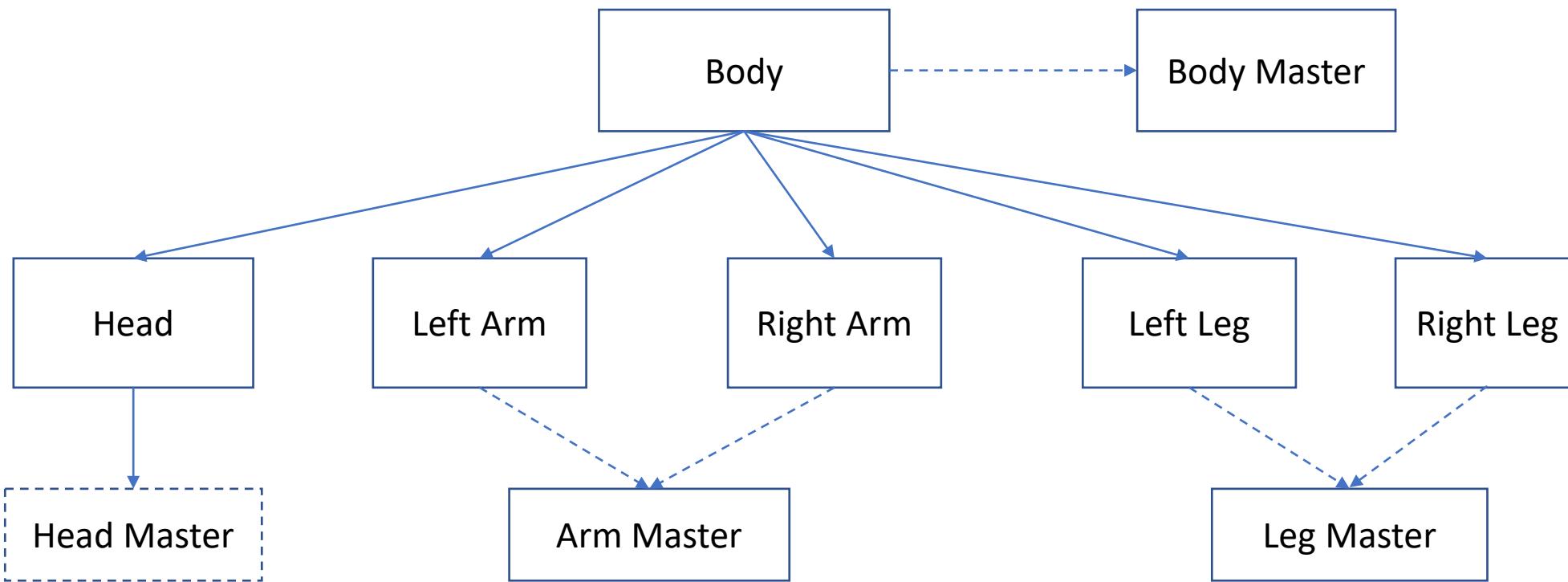
Hierarchical Modeling

- Early versions of OpenGL directly supported hierarchical modeling
- They provided a matrix stack that made it easy to traverse the model
- With vertex shaders we can no longer do this, since the vertex shaders control how the matrices are used
- This makes the implementation more complex, but also more flexible

Hierarchical Modeling

- Since the stick man is made up of cylinders it would be a good idea to have **a procedure that produces cylinders**
- This procedure is parameterized by the **radius** of the cylinder, the **height** of the cylinder and the number of sides
- Since the cylinder is made up of polygons, the third parameter is used to specify the **number of polygonal sides** that we will use to approximate the cylinder

Hierarchical Modeling



Hierarchical Modeling

- The result produced by this function is a struct called Master
- This contains all the information required to draw the cylinder:
 - The vertex array object
 - The number of indices for glDrawElements
 - The array element buffer identifier, which is unfortunately not part of the vertex array object

Hierarchical Modeling

```
struct Master {  
    GLuint vao;  
    int indices;  
    GLuint ibuffer;  
};
```

```
Master *body;  
Master *head;  
Master *leg;  
Master *arm;
```

Hierarchical Modeling

- We can view the cylinder as having two sets of vertices, one for the top and one for the bottom
- The x and y coordinates for these vertices will be the same, only the z values will be different
- These vertices lie on a circle, so we can use sin and cos to compute their x and y coordinates
- The code for this is shown on the next slide

Hierarchical Modeling

```
x = new double[sides];
y = new double[sides];
dangle = 6.28/sides;
angle = 0.0;
for(i=0; i<sides; i++) {
    x[i] = radius*cos(angle);
    y[i] = radius*sin(angle);
    angle += dangle;
}
```

Hierarchical Modeling

- The top and bottom of the cylinder are composed of triangles
- A vertex at the center is part of all of these triangles
- So if we have n sides we will have $2(n+1)$ vertices, each of which has 3 coordinates, so we will need $3*2*(n+1)$ floating point values for the vertices

Hierarchical Modeling

- The following slide shows the code for creating the bottom vertices
- The code for the top vertices is essentially the same except we use height for the z coordinate
- We can use the same vertices for the sides of the cylinder

Hierarchical Modeling

```
vertices = new GLfloat[3*2*(sides+1)];  
j = 0;  
  
/* bottom */  
vertices[j++] = 0.0;  
vertices[j++] = 0.0;  
vertices[j++] = 0.0;  
for(i=0; i<sides; i++) {  
    vertices[j++] = x[i];  
    vertices[j++] = y[i];  
    vertices[j++] = 0.0;  
}  
}
```

Hierarchical Modeling

- The code for the indices for the top and bottom are shown on the next slide
- We need to fix the last vertex of the last triangle so it points to the first vertex
- Note: there are `sides` triangles on the top, `sides` triangles on the bottom, and `2*sides` triangles for the sides of the cylinder, and 3 vertices per triangle

Hierarchical Modeling

```
indices = new GLushort[3*4*sides];
j=0;

/* bottom */
for(i=0; i<sides; i++) {
    indices[j++] = 0;
    indices[j++] = i+1;
    indices[j++] = i+2;
}
indices[j-1] = 1;

/* top */
base = sides+1;
for(i=0; i<sides; i++) {
    indices[j++] = base;
    indices[j++] = base+i+1;
    indices[j++] = base+i+2;
}
indices[j-1] = base+1;
```

Hierarchical Modeling

- For each segment of the side we use two triangles
- This produces the rectangle that each side is made of

Hierarchical Modeling

```
for(i=1; i<sides; i++) {  
    indices[j++] = i;  
    indices[j++] = base+i;  
    indices[j++] = i+1;  
    indices[j++] = base+i;  
    indices[j++] = base+i+1;  
    indices[j++] = i+1;  
}  
indices[j++] = sides;  
indices[j++] = base+sides;  
indices[j++] = 1;  
indices[j++] = base+sides;  
indices[j++] = base+1;  
indices[j++] = 1;
```

Hierarchical Modeling

- Finally the code on the next slide constructs the buffers that we need for the cylinder
- The complete code for the example is on Canvas where you can see all the details
- With the cylinder procedure our init() procedure is now much shorter, see the following slide

Hierarchical Modeling

```
glGenBuffers(1, &vbuffer);
 glBindBuffer(GL_ARRAY_BUFFER, vbuffer);
 glBufferData(GL_ARRAY_BUFFER, 3*2*(sides+1)*sizeof(GLfloat), vertices, GL_STATIC_DRAW);

 glGenBuffers(1, &ibuffer);
 result->ibuffer = ibuffer;
 glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibuffer);
 glBufferData(GL_ELEMENT_ARRAY_BUFFER, 3*4*sides*sizeof(GLushort), indices, GL_STATIC_DRAW);

 vPosition = glGetUniformLocation(program,"vPosition");
 glVertexAttribPointer(vPosition, 3, GL_FLOAT, GL_FALSE, 0, 0);
 glEnableVertexAttribArray(vPosition);
```

Hierarchical Modeling

```
void init() {  
    int vs;  
    int fs;  
  
    vs = buildShader(GL_VERTEX_SHADER, "example5.vs");  
    fs = buildShader(GL_FRAGMENT_SHADER, "example5.fs");  
    program = buildProgram(vs,fs,0);  
    dumpProgram(program,"example 5");  
  
    glUseProgram(program);  
  
    body = cylinder(0.2,1.5,10);  
    leg = cylinder(0.2,0.9,10);  
    arm = cylinder(0.15,0.55,10);  
    head = cylinder(0.5,0.4,8);  
}
```

Enabling Code Reuse

- Note that we have parameterized this procedure so it can be used to generate a wide range of cylinders
- Whenever we write a procedure that produces geometry it's a good idea to consider how it can be parameterized
- This often reduces the number of procedures in a program and makes it easier to reuse the code that we write

Displaying the Model

- The display procedure draws the stick man
- We already have masters for all the objects we need to draw, all we need to do is set up the transformations
- We will need a stack of matrices, which we can get from the standard template library:

```
stack<glm::mat4> matrixStack;
```

Displaying the Model

- Since we have a different model matrix for each cylinder, we have to split the model matrix from the view and perspective matrices
- The vertex shader will now be responsible for combining these matrices
- The body is the root of our modeling hierarchy, the code for drawing it is on the next slide

Displaying the Model

```
/* draw body */  
glBindVertexArray(body->vao);  
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, body->ibuffer);  
glUniformMatrix4fv(modelLoc, 1, 0, glm::value_ptr(model));  
glDrawElements(GL_TRIANGLES, body->indices, GL_UNSIGNED_SHORT, NULL);
```

Displaying the Model

- In this case most of what we need comes directly from the Master struct for the body
- The other body parts are more complicated, since each has its own transformation matrix
- We need to push the current model matrix onto the stack, set up the new model matrix
- Then draw the cylinder and restore the model matrix

Displaying the Model

```
/* draw the right leg */  
matrixStack.push(model);  
model = glm::rotate(model, -2.3f, glm::vec3(0.0, 1.0, 0.0));  
glUniformMatrix4fv(modelLoc, 1, 0, glm::value_ptr(model));  
glBindVertexArray(leg->vao);  
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, leg->ibuffer);  
glDrawElements(GL_TRIANGLES, leg->indices, GL_UNSIGNED_SHORT, NULL);  
model = matrixStack.top();  
matrixStack.pop();
```



Note the use of radians

Displaying the Model

- The other body parts have basically the same code
- The complete code for this procedure is Canvas
- It is quite long, but straight forward
- The vertex shader for this example is shown on the next slide

Vertex Shader

```
/*
 * Simple vertex shader for example five
 */

in vec4 vPosition;
uniform mat4 model;
uniform mat4 viewPerspective;

void main() {

    gl_Position = viewPerspective * model * vPosition;

}
```

Fragment Shader

- The vertex shader combines the two matrices and then transforms the vertex
- The fragment shader uses a uniform for the colour of the cylinder
- So we can use the same shader for all the cylinders and just change the colour through the uniform

Fragment Shader

```
/*
 * Simple fragment shader for example five
 */

uniform vec4 colour;

void main() {
    gl_FragColor = colour;
}
```

Note

- The code that we are using to draw the cylinders is very similar (it was produced using cut and paste)
- This suggests that it should be placed in a procedure
- We could produce a drawInstance procedure that draws each of the instances
- The instances could be created in the init() procedure
- This will make our program shorter and a bit easier to follow

Animation in Hierarchical Models

- There is one last thing that we can do – make the man move
- The walking motion is produced by using a rotation about the x axis
- When one leg is rotating forward the other leg is rotating backwards
- This can be done by having opposite signs on the rotation angles

Hierarchical Modeling

- We need to generate the `walk` angle that produces the walking motion
- This angle varies from -0.7 to +0.7 radians and is updated each time the stickman is drawn
- This can easily be done in the display loop at the end of the `main()` procedure, we only need to keep track of the direction we are moving, which can be done using a static local variable

Hierarchical Modeling

- We can add other motions to the model in similar ways, such as an arm wave ([try it!](#))
- Hierarchical models make this type of animation easy, all we need to do is modify [the transformation between the nodes in the hierarchy](#)
- If we added a translation to the body we could have the stickman move at the same time as his legs are walking

Scene Graphs

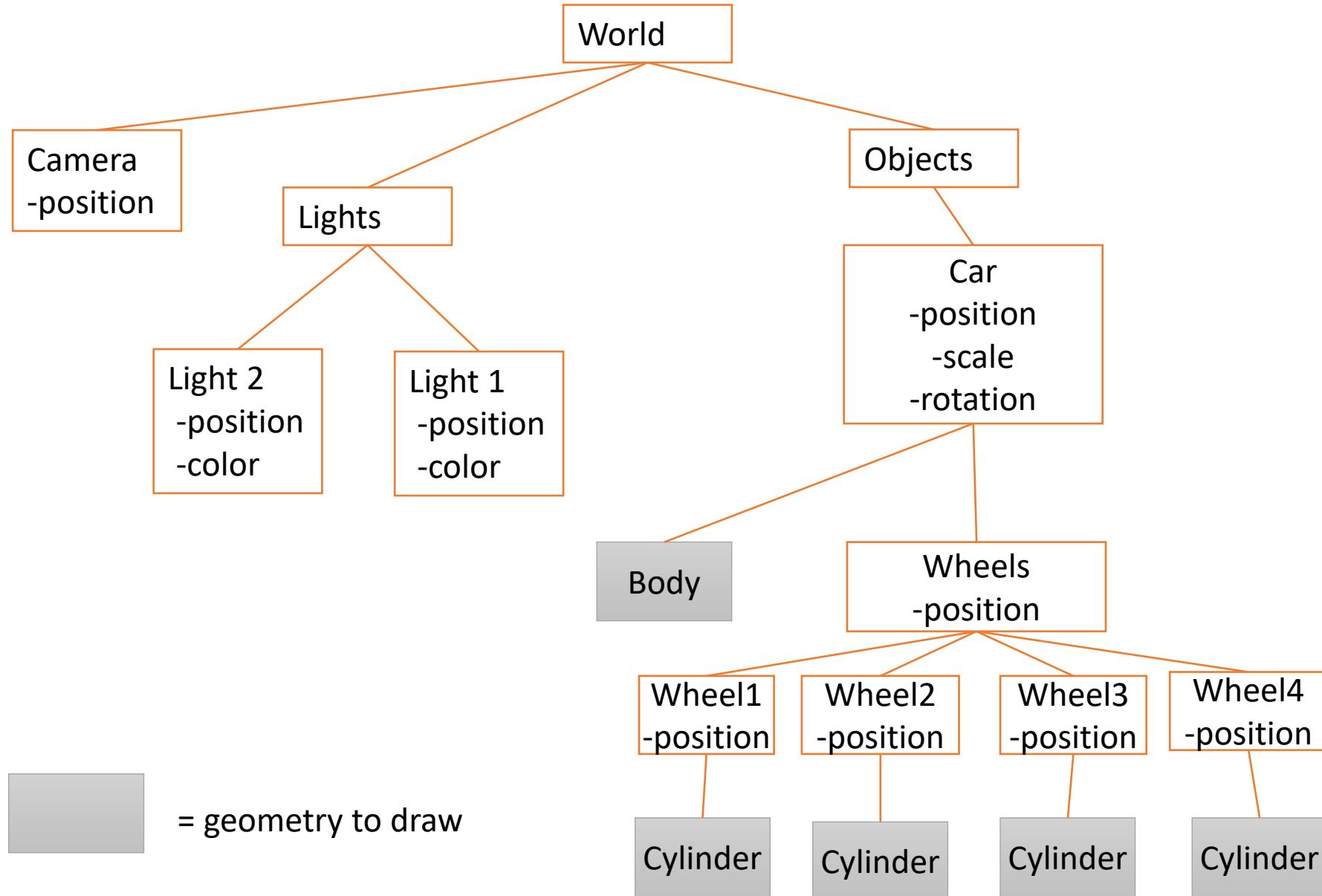
- The notion of hierarchical modeling can be taken one step further
- We can formalize the notions of nodes and edges and represent them with individual classes
- We can then construct a graph structure from instances of these classes
- This gives us what's called [a scene graph](#)

Scene Graph

- A scene graph system has classes for different types of graphical information, such as polygonal meshes, transformations, materials, lights, etc
- It also has ways of connecting up instances to form a complete graph that describes the object
- There are quite a few scene graph systems, but none of them are standards in the same way that OpenGL is
- The closest to a standard is Open Scene Graph
<http://www.openscenegraph.org/>

Scene Graph

- We will create scene graphs with two types of nodes:
 - Abstract (non-drawing) nodes: used to specify lights, view, and to organize objects into hierarchical models. These nodes may specify transforms, material properties, colours which will apply to all nodes below them
 - Geometry (drawn) nodes: these are the shapes which will be rendered. These are always LEAF nodes



Summary

- In today's lecture you learned:
 - Hierarchical Models
 - How hierarchical models can be implemented in OpenGL
 - Scene Graphs

Next Classes

- Introduction to curves: Implicit and Parametric Geometry

CSCI 3090

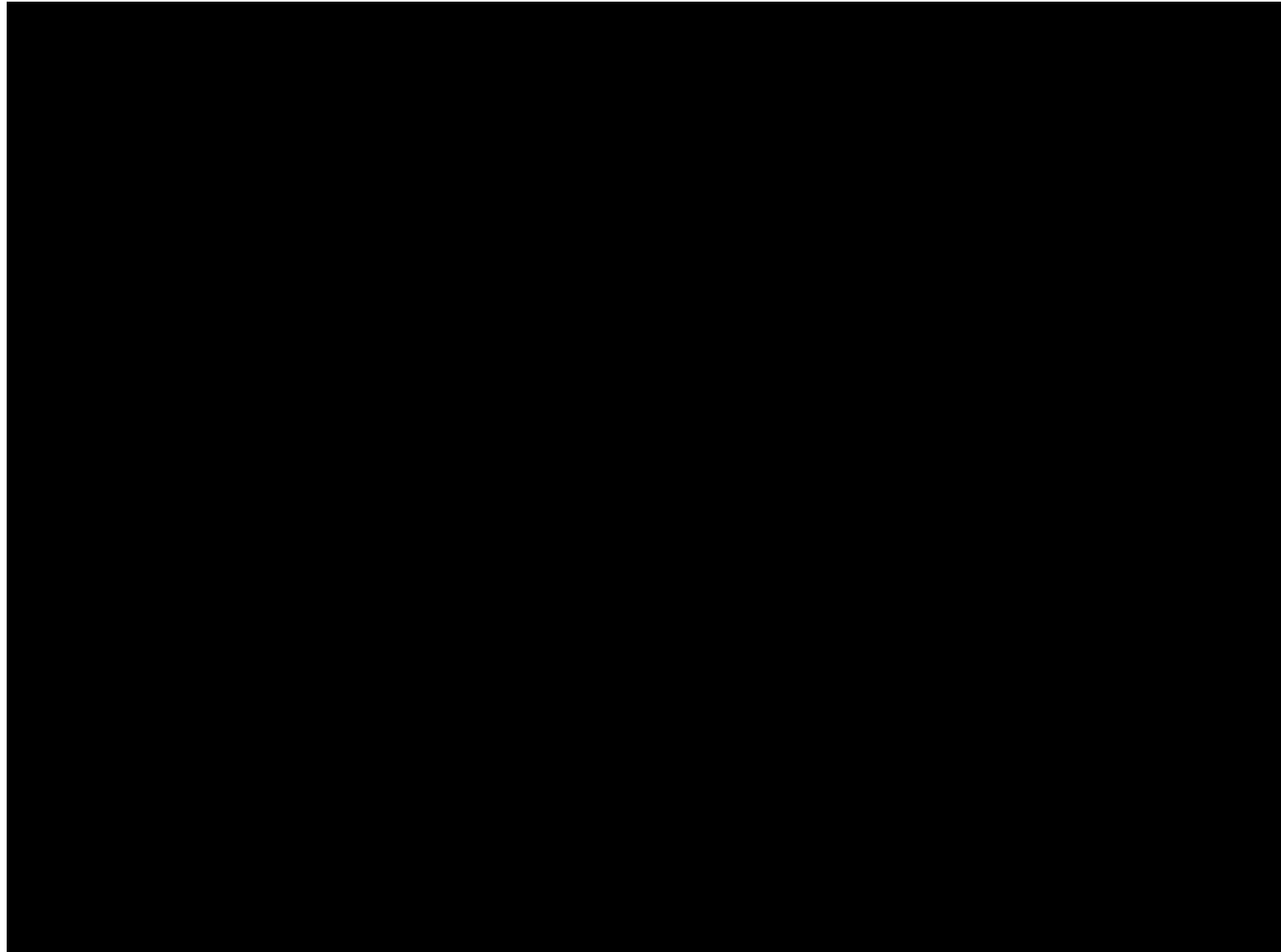
Implicit and Parametric

Representations

Mark Green

Faculty of Science

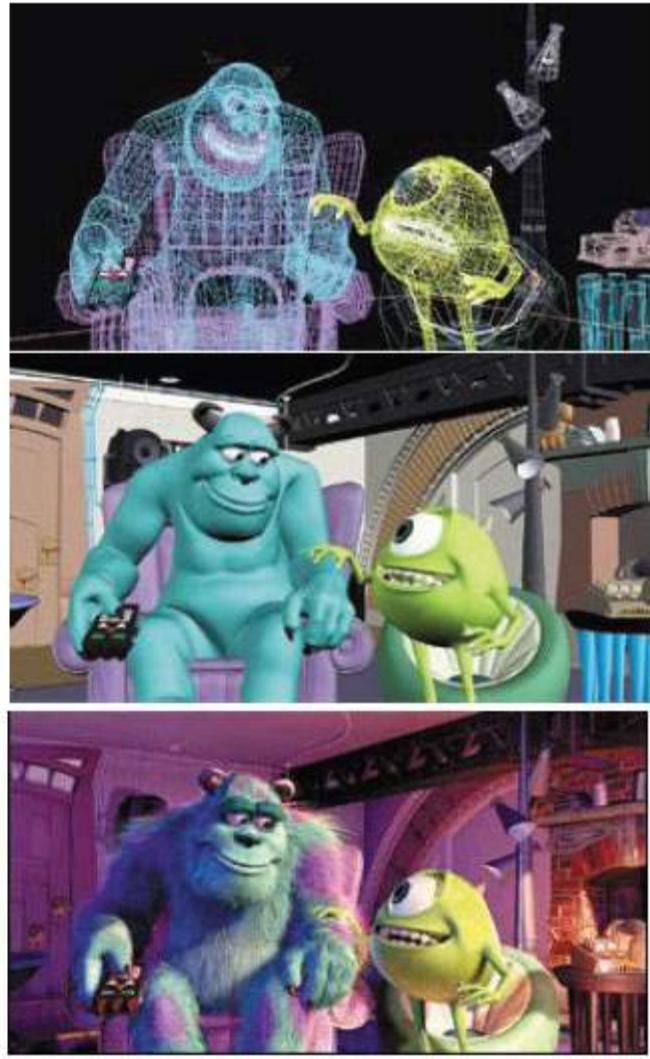
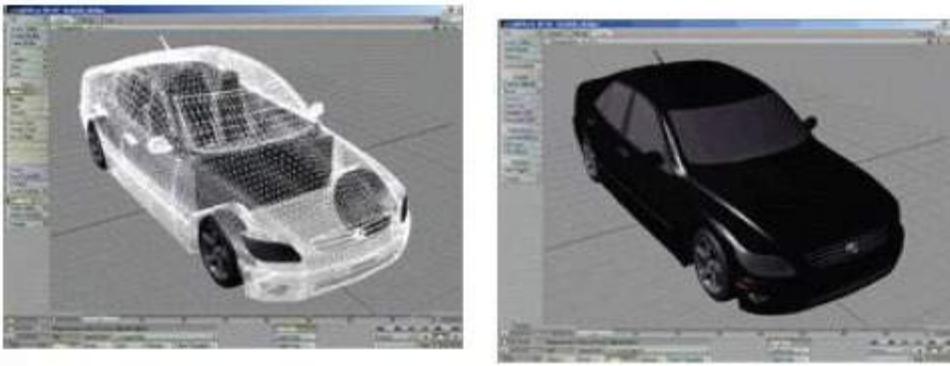
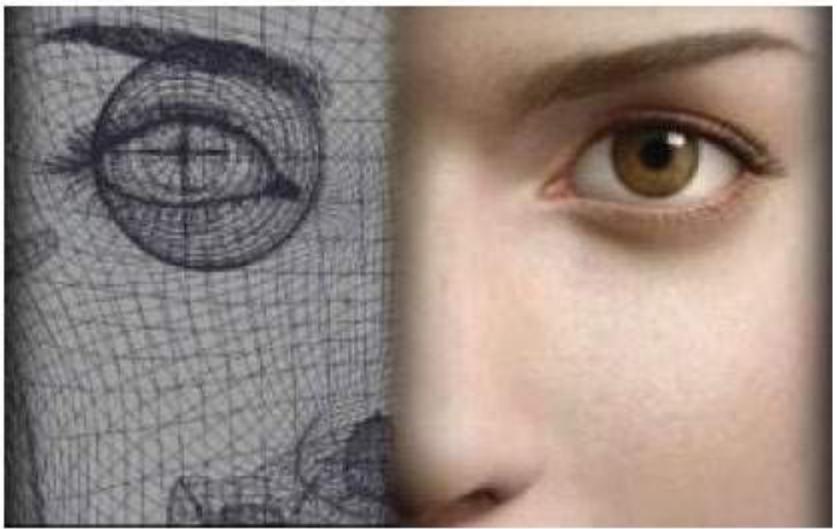
Ontario Tech



Goals

- By the end of today's class, you will:
 - Understand why curved modeling primitives are required
 - Describe the difference between implicit and parametric representations of curves
 - Understand piecewise representations and continuity

Implicit Representations



Vertex Specification Woes...

- Polygons are relatively low level, they are quite general, but we need a lot of them to represent objects with interesting shapes
- In particular curved objects are hard to represent with polygons, they are basically flat, so they are a poor approximation to curves
- We will look at two representations that are higher level and are good at representing curved objects

Implicit Representations

- Implicit representations are based on giving an equation for the object
- In 2D this equation has the form

$$f(x, y) = 0 \text{ or } f(\bar{p}) = 0$$

- The function $f(x, y)$ is the representation, in this case it's the representation of a curve
- All the points that satisfy this equation are on the curve

Implicit Line

- Direction of a line is a vector: $\vec{d} = \overline{p_1} - \overline{p_0}$
- Any vector from p_0 to any point on the line must be parallel to \vec{d}
- So, any point on the line must have a perpendicular vector $\vec{d}^\perp = (d_y, -d_x) \equiv \vec{n}$
- Check: $\vec{d} \cdot \vec{d}^\perp = 0$

Implicit Line

- Given: $p_1=[x_1, y_1]; p_2=[x_2, y_2]$
- What is \vec{n} ?

Implicit Line (Normal Form)

- Given: $p_0 = [x_0, y_0]$; $p_1 = [x_1, y_1]$
- $\vec{n} = [y_1 - y_0, x_0 - x_1]$
- $(\vec{n} \cdot (q - p_0)) = 0$ is satisfied for all points q on the line

Implicit Circle

- For example, the implicit representation of a circle is:

$$(x - x_c)^2 + (y - y_c)^2 - r^2 = 0$$

- Or, in vector notation

$$\|\bar{p} - \bar{p}_c\|^2 = r^2$$

- Here (x_c, y_c) is the center of the circle and r is its radius
- This is a very compact representation, but we can't easily compute the points on the circle from this representation

Implicit Representations

- This is the main problem with implicit representations, they tend to be very **difficult to display**
- In the scientific visualization part of the course we will deal more with implicit representations, and some of their display algorithms
- For now we will just concentrate on some of their basic properties

Implicit Representations

- We can define curves and surfaces in 3D space in a similar way, that is:

$$f(x, y, z) = 0$$

- The set of points that satisfy this equation either lie on the curve or surface
- In 3D we are usually interested in surfaces, so we will concentrate on them
- The implicit representation of a sphere is:

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

Implicit Representations

- Again this representation is compact, and captures the curvature of the sphere, but gives us no way of drawing it
- In order to display these objects we need to know their normal vectors
- For implicit surfaces this is relatively easy to compute, it is just the gradient:

$$\nabla f = \begin{pmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} & \frac{\partial f}{\partial z} \end{pmatrix}$$

Implicit Representations

- Given $f(x,y,z)$ in the form of an equation, the gradient or normal vector is easy to compute
- It also gives us a hint for drawing these objects
- Relatively close to the surface we can follow the gradient to the intersection point
- Once we have found one point on the surface we can use the gradient and tangents to find more points and work our way around the surface
- It is still not very easy...

Parametric Representations

Parametric Representations

- While the implicit representation was compact and could easily handle curved surfaces, it was not easy to draw
- Parametric representations are also quite compact and good at representing curved surfaces, but they are **much easier to draw**
- They have been used in computer graphics for many years

Basics of Parametric Form

- We will start in 2D since it is simpler
- The basic idea is to have a **parameter** that varies along the curve
- We then use a function of this parameter to produce the points along the curve:

$$(x, y) = \bar{p} = f(t)$$

- As we vary the value of t over its range we generate all of the points on the curve

Parameterization

- A parametric representation for a circle is

$$\begin{aligned}x &= r\cos(t) \\y &= r\sin(t)\end{aligned}$$

For t varying between 0 and 2π

- Note that the parameterization is not unique:

$$\begin{aligned}x &= t - 1 \\y &= 2t - t^2\end{aligned}$$

For t varying between 0 and 2 also produces a circle!



Arc Length Parameterization

- The other common parameterization is by arc length, s , this is **the distance along the curve**
- For an arc length parameterization we have:

$$\left| \frac{df(s)}{ds} \right|^2 = c$$

- Where c is a constant
- This is useful for some mathematical operations but not as common as the unit parameterization

Unit Parameterization

- With this representation drawing the circle is easy, just use enough values of t to get a dense enough set of points
- We can choose any parameterization we like, preferably one that is convenient
- There are two standard parameterizations, the most common one is to use the interval 0 to 1 – **the unit parameterization**
- The curve starts at $u=0$ and ends at $u=1$

Unit Parameterization of a Circle

- A unit parametric representation for a circle is

$$\begin{aligned}x &= r\cos(2\pi u) \\y &= r\sin(2\pi u)\end{aligned}$$

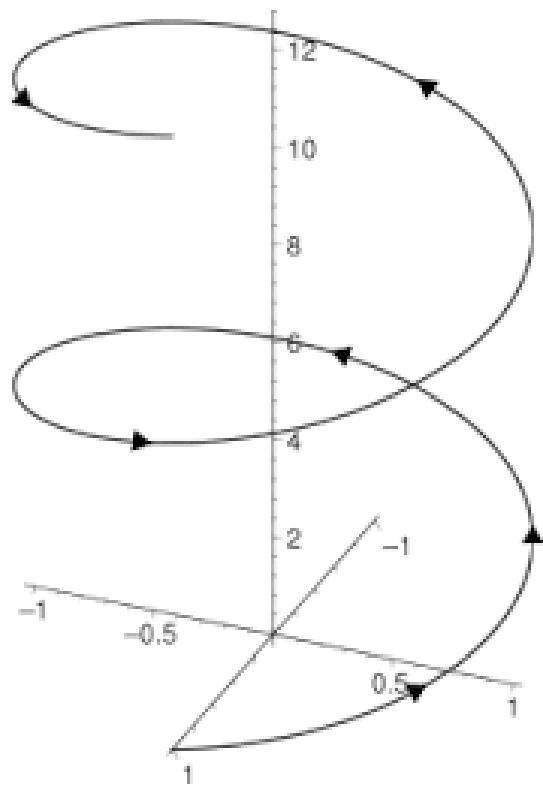
For u varying between 0 and 1

Parametric Representations

- We can generalize the curves to 3D to get:
$$(x, y, z) = p = f(t)$$
- We can also use this technique to represent **surfaces**, in this case the representation is:
$$(x, y, z) = p = f(u,v)$$
- In this case we use **two parameters** to trace out the surface, as we vary these parameters we get all of the points on the surface

3D Parametric Curves

- $f(t) = (\cos(t), \sin(t), t), 0 \leq t \leq n2\pi$



Surface Dimensionality

- For any n dimensional solid, its surface will be n-1 dimensional
- Consider the surface of the earth, we can specify any point using its longitude and latitude
- Similarly we can use two parameters for any 3D surface
- We will mainly look at curves, but the techniques generalize to surfaces

Piecewise Curves

- Unfortunately, using one function isn't as flexible as we would like it to be
- We can get some simple shapes that way, but for complex shapes it soon becomes quite difficult
- We solve this problem using a **piecewise representation**
- We basically divide the curve up into pieces and then use a separate representation for each piece

Piecewise Curves

- This has the disadvantage of requiring more than one function, but it has two main advantages:
 - We can use simpler functions, easier to program and work with
 - More general, we can represent any curve, just need to use enough pieces

Continuity

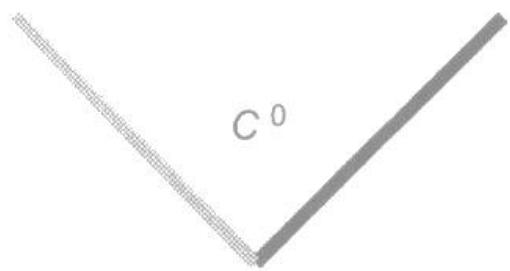
Parametric Representations

Continuity

- When we do a piecewise representation we need to consider **how the pieces fit together: the continuity**
- a curve s is said to be C^n -continuous if its n^{th} derivative $d^n s/dt^n$ is continuous → **parametric continuity**: shape & speed
- not only for individual curves, but also in particular for where segments connect

C⁰ Continuity

- We say that a curve is C⁰ continuous if all the pieces match up at their boundaries

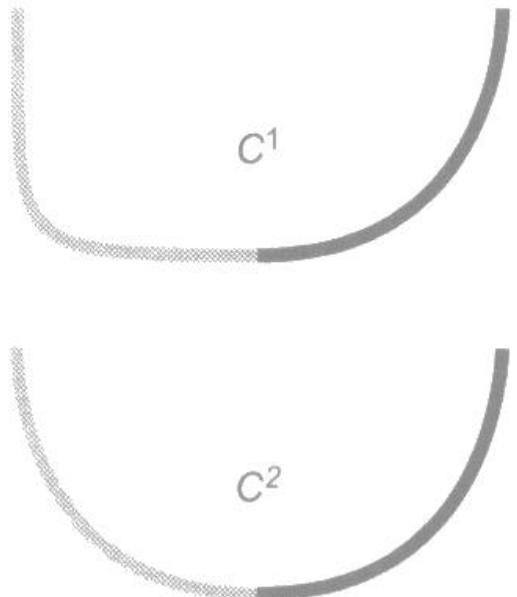


Consider two curve pieces $f_1(t)$ and $f_2(t)$, they are C⁰ continuous if:

$$f_1(1) = f_2(0)$$

If a curve is C⁰ there are no gaps in the curve, but it may not be smooth

C1 & C2 Continuity



- A curve is C^1 continuous if the first derivatives are equal across the joins
- In general a curve is C^n continuous if all of the first n derivatives are equal across all of the joins
- We will usually be interested in C^1 and C^2 curves

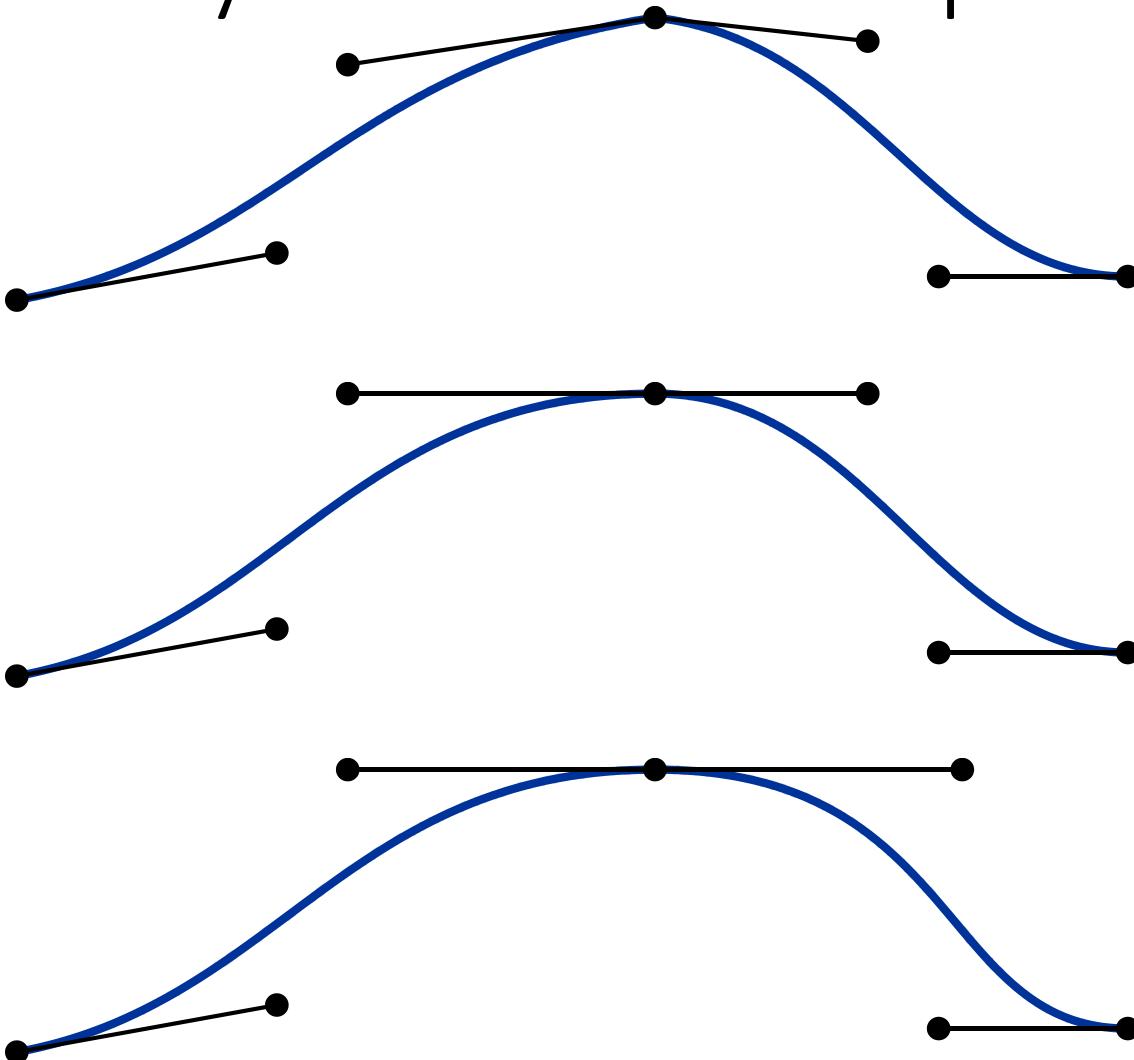
Parametric Representations

- The parameterization can cause problems for continuity, particularly with the derivatives
- The derivatives might point in the same direction, but their magnitude might be different due to the parameterizations
- These leads to the idea of geometric continuity, which relies more on the geometry of the curve

Geometric Continuity

- **geometric continuity:** two curves are G^n -continuous if they have proportional n^{th} derivatives (same direction, speed can differ)
- G^n follows from C^n , but not the other way
- Car bodies need at least G^2 -continuity for aerodynamics

Continuity Criteria: Examples



$G^0 = C^0$

G^1

C^1

Parametric Representations

- Two curves are G^1 continuous if they satisfy
$$f_1'(1) = kf_2'(0)$$
- Where k is a scalar constant, there are similar definitions for G^n
- If a curve is C^n continuous it will also be G^n continuous, but not the other way around

Summary

- In today's lecture you learned:
 - Implicit and Parametric Representations
- What are implicit representations best for?
- What are parametric representations best for?

CSCI 3090

Parametric Curves

Mark Green
Faculty of Science
Ontario Tech

Goals

- By the end of today's class, you will be equipped to:
 - Describe the difference between local and global control
 - Determine whether a curve interpolates a given point
 - Describe how blending functions combine to make curves

Parametric Representations

Parametric Representations

- For parametric representations we need some function, $f(t)$, that gives the points along the curve
- The question is what function should we use?
- We could use trigonometric functions, but they are hard to deal with
- Instead we will use polynomials, they are general and easy to work with

Canonical Form

- In general a polynomial has the form:

$$f(t) = \sum_{i=0}^n a_i t^i$$

Canonical Representation

- In our case the a_i are vectors since $f(t)$ is a vector -> remember $f(t)$ specifies (x,y) !
- This representation is good for computing points on the curve
- But, it's not particularly easy to find the a_i for a desired curve

$$f(u) = \sum_{i=0}^n a_i u^i$$

Blended Form

- There is a second form that is quite useful:

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

- Here the c_i are still vectors, and the $b_i(t)$ are polynomials called basis or blending functions
- We can choose the $b_i(t)$ in any way that's convenient to us, in particular to make it easy to fit the curve

Canonical Form -> Blended Form

$$f(u) = \sum_{i=0}^n a_i u^i$$

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

For the canonical case,
each $b_i(u) = u^i$

Canonical Form as a Vector Expression

$$f(u) = \sum_{i=0}^n a_i u^i$$

- Canonical form as a vector expression has

$$\begin{aligned}\mathbf{a} &= [a_0 \ a_1 \ a_2 \ \dots \ a_n] \\ \mathbf{u} &= [1 \ u \ u^2 \ u^3 \ u^4 \ \dots \ u^n]\end{aligned}$$

- So $f(u) = \mathbf{u} \cdot \mathbf{a}$

The Linear Case

- The simplest form of polynomial is the linear one:

$$f(u) = a_0 + ua_1$$

- If p_0 and p_1 are the end points, then we can also write the line as:

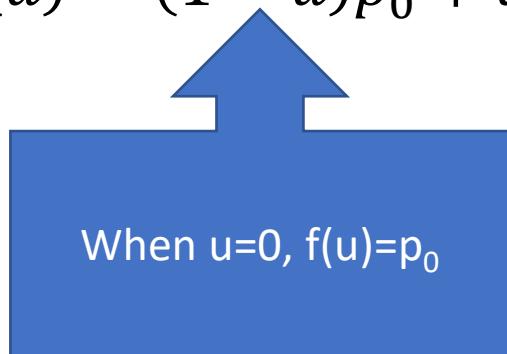
$$f(u) = (1 - u)p_0 + up_1$$

- The **p** vector contains our **control points**

The Linear Case

- If p_0 and p_1 are the end points, then we can also write the line as:

$$f(u) = (1 - u)p_0 + up_1$$



- The \mathbf{p} vector contains our **control points**

The Linear Case

- If p_0 and p_1 are the end points, then we can also write the line as:

$$f(u) = (1 - u)p_0 + up_1$$



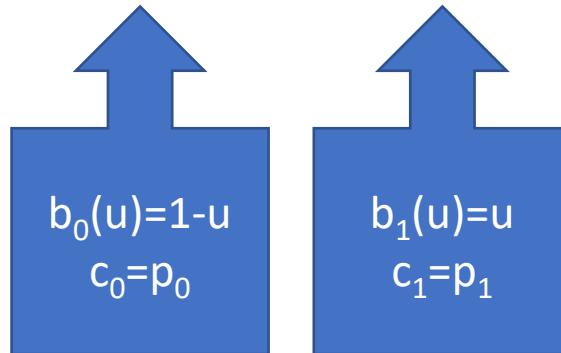
When $u=1$, $f(u)=p_1$

- The \mathbf{p} vector contains our **control points**

The Linear Case

- If p_0 and p_1 are the end points, then we can also write the line as:

$$f(u) = (1 - u)p_0 + up_1$$



Recall blended form:

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

Control Points

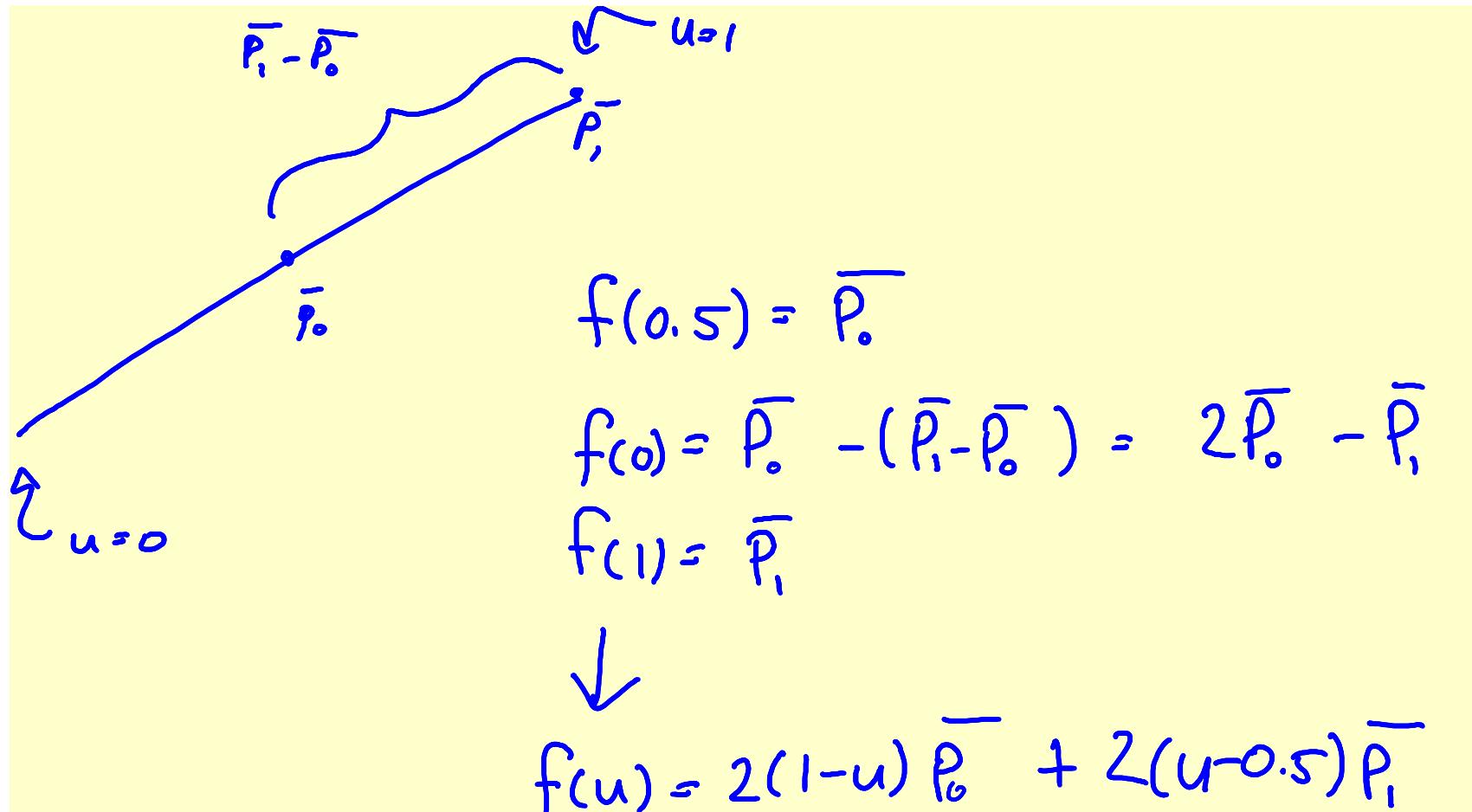
- In this representation we call p_0 and p_1 control points, since they *control* the shape of the curve

Control Points

- Note that we could choose many different kinds of control points for a line:
 - The mid point and one end point
 - One end point and the vector to the other end point
 - The mid point, slope and distance to one of the end points
- For each choice of control points we will have different blending functions, but the same canonical representation
- Given control points, we need to *know* what they represent in order to use equations properly

Control Points

- Consider a line defined by two control points:
 - The mid point and one end point
- What is the expression of the line, using this set of control points?



The Linear Case

- Let's return to p_0 and p_1 being the end points:

$$f(u) = (1 - u)p_0 + up_1$$

Solving for Coefficients (a)

- Now that we have two representations of the curve we can just equate them to get a_i :

- $f(u) = p_0 + (p_1 - p_0)u = a_0 + ua_1$
- $f(0) = p_0 = a_0$
- $f(1) = p_0 + p_1 - p_0 = p_1 = a_0 + a_1$

Thus: $a_0 = p_0$, $a_1 = p_1 - p_0$

- In general the relationship is not as easy to determine, so we need a general technique: matrix form

Constraint Matrix

- $a_0 = p_0, p_1 = a_0 + a_1$
- We use the known values to set up a set of equations, substituting values for u :
 - $p_0 = f(0) = a_0 + 0a_1$
 - $p_1 = f(1) = a_0 + 1a_1$
- We can put this into matrix form:

$$\begin{bmatrix} p_0 \\ p_1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \end{bmatrix}$$
$$p = Ca$$

- C is called the constraint matrix

Blending Matrix

- We **invert** the constraint matrix to give $B = C^{-1}$, which is called the blending matrix
- From $p = Ca$ and $B = C^{-1}$ we have the following relationship:

$$a = Bp$$



Blended Form

- And the canonical representation becomes
 - $f(u) = \mathbf{u} \cdot \mathbf{a} = \mathbf{u}B\mathbf{p}$
 - $\mathbf{u} = [1 \; u \; u^2 \; u^3 \; \dots \; u^n]$
- We now have a completely general technique that is **independent of the polynomial degree**
- All we need is **B** and the **control points** and we can produce the canonical polynomial

Example: midpoint/endpoint control

- Find the basis matrix for a line segment parameterized on its halfway point ($u=0.5$) and end point ($u=1$)
- $p_0 = f(0.5) = u^0 a_0 + u^1 a_1 = 1a_0 + 0.5a_1$
- $p_1 = f(1) = u^0 a_0 + u^1 a_1 = 1a_0 + 1a_1$
- So $C = \begin{bmatrix} 1 & 0.5 \\ 1 & 1 \end{bmatrix}, B = \begin{bmatrix} 2 & -1 \\ -2 & 2 \end{bmatrix}$
- And $f(u) = \mathbf{u}B\mathbf{p}$, so we have a parameterization based on the control points as desired

Checking the expression...

$$\begin{matrix} u & B & P \\ \begin{bmatrix} 1 & u \end{bmatrix} & \begin{bmatrix} 2 & -1 \\ -2 & 2 \end{bmatrix} & \begin{bmatrix} P_0 \\ P_1 \end{bmatrix} \end{matrix}$$

$$\begin{matrix} uB \\ \begin{bmatrix} 2-2u & -1+2u \end{bmatrix} \end{matrix} \begin{bmatrix} P_0 \\ P_1 \end{bmatrix} =$$

$$(2-2u)P_0 + (2u-1)P_1 = 2((1-u)\bar{P}_0 + 2(u-\alpha_s)\bar{P}_1)$$

Same as we found
'manually'

Polynomial Degree

- While linear polynomials got us started, we really want something that's curved
- Most of the curves we use in graphics are **cubic** for the following reasons:
 - High enough degree to produce smooth curves
 - Low enough degree to be controllable
- The mathematics is basically the same for any degree polynomial, but it turns out that the cubic works out best most of the time

Parametric Representations

- Note that once we have evaluated \mathbf{uB} , we are just multiplying the control points by constants, that is we have a **linear combination of the control points**
- To see how this works for cubic curves we will look at the **Hermite curve**, in this case we **specify the position and derivatives** at the two end points, that is we have $f(0)$, $f'(0)$, $f(1)$ and $f'(1)$ and we want to find the polynomial coefficients

Finding the Blend Matrix: Hermite Curve

- Hermite Curve is defined through specifying the position and derivatives at the two end points:
 - $f(0)$
 - $f'(0)$
 - $f(1)$
 - $f'(1)$
- Q: how to find the polynomial coefficients for canonical form?

Parametric Representations

- Construct a system of equations:

$$p_0 = f(0) = a_0 + 0a_1 + 0^2 a_2 + 0^3 a_3$$

$$p_1 = f'(0) = 0a_0 + a_1 + 0^1 2a_2 + 0^2 3a_3$$

$$p_2 = f(1) = a_0 + 1a_1 + 1^2 a_2 + 1^3 a_3$$

$$p_3 = f'(1) = 0a_0 + a_1 + 1^1 2a_2 + 1^2 3a_3$$

- Which gives us the following (recall $p = Ca$):

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 2 & 3 \end{bmatrix} \quad B = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -3 & -2 & 3 & -1 \\ 2 & 1 & -2 & 1 \end{bmatrix}$$

Hermite Curve

- $$f(u) = \begin{bmatrix} 1 \\ u \\ u^2 \\ u^3 \end{bmatrix}^T \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -3 & -2 & 3 & -1 \\ 2 & 1 & -2 & 1 \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{bmatrix}$$
- $$f(u) = (1 - 3u^2 + 2u^3)p_0 + (u - 2u^2 + u^3)p_1 + (3u^2 - 2u^3)p_2 + (-u^2 + u^3)p_3$$

Putting the Pieces Together

- Now lets come back to the problem of putting curve segments together to produce a piecewise curve
- Each piece of the curve will have a range of parameter values, we will usually use 0 thru 1 as this range
- Similarly there will be a range of parameter values that cover the complete curve, typically this will range from 0 to n, where there are n pieces

Knots

- We can perform a simple translation from the global parameter range to the local parameter range of each piece
- In the global parameter space there are parameter values where one curve segment stops and the next curve segments starts
- These places are called **knots**, and for simple curves they have the values: 0, 1, 2, ... n
- These are **parameter values, not positions**

Maintaining Continuity Between Pieces

- If the pieces are independent how do we maintain this continuity?
 - Shared Point Scheme
 - Dependency Scheme
 - Explicit Equations

Continuity: Shared Point Scheme

- The control point at the end of one piece is the control point at the start of the next piece
- This assumes that the pieces are defined in terms of values at their end points
- In general this is too restrictive

Continuity: Dependency Scheme

- Copy the values at the end of one piece to the start of the next piece, called a dependency scheme
- This doesn't force us to have a control point at the end of each piece, we can just evaluate the curve at that point and pass this value on to the next piece

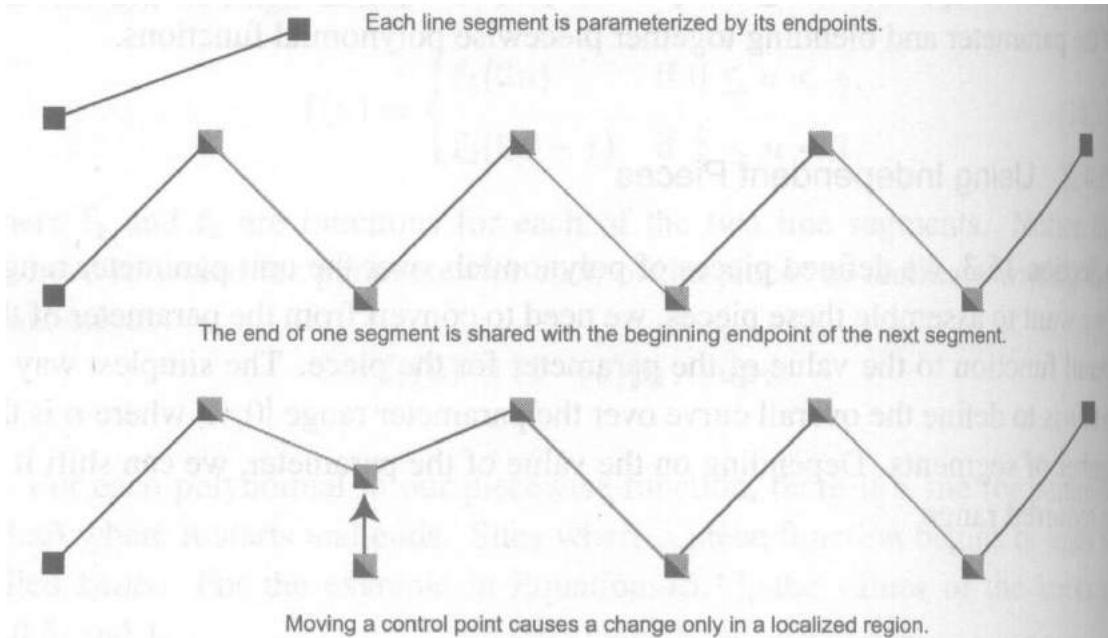
Continuity: Explicit Equation Scheme

- Use an explicit equation to connect the two pieces
- This could be a constraint on the control points, or on the curve value at particular points
- All three of these schemes introduce constraints or connections between the pieces, so a change in one piece can propagate along the curve

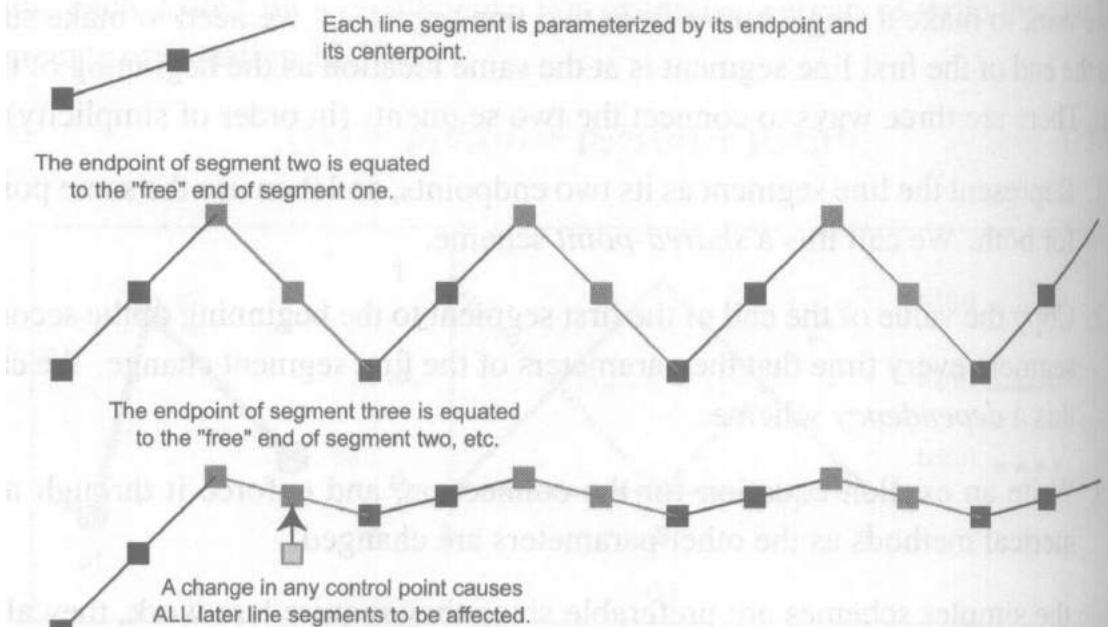
Local Control

- A curve has **local control** if a change in a control point only effects a small number of pieces
- That is, changing a control point only effects the local shape of the curve, it has no impact on the parts of the curve that are far away
- Locality is important in design, we don't want a small change to have a big impact on the curve
- We want to be able to work on one part of the curve at a time

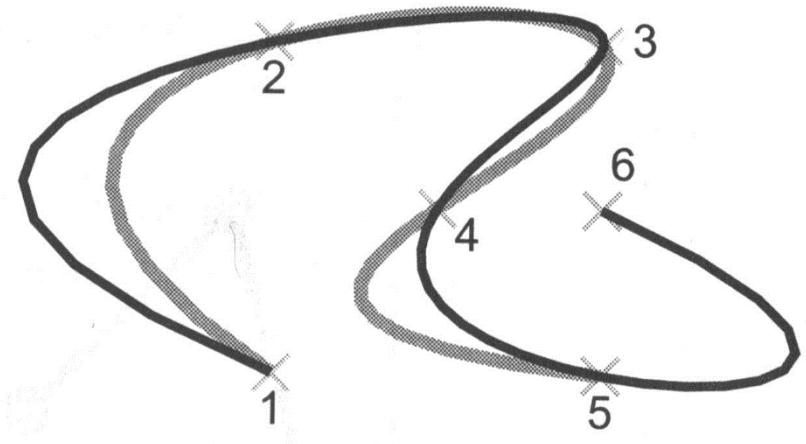
Shared control points: local control



Dependency scheme: global control



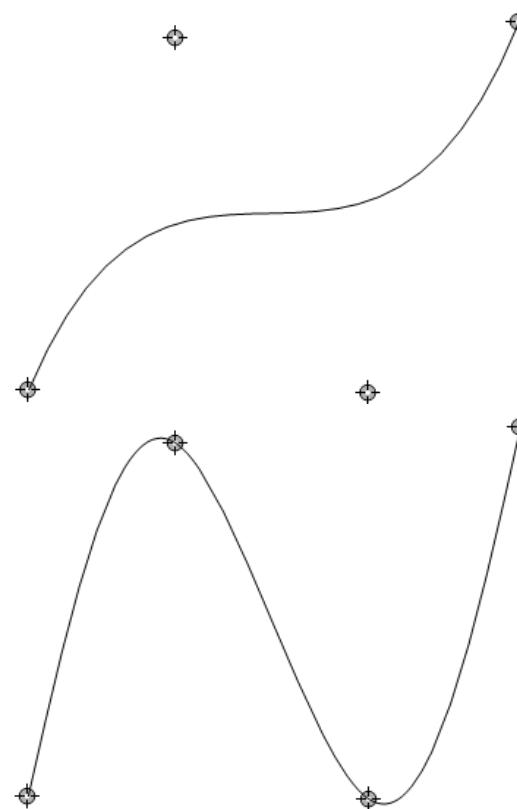
Local vs. Global Control



- Local control: move a control point, only attached segments affected
- Global control: move a control point, all curves affected
 - Shown: adding a 6th point moves curve

Interpolating vs. Approximating

- two different curves schemes: curves do not always go through all control points
 - **approximating curves**
not all control points are on the resulting curve
 - **interpolating curves**
all control points are on the resulting curve



Properties of Cubics

- We would like to have curves which satisfy:
 1. Each piece of the curve is cubic
 2. The curve interpolates the control points
 3. The curve has local control
 4. The curve has C^2 continuity
- However, no such curve is possible... we use various sorts of cubics which satisfy 3 of 4 conditions, depending on our needs

Summary

- In today's lecture you learned:
 - Parametric Representations
 - Blending functions
 - Control points

Next Class

- Advanced curves (study this lecture in between!)

CSCI 3090

Advanced Curves

Mark Green

Faculty of Science

Ontario Tech



Goals

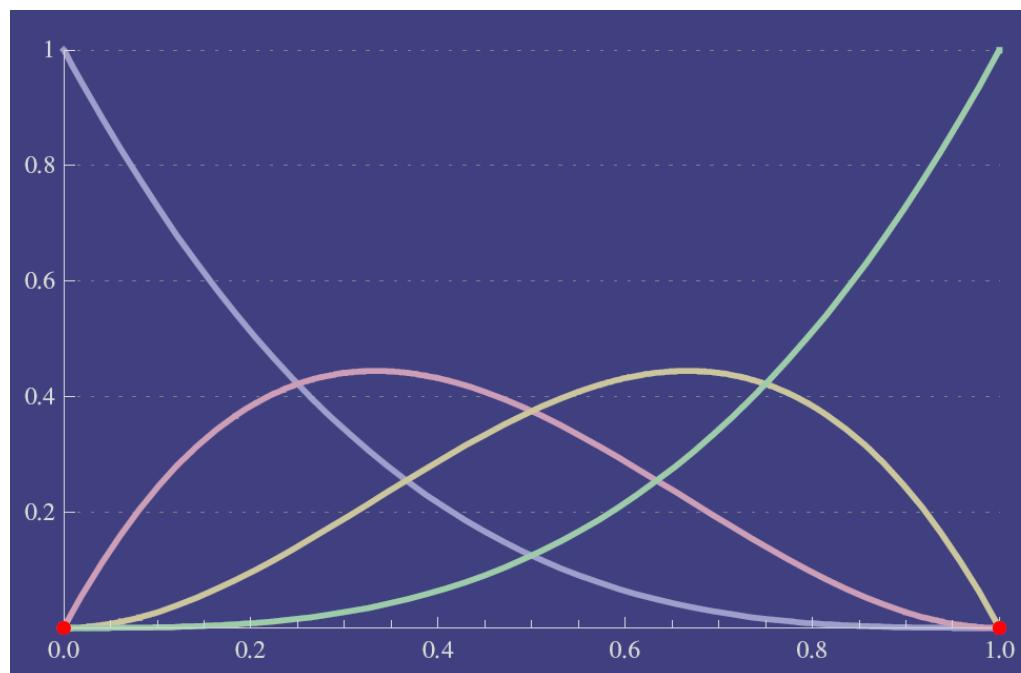
- By the end of today's class, you will:
 - Understand the important properties of Bezier curve
 - Understand subdivision surfaces and how they can be used to refine a 3D mesh

Bezier Curves

- The Bezier curve is probably the most important curve in computer graphics, demonstrated the power of parametric curves
- Developed by Pierre Bezier in the early 1970s
- Aim was to make car design easier, wanted car designers to be able to use CAD tools
- Problem: existing curves were too complicated and weren't intuitive to use, couldn't give them to car designers
- This was the motivation behind the development of the Bezier curve

Bézier Curves: Properties

- **approximating curve:** only first & last control points are interpolated
- No guarantee of continuity at joins, must be enforced by program



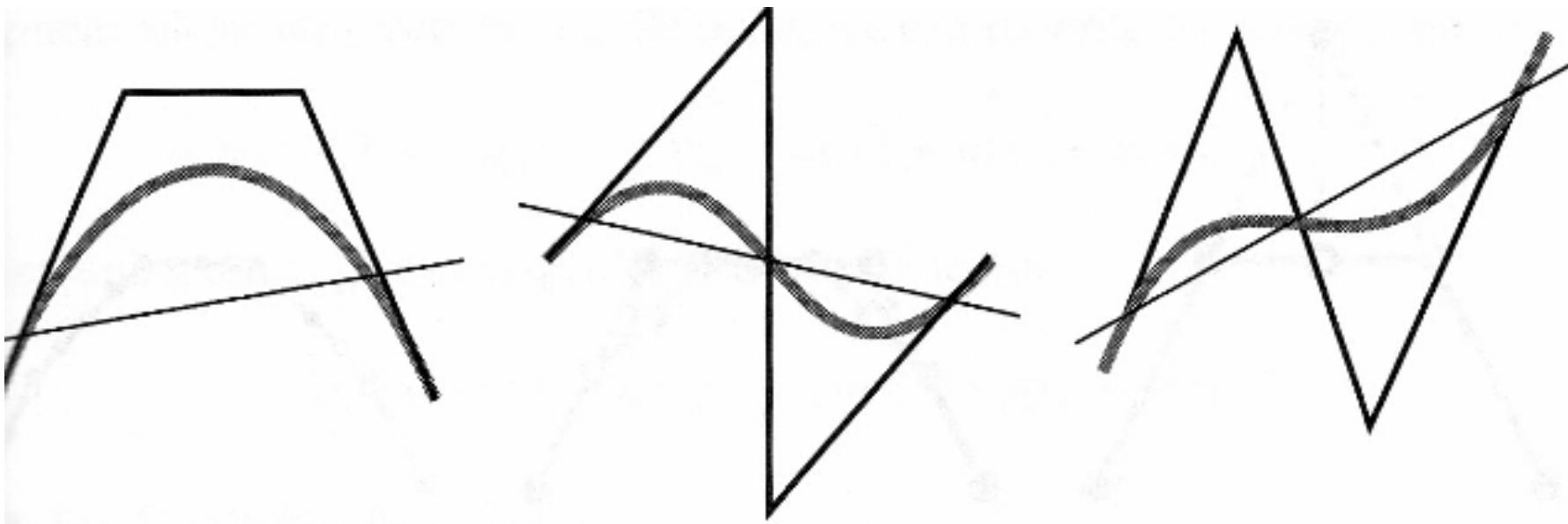
Bezier Curves

- For piecewise curves, it has **very good local control properties**, with changes effecting at most two pieces

Bézier Properties

- Bounded by convex hull:
 - The **convex hull** is the smallest polygon that contains the control points
 - Curve will be within this hull
- Variation diminishing property:
 - any line intersects the curve no more times than it intersects the lines connecting the control points
 - i.e., The curve wiggles no more than the lines connecting the control points

Variation Diminishing Property



the number of intersection points of any straight line with a Bézier curve is at most the number of intersection points of the same straight line with the control polygon of the curve.

Bézier Properties

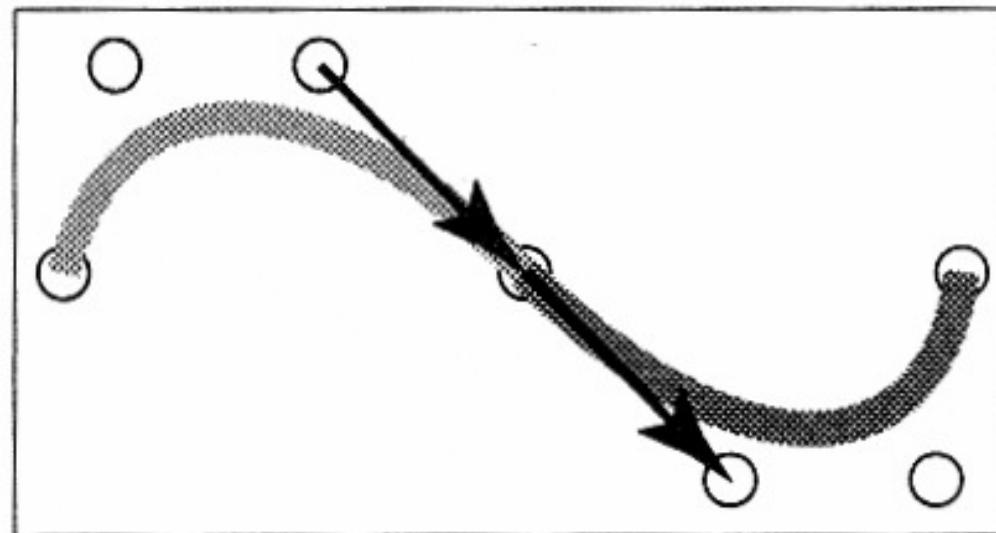
- Bézier curves are **affine invariant**: we can translate, rotate, scale and skew the control points and the curve will be transformed in the same way
- Instead of trying to transform the curve, we **just need to transform its control points**
- In addition **the curves are symmetric**, we can reverse the order of the control points and the curve has the same shape

Bézier Continuity

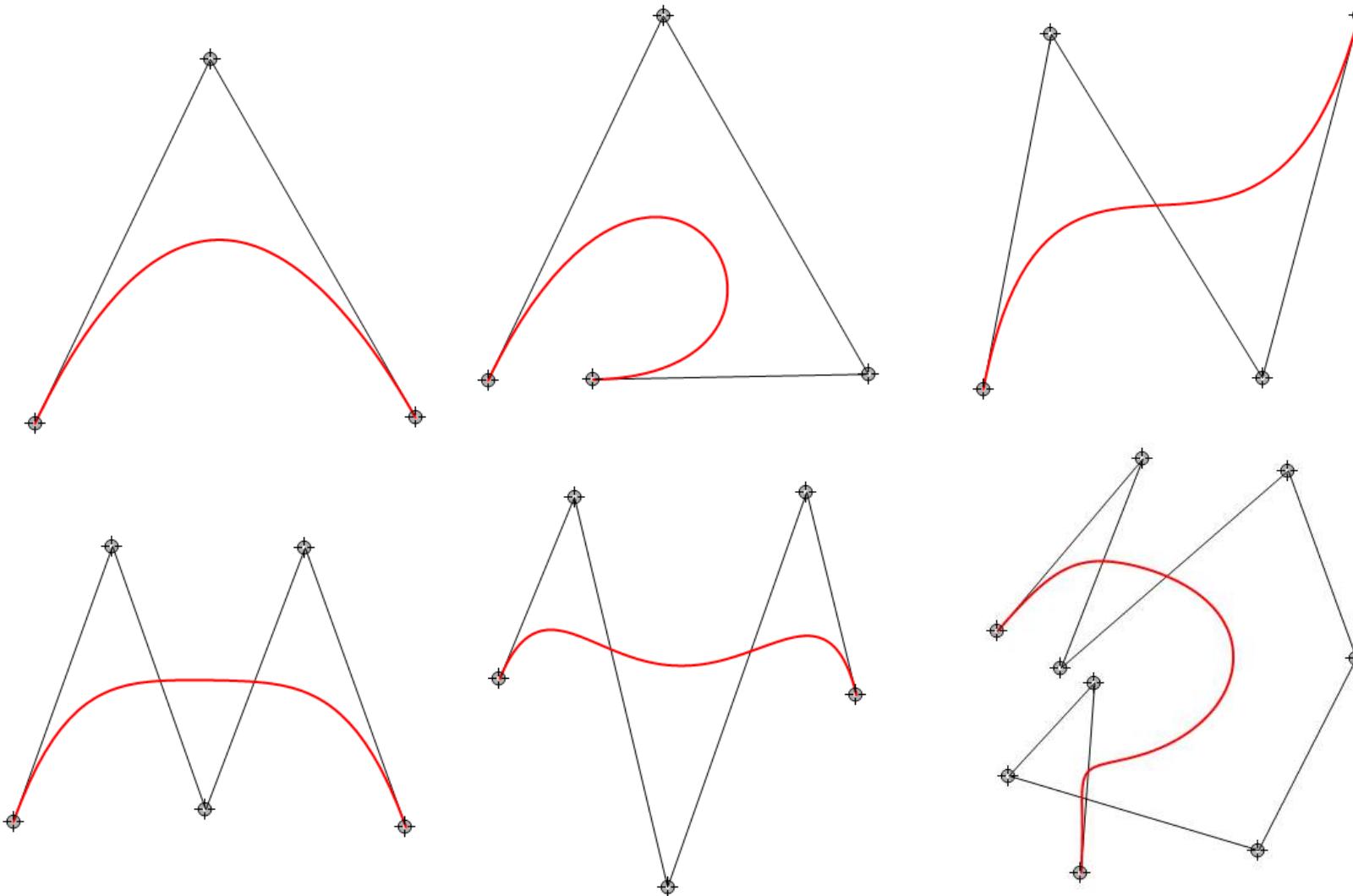
- When it comes to piecing together Bézier curves we need to be careful
- Usually a **shared control point scheme** is used where the last control point of one piece is the first control point of the next
- This only guarantees C^0 continuity since the curve derivatives are controlled by other control points

Enforcing Continuity

- If we want the curve to be G^1 continuous the three control points at the join must be collinear, for C^1 the two line segments must be of equal length



Bézier Curves: Examples



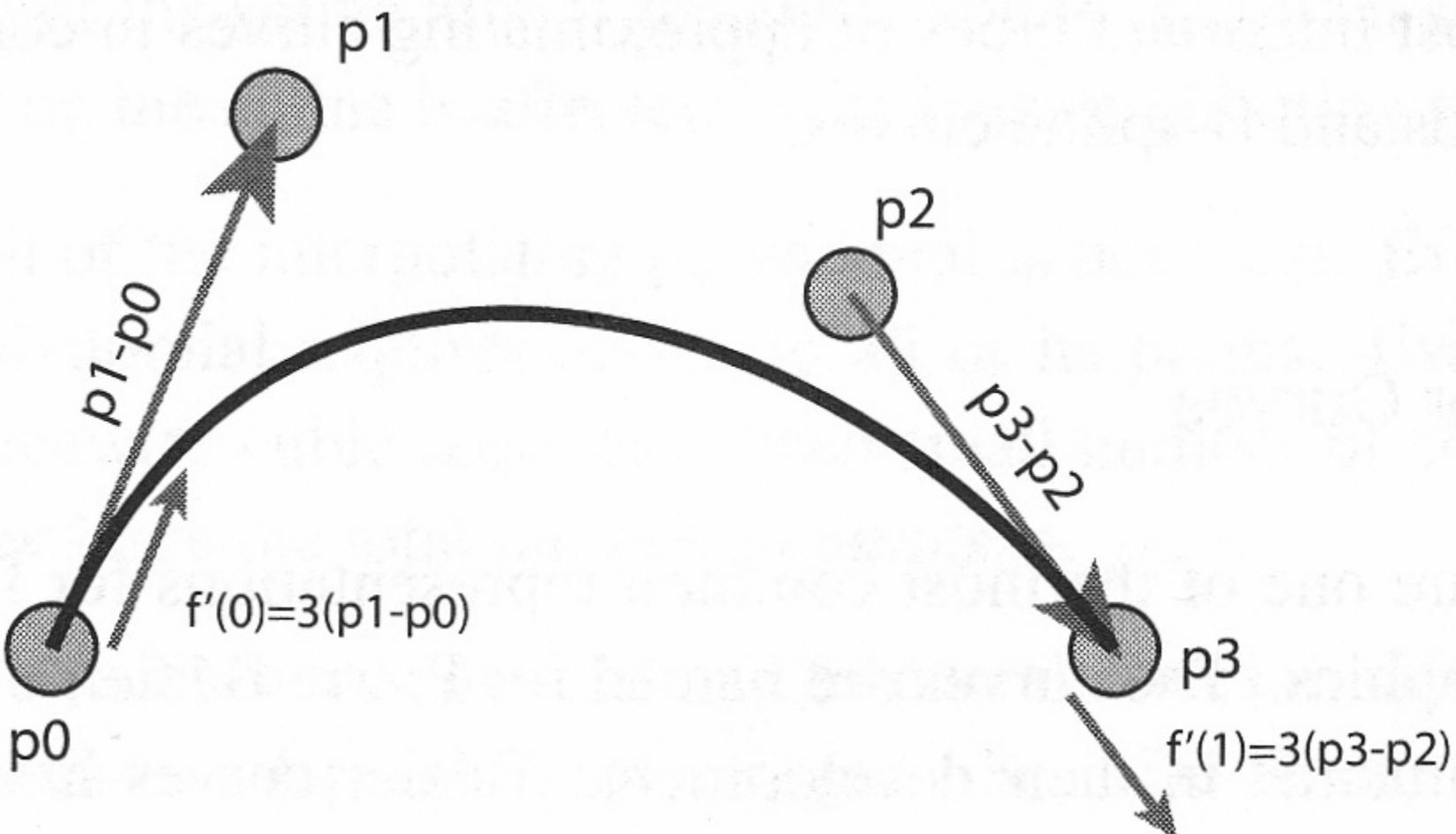
Bezier Curves

- In addition, manipulating the control points has an **intuitive impact** on the curve's shape
- It is very easy to design with Bezier curves
- The main disadvantage is they only guarantee C^0 continuity and if some constraints are placed on the control points C^1 continuity can be achieved

Cubic Bezier Curve Definition

- Interpolates its first and fourth control points and approximates the second and third
- The derivative at $u=0$ is three times the vector from the first to the second control point
- The derivative at $u=1$ is three times the vector from the third to the fourth control point
- Thus, the shape of the curve is controlled by moving the second and third control points, while the position is controlled by the first and fourth

Bezier Curve Diagram



Cubic Bezier Control Points

- Putting all of this information together we get the following:

$$p_0 = f(0) = a_0$$

$$p_3 = f(1) = a_0 + a_1 + a_2 + a_3$$

$$3(p_1 - p_0) = f'(0) = a_1$$

$$3(p_3 - p_2) = f'(1) = a_1 + 2a_2 + 3a_3$$

Rearrange...

$$p_0 = a_0$$

$$p_1 = a_0 + \frac{1}{3}a_1$$

$$p_2 = a_0 + \frac{2}{3}a_1 + \frac{1}{3}a_2$$

$$p_3 = a_0 + a_1 + a_2 + a_3$$

Bezier Constraint Matrix

$$p_0 = a_0$$

$$p_1 = a_0 + \frac{1}{3}a_1$$

$$p_2 = a_0 + \frac{2}{3}a_1 + \frac{1}{3}a_2$$

$$p_3 = a_0 + a_1 + a_2 + a_3$$

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1/3 & 0 & 0 \\ 1 & 2/3 & 1/3 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

Constraint Matrix -> Blending Matrix

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1/3 & 0 & 0 \\ 1 & 2/3 & 1/3 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

$$B = C^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -3 & 3 & 0 & 0 \\ 3 & -6 & 3 & 0 \\ -1 & 3 & -3 & 1 \end{bmatrix}$$

<http://www.bluebit.gr/matrix-calculator/>

Cubic Bezier Bernstein Polynomials

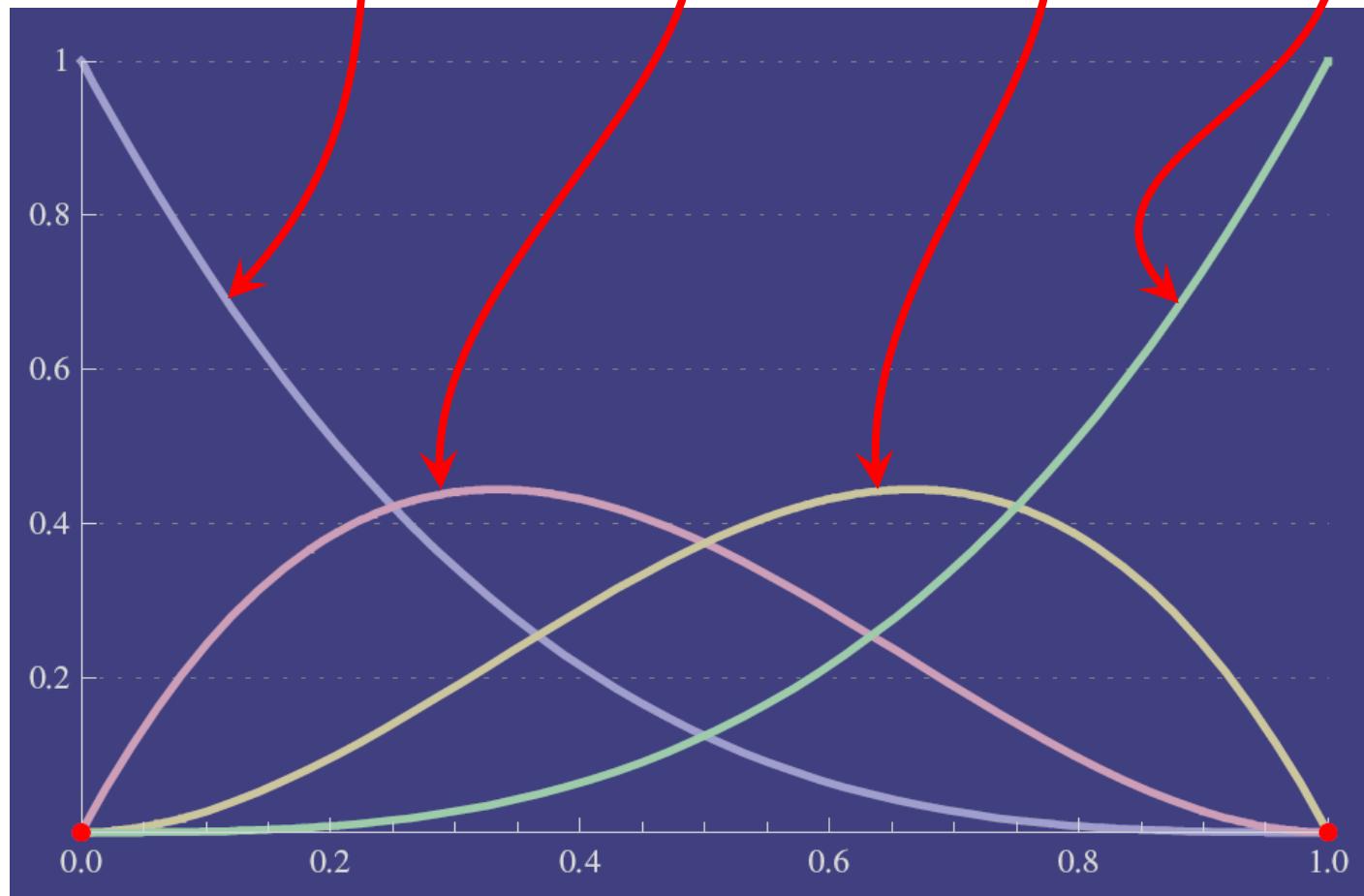
- If we multiply out \mathbf{uBp} and simplify, we get the following result:

$$f(u) = \sum_{i=0}^3 b_{i,3} p_i$$
$$b_{0,3} = (1-u)^3$$
$$b_{1,3} = 3u(1-u)^2$$
$$b_{2,3} = 3u^2(1-u)$$
$$b_{3,3} = u^3$$
$$B = C^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -3 & 3 & 0 & 0 \\ 3 & -6 & 3 & 0 \\ -1 & 3 & -3 & 1 \end{bmatrix}$$

- The blending functions are called the Bernstein polynomials

Bernstein Polynomials Visualized

$$P(t) = P_0(1-t)^3 + P_1 3t(1-t)^2 + P_2 3t^2(1-t) + P_3 t^3$$



Bézier Curves: Blending Functions

- Evaluating a point on a Bezier curve:

$$\begin{aligned} p(t) &= p_0 b_{0,3}(t) + p_1 b_{1,3}(t) + p_2 b_{2,3}(t) + p_3 b_{3,3}(t) \\ &= p_0 (1-t)^3 + p_1 3t(1-t)^2 + p_2 3t^2(1-t) + p_3 t^3 \end{aligned}$$

Curves and Smooth Surfaces

Freeform Surfaces

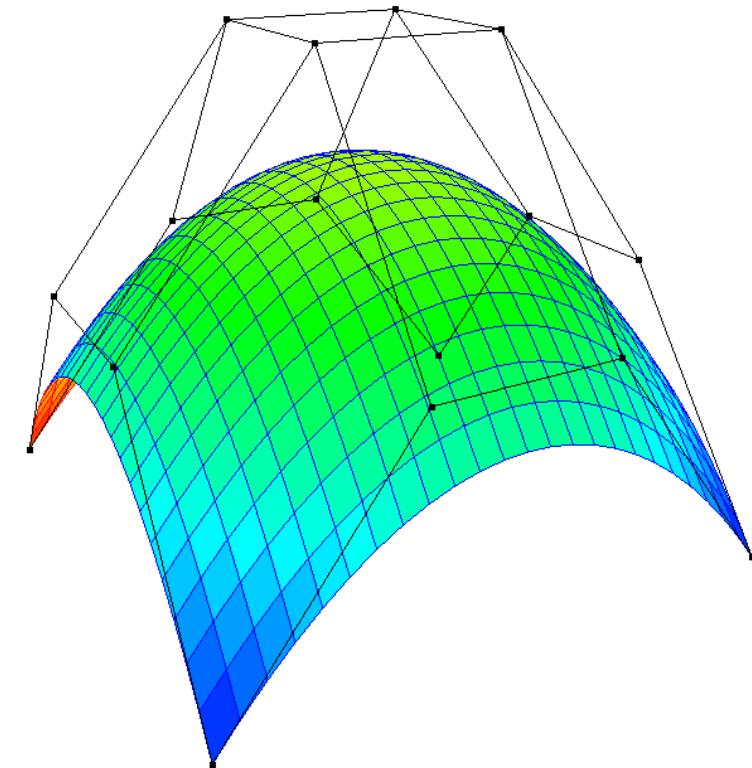
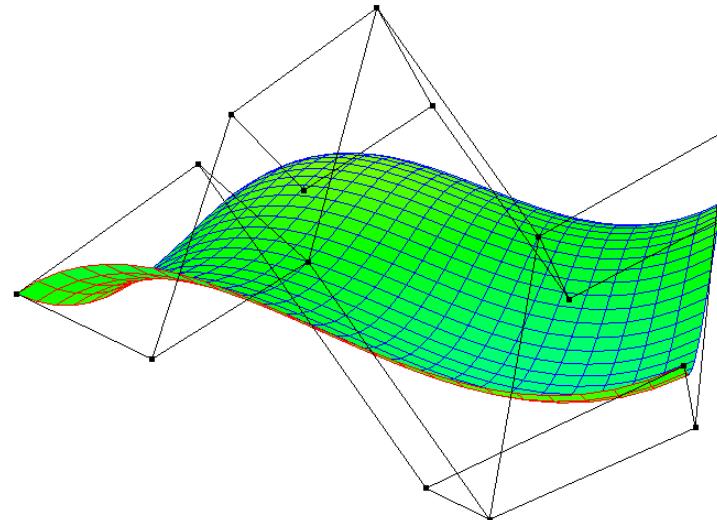
Freeform Surfaces

- base surfaces on parametric curves
- Bézier curves → Bézier surfaces/patches
- mathematically:
application of curve formulations
along two parametric directions

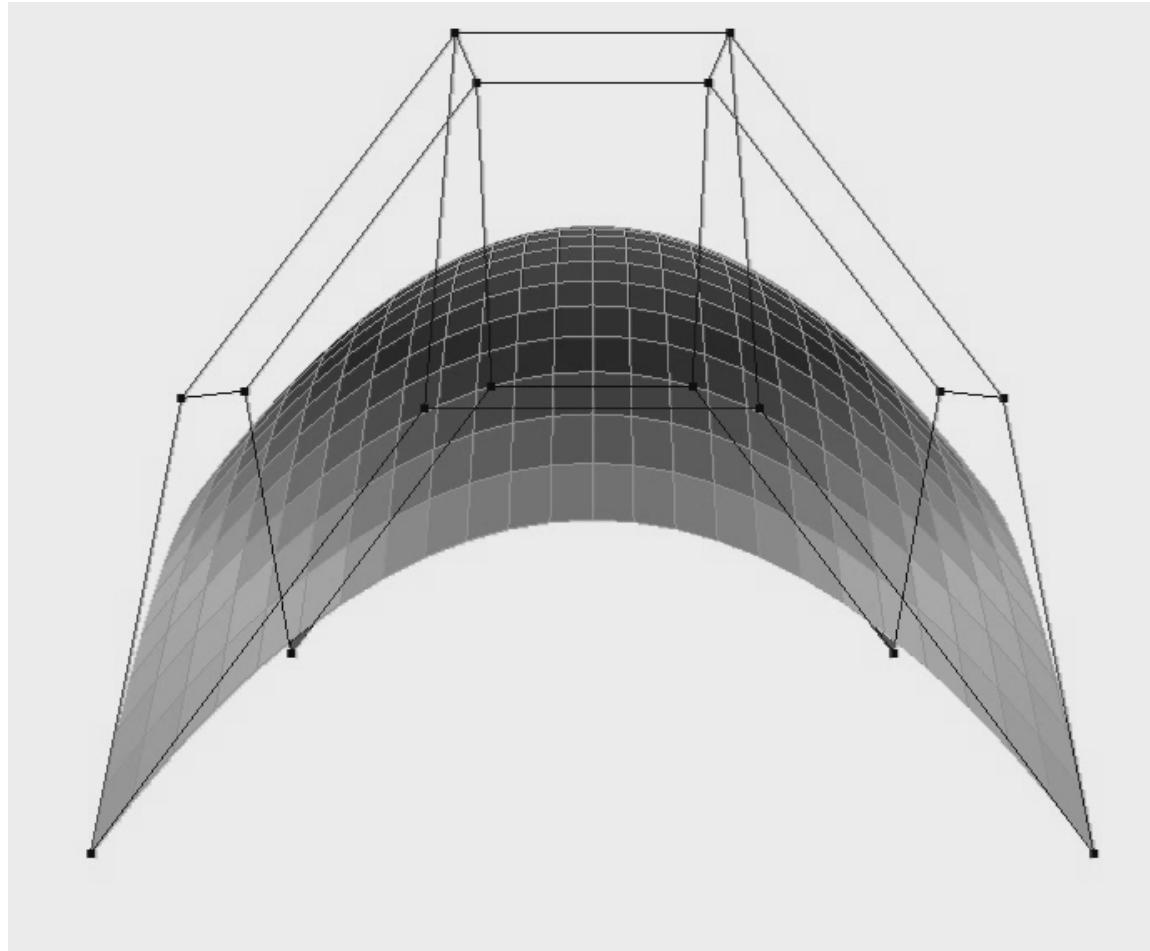
Freeform Surfaces: Principle

- Bézier surface: control mesh with $m \times n$ control points now specifies the surface:

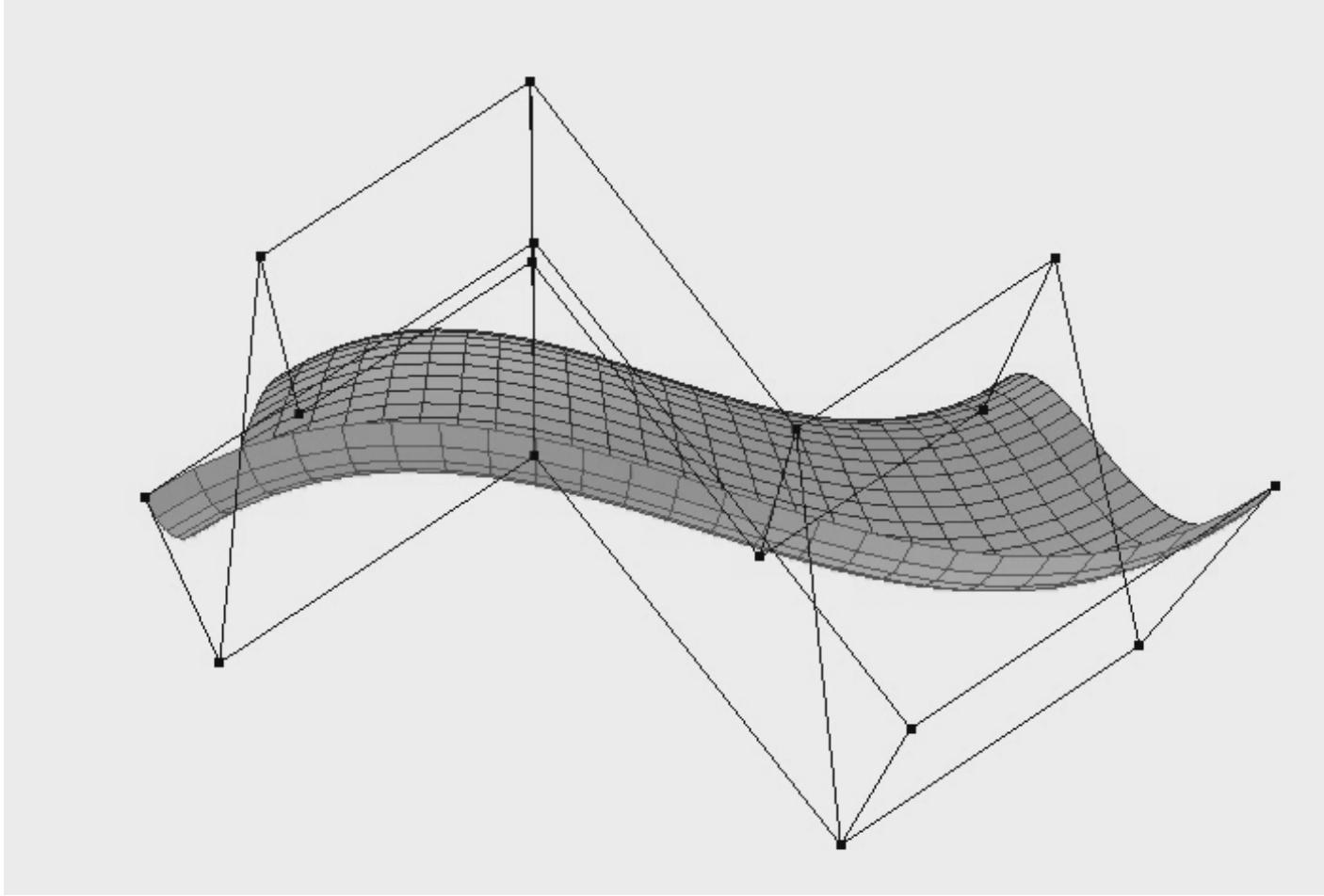
$$P(u,v) = \sum_{j=0}^m \sum_{i=0}^n P_{j,i} B_{j,m}(v) B_{i,n}(u)$$



Freeform Surfaces: Examples



Freeform Surfaces: Examples



The Utah Teapot

- famous model used early in CG
- modeled from Bézier patches in 1975
- used frequently in CG techniques as an example along with other “famous” models like the Stanford bunny



The Utah Teapot



Freeform Surfaces: How to Render?

