



COMP 371

Computer Graphics

Session 1

Course Overview - Syllabus

Lecture Overview

- Administrative Issues
- Modeling
- Animation
- Rendering
- About OpenGL Programming

Instructors

Name: **Kaustubha Mendhurwar**

Office: **ER 1101**

Email: kaustubha.mendhurwar@concordia.ca

Research Interests:

3D Graphics/Animation

Virtual/Augmented Reality

Course Website

- This course uses Moodle for hosting course content, announcements, lecture slides, notices, assignments, course related submissions by students, schedule changes, scheduling of demos, etc.
- Please visit this site regularly.
- The lecture notes will be posted on the course website in Moodle the day of (or the day after) the lecture.

Course Prerequisites

Courses

- COMP 232 or COEN 231
- COMP 352 or COEN 352

Very good programming skills in C/C++

Basic Data Structures

- COMP 232 or COEN 231
- COMP 352 or COEN 352

Geometry

Simple Linear Algebra

Course Components

□ Programming

- 2 Timed take home OpenGL based quizzes 20% each
- 1 team project split into three submissions carrying respectively 10%, 20% & 30% weight
- Lab participation/performance (No weight but highly recommended if you want to do well in this course)

□ Assignment Grading

- Specifications - functionality and features
- Style and documentation
- Aesthetics → Quality
- Appealing extras → bonus marks

Computer Graphics

Study of Computer Graphics

- Capturing (imaging) the world of interest (real or imaginary) in a pixel, such that it
- appears realistic (when the world is real),
- appears as intended by the artist (when the world is imaginary)



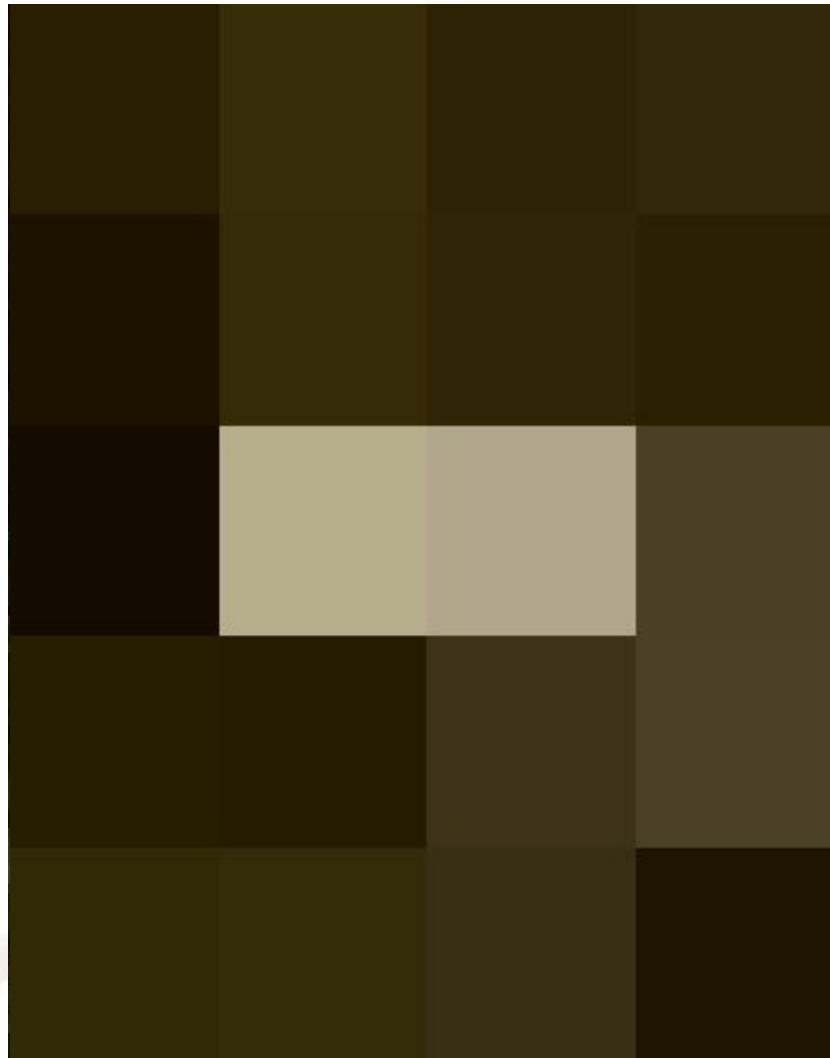
Acknowledgement:
Ray traced image using POVRAY

What is a Pixel?

Usually, a rectangular area on a 2D display surface with two properties, one fixed and the other under computer control :

- **area of the pixel is fixed**
- **color of the pixel is set under program control**

What is a Pixel?



Computer Graphics

Important Attributes of the World (real or Imaginary)?

Two very important and rather complex attributes:

- **visual look or appearance due to light reflection, refraction, occlusion, material, world lighting, ...**
- **dynamic behavior due to interaction with other elements of the world -- movement, collision, elastic effects, ...**

Computer Graphics

Three Fundamental Tasks

- **Modelling the world,**
- **Simulating behavior of objects in the world,**
- **Presenting (rendering) the world.**

For interactive applications

- **Interacting with virtual object.**
- **Geometry and Physics are traditional tools**

Computer Graphics

Different Digital Representations of the World

- **Digital Images**
- **Segmented Regions**
- **3D Geometric Objects**
- **Symbolic Descriptions**



Different branches of Visual Computing depend on transforming the data amongst these representations

Computer Graphics

What is Computer Graphics?

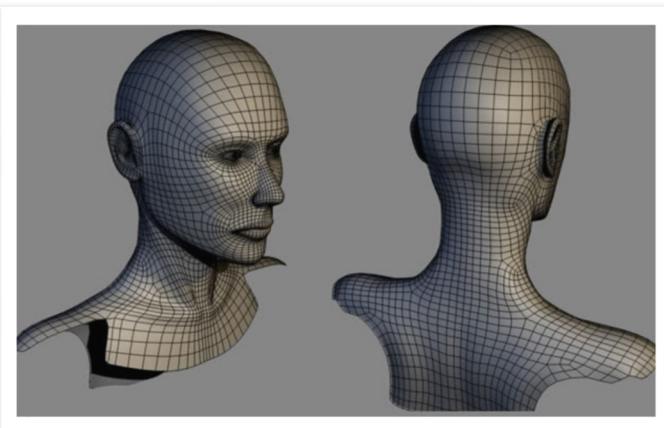
- **Computer graphics** is thus concerned with all aspects of producing images using a computer:
- **Modeling** - how to represent shape of objects
- **Animation** - how to represent and control motion
- **Rendering** - how to create images of objects
- **Image Processing** - how to manipulate & edit images

This course will cover the above theoretical aspects and provide practical training using OpenGL graphics library

Computer Graphics

Modeling

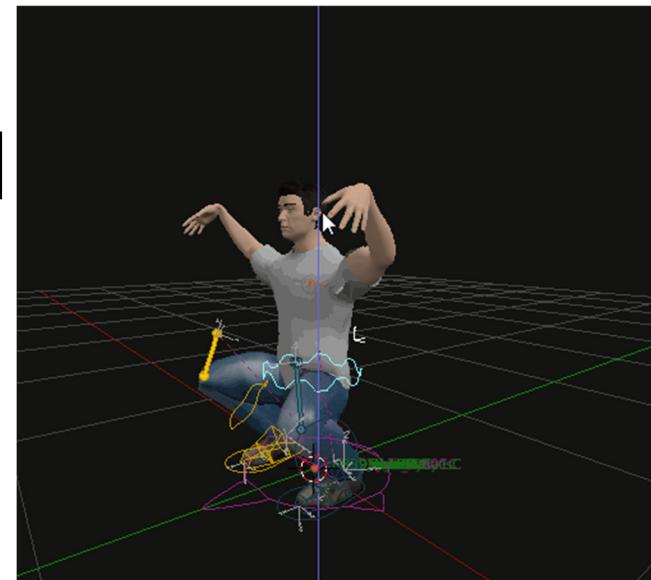
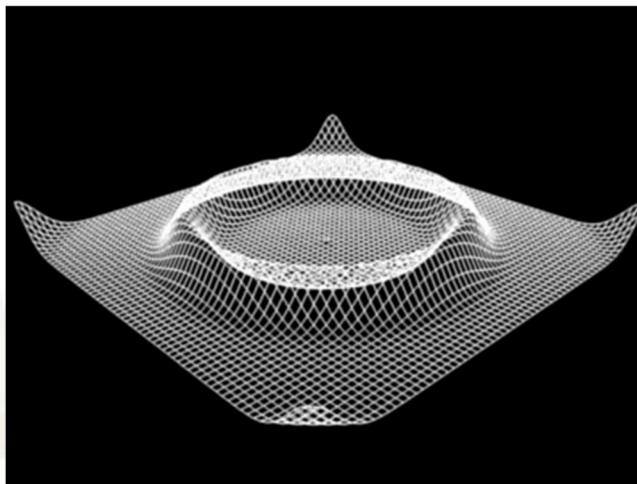
- How to represent objects
- Turn the real into a virtual representation
- Describe (approximate) the real world or fictional objects using mathematics
- If the image does not exist in real life, a blueprint is drawn by an artist



Computer Graphics

Animation

- **Control and represent motion of the objects**
 - **keyframe - inverse kinematics**
 - **performance-based**
 - **procedural**
 - **physics based [simulation]**



Computer Graphics

Rendering

- How to create images of objects

- Color
- Lighting
- Shading
- Surface detail (texture)
- Reflection, transparency, shadows



Wire-frame

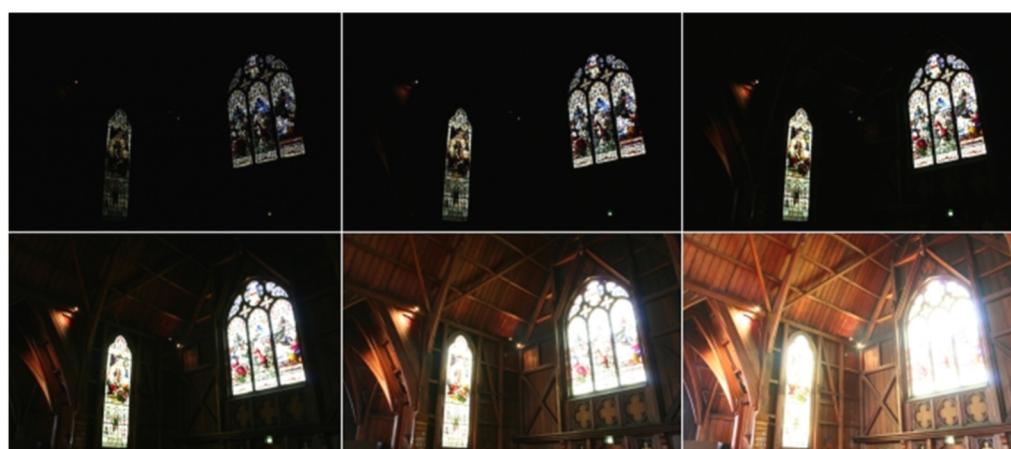
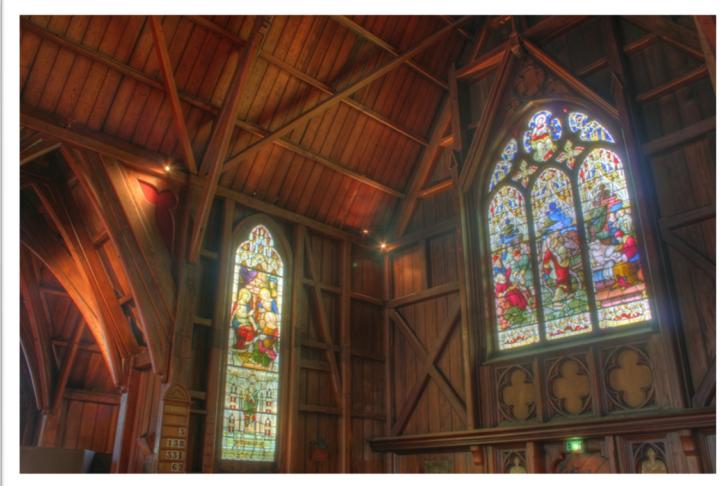


Final Render

Computer Graphics

Image Processing

- Various Effects
 - Scene Completion
 - example shown later
 - High Dynamic Range
 - increase ratio of light to dark



Computer Graphics

Computer Graphics Goals

1. Create synthetic images which cannot be distinguished from reality
 - ❑ Autodesk's Fake or photo Quiz
 - ❑ <http://area.autodesk.com/fakeorfoto>

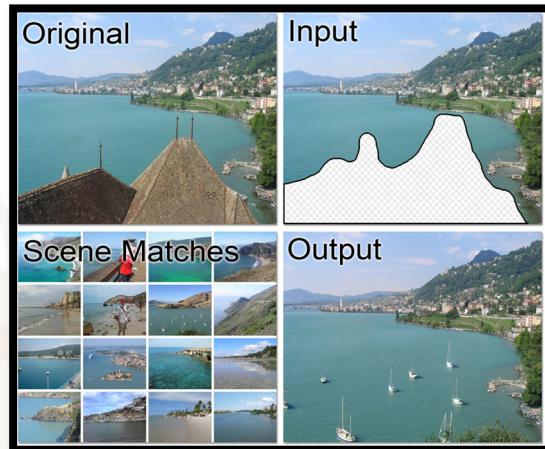


Computer Graphics

Computer Graphics Goals

2. Create a new reality (not necessarily scientific)

- Scene completion using Millions of photos
 - James Hays, Alexei Efros
- Non-photorealistic rendering
 - Aaron Hertzmann, Denis Zorin



Computer Graphics

Syllabus - Course Outline

- Posted on Moodle

Computer Graphics

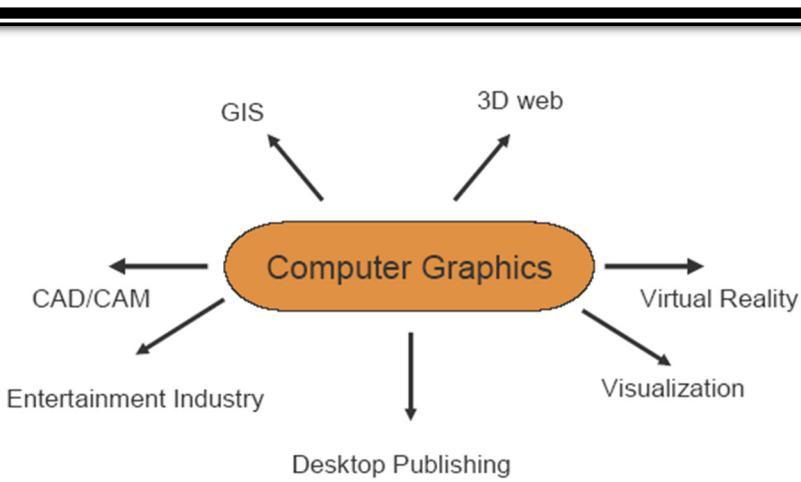
You will learn a lot!

- Fundamental concepts of 3D Graphics
- Common algorithms used in Graphics
- How to write programs using OpenGL
- How to deliver a project focused on 3D Graphics
- Many core CG concepts which the industry is looking for in CS employees!
- You do need programming knowledge, in particular C/C++, to take this course
- **WARNING: HIGH LOAD COURSE**

Computer Graphics

You will be able to

- Write C/C++ and OpenGL applications such as:
 - computer games and other graphical applications
- Take programming jobs for the 3D graphics and visual domain



Computer Graphics

Resources

- **Can run OpenGL on any system**
 - Windows: check graphics card properties for level of OpenGL supported
 - Linux: Mesa (software implementation of OpenGL)
 - Mac: need extensions for 3.1 equivalence
- **Get GLFW from web**
- **Get GLEW from web**
- **You will have remote access to labs with OpenGL already installed**

Computer Graphics

After this course

- **COMP 376 – Intro to Game Development**
- **COMP 476 – Advanced Game Development**
- **COMP 477 – Animation**

Computer Graphics

Slides Acknowledgements

- **Concordia University**
 - **Prof. Thomas Fevens**
 - **Prof. A Ben Hamza**
 - **Prof. Tiberiu Popa**
 - **Prof. Charalambos Poullis**
- **Others**
 - **E. Angel, P. Shirley, J. Barbic, R. Barzel, A. van Dam**

Computer Graphics

Next Lecture

- **Introduction to Computer Graphics**
 - **Graphics pipeline**
 - **OpenGL API**
 - **Primitives: lines, polygons**
 - **Attributes: color**
 - **Example**



COMP 371

Computer Graphics

Session 2

Introduction to Computer Graphics

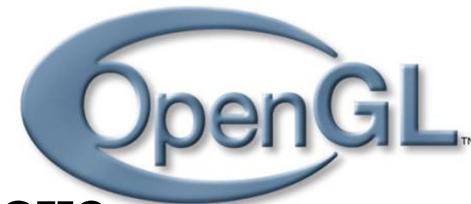
Lecture Overview

- OpenGL API
- Graphics pipeline
- Primitives: lines, polygons
- Attributes: color
- Example

OpenGL

What is OpenGL

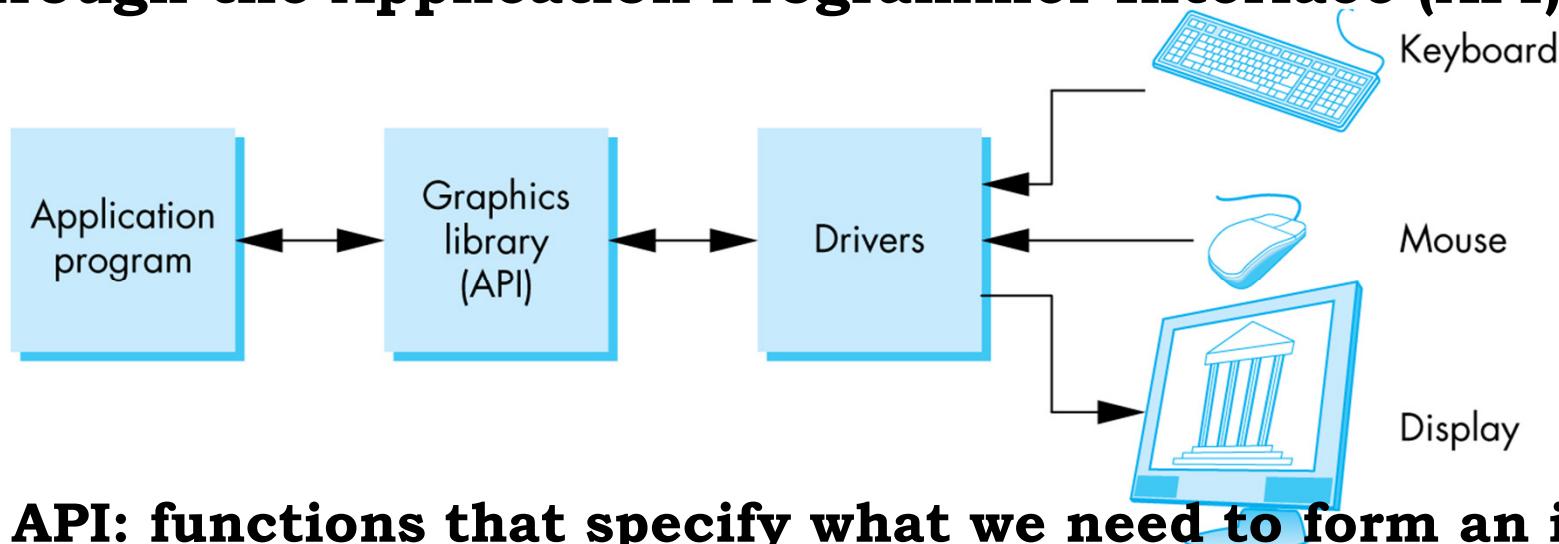
- **Low-level graphics library (API) for 2D/3D interactive graphics**
- **Originated from SGI's GL in 1992**
- **Version Used for this course – 3.1 or above**
- **New version 5.0 or Vulkan?**
- **Managed by Khronos Group (non-profit consortium)**
- **API is governed by Architecture Review Board (part of Khronos)**
- **Used in broadcasting, CAD/CAM/CAE, entertainment, medical imaging, and virtual reality to produce and display incredibly compelling 2D & 3D graphics**



OpenGL

OpenGL API

Programmer manages the graphics system functionality through the Application Programmer Interface (API)



- **API: functions that specify what we need to form an image**
 - object(s), viewer, light source(s), materials
- **Provides access to other information**
 - input from devices such as mouse and keyboard

OpenGL

OpenGL is Cross-Platform

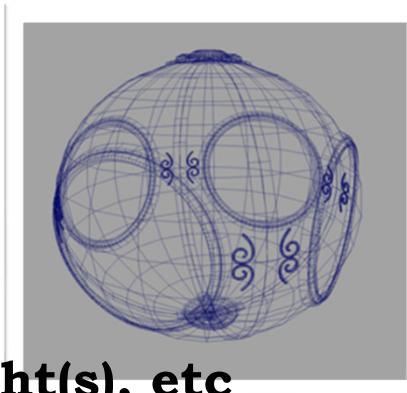
- **Available for Windows, Linux, Mac**
 - A freeware OpenGL implementation is also available on Linux called Mesa
- **Same code will work on different OSes**
 - most of the time with no modifications!

OpenGL

What an OpenGL program does?

From the programmer's point of view:

- Specifies the geometric object(s) in the scene
- Describes the properties of the object(s)
 - Color, material properties i.e. how object reflects light
- Defines how object(s) are viewed
 - Camera positioning, type
- Specifies light source(s)
 - Light positioning, type
- Creates animations
 - move object(s), camera, light(s), etc
- Specifies rendering properties
 - antialiasing, etc.



OpenGL

How OpenGL works?

OpenGL is a state machine

- You give it orders to set the current state of any one of its internal variables, or to query for its current status
 - State variables: color, camera position, light position, material properties, etc
- The current state persists until set to new values by the program
 - Example: once we set a color, that color remains the current color until it is changed through a color-altering function

OpenGL

Library Organization

- **OpenGL (GL)** - core graphics
 - function names begin with letters gl
- **OpenGL Framework Library (GLFW)** - provides programmers with the ability to create and manage windows and OpenGL contexts, as well as handling of input devices and events.
 - function names begin with letters glfw
- **OpenGL Extension Wrangler Library (GLEW)** - provides efficient run-time mechanisms for determining which OpenGL extensions are supported on the target platform.
 - function names begin with letters glew

OpenGL

Other libraries (some examples)

- **GLM – OpenGL Mathematics**
 - C++ math library based on the OpenGL Shading Language (GLSL) Specifications
- **OGRE – Object-Oriented Graphics Rendering Engine**
 - scene-oriented, real-time, 3D rendering engine. It is written in C++ and is designed to make it easier to write programs that use hardware-accelerated 3D graphics.

OpenGL

Immediate-mode Graphics [up to v3.1]

- Primitives are not stored in the system but rather passed through the system for possible rendering as soon as they become available
- Each time a vertex is specified in application, it is sent to the GPU
 - Creates data transfer bottleneck between CPU and GPU
 - Removed since OpenGL 3.1
- The scene is re-drawn for each frame

OpenGL

Retained mode Graphics [v3.1 onwards]

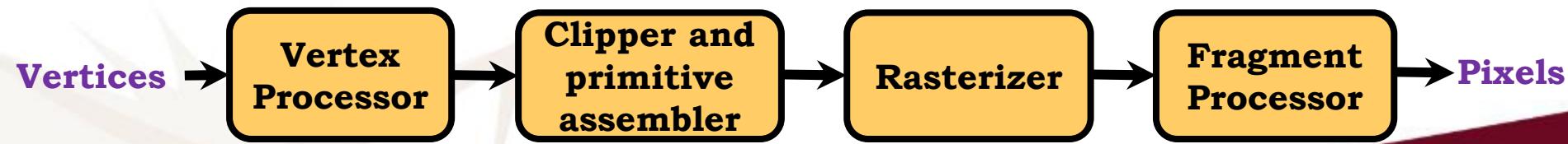
- We want to minimize the data transfers from CPU to GPU
- Solution:
 - Put all vertex and attribute data in array
 - Send array to GPU to be rendered immediately

This is almost OK but problem is we would have to send array over each time we need another render of it
- Revised solution:
 - Put all vertex and attribute data in array
 - Send array over and store on GPU for multiple renderings

OpenGL

Graphics Pipeline [simplified]

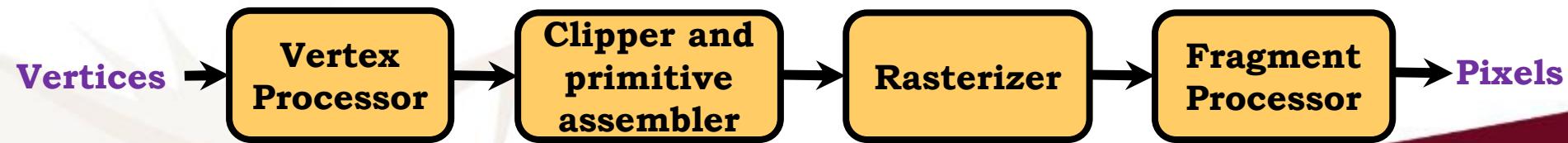
- A scene contains a set of objects
- Each object comprises a set of geometric primitives
- Each primitive comprises a set of vertices
- A complex scene can have millions of vertices that define the objects
- All vertices must be processed efficiently in a similar manner to form an image in the framebuffer
- Pipeline consists of four major steps in the imaging process



OpenGL

Graphics Pipeline [simplified]

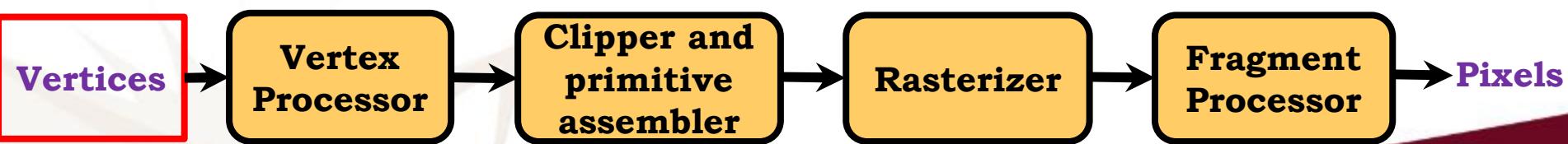
- Implemented by OpenGL, graphics drivers and the graphics hardware
 - programmer does not have to implement the pipeline
 - pipeline is configurable using special programs called “shaders”



OpenGL

Vertex

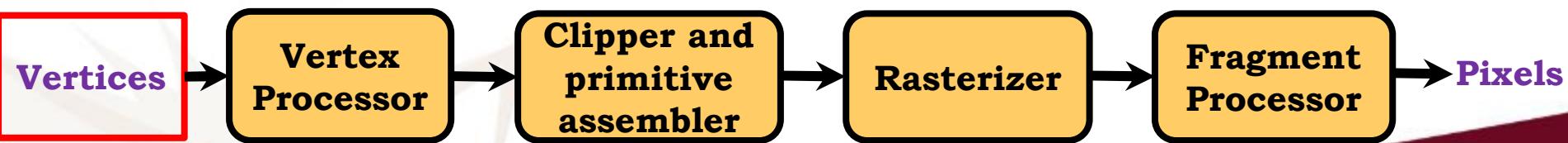
- The GPU is optimized to render triangles through the Graphics Pipeline
- Virtual Worlds can be Modeled using thousands of triangles. Curved surfaces can be approximated using triangles.
- Each triangle is composed of 3 vertices : position and other attributes (color, normal, material parameters, etc)



OpenGL

Vertices

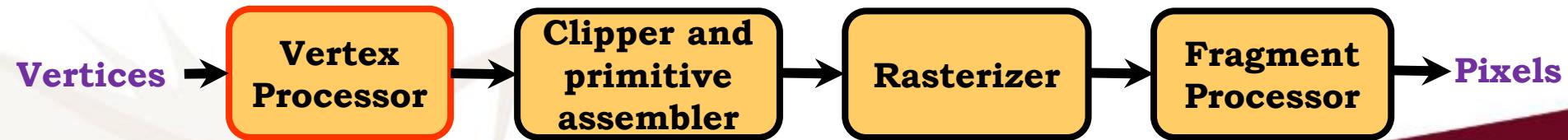
- Vertices are defined in world coordinates
- In OpenGL [immediate mode, deprecated]:
 - `void glVertex{234}{sfid}[v](TYPE coords,...)`
 - 2,3,4: the vertex dimensions
 - sfid: short, float, int, double
 - TYPE: short, float, int, double - for portability use GLtype
- E.g: `GLfloat x,y,z;`
 - `glVertex3f(x,y,z);` → sends vertex (x,y,z) down the pipeline



OpenGL

Vertex Processing

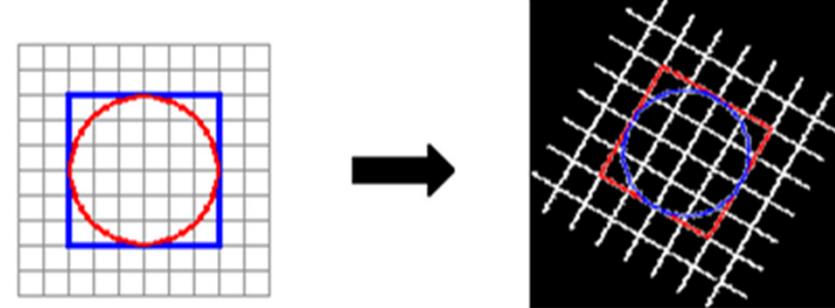
- Two major functions:
 - carry out coordinate transformations
 - compute a color for each vertex



OpenGL

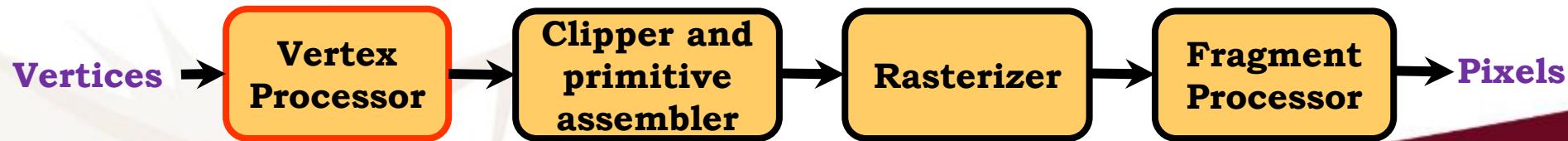
Vertex Processing → Coordinate Transform

- Transformations are functions that map points from one place to another
- Transformations in world coordinates



Deprecated

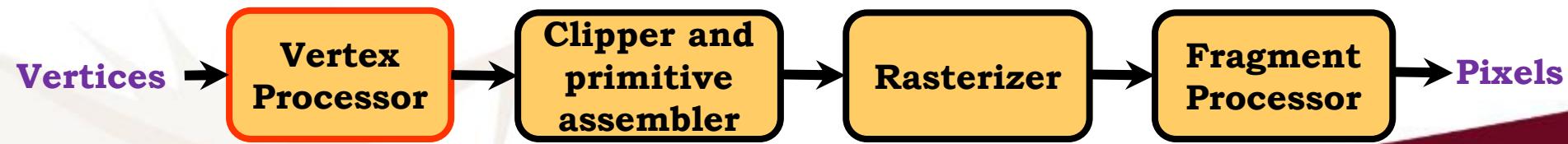
```
void glRotate{f,d}(GLtype angle, GLtype x, GLtype y, GLtype z)  
void glScale{f,d}(GLtype sx, GLtype sy, GLtype sz)  
void glTranslate{f,d}(GLtype tx, GLtype ty, GLtype tz)
```



OpenGL

Vertex Processing → Coordinate Transform

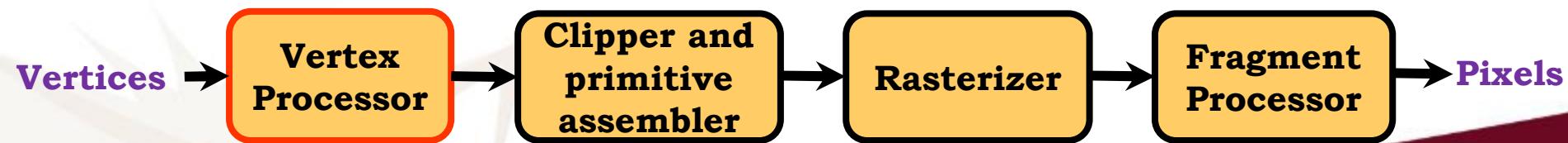
- Almost every step in the rendering pipeline involves a change of coordinate systems.
- Transformations are central to understanding 3D computer graphics.
- The best way to implement transformations is with matrix operations
- In 3D, transformation matrices are 4x4 matrices.



OpenGL

Vertex Processing → Coordinate Transform

- The position of each vertex will go through a sequence of transformations from the Model Coordinate System, to the Screen Coordinate System
- Every vertex on a Model is transformed using a transformation matrix taking into account the position of the model in the world, the camera position and projection parameters
- This Matrix is called the **Model-View-Projection Matrix**



OpenGL

Vertex Processing → Coordinate Transform

- Positions in each space can be computed as follow:

$$- P^W = M^{World} * P^M$$

$$- P^V = M^{View} P^W$$

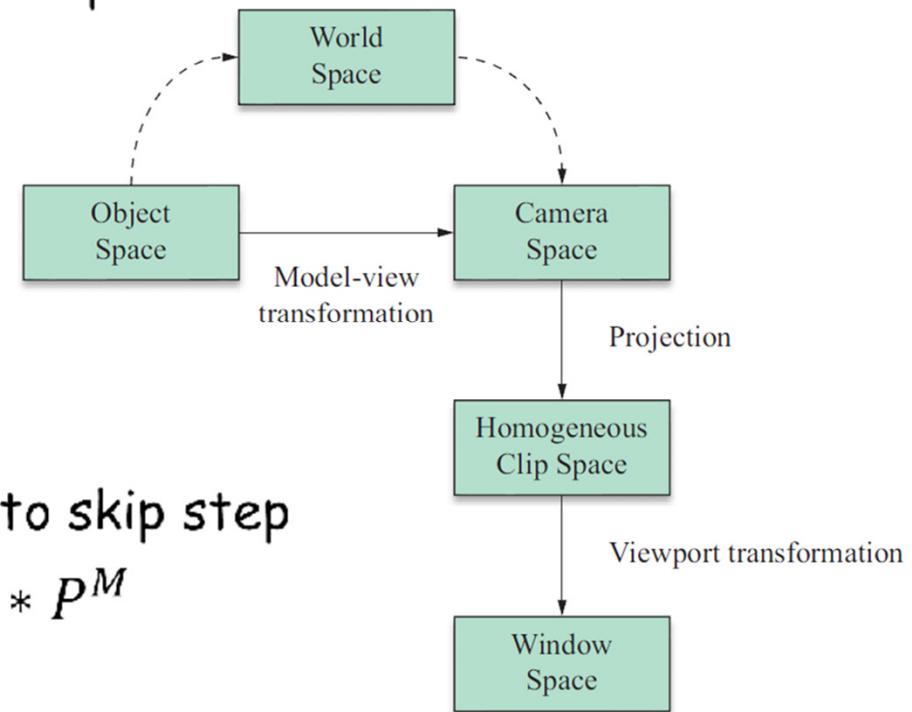
$$- \tilde{P} = M^{Proj} * P^V$$

$$- P^N = \tilde{P} / \tilde{w}$$

$$- P^D = M^{Viewport} * P^N$$

- Matrices can be concatenated to skip step

$$- \tilde{P} = M^{Proj} M^{View} M^{World} * P^M$$



OpenGL

Vertex Processing → Vertex color

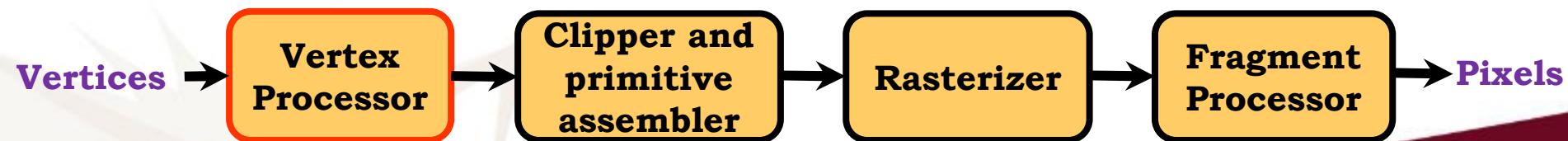
- Assignment of vertex colors

- Simple
- explicitly define color for each vertex [immediate mode]

```
void glColor{3,4}{b,s,i,f,d,ub,us,ui}(GLtype red, GLtype green,  
GLtype blue, [GLtype alpha])
```

Deprecated

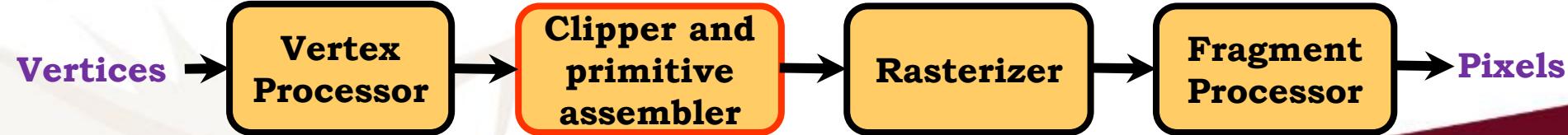
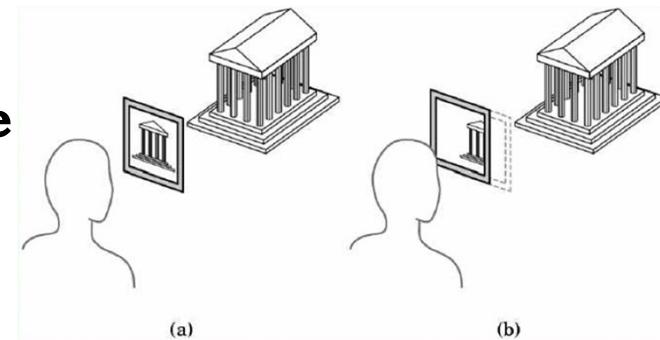
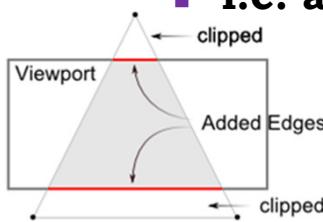
- Complex
- compute the color from a physically realistic lighting model which takes into account the surface properties of the object and the characteristic light sources in the scenes



OpenGL

Clipping and Primitive assembling

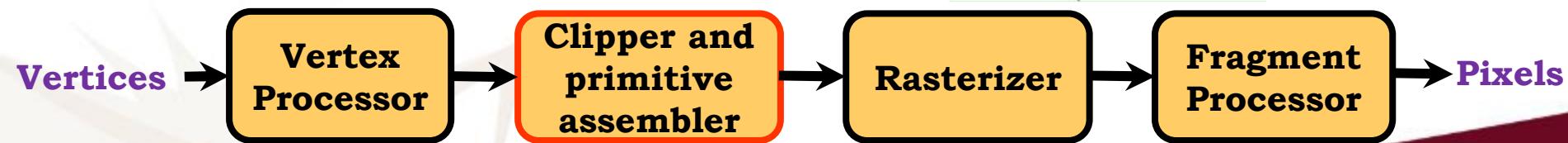
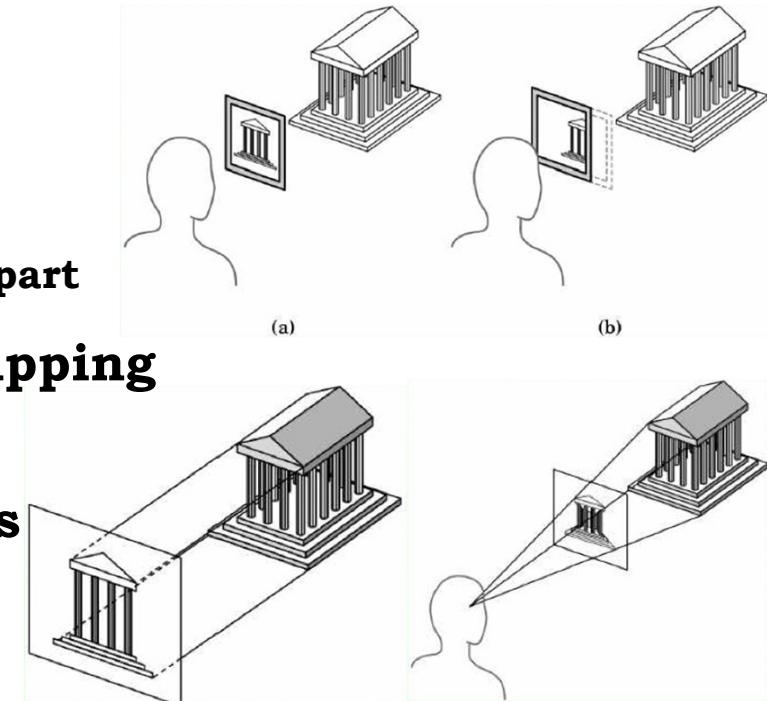
- No imaging system can see the whole world at once → Use a **clipping volume** to define visible area
- Clipping is done mostly automatically by defining the **viewport**
- Clipping is performed on primitives
- Primitive assembly must be done before
 - i.e. assemble sets of vertices into primitives



OpenGL

Clipping and Primitive assembling

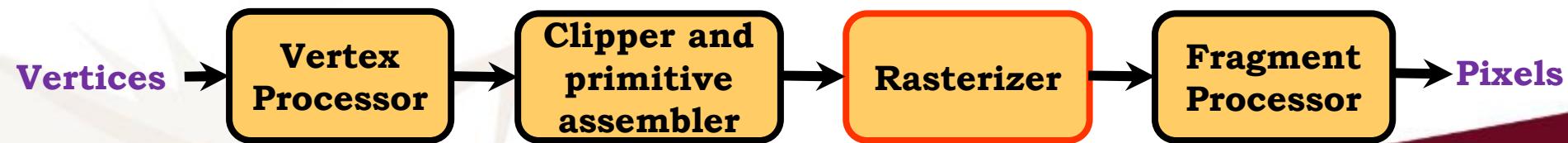
- Possible cases:
 - visible → continue processing
 - non-visible → stop processing
 - partially visible → continue with visible part
- The projections of objects in the clipping volume appear in the final image
- Projections are again represented as transformations
 - Orthographic vs Perspective



OpenGL

Rasterization

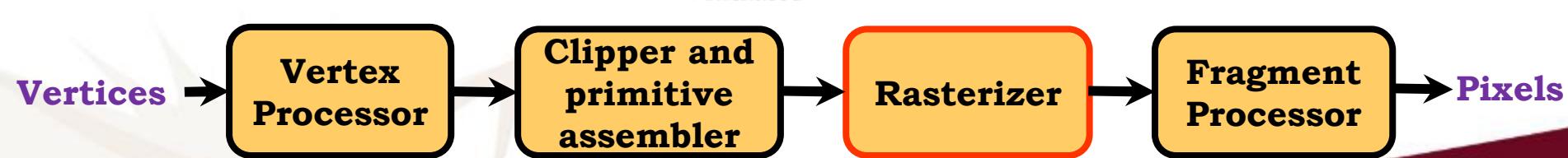
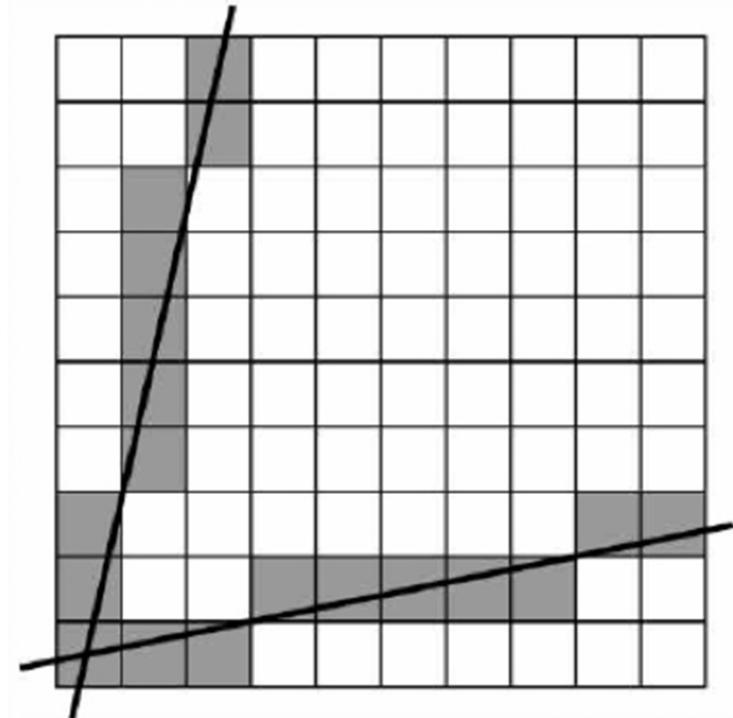
- The primitives that emerge from the clipper are still represented in terms of their vertices and must be further processed to generate pixels in the frame buffer
 - Example: three vertices defining a triangle filled with a solid color → the rasterizer determines which pixels in the frame buffer are inside the polygon
- Transforms vertices to window coordinates
- The output is a set of fragments for each primitive
 - A fragment is a potential pixel which carries information e.g. color, position, depth



OpenGL

Rasterization

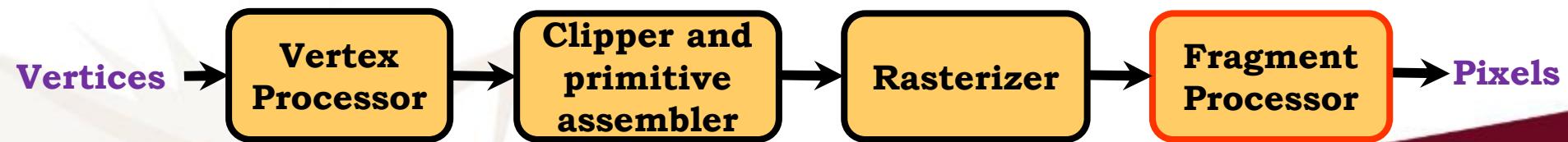
- Transform vertices to window coordinates?
 - Vertex values are interpolated inside the triangle
- Antialiasing



OpenGL

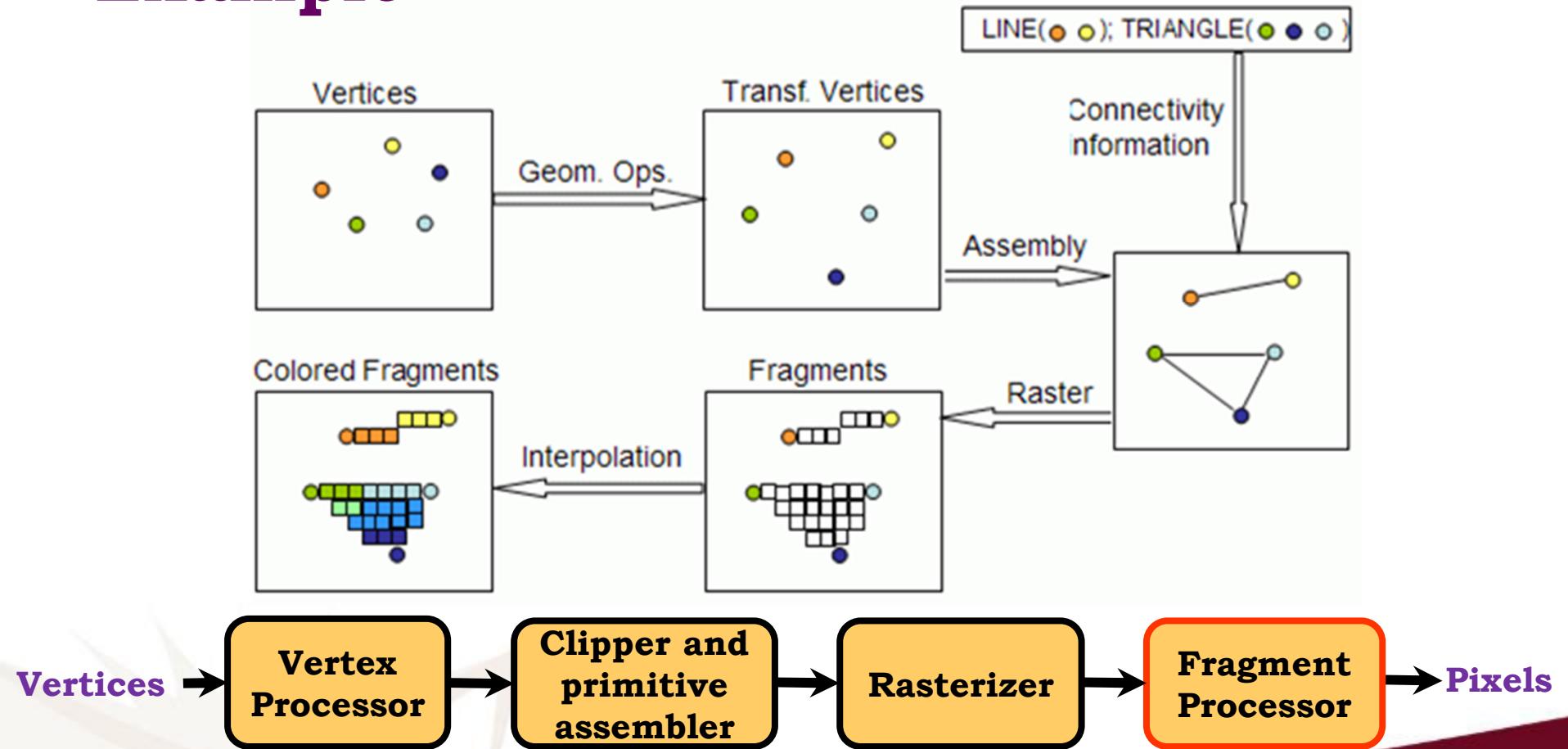
Fragment Processing

- The fragment processor takes in the fragments generated by the rasterizer and updates the pixels in the frame buffer
- Processing examples:
 - some fragments may not be visible
 - color may be altered because of texture or bump mapping
 - create translucent effects by combining the fragment's color with the color of the pixel



OpenGL

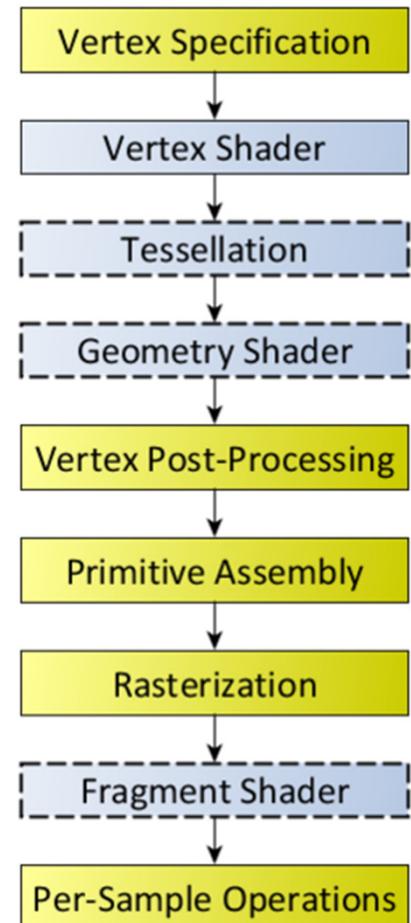
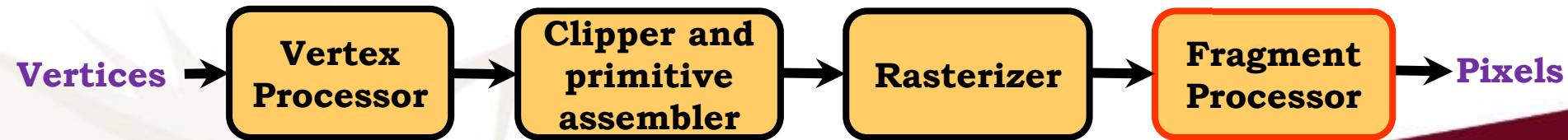
Example



OpenGL

Graphics Pipeline [detailed]

- Implemented by OpenGL, graphics drivers and the graphics hardware
 - programmer does not have to implement the pipeline
 - unless extra control is required → pipeline is reconfigurable using special programs called “shaders” → vertex, tessellator, geometry and fragment



OpenGL

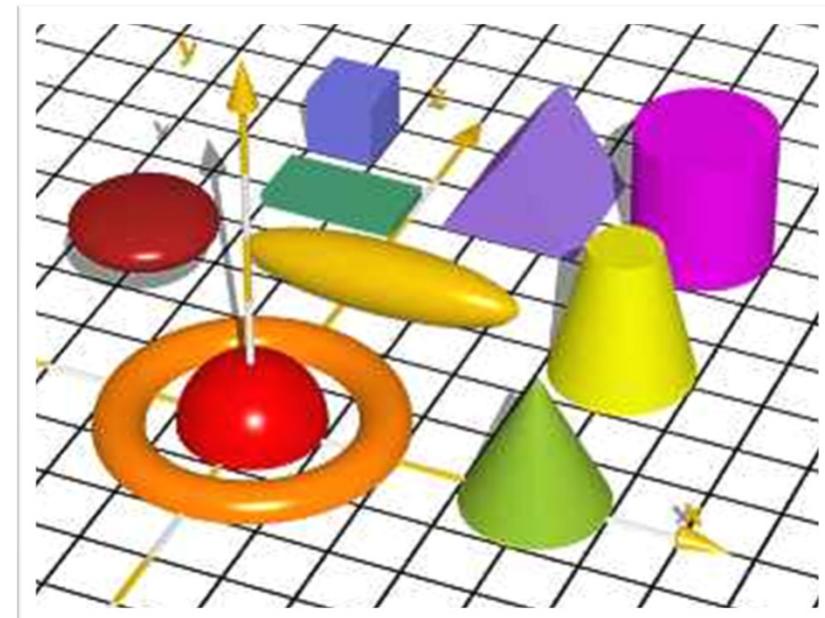
Primitives & Attributes

- In addition to the functions a graphics API needs to provide a way to represent objects in terms of geometry and appearance
 - Geometry → Primitives
 - Appearance → Attributes

OpenGL

Primitives

- Most APIs support a limited set of primitives including
 - Points (1D object)
 - Line segments (2D objects)
 - Polygons (3D objects)
 - Some curves and surfaces
 - Quadrics
 - Parametric polynomial
- Primitives are specified via vertices



OpenGL

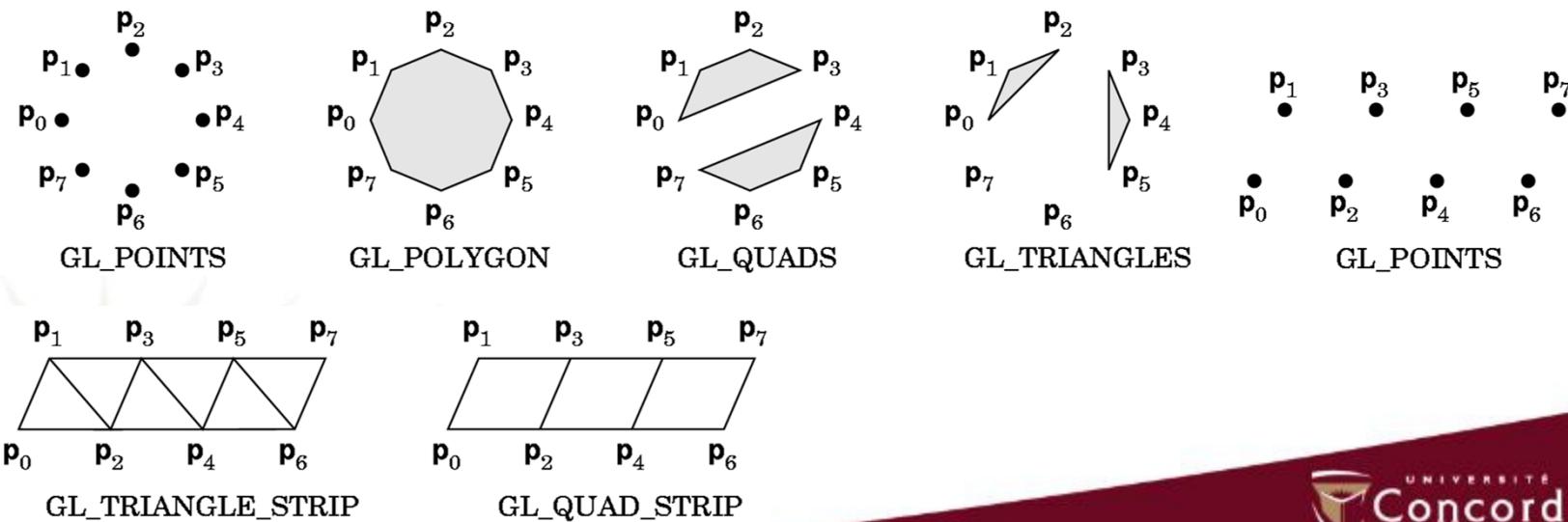
Polygon Representation

- There are different types of Primitives supported by the Graphics Hardware
- Choosing the right primitive type is important to ensure your geometry renders correctly and efficiently