- freeform surface specification yields
 - points on the surface (evaluating the sums)
 - order of points (through parameter order)
- extraction of approximate polygon mesh
 - chose parameter stepping size in u and v
 - compute the points for each of the steps
 - create polygon mesh using the inherent order
- can be created as detailed as necessary

Curves and Surfaces: Summary

- need to model smooth curves & surfaces
- use of control points
- polynomial descriptions
- continuity constraints C^n/G^n ,
important both for curves and surfaces
- surfaces from curves

Next Class

- Rendering

CSCI 3090

Viewing and Perspective

Mark Green

Faculty of Science

Ontario Tech

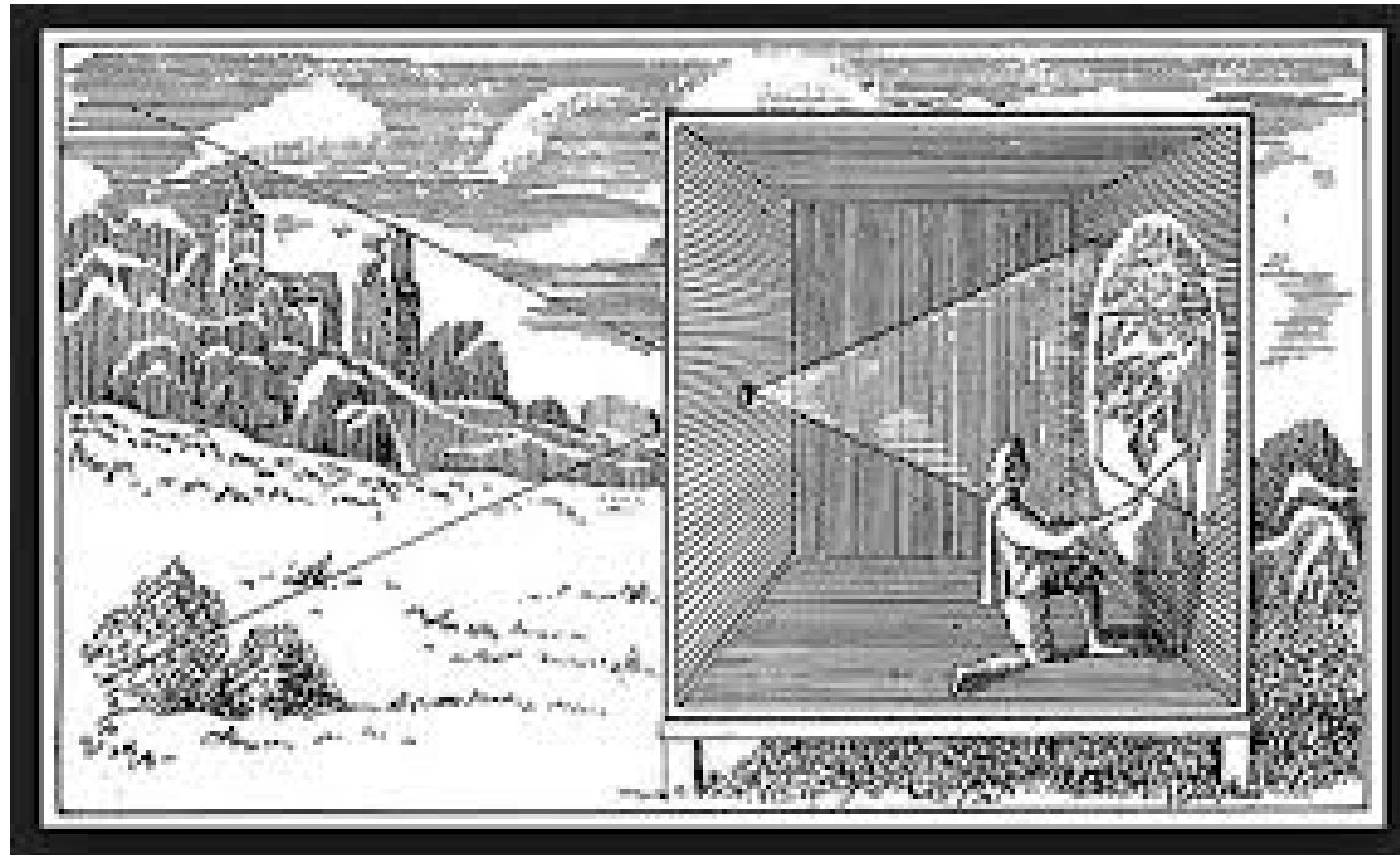
Art History

<https://youtu.be/bkNMM8uiMww>

Art History

- This is a very Euro-centric view of art history and mathematics
- The Greeks did know about perspective, the Europeans lost this information when they trashed libraries
- The Arab world kept this knowledge and the theory of perspective was further developed at their universities
- It wasn't until the 1400s that the Europeans rediscovered perspective and then claimed that they invented it

Camera Obscura



Goals

- By the end of today's class, you will be equipped to:
 - Describe the transformation matrix used to create perspective and orthogonal projections

Rendering Intro

Introduction

- Rendering is the process of converting a model to an image
- This is a very broad topic and it covers a wide range of techniques
- We will look at the basics now, and come back to look at more advanced techniques later

Photorealistic Rendering

- Most of the rendering techniques are aimed at photorealistic image production
- They aim to produce an image that is similar to what a camera would produce
- We want the image to be as realistic as possible, not be able to distinguish the computer generated image from real life
- This has been one of the major goals of computer graphics over the past 40 years

Non-Photorealistic Rendering

- There is also non-photorealistic rendering
- This is producing an image that doesn't correspond to real life, something that wouldn't be produced by a camera
- There are many uses of this type of rendering:
 - Illustrations
 - Visualization
 - Art

Non-Photorealistic Rendering

- In non-photorealistic rendering the image has some purpose, it is being used to illustrate some concept or process
- We are trying to draw attention to particular parts of the image, highlight the important parts
- Quite often realism gets in the way
- We will look more at these techniques in the visualization part of the course

Rendering Process

- There are three standard steps in any rendering process:
 - Viewing and projection
 - Hidden surface removal
 - Determining surface colour
- We will look at all three of these processes in some detail
- Apply to photo- and non-photorealistic rendering

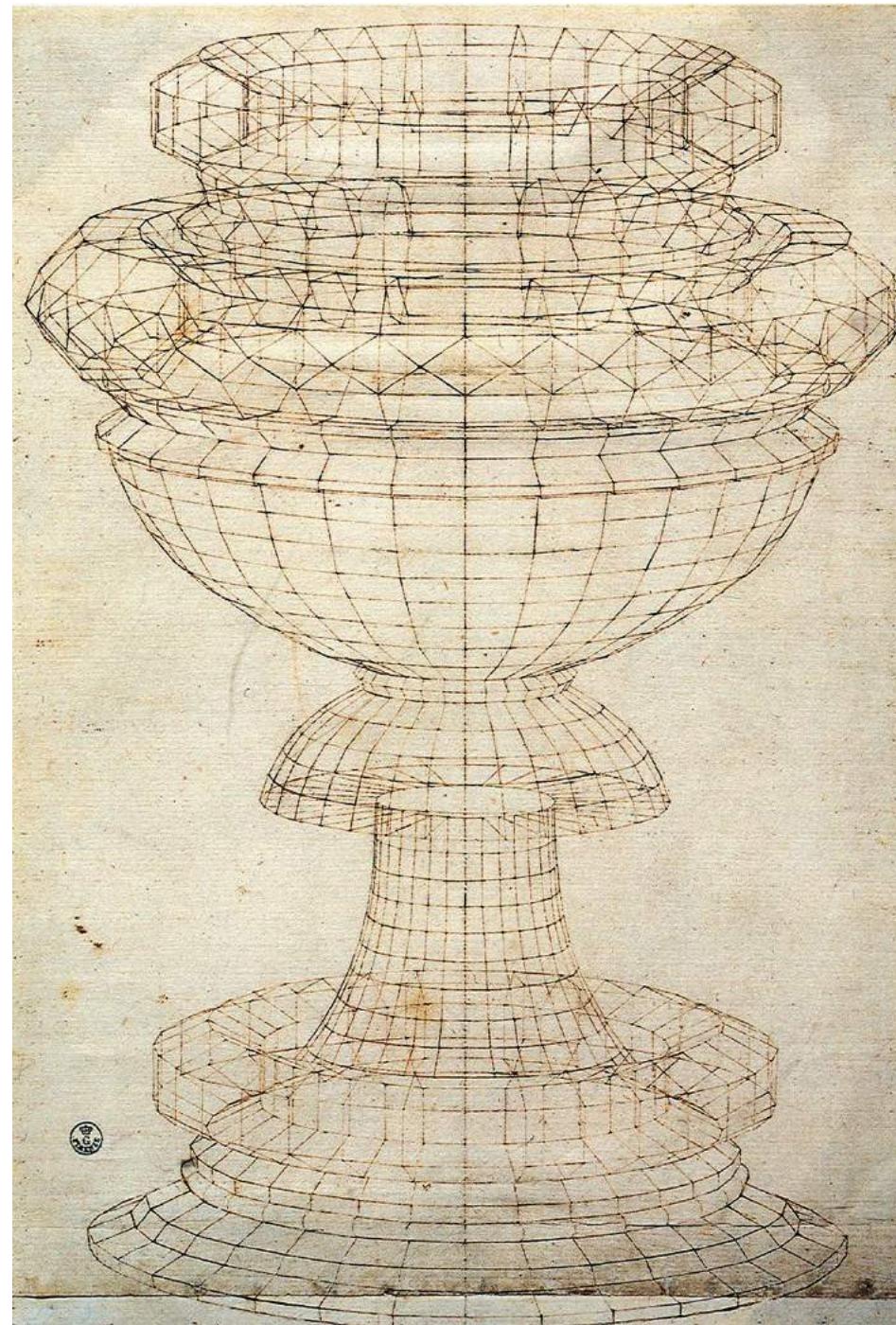
Viewing and Projection

Rendering

Rendering Process

- There are three standard steps in any rendering process:
 - Viewing and projection
 - Hidden surface removal
 - Determining surface colour
- We will look at all three of these processes in some detail
- Apply to photo- and non-photorealistic rendering

Paolo Uccello,
1430-1440



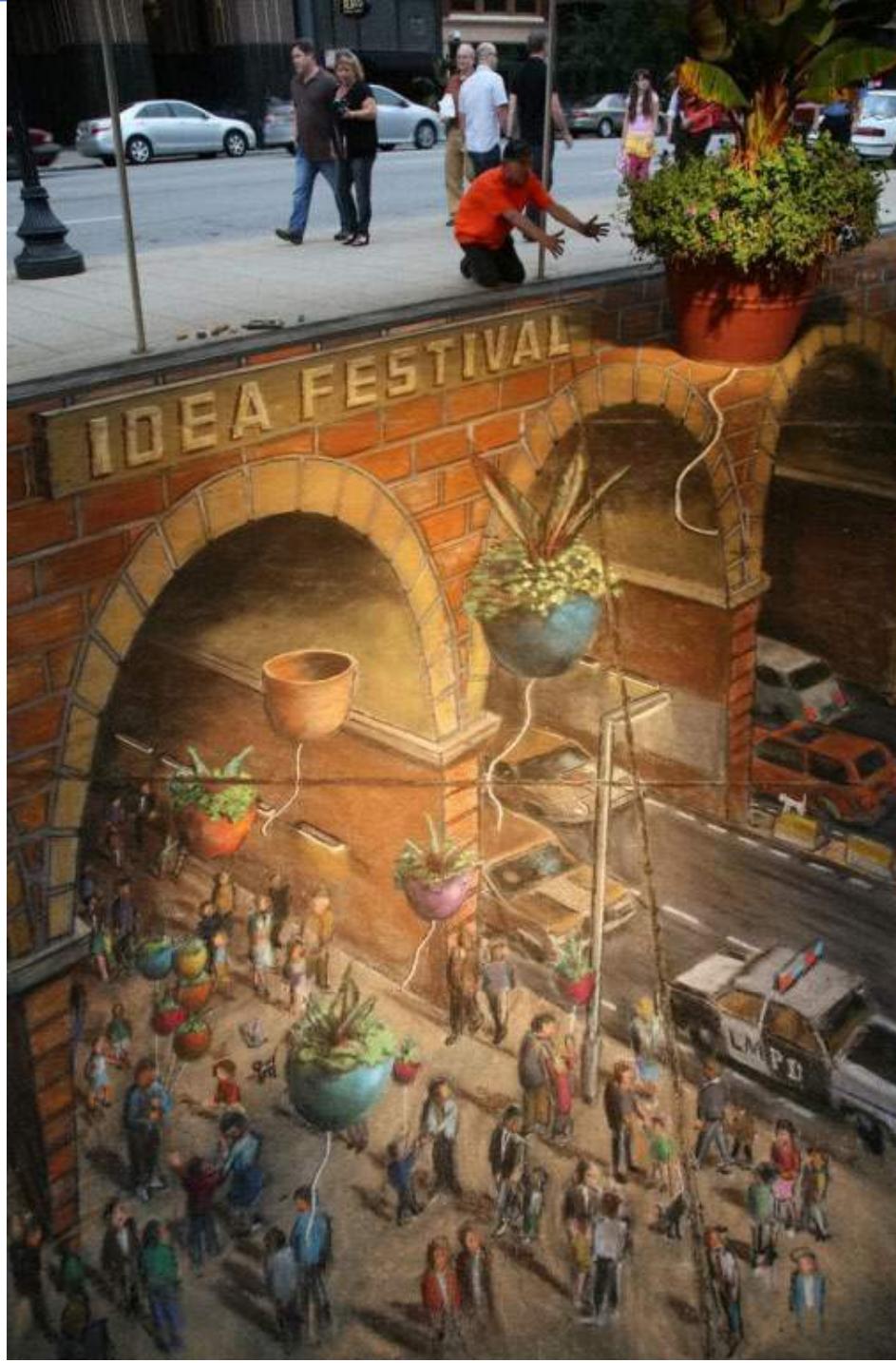


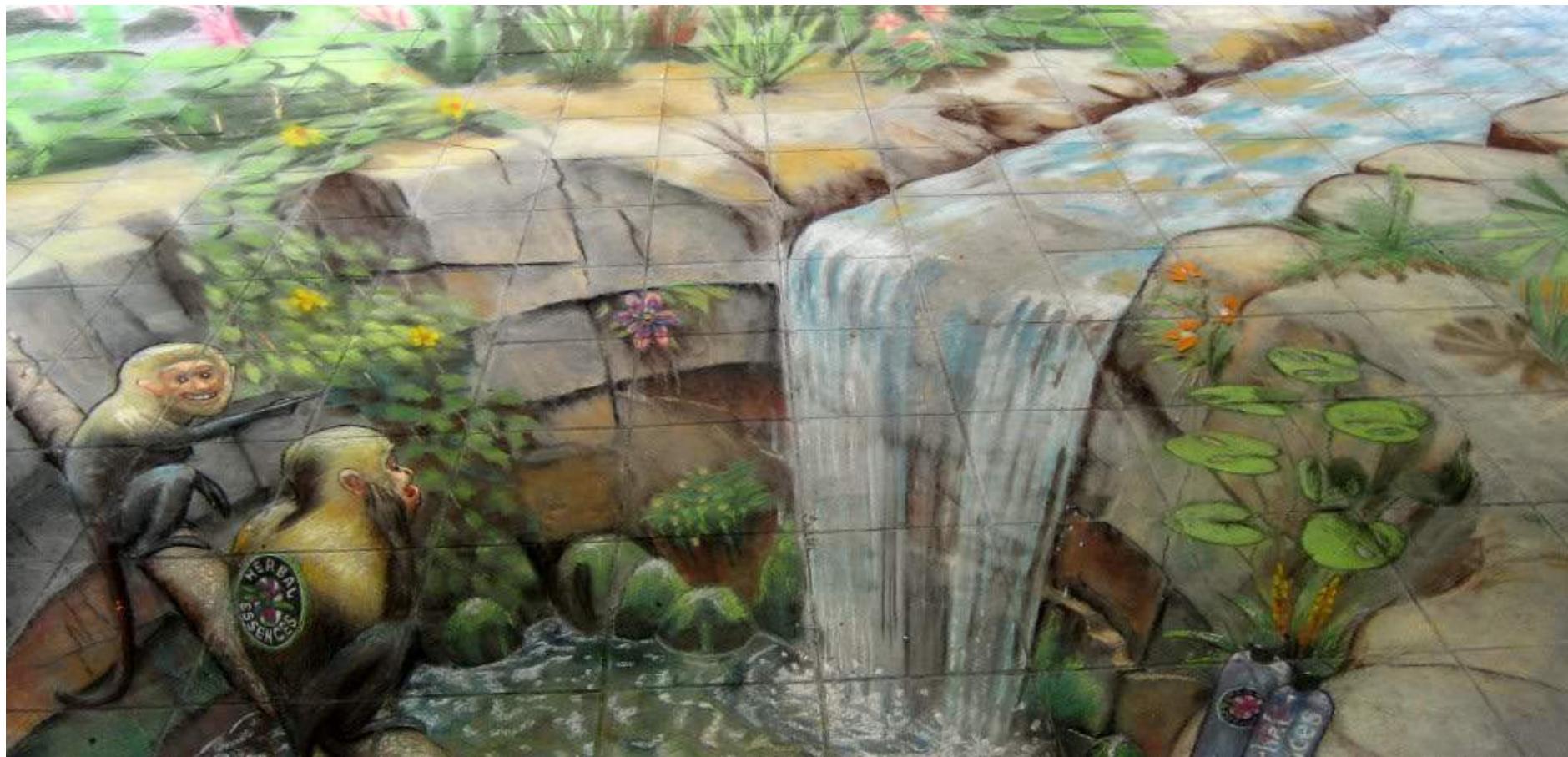
Julian Beever Anamorphic Drawings





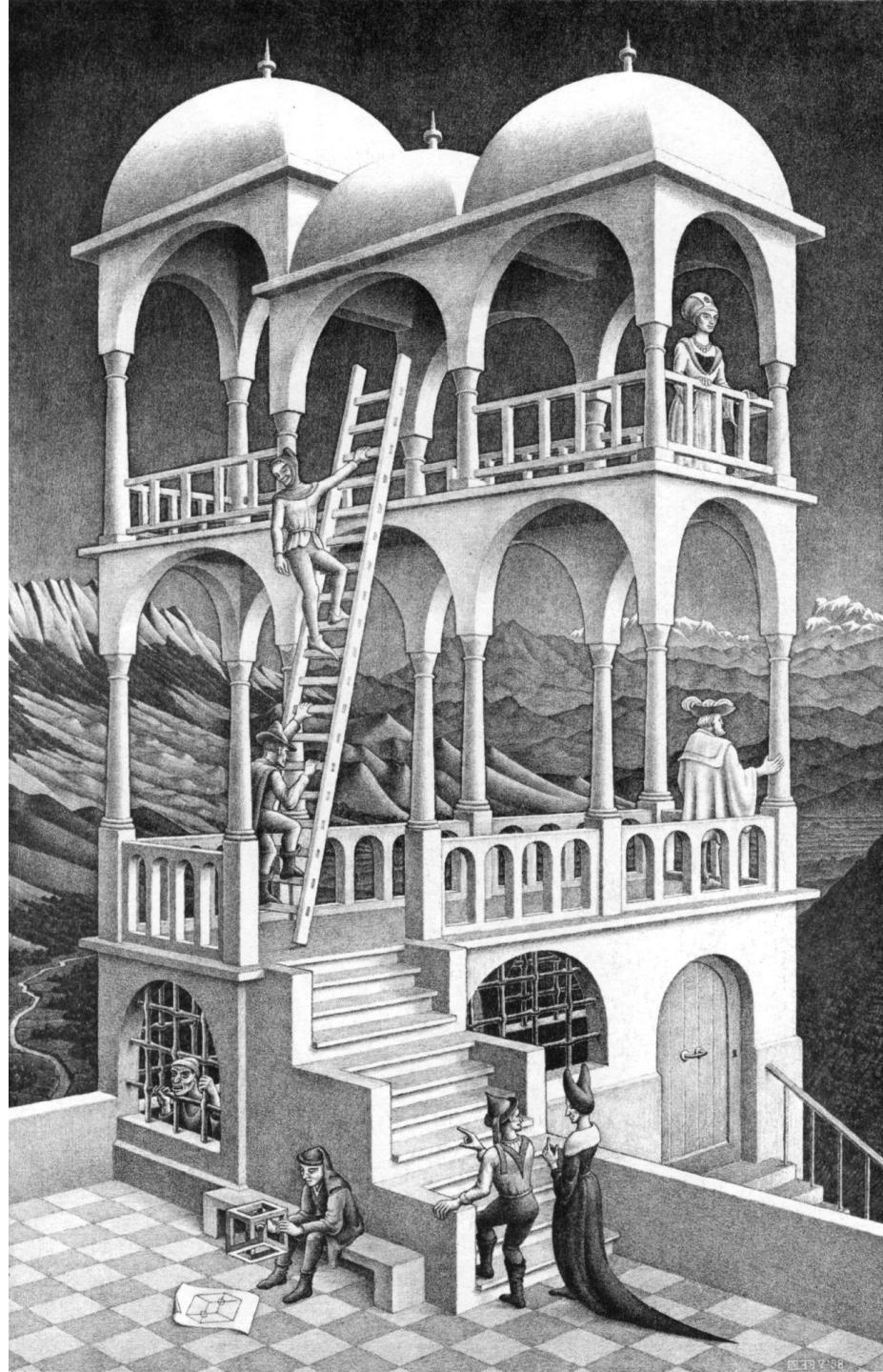








<http://www.fotosbuzz.com/julian-beever-arte-en-3d>



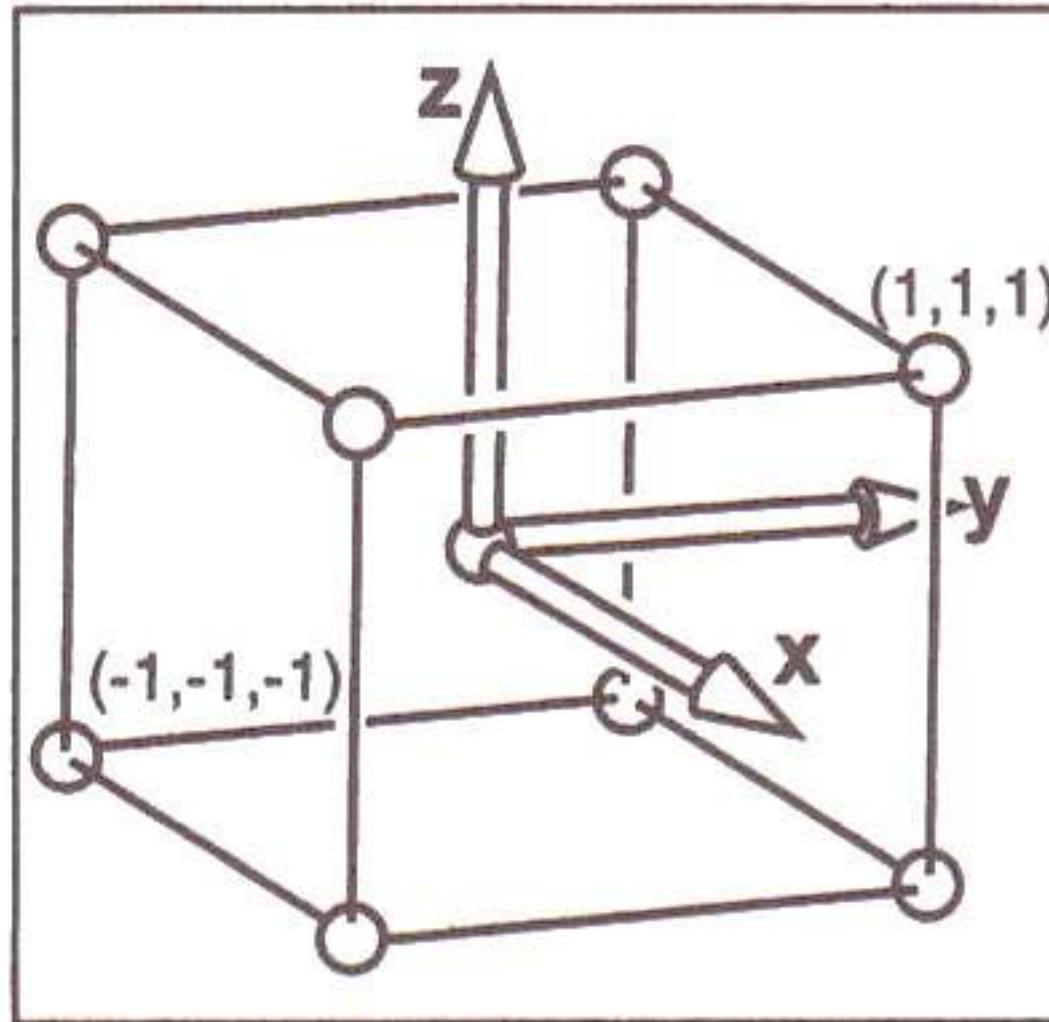
A Stepwise Approach

- Viewing is often divided into a number of steps, with each step producing its own matrix
- There are many ways of doing viewing, depending upon the coordinate systems, projections, and other factors
- By dividing it into steps we can produce a matrix for each step, and then multiply them together in the end
- This makes it easier to mix and match the different ways of doing things

Canonical View Volume

- The first thing we need to do is map the canonical view volume onto the screen
- The canonical view volume is a cube centered on the origin with each axis going from -1 to +1
- In other words all the points in the canonical view volume satisfy the set expression $(x, y, z) \in [-1, +1]^3$

Canonical View Volume



Canonical View Volume -> Pixel Raster

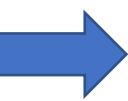
- We map the canonical view volume onto a screen that has n_x by n_y pixels
 - $x=-1$ to the left side of the screen
 - $x=+1$ to the right side
 - $y =-1$ to the bottom
 - $y=+1$ to the top
- The pixel coordinates are for the [center of the pixel](#), and each pixel has size 1×1 in screen coordinates, this is an important detail that doesn't make much difference to the math now

Canonical View Volume -> Pixel Raster

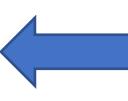
- This is basically a scale followed by a translate
- We need to scale from $[-1, +1]$ to pixel coordinates, and then translate so the origin is in the lower left corner instead of in the middle
- We usually preserve the z coordinate, so we can use it in hidden surface removal
- This produces a 4×4 matrix which just passes the z coordinate on
 - we show a 3×3 version to save some space and make the concept clearer (z row omitted)

Screen Transformation

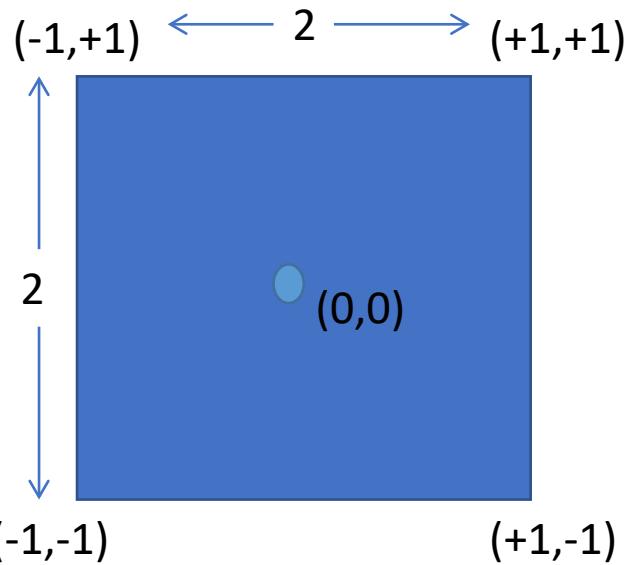
- So the pixel coordinates go from $(0,0)$ to (n_x-1, n_y-1) which really cover an area of $(-0.5, -0.5)$ to $(n_x-0.5, n_y-0.5)$ (square pixels)
- When we map onto the screen we drop the z coordinate, so the matrix that we need is:

Scale from size 2 to size n_x 

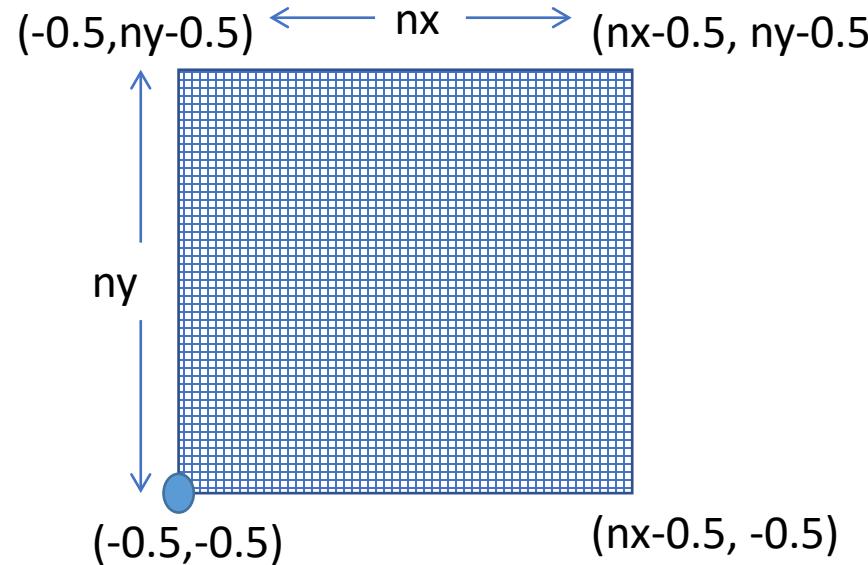
$$\begin{bmatrix} \frac{n_x}{2} & 0 & \frac{n_x - 1}{2} \\ 0 & \frac{n_y}{2} & \frac{n_y - 1}{2} \\ 0 & 0 & 1 \end{bmatrix}$$

Translate to center 

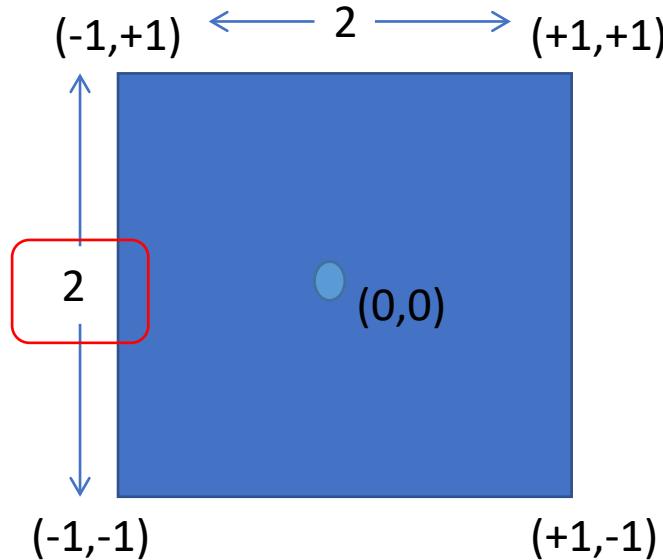
Screen Transformation



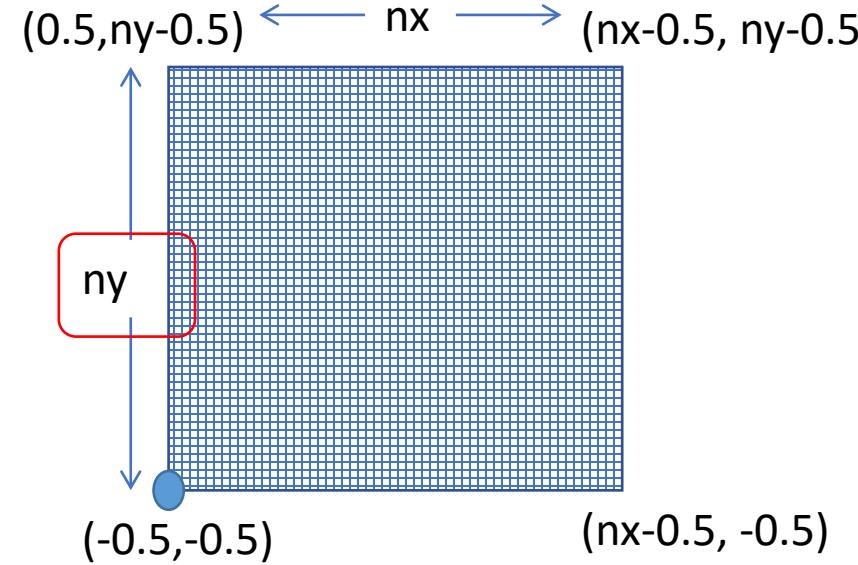
The (x,y) plane of the canonical view volume



Screen coordinates specifying pixel centres

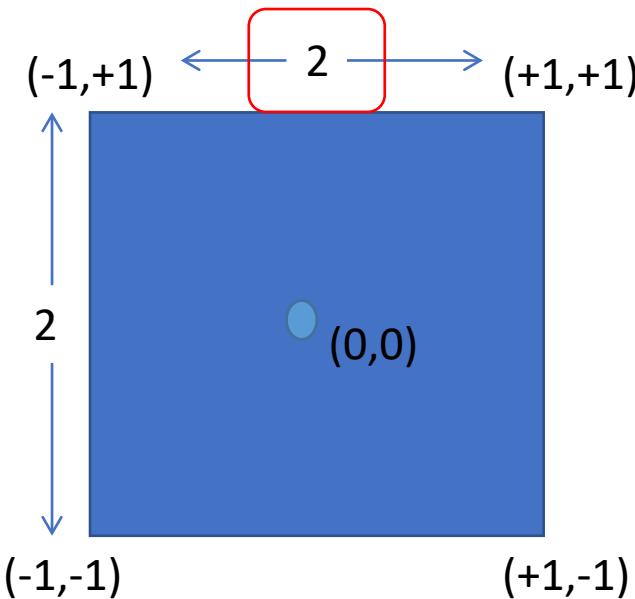


The (x, y) plane of the canonical view volume

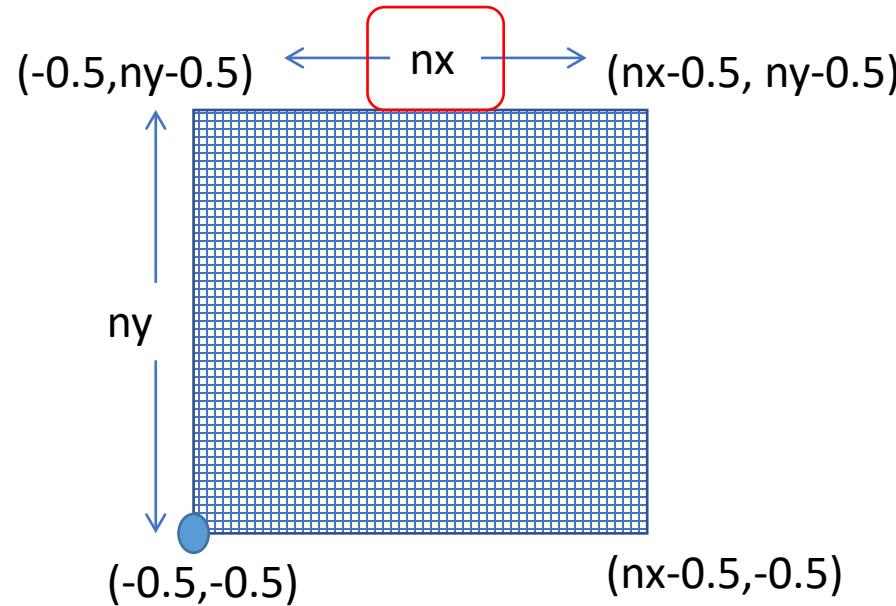


Screen coordinates specifying pixel centers

$$\begin{bmatrix} \frac{n_x}{2} & 0 & \frac{n_x - 1}{2} \\ 0 & \frac{n_y}{2} & \frac{n_y - 1}{2} \\ 0 & 0 & 1 \end{bmatrix}$$

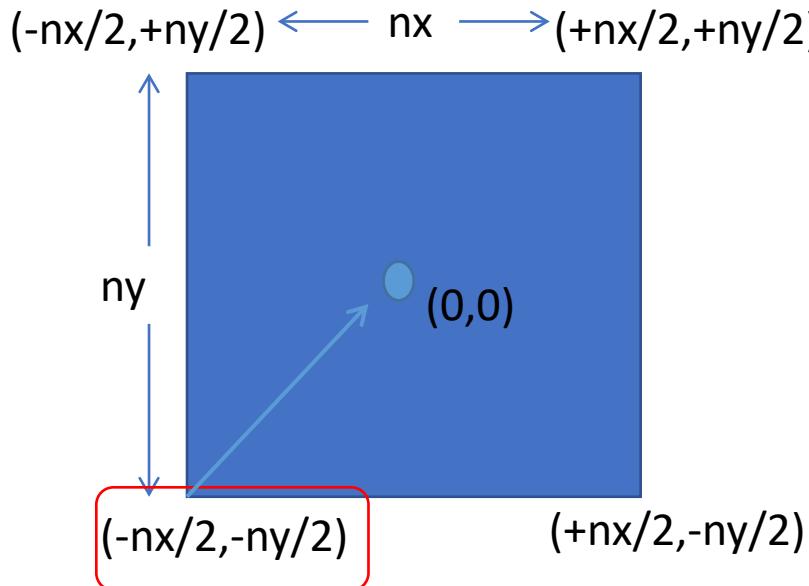


The (x, y) plane of the canonical view volume

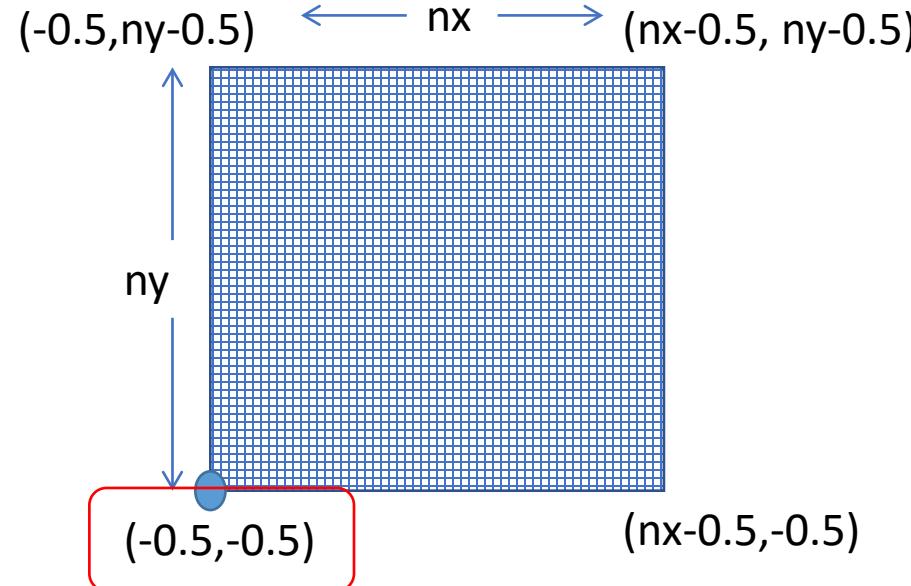


Screen coordinates specifying pixel centers

$$\begin{bmatrix} \frac{n_x}{2} & 0 & \frac{n_x - 1}{2} \\ 0 & \frac{n_y}{2} & \frac{n_y - 1}{2} \\ 0 & 0 & 1 \end{bmatrix}$$



The **scaled (x,y)** plane of
the canonical view volume



Screen coordinates
specifying pixel centers

$$\begin{bmatrix} \frac{n_x}{2} & 0 \\ 0 & \frac{n_y}{2} \\ 0 & 0 \end{bmatrix} \quad \boxed{\begin{bmatrix} \frac{n_x - 1}{2} \\ \frac{n_y - 1}{2} \\ 1 \end{bmatrix}}$$

Which way is up?

- We have assumed that the screen origin is in the lower left corner and that the y axis points up
- Some displays have their origin in the upper left corner with the y axis pointing down
- This can be handled by doing a reflection about a horizontal line through the middle of the screen
- This is the same as multiplying the y scale factor by -1, the rest of the matrix stays the same

Viewing

- Using the canonical view volume is just like drawing in a restricted 2D space -> z is ignored
- Our models are in 3D space, and they could potentially use all of this space
- But, we need some way of getting this 3D information onto the 2D screen
 - We need to remove one dimension

Projection

- There are many types of projections, and they have been studied by artists and mathematicians for at least 2,500 years, so we cannot cover all of them in this course!
- Two basic types:
 - Parallel (orthographic)
 - Perspective

Parallel Projections

- The parallel projections are **not photorealistic**, since they do not mimic how our eye works, but they are very useful for technical illustrations
- Parallel projections **preserve angles, line lengths and other important geometrical quantities**, so you can accurately measure off of an image produced using a parallel projection
- We will look at one particular type of parallel projection called the **orthographic projection**

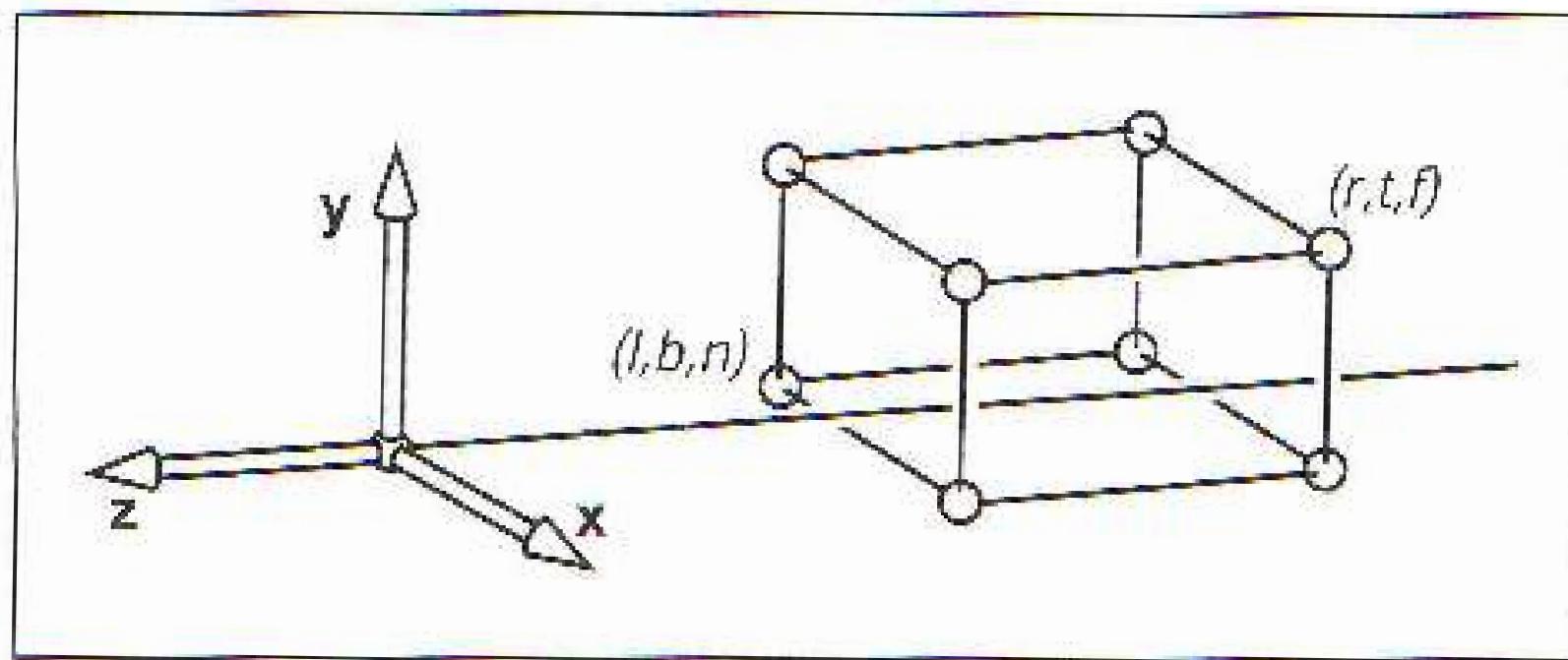
Orthographic Projection

- In an orthographic projection the volume occupied by our model is an axis aligned box, that is the sides of the box are perpendicular to the coordinate axis
- This box is defined by 6 planes:
 - $x = l$ (the left plane)
 - $x = r$ (the right plane)
 - $y = b$ (the bottom plane)
 - $y = t$ (the top plane)
 - $z = n$ (the near plane)
 - $z = f$ (the far plane)

Orthographic Projection

- We can view our space as $[l,r] \times [b,t] \times [n,f]$
- Now we have to deal with coordinate systems
- On the screen we would like to think of the y axis pointing up and the x axis pointing right (right hand rule)
- In order for this to happen we must be looking down the negative z axis, that is the positive z axis is pointing out of the screen
- So, we can view all of the z coordinates in our model as being negative, resulting in $n > f$

Orthographic Projection



Orthographic Projection

- May seem odd that $n > f$, however, we also have $|n| < |f|$, so if we think in terms of absolute values everything is okay
- Some people have used left handed coordinate systems to get around this problem, but this can actually make the problem worse and not better

Orthographic Space -> Canonical Space

- We are going to transform our orthographic space into the canonical view volume, since we already know how to deal with the canonical view volume
- This involves both a scale and a translation
- We need to scale to the $[-1, +1]$ space of the canonical view volume, and then translate our space so it is centered on the canonical origin

Viewing Matrix

- The matrix we need is:

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

- Combining this with the previous matrix gives

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

Orthographic Volume to Screen

- If we have a point within the orthographic view volume we can multiply this point by M_o to get its position on the screen
- Similarly we can multiply all the vertices of a polygon by M_o to get their positions in screen space
- This will be a 2D polygon that is the projection of the 3D one, which we can now draw

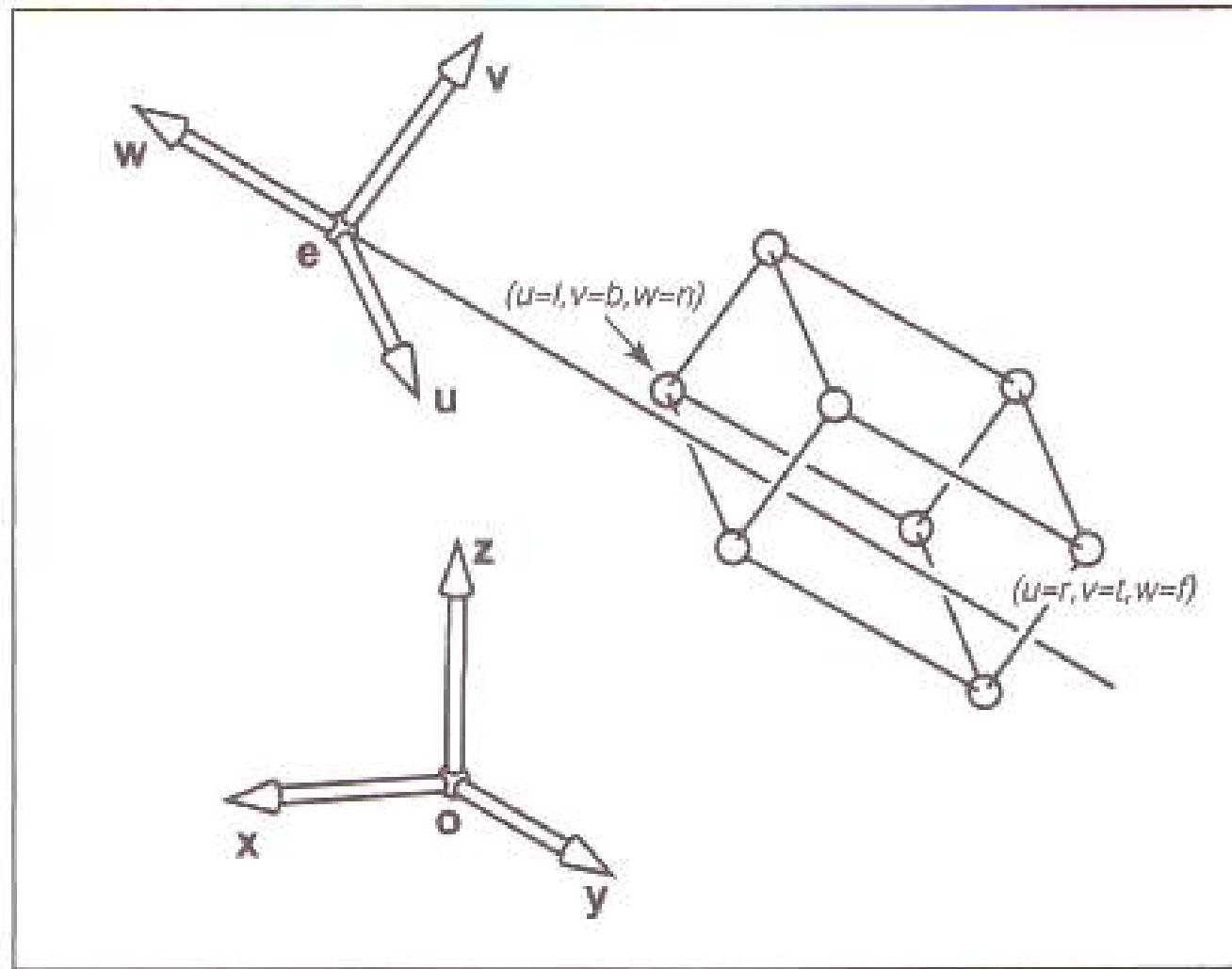
What are we looking at?

- So far we have assumed that the viewer is at a fixed position and looking in a fixed direction
- Right now our eye is at the origin and looking down the $-z$ axis
- The `lookAt()` procedure in `glm` allows us to specify the eye position, the direction that we are looking and the up vector
- We would like to add this to our viewing process – we need a matrix to do it

Look At Parameters

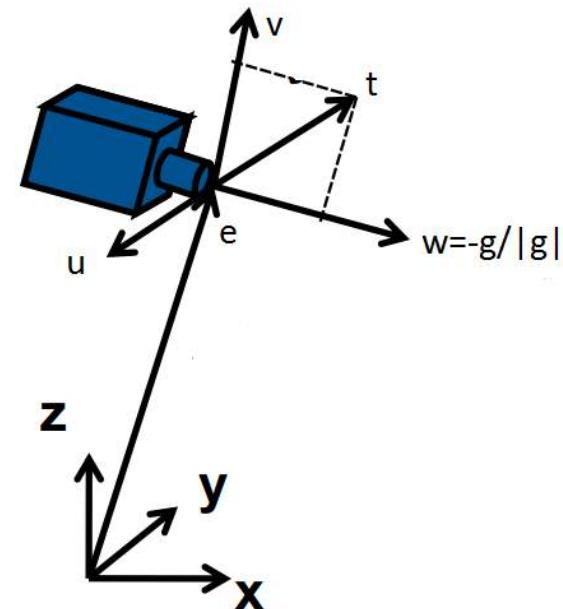
- We have the following pieces of information:
 - e – the eye position
 - g – the direction the viewer is looking
 - t – the view *up* vector
- We want to construct a *uvw* coordinate system at the viewer's eye and then rotate and translate it onto the *xyz* axis and the origin

Viewing



Viewing Transformation

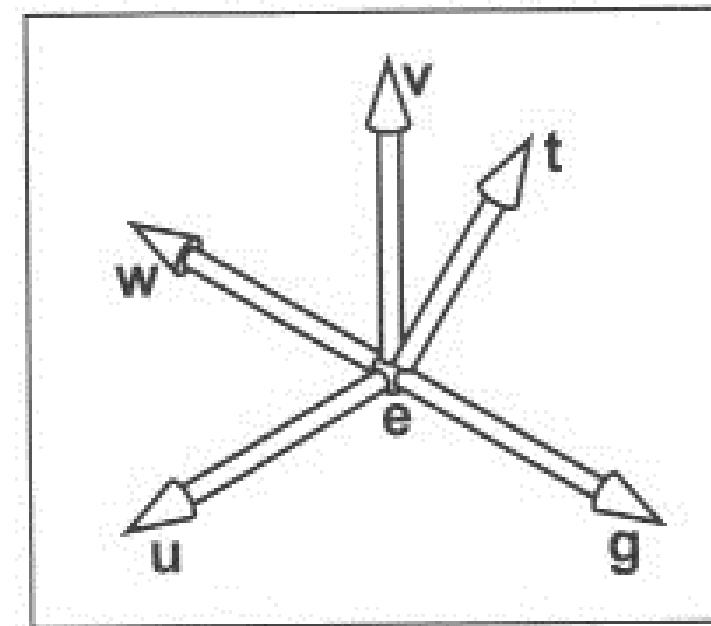
- Goal:
 - Camera: at origin, view along $-z$, y upwards (right handed)
 - Translation of e to the origin
 - Rotation of g to $-Z$ -axis
 - Rotation of *projection* of t to y



Viewing Matrix

- First want to rotate g so it points along the $-z$ axis, so we have the following for w :

$$w = -g/|g| \rightarrow \text{rotate onto } z$$



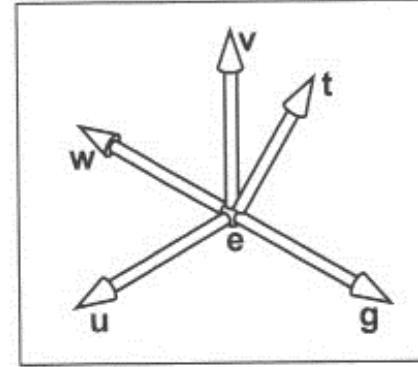
Viewing Matrix

$$w = -g/|g| \rightarrow \text{rotate onto } z$$

- Next we need a second vector which is perpendicular to w. We can use t which we know is not collinear to w:

$$u = (t \times w)/|t \times w| \rightarrow \text{rotate onto } x$$

Note: u is perpendicular to t and w

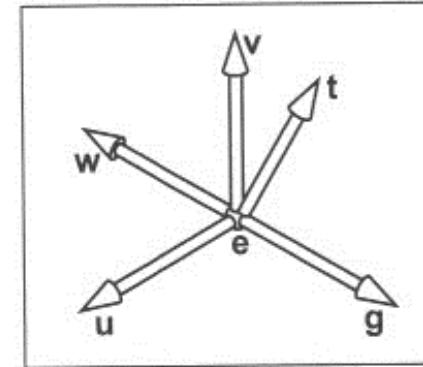


Viewing Matrix

$w = -g/|g| \rightarrow$ rotate onto z

$u = (t \times w)/|t \times w| \rightarrow$ rotate onto x

- Finally, we can get the third vector which is perpendicular to w and u:
 $v = w \times u \rightarrow$ this is a *projection of t*; rotate it onto y



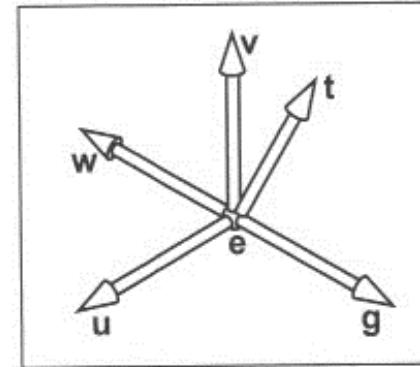
Note: this is our new ‘up’ vector as v is perpendicular to u, which is perpendicular to t.

Viewing Matrix

$w = -g/|g| \rightarrow$ rotate onto z

$u = (t \times w)/|t \times w| \rightarrow$ rotate onto x

$v = w \times u \rightarrow$ rotate onto y



W points opposite to the gaze; v is in the same plane as g and t.

- Use these to create a rotation matrix which will move uvw onto xyz .
- Translate the eye to the origin.

Rotate and Translate the Look Position and Vector

- The matrix that we need is:

$$M_v = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -e_x \\ 0 & 1 & 0 & -e_y \\ 0 & 0 & 1 & -e_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- To transform the points we need to construct the matrix $M = M_o M_v$
- We then multiply all of the points by M to get their positions on the screen

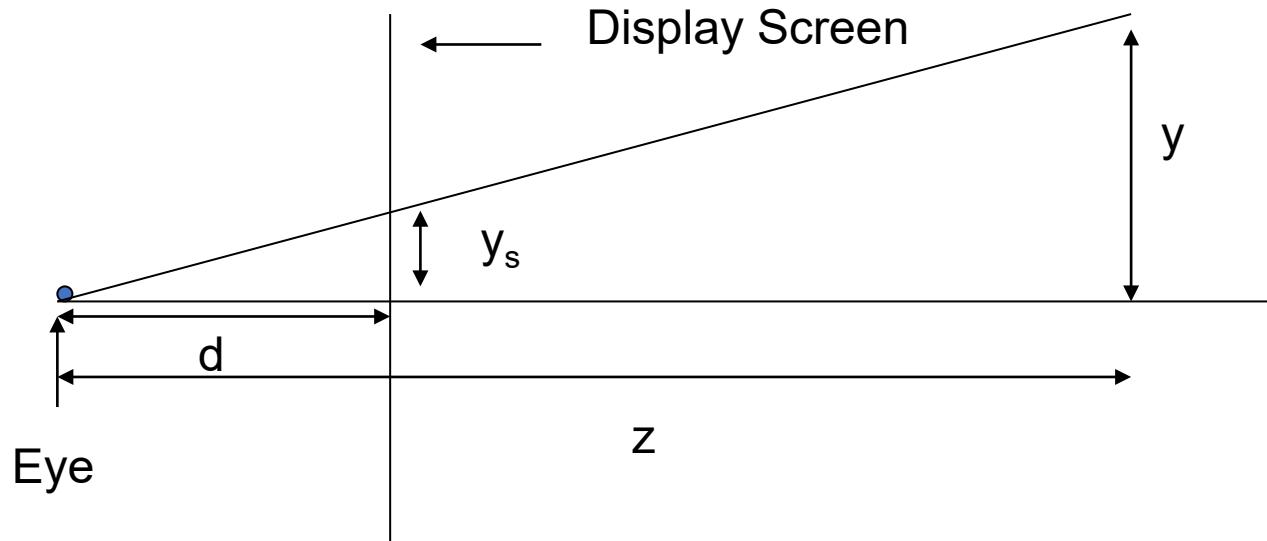
Perspective Projection

Perspective Projection

- Now we can do viewing with orthographic projections, so now we will turn our attention to the perspective projection
- In a perspective projection objects that are farther away appear to be smaller than ones that are closer
- The size of the object is scaled by their distance from the viewer
- This is what we normally see in the real world

Perspective Geometry

- The geometry of the perspective transformation is really quite simple, it is all based on similar triangles
- We can see this in a 2D projection



Similar Triangles to Define Geometry

- From the diagram we can see that
$$y_s = (d/z)y$$
- To get a perspective projection we just need to divide by z , or the distance to the point, d is basically a scaling factor
- Note that any point along the line to y will project onto the same point on the screen
- The projection basically takes this line and converts it to a line parallel to the z axis

Matrix Divide?

- Now we need to construct a matrix that will divide by z , but how do we do this?
- Matrices don't divide, they only multiply and add
- Homogeneous coordinates come to the rescue!

Homogeneous Coordinates

- Recall that we have:
 - $(hx, hy, hz, h) = (x, y, z, 1)$
- That is we divide by the fourth coordinate to get Cartesian coordinates
 - $(2,4,8,2) = (1,2,4,1)$

Homogeneous Coordinates

- So far we have kept our homogenous coordinate at 1, so we could avoid the division, but now we want to perform the division
- To solve our problem we only need to get the **z coordinate in the homogenous position**, then we will have a divide by z
- This can be done quite easily with a matrix

Perspective Transformation

- For our simple perspective transformation we have:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{d} & 0 \end{bmatrix}$$

Perspective Transformation

- If we multiply the point $(x, y, z, 1)$ by this matrix we get the point $(x, y, z, z/d)$:

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

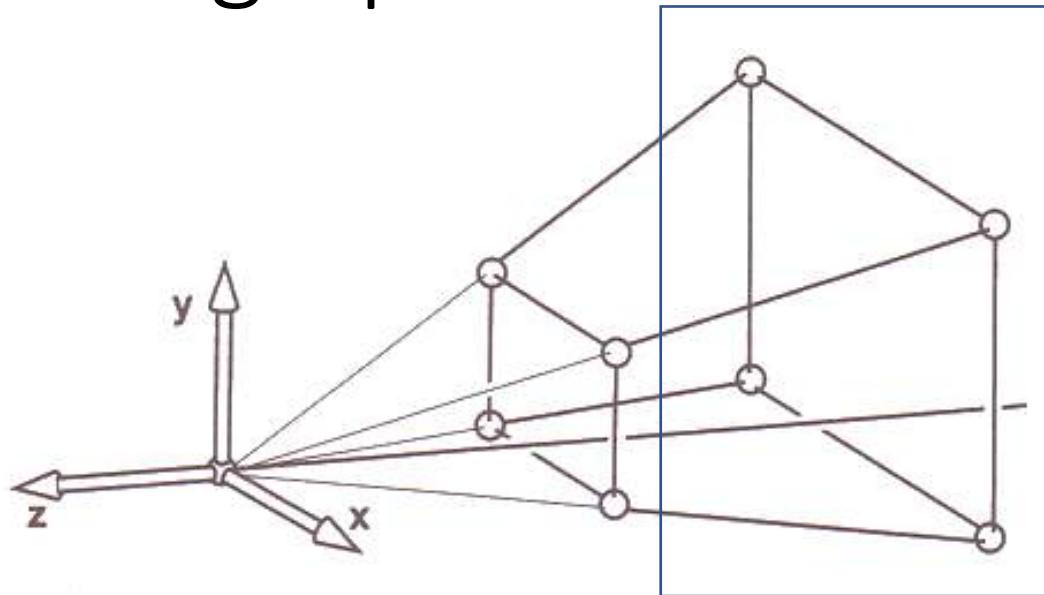
- When we divide by the homogeneous coordinate we get $(dx/z, dy/z, d, 1)$!

Perspective to Orthographic

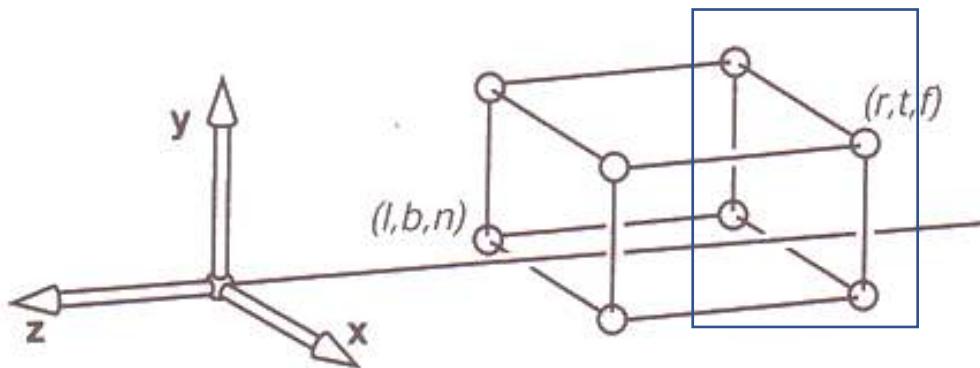
- Any transformation that produces the [divide by \$z\$](#) results in a perspective projection, so we might as well choose a convenient one
- One convenient projection converts the perspective pyramid (*the frustum*) into the orthographic volume:
 - leaves the $z=n$ plane alone
 - squishes down the pyramid so the $z=f$ end is the same size as the $z=n$ end
- We can do this by [scaling and dividing by \$z\$](#)
- The $z=n$ plane will be our projection plane, or the position of the display screen

Perspective -> Orthographic

Perspective



Orthographic



Viewing

- The matrix that does this for us is:

$$M_p = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{n+f}{n} & -f \\ 0 & 0 & \frac{1}{n} & 0 \end{bmatrix}$$

- This gives us:

$$M_p \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \frac{n+f}{n} - f \\ \frac{z}{n} \end{bmatrix} = \begin{bmatrix} \frac{nx}{z} \\ \frac{ny}{z} \\ n + f - \frac{fn}{z} \\ 1 \end{bmatrix}$$

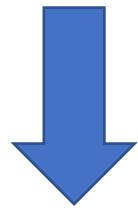
Remember this it
will come back to
haunt us

Final Matrix

- Note that this gives us the proper perspective division for x and y, but we end up with a **non-linear function for z**
- ... don't worry for now: no z in the pixel array!
- We will come back to this when we talk about hidden surface algorithms
- The matrix that we now apply to our points is
 $M = M_o M_p M_v$
- Again this transforms 3D polygons into polygons in 2D space

$M_o M_p M_v$

Orthogonal to canonical

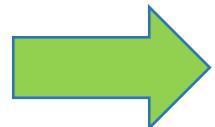


$$\begin{bmatrix} \frac{n_x}{2} & 0 & 0 & \frac{n_x-1}{2} \\ 0 & \frac{n_y}{2} & 0 & \frac{n_y-1}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & \frac{2}{n-f} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0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



Viewing angle to xyz

Perspective to orthogonal



$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{n+f}{n} & -f \\ 0 & 0 & \frac{1}{n} & 0 \end{bmatrix}$$

$$\begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -e_x \\ 0 & 1 & 0 & -e_y \\ 0 & 0 & 1 & -e_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

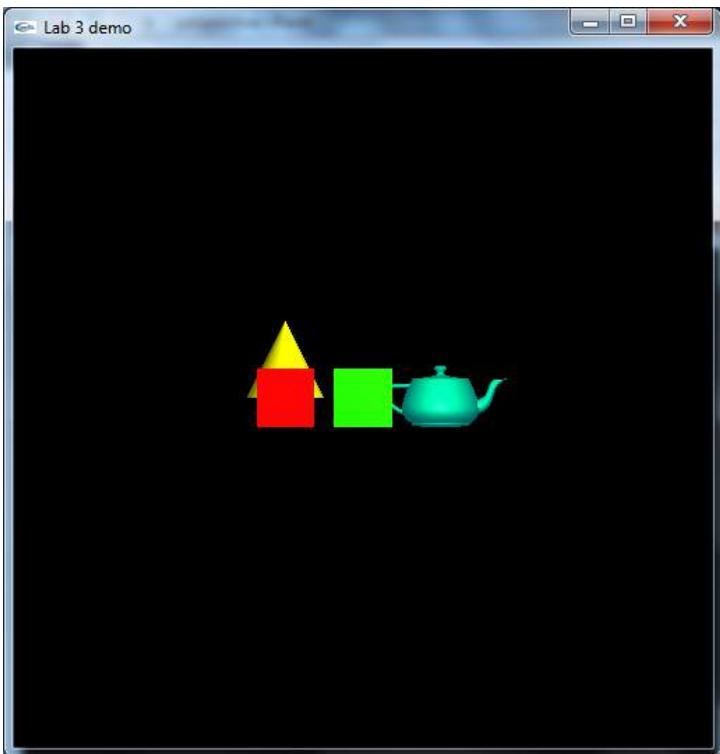
Now you know why we use the
glm library for constructing
viewing and projection matrices

Projection Matrix

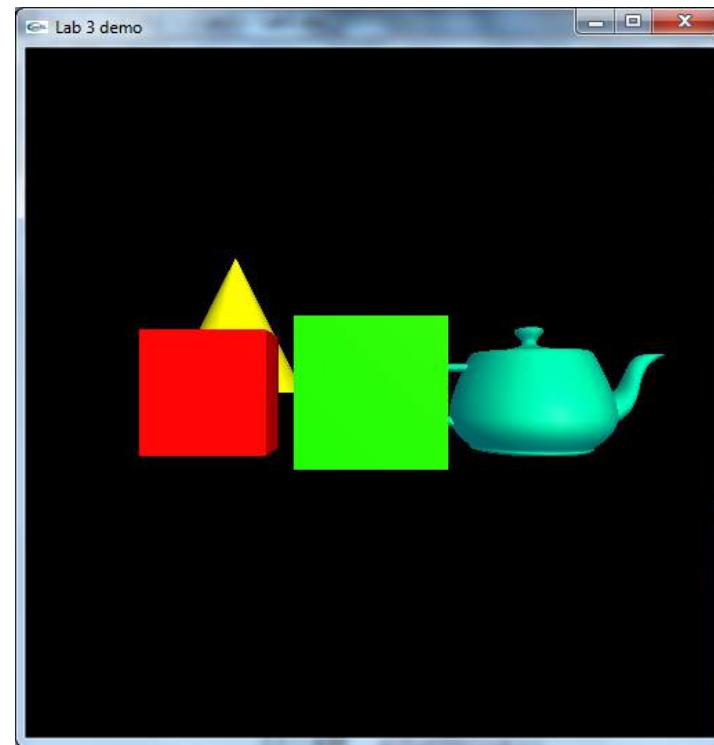
- Sometimes we combine M_o and M_p into one matrix called $M_{\text{projection}}$
- This is just the product of the two matrices
- At this point we can project and compute the viewing transformation

Comparing Projections

Orthogonal



Perspective



Summary

- Photorealistic and non-photorealistic rendering
- Viewing transformation
- Orthographic projection
- Perspective projection

Next Class

- Hidden surface algorithms

CSCI 3090

Hidden Surface

Mark Green
Faculty of Science
Ontario Tech

Some things should remain hidden.



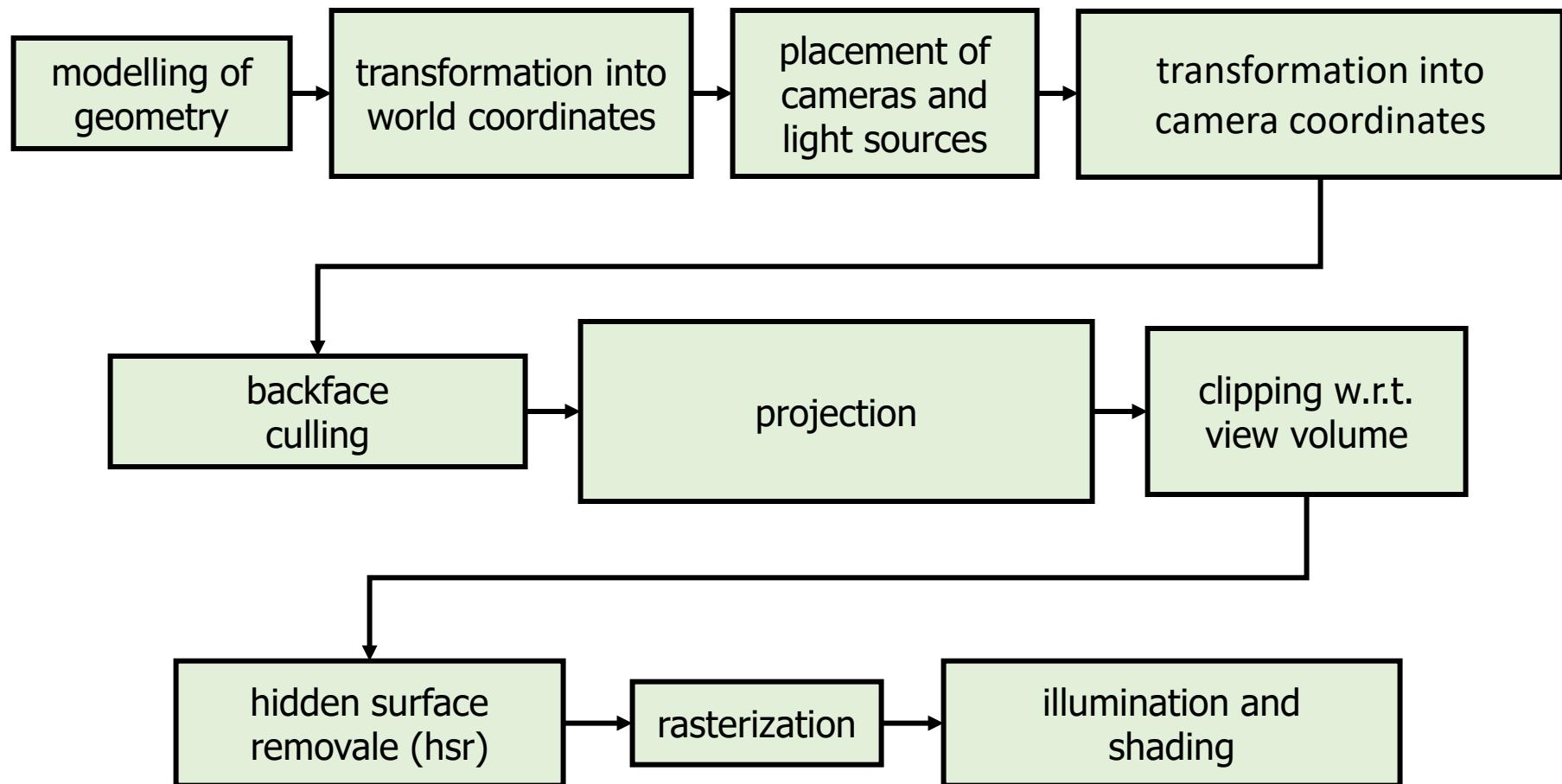
Goals

- By the end of today's class, you will be equipped to:
 - Describe the algorithms used to eliminate hidden surfaces

Rendering Process

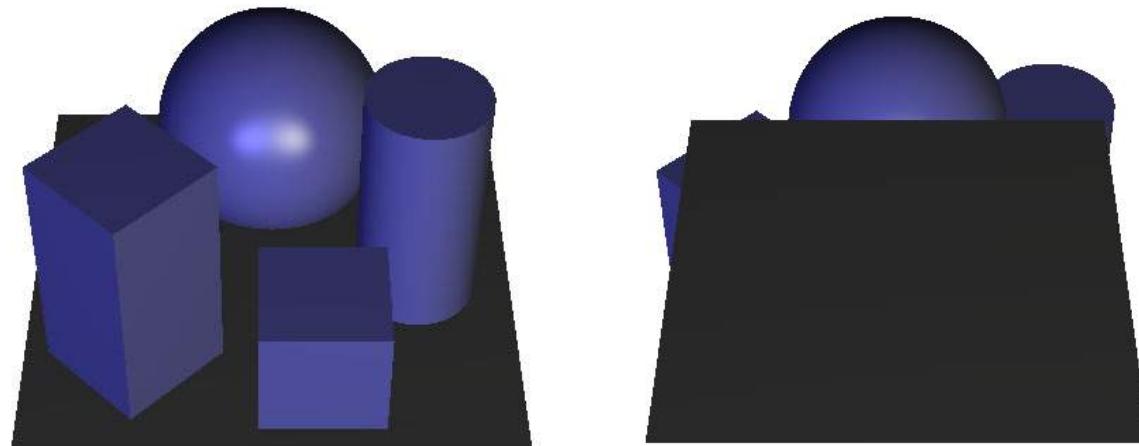
- There are three standard steps in any rendering process:
 - Viewing and projection
 - Hidden surface removal
 - Determining surface colour
- We will look at all three of these processes in some detail
- Apply to photo- and non-photorealistic rendering

Rendering Pipeline



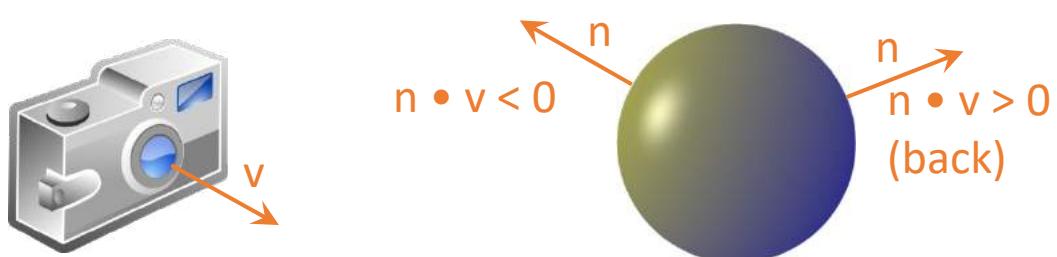
Hidden Surface Removal: Motivation

- model parts independently processed by rendering pipeline: show front parts only
- avoid unnecessary processing



Back Face Culling

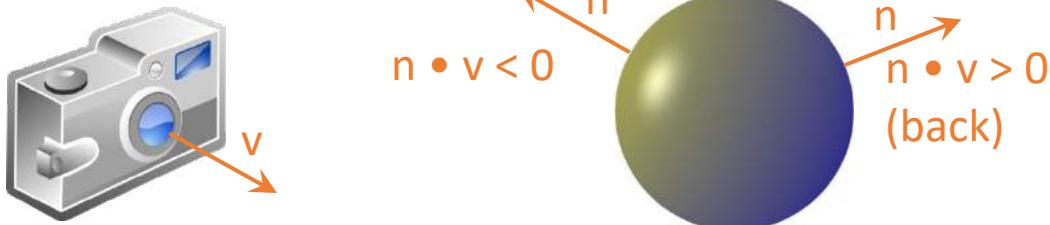
- back faces (usually) not visible
- reduction of computation *for closed objects*
- removal early in the pipeline
- reduction of polygon count by approx.
 $\frac{1}{2}$ of the total polygon number
- computation: discard polygons where dot product of surface normal with view direction > 0



Back Face Culling

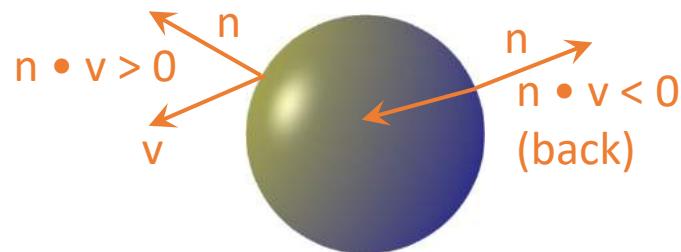


- The sign to test for depends on how you define \vec{v}
- \vec{v} is the look direction (i.e. look at point – eye)
- Back faces have dot product > 0



Back Face Culling

- The sign to test for depends on how you define \vec{v}
- \vec{v} is from the polygon to the eye
- Back faces have dot product < 0



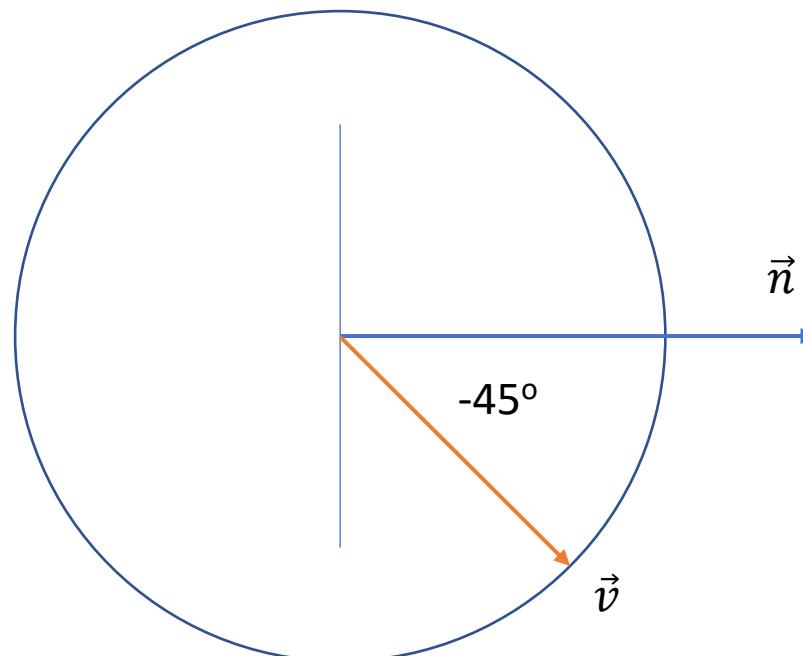
Why dot product works...

- Recall: dot product involves cosine

$$\mathbf{A} \cdot \mathbf{B} = \|\mathbf{A}\| \|\mathbf{B}\| \cos \theta$$

- Cosine is +ve from -90° to 90°

Let's translate v and n to the origin to compare the angles...



Cosine is > 0 – looking at the back

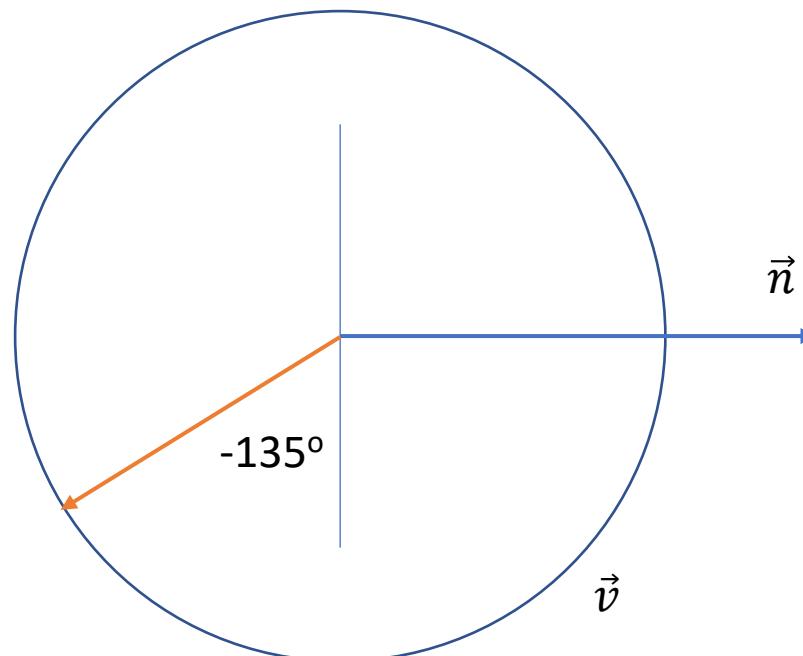
Why dot product works...

- Recall: dot product involves cosine

$$\mathbf{A} \cdot \mathbf{B} = \|\mathbf{A}\| \|\mathbf{B}\| \cos \theta$$

- Cosine is +ve from -90° to 90°

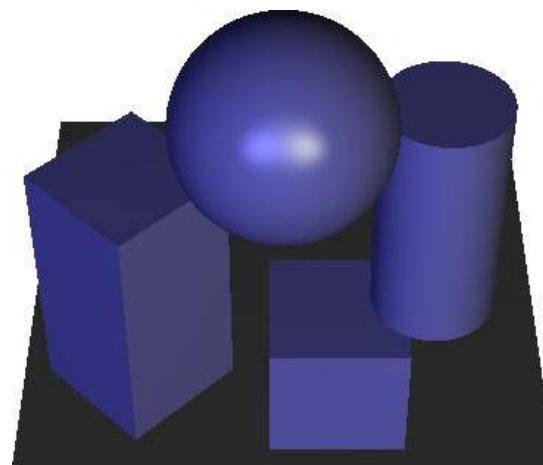
Let's translate v and n to the origin to compare the angles...



Cosine is < 0 – looking at the front

Back Face Culling

- back face culling as HSR technique?
 - (usually) not sufficient due to partial overlaps of several objects in the scene
 - would work reliably only if no overlaps occur
 - e.g., if only one convex object is shown
- order of depicted objects (overlapping) with only back face culling still depends on rendering sequence
→ idea for real HSR technique



Hidden Surface

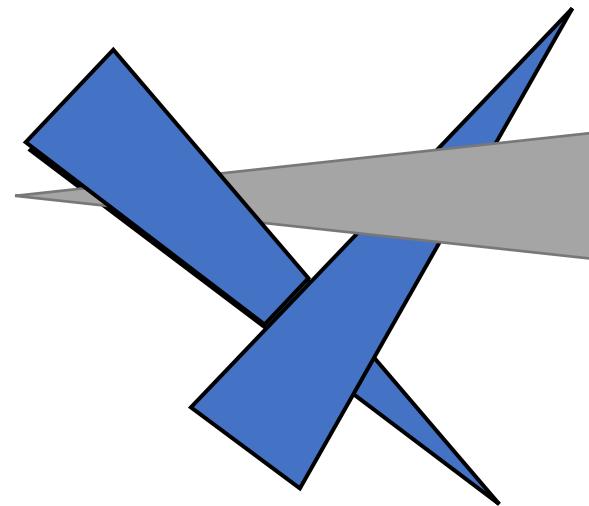
- Most hidden surface elimination is now done in hardware, with a few exceptions
- One of the main exceptions is high quality rendering for film and video, which sometimes still uses software algorithms
- There are three broad classes of algorithms that are in common use today:
 - Painter's algorithm
 - Z buffer algorithms
 - BSP Tree algorithms

Painter's Algorithm

Hidden Surface Algorithms

Painter's Algorithm

- hidden surface removal borrowed from van Gogh & co: draw scene from back to front
- sort scene's triangles from back to front
- start rendering triangles from the back
- problems:
 - cyclic overlaps
 - performance:
sorting is $O(n \log n)$
 - wasted drawing of invisible items



Z-Buffer Algorithms

Hidden Surface Algorithms

Z-Buffer

- The z-buffer or **depth buffer algorithm** is the one used in hardware
- Once we have finished all of our viewing transformations we have an (x,y) position in screen space, **plus a z' value**
- This z' value should be **related to the distance from the viewer to the point**

Z-Buffer

- The projection of a 3D polygon is a 2D polygon, the vertices of the 2D polygon are the projections of the vertices of the 3D polygon
- There are efficient algorithms for computing the **pixels covered by a polygon**, in the process we can **interpolate the z values from the projected vertices**

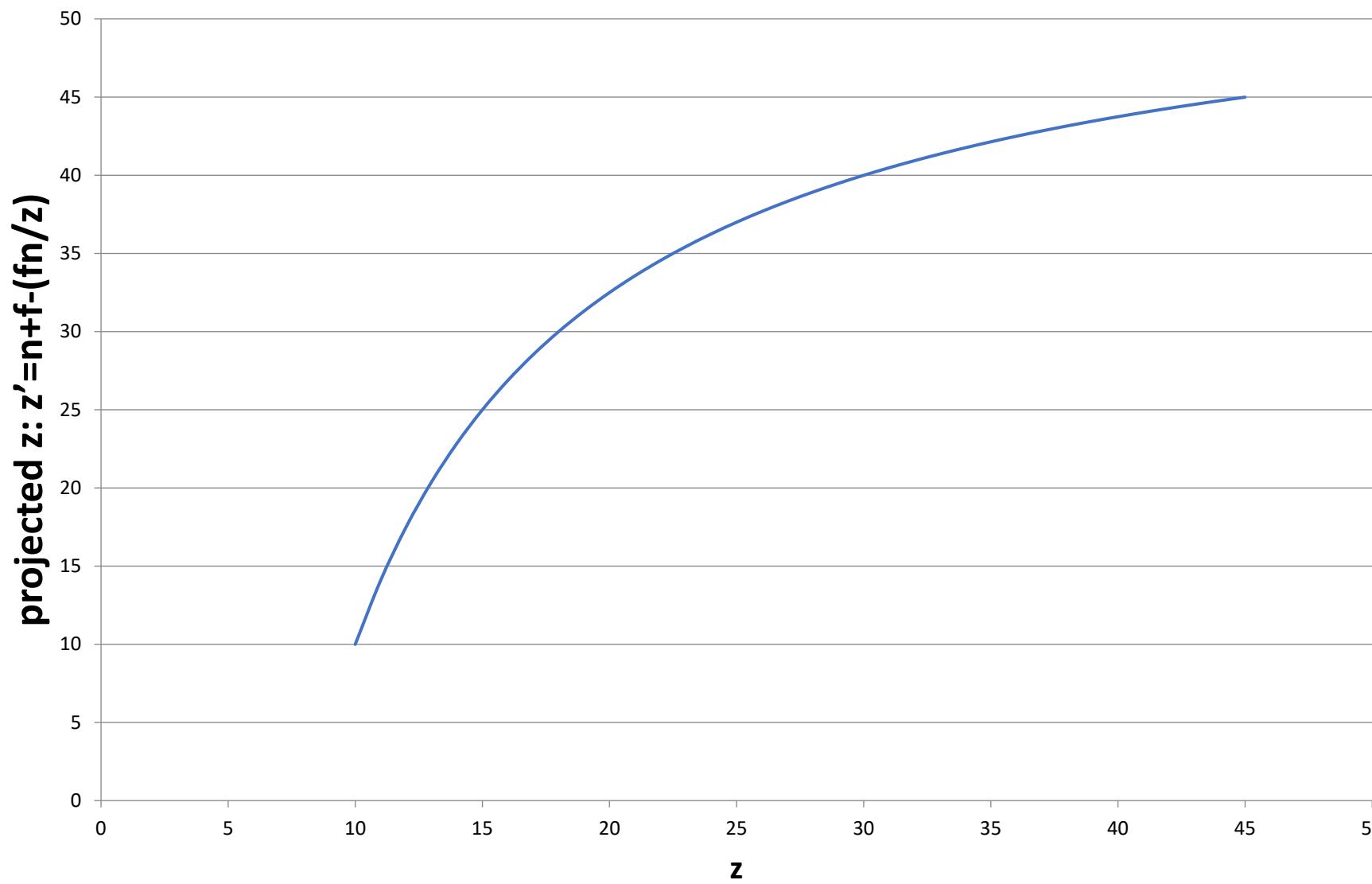
Z-Buffer

- Our projection gives us the following for the projected z' values:

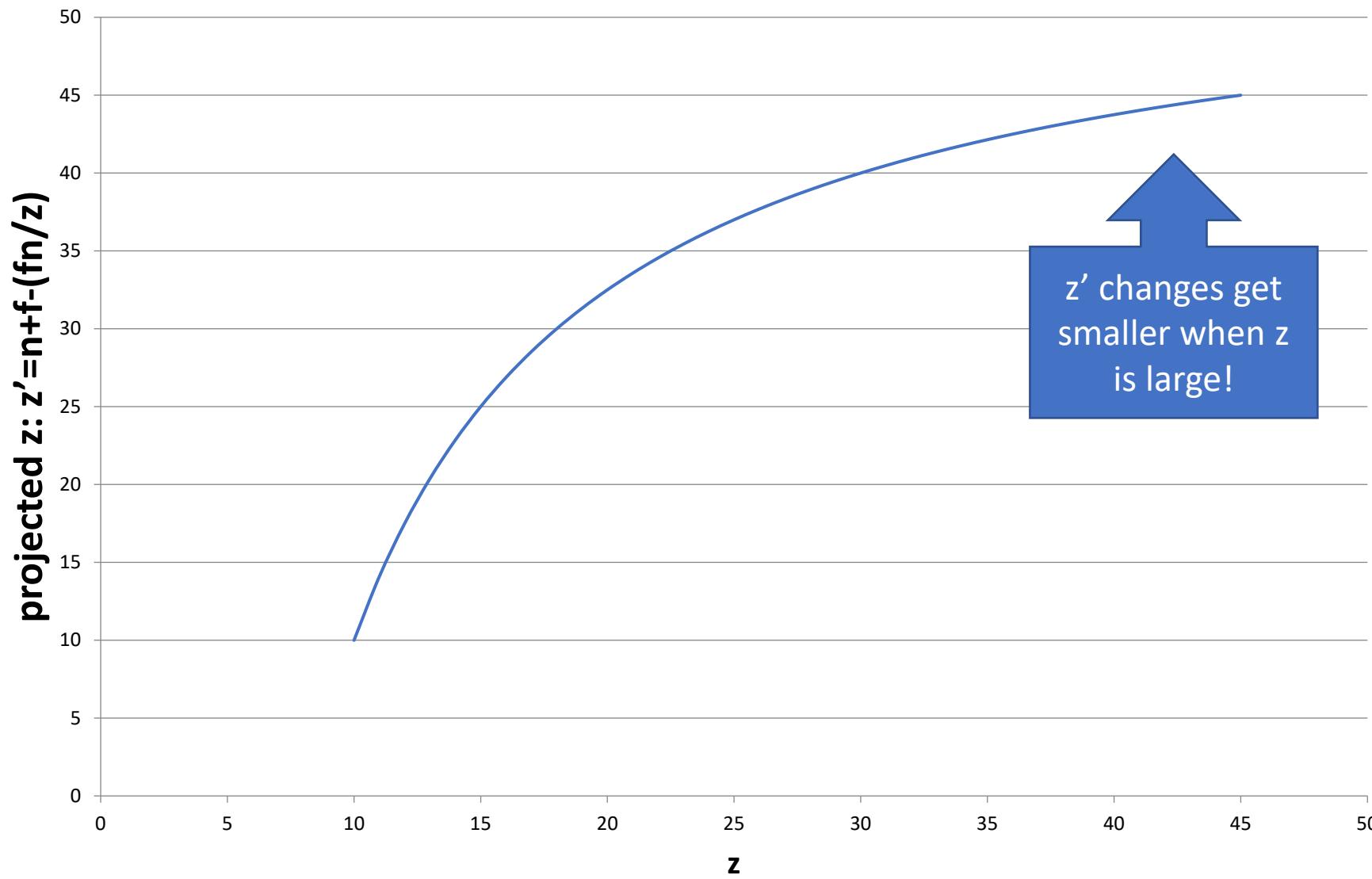
$$M_p \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \frac{n+f}{n} - f \\ \frac{z}{n} \end{bmatrix} = \begin{bmatrix} \frac{nx}{z} \\ \frac{ny}{z} \\ \frac{n+f-fn}{z} \\ 1 \end{bmatrix}$$

- This is a non-linear function of z for values of z that lie within the view volume (note: *monotonic from n to f*)
- Question: value at $z = n$? $z = f$? $z = 2f$?

Projected z for n=10, f=45



Projected z for n=10, f=45



Z-Buffer

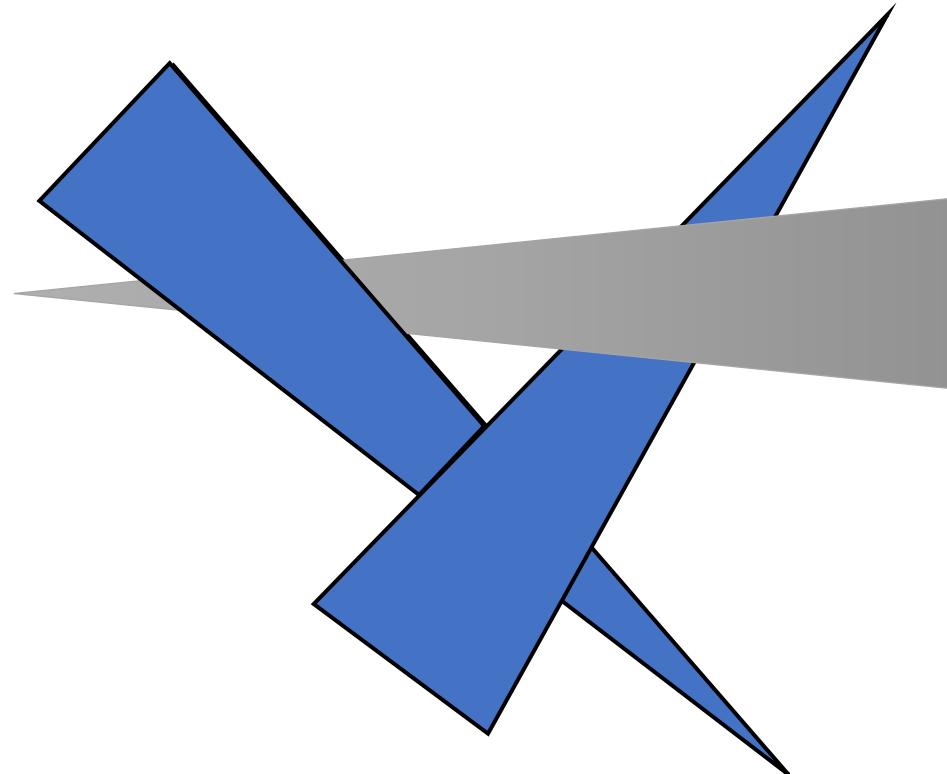
- Create a z-buffer that has the same resolution as the screen
- All positions are initialized to the furthest point: f , the far plane
- To write a pixel:
 - check the current z buffer value
 - if the new z' value is smaller than the one in the z buffer, write the pixel and update, otherwise ignore the pixel

Efficiency

- In producing an image we may write to a given pixel many times, but the one that remains at the end is the one closest to the viewer
- This algorithm isn't perfect, since it can waste time on polygons that aren't visible, but its simplicity and hardware implementation often makes up for the inefficiency

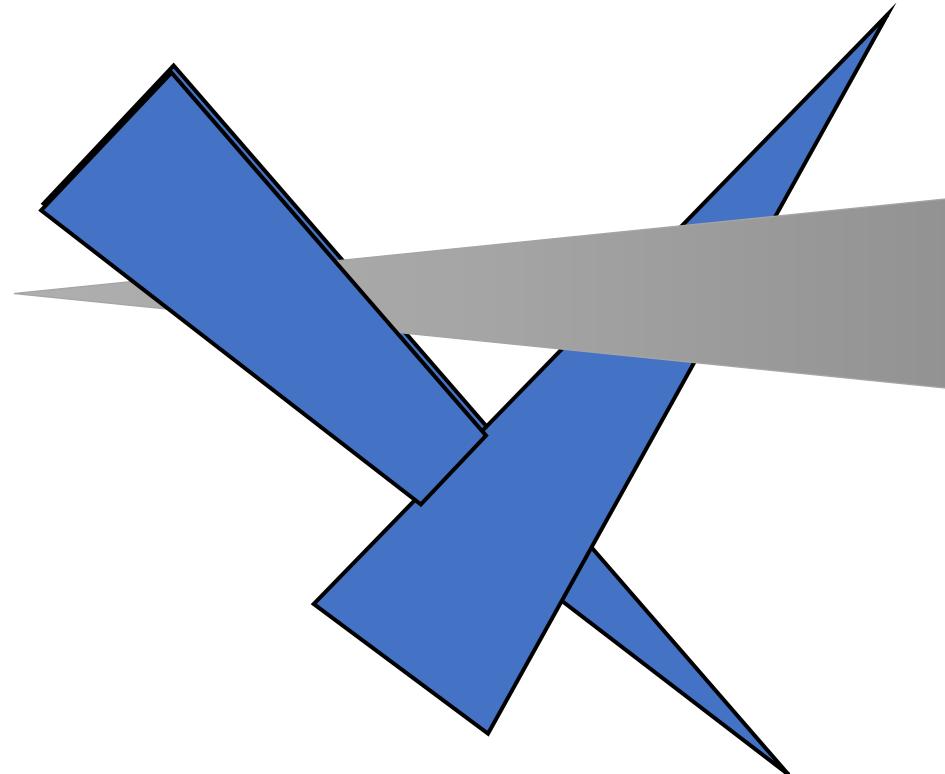
Comparison to Painter's Algorithm

- Cyclic overlaps?
- Performance?



Comparison to Painter's Algorithm

- Cyclic overlaps
 - Ok!
- Performance
 - No sorting
 - Memory required for z-buffer
 - Wasted drawing of hidden surfaces



Implementation Considerations

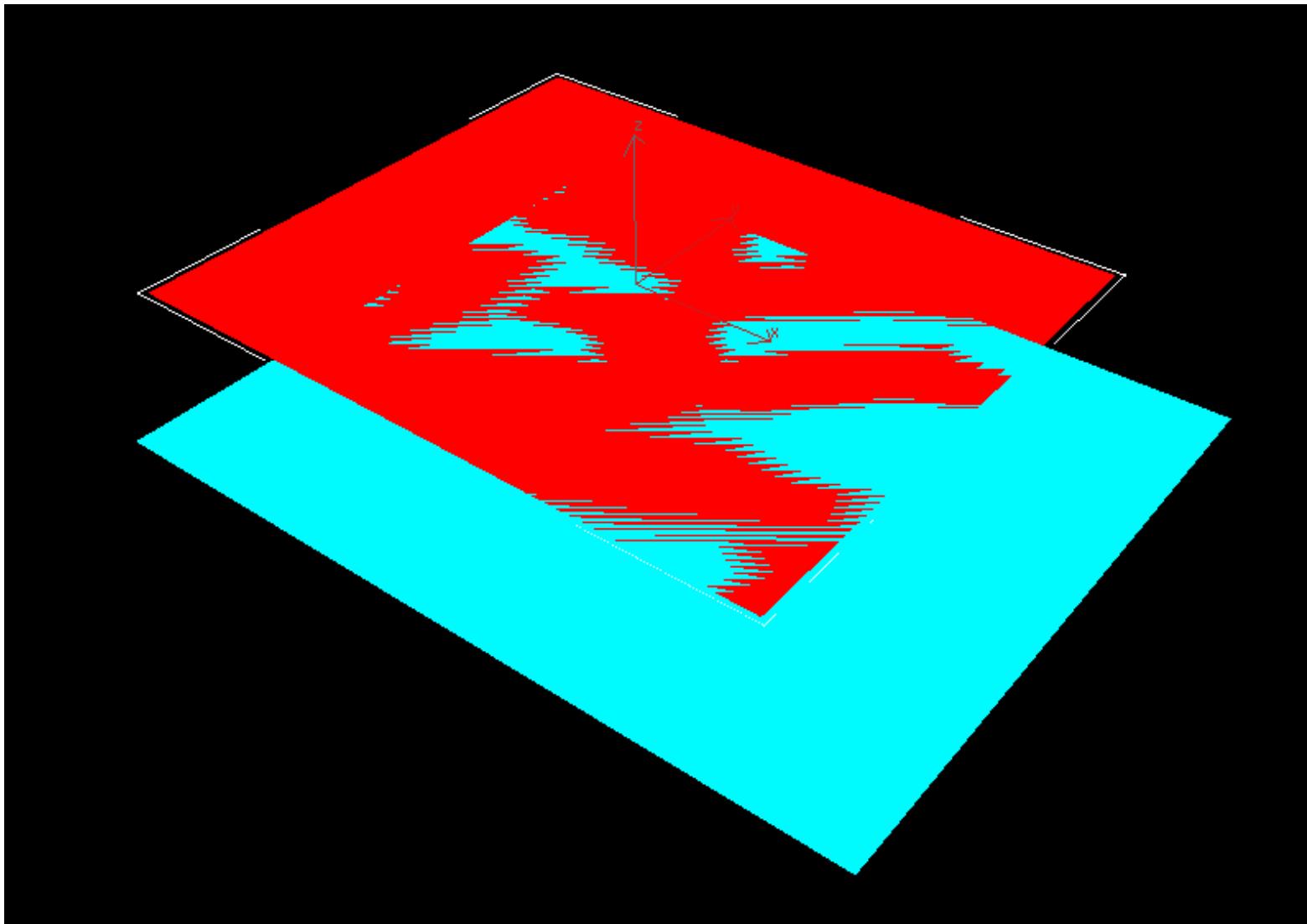
- Computers do not have infinite precision!
- Today, on most graphics displays the z buffer is made up of 32 bit or 64 bit integers
- We scale the z values so they fit within this range

Implementation Considerations

- Orthographic (parallel) projection:
 - z values will vary linearly from n to f
 - we have B possible values (which is usually 2^{32})
 - so we have **B bins of z values**, with each bin having the size: $\Delta z = (f-n)/B$
 - any z in a given bin **gets same z-buffer value**

Implementation Considerations

- Example:
 - 32 bit buffer: 4,294,967,296 bins
 - F=5000, n=1
 - $\Delta z = (f-n)/B = 1.1 \times 10^{-6}$
 - So, 5.000001 and 5.000002 are in the same bin
 - Will not be distinguished by the algorithm!
 - This may seem small, but small differences can be introduced by the viewing transformations
 - Especially problematic for co-planar polygons

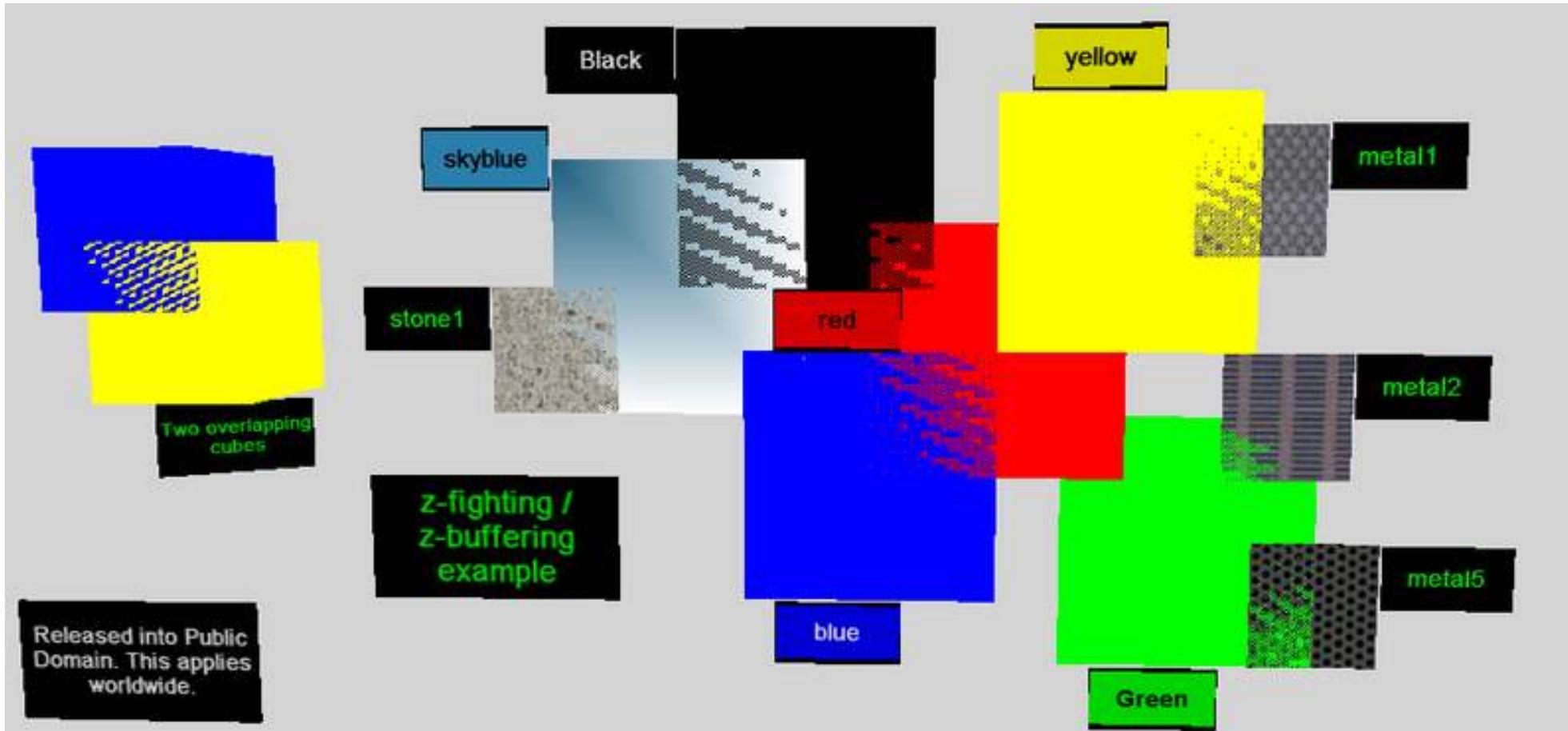


<http://en.wikipedia.org/wiki/File:Z-fighting.png>

Implementation Considerations

- As long as all the polygons are more than Δz apart there will be no problems
- If they are closer, the z buffer will think they have the same z value, so it won't be able to determine the correct pixel
- If we are lucky, it will always choose the pixels from the same polygon
- In practice we are usually unlucky, and we will get a mixture of the two polygons

“Z-fighting”



Avoiding Z-fighting

- To avoid this problem we need to carefully select n and f to give the smallest possible Δz
 - Maximize precision in the depth range we will use
- Many beginning graphics programmers set f to a large value, just because they think its safe...
 - Why is this a problem?
- The problem is **more serious with the perspective projection**

Z-Fighting in the Perspective Projection

- $z_buffer_value = (1 << N) * (a + b / z_w)$
 - N = number of bits of Z precision
 - $a = f / (f-n)$
 - $b = fn / (n-f)$
 - z_w = distance from the eye to the object
- Consequence:
 - For smaller z_w , more precision (closer objects have more detail)
 - As z_w increases, z_buffer_value loses resolution

Z-Fighting in the Perspective Projection

- From before we know that the projected z' value is related to z_w (the world z value) in the following way:
 - $z' = n + f - fn/z_w$
- Which gives the following relationship upon differentiation:
 - $\Delta z \cong fn\Delta z_w/z_w^2$
 - $\Delta z_w \cong z_w^2\Delta z/(fn)$
 - Where Δz_w is the bin size in world coordinates

Never set n=0

- In model or world space the bins don't have a uniform size and become larger as z_w becomes larger
- The worst case occurs at $z_w=f$ where the bin size becomes
 - $\Delta z_w \cong \frac{z_w^2 \Delta z}{fn} = \frac{f \Delta z}{n}$ at $z_w=f$
 - Note that this implies that we should never set n=0, otherwise the bins will become extremely large (imprecise) as we move away from the eye

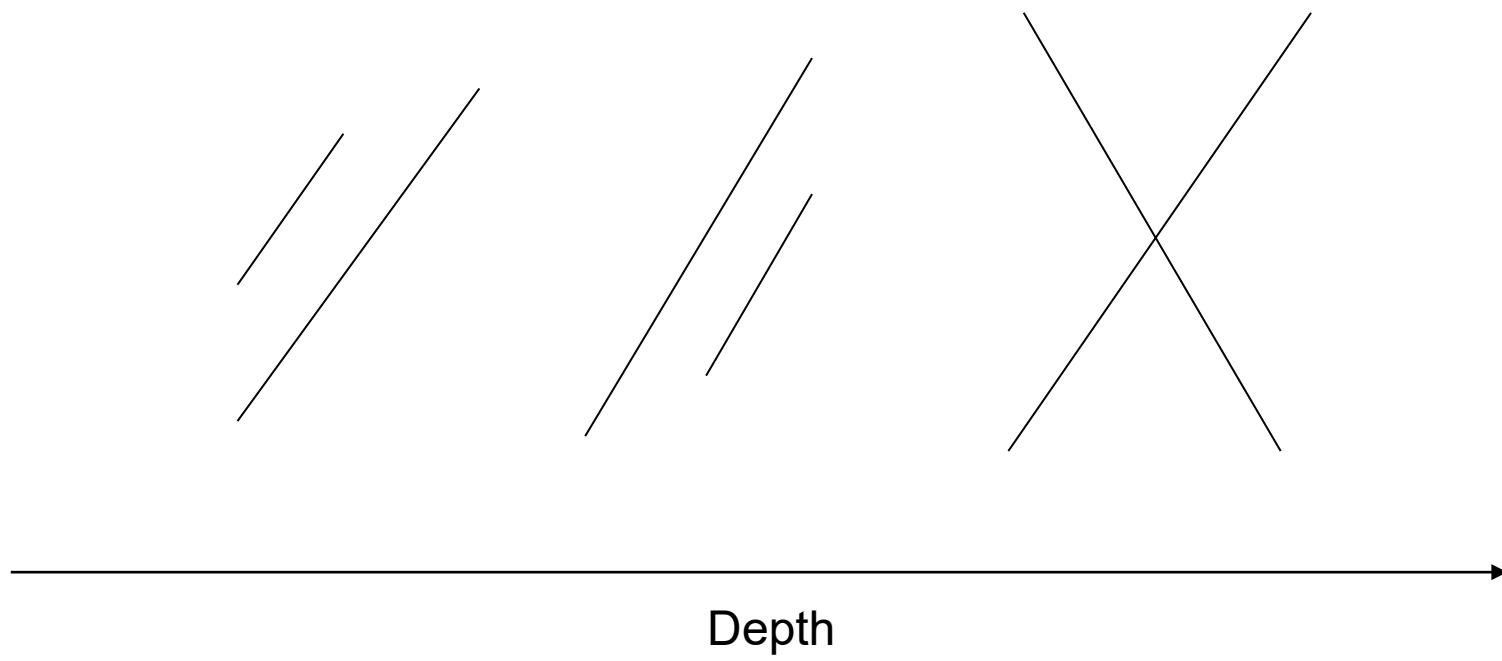
Z-Buffer in the Perspective Projection

- Bins become larger further away from the eye
- Objects further away are less important, so this is natural... but can still produce an ugly image!
- High precision close to the near plane may be wasting precision if this depth range is not used
- Floating point z buffers are beginning to appear on modern high end graphics cards, which don't suffer as much from this problem

BSP-Tree

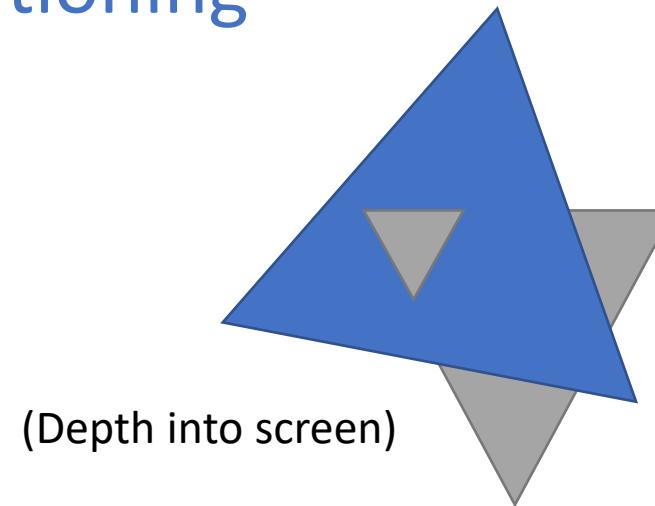
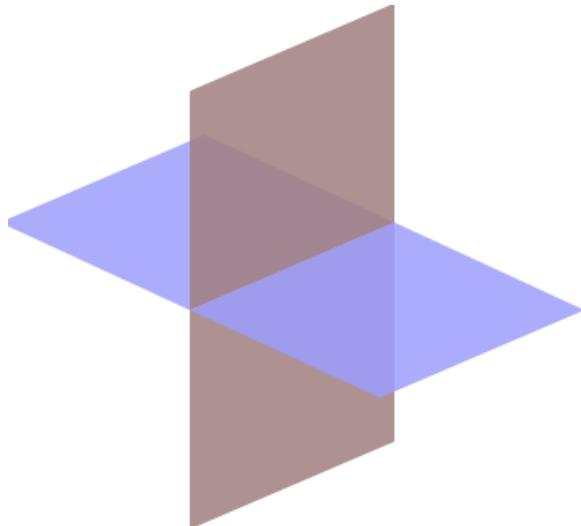
Hidden Surface Algorithms

Sorting Line Segments



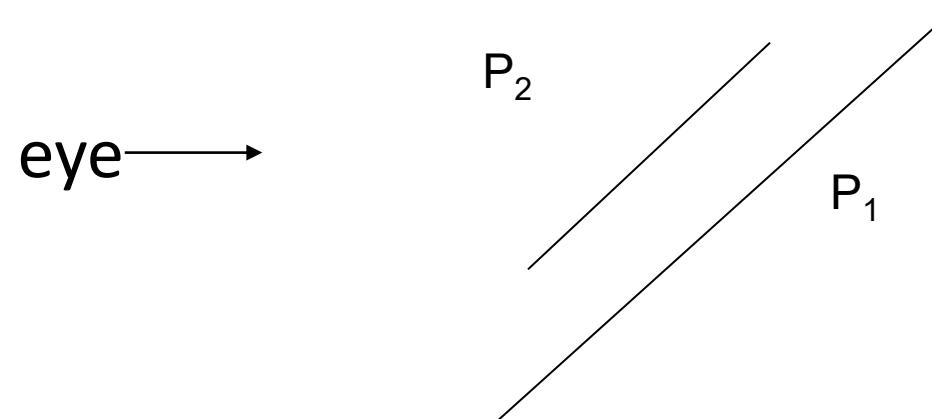
BSP Tree

- No way to sort the polygons without splitting
- Solve the problem with a new data structure that works for all eye positions: **the BSP Tree**
- **BSP = Binary Space Partitioning**



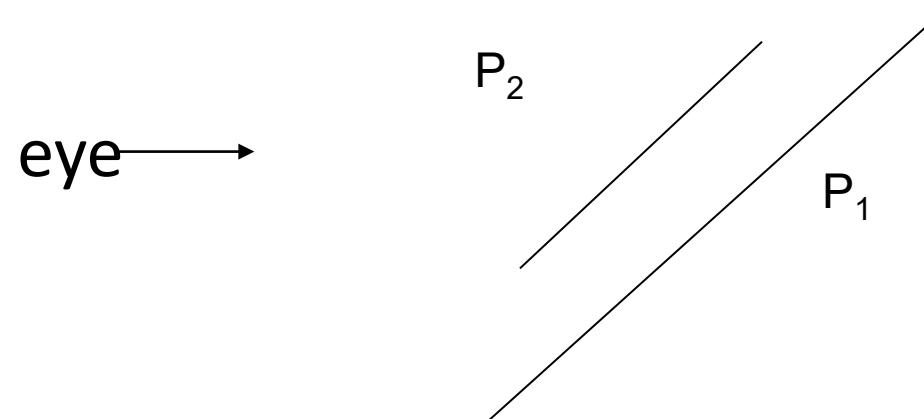
BSP Algorithm

- We will start by assuming that the polygons don't intersect, or cross in the depth direction, this gives us the situation below:



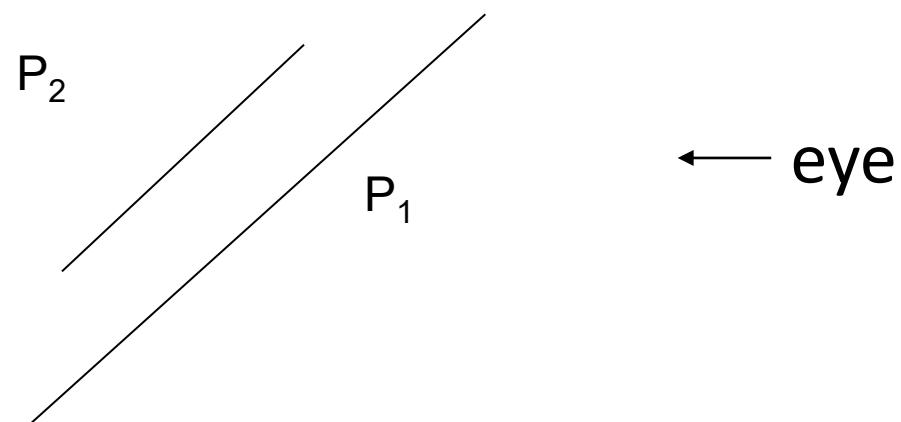
BSP Algorithm

- If the eye is on the left we want to draw P1 first followed by P2:



BSP Algorithm

- If the eye is on the right we want to draw P2 first followed by P1:



BSP Algorithm

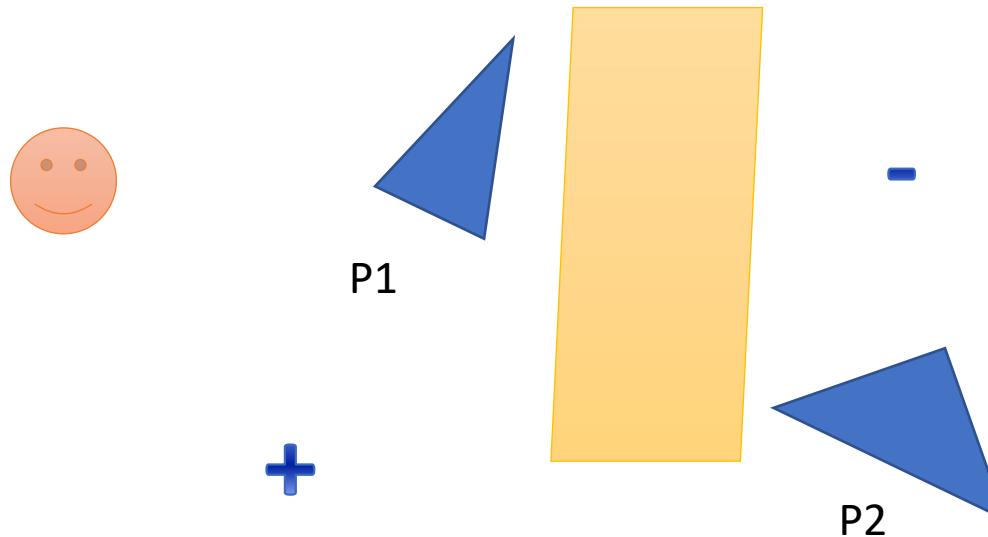
- Algorithmic Intuition:
 - Each polygon lies in a plane, and this plane has an equation $f(x,y,z) = 0$
 - this equation can partition the space of the plane into two parts: $f(x,y,z) > 0$ and $f(x,y,z) < 0$

BSP Algorithm

- One part will be on the plus side of the polygon, and the other part will be on the negative side of the polygon
- If the **eye position** is on the plus side, we should draw the polygons on the negative side first, followed by the current polygon, and then all the polygons on the plus side
- ... a lot like a binary tree!

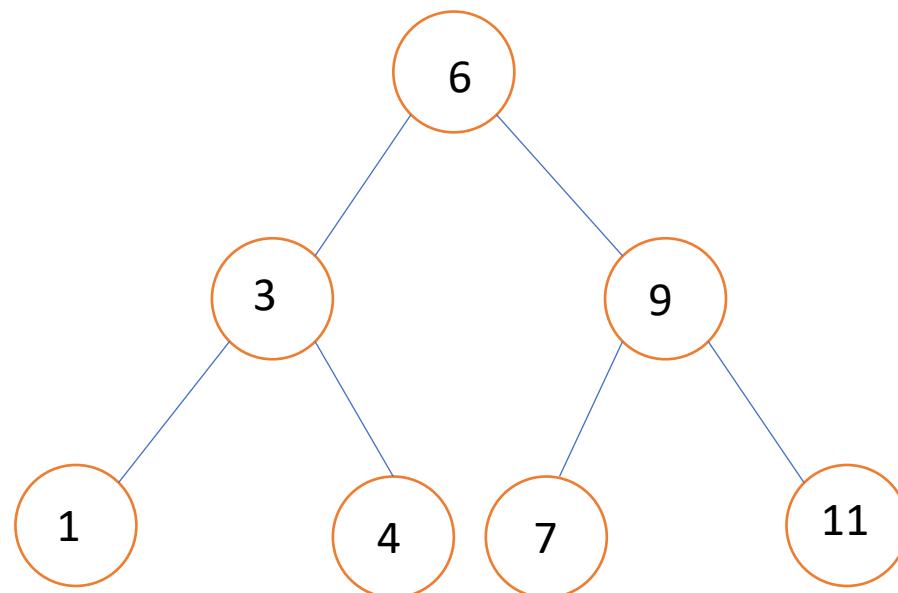
Example

- Plane equation: $x + 2y + 2z - 2 = 0$
- Eye position: $(1, 0, 1)$, $f(x, y, z) = 1 > 0$
 - Draw polygons on negative side first (P_2, P_1)



BSP Algorithm

- In a binary search tree we use the < operation on numbers (or letters) to order the nodes



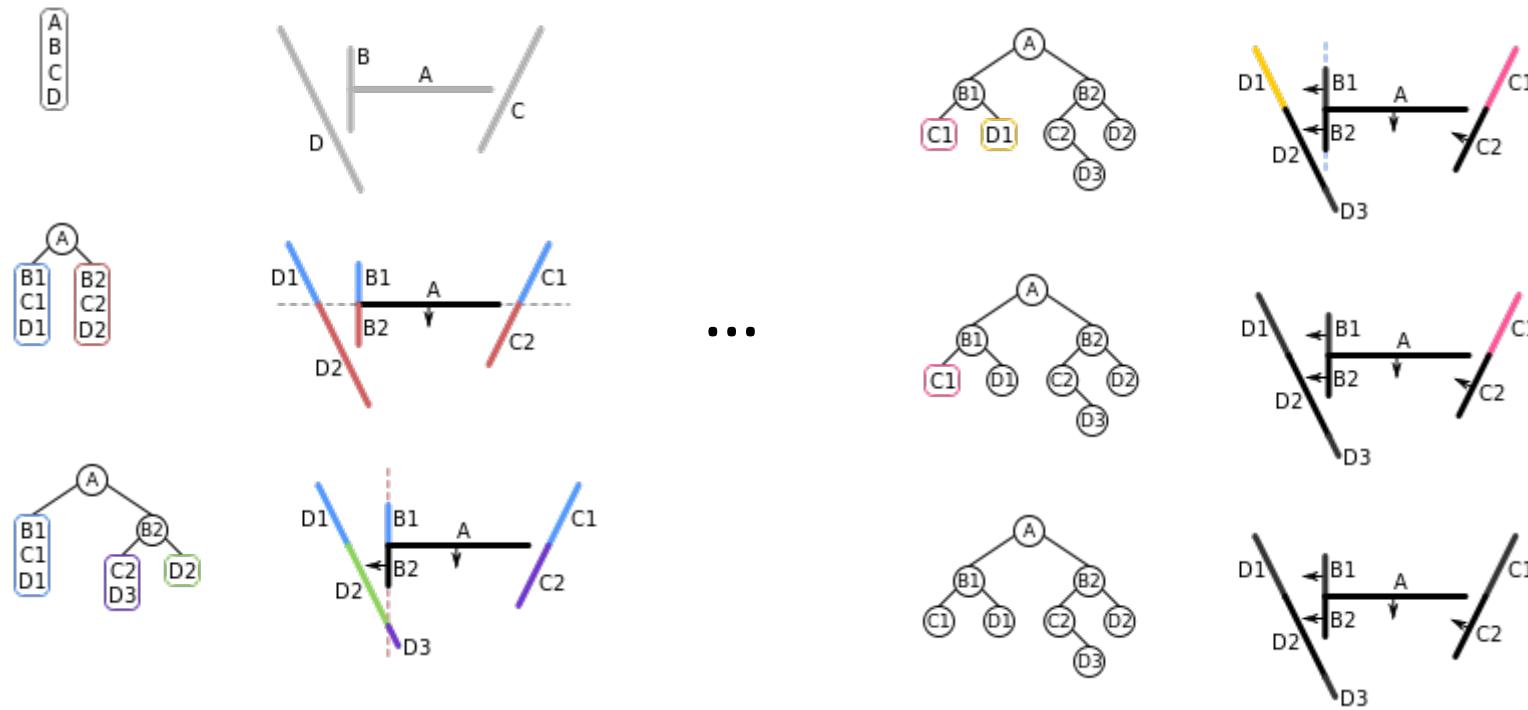
BSP Algorithm

- In the case of a BSP tree we use the plane equation to order the nodes
- Each node of the tree has a polygon and its (implicit) plane equation
- The plus branch for a node contains all the polygons on its plus side, and the negative branch contains all the polygons on its negative side

No need for Z-Buffer

- BSP can split up complex polygons into smaller pieces which can be rendered with the painter's algorithm
- No need for z-buffer
- But number of polygons may increase dramatically
- *Doom* was an early 3D computer game that used BSP trees, before most graphics cards had z buffers

Intuition: Example Partitioning



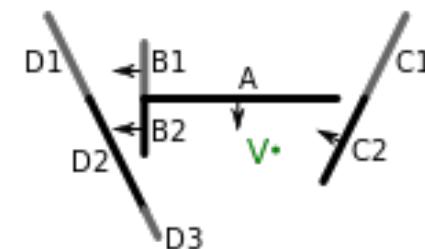
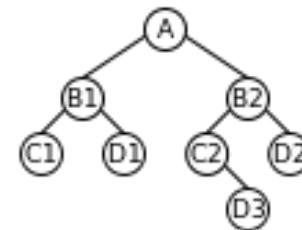
- Order of and position of partitioning line/plane is important. Many algorithms for this...

Example from: http://en.wikipedia.org/wiki/Binary_space_partitioning

Intuition: Traversing the BSP Tree

- Viewing point is in front of A
 - Draw items behind A (B1, C1, D1)
 - Draw A
 - Draw items in front of A (B2, C2, D3, D3)

(recursive at each level)



Example from: http://en.wikipedia.org/wiki/Binary_space_partitioning

BSP Data Structure

- So our data structure looks like this:

```
Struct BSP_tree {  
    polygon      poly;  
    plane       plane;  
    BSP_tree *plus;  
    BSP_tree *minus;  
};
```

BSP Tree Traversal Algorithm

```
draw(BSP_tree tree, point eye) {  
    if(tree.empty())  
        return;  
    if(tree.plane(eye) < 0) {  
        draw(tree.plus,eye);  
        draw poly;  
        draw(tree_MINUS,eye);  
    } else {  
        draw(tree_MINUS,eye);  
        draw poly;  
        draw(tree_PLUS,eye);  
    }  
}
```

BSP Advantages

- Note that once we have the BSP tree we can draw it correctly, no matter where the viewer is located, without recalculating the tree
- One of the advantages of BSP trees is they don't rely on any graphics hardware, they can be used with any graphics display
- They are also useful for other types of geometrical algorithms

Constructing BSP Trees

- BSP trees are easy to display, but how do we build one?
- This is where things become difficult, there is no efficient algorithm for constructing an optimal BSP tree
- Fortunately, there are algorithms that construct nearly optimal trees in a reasonable length of time

BSP Tree Construction

- Assumption: polygons do not intersect (their planes don't intersect)
- Choose one of the polygons to be the root of the tree
- We then add the rest of the polygons to the tree one by one, using the plane equations as the ordering operation

BSP Tree Construction

- We test all the vertices of the polygon in the plane equation at the current node
- If they are all positive, the polygon is inserted in the plus subtree, otherwise the polygon is inserted into the minus subtree
- Remember the assumption:
 - Since the polygons don't intersect they will either be all positive or all negative

BSP Tree Construction

- build(pl)
 - pl is the list of polygons
 - Uses choose(pl) to get poly, the next polygon to insert, and remove poly from pl
- insert(node, poly)
 - node is the insertion point
 - poly is the polygon to be inserted

BSP Tree Construction

```
build(polygon_list pl) {  
    BSP_tree root;  
    polygon poly;  
  
    poly = choose(pl);  
    root.set(poly);  
    while(pl not empty) {  
        poly = choose(pl);  
        insert(root,poly);  
    }  
}
```

BSP Tree Construction

```
insert(BSP_tree node, polygon poly) {
    if(node.plane(poly) < 0) {
        if(node.minus is empty)
            node.minus.set(poly);
        else
            insert(node.minus,poly);
    } else if(node.plane(poly) > 0) {
        if(node.plus is empty)
            node.plus.set(poly);
        else
            insert(node.plus,poly);
    }
}
```

Removing the Assumption

- When some of the vertices lie on one side of the plane, and the rest of the vertices lie on the other we need to split the polygon
- Some of the pieces go in one subtree and the rest of the pieces go in the other subtree

Splitting

- For simple polygons the split will result in two smaller polygons, but this is not always the case
- We will assume we get just two parts, plus_side and minus_side, but remember both of these could be a list

Making insert() general

- We need to add one more case to insert, this will occur when `node.plane(poly) == 0`
 - `split()` uses the plane of the polygon at the current node to split the polygon that we are adding
- The pieces are then added recursively to the BSP tree
- If the poly is in the plane (`node.plane(poly)==0` for all vertices) just put it in the plus or minus side, doesn't matter.

Splitting

```
insert(BSP_tree node, polygon poly) {
    polygon plus_side;
    polygon minus_side;
    ...
} else {
    split(node.plane, plus_side, minus_side);
    if(node.plus is empty)
        node.plus.set(plus_side);
    else
        insert(node.plus, plus_side);
    if(node.minus is empty)
        node.minus.set(minus_side);
    else
        insert(node.minus, minus_side);
}
```

Efficiency

- When we display a BSP tree we visit all of the nodes, so the shape of the tree isn't important
- Efficiency is determined by the overall size of the tree, that is the number of nodes
- Each time we split a polygon, we increase the number of nodes, so we want to minimize splits
- Our choice of polygon for the root, and the order that we add the other polygons determines the number of splits

Efficiency

- Thus, the choose() procedure determines the efficiency of the algorithm
- Each time we call choose() we want to select the polygon that splits the fewest of the polygons that remain on the list
- This is very inefficient, since we would need to check each polygon on the list and compare it against all of the others

Efficiency

- Two techniques have been used to make choose() more efficient
- The first technique is to just randomly choose a polygon from the list
- This does a reasonable job most of the time, but it can produce some very bad trees
- The main advantage of this approach is that it is very fast

Efficiency

- Since the tree is constructed once, we can take a bit more time
- A better technique is to randomly select a small number of polygons from the list, and then choose the one that splits the fewest polygons
- Experiments have shown that selecting just 5 polygons at random produces a near optimal BSP tree

Summary

- Algorithms for hidden surface removal:
 - Back-face culling
 - Z-Buffer
 - Painter's Algorithm
 - BSP trees

Next Class

- Lighting and illumination models

CSCI 3090

Lighting Models

Mark Green

Faculty of Science

Ontario Tech

Rendering Process

- There are three standard steps in any rendering process:
 - Viewing and projection
 - Hidden surface removal
 - Determining surface colour
- We will look at all three of these processes in some detail
- Apply to photo- and non-photorealistic rendering

Goals

- By the end of today's class, you will be:
 - Understand the difference between local and global illumination
 - Describe the standard local illumination model
 - Relate the lighting models to physical reality

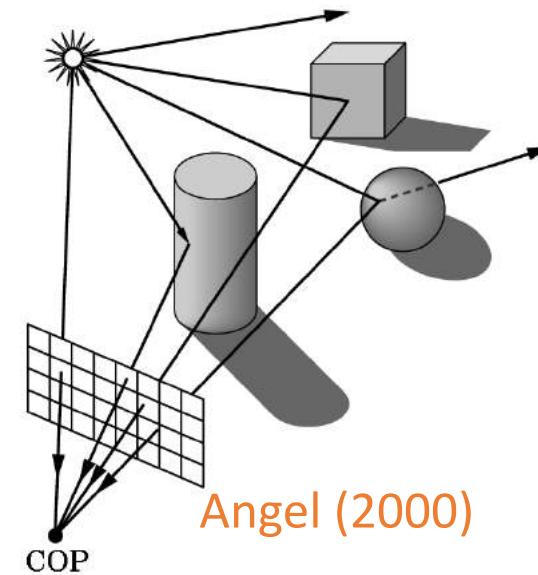
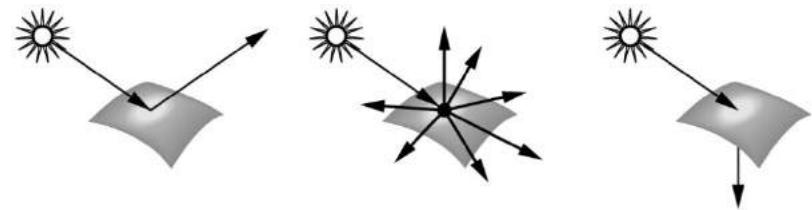
Lighting Overview

- Illumination models
 - light sources
 - light interaction with surfaces
 - Phong illumination model
- Shading
 - application of illumination models to rendering polygons, pixel-by-pixel
 - efficiency vs. quality
 - flat, Gouraud, Phong shading

Illumination Models

Real World

- surfaces emit, absorb, reflect, and scatter light
- light intensity and color dependent on surface position and orientation w.r.t. the light source
- light is usually reflected or refracted several times
- usually several sources of light
- final intensity/color at a point is sum of several light paths ending at that point



Surface Colour

- The model presented here is semi-physical in the sense that it is motivated by physics and empirical observations, but **it is not physically accurate**
- We will model **simple light reflection** and the interaction between the colour of the light and the colour of the object
 - We are not considering emissive objects right now

Illumination Models

- Mathematical description necessary
- Leads to equation using integrals: the rendering equation
 - it's usually not solvable analytically
 - we need numerical solutions and approximation!
 - simplifying light sources
 - simplifying materials
 - simplifying computation
 - speed-up

Illumination Models

- Description of the factors that influence the color and light intensity at a point
- Local illumination models:
 - only pixel-light interaction, pipeline approach, local information
 - heuristic approximation, simpler than global
 - Phong illumination model & polygonal shading
- Global illumination models:
 - consider all objects in a scene when computing light at a specific point
 - e.g., radiosity and raytracing

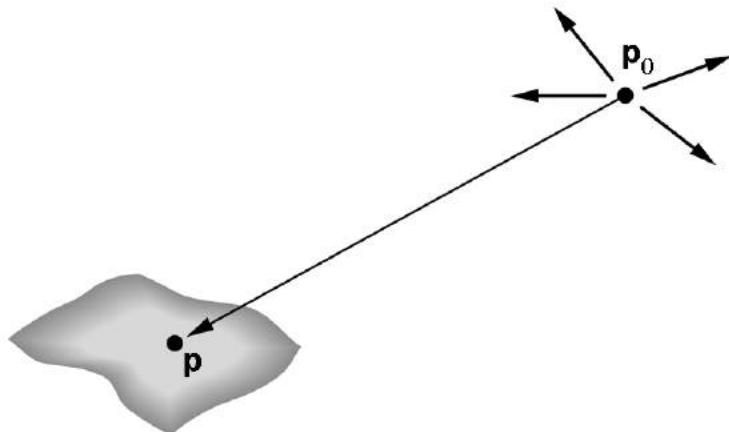
Local Illumination Model

- Our lighting model is made up of three components:
 - Ambient reflection
 - Diffuse reflection
 - Specular reflection

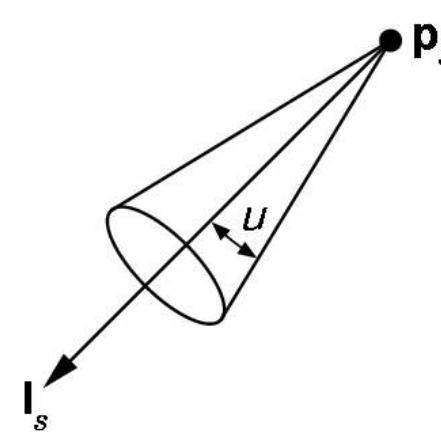
Light Sources in Computer Graphics

- point light source: has only position and no size, sends light equally in all directions
→ point in 3D & intensity
- spot light source: point light source sending out a cone of light with light intensity decreasing towards cone border
→ point + vector in 3D, angle, attenuation
- directional light source: sends directed, parallel light rays, has no position
→ vector in 3D & intensity

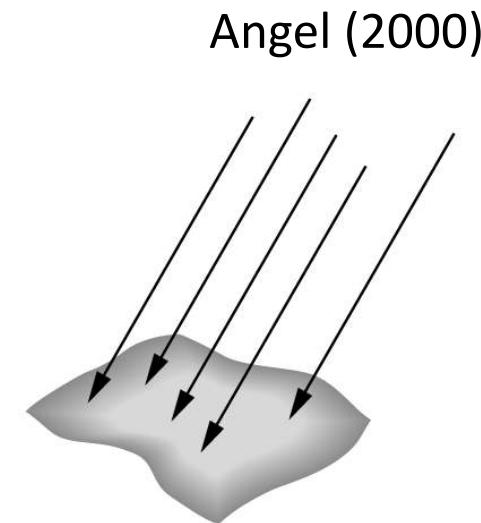
Light Sources in Computer Graphics



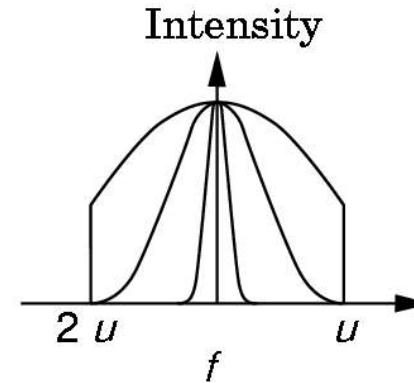
- point light
- spot light intensity defined by cosine function depending on angle from center ray, exponent



spot light



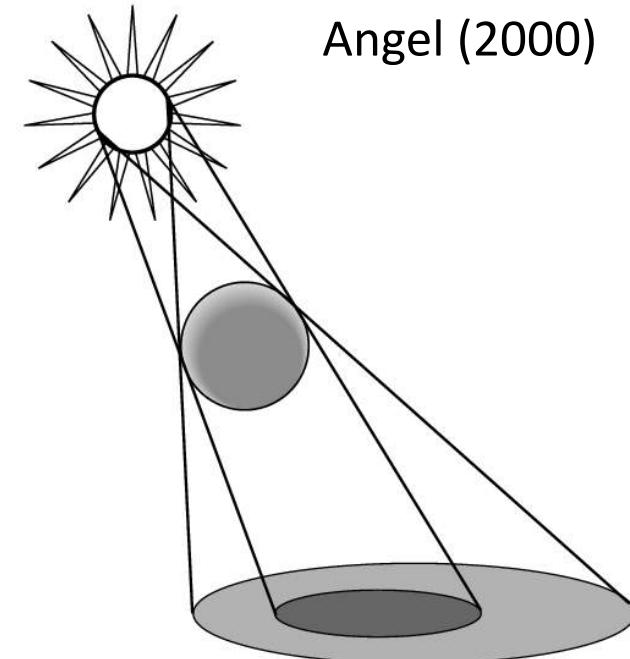
directional light



Angel (2000)

Light Sources in Computer Graphics

- results from simplification to point light sources?
 - sharp contrast between light and shade,
no gradual change (penumbra)
 - area light sources have
to be approximated by
several point lights to
have a penumbra



Light Sources in Computer Graphics

- light attenuation
 - light intensity reduces with growing distance
 - theoretically: $I \sim 1/d^2$
 - reality does not follow exactly this – why?
 - CG: $I \sim 1/(a+bd+cd^2)$, often only linear: $a,c = 0$
- light color
 - heuristic: color modeled using RGB values
 - approximation because light behavior depends on wave length – examples?

Phong Illumination Model (1973)

- most common CG model for illumination:

$$I_{Phong} = I_a + I_d + I_s \quad (\text{by Bùi Tường Phong})$$

- ambient light: base illumination of scene
 - simulates light scattering on objects
 - necessary because repeated diffuse reflection is not considered in local illumination model
 - depends on color of all objects in scene
 - should always be kept very small
- diffuse light: light from diffuse reflection
- specular light: light from mirror-like reflection

Ambient Light

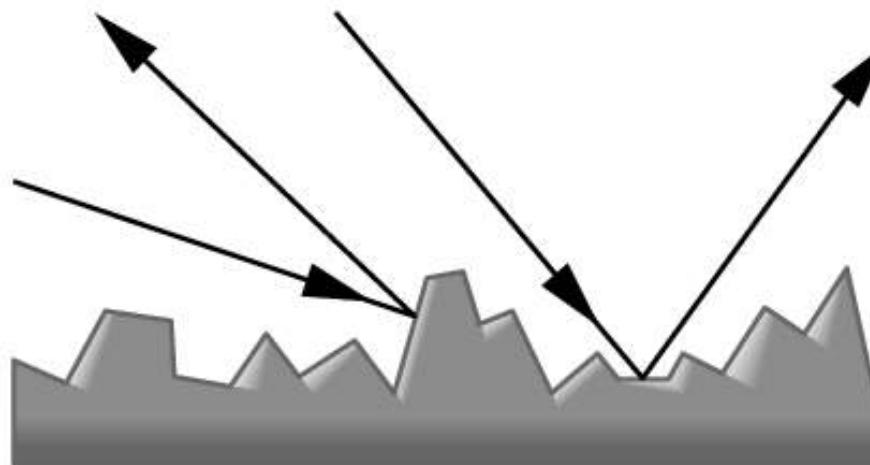
- Ambient light is the background light in the environment and it is modeled by a constant light level c_a
 - c_a is a vector that is made up of three components: red, green and blue
- Each object has a reflection coefficient, c_r , this is also a vector with red, green and blue components
- The amount of ambient reflection for an object is given by $I_a = c_a c_r$, where the multiplication is component-wise

Diffuse Reflection

- Diffuse reflection is uniform in all directions above the surface of the object
- The amount of diffuse reflection is proportional to the amount of light striking the object, and this can be approximated by [Lambert's cosine law](#)

Light Interaction with Surfaces

- diffuse reflection: equal reflection in all directions on rough surfaces, depends only on θ and not observer – examples?
- due to light scattering on rough surfaces on randomly oriented microscopic facets



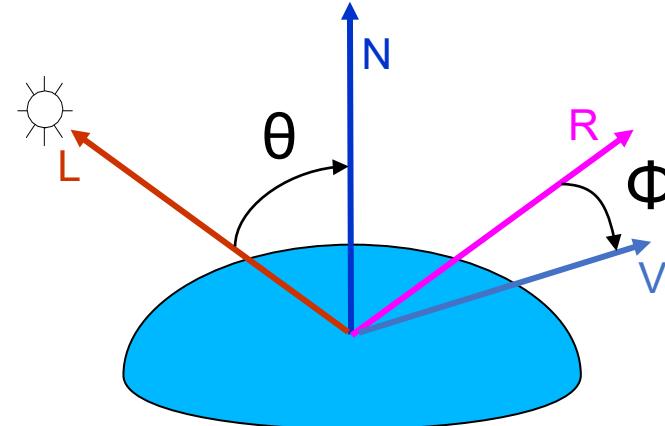
Angel (2000)

Diffuse Reflection

- Lambert's law states that the angle θ between L & N determines diffuse reflection:

$$I_d = c_l c_r \cos\theta = c_l c_r (L \cdot N)$$

- for normalized L, N
- $\theta = 0^\circ \rightarrow$ max. intensity
- $\theta = 90^\circ \rightarrow$ 0 intensity
- What are the constants of proportionality?



L – vector to light source

N – surface normal vector

R – reflected light ray

V – vector to viewer/observer

Diffuse Reflection

- Our empirical model states that the diffuse reflection is given by:
 - $I_d = c_l c_r (L \cdot N)$
 - c_l is the (rgb) colour of the diffuse light, c_r is the (rgb) reflective colour of the object
- What happens if the dot product is negative? We don't want to have negative light!

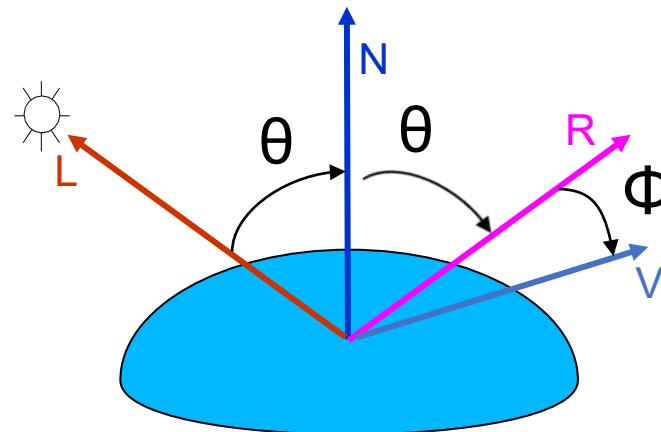
Diffuse Reflection

- If the dot product is negative, the **normal must be pointing away from the light source**
- That is, part of the object's surface must be between the light source and this point
- In this case the amount of reflected light should be zero, so our revised model is:
 - $I_d = c_l c_r \max(0, L \cdot N)$

Combining Ambient and Diffuse

- We can put together the two components that we have seen so far to get:
 - $I_{a+d} = c_r(c_a + c_l \max(0, L \cdot N))$
- This model produces objects that don't have highlights, they just have a flat matte appearance
- The next thing we need to do is add the highlights or specular reflection

Mirror Reflection



L – vector to light source

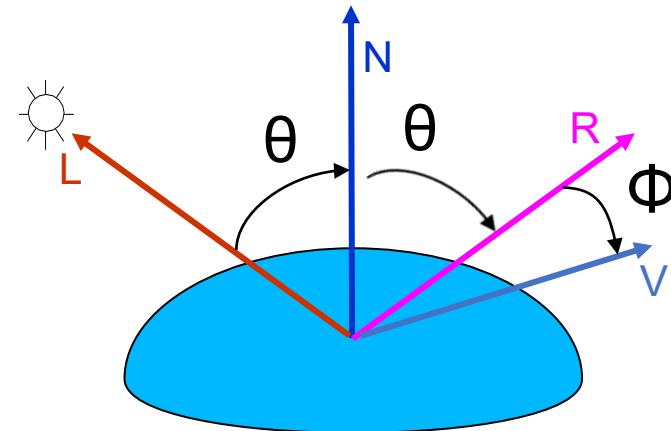
N – surface normal vector

R – reflected light ray

V – vector to viewer/observer

Specular Highlights

- pure mirror reflection would produce a infinitely small highlight
- angle Φ between R & V determines perceived brightness
- maximal reflection if $R = V$ ($\Phi = 0$)
- Attempt 1:
$$I_s = c_l(V \cdot R)$$
 - highlights are too large



L – vector to light source

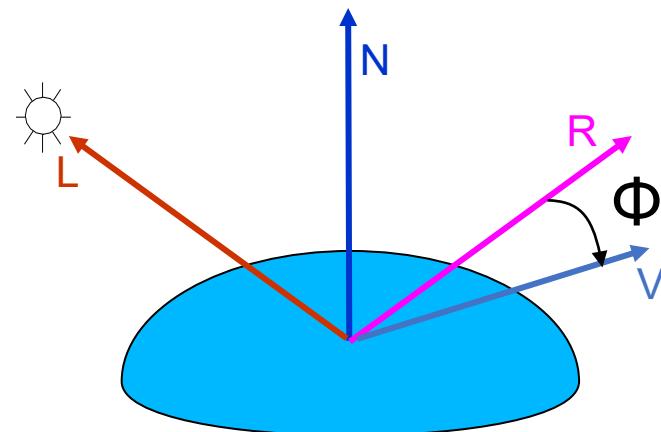
N – surface normal vector

R – reflected light ray

V – vector to viewer/observer

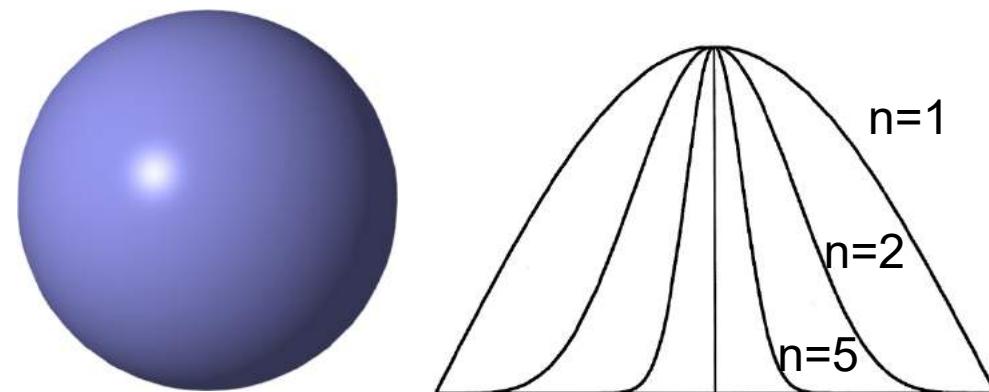
Light Interaction with Surfaces

- **directed reflection:** reflection only for small Φ : smooth surfaces – examples?
- physical reality
 - non-symmetric reflection around R
 - materials with **anisotropic** reflection
(reflection depends on angle Φ AND direction of L w.r.t. surface, e.g. velvet)

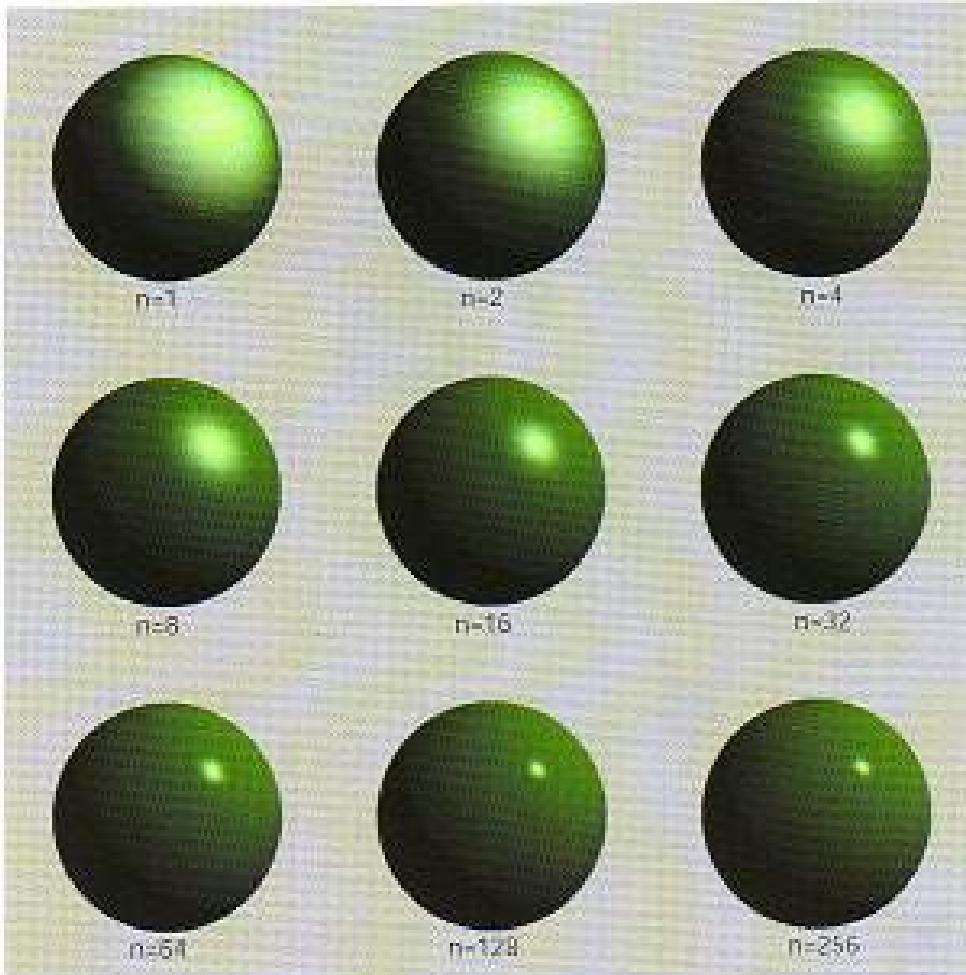


Specular Highlights

- Since $(V \cdot R)$ is less than 1, we can enhance this function's peak by taking it to a power, n , called the shininess of the surface
 - $I_s = c_l \max(0, V \cdot R)^n$
 - Max function avoids negative reflection
 - Metals: $n \approx 100$



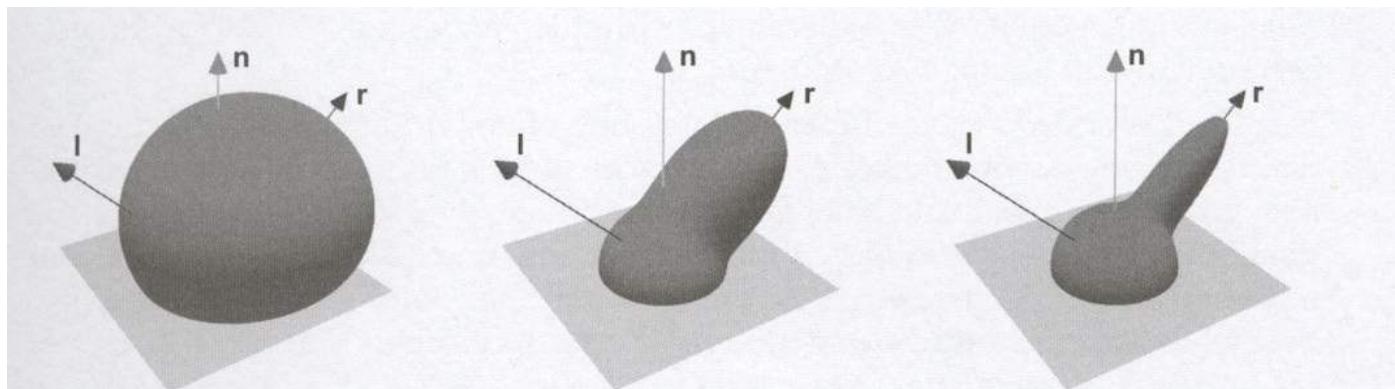
Specular Highlights



Specular reflection
for different values
of the shininess, n

Phong Illumination Model

- 3D reflection functions for different specular exponents: $n = 2, 20, 100$



Bender/Brill (2003)

Specular Reflection

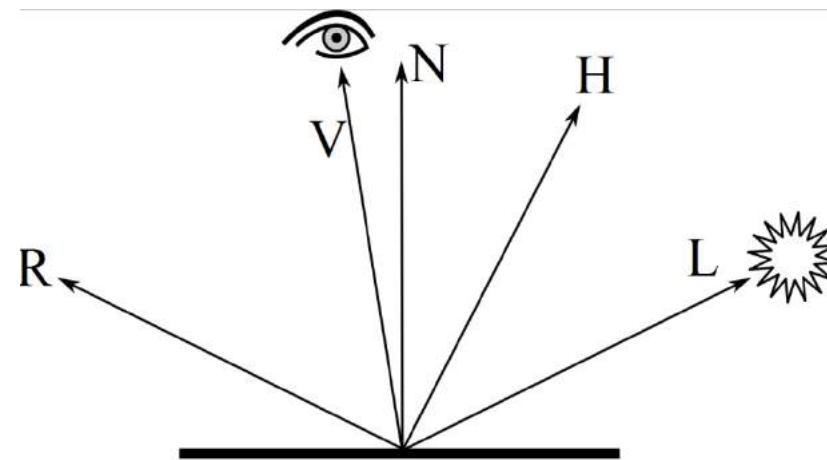
- $I_s = c_l \max(0, V \cdot R)^n$
- We can compute R in the following way:

$$R = -L + 2(L \cdot N)N$$

Specular Reflection: Blinn-Phong

- Another way of computing specular reflection is the Blinn-Phong approximation, based on the **half vector, H**, which is the vector half way between V and L:

$$H = (V + L)/|V + L|$$



Specular Reflection: Blinn-Phong

$$H = (V + L)/|V + L|$$

- Advantage: for directional (e.g. distant) light sources, L is constant, so H needs only to be calculated once for the entire scene!
- Disadvantage: need to compute a squareroot in the normalization – but we need to do this for R as well

Specular Reflection: Blinn-Phong

- With H we compute the specular reflection as:

$$I_s = c_l(H \cdot N)^n$$

Phong vs. Phong-Blinn

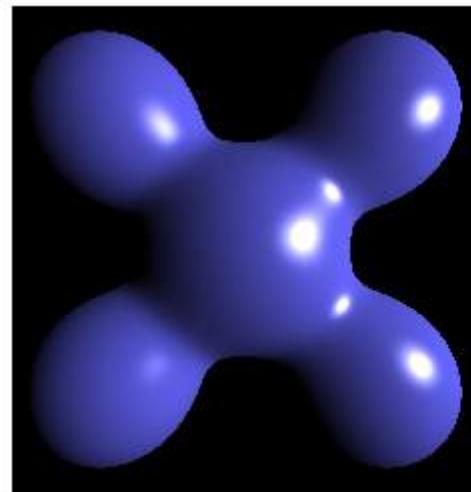
- The halfway angle $N \cdot H$ is smaller than the true Phong angle $R \cdot V$
- To approximate true Phong correctly, we need a larger exponent $n' > n$:

$$I_s = c_l \max(0, V \cdot R)^n$$

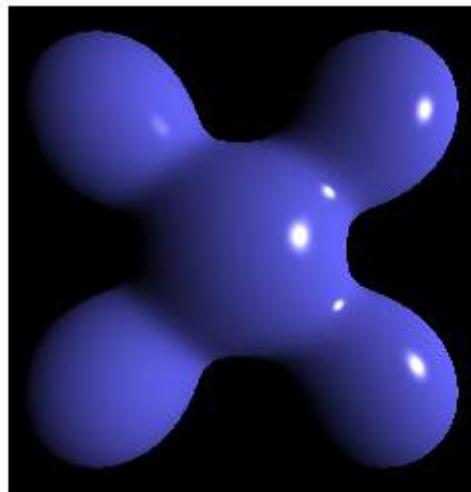
Approximated by:

$$I_s = c_l (H \cdot N)^{n'}$$

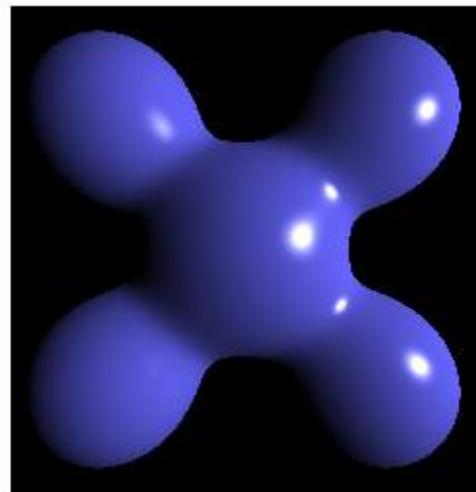
- Both are a gross approximation, so it really doesn't matter



Blinn–Phong



Phong



**Blinn–Phong
(higher exponent)**

http://en.wikipedia.org/wiki/File:Blinn_phong_comparison.png

Material Colour in Specular Reflection

- We can also scale the specular reflection by a colour c_p , particularly when we want to model metals:
- $I_s = c_l c_p \max(0, V \cdot R)^n$

Or

- $I_s = c_l c_p \max(0, N \cdot H)^n$
- c_p is usually a function of the object colour

Multiple Lights

- What happens if we have multiple lights?
 - We just sum the diffuse and specular reflections over the different light sources

Surface Colour

- Putting all of this together we get:

$$I = c_r(c_a + c_l \max(0, N \cdot L)) + c_l c_p (R \cdot V)^n$$

or

$$I = c_r(c_a + c_l \max(0, N \cdot L)) + c_l c_p (H \cdot N)^n$$

Optional – used for metals

Phong Illumination Model

$$I = I_a + I_d + I_s$$

$$I = c_r c_a + \sum_i (c_r c_{l_i} \max(0, L \cdot N) + c_p c_{l_i} \max(0, R \cdot V)^n)$$

Flat Shading

- no interpolation
- all pixels same color
- two methods:
 - one point per triangle/quad
 - average of triangle's/quad's vertices
- low quality: single primitives easily visible
- fast computation & easy implementation

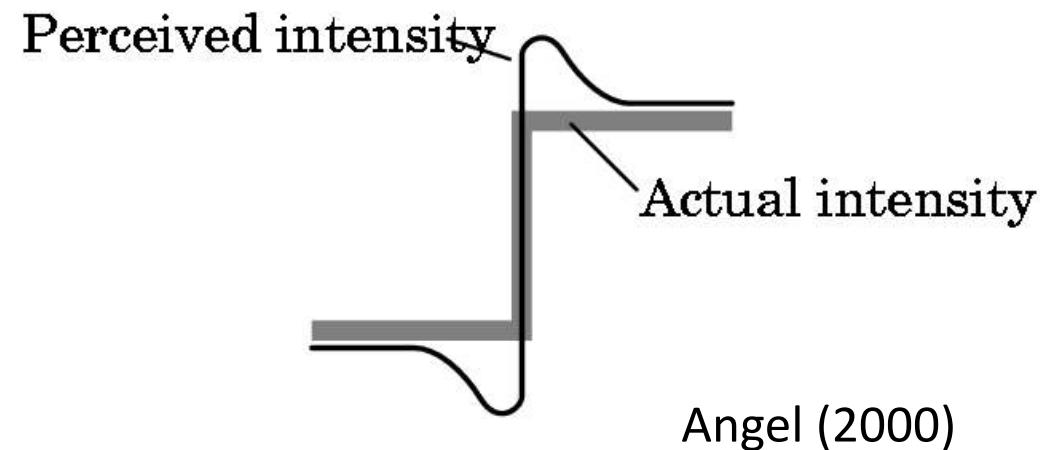


Flat Shading: Edge Perception

- discontinuities easily visible and distracting
 - reason in human perception
 - contrast are enhanced by visual system
 - perceived brightness differences are bigger physical reality



Mach band effect



Smooth Shading

- How do we apply our light model to the object?
- In the past OpenGL used Gouraud shading:
 - Apply the model at each vertex to get a colour for each vertex
 - The vertex colours are then interpolated across the polygon to get the colour at each pixel
- We now compute this in our shader programs, so we are free to use whatever computation we like

Gouraud Shading Problem

- Specular reflection doesn't look good:
 - The highlights can be smaller than a polygon, in which case none of the vertices have a specular reflection component so it can't be interpolated across the polygon
 - Even if a vertex catches the specular highlight, the sharpness of the specular reflection is lost through interpolation

Gouraud Shading (1971)

- computation of colors at all vertices
- linear interpolation of colors over polygon
- more computation but better quality than flat shading
- highlights problematic: highlight shapes and highlights in the middle of triangles

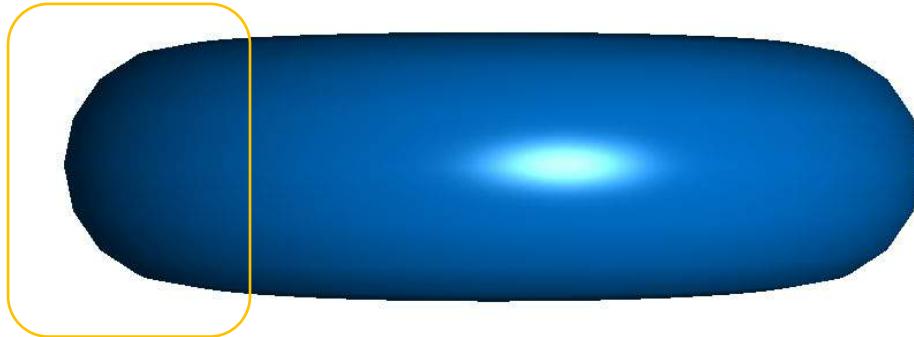


Phong Shading

- Specular problems are solved by interpolating the normal vectors over the surface of the polygon and computing the colour at each pixel
- This is called Phong shading
- This is much more expensive than just interpolating the colours, but is now available in dedicated hardware

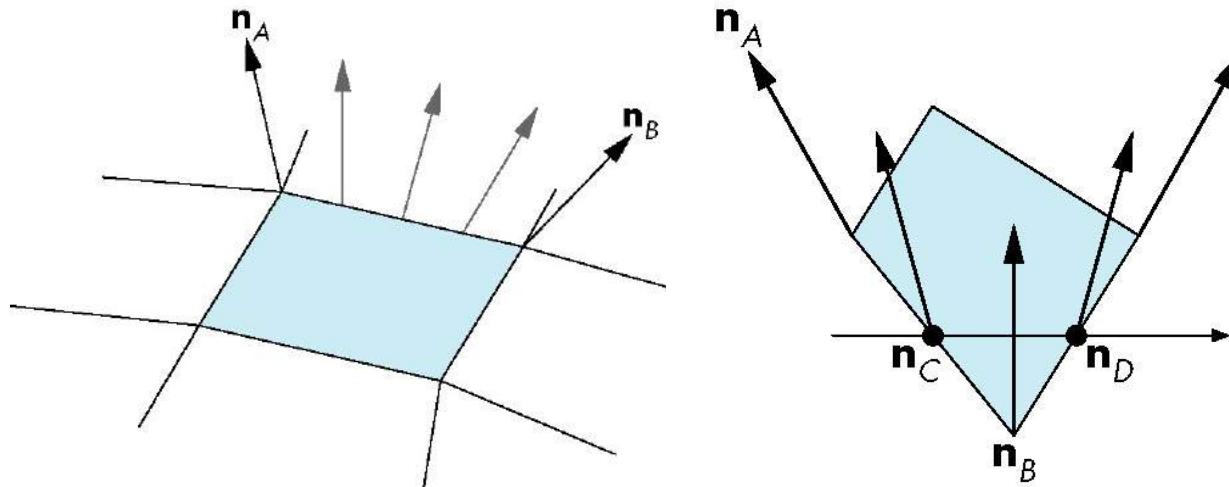
Phong Shading (1973)

- linear interpolation of normals for each pixel
- color computation for each pixel separately
- best quality, highlights are shown correctly
- computationally more expensive
- problems:
 - polygons still visible at silhouettes



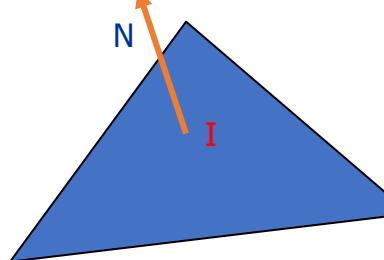
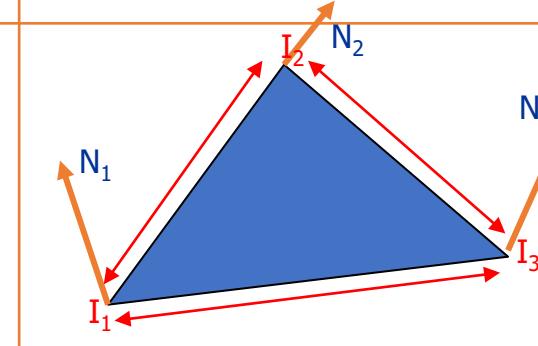
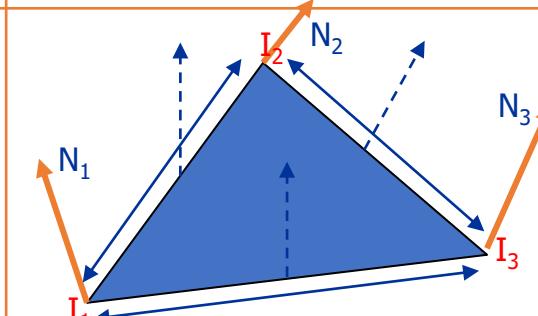
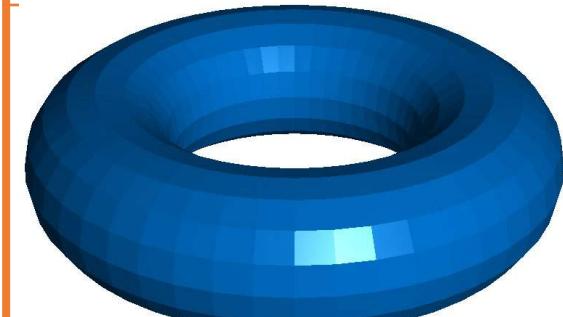
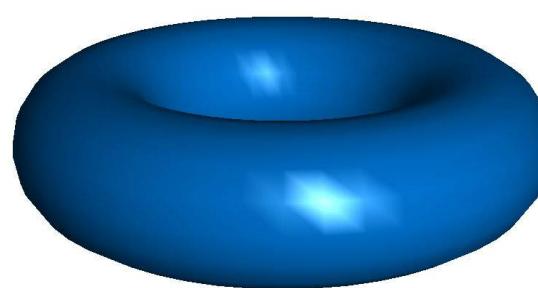
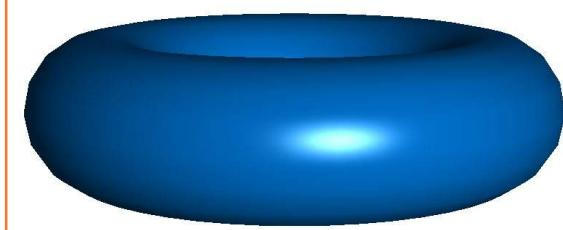
Phong Shading: Normal Interpolation

- first: normal vector interpolation along edges on per-scanline basis
- second: normal vector interpolation between edges along scanlines on per-pixel basis



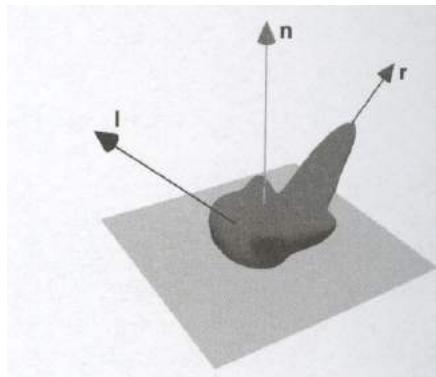
Angel (2003)

Polygonal Shading: Comparison

Flat	Gouraud	Phong
		
one color value for entire polygon	one color value per vertex & interpolation inside the triangle	vertex normals interpolated and one color value per pixel
		

BRDF: Bidirectional Reflectance Distribution Function (advanced!)