OpenGL

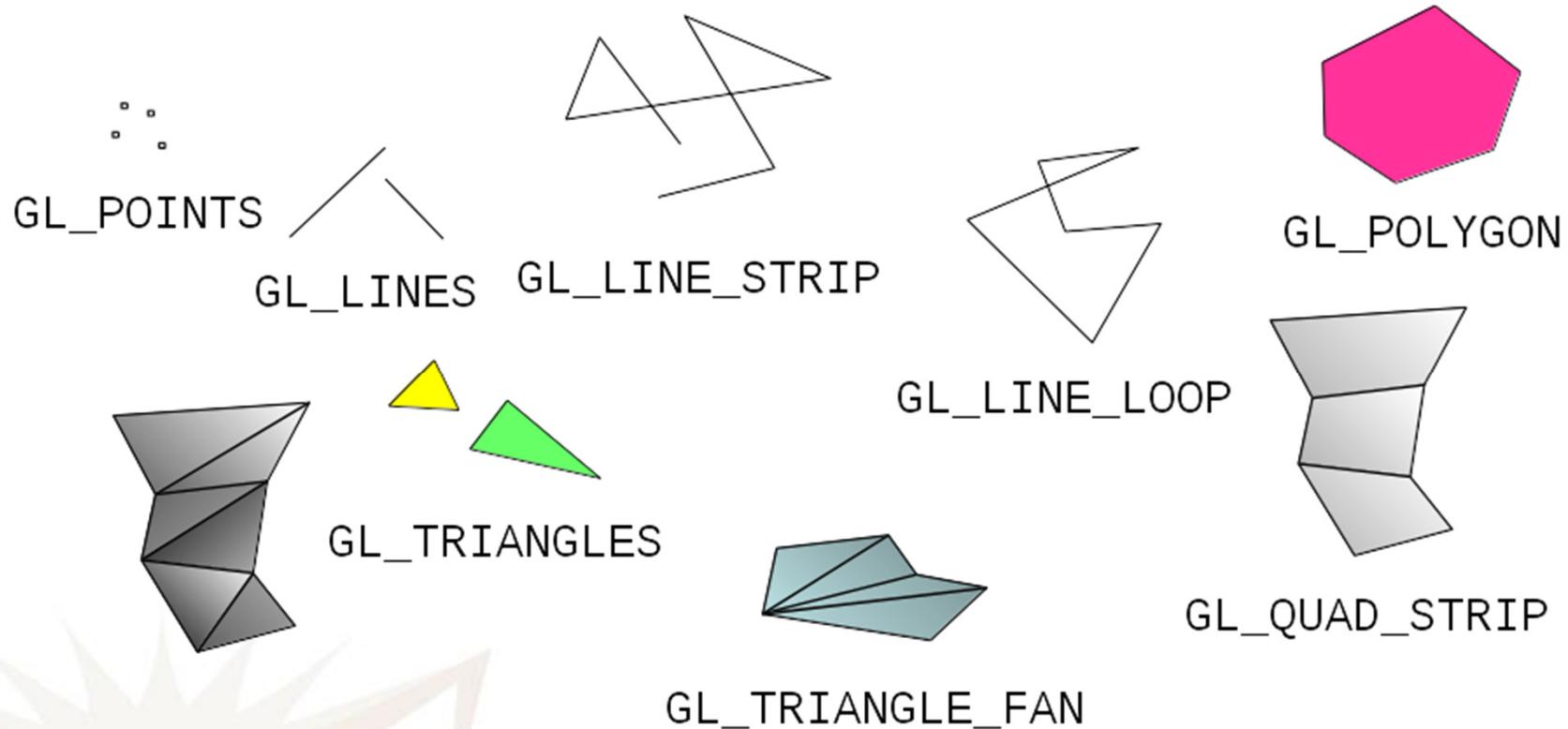
OpenGL Primitive Types

- **Polygons (GL_POLYGON)** successive vertices define line segments, last vertex connects to first
- **Triangles and Quadrilaterals (GL_TRIANGLES, GL_QUADS)** successive groups of 3 or 4 interpreted as triangles or quads
- **Strips and Fans (GL_TRIANGLE_STRIP, GL_QUAD_STRIP, GL_TRIANGLE_FAN)** joined triangles or quads that share vertices



OpenGL

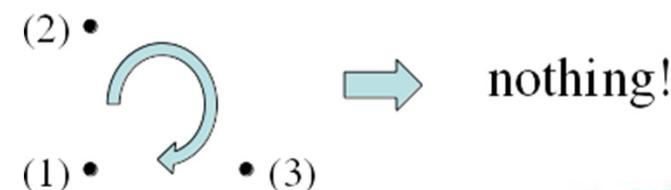
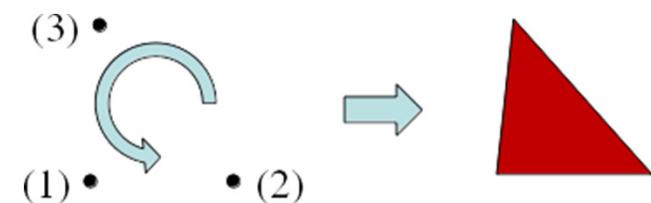
OpenGL Primitive Types



OpenGL

Winding Order

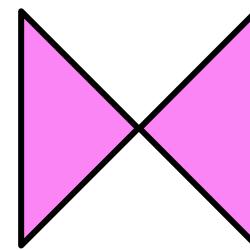
- Polygons in OpenGL are usually viewed from one-side, meaning they can only be viewed from their “front” side
- In order to draw the “front” side of a polygon, the vertices must be specified in counter-clockwise order from the perspective of the camera
- To get this behavior Backface Culling must be enabled
 - `glCullFace(GL_BACK);`



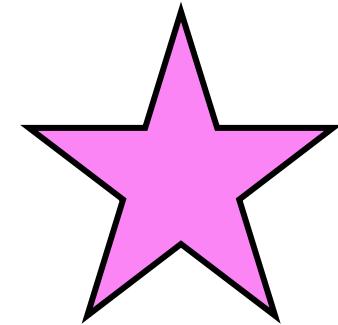
OpenGL

Polygon Issues

- OpenGL (driver) will convert quads and polygons into triangles and only display triangles
 - Simple: edges cannot cross
 - Convex: All points on line segment between two points in a polygon are also in the polygon
 - Flat: all vertices are in the same plane



Non-Simple Polygon

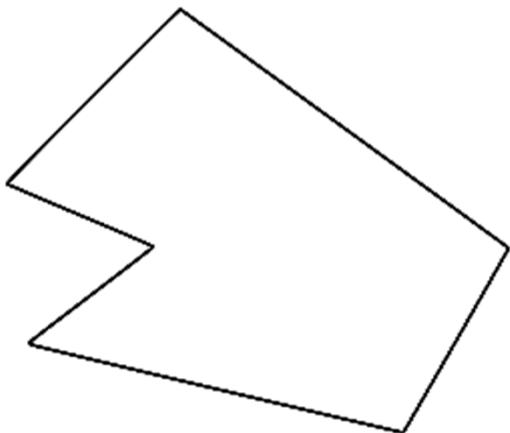


Non-Convex Polygon

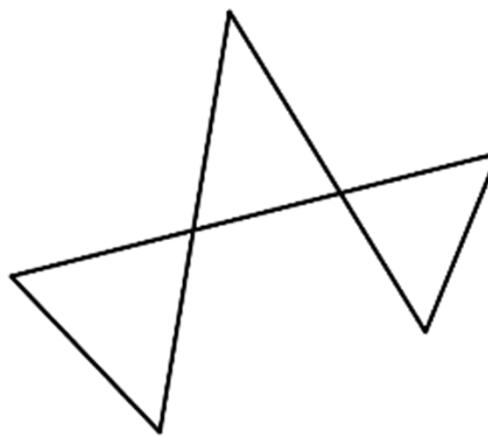
- Application program must tessellate a concave polygon into triangles (triangulation)
- OpenGL 4.1 contains a tessellator

OpenGL

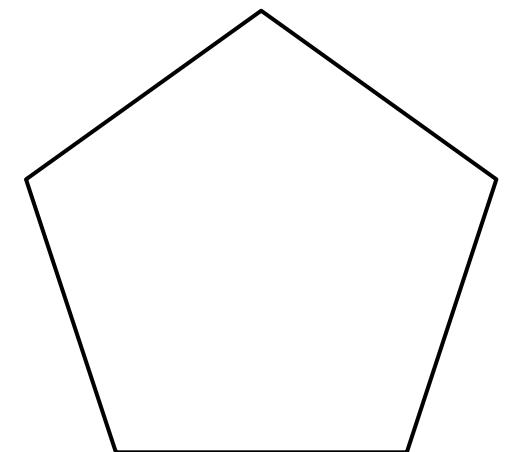
Examples



Simple Non-Convex Polygon



Complex Polygon



Simple and Convex Polygon

OpenGL

Polygon Issues

- Why only simple and convex?
 - Non-convex and non-simple polygons are expensive to process and render
- Convexity and simplicity is expensive to test
- Behavior of OpenGL implementation on disallowed polygons is “undefined”
- Some tools available in GLU (an older GL Utility functions library) for decomposing complex polygons (tessellation)
- Triangles are most efficient → use triangles in assignments

OpenGL

Attributes

- **Attributes determine the appearance of objects**
 - **Color (points, lines, polygons)**
 - **Size and width (points, lines)**
 - **Stipple pattern (lines, polygons)**
 - **Polygon mode**
 - **Display as filled: solid color or stipple pattern**
 - **Display edges**
 - **Display vertices**

OpenGL

Color

- Electromagnetic radiation
- Humans can see only part of the spectrum

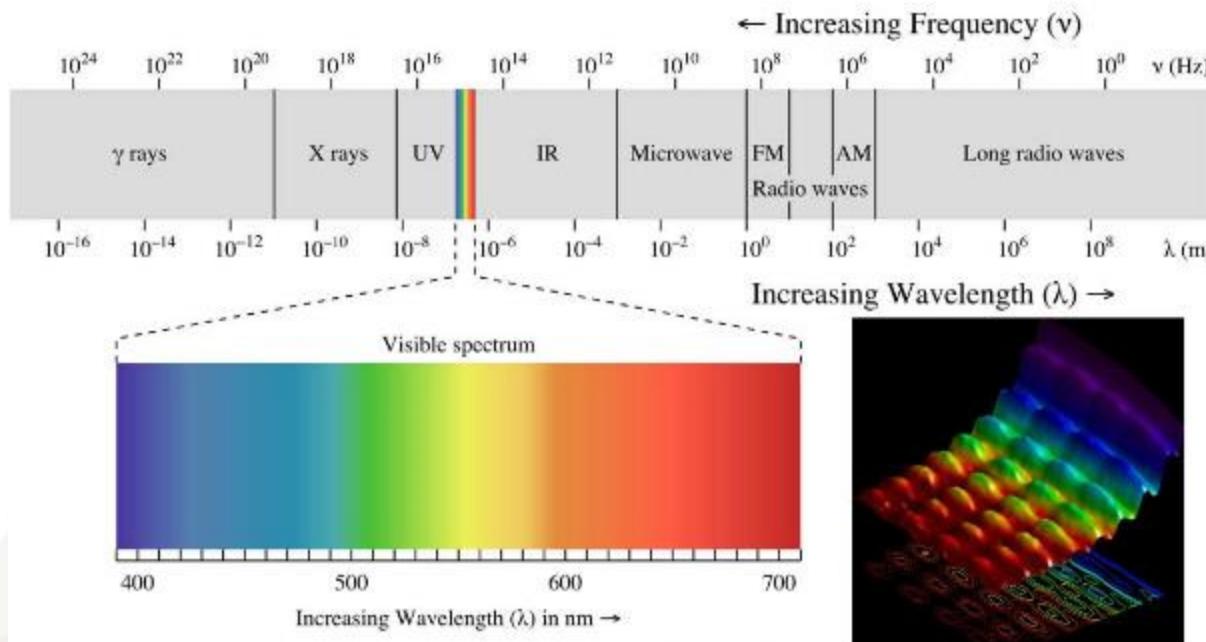
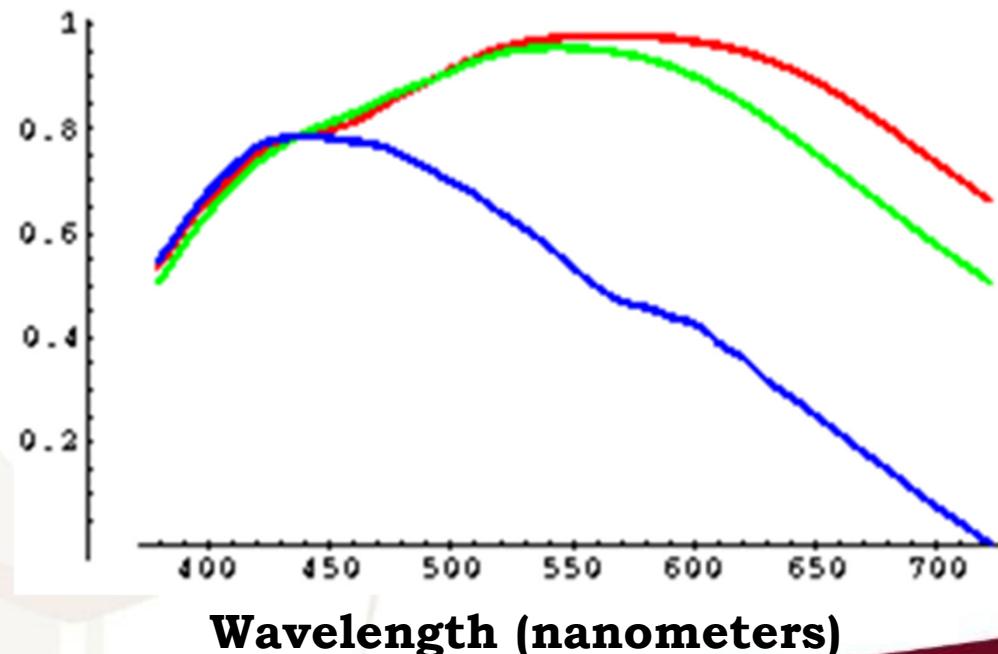


Image Courtesy: Socratic.org

OpenGL

Color

- The human eye has three types of cones which respond to particular wavelengths
- The wavelengths correspond to colors R, G, B

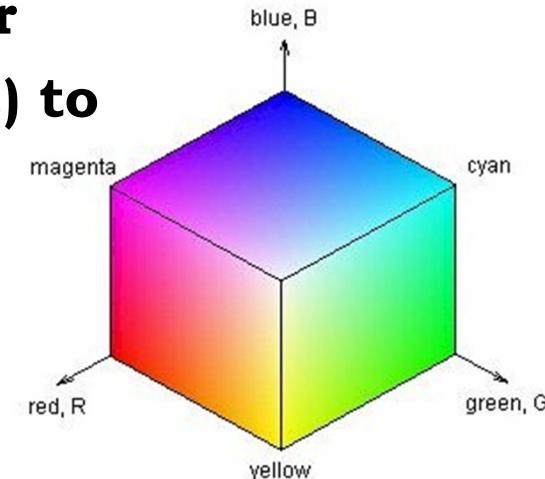


Source: Vos & Walraven

OpenGL

Color Models

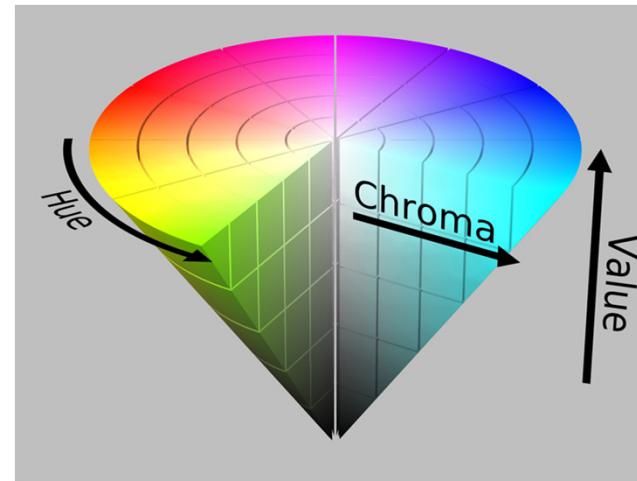
- **RGB (Red, Green, Blue)**
 - Each color component is stored separately in the frame buffer
 - Usually 8 bits per component in buffer
 - Color values can range from 0.0 (none) to 1.0 (all) using floats or over the range from 0 to 255 using unsigned bytes
- Convenient for display
- Can be unintuitive (3 floats in OpenGL)



OpenGL

Color Models

- **HSV (Hue, Saturation, Value)**
 - **Hue:** what color
 - **Saturation:** how far away from gray
 - **Value:** how bright



- **Other color models used for printing [CMYK], movies, etc**

OpenGL

Example

- Minimal example of using OpenGL v4.5 [retained mode] for drawing a triangle

- Example output:

Renderer: NVIDIA GeForce GTX ...

OpenGL version supported: 4.5.0

```
#include "../GL/glew.h"      // include GL Extension Wrangler
#include "../GLFW/glfw3.h"    // include GLFW helper library
#include <stdio.h>

int main () {
    /// Initialize GL context and O/S window using the GLFW helper library
    if (!glfwInit ()) {
        fprintf (stderr, "ERROR: could not start GLFW3\n");
        return 1;
    }

    /// Create a window of size 640x480 and with title "Lecture 2: First Triangle"
    GLFWwindow* window = glfwCreateWindow (640, 480, "Lecture 2: First Triangle", NULL, NULL);
    if (!window) {
        fprintf (stderr, "ERROR: could not open window with GLFW3\n");
        glfwTerminate();
        return 1;
    }
    glfwMakeContextCurrent (window);

    /// Initialize GLEW extension handler
    glewExperimental = GL_TRUE;    //Needed to get the latest version of OpenGL
    glewInit ();

    /// Get the current OpenGL version
    const GLubyte* renderer = glGetString (GL_RENDERER); // Get renderer string
    const GLubyte* version = glGetString (GL_VERSION);   // Version as a string
    printf ("Renderer: %s\n", renderer);
    printf ("OpenGL version supported %s\n", version);

    /// Enable the depth test i.e. draw a pixel if it's closer to the viewer
    glEnable (GL_DEPTH_TEST); // Enable depth-testing
    glDepthFunc (GL_LESS);   // The type of testing i.e. a smaller value as "closer"

    ///THE CODE GOES HERE

    // Close GL context and any other GLFW resources
    glfwTerminate();
    return 0;
}
```

OpenGL

Example – Draw Triangle

Draw a triangle → define 3 points

- **for now we ignore transformations and projections**
- **create an array with a total of 9 numbers corresponding to 3 points x 3 dimensions each x, y ,z**

```
float points[ ] = {  
    0.0f,  0.5f,  0.0f,  
    0.5f, -0.5f,  0.0f,  
   -0.5f, -0.5f,  0.0f  
}
```

OpenGL

Example – Draw Triangle

Draw a triangle → define 3 points

- for now we ignore transformations and projections
- create an array with a total of 9 numbers corresponding to 3 points x 3 dimensions each x, y ,z

Next step is to copy this part of memory onto the graphics card in a unit called **Vertex Buffer Object (VBO)**

- generate an empty buffer
- set it as the current buffer i.e. bind
- copy the points

Next step is to setup the **Vertex Array Object (VAO)**

- collection of VBOs to hold vertex points, texture coordinates, colors, etc.
- setup one per mesh → when we want to draw, we bind the VAO and draw
- glVertexAttribPointer defines the layout for attribute e.g. 0

```
float points[ ] = {  
    0.0f,      0.5f,      0.0f,  
    0.5f,     -0.5f,      0.0f,  
   -0.5f,     -0.5f,      0.0f  
};
```

```
GLuint vao = 0;  
 glGenVertexArrays(1, &vao);  
 glBindVertexArray(vao);  
  
 glEnableVertexAttribArray(0);  
 GLuint vbo = 0;  
 glGenBuffers(1, &vbo);  
 glBindBuffer(GL_ARRAY_BUFFER, vbo);  
 glBufferData(GL_ARRAY_BUFFER, 9 * sizeof(float), points,  
 GL_STATIC_DRAW);  
 glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, NULL);
```

OpenGL

Example – Shaders

Need to define how to draw the VAO → Programmable Shaders

- Looks a lot like programming in C
- You can load the shaders from a file or write them in-line
- Note that the attribute pointer matches the input variables in the shader

Vertex Shader

- describes where the 3D points will end up on the display, runs once per vertex
- 1 input variable → vec3 (matches the VBO attr. pointer)
- gl_Position is a reserved name → vec4

```
const char* vertex_shader =
"#version 400\n"
"in vec3 vp;"
"void main () {"  
"    gl_Position = vec4 (vp, 1.0);"
"}";  
  
const char* fragment_shader =
"#version 400\n"
"out vec4 frag_colour;"  
"void main () {"  
"    frag_colour = vec4 (0.5, 0.0, 0.5, 1.0);"
"}";
```

Fragment Shader

- describes the colours of fragments, runs once per fragment
- 1 output variable → vec4 (red, green, blue, alpha)
- values range from 0-1

OpenGL

Example – Using Shaders

- In order to use the shaders
- Load the strings into a GL shader
- Compile
- The compiled shaders must be combined into a single executable GPU shader program
- create an empty program
- attach the shaders
- link them together

```
GLuint vs = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vs, 1, &vertex_shader, NULL);
glCompileShader(vs);

GLuint fs = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fs, 1, &fragment_shader, NULL);
glCompileShader(fs);
```

```
GLuint shader_programme = glCreateProgram();
glAttachShader(shader_programme, fs);
glAttachShader(shader_programme, vs);
glLinkProgram(shader_programme);
```

OpenGL

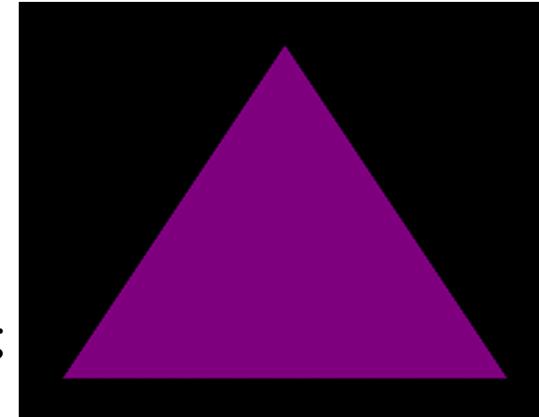
Example – Drawing

To draw our VAO

- Clear the drawing surface
- Specify which GPU shader program to use
- Set the VAO as the input variables for further drawing
- Draw
- Triangles
- Starting from point 0
- 3 points at a time

```
while (!glfwWindowShouldClose (window)) {  
    // wipe the drawing surface clear  
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);  
    glUseProgram (shader_programme);  
    glBindVertexArray (vao);  
    // draw points 0-3 from the currently bound VAO with current in-use shader  
    glDrawArrays (GL_TRIANGLES, 0, 3);  
    // update other events like input handling  
    glfwPollEvents ();  
    // put the stuff we've been drawing onto the display  
    glfwSwapBuffers (window);  
}
```

Display Loop



Result

OpenGL

Review

- **OpenGL API**
- **Graphics Pipeline**
- **Primitives: vertices, lines, polygons**
- **Attributes: color**
- **Example**

OpenGL

Next Lecture

- ❑ **Input and Interaction**
 - ❑ Client/Server Model
 - ❑ Callbacks
 - ❑ Double Buffering
 - ❑ Hidden Surface Removal
 - ❑ Simple Transformations



COMP 371

Computer Graphics

Session 3
INPUT AND INTERACTION



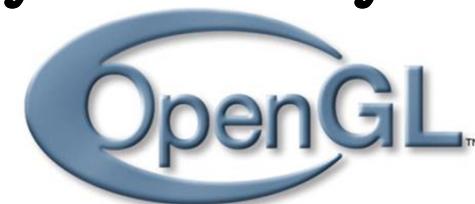
Lecture Overview

- Review of last class
- Input and Interaction
- Clients and Servers, Input Modes
- Drawing
- Double Buffering
- Hidden Surface Removal

OpenGL

What is OpenGL

- OpenGL is a computer graphics rendering API
 - generate high-quality images by rendering with geometric and image primitives
- Cross platform → widely used → forms the basis of many interactive applications which include 3D graphics
- Allows you to focus on the graphics component of your application and not worry about any dependencies
 - operating system independent
 - window system independent



OpenGL

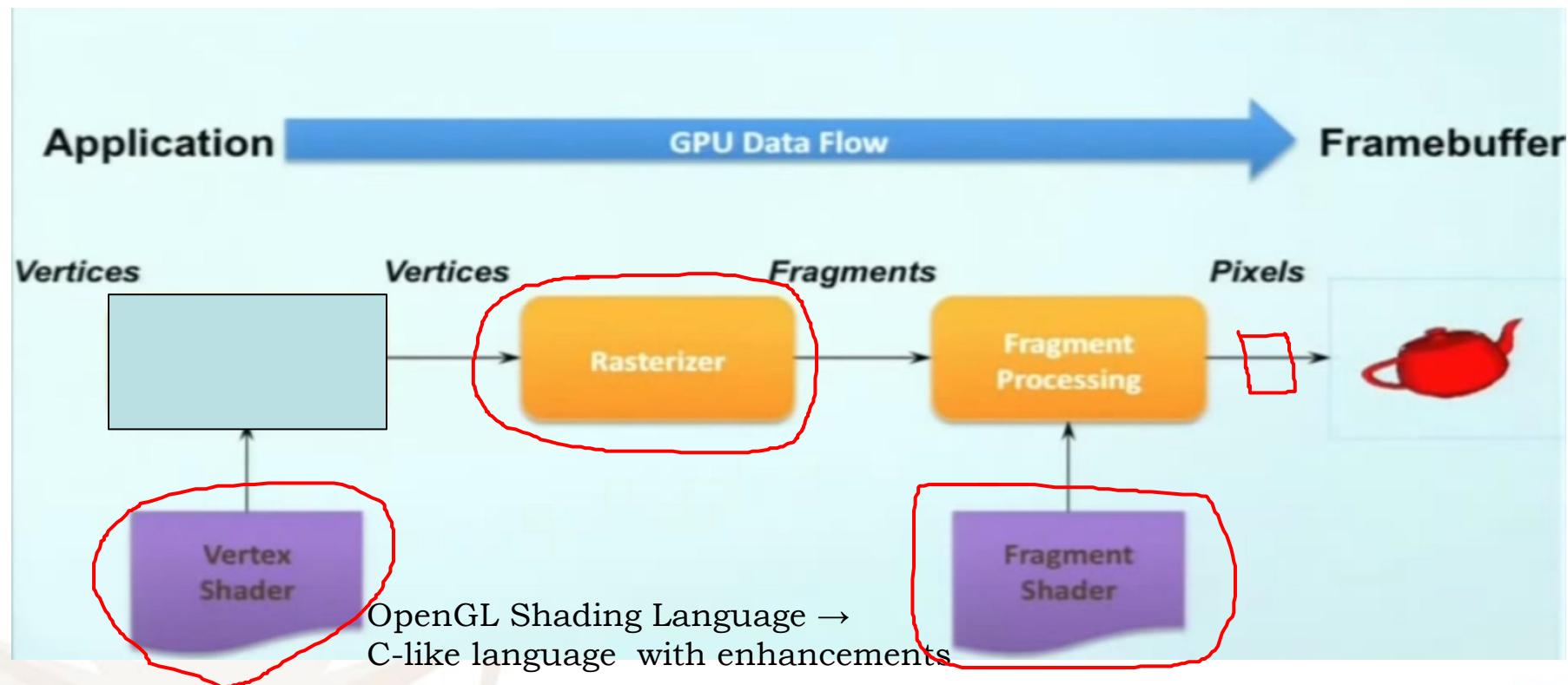
OpenGL 3.1 and Onwards

We'll focus on the latest versions of OpenGL

- They enforce a new paradigm to work with OpenGL
 - Allows more efficient use of GPU resources
- If you are familiar with “classic” graphics pipelines, modern OpenGL will not support in future versions
 - Fixed-function graphics operations
 - e.g. glVertex*, glColor*, etc.
 - lighting e.g. glLight*
 - transformations glTranslate*, glScale*, glRotate*
- All applications must use shaders for their graphics processing
 - they must provide at least a vertex and fragment shader

OpenGL

Graphics Pipeline [Simplified]



Slide credit: Ed Angel,
SIGGRAPH 2014

OpenGL

[Minimal] OpenGL application

- **Create shader programs [vertex, fragment]**
- **Create containers [buffer objects], load data into them and pass them over to the GPU**
- **“Connect” data locations with shader variables**
- **Render**

OpenGL

Application Framework Requirements

- OpenGL application needs to draw on screen [window]
 - communicate with native windowing system
- Libraries used in course:
 - GLFW is a simple, open-source library → up-to-date [compared to GLUT] that deals with window management, input processing, etc.
 - GLEW is a simple, open-source library → handles the extensions

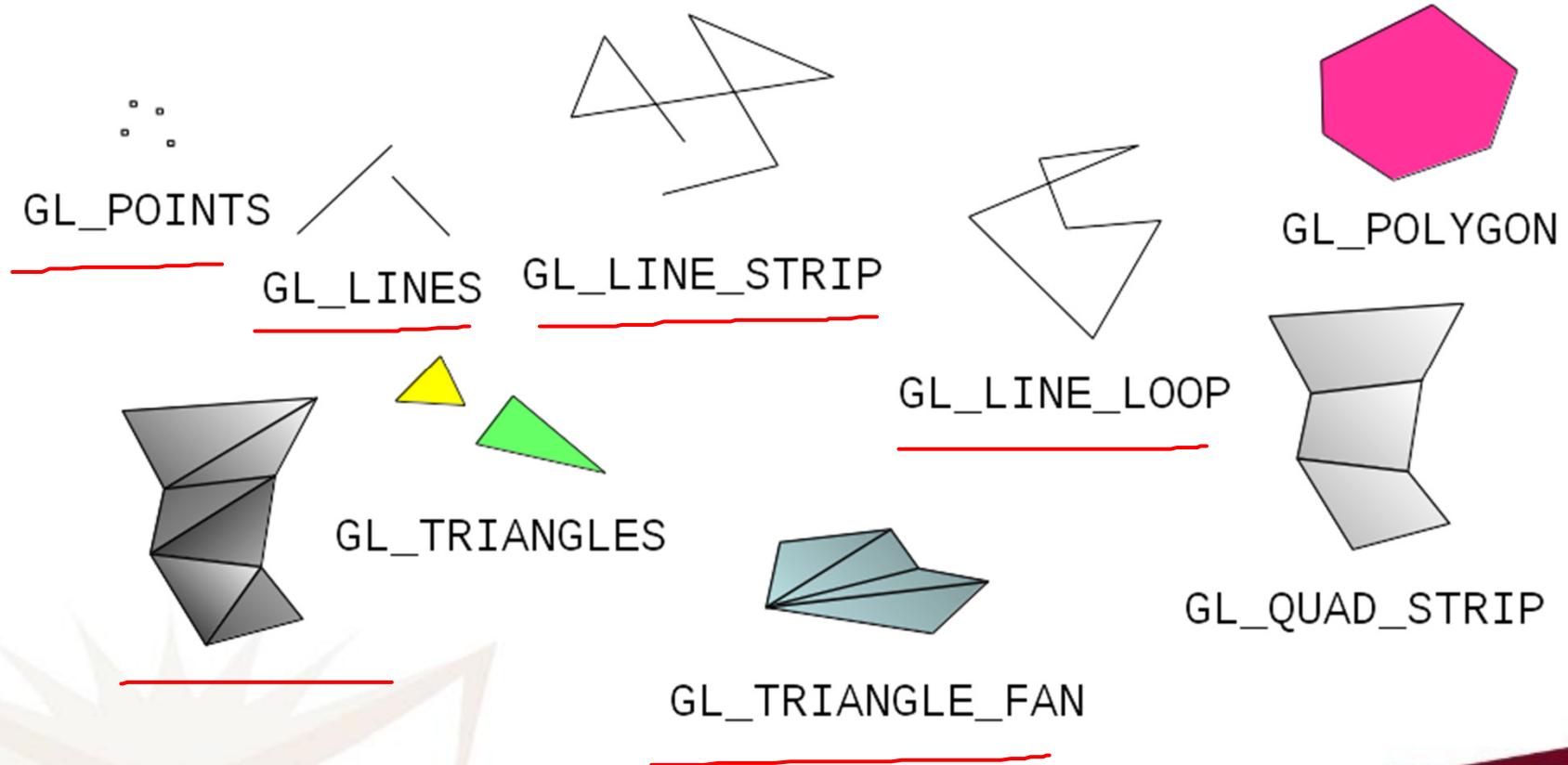
OpenGL

Primitive Representation

- Primitives are represented using vertices
- A vertex is a positional coordinate in 3D Euclidean space
 - stored in 4D homogeneous coordinates [x, y, z, w]
 - allow to give common representation to positions as well as vectors [more on this later]
- Vertex data must be stored in vertex buffer objects (VBOs)
- VBOs must be stored in vertex array object (VAOs)

OpenGL

OpenGL Primitive Types



Input and Interaction

Client - Server

- This model was popularized by the X Window System
 - Servers perform tasks for clients
- A workstation with a display, a keyboard, and a pointing device is a **graphics server**
- OpenGL application programs are **clients** that use the graphics server
- Server can provide output services on its display and input services through the input devices e.g. keyboard, mouse, etc.

Input and Interaction

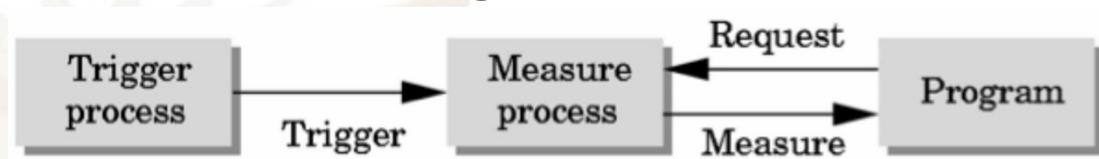
Input Modes

- Input devices contain a trigger which can be used to send a signal to the operating system
 - Button on mouse
 - Pressing or releasing a key e.g. Return or Enter
- When triggered, input devices return information i.e. measure to the system
 - Mouse returns position information
 - Keyboard returns ASCII code
- The application can obtain the measure of a device in three modes
 - request, sample and event modes

Input and Interaction

Request Mode

- Input provided to program only when user triggers the device
- Typical of keyboard input
 - Can erase (backspace), edit, correct until enter (return) key (the trigger) is released
 - a C program requires a character input → e.g. `scanf("%d",&x);`
 - halts when `scanf` is encountered and waits
 - keyboard buffer contents are returned to the program only after hitting Enter or Return



Input and Interaction

Sample Mode [Polling]

- Input is immediate
- No trigger is needed
- The user must have positioned the pointing device or entered data using the keyboard before the function call

Input and Interaction

Request & Sample Modes

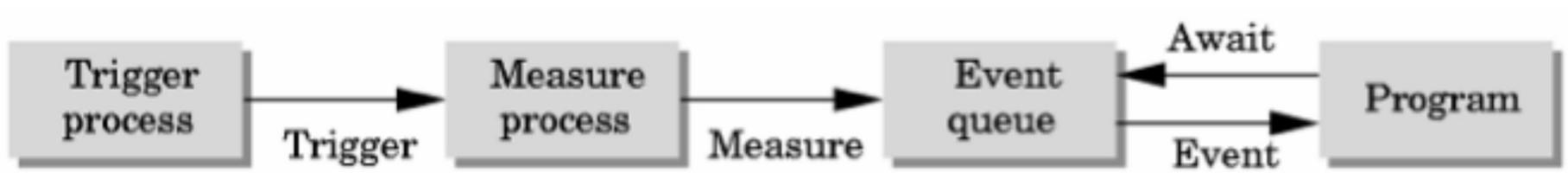
- User must identify which device will provide the input!
 - very hard to use for graphical applications e.g. games, simulators, etc.
 - don't know what devices the user will use at any point in the game/simulation



Input and Interaction

Event Mode

- Most systems have more than one input device, each of which can be triggered at an arbitrary time by a user
- Each trigger generates an event whose measure is put in an event queue which can be examined by the user program



Input and Interaction

Event Types

- **Window: resize, expose, iconify**
- **Mouse: click one or more buttons**
- **Motion: move mouse**
- **Keyboard: press or release a key**

Input and Interaction

Callbacks

- Programming interface for event-driven input
- Define a callback function for each type of event the graphics system recognizes
- This user-supplied function is executed when the event occurs
- **GLFW example:**

Register the callback function for when mouse moves

```
glfwSetCursorPosCallback(window, myCursorMovedCallback);
```

Callback functions have different input parameters according to the type of event

Provide the implementation of the callback function with the same name

```
static void myCursorMovedCallback(GLFWwindow* window, double xpos, double ypos)  
{ ..... }
```

Input and Interaction

GLFW Callback Input Function Signatures

```
typedef void(* GLFWmousebuttonfun )(GLFWwindow *, int, int, int)
The function signature for mouse button callbacks. More...

typedef void(* GLFWcursorposfun )(GLFWwindow *, double, double)
The function signature for cursor position callbacks. More...

typedef void(* GLFWcursoreenterfun )(GLFWwindow *, int)
The function signature for cursor enter/leave callbacks. More...

typedef void(* GLFWscrollfun )(GLFWwindow *, double, double)
The function signature for scroll callbacks. More...

typedef void(* GLFWkeyfun )(GLFWwindow *, int, int, int, int)
The function signature for keyboard key callbacks. More...

typedef void(* GLFWcharfun )(GLFWwindow *, unsigned int)
The function signature for Unicode character callbacks. More...

typedef void(* GLFWcharmodsfun )(GLFWwindow *, unsigned int, int)
The function signature for Unicode character with modifiers callbacks. More...

typedef void(* GLFWdropfun )(GLFWwindow *, int, const char **)
The function signature for file drop callbacks. More...
```

Input and Interaction

GLFW Callback Window Function Signatures

typedef struct GLFWwindow GLFWwindow	Opaque window object. More...
typedef void(*) GLFWwindowposfun)(GLFWwindow *, int, int)	The function signature for window position callbacks. More...
typedef void(*) GLFWwindowsizefun)(GLFWwindow *, int, int)	The function signature for window resize callbacks. More...
typedef void(*) GLFWwindowclosefun)(GLFWwindow *)	The function signature for window close callbacks. More...
typedef void(*) GLFWwindowrefreshfun)(GLFWwindow *)	The function signature for window content refresh callbacks. More...
typedef void(*) GLFWwindowfocusfun)(GLFWwindow *, int)	The function signature for window focus/defocus callbacks. More...
typedef void(*) GLFWwindowiconifyfun)(GLFWwindow *, int)	The function signature for window iconify/restore callbacks. More...
typedef void(*) GLFWframebuffersizefun)(GLFWwindow *, int, int)	The function signature for framebuffer resize callbacks. More...

Input and Interaction

GLFW Error Handling

- Define an error callback function

```
void error_callback(int error, const char* description)
{
    fputs(description, stderr);
}
```

- Register the callback function

glfwSetErrorCallback(error_callback);

Input and Interaction

GLFW Main Event Loop

- Main loop processes the events in the event queue
- Processing events will cause the window and input callbacks associated with those events to be called
 - i.e. “Call back” functions registered by the client

void glfwPollEvents (void) processes only those events that are already in the event queue and then returns immediately

Drawing

Double Buffering

- When [re]-drawing on the display we must do so at a high enough rate that we cannot notice the clearing and redrawing
 - Screen refreshing: common 60-100 Hz (times per second)
 - if frame buffer contents are unchanged → OK
 - if the contents change i.e. animation → flickering, undesirable
- **Solution:** Use two different frame buffers
 - Front: always the one displayed
 - Back: one into which we draw
- The buffers are swapped only after an explicit function call by the application program i.e.

void glfwSwapBuffers (GLFWwindow * window)

Drawing

Controlling Speed

- GPUs can render millions of primitives per second
 - Running a simple program with a rotating cube will result in the cube being rendered thousands of times per second → blur on the display
- **Solution:** Use two different frame buffers
 - Use timing mechanisms provided by libraries or OS to put delays
 - Lock the swapping of buffers to the screen refresh rate
 - glfwSwapInterval(1) → blocks swapping until the monitor has done at least one vertical redraw since the last buffer swap
 - puts an upper limit on the frame rate at the monitor's refresh rate

Drawing

Hidden Surface Removal

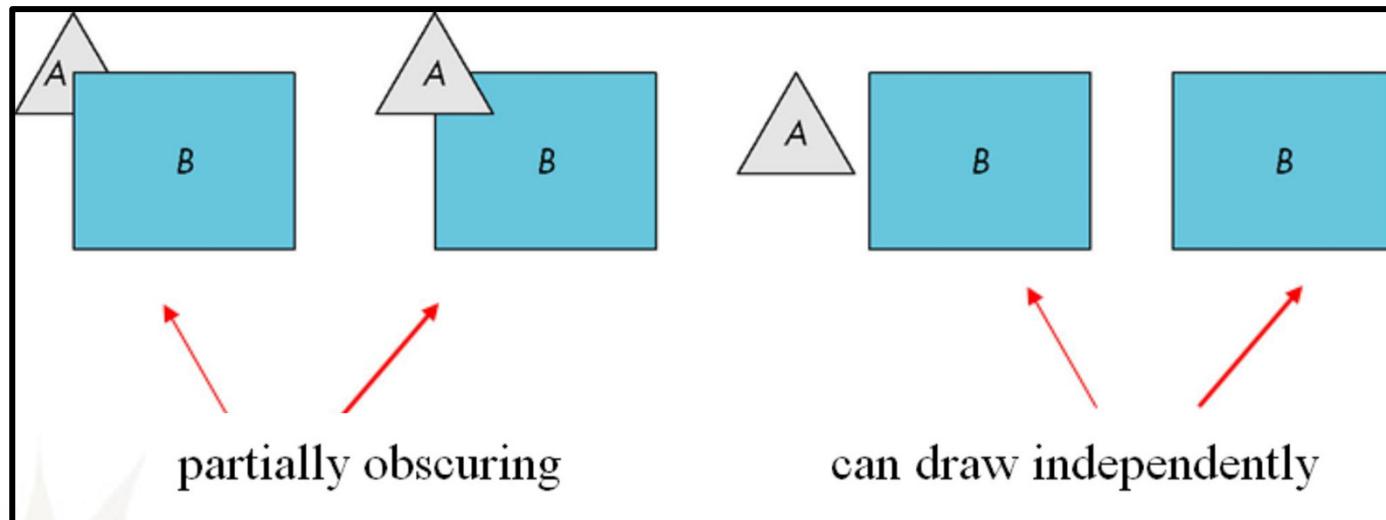
- **Objective:** Remove surfaces which are not visible to the viewer
- Many solutions have been proposed
 - Object-space algorithms
 - Image-space algorithms

Drawing

Hidden Surface Removal (Object Space)

Object-space Approach:

- Use pairwise testing between polygons (objects)



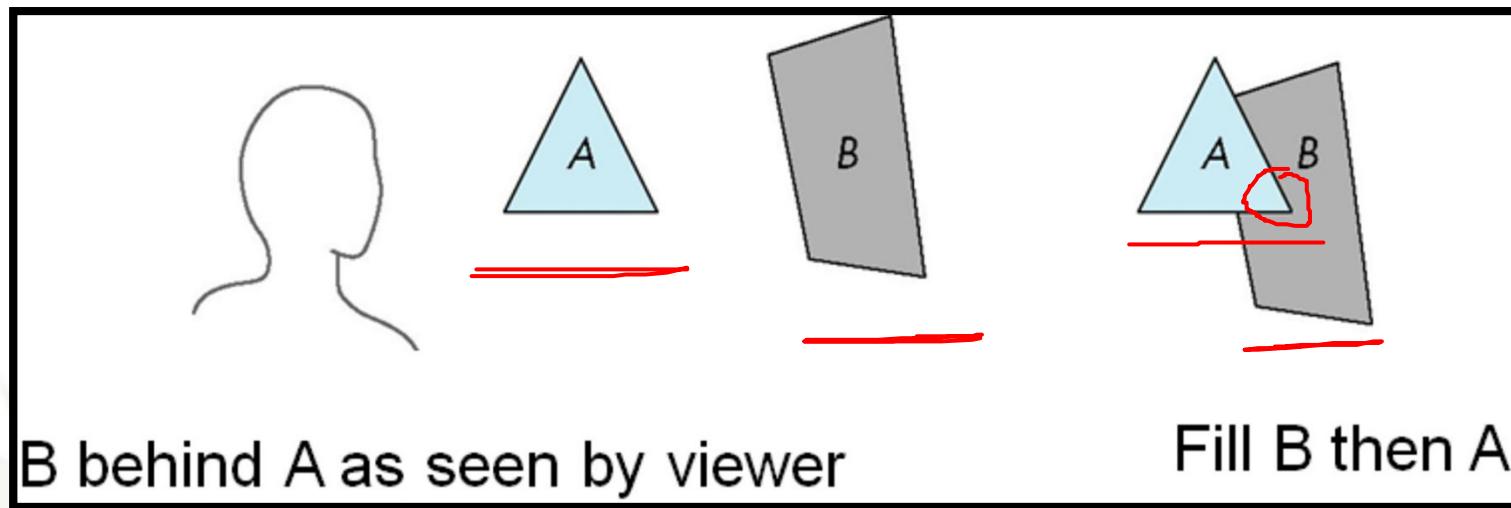
- Worst case complexity $O(n^2)$ for n polygons

Drawing

Hidden Surface Removal (Object Space)

Painter's Algorithm

- Render polygons in a back to front order so that polygons behind others are simply painted over



E. Angel and D. Shreiner: Interactive Computer Graphics 6E © Addison-Wesley 2012

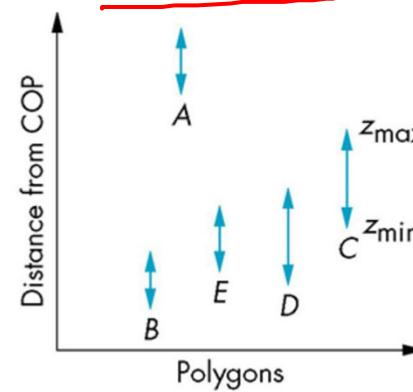
Drawing

Hidden Surface Removal (Object Space)

Depth Sort: Requires ordering of polygons first

- $O(n \log n)$ calculation for ordering
- Not every polygon is either in front or behind all other polygons
- Order polygons and deal with easy cases first, harder later

Polygons sorted by distance from COP



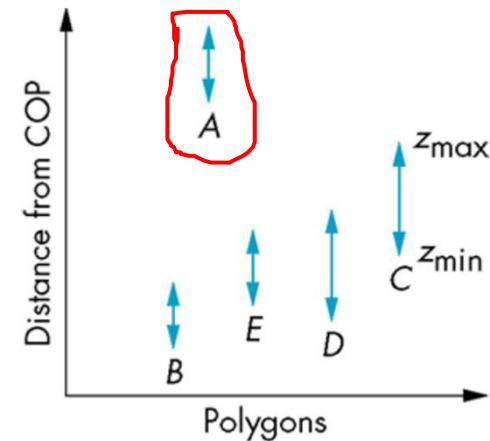
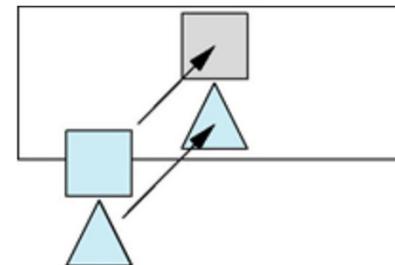
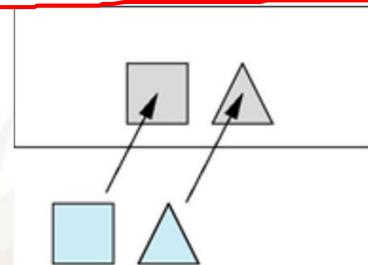
Drawing

Hidden Surface Removal (Object Space)

Depth Sort:

EASY CASES

- A lies behind all other polygons
 - Can render
- Polygons overlap in z but not in either x or y
 - Can render independently

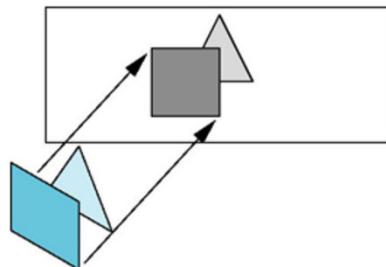


Drawing

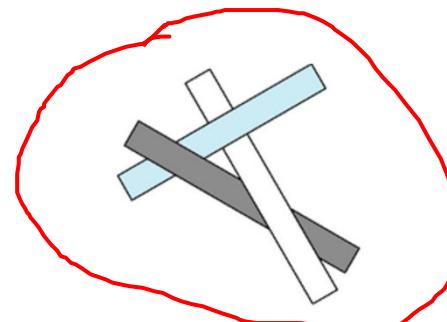
Hidden Surface Removal (Object Space)

Depth Sort:

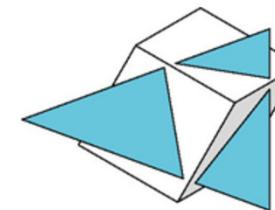
HARD CASES



Overlap in all directions
but can one is fully on
one side of the other



cyclic overlap



penetration

Drawing

Hidden Surface Removal (Object Space)

Painter's Algorithm:

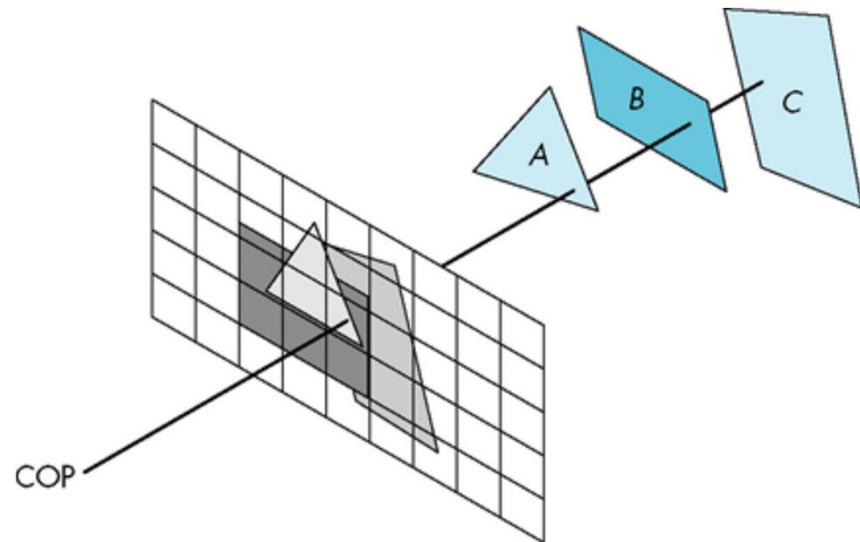
- **Pros**
 - Simple (most of the time)
 - Can handle transparency well
 - In special cases you don't have to sort e.g. heightfield
- **Cons**
 - Cannot handle complex geometry well
 - Sorting can be expensive
- **OpenGL does not provide an implementation**
 - **Must be implemented by the programmer**

Drawing

Hidden Surface Removal (Image Space)

Image-Space Approach:

- Look at each projector (nm for an $n \times m$ frame buffer) and find closest of k polygons
- Complexity $O(nmk)$
- Ray tracing
- z-buffer

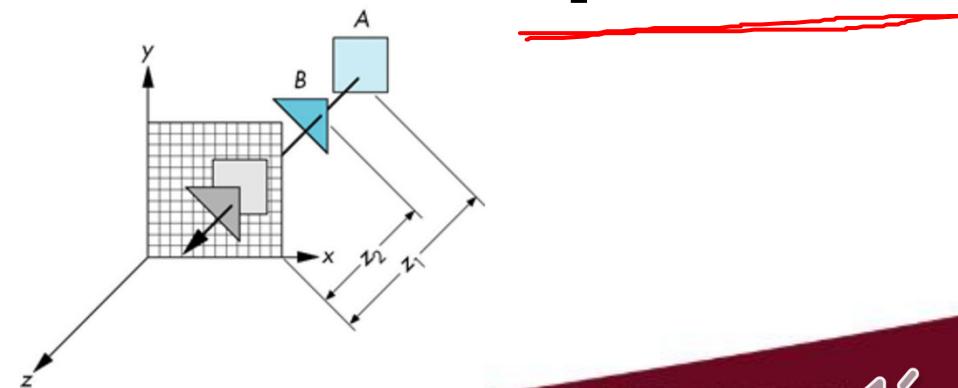


Drawing

Hidden Surface Removal (Image Space)

Z-Buffer Algorithm:

- Use a buffer called the z or depth buffer to store the depth of the closest object at each pixel found so far
- As we render each polygon, compare the depth of each pixel to depth in z buffer
- If less, place shade of pixel in color buffer and update z-buffer



Drawing

Hidden Surface Removal (Image Space)

Z-Buffer Algorithm:

- Pros
 - Simple (no sorting or splitting)
 - Independent of geometric primitives
- Cons
 - Memory intensive (but memory is cheap now)
 - Tricky to handle transparency and blending
 - Depth-ordering artifacts
- OpenGL implements z-Buffering
 - Must be enabled

Drawing

z-Buffer in GLFW and OpenGL

- Ask for a depth buffer when you create your window
 - glfwWindowHint(GL_DEPTH_BITS, 24); //Default value
- Call to glEnable(GL_DEPTH_TEST) in your program's initialization routine, after a context is created and made current
- Set the depth function using glDepthFunc(GL_LESS);
- Every time you draw call glClear(GL_DEPTH_BUFFER_BIT)
 - typically bitwise OR'd with other values e.g.
 - glClear(GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT);

Review

- Input and Interaction**
 - Clients and Servers, Input Modes**
- Drawing**
 - Double Buffering, Hidden Surface Removal**

Next Lecture

Transformations



COMP 371

Computer Graphics

Session 4
GEOMETRY AND
TRANSFORMATIONS



Lecture Overview

- **Geometry and Transformations**
 - Introduce the elements of geometry
 - Scalars, Vectors, Points
- **Coordinate Systems**
 - Homogeneous Coordinates

Transformations

Basic Elements

- **Geometry is the study of relationships among the objects in an n-dimensional space**
 - In computer graphics, we are interested in objects that exist in three dimensions
- **Want a minimum set of primitives from which we can build more sophisticated objects**
 - We will need three basic elements
 - Scalars, Vectors, Points

Transformations

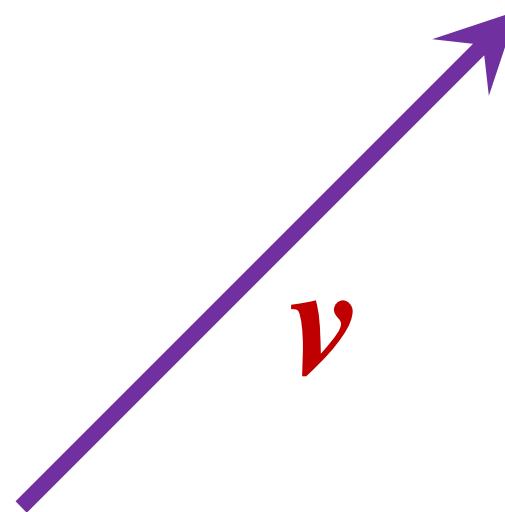
Scalars

- Scalars can be defined as members of sets which can be combined by two operations (addition and multiplication) obeying some fundamental axioms (associativity, commutativity, inverses)
- Examples include the real and complex number systems under the ordinary rules with which we are familiar
- Scalars alone have no geometric properties
- Scalars a, b, c form a **scalar field**
- Operations $a+b, a.b, 0, 1, -a, ()^{-1}$

Transformations

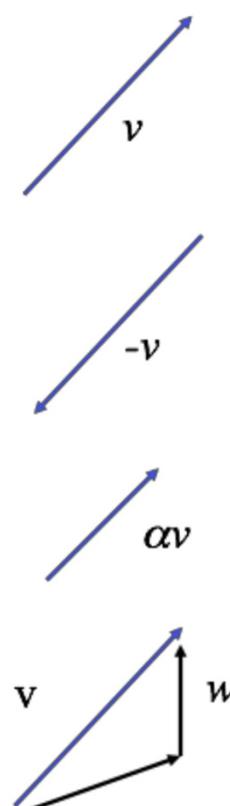
Vectors

- **Physical definition:** a vector is a quantity with two attributes
 - Direction
 - Magnitude
- Form a vector space
- Examples include
 - Force
 - Velocity
 - Directed line segments
 - Most important example for graphics
 - Can map to other types



Transformations

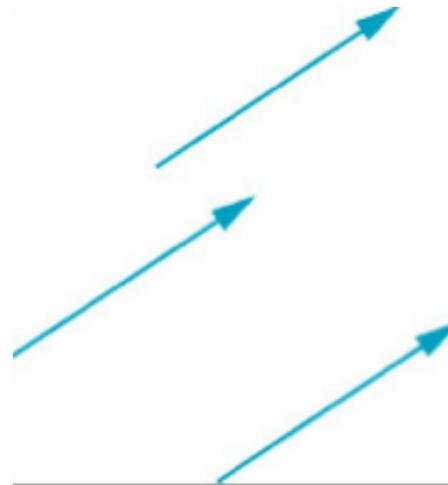
Vector Operations

- Every vector has an inverse
 - Same magnitude but points in opposite direction
 - Every vector can be multiplied by a scalar
 - There is a zero vector
 - Zero magnitude, undefined orientation
 - The sum of any two vectors is a vector
 - Use head-to-tail axiom
 - The magnitude of a vector a is the scalar $\|a\|$
 - A unit vector is any vector with magnitude one
- 

Transformations

Vectors Lack Position

- These vectors are identical
 - Same direction and magnitude

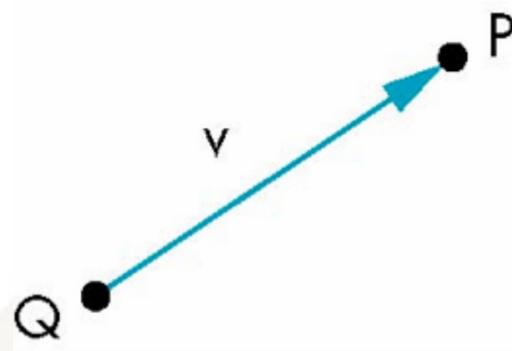


- Vector spaces insufficient for geometry
 - Need points

Transformations

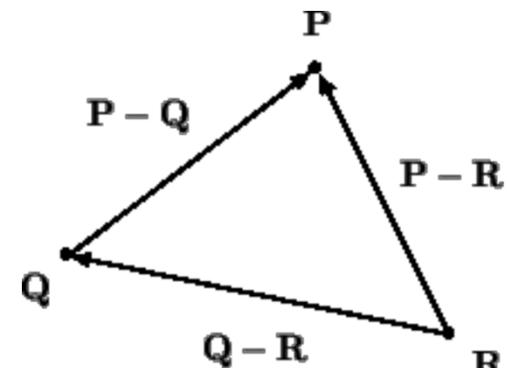
Points

- Location in space
- Operations allowed between points and vectors
 - Point-point subtraction yields a vector
 - Equivalent to point-vector addition



$$v = P - Q$$

$$P = v + Q$$

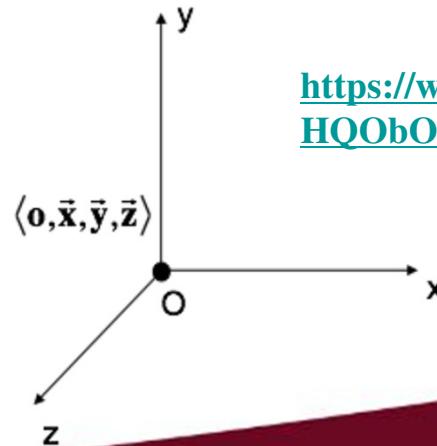
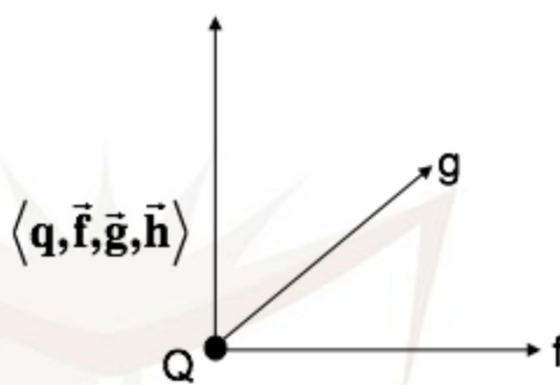


We can interpret vectors as *displacements*, instructions to get from one point in space to another

Transformations

Coordinate Systems

- Every vector can be multiplied by a scalar to scale the vector's magnitude without changing its direction
- $\|k \mathbf{a}\| = k \|\mathbf{a}\|$
- In 2D, we can represent a vector as a linear combination, or weighted sum, of any two non-parallel basis vectors
- In 3D, requires three non-parallel, non coplanar basis vectors



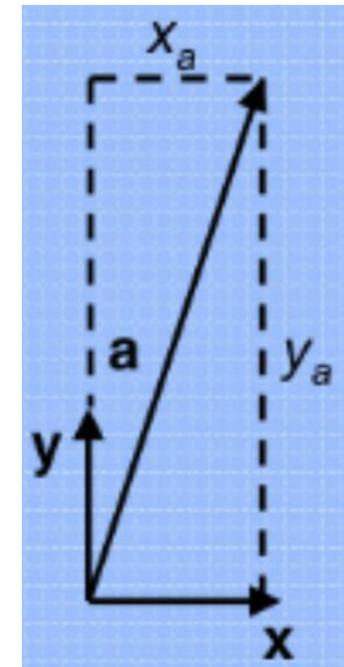
https://www.youtube.com/playlist?list=PLZHQObOWTQDPD3MizzM2xVFitgF8hE_ab

Transformations

Coordinate Systems

- Basis vectors that are unit vectors at right angles to each other are called **orthonormal**
- The x-y Cartesian coordinate system is a **special orthonormal system**
- Vectors are commonly represented in terms of their **Cartesian coordinates**

$$\mathbf{a} = (X_a, Y_a) \quad \mathbf{a}^T = [X_a \; Y_a]$$

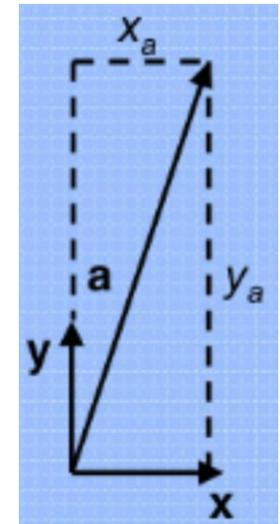


Transformations

Coordinate Systems

- Vectors expressed by orthonormal coordinates

$$\mathbf{a} = (\mathbf{X}_a, \mathbf{Y}_a)$$



- have the very useful property that their magnitudes can be calculated according to the Pythagorean Theorem

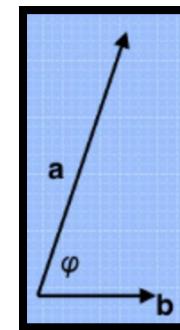
$$\| \mathbf{a} \| = \sqrt{(\mathbf{X}_a^2 + \mathbf{Y}_a^2)}$$

Transformations

Dot Product

- We can multiply two vectors by taking the **dot (or inner) product**
- The dot product is defined as

$$\mathbf{a} \cdot \mathbf{b} = \| \mathbf{a} \| \| \mathbf{b} \| \cos\varphi$$



where φ is the angle between the two vectors

- Note that the dot product takes two vectors as arguments, but is often called the **scalar product** because its result is a scalar

$$\mathbf{A} \cdot \mathbf{B} = \sum_{i=1}^n A_i B_i = A_1 B_1 + A_2 B_2 + \cdots + A_n B_n$$

Transformations

Dot Product Properties

- It's often useful in graphics to know the cosine of the angle between two vectors, and we can find it with the dot product

$$\cos \varphi = \mathbf{a} \cdot \mathbf{b} / (\|\mathbf{a}\| \|\mathbf{b}\|)$$

- We can use the dot product to find the **projection** of one vector onto another. The scalar $\mathbf{a} \rightarrow \mathbf{b}$ is the magnitude of the vector \mathbf{a} projected at a right angle onto vector \mathbf{b} , and

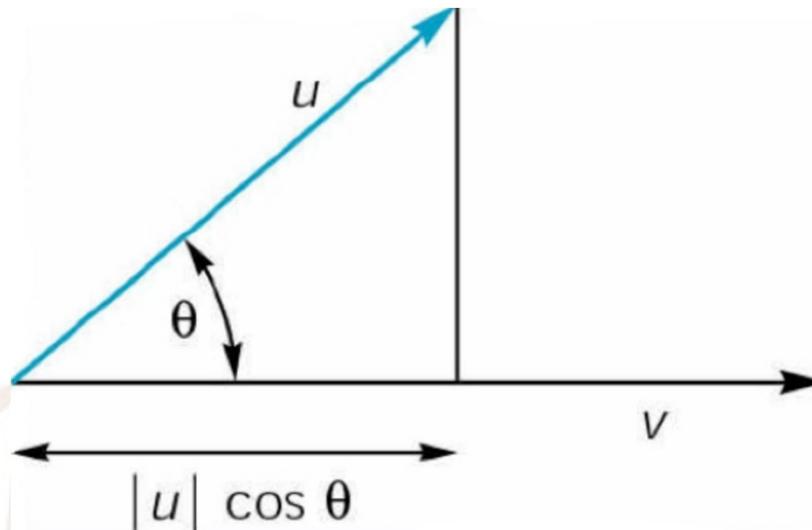
$$\mathbf{a} \rightarrow \mathbf{b} = \|\mathbf{a}\| \cos \varphi = \mathbf{a} \cdot \mathbf{b} / \|\mathbf{b}\|$$

- Dot products are commutative and distributive:
 - $\mathbf{a} \cdot \mathbf{b} = \mathbf{b} \cdot \mathbf{a}$
 - $\mathbf{a} \cdot (\mathbf{b} + \mathbf{c}) = \mathbf{a} \cdot \mathbf{b} + \mathbf{a} \cdot \mathbf{c}$
 - $(k\mathbf{a}) \cdot \mathbf{b} = \mathbf{a} \cdot (k\mathbf{b}) = k(\mathbf{a} \cdot \mathbf{b})$

Transformations

Dot Product – Projection Example

- Dot product projects one vector onto another vector
- $\mathbf{u} \cdot \mathbf{v} = u_1 v_1 + u_2 v_2 + u_3 v_3 = \| \mathbf{u} \| \| \mathbf{v} \| \cos(\theta)$
- Projection_v $\mathbf{u} = (\mathbf{u} \cdot \mathbf{v}) \mathbf{v} / \| \mathbf{v} \|^2$

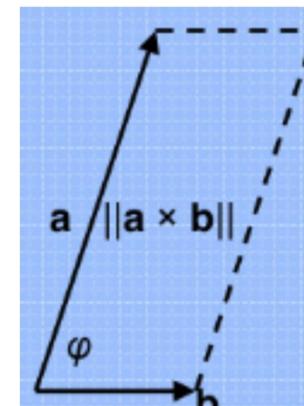


Transformations

Cross Product

- The cross product is another vector multiplication operation, usually used only for 3D vectors
- The direction of $\mathbf{a} \times \mathbf{b}$ is orthogonal to both \mathbf{a} and \mathbf{b}
- The magnitude is equal to the area of parallelogram formed by the two vectors. It is given by

$$\| \mathbf{a} \times \mathbf{b} \| = \| \mathbf{a} \| \| \mathbf{b} \| \sin \varphi$$



Transformations

Cross Product Properties

- Cross products are distributive
 - $\mathbf{a} \times (\mathbf{b} + \mathbf{c}) = \mathbf{a} \times \mathbf{b} + \mathbf{a} \times \mathbf{c}$
 - $(k\mathbf{a}) \times \mathbf{b} = \mathbf{a} \times (k\mathbf{b}) = k(\mathbf{a} \times \mathbf{b})$
- Cross products are intransitive; in fact,
 - $\mathbf{a} \times \mathbf{b} = -\mathbf{b} \times \mathbf{a}$
- For all \mathbf{a} , $\mathbf{a} \times \mathbf{a} = \mathbf{0}$
 - because of the sin in the magnitude calculation
- In x-y-z Cartesian space,
 - $\mathbf{x} \times \mathbf{y} = \mathbf{z}$
 - $\mathbf{y} \times \mathbf{z} = \mathbf{x}$
 - $\mathbf{z} \times \mathbf{x} = \mathbf{y}$

$$\mathbf{x} \times \mathbf{y} = \mathbf{z} \quad \mathbf{y} \times \mathbf{z} = \mathbf{x} \quad \mathbf{z} \times \mathbf{x} = \mathbf{y}$$

Transformations

Cross Product Calculation

$$\mathbf{i} = \mathbf{j} \times \mathbf{k} \quad \mathbf{k} \times \mathbf{j} = -\mathbf{i} \quad \mathbf{i} \times \mathbf{i} = \mathbf{j} \times \mathbf{j} = \mathbf{k} \times \mathbf{k} = \mathbf{0}$$

$$\mathbf{j} = \mathbf{k} \times \mathbf{i} \quad \mathbf{i} \times \mathbf{k} = -\mathbf{j} \quad \mathbf{u} = u_1\mathbf{i} + u_2\mathbf{j} + u_3\mathbf{k}$$

$$\mathbf{k} = \mathbf{i} \times \mathbf{j} \quad \mathbf{j} \times \mathbf{i} = -\mathbf{k} \quad \mathbf{v} = v_1\mathbf{i} + v_2\mathbf{j} + v_3\mathbf{k}$$

$$\begin{aligned}\mathbf{u} \times \mathbf{v} &= (u_1\mathbf{i} + u_2\mathbf{j} + u_3\mathbf{k}) \times (v_1\mathbf{i} + v_2\mathbf{j} + v_3\mathbf{k}) \\&= u_1v_1(\mathbf{i} \times \mathbf{i}) + u_1v_2(\mathbf{i} \times \mathbf{j}) + u_1v_3(\mathbf{i} \times \mathbf{k}) + \\&\quad u_2v_1(\mathbf{j} \times \mathbf{i}) + u_2v_2(\mathbf{j} \times \mathbf{j}) + u_2v_3(\mathbf{j} \times \mathbf{k}) + \\&\quad u_3v_1(\mathbf{k} \times \mathbf{i}) + u_3v_2(\mathbf{k} \times \mathbf{j}) + u_3v_3(\mathbf{k} \times \mathbf{k})\end{aligned}$$

Transformations

Cross Product Calculation

$$\begin{aligned}\mathbf{u} \times \mathbf{v} &= u_1 v_1 \mathbf{0} + u_1 v_2 \mathbf{k} - u_1 v_3 \mathbf{j} - \\&\quad u_2 v_1 \mathbf{k} - u_2 v_2 \mathbf{0} + u_2 v_3 \mathbf{i} + \\&\quad u_3 v_1 \mathbf{j} - u_3 v_2 \mathbf{i} - u_3 v_3 \mathbf{0} \\&= (u_2 v_3 - u_3 v_2) \mathbf{i} + (u_3 v_1 - u_1 v_3) \mathbf{j} + (u_1 v_2 - u_2 v_1) \mathbf{k}\end{aligned}$$

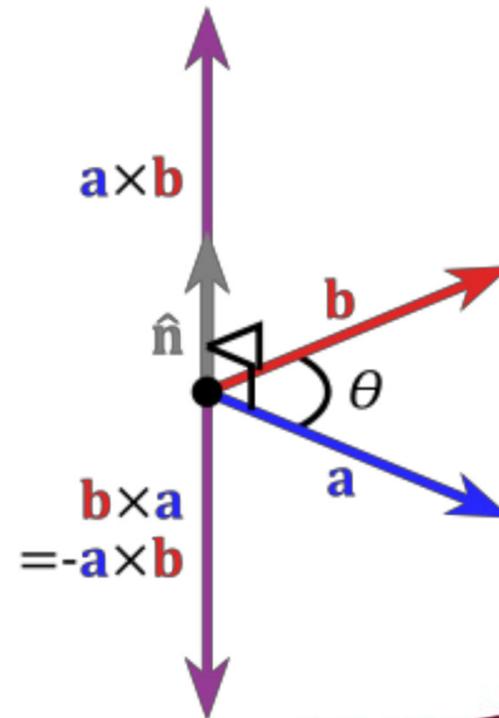
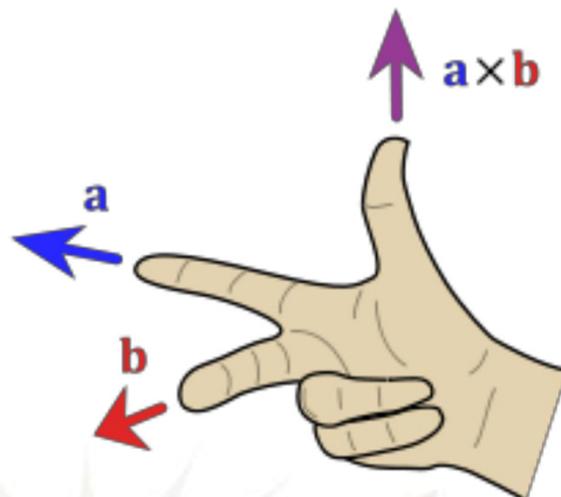
Resulting vector $\mathbf{s} = s_1 \mathbf{i} + s_2 \mathbf{j} + s_3 \mathbf{k} = \mathbf{u} \times \mathbf{v}$

$$\begin{pmatrix} s_1 \\ s_2 \\ s_3 \end{pmatrix} = \begin{pmatrix} u_2 v_3 - u_3 v_2 \\ u_3 v_1 - u_1 v_3 \\ u_1 v_2 - u_2 v_1 \end{pmatrix}$$

Transformations

Cross Product

Right-hand rule determines the direction of the orthonormal vector



Transformations

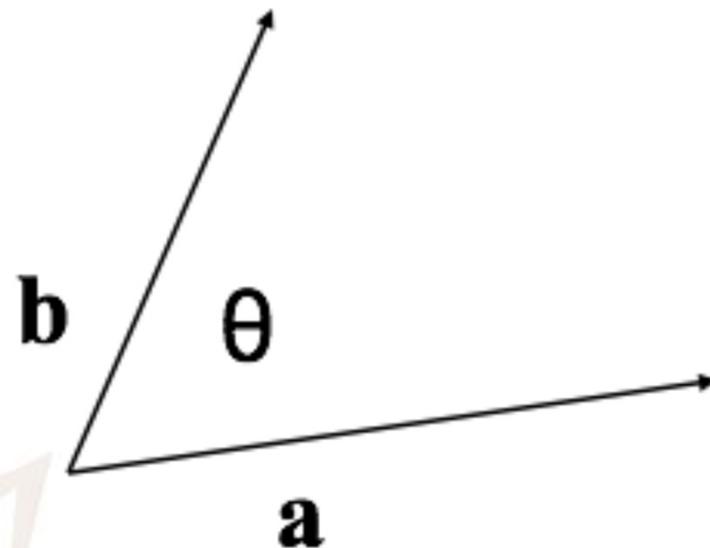
Normal Vectors

- A **normal vector** is a vector perpendicular to a surface. A **unit normal** is a **normal vector of magnitude one**
- **Normal vectors are important to many graphics calculations**
- If the surface is a polygon containing the points a , b , c , one normal vector $\mathbf{n} = (\mathbf{b} - \mathbf{a}) \times (\mathbf{c} - \mathbf{a})$
- This vector points into the polygon if a , b , and c are arranged **clockwise**; it points outward if they are arranged **counterclockwise**

Transformations

Example: Angle between vectors

- How do you find the angle θ between vectors a and b ?



Transformations

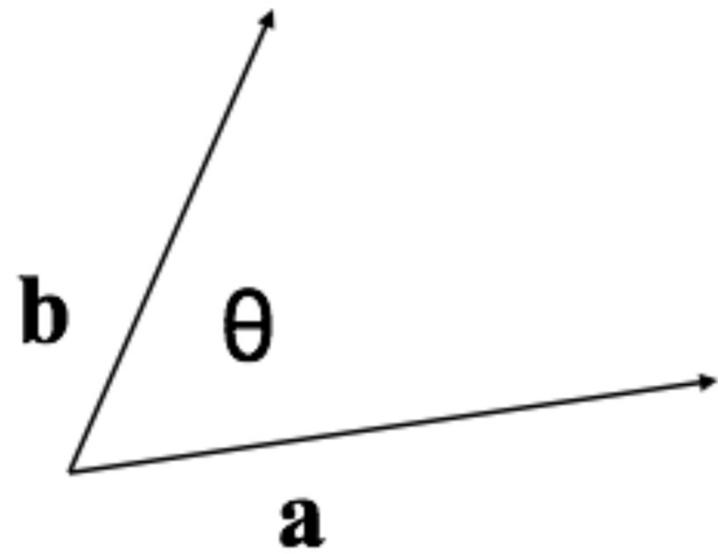
Example: Angle between vectors

- How do you find the angle θ between vectors \mathbf{a} and \mathbf{b} ?

$$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}| |\mathbf{b}| \cos \theta$$

$$\cos \theta = \left(\frac{\mathbf{a} \cdot \mathbf{b}}{|\mathbf{a}| |\mathbf{b}|} \right)$$

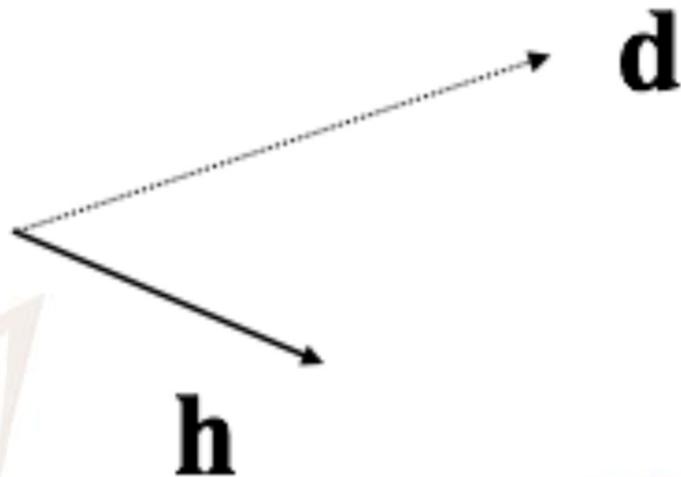
$$\theta = \cos^{-1} \left(\frac{\mathbf{a} \cdot \mathbf{b}}{|\mathbf{a}| |\mathbf{b}|} \right)$$



Transformations

Example: Align Two vectors

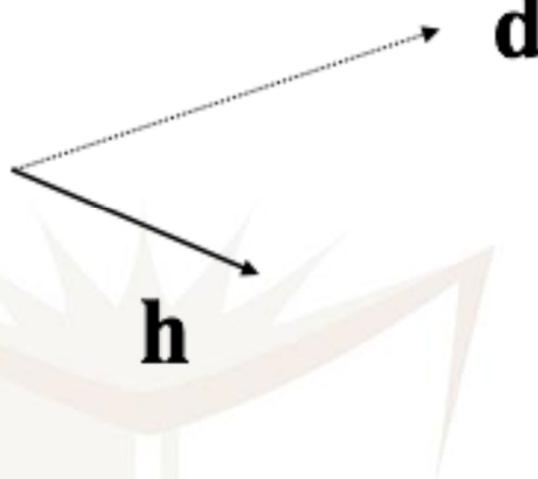
- We are heading in direction **h**. We want to rotate so that we will align with a different direction **d**. Find a unit axis **a** and an angle θ to rotate around.



Transformations

Example: Align Two vectors

- We are heading in direction \mathbf{h} . We want to rotate so that we will align with a different direction \mathbf{d} . Find a unit axis \mathbf{a} and an angle θ to rotate around.



$$\mathbf{a} = \frac{\mathbf{h} \times \mathbf{d}}{|\mathbf{h} \times \mathbf{d}|}$$

$$\theta = \sin^{-1} \left(\frac{|\mathbf{h} \times \mathbf{d}|}{|\mathbf{h}| |\mathbf{d}|} \right)$$

$$\theta = \cos^{-1} \left(\frac{\mathbf{h} \cdot \mathbf{d}}{|\mathbf{h}| |\mathbf{d}|} \right)$$

$$\theta = \tan^{-1} \left(\frac{|\mathbf{h} \times \mathbf{d}|}{\mathbf{h} \cdot \mathbf{d}} \right)$$

theta = atan2(| $\mathbf{h} \times \mathbf{d}$ |, $\mathbf{h} \cdot \mathbf{d}$)

Transformations

Sample Vector Class in C++

```
#include <vector>
#include <iostream>
int main()
{
5.    std::vector<int> q;
        q.push_back(10); q.push_back(11); q.push_back(12);

        std::vector<int> v;
        for(int i=0; i<5; ++i){
10.       v.push_back(i);
    }
    // v contains 0 1 2 3 4

    std::vector<int>::iterator it = v.begin() + 1;
15.    // insert 33 before the second element:
        it = v.insert(it, 33);
        // v contains 0 33 1 2 3 4
        // it points to the inserted element
20.    //insert the contents of q before the second element:
        v.insert(it, q.begin(), q.end());
        // v contains 0 10 11 12 33 1 2 3 4
        // iterator 'it' is invalid
```

```
25.    it = v.begin() + 3;
        // it points to the fourth element of v
        // insert three time -1 before the fourth element:
        v.insert(it, 3, -1);
        // v contains 0 10 11 -1 -1 -1 12 33 1 2 3 4
30.    // iterator 'it' is invalid
        // erase the fifth element of v
        it = v.begin() + 4;
        v.erase(it);
        // v contains 0 10 11 -1 -1 12 33 1 2 3 4
35.    // iterator 'it' is invalid
        // erase the second to the fifth element:
        it = v.begin() + 1;
40.    v.erase(it, it + 4);
        // v contains 0 12 33 1 2 3 4
        // iterator 'it' is invalid
        // clear all of v's elements
45.    v.clear();

        return 0;
}
```

Transformations

Matrix

- A rectangular array of numbers

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \in \mathbb{R}^{m \times n}$$

- Square matrix if $m = n$
- In graphics often $m = n = 3$ and $m = n = 4$

Transformations

Matrix Addition

$$\begin{aligned}\mathbf{A} + \mathbf{B} &= \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{bmatrix} \\ &= \begin{bmatrix} a_{11} + b_{11} & a_{12} + b_{12} & \cdots & a_{1n} + b_{1n} \\ a_{21} + b_{21} & a_{22} + b_{22} & \cdots & a_{2n} + b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} + b_{m1} & a_{m2} + b_{m2} & \cdots & a_{mn} + b_{mn} \end{bmatrix}\end{aligned}$$

Transformations

Matrix Multiplication With a Scalar

$$s\mathbf{M} = \mathbf{M}s = \begin{bmatrix} sm_{1,1} & sm_{1,2} & \dots & sm_{1,n} \\ sm_{2,1} & sm_{2,2} & \dots & sm_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ sm_{m,1} & sm_{2,2} & \dots & sm_{m,n} \end{bmatrix}$$

Transformations

Matrix Multiplication

$$\mathbf{AB} = \mathbf{C}, \quad \mathbf{A} \in \mathbf{R}^{p,q}, \mathbf{B} \in \mathbf{R}^{q,r}, \mathbf{C} \in \mathbf{R}^{p,r}$$

$$(\mathbf{AB})_{i,j} = \mathbf{C}_{i,j} = \sum_{k=1}^q a_{i,k} b_{k,j}, \quad i \in 1..p, j \in 1..r$$

Transformations

Matrix-Vector Multiplication

$$\mathbf{Ax} = \mathbf{y}, \quad \mathbf{A} \in \mathbf{R}^{p,q}, \mathbf{x} \in \mathbf{R}^q, \mathbf{y} \in \mathbf{R}^p$$

$$(\mathbf{Ax})_i = \mathbf{y}_i = \sum_{k=1}^q a_{i,k}x_k$$

Transformations

Identity Matrix

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix} \in \mathbf{R}^{n \times n}$$

$$\mathbf{M}\mathbf{I} = \mathbf{IM} = \mathbf{M}, \quad \text{for any } \mathbf{M} \in \mathbf{R}^{n \times n}$$

Transformations

Matrix Inverse

- If a square matrix M is non-singular, there exists a unique inverse M^{-1} such that

$$MM^{-1} = M^{-1}M = I$$

$$(MPQ)^{-1} = Q^{-1}P^{-1}M^{-1}$$

Transformations

Matrix Inverse

- Vectors are column vectors
- “Column major” ordering
- Matrix elements stored in array of floats
 - float M[16];
 - Corresponding matrix elements:

$$\begin{bmatrix} m[0] & m[4] & m[8] & m[12] \\ m[1] & m[5] & m[9] & m[13] \\ m[2] & m[6] & m[10] & m[14] \\ m[3] & m[7] & m[11] & m[15] \end{bmatrix}$$

Transformations

Matrix Inverse

```
#include <iostream>
#include <random>

class CMatrix {
private:
    int* m_ptValues;
    int m_totalSize;
    int m_rows;
    int m_columns;
public:
    CMatrix(int rows, int columns); // base ctor.
    CMatrix(const CMatrix& rhs); // copy ctor.
    CMatrix& operator=(const CMatrix& rhs); // assign. ctor.
    ~CMatrix(); // dtor.
    int& operator()(int row, int column);
    int& operator()(int row, int column) const;
    CMatrix& operator+=(int scalar);
    CMatrix operator+(int scalar);
    CMatrix& operator-=(int scalar);
    CMatrix operator-(int scalar);
    CMatrix& operator*=(int scalar);
    CMatrix operator*(int scalar);
    CMatrix& operator*=(const CMatrix& rhs);
    CMatrix operator*(const CMatrix& rhs);
    CMatrix& operator+=(const CMatrix& rhs);
    CMatrix operator+(const CMatrix& rhs);
    void reshape(int newRows, int newColumns);
    void show(); //used for dev. only
    void range(int start, int defaultStep=1);
    void fill(int value);
    void randint(int lowerBound, int upperBound);
};
```

Transformations

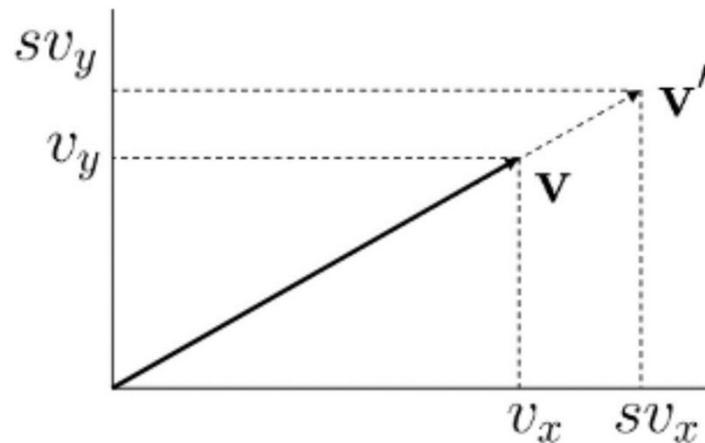
Linear Transformations

- **Scaling, shearing, rotation, reflection of vectors, and combinations thereof**
- **Implemented using matrix multiplications**

Transformations

Scaling

- Uniform scaling matrix in 2D



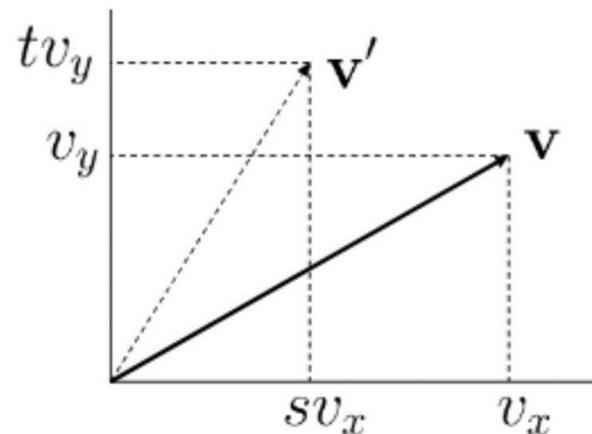
- Analogous in 3D

$$\begin{bmatrix} s & 0 \\ 0 & s \end{bmatrix} \mathbf{v} = \begin{bmatrix} s & 0 \\ 0 & s \end{bmatrix} \begin{bmatrix} v_x \\ v_y \end{bmatrix} = \begin{bmatrix} v'_x \\ v'_y \end{bmatrix} = \mathbf{v}'$$

Transformations

Scaling

- Non-Uniform scaling



- Analogous in 3D

$$\begin{bmatrix} s & 0 \\ 0 & t \end{bmatrix} \mathbf{v} = \begin{bmatrix} s & 0 \\ 0 & t \end{bmatrix} \begin{bmatrix} v_x \\ v_y \end{bmatrix} = \begin{bmatrix} v'_x \\ v'_y \end{bmatrix} = \mathbf{v}'$$

Transformations

Shearing

- Shearing along X-axis in 2D

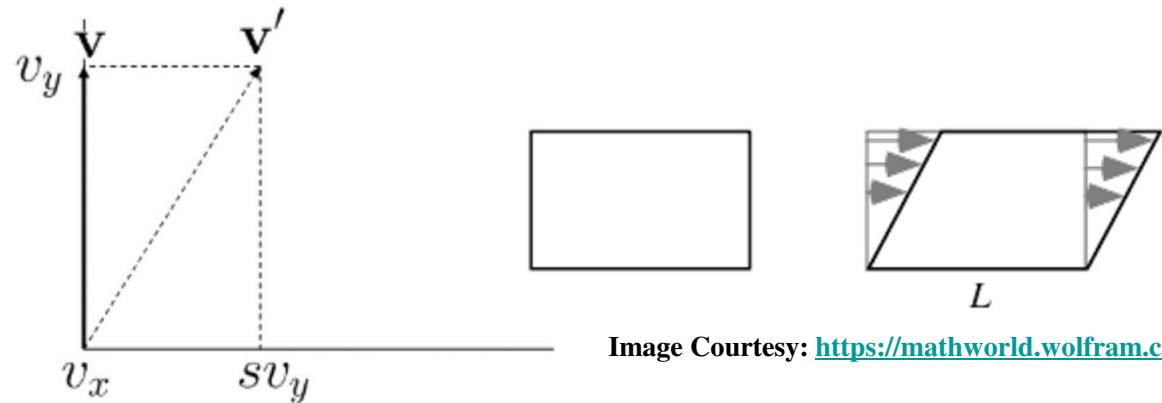


Image Courtesy: <https://mathworld.wolfram.com/>

- Analogous in 3D

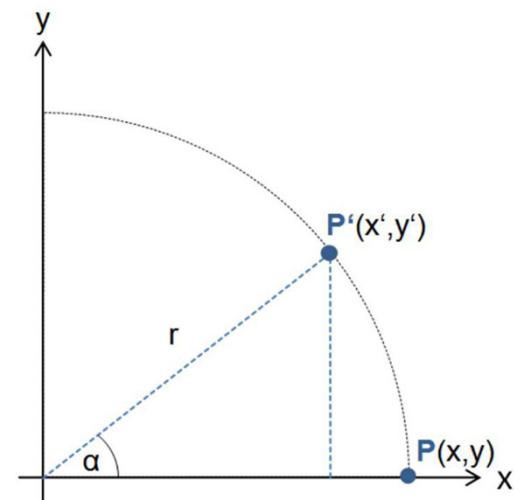
$$\mathbf{v}' = \begin{bmatrix} 1 & s \\ 0 & 1 \end{bmatrix} \mathbf{v}$$

Transformations

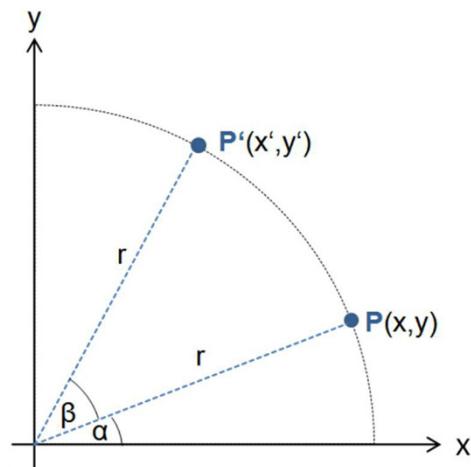
Rotation (2D)

- Convention: positive angles rotate counterclockwise
- Rotation matrix

$$\mathbf{R}(\theta) = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix}$$



$$\begin{aligned}\cos \alpha &= \frac{x'}{r} \rightarrow x' = r * \cos \alpha = x * \cos \alpha \\ \sin \alpha &= \frac{y'}{r} \rightarrow y' = r * \sin \alpha = x * \sin \alpha\end{aligned}$$

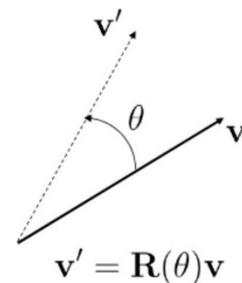


$$\begin{aligned}x &= r * \cos \alpha \\ y &= r * \sin \alpha\end{aligned}$$

$$\begin{aligned}x' &= r * \cos(\alpha + \beta) \\ y' &= r * \sin(\alpha + \beta)\end{aligned}$$

$$\begin{aligned}x' &= r * \cos(\alpha + \beta) \\ &= r * (\cos \alpha * \cos \beta - \sin \alpha * \sin \beta) \\ &= r * \cos \alpha \cos \beta - r * \sin \alpha \sin \beta \\ &= x * \cos \beta - y * \sin \beta\end{aligned}$$

$$\begin{aligned}y' &= r * \sin(\alpha + \beta) \\ &= r * (\sin \alpha * \cos \beta + \cos \alpha * \sin \beta) \\ &= r * \sin \alpha \cos \beta - r * \cos \alpha \sin \beta \\ &= y * \cos \beta + x * \sin \beta\end{aligned}$$



$$v' = \mathbf{R}(\theta)v$$

Transformations

Rotation (3D)

- Rotation around coordinate axes

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

X component remains unchanged

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

Y component remains unchanged

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

Z component remains unchanged

Transformations

Rotation (3D)

- Concatenation of rotations around x, y, z axes

$$\mathbf{R}_{x,y,z}(\theta_x, \theta_y, \theta_z) = \mathbf{R}_x(\theta_x)\mathbf{R}_y(\theta_y)\mathbf{R}_z(\theta_z)$$

Euler angles

- Result depends on matrix order

$$\mathbf{R}_x(\theta_x)\mathbf{R}_y(\theta_y)\mathbf{R}_z(\theta_z) \neq \mathbf{R}_z(\theta_z)\mathbf{R}_y(\theta_y)\mathbf{R}_x(\theta_x)$$

Transformations

Rotation (3D)

- Around arbitrary axis

$$\mathbf{R}(\mathbf{a}, \theta) = \begin{bmatrix} 1 + (1 - \cos(\theta))(a_x^2 - 1) & -a_z \sin(\theta) + (1 - \cos(\theta))a_x a_y & a_y \sin(\theta) + (1 - \cos(\theta))a_x a_z \\ a_z \sin(\theta) + (1 - \cos(\theta))a_y a_x & 1 + (1 - \cos(\theta))(a_y^2 - 1) & -a_x \sin(\theta) + (1 - \cos(\theta))a_y a_z \\ -a_y \sin(\theta) + (1 - \cos(\theta))a_z a_x & a_x \sin(\theta) + (1 - \cos(\theta))a_z a_y & 1 + (1 - \cos(\theta))(a_z^2 - 1) \end{bmatrix}$$

- Rotation axis \mathbf{a}
 - \mathbf{a} must be a unit vector: $|\mathbf{a}| = 1$
- Right-hand rule applies for direction of rotation
 - Counterclockwise rotation

Transformations

Homogeneous Coordinates

- We add a 4th coordinate to vectors and points
- Usually for 3D points we choose $w = 1$
- For 3D vectors $w = 0$
- Benefit of homogeneous coordinates:
 - vectors and points both represented by 4 coordinates

Transformations

Homogeneous Coordinates

- First consider the following linear transformations

M	Matrix-vector multiplication	Matrix-point multiplication
	$\mathbf{M}\vec{v} = \begin{bmatrix} m_{xx} & m_{xy} & m_{xz} \\ m_{yx} & m_{yy} & m_{yz} \\ m_{zx} & m_{zy} & m_{zz} \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}$	$\mathbf{Mp} = \begin{bmatrix} m_{xx} & m_{xy} & m_{xz} \\ m_{yx} & m_{yy} & m_{yz} \\ m_{zx} & m_{zy} & m_{zz} \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix}$
Rotation matrix	Rotates vector direction	Moves point by rotating <i>about origin of frame</i>
Scale matrix	Scales vector magnitude	Moves point <i>towards or away from origin of frame</i>

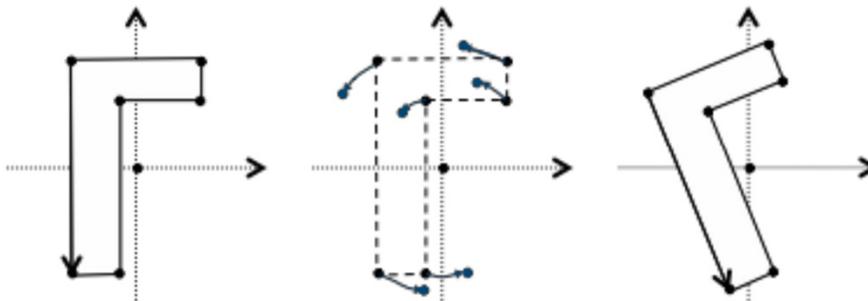
Transformations

Rotating an object

- Object defined as collection of points $\mathbf{p}_1 \dots \mathbf{p}_N$
- Apply rotation matrix to every point:

$$\mathbf{p}'_i = \mathbf{R} \mathbf{p}_i, \text{ for all } i$$

- Rotates object about origin of frame



- Also rotates all vectors in object:

$$\vec{\mathbf{v}} = \mathbf{p}_2 - \mathbf{p}_1$$

$$\vec{\mathbf{v}}' = \mathbf{p}'_2 - \mathbf{p}'_1 = (\mathbf{R}\mathbf{p}_2) - (\mathbf{R}\mathbf{p}_1) = \mathbf{R}(\mathbf{p}_2 - \mathbf{p}_1) = \mathbf{R}\vec{\mathbf{v}}$$

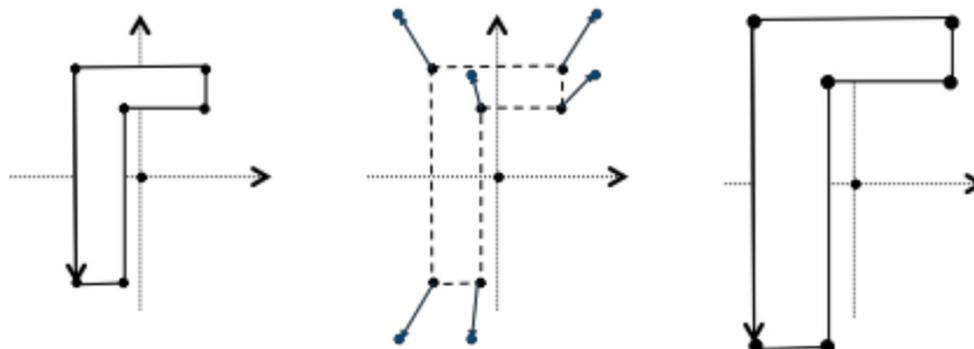
Transformations

Scaling an object

- Apply scale matrix to every point:

$$\mathbf{p}'_i = \mathbf{S} \mathbf{p}_i, \text{ for all } i$$

- Scale object about origin of frame



- Also scales all vectors in object

$$\vec{v} = \mathbf{p}_2 - \mathbf{p}_1$$

$$\vec{v}' = \mathbf{p}'_2 - \mathbf{p}'_1 = (\mathbf{S}\mathbf{p}_2) - (\mathbf{S}\mathbf{p}_1) = \mathbf{S}(\mathbf{p}_2 - \mathbf{p}_1) = \mathbf{S}\vec{v}$$

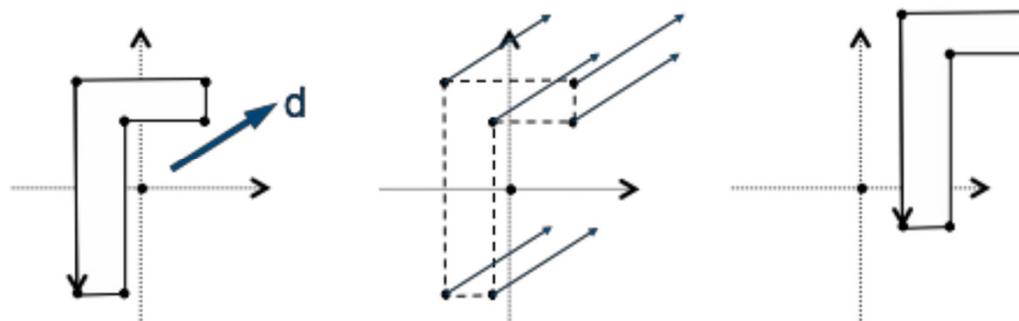
Transformations

Moving [Translating] an object

- Add displacement vector to each point:

$$\mathbf{p}'_i = \mathbf{p}_i + \vec{\mathbf{d}}, \quad \text{for all } i$$

- *Translates* the object:



- *Vectors don't change:*

$$\vec{\mathbf{v}} = \mathbf{p}_2 - \mathbf{p}_1$$

$$\vec{\mathbf{v}}' = \mathbf{p}'_2 - \mathbf{p}'_1 = (\mathbf{p}_2 + \vec{\mathbf{d}}) - (\mathbf{p}_1 + \vec{\mathbf{d}}) = (\mathbf{p}_2 - \mathbf{p}_1) + (\vec{\mathbf{d}} - \vec{\mathbf{d}}) = \vec{\mathbf{v}} + 0 = \vec{\mathbf{v}}$$

Transformations

General Object Transformation

- Some matrix M and displacement d :
 - $p' = Mp + d$
 - $v' = Mv$
 - Different rule for points vs vectors
- Hard to compose transformations
- Hard to invert, etc.
- ...so introduce Homogeneous Coordinates

Transformations

Homogeneous Coordinates

- **Simple:** a trick to unify/simplify computations
- **Advanced:** projective geometry
 - Interesting mathematical properties
 - Good to know, but less immediately practical
- **We will use some aspect of this when we do perspective projection later on**

Transformations

Homogeneous Coordinates

- Add an extra component: 1 for point, 0 for vector

$$\mathbf{p} = \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} \quad \vec{\mathbf{v}} = \begin{bmatrix} v_x \\ v_y \\ v_z \\ 0 \end{bmatrix}$$

- Combine M and d into single 4x4 matrix:

$$\begin{bmatrix} m_{xx} & m_{xy} & m_{xz} & d_x \\ m_{yx} & m_{yy} & m_{yz} & d_y \\ m_{zx} & m_{zy} & m_{zz} & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Transformations

Homogeneous Coordinates

- Generalized homogeneous point

$$\mathbf{p}_h = wp_x\mathbf{x} + wp_y\mathbf{y} + wp_z\mathbf{z} + w\mathbf{o}$$

$$\begin{bmatrix} wp_x \\ wp_y \\ wp_z \\ w \end{bmatrix}$$

- Obtain 3D coordinates of point

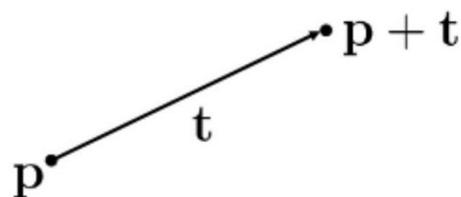
- divide by homogeneous coordinate w

$$\mathbf{p} = p_x\mathbf{x} + p_y\mathbf{y} + p_z\mathbf{z} + \mathbf{o}$$

$$\begin{bmatrix} wp_x/w \\ wp_y/w \\ wp_z/w \\ w/w \end{bmatrix}$$

Transformations

Translation

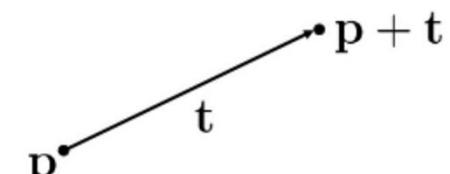


$$\mathbf{p} = \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix}$$

$$\mathbf{t} = \begin{bmatrix} t_x \\ t_y \\ t_z \\ 0 \end{bmatrix}$$

$$\mathbf{p} + \mathbf{t} = \begin{bmatrix} p_x + t_x \\ p_y + t_y \\ p_z + t_z \\ 1 \end{bmatrix}$$

Matrix notation



$$\mathbf{p} + \mathbf{t} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} = \begin{bmatrix} p_x + t_x \\ p_y + t_y \\ p_z + t_z \\ 1 \end{bmatrix}$$

Translation matrix

Transformations

Transformations

- Add a 4th row/column to 3x3 transformation matrices
- Example: rotation

$$R(a, \theta) \in \mathbb{R}^{3 \times 3}$$

$$\begin{bmatrix} R(a, \theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Transformations

Transformations

- Concatenation of transformations:
 - Arbitrary transformations (scale, shear, rotation, translation)

$$\mathbf{M}_3, \mathbf{M}_2, \mathbf{M}_1 \in \mathbf{R}^{4 \times 4}$$

- Build “chains” of transformations
- $\mathbf{p}'_h = \mathbf{M}_3\mathbf{M}_2\mathbf{M}_1\mathbf{p}_h$
- Again: Result depends on order

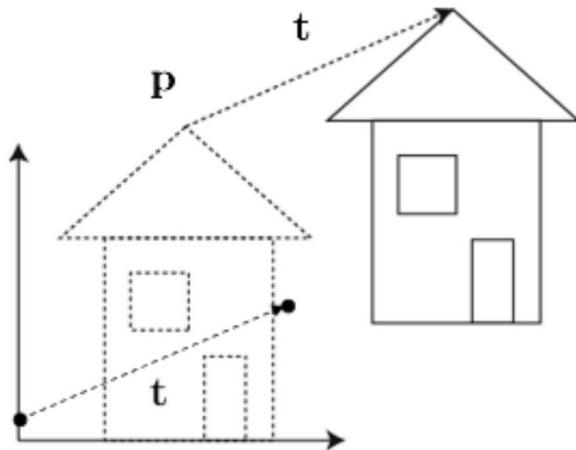
Transformations

Affine Transformations

- **Generalization of linear transformations**
 - Scale, shear, rotation, reflection (linear)
 - Translation
- **Preserve straight lines, parallel lines**
- **Implementation using 4x4 matrices and homogeneous coordinates**

Transformations

Translation



$$\mathbf{p} = \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} \quad \mathbf{t} = \begin{bmatrix} t_x \\ t_y \\ t_z \\ 0 \end{bmatrix}$$

$$\mathbf{p}' = \mathbf{p} + \mathbf{t} = \begin{bmatrix} p_x + t_x \\ p_y + t_y \\ p_z + t_z \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} p'_x \\ p'_y \\ p'_z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix}$$

$$\mathbf{p}' = \mathbf{T}(\mathbf{t})\mathbf{p}$$

Transformations

Inverse Translation

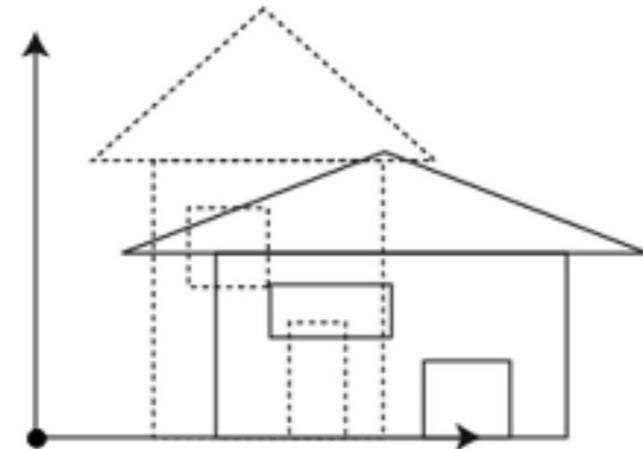
$$\mathbf{T}(\mathbf{t})^{-1} = \mathbf{T}(-\mathbf{t})$$

$$\mathbf{T}(\mathbf{t}) = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{T}(-\mathbf{t}) = \begin{bmatrix} 1 & 0 & 0 & -t_x \\ 0 & 1 & 0 & -t_y \\ 0 & 0 & 1 & -t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Transformations

Scaling

- Origin does not change



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

Transformations

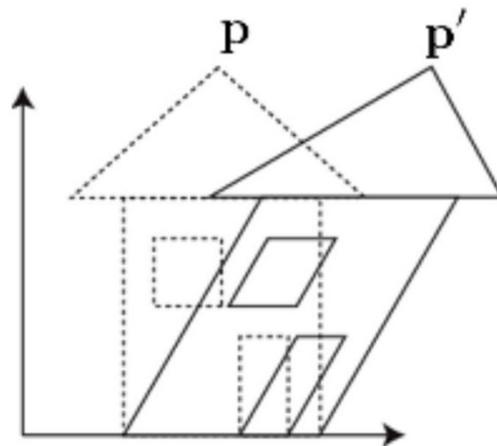
Inverse Scaling

- Origin does not change

$$\mathbf{S}(s_x, s_y, s_z)^{-1} = \mathbf{S}(1/s_x, 1/s_y, 1/s_z)$$

Transformations

Shear



$$\mathbf{p}' = \begin{bmatrix} 1 & z \\ 0 & 1 \end{bmatrix} \mathbf{p}$$

- Pure shear if only one parameter is non-zero

$$\mathbf{Z}(z_1 \dots z_6) = \begin{bmatrix} 1 & z_1 & z_2 & 0 \\ z_3 & 1 & z_4 & 0 \\ z_5 & z_6 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Transformations

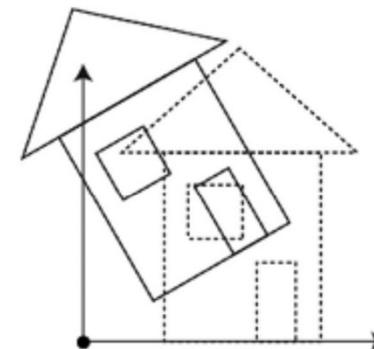
Rotation around coordinate axis

- Origin does not change

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

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

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



Transformations

Rotation around arbitrary axis

- Origin does not change
- Angle θ , unit axis \mathbf{a}

$$c_\theta = \cos \theta, s_\theta = \sin \theta$$

$$\mathbf{R}(\mathbf{a}, \theta) = \begin{bmatrix} a_x^2 + c_\theta(1 - a_x^2) & a_x a_y (1 - c_\theta) - a_z s_\theta & a_x a_z (1 - c_\theta) + a_y s_\theta & 0 \\ a_x a_y (1 - c_\theta) + a_z s_\theta & a_y^2 + c_\theta(1 - a_y^2) & a_y a_z (1 - c_\theta) - a_x s_\theta & 0 \\ a_x a_z (1 - c_\theta) - a_y s_\theta & a_y a_z (1 - c_\theta) + a_x s_\theta & a_z^2 + c_\theta(1 - a_z^2) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Transformations

Rotation Matrix Properties

- **Orthonormal**
 - Rows, columns are unit length and orthogonal
- **Inverse of rotation matrix:**
 - its transpose

$$\mathbf{R}(\mathbf{a}, \theta)^{-1} = \mathbf{R}(\mathbf{a}, \theta)^T$$

Transformations

Summary

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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

$$\mathbf{R}_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{R}_y(\theta) = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{R}_z(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{T}(\mathbf{d}) = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Review

- ❑ **Geometry and Transformations**
 - ❑ Introduce the elements of geometry
 - ❑ Scalars, Vectors, Points
- ❑ **Coordinate Systems**
- ❑ **Homogeneous Coordinates**

Next Lecture

Viewing and Projections



COMP 371

Computer Graphics

Session 5

VIEWING AND PROJECTIONS

Lecture Overview

- Review of last week's lecture
- Coordinate Systems/Transformations
- Viewing and projections
 - Orthographic/Perspective projections
- OpenGL viewing and projections

Transformations

Scalars, Vectors, Points

- **Scalars**
 - no geometric properties
- **Vectors**
 - direction and magnitude
- **Points**
 - location in space

Transformations

Coordinate Systems

- Consider a basis v_1, v_2, \dots, v_n
- A vector is written $v = a_1v_1 + a_2v_2 + \dots + a_nv_n$
- The list of scalars $\{a_1, a_2, \dots, a_n\}$ is the representation of v with respect to the given basis
- We can write the representation as a row or column array of scalars

Example:

$$v=2v_1+3v_2-4v_3$$

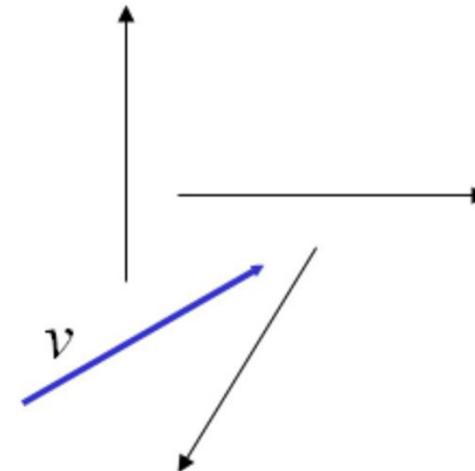
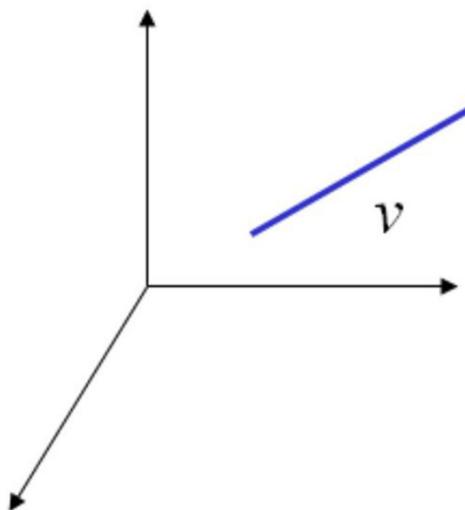
$$a=[2 \ 3 \ -4]^T$$

$$a = [a_1 \ a_2 \ \dots \ a_n]^T = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix}$$

Transformations

Coordinate Systems

- Which one is correct?