- Combine effects of diffuse and specular reflection plus other material properties
- Four dimensional function dependent on the incoming and outgoing light directions
- Closer to physical reality



Bender/Brill (2003)

Increasing Realism



Raytracing
(credit: Ken Musgrave)



Texture Mapping
(credit: Massimo Righi)

Summary

- Algorithms for hidden surface removal:
 - BSD Trees
- Lighting:
 - Lighting is simulated as a sum of ambient, diffuse, and specular
 - Phong illumination model
 - Relation to reality
 - Dependency on normals and reflectance angle

CSCI 3090

OpenGL Lighting

Mark Green

Faculty of Science

Ontario Tech

Introduction

- In the previous lecture we discussed local illumination, presented the Phong model
- In this lecture we will show how this can be implemented in OpenGL
- Lighting models are implemented in shader programs, so we will learn more about shader programming as well

C Code

- This lecture is based on two example programs, example6 and example6a, both are on Canvas
- Most of what we will be doing is in shader programs, so we will start by constructing a C program that we will use for most of the examples
- This saves writing and compiling a new program for each of the examples
- This program is based on the program from laboratory two, so we will use our vase as our test object

C Code

- There are two main changes to this program
- The first change is to use separate viewing and projection matrices for our vertex program
- This is a relatively simple change that only affects a few lines in our display() procedure

C Code

```
void display () {  
    glm::mat4 view;  
    int modelViewLoc;  
    int projectionLoc;  
    int normalLoc;  
  
    ...  
    modelViewLoc = glGetUniformLocation(program,"modelView");  
    glUniformMatrix4fv(modelViewLoc, 1, 0, glm::value_ptr(view));  
    projectionLoc = glGetUniformLocation(program,"projection");  
    glUniformMatrix4fv(projectionLoc, 1, 0, glm::value_ptr(projection));  
    normalLoc = glGetUniformLocation(program,"normalMat");  
    glUniformMatrix3fv(normalLoc, 1, 0, glm::value_ptr(normal));
```

C Code

- The second changes supports multiple vertex and fragment programs without changing the C code
- We can enter the names of the vertex and fragment programs on the command line
- All our shader program names will be of the form example6x.vs or example6y.fs, so to save some typing we will only enter x and y

C Code

- We have two global variables that store these values: `vertexName` and `fragmentName`, they are both `char*` values
- The code on the next slide is added at the beginning of our main procedure
- It provides default values if the user doesn't provide command line parameters

C Code

```
if(argc > 1) {
    vertexName = argv[1];
} else {
    vertexName = "a";
}
if(argc > 2) {
    fragmentName = argv[2];
} else {
    fragmentName = "a";
}
```

C Code

- Finally in the init() procedure we need to make the following change:

```
sprintf(vname,"example6%s.vs", vertexName);
sprintf(fname,"example6%s.fs", fragmentName);
vs = buildShader(GL_VERTEX_SHADER, vname);
fs = buildShader(GL_FRAGMENT_SHADER, fname);
program = buildProgram(vs,fs,0);
```

Example A

- Our first example is a straight forward translation of the Phong light model
- We will assume that we have a directional light, since that is easier to code
- Our vertex shader is quite simple, all it needs to do is transform the vertex position and normal
- The code is on the next slide

Example A

```
in vec4 vPosition;  
in vec3 vNormal;  
uniform mat4 modelView;  
uniform mat4 projection;  
uniform mat3 normalMat;  
out vec3 normal;  
  
void main() {  
  
    gl_Position = projection * modelView * vPosition;  
    normal = normalMat* vNormal;  
}
```

Example A

- The fragment shader is more complicated
- First we need to decide on the coordinate system we are using
- At the fragment shader we have basically two coordinate systems, eye coordinates and projection coordinates
- It makes little sense to use projection coordinates, so we will use eye coordinates

Example A

- Recall: in eye coordinates the eye is at the origin and looking down the z axis
- The first part of the fragment shader is shown on the next slide
- L is the vector in the light direction, colour is the colour of the object, Lcolour is the colour of the light source and H is the half vector
- n is the shininess of the object, used in specular reflection

Example A

```
in vec3 normal;
```

```
void main() {
    vec3 N;
    vec3 L = vec3(1.0, 1.0, 0.0);
    vec4 colour = vec4(1.0, 0.0, 0.0, 1.0);
    vec4 Lcolour = vec4(1.0, 1.0, 1.0, 1.0);
    vec3 H = normalize(L + vec3(0.0, 0.0, 1.0));
    float diffuse;
    float specular;
    float n = 100.0;
```

Example A

- The following slide shows the rest of the code
- We first normalize the normal vector, why?
- After normalizing the light direction we compute $N \cdot L$, if this is negative we have no diffuse or specular light components
- If positive we compute the specular light component, make sure that it is not negative

Example A

```
N = normalize(normal);
L = normalize(L);
diffuse = dot(N,L);
if(diffuse < 0.0) {
    diffuse = 0.0;
    specular = 0.0;
} else {
    specular = pow(max(0.0, dot(N,H)),n);
}

gl_FragColor = min(0.3*colour + diffuse*colour*Lcolour + Lcolour*specular, vec4(1.0));
gl_FragColor.a = colour.a;
```

Example A

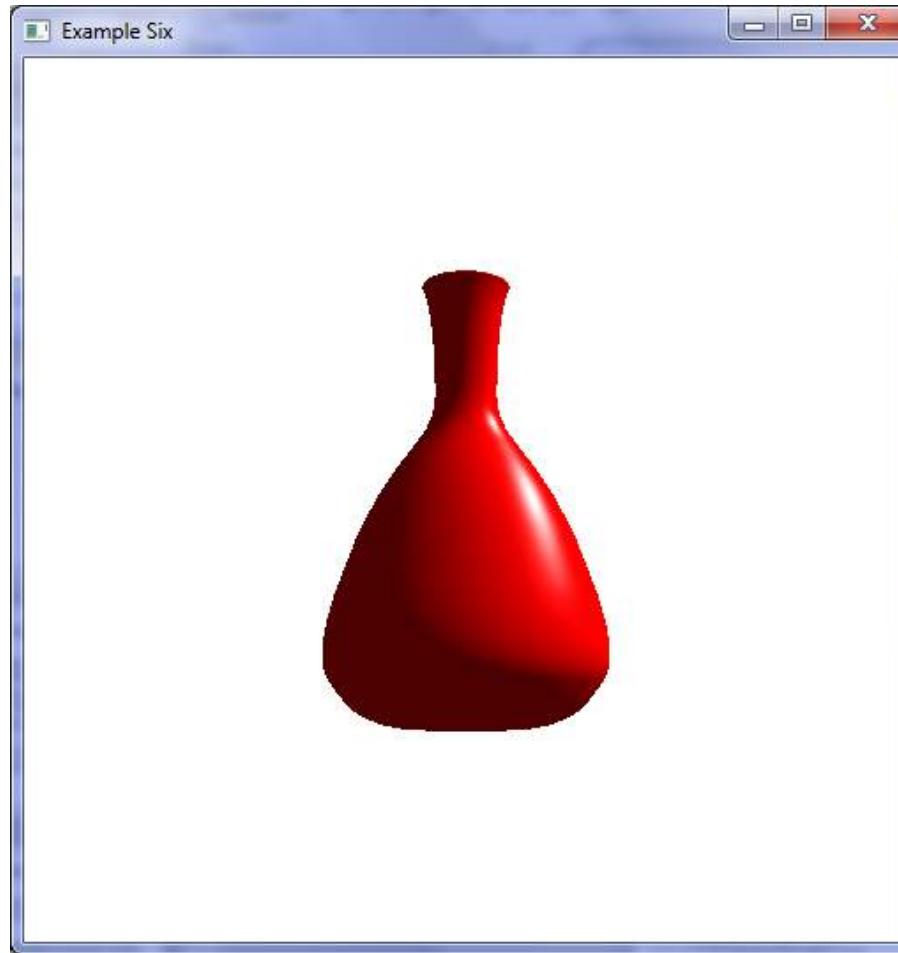
- Finally we combine the three light components
- In this example we have assumed that the ambient light level is 0.3
- Note the use of the min function to make sure that none of the colour components exceeds 1.0
- The result is shown on the next slide

Example A

```
Command Prompt
D:\Courses\CSCI 3090\Examples\Example6\x64\Debug>example6 a a
X range: 23.415314 146.176971
Y range: -7.925004 191.016907
Z range: -15.912292 106.849106
center: 84.796143 91.545952 45.468407

D:\Courses\CSCI 3090\Examples\Example6\x64\Debug>
```

Example A



Example A

- This illustrates the basic idea, but the fragment shader is far from ideal
- Many of the variables should be uniforms, so we can change them from our C program
- This includes L, colour, Lcolour and n, plus the ambient light level
- We will examine how this can be done once we have looked at the other types of light sources

Example B

- For a point light source the light has a position in space, so we need to compute the vector to the light source
- We do this by subtracting the position of the pixel we are shading from the position of the light source
- After normalization, we can use this vector in our computations in the same way as a directional light source

Example B

- We start by making one change to our vertex program
- We need to transform the vertex position to eye space, this will then be interpolated over the polygon's surface
- It will be passed to our fragment program as the pixel's position
- The vertex shader code is shown on the next slide

Example B

```
in vec4 vPosition;  
in vec3 vNormal;  
uniform mat4 modelView;  
uniform mat4 projection;  
uniform mat3 normalMat;  
out vec3 normal;  
out vec4 position;  
  
void main() {  
  
    gl_Position = projection * modelView * vPosition;  
    position = modelView * vPosition;  
    normal = normalMat* vNormal;  
}
```

Example B

- There is a new output variable, position, and a line in the program that computes its value
- The new fragment shader is shown on the next slide
- We have a new variable Lposition, which is the position of the light source
- We use this to compute the value of L, which is used in the remaining computations

Example B

```
in vec3 normal;  
in vec4 position;  
  
void main() {  
    vec3 N;  
    vec3 Lposition = vec3(500.0, 500.0, 500.0);  
    vec4 colour = vec4(1.0, 0.0, 0.0, 1.0);  
    vec4 Lcolour = vec4(1.0, 1.0, 1.0, 1.0);  
    vec3 H;  
    float diffuse;  
    float specular;  
    float n = 100.0;  
    vec3 L;  
  
    N = normalize(normal);  
    L = normalize(Lposition - position.xyz);  
    H = normalize(L + vec3(0.0, 0.0, 1.0));
```

Example B

Example Six

— □ ×



Example C

- I find that working in eye coordinates is difficult, hard to know where to put the lights
- The original version of OpenGL used eye coordinates for lights, since that was easier in hardware
- But, we don't need to do that, we are free to use whatever coordinate system we like in our code

Example C

- There is one problem with using eye coordinates for lights:
The lights move with the eye, when eye position changes light position changes
- If lights are part of the model this is a problem, each time the eye moves the light position needs to be changed
- This greatly complicates our program code

Example C

- So, lets convert our programs to use world coordinates
- To do this our fragment shader will need the world coordinate version of the eye position, vertex coordinate and normal vector
- The vertex coordinates and normal vectors we can easily handle in the vertex shader, just pass their values to the next stage

Example C

```
in vec4 vPosition;  
in vec3 vNormal;  
uniform mat4 modelView;  
uniform mat4 projection;  
uniform mat3 normalMat;  
out vec3 normal;  
out vec4 position;  
  
void main() {  
  
    gl_Position = projection * modelView * vPosition;  
    position = vPosition;  
    normal = vNormal;  
}
```

Example C

- Note that this is more efficient, since we have fewer matrix multiplies
- The eye position can be sent to the fragment program as a uniform variable
- It can then be used in the computation of the half vector, this is the only change
- The fragment shader code is shown on the next slide

Example C

```
in vec3 normal;
in vec4 position;
uniform vec3 eye;

void main() {
    vec3 N;
    vec3 Lposition = vec3(500.0, 500.0, 800.0);
    vec4 colour = vec4(1.0, 0.0, 0.0, 1.0);
    vec4 Lcolour = vec4(1.0, 1.0, 1.0, 1.0);
    vec3 H;
    float diffuse;
    float specular;
    float n = 100.0;
    vec3 L;
    vec3 e;

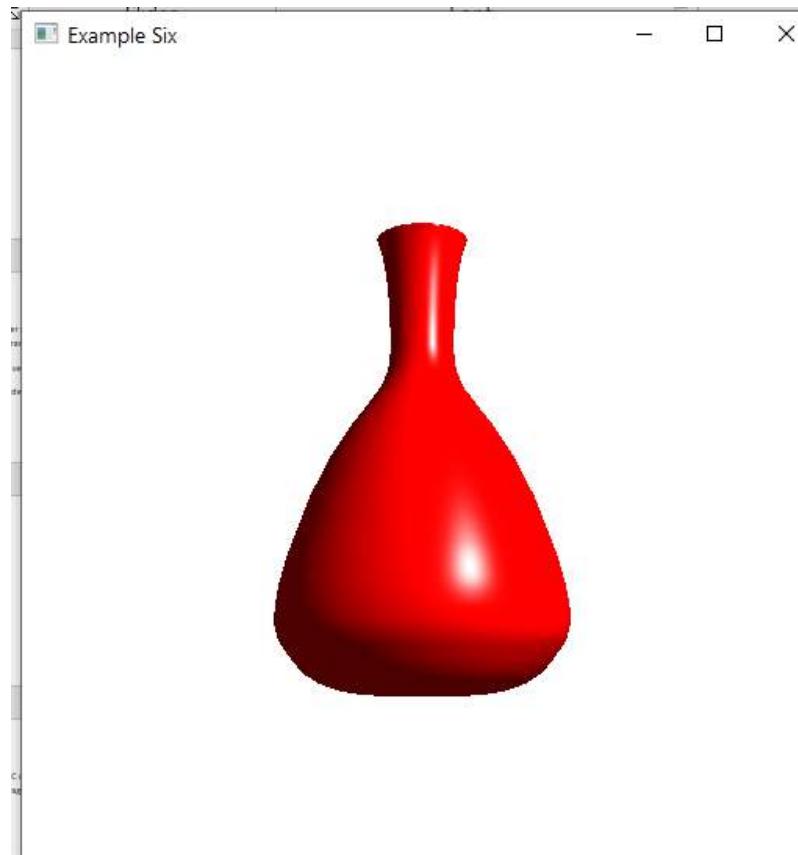
    N = normalize(normal);
    L = normalize(Lposition - position.xyz);
    e = normalize(eye - position.xyz);
    H = normalize(L + e);
```

Example C

- We also need to make a small modification to our C code
- We need to send the current eye position to the fragment shader
- The following code is added to display():

```
eyeLoc = glGetUniformLocation(program, "eye");
glUniform3f(eyeLoc, eyex, eyey, eyez);
```

Example C



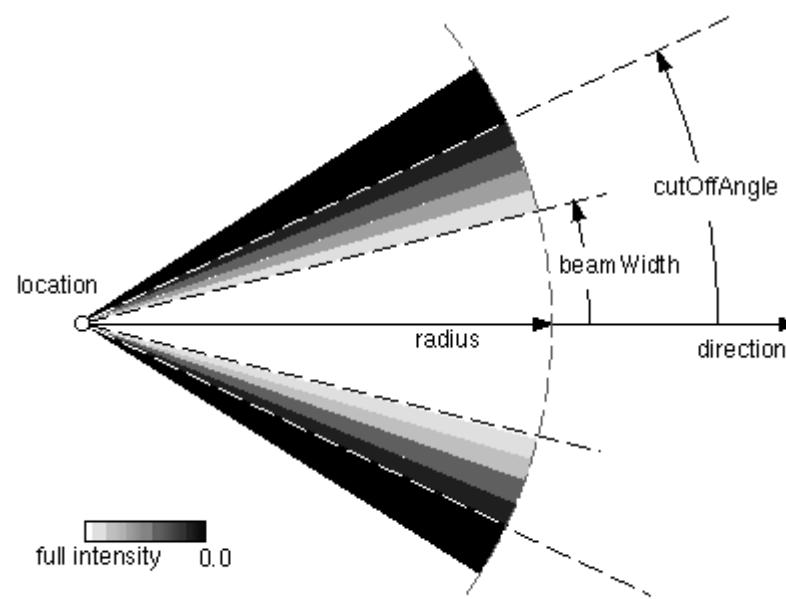
Example D

- The third kind of light that we had was a spotlight
- A spotlight is similar to a point light in that it has a position, but it also has a direction
- The direction is the axis of a cone, points inside this cone receive light, points outside don't
- Inside the cone we have an exponential decay of the light intensity

Example D

- The size of the cone is defined by an angle, called the cut off angle
- We have the direction to the light source, and the direction of the spot light, the dot product will give the angle, or more correctly the cosine of the angle between these two vectors
- If the angle is greater than the cut off angle the light intensity is zero

Example D



Example D

- It is easier to work in terms of cosines than angles, if the two vectors are perfectly aligned the cosine is 1, the cosine decreases as the vectors diverge
- Our cut off angle will now be in terms of the cosine and anything less than this cosine will not be illuminated by the light
- The first part of our fragment shader is on the next slide

Example D

```
void main() {  
    vec3 N;  
    vec3 Lposition = vec3(500.0, 500.0, 800.0);  
    vec4 colour = vec4(1.0, 0.0, 0.0, 1.0);  
    vec4 Lcolour = vec4(1.0, 1.0, 1.0, 1.0);  
    vec3 spotDirection = vec3(500.0, 500.0, 750.0);  
    float spotCutoff = 0.99;  
    float spotExp = 200.0;  
    float spotCos;  
    float atten;  
    vec3 H;  
    float diffuse;  
    float specular;  
    float n = 100.0;  
    vec3 L;  
    vec3 e;
```

Example D

- The atten variable is the amount that the diffuse and specular reflections will be attenuated
- The next part of the fragment code is shown on the next slide
- The dot product between L and the spot direction are used to compute atten, which then multiples diffuse and specular

Example D

```
N = normalize(normal);
L = normalize(Lposition - position.xyz);
e = normalize(eye - position.xyz);
H = normalize(L + e);
spotCos = dot(L, normalize(spotDirection));
if(spotCos < spotCutoff) {
    atten = 0;
} else {
    atten = pow(spotCos,spotExp);
}
diffuse = dot(N,L) * atten;
if(diffuse < 0.0) {
    diffuse = 0.0;
    specular = 0.0;
} else {
    specular = pow(max(0.0, dot(N,H)),n) * atten;
}
```

Example D

Example Six

— □

Example Six

— □



Distance Attenuation

- At this point we could add distance attenuation to our fragment shader
- This adds three more variables for the polynomial coefficients
- The length function can be used to compute the length of the vector from the pixel to the light source
- This is all we need to evaluate the distance attenuation polynomial

Gouraud Shading

- We can move the light model evaluation to the vertex shader, this will give us Gouraud shading
- This will give us a colour at each vertex which will be interpolated across the polygon
- Our fragment shader will now be trivial, just one line of code to copy the colour value

Multiple Light Sources

- For multiple light sources we just need to do a sum over all the lights, this is easy to do in a for loop
- But, right now we have a lot of variables in our fragment shader, adding more lights will multiply the number of variables and it will be hard to keep track of things
- We need a better solution, a better way of packaging this information

Multiple Light Sources

- In addition, we would like to have all the light model parameters controlled by our C code, which means they must be uniform variables
- One way of doing this is to put all the information for a light into a structure, which are supported by GLSL
- The following slide shows how this might look

Multiple Light Sources

```
struct LightProperties {  
    bool isEnabled;  
    bool isLocal;  
    bool isSpot;  
    vec3 ambient;  
    vec3 position;  
    vec3 coneDirection;  
    float spotCutOff;  
    float spotExponent;  
};
```

Multiple Light Sources

- Can then have an array of these structures, one entry for each light:

```
const int MaxLights = 10;
```

```
Uniform LightProperties Lights[MaxLights];
```

- A for loop can then be used in the fragment shader to process each of the lights, similar to what we have done already

Multiple Light Sources

- Note the use of the Boolean flags, we can easily access the individual structure fields, so we can easily change the isEnabled field during display
- The isLocal and isSpot flags can be used to determine the type of light that we have
- If isLocal is false the position field can be interpreted as the direction to the light source

Multiple Light Sources

- We can set up a similar structure for material properties, such as the ambient, diffuse and specular reflectance
- We can also include the shininess for specular reflection
- This nicely packages things at the shader level, but what happens in our C program?
- How do we set the values in this array of structures?

Uniform Blocks

- A number of uniform variables can be combined into a uniform block
- Instead of sending the values individually to the shader, we can send the whole block at once
- This saves calls to OpenGL and makes it easier to modify the set of uniform variables
- Example6a illustrates one way of doing this, there is another way of doing this, but it is much more complicated
- We will stick with the simple way

Uniform Blocks

- The best place to start is with the fragment shader, at the global level we have the following uniform block declaration:

```
layout(std140, binding=1) uniform Light {  
    vec4 Lposition;  
    vec4 Lcolour;  
    vec4 spotDirection;  
    float spotCutoff;  
    float spotExp;  
};
```

Uniform Blocks

- Note that this looks like a C struct
- It starts with a layout clause, with two items in it
- The first std140 tells the GLSL compiler how we want it to layout the memory for the uniform block
- The second is a binding that we will use to refer to this uniform block in our C program
- In our C program we want to have a C struct that is similar to this, but there is a problem

Uniform Blocks

- We are dealing with two different compilers, and they can allocate storage in their own way
- The std140 item tells the GLSL compiler to layout memory in a particular way that we can copy in our C program
- It uses a set of rules for this, the two most important are:
 - Float, int and Boolean variables occupy 4 bytes, are aligned on a 4 byte boundaries
 - Vec3 and vec4 variables occupy 16 bytes, are aligned on 16 byte boundaries

Uniform Blocks

- To make things easy we will always use vec4 and put all of them at the start of the uniform block
- In our C code a vec4 becomes an array of GLfloat of length 4
- With that in mind the next slide shows how we lay out our C struct and initialize its value
- Note that we have 2 padding values at the end of the struct so it ends up being a multiple of 16 bytes long
- By doing this we can have an array of lights without changing the structure

Uniform Blocks

```
struct Light {  
    GLfloat Lposition[4];  
    GLfloat Lcolour[4];  
    GLfloat spotDirection[4];  
    GLfloat spotCutoff;  
    GLfloat spotExp;  
    GLfloat padding[2]; // pad structure to a multiple of 16 bytes  
} light = {  
    { 500.0, 500.0, 800.0, 1.0},  
    { 1.0, 1.0, 1.0, 1.0},  
    { 500.0, 500.0, 750.0, 1.0},  
    0.85, 200.0, 0.0, 0.0  
};
```

Uniform Blocks

- We need to send these values to the GPU, and we do it using a buffer, the same way we sent vertex data to the GPU
- We start with a global variable for the buffer identified:

```
GLuint lightBuffer;
```

- Next we set up the buffer, the same way we've done other buffers in the init() procedure

```
glGenBuffers(1, &lightBuffer);
 glBindBuffer(GL_UNIFORM_BUFFER, lightBuffer);
 glBufferData(GL_UNIFORM_BUFFER, sizeof(light), &light,
 GL_STATIC_DRAW);
```

Uniform Blocks

- Finally we need to specify the uniform block to use in the display() procedure, just before glDrawElements:

```
glBindBufferBase(GL_UNIFORM_BUFFER, 1, lightBuffer);
```
- The second parameter is the value of binding that we used in the fragment shader, the third parameter is the identifier of the buffer in our C program
- We could change any of the values in our light struct, and then just used glBufferSubData to write it to the GPU
- This would give us a way of animating all the properties of our light source

Summary

- Examined how we can implement the Phong light model in OpenGL
- Examined the different types of light sources and the coordinate systems we can use for lighting
- Since this is fully programmable, there are many other things that we can do

CSCI 3090

Ray Tracing

Part One

Mark Green
Faculty of Science
Ontario Tech

The Compleat Angler (1978)

- <https://www.youtube.com/watch?v=WV4qXzM641o>
- <https://www.youtube.com/watch?v=zbjm9TfVMOQ>



Learning Objectives

- By the end of today's class, you will be equipped to:
 - Describe the general concepts of ray tracing
 - Calculate the intersections of rays with spheres

Activity

- In the five following slides, you will find images created by raytracing using the open source raytracer YafaRay (www.yafaray.org).
- What parts of these images would be:
 - Difficult
 - Not possible

Using the rendering we have learned to date?



Loft in New York - Gab 2009 - Blender/Yafarey



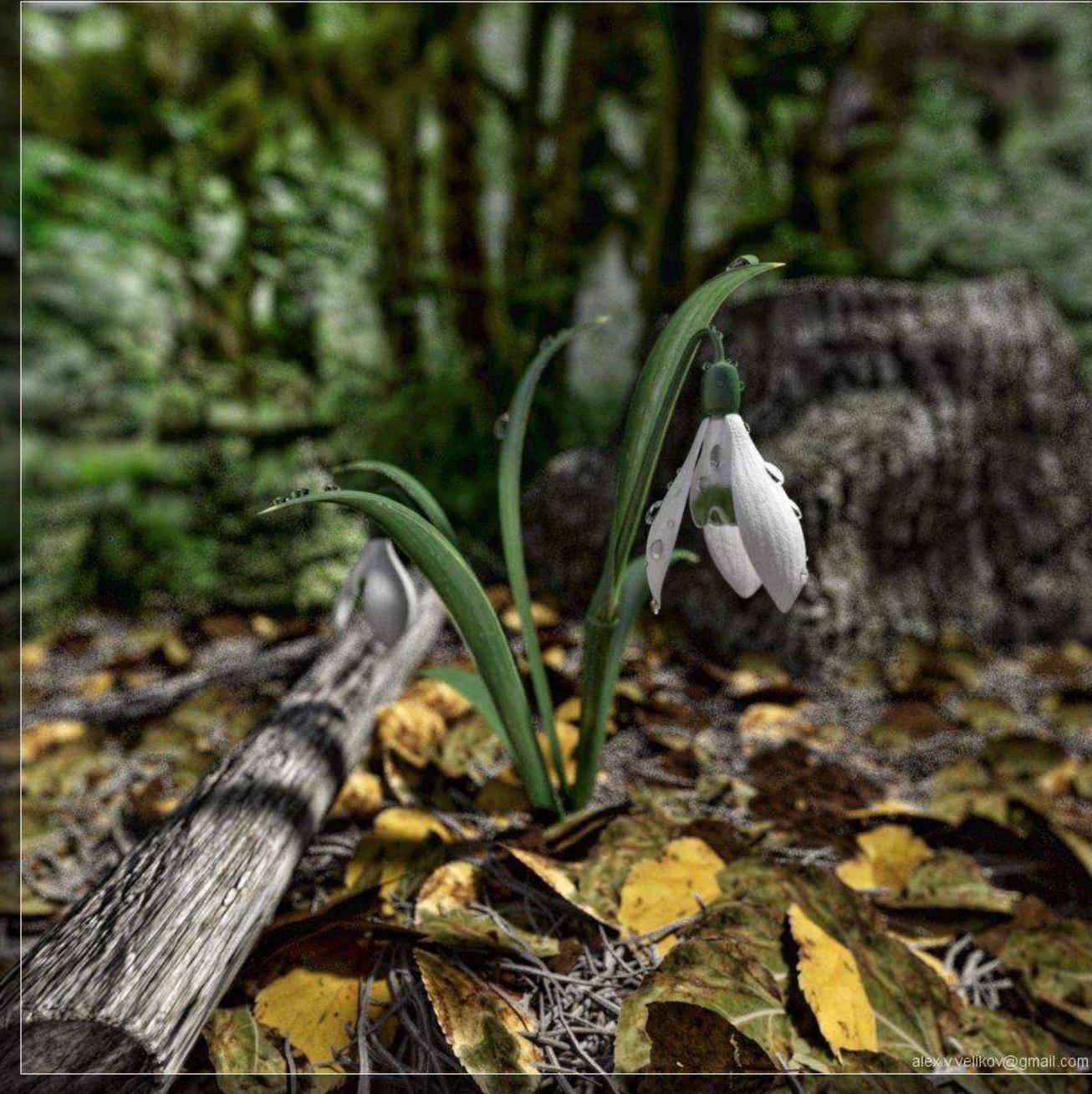
Gab 2009 Blender / Yafaray



alex.v.velikov@gmail.com



Gab , Sebi , Pikrokola



alex.v.velikov@gmail.com

More Examples

PoVRay Hall of Fame:
<http://hof.povray.org/>



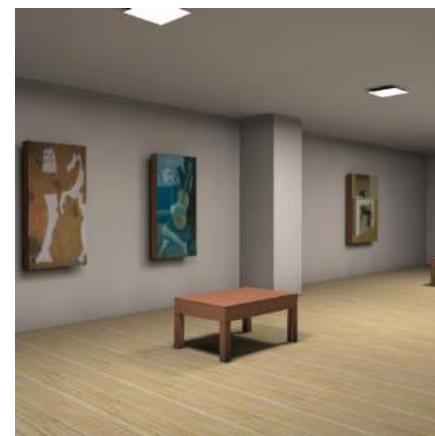
Copyright 2000 Gilles Tran

Ray Tracing

Introduction

Global Illumination: Introduction

- So far: only one interaction between light (from light source) and surface considered, what we know at the pixel
→ local illumination models
- now: model several steps of reflection
→ global illumination models
- two techniques: **ray tracing** & radiosity



Coombe et al., 2004

Properties of Light (CG-simplified)

- principle/law of conservation of energy
 - no energy can be created or destroyed
 - incoming equals outgoing energy everywhere
 - cg is a special case – visible light as single form of energy:
 - light energy
may convert
into other forms
 - other energy types
may convert
into light energy



Properties of Light (CG-simplified)

- duality of light
 - light as waves (part of the electromagnetic spectrum)
→ recall light model lecture
 - light as particles (carrying certain amounts of energy)
→ use in raytracing & radiosity
- cg-light follows straight paths
(sorry Einstein!)



Local Illumination Revisited

- previously: Phong illumination model
(ambient, diffuse, and specular reflection)
→ only coarse **approximation** of reality
- goal: look at real physical conditions
 - use particle model of light
 - derive reflected amount of light depending on
 - incoming light from different directions
 - more realistically modeled reflection properties
 - own emitted light

Basic Idea

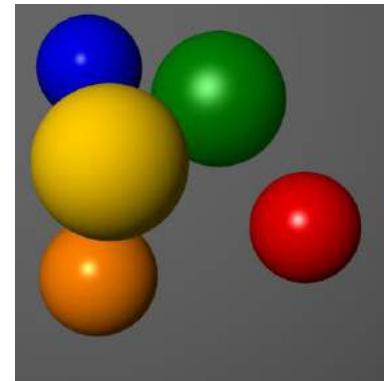
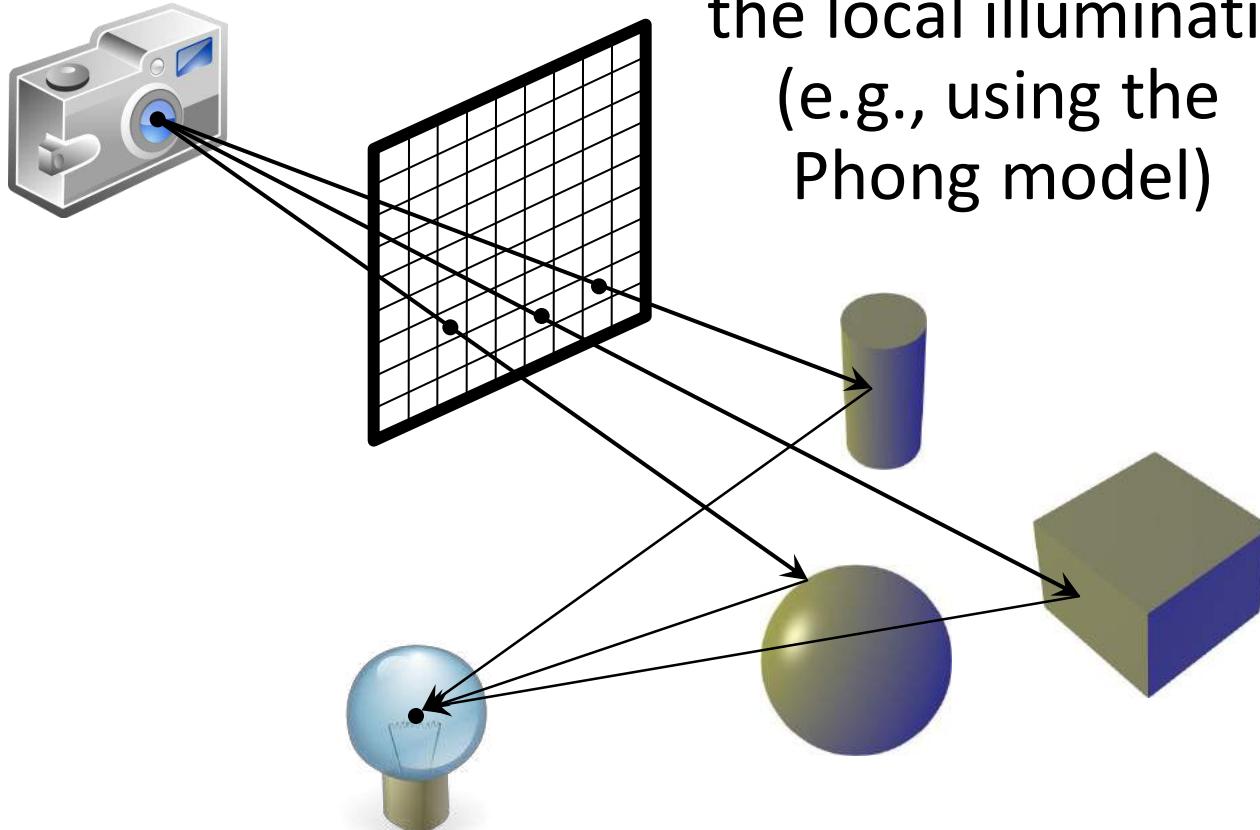
- When we look at the screen our eyes see rays of light coming from the screen
- Can be viewed as one ray per pixel, since pixels have constant colour
- A ray that originates at the pixel and strikes the eye, contains information on one small part of the scene
- But, how do we compute the pixel colour?

Basic Idea

- We can just reverse the ray and trace (or follow) it into the scene
- The ray goes from the pixel and then strikes an object in the scene, or continues on into the background
- We now need to compute the colour of the object where the ray hits it
- We need to determine the light contributions from other objects in the scene

Raycasting

- shoot rays; for each intersection compute the local illumination (e.g., using the Phong model)

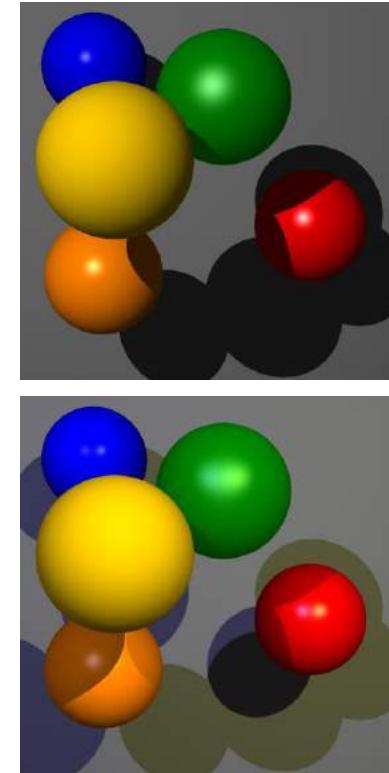
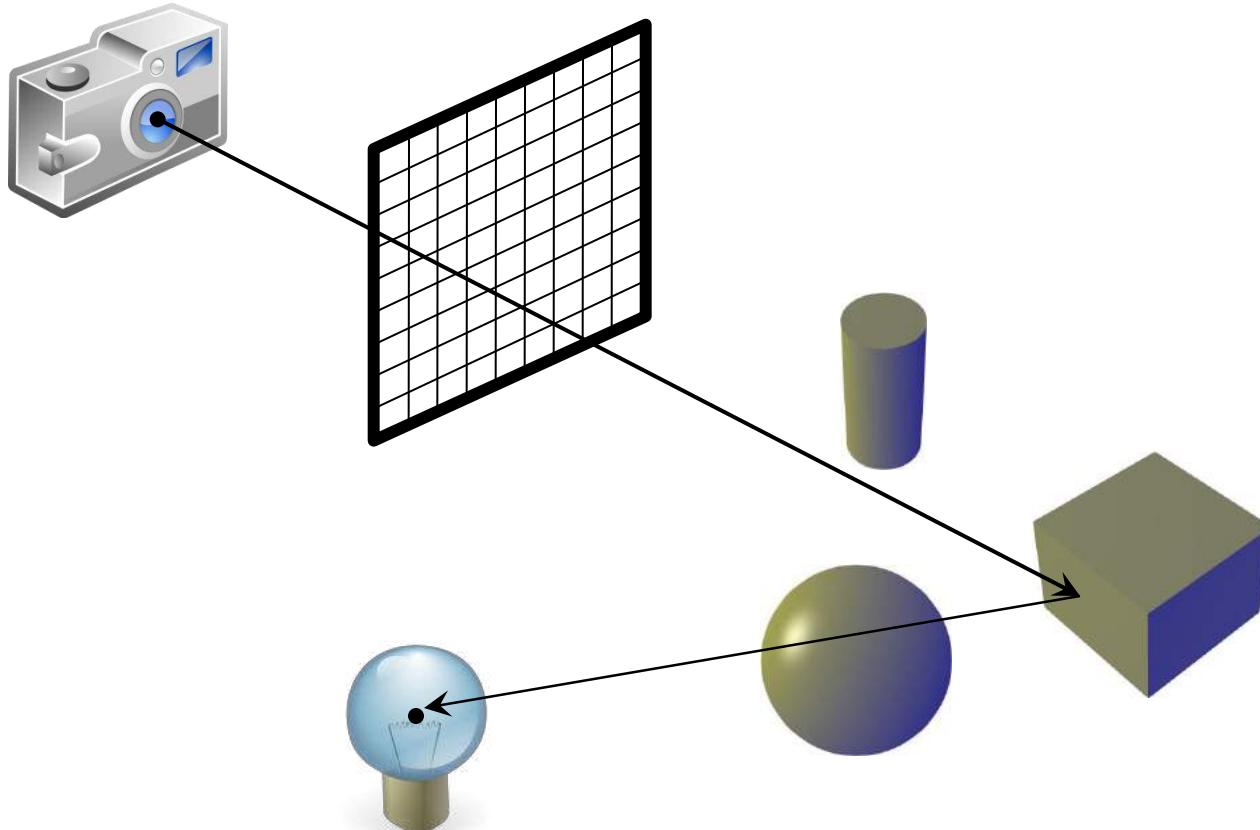


Basic Idea

- We can trace a ray from the intersection point to each of the light sources
- If there is a clear path, we can compute the diffuse and specular light components produced by that light
- If there isn't a clear path, the object is in shadow from that light source, and only the ambient component is used

Generating Shadows

- shadow rays to test intersections between point and light source



Basic Idea

- The ray can also reflect off of the surface that it hit, using standard reflection
- We can then follow this ray to see where it goes
- If it strikes another object, we need to compute the colour at that object, since that's the reflected colour for the object that generated the reflection ray

Basic Idea

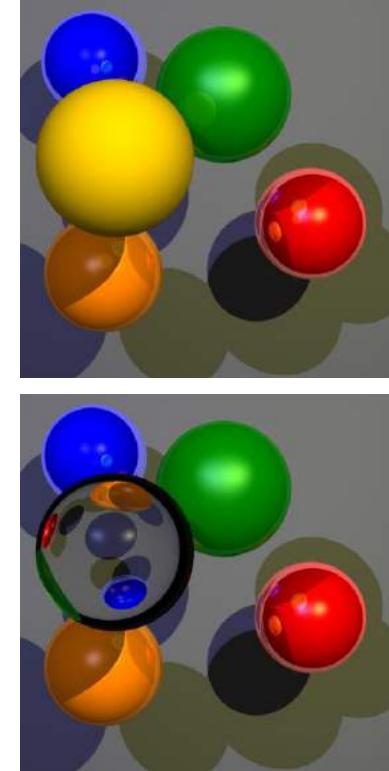
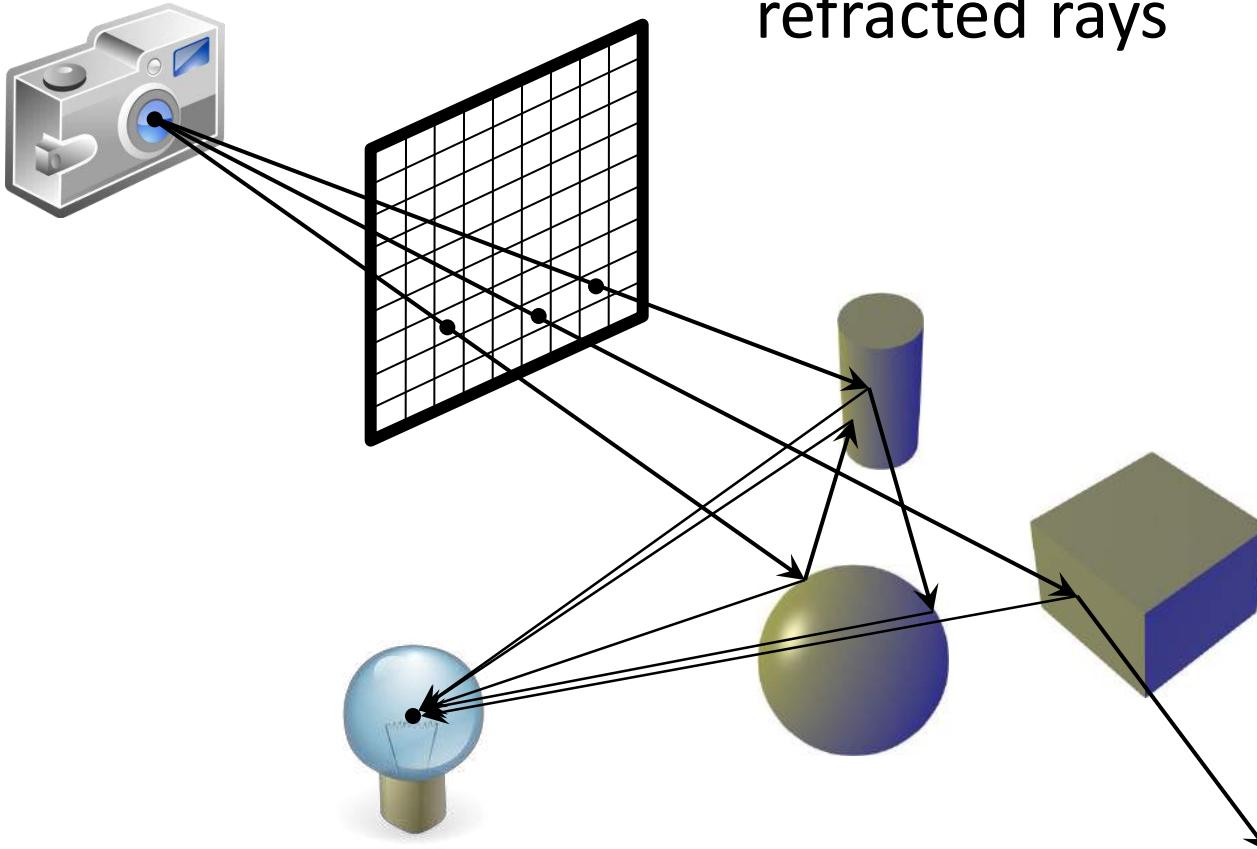
- If the object is transparent, we need to trace a ray through the object to find the colour on the other side
- The ray can be refracted in this process, that is the path of the ray is bent
- After it emerges from the object we keep on tracing the ray until it hits another object and we can compute the colour there

Basic Idea

- Each time a reflected or refracted ray hits another object we apply the ray tracing procedure recursively with the new ray starting at the intersection point
- Thus, the original ray through the pixel can generate a large number of rays as it traces the light bouncing around the scene
- We usually stop tracing once we have reached a certain level

True Raytracing (Recursion)

- recursion: also follow reflected rays and refracted rays



Phong Illumination Model

$$I = I_a + I_d + I_s$$

$$I = c_r c_a + \sum_i (c_r c_{l_i} \max(0, L \cdot N) + c_p c_{l_i} \max(0, R \cdot V)^e)$$

Blinn-Phong Illumination Model

$$I = I_a + I_d + I_s$$

$$I = c_r c_a + \sum_i (c_r c_{l_i} \max(0, L \cdot N) + c_p c_{l_i} \max(0, N \cdot H)^e)$$

Phong Illumination Model

- We can also introduce material properties which specify *how much* (e.g. a scalar) a/d/s reflections there are:

$$I = k_a I_a + k_d I_d + k_s I_s$$

$$\begin{aligned} I = & k_a c_r c_a \\ & + \sum_i (k_d c_r c_{l_i} \max(0, L \cdot N) + k_a c_p c_{l_i} \max(0, R \cdot V)^e) \end{aligned}$$

Phong Illumination Model

- We can also introduce material properties which specify *how much* (e.g. a scalar) a/d/s reflections there are:

$$I = k_a I_a + k_d I_d + k_s I_s$$

$$I = k_a c_r c_a + \sum_i (k_d c_r c_{l_i} \max(0, L \cdot N) + k_a c_p c_{l_i} \max(0, R \cdot V)^e)$$

Material colour for ambient and diffuse reflection

Material colour for specular reflection.
This is often omitted; used only for metals.

Phong Illumination Model

- We can also introduce material properties which specify *how much* (e.g. a scalar) a/d/s reflections there are:

$$I = k_a I_a + k_d I_d + k_s I_s$$

$$I = k_a c_r c_a + \sum_i (k_d c_r c_{l_i} \max(0, L_i \cdot N))$$

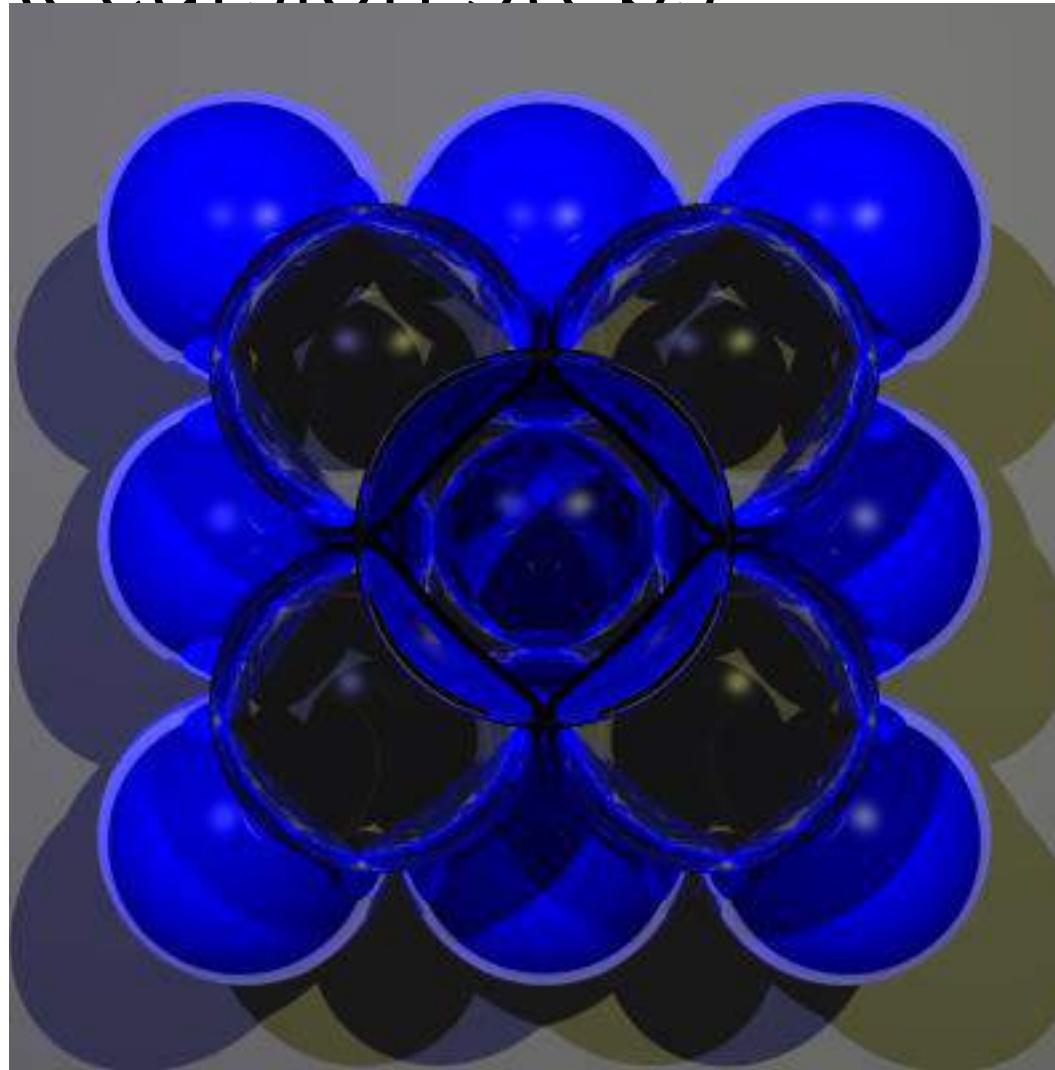
Color of the ambient light (not summed over lights); for A2 assume (1,1,1), therefore this term can be omitted

Adapted Illumination Model

- enhance the Phong model with terms for reflected (I_r) and refracted portion (I_t)
(model by Whitted):

$$\begin{aligned} I = & k_a c_r c_a \\ & + \sum_i (k_d c_r c_{l_i} \max(0, L \cdot N) + k_s c_p c_{l_i} \max(0, R \cdot V)^e) \\ & + \alpha I_r + \beta I_t \end{aligned}$$

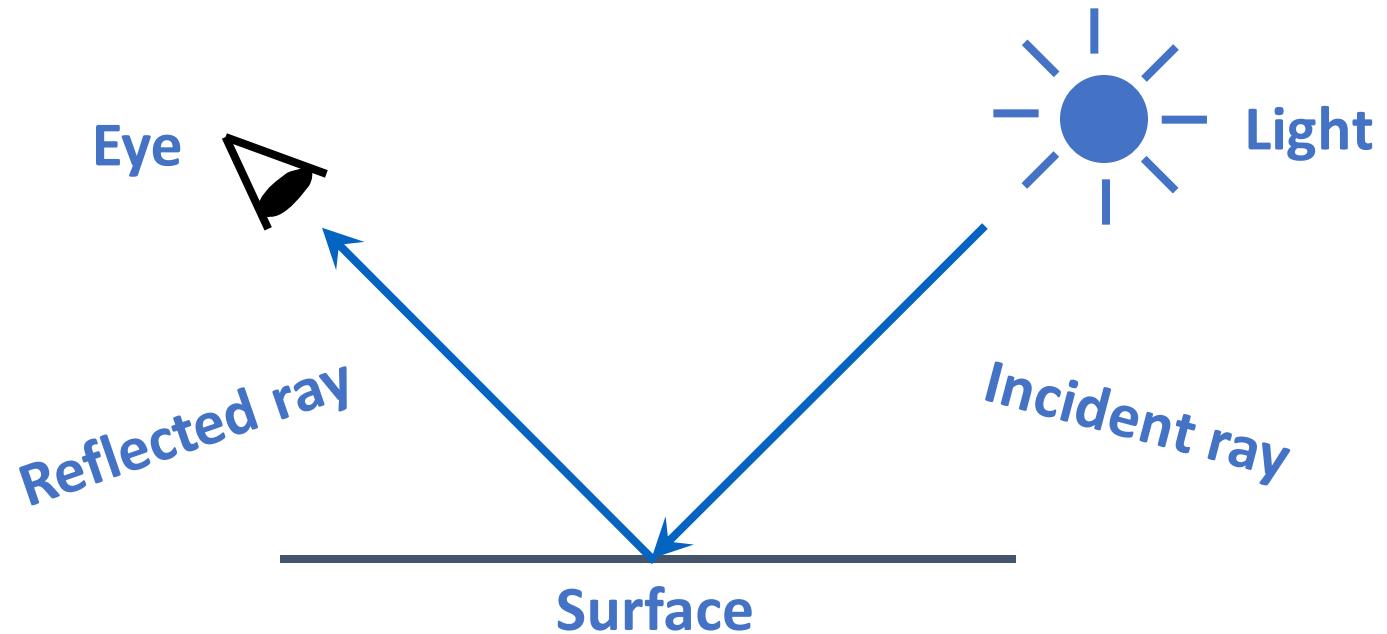
Raytracing Recursion Stems



Recursion 5

The Basic Idea

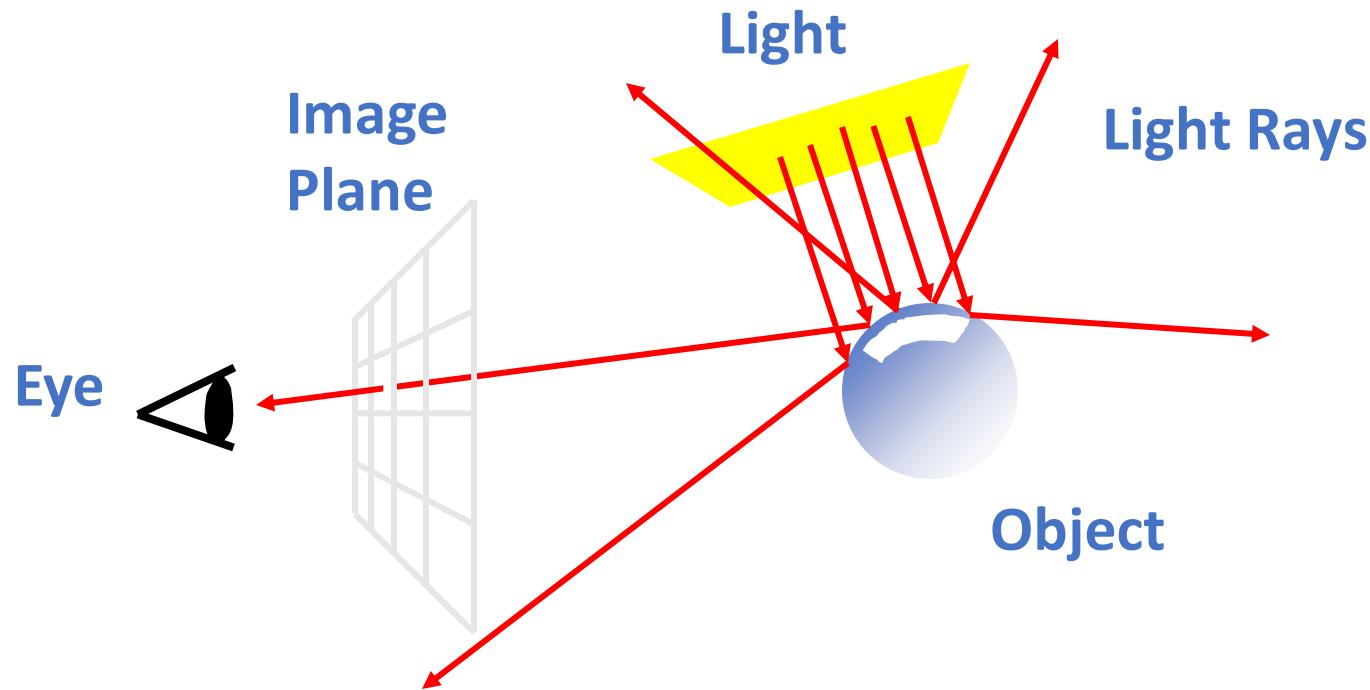
- Simulate light rays from light source to eye



But... we trace rays from the eye
to the object (then to the light)?

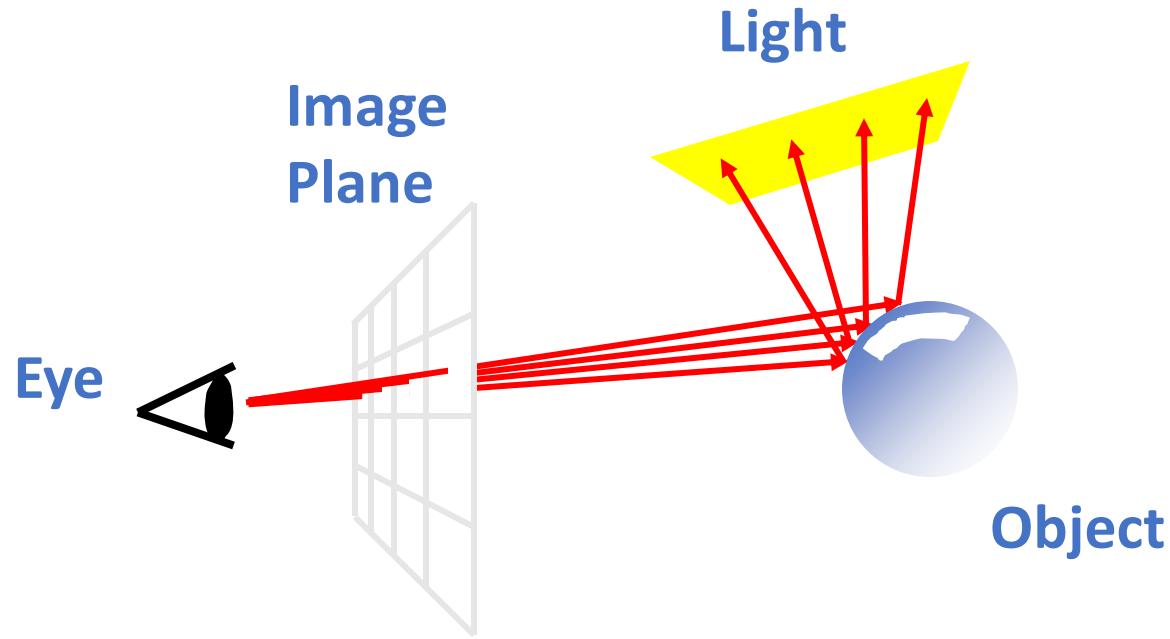
“Forward” Ray-Tracing

- Trace rays from light
- Lots of work for little return



“Backward” Ray-Tracing

- Trace rays from eye instead
- Do work where it matters

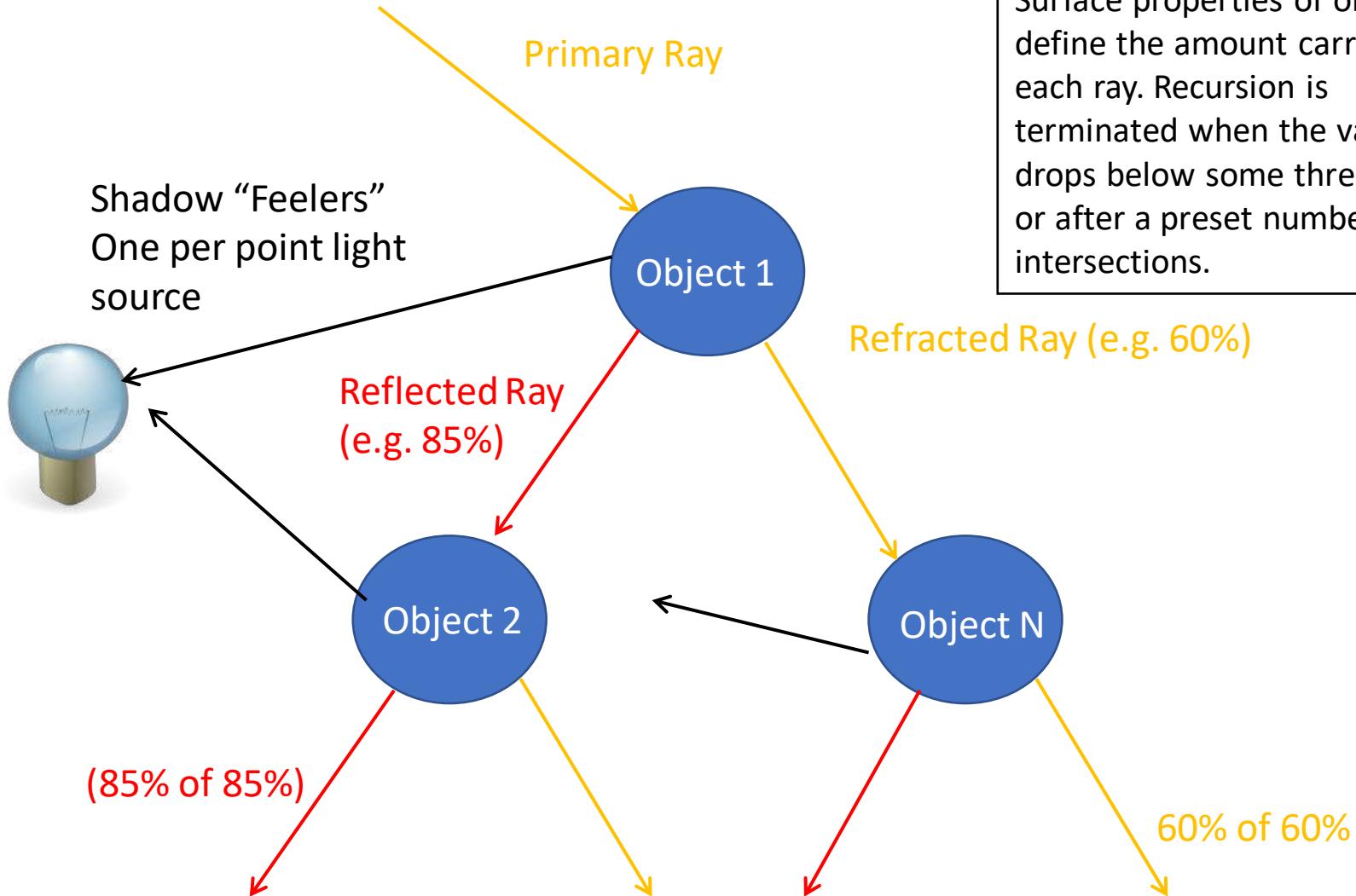


This is what most people mean by “ray tracing”.

Efficiency of Backward Ray Tracing

- Each of the original rays can generate many more rays, and the original ray is the actual pixel colour
- We can view the collection of rays as a tree, with the pixel as the root
- Each intersection forms a node, with the rays generated at that intersection leading to its children
- This is called a **ray tree**, and can be recursively evaluated to produce the pixel colour

The Ray Tree



Efficiency Considerations

- This tree can quickly grow quite large
 - ray tracing can become quite expensive if we are not careful
- We want to control the number of rays
- We also want to make the intersection computations as efficient as possible since we need to do it a lot of times

Stopping Conditions

- Returning to the question of when do we stop tracing rays, there are two possible termination conditions:
 1. Stop tracing when the ray tree has reached a maximum depth
 2. Stop tracing when the contribution to the colour is below a certain level.
- Usually both of these conditions are used, and we usually give the user control over them

Finding Intersections

- When we trace a ray we want to find the closest object, not just any object that the ray intersects
- Only the closest object contributes, since the other objects are hidden by it
- Naïve approach is to test every object in the scene, and then select the one with the closest hit, only works for very small scenes

Coordinate Systems

- First we need to put everything in a common coordinate system so we can generate rays
- We know that we need to generate a ray from the eye to each pixel on the screen, and then trace this ray into the scene or model, but which coordinate system is used?

Eye Coordinates

- Earlier in rendering we assumed that the eye was at the origin looking down the z axis
- The display screen was perpendicular to the z axis
- We can use this coordinate system for ray tracing, in this case we need to **transform the model so it is in the eye coordinate system**
- We only need to do this once, so it is not a major efficiency concern

Model Coordinates

- Alternatively, we can place the eye and the display screen in the model coordinate system
- This transformation needs to be applied to each of the rays that go through pixels
- But again this is a relatively simple transformation, not a major efficiency concern

Intersection

- This is the most expensive operation in ray tracing, so we need to be careful here
- Mathematically we represent a ray in the following way:
 - $p(t) = e + td, \quad t > 0$
- Where e is the starting point of the ray, d is a unit vector in the ray direction and t is the parameter that gives the points on the ray
- Note that **t will always be greater than zero**, otherwise the intersection is behind the eye

Intersection

- One of the easiest objects to intersect is the sphere
- Given the implicit representation of the sphere $f(p) = 0$, the intersection points can be computed by solving:

$$f(e + td) = 0$$

- In vector form the equation of a sphere is:

$$(p - c) \cdot (p - c) - R^2 = 0$$

- where c is the center and R is the radius

Intersection

- On substituting into this equation we get:

$$(e + td - c) \cdot (e + td - c) - R^2 = 0$$

- Or

$$(td + (e - c)) \cdot (td + (e - c)) - R^2 = 0$$

$$(d \cdot d)t^2 + 2(d \cdot (e - c))t + (e - c) \cdot (e - c) - R^2 = 0$$

- This is just a quadratic equation in t that can be written in the form:

$$At^2 + Bt + C = 0$$

- The quadratic formula can be used to solve this equation

Intersection

$$(d \cdot d)t^2 + 2(d \cdot (e - c))t + (e - c) \cdot (e - c) - R^2 = 0$$

$$A = (d \cdot d)$$

$$B = 2(d \cdot (e - c))$$

$$C = (e - c) \cdot (e - c) - R^2$$

Where e = eye, c = sphere center, d = ray direction

- The quadratic formula can be used to solve this equation:

$$\text{roots} = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$$

Intersection

- We know that there are two roots to this equation, if the roots are complex we know that the ray doesn't hit the sphere
- Otherwise we have real roots, and we want the smaller of the two roots (Why?)
- The discriminant can be used to quickly check whether the roots are real, and only then do we proceed to computing the smallest root

Intersection

- The sphere intersection calculation illustrates one of the main ray tracing techniques
- If we have an implicit representation all we need to do is substitute the parametric ray into this equation and solve
- In the case of the sphere the solution is easy, for most other objects it is not

Summary

- Ray Tracing
 - Basic concepts
 - Reflection rays, refraction rays, shadow rays
 - Forward and backward ray tracing
 - Intersections with spheres

Next Class

- General Intersections
- Accelerated Ray Tracing
- Recursive Ray Tracing: Shadows, Refraction & Reflection
- Distributed Ray Tracing



CSCI 3090

Ray Tracing

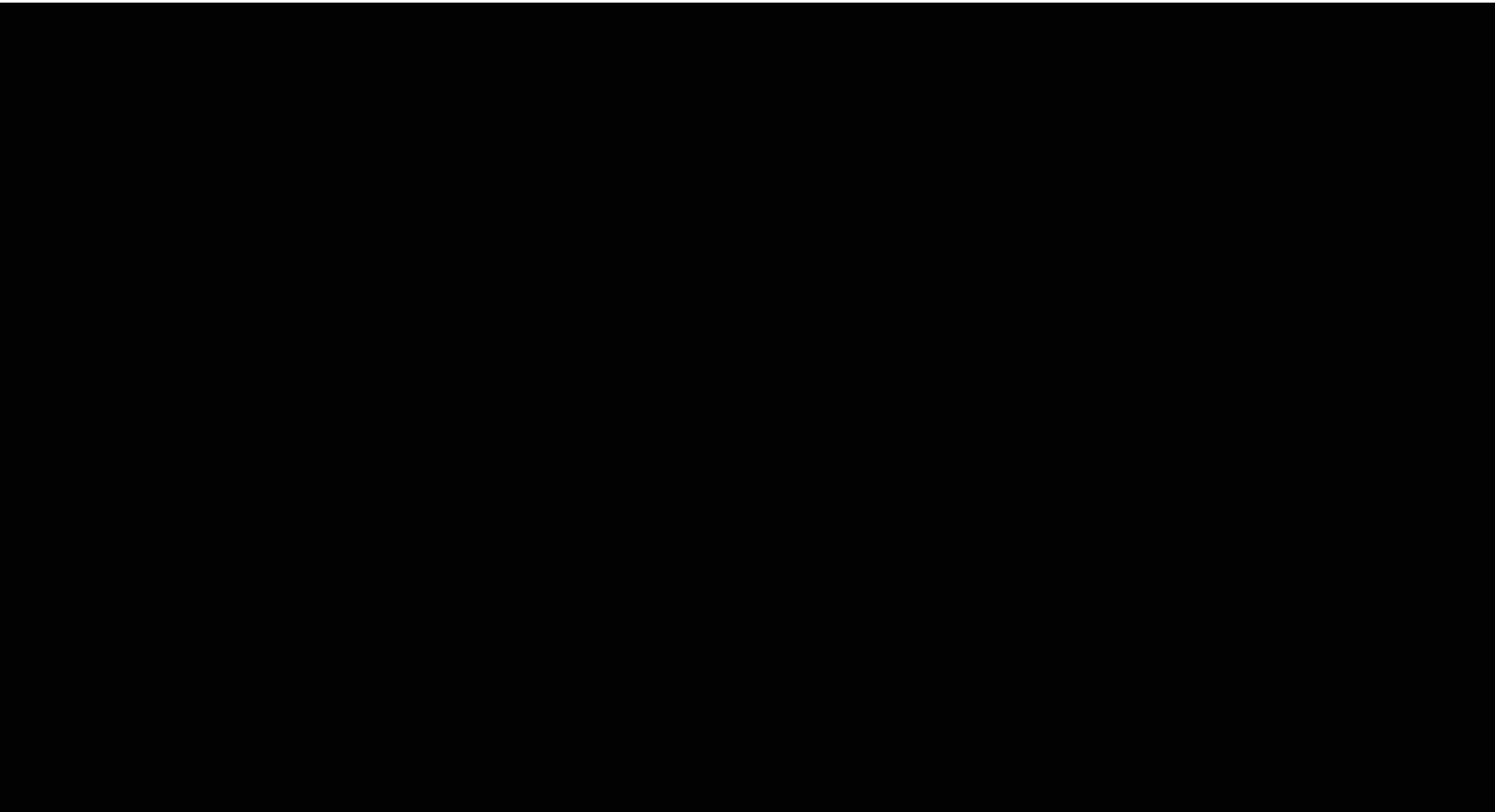
Part Two

Mark Green

Faculty of Science

Ontario Tech

Boundin'



Learning Objectives

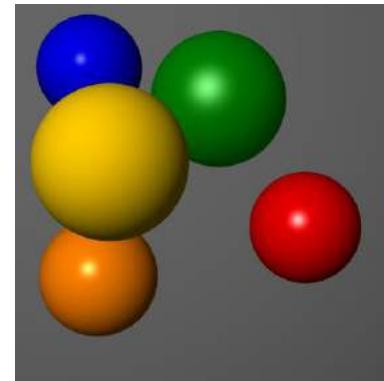
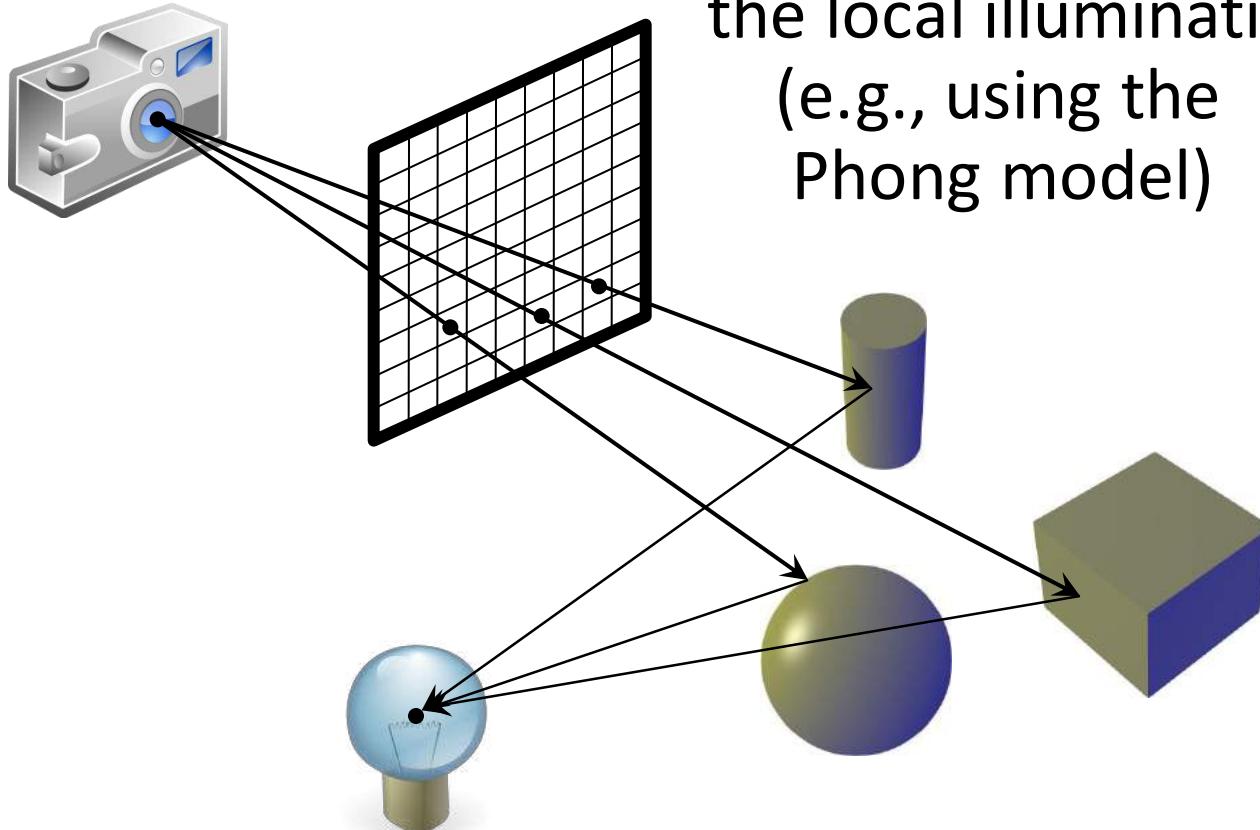
- By the end of today's class, you will be equipped to:
 - Calculate the intersections of rays with simple geometric shapes
 - Create a simple raytracer which uses shadows, refraction, and iterative reflection

Introduction to Ray Tracing

Review

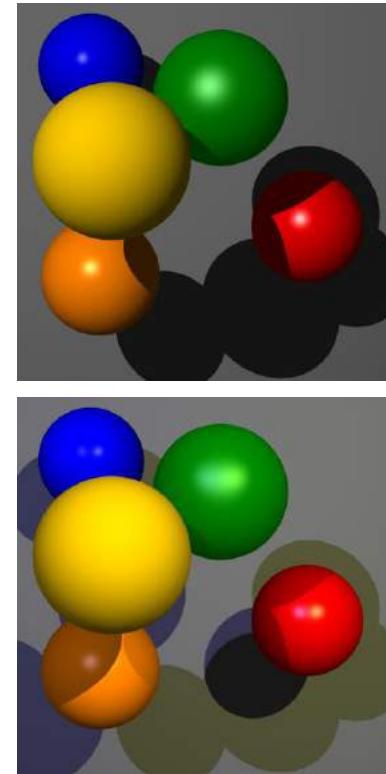
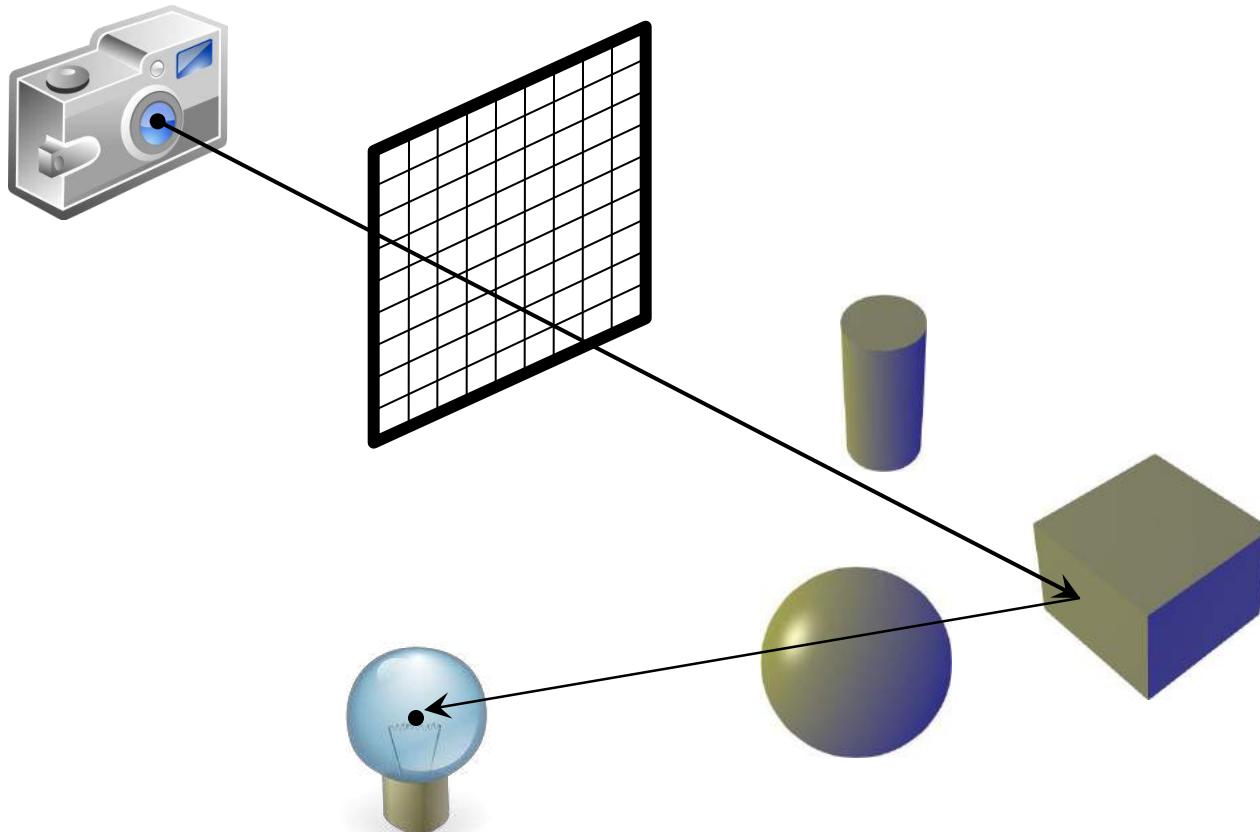
Raycasting

- shoot rays; for each intersection compute the local illumination (e.g., using the Phong model)



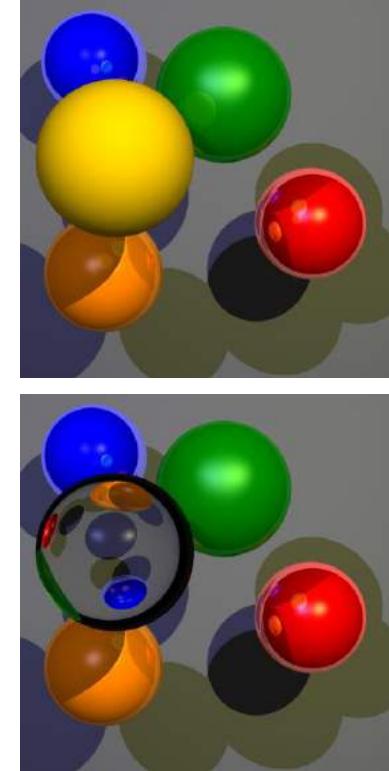
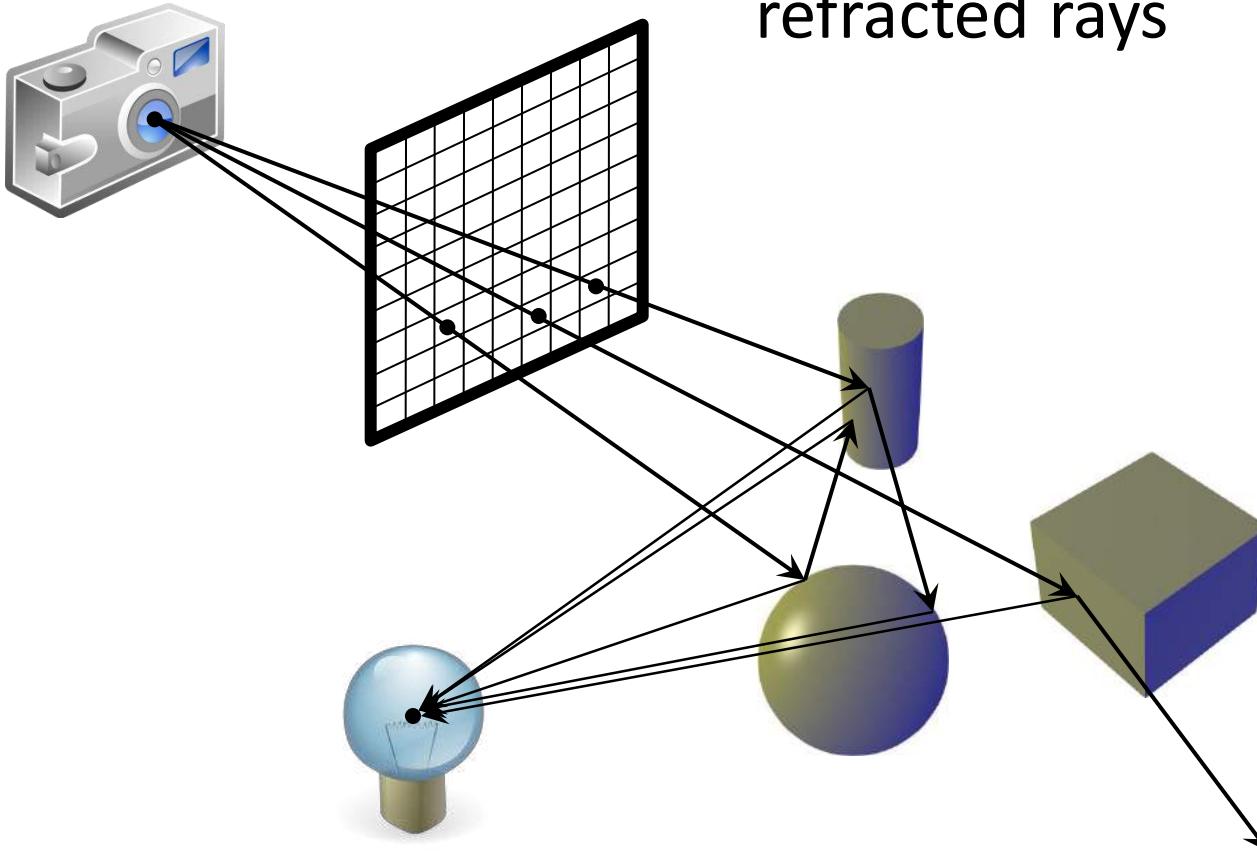
Generating Shadows

- shadow rays to test intersections between point and light source



True Raytracing (Recursion)

- recursion: also follow reflected rays and refracted rays



Adapted Illumination Model

- enhance the Phong model with terms for reflected (I_r) and refracted portion (I_t)
(model by Whitted):

$$\begin{aligned} I = & k_a c_r c_a \\ & + \sum_i (k_d c_r c_{l_i} \max(0, L \cdot N) + k_s c_p c_{l_i} \max(0, R \cdot V)^e) \\ & + \alpha I_r + \beta I_t \end{aligned}$$

Intersection

- This is the most expensive operation in ray tracing, so we need to be careful here
- Mathematically we represent a ray in the following way:

$$p(t) = e + td, t > 0$$

Where e is the starting point of the ray, d is a unit vector in the ray direction and t is the parameter that gives the points on the ray

- Note that **t will always be greater than zero**, otherwise the intersection is behind the eye

Intersection

- One of the easiest objects to intersect is the sphere
- Given the implicit representation of the sphere $f(p) = 0$, the intersection points can be computed by solving:

$$f(e + td) = 0$$

- In vector form the equation of a sphere is:

$$(p - c) \cdot (p - c) - R^2 = 0$$

- where c is the center and R is the radius

Intersection

- On substituting into this equation we get:
 - $(e + td - c) \cdot (e + td - c) - R^2 = 0$
- Or
 - $(td + (e - c)) \cdot (td + (e - c)) - R^2 = 0$
 - $(d \cdot d)t^2 + 2(d \cdot (e - c))t + (e - c) \cdot (e - c) - R^2 = 0$
- This is just a quadratic equation in t that can be written in the form:
 - $At^2 + Bt + C = 0$
- The quadratic formula can be used to solve this equation

Intersection

- $(d \cdot d)t^2 + 2(d \cdot (e - c))t + (e - c) \cdot (e - c) - R^2 = 0$
 - $A = (d \cdot d)$
 - $B = 2(d \cdot (e - c))$
 - $C = (e - c) \cdot (e - c) - R^2$
- Where e = eye, c = sphere center, d = ray direction
- The quadratic formula can be used to solve this equation:

$$roots = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$$

General Intersection

Intersection

- For a general polygon the intersection calculation is a two step process
- The first step determines where the ray intersects the polygon's plane, as long as the ray isn't parallel to the plane
- The second step is determines whether the intersection point is inside or outside the polygon

General Polygon

- Given a polygon vertex p_1 and the normal n , to the polygon, the plane equation is given by:

$$(p - p_1) \cdot n = 0$$

For any point p on the plane

- Replacing p by $e + td$ gives the following solution:

$$(e + td - p_1) \cdot n = 0$$

Rearrange:

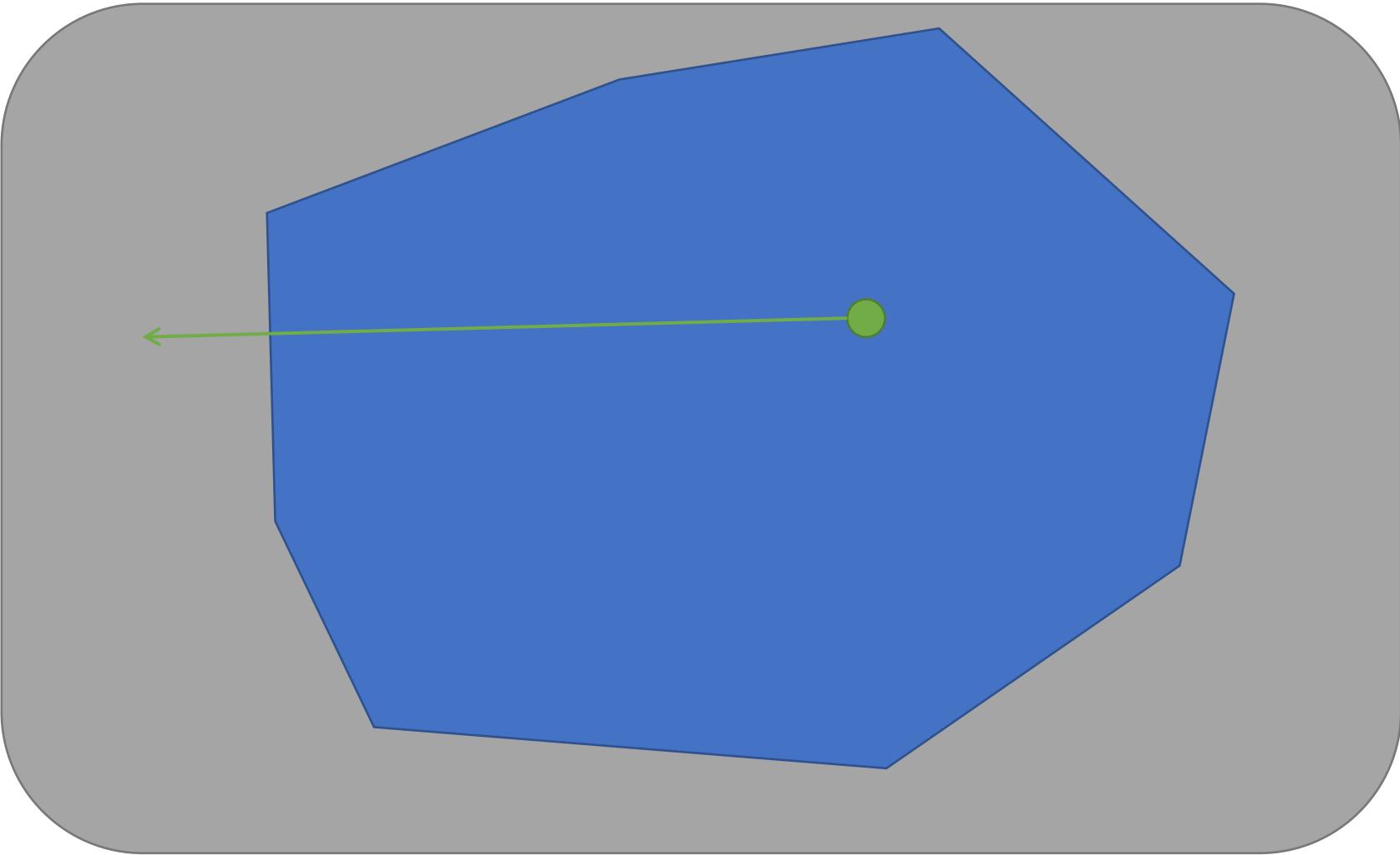
$$t = ((p_1 - e) \cdot n) / (d \cdot n)$$

- If $(d \cdot n) = 0$ the ray is parallel to the plane

General Polygon Intersection

- We have the intersection point (substitute t into the ray equation)
- Project the polygon and intersection point onto one of the coordinate planes
- Draw a line from the intersection point to infinity and count the number of times it intersects a polygon edge
- If the count is odd, the point is inside the polygon

1 intersection: inside



Intersection

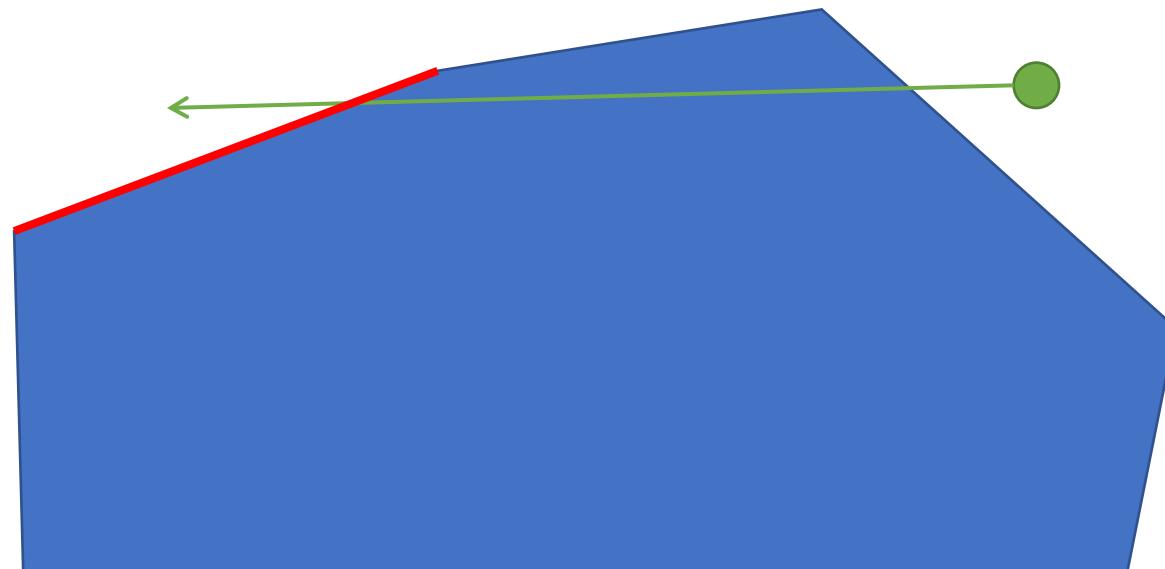
- Why does this work?
- Each time the line intersects an edge it is either entering or leaving the polygon
- If we have an even number of intersections, there must be an equal number of entering and leaving, so the original point must be outside
- With an odd number we have entered more than we have left, so the original point must be inside the polygon

Which line should we use for testing?