- → Both are correct because vectors have no fixed location

Transformations

Frames

- A coordinate system is insufficient to represent points
- If we work in an affine space we can add a single point P_0 , the origin, to the basis vectors to form a *frame*

Transformations

Representation in a Frame

- **Frame determined by (P_0, v_1, v_2, v_3)**
- **Within this frame, every vector can be written as**

$$v = a_1v_1 + a_2v_2 + \dots + a_nv_n$$

- **Every point can be written as**

$$P = P_0 + b_1v_1 + b_2v_2 + \dots + b_nv_n$$

Transformations

Confusing Points and Vectors

- Consider the point and the vector

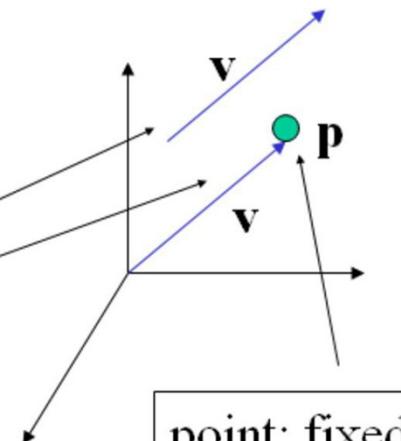
$$P = P_0 + b_1 v_1 + b_2 v_2 + \dots + b_n v_n$$

$$v = a_1 v_1 + a_2 v_2 + \dots + a_n v_n$$

- They appear to have the similar representations

$$p = [b_1 \ b_2 \ b_3] \quad v = [a_1 \ a_2 \ a_3]$$

Vector can be placed anywhere



- which confuses the point with the vector

A vector has no position

Transformations

A Single Representation

- If we define $0 \cdot P = 0$ and $1 \cdot P = P$ then we can write

$$v = a_1 v_1 + a_2 v_2 + a_3 v_3 = [a_1 \ a_2 \ a_3 \ 0] [v_1 \ v_2 \ v_3 \ P_0]^T$$

$$P = P_0 + b_1 v_1 + b_2 v_2 + b_3 v_3 = [b_1 \ b_2 \ b_3 \ 1] [v_1 \ v_2 \ v_3 \ P_0]^T$$

- Thus we obtain the four-dimensional homogeneous coordinate representation

$$\mathbf{v} = [a_1 \ a_2 \ a_3 \ 0]^T$$

$$\mathbf{p} = [b_1 \ b_2 \ b_3 \ 1]^T$$

Transformations

Homogeneous Coordinates

- The homogeneous coordinates form of a three dimensional point $[x \ y \ z]$ is given as
 - $\mathbf{p} = [x' \ y' \ z' \ w]^T = [wx \ wy \ wz \ w]^T$
- We return to a three dimensional point (for $w \neq 0$) by
 - $x \leftarrow x'/w$
 - $y \leftarrow y'/w$
 - $z \leftarrow z'/w$
- If $w=0$, the representation is that of a vector

Transformations

Homogeneous Coordinates

- Homogeneous coordinates are key to all computer graphics systems
 - All standard transformations (rotation, translation, scaling) can be implemented with matrix multiplications using 4×4 matrices
 - -Hardware pipeline works with 4 dimensional representations
 - -For orthographic viewing, we can maintain $w=0$ for vectors and $w=1$ for points
 - For perspective we need a perspective division

Transformations

Change of Coordinate Systems

- Consider two representations of the same vector with respect to two different bases.
The representations are

$$\mathbf{a} = [\alpha_1 \alpha_2 \alpha_3]$$

$$\mathbf{b} = [\beta_1 \beta_2 \beta_3]$$

where

$$\begin{aligned}\mathbf{v} &= \alpha_1 \mathbf{v}_1 + \alpha_2 \mathbf{v}_2 + \alpha_3 \mathbf{v}_3 = [\alpha_1 \alpha_2 \alpha_3] [\mathbf{v}_1 \mathbf{v}_2 \mathbf{v}_3]^T \\ &= \beta_1 \mathbf{u}_1 + \beta_2 \mathbf{u}_2 + \beta_3 \mathbf{u}_3 = [\beta_1 \beta_2 \beta_3] [\mathbf{u}_1 \mathbf{u}_2 \mathbf{u}_3]^T\end{aligned}$$

Transformations

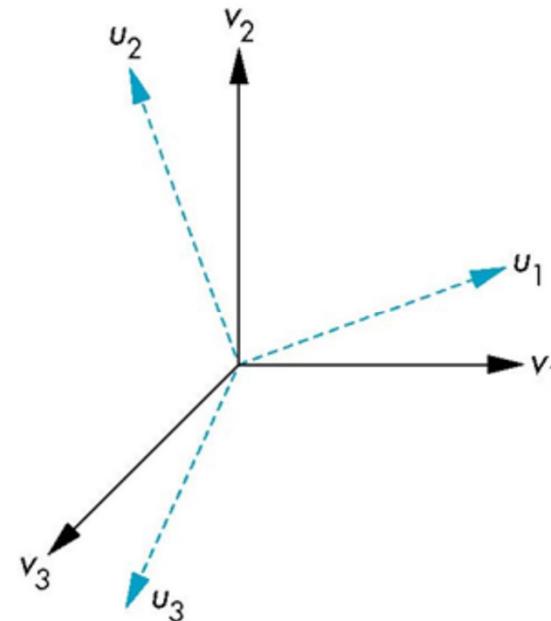
Representing Second Basis in Terms of First

- Each of the basis vectors, \mathbf{u}_1 , \mathbf{u}_2 , \mathbf{u}_3 , are vectors that can be represented in terms of the first basis

$$\mathbf{u}_1 = \gamma_{11}\mathbf{v}_1 + \gamma_{12}\mathbf{v}_2 + \gamma_{13}\mathbf{v}_3$$

$$\mathbf{u}_2 = \gamma_{21}\mathbf{v}_1 + \gamma_{22}\mathbf{v}_2 + \gamma_{23}\mathbf{v}_3$$

$$\mathbf{u}_3 = \gamma_{31}\mathbf{v}_1 + \gamma_{32}\mathbf{v}_2 + \gamma_{33}\mathbf{v}_3$$



Transformations

Matrix Form

- The coefficients define a 3 x 3 matrix

$$\mathbf{M} = \begin{bmatrix} \gamma_{11} & \gamma_{12} & \gamma_{13} \\ \gamma_{21} & \gamma_{22} & \gamma_{23} \\ \gamma_{31} & \gamma_{32} & \gamma_{33} \end{bmatrix}$$

and the bases can be related by

$$\mathbf{a} = \mathbf{M}^T \mathbf{b}$$

Transformations

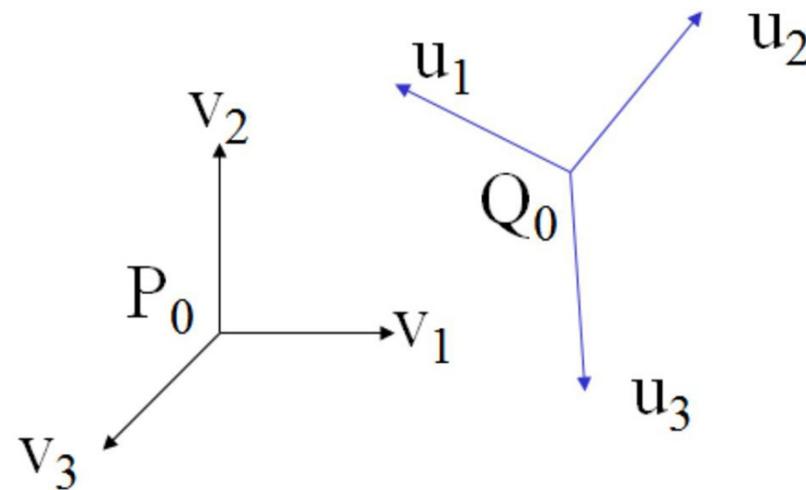
Change of Frames

- We can apply a similar process in homogeneous coordinates to the representations of both points and vectors

Consider two frames:

$$(P_0, v_1, v_2, v_3)$$

$$(Q_0, u_1, u_2, u_3)$$



- Any point or vector can be represented in either frame
- We can represent Q_0 , u_1 , u_2 , u_3 in terms of P_0 , v_1 , v_2 , v_3

Transformations

Representing One Frame in Terms of the Other

Extending what we did with change of bases

$$u_1 = \gamma_{11}v_1 + \gamma_{12}v_2 + \gamma_{13}v_3$$

$$u_2 = \gamma_{21}v_1 + \gamma_{22}v_2 + \gamma_{23}v_3$$

$$u_3 = \gamma_{31}v_1 + \gamma_{32}v_2 + \gamma_{33}v_3$$

$$Q_0 = \gamma_{41}v_1 + \gamma_{42}v_2 + \gamma_{43}v_3 + \gamma_{44}P_0$$

defining a 4×4 matrix

$$M = \begin{bmatrix} \gamma_{11} & \gamma_{12} & \gamma_{13} & 0 \\ \gamma_{21} & \gamma_{22} & \gamma_{23} & 0 \\ \gamma_{31} & \gamma_{32} & \gamma_{33} & 0 \\ \gamma_{41} & \gamma_{42} & \gamma_{43} & 1 \end{bmatrix}$$

Transformations

Working with Representations

- Within the two frames any point or vector has a representation of the same form

$\mathbf{a} = [a_1 \ a_2 \ a_3 \ a_4]$ in the first frame

$\mathbf{b} = [b_1 \ b_2 \ b_3 \ b_4]$ in the second frame

where $a_4 = b_4 = 1$ for points and $a_4 = b_4 = 0$ for vectors and

$$\mathbf{a} = \mathbf{M}^T \mathbf{b}$$

- The matrix \mathbf{M} is 4×4 and specifies an affine transformation in homogeneous coordinates

Transformations

The World and Camera Frames

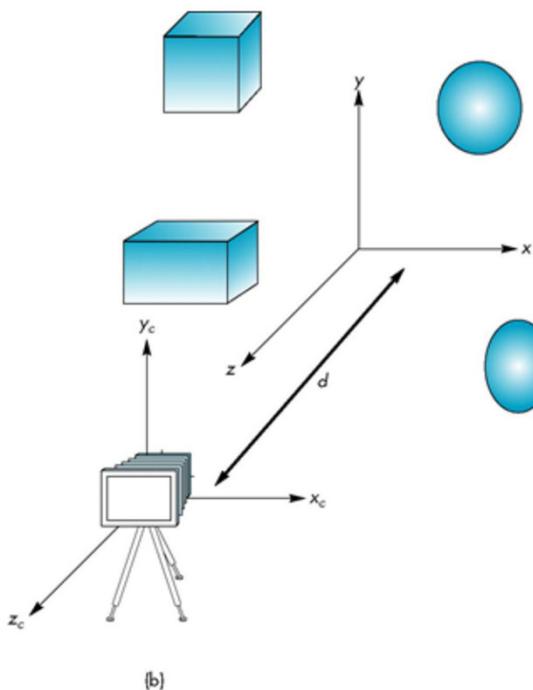
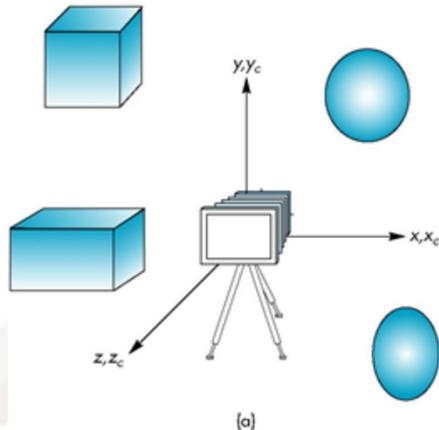
- When we work with representations, we work with n-tuples or arrays of scalars
- Changes in frame are then defined by 4 x 4 matrices
- In OpenGL, the base frame that we start with is the world frame
- Eventually we represent entities in the camera frame by changing the world representation using the model-view matrix
- Initially these frames are the same ($M=I$)

Transformations

Example: Moving the Camera

- If objects are on both sides of $z=0$, we must move camera frame

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -d \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Transformations

Translation Matrix

- A translation using a 4×4 matrix T in homogeneous coordinates $\mathbf{p}' = T\mathbf{p}$ where

- $T = T(d_x, d_y, d_z) = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$

Transformations

Rotation Matrix

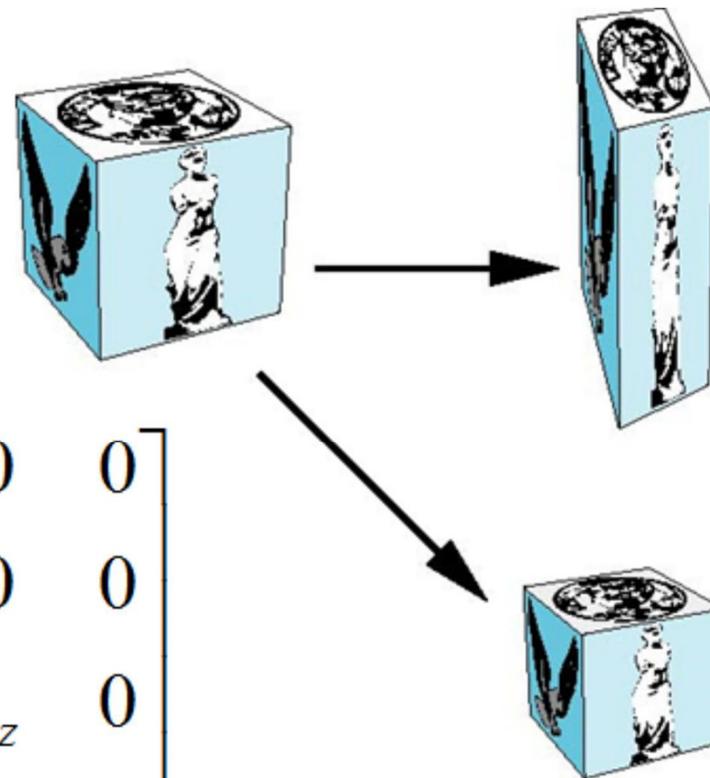
$$\mathbf{R} = \mathbf{R}_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{R} = \mathbf{R}_z(\theta) = \begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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

Transformations

Scaling Matrix

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



Transformations

Inverses

- Although we could compute inverse matrices by general formulas, we can use simple geometric observations
- Translation: $\mathbf{T}^{-1}(d_x, d_y, d_z) = \mathbf{T}(-d_x, -d_y, -d_z)$
- -Rotation: $\mathbf{R}^{-1}(q) = \mathbf{R}(-q)$
 - Holds for any rotation matrix
 - Note that since $\cos(-q) = \cos(q)$ and $\sin(-q) = -\sin(q)$
- $\mathbf{R}^{-1}(q) = \mathbf{R}^T(q)$
- -Scaling: $\mathbf{S}^{-1}(s_x, s_y, s_z) = \mathbf{S}(1/s_x, 1/s_y, 1/s_z)$

Transformations

Concatenation

- We can form arbitrary affine transformation matrices by multiplying together rotation, translation, and scaling matrices
- Because the same transformation is applied to many vertices, the cost of forming a matrix $M=ABCD$ is not significant compared to the cost of computing Mp for many vertices p
- The difficult part is how to form a desired transformation from the specifications in the application

Transformations

Order of Transformations

- Note that matrix on the right is first applied
- Mathematically, the following are equivalent

$$p' = ABCp = A(B(Cp))$$

Transformations

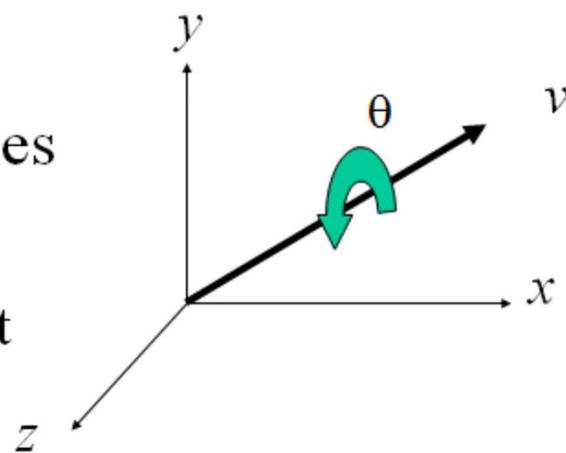
General Rotation About the Origin

A rotation by θ about an arbitrary axis can be decomposed into the concatenation of rotations about the x , y , and z axes

$$\mathbf{R}(\theta) = \mathbf{R}_z(\theta_z) \mathbf{R}_y(\theta_y) \mathbf{R}_x(\theta_x)$$

θ_x θ_y θ_z are called the Euler angles

Note that rotations do not commute
We can use rotations in another order but with different angles



Transformations

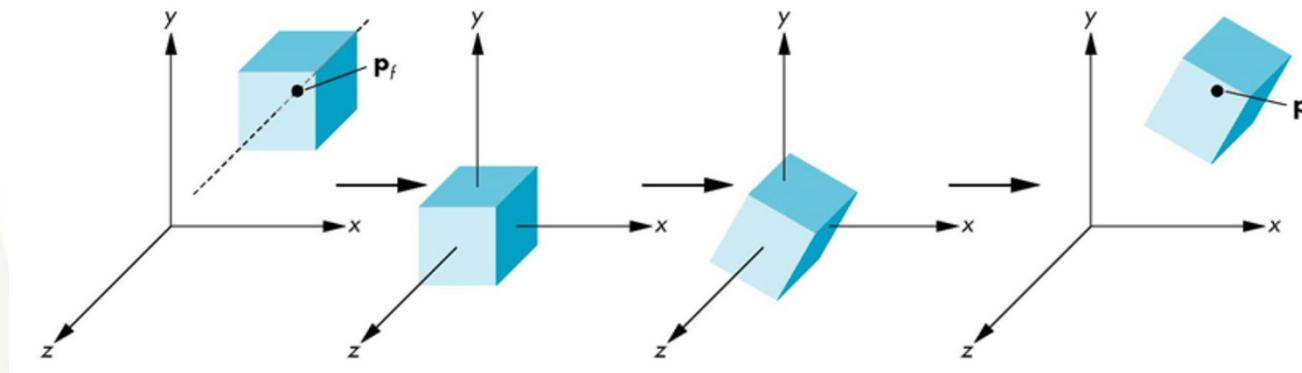
Rotation About a Fixed Point other than the Origin

Move fixed point to origin

Rotate

Move fixed point back

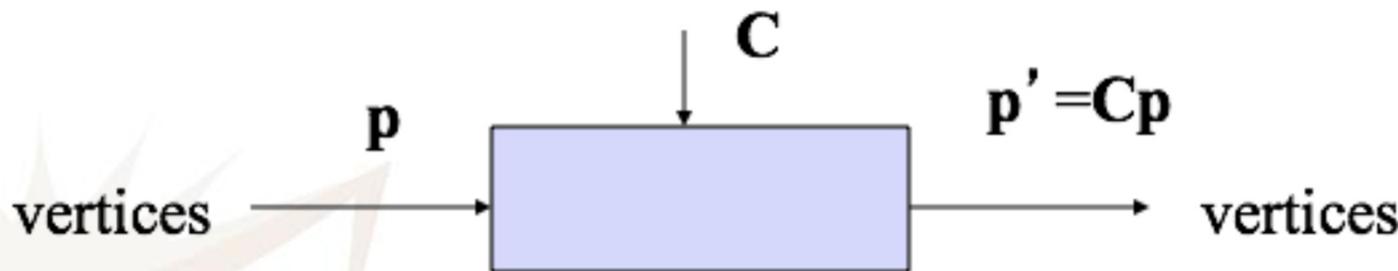
$$\mathbf{M} = \mathbf{T}(p_f) \mathbf{R}(\theta) \mathbf{T}(-p_f)$$



Transformations

OpenGL Transformations

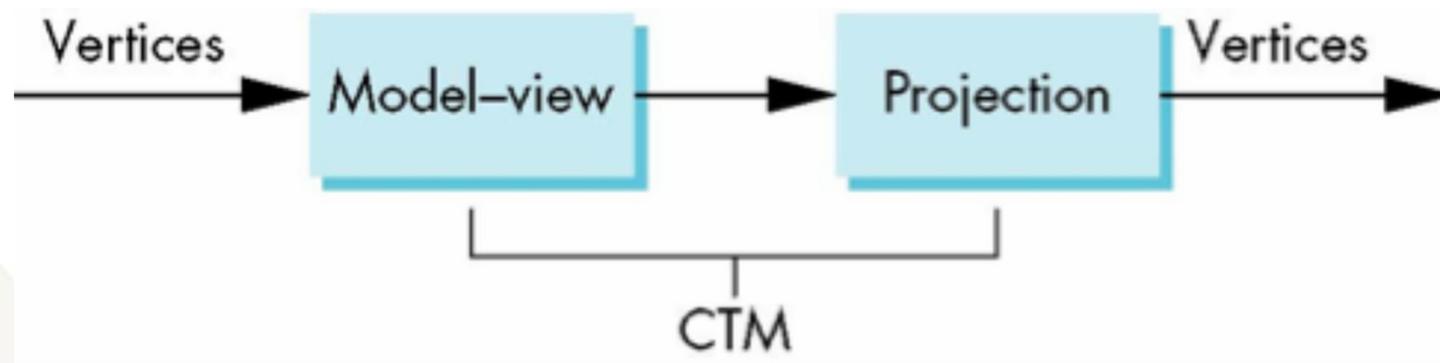
- Conceptually there is a 4×4 homogeneous coordinate matrix, the *current transformation matrix* (CTM) that is part of the state and is applied to all vertices that pass down the pipeline
- The **CTM** is defined in the user program and loaded into a transformation unit



Transformations

CTM in OpenGL

- OpenGL had a **model-view** and a **projection** matrix in the pipeline which were concatenated together to form the CTM
- We will emulate this process



Transformations

Rotation, Translation, Scaling

- `glm::mat4 m = glm::mat4(1.0);` **Identity Matrix**
- **Multiply by rotation matrix of theta in degrees/radians where (v_x , v_y , v_z) define axis of rotation**
- `glm::mat4 r = glm::rotate(theta, vx, vy, vz);`
`m = r * m`
or `glm::rotate(m, theta, vx, vy, vz);`
- **Do the same with translation and scaling:**
- `glm::mat4 s = glm::scale(sx, sy, sz);`
`glm::mat4 t = glm::translate(tx, ty, tz);`
`m = m * s * t;`

View and Projection

Viewing and Projection

- Our eyes collapse 3-D world to 2-D retinal image (brain then has to reconstruct 3D)
- In CG, this process occurs by **projection**
- **Projection has two parts:**
 - Viewing transformations: camera position and direction
 - Perspective/orthographic transformation: reduces 3D to 2D
- Use **homogeneous transformations**

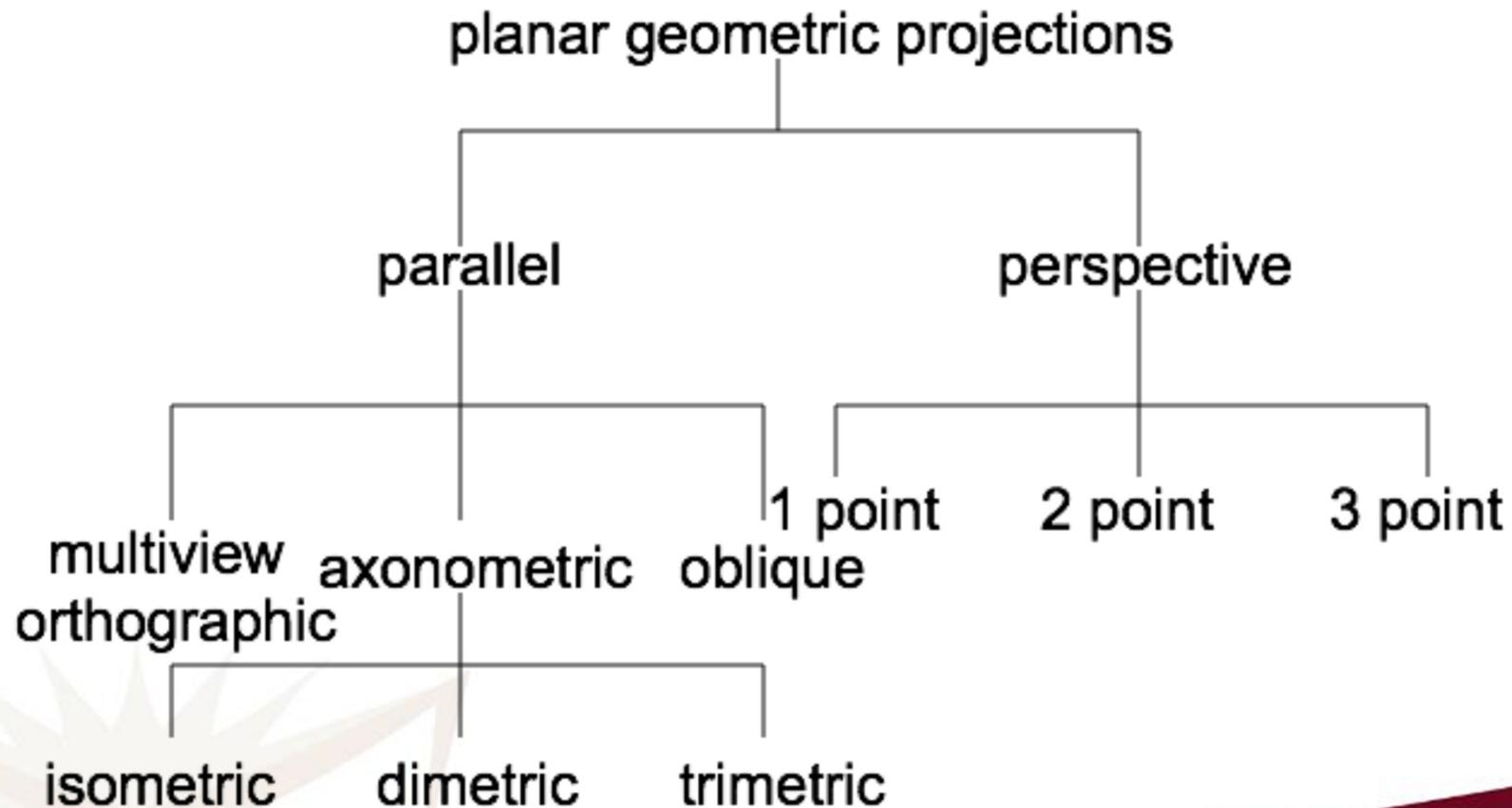
View and Projection

Geometric Projections

- Computer graphics treats all projections the same and implements them with a single pipeline
- Classical viewing developed different techniques for drawing each type of projection
- Fundamental distinction is between parallel and perspective viewing even though mathematically parallel viewing is the **limit of perspective viewing**

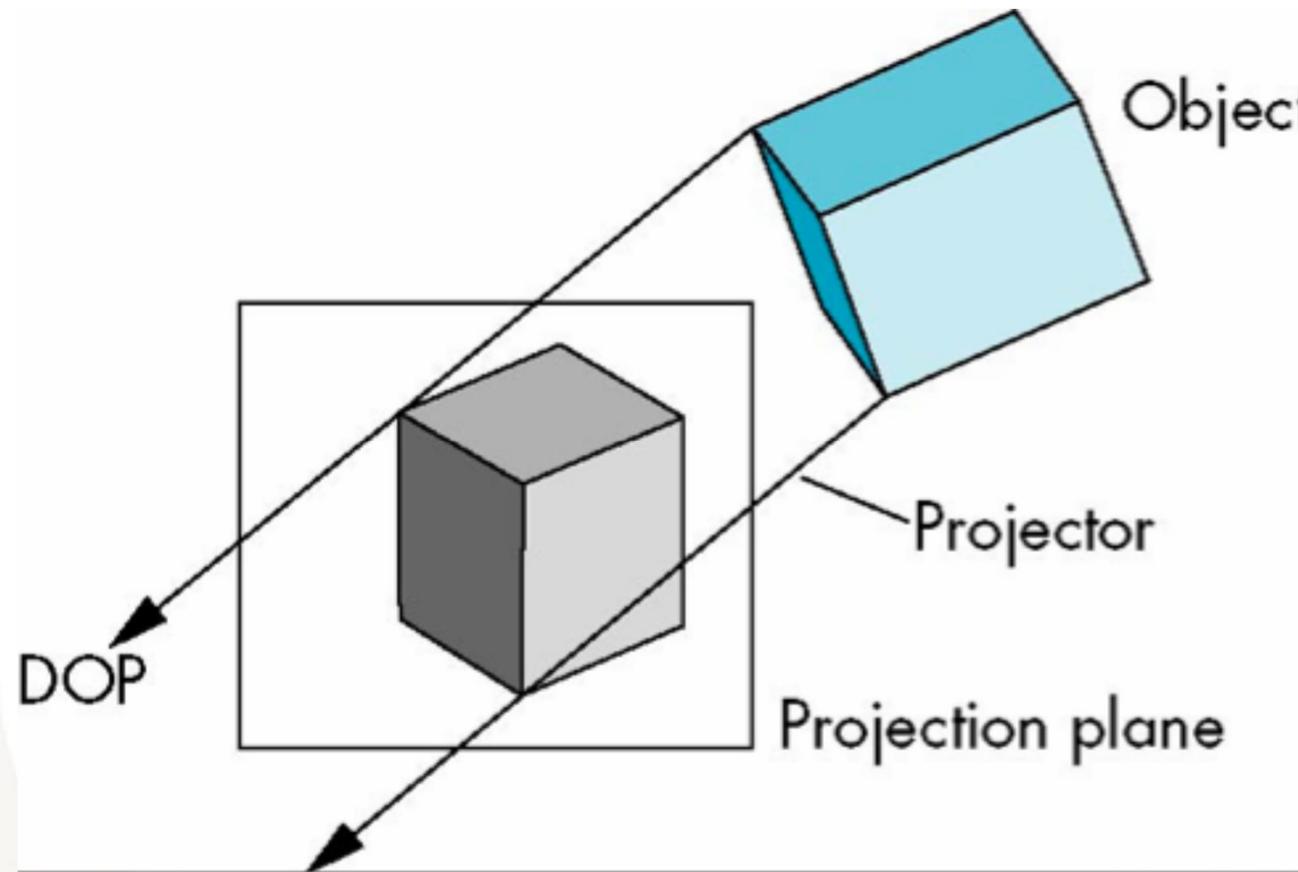
View and Projection

Geometric Projections



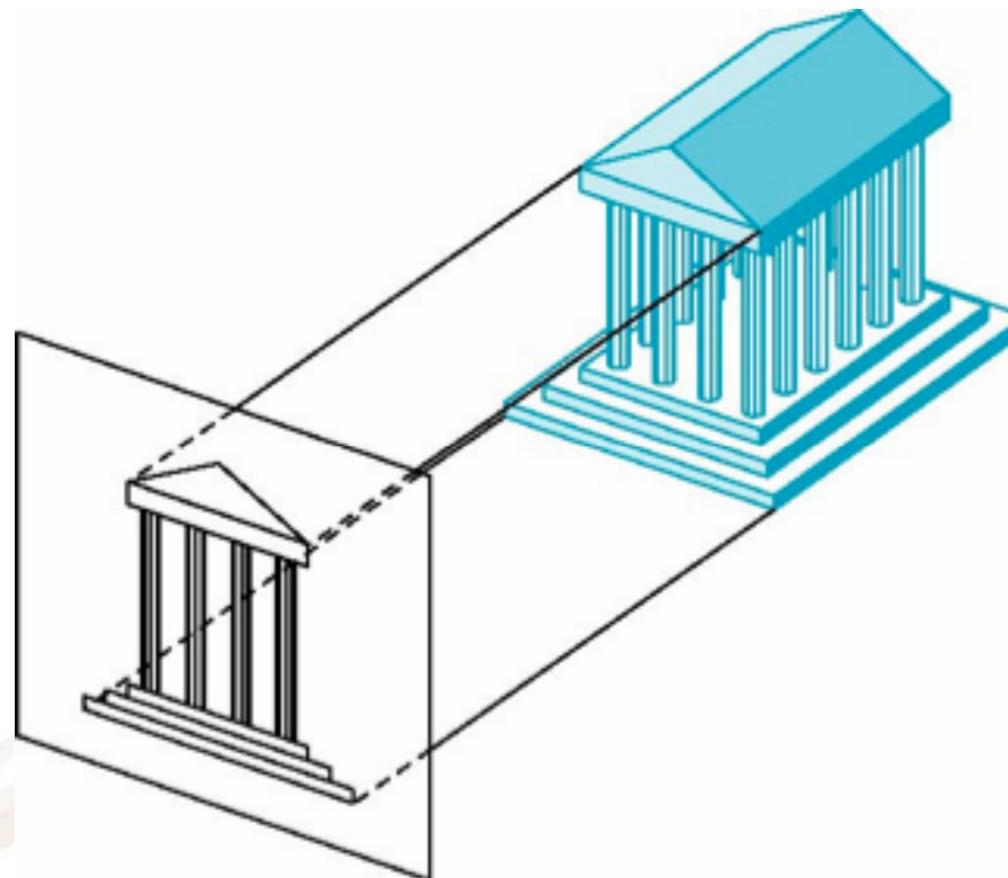
View and Projection

Parallel Projection



View and Projection

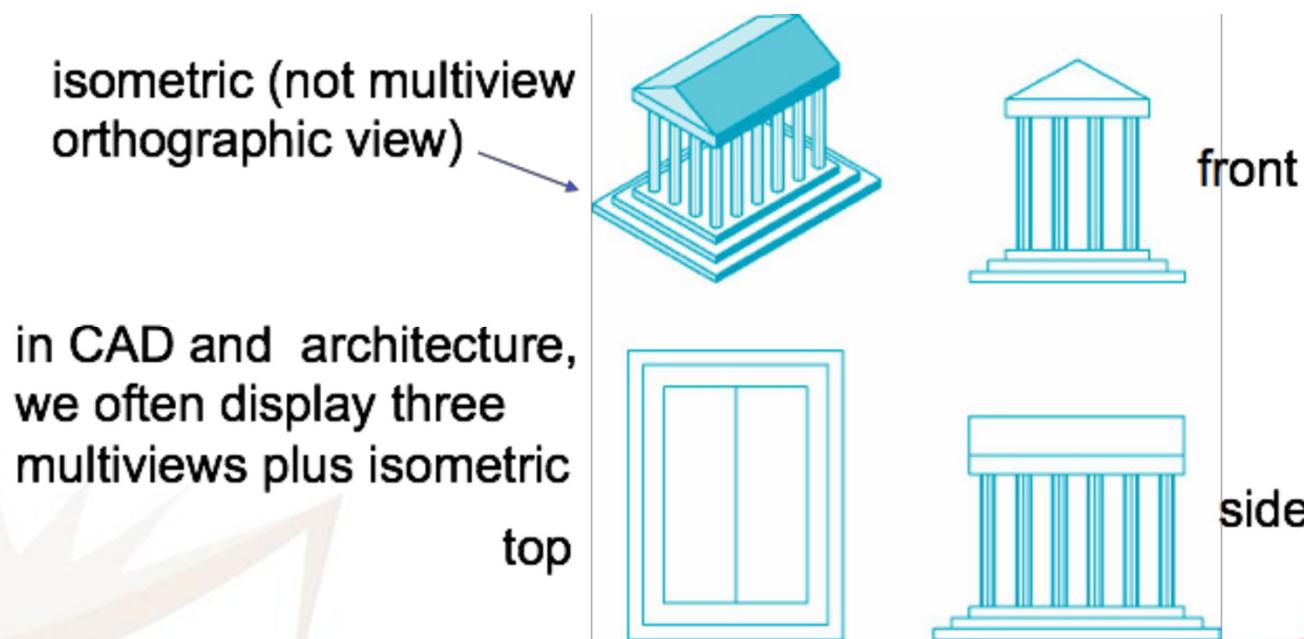
Orthographic Projection



View and Projection

Mult-view Orthographic Projection

- **Projection plane parallel to principal face**
- **Usually form front, top, side views**



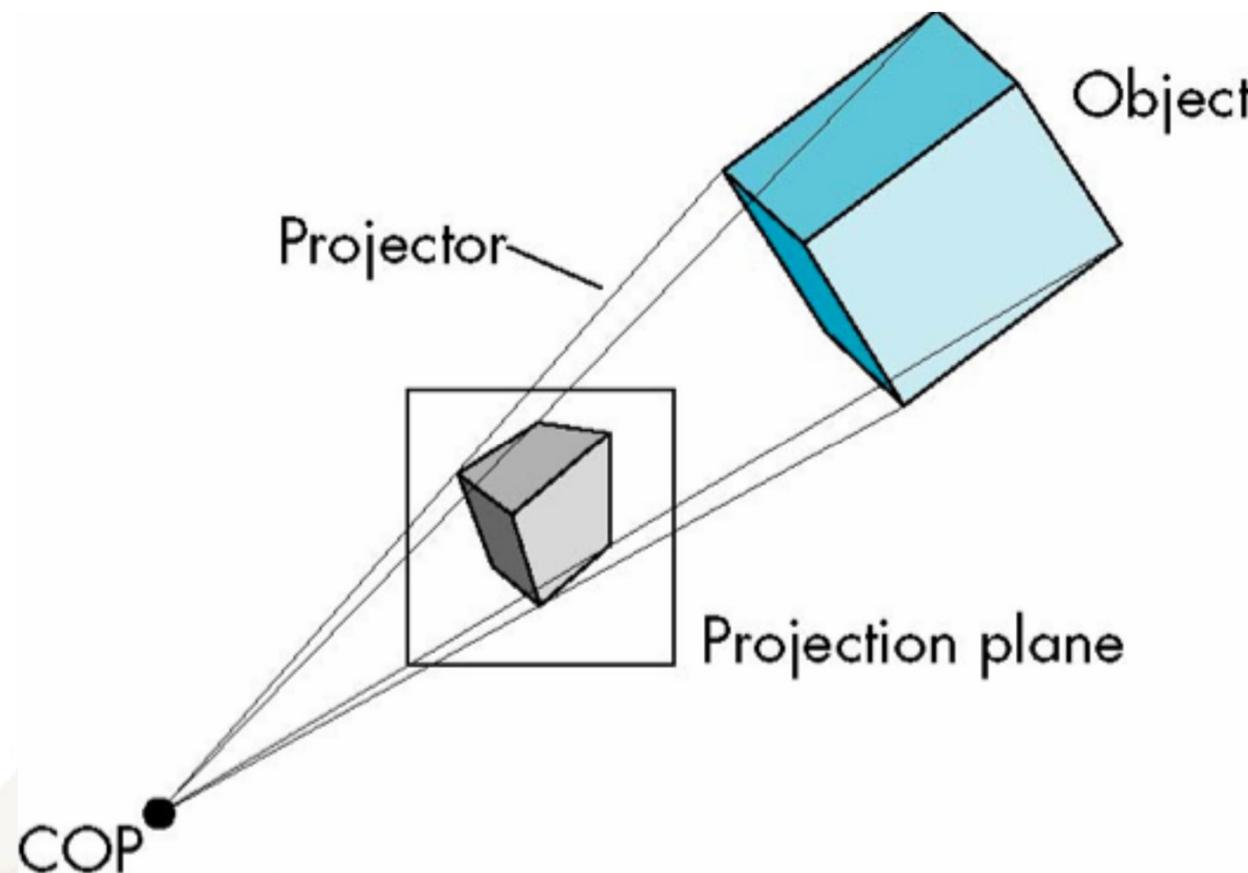
View and Projection

Advantages and Disadvantages

- **Preserves both distances and angles**
 - Shapes preserved
 - Can be used for measurements
 - Building plans
 - Manuals
- **Cannot see what the object really looks like because many surfaces hidden from view**
 - Often we add the isometric

View and Projection

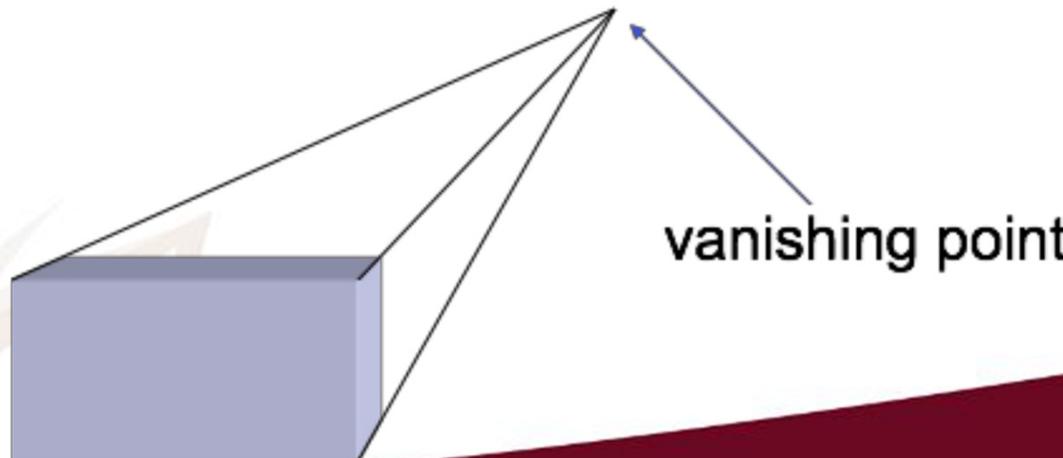
Perspective Projection



View and Projection

Vanishing Points

- Parallel lines (not parallel to the projection plane) on the object converge at a single point in the projection (*the vanishing point*)
- Drawing simple perspectives by hand uses these vanishing point(s)



View and Projection

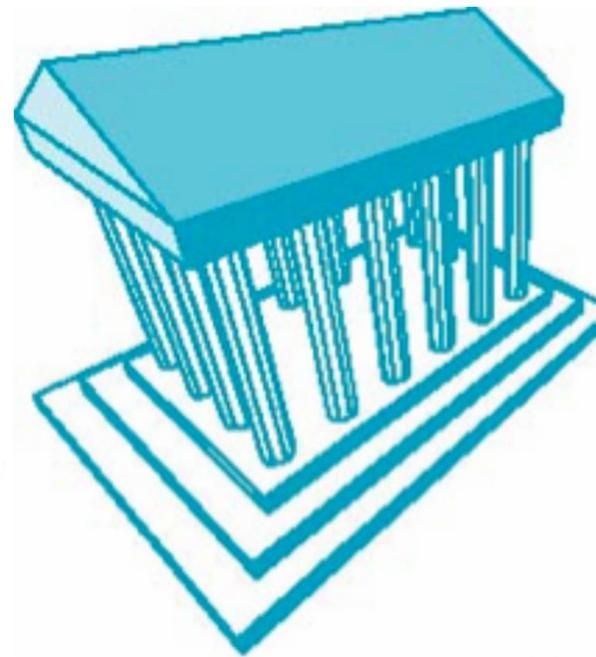
Vanishing Points



View and Projection

Three-Point Perspective

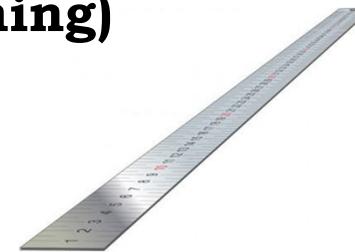
- No principal face parallel to projection plane
- Three vanishing points for cube



View and Projection

Advantages and Disadvantages

- Objects further from viewer are projected smaller than the same sized objects closer to the viewer (**diminution**)
 - Looks realistic
- Equal distances along a line are not projected into equal distances (**non-uniform foreshortening**)



- Angles preserved only in planes parallel to the projection plane
- More difficult to construct by hand than parallel projections (but not more difficult by computer)

View and Projection

Computer Viewing

- There are three aspects of the viewing process, all of which are implemented in the pipeline,
 - Positioning the camera
 - Setting the model-view matrix
 - Selecting a lens
 - Setting the projection matrix
 - Clipping
 - Setting the view volume

View and Projection

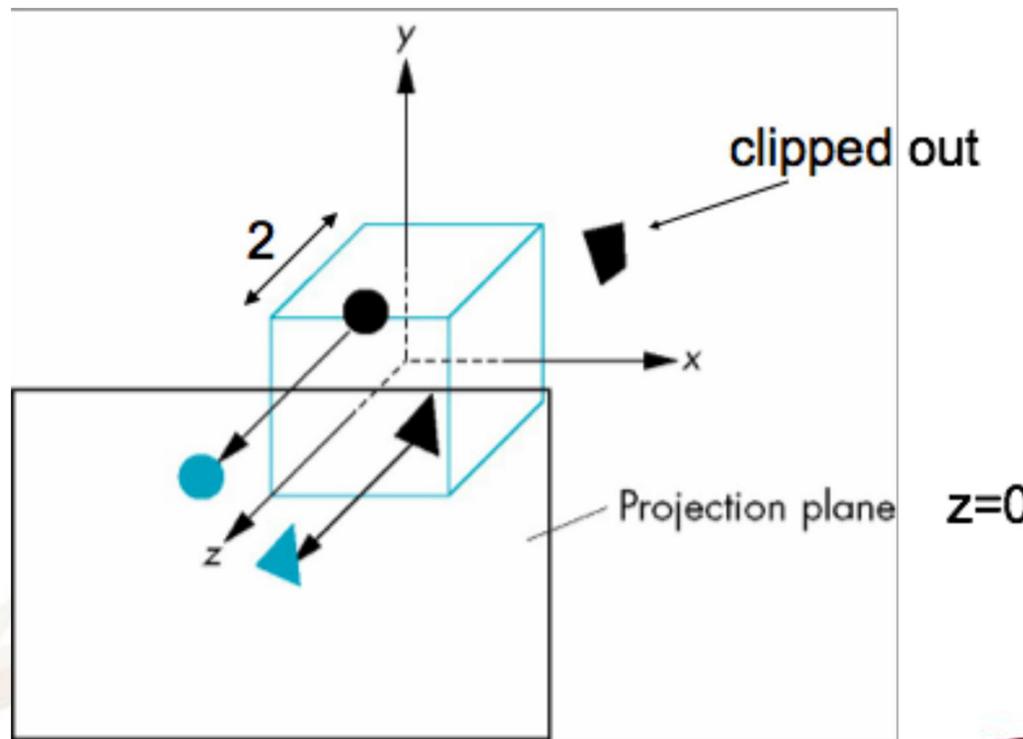
The OpenGL Camera

- In OpenGL, initially the object and camera frames are the same
 - Default model-view matrix is an identity
- The camera is located at origin and points in the negative z direction
- OpenGL also specifies a default view volume that is a cube with sides of length 2 centered at the origin
 - Default projection matrix is an identity

View and Projection

Default Projection

- Default projection is orthogonal



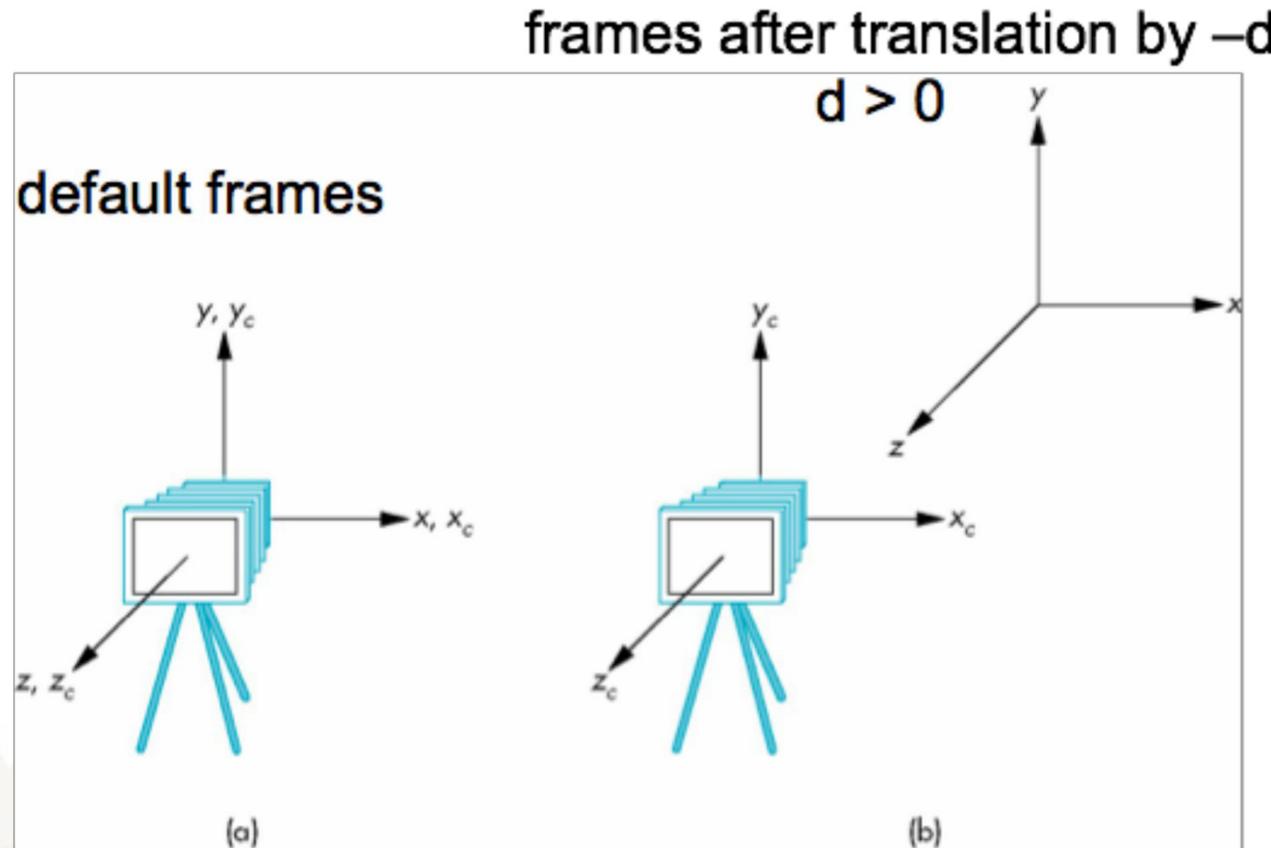
View and Projection

Moving the Camera Frame

- If we want to visualize objects with both positive and negative z values we can either
 - Move the camera in the positive z direction
 - Translate the camera frame
 - Move the objects in the negative z direction
 - Translate the world frame
- Both of these views are equivalent and are determined by the model-view matrix
 - Want a translation (`Translate(0.0,0.0,-d);`)
 - $d > 0$

View and Projection

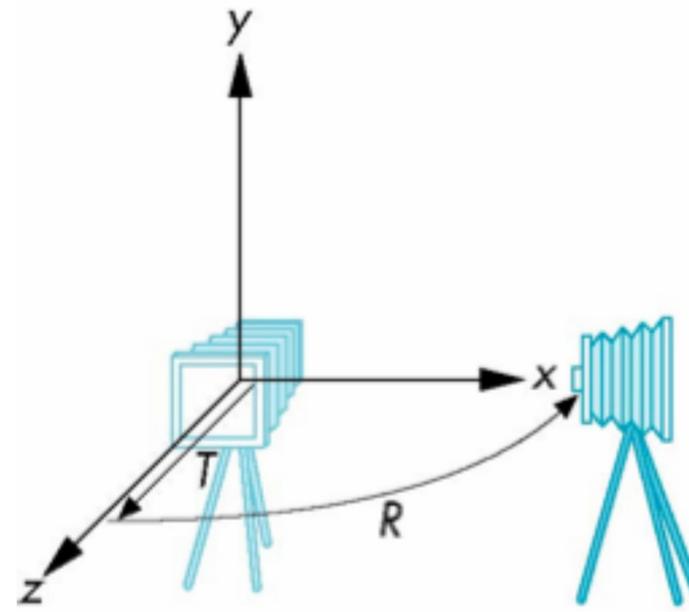
Moving Camera back from Origin



View and Projection

Moving the Camera

- We can move the camera to any desired position by a sequence of rotations and translations
- Example: side view
 - Rotate the camera
 - Move it away from origin
 - Model-view matrix $C = TR$



View and Projection

OpenGL Code

- Remember that last transformation specified is first to be applied

```
glm::mat4 camera_matrix = glm::mat4(1.0);
```

```
glm::translate(camera_matrix, tx, ty, tz);
```

```
glm::rotate(camera_matrix, 90.0*3.14/180.0, 0.0, 1.0, 0.0);
```