- One good choice for a line is one parallel to the x axis, in this case the intersection calculation is easy (just test the y coordinates)
- If one end point is above the line and the other is below, plus they are on the right side of the intersection point, then we have an intersection
- Note, if the projection onto the x – y plane is a line, choose one of the other planes

Counting Edge Crossings

- Test segments with a line parallel to the x-axis
- 1 end point above, 1 below
- Both on the same side of the intersection point with the plane
- = the ray crosses an edge



Intersection

- There are more efficient computations in the case of triangles
- Over the years intersection techniques for most of the common objects and modeling techniques have been produced
- These techniques are not as efficient as the sphere and polygon techniques, so we need to be careful when we use them

Intersection

- Intersection efficiency is a major concern in ray tracing, we have a lot of rays and they all must potentially be compared to all the objects in the scene
- We need some way of making this more efficient
- A key observation is that most of the objects won't be hit by a particular ray, so we really want to perform non-intersection computations as quickly as possible
- We only need to compute the intersection point in rare cases

Bounding Volumes

- Most objects are quite complex, say many polygons, so if we could quickly eliminate them the computations would be quicker
- One way of doing this is to enclose them in a simpler shape
- For example we could enclose them in a sphere, and then test for intersecting the sphere

Bounding Volumes

- Intersecting a sphere is very efficient, we can eliminate hundreds if not thousands of polygons with one test
- Axis aligned bounding boxes can also be used, a rectangular prism where the faces are parallel to the coordinate planes
- In this special case the intersection computation are also very fast

Bounding Volumes

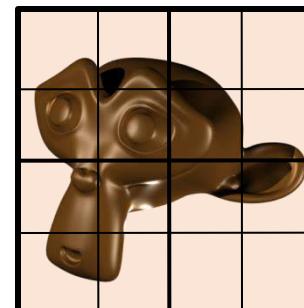
- Works best when the bounding volumes closely contains the object, very little space without polygons
- Reduces possibility of false positives
- Can add bounding volumes to modeling hierarchy, produces a hierarchy of bounding volumes
- Can give a much better fit

Accelerating Raytracing

- highly parallelizable technique
 - today's processors are dual-core or quad-cores
 - GPUs are even more parallelized (pipelines)
- space partitioning
 - bounding volume hierarchies
 - binary space partitioning
 - uniform subdivision grids (octrees)



© nVidia



Space Partitioning

- We want the closest intersection, the one closest to the start of the ray
- It is more likely that this intersection will occur close to the start of the ray, not far away from it

Space Partitioning

- Also the intersection will not occur behind the ray, and it won't occur in a direction that the ray is not going
- In fact the ray really only passes through a very small part of the 3D space, it misses most of this space so we really don't need to look there
- So we need some way of organizing the objects in the scene so we only look at the ones where the ray will pass and ignore the rest of them

Space Partitioning

- We want to test the objects in an intelligent order, the most likely ones first, but we need to make sure that we haven't missed any closer ones
- One approach would be to sort the objects on distance, and the start checking the closest one first
- This will work, but its very inefficient, since we would need to sort all of the objects for each ray

Space Partitioning

- But this does hint at a solution
- We can build a grid on top of the scene, and then assign the objects to the grid cells that they occupy
- Each cell should hold a small number of objects, ideally 3 to 5
- We start the intersection testing at the cell where the ray starts
- If it's a pixel ray, we find the first cell that the ray hits

Space Partitioning

- The objects in this cell are the closest ones to the ray origin, so if the ray hits one of them we are done
- We can't have a closer intersection, otherwise that object would be in the same cell
- If we don't have an intersection, we move to the next cell that the ray would travel through

Space Partitioning

- This approach only examines a small number of cells, if we have an $n \times n \times n$ grid in the worst case it will examine $n\sqrt{3}$ cells
- If the objects are evenly distributed over the cells this results in a significant time savings, on the order of $1/n^2$
- But in practice the objects are rarely evenly distributed

Space Partitioning

- There are several problems with this approach
- As we increase n to get more efficiency the data structure grows quite quickly, on the order of n^3
- As the cell sizes get smaller its likely that an object will appear in many cells, and if we are not careful we will do the intersection computation multiple times

Space Partitioning

- Many of the **cells will be empty**, so we will waste time tracing through them
- Computing the next cell to visit takes time, and if they are empty we are wasting it
- Thus there is a lower limit on the cell size, which limits the amount of speed up
- Several techniques have been developed to improve grid performance

Adaptive Space Partitioning

- Observation: we want the empty cells to be as large as possible so we can get through them quickly
- We can do this by using [an adaptive grid where the cell sizes can vary](#)
- This is a more complicated data structure and its implementation is not as efficient, but it can improve performance by reducing the number of empty cells

A simplified Raytracer

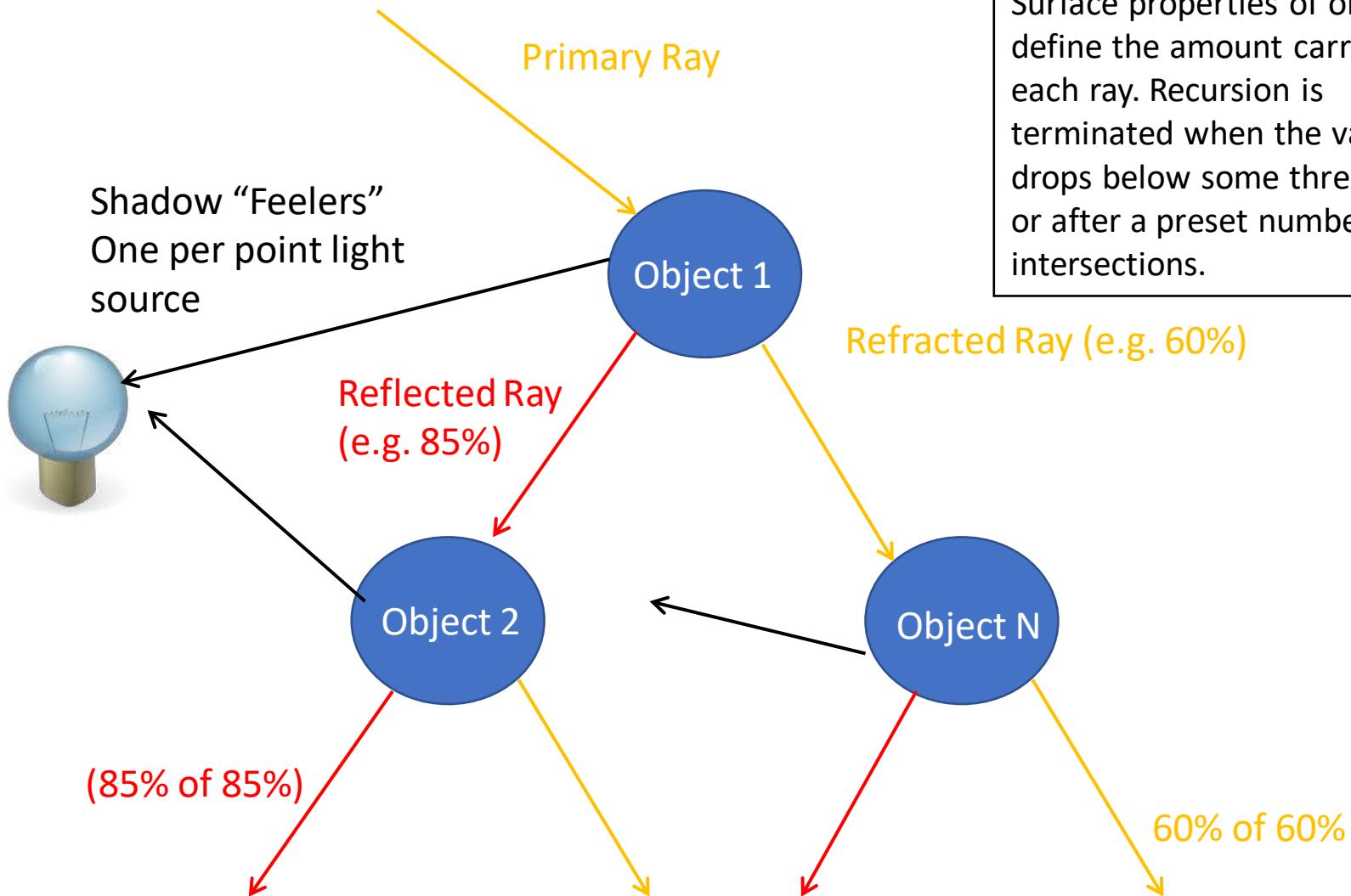
```
Procedure RayTrace()
{
    for y=1 to -1 by 2/(Y-1) do
        for x=1 to -1 by 2/(X-1) do {
            ray.origin=[0,0,0];
            ray.dir=[x, y, 1];
            PutPixel(Render(ray));
        }
}

Function Render(ray)
{
    object=QueryScene(ray);
    if (object==NIL) return backgroundColour;
    return Shade(object, ray);
}

Function QueryScene(ray)
{
    closest=NIL;
    distance=INFINITY;
    foreach object in sceneList do {
        if intersect(ray, object)
            if (object.distance<distance) {
                closest=object;
                distance=object.distance;
            }
    }
    return closest;
}
```

Recursive Raytracing

The Ray Tree



Hard Shadows

- A sharp transition from the shadow to the illuminated region
- Easy to produce
- Construct a shadow “feeler” ray for each light source

Shadow Ray

- Origin: the intersection of the incident ray and the object (point p)
- Direction: subtract p from the position of the light source and normalize
 - If the light source is infinitely far away we just use the light source direction as the ray direction
- Determine if the ray intersects another object before it reaches the light source
 - if it does, p is in shadow!

Shadows

- What do we do if the ray hits the light?
- Gather the 3 component contributions of the Phong model
- Note that this only works for point light sources and ones that are infinitely far away

Reflections

- For reflection we need to compute a **reflection direction**, this will be the direction of the new ray
 - Origin: the intersection of the incident ray and the object (point p)
 - Direction: reflected direction is a function of the original ray d and the surface normal:

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

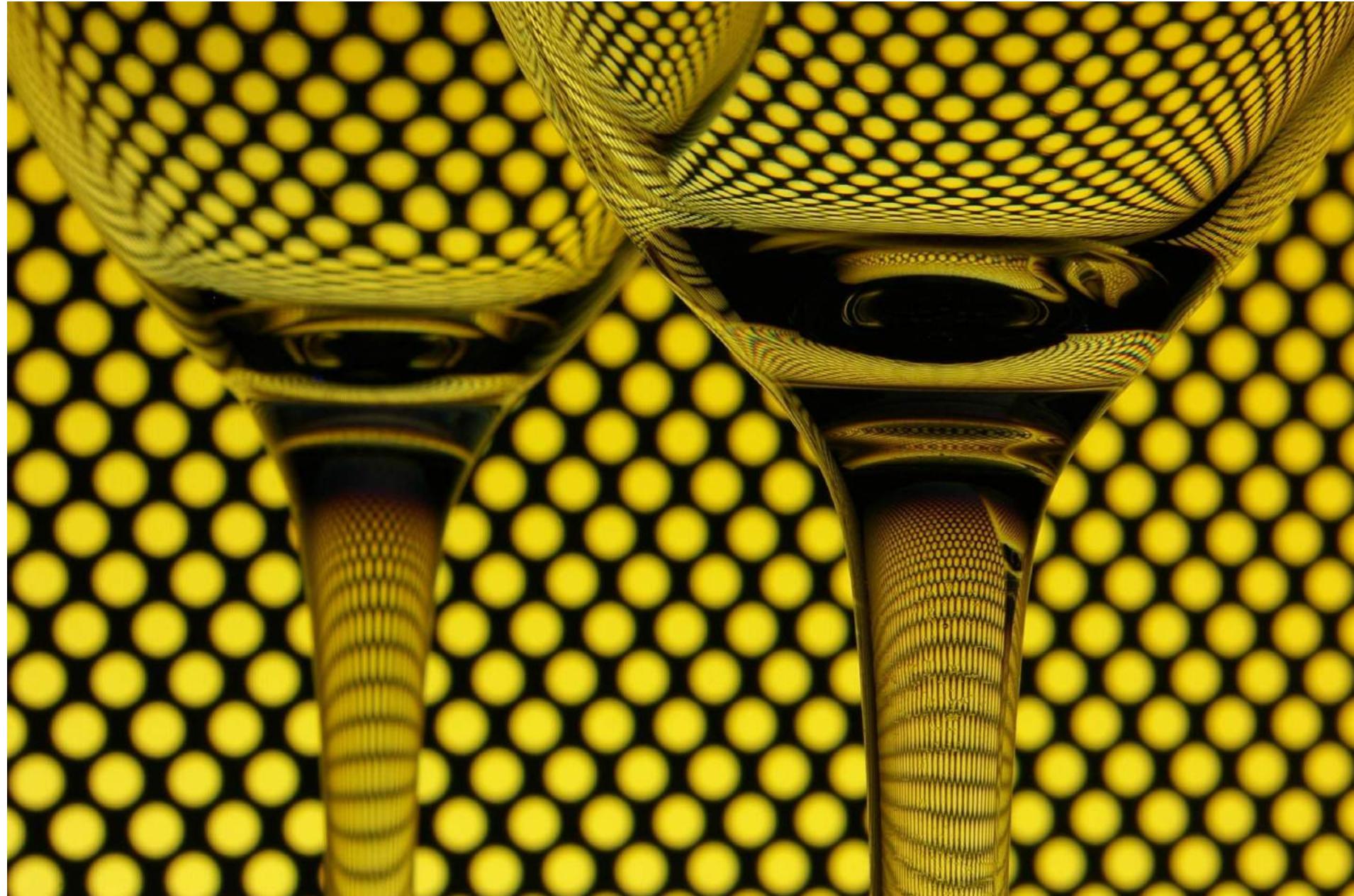
Note: In this case we are using a perfect mirror reflection, not the cone used in specular reflection

Reflection

- We then intersect this ray r with the closest object, make sure that $t > 0$
 - If $t=0$ we find the original intersection point!
- We then apply the ray tracing computation recursively to compute the colour of the point that the reflection ray hits

Reflection

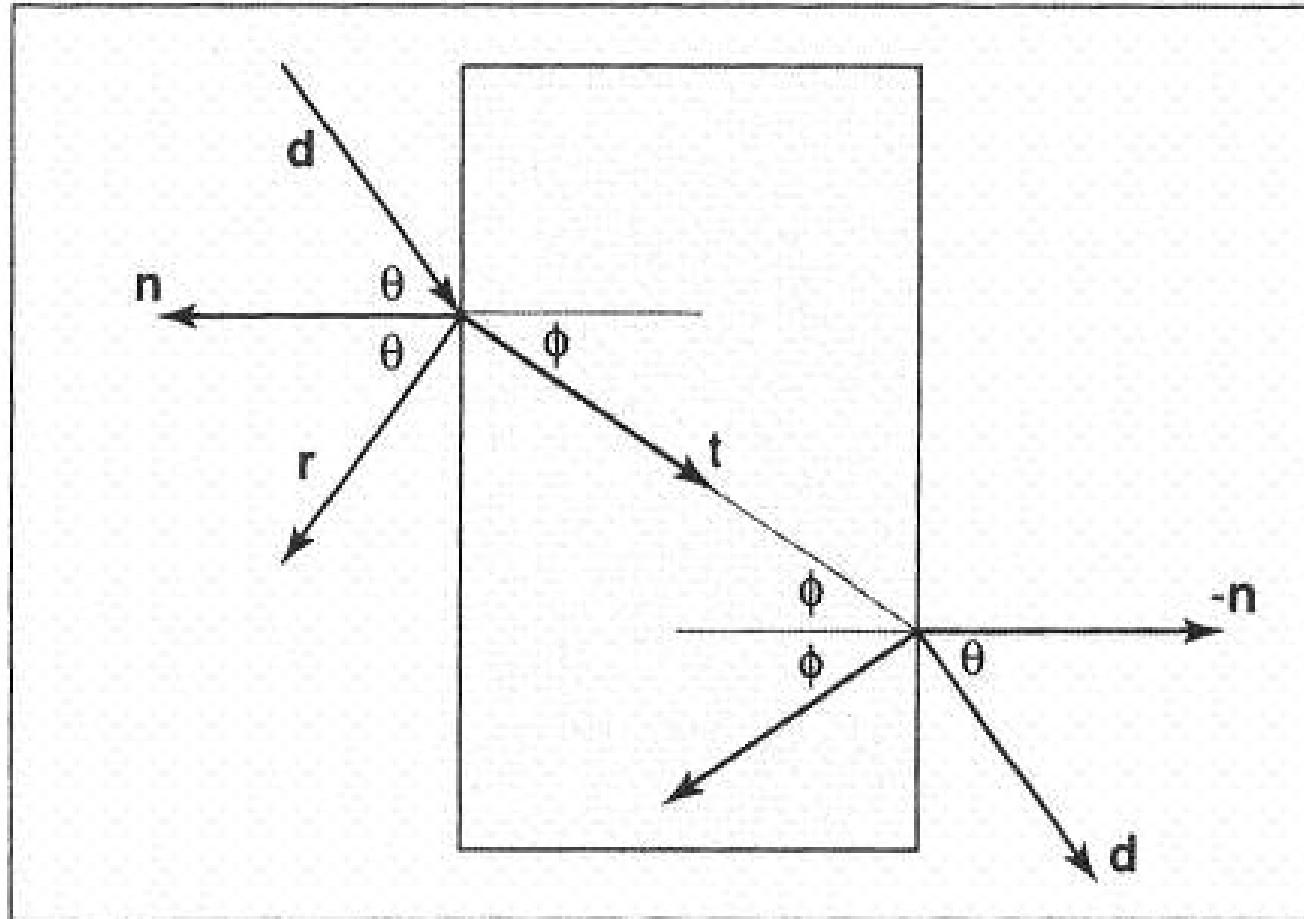
- What do we do with this colour?
- The reflection basically contributes to the specular light component, so we multiply the reflected colour by the specular reflection coefficient
- Since these coefficients are less than 1, each reflection contributes less to the overall value of the pixel, giving us a way to terminate the ray tracing



Refraction

- We can have objects that both reflect and refract light (are partially transparent), these objects are called dielectrics
- Example of these objects include glass, water and even air
- The next slide shows Snell's law for refraction

Refraction

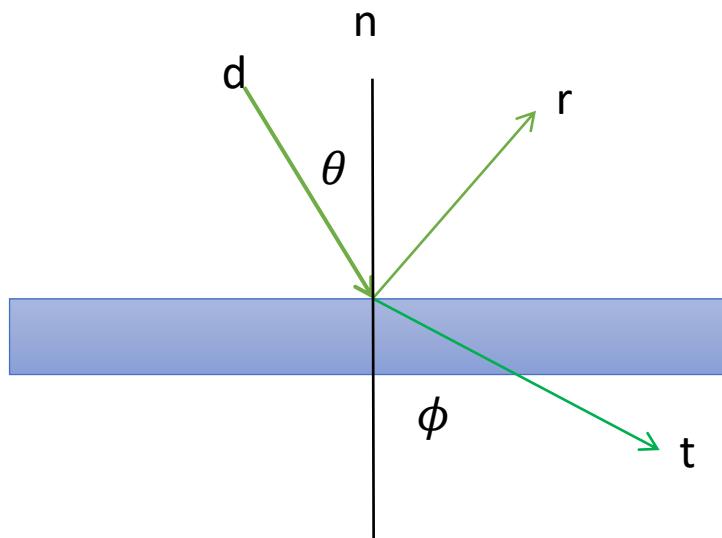


Refraction

- Snell's law is given by:

$$\eta \sin \theta = \eta_t \sin \phi$$

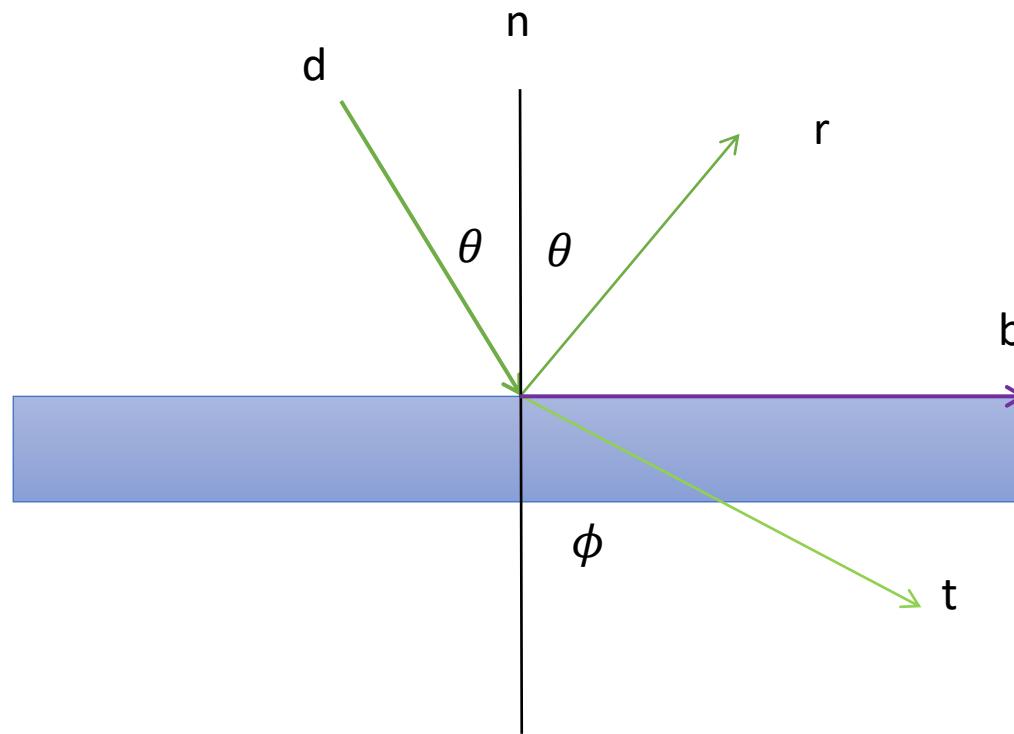
Where η and η_t are the indices of refraction in the two media, θ is known, solve for ϕ



Refraction

- This gives us an angle, but it **doesn't give us a direction vector t**
- To get this we need to consider the geometry in the next slide
- We can express our vector t in terms of two **basis vectors b and n :**
$$t = b \sin \phi - n \cos \phi$$
- The problem is we need to find the vector b

Refraction



Refraction

- Since \mathbf{d} lies in the same plane as \mathbf{b} , we can use it to solve for \mathbf{b} :
- $d = b \sin \theta - n \cos \theta$
- $b = \frac{d + n \cos \theta}{\sin \theta}$

...putting this together gives:

$$t = \frac{\eta(d - n(d \cdot n))}{\eta_t} - n \sqrt{1 - \frac{\eta^2(1 - (d \cdot n)^2)}{\eta_t^2}}$$

Refraction

- What happens if the term in the square root is negative?
- In this case we have **total internal reflection**, the ray is not refracted and instead is reflected from the surface
- If this occurs when the ray enters the material all the energy goes into reflection
- If it occurs when the ray is leaving the material, there is no refracted light contribution

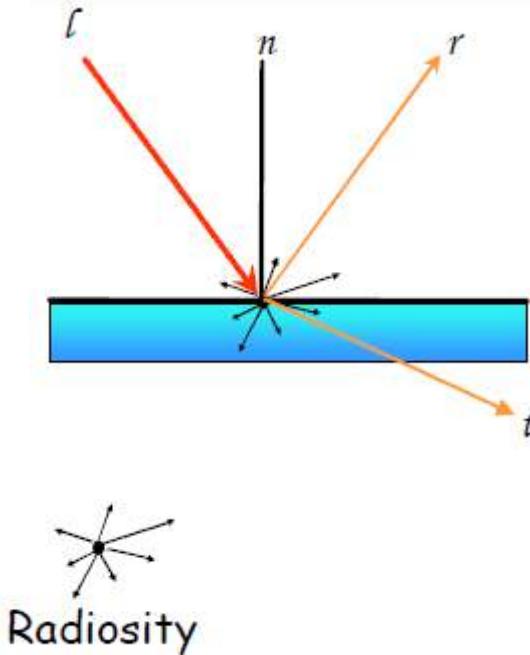
Refraction

- We now have the rays, but what about the **intensity** of the reflected and refracted light?
- The amount of reflection is a function of θ (the incident angle), and is given by the **Fresnel equations**
- These equations are not easy to solve, so we use an approximation

Types of Loss of Intensity

- Specular reflection r
- Transmission (refraction) t
- + diffuse reflection!!

This diffuse reflection is radiosity
(which we will ignore)!



Reflection Intensity

- The Schlick approximation is reasonably accurate and is given by:

$$R(\theta) = R_0 + (1 - R_0)(1 - \cos \theta)^5$$

$$R_0 = \left(\frac{\eta_t - 1}{\eta_t + 1} \right)^2$$

- $R(\theta)$ could be multiplied by a specular reflection coefficient to give the total specular reflection from the surface
- For our purposes, you may approximate with a constant factor, e.g. 40% reflection

Attenuation through Refraction

- Beer's Law gives us attenuation through a material.
Light intensity I is calculated as:

$$I(s) = I(0) e^{-\ln(a)s}$$

- Where s is the distance that the ray has traveled through the material
- One of the problems is that a is usually not known, so we usually need to guess at a values and do some trial and error experiments

A recursive Raytracer

```
Function Render(ray)
{
    object=QueryScene(ray);
    if (object==NIL) return backgroundColour;
    intensity=Shade(object, ray);
    if (object.reflect>0.0) intensity=intensity+
        object.reflect*Render(Reflect(object, ray));
    if (object.refract>0.0) intensity=intensity+
        object.refract*Render(Refract(object, ray));
    return intensity;
}

Function Shade(object, ray)
{
    intensity=AmbientIntensity;
    lightRay.org=object.intersect;
    foreach light in LightList do {
        lightRay.dir=light.dir;
        if (QueryScene(lightRay)=NIL))
            intensity=intensity+RefFunc(ray, object, light);
    }
    return intensity;
}
```

A recursive Raytracer

```
Function Render(ray)
{
    object=QueryScene(ray);
    if (object==NIL) return backgroundColour;
    intensity=Shade(object, ray);
    if (object.reflect>0.0) intensity=intensity+
        object.reflect*Render(Reflect(object, ray));
    if (object.refract>0.0) intensity=intensity+
        object.refract*Render(Refract(object, ray));
    return intensity;
}

Function Shade(object, ray)
{
    intensity=AmbientIntensity;
    lightRay.org=object.intersect;
    foreach light in LightList do {
        lightRay.dir=light.dir;
        if (QueryScene(lightRay)=NIL))
            intensity=intensity+RefFunc(ray, object, light);
    }
    return intensity;
}
```

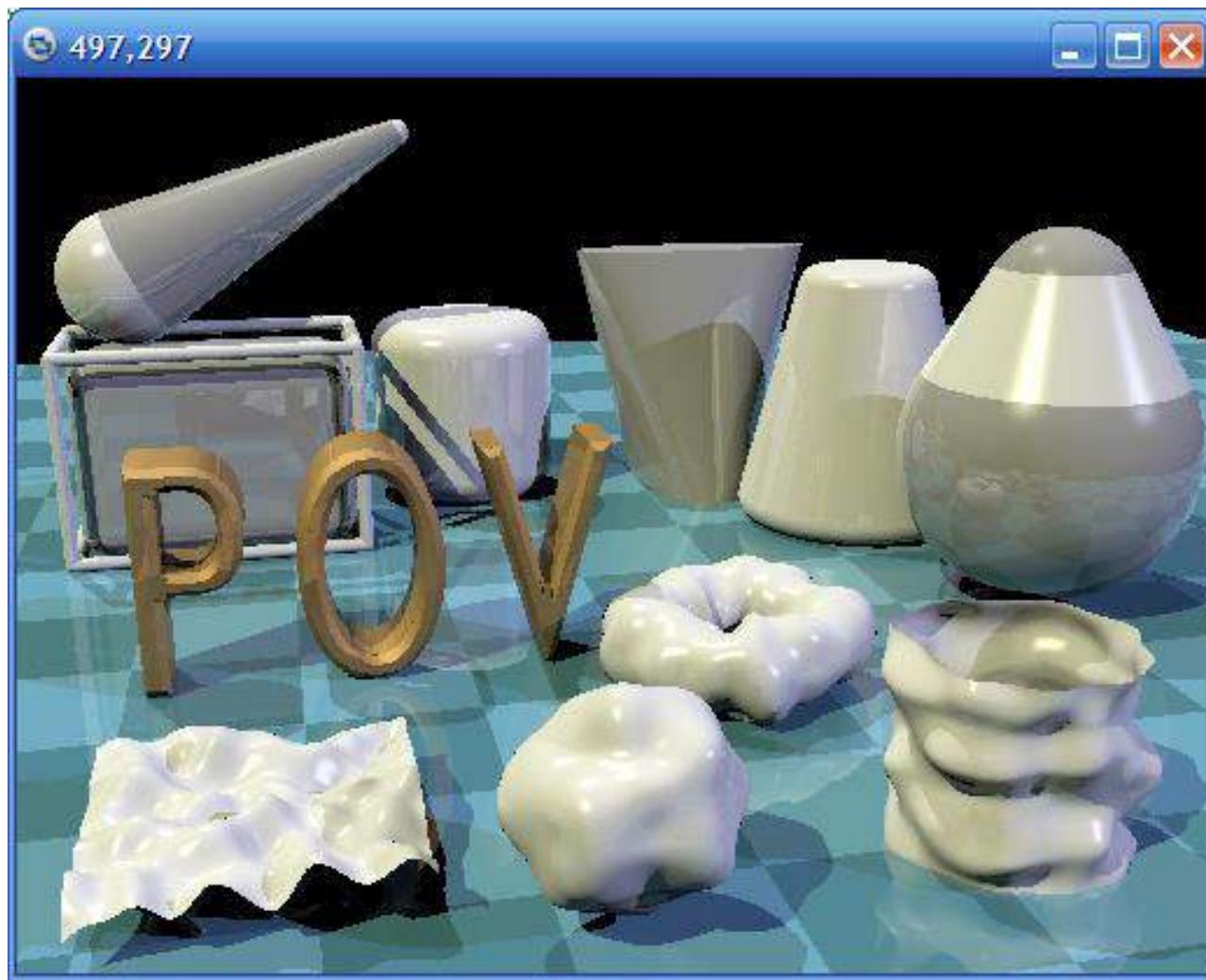
Distributed RayTracing

Advanced

Distributed Ray Tracing

- There are two main problems with classical ray tracing
 1. The images are too realistic, they look too clean and too sharp, the images are always in focus
 2. There is a lot of aliasing, particularly around the edges of objects and shadows
- Both of these problems are illustrated in the following slide

Distributed Ray Tracing



Distributed Ray Tracing

- The whole image looks just too real, too perfect, there is no dirt, no roughness
- Both of these problems are caused by point sampling:
 - We have only one ray per pixel
 - There is one ray to the light source
 - One reflected ray
 - One refracted ray

Distributed Ray Tracing

- Consider a line of pixels, where the first ray hits the same object
- The reflected ray for the first pixel may hit an object, while the reflected ray for the next pixel may just miss that object
- This can occur along the entire row of pixels, with every other reflected ray missing the object
- This will produce a very jagged effect

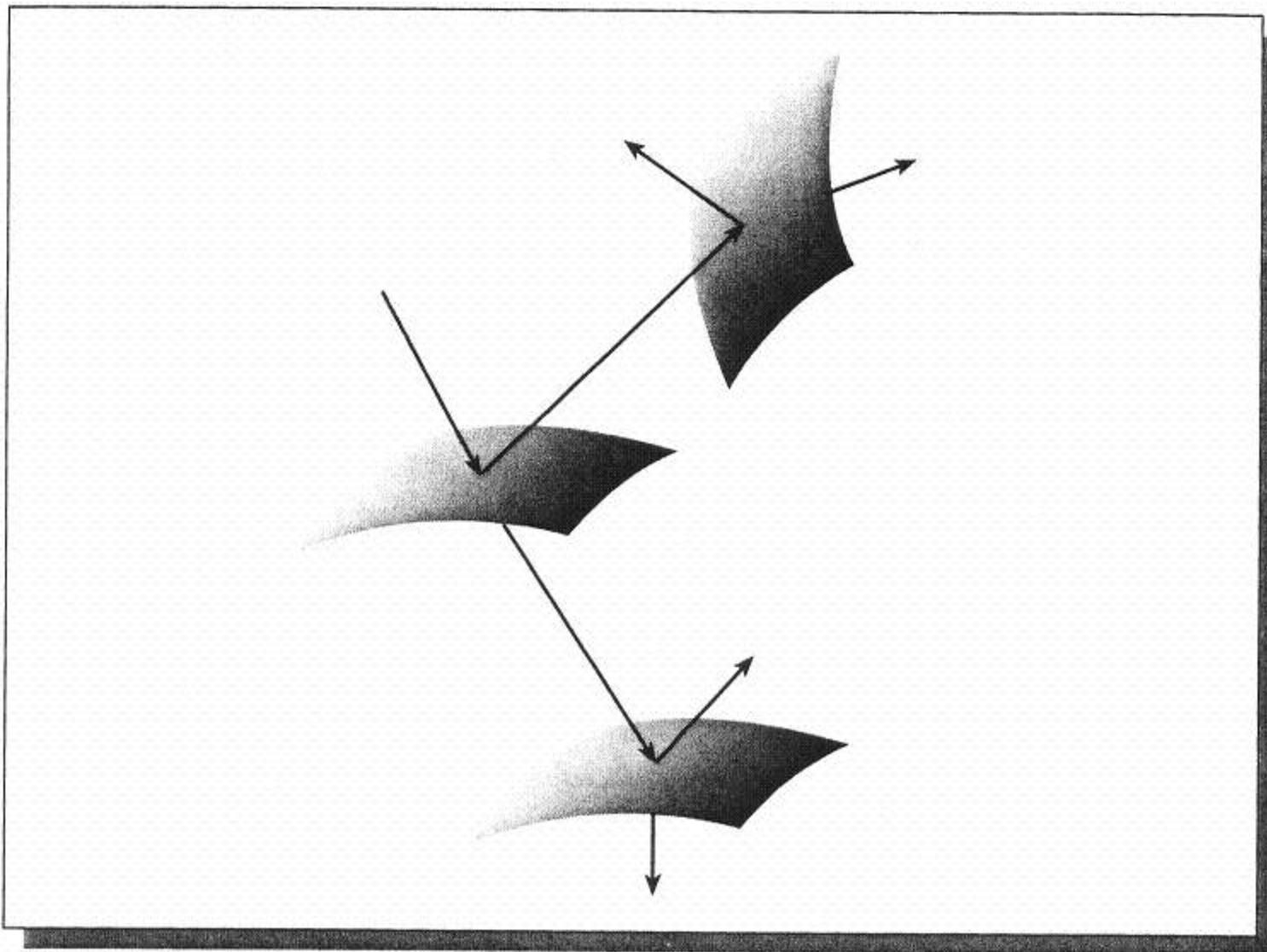
Distributed Ray Tracing

- The same thing happens with shadows
- There is a sharp transition from shadow to no shadow since all the lights are point sources and we are using just one shadow ray
- A number of solutions have been developed to these problems, but one of the most effective ones is distributed ray tracing

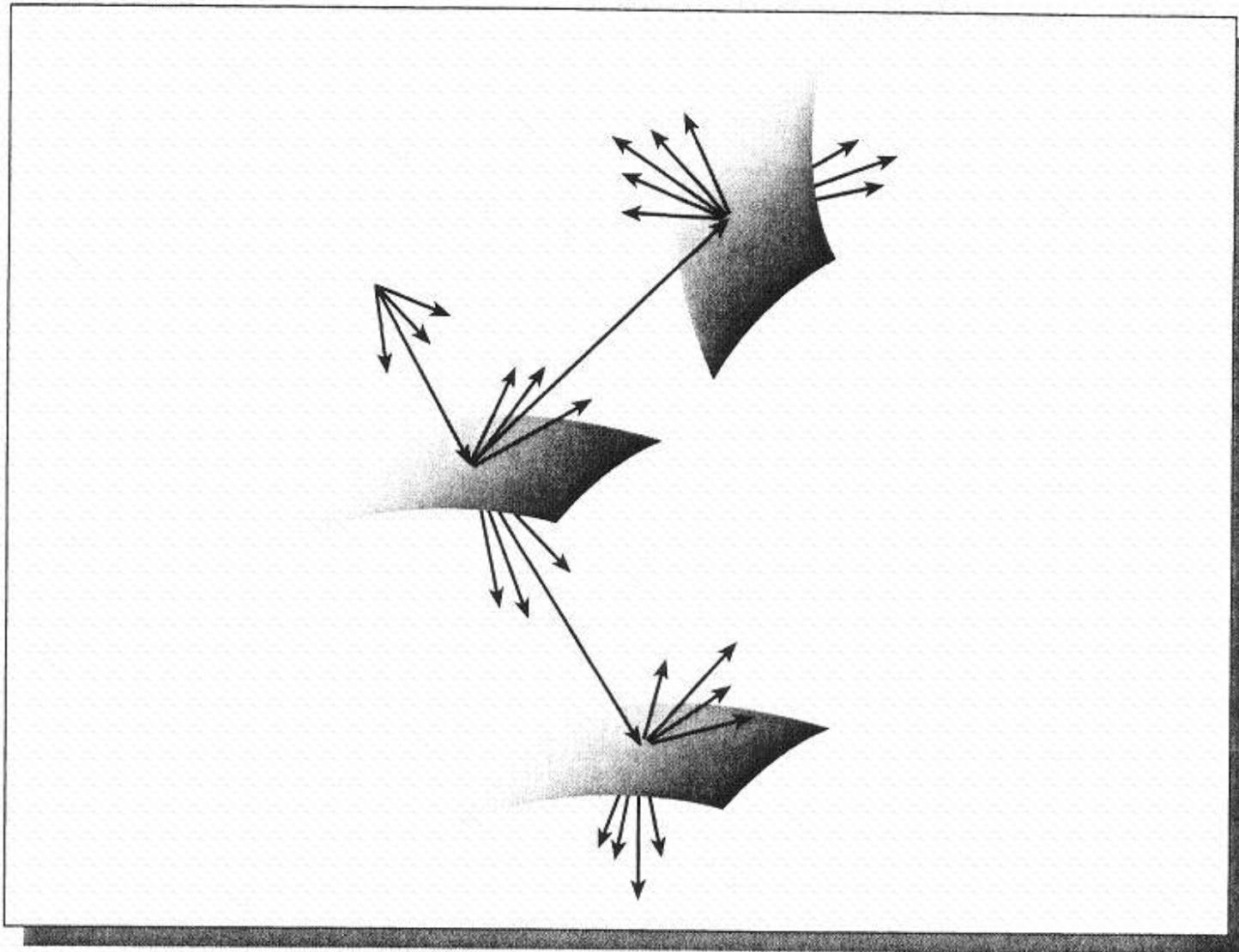
Distributed Ray Tracing

- The basic idea is to use more than one ray per pixel, but we need to be careful how we do this (see next slide)
- The easiest approach is to divide the pixel into smaller squares and then use one ray for each of these smaller squares
- This produces a smaller sampling grid, but its still regular, so it doesn't solve the problem

Classical Ray Tracing



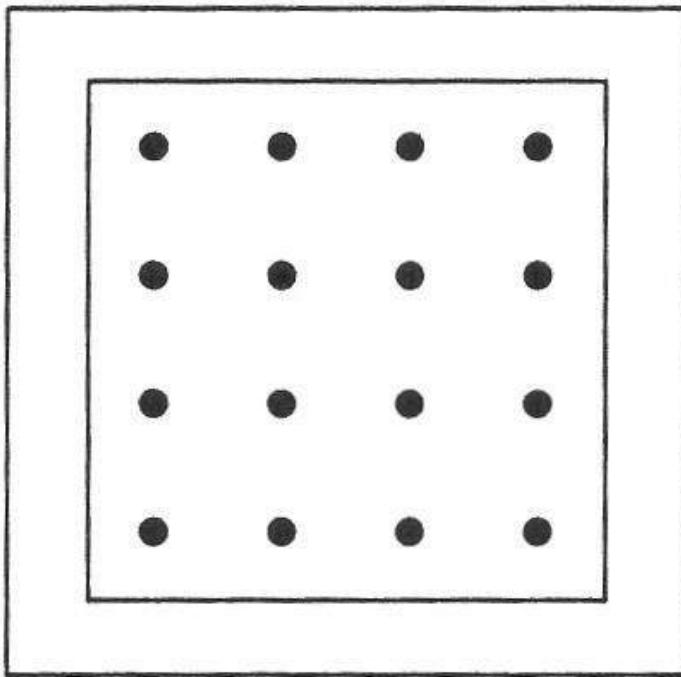
Distributed Ray Tracing



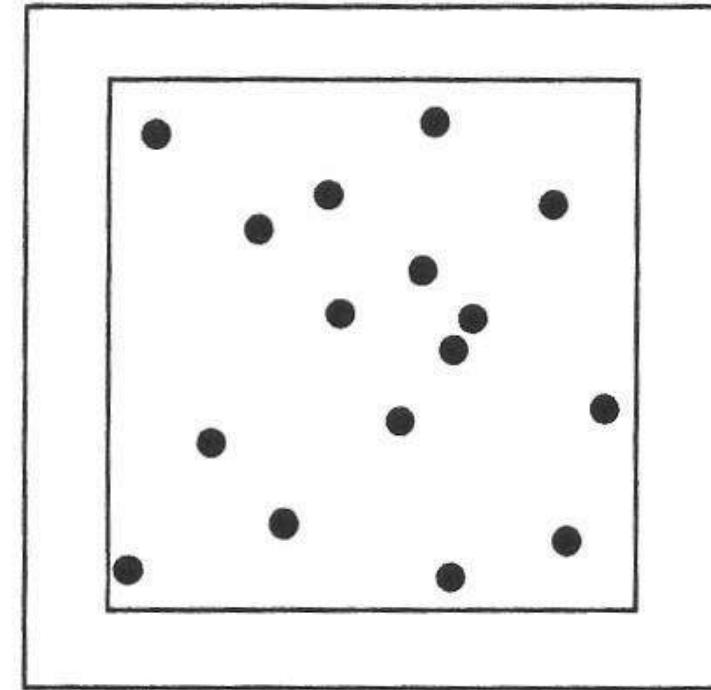
Distributed Ray Tracing

- The next approach is to randomly distribute the rays through the pixel
- Each time that we generate a ray we select a random position within the pixel
- We use a large number of rays for each pixel and then average the results
- This approach is quite inefficient, we need to generate a large number of rays to get a good sample for each pixel

Distributed Ray Tracing



Uniform Grid



Pure Random Sampling

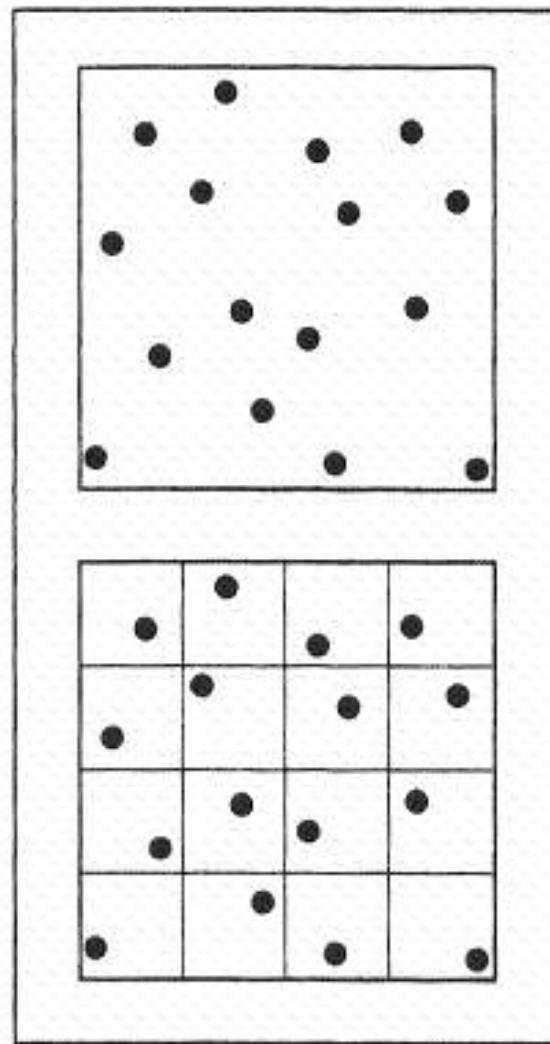
Distributed Ray Tracing

- The problem with random generation is it doesn't uniformly sample the pixel unless we use a lot of rays
- The grid sampling approach uniformly samples the pixel, but really doesn't solve our problem
- The solution is to combine the two approaches to get the advantages of both approaches

Distributed Ray Tracing

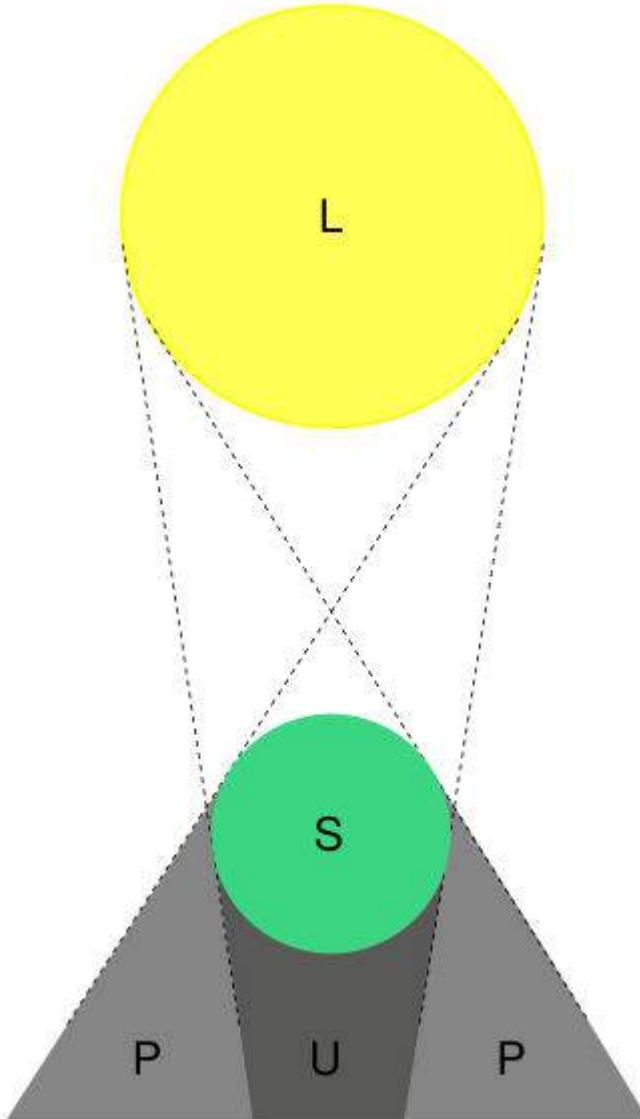
- Divide the pixel into a grid of smaller cells, then [within each cell](#) randomly generate a random position for the ray
 - The grid ensures that the entire pixel is sampled in a uniform way with a relatively small number of rays
 - The random position within each cell breaks down the uniform structure so the aliasing isn't as noticeable
- This approach adds a bit of noise in exchange for eliminating the aliasing
- Usually a 5x5 grid within each pixel is good enough to produce a reasonable image

Distributed Ray Tracing



Soft Shadows

- We rarely find sharp shadows in the real world, the shadows we see have a soft transition
- This transition region is called the penumbra, while the solid shadow area is called the umbra



Soft Shadows

- We can use distributed ray tracing to solve this problem as well
 - The light source must have a finite area, **it cannot be a single point**
 - The surface of the light source is divided into a number of cells, similar to what we did with pixels

Distributed Ray Tracing

- Instead of using one shadow ray we now use **multiple shadow rays**, one for each of the cells on the light source
- When we shot the ray we generate a random position within one of the cells
- The values from the different shadow rays are then averaged to produce the shadow value for the point

Distributed Ray Tracing

- If a point is in the umbra all of the shadow rays will be blocked and we will get the same result as before
- If the point is in the penumbra some of the shadow rays will be blocked and others will reach the light source
- In this case the average value will not be as dark as a shadow, so we will get a penumbra effect

Distributed Ray Tracing

- There are a number of other effects we can get using this approach including:
 - Depth of field or focus
 - Glossy reflections
 - Motion blur
- Note that all of these will require the use of even more rays

Efficiency of Distributed Ray Tracing

- One of the problems with distributed ray tracing is the **number of rays rapidly increases**
- If we use just 25 rays per pixel and just one light source with 25 cells we increase the number of rays by a factor of at least 625, even more if there are multiple objects to shadow
- The problem becomes worse if there are more light sources
- If we add other effects the number of rays grows even faster, even exponentially

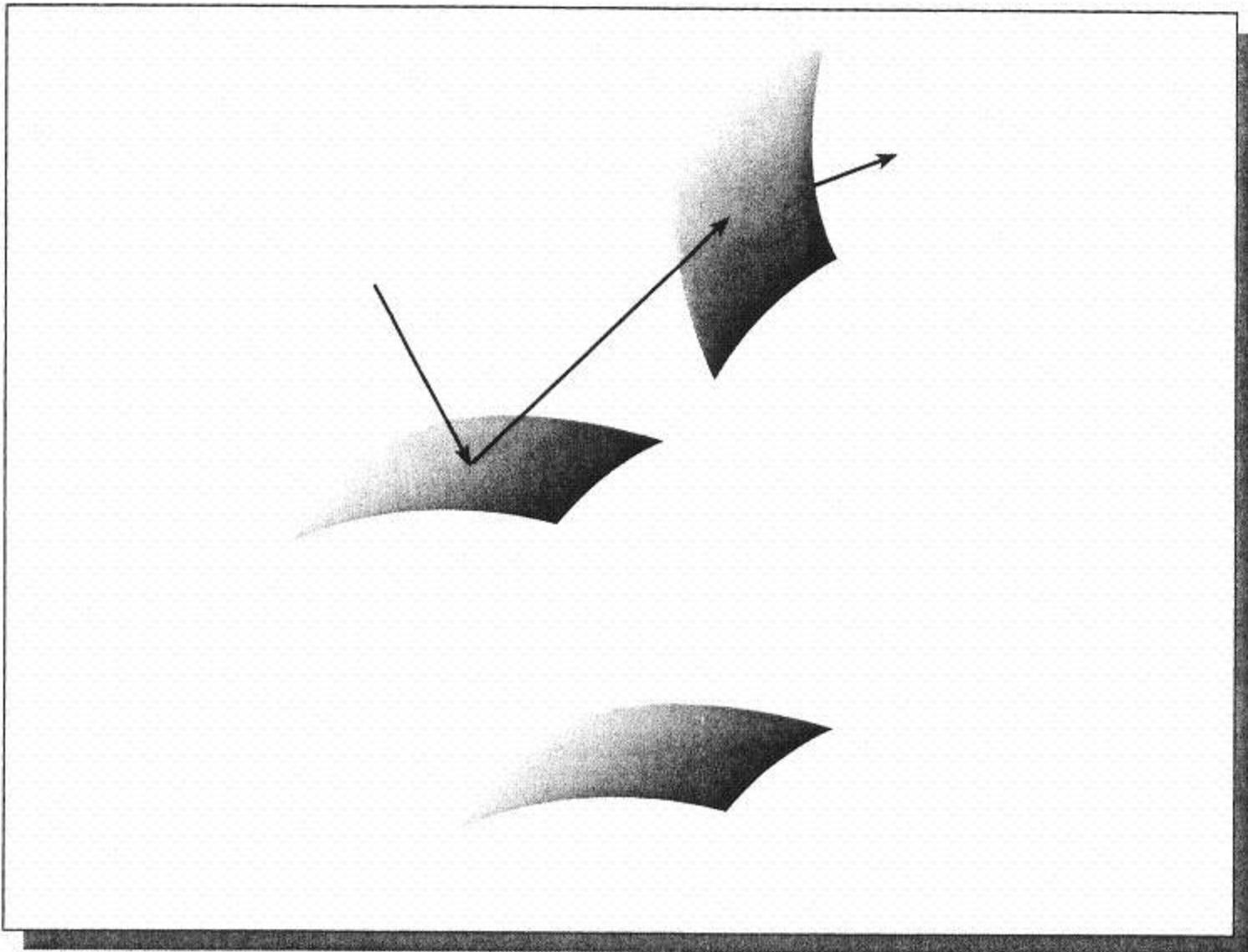
Path Tracing

- One alternative to distributed ray tracing is path tracing
- In this approach each ray follows one path through the scene (no branching!)
- We still generate a number of rays for each pixel, usually in the range of several hundred to thousands

Path Tracing

- When the ray hits an object we randomly decide to generate a shadow, reflection or refraction ray
- We then randomly generate a direction for this ray using the same techniques as distributed ray tracing
- This continues until the ray leaves the scene or has explored enough levels

Path Tracing



Path Tracing

- The main advantage of path tracing is we don't get the explosive growth in rays at each intersection
- When we are deep in the tree using a large number of rays at each intersection may not be very useful, due to the small contribution they make to the result
- But, this is the point where we generate the largest number of rays

Path Tracing

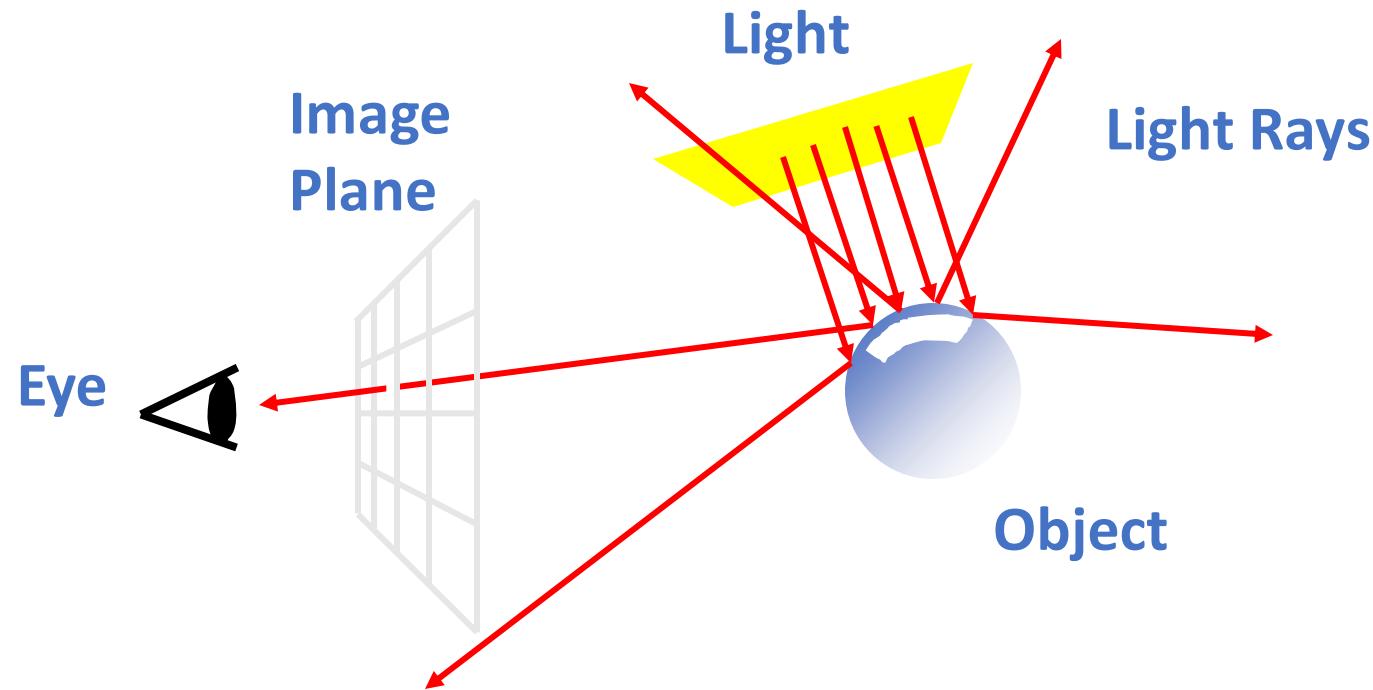
- With path tracing its easy to control the number of rays
- Once the initial ray is generated no additional rays are generated, so it is safe to generate a large number of rays for each pixel
- Since each ray only explores one path through the ray tree we need to generate a large number of them in order to get a good average value
- But how do we know how many rays to generate?

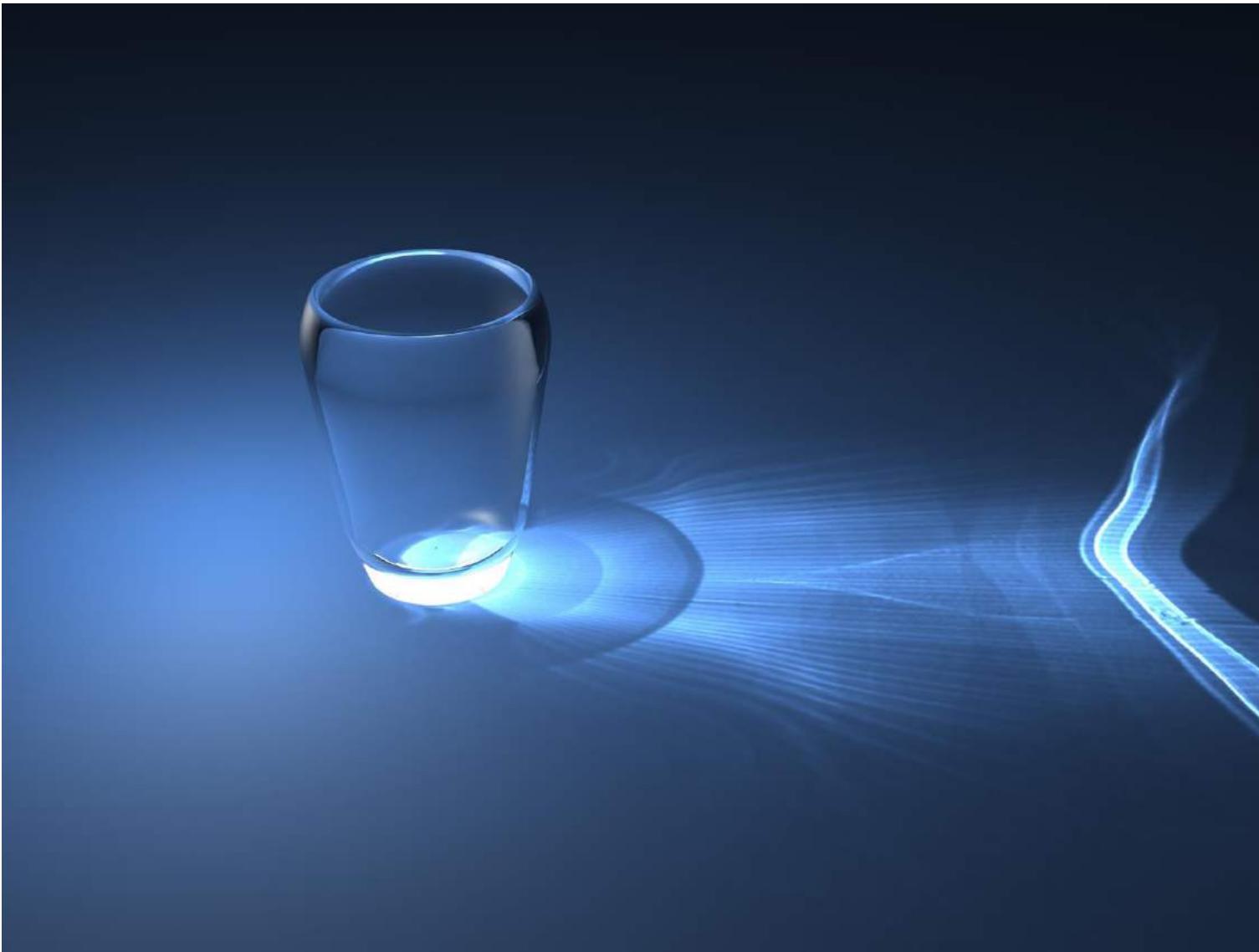
Path Tracing

- We are basically doing a Monte Carlo simulation of light transfer
- We know with Monte Carlo simulations we need to generate a large number of samples
- We also know that there are ways for controlling the sampling, so we can do this reasonably efficiently

“Forward” Ray-Tracing

- Previously “Lots of work for little return”
- But: useful for “caustics”





<http://donbertone.deviantart.com/art/Caustics-25195916>

Summary

- Recursive Ray Tracing: Shadows, Refraction & Reflection
- Distributed Ray Tracing and Path Tracing

Next Class

- Texture Mapping

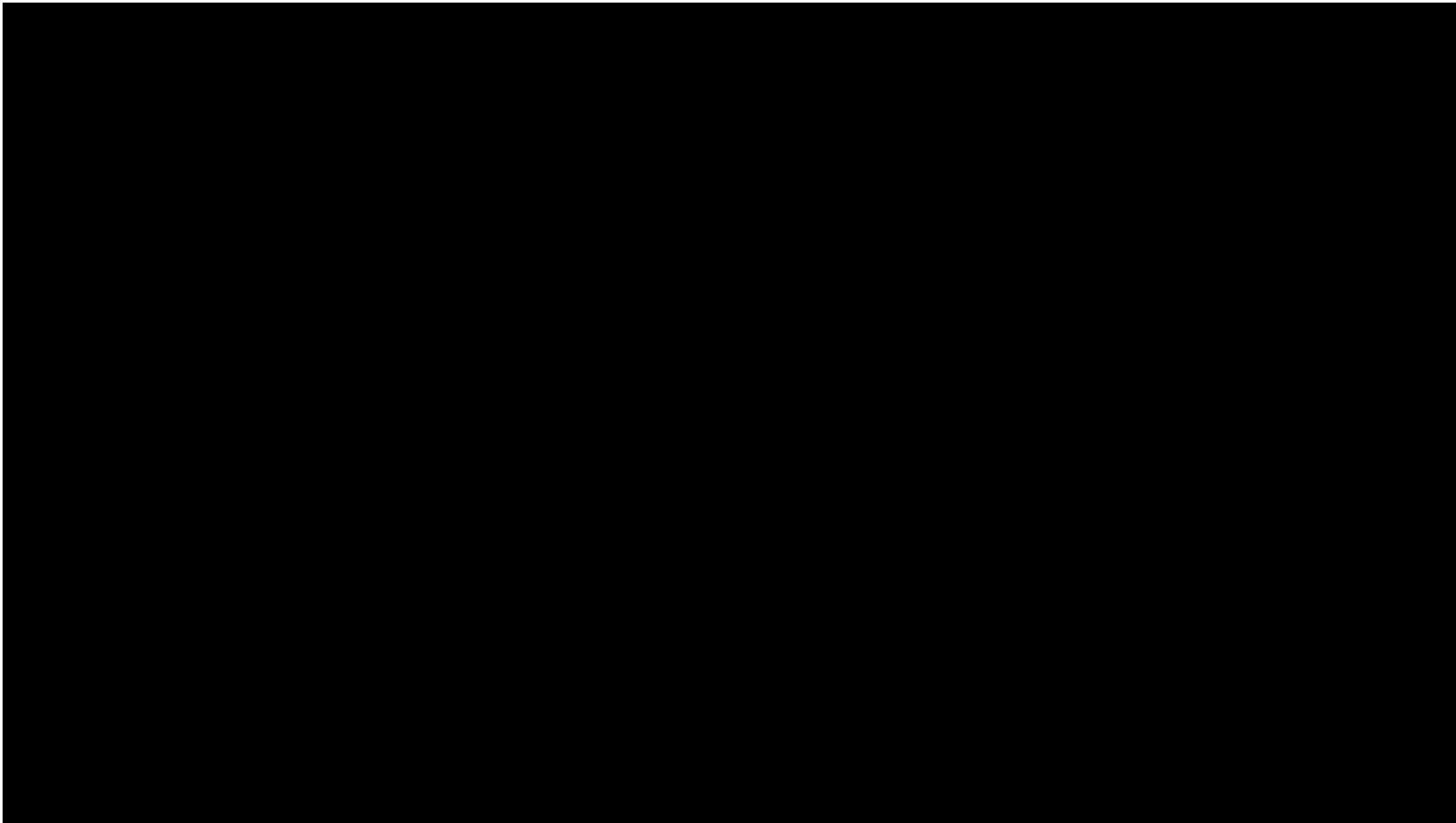
CSCI 3090

Texture Mapping

Part One

Mark Green
Faculty of Science
Ontario Tech

Caminandes



Goals

- By the end of today's class, you will be equipped to:
 - Use texture maps in your OpenGL programs
 - Describe the different methods of generating textures
 - Define mipmaps

Green Frog

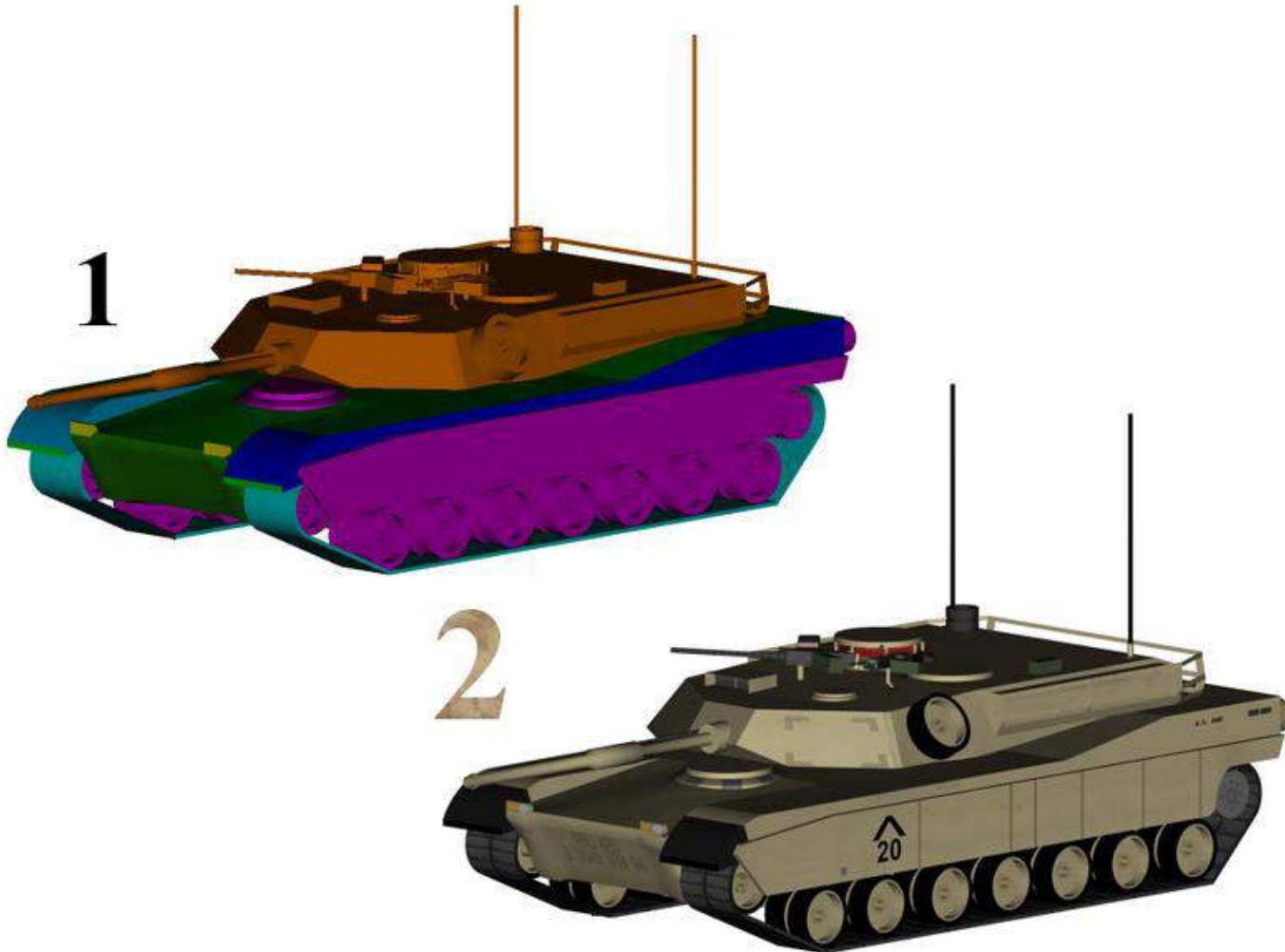


2006 © Massimo Righi

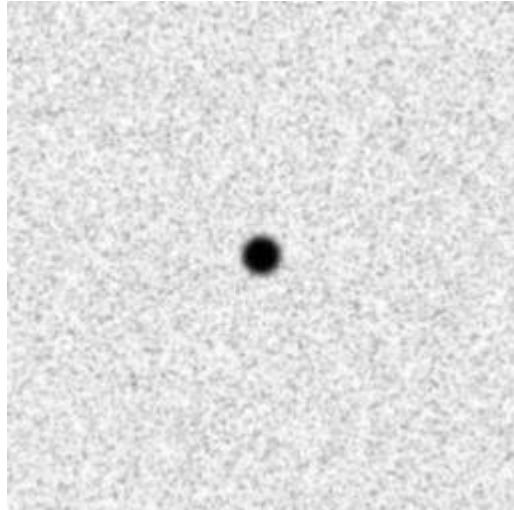
Texture Mapping

- We want our object to look like something in the real world...
- Why not just take a picture of the real world object and paste it onto our model?

Surface Colour



Bump Mapping



Surface Colour



Texture Mapping

- If we were modeling a brick wall we could model each brick individually, but this would take a lot of time
- We could just use a small number of polygons and map a picture of a brick wall onto them



Texture Mapping

- We don't take a picture of the entire wall, we just take a picture of a portion of it, say a few dozen bricks
- We can then repeat this pattern along the wall, if we are careful people won't notice that it's a repeating pattern
- Take your own images or buy commercial texture collections

Texture Mapping

- The real problem is **how do we map these images onto the surface of our model?**
- The images are 2D, they are made up of pixels, that is there are n_x by n_y pixels in the image
- The **surface** of our object is also 2D, so we need some way of mapping a point on the surface of our object to a point in the texture image

Texture Coordinates

- To do this we need a coordinate system on the object surface, these coordinates are called **texture coordinates**
- One of the easiest ways to do this is to **attach texture coordinates to the vertices**, these coordinates are then interpolated over the polygon the same way that normal vectors and z are
- We can also use expressions to generate texture coordinates, but this can be a bit tricky

Specifying Texture Coordinates

- We can specify texture coordinates the same was we specify vertex coordinates and normal vectors
- They are copied into the array buffer and we have an attribute variable in our vertex shader that receives them
- We typically use (u,v) for the texture coordinates (recall parameterization lesson!)
- Now we need to [map \$\(u,v\)\$ onto the texture image](#)

Texture Mapping

- Texture images can be of any size, and we might want to use different resolution textures with the same object
 - don't want (u,v) to be directly related to image size
- Instead we have a standard range for u and v , the uv point $(0,0)$ maps to the lower left corner of the image and the uv point $(1,1)$ maps to the upper right corner

Texture Mapping

- If we want one copy of the image to cover the entire surface of the object we restrict u and v to the range 0 to 1
- The texture can repeat if (u,v) goes outside of this range
 - Example: if u and v range from 0 to 2 we will have four copies of the texture
- For this to work seamlessly **the edges of the texture must match up** so we don't see the joins between the texture patches

Texture Mapping

- Now that we have our texture coordinates it looks like the mapping problem is simple
- We take the fractional part of u and v , and multiply that by n_x and n_y to get the pixel coordinates in the texture image
- While this is mathematically correct, it quite often doesn't produce a very good result

Texture Mapping

- We reference the texture image when we are computing the colour of a pixel
 - the (u,v) coordinates correspond to a pixel on the screen
- We then use a pixel in the texture image to colour the screen pixel
- If the screen pixel and the texture pixel are about the same size this works okay, but if their sizes are very different we have problems

Screen Pixels << Texels

- Several screen pixels will map onto the same texture pixel, so they will all have the same colour
- This gives the image a blocky effect, since the texture pixels are magnified
- This problem occurs when the viewer gets too close to the object

Screen Pixels << Texels

- We can solve this problem by using a higher resolution texture, or preventing the user from getting so close (adjust the near plane)
- This can be a problem in games, good game and level design can solve this problem most of the time

Screen Pixels >> Texels

- One screen pixel covers several texels
- This causes nasty visual artifacts particularly when the user is moving or the object is animated
- Harder to solve

Aliasing

- If we just choose the closest texture pixel the image may not look right, and if the viewer or object moves the choice of pixel will change
- The surface will tend to “bubble” as the object moves
- This is an example to the problem of [aliasing](#), which is a major issue in computer graphics

Sampling

- In order to solve this problem we can't do a point sample of the texture
- We must **average all of the texture pixels that are covered by the screen pixel**
- This will produce a blurry image, but it will contain all the information that should be in the screen pixel
- This usually occurs when the objects are further away, therefore, this won't be noticed

Averaging is Expensive

- There is a problem here, averaging the texture pixels produces a good image, but its very expensive
- For each screen pixel we may need to average hundreds of texture pixels, which is hard to do in real time
- In high quality rendering the texture pixels are averaged carefully, but for interactive applications we can't afford to do this

Speed Up

- A number of techniques have been developed to make this more efficient
- In some cases the textures are blurred before they are used
- Another approach is to just average a few close pixels, usually 4
- These approaches work to a point, but once the size mismatch passes a certain point they no longer work

Mipmap

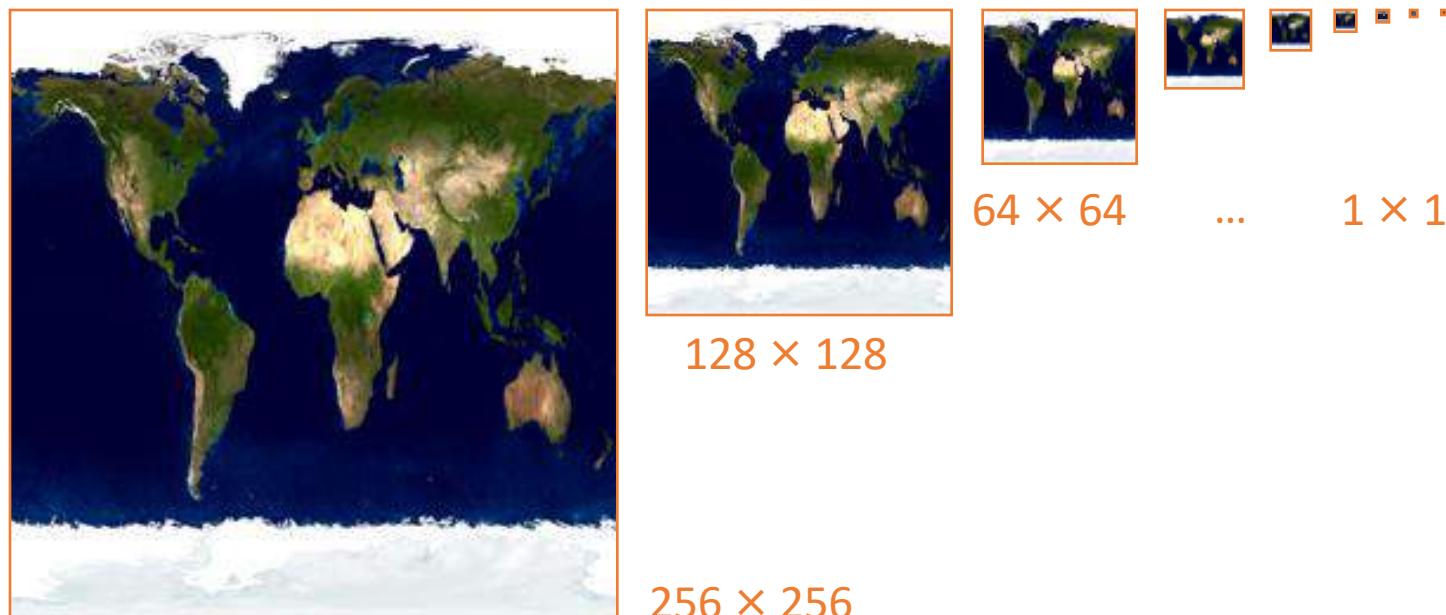
- One solution to this problem is to use a mipmap, which is basically a hierarchy of textures
- This is an approximation, it is very efficient and it works most of the time
- In order to use this technique the width and height of the texture image must be a power of two, they can be different powers, for example 128x64

Mipmap Hierarchy

- We create a hierarchy of texture maps in the following way
- The base of the hierarchy is the original image
- The next level is one quarter of the size, and each pixel is the average of 4 pixels in the original image
- We continue this process until one of the dimensions is 1

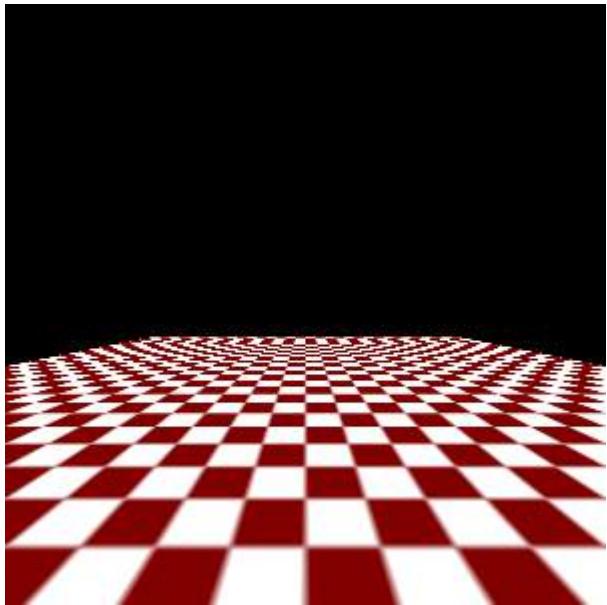
Texture Mapping: Mip Mapping

- optimal texture mapping (speed & quality):
texel size \approx pixel size
- idea: use stack of textures and select the most appropriate one w.r.t. situation

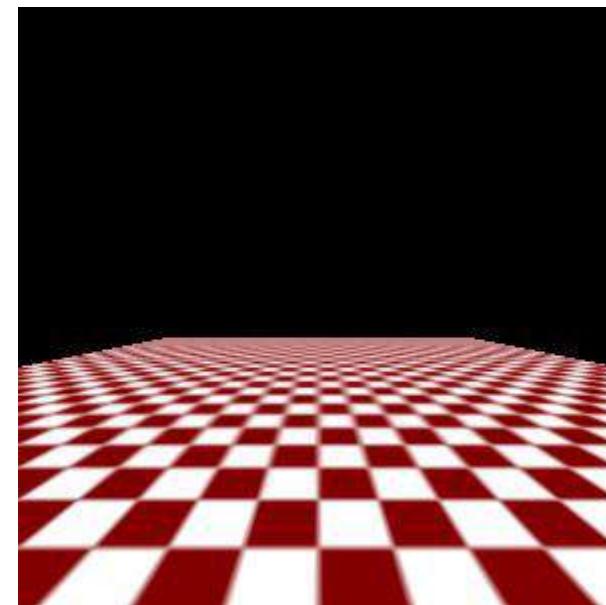


Mipmapping

No Mipmap



Mipmapping



Alternative Interpolations

- GL_NEAREST_MIPMAP_NEAREST
- GL_LINEAR_MIPMAP_NEAREST
- GL_NEAREST_MIPMAP_LINEAR
- GL_LINEAR_MIPMAP_LINEAR

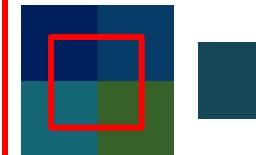
Texture Mapping: Mip Mapping

- optimal texture mapping (speed & quality):
texel size \approx pixel size
- interpolation: **GL_NEAREST_MIPMAP_NEAREST**
select nearest mipmap level, select nearest pixel of 2×2 neighbourhood



Texture Mapping: Mip Mapping

- optimal texture mapping (speed & quality):
texel size \approx pixel size
- interpolation: **GL_LINEAR_MIPMAP_NEAREST**
select nearest mipmap level, linearly interpolate pixel in 2×2 neighbourhood



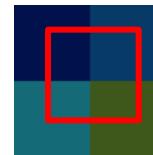
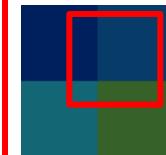
Texture Mapping: Mip Mapping

- optimal texture mapping (speed & quality):
texel size \approx pixel size
- interpolation: **GL_NEAREST_MIPMAP_LINEAR**
select 2 adjacent mipmap levels, select nearest pixel in 2×2 neighbourhoods,
then interpolate between them



Texture Mapping: Mip Mapping

- optimal texture mapping (speed & quality):
texel size \approx pixel size
- interpolation: **GL_LINEAR_MIPMAP_LINEAR**
select 2 adjacent mipmap levels, linearly interpolate pixel in 2×2 neighbourhoods,
then interpolate between them



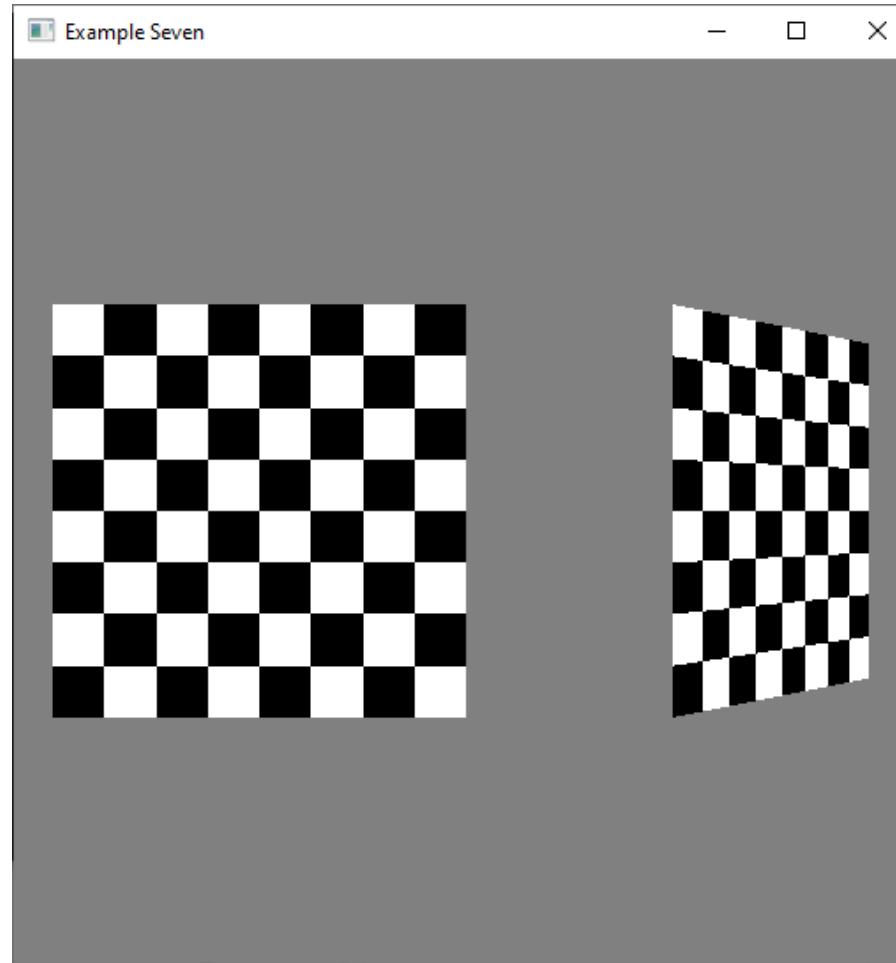
Mip Mapping

- Since the textures don't usually change at run time, we can pre-compute the mipmap and use it whenever the texture is required
- OpenGL has a procedure for doing this for us, so we really don't need to worry about the details
- A mipmap requires more storage than a simple texture (less than twice – why?), but it gives us much better quality
- This is a very common technique that is used in most programs

Example

- Now that we know a bit about textures lets see how they are done in OpenGL
- An old version of the Red book had a simple example program called checker.c
- I have updated this program to the new version of OpenGL, it is Example7 on Canvas
- We will take a look at this program, the next slide shows its output

Example



Procedural Texture

- This program draws two quadrilaterals (each with two triangles) and applies a checkerboard texture to both of them
- The checkerboard texture is created inside the program, this is somewhat unusual since we normally load textures from files
 - This is called a **procedural texture** and it saves all the issues involved with file formats and reading images, etc.

Example

```
void makeCheckImage(void)
{
    int i, j, c;

    for (i = 0; i < checkImageWidth; i++) {
        for (j = 0; j < checkImageHeight; j++) {
            c = (((((i&0x8)==0)^((j&0x8)==0)))*255;
            checkImage[i][j][0] = (GLubyte) c;
            checkImage[i][j][1] = (GLubyte) c;
            checkImage[i][j][2] = (GLubyte) c;
            checkImage[i][j][3] = (GLubyte) 255;
        }
    }
}
```

Example

- The array `checkImage` contains our texture, it is 64x64 pixels
- Each pixel has a red, green, blue and alpha component
- The alpha component is the amount of transparency for the pixel
- In this case we are using one byte to store each component, so the components go from 0 to 255

Example

- Now that we have our texture image, we construct a texture object to contain all of the information on the texture
- Texture objects allow us to have multiple textures in the same program and switch quickly between different textures
- Texture objects have integer names, like buffer objects and shaders, and the `glGenTextures()` procedure is used to create these names

Example

- The texture object is created in the init() procedure, the texture part of this will be explained here
- After the call to makeCheckImage we have two lines that create the texture object
- First create the integer name for the texture object and then bind it
- Note: this is similar to what we did for vertex data

Example

```
makeCheckImage();  
  
glGenTextures(1, &texName);  
 glBindTexture(GL_TEXTURE_2D, texName);  
  
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, checkImageWidth,  
             checkImageHeight, 0, GL_RGBA, GL_UNSIGNED_BYTE,  
             &checkImage[0][0][0]);  
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);  
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);  
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);  
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
```

Example

- The `glTexImage2D()` sets up the texture image
- The first parameter is the binding point, the type of texture
- The second parameter is the texture's level within the mipmap, in this case it's the base level
- The third parameter is the internal format for the texture, there are many options here that govern how the texture is used
 - For simple texturing `GL_RGBA` is a good choice

glTexImage2D

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0,  
GL_RGBA, GL_UNSIGNED_BYTE, &checkImage[0][0][0]);
```

- The next two parameters are the width and height of the image, this must match the actual size of the image that we are using
- The next parameter is the size of the border around the texture, it can be 0 or 1
- The seventh parameter is the format of the pixels in the image we are using to define the texture, this could be different from the internal format

glTexImage2D

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0,  
GL_RGBA, GL_UNSIGNED_BYTE, &checkImage[0][0][0]);
```

- The eighth parameter is the data type used for the pixel data
- In this case we are using unsigned bytes, but they could ints or floats
- The final parameter is a pointer to the actual image data
- There are a lot parameters, but most of them don't change very much from one application to the next

Example

- The next two procedures state what happens when the texture coordinates go outside the 0 to 1 range
- There are two options here:
 - `GL_CLAMP`: values outside the range are clamped to 0 or 1
 - `GL_REPEAT`: the texture is repeated when the coordinate goes outside the 0 to 1 range

Example

- The next two procedure specify what happens when the screen and texture pixel sizes aren't the same
- The MAG filter is used when the screen pixel is smaller than the texture pixel, the two possible values are:
 - GL_NEAREST: the nearest texture pixel is used
 - GL_LINEAR: a weighted average of the nearest 4 texture pixels is used

Example

- The MIN filter is used when the screen pixel is larger than the texture pixel
- The GL_NEAREST and GL_LINEAR values can be used here as well
- Recall there are also options for using a mipmap, but we will discuss how this can be done later

Example

- We now have all the information that we need for the texture object
- The next thing we need to do is create the object and add texture coordinates to it
- Texture coordinates are specified in essentially the same way as vertex coordinates and normal vectors
- The next slide shows the vertex coordinates and texture coordinates

Example

```
GLfloat vertices[] = {  
    -2.0, -1.0, 0.0,  
    -2.0, 1.0, 0.0,  
    0.0, 1.0, 0.0,  
    0.0, -1.0, 0.0,  
    1.0, -1.0, 0.0,  
    1.0, 1.0, 0.0,  
    2.41421, 1.0, -1.41421,  
    2.41421, -1.0, -1.41421};
```

```
GLfloat texCoords[] = {  
    0.0, 0.0,  
    0.0, 1.0,  
    1.0, 1.0,  
    1.0, 0.0,  
    0.0, 0.0,  
    0.0, 1.0,  
    1.0, 1.0,  
    1.0, 0.0};
```

Example

- After this we set up the array buffer with the vertex coordinates and texture coordinates
- The code for doing this is on the next slide
- This is basically the same as what we have seen before

Example

```
glGenBuffers(1, &vbuffer);
 glBindBuffer(GL_ARRAY_BUFFER, vbuffer);
 glBufferData(GL_ARRAY_BUFFER, sizeof(vertices)+sizeof(texCoords), NULL, GL_STATIC_DRAW);
 glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(vertices), vertices);
 glBufferSubData(GL_ARRAY_BUFFER, sizeof(vertices), sizeof(texCoords), texCoords);
```

Example

- Finally we need to map the attribute values to the attribute variables in our shader program
- This is shown on the next slide, again this is similar to what we have seen before

Example

```
vPosition = glGetAttribLocation(program, "vPosition");
glVertexAttribPointer(vPosition, 3, GL_FLOAT, GL_FALSE, 0, 0);
 glEnableVertexAttribArray(vPosition);
vTexture = glGetUniformLocation(program, "vTexture");
glVertexAttribPointer(vTexture, 2, GL_FLOAT, GL_FALSE, 0, (void*) sizeof(vertices));
 glEnableVertexAttribArray(vTexture);
```

Example

- The display function is shown on the next slide
- It looks basically the same as what we have seen before
- Note that we don't need to do anything special to indicate that we are using texture mapping

Example

```
void display (void) {  
    glm::mat4 view;  
    int modelViewLoc;  
  
    view = projection * glm::translate(glm::mat4(1.0), glm::vec3(0.0, 0.0, -3.6));  
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);  
    glUseProgram(program);  
    modelViewLoc = glGetUniformLocation(program,"modelView");  
    glUniformMatrix4fv(modelViewLoc, 1, 0, glm::value_ptr(view));  
  
    glBindTexture(GL_TEXTURE_2D, texName);  
    glBindVertexArray(objVAO);  
    glDrawElements(GL_TRIANGLES, 3*triangles, GL_UNSIGNED_INT, NULL);  
}
```

Example

- Now we turn to the vertex and fragment shaders, both of which are quite simple
- The vertex shader is shown on the next slide
- The vertex coordinates are processed in the standard way
- The texture coordinates are assigned to the texCoord variable which will be interpolated over the triangle and passed to the fragment program

Example

```
#version 330 core  
  
in vec4 vPosition;  
in vec2 vTexture;  
  
uniform mat4 modelView;  
  
out vec2 texCoord;  
  
void main(void) {  
    gl_Position = modelView * vPosition;  
    texCoord = vTexture;  
}
```

Example

- The fragment program is shown on the following slide
- We have a uniform variable `tex`, which is automatically bound to our texture
- This is a sampler variable that is used to sample a texture map
- The texture function uses this sampler and the texture coordinates to retrieve the texture value to be used for the current pixel

Example

```
#version 330 core  
  
in vec2 texCoord;  
uniform sampler2D tex;  
  
void main(void) {  
  
    gl_FragColor = texture(tex, texCoord);  
  
}
```

Discussion

- We have just seen the very basics of texture mapping
- There are many variations, we have just used the texture as the colour, we could use it in our light models as well
- There are many ways of computing texture coordinates
- We will explore some of these issues in the next lecture

Summary

- Texture mapping
 - Selecting texture coordinates
 - Different blending functions

Next Class

- Advanced Texture-Mapping

CSCI 3090

Texture Mapping

Part Two

Mark Green
Faculty of Science
Ontario Tech

Genesis Effect

- <https://youtu.be/Qe9qSLYK5q4>

Goals

- By the end of today's class, you will be equipped to:
 - Use different types of texture maps in your OpenGL programs
 - Describe the different methods of generating textures

File Based Textures

- Most of the time when we use an image for a texture it is read from a file, not computed in our program
- It is time to explore how we do that with OpenGL
- We will start by taking our vase object from example 6 and apply our checkboard texture to it
- This tests our ability to read texture coordinates from an OBJ file

File Based Textures

- Our vase model has texture coordinates, we just need to extract them and then copy them into our array buffer
- The code on the following slide shows how this is done
- This is similar to what we have done before
- Note: the texture coordinates are stored after the vertex coordinates and normal vectors

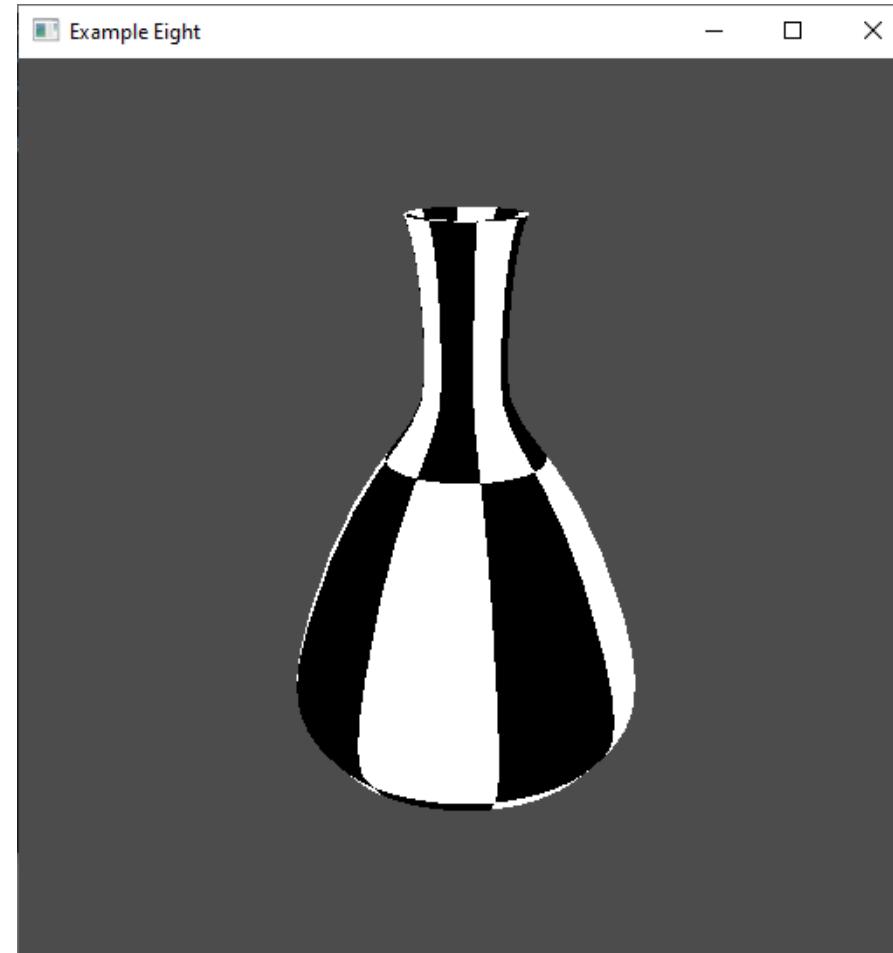
File Based Textures

```
nt = shapes[0].mesh.texcoords.size();
texcoords = new GLfloat[nt];
for(i=0; i<nt; i++) {
    texcoords[i] = shapes[0].mesh.texcoords[i];
}
...
glBufferData(GL_ARRAY_BUFFER, (nv+nn+nt)*sizeof(GLfloat), NULL,
GL_STATIC_DRAW);
...
glBufferSubData(GL_ARRAY_BUFFER, (nv+nn)*sizeof(GLfloat),
nt*sizeof(GLfloat), texcoords);
```

File Based Textures

- The texture is set up in the same way as the previous example
- The code is basically cut and paste
- The result that we get is shown on the following slide

File Based Textures



File Based Textures

- The texture coordinates are not ideal for this texture, we can clearly see the aliasing between the squares
- We can partly solve this problem by using GL_LINEAR for our MAG filter
- When we do this we get the image on the next slide

File Based Textures



Comparison



GL_NEAREST

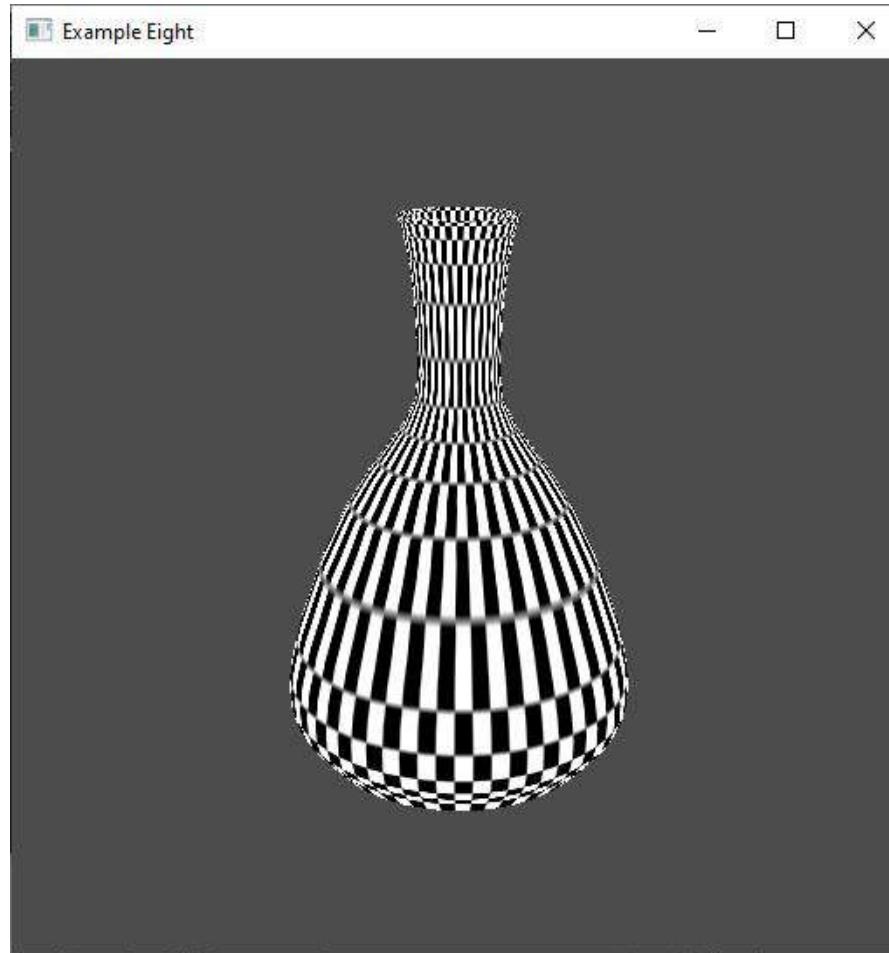


GL_LINEAR

File Based Textures

- Let's continue exploring this before we turn to reading image files
- In the vertex shader I multiply the texture coordinates by 8 and this produces the image on the next slide
- Notice the high level of aliasing around the top of the vase

File Based Textures



File Based Textures

- This cries out for some type of filtering, so let's try mipmaps
- We could construct the mipmaps on our own, which would be the best choice for high quality
- But, OpenGL provides a mipmap generation function, `glGenerateMipmap`, so we will use it
- The code is on the following slide

File Based Textures

```
glGenTextures(1, &texName);
```

```
glBindTexture(GL_TEXTURE_2D, texName);
```

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, checkImageWidth,  
            checkImageHeight, 0, GL_RGBA, GL_UNSIGNED_BYTE,  
            &checkImage[0][0][0]);
```

```
glGenerateMipmap(GL_TEXTURE_2D);
```

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
```

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

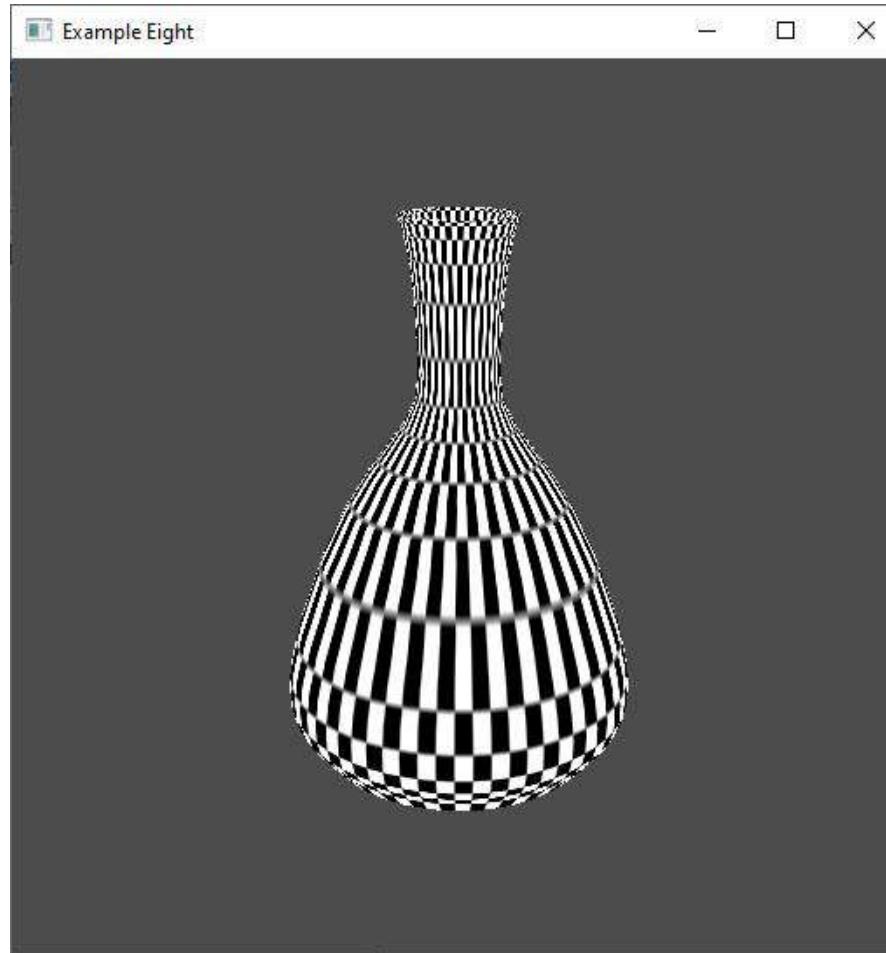
```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
```

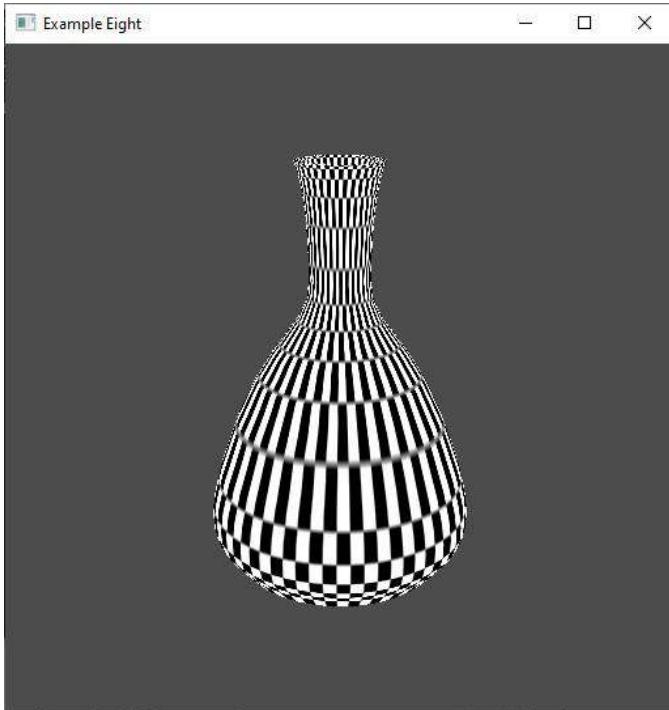
File Based Textures

- The two lines that are changed are highlighted in red
- The result is shown on the next slide
- Note how the severely aliased part of the vase has been replaced by a blur
- This is a difficult texture to deal with due to its high spatial frequencies

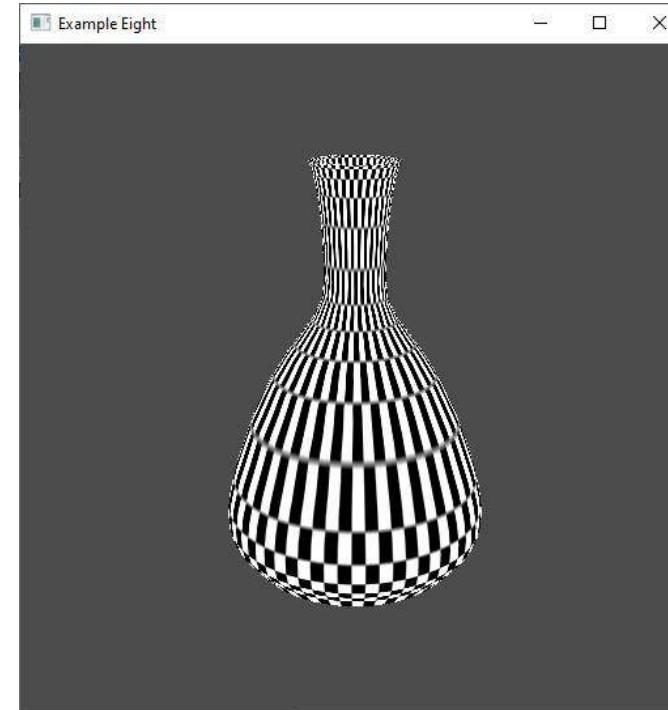
File Based Textures



Comparison



No Mipmap



Mipmap

File Based Textures

- Before moving on to reading files we should examine the vertex and fragment shaders
- The vertex shader is shown on the next slide, this is basically similar to what we have seen before
- The fragment shader is shown on the following slide, it is the same as the previous example

File Based Textures – Vertex Shader

```
in vec4 vPosition;
in vec3 vNormal;
in vec2 vTexcoord;

uniform mat4 modelView;
uniform mat4 projection;
uniform mat3 normalMat;
out vec2 texCoord;
out vec3 normal;

void main(void) {
    gl_Position = projection * modelView * vPosition;
    normal = normalMat * normal;
    texCoord = vTexcoord;
}
```

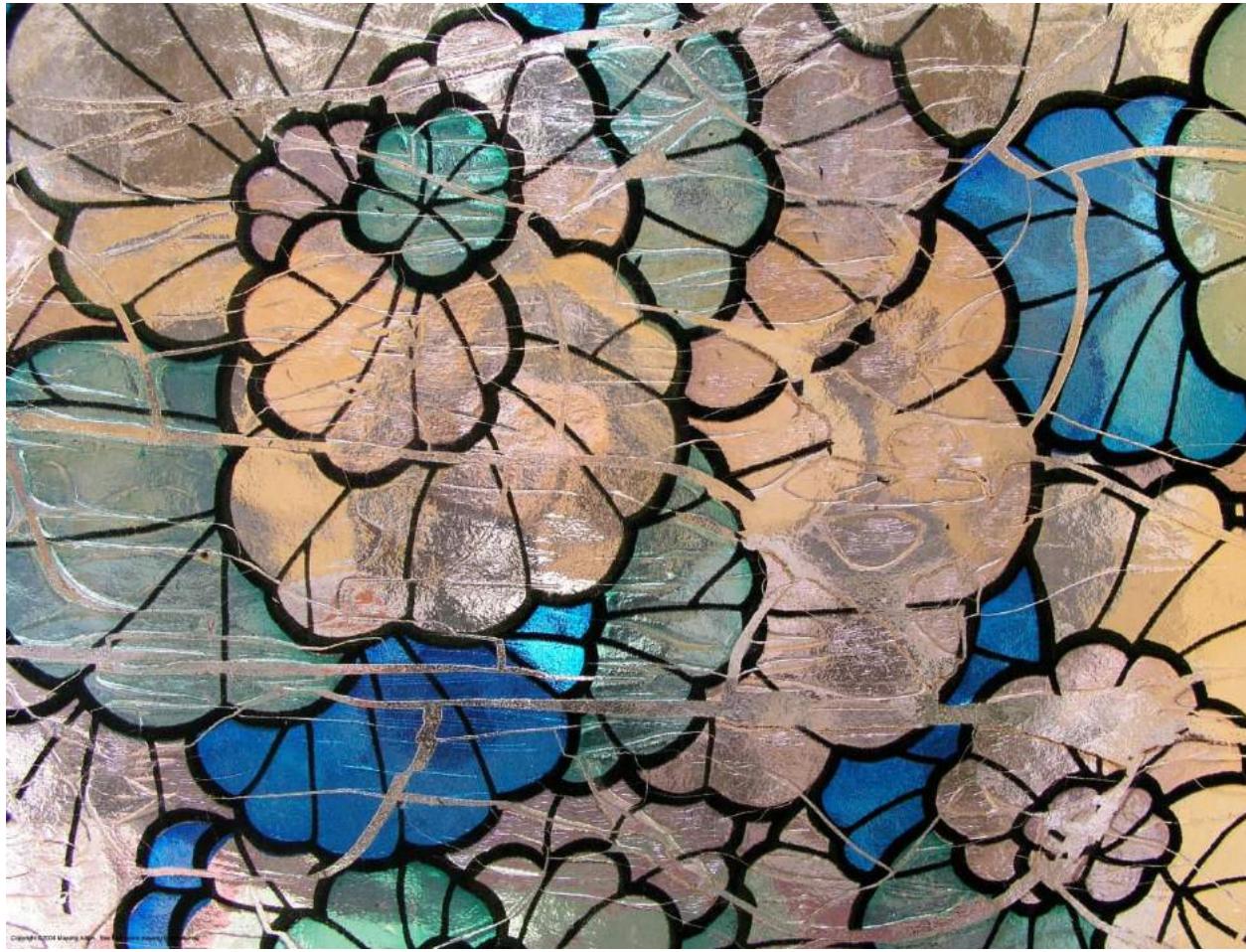
File Based Textures – Fragment Shader

```
in vec2 texCoord;  
uniform sampler2D tex;  
  
void main(void) {  
    gl_FragColor = texture(tex, texCoord);  
}
```

File Based Textures

- Vases usually don't look this way, we would like to have something a bit more natural
- The next slide shows a glass texture that would make our vase look more natural
- When we apply this texture to our vase we get the image on the following slide
- All we need to do is read this texture in and apply it
- The code for this part of the lecture is in example8.zip on Canvas

File Based Textures



File Based Textures



File Based Textures

- The texture is a JPEG file, so we will need to read it, but we don't want to write code for decoding JPEGs
- Instead we will use one of the standard image libraries, `Freelimage`
- This library doesn't have a lot of functionality (we don't need it), but it is very good at basic image manipulation

Freelimage

- You will find the standard binary distribution for Freelimage under resources on Canvas
- The documentation is there as well
- The source code is available from the Freelimage website:
<http://freeimage.sourceforge.net/>

File Based Textures

- Before reading the file need to examine how the image will be stored in memory
- Each pixel will be stored as three bytes, RGB, one after the other
- The pixels in one horizontal line are stored adjacent to each other
- The first pixel on the line will be the first 3 bytes, the next pixel will be the next 3 bytes, etc.

File Based Textures

- For each horizontal line we will have $3 * \text{width}$ bytes, where width is the width of the image
- There are height lines in the image, which are stored one after the other
- This is a packed representation of the image, and this is what OpenGL expects
- Unfortunately, this is not what FreeImage gives us

File Based Textures

- *Freelimage* allocates image memory so each horizontal line starts on a word boundary
- This is for performance reasons
- Our pixels are 3 bytes long, and the words are either 4 or 8 bytes, so it's unlikely that a horizontal line in our image will be an exact multiple of the word length
- *Freelimage* provides a way of handling this

File Based Textures

- For setting up a texture we need more than the bytes in the image, we also need its size
- Our image reading procedure will return a structure that contains all the information we need
- This is shown on the next slide
- The bytes field is the number of bytes in each pixel, for now its 3, but we could find images with 4 bytes so this is good to have

File Based Textures

```
struct textureStruct {  
    int height;  
    int width;  
    int bytes;  
    unsigned char *data;  
};
```

File Based Textures

- Our loadImage procedure is divided into two parts:
 - Reading the image and filling in the texture structure
 - Processing the pixels
- The first part of the procedure is shown on the next slide
- Note that we need to know the file type before we can read it

File Based Textures

```
bitmap = FreeImage_Load(FIF_JPEG, filename, JPEG_DEFAULT);
height = FreeImage_GetHeight(bitmap);
width = FreeImage_GetWidth(bitmap);
bytes = FreeImage_GetBPP(bitmap)/8;
printf("image size: %d %d %d\n", width, height, bytes);
```

```
data = new unsigned char[width*height*bytes];
result->height = height;
result->width = width;
result->bytes = bytes;
result->data = data;
```

File Based Textures

- In the second part of the procedure we process the image one line of pixels at a time
- We move the pixels from the internal FreedImage storage to the array that we use to pass the pixels to OpenGL
- This code is shown on the following slide

File Based Textures

```
k = 0;
for(j=0; j<height; j++) {
    bits = FreeImage_GetScanLine(bitmap,j);
    for(i=0; i<width; i++) {
        data[k++] = bits[FI_RGBA_RED];
        data[k++] = bits[FI_RGBA_GREEN];
        data[k++] = bits[FI_RGBA_BLUE];
        bits += 3;
    }
}
FreeImage_Unload(bitmap);
return(result);
```

File Based Textures

- Now that we have the texture in memory, the rest of the code is easy
- We basically need to make a few small changes to the code that we have already written
- The code that's changed is highlighted in red, it is basically two lines of code

File Based Textures

```
texture = loadImage("glasswork.jpg");

glGenTextures(1, &texName);
glBindTexture(GL_TEXTURE_2D, texName);

glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, texture->width, texture->height,
            0, GL_RGB,GL_UNSIGNED_BYTE, texture->data);

glGenerateMipmap(GL_TEXTURE_2D);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
```

File Based Textures

- The vertex and fragment shaders are unchanged
- We could make the image look better by adding a specular highlight
- We can use the texture value as the ambient and diffuse component and then add the specular code that we have already written

Texture Values

- The values that we retrieve from the texture map can be used in any computation, they can even be used in vertex shaders
- We don't need to store just colours in our texture map, with three components it could also be a vector or position
- We can also use more than one texture map at a time

Multiple Textures

- Many applications use more than one texture on an object
- One texture could be a base texture, while another could be a detail texture
- The textures could be used in different parts of the light model, one for diffuse and another for specular
- There are many possible combinations, we will examine the details of working with multiple textures

Multiple Textures

- With multiple textures we may need to have more than one set of texture coordinates
- This is easy to do, we set up the second set of texture coordinates in the same way as the first one
- Our vertex and fragment shaders then have one extra `vec2` attribute variable for the second set of texture coordinates

Multiple Textures

- Next we need to deal with multiple textures
- Each texture is associated with a texture unit, and there are at least 80 texture units
- The `glActiveTexture` function is used to select one of these units
- After it is selected we can use `glBindTexture` to associate a texture with the texture unit

Multiple Textures

- The default texture units are numbered 0 through 79
- To associate a texture with a uniform sampler variable in one of our shaders we just need to assign the texture unit number to that variable
- This can be done using the `glUniform1i` procedure, the same as we have set other uniform variables

Multiple Textures

- Our next example shows how we can use multiple textures and will improve our loadImage function
- In this example we use two textures, one with an alpha channel so we can easily combine them
- We will use one set of texture coordinates, but in the vertex shader we will rotate them to produce two sets of texture coordinates
- The code for this example is in the file example9.zip on Canvas

Multiple Textures

- Our texture files are PNG files, one has 3 bytes per pixel and the other has 4 bytes per pixel
- Thus we need to change our loadImage function to deal with these changes
- The first change just changes some parameters in the load call:
`bitmap = FreeImage_Load(FIF_PNG, filename, PNG_DEFAULT);`
- The other changes are on the next slide

Multiple Textures

```
for(j=0; j<height; j++) {  
    bits = FreeImage_GetScanLine(bitmap,j);  
    for(i=0; i<width; i++) {  
        data[k++] = bits[FI_RGBA_RED];  
        data[k++] = bits[FI_RGBA_GREEN];  
        data[k++] = bits[FI_RGBA_BLUE];  
        if(bytes > 3) {  
            data[k++] = bits[FI_RGBA_ALPHA];  
        }  
        bits += bytes;  
    }  
}
```

Multiple Textures

- The changes are shown in red
- If there is an alpha component we need to retrieve its value as well
- The constant increment by 3 is now replaced by an increment by bytes
- The code for processing the first texture is shown on the next slide, the changes are highlighted in red

Multiple Textures

```
glGenTextures(1, &texName1);
glActiveTexture(GL_TEXTURE0);
 glBindTexture(GL_TEXTURE_2D, texName1);

texture1 = loadImage("test.png");

glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, texture1->width,
             texture1->height, 0, GL_RGBA, GL_UNSIGNED_BYTE,
             texture1->data);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
```

Multiple Textures

- The code for the second texture is basically the same
- The following code is used to set the values of the uniform sampler variables in the fragment shader:

```
tex1 = glGetUniformLocation(program, "tex1");
tex2 = glGetUniformLocation(program, "tex2");
glUniform1i(tex1, 0);
glUniform1i(tex2, 1);
```

Multiple Textures

- The code for the vertex shader is shown on the next slide
- The “time” uniform variable is updated in the display loop in the main() procedure and sent to the vertex shader on each update
- The time variable is used to construct a rotation matrix, which is applied to the two sets of texture coordinates

Multiple Textures

```
in vec4 vPosition;
in vec2 vTexture;

uniform mat4 modelView;
uniform float time;

out vec2 texCoord1;
out vec2 texCoord2;

void main(void) {
    mat2 m = mat2(vec2(cos(time), sin(time)),
                  vec2(-sin(time), cos(time)));
    gl_Position = modelView * vPosition;
    texCoord1 = vTexture * m;
    texCoord2 = vTexture * transpose(m);
}
```

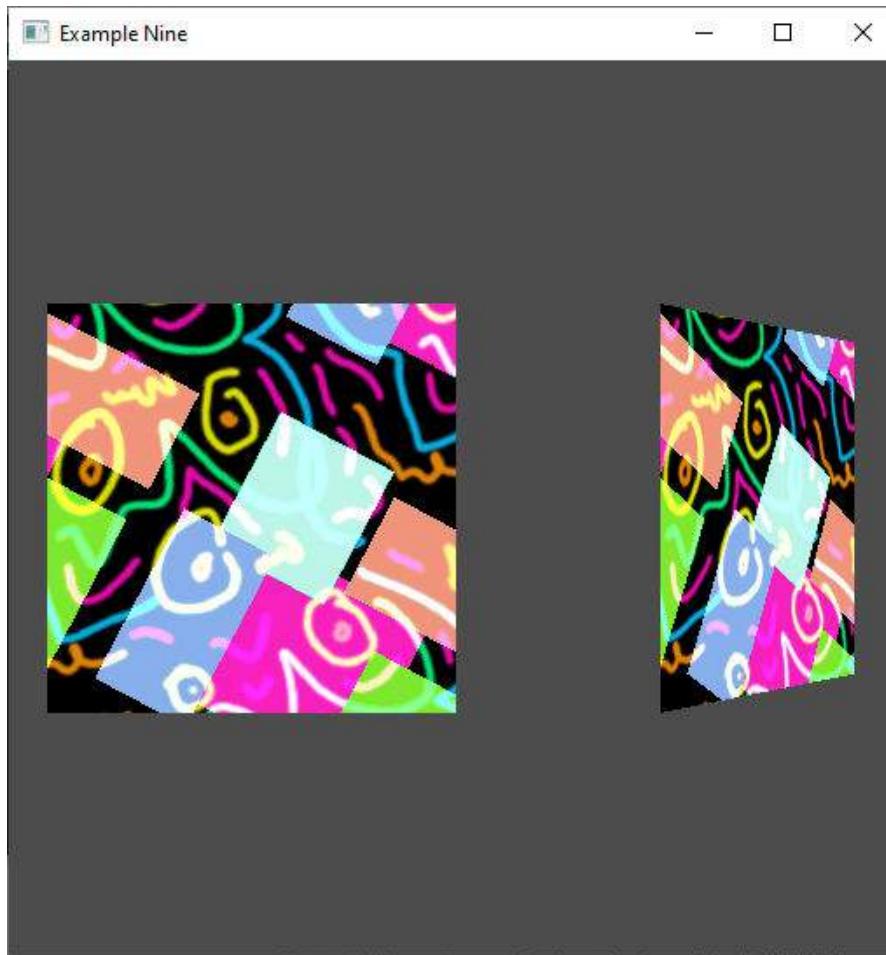
Multiple Textures

- Since we use the transpose of m for the second set of coordinates the textures rotate in opposite directions
- The fragment shader is shown on the next slide
- All it does is add the two textures together and use that as the final pixel colour

Multiple Textures

```
in vec2 texCoord1;  
in vec2 texCoord2;  
uniform sampler2D tex1;  
uniform sampler2D tex2;  
  
void main(void) {  
  
    gl_FragColor = texture(tex1, texCoord1) + texture(tex2, texCoord2);  
  
}
```

Multiple Textures



Advanced Texture Mapping

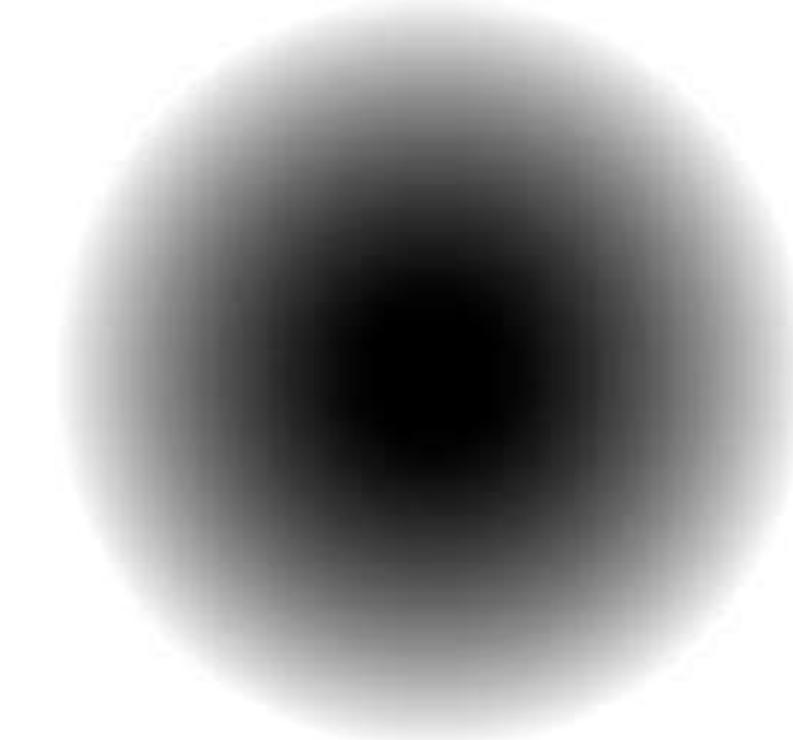
- This is called simple 2D texturing, and it's the simplest form of texture mapping
- We will now look briefly at some of the more advanced forms of texture mapping
- So far we have just used a 2D image as our texture map, we can also use 1D and 3D images as well
- In OpenGL they are handled in basically the same way as 2D images, just the names of the functions change

Advanced Texturing

- A 1D image is more or less like a line, but its colour can change at each pixel
- Each vertex just needs one texture coordinate
- 1D textures are often used for contour lines and scales on the surfaces of objects

1D texture

- 1D textures can create interesting effects, depending on how they are applied



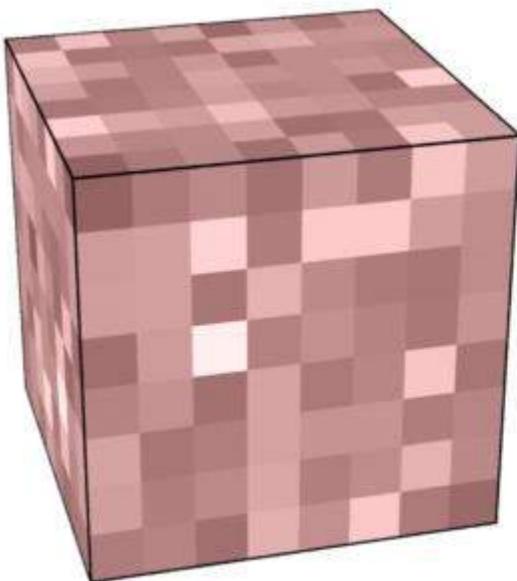
3D Texturing

- A 3D image can be viewed as a volume, so we are basically sampling a volume in 3D space
- Why would we want to do this?
- Consider an object made of wood, if we cut this object open we expect to see a wood grain texture inside of it
- If our texture is 3D we can have a wood texture throughout the whole object

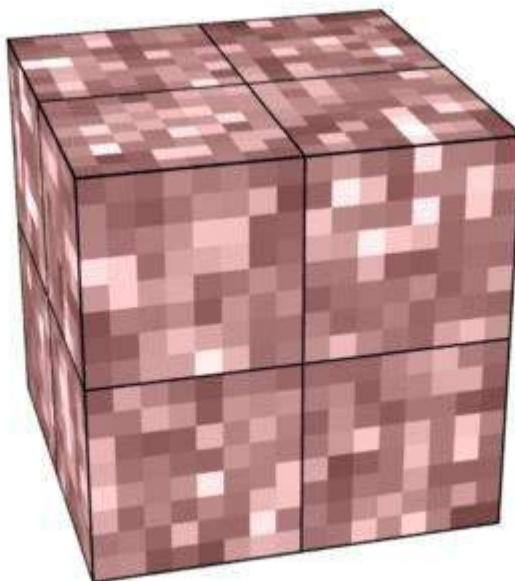
Advanced Texturing

- Another example is a cloud or smoke
- This type of object is partially transparent, so we can see inside of it from the outside
- Thus, we need to have a texture inside of this object as well
- One of the main problems with 3D textures is they are large, and thus they are hard to filter to get rid of aliasing problems

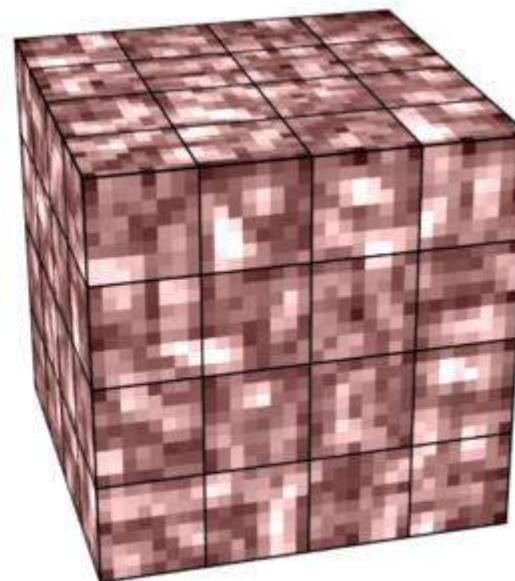
3D Texture “Brick Mapping”



Level 0



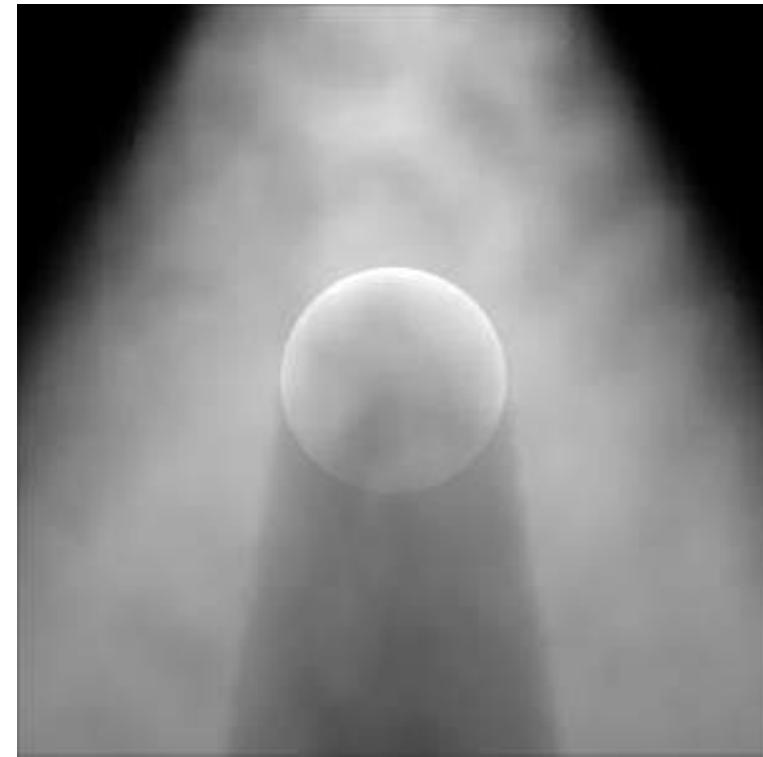
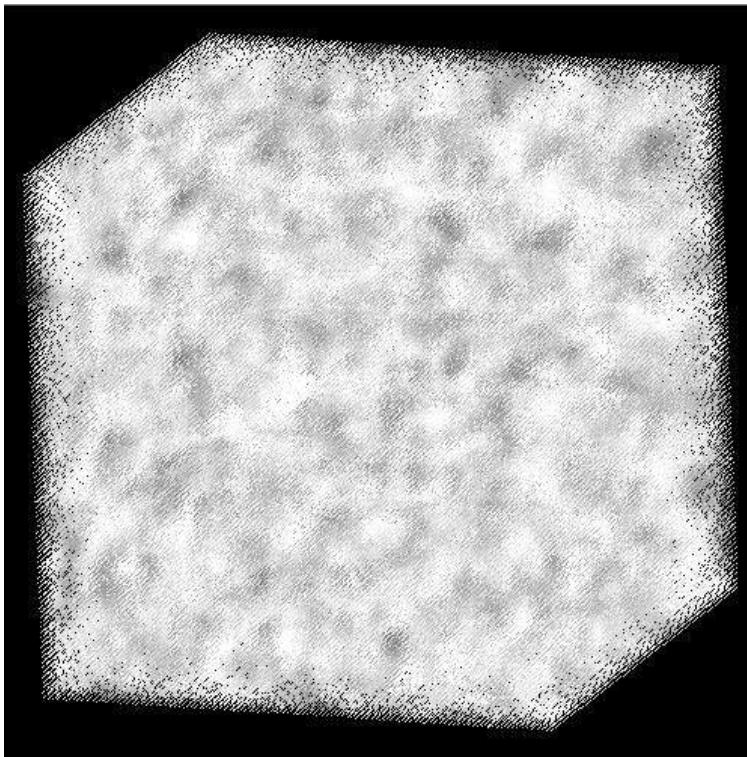
Level 1



Level 2

http://hradec.com/ebooks/CGI/RMS_1.0/rman/prman_technical_rendering/AppNotes/baking3d.html

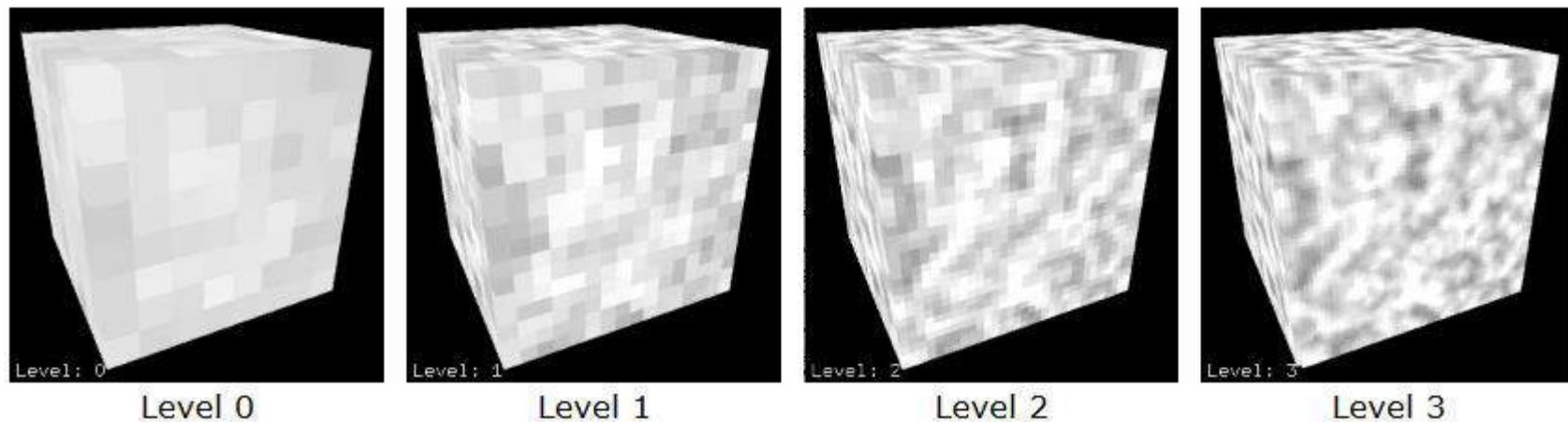
3D texture application



Procedural Textures

- This brings up the idea of procedural textures
- Instead of storing an image a procedure is used to generate a texture value when its needed
- This solves the storage problem, and the procedure can filter the texture based on its knowledge of the technique used to compute the texture value
- Our ‘checker’ map is an example of this

Procedural “Smoke” 3D Texture



http://hradec.com/ebooks/CGI/RMS_1.0/rman/prman_technical_rendering/AppNotes/baking3d.html

Procedural Textures

- With modern graphics hardware its now possible to do **procedural textures** in real time
- So far we have used the texture values to modify the surface colour directly, but there are other things we can do with texture values
- Modify the normal vector before it is used to compute the colour of the pixel
- This is called a normal map (“bump map”), and it can be used to produce bumpy looking surfaces

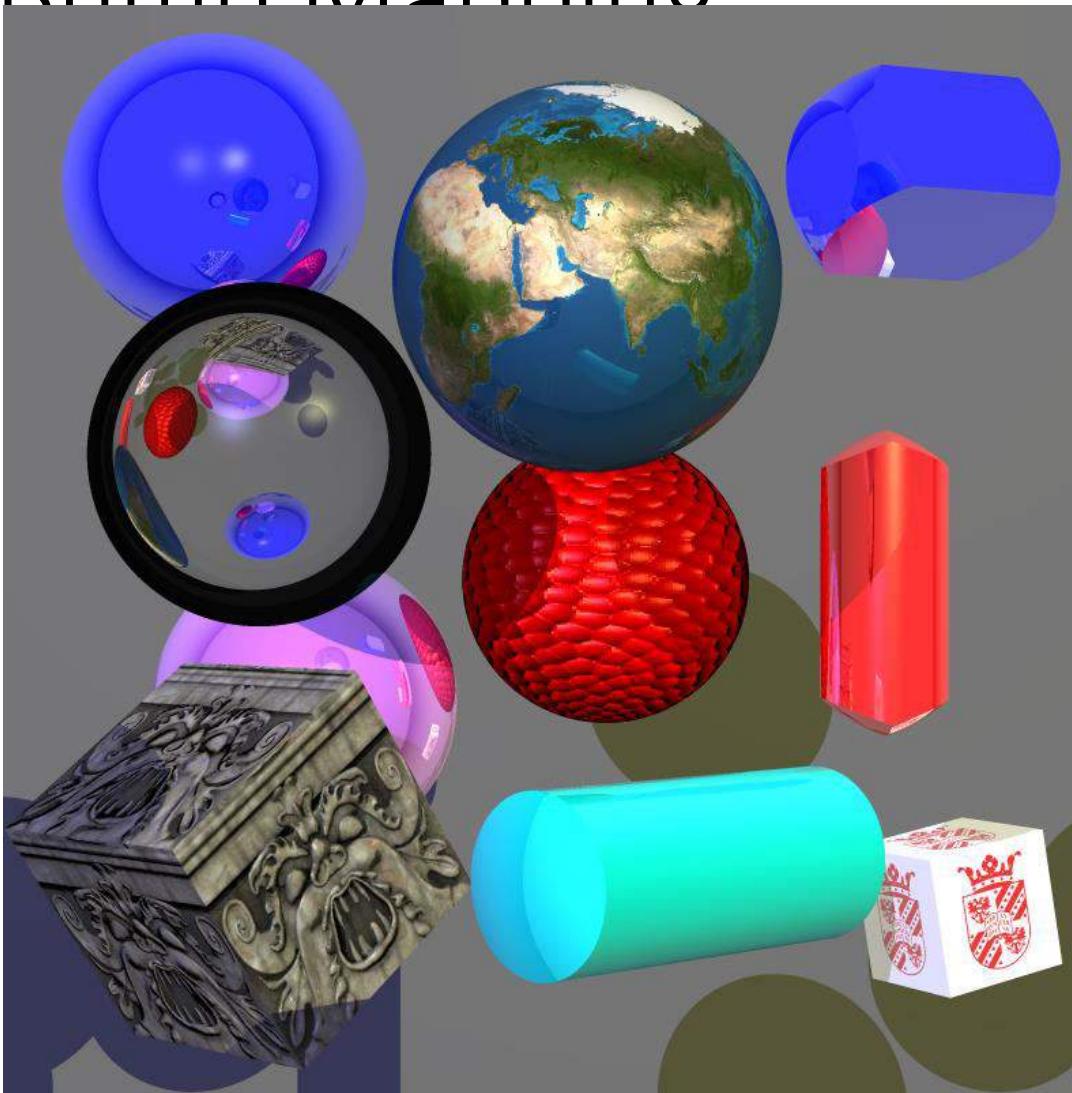
Texture Mapping: Bump Mapping

- recall: the brick wall
- texture captures visuals
- problem: illumination captured in the texture, conflicting visuals
- solution part 1: capture texture under diffuse light only – how?
- solution part 2: **bump mapping** → change illumination handling with texture



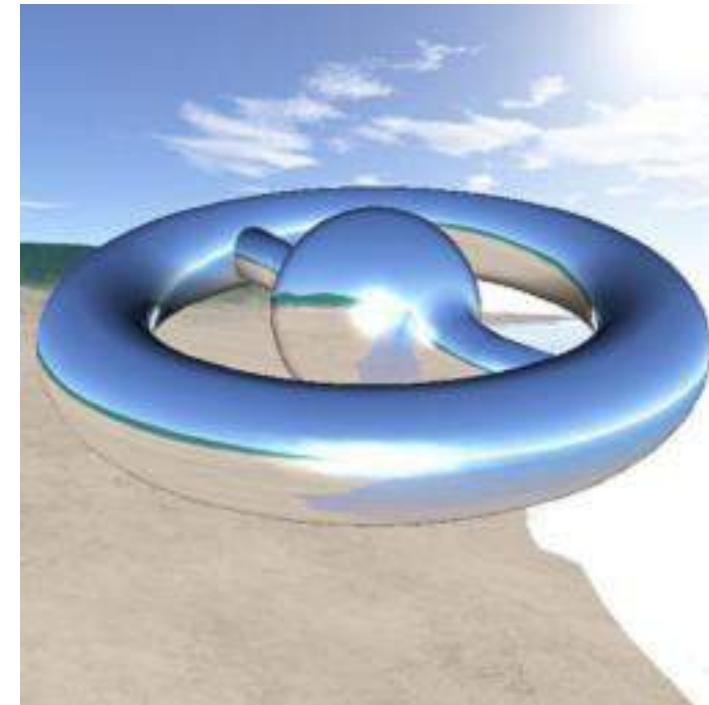
Texture Mapping: Bump Mapping

- bump maps:
vector offsets
to the normal
vectors
- illumination
computed
as usual
- bump maps
should match
visual texture



Affecting other Properties

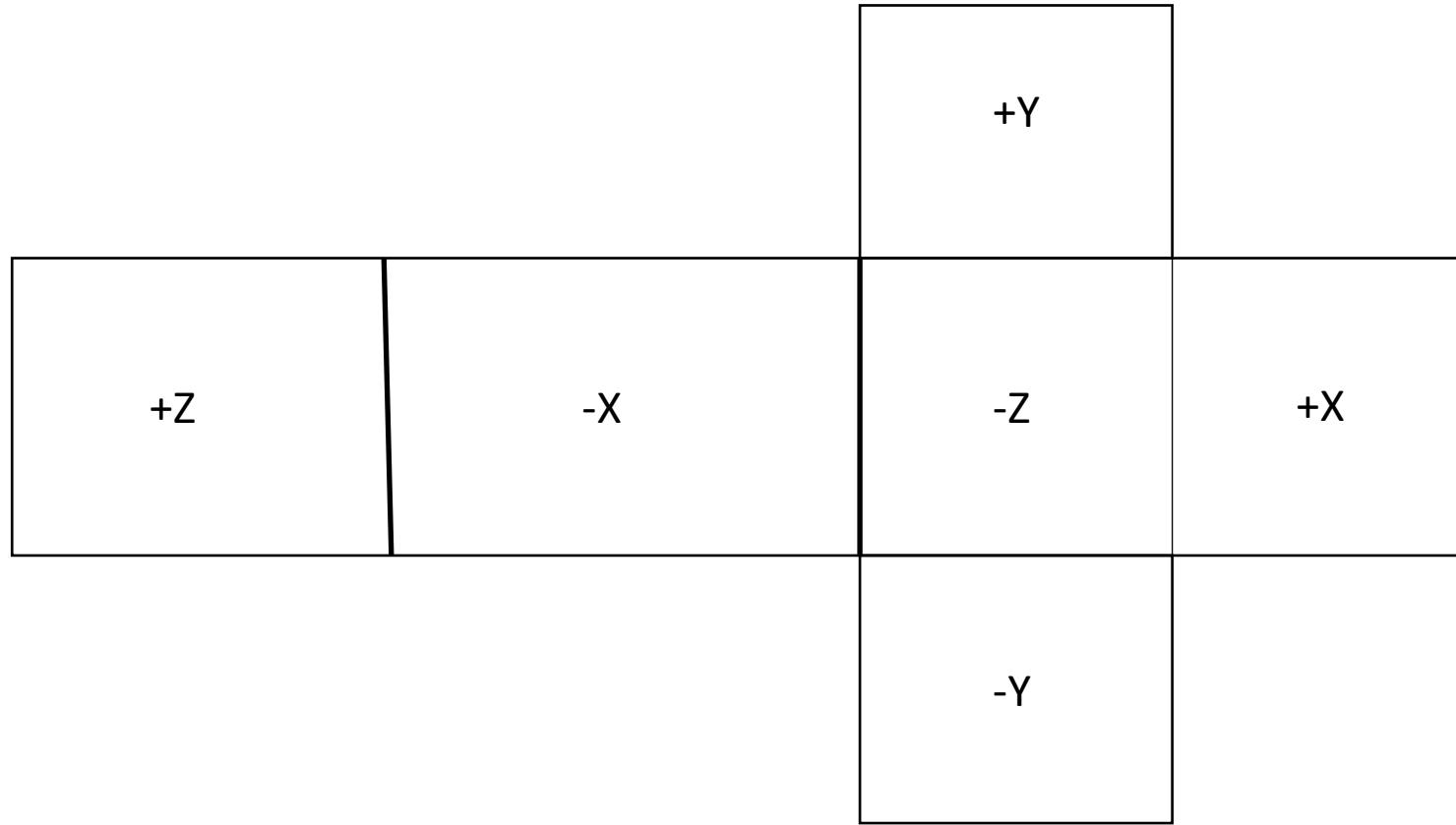
- actual surface positions
(as opposed to normal vectors only):
displacement mapping
- transparency
- simulation of reflection:
environment mapping
- many more things
with GPU processing



Environment Maps

- So far we have considered one texture, a variation on this is to have 6 textures, two for each coordinate plane in the positive and negative directions
- See the next slide for how this is laid out
- We now use a 3D texture coordinate (s, t, r) , the coordinate with the largest magnitude determines the 2D image that we use
- The other 2 coordinates are treated as standard 2D texture coordinates

Environment Maps



Environment Maps

- There are multiple ways that we can compute the texture coordinates
- In a box map either the normal vector at the vertex, or the vector from the center of the object to the vertex are used as texture coordinates
- This is better than using a spherical map since there is less distortion around the poles, but we need to have 6 texture maps instead of one
- Environment maps are quite often used for reflective objects to capture reflections of other objects in the environment

Environment Maps

- In this case we want the texture coordinates to map onto the reflected light direction from the eye's perspective
- The texture coordinates are typically generated in real-time, which can be done in hardware
- We need two vectors, the unit vector u from the origin to the vertex/pixel and the normal vector n at the vertex, both of these must be in the eye coordinate system
- Alternatively, in world coordinates u is the vector from the eye to the vertex/pixel

Environment Maps

- We then compute the reflection direction as:
$$r = u - 2n(n \cdot u)$$
- r is then used as the texture coordinates
- We can automatically create the texture map in the following way:
 - Position the camera at the object center
 - Point the camera along one of the 6 axis directions
 - Render the scene without the textured object
 - Save the resulting image as one of the texture images

Environment Maps

- A relatively low resolution can be used for this, since the texture will only occupy part of the screen
- This process can be repeated each time the image is rendered, or it can be done once at application initialization and only changed when something in the environment changes
- This is a cheap way of getting effects that are similar to global illumination

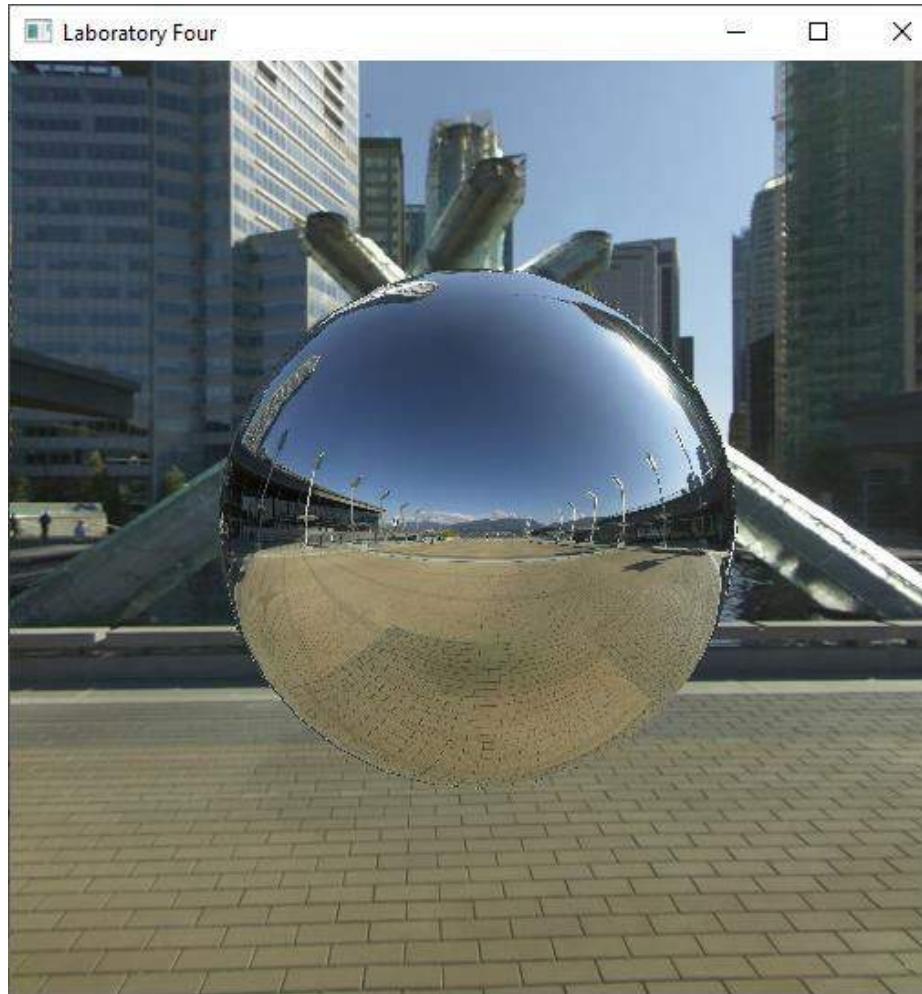
Environment Maps

- To see how this works we will construct an interesting example
- This example is based an environment map for the outside of the Vancouver Convention Centre
- We will start by constructing a cube and then use this environment map to texture it
- This will be the background for our scene
- The code for this example is example10.zip on Canvas

Environment Maps

- Next we will put a sphere in the middle of the scene and make it completely reflective
- It will reflect the environment as shown on the next slide
- The same texture is used for the sphere, but with a different fragment program

Environment Maps



Environment Maps

- We already have the sphere code, so that's easy
- We add a procedure, background() that constructs the cube that is used as the background
- The init() procedure also loads the cube texture that we use

Environment Maps

- This example comes with a texture loading library that will load the cube map for us
- The procedure `loadCube()` does this, its parameter is the folder that contains the 6 images that make up the cube texture
- The code at the end of `init()` that does this is shown on the next slide

Environment Maps

```
texture = loadCube("../VancouverConventionCentre");
glGenTextures(1, &envMap);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_CUBE_MAP, envMap);
for (i = 0; i<6; i++) {
    glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, 0, GL_RGBA, texture->width, texture->height,
0, GL_RGB, GL_UNSIGNED_BYTE, texture->data[i]);
}
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
```

Environment Maps

- The display() procedure needs to display the two models, this is just an extension of what we have seen before
- The models have different shader programs, we just need to make sure we are using the right one
- Finally we have the code to compile and link the shader programs

Environment Maps

```
sprintf(vertexName, "example10%s.vs", vertex);
vs = buildShader(GL_VERTEX_SHADER, vertexName);
sprintf(fragmentName, "example10%s.fs", fragment);
fs = buildShader(GL_FRAGMENT_SHADER, fragmentName);
program = buildProgram(vs, fs, 0);
vs = buildShader(GL_VERTEX_SHADER, "cube.vs");
fs = buildShader(GL_FRAGMENT_SHADER, "cube.fs");
program2 = buildProgram(vs, fs, 0);
dumpProgram(program, "Example 10 shader program");
```

Environment Maps

- Now we turn to the shaders
- The vertex shaders for the two programs are the same so we will only look at one of them
- It transforms the position to eye coordinates for rendering
- It also passes the position and normal vector to the fragment shader untouched

Environment Maps

```
#version 330 core

uniform mat4 modelView;
uniform mat4 projection;
in vec4 vPosition;
in vec3 vNormal;
out vec3 normal;
out vec3 position;

void main() {
    gl_Position = projection * modelView * vPosition;
    position = vPosition.xyz;
    normal = vNormal;
}
```

Environment Maps

- The fragment shader for the background is straight forward
- We just use the world space position as the index into the environment map
- This makes sense see we have essentially painted this map onto the surface of the cube

Environment Maps

```
#version 330 core  
  
in vec3 normal;  
in vec3 position;  
uniform samplerCube tex;  
uniform vec3 Eye;  
  
void main() {  
    gl_FragColor = texture(tex,normalize(position));  
}
```

Environment Maps

- The fragment shader for the sphere is a bit more complicated
- We want to reflect off the sphere into the environment map
- The reflect procedure assists with this
- The first parameter to this function is the incident vector and the second is the normal
- The result is the reflection vector that is used as the texture coordinates for the environment map

Environment Maps

```
#version 330 core

in vec3 normal;
in vec3 position;
uniform samplerCube tex;
uniform vec4 colour;
uniform vec3 Eye;
uniform vec3 light;
uniform vec4 material;

void main() {
    vec3 tc;

    tc = reflect(position - Eye, normal);
    gl_FragColor = texture(tex,tc);

}
```

Environment Maps

- Try this example and change the viewing direction to see how it responds
- This example is the starting point for the next assignment

Summary

- Examined several OpenGL programs that illustrate how texture mapping can be used
- Use of multiple textures
- Briefly described other uses of textures
- Will come back to procedural textures later

CSCI 3090

Graphics Application

Development

Mark Green

Faculty of Science

Ontario Tech

Goals

- In this part of the course you will:
 - Be introduced to the structure of graphics applications with emphasis on visualization
 - Be introduced to the problem of large data sets and some potential approaches to dealing with them
 - Examine a case study of visualizing data produced by a simulation of a hurricane

Introduction

- So far we have produced some programs that do graphics, you have a feel for how we develop these programs
- Most graphics applications are more than what we have seen
- Quite often they are made up of several programs, need to deal with multiple file formats

Introduction

- Start by examining some of the general principles and ideas
- Then move on to a case study based on Hurricane data, show how we develop real graphics applications
- Bias towards visualization application, since that is the subject of the case study

Process - Data

- Start with a source of data, can be form:
 - Collected data – field data
 - Experimental data – results of an experiment
 - Result of a computation
 - Output from a design program
- Rarely have control over the data format, hopefully someone tells us how its formatted

Process - Data

- Already seen one file formats, OBJ, relatively simple format produced by design programs
- There are many other file formats of this nature, they can be text or binary
- If they are binary we may have problems reading them, number representations, big vs. little endian, etc.

Process - Data

- If we are lucky there is a library for reading and writing the file format, otherwise we need to do this ourselves
- Quite often the raw data is not in the format that we need
- We need to write a program that converts the data into a more convenient format, another set of files
- This can be a complicated process, may need to combine data from several sources

Process - Data

- Example: collection of environmental data
- May need to combine with geographical data to assign it a position in space, understand why there are no readings from below ground
- May need to combine with weather data
- Produce new composite file with collected data and the context in which it was collected

Process – Preprocessing

- We would like to have our data on a grid, preferably a uniform grid, this makes the rest of the processing easier
- If the data is irregularly distributed need a step that interpolates the data to a regular grid
- Make sure that this makes sense for the data, it does for most physical measurements

Process - Extraction

- Once the data is in a reasonable format need to extract the parts we are interested in
- This can vary from easy to quite complicated
- The easy cases, when we just need to extract a subset of the variables
- Read in the dataset, extract the variables, write them to a new file
- This could be slow for a large dataset, but its reasonably easy to code

Process - Extraction

- In other cases this could be a complex process
- Example: wind velocity, what is a meaningful way of displaying this data?
- Could just display a vector at each grid point, but this only gives local view
- Compute the path of a particle as it moves through the velocity field, gives a better idea of the structure of the data

Process - Extraction

- Trace multiple paths, see how they interact, where there are unique features in the flow
- Do we trace the particle through a single time step of data, or do we trace it through multiple time steps?
- First gives us a picture of the flow at one point in time
- Second gives us a picture of how the flow evolves
- Both are valuable, depends what we are interested in

Process - Display

- Finally we get to the graphics!
- We need to convert the extracted data into some graphical representation
- For physical data or designs this is easier, we already have a coordinate system, an easy mapping from data to geometry
- For non-physical data this is much harder, need to figure out how we map the data to geometry or something we can display

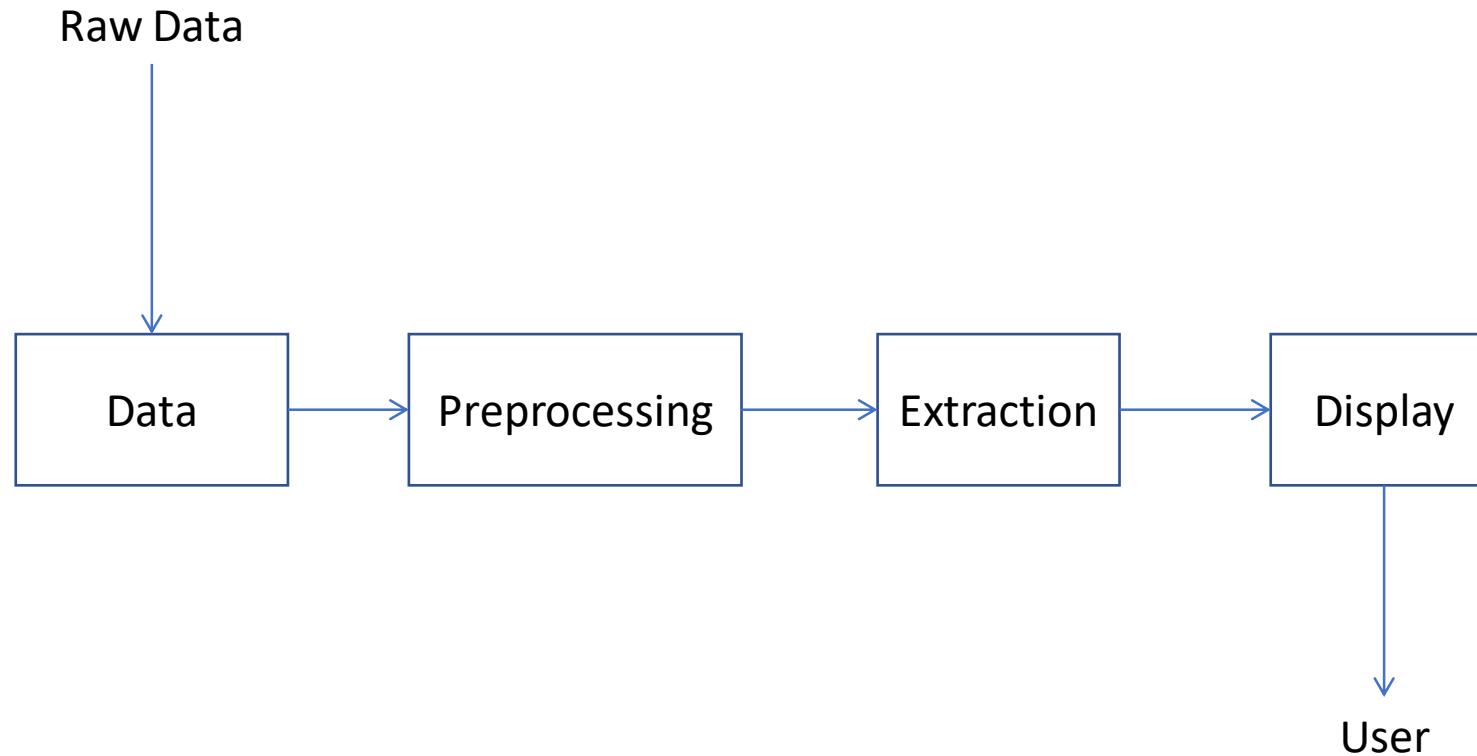
Process - Display

- This mapping from data to geometry is the heart of the display problem, where we spend most of our time
- Visualization researchers spend most of their time on these mappings, called visualization techniques
- How interactive is the visualization? How can the user interact with the data?

Process - Display

- Some cases the data is static, the user can only change the view point
- Can have control over the mapping, the part of the data that is concentrated on
- May want to go further back and control how the data is extracted (a background process?), or even how the data is generated

Process



Process - Scale

- This is the classical view, we have a set of programs or processes
- Each program has one or more input files and produces an output file
- This was okay when the datasets were relatively small
- The data set for the case study consists of 624 files, each is 100MBytes, a mere 62GBytes
- This is considered to be on the small side, easily fits on one disk

Process - Scale

- The data is stored on a stack of DVDs and on the hard drive of one of my servers
- Supercomputers: thousands of CPUs/GPUs, can generate large amounts of data very quickly
- Hard to save data to disk, need hundreds of disks, writing the data can take a considerable amount of time, waste of computing time

Process - Scale

- Can now build relatively cheap sensors that can be placed around the world
- Each sensor sends a small set of data, maybe 1KByte every few seconds or minutes
- With a large number this can soon add up to a significant amount of data
- This is streaming over the Internet to servers that capture the data and consolidate it

Process - Scale

- In the past we could have all the data for a graphics application on the same workstation
- Visualization was a post process after the data had been computed or collected
- Data now larger than workstation can handle, even with increases in storage and processing power
- Need to have another solution

Process - Scale

- Rather than write the data to disk, can we perform the visualization on the supercomputer?
- Don't try to move the data off the supercomputer, build the visualization on the supercomputer while the computation is running
- Display the results of the visualization on a separate workstation

Process - Scale

- This is called in-situ visualization, reduces the amount of data that must be transmitted
- Also reduces the amount of data that must be saved – no need to save data on disk for visualization
- Ideally distribute the visualization over the supercomputer so the data doesn't need to be moved between processors
- Moving data between a large number of processors is very expensive

Computational Steering

- If the visualization is running on the supercomputer at the same time as the computation, why not interact directly with the computation?
- View the results of the visualization and change the parameters to the computation at the same time
- Called computational steering, ability to control computation while it is running

Computational Steering

- Most computations are started and don't look at the results until they are finished
- Can take hours or days to finish, but what happens if something goes wrong?
- Like to be able to peek at a computation to see if something has gone wrong, or if there is something interesting to explore
- Take action before the computation finishes, save valuable computer time

Computational Steering

- One of the grand challenges of scientific computing
- Can we do this without significantly impacting the way we write scientific code
- Some of these programs have been evolving for decades, difficult to make major changes to them
- Can we provide a framework that does this (semi) automatically?