View and Projection

The `glm::lookAt` Function

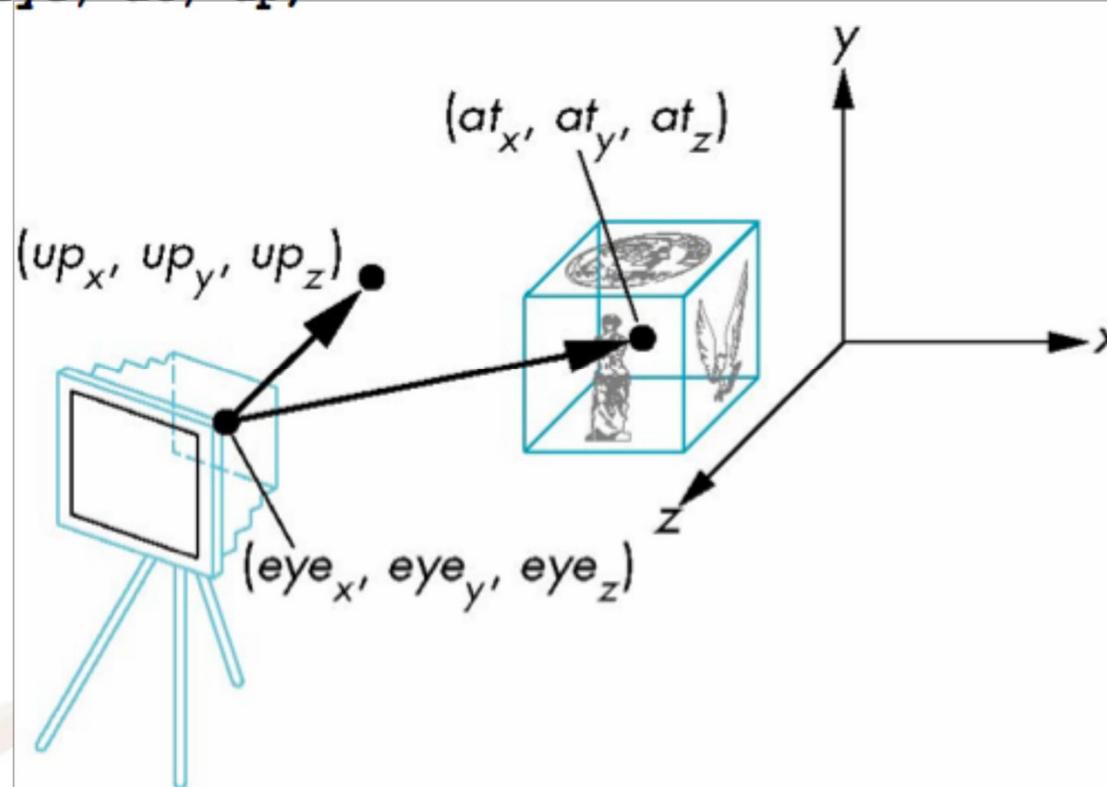
- The GLM library contains the function `glm::lookAt` to form the required model-view matrix through a simple interface
- Example:

```
glm::mat4 camera_matrix = glm::lookAt(glm::vec4 eye,  
                                     glm::vec4 at, glm::vec4 up);
```

View and Projection

The `glm::lookAt`

`LookAt(eye, at, up)`



View and Projection

Projections and Normalization

- The default projection in the eye (camera) frame is orthogonal
- For points within the default view volume

$$x_p = x$$

$$y_p = y$$

$$z_p = 0$$

- Most graphics systems use view normalization
 - All other views are converted to the default view by transformations that determine the projection matrix
 - Allows use of the same pipeline for all views

View and Projection

Homogeneous Coordinate Representation

default orthographic projection

$$x_p = x$$

$$y_p = y$$

$$z_p = 0$$

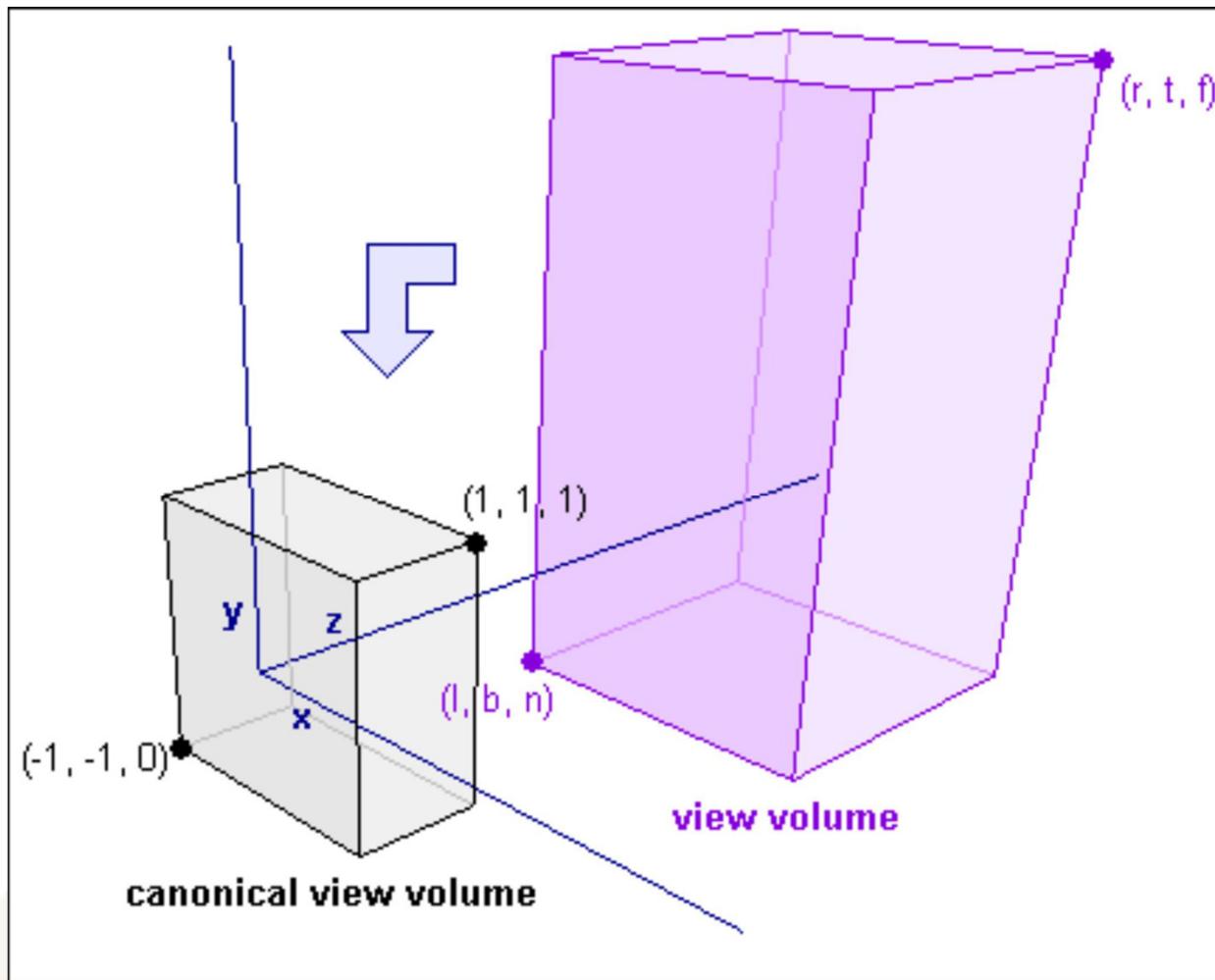
$$w_p = 1$$

$$\mathbf{p}_p = \mathbf{M}\mathbf{p}$$

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

In practice, we can let $\mathbf{M} = \mathbf{I}$ and set the z term to zero later

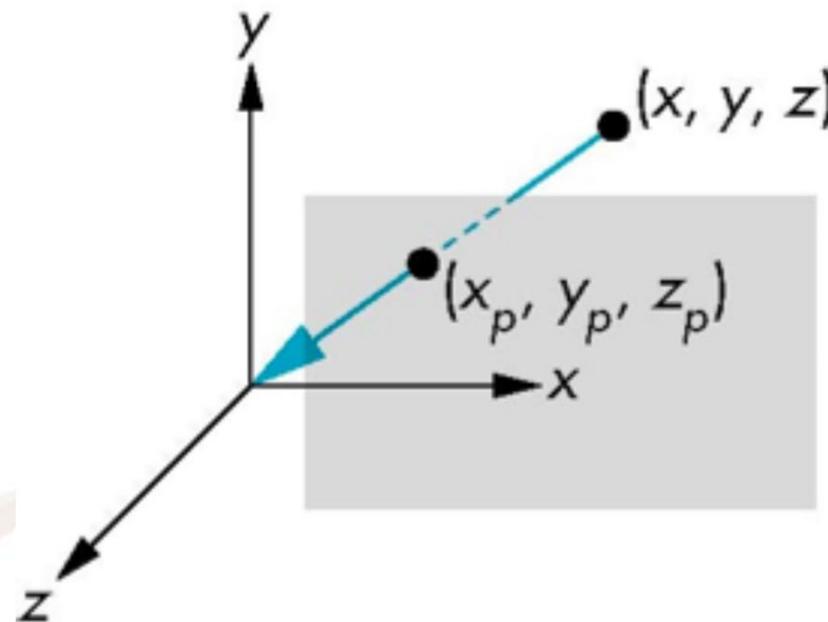
View and Projection



View and Projection

Simple Perspective

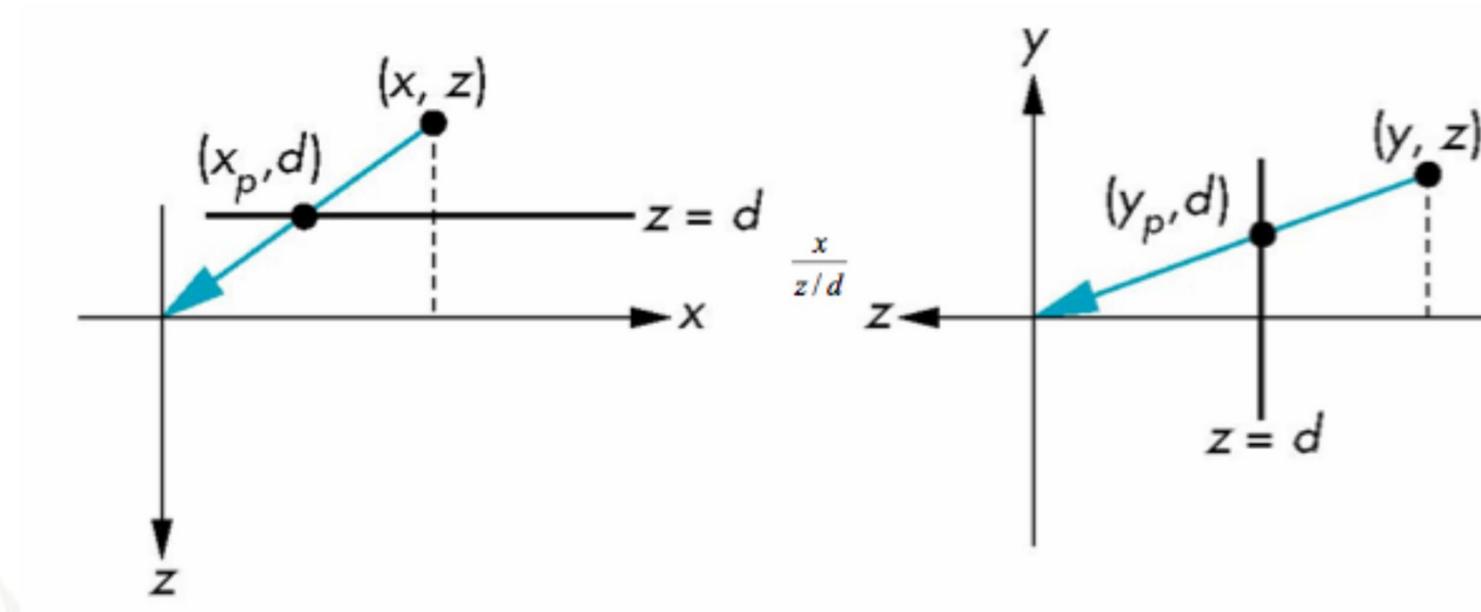
- Center of projection at the origin
- Projection plane $z = d$, $d < 0$



View and Projection

Perspective Equations

- Consider top and side views



$$x_p = \frac{x}{z/d}$$

$$y_p = \frac{y}{z/d}$$

$$z_p = d$$

View and Projection

Homogeneous Coordinate Form

consider $\mathbf{q} = \mathbf{Mp}$ where $\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix}$

$$\mathbf{q} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \Rightarrow \mathbf{p} = \begin{bmatrix} x \\ y \\ z \\ z/d \end{bmatrix}$$

View and Projection

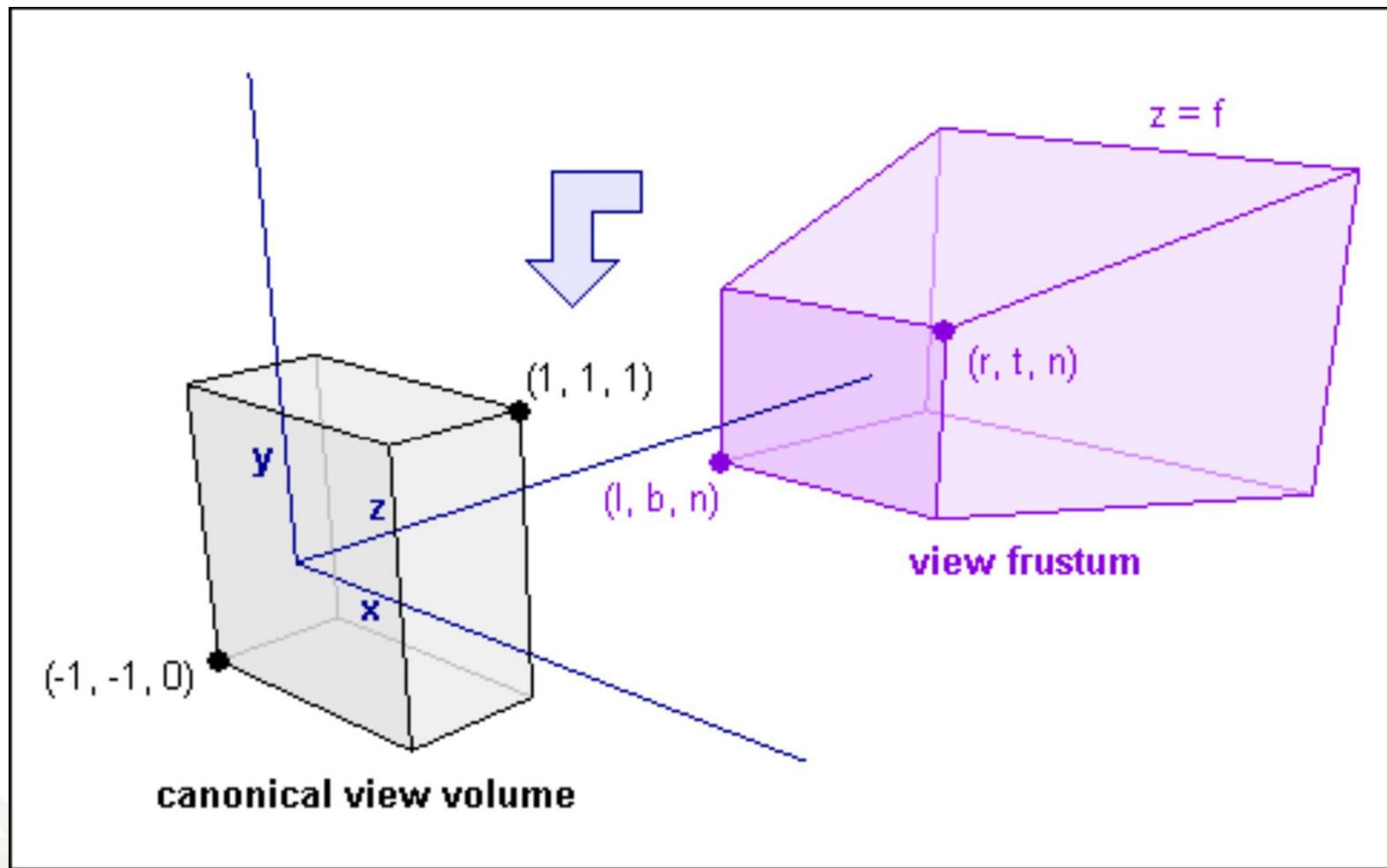
Perspective Division

- However $w \neq 1$, so we must divide by w to return from homogeneous coordinates
- This perspective division yields

$$x_p = \frac{x}{z/d} \quad y_p = \frac{y}{z/d} \quad z_p = d$$

- the desired perspective equations

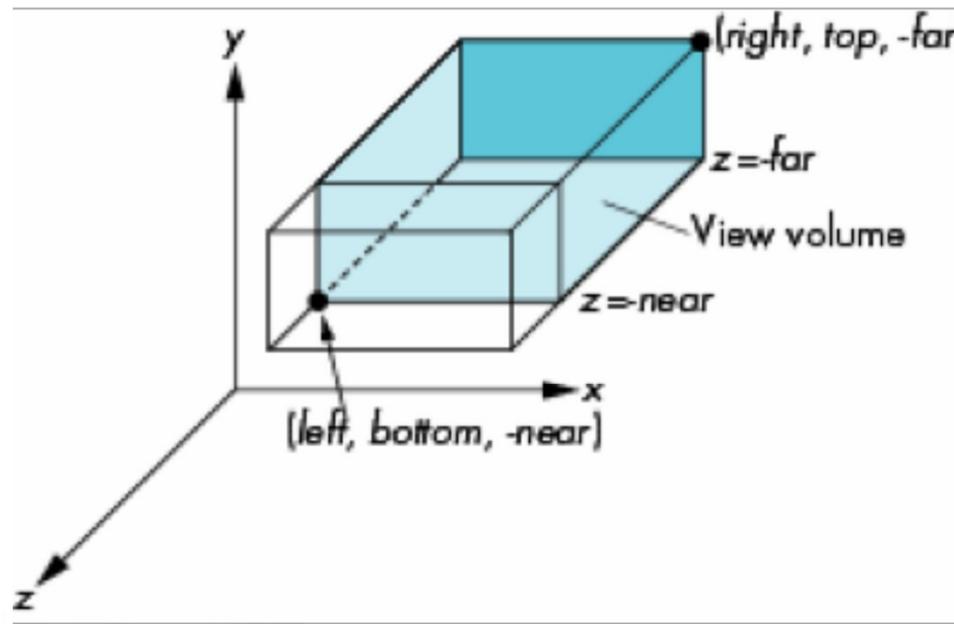
View and Projection



View and Projection

OpenGL Orthogonal Viewing

- `glm::ortho(left, right, bottom, top, near, far)`

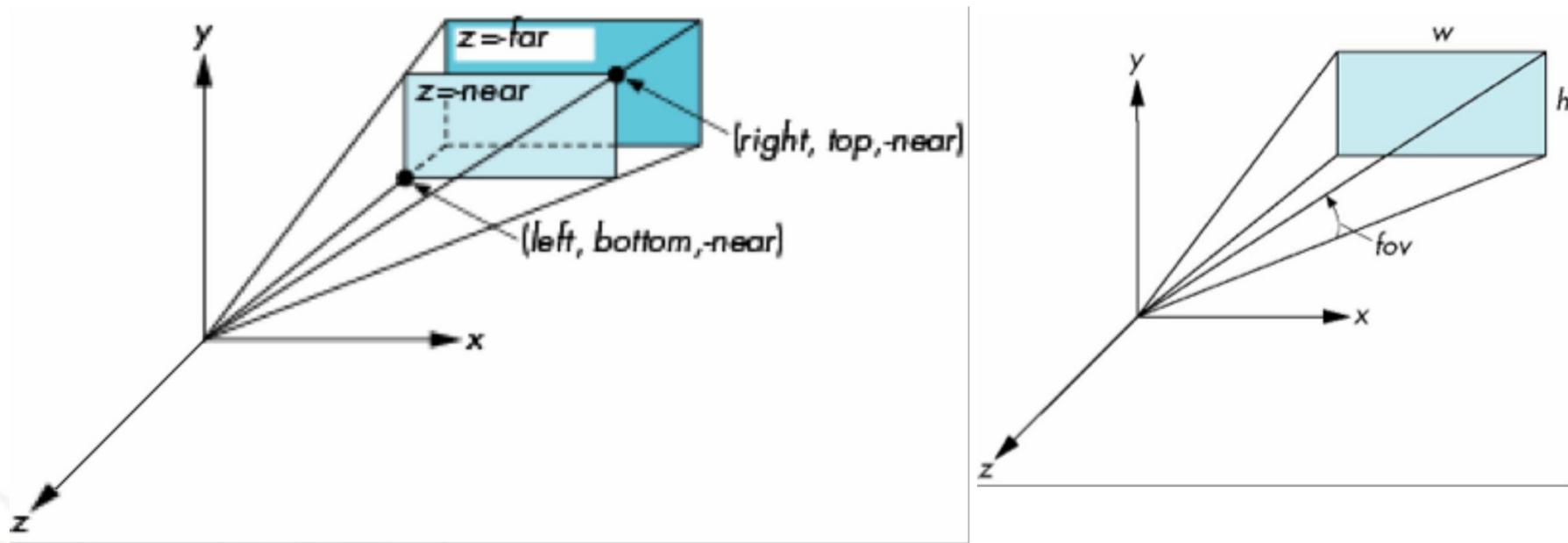


near and far measured from camera

View and Projection

OpenGL Perspective Viewing

- `glm::perspective(fovy, aspect, near, far)`



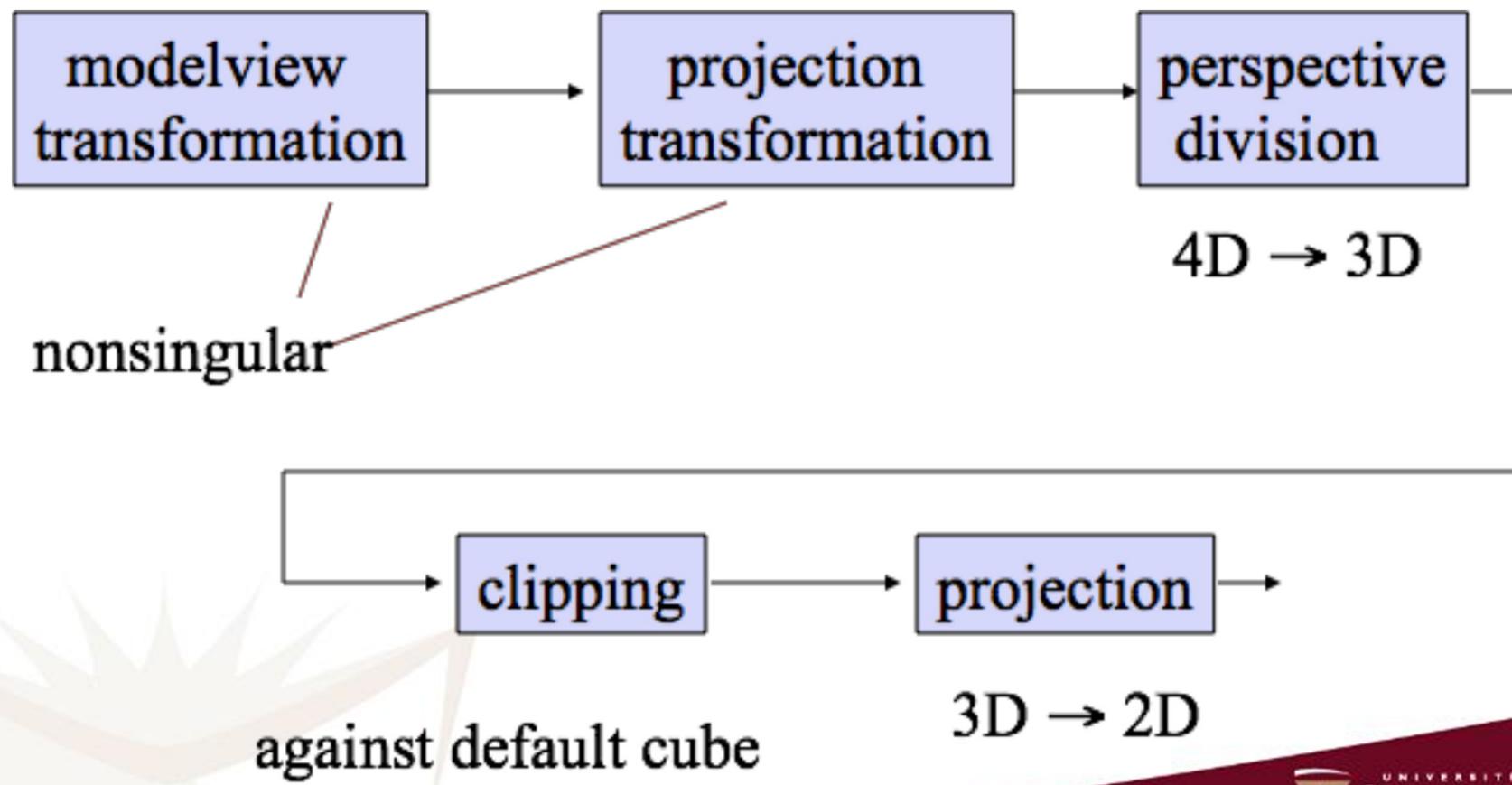
View and Projection

Normalization

- Rather than derive a different projection matrix for each type of projection, we can convert all projections to orthogonal projections with the default view volume
- This strategy allows us to use standard transformations in the pipeline and makes for **efficient clipping**

View and Projection

Pipeline View



View and Projection

Notes

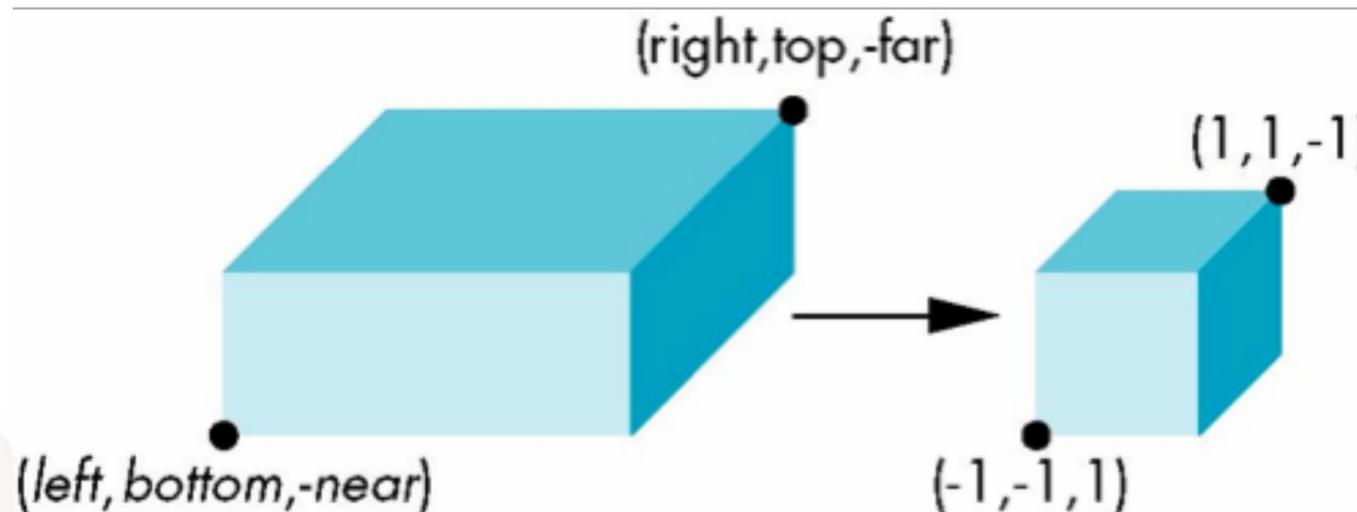
- We stay in four-dimensional homogeneous coordinates through both the model-view and projection transformations
 - Both these transformations are nonsingular
 - Default to identity matrices (orthogonal view)
- Normalization lets us clip against simple cube regardless of type of projection
- Delay final projection until end
 - Important for hidden-surface removal to retain depth information as long as possible

View and Projection

Orthogonal Normalization

- `glm::ortho(left, right, bottom, top, near, far)`

normalization \Rightarrow find transformation to convert specified clipping volume to default



View and Projection

Orthogonal Matrix

- Two steps
 - Move center to origin
 - $T(-(\text{left}+\text{right})/2, -(\text{bottom}+\text{top})/2, (\text{near}+\text{far})/2))$
 - Scale to have sides of length 2
 - $S(2/(\text{left}-\text{right}), 2/(\text{top}-\text{bottom}), 2/(\text{near}-\text{far}))$

$$\mathbf{P} = \mathbf{S}\mathbf{T} = \begin{bmatrix} \frac{2}{\text{right} - \text{left}} & 0 & 0 & -\frac{\text{right} - \text{left}}{\text{right} - \text{left}} \\ 0 & \frac{2}{\text{top} - \text{bottom}} & 0 & -\frac{\text{top} + \text{bottom}}{\text{top} - \text{bottom}} \\ 0 & 0 & \frac{2}{\text{near} - \text{far}} & \frac{\text{far} + \text{near}}{\text{far} - \text{near}} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

View and Projection

Final Projection

- Set $z = 0$
- Equivalent to the homogeneous coordinate transformation

$$\mathbf{M}_{\text{orth}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Hence, general orthogonal projection in 4D is

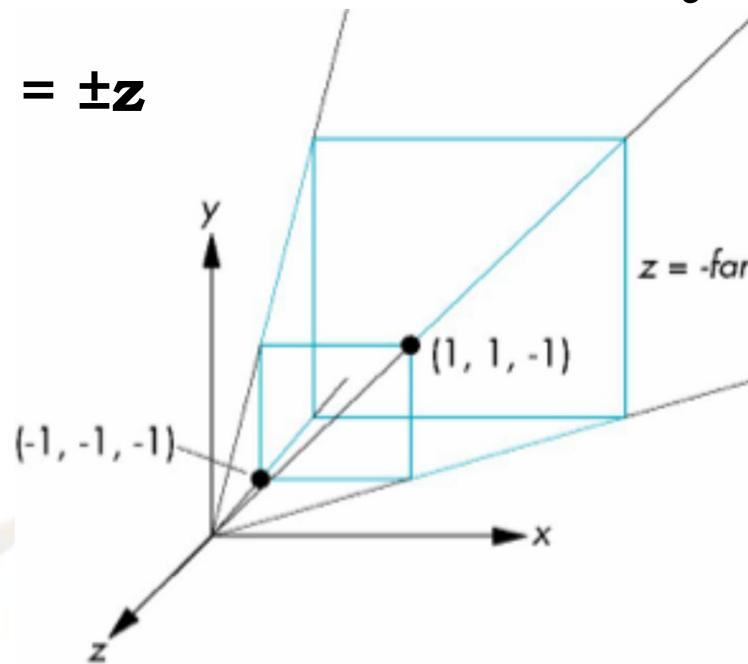
$$\mathbf{P} = \mathbf{M}_{\text{orth}} \mathbf{S} \mathbf{T}$$

View and Projection

Simple Perspective

- Consider a simple perspective with the COP at the origin, the near clipping plane at $z = -1$, and a 90 degree field of view determined by the planes

$$x = \pm z, y = \pm z$$



View and Projection

Perspective Matrices

- Simple projection matrix in homogeneous coordinates

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

- Note that this matrix is independent of the far clipping plane

View and Projection

Generalization

$$\mathbf{N} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \alpha & \beta \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

after perspective division, the point $(x, y, z, 1)$ goes to

$$x'' = x/z$$

$$y'' = y/z$$

$$z'' = -(\alpha + \beta/z)$$

which projects orthogonally to the desired point
regardless of α and β

View and Projection

Picking a and b

If we pick

$$\alpha = \frac{\text{near} + \text{far}}{\text{far} - \text{near}}$$

$$\beta = \frac{2\text{near} * \text{far}}{\text{near} - \text{far}}$$

the near plane is mapped to $z = -1$

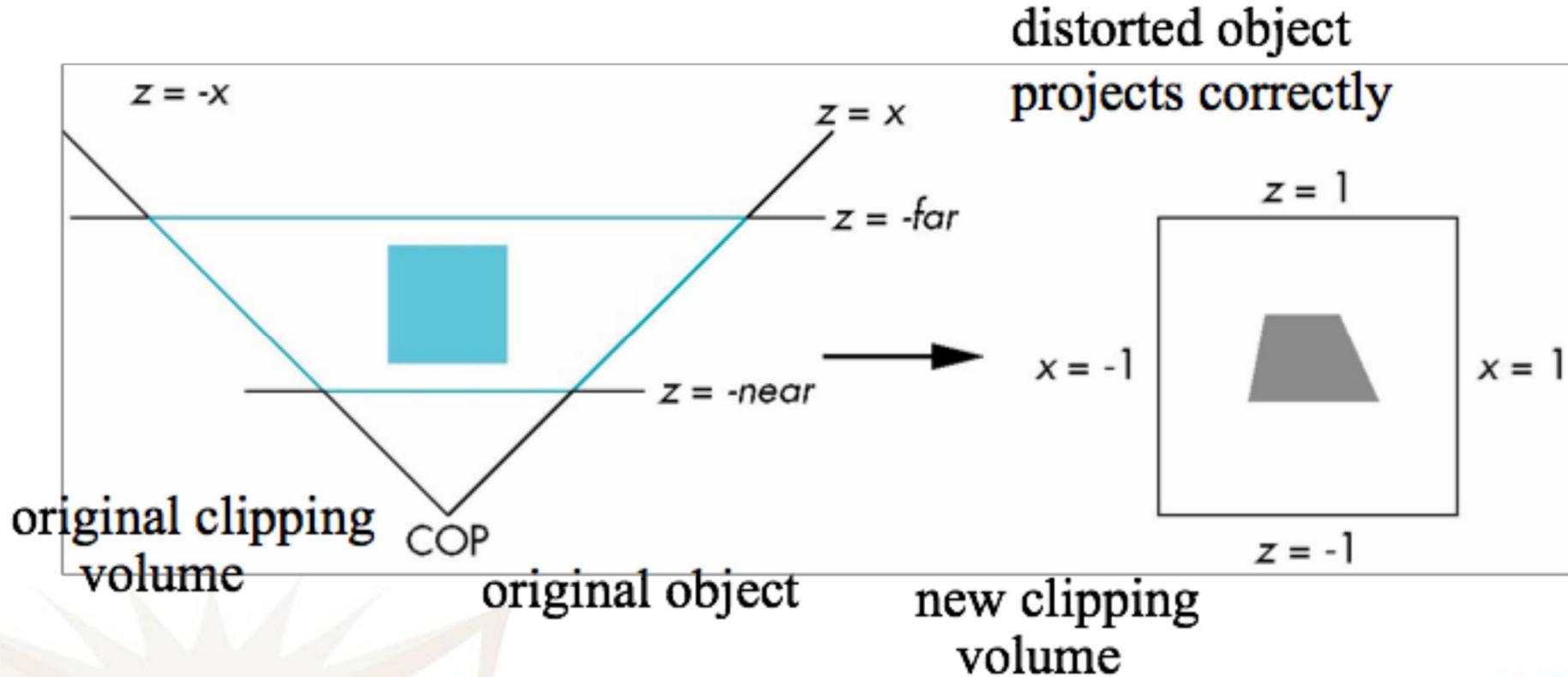
the far plane is mapped to $z = 1$

and the sides are mapped to $x = \pm 1, y = \pm 1$

Hence the new clipping volume is the default clipping volume

View and Projection

Normalization Transformation



View and Projection

Why do we do it this way?

- Normalization allows for a **single pipeline** for both perspective and orthogonal viewing
- We stay in four dimensional homogeneous coordinates as long as possible to retain three-dimensional information needed for hidden-surface removal and shading
- We simplify clipping

Review

- Coordinate Systems**
- Homogeneous Coordinates**
- Viewing**
- Projections**

Next Lecture

- Building models**

Support Slides

Shear Matrix

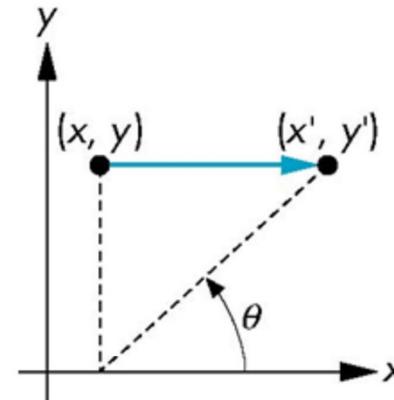
- Consider simple shear along x axis

$$x' = x + y \cot \theta$$

$$y' = y$$

$$z' = z$$

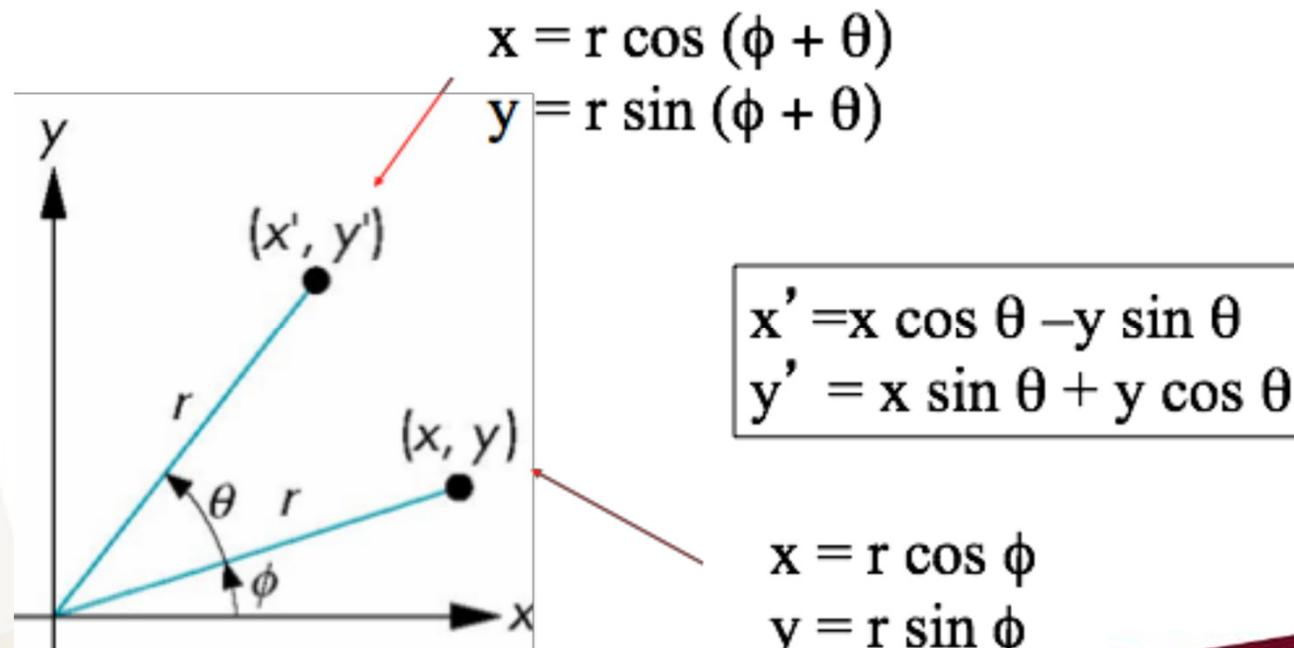
$$H(\theta) = \begin{bmatrix} 1 & \cot \theta & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Support Slides

Rotation Matrix (2D)

- Consider rotation about the origin θ degrees
radius stays the same, angle increases by θ



Support Slides

Rotation about the Z axis

- Rotation about z axis in three dimensions leaves all points with the same z
 - Equivalent to rotation in two dimensions in planes of constant z

$$\begin{aligned}x' &= x \cos \theta - y \sin \theta \\y' &= x \sin \theta + y \cos \theta \\z' &= z\end{aligned}$$

- or in homogeneous coordinates

$$\mathbf{p}' = \mathbf{R}_z(\theta)\mathbf{p}$$



COMP 371

Computer Graphics

Session 6
BUILDING MODELS



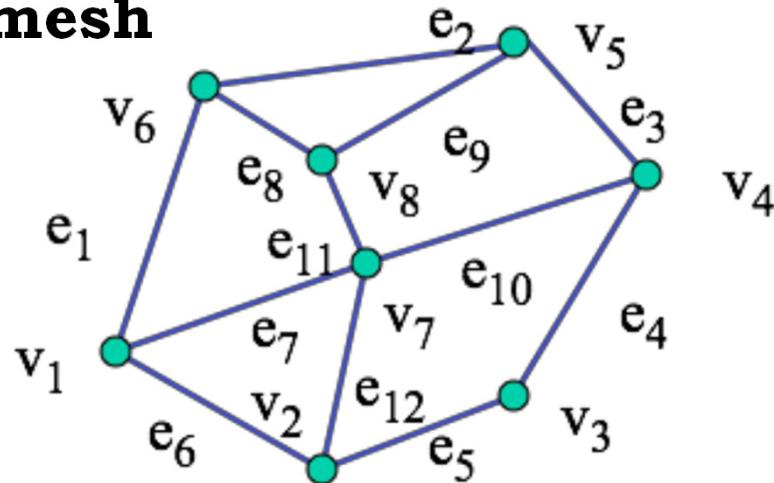
Lecture Overview

- Building Models
- Introduce simple data structures for building polygonal models

Building Models

Representing a Mesh

- Consider a mesh



- There are 8 nodes and 12 edges
- 5 interior polygons
- 6 interior (shared) edges
- Each vertex has a location $v_i = (x_i \ y_i \ z_i)$

Building Models

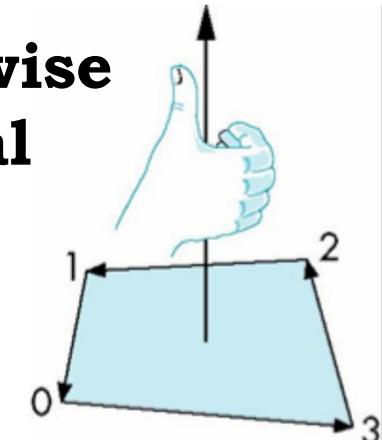
Simple Representation

- Define each polygon by the geometric locations of its vertices
- `glm::vec3 v1 = glm::vec3(x1, y1, z1);`
`glm::vec3 v2 = glm::vec3(x2, y2, z2);`
- ...
- Inefficient and unstructured

Building Models

Inward and Outward Facing Polygons

- The order $\{v_1, v_2, v_7\}$ and $\{v_2, v_7, v_1\}$ are equivalent in that the same polygon will be rendered by OpenGL but the order $\{v_1, v_7, v_2\}$ is different
- The first two describe outwardly facing polygons
- Use the right-hand rule = counter-clockwise encirclement of outward-pointing normal
- OpenGL can treat inward and
- outward facing polygons differently



Building Models

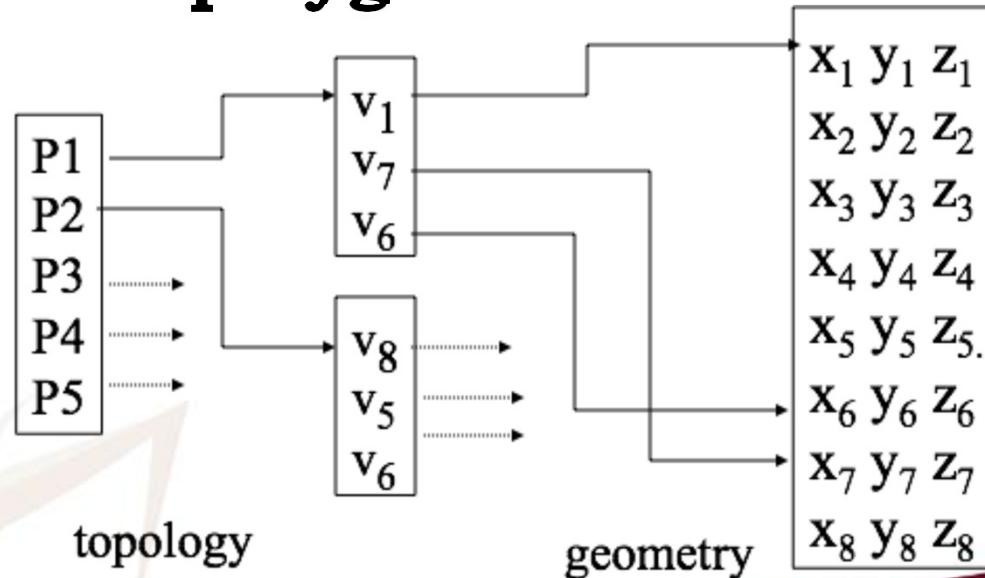
Geometry vs Topology

- Generally it is a good idea to look for data structures that separate the geometry from the topology
 - Geometry: locations of the vertices
 - Topology: organization of the vertices and edges
 - Example: a polygon is an ordered list of vertices with an edge connecting successive pairs of vertices and the last to the first
 - Topology holds even if geometry changes

Building Models

Vertex Lists

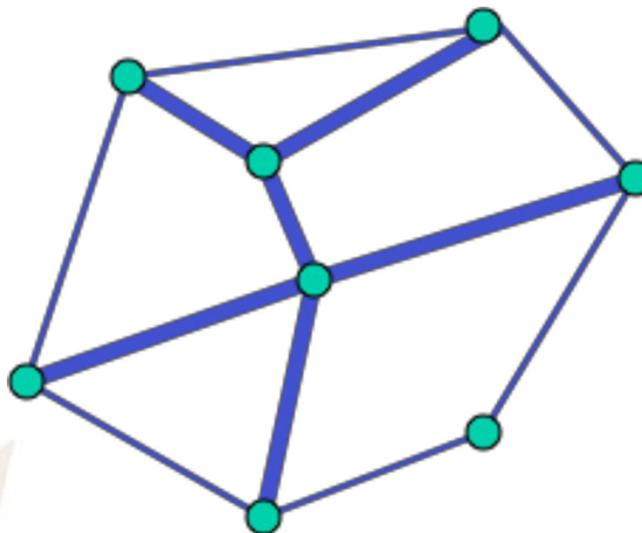
- Put the geometry in an array
- Use pointers from the vertices into this array
- Introduce a polygon list



Building Models

Shared Edges

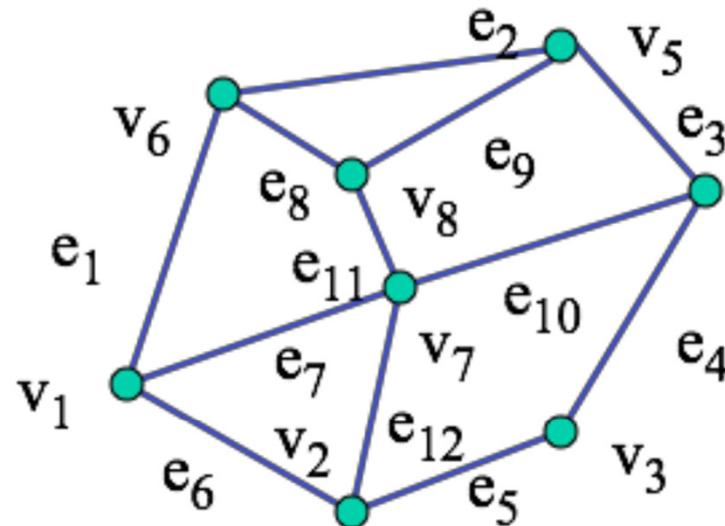
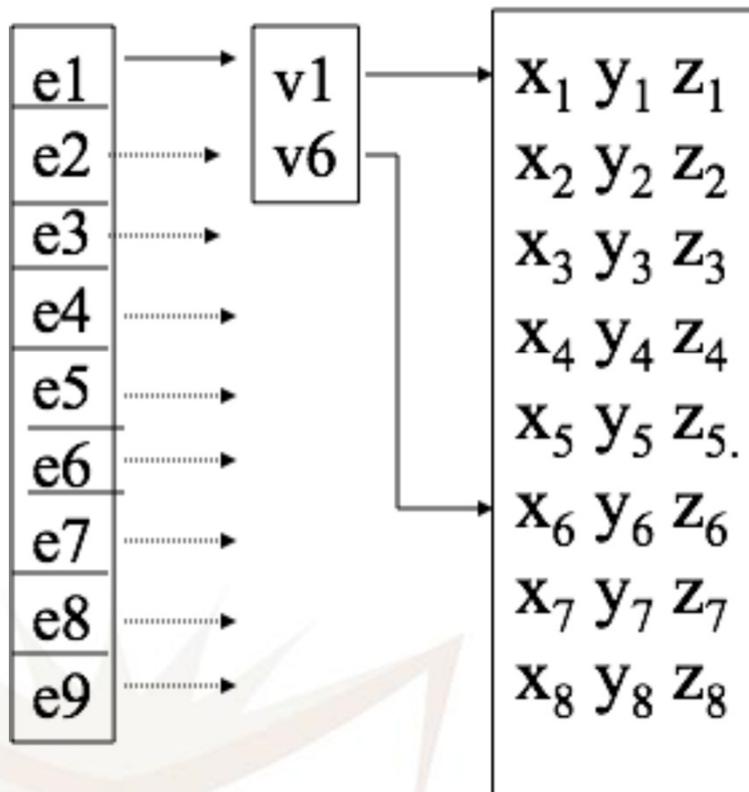
- Vertex lists will draw filled polygons correctly but if we draw the polygon by its edges, shared edges are drawn twice



- Can store mesh by edge list

Building Models

Edge List



Note polygons are
not represented

Building Models

Vertex Arrays

- OpenGL provides a facility called vertex arrays that allows us to store array data in the implementation
- attributes:
 - position, color, texture coordinates, indices, normals

Review

- ❑ Building models
- ❑ vertex, edge lists

Next Lecture

- ❑ Shadows with projections



COMP 371

Computer Graphics

Session 7

SHADOWS WITH PROJECTIONS



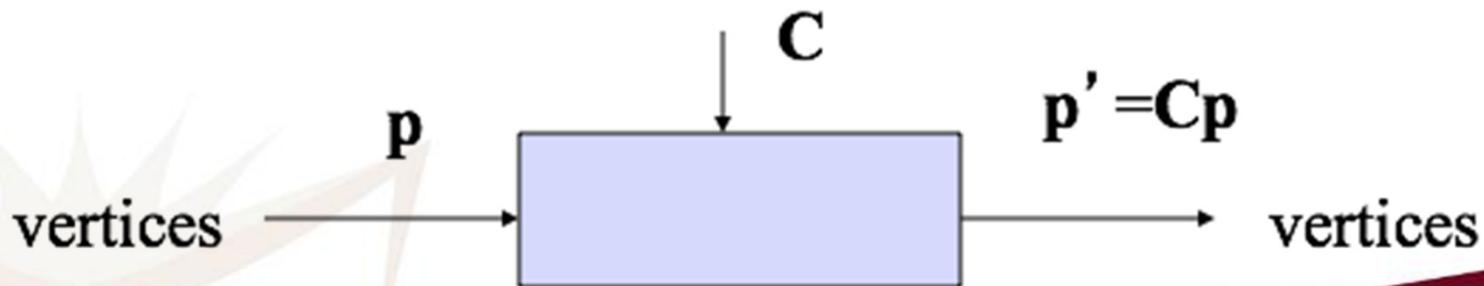
Lecture Overview

- Review of last class
- Shadow algorithms
 - shadows with projections
 - shadow maps

Recap

OpenGL Transformations

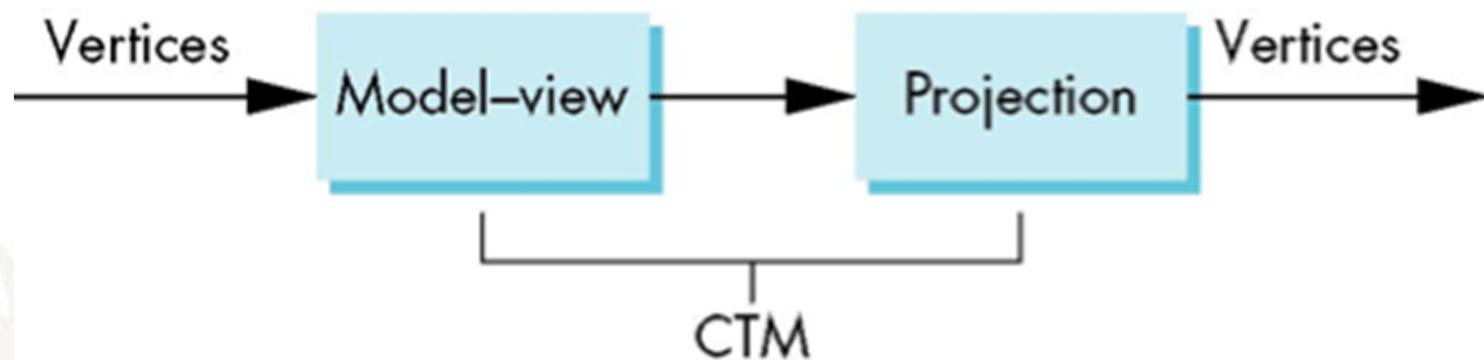
- Conceptually there is a 4×4 homogeneous coordinate matrix, the *current transformation matrix* (CTM) that is part of the state and is applied to all vertices that pass down the pipeline
- The **CTM** is defined in the user program and loaded into a transformation unit



Recap

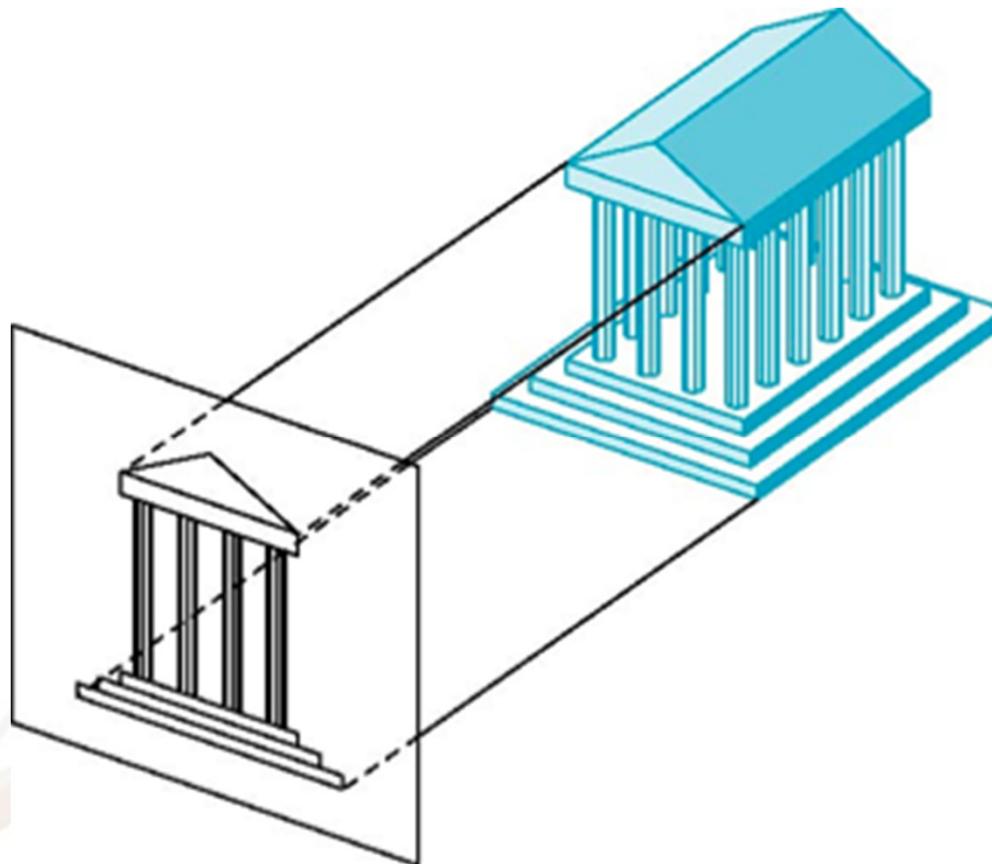
CTM in OpenGL

- OpenGL has a *model-view and a projection matrix* in the pipeline which were concatenated together to form the **CTM**
- We will emulate this process