Case Study

- Based on a simulation of hurricane Isabel using the WRF model courtesy of NCAR and NSF
- This is a 48 hour simulation with data files produced every hour
- The simulation is done on a 500x500x100 grid with x being longitude, y being latitude and z being height
- There is one variable per time point in a file, with each variable being a float value

Case Study

- Thus each file is 100MBytes
- The variable that are computed are:
 - QCLOUD – cloud water
 - QGRAUP – cloud graupel – small hail
 - QICE – cloud ice
 - QRAIN – cloud rain
 - QSNOW – cloud snow
 - QVAPOR – water vapor
 - CLOUD – total cloud – QCLOUD + QICE
 - PRECIP – total precipitation – QGRAUP+QRAIN+QSNOW
 - P – pressure
 - TC – temperature
 - U, V, W – X, Y and Z wind velocity

Case Study

- Thus we have 13 variables over 48 time steps, which gives us 624 data files, 62.4 Gbytes of data
- There is also an HGT.bin file that gives the height at each of the 500x500 surface positions
- Locations where there is no valid data (below surface level) have the value of 1e35

Case Study

- This is all binary data, and the endian of the data is the opposite of modern computers
- We will start by looking at the wind velocity data, we want a picture of how the wind velocity varies over 3D location
- We will do this by tracing particle paths within a single time slice
- Since the components of the wind velocity are in three different files the first thing we need to do is combine them

Case Study - Convert

- In addition we also need to change the endian of the data
- We can do this all in a program we call convert
- This program reads in the U, V, W files separately and converts the endianness of the values
- The code for doing this is shown on the next slide

Case Study - Convert

```
float* read_and_convert(char* filename) {
    FILE* infile;
    float *result;
    int i;

    infile = fopen(filename,"r");
    printf("creating array\n");
    result = (float*) malloc(XDIM*YDIM*ZDIM*sizeof(float));
    printf("reading data\n");
    fread(result,XDIM*YDIM*ZDIM*sizeof(float),1,infile);

    printf("converting data\n");

    for(i=0; i<XDIM*YDIM*ZDIM; i++) {
        result[i] = flip(result[i]);
        if(result[i] > 1e20)
            result[i] = 0.0;
    }

    return(result);
}
```

Case Study - Convert

- Once the three files have been read we assemble the U, V and W values
- For each grid point the U, V and W values are stored in consecutive float values
- Once this has been done the whole data array is written to a single file
- The size of this file is 300MBytes
- The code for doing this is shown on the next slide

Case Study - Convert

```
output_data(float *udata, float *vdata, float *wdata) {
    FILE *outfile;
    int i;
    int j;
    float *data;

    data = (float*) malloc(3*XDIM*YDIM*ZDIM*sizeof(float));

    j = 0;
    for(i=0; i<XDIM*YDIM*ZDIM; i++) {
        data[j++] = udata[i];
        data[j++] = vdata[i];
        data[j++] = wdata[i];
    }

    outfile = fopen("output", "w");
    fwrite(data,3*XDIM*YDIM*ZDIM, sizeof(float), outfile);

    fclose(outfile);
}
```

Case Study - Convert

- The whole conversion process takes a few seconds
- We can do all the processing in memory, it only requires 600MBytes
- If we wanted to store all 48 time slices we would need 14.4GBytes of storage
- While this is more than we find on most laptops, it is not out of line for a desktop computer

Case Study - Trace

- Now that we have all three components of the velocity in one file we can examine how we compute the particle traces
- This is done by a program called trace
- The idea is to put a particle in the velocity field and compute its motion using basic physics
- At each grid point we have a velocity value, but its unlikely that a particle will be a grid point, so we will need to interpolate velocity

Case Study - Trace

- Basically we need to integrate velocity over time to get the particle's path
- We need to do this numerically, so we need to select an integration technique
- The trace program implements two of them:
 - Euler – fast but not very accurate
 - 2nd order Runge-Kutta – more accurate, but requires two velocity evaluations per time step

Case Study - Trace

- Now for the details
- How many particle paths do we need?
- The current version of trace computes 1000 paths, this is sufficient to see the general structure of the flow
- How do we generate the initial particle position?
- A random position is generated, the length of the velocity vector at that position is computed and the position is accepted if the length is greater than a threshold

Case Study - Trace

- Finding initial particle positions, called seeding is a difficult problem
- Choosing random positions is a good general strategy, but could miss important features
- If we know the location of the interesting features we can randomly generate seeds within a sphere or along a line at that location
- With a small data set this can be under interactive control, move the sphere or line and see how the paths change

Case Study - Trace

- It would be nice if our program could automatically select interesting locations
- But what is an interesting location?
- An area of high velocity could be interesting, would generate longer paths
- For weather this is a good guess
- High variances in velocity could also be interesting, for example a circular flow that could develop into a tornado
- What's interesting probably depends upon the application and what we are looking for

Case Study - Trace

- How do we terminate the path?
- There are several possible termination conditions (we typically use all of them):
 - The velocity is close to zero, so the particle isn't moving
 - The particle moves outside of the velocity field
 - The path has reached a maximum length
 - The maximum number of time steps has been reached

Case Study - Trace

- What is the output of this program?
- A single file that contains samples along the particle paths
- We don't save all the particle positions, we use a time step of 0.01, but only record every 1.0 (we call the unit seconds, but there really isn't any time here)
- The file consists of a header, then the paths, one after the other

Case Study - Trace

- The header has the following contents:
 - The maximum number of paths, and the maximum length of each path
 - The minimum and maximum x value
 - The minimum and maximum y value
 - The minimum and maximum z value
- Each path contains the following information:
 - The length of the path
 - The points along the path, one point per line

Case Study – Trace Output

```
1000 1001
2480.000000 3320.000000
2370.000000 4170.000000
0.035000 19.835000
104
3077.171143 3294.382813 6.054138
3080.971191 3294.533447 6.056266
3084.009766 3294.504395 6.057744
3087.594238 3294.385986 6.040550
3090.654541 3294.332031 6.028432
3093.555420 3294.291748 6.011816
3096.310059 3294.252686 6.001551
3098.971924 3294.314453 6.009079
3101.619385 3294.513916 6.025098
3104.234375 3294.847900 6.051176
3106.795654 3295.199951 6.076557
```

Case Study - Trace

- Now turn to the program code
- This program was originally written to evaluate the feasibility of running path tracing in parallel on many computers
- Thus, there is some code that we can safely ignore for our purposes
- The main data structure in the program is the field array, which contains the velocity field, this is a $500 \times 500 \times 100 \times 3$ array

Case Study -Trace

- Start by examining the code for seeding the path, shown on the next slide
- A loop until an acceptable starting point is reached
- A random starting position within the grid is generated, the magnitude of the velocity at the closest grid point is computed
- If it is large enough the point is acceptable, otherwise the process is repeated

Case Study - Trace

```
acceptable = 0;
while(!acceptable) {
    x = (Xmin+Xmax)/2.0 + (Xmax - Xmin)*(drand48()-0.5);
    y = (Ymin+Ymax)/2.0 + (Ymax - Ymin)*(drand48()-0.5);
    z = (Zmin+Zmax)/2.0 + (Zmax - Zmin)*(drand48()-0.5);
    r = (x-Xmin)/dx;
    s = (y-Ymin)/dy;
    t = (z-Zmin)/dz;
    v = field[t][s][r][0]*field[t][s][r][0]
        + field[t][s][r][1]*field[t][s][r][1]
        + field[t][s][r][2]*field[t][s][r][2];
    if(v > 20.0)
        acceptable = 1;
    else
        rejected++;
}
```

Case Study - Trace

- Next we need to compute the path, this is done in the trace procedure
- The main part of this procedure is a loop until the end of the path is reached
- The first part of this loop is shown on the next slide
- Get the velocity at the current point, select the integration scheme and compute the new position of the particle

Case Study - Trace

```
while(t < max) {
    getVelocity(x/LONG, y/LAT, z, &vx, &vy, &vz);
    if(integrator == EULER) {
        dx = vx*dt;
        dy = vy*dt;
        dz = vz*dt;
    } else if(integrator == RK2) {
        x1 = x+0.5*vx*dt;
        y1 = y+0.5*vy*dt;
        z1 = z+0.5*vz*dt;
        getVelocity(x1/LONG, y1/LAT, z1, &vx, &vy, &vz);
        dx = vx*dt;
        dy = vy*dt;
        dz = vz*dt;
    }
    x = x+dx;
    y = y+dy;
    z = z+dz;
    dlen = sqrt(dx*dx + dy*dy + dz*dz);
    len += dlen;
```

Case Study - Trace

- The length of the path is then updated
- The following code is used to add the current position to the path if a reporting point has been reached:

```
if(nextDump <= t) {  
    path.x[j] = x;  
    path.y[j] = y;  
    path.z[j] = z;  
    j++;  
    nextDump += report;  
}  
t = t+dt;
```

Case Study - Trace

- Now we need to check the various termination conditions, shown on the next slide
- Check to see if the particle has gone outside of the grid and whether the distance travelled is too short
- At the end we set the path length in the path data structure and write it to the output file

Case Study - Trace

```
if(x < Xmin*LONG || x > Xmax*LONG) {  
    early++;  
    goto exit;  
}  
if(y < Ymin*LAT || y > Ymax*LAT) {  
    early++;  
    goto exit;  
}  
if(z < Zmin || z > Zmax) {  
    early++;  
    goto exit;  
}  
if(dlen < 0.1) {  
    early++;  
    goto exit;  
}  
}
```

Case Study - Trace

- The final piece of code to examine is the velocity interpolation code, the start of this code is shown on the next slide
- This shows the interpolation for the x component of the velocity, based on the 8 neighbouring grid points
- The code for the y and z components is similar
- After running this program we have a set of traces in a text file

Case Study - Trace

```
i = (x - Xmin)/dx;  
r = (x-Xmin) - i*dx;  
j = (y - Ymin)/dy;  
s = (y-Ymin) - j*dy;  
k = (z - Zmin)/dz;  
t = (z-Zmin) - k*dz;
```

```
*vx = field[k][j][i][0]*(1-r)*(1-s)*(1-t) + field[k][j][i+1][0]*r*(1-s)*(1-t)  
+ field[k][j+1][i][0]*(1-r)*s*(1-t) + field[k][j+1][i+1][0]*r*s*(1-t)  
+ field[k+1][j][i][0]*(1-r)*(1-s)*t + field[k+1][j][i+1][0]*r*(1-s)*t  
+ field[k+1][j+1][i][0]*(1-r)*s*t + field[k+1][j+1][i+1][0]*r*s*t;
```

Case Study - Display

- Now that we have the paths its time to display them
- We won't go into the details of the code, since that is part of the assignment, but we aren't doing that assignment this year
- The easiest way to draw the paths is as a line strip, each path is drawn as a single line strip
- Our program first reads the data produced by the trace program
- This data contains the x, y and z extents of the data, so we can scale the points as we read them in

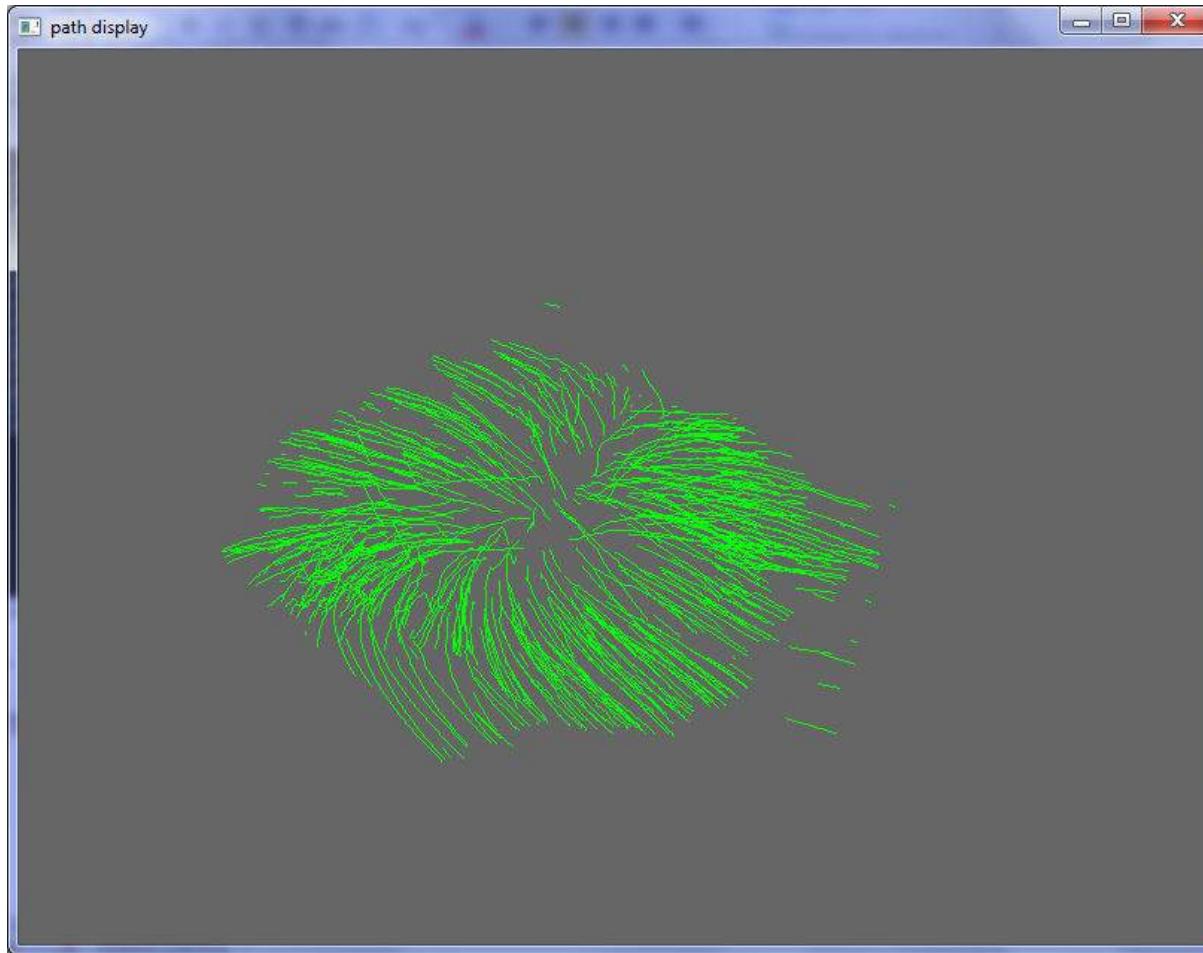
Case Study - Display

- We also know the number of paths and the maximum length of each path, so we can pre-allocate storage before we start reading the data
- Since the data is static, we can set up the array buffers and index buffers once
- The easiest approach is to do one set of buffers per path, we did something similar to this in the instancing example

Case Study - Display

- My version of the program uses the old version of OpenGL, so it is not as efficient as it could be
- The program rotates the data about the vertical axis and gives the user control over the view point
- A screen capture of this program is shown on the next slide

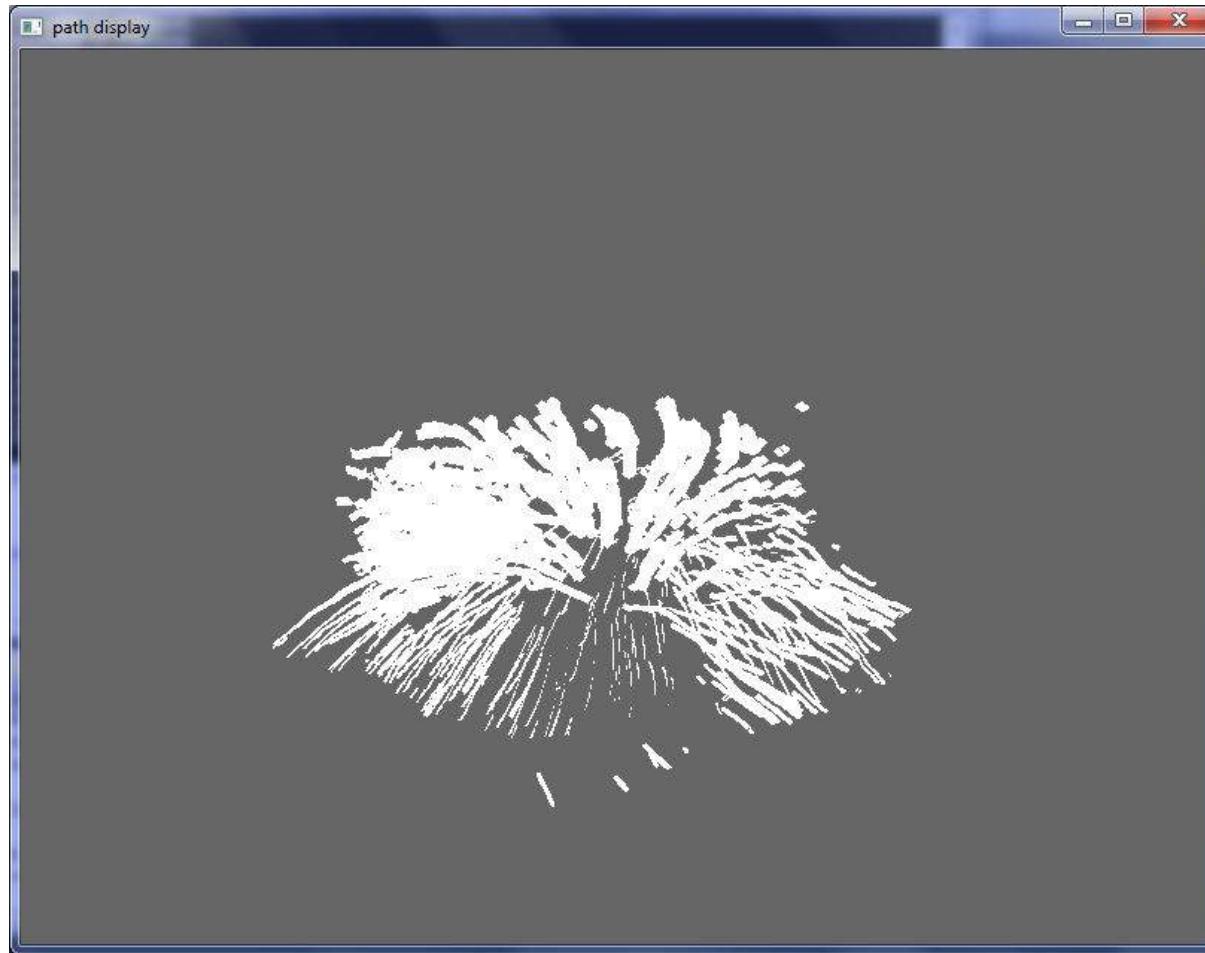
Case Study - Display



Case Study - Display

- A more interesting representation can be produced by building a tube to display the path
- In my case I used a 4 sided tube, which is the easiest to program that also gives reasonable results
- The cross section of the tube must be perpendicular to the direction of the path, this is shown in the next slide

Case Study - Display



Case Study - Display

- The easiest way to do this is to compute the vector between adjacent points on the path
- Now find two vectors that are perpendicular to this vector, cross product is your friend here
- The point on the path is the center of the tube, the two vectors you have just computed can be used to find the vertices of the tube at the current position

Case Study - Display

- You have tube vertices at the previous point, and tube vertices at the current point, now all you need to do is construct triangles
- Each tube side will have 2 triangles, for a total of 8 triangles for a tube segment

Details

- Two points at ends of segment:

$$P_1 = (x_1, y_1, z_1) \quad P_2 = (x_2, y_2, z_2)$$

- Find the vector between points

$$dx = x_2 - x_1 \quad dy = y_2 - y_1 \quad dz = z_2 - z_1$$

$$V_1 = (dx, dy, dz)$$

- Assume that both dx and dy are not zero

$$V_2 = (dy, -dx, 0)$$

Details

- V_2 is perpendicular to V_1 , try the dot product
- The final vector that we need is:

$$V_3 = V_1 \times V_2$$

- Our 4 points at P_1 are:

$$P_1 + V_2 + V_3$$

$$P_1 + V_2 - V_3$$

$$P_1 - V_2 + V_3$$

$$P_1 - V_2 - V_3$$

Case Study

- We have seen what we can do with wind velocity, but what about the other variables?
- These variables are scalars (temperature, pressure, etc.) they can't be interpreted as vectors
- In addition, they are on a 3D grid, we can't use a simple 2D plot to display them
- We need a different set of visualizations, ones that allow us to view the data in context

Case Study

- One approach is to view the data as a 3D texture, we have data values on a 3D grid, this can be viewed as our texture pixels
- The data is currently floating point numbers, they will need to be converted into colours to be used in a texture
- This gives us our extraction and display programs
- The extraction program produces the textures, the display program displays them

Case Study - Texture

- The texture program converts a data file, 25,000,000 floating point values into a texture map
- This can be done by iterating over the grid and converting each floating point value into an RGB colour
- The code for doing this is shown on the next slide

Case Study - Texture

```
void build_texture() {
    int i,j,k;
    unsigned char r, g, b;
    float value;

    for(i=0; i<ZDIM; i++) {
        for(j=0; j<YDIM; j++) {
            for(k=0; k<XDIM; k++) {
                value = flip(field[i][j][k]);
                if(value > 1e20)
                    value = vmin;
                colour(value, r, g, b);
                texture[i][j][k][0] = r;
                texture[i][j][k][1] = g;
                texture[i][j][k][2] = b;
            }
        }
    }
}
```

Case Study - Texture

- The colour procedure converts the data value into the r, g and b components of the colour
- Note, we again have to use our flip procedure to change the endianness of the data
- There are many ways that we can convert a value into a colour, the colour procedure hides these choices
- The current version uses grey scale and hue in the HSV colour space, but it can easily be extended

Case Study - Texture

- The heart of the colour procedure is shown on the next slide
- It is basically a switch statement based on the colour scheme to be used
- In both cases the value is scaled to an appropriate range, the hue values go from 0 to 360
- They are then converted into the RGB values for the texel

Case Study - Texture

```
switch(Space) {  
    case 1: // grey scale  
        scale = 256.0/(vmax-vmin);  
        v = (value-vmin)*scale;  
        r = (unsigned char) v;  
        g = (unsigned char) v;  
        b = (unsigned char) v;  
        break;  
    case 2: // HSV  
        scale = 360.0/(vmax-vmin);  
        h = (value-vmin)*scale;  
        s = 0.5;  
        v = 0.5;  
        HSVtoRGB(R, G, B, h, s, v);  
        r = (unsigned char) (R*256.0);  
        g = (unsigned char) (G*256.0);  
        b = (unsigned char) (B*256.0);  
        break;  
}
```

Case Study - Texture

- This produces a relatively large texture, on the order of 75MBytes
- I did experiment with filtering the texture to a lower resolution, but there is still a bug in that code
- It turns out that the smaller texture isn't significantly faster than the large texture in the display program, so there was no need to waste time on debugging the code

Case Study - Volume

- Now we turn to the display program, we have a 3D texture, but how do we display it?
- There are many ways of doing this, we will start with one of the simplest
- We will insert a plane into the 3D space of the texture and use its coordinates as the texture coordinates
- This will give us a view of a slice through the texture space, one 2D plane worth of data

Case Study - Volume

- Our texture space is the unit cube, so our plane will be placed inside a unit cube, directly mapping from plane coordinates to texture coordinates
- To simplify the code, the plane will be parallel to the y axis, and we will be able to translate the plane along the y axis and rotate it about the z axis
- The code for creating the plane is shown on the next slide

Case Study - Volume

```
GLfloat vertices[] = {
    0.0, 0.5, 0.0,
    0.0, 0.5, 1.0,
    1.0, 0.5, 1.0,
    1.0, 0.5, 0.0,
};

GLuint indices[] = {0, 1, 2, 2, 3, 0 };

glGenBuffers(1, &vbuffer);
 glBindBuffer(GL_ARRAY_BUFFER, vbuffer);
 glBindBuffer(GL_ARRAY_BUFFER, sizeof(vertices), NULL, GL_STATIC_DRAW);
 glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(vertices), vertices);

glGenBuffers(1, &ibuffer);
 glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibuffer);
 glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices,
 GL_STATIC_DRAW);
```

Case Study - Volume

- There is nothing really new here
- The coordinate data is mapped to the vPosition variable in the vertex shader in the standard way
- Now we need to load in the 3D texture
- The texture file is read into an array called texture, the following code is then used to create the texture map

Case Study - Volume

```
glGenTextures(1, &texName);
 glBindTexture(GL_TEXTURE_3D, texName);

 glTexImage3D(GL_TEXTURE_3D, 0, GL_RGB, XDIM,
              YDIM, ZDIM, 0, GL_RGB, GL_UNSIGNED_BYTE,
              &texture[0][0][0][0]);

 glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_S, GL_REPEAT);
 glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_T, GL_REPEAT);
 glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_R, GL_REPEAT);
 glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
 glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
```

Case Study - Volume

- Instead of using `glTexImage2D` we use `glTexImage3D`, with one extra parameter to give the size of the texture in the third dimension
- To interact with the visualization, four cases are added to the keyboard function, as shown on the next slide
- The first two are for translation and the second two are for rotation

Case Study - Volume

```
if (key == GLFW_KEY_T && action == GLFW_PRESS) {
    dy -= 0.05;
    if (dy < -0.5)
        dy = -0.5;
}
if (key == GLFW_KEY_Y && action == GLFW_PRESS) {
    dy += 0.05;
    if (dy > 0.5)
        dy = 0.5;
}
if (key == GLFW_KEY_R && action == GLFW_PRESS)
    angle += 0.02;
if (key == GLFW_KEY_F && action == GLFW_PRESS)
    angle -= 0.02;
```

Case Study - Volume

- The modifications to the display code are shown on the next slide
- Note that the model, view and projection matrices are separated, due to the way that we compute the texture coordinates in the vertex shader
- Also note how the rotation is handled, the model is centered on $y = 0.5$, so we need to translate to the origin before the rotation

Case Study - Volume

```
model = glm::translate(glm::mat4(1.0), glm::vec3(0.0, -0.5, 0.0));
model = glm::rotate(model, angle, glm::vec3(0.0, 0.0, 1.0));
model = glm::translate(model, glm::vec3(0.0, 0.5, 0.0));
model = glm::translate(model, glm::vec3(0.0, dy, 0.0));

view = glm::lookAt(glm::vec3(eyex, eyey, eyez),
                   glm::vec3(0.5, 0.0, 0.5),
                   glm::vec3(0.0f, 0.0f, 1.0f));

glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glUseProgram(program);
modelLoc = glGetUniformLocation(program, "model");
 glUniformMatrix4fv(modelLoc, 1, 0, glm::value_ptr(model));
viewLoc = glGetUniformLocation(program, "view");
 glUniformMatrix4fv(viewLoc, 1, 0, glm::value_ptr(view));
projectionLoc = glGetUniformLocation(program, "projection");
 glUniformMatrix4fv(projectionLoc, 1, 0, glm::value_ptr(projection));

 glBindTexture(GL_TEXTURE_3D, texName);
```

Case Study - Volume

- Now we can examine the shaders, the vertex shader is shown on the next slide
- The texture coordinates are produced by only applying the model matrix to the vertex position
- All three matrices are applied to the vertex coordinates

Case Study - Volume

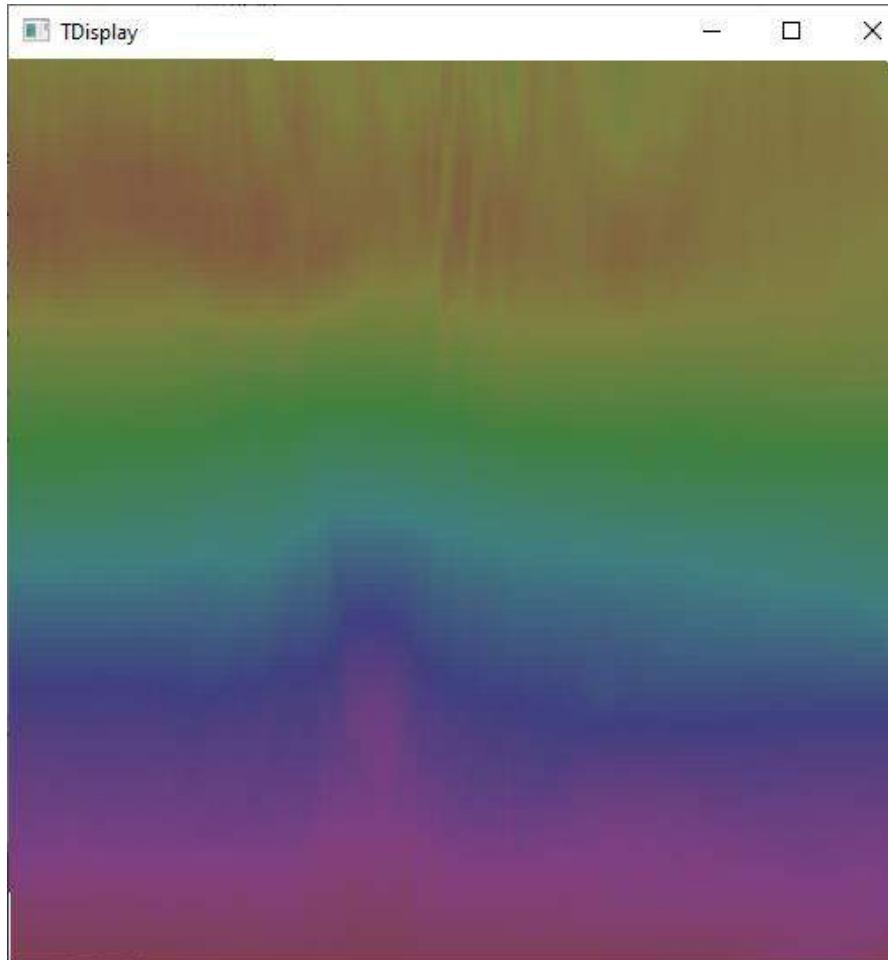
```
in vec4 vPosition;  
  
uniform mat4 model;  
uniform mat4 view;  
uniform mat4 projection;  
  
out vec3 texCoord;  
  
void main(void) {  
  
    gl_Position = projection * view * model * vPosition;  
    texCoord = (model * vPosition).xyz;  
  
}
```

Case Study - Volume

- The fragment shader is basically the same as we have seen before, with one small change to the texture type:

```
in vec3 texCoord;  
uniform sampler3D tex;  
  
void main(void) {  
  
    gl_FragColor = texture(tex, texCoord);  
  
}
```

Temperature



Pressure



Case Study

- We can easily interact with the visualization to get different views of the data
- The program could be extended to allow us to view the volume along the x or z axis, the x axis probably won't be much different, but the z axis could be interesting
- But this doesn't give us a complete picture of the data
- A more advanced technique that gives us some of this information is volume rendering

Volume Rendering

- We start with a 3D uniform rectilinear grid of scalars and we want to produce a visualization
- We would like to have X-ray vision into the volume so we can see more than one voxel at each point on the screen
- We would like some way of seeing the overall structure, everything that is in the volume in one image

Volume Rendering

- All volume rendering techniques can be divided into two parts:
 - Determine the voxels, or points in the dataset, that contribute to each pixel on the screen
 - Determine the colour of a pixel as a function of the voxels that map onto it
- Most of the time these two parts are at least semi-independent, but in each implementation there will be connections between them

Volume Rendering

- The method used to determine the pixel colour is called the transfer function
- It is usually a function of the current pixel colour and the current voxel, and sometimes the path from the pixel to the voxel
- There are many different transfer functions, we will look at them after we have determined the voxels to visit

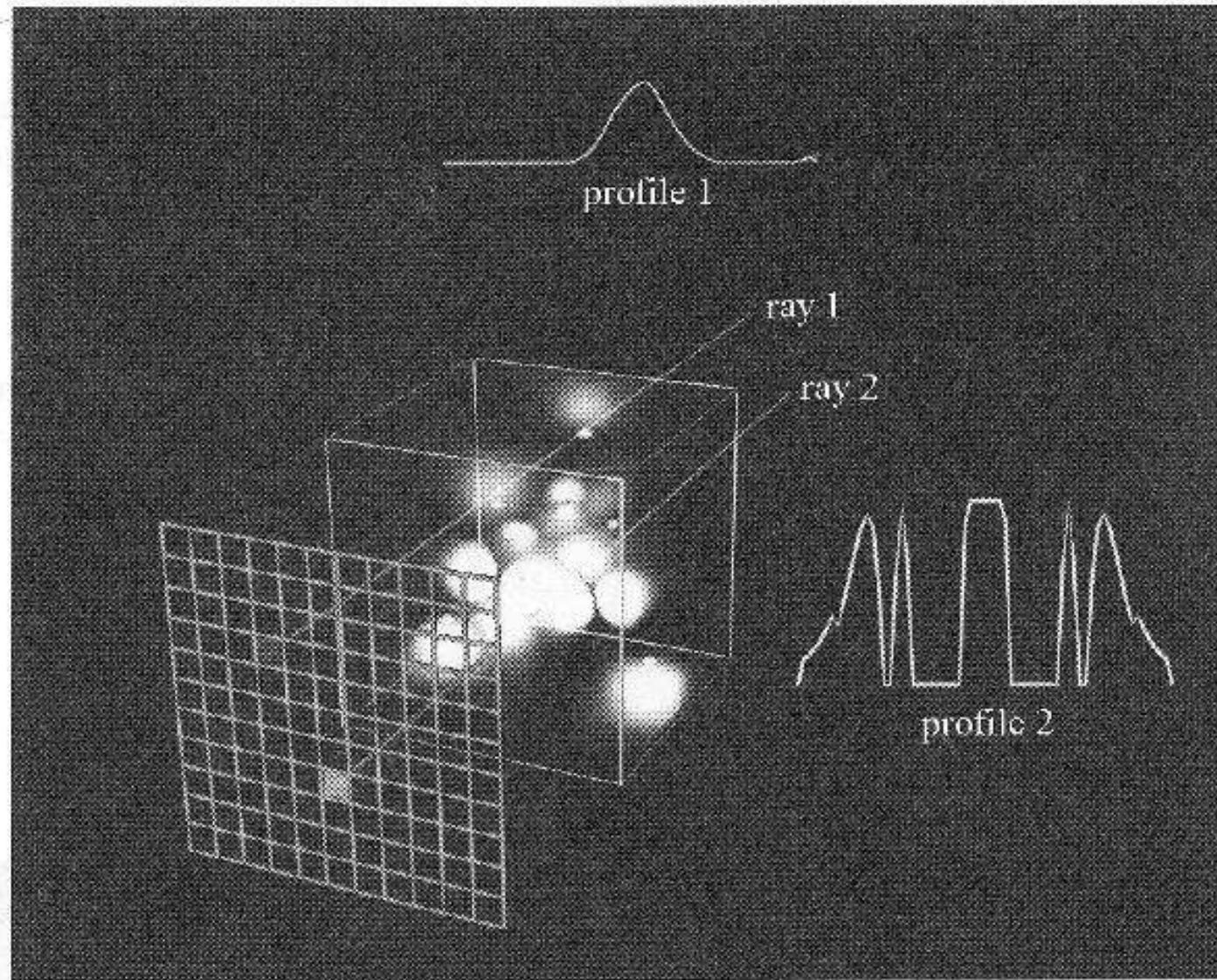
Volume Rendering

- There are two types of methods for determining the voxels that contribute to each pixel value:
 - Start with the pixel and search through the volume for the voxels
 - Start with the voxels and determine the pixels that they map to
- The first type of method is easier to explain

Volume Rendering

- We will assume an orthographic projection for all the techniques, perspective is possible for many of them, but it is more complicated
- Consider an individual pixel in the image
- For this pixel we build a ray:
 - The start of the ray is the center of the pixel
 - The direction of the ray is perpendicular to the screen

Volume Rendering



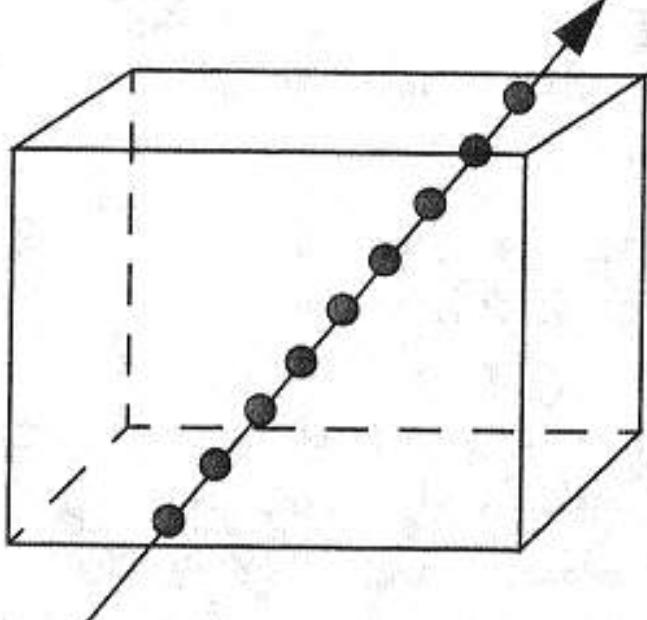
Volume Rendering

- This ray will pass through the volume and we can use it to sample the values along its path
- At each point along the path we evaluate the transfer function and use it to update the pixel value
- We have a ray for each pixel, so this can end up being an expensive computation, but possible on modern computers

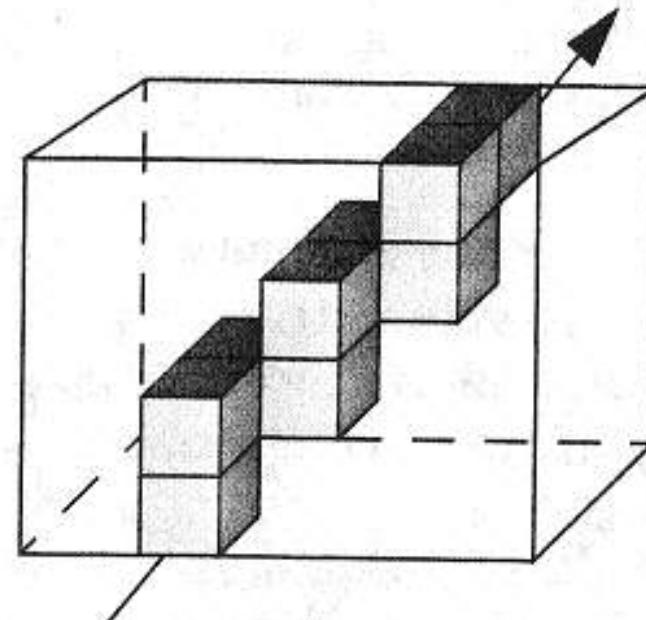
Volume Rendering

- How do we use the ray to sample the volume?
- There are two basic approaches here, we can either uniformly sample the volume or sample the volume at each voxel the ray strikes
- The first approach is more expensive, but typically leads to better results

Volume Rendering



Uniform sampling



Voxel by voxel traversal

Volume Rendering

- We start by determining the points where the ray enters and leaves the volume
- This extent is then divided into a number of uniform length steps
- We then sample the volume at each step along the ray
- Since the step locations don't correspond to data points we need to do some form of interpolation at each step

Volume Rendering

- One approach is to select the closest data point, but this doesn't produce very good results
- Another approach is based on linear interpolation
- Each voxel or cell has 8 data points, we linearly interpolate these 8 values based on the distance from the step to each of the corners

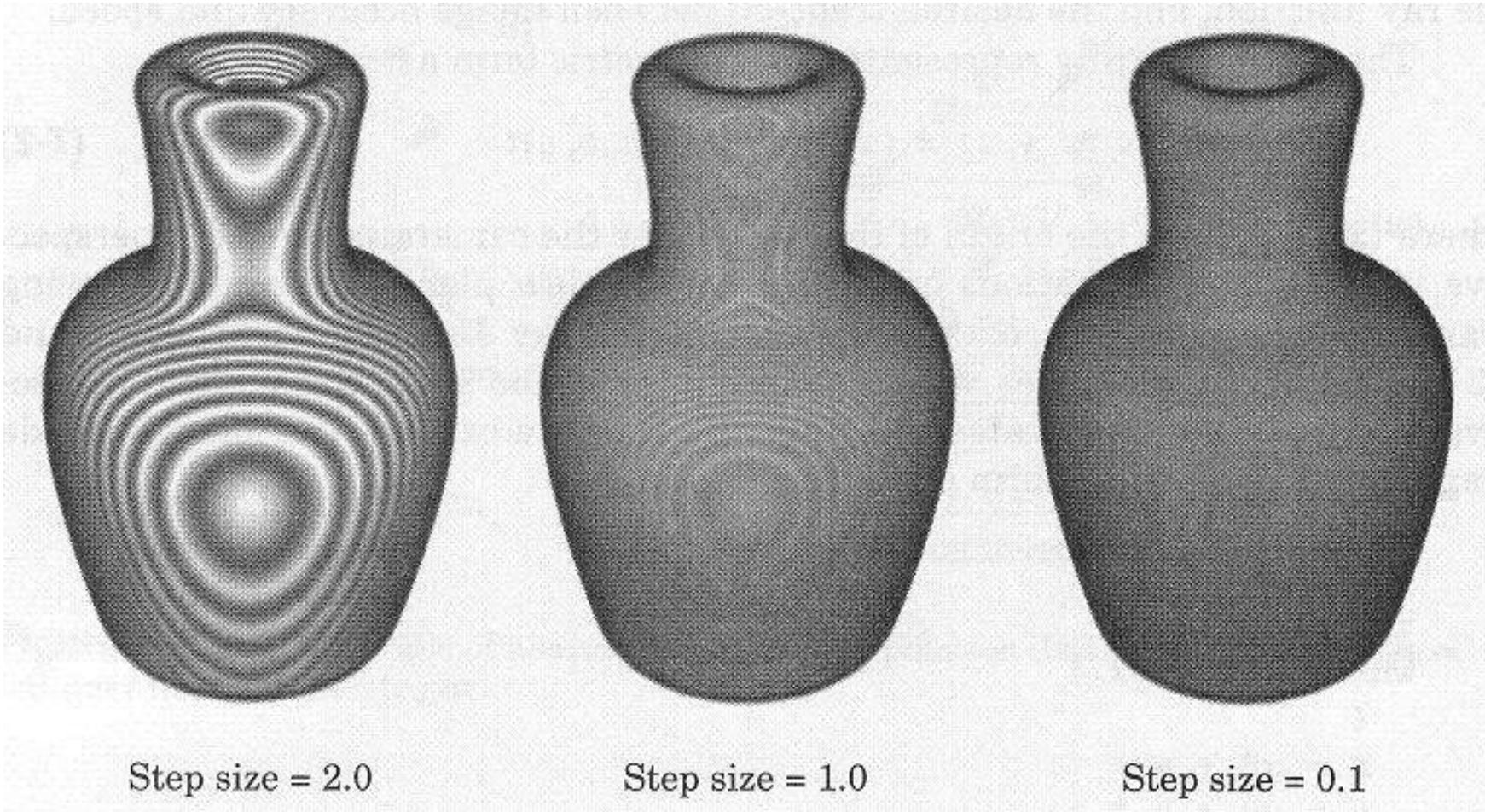
Volume Rendering

- The next thing to be considered is the number of steps
- The more steps, the longer it will take, so we want to keep the number of steps down
- If the steps are too large there is a chance that we could miss important features, or we could get aliasing which results in poor quality images

Volume Image

- The next slide shows three images of the same object
- These images were produced with step sizes of 2.0, 1.0 and 0.1 voxels
- It is clear that a step size of 2 voxels is too large and a step size of 0.1 voxels is too expensive
- A good choice is a step size of 1.0 to 0.5 voxels

Volume Rendering



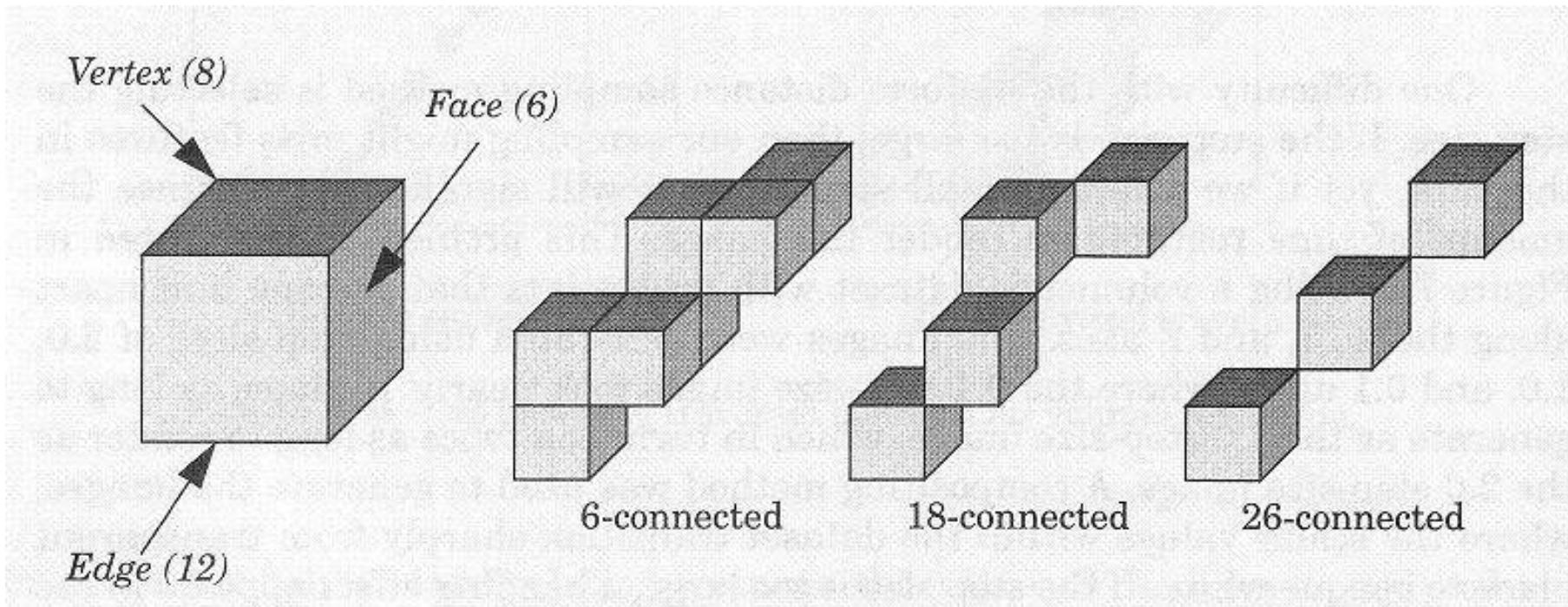
Volume Rendering

- The other approach to sampling the volume is to have the ray follow the voxels through the volume
- In this case we don't need to worry about interpolation and to some extent the step size
- The problem is that the path of the line is no longer straight, it goes in discrete jumps from one voxel to the next

Volume Rendering

- When we go from one voxel to the next we need to determine what we mean by the next voxel
- There are three different possibilities:
 - 6 connected: the two voxels share a face
 - 18 connected: the two voxels share a face or an edge
 - 26 connected: the two voxels share a face, edge or vertex
- These three possibilities are shown in the next slide

Volume Rendering



Volume Rendering

- 26 connected will produce the shortest path through the volume, but it might miss important details
- 6 connected will produce a much longer path, but will quite often consider voxels that only make a small contribution
- For most application 18 connected is a good compromise if speed is not a major concern

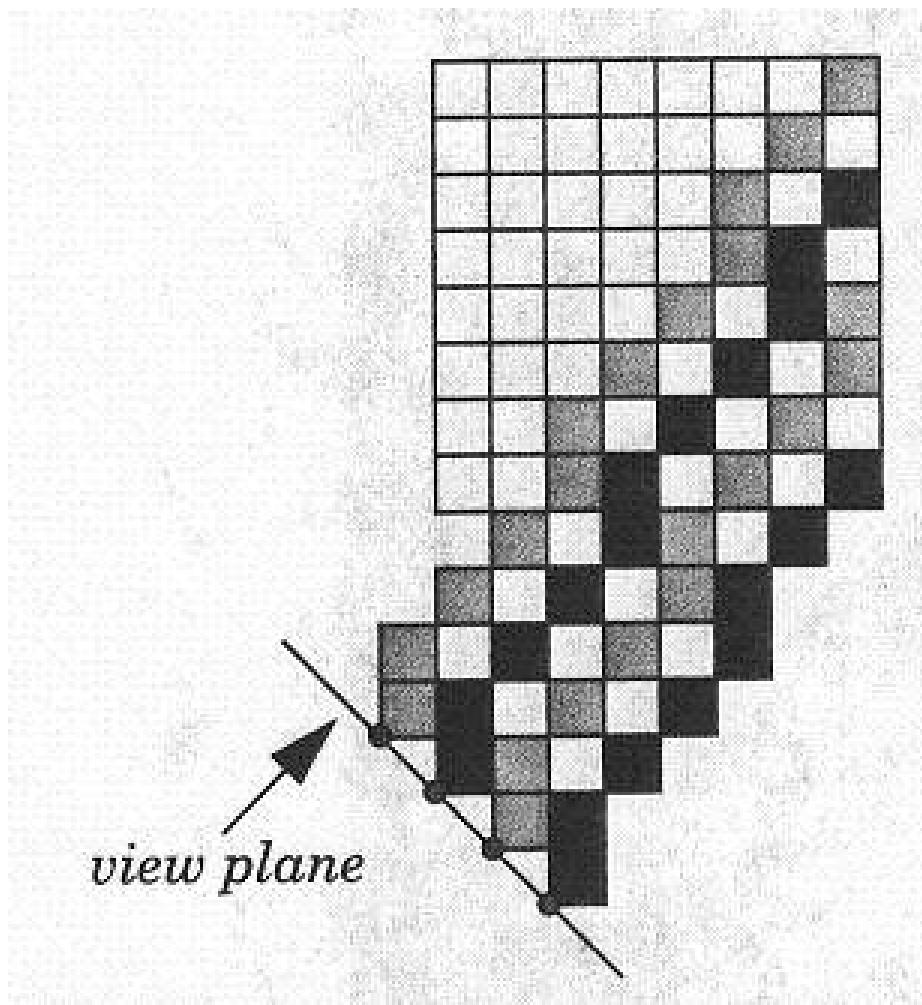
Volume Rendering

- One of the advantages of this approach is it can be optimized if an orthographic projection is used
- In this case the rays will take parallel paths through the volume, so we only need to compute the sequence of voxels visited once
- We can then translate this sequence by the appropriate amount for each pixel
- This can all be done with integer arithmetic

Volume Rendering

- There is one problem with both of these techniques, they work best if the view plane is parallel to the volume
- That is, the rays are perpendicular to one of the volume's sides
- If this isn't the case then some of the voxels in the volume will be missed, or we won't get a uniform sampling of the volume, this is shown in the next slide

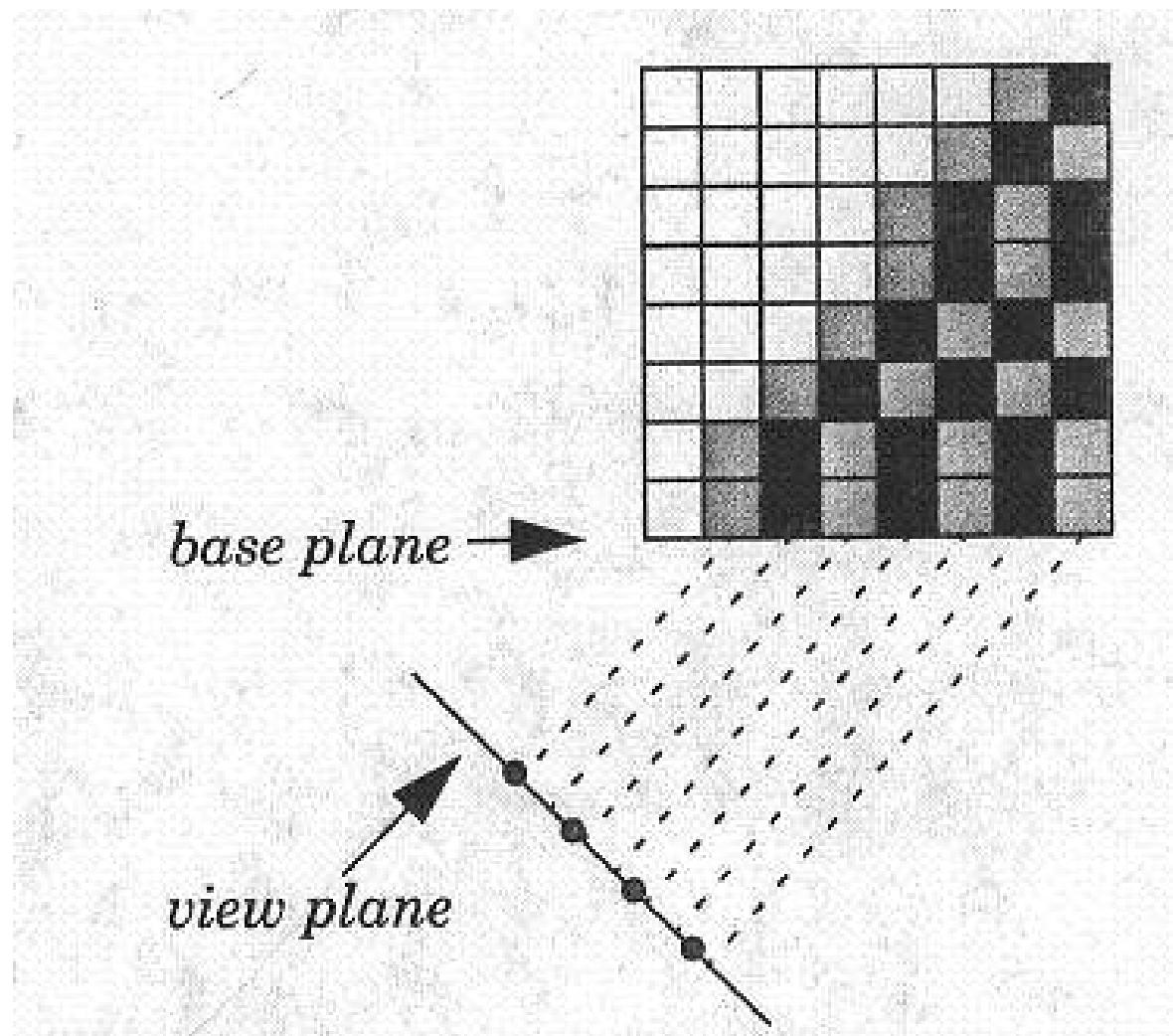
Volume Rendering



Volume Rendering

- This problem can at least be partially solved by starting the rays at the base of the volume and collecting the image on one surface of the volume
- This image must then be warped to the view plane as shown in the next slide
- This is a common problem that we will see in volume rendering, and the general solution involves warping to the view plane

Volume Rendering



Volume Rendering

- The other approach is to start with the voxels in the volume and determine the pixels they map to
- This is more efficient, but the images are not as good
- This is the approach that has been used in the past, due to efficiency, but is not used very much now

Volume Rendering

- The problem with this approach is determining the order to visit the voxels
- You want to start at the back of the volume, the voxels furthest away from the view plane and work forward
- This is similar to hidden surface, if we don't work from the back we get the wrong value for the transfer function

Volume Rendering

- If the view plane is parallel to the volume this is fairly easy to do
- In this case we have a well defined back to front order for the voxels
- We start at the plane furthest away from the view plane and work forward
- It is then relatively easy to determine the view plane pixel that each row of voxels maps to

Volume Rendering

- The earliest volume rendering algorithms used this approach
- If the view plane is not parallel to the volume then we have problems
- In this case producing a good voxel ordering is impossible, there will always be cases where voxels are visited in the wrong order, and this problem gets worse as the angle between the planes increases

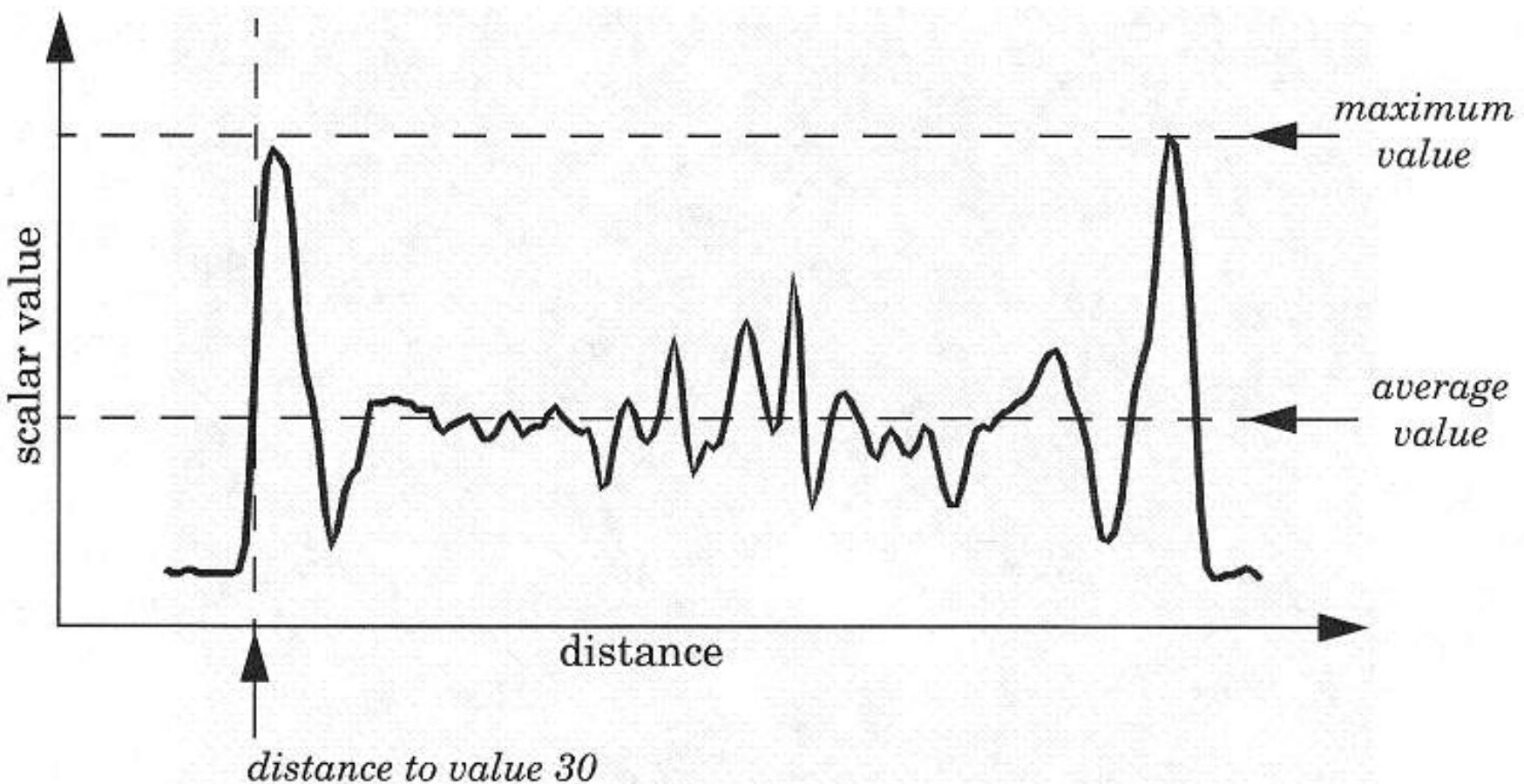
Volume Rendering

- One solution is to find the closest volume plane to the view plane, and then perform the volume rendering to the volume plane
- This gets us back to the previous case, which we can do
- Once we have finished constructing the image we then warp it to the image plane
- This works, but it doesn't give particularly good results

Volume Rendering

- We now turn our attention to the transfer functions, which provide the colour for each pixel
- The transfer function is evaluated at each sample point along the ray
- It can be a function of the current pixel value, the voxel value, the distance to the voxel, the gradient at the voxel or any other quantity that can be computed at the voxel or along the path
- The next slide shows a typical cross section through a volume

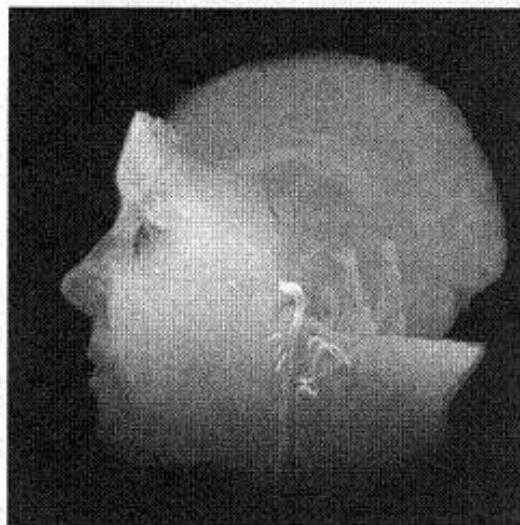
Volume Rendering



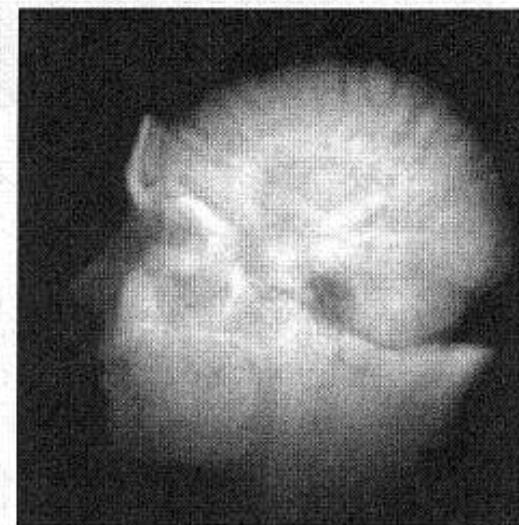
Volume Rendering

- There are a number of simple transfer functions that are quite easy to compute
- Four of them are shown on the next slide
- The maximum value or maximum intensity function takes the largest voxel value that is encountered along the ray
- This is fairly easy to understand and isn't effected very much by noise in the data

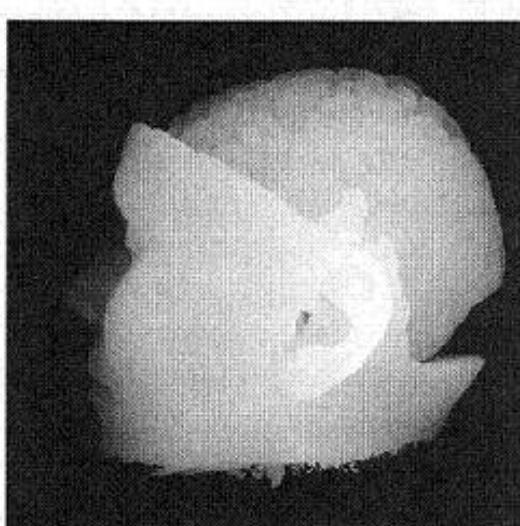
Volume Rendering



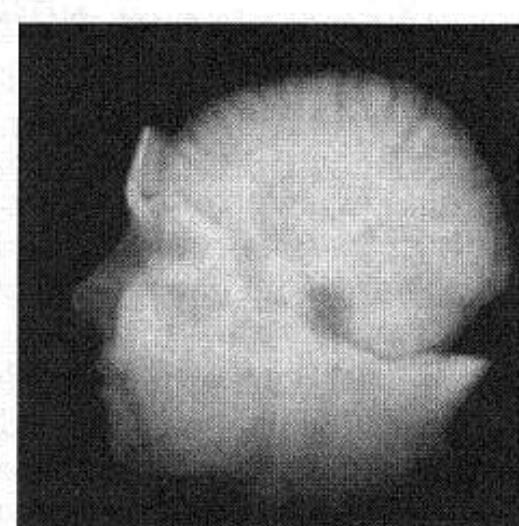
Maximum value



Average value



Distance to value 30



Composite

Volume Rendering

- The average value function computes the average voxel value along the ray, this tends to be more representative of the values along the ray
- The distance to value computes the distance to a particular voxel value and uses this as the pixel colour
- This tends to produce a more solid looking image

Volume Rendering

- The composite function is the most complex one
- This function views the voxel value as an opacity, the opposite of transparency
- The opacity is accumulated along the ray to give the pixel values
- Regions of low opacity tend to look transparent and allow us to look through the volume

Case Study – Volume Rendering

- The average transfer function is fairly easy to compute, so we will try it with some of our data
- We need to make very few changes to the volume program:
 1. Change the texture wrap to clamp
 2. Add another program parameter that identifies the set of shaders to be used

Case Study – Volume Rendering

- We can easily do uniform sampling for tracing the ray with a GPU, we only need to position the plane at the sample points
- We start by moving the sample plane to the back of the volume in the vertex shader
- We then compute a vector in the sampling direction, which is passed to the fragment shader
- The code for this is shown on the next slide

Case Study – Volume Rendering

```
in vec4 vPosition;  
  
uniform mat4 model;  
uniform mat4 view;  
uniform mat4 projection;  
  
out vec3 texCoord;  
out vec3 dir;  
  
void main(void) {  
    vec4 position = vPosition;  
  
    gl_Position = projection * view * model * vPosition;  
    position.y -= 0.5;  
    texCoord = (model * position).xyz;  
    dir = (model * vec4(0.0, 1.0, 0.0, 0.0)).xyz;  
}
```

Case Study – Volume Rendering

- The code for the fragment shader is shown on the next slide
- It is basically a for loop that iterates through the sample locations
- At each sample location a texture lookup is performed and added to the total
- This is then divided by the number of slices to give the average, which is used as the pixel colour

Case Study - Volume

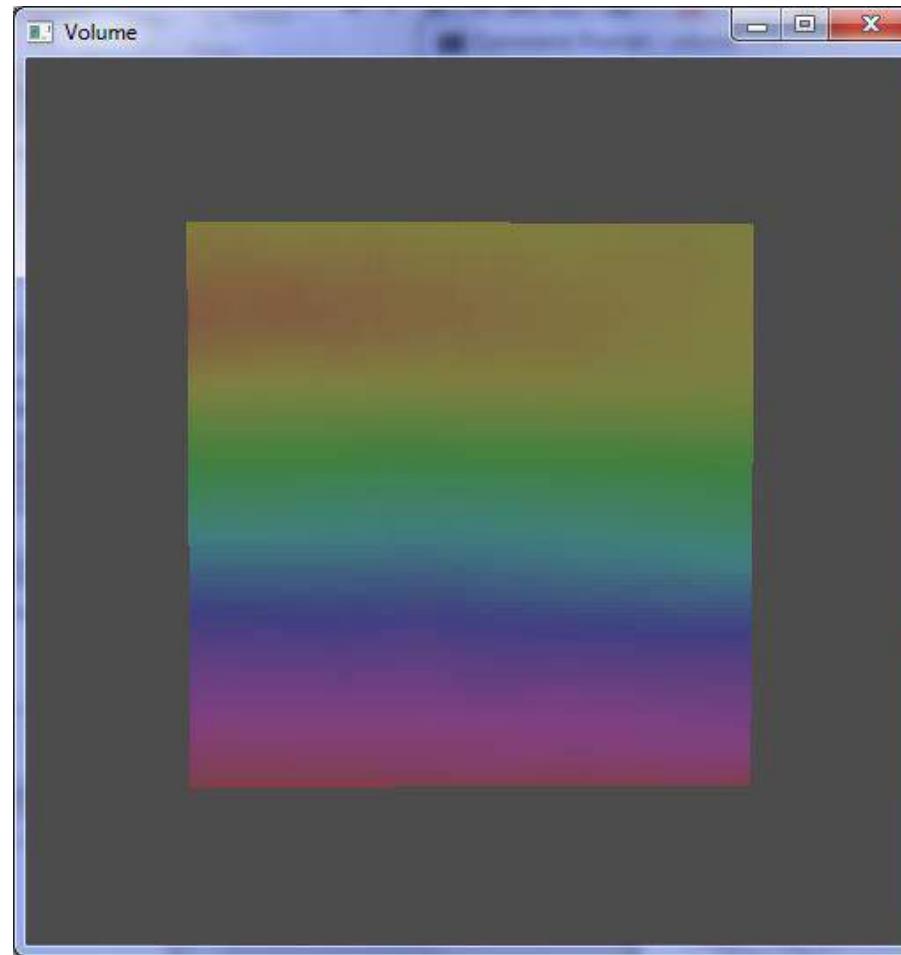
```
in vec3 texCoord;
in vec3 dir;
uniform sampler3D tex;

void main(void) {
    int slices = 10;
    float step = 0.1;
    vec4 average;
    int i;
    vec3 coord = texCoord;

    average = vec4(0.0, 0.0, 0.0, 1.0);
    for(i=0; i<slices; i++) {
        average += texture(tex, coord);
        coord += step*dir;
    }
    average /= slices;

    gl_FragColor = average;
}
```

Case Study - Temperature



Volume Rendering

- Opacity gives us our first way of separating structures within the volume
- Each structure will have a range of voxel values
- To extract a particular structure we make the opacity for its voxel values high and the opacity for other voxel values low
- We can extract several structures at the same time using this approach

Volume Rendering

- This approach works well when each structure has a well defined range of voxel values, and these ranges don't overlap
- The problem becomes harder when there is an overlap between the ranges, and in that case we need to use extra information to separate them, but where does this information come from?

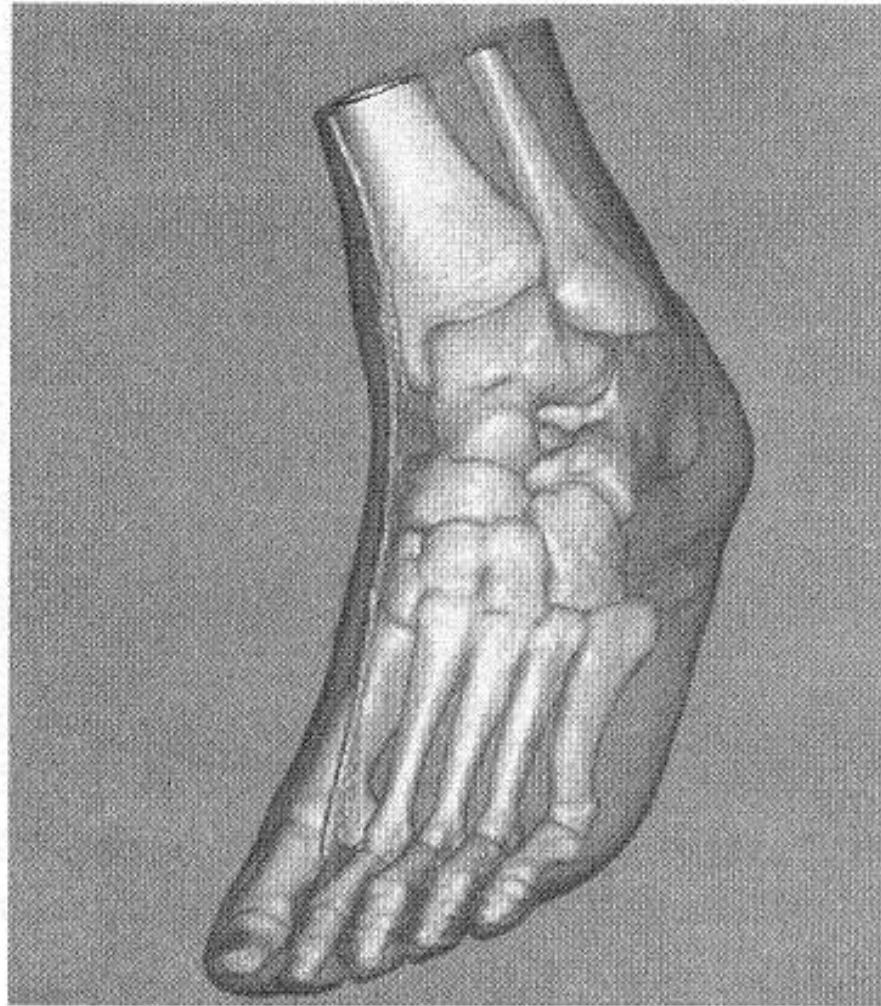
Volume Rendering

- One approach is to use the magnitude of the gradient vector
- The basic idea is that the voxel values will be changing quickly in the area between structures
- The voxel value itself may not unambiguously indicate a new structure, but the rate at which the voxel values are changing could indicate it

Volume Rendering

- The next slide shows a visualization produced using the gradient magnitude to classify the structures
- One of the problems with using the gradient is we must have some way to compute it
- We only have voxel values at the data points, and these are interpolated across the voxel in many cases

Volume Rendering



Volume Rendering

- A finite difference approach can be used to compute the gradient, using central difference formulas to increase the accuracy
- The problem with using the gradient is the expense of computing it
- One approach is to compute the gradient once, and store it with the voxel, unfortunately this uses a lot of space

Volume Rendering

- The voxel value is either 8 or 16 bits, but the gradient is 3 floating point numbers, or 12 bytes
- Only the magnitude could be stored at the voxel, but that leads to poorer results when interpolation is used
- Another approach is to classify all of the voxels in a preprocessing step and store the classification in a separate volume

Volume Rendering

- This also causes problems with interpolation
- We have a structure identifier at each voxel, interpolation doesn't make any sense with this
- What is the average of muscle and bone?
- We could use the data point values to guide the interpolation within the voxel

Volume Rendering

- A number of special purpose voxel classification schemes have been developed for particular applications
- A good example of this is medical visualization where it can be hard to distinguish between the different types of tissues
- The topic is beyond this course

Case Study – Volume Rendering

- We can apply this approach to the pressure data
- In the middle of the hurricane there is a low pressure area that was hard to see in the plane visualization
- We can use the grey scale texture to extract the area of low pressure and give it a solid colour, all the other pixels will have a background colour

Case Study – Volume Rendering

- The fragment shader that does this is shown on the next slide
- We start by extracting a value at the current texture coordinate, since it's a grey scale texture we extract one of the colour components
- Then check to see if it is in the range 0.1 to 0.5
- The lower limit excludes the land values and the upper limit extract the low pressure data

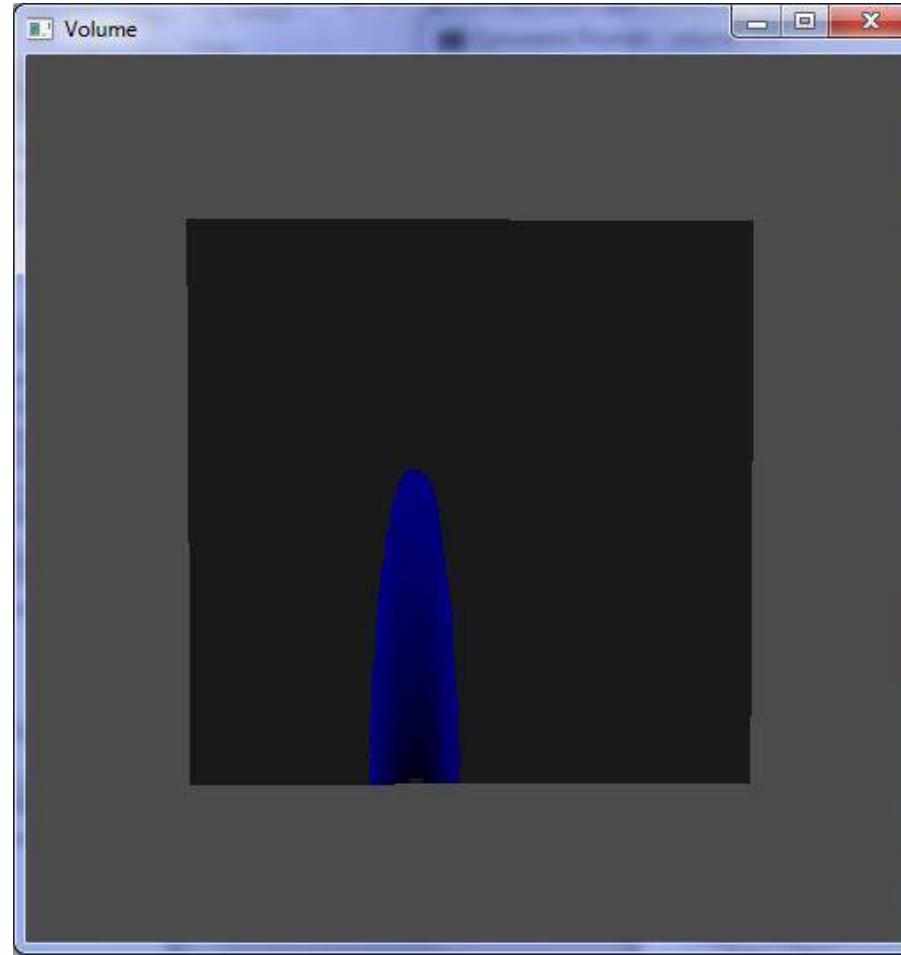
Case Study – Volume

```
in vec3 texCoord;
in vec3 dir;
uniform sampler3D tex;

void main(void) {
    int slices = 10;
    float step = 0.1;
    vec4 colour;
    int i;
    vec3 coord = texCoord;
    float value;

    colour = vec4(0.1, 0.1, 0.1, 1.0);
    for(i=0; i<slices; i++) {
        value = texture(tex, coord).r;
        if((value > 0.1) && (value < 0.5))
            colour = vec4(0.0, 0.0, value, 1.0);
        coord += step*dir;
    }
    gl_FragColor = colour;
}
```

Case Study – Pressure



Case Study – Volume Rendering

- In both cases we have done a very coarse sampling of the texture
- We could get better results by increasing the number of slices, but this will slow down the display
- We've also used nearest neighbour for our sampling, we could try linear sampling
- There are many variations that we could try on these basic shaders

Volume Rendering Rendering

- There is one last thing to consider
- We are producing images of 3D structures, and we know that proper lighting makes it easier to interpret these structures
- Should we consider applying lighting or illumination to our volume renderings?
- The following slide shows a comparison that indicates the effectiveness of lighting

Volume Rendering



Maximum intensity



Composite (unshaded)



Composite (shaded)

Volume Rendering

- It has been argued that illumination isn't a good idea, particularly in medical applications
- Volume rendering produces a colour for each pixel, the illumination process will alter this colour possibly changing the interpretation of the data
- On the other hand without illumination its difficult to interpret the 3D structure of the data, this could be equally important, knowing where to make the cut in surgery

Volume Rendering

- How do we illuminate a volume?
- Before we had surfaces and normal vectors, but now we don't
- We could view each voxel as a micro-surface, it gives the position to be illuminated
- We could try to estimate a normal vector from the gradient, and possibly information from adjacent voxels

Volume Rendering

- In some cases this works, particularly when the different structures are well separated
- It also has the advantage of being relatively efficient
- A more accurate way of computing illumination is to use transport theory, a model of how light is scattered and absorbed in a volume

Volume Rendering

- The light arriving at a pixel along a ray starting at t_0 and traveling in direction ω is given by:

$$I(t_0, \omega) = \int_{t_0}^{\infty} Q(t) e^{-\int_{t_0}^t \sigma_a(t') + \sigma_{sc}(t') dt'} dt$$

- Where:
 - $\sigma_a(t)$ is the absorption of light
 - $\sigma_{sc}(t)$ is the scattering of light

Volume Rendering

- The quantity $Q(t)$ is given by:

$$Q(t) = E(t) + \sigma_{sc}(t) \int_{4\pi} \rho_{sc}(\omega' \rightarrow \omega) I(t, \omega') d\omega'$$

- Where:
 - $E(t)$ is the light emitted at t
 - $\rho_{sc}(\omega' \rightarrow \omega)$ the scattering function, the amount of light scattered from direction ω' to direction ω
- This is all integrated over the solid angle arriving at the point t from all directions

Volume Rendering

- While these equations are accurate, they are very difficult to solve
- A number of approximations to them have been developed, but they end up looking like the composition transfer function in their simplest case

Volume Rendering

- This covers the basic ideas behind volume rendering
- There are a number of other techniques, but they are mainly variations on the ideas that we have already seen
- With faster hardware volume rendering is becoming a standard visualization technique

Summary

- Examined the basic structure of graphics applications with an emphasis on visualization
- Examined the extraction of particle paths from velocity fields, and their display
- Presented a survey of volume rendering techniques
- Used volume rendering to explore some of the variable in our hurricane Isabel data set

CSCI 3090

Colour

Mark Green
Faculty of Science
Ontario Tech

Goals

- By the end of todays class you will:
 - Have a basic understanding of colour vision
 - Understand the basic colour spaces
 - Understand the issues involved in using colour to represent data

Introduction

- An understanding of colour is important for advanced topics in computer graphics
- A knowledge of colour is also useful for visualization
- Colour is a complex topic since it not only involves physics, but also the physiology and psychology of the human vision system

Colour

- Humans have two types of vision systems
- They have a colour vision system that is used under normal lighting conditions, this will be our main interest in this section
- There is also a black and white vision system that is used under low light conditions
- This system basically measures light intensity

Colour

- Both vision systems are based on detectors that give a single value for all of the light that they receive
- The human vision system can see radiant energy in the 380 to 800 nm range, so this detector is computing some form of weighted average over this range of wavelengths

Colour

- The detector response is given by:

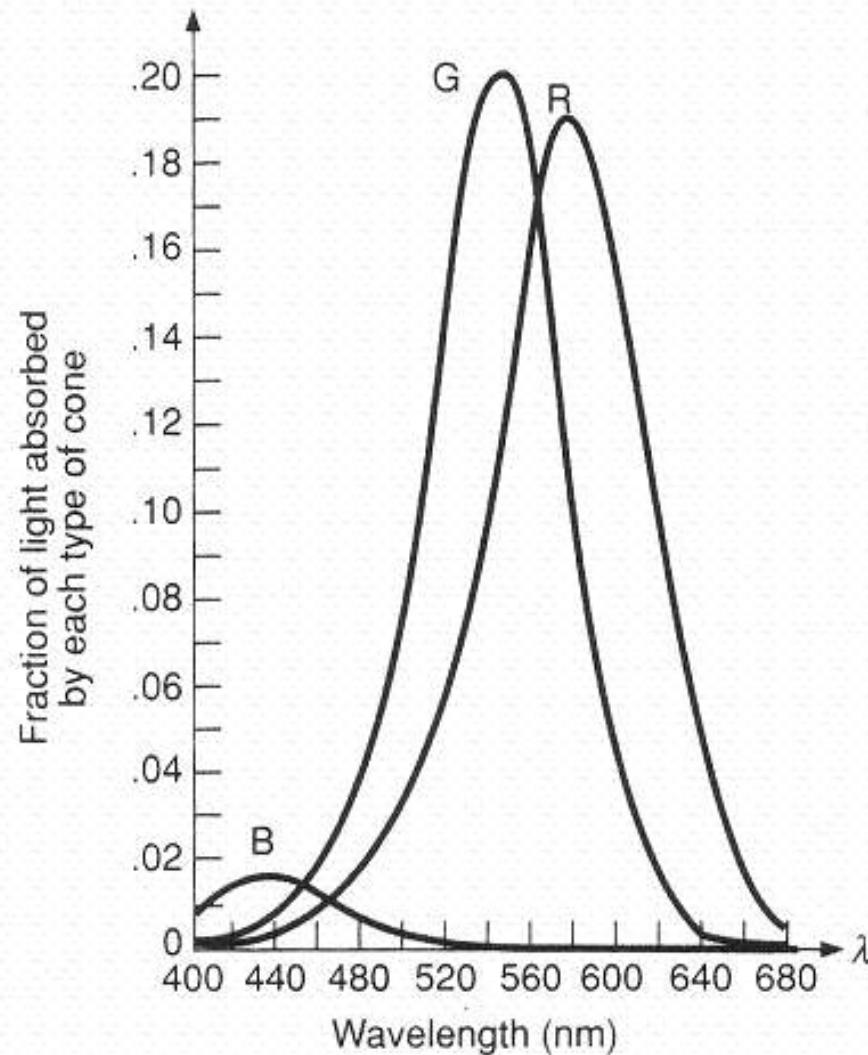
$$response = k \int_{380}^{800} w(\lambda)L(\lambda)d\lambda$$

- In this equation λ is the wavelength, $L(\lambda)$ is the amount of light reaching the detector, as a function of wavelength and $w(\lambda)$ is the weighting function
- The values of $w(\lambda)$ are usually tabulated, since they are often determined experimentally

Colour

- The human colour vision system is based on three types of detectors, which are called cones
- The $w(\lambda)$ for these three detectors are shown on the next slide
- Note that they overlap in λ , so that various combinations of these three signals can produce the same colour
- They are not a linearly independent coordinate system

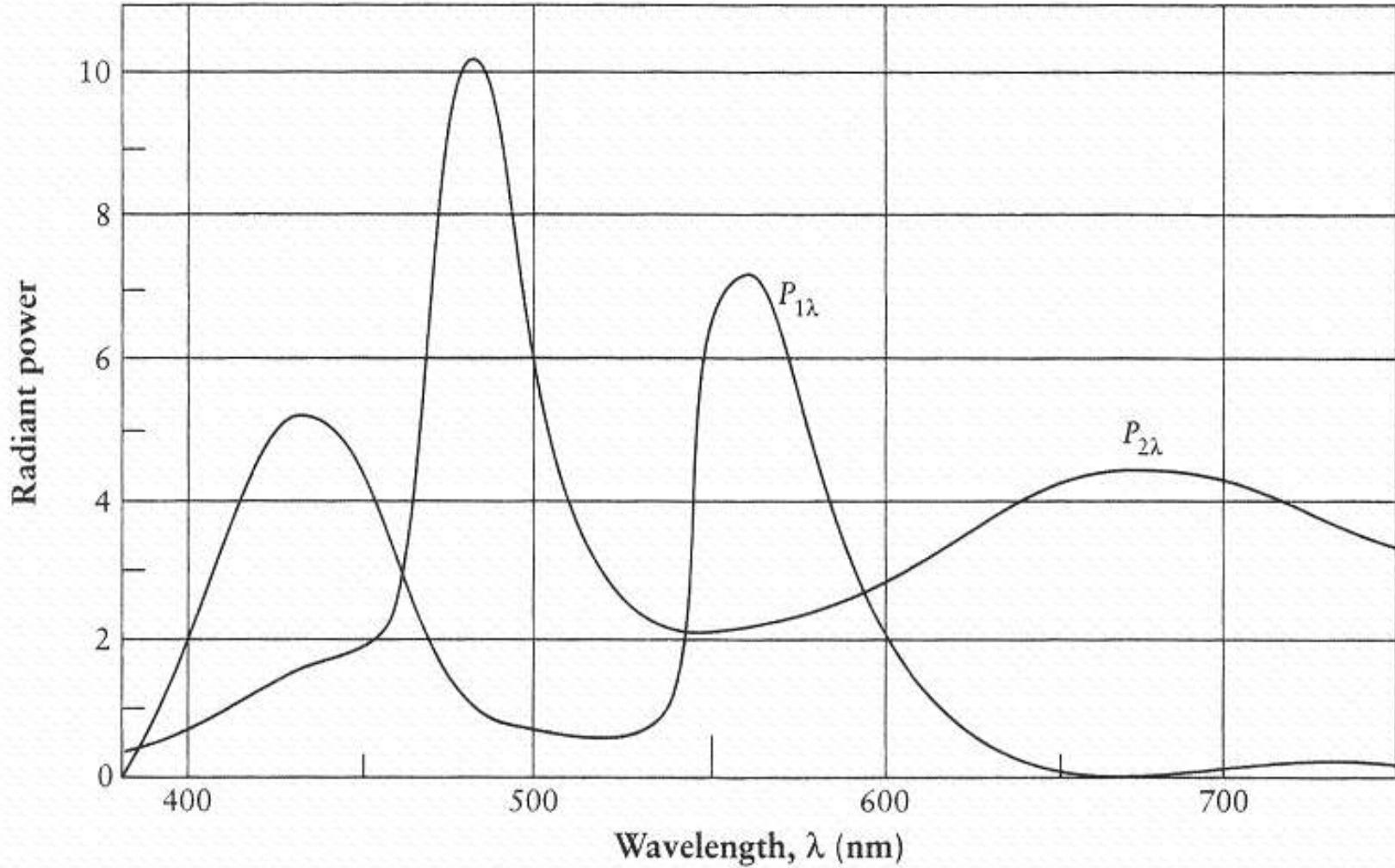
Colour



Colour

- We can have two spectra that produce the same colour, in this case the spectra are called metamers
- An example of this is shown on the next slide
- These two spectra look totally different, but when viewed they have the exact same colour
- This can sometimes be used to our advantage
- We can choose the spectra that is easier to produce and use that to generate the colour

Colour



Colour

- Since colour is perceptual as well as physical, it is difficult to develop a consistent mathematical theory for it
- In fact the $w(\lambda)$ for the human vision system were not accurately determined until the 1980s
- Before that an indirect technique was used to study our colour vision system

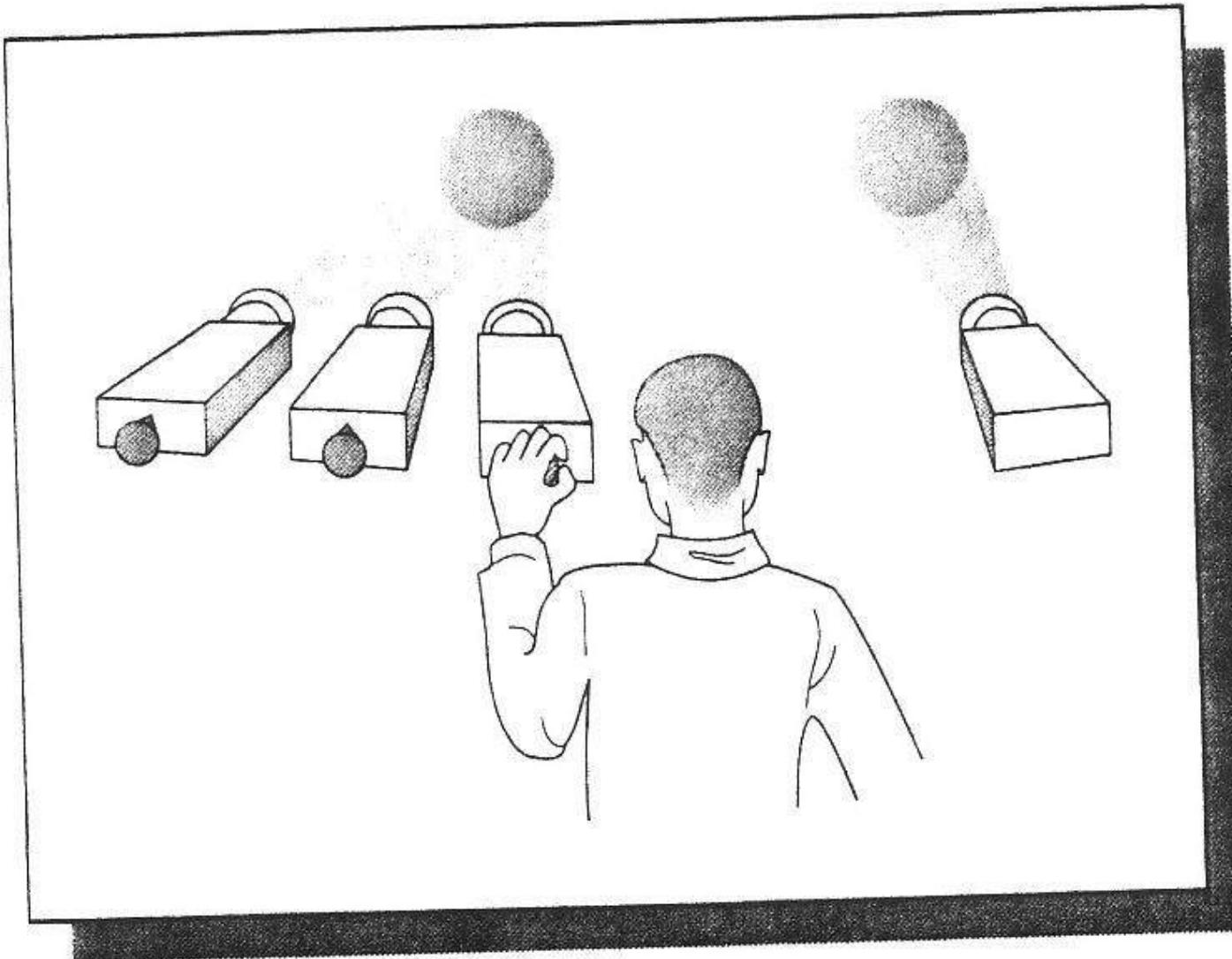
Colour

- This approach was based on using three coloured lights with independent dimmer controls
- These lights were projected on a screen so they exactly overlapped
- Different colours could be produced by setting the dimmers to different values
- This could be used to match a given colour

Colour

- To do a colour match a fourth light was used with a colour filter
- This light was projected onto a different part of the screen and people adjusted the dimmers to match it
- Usually red, green and blue lights were used, and it was found that there were some colours that couldn't be match

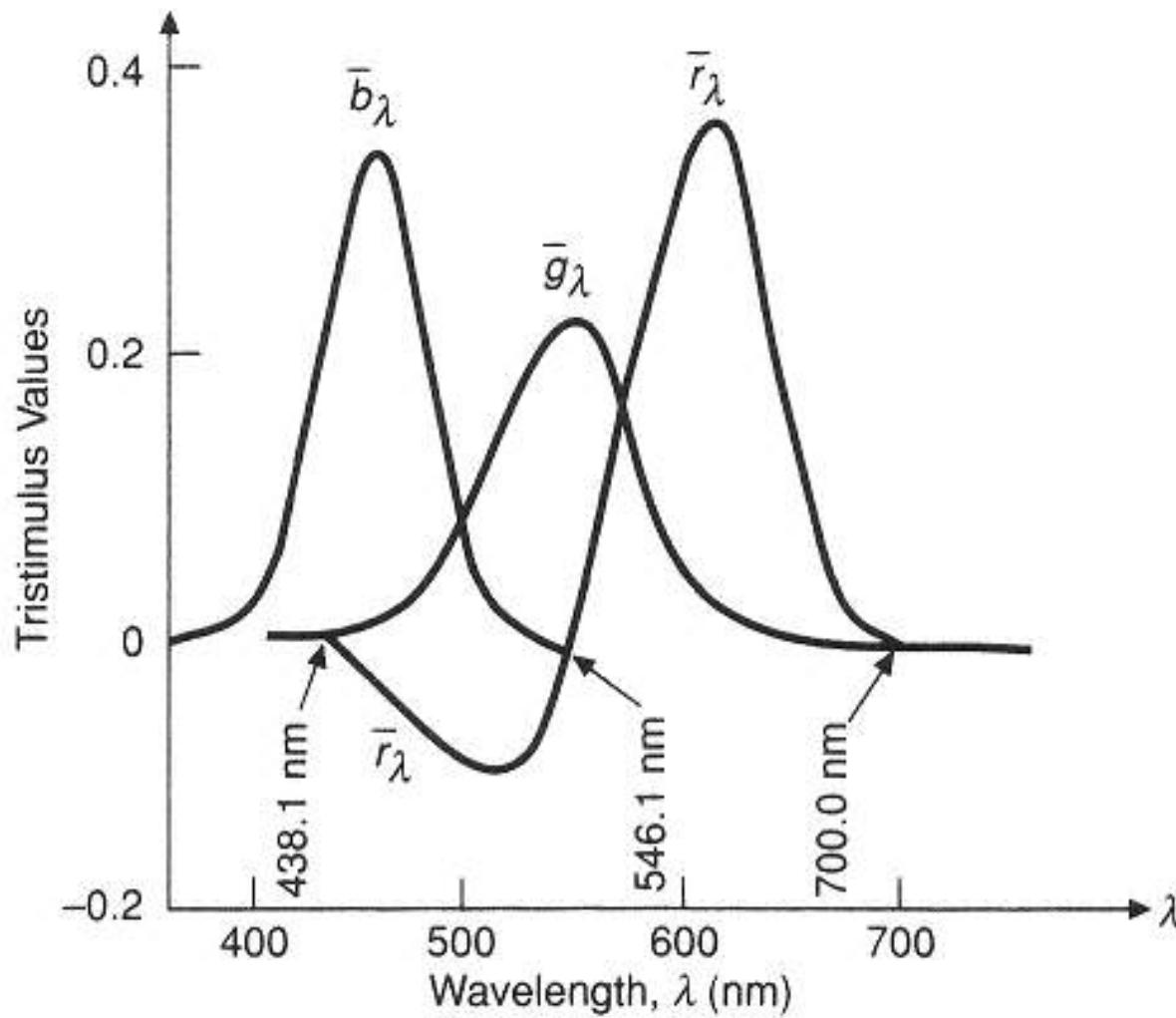
Colour



Colour

- The following slide shows the amount of red, green and blue required to match different visible wavelengths
- Note that in some cases we need a negative amount of red, which is not physically possible
- It turns out that we can't choose three physical colours that can be combined to produce the complete spectrum