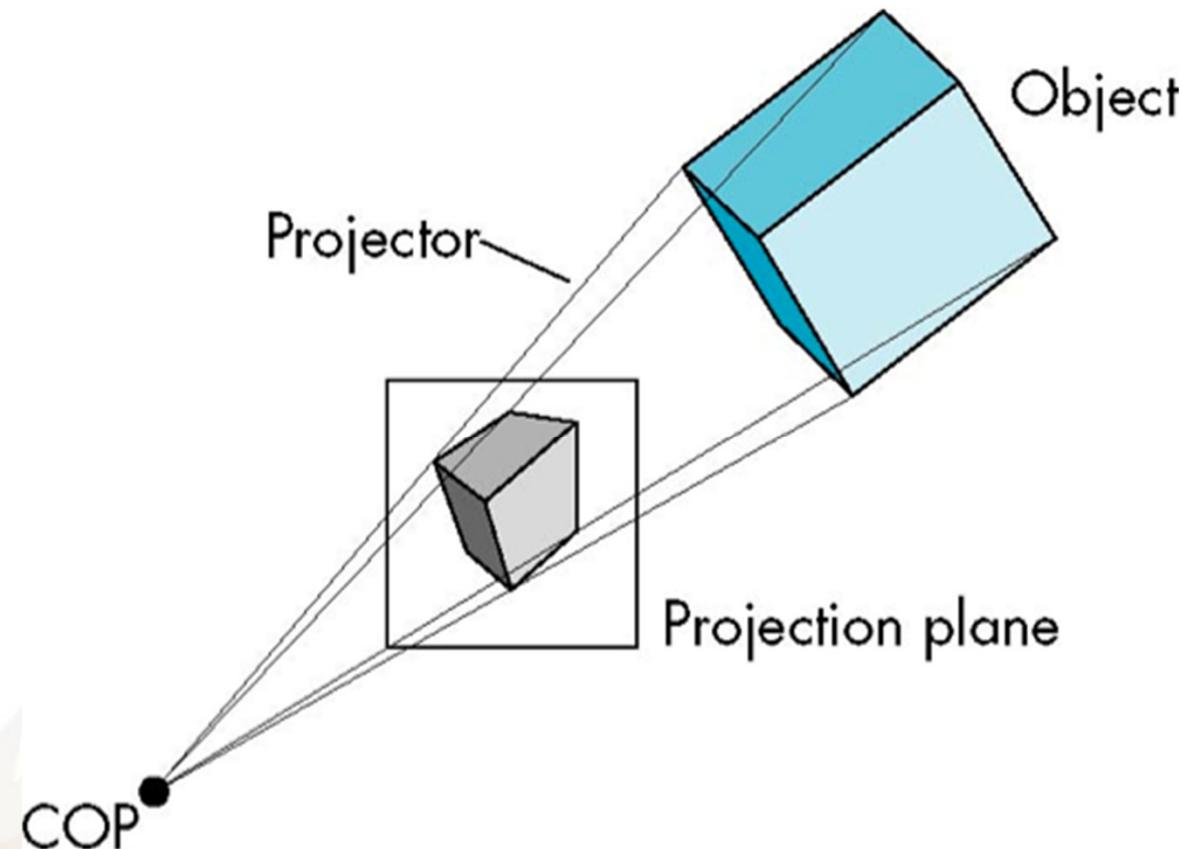
Recap

Orthographic Projection



Recap

Perspective Projection



Recap

Three-Point Perspective

- No principal face parallel to projection plane
- Three vanishing points for cube



Recap

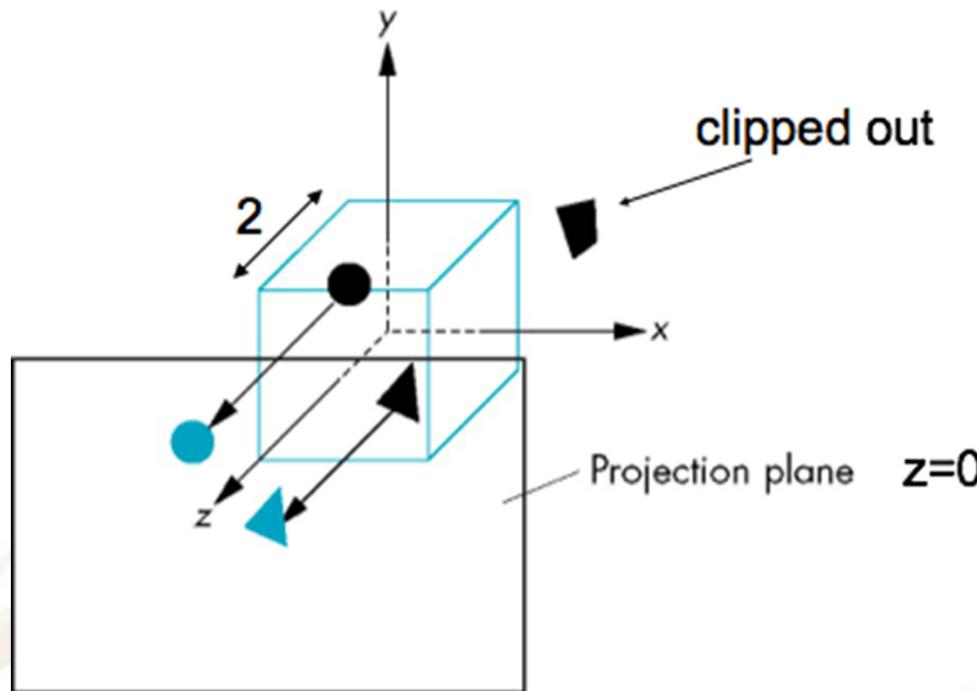
Computer Viewing

- There are three aspects of the viewing process, all of which are implemented in the pipeline,
 - Positioning the camera
 - Setting the model-view matrix
 - Selecting a lens
 - Setting the projection matrix
 - Clipping
 - Setting the view volume

Recap

Default Projection

- Default projection is orthogonal



Recap

The `glm::lookAt` Function

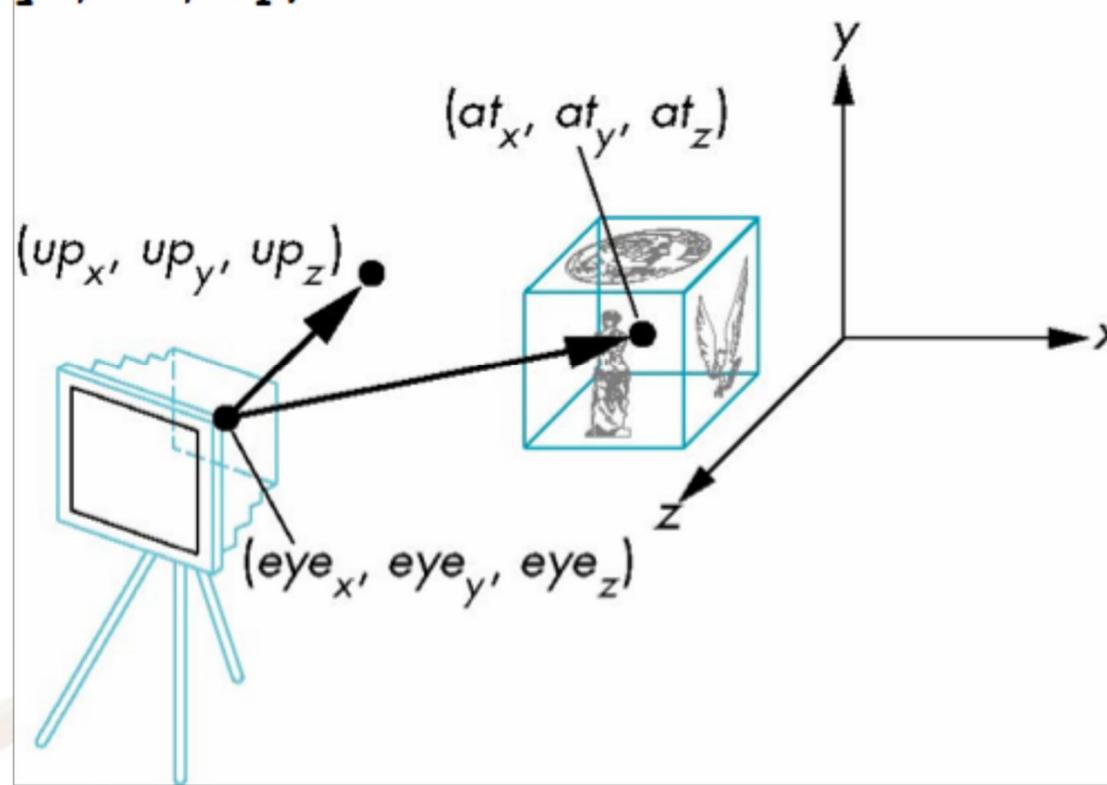
- The GLM library contains the function `glm::lookAt` to form the required model-view matrix through a simple interface
- Example:

```
glm::mat4 camera_matrix = glm::lookAt(glm::vec4 eye,  
                                     glm::vec4 at, glm::vec4 up);
```

Recap

The `glm::lookAt` Function

`LookAt(eye, at, up)`



Recap

Projections and Normalization

- The default projection in the eye (camera) frame is **orthogonal**
- For points within the default view volume

$$x_p = x$$

$$y_p = y$$

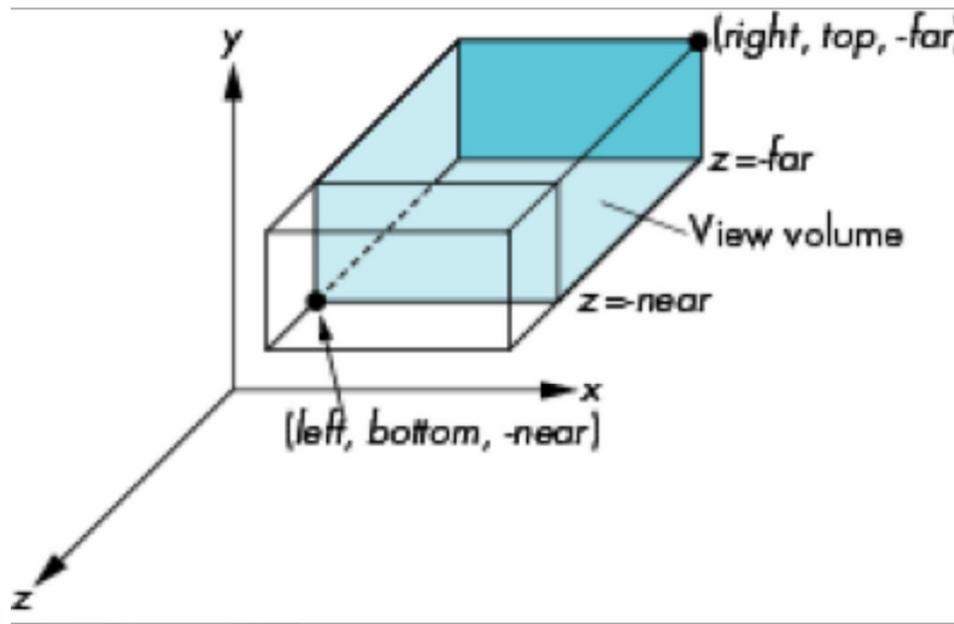
$$z_p = 0$$

- Most graphics systems use **view normalization**
 - All other views *are converted to the default view* by transformations that determine the projection matrix
 - Allows use of the same pipeline for all views

Recap

OpenGL Orthogonal Viewing

- `glm::ortho(left, right, bottom, top, near, far)`

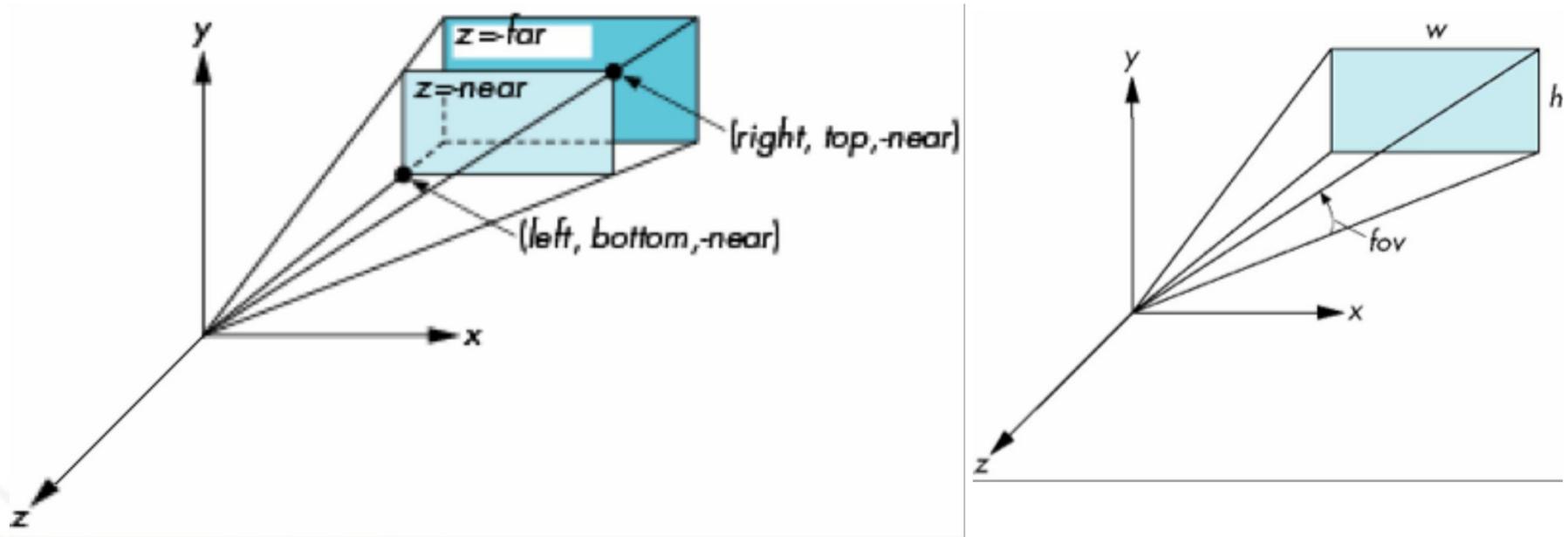


near and far measured from camera

Recap

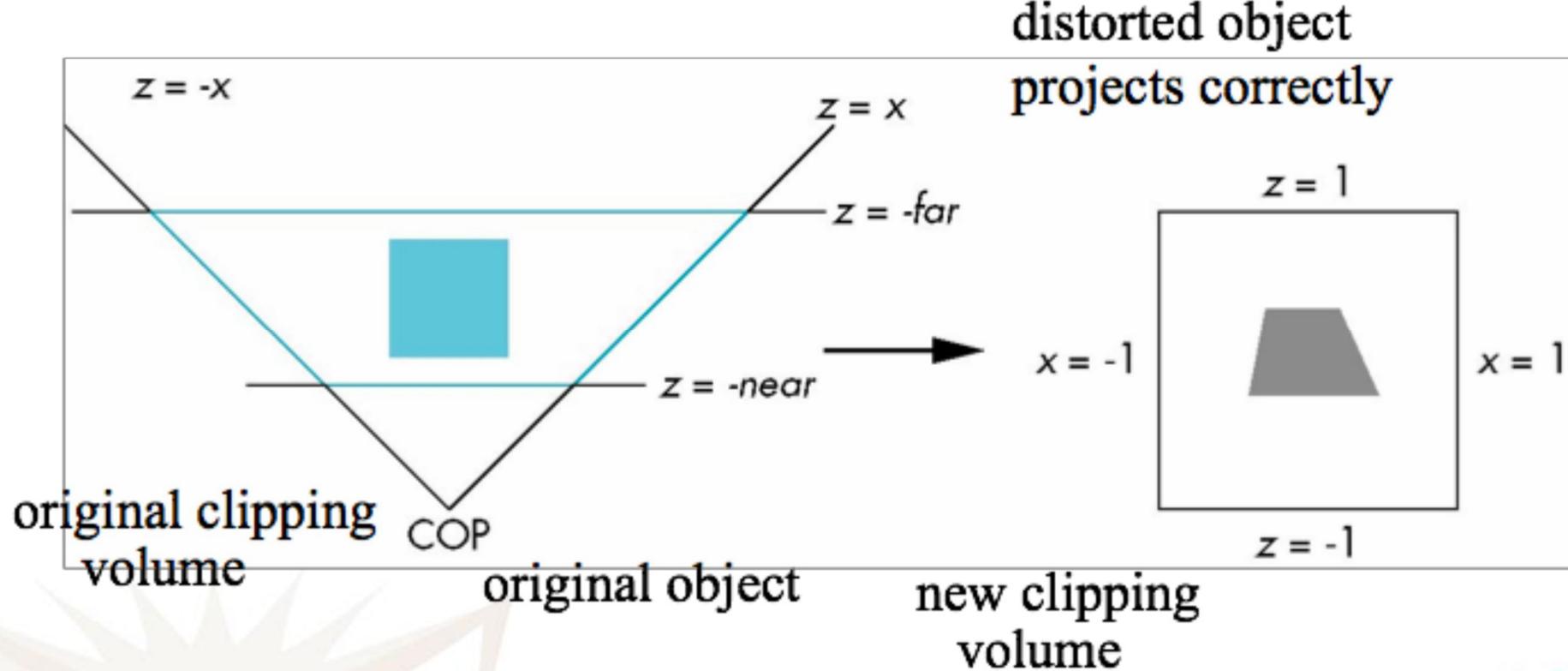
OpenGL Perspective Viewing

- **glm::perspective(fovy, aspect, near, far)**

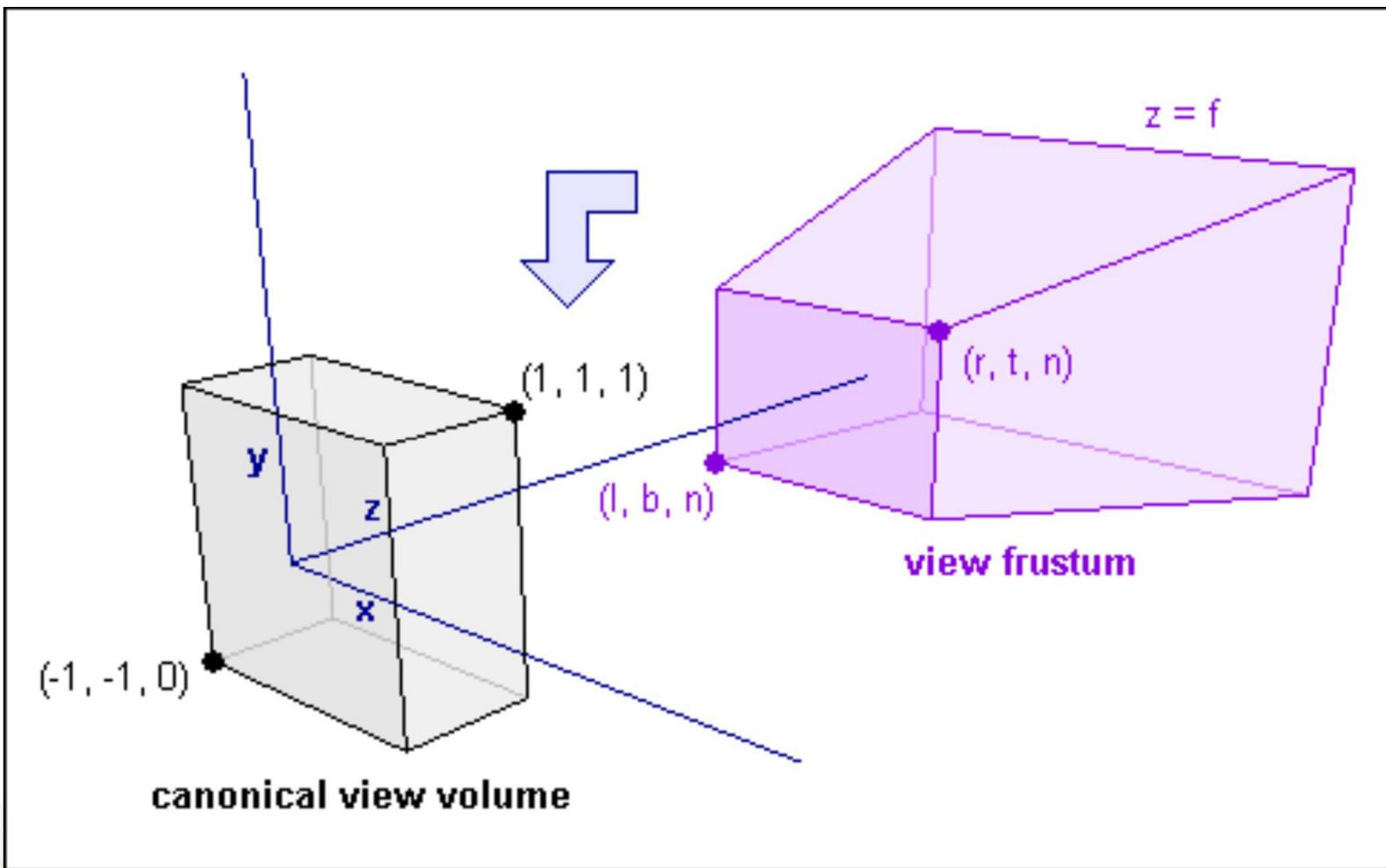


Recap

Normalization Transformation



Recap



Shadows

Importance of Shadows

- Important role in understanding
 - the **position** and **size** of the **occluder**
 - the **geometry** of the ***occluder***
 - the **geometry** of the ***receiver***

Shadows

Importance of Shadows



Image credit: [A survey of Real-Time Soft Shadows Algorithms](#)

Shadows

Importance of Shadows



Image credit: [A survey of Real-Time Soft Shadows Algorithms](#)

Shadows

Importance of Shadows

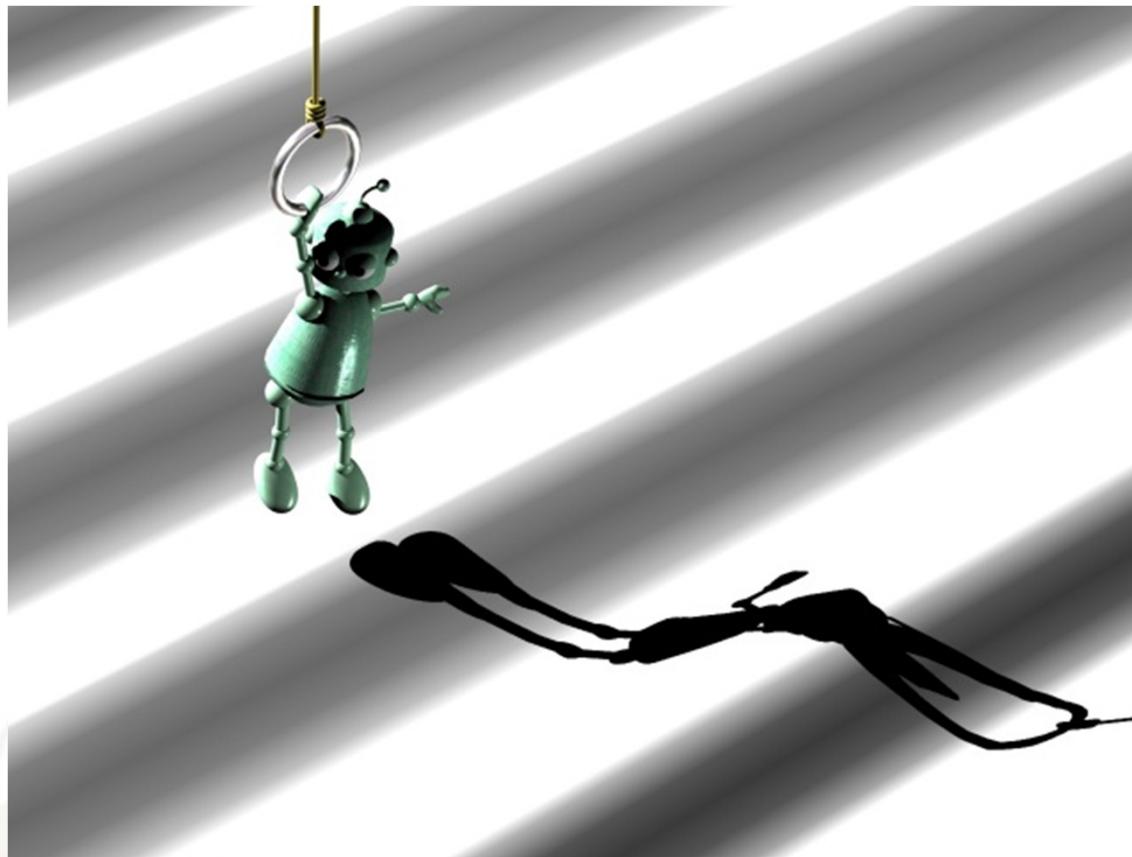


Image credit: [A survey of Real-Time Soft Shadows Algorithms](#)

Shadows

Importance of Shadows

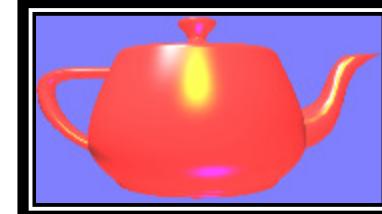
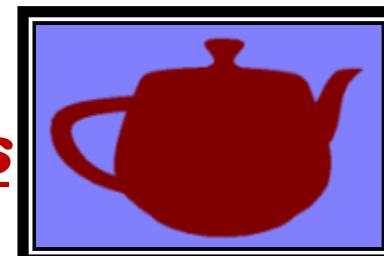


Image credit: [A survey of Real-Time Soft Shadows Algorithms](#)

Shadows

Need for Illumination?

- Important for perception and understanding of 3D scenes
- Has *visual cues for humans*
- Provides information about
 - Positioning of light sources
 - Characteristics of light
 - Sources
 - Materials
 - Viewpoint



Shadows

Light Sources

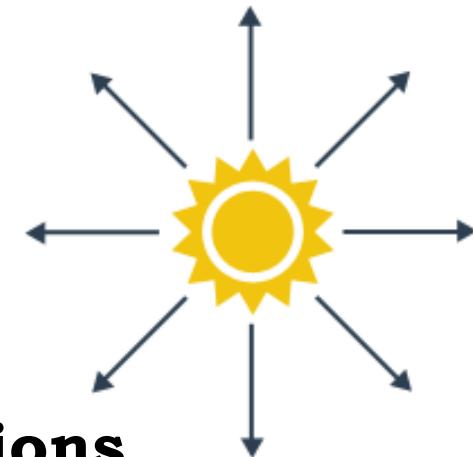
- **Point Light**
- **Spot Light**
- **Directional Light**
- **Area Light**

Shadows

Light Sources

Point Light

- **Isotropic**
 - light rays are emitted in all directions
- All light is emitted by a single point
- Easy to compute (defined just by position)
- Not realistic → Results in “hard” shadows
 - E.g. Light Bulb



Shadows

Light Sources

Spot Light

- **Anisotropic**
 - Emits light in a **cone of directions**
- **Extension of point light**
- **Easy to compute (defined by *position, direction and cone angle*)**
- **Dramatic Effects → “hard” shadows**
 - E.g. Theater Spot Light

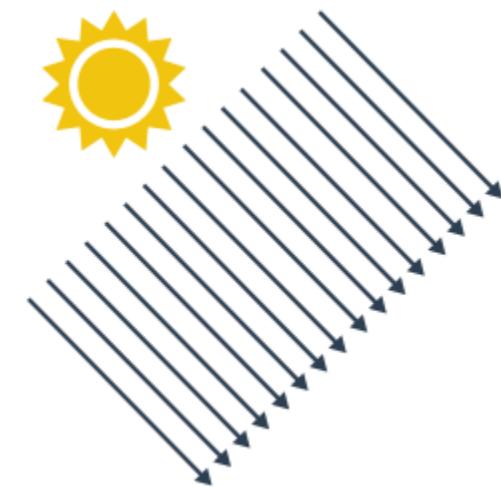


Shadows

Light Sources

Directional Light

- All light rays are parallel
- Light is located at infinity
(Described by *direction only, no position*)
- For light sources that are infinitely far away
- Intensity doesn't change based on distance
- Model **Sun light** → Results in “hard” shadows

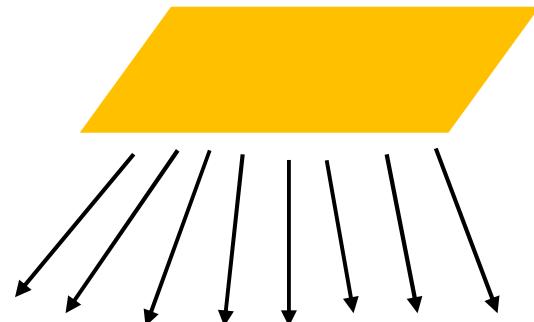


Shadows

Light Sources

Area Light

- **Realistic**
- **Light rays are emitted from a surface**
- **Computationally expensive (defined by *size, shape, position, intensity*)**
- **produces “soft” shadows and smoother lighting**
 - E.g. Photography Lights



Shadows

Need for Shadows?

- **Areas hidden from a light source**
- **Has visual depth cues for humans**
 - Provide info on relative locations
 - Provide info on relative motions
 - Provide info on shape of casting & receiving objects
 - Provide realism to scenes
- **Provides information about**
 - **Scene Lighting**
 - **Characteristics of light sources**

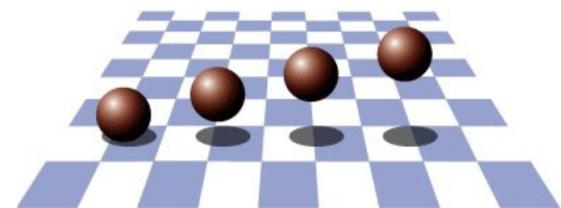
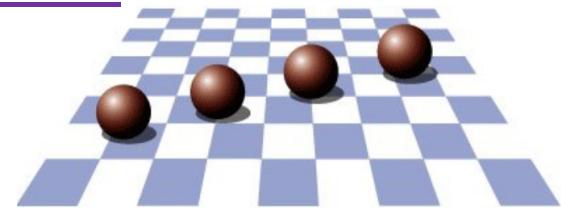


Image Courtesy: Stefanie Zollmann

<https://www.youtube.com/watch?v=XZO8-Df3mxA>

Shadows

Soft Vs Hard Shadows

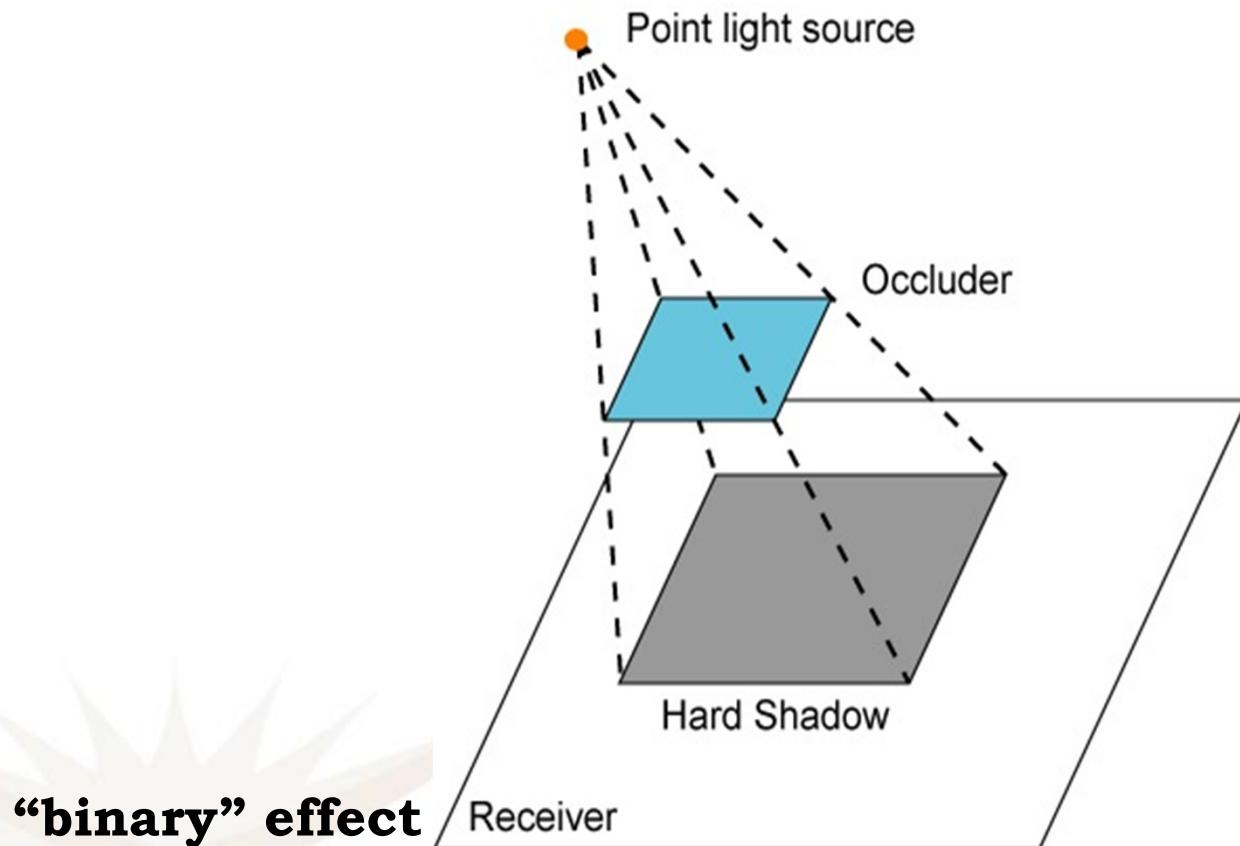


Image credit: [A survey of Real-Time Soft Shadows Algorithms](#)

Shadows

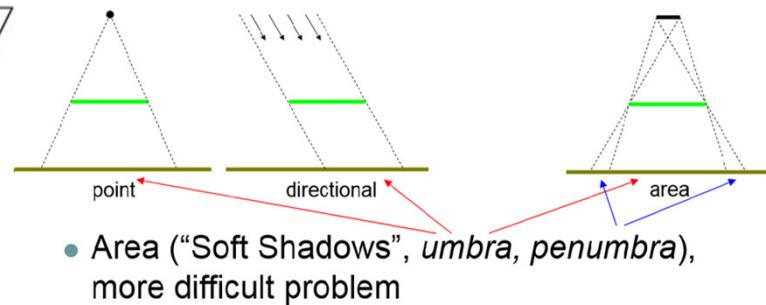
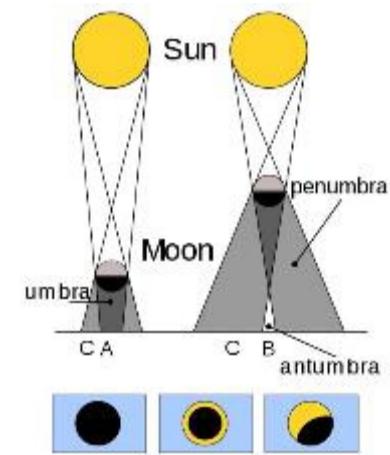
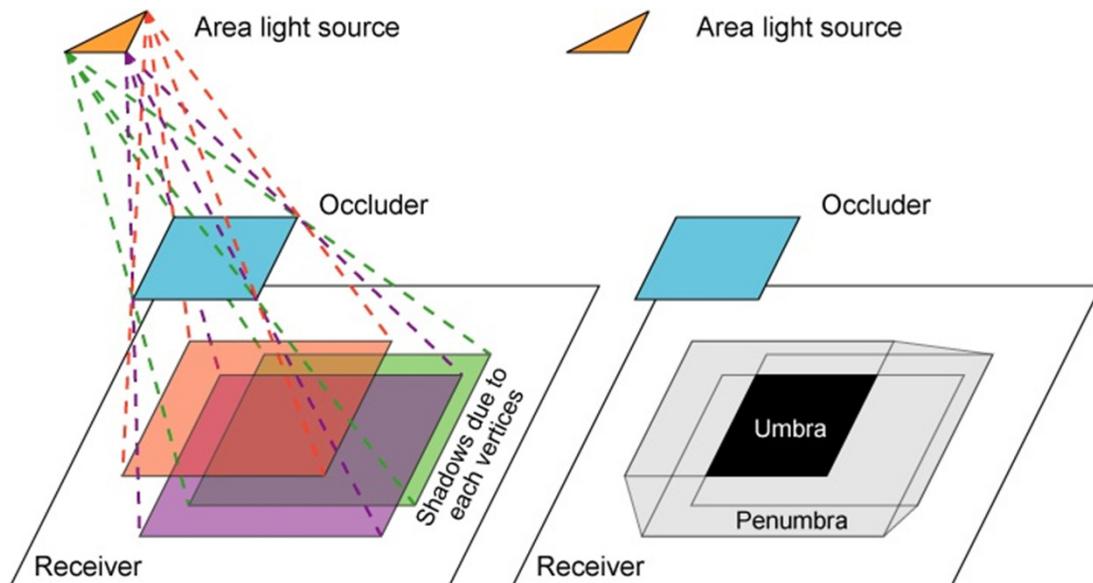
Soft Vs Hard Shadows



Image credit: [A survey of Real-Time Soft Shadows Algorithms](#)

Shadows

Soft Vs Hard Shadows



- **umbra** region (if it exists): light source is totally blocked from the receiver
- **penumbra** region in which the light source is partially visible
- The determination of the umbra and penumbra is a **difficult task in general**, as it amounts to solving visibility relationships in 3D

Image credit: [A survey of Real-Time Soft Shadows Algorithms](#)

Shadows

Soft Vs Hard Shadows



Image credit: [A survey of Real-Time Soft Shadows Algorithms](#)

Shadows

Soft Vs Hard Shadows

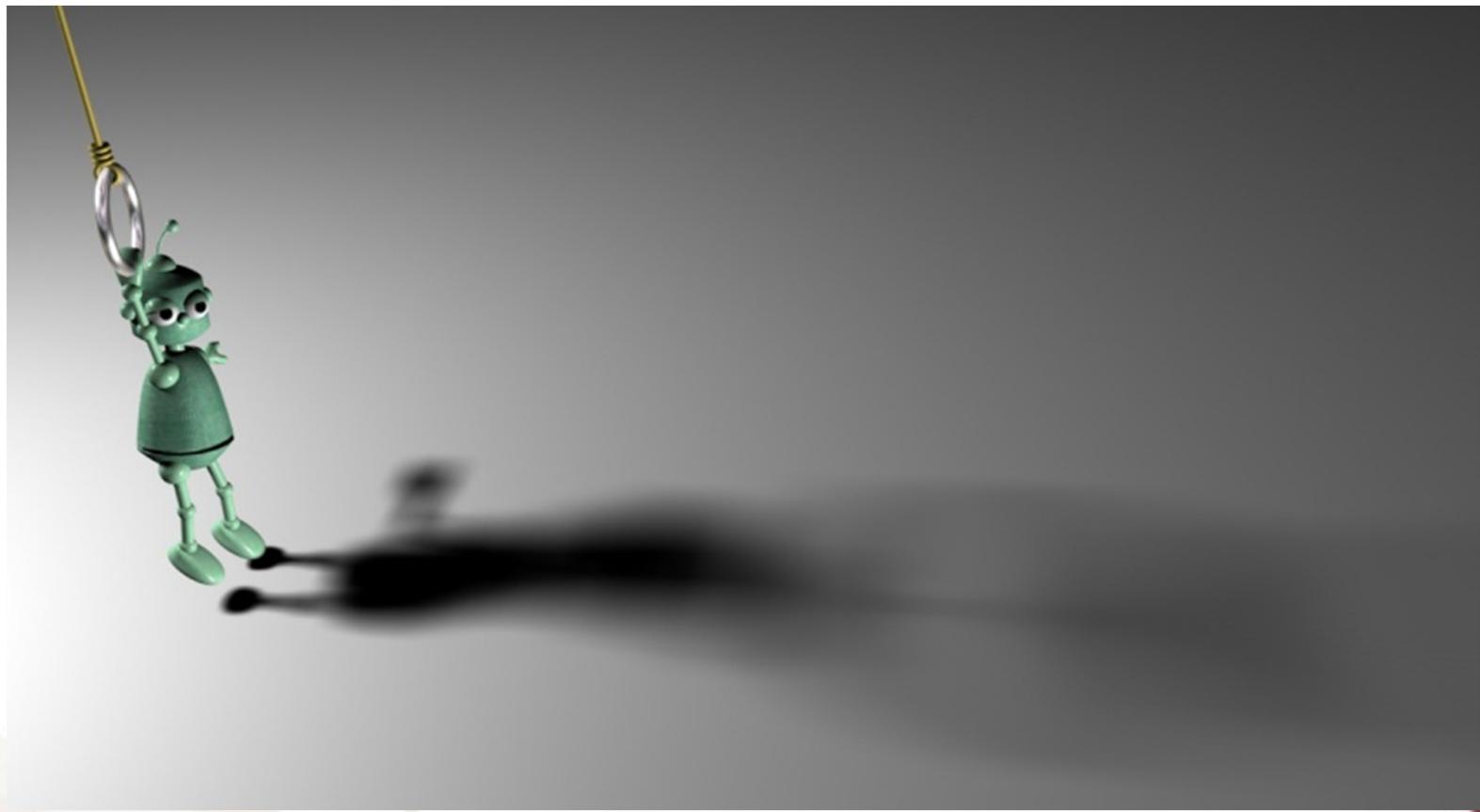
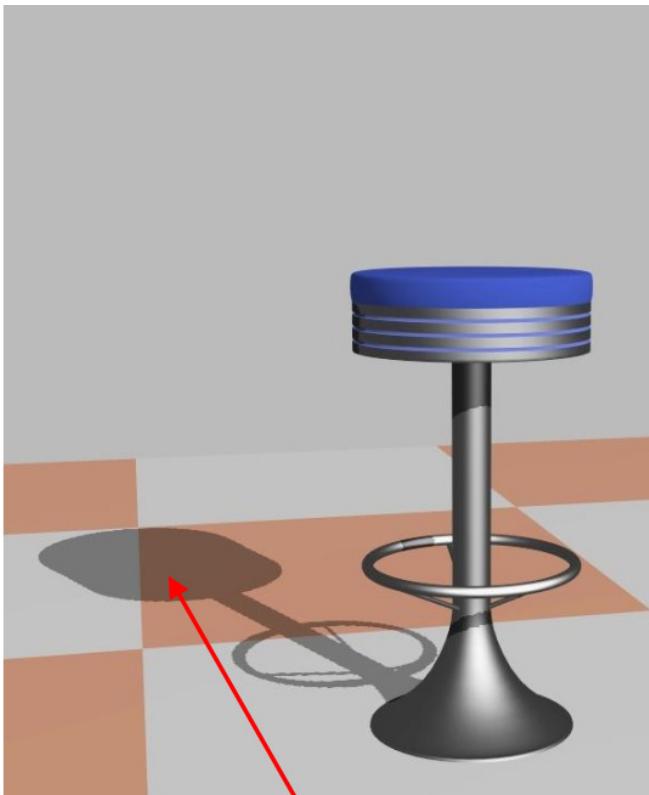


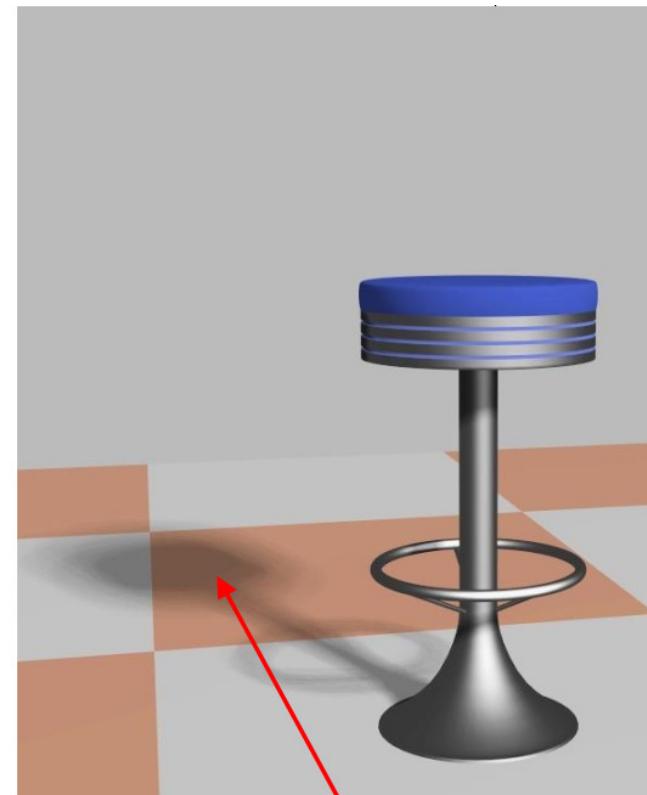
Image credit: [A survey of Real-Time Soft Shadows Algorithms](#)

Shadows

Soft Vs Hard Shadows



Hard Shadow



Soft Shadow

Shadows

Light in the Eye

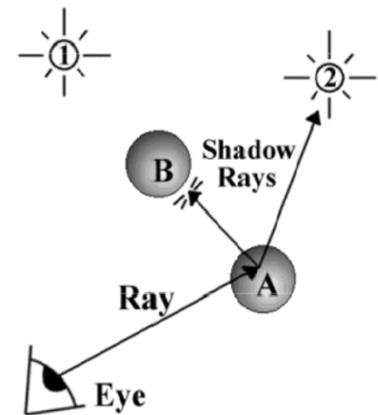
- When do we not see shadows in a real scene?
- When the only light source is a point source at the eye or center of projection
 - Shadows are behind objects and not visible
- Shadows are a global rendering issue
 - Is a surface visible from a source
 - May be obscured by other objects

Shadows

Shadow Algorithms

- With image level or local visibility tests

- Approximate and inexpensive
 - Projective (planar) shadows
 - Shadow maps
 - Shadow volume



- With world level global visibility tests

- Accurate yet expensive
 - Ray casting or ray tracing
 - Radiosity
 - Montecarlo simulation

Shadows

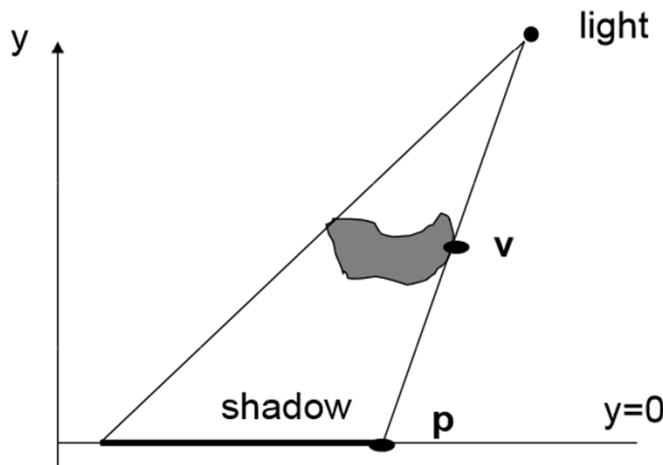
Projective Shadows

- Oldest methods
 - Used in flight simulators to provide visual clues
- Projection of an occluder polygon onto a receiver polygon is called a shadow polygon
- Given a point light source and a polygon, the vertices of the shadow polygon are the projections of the original polygon's vertices from a point source onto a receiver surface

Shadows

Projective Shadows

- Special case: the shadow receiver is an axis plane
 - Just project all the polygon vertices to that plane and form shadow polygons



Given:

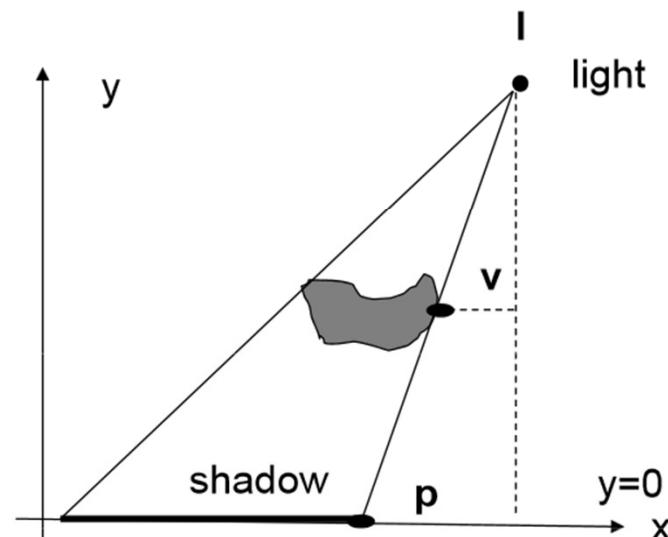
- Light position \mathbf{l}
- Plane position $y = 0$
- Vertex position \mathbf{v}

Calculate: \mathbf{p}

Shadows

Projective Shadows

- We can use similar triangles to solve p



$$\frac{p_x - l_x}{v_x - l_x} = \frac{l_y}{l_y - v_y}$$

$$p_x = \frac{l_y v_x - l_x v_y}{l_y - v_y}$$

- Same principle applied to different axis planes

Shadows

Projective Shadows

- **Projection matrix for plane $y = 0$**

$$M = \begin{bmatrix} l_y & -l_x & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & -l_z & l_y & 0 \\ 0 & -1 & 0 & l_y \end{bmatrix}$$

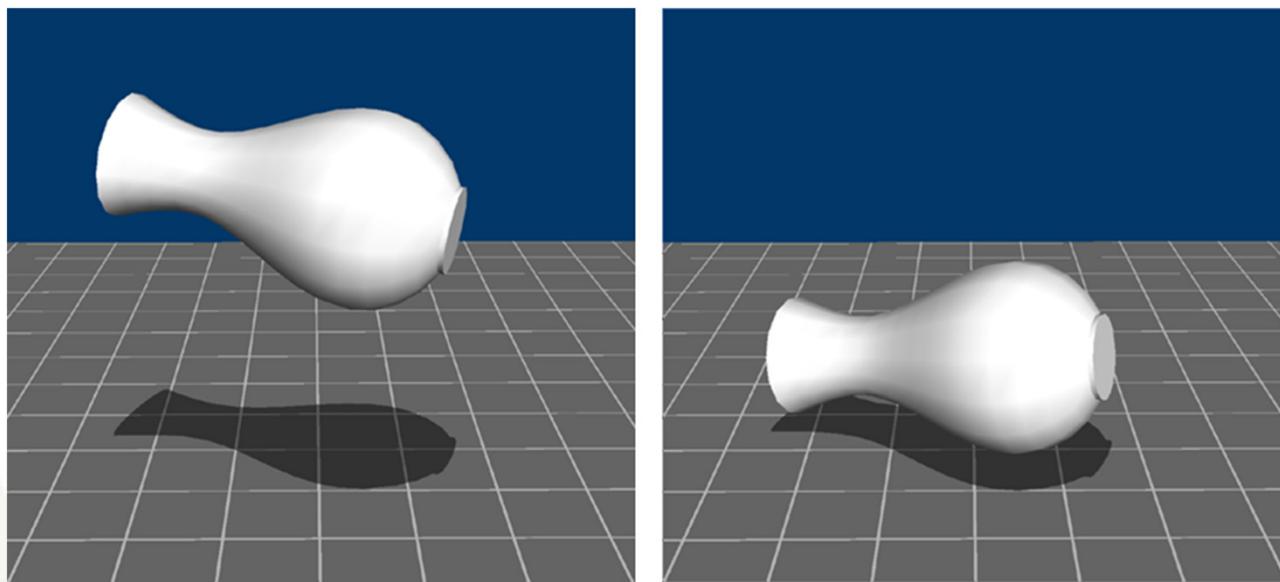
- **Projection matrix for casting a shadow on a general plane $n \cdot x + d = 0$**

$$M = \begin{pmatrix} n \cdot l + d - l_x n_x & -l_x n_y & -l_x n_z & -l_x d \\ -l_y n_x & n \cdot l + d - l_y n_y & -l_y n_z & -l_y d \\ -l_z n_x & -l_z n_y & n \cdot l + d - l_z n_z & -l_z d \\ -n_x & -n_y & -n_z & n \cdot l \end{pmatrix}$$

Shadows

Cast Shadows on Planar Surfaces

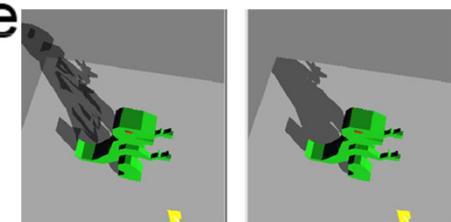
- Render (draw) the object polygons and then the shadow polygons (with suitable alpha value to make them look dark)



Shadows

Projective Shadows (Limitations)

- Shadow polygon generation (z fighting)
 - Add an offset to the shadow polygons (`glPolygonOffset`)
 - Draw receivers first, turn z-test off, then draw the shadow polygons. After this, draw the rest of the scene.
- Shadow polygons fall outside the receiver
 - Using stencil buffer – draw receiver and update the stencil buffer
- Shadows have to be rendered at each frame
 - Render into texture
- Restrictive to planar objects
 - e.g. car on street, player on ground*

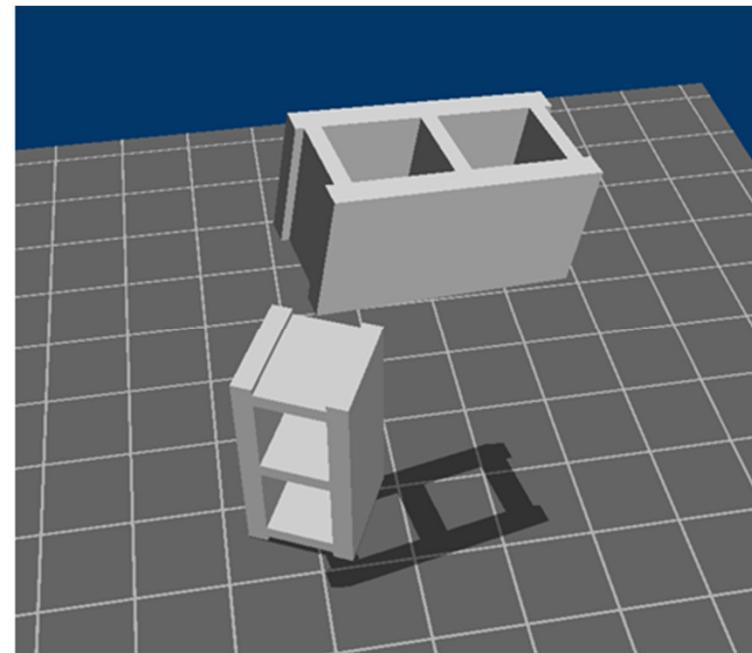
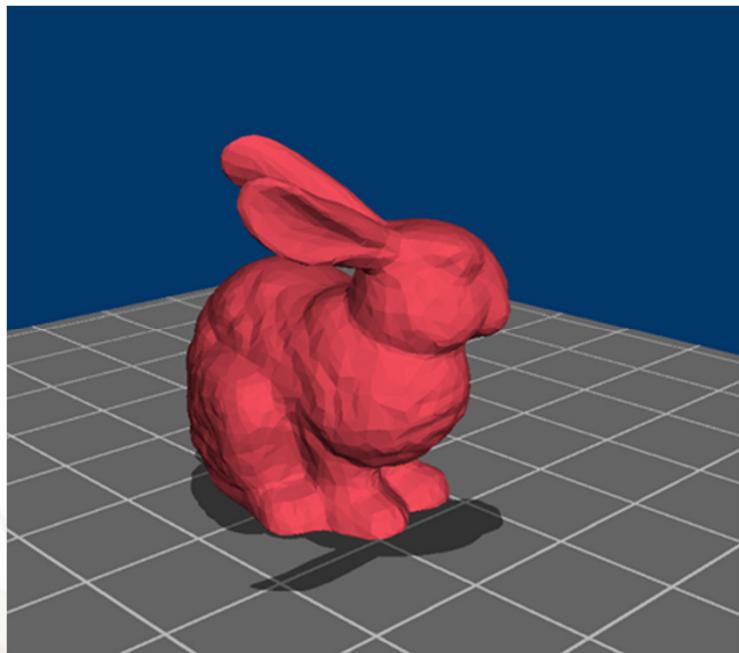


No self-shadows

Shadows

Projective Shadows (Limitations)

- Does not produce self-shadows, shadows cast on other objects, shadows on curved surfaces, etc.