Colour



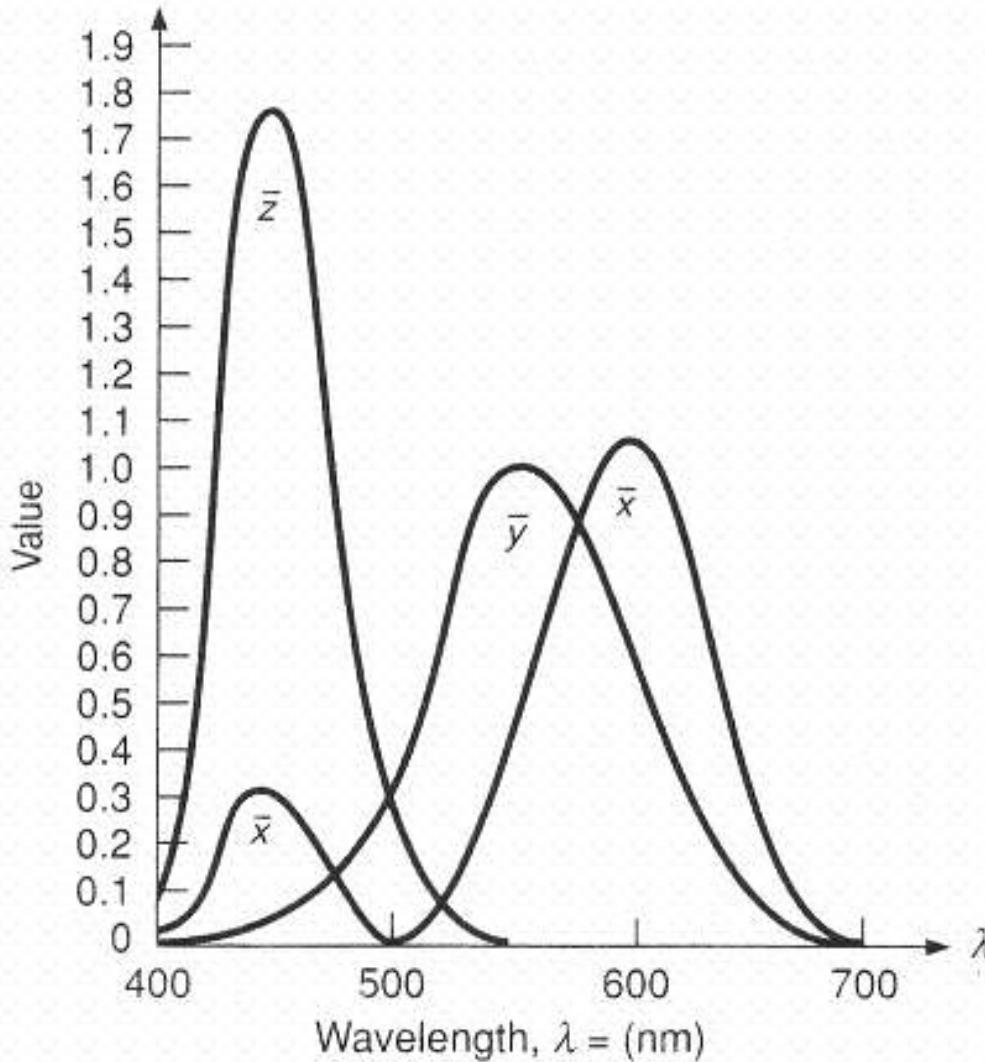
Colour

- This looks to be a problem, we would really like to have one system for describing colour, that is a colour standard
- In the 1930s the CIE worked out such a system
- They made the observation that since they were producing a standard they didn't need to use real colours, they could invent their own three basis colours or primaries

Colour

- The CIE primaries are called X, Y and Z and their weight functions are shown on the next slide
- Y represents the luminance or brightness of the colour
- X and Z represent the hue of the colour
- This is a particularly convenient way of representing colours

Colour



Colour

- The X, Y, Z colour space is not particularly easy to work with, so we normally project it onto the X+Y+Z=1 plane using the following equations:

$$x = \frac{X}{X + Y + Z}$$

$$y = \frac{Y}{X + Y + Z}$$

$$z = \frac{Z}{X + Y + Z}$$

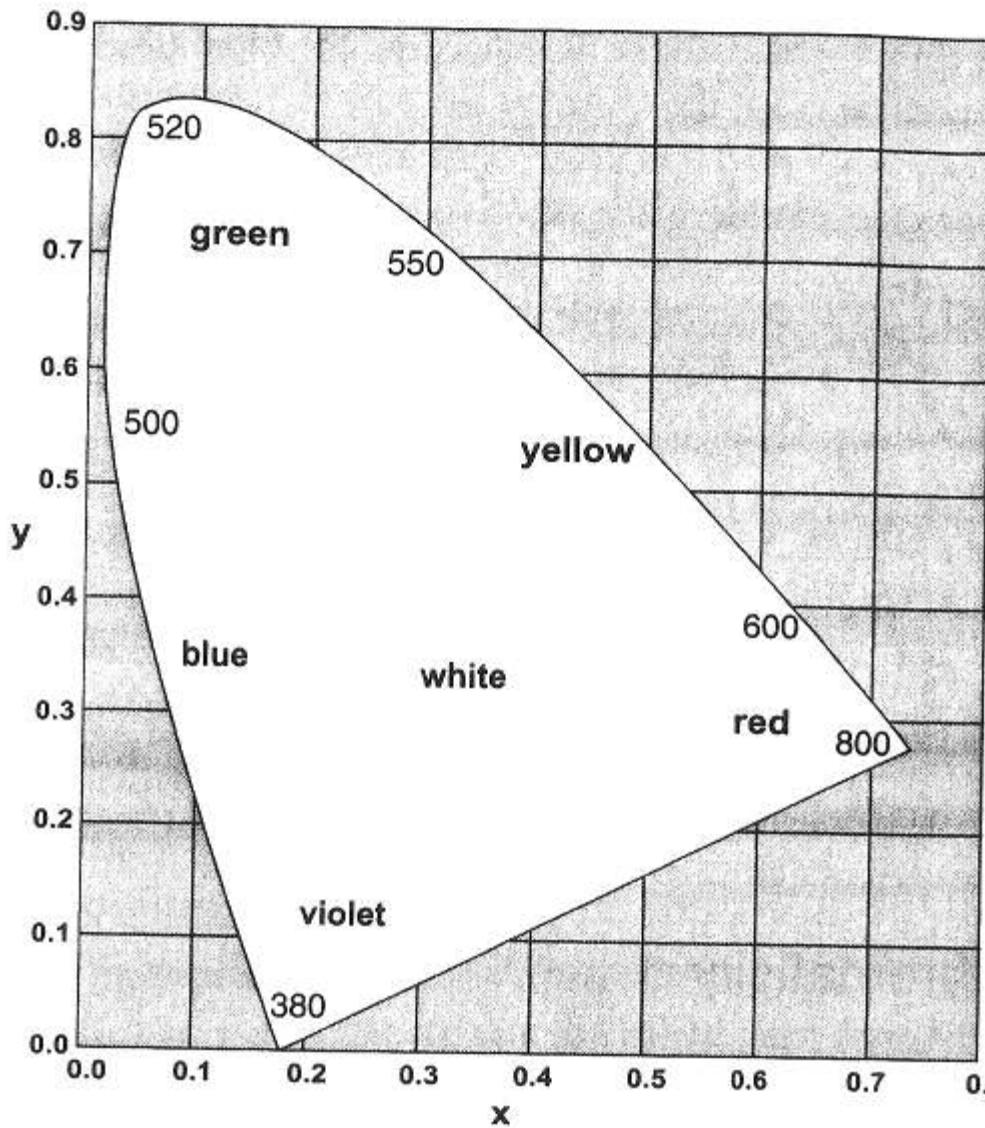
Colour

- Note that $z = 1-x-y$, so we really only need two coordinates
- This transformation leaves us with the hue having removed the luminance from the colour
- Luminance is the amount of white in a colour, and hue is the basic colour
- For example pink and red have the same hue, we get pink by adding white to red

Colour

- We can plot can plot x and y for the CIE colour space, this plot is shown in the next slide
- Note that even though x and y could go from 0 to 1, only a portion of this space corresponds to real colours
- This type of diagram is called a gamut and gives the range of colours available in a colour space

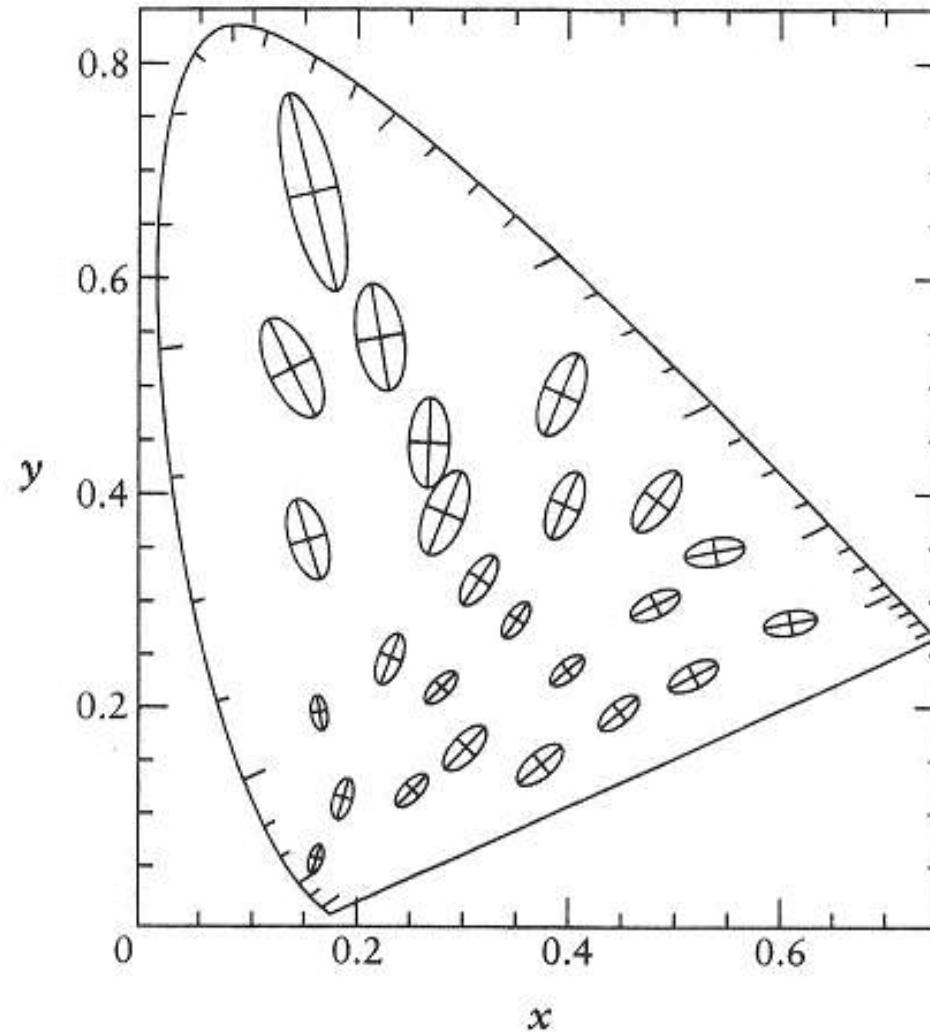
Colour



Colour

- There is a major problem with both the RGB and XYZ colour spaces
- Neither of these spaces are perceptually linear
- That is, a difference in one part of the space may look larger or smaller than a difference in another part of the space
- This is shown in the next slide where the ellipses show regions of perceptually equal colours
- Notice how the sizes and orientations vary over the gamut

Colour



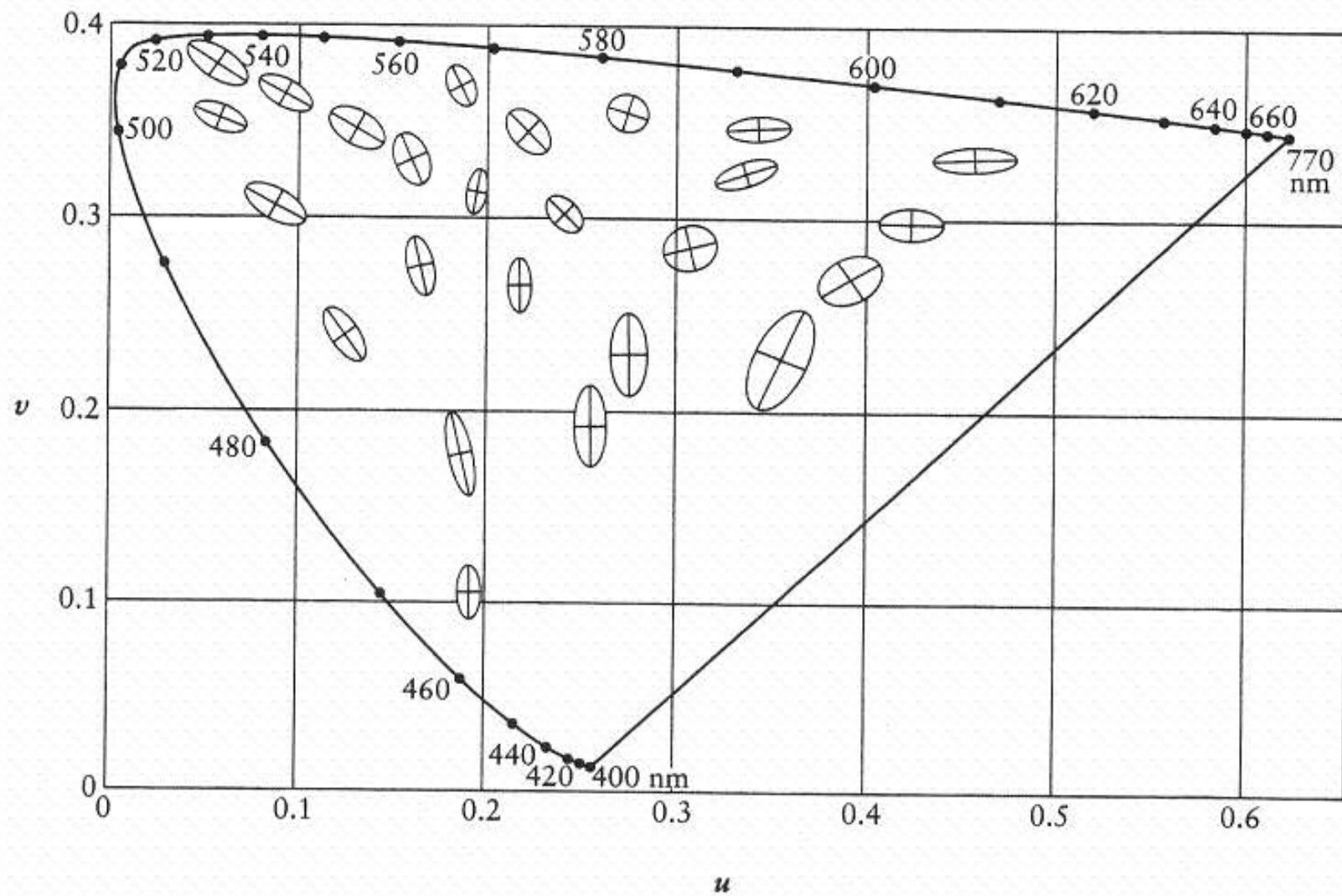
Colour

- This non-linearity can cause a number of problems
- First we need to be careful when we interpolate from one colour value to another, since the interpolation may not be perceptually linear
- Second if we use colour to visualize a scalar we may perceive differences that really aren't there, this misleads the viewer

Colour

- We really want a colour space that is perceptually linear
- Differences in colours map linearly to differences in the colour space
- There have been a number of attempts to do this and completely linear spaces are hard to produce
- The $L^*u^*v^*$ and $L^*a^*b^*$ are close as shown in the next slide (u^*v^* plane)

Colour



Colour

- Both of these colour spaces are based on a reference white colour (X_n , Y_n , Z_n) where Y_n is scaled to 100
- Both of these colour spaces use the same value for L^* , which is given by:

$$L^* = \begin{cases} Y / Y_n \geq 0.008856 & 116 \left(\frac{Y}{Y_n} \right)^{1/3} - 16 \\ Y / Y_n \leq 0.008856 & 903.3 \left(\frac{Y}{Y_n} \right) \end{cases}$$

Colour

- For u^* and v^* we have

$$u^* = 13L^*(u' - u'_n)$$

$$v^* = 13L^*(v' - v'_n)$$

$$u' = \frac{4X}{X + 15Y + 3Z}$$

$$v' = \frac{9Y}{X + 15Y + 3Z}$$

$$u'_n = \frac{4X_n}{X_n + 15Y_n + 3Z_n}$$

$$v'_n = \frac{9Y_n}{X_n + 15Y_n + 3Z_n}$$

Colour

- For a^* and b^* we have the following:

$$a^* = 500L^* \left[f\left(\frac{X}{X_n}\right) - f\left(\frac{Y}{Y_n}\right) \right]$$

$$b^* = 500L^* \left[f\left(\frac{Y}{Y_n}\right) - f\left(\frac{Z}{Z_n}\right) \right]$$

$$f(r) = \begin{cases} r \geq 0.008856 & r^{1/3} \\ r < 0.008856 & 7.787r + 16/116 \end{cases}$$

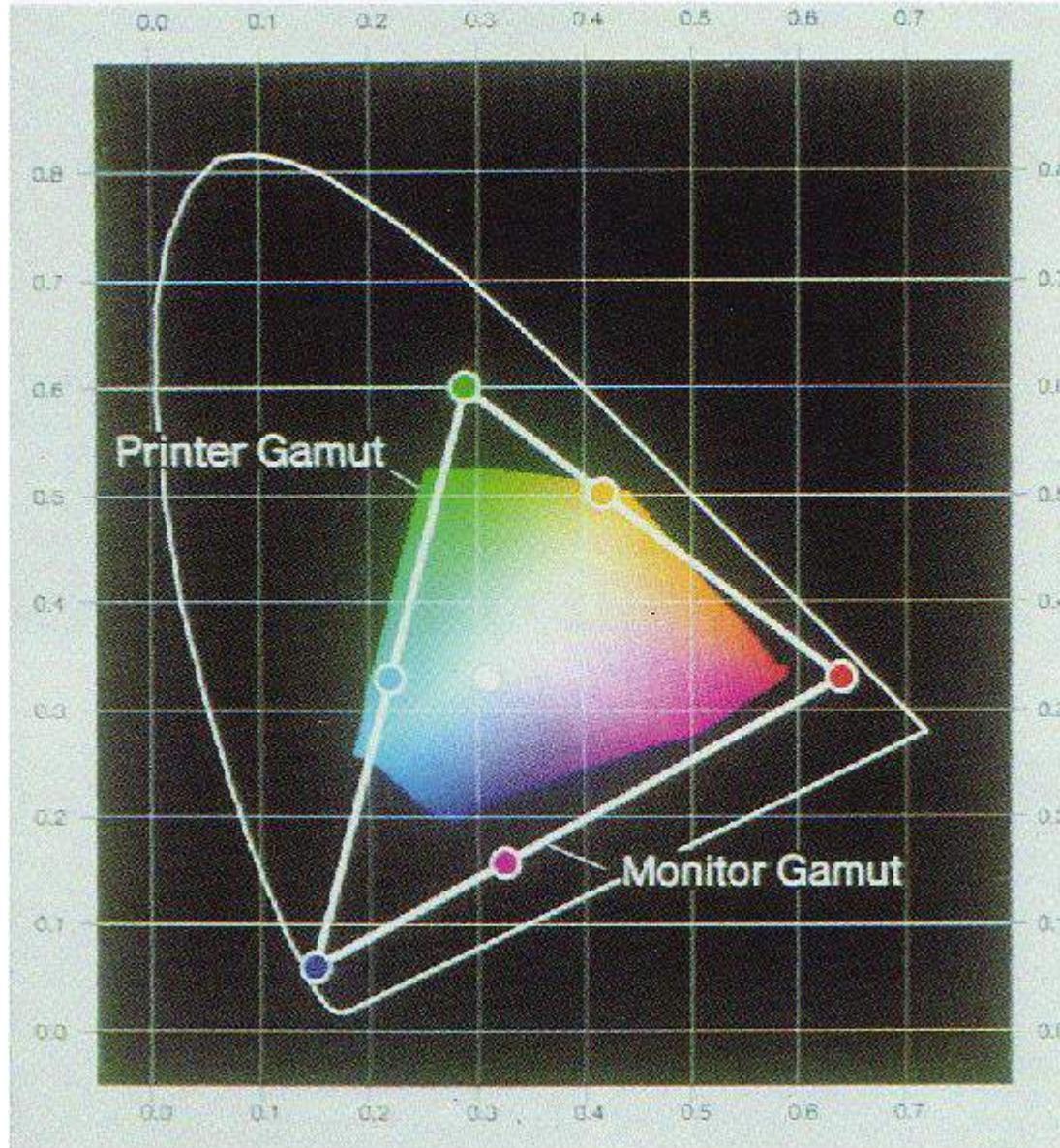
Colour

- We can develop similar formula for going from $L^*u^*v^*$ or $L^*a^*b^*$ to XYZ
- How does this relate to real devices?
- For any monitor there is a simple transformation from XYZ to RGB space and back
- We treat RGB and XYZ as vectors and then use a 3x3 matrix to transform them

Colour

- We can do the same thing with printers, which use a different colour space
- The gamuts of a typical monitor and printer are shown on the next slide
- Note that they are not equal
- There are colours that we can display on a monitor, but we cannot print
- This is a major problem in the publishing industry

Colour



Colour

- We need a matrix to convert XYZ to RGB, but where does this matrix come from?
- This is the problem, it varies from one monitor to another and we need to measure the monitor we are using in order to compute the matrix
- The procedure for doing this is described in textbooks on colour and advanced graphics

Summary

- Colour is a complex subject, and we have only scratched the surface
- We need to be careful when we use colour in visualization so we don't mislead the viewer
- For commercial production we need to calibrate all of our display devices so we produce consist colour across all of them

CSCI 3090

Graphics Hardware

Mark Green

Faculty of Science

Ontario Tech

Goals

- By the end of this lecture you will have seen:
 - The evolution of graphics hardware over the past few decades
 - The unified shader architecture
 - Techniques for scaling performance within a family of GPUs

How it Began

- In the mid 1990s companies started producing 3D graphics cards for PCs
- Many of the designs were produced by engineers that worked at the big workstation companies
- Their initial designs looked similar to graphics on workstations
- The graphics card were fixed function, there was no way to program them
- Concentration was on mass production of chips that would give performance at a low cost

How it Began

- Started calling these chips GPUs
- GPUs were beginning to have a significant amount of computing power, much more than CPUs
- But with the fixed function pipeline they were inflexible, no ability to program them
- CPU could be programmed, but not the GPU
- To take full advantage of this computing power we needed a way to program GPUs

3D Acceleration

- This first appeared at the pixel level, motivated by the use of multiple textures
- Can apply multiple textures to the same object, each texture provides a different visual effect
- How do we combine these textures?
- Started with just a small set of fixed functions and only 2 textures, but as more textures were added, could have up to four on one object, the different combinations became too large

3D Acceleration

- Could keep adding more parameters and options to the texture functions, but this was getting out of control
- The solution was to use a program to specify how the textures were to be combined
- This was done in a simple assembly language that had about a dozen instruction types, the programs could only be a few instructions long, 4 or 8 instructions
- Initially called texture combiners

3D Acceleration

- This was the start of what is now called fragment shaders or fragment programs
- These are programs that are executed for each pixel drawn by the GPU
- A fragment is a potential pixel, if it passes z-buffer and other tests and gets to the screen it is then called a pixel
- Other lighting and per pixel effects can now be performed in fragment shaders

3D Acceleration

- The original fragment shaders performed only integer arithmetic and had fairly short programs, on the order of 4 to 8 instructions
- They evolved to using floating point arithmetic, much longer programs (at least 512 instructions) and limited forms of branching

3D Acceleration

- Since fragment shaders were a success the idea was extended to geometry to produce vertex shaders or vertex programs
- These are programs that are applied to each vertex in a model
- These programs can perform standard operations, such as transformations, projections and lighting, but they can also be programmed to perform non-standard operations

3D Acceleration

- Vertex shaders have always been floating point and have more sophisticated programming features
- Thus they appeared later in the evolution of programmable graphics hardware
- Vertex shaders were fairly limited in the types of computations they could perform
- They take a vertex as input and produce a vertex as output, it is assumed that the output vertex is in screen coordinates

3D Acceleration

- Evolved to use textures, control structures, and procedures
- The programs can be relatively long, up to 65,000 instructions
- In reality we don't want them very long, since each vertex must go through the program, so a long program will slow down display significantly
- Both fragment and vertex shaders make heavy use of vector processing, most instructions operate on 4 component vectors as well as scalars

3D Acceleration

- The architecture of GPU has changed significantly since fragment and vertex shaders were first introduced
- This lead to the ability to define new types of shaders
- We will examine these changes when we examine some typical graphics chips in detail
- Now we will turn to some more practical concerns

3D Acceleration

- When programmable shaders first appeared they were programmed in assembly language
- The programs were relatively short, so this wasn't much of a problem
- There were two ways of doing this in OpenGL, either a procedure call for each instruction, or the whole program could be stored in a text string and passed as a parameter in a single procedure call

3D Acceleration

- There were a number of problems with this approach:
 - There was little to no feedback on errors, the program just didn't work – the screen was blank
 - The instruction set changed from one graphics card to another resulting in a programming nightmare
 - As programs got longer the whole process became harder to manage, try to figure out a long program that is represented by a sequence of procedure calls in another programming language

3D Acceleration

- If we are programming, why not use a programming language?
- This was the next advance, a number of special purpose programming languages for vertex and fragment shaders, largely based on C
- The three main ones are:
 - Cg – developed by Nvidia, a C based language mainly for their chips – not really used now
 - OpenGL Shading Language – part of the OpenGL 2.0 standard, called GLSL
 - HLSL – this is a Microsoft product that is part of DirectX, this is related to Cg and they were developed together

3D Acceleration

- Cg and GLSL are reasonably close to each other
- They are both based on C and share a large number of concepts
- In many cases it is relatively easy to move programs between the two languages
- One complicating factor is the languages for vertex and fragment shaders were slightly different due to the differences between the processors

3D Acceleration

- While Cg was the first popular high level language for GPU programming, it has largely been displaced by GLSL
- Problem: could only be used on Nvidia GPUs, so programs weren't portable
- Need to have a standard that runs everywhere for wide acceptance, something Nvidia still hasn't learned

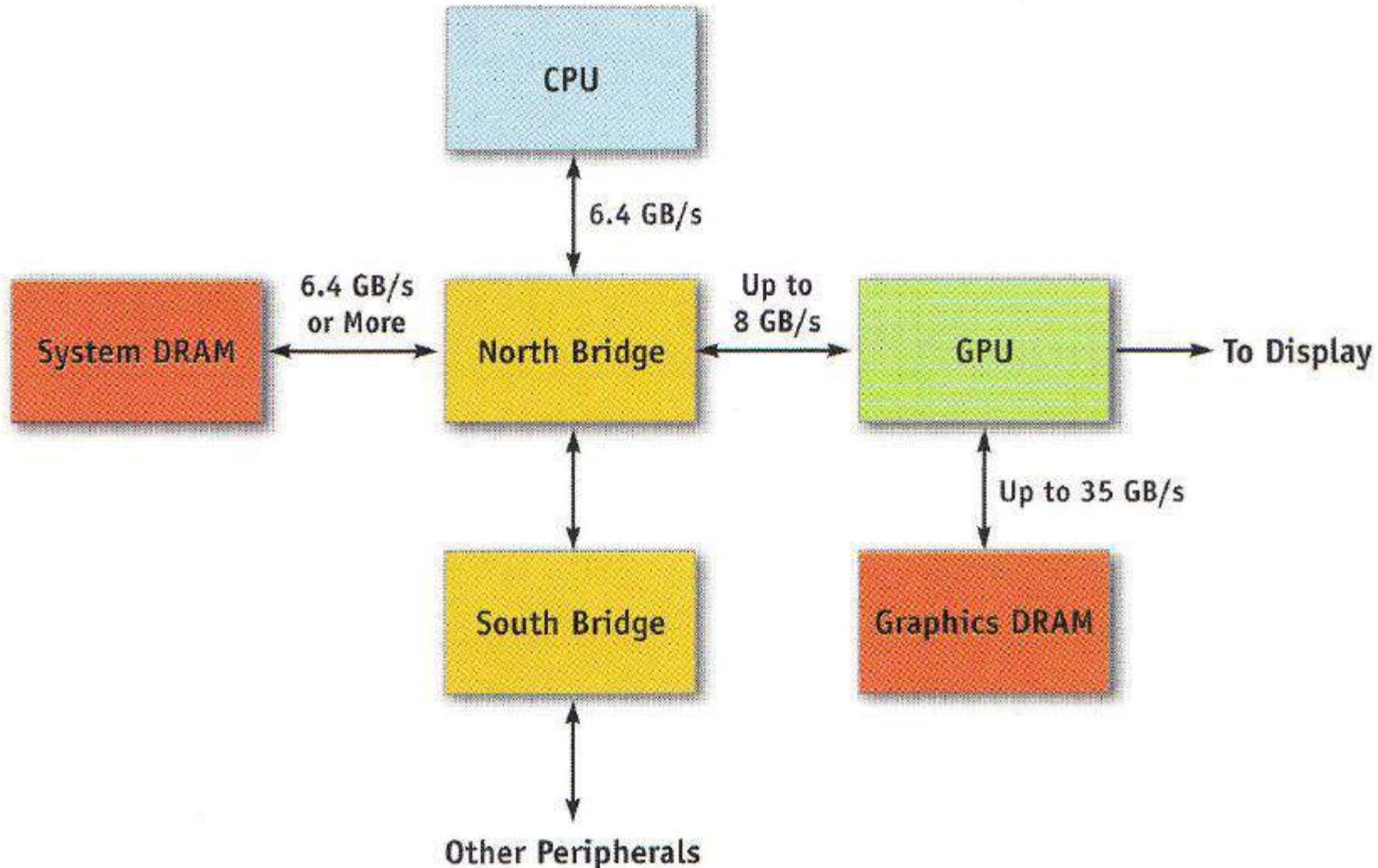
Fixed GPU Architecture

- We now turn to the architecture and programming of modern GPUs
- We will start with the Nvidia GeForce 6 series, this is the height of the old style of GPU architecture
- It illustrates some of the problems that we can get into if we are not careful
- The high level architecture is shown on the next slide along with some indication of the bandwidths involved for the various data paths

Note

- We have lots of information on the older GPUs, this was widely published by the manufacturers
- Largely used or marketing their designs, how advanced they were
- A lot of this information has dried up over the years
- Companies are now trying to protect their advances

Fixed GPU Architecture



Fixed GPU Architecture

- There are basically two modern interfaces to graphics cards, AGP and PCI Express
- AGP has largely disappeared and few new graphics cards use it, mainly industrial applications
- The basic bandwidth of AGP is 256 MB/sec
- The AGP standard also includes AGP 2x, 4x and 8x, but 8x is relatively rare in practice

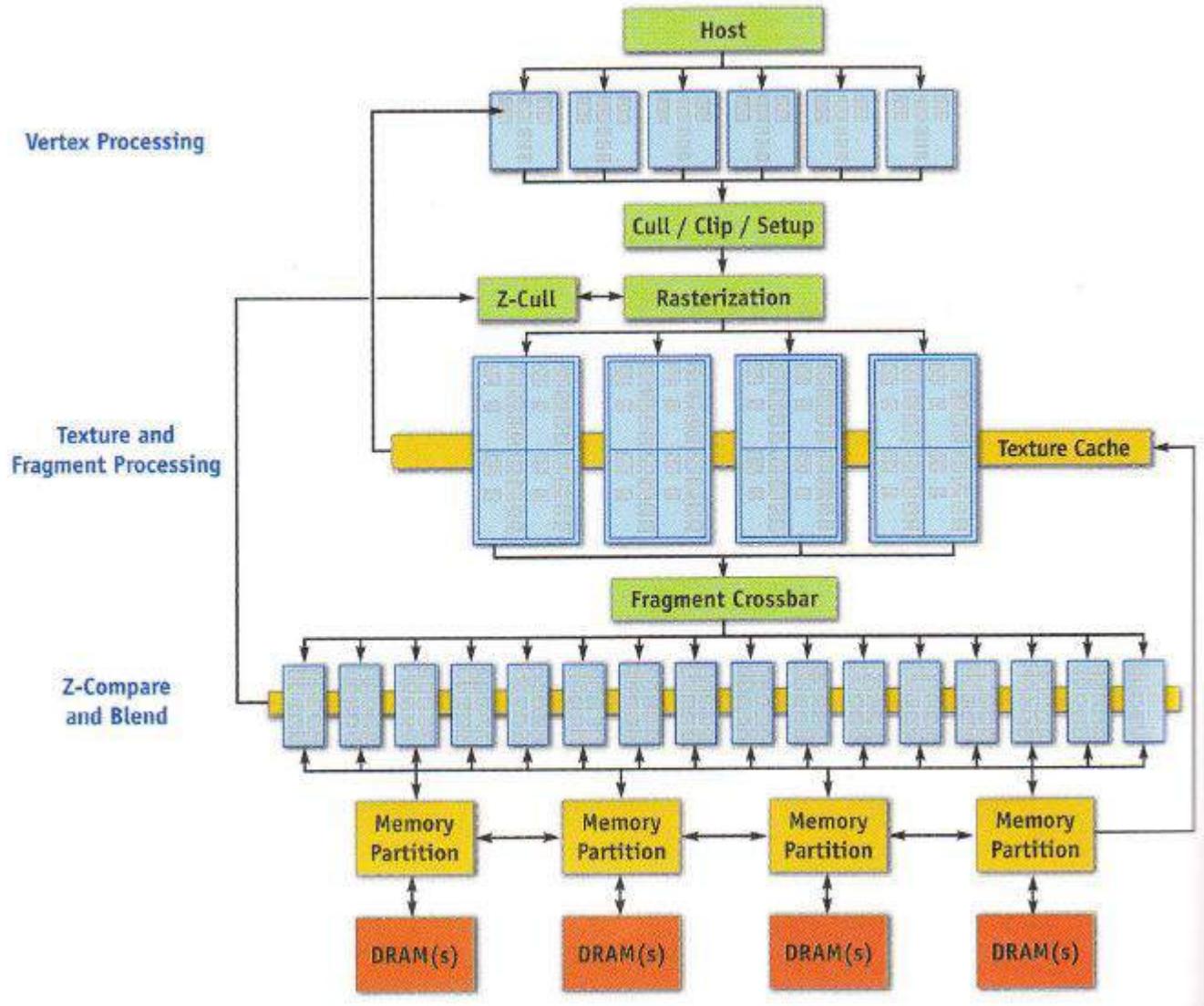
Fixed GPU Architecture

- PCI Express was introduced in 2004 and has now become the standard for graphics cards
- Most graphics cards use PCI Express 16x which has a bandwidth of 8 GB/sec, over 30x faster than base AGP
- It's not unusual to find multiple graphics card slots on motherboards to give increased performance

Fixed GPU Architecture

- Also notice that the bandwidth to memory on a graphics card can be significantly higher than the bandwidth for a CPU
- We are taking advantage of the special memory access patterns that occur in graphics
- The next slide shows the architecture of the GeForce 6 Series GPU
- Before we look at the architecture in detail we need to understand what the CPU is sending to the GPU

Fixed GPU Architecture



Fixed GPU Architecture

- The following information is sent to the GPU:
 - Vertex and fragment programs
 - Geometrical primitives
 - Attributes: material, lighting, matrices, etc
 - Commands
- The graphics card usually deals with a small number of primitives, typically points, lines, triangles and sometimes quads

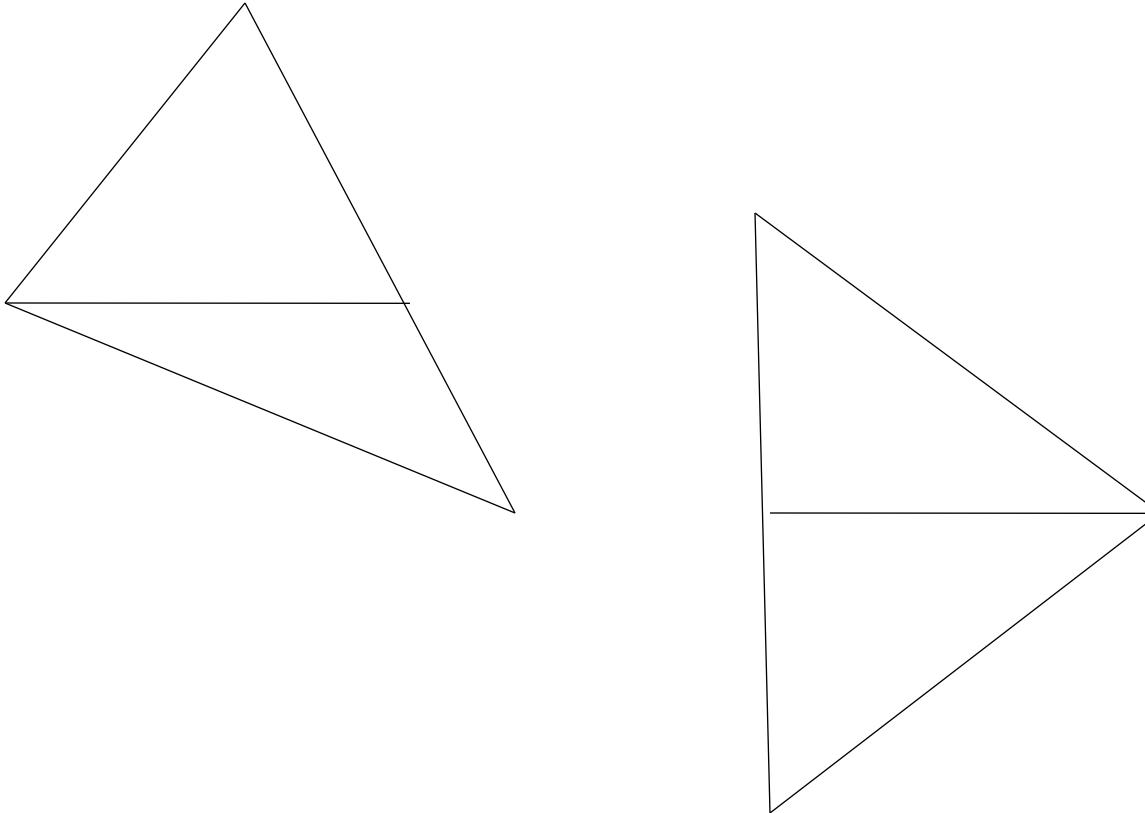
Fixed GPU Architecture

- We have seen that basically any geometrical model can be converted into triangles, so this isn't a problem
- The conversion to triangles is usually done in the driver, so the programmer rarely sees this
- Modern versions of OpenGL force us to use only triangles
- At the GPU the triangle is divided into three vertices, and each vertex is sent through the vertex program separately

Fixed GPU Architecture

- At the output of the vertex processor the vertices are reassembled into triangles and are then converted into screen aligned quads
- In a screen aligned quad the upper and lower edges are scan lines or rows of pixels on the screen
- Any triangle can be decomposed into at most 2 screen aligned quads with the possibility that both of them will have a degenerate edge

Fixed GPU Architecture



Fixed GPU Architecture

- Why do we want to do this?
- This makes scan conversion easier, determining the pixels that are covered by the triangle
- At each vertex we have coordinates, plus colours and other attributes
- The attributes, along with the z coordinate must be interpolated across the surface of the polygon

Fixed GPU Architecture

- It is easier to compute the x and y coordinates if we move along one row or column of the image
- Example: for each row the y value is constant, and x is incremented by 1
- For the other attributes we need to interpolate along the scan line, which means we need to have the values at the start and end of the scan line
- This is easy to do by interpolating between the vertices

Fixed GPU Architecture

- Some graphics chips allow quads in addition to triangles, they can be decomposed into at most 3 screen aligned quads, so this is easy to accommodate
- Now we can follow through the architecture
- The vertices are first sent through the vertex processors
- There were several of these processors, the number of processors determines the performance and cost of the graphics card

Fixed GPU Architecture

- The processing of each vertex is done independently so we can process many of them in parallel
- After processing they are reassembled into primitives so they can be clipped
- We may also perform culling at this step, remove the primitives that aren't visible or hidden by other primitives
- They are then rasterized using the process described earlier

Fixed GPU Architecture

- The individual fragments are then processed by the fragment processors
- Again this is done in parallel, since each of the pixels can be treated independently
- Finally we do the final z compare for hidden surface removal and blend the resulting pixels into the frame buffer
- Typically the memory is divided into multiple partitions to increase the memory bandwidth

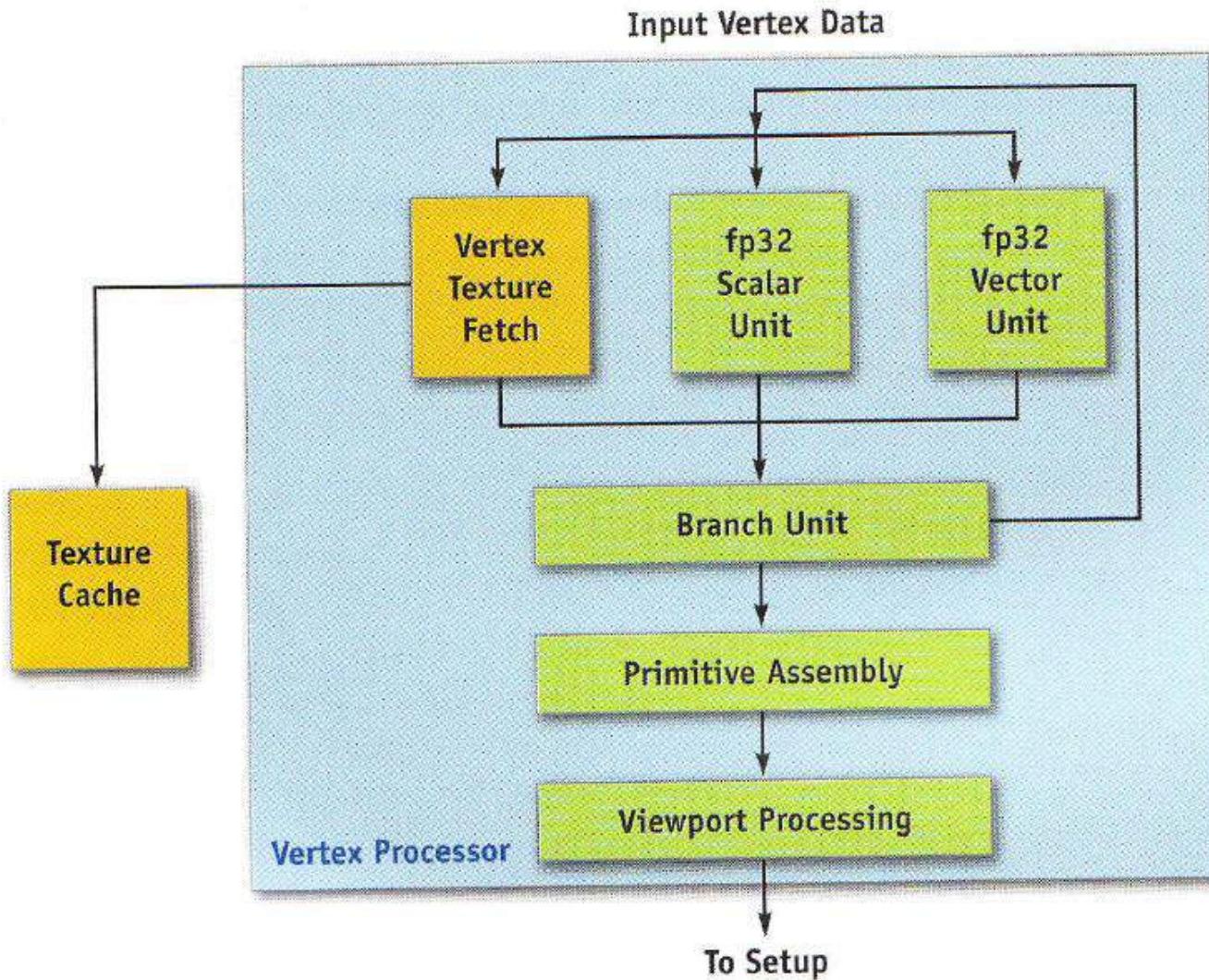
Fixed GPU Architecture

- The resulting pixels may be written on the screen, but with modern graphics hardware they can be written into any location within memory
- These are called render targets and the most common one is textures, but there are other types of off screen memories
- Some graphics cards can write to multiple render targets, this is called MRT

Fixed GPU Architecture

- The architecture of a vertex processor is shown on the next slide
- The processor can do a texture fetch, a scalar operation and a vector operation in parallel
- Since a texture fetch involves a memory operation it takes longer than the other two, so we attempt to issue several arithmetic operations for each texture fetch

Fixed GPU Architecture



Fixed GPU Architecture

- The vertex processor typically does several vector-matrix multiplies per vertex so its easy to keep the arithmetic units busy
- The instruction set is optimized for this type of processing with an emphasis on vector operations
- For example there are instructions for dot and cross products, and they execute in a single instruction time

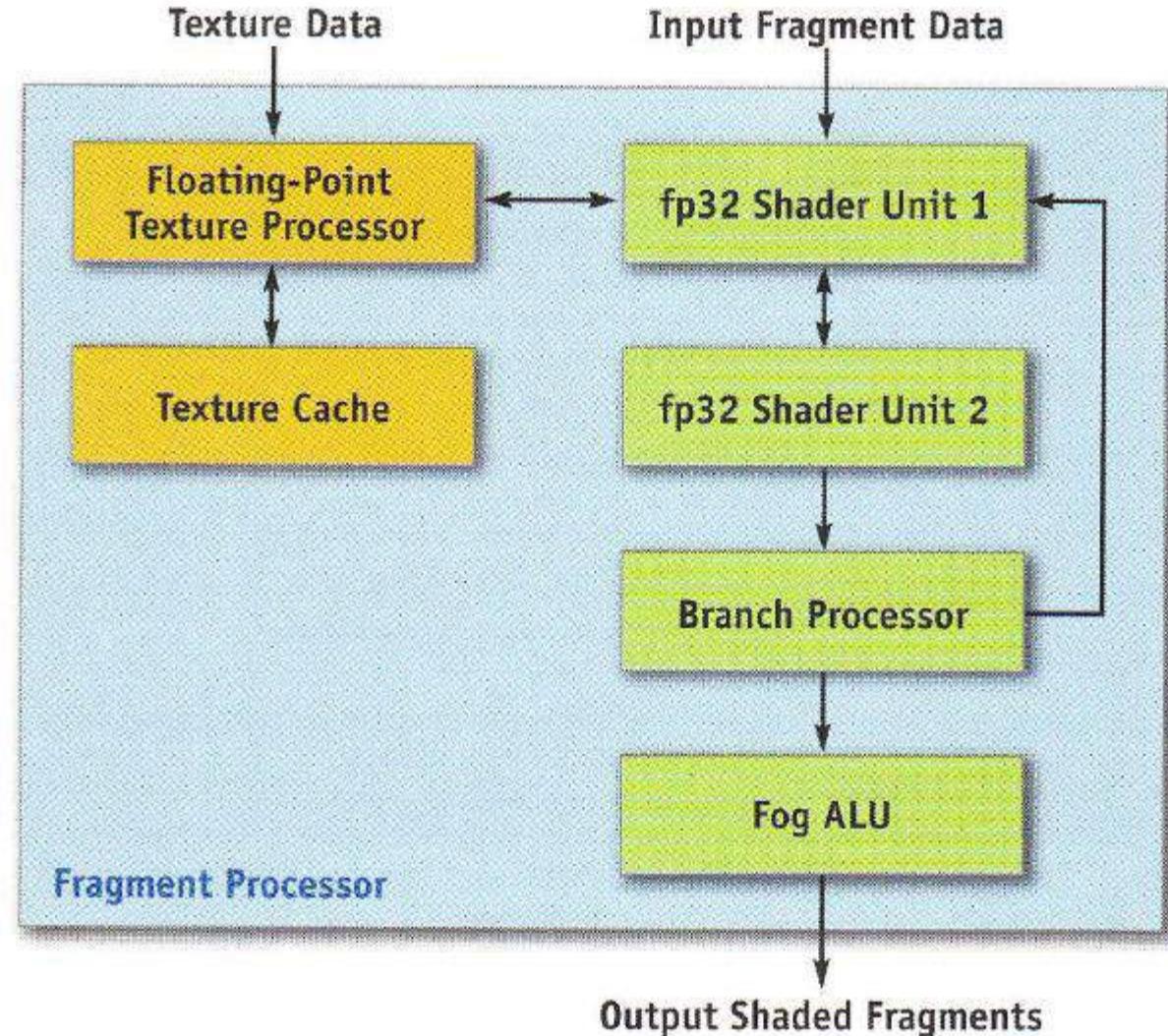
Fixed GPU Architecture

- There are also instructions for most of the trig functions and other operations typically found in graphics
- In this architecture the vertex processors are MIMD processors, with each processor working independently
- There is no penalty for branches and subroutine calls within the code
- In this architecture there were between 2 and 10 vertex processors

Fixed GPU Architecture

- Most vertex processors have a vertex cache where they store the most recent vertices they have processed
- This optimizes the processing of certain types of meshes where a vertex appears in several neighbouring triangles
- The architecture of the fragment processor is shown in the next slide

Fixed GPU Architecture



Fixed GPU Architecture

- Each triangle has three vertices but it can cover many pixels, therefore the fragment processors must have greater throughput
- There are two main ways of achieving this:
 - Use shorter programs, we have limited control over this
 - Use more processors
- The second approach is usually taken, not unusual to have hundreds of fragment processors

Fixed GPU Architecture

- If there are so many processors they must be simpler, which we can see in their architecture
- There are two processors that allow us to perform two operations in parallel
- These are typically shading and texture operations, with one texture fetch per shader unit
- The texture processor can perform the standard texture filtering operations

Fixed GPU Architecture

- Fragment processors can branch, but there are restrictions on this
- The fragment processors are usually SIMD processors, so if one processor branches they must all branch
- This is handled by having all processors follow both branches, but just saving the results for their branch
- This limits the amount of branching that can be performed

Fixed GPU Architecture

- With this approach a branch doesn't save time, it increases the processing time since each processor needs to do more work
- There are some techniques for reducing branching
- Such as using conditional expressions instead of branches and unrolling for loops

Unified Shader Architecture

- There is a major problem with the NVIDIA 6000/7000 series architecture and similar chips
- Both the vertex and fragment processors are dedicated processors that have a fixed place in the pipeline
- There is a fixed number of each of these processors, with fragment processors out numbering vertex processors
- This leads to bottleneck problems

Unified Shader Architecture

- If the vertex programs are long, it will take a considerable amount of time to process each vertex
- While this is happening the fragment processors will be idle waiting for input
- Similarly, if the fragment programs are long and require a lot of processing the vertex processors will need to wait, since there will be no free fragment processors to accept their output

Unified Shader Architecture

- This results in idle processors, possibly for a significant amount of the time
- The ratio between vertex and fragment processors is determined by examining a wide range of applications and taking the average
- Unfortunately, this average may not fit any of the real applications, since the average application may not exist

Unified Shader Architecture

- To improve performance we need to make better use of the vertex and fragment processors
- Originally these processors were quite different
- Vertex processors were floating point, fragment processors were integer
- Only fragment processors used textures
- Vertex processors were MIMD and fragment processors were SIMD

Unified Shader Architecture

- Over time these differences started to disappear
- With floating point colour, floating point was added to fragment processors
- Vertex processors can now access texture maps
- Fragment programs are becoming more complex, so they would benefit from a MIMD architecture, etc

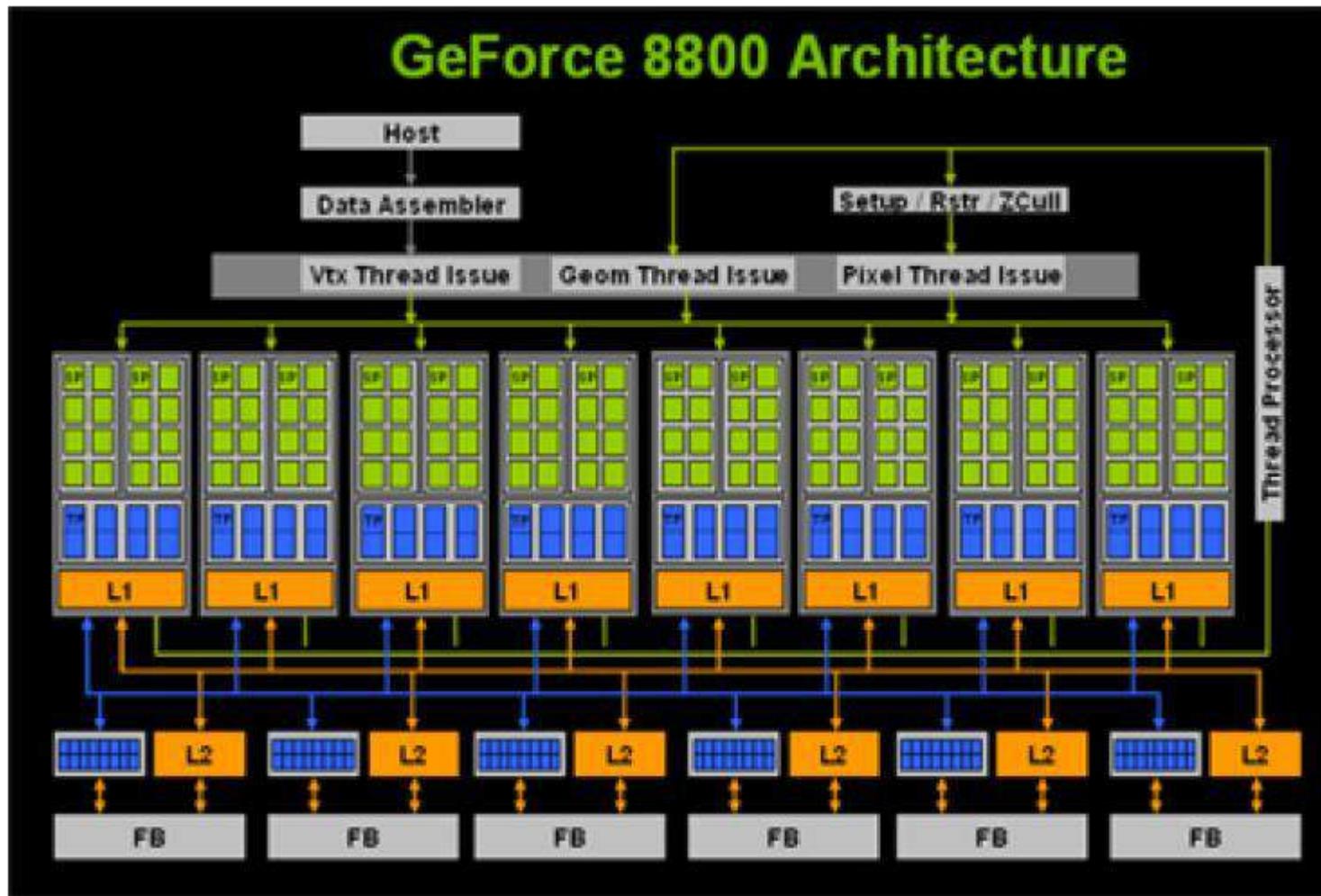
Unified Shader Architecture

- Why not combine the two types of processors, so they can be switched between vertex and fragment programs depending upon application demand?
- This would make better use of hardware resources and increase performance
- This was the motivation behind the NVIDIA 8800 GPU
- This GPU uses a unified shader architecture where a shader program can be run on any processor

Unified Shader Architecture

- The architecture of the NVIDIA 8800 is shown on the next slide
- The processor is divided into 8 Texture Processing Clusters (TPC) which are in the middle of the diagram
- Each TPC is further divided into two Streaming Multiprocessors (SM), which each have 8 streaming processors (SP)
- This gives a total of 128 streaming processors on each GPU

Unified Shader Architecture



Unified Shader Architecture

- Each streaming processor can be executing multiple threads, which are supported at the hardware level
- These threads can be running a geometry, vertex or fragment shaders
- The number of threads devoted to each type of program depends upon the current work load and can be changed dynamically while the GPU is running

Unified Shader Architecture

- When the GPU receives input from the CPU it is first allocated to a vertex thread that will execute its vertex program
- This allocation is handled by the thread scheduler at the top of the diagram
- The output from the vertex program is fed back to the thread scheduler to be sent to either a geometry or fragment program
- Data can be pumped through this loop several times before it eventually gets sent to the frame buffer

Unified Shader Architecture

- Each TPC has 4 texture units, each with a texture address unit and two texture filtering units
- These are shared by the streaming processors in the TPC
- If a SP has to wait for a texture operation it switches to another thread, so it is not idle while the texture fetch is occurring
- All the SP in a TPC also share an L1 cache

Unified Shader Architecture

- The L2 caches are shared between the TPCs
- Unlike the previous generations of GPUs, the processors have no vector instructions, all of the instructions are scalar
- For vector processing several SPs can be grouped together in an SIMD arrangement
- This optimizes both scalar and vector processing

Unified Shader Architecture

- The allocation of chip area on a GPU is very different from that of a CPU
- On a CPU a large portion of the area is allocated to cache memory, only a small portion is allocated to computing
- The reverse is the case on a GPU, most of the area is devoted to computing, while there is very little cache
- This difference reflects the different nature of the computations

Unified Shader Architecture

- Graphics computations are pipelined, the output of one stage is the input to the next stage
- The only results that are stored in memory are the final results that go into the frame buffer
- The only memory reference is texture maps, so a GPU performs many more computations than it does memory references
- There are no complicated data structures here

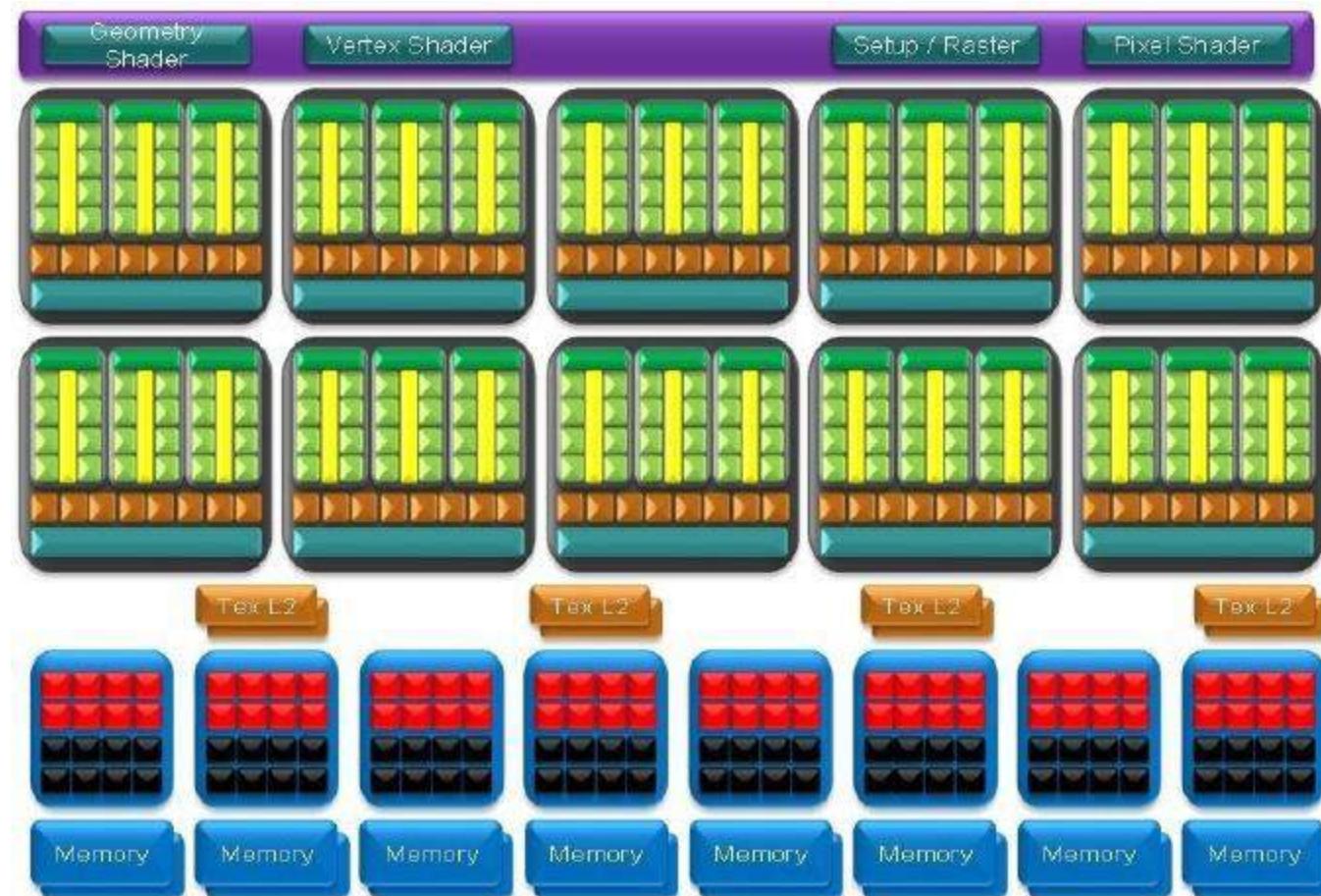
Unified Shader Architecture

- On the other hand CPUs handle general computations, which can involve a large number of memory references and complicated data structures
- This makes large caches more important in a CPU, but at the cost of computing performance
- Thus, many compute intensive applications run faster on a GPU than a CPU, which is becoming a trend in high performance computing

The Next Step

- The next iteration on this design is the GTX 280 architecture, which is shown on the next slide
- This architecture adds more processors with a total processor count of 240
- This is organized as 10 TPCs, three SMs per TPC and 8 SPs per SM (the same as the previous architecture)

The Next Step



The Next Step

- The individual streaming processors are essentially the same as the NVIDIA 8800 with some small improvements
- One of the main differences is in the organization of the TPC, which is shown on the next slide
- The SMs operate as MIMD machines, but the processors inside an SM operate more like an SIMD machine

The Next Step



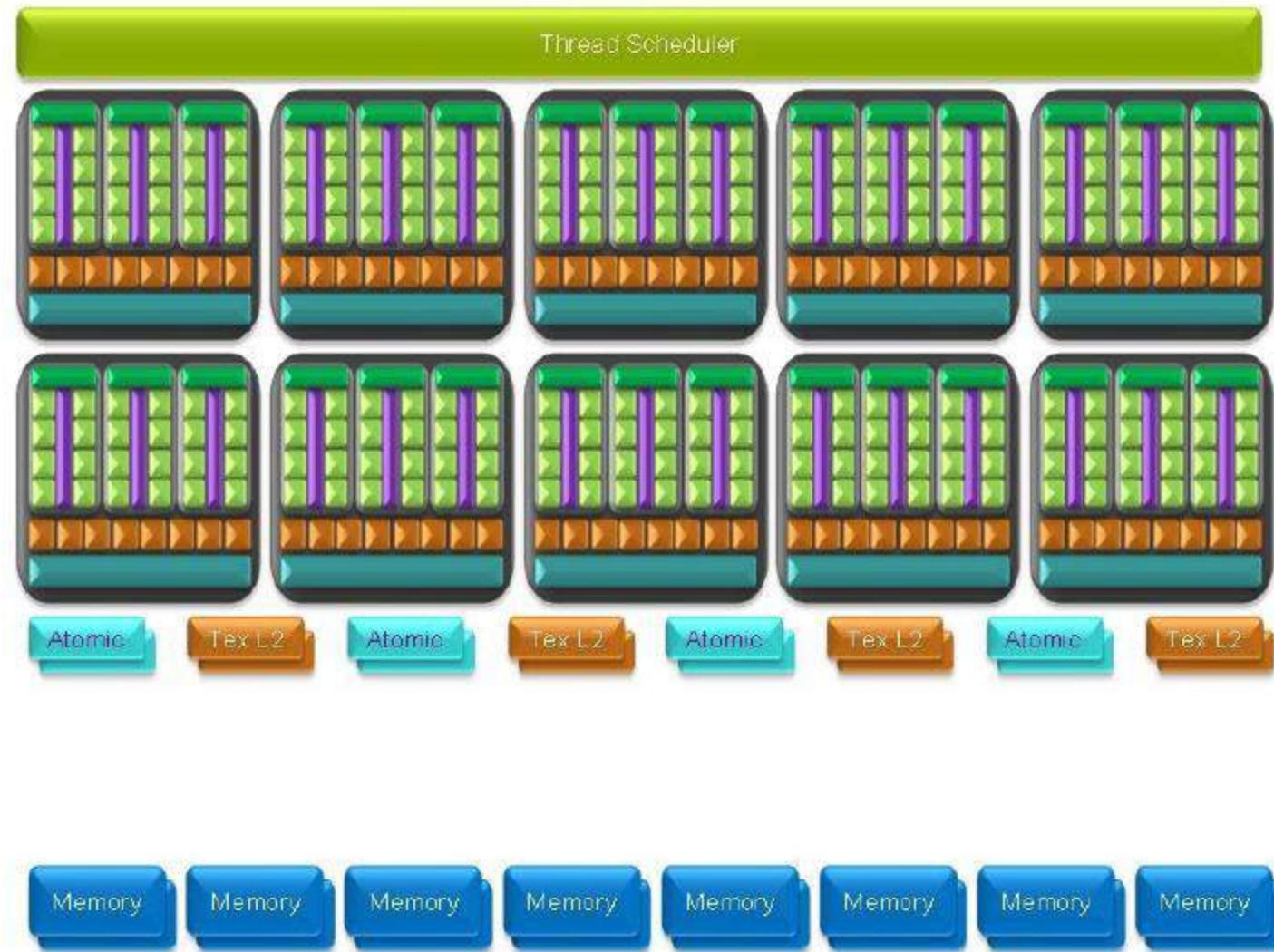
The Next Step

- There are more texture units in an SM, giving better texture performance
- A greater number of hardware threads are also supported
- From a graphics point of view, this is a fine tuning of the architecture to deliver higher performance
- One of the main differences is the GPU can now operate in compute mode, which doesn't perform graphics

The Next Step

- The compute mode architecture is shown on the next slide
- In this mode the emphasis is on accelerating scientific and engineering computations
- In this mode there is only one type of thread, and thus one type of program
- This optimizes performance on large arrays of data

The Next Step



Current GPUs

- The following slide shows the architecture of the NVidia Pascal GP100(recent high end GPU)
- The bulk of this chip are the computational cores, 3840 of them
- How do we get this many cores on a chip, a CPU chip has at most 10?
- GPU cores are quite different, they are a SIMD architecture (Single Instruction Multiple Data)
- There is one instruction decode unit for a group of 32 cores, all the cores perform the same instruction in lock step

Pascal GP100 Architecture



Current GPUs

- Instruction decode on a CPU is very complicated, larger than the instruction execution part
- In a GPU this is simplified considerably, so instruction decode takes up much less room
- The emphasis is on the computational units
- The next slide show the GP100 SIMD unit, which is called a streaming multiprocessor, or SM
- The instruction decode is at the top of the chip, a relatively small area
- Most of the chip area is devoted to the computational elements

SIMD Processor



Current GPUs

- The SM units in the GP100 have fewer cores than previous architectures, at least half the number
- But they have the same amount of memory, this gives each core more memory
- This translates into more local memory for our shader programs
- The GP100 organizes the SMs into 6 Graphics Processing Clusters, or GPCs, this is a new level of organization for NVidia chips

NVidia GP100

- The GP100 introduces a new memory organization where the memory chips (actually the dies) are stack one on top of the other
- The dies in a stack are linked together by up to 5000 wires running between them
- Current system have 4 dies in each stack significantly increasing the memory bandwidth
- Going from 288 GB/s on previous architectures to 732 GB/s on the GP100

One More Crank

- Turing architecture was introduced almost two years ago, a significant evolution
- High level architecture of chip is on the next slide, looks somewhat similar to earlier GPUs
- The RTX 2080Ti has 6 GPCs, the RTX 2080 has 4 GPCs and the other models have 3 GPCs
- It is basically the same architecture it is just scaled for greater performance

Turing Architecture



Turing Architecture

- A Turing SM is shown in the next slide, there are several differences with previous designs:
 - The floating point and integer units can work in parallel
 - The tensor cores are used for machine learning, they are tuned to large matrix operations
 - The RT core is used for ray tracing, it can do ray-triangle intersections and bounding volume hierarchies
 - A program on an SM can spawn other programs, like starting a new process

Turing SM



Turing Architecture

- Putting different types of computations on the GPU, more specialized units
- Also the ability to launch programs
- Beginning to look like a CPU, not quite there yet
- When will we have a GPU that has its own CPU?
- This is the logical evolution and will turn everything upside-down, soon the GPU will need its own GPU

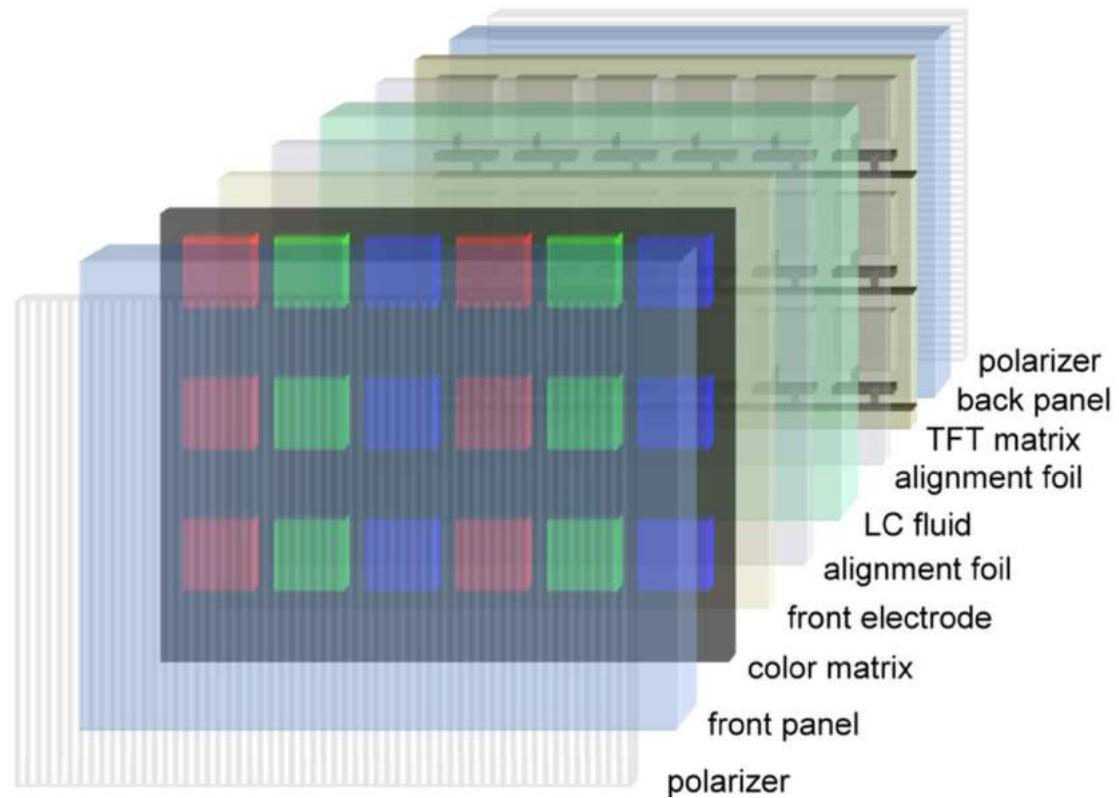
Displays

- This is where all the fun is, largely ignored by graphics researchers until about 8 years ago
- Used to view monitors and projectors as black boxes, not very interesting, just used to display the results of our work
- Now we know that there are cool things inside of these boxes
- In particular Spatial Light Modulators (SLM)
- A SLM dynamically changes (modulates) the optical properties (light) of individual pixels (spatial)

Displays

- There are many types of SLMs, we will look at only a small number, then how they can be used
- One of the most common is the LCD (liquid crystal)
- The structure of the a typical LCD panel is shown on the next slide
- LCDs work by changing the polarization of light, so we have polarizers on both sides of the panel
- The polarizers are thin sheets that pass light of only one polarity

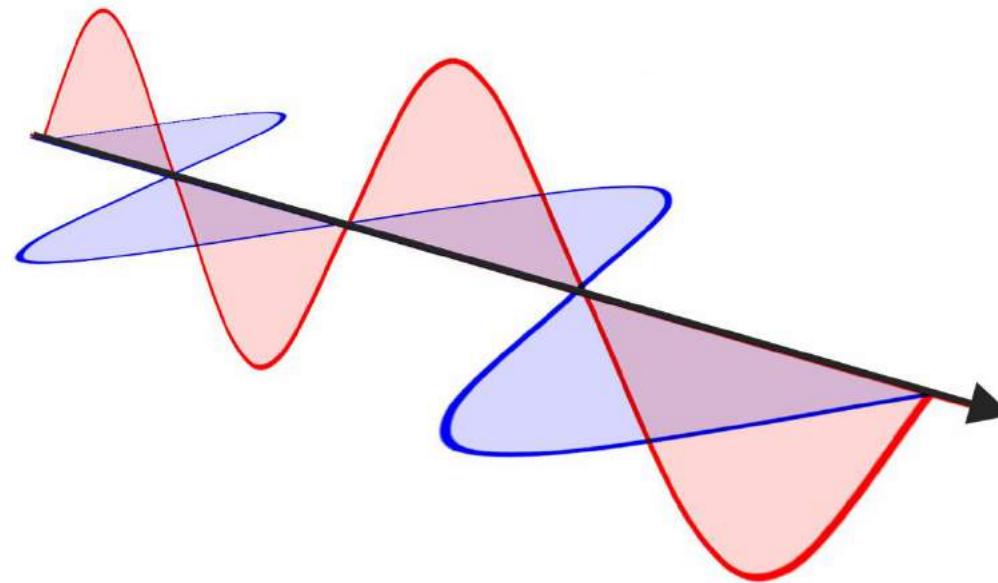
LCD



LCD

- What's polarity?
- Light can be viewed as a wave (it can also be viewed as a particle), so think of a sine wave traveling through 3D space
- The wave travels in a straight line, but the vibrations could be about any angle around this line
- The next slide shows two possible vibration angles
- In polarized light all the waves have the same angle
- If we only have the blue wave in the following slide we have polarized light

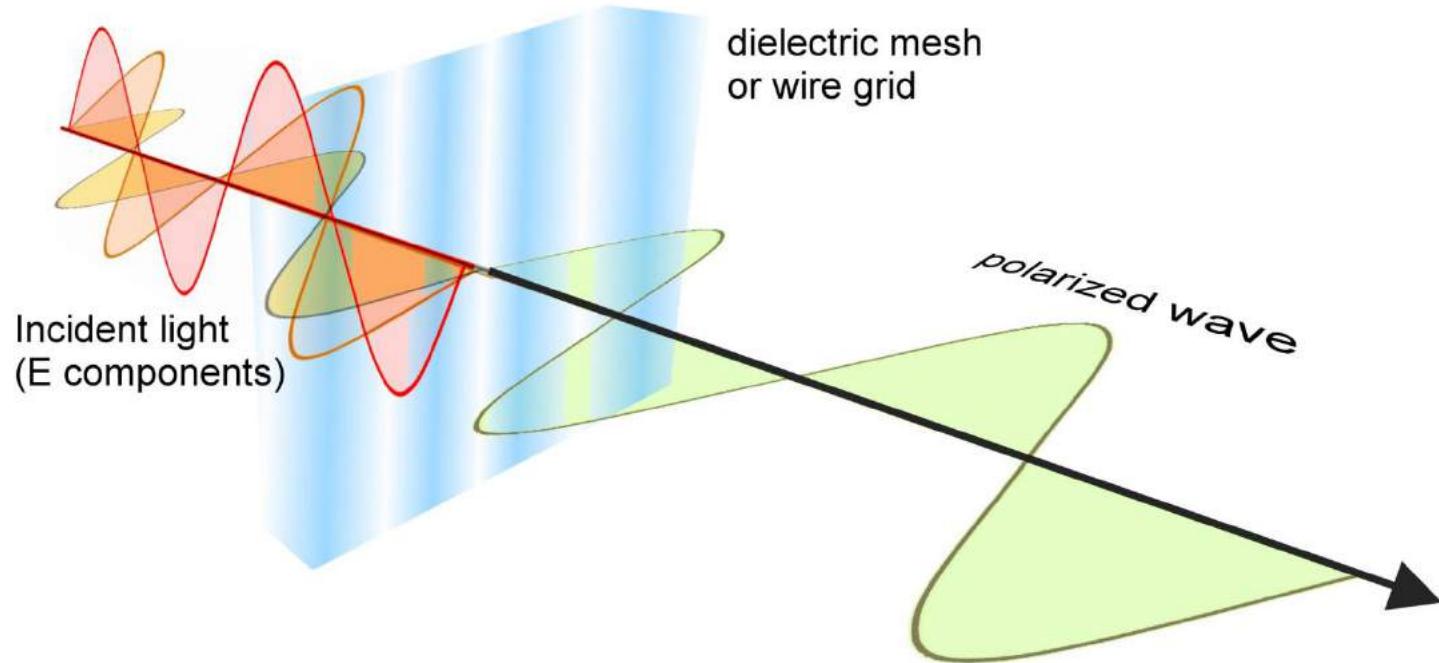
LCD



LCD

- This is linear polarization, which is the simplest kind
- A polarizer is used to select light with a particular polarization, see the next slide
- We can also have optical elements that shift the polarization
- Liquid crystals are long molecules that can be aligned by applying a voltage to them, when this occurs their optical properties change
- They are arranged in small cells, with three cells for each colour pixel

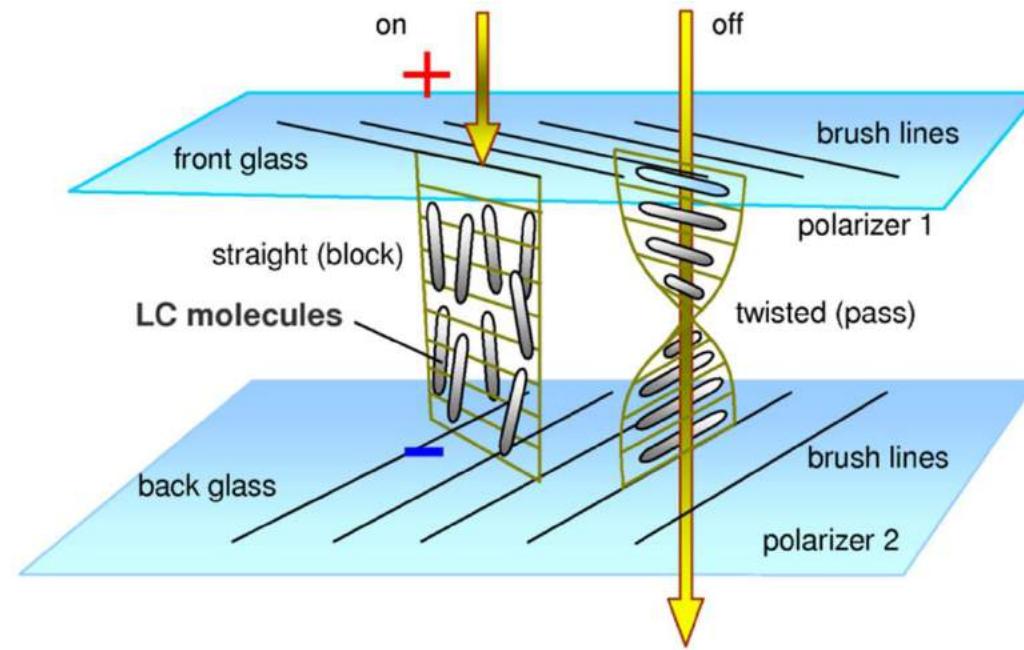
LCD



LCD

- The behavior of an individual LCD cell is shown in the next slide
- On the left a positive voltage is applied to the top layer and light is blocked
- On the right light passes through the cell, but its polarization is changed
- In this mode the LCD is basically transparent
- In between we get various levels of transparency, which can be achieved in several ways

LCD



LCD

- So far we can't see anything
- An LCD display has a backlight that provides the light required to see what is on the LCD
- Originally the backlights were fluorescent, but they are now mainly LED
- The light should be as even as possible to avoid intensity variations across the display
- The fun starts when we use more than one LCD panel
- Each panel can do 256 levels of red, green and blue

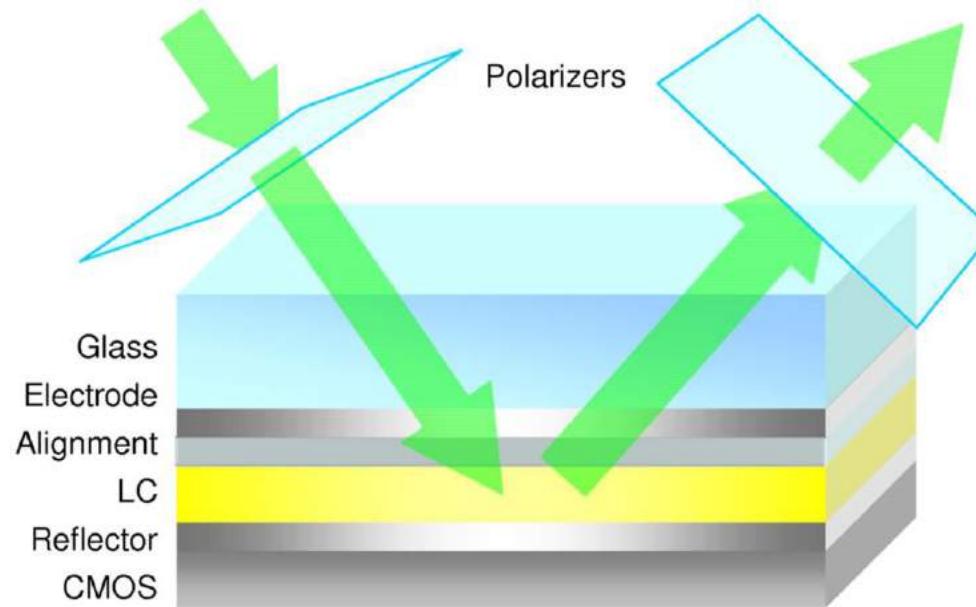
LCD

- If we stack two panels their images are multiplied
- This gives us $256 \times 256 = 65,536$ levels for each of red, green and blue
- This is called a High Dynamic Range (HDR) display
- The bottom panel becomes a pixel accurate variable backlight
- An LCD is called a transmissive SLM, since it modifies the light that passes through it

Reflective Displays

- LCDs change the light that is transmitted through them, we can also change the light that is reflected off an SLM
- Look at two common technologies
- LCoS (liquid crystal on silicon) is based on applying a coating of liquid crystal on top of a CMOS chip
- A reflective surface is placed on top of the CMOS layer and below the liquid crystal
- Charges applied to the liquid crystal are used to change its polarizing properties, and thus whether light is reflected

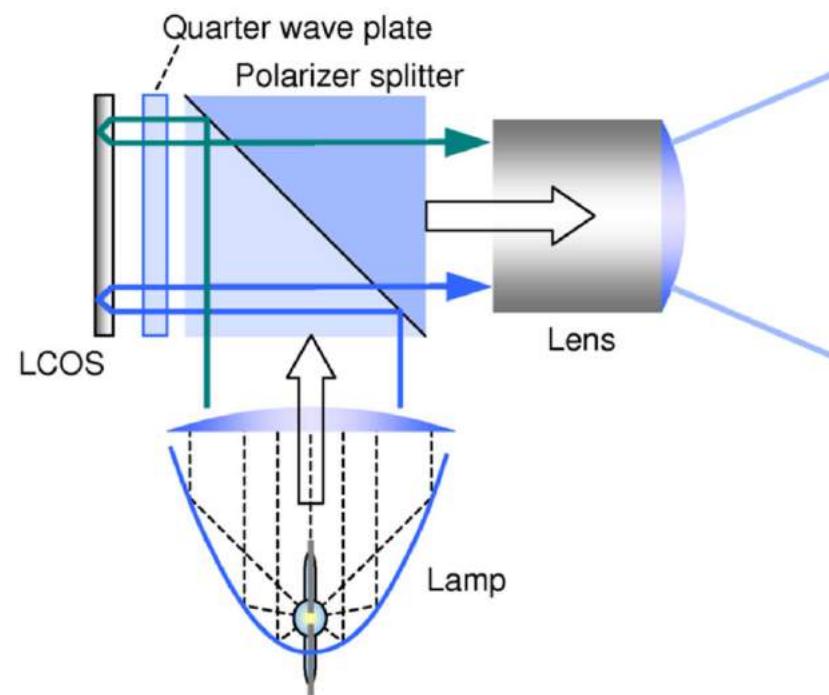
LCoS



LCoS

- This removes the need for a backlight
- It can also have smaller pixels, which can be used for higher resolution or smaller displays
- F-LCoS (Ferroelectric LCoS) can produce pixels smaller than $8 \mu\text{m}$, these displays are used in head-mounted displays
- Can have full HD in a 1" display
- LCoS is also used in projectors as shown on the next slide

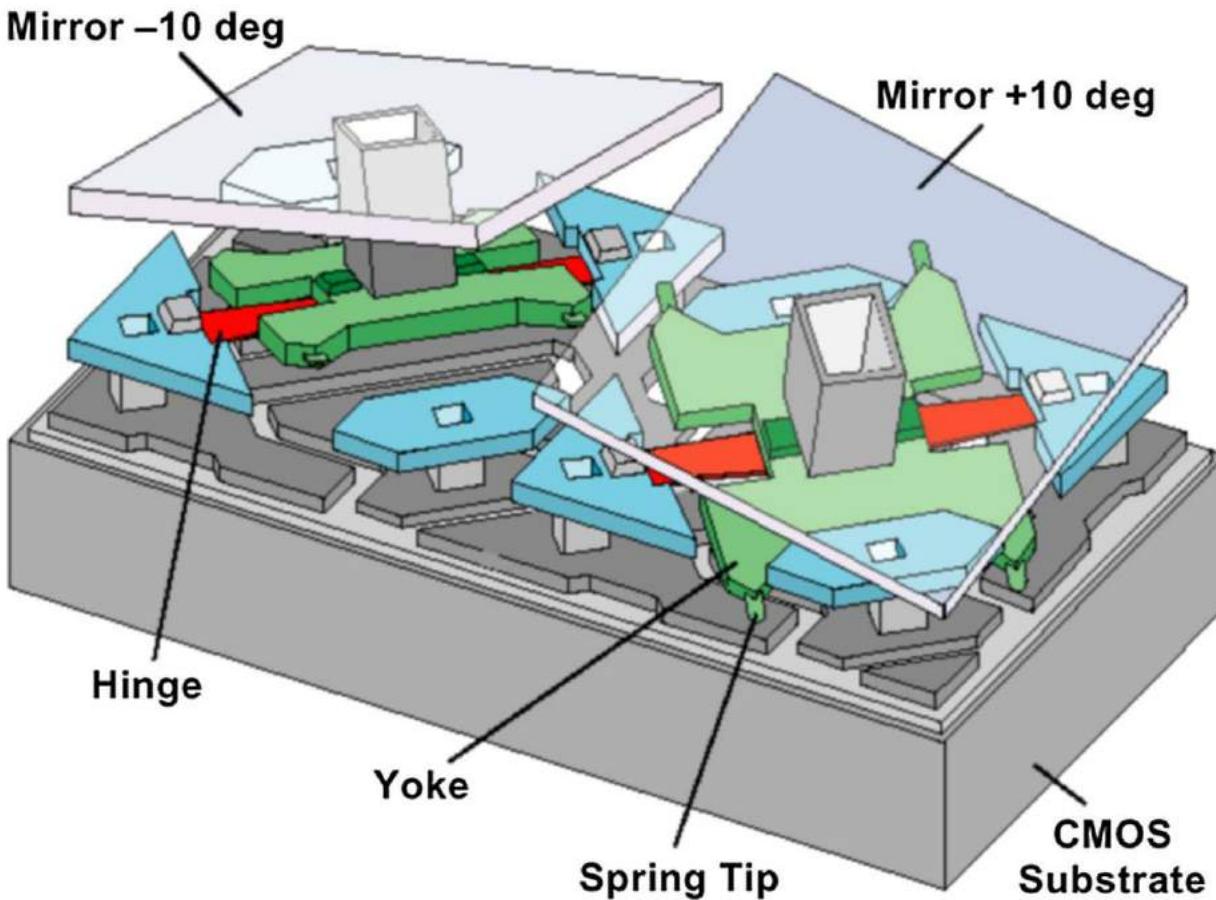
LCoS



DMD

- Digital micromirror devices (DMD) are based on fabricating very small mirrors on a silicon chip
- These mirrors can move using silicon hinges and springs, see the next slide
- In one configuration the mirror reflects light towards the viewer and in the other the light is reflected away from the viewer, this gives basically a binary signal
- We can get grey levels by controlling the amount of time the mirror points towards the viewer
- If it is only half a pixel time we can get half intensity

DMD



DMD

- DMDs switch very quickly, much faster than LCDs
- They are also very efficient, since there is very little light loss in the reflection, can be used for very bright projectors
- The main use of DMDs is in projectors
- At the low end a single DMD can produce colour by sequentially displaying red, green and blue images
- Even low end DMDs operate at over 200Hz, so this works okay

DMD

- High end projectors use three DMDs and then optical elements to combine the three images
- This is the technology that is now used in movie theatres
- They are sometimes called DLP projectors, it's the same technology
- Texas Instruments developed DMD and until recently this was patented, which kept the cost high

Summary

- In this section we have investigated graphics hardware, looking at how graphics devices have evolved over the past few decades
- The architecture of modern GPUs has been presented along with the techniques that have been developed to program them
- Examined the three main display technologies:
 - LCD
 - LCoS
 - DMD