Shadows

SHADOW/VIEW DUALITY

- A point is in shadow if it not visible from the perspective of the light source
- Use the view from the light source to compute shadows

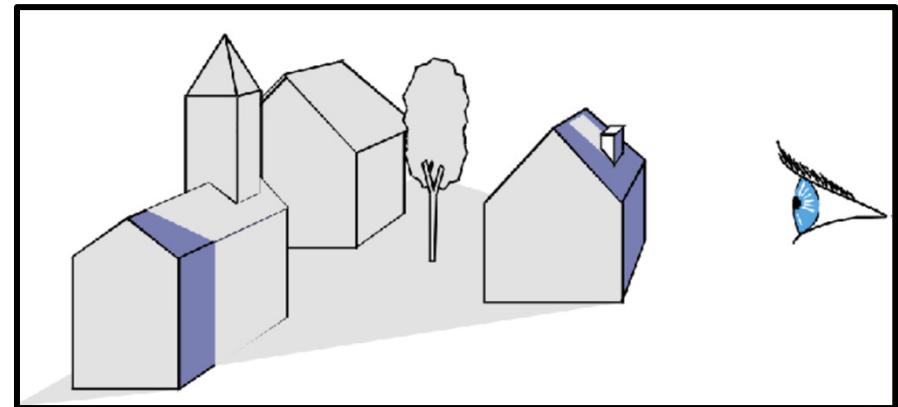
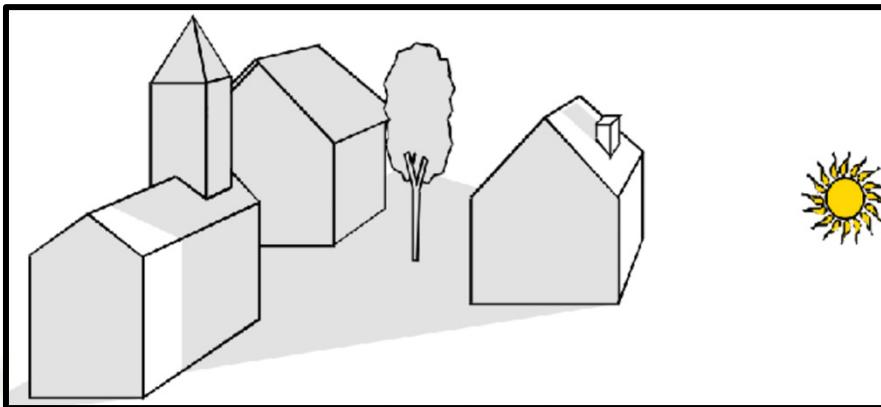
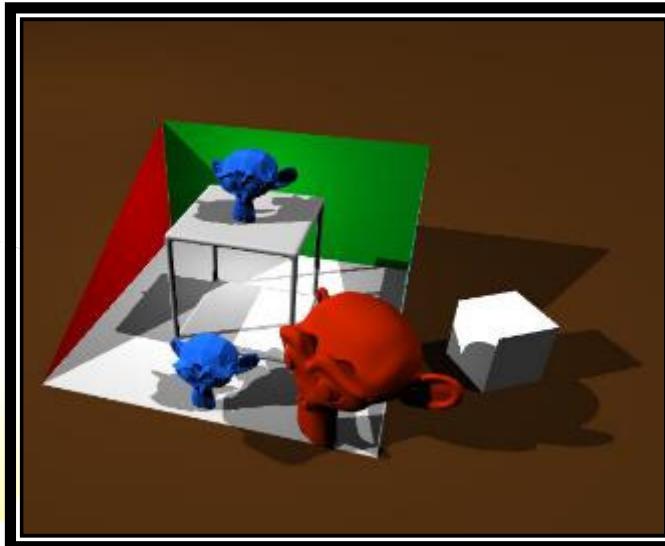


Image Courtesy: Stefanie Zollmann

Shadows

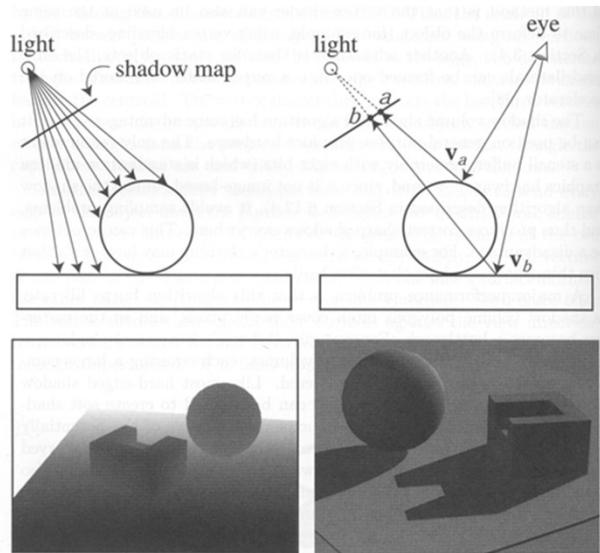
Shadow Mapping



Shadows

Shadow Mapping

- Requires 2 passes through the pipeline
- Compute shadow map (storing depth from light source)
- Render final image, check shadow map to see if points are in shadow

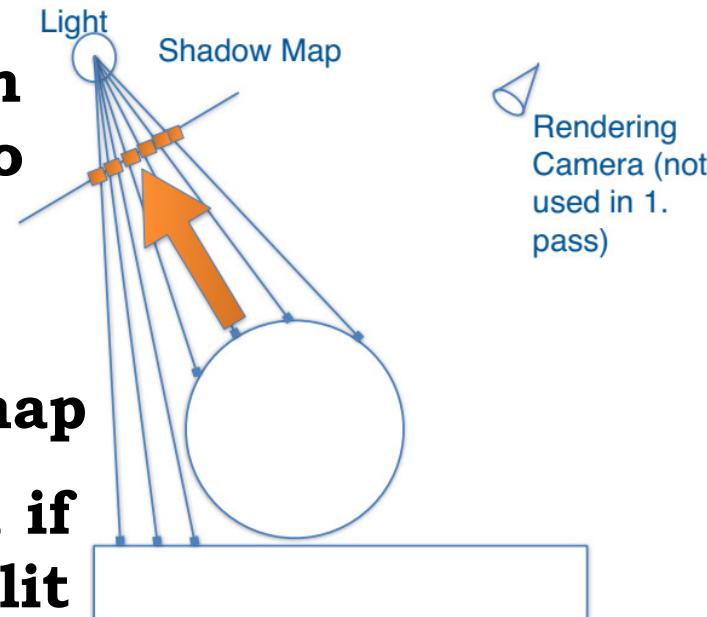


- Shadow map calculation is independent of eye position
- Shadow map is loaded once
- If eye moves, no need for recalculation; If objects move, recalculation required

Shadows

Shadow Maps (First Pass)

- Render a scene from a light source, depth buffer will contain the distances from the source to each fragment
- Store these depths in a texture called a depth map or shadow map
- Image in shadow map indicates, if rendered with a light, anything lit is not in shadow
- Transform the geometry into light-view space



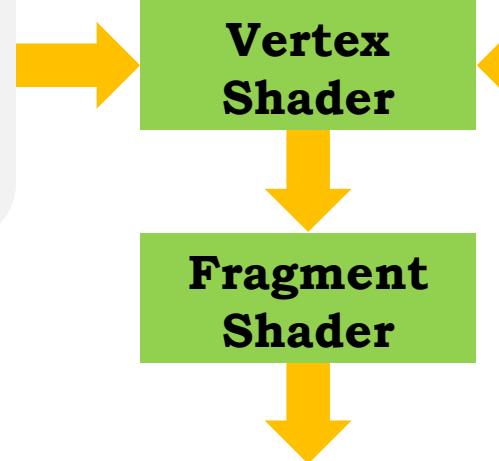
Shadows

Shadow Mapping (First Pass)

Position Transform

Light Space
View

Light Space
Projection



World Space Scene



Depth Buffer

Credit

Shadows

Shadow Mapping (First Pass)

Position Transform

Light Space View

Light Space Projection

```
// depthMVP contains Model-View-Projection Matrix  
// for light source  
uniform mat4 depthMVP;
```

```
void main(){  
    gl_Position = depthMVP * vertexPosition;  
}
```

Depth Buffer



World Space Scene



Credit

Shadows

Shadow Mapping (First Pass)

Position Transform

Light Space
View

Light Space
Projection



Vertex
Shader



World Space Scene

Fragment
Shader



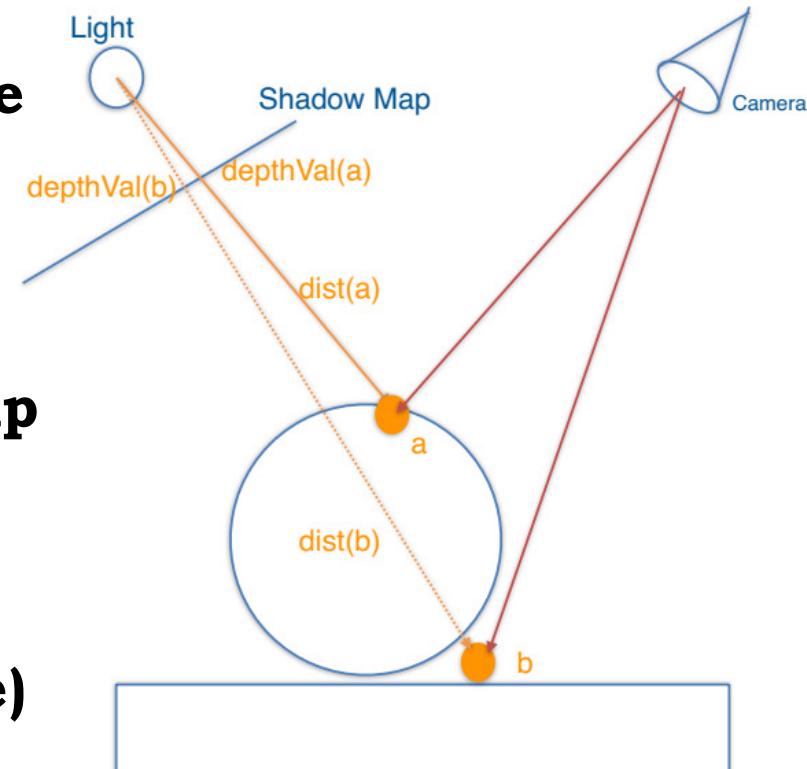
Depth Buffer

Credit

Shadows

Shadow Mapping (Second Pass)

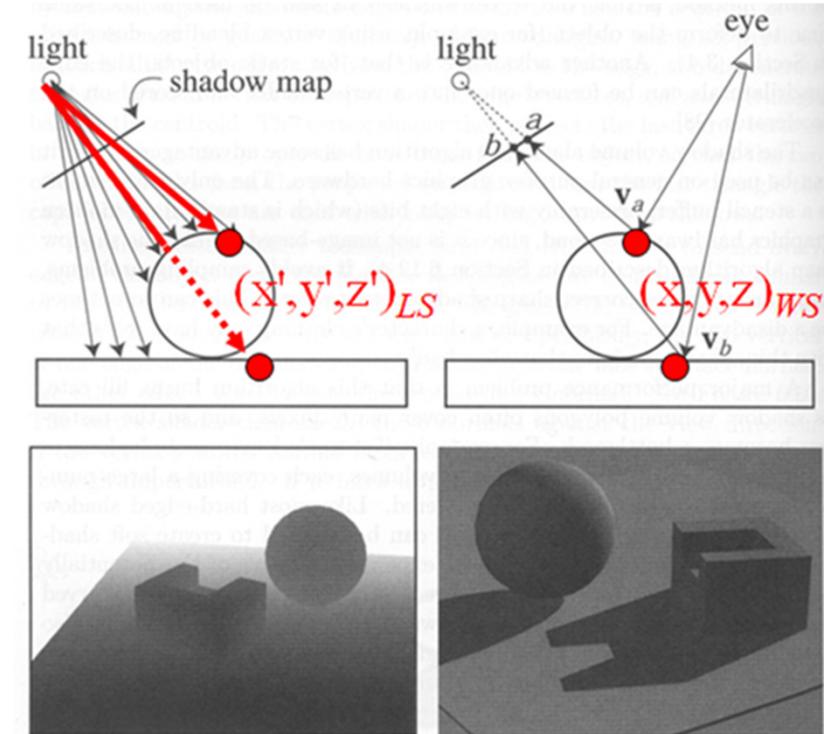
- During the final rendering, compare the distance from the fragment to the light source with the distance in the shadow map
- If the depth in the shadow map is less than the distance from the fragment to the light source the fragment is in shadow (from this light source)
- Otherwise, use rendered color



Shadows

Shadow Mapping (Second Pass)

- We have a 3D point $(x, y, z)_{WS}$
- How do we look up the depth from the shadow map?
- Use the 4x4 perspective projection matrix from the light source to get $(x', y', z')_{LS}$
 - If $\text{ShadowMap}(x', y') < z'$ then point is in shadow
 - If $\text{ShadowMap}(x', y') \geq z'$ then point is lit



Shadows

Shadow Mapping (Second Pass)

Vertexshader:

```
// Output position of the vertex, in clip space : MVP * position  
gl_Position = MVP * vec4(vertexPos_modelspace, 1);  
  
// Same, but with the light's view matrix  
shadowCoord = textureSpace * DepthMVP * vec4(vertPos_modelspace, 1);
```

Fragmentshader:

```
float visibility = 1.0;  
float depthVal = texture(shadowMap, shadowCoord.xy).z;  
float dist = shadowCoord.z;  
if (dist > depthVal){  
    visibility = 0.0;  
}
```

```
outputColor =  
// Ambient : simulates indirect lighting  
ambientColor +  
// Diffuse : "color" of the object  
visibility * diffuseColor  
// Specular : reflective highlight, like a mirror  
visibility * specularColor;
```

- **Transform points in light space**
- **Transform points in shadow map**
- **Compute visibility based on distance**
- **Use Vertex and Fragment Shader to perform these operations**

Credit

Shadows

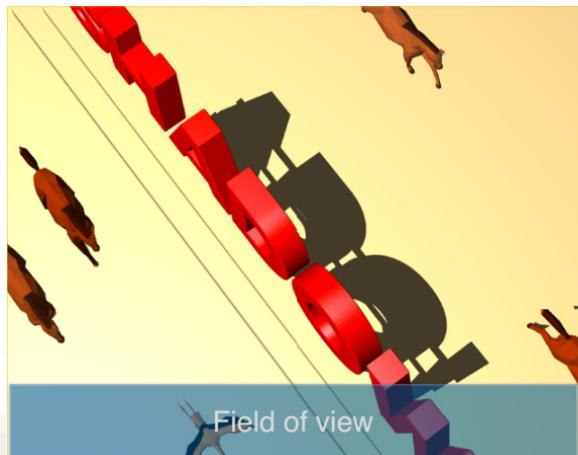
Shadow Mapping



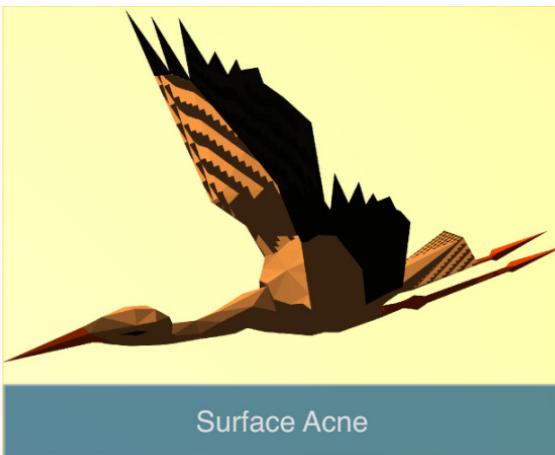
Shadows

Shadow Mapping (Limitations)

- Field of View
- Surface Acne
- Aliasing



Field of view



Surface Acne



Shadow Map Aliasing

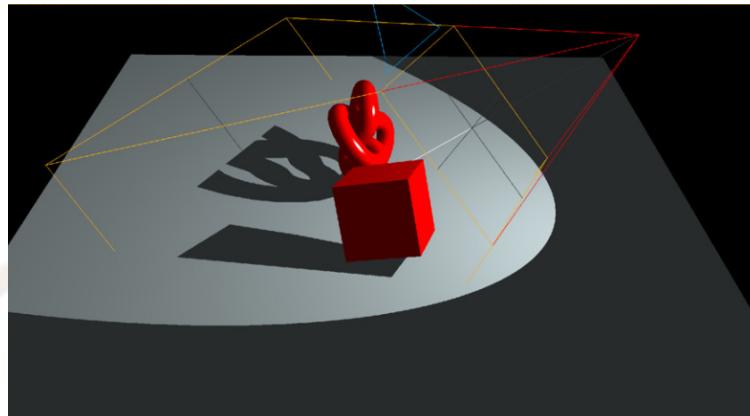
Credit

Shadows

Shadow Mapping (Limitations)

Field of View

- What if object is outside field of view of shadow map?
- No shadows or partial shadows
- For spotlights, can be changed by tweaking its range
- Problem, in particular, for larger scenes



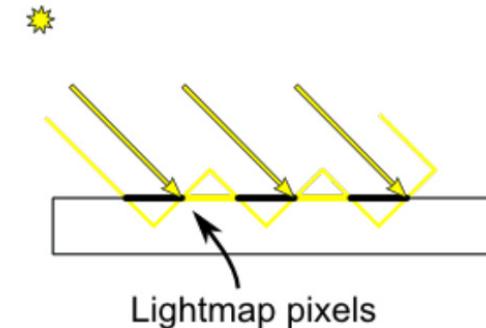
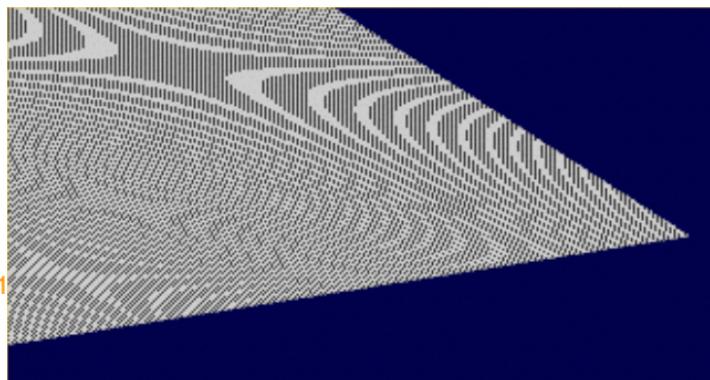
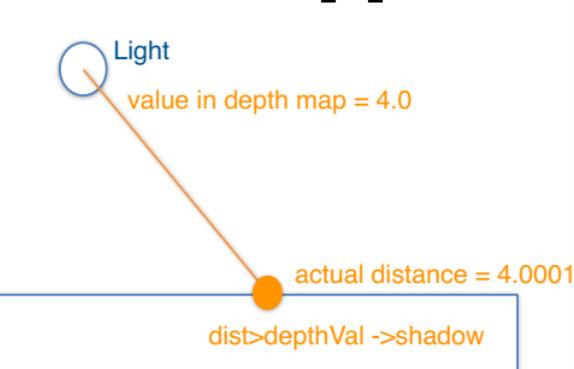
Credit

Shadows

Shadow Mapping (Limitations)

Surface Acne

- Self-shadowing problem due to precision and depth map resolution
 - Depth value in map can differ from actual distance between object and light source
 - Sampling problem: neighboring vertices map to the same depth map pixel



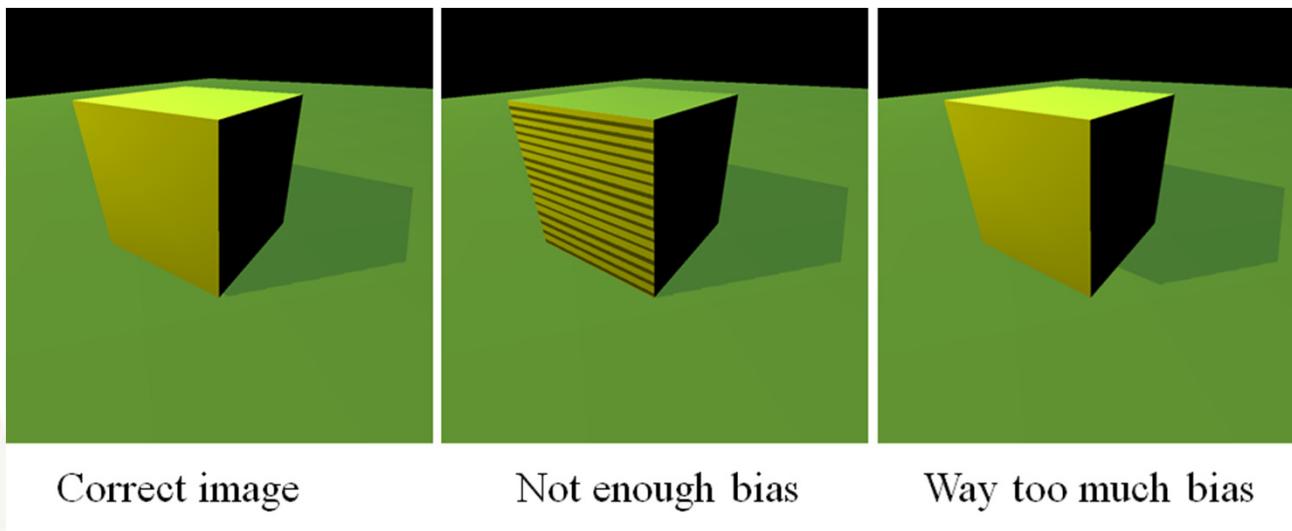
Credit

Shadows

Shadow Mapping (Limitations)

Surface Acne (Solution)

- **Shadow map bias for shadow test:**
 $\text{ShadowMap}(x,y) + \text{bias} < \text{dist}$



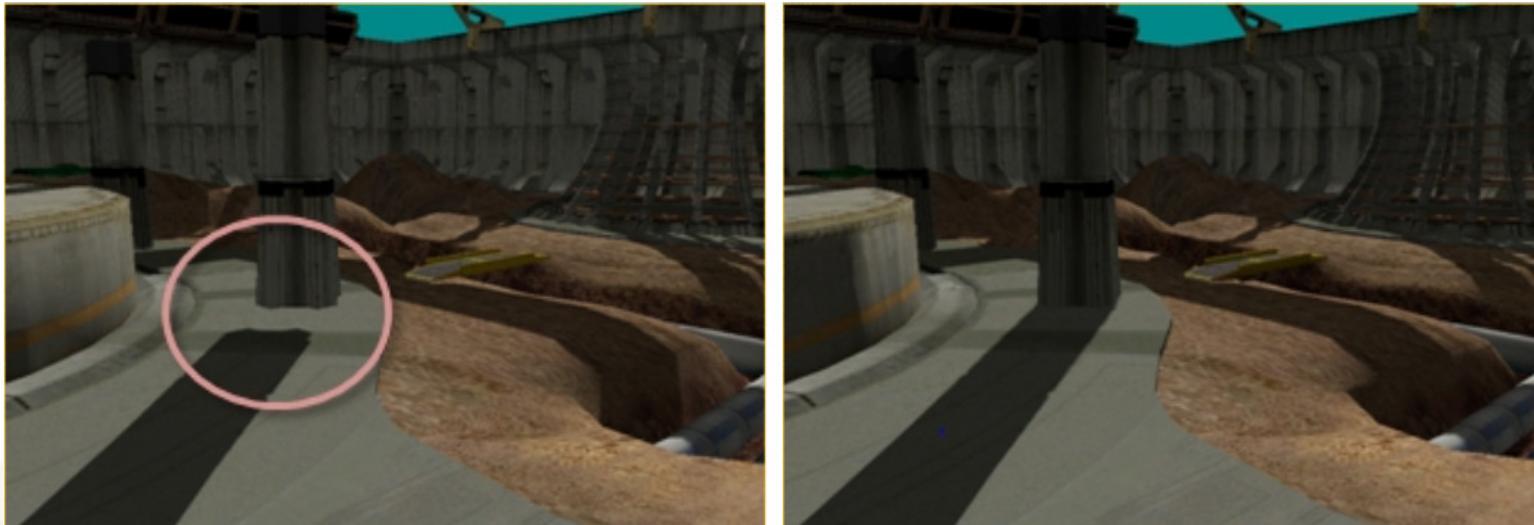
Credit

Shadows

Shadow Mapping (Limitations)

Surface Acne (Solution)

- Choosing a good bias value can be tricky - otherwise Peter Panning



Credit

Shadows

Shadow Mapping (Limitations)

Shadow Map Aliasing

- Quality depends on shadow map resolution
- Higher resolution:
 - Higher quality
 - More memory required



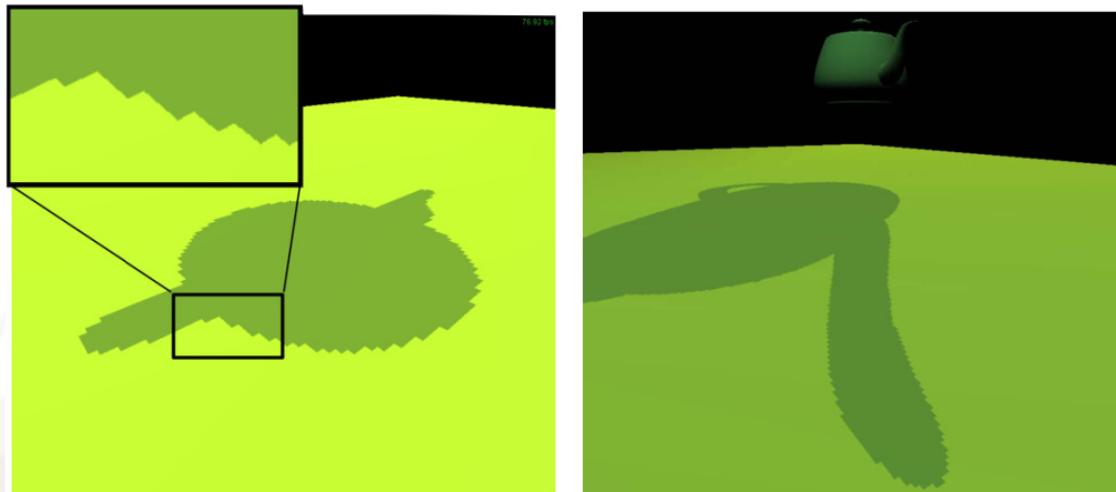
Credit

Shadows

Shadow Mapping (Limitations)

Shadow Map Aliasing

- Under-sampling of a shadow map (Jagged shadow edges)
- Re-projection aliasing – especially bad when the camera & light are opposite each other



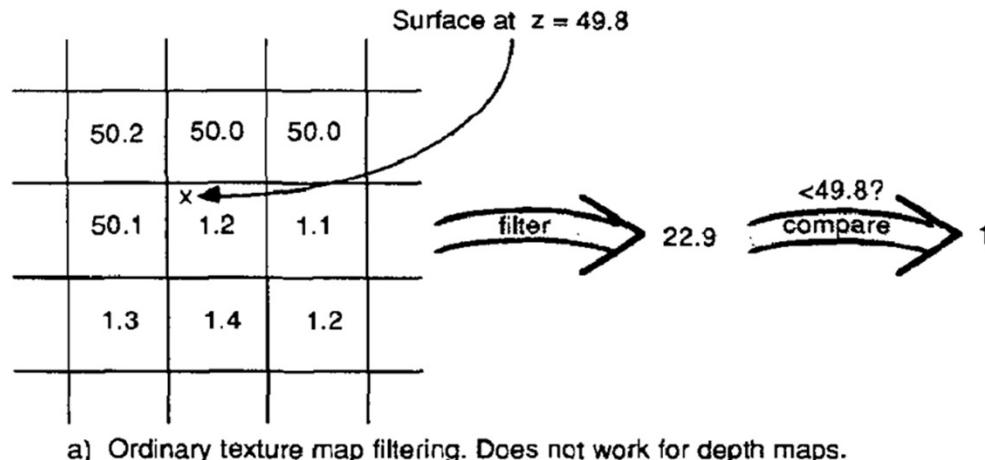
Credit

Shadows

Shadow Mapping (Limitations)

Percentage Closer Filtering (Solution)

- Should we filter the depth? (weighted average of neighboring depth values)
- No... filtering depth is not meaningful



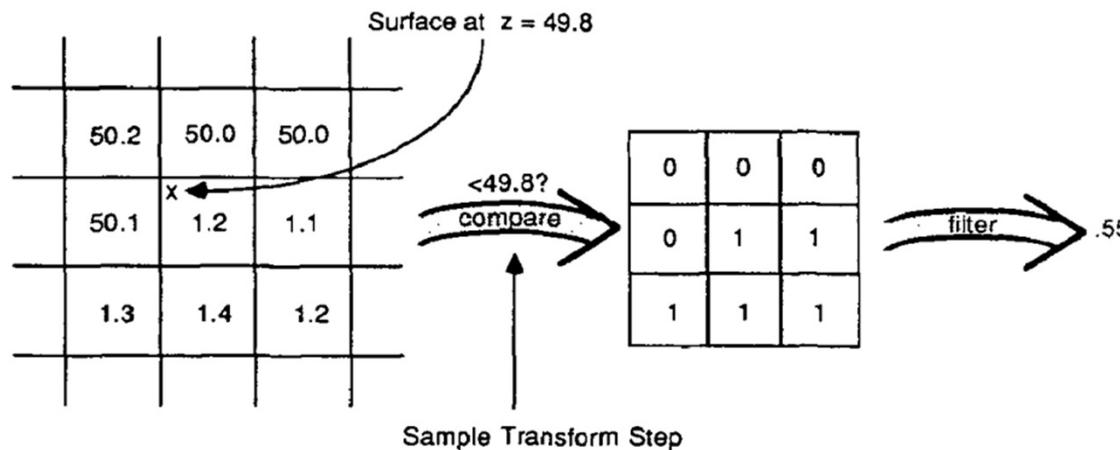
Credit

Shadows

Shadow Mapping (Limitations)

Percentage Closer Filtering (Solution)

- Instead filter the result of the test (weighted average of comparison results)
- But makes the bias issue more tricky



Credit

Shadows

Shadow Mapping (Limitations)

Percentage Closer Filtering (Solution)

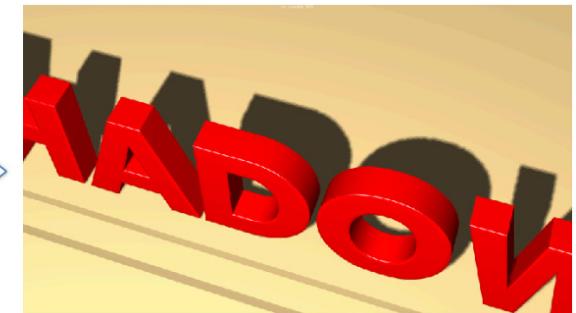
- Quality increases with Percentage closer filtering
- Bigger Filter → Fake softer appearance of shadows



1024x1024



1024x1024



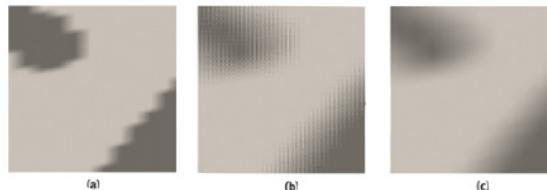
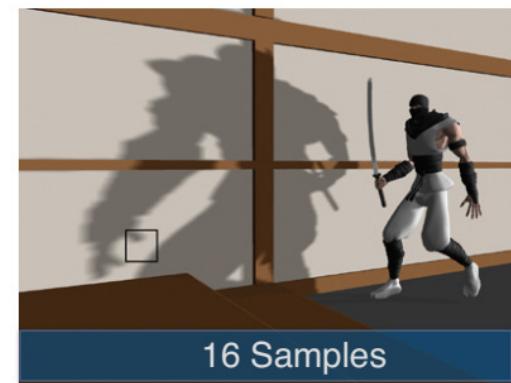
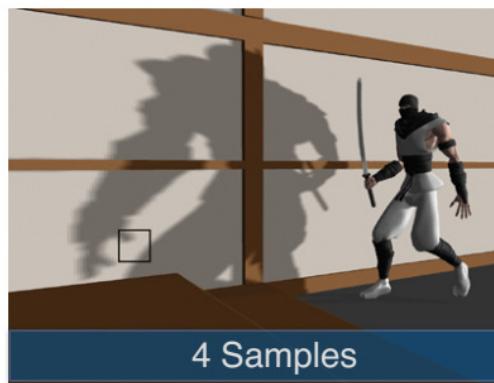
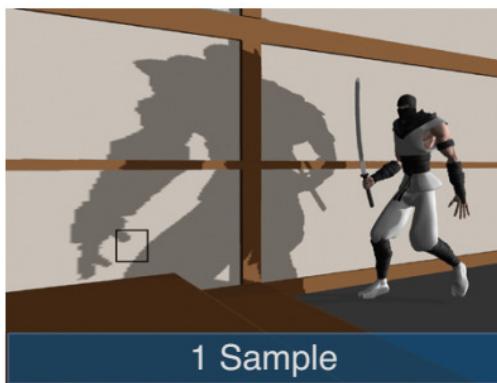
Credit

Shadows

Shadow Mapping (Limitations)

Percentage Closer Filtering (Solution)

- Results for different filter size



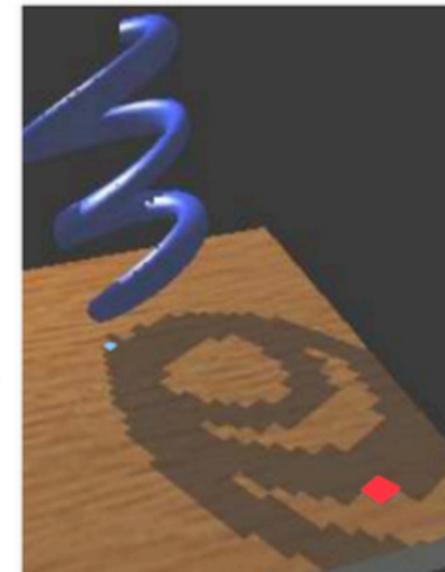
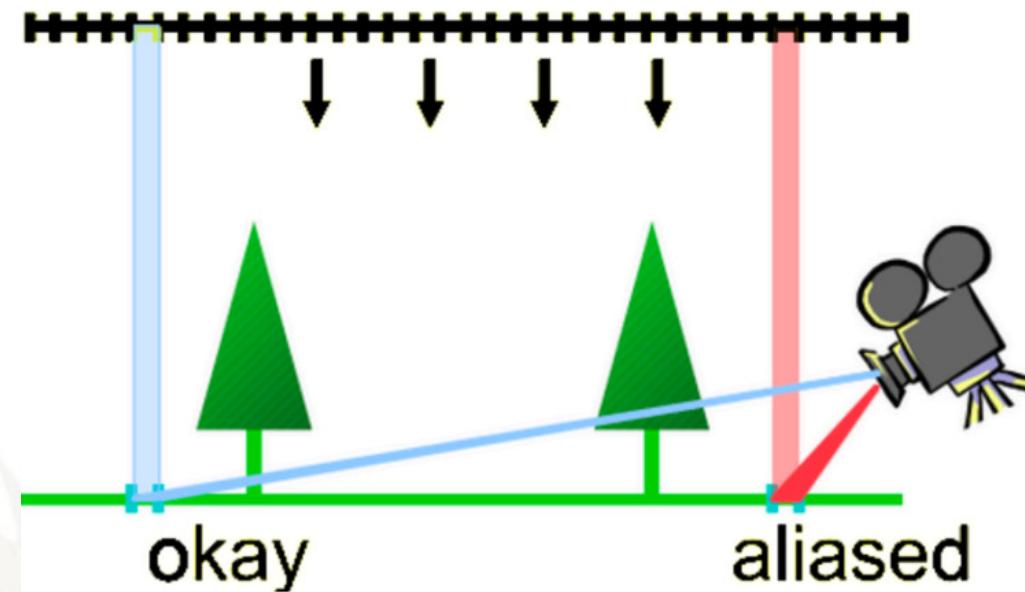
Credit

Shadows

Shadow Mapping (Limitations)

Perspective Aliasing

- View space resolution Vs. shadow map resolution



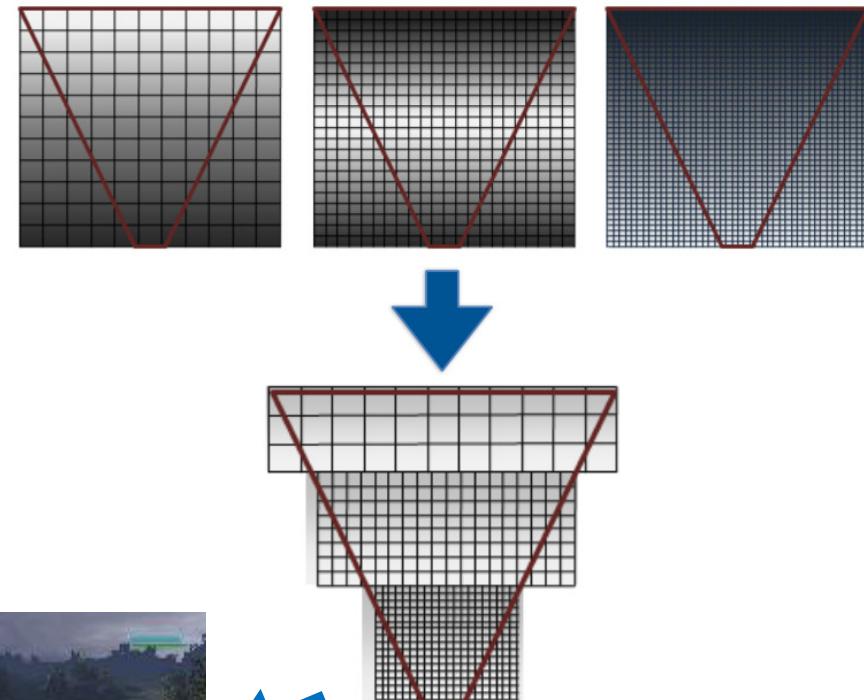
Credit

Shadows

Shadow Mapping (Limitations)

Cascade Shadow Maps

- Providing higher resolution of the depth texture near the viewer and lower resolution far away
- Splitting the camera view frustum and creating a separate depth-map for each partition

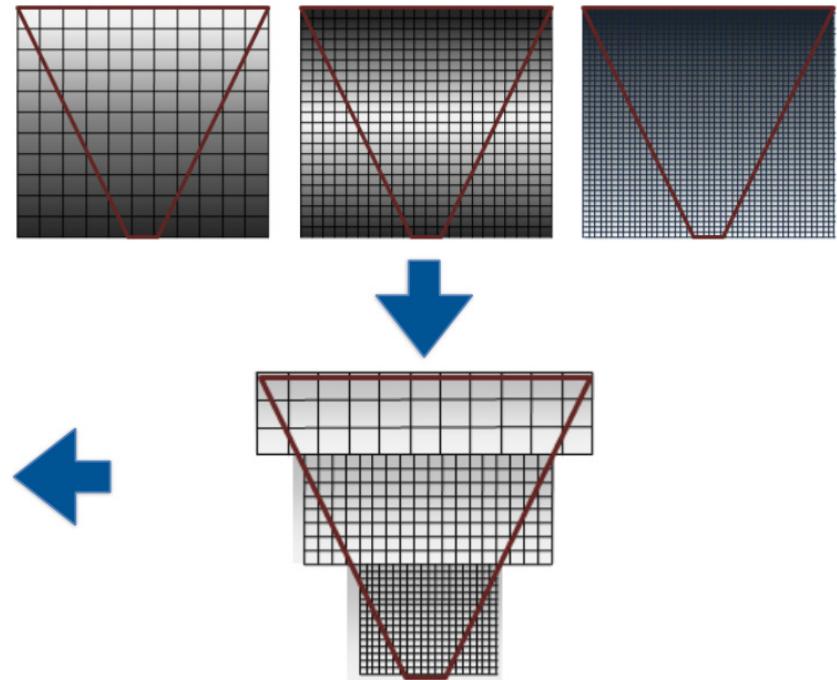
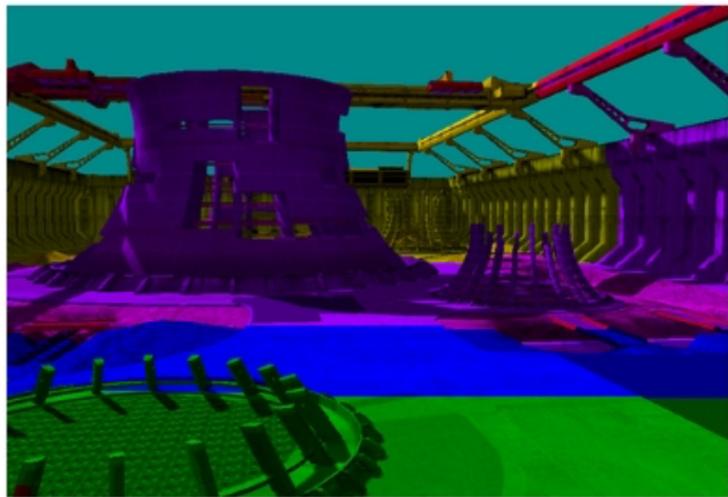


Credit

Shadows

Shadow Mapping (Limitations)

Cascade Shadow Maps

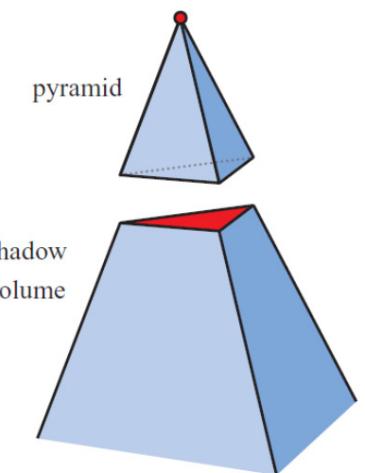
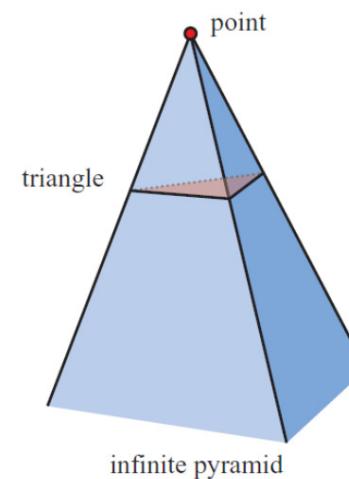
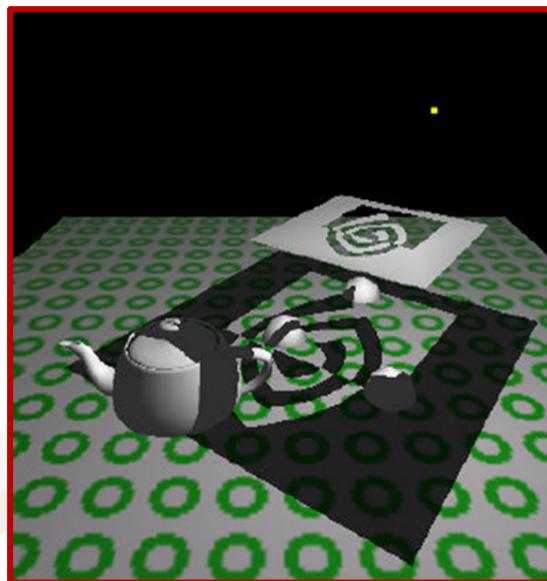


Credit

Shadows

Shadow Volumes

- In the real world, the shadow cast by an object blocking a light is a volume, not merely some two-dimensional portion of a plane
- An algorithm that models shadow regions as volumes



Credit

Shadows

Shadow Volumes

Two stages

- Compute the shadow volume formed by a light source and a set of shadowing objects
- Check if the point is inside/outside the shadow volume
 - Inside → shadowed
 - Outside → illuminated by light source

Shadows

Shadow Volumes

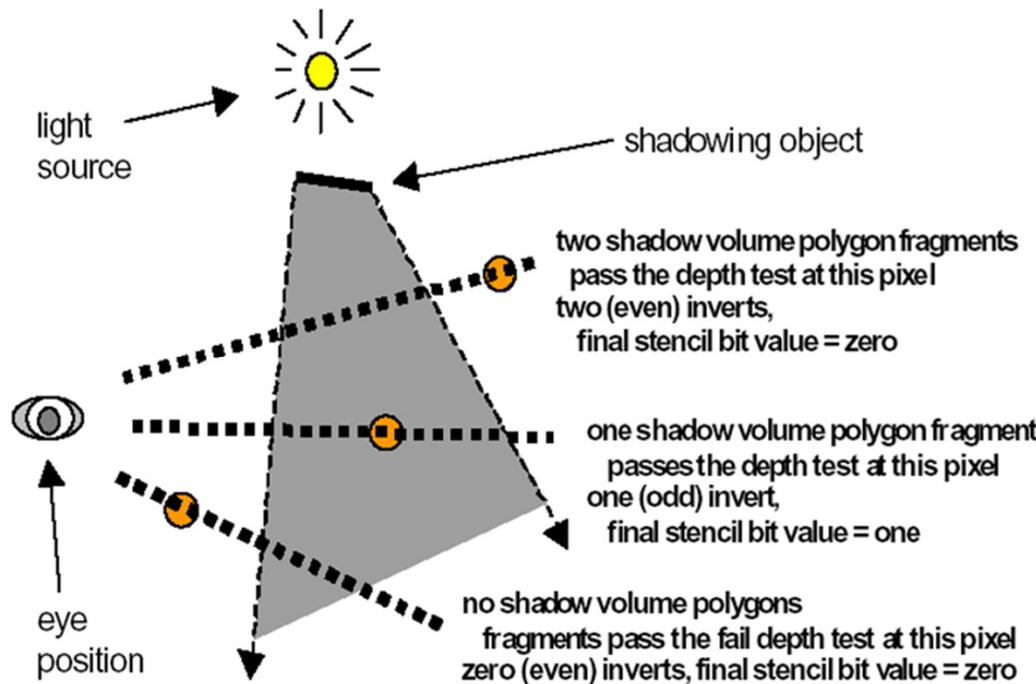
Decision of Point Inside/Outside Shadow

- Get the polygonal boundary representation for the shadow volume
- Render the scene with the lights enabled
- Clear the stencil buffer, and render the shadow volume
 - Whenever a rendered fragment of the shadow volume is closer than the depth of the other objects, invert a bit in the stencil value for that pixel
- After rendering, if the bit is on, then it means the fragment is inside the shadow volume, so must be shadowed
- Otherwise outside the shadow

Shadows

Shadow Volumes

Decision of Point Inside/Outside Shadow



Shadows

Shadow Volumes

Advantages

- It can be used on general-purpose graphics hardware
 - Only using the stencil buffer
- Finer resolution of shadow edges (no box artifacts)

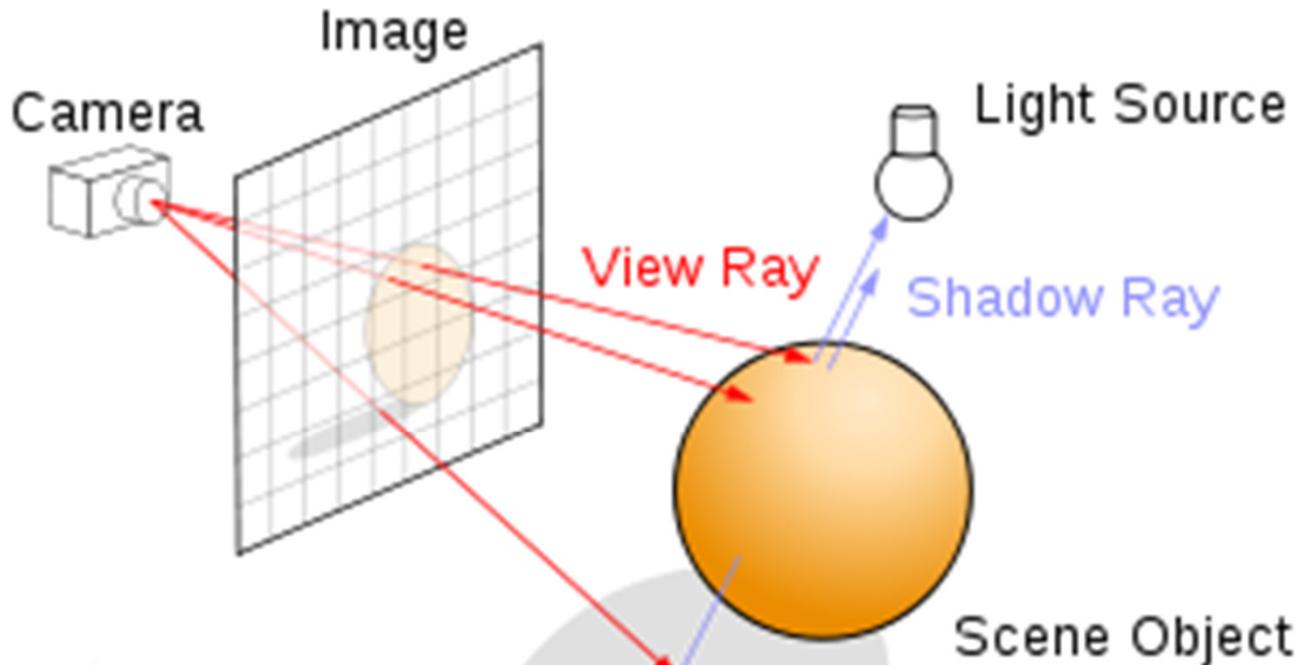


Disadvantages

- Bottle neck at the rasterizer
- Many shadow volumes covering many pixels

Shadows

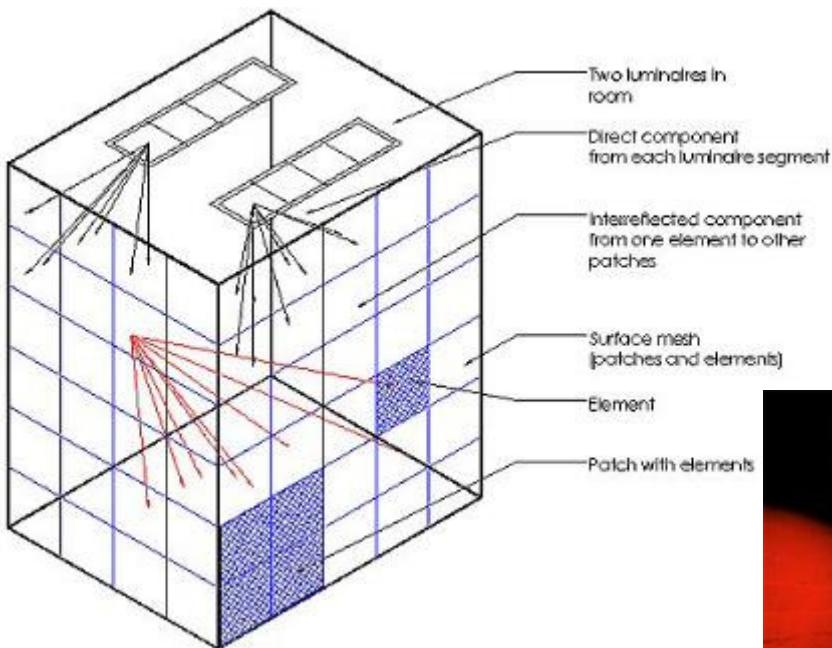
Ray Tracing



- ❑ For each visible face, trace ray to each light source
- ❑ If intersects another object before light, face is in shadow

Shadows

Radiosity



Review

- ❑ **Shadow algorithms**
 - ❑ shadows with projections
 - ❑ shadow maps
 - ❑ mitigating aliasing problems
- ❑ **shadow volumes**
- ❑ **Brief overview of**
 - ❑ Ray tracing
 - ❑ Radiosity

Next Lecture

Hierarchical Modeling



COMP 371

Computer Graphics

Session 8

HIERARCHICAL MODELING



Lecture Overview

□ Hierarchical Modeling

Hierarchical Modeling

Hierarchical Models

- Many graphical objects are structured
- Structure often is naturally hierarchical
- Exploit structure for
 - Efficient rendering
 - Example: tree leaves
 - Concise specification of model parameters
 - Example: joint angles
 - Physical realism



Image credit

Hierarchical Modeling

Instance Transformation

- Often we need several instances of an object
 - Wheels of a car
 - Arms or legs of a character
 - Chess pieces
 - Chairs and desks in a lecture hall



Hierarchical Modeling

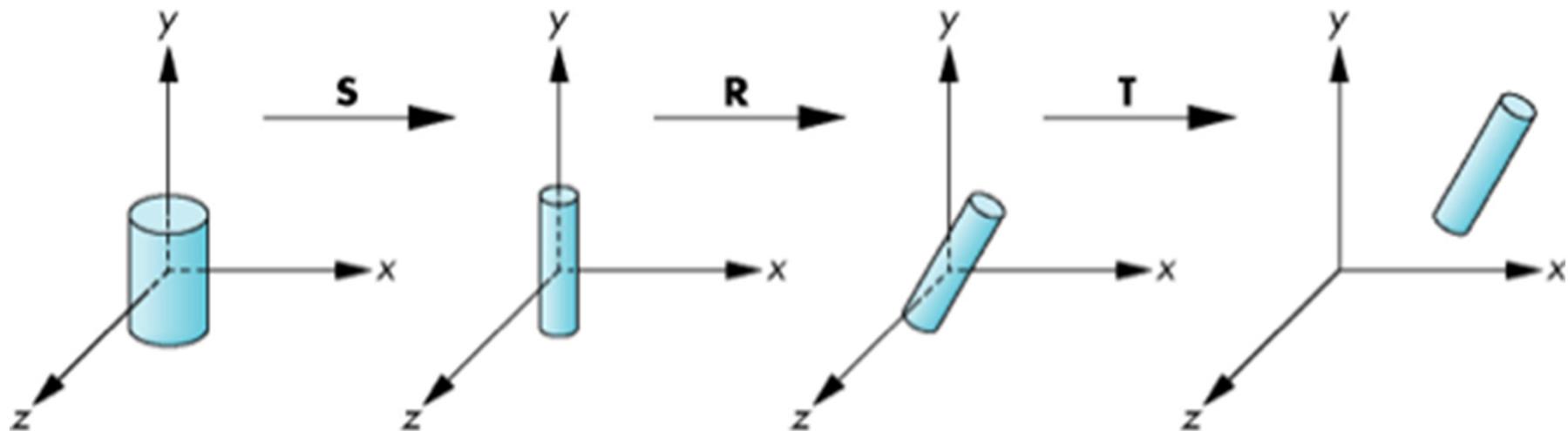
Instance Transformation

- Instances can be shared across space or time
- Write a function that renders the object in “standard” configuration (object space)
- Apply transformations to different instances (world space)
- Typical order: scaling, rotation, translation



Hierarchical Modeling

Sample Instance Transformation



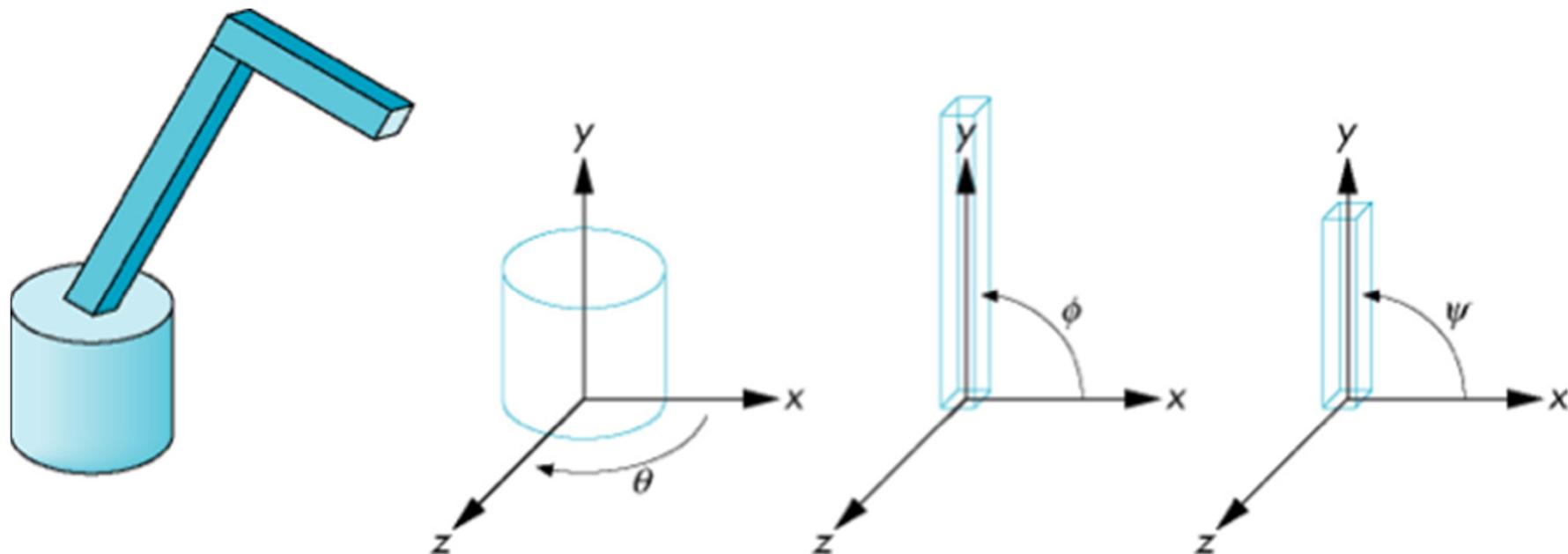
Hierarchical Modeling

Example



Hierarchical Modeling

Robot Arm : Drawing a Compound Object



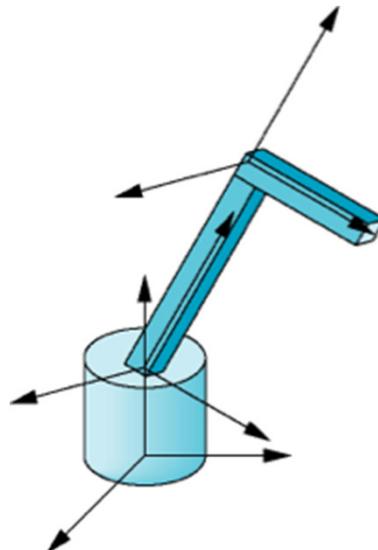
robot arm

parts in their own
coordinate systems

Hierarchical Modeling

Articulated Models

- Robot arm is an example of an articulated model
 - Parts connected at joints
 - Can specify state of model by giving all joint angles



Hierarchical Modeling

Relationships in Robot Arm

- **Base rotates independently**
 - Single angle determines position
- **Lower arm attached to base**
 - Its position depends on rotation of base
 - Must also translate relative to base and rotate about connecting joint
- **Upper arm attached to lower arm**
 - Its position depends on both base and lower arm
 - Must translate relative to lower arm and rotate about joint connecting to lower arm

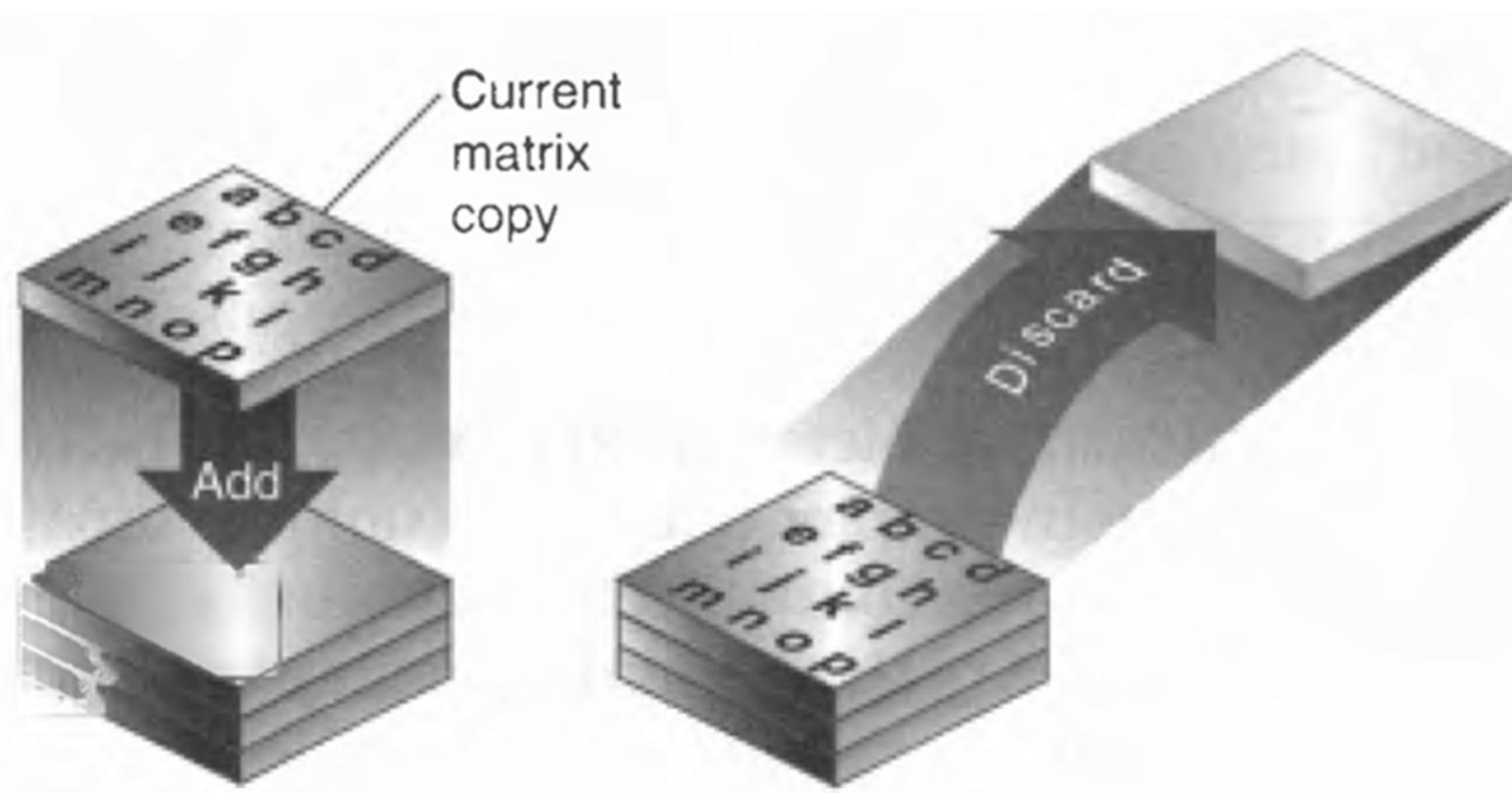
Hierarchical Modeling

Relationships in Robot Arm

- **Rotation of base:** R_b
 - Apply $M = R_b$ to base
- **Translate lower arm relative to base:** T_{lb}
- **Rotate lower arm around joint:** R_{lb}
 - Apply $M = R_b T_{lb} R_{lb}$ to lower arm
- **Translate upper arm relative to lower arm:** T_{ul}
- **Rotate upper arm around joint:** R_{ul}
 - Apply $M = R_b T_{lb} R_{lb} T_{ul} R_{ul}$ to upper arm

Hierarchical Modeling

Matrix Stack



Review

□ Hierarchical Modeling

Next Lecture

Splines



COMP 371

Computer Graphics

Session 9
SPLINES



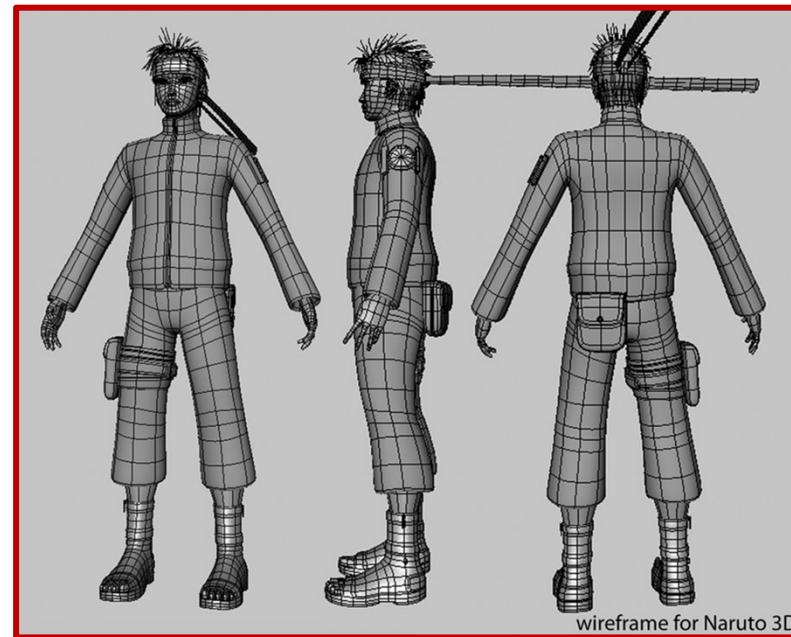
Lecture Overview

- **Review of last class**
 - Hierarchical Modeling
- **Splines**
 - Hermite Splines
 - Bezier Splines
 - Catmull-Rom Splines
 - Other Cubic Splines

Hierarchical Modeling

Modeling Complex Shapes

- We want to build models of very complicated objects
- Complexity is achieved using simple pieces
 - polygons,
 - parametric curves and surfaces
 - implicit curves and surfaces
- In this lecture we will focus on **parametric curves**



[Image credit](#)

Parametric Curves

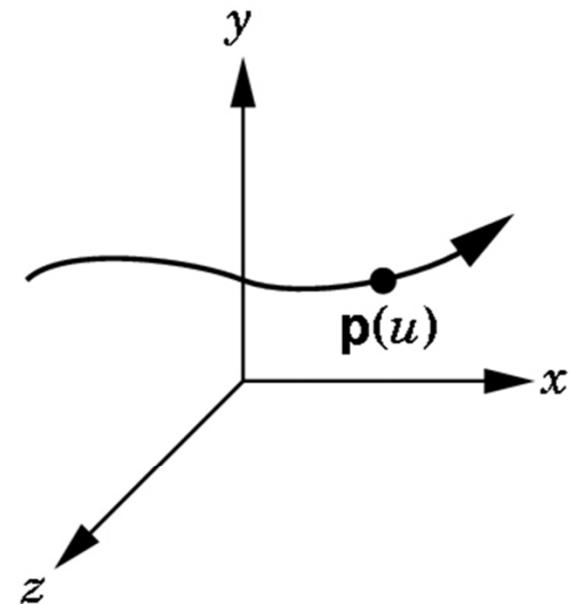
Curve Representation Requirements

- What do we need from curves in Computer Graphics?
 - Local control of shape (so that it's easy to build and modify)
 - Stability
 - Smoothness and continuity
 - Ability to evaluate derivatives
 - Ease of rendering

Parametric Curves

Curve Representation

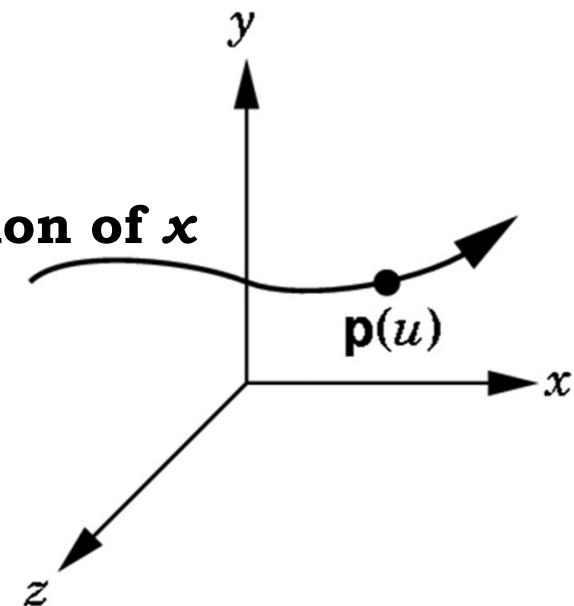
- **Explicit:** $y = f(x)$
 - Must be a function (single valued)
 - **Limitation:** vertical lines, circle?



Parametric Curves

Curve Representation

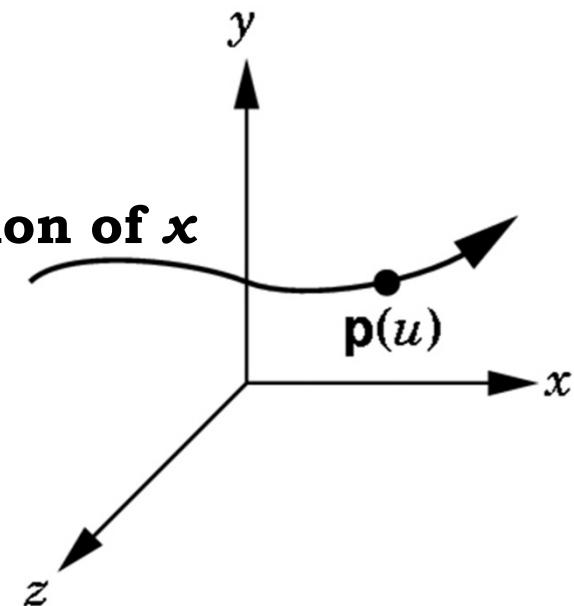
- **Explicit:** $y = f(x)$
 - Must be a function (single valued)
 - **Limitation:** vertical lines, circle?
- **Implicit:** $f(x, y) = 0$
 - **Pros:** y can be a multiple valued function of x
 - **Cons:** Hard to specify, modify, control



Parametric Curves

Curve Representation

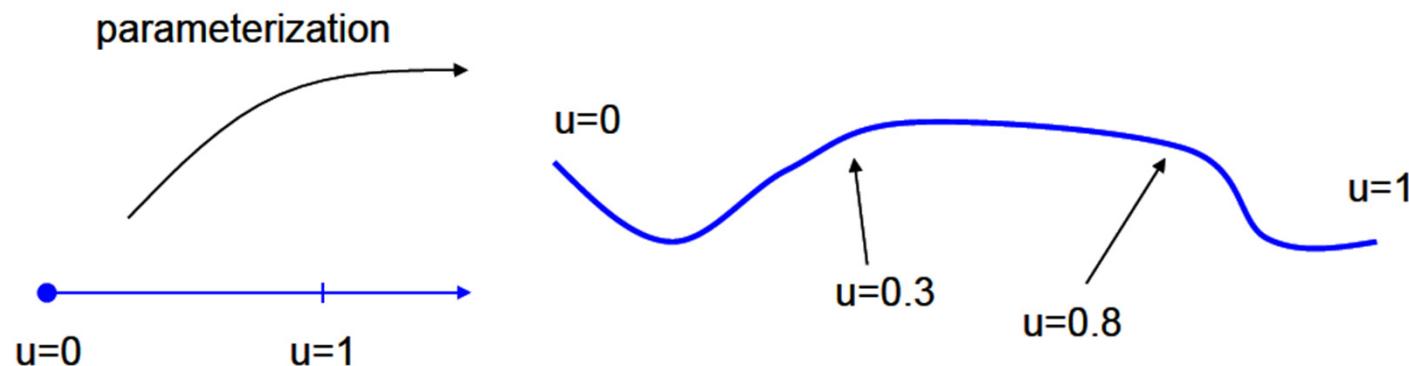
- **Explicit:** $y = f(x)$
 - Must be a function (single valued)
 - **Limitation:** vertical lines, circle?
- **Implicit:** $f(x, y) = 0$
 - **Pros:** y can be a multiple valued function of x
 - **Cons:** Hard to specify, modify, control
- **Parametric:** $(x, y) = (f(u), g(u))$
 - **Pros:** Easy to specify, modify, control
 - **Cons:** Extra “hidden” variable u , i.e. the parameter



Parametric Curves

Parameterization of a Curve

- **Parameterization of a curve:** how a change in u moves you along a given curve in xyz space
- **Parameterization is not unique.** It can be slow, fast, with continuous/discontinuous speed, clockwise (CW) or CCW, etc.

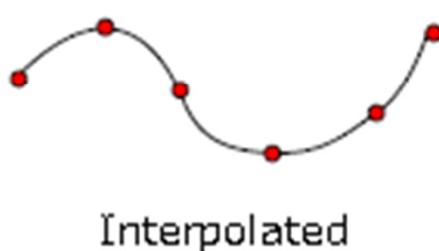


<https://colalg.math.csusb.edu/~devel/precalcdemo/param/src/param.html#:~:text=A%20parametric%20representation%20of%20a,different%20pairs%20of%20parametric%20equations.>

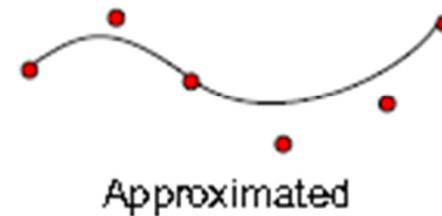
Parametric Curves

Interpolation Vs Approximation

- There are two ways to “connect the dots”:
- **Interpolation** → polynomial sections are fitted so that all the control points are connected
- **Approximation** → polynomial curve is plotted so that some, or all, of the control points are not on the curve path



Interpolated

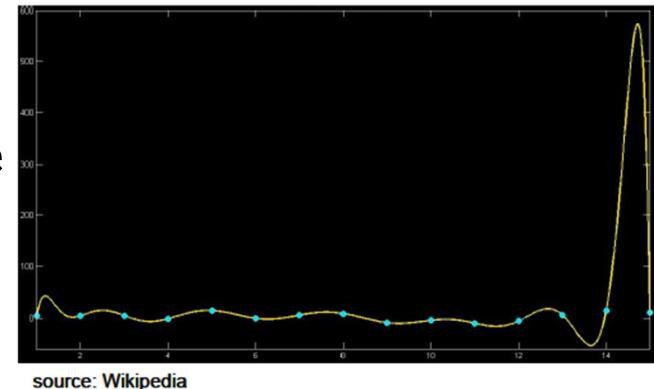


Approximated

Parametric Curves

Polynomial Interpolation

- An n^{th} degree polynomial fits a curve to $n+1$ points
 - called Lagrange Interpolation
 - result is a curve that is too wiggly, change to any control point affects the entire curve i.e. non-local
 - this method is poor
- We usually want the curve to be as smooth as possible
 - minimize wiggles
 - high-degree polynomials are bad

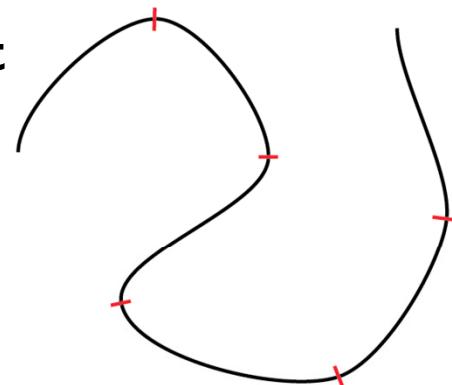


Lagrange interpolation,
degree=15

Parametric Curves

Splines: Piecewise Polynomials

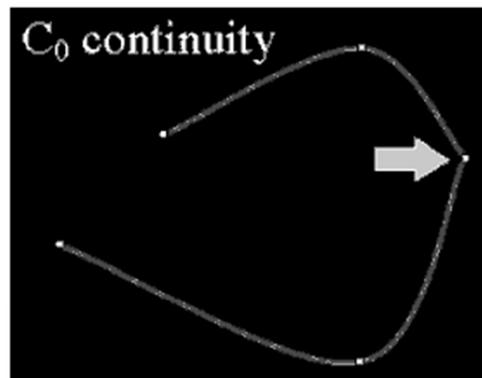
- A **spline** is a piecewise polynomial
 - Curve is broken into consecutive segments
 - Each segment is a low-degree polynomial interpolating i.e. passing through the control points
- **Cubic** piecewise polynomials are the most common
 - They are the lowest order polynomials that
 - interpolate two points and
 - allow the gradient at each point to be defined (C^1 continuity possible)
 - Piecewise definition gives local control
 - Higher or lower degrees are possible, of course



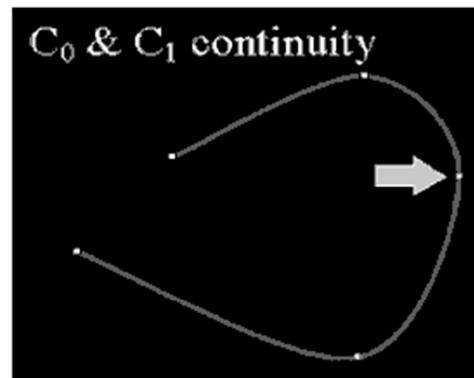
Parametric Curves

Splines: Piecewise Polynomials

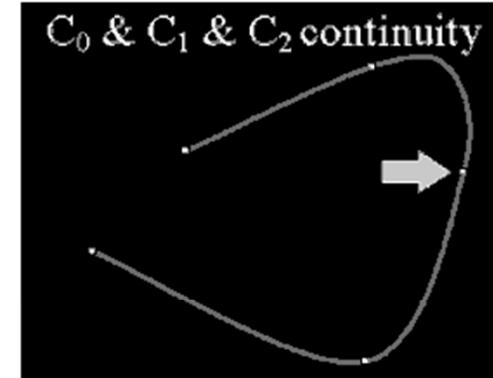
- **Spline:** many polynomials pieced together
- Want to make sure they fit together nicely



C_0 continuity
Continuous in position



C_0 & C_1 continuity
Continuous in position and tangent vector



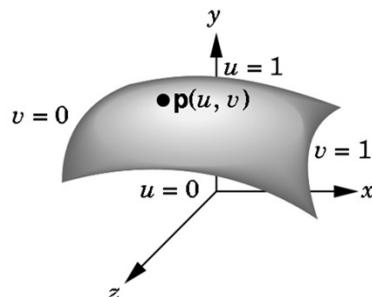
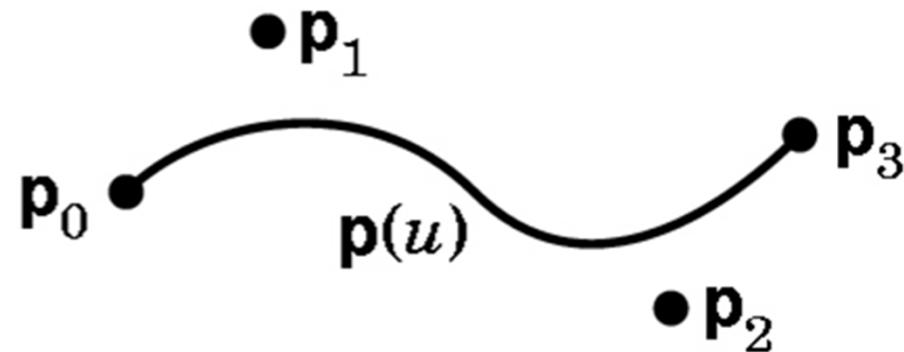
C_0 & C_1 & C_2 continuity
Continuous in position, tangent, and curvature

curvature: a measure of how rapidly a curve is bending at a point

Parametric Curves

Splines

- **Types of splines:**
 - Hermite Splines
 - Bezier Splines
 - Catmull-Rom Splines
 - Natural Cubic Splines
 - B-Splines
 - NURBS
- **Splines can be used to model both curves & surfaces**



Parametric Curves

Cubic Curves in 3D

- **Cubic polynomial:**
 - $p(u) = au^3 + bu^2 + cu + d = [u^3 \ u^2 \ u \ 1] [a \ b \ c \ d]^T$
 - a, b, c, d are 3D-vectors, u is a scalar
- **Three cubic polynomials, one for each coordinate:**
 - $x(u) = a_x u^3 + b_x u^2 + c_x u + d_x$
 - $y(u) = a_y u^3 + b_y u^2 + c_y u + d_y$
 - $z(u) = a_z u^3 + b_z u^2 + c_z u + d_z$
- **In matrix notation:** $[x(u) \ y(u) \ z(u)] = [u^3 \ u^2 \ u \ 1] \begin{bmatrix} a_x & a_y & a_z \\ b_x & b_y & b_z \\ c_x & c_y & c_z \\ d_x & d_y & d_z \end{bmatrix}$
- **Or simply:** $p = [u^3 \ u^2 \ u \ 1] A$

Parametric Curves

Cubic Hermite Splines

- Interpolating piecewise cubic polynomial
- We want a way to specify the end points and the tangent at the end points!



Hermite Specification

Parametric Curves

Deriving Hermite Splines

- **Four constraints:** value and slope (in 3-D, position and tangent vector) at beginning and end of interval $[0,1]$:

$$\left. \begin{array}{l} p(0) = p_1 = (x_1, y_1, z_1) \\ p(1) = p_2 = (x_2, y_2, z_2) \\ p'(0) = p'_1 = (x'_1, y'_1, z'_1) \\ p'(1) = p'_2 = (x'_2, y'_2, z'_2) \end{array} \right\} \text{user constraints}$$

- Assume cubic form: $p(u) = au^3 + bu^2 + cu + d$
- Four unknowns: a, b, c, d

Parametric Curves

Deriving Hermite Splines

- Assume cubic form:

$$p(u) = au^3 + bu^2 + cu + d \rightarrow P'(u) = 3au^2 + 2bu + c$$

$$p_1 = p(0) = d \quad \text{and} \quad p'_1 = p'(0) = c$$

$$p_2 = p(1) = a + b + c + d \quad \text{and} \quad p'_2 = p'(1) = 3a + 2b + c$$

- Linear system: 12 equations for 12 unknowns (however, can be simplified to 4 equations for 4 unknowns)
- Unknowns: a, b, c, d (each of a, b, c, d is a 3-vector)

Parametric Curves

Deriving Hermite Splines

$$\begin{aligned}d &= p_1 \\a + b + c + d &= p_2 \\c &= \bar{p}_1 \\3a + 2b + c &= \bar{p}_2\end{aligned}$$



Rewrite this 12x12 system
as a 4x4 system:

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{bmatrix} \begin{bmatrix} a_x & a_y & a_z \\ b_x & b_y & b_z \\ c_x & c_y & c_z \\ d_x & d_y & d_z \end{bmatrix} = \begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ \bar{x}_1 & \bar{y}_1 & \bar{z}_1 \\ \bar{x}_2 & \bar{y}_2 & \bar{z}_2 \end{bmatrix}$$

Parametric Curves

The Cubic Hermite Spline Equation

- After inverting the 4x4 matrix, we obtain:

$$[x \ y \ z] = [u^3 \ u^2 \ u \ 1] \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ \bar{x}_1 & \bar{y}_1 & \bar{z}_1 \\ \bar{x}_2 & \bar{y}_2 & \bar{z}_2 \end{bmatrix}$$

↑ ↑ basis control matrix
point on parameter (what the user gets to pick)
the spline vector

- This form is typical for splines
 - basis matrix and meaning of control matrix change with the spline type

Parametric Curves

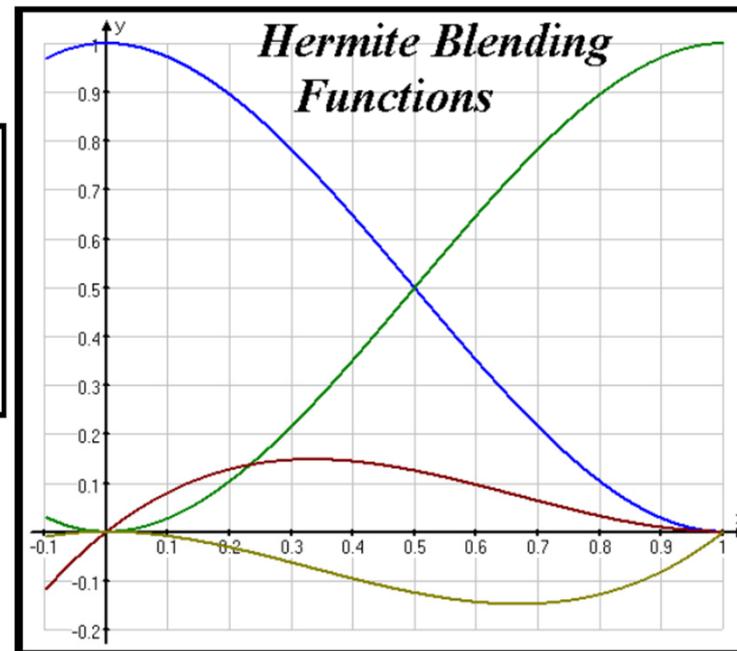
Four Basis Functions for Hermite Splines

- Every cubic Hermite spline is a linear combination (blend) of these 4 functions

transpose

$$p(u) = \begin{bmatrix} 2u^3 - 3u^2 + 1 \\ -2u^3 + 3u^2 \\ u^3 - 2u^2 + u \\ u^3 - u^2 \end{bmatrix}^T \begin{bmatrix} p_1 \\ p_2 \\ \bar{p}_1 \\ \bar{p}_2 \end{bmatrix}$$

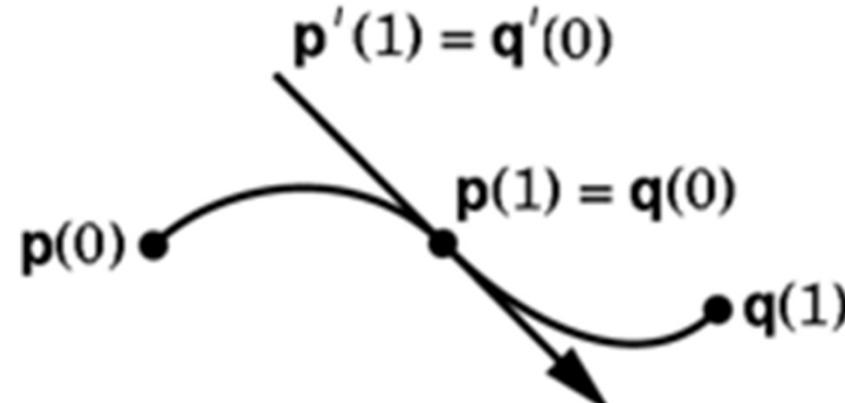
4 Basis Functions



Parametric Curves

Piecing Together Hermite Splines

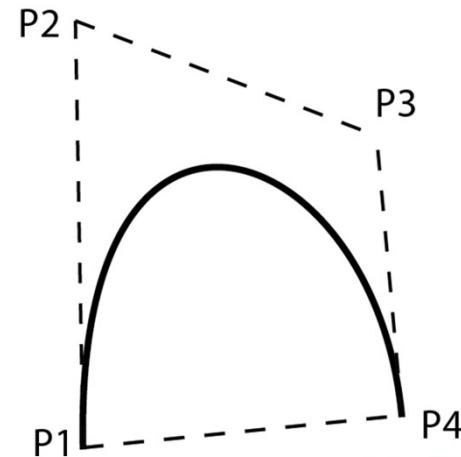
- Easy to make a multi-segment Hermite Spline:
 - each segment is specified by a *cubic Hermite curve*
 - just specify the position and tangent at each “joint” (called **knot**)
 - the pieces fit together with matched positions and first derivatives
 - gives **C_1 continuity**
 - *local control*



Parametric Curves

Bezier Splines

- Variant of the Hermite spline
 - Interpolates endpoints
 - Approximation of other two control points
- Instead of endpoints and tangents, four control points
 - points P_1 , P_4 are on the curve (end points) $\rightarrow p(0) = P_1$ and $p(1) = P_4$
 - points P_2 , P_3 are off the curve
 - $p'(0) = 3(P_2-P_1)$
 - $p'(1) = 3(P_4-P_3)$
- Basis matrix is derived from the Hermite basis (or from scratch)
- **Convex Hull property:** curve contained within the convex hull of control points
- Scale factor “3” is chosen to make “velocity” approximately constant



Parametric Curves

The Bezier Spline Matrix

$$[x \ y \ z] = [u^3 \ u^2 \ u \ 1] \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ -3 & 3 & 0 & 0 \\ 0 & 0 & -3 & 3 \end{bmatrix} \begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \\ x_4 & y_4 & z_4 \end{bmatrix}$$

Hermite basis

Bezier to Hermite

Bezier
control matrix

$$= [u^3 \ u^2 \ u \ 1] \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \\ x_4 & y_4 & z_4 \end{bmatrix}$$

Bezier basis

Bezier
control matrix

Parametric Curves

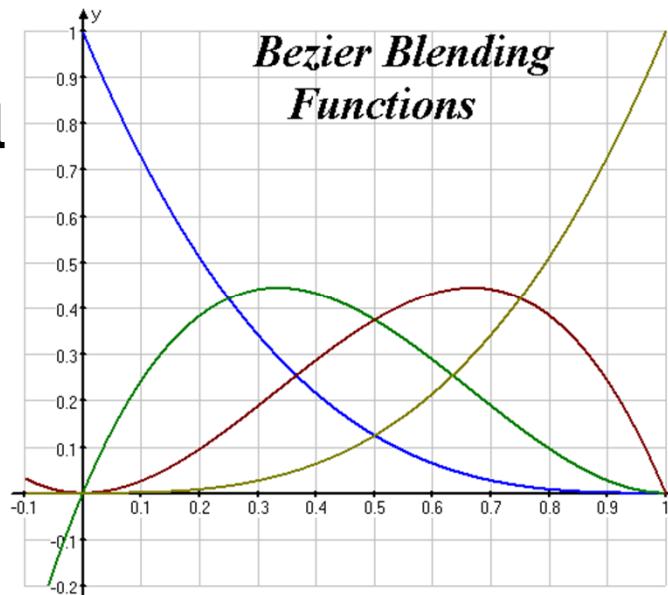
Bezier Blending Functions

- Also known as the order 4, degree 3 Bernstein polynomials
- Nonnegative, sum to 1; $0 \leq u \leq 1$
- The entire curve lies inside the polyhedron bounded by the control points

$$b_{\nu,n}(x) = \binom{n}{\nu} x^{\nu} (1-x)^{n-\nu}, \quad \nu = 0, \dots, n.$$

$$\binom{n}{k} = \frac{n!}{k! (n-k)!} \quad \text{for } 0 \leq k \leq n,$$

$$p(t) = \begin{bmatrix} (1-t)^3 \\ 3t(1-t)^2 \\ 3t^2(1-t) \\ t^3 \end{bmatrix}^T \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{bmatrix}$$



Parametric Curves

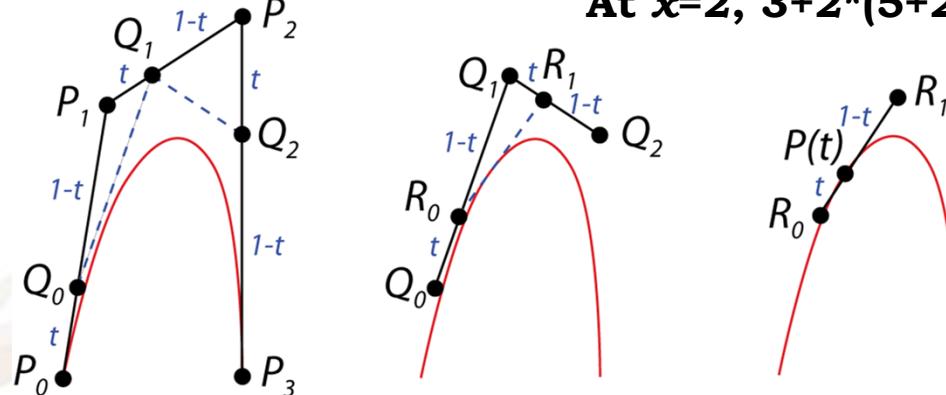
DeCasteljau Construction

- Efficient algorithm to evaluate Bezier splines
- Similar to Horner rule for polynomials

$$p(x) = a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + a_n x))).$$

$$p(x) = x^4 + 2x^3 + 4x^2 + 5x + 3 \rightarrow a_0=3, a_1=5, a_2=4, a_3=2, a_4=1$$

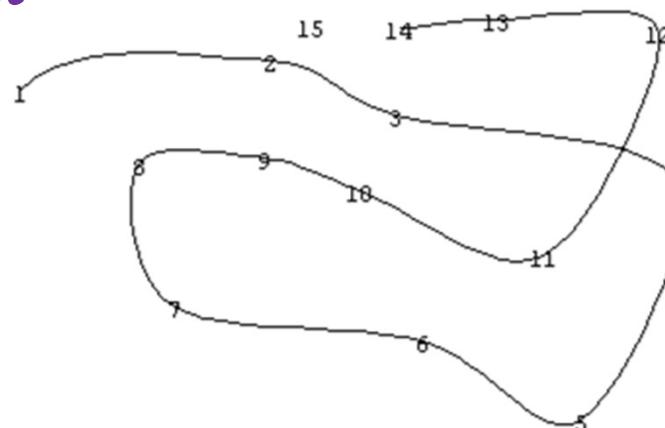
At $x=2, 3+2*(5+2*(4+2*(2+2*1))) = 61$



Parametric Curves

Catmull-Rom Splines

- With Hermite Splines, the designer must arrange for consecutive tangents to be co-linear, to get C_1 continuity. Similar for Bezier. This gets tedious.
- Catmull-Rom:** an interpolating cubic spline with built in C_1 continuity.
- Compared to Hermite/Bezier: *fewer control points required, but less freedom*



Parametric Curves

Constructing the Catmull-Rom Spline

- Suppose we are given n control points in 3-D: p_1, p_2, \dots, p_n .
- For a Catmull-Rom spline, we set the tangent at p_i to $s^*(p_{i+1} - p_{i-1})$ for $i=2, \dots, n-1$, for some s (often $s=0.5$)
- **s is tension parameter:** determines the magnitude (but not direction!) of the tangent vector at point p_i
- What about endpoint tangents? Use extra control points p_0, p_{n+1} .
- Now we have positions and tangents at each knot. This is a **Hermite specification** → use Hermite formulas to derive the spline
- Note: curve between p_i and p_{i+1} is completely determined by $p_{i-1}, p_i, p_{i+1}, p_{i+2}$.

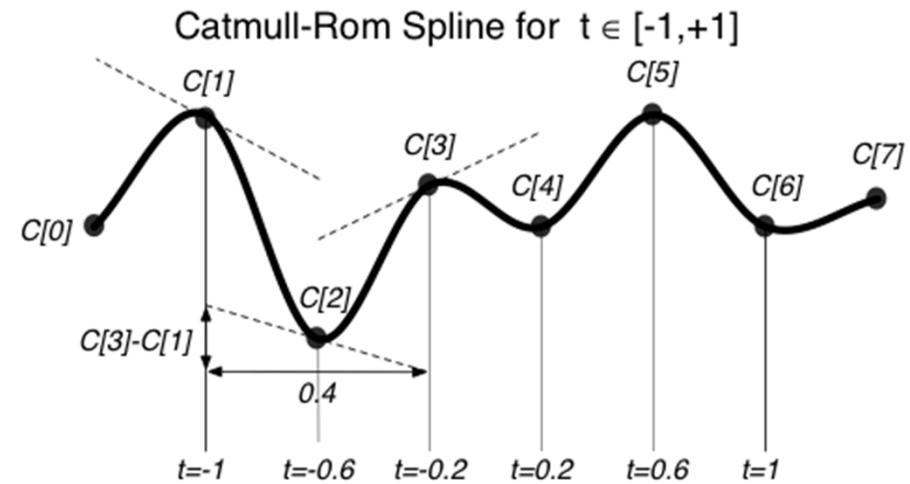
Parametric Curves

Catmull-Rom Spline Matrix

- Derived in a way similar to Hermite and Bezier
- Parameter s is typically set to $s = 0.5$

$$\begin{bmatrix} x & y & z \end{bmatrix} = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \begin{bmatrix} -s & 2-s & s-2 & s \\ 2s & s-3 & 3-2s & -s \\ -s & 0 & s & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \\ x_4 & y_4 & z_4 \end{bmatrix}$$

basis control matrix



Parametric Curves

Spline with More Continuity?

- So far, only C_1 continuity
- How could we get C_2 continuity at control points?
- Possible answers:
 - Use higher degree polynomials
 - degree 4 = quadtric, degree 5 = quintic, etc but these get computationally expensive, and sometimes wiggly
 - Give up local control → natural cubic splines
 - A change to any control point affects the entire curve
 - Give up interpolation → cubic b-splines
 - Curve goes near, but not through the control points

Parametric Curves

Comparison of Basic Cubic Splines

Type	Local Control	Continuity	Interpolation
Hermite	YES	C1	YES
Bezier	YES	C1	YES
Catmull-Rom	YES	C1	YES
Natural	NO	C2	YES
B-Splines	YES	C2	NO

Summary:

Cannot get C2, interpolation and local control with cubics

Parametric Curves

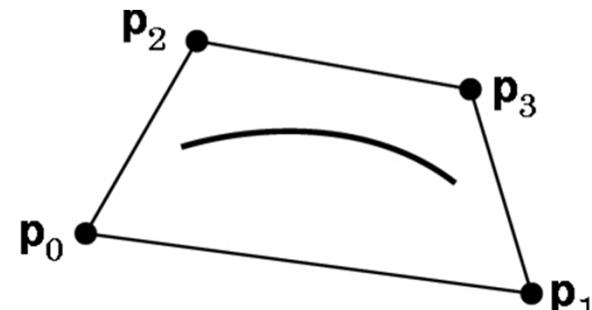
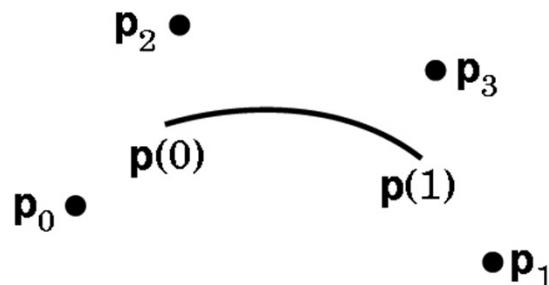
Natural Cubic Splines

- If you want 2nd derivatives at joints to match up, the resulting curves are called **natural cubic splines**
- It's a simple computation to solve for the cubics' coefficients
- See Numerical Recipes in C book for code
- Finding all the right weights is a global calculation
- solve tridiagonal linear system

Parametric Curves

B-Splines

- Give up interpolation
 - the curve passes near the control points
 - best generated with interactive placement (because it's hard to guess where the curve will go)
- Curve obeys the convex hull property
- C_2 continuity and local control are good compensation for loss of interpolation



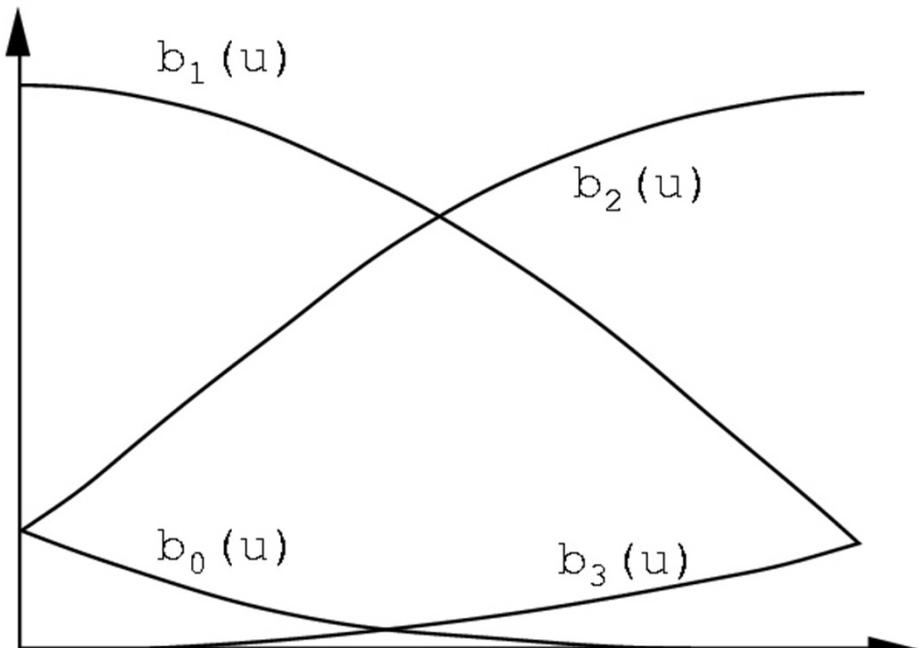
Parametric Curves

B-Spline Basis

- We always need 3 more control points than the number of spline segments

$$M_{Bs} = \frac{1}{6} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix}$$

$$G_{Bsi} = \begin{bmatrix} P_{i-3} \\ P_{i-2} \\ P_{i-1} \\ P_i \end{bmatrix}$$



Parametric Curves

Other Common Types of Splines

- **Non-uniform Splines**
- **Non-uniform Rational cubic curves (NURBS)**
 - NURBS are very popular and used in many commercial packages

Parametric Curves

How to Draw Spline Curves

- Basis matrix equation allows same code to draw any spline type
 - Method 1: brute force (incremental)
 - Calculate the coefficients
 - For each cubic segment, vary u from 0 to 1 (fixed step size e.g 0.1)
 - Plug in u value, matrix multiply to compute position on curve
 - Draw line segment from last position to current position
 - What's wrong with this approach?
 - Draws in even steps of u
 - Even steps of u does not mean even steps of x
 - Line length will vary over the curve
 - Want to bound line length
 - too long: curve looks jagged
 - too short: curve is slow to draw

Parametric Curves

How to Draw Spline Curves

- **Method 2:** recursive subdivision - vary step size to draw short lines

```
Subdivide( $u_0$ ,  $u_1$ , maxlinelength)
```

```
 $u_{\text{mid}} = (u_0 + u_1)/2$ 
```

```
 $x_0 = f(u_0)$ 
```

```
 $x_1 = f(u_1)$ 
```

```
if  $|x_1 - x_0| > \text{maxlinelength}$ 
```

```
    Subdivide( $u_0$ ,  $u_{\text{mid}}$ , maxlinelength)
```

```
    Subdivide( $u_{\text{mid}}$ ,  $u_1$ , maxlinelength)
```

```
else drawline( $x_0$ ,  $x_1$ )
```

- **Method 3:** Variant on Method 2 - vary step size to get smooth curvature

- replace condition in “if” statement with straightness criterion
- draws fewer lines in flatter regions of the curve

Review

- ❑ Piecewise cubic is generally sufficient
- ❑ Define the specification [conditions/properties] on the curves and their continuity
- ❑ Most important:
 - ❑ basic curve properties (what are the conditions, controls, and properties for each spline type)
 - ❑ generic matrix formula for uniform cubic splines
 - ❑ $p(u) = u \mathbf{x}$ basis \mathbf{x} control
 - ❑ given a definition, derive a basis matrix (do not memorize the matrices themselves)



COMP 371

Computer Graphics

Session 10
RASTERIZATION



Lecture Overview

- Review of last class**
- Line Scan conversion**
- Polygon Scan conversion**
- Antialiasing**

RECAP

What have we seen so far?

- We digitally model the *world of interest* using geometric primitives (edges, faces, surfaces,...)
- For rendering this world, we must compute part(s) of the world that are *visible within a pixel*. We then compute the pixel color such that it visually represents that part of the world as best as it can
- Edges (geometrically approximated by straight lines) dominate visual form perception by humans
- How do we render edges in our digitally created world?

RECAP

What have we seen so far?

- Point classification is a very fundamental operation in computer graphics. It is defined as follows:
 - Given a point P and a geometric entity E , classify the point as being **outside**, **on** or **inside** the entity
- Example:
 - $P = (x_0, y_0)$
 - Let E be a circle with equation: $x^2 + y^2 = r^2$
 - Let $E_{\text{centre}} = (x_c, y_c)$ and $d = \sqrt{(x_0 - xc)^2 + (y_0 - yc)^2}$
 - Then P is inside if $d^2 < r^2$, on if $d^2 = r^2$ and outside if $d^2 > r^2$

Rasterization

Rasterization

- The raster display is a matrix of picture elements also called **pixels**.
 - Each pixel has a color value assigned
- A **frame buffer stores the values for each pixel**
- The task of displaying a world modelled using primitives like lines, polygons, filled/patterned areas, etc. can be carried out in two steps
 - determine the pixels through which the primitive is visible, a process called **Rasterization or scan conversion**
 - determine the color value to be assigned to each such pixel

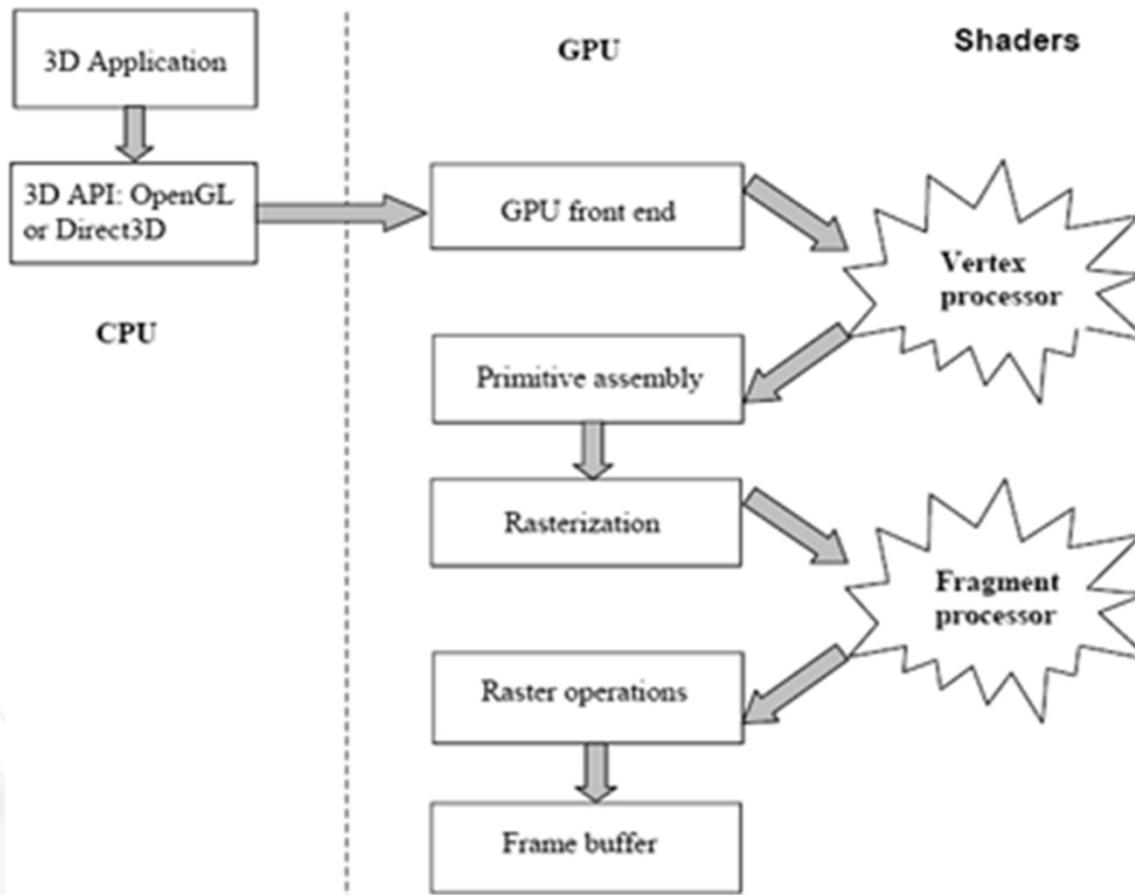
Rasterization

Rasterization (Scan Conversion)

- Final step in pipeline: Rasterization
- From screen coordinates (float) to pixels (int)
- Writing pixels into frame buffer
- Separate buffers:
 - depth (z-buffer)
 - display (frame buffer)
 - shadows (stencil buffer)
 - blending (accumulation buffer)

Rasterization

Programmable GPU Pipeline



- **GPUs have most of the Rasterization algorithms implemented in hardware!**

Rasterization

Scan Converting a Line Segment

- The line is a powerful element used since the days of Euclid *to model the edges in the world*



- Given a line segment defined by its endpoints, generate one fragment for each pixel that is intersected (or covered) by the line segment and assign a color

Rasterization

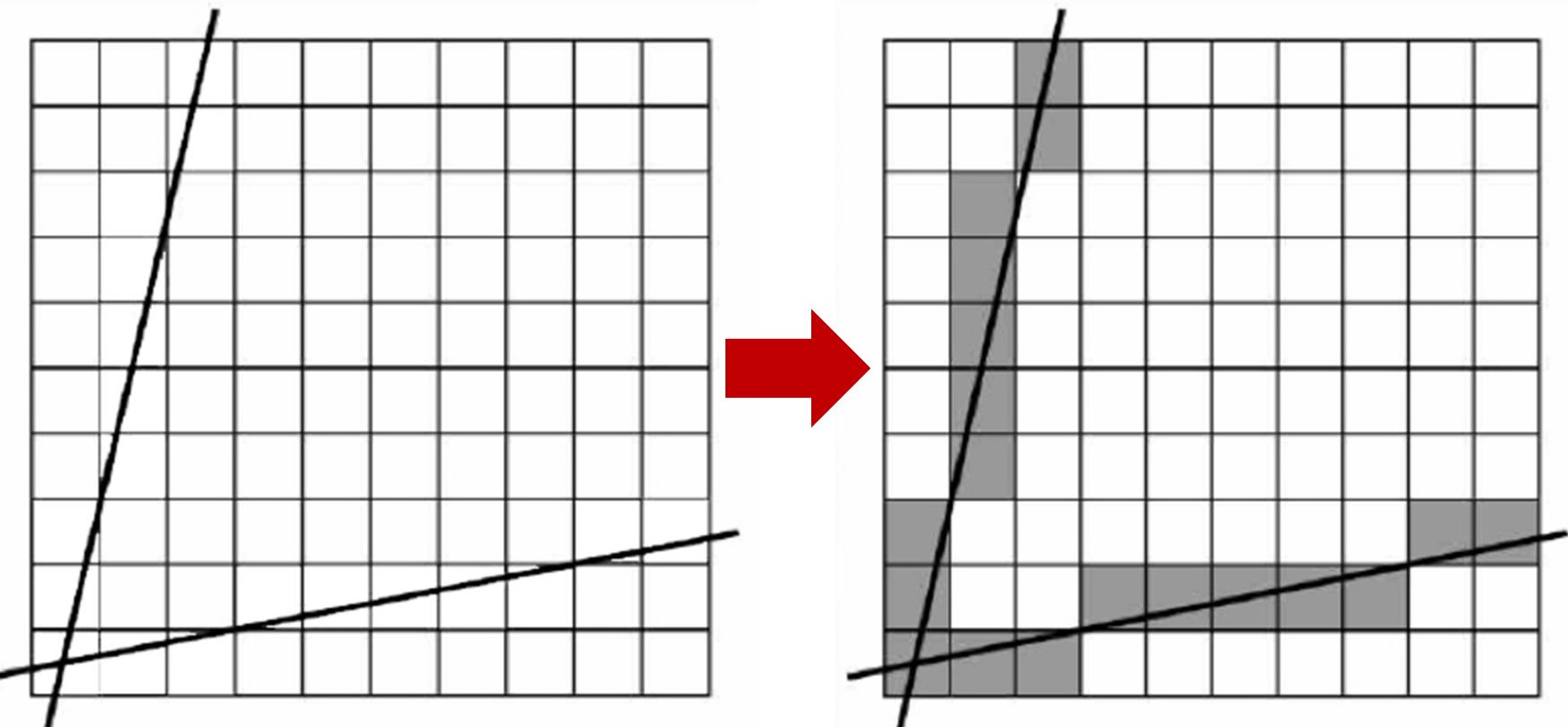
Scan converting lines

Requirements

- the chosen pixels should lie as *close to the ideal line* as possible
- the sequence of pixels should be as straight as possible. All lines should appear to be of constant brightness independent of their length and orientation
- should start and end accurately
- should be drawn as rapidly as possible
- should be possible to draw lines with different width and line styles

Rasterization

Rasterizing a line



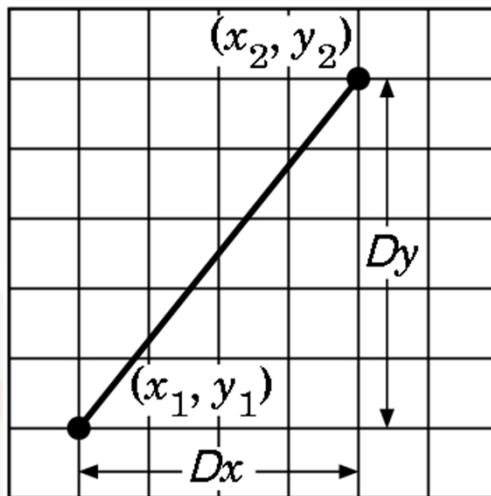
Rasterization

Digital Differential Analyzer (DDA)

- For a line segment joining points $P(x_1, y_1)$ and $P(x_2, y_2)$
- Represent line as

$$y = mx + h \quad \text{where} \quad m = \frac{y_2 - y_1}{x_2 - x_1} = \frac{\Delta y}{\Delta x}$$

- Then, if $\Delta x = 1$ pixel, we have $\Delta y = m\Delta x = m$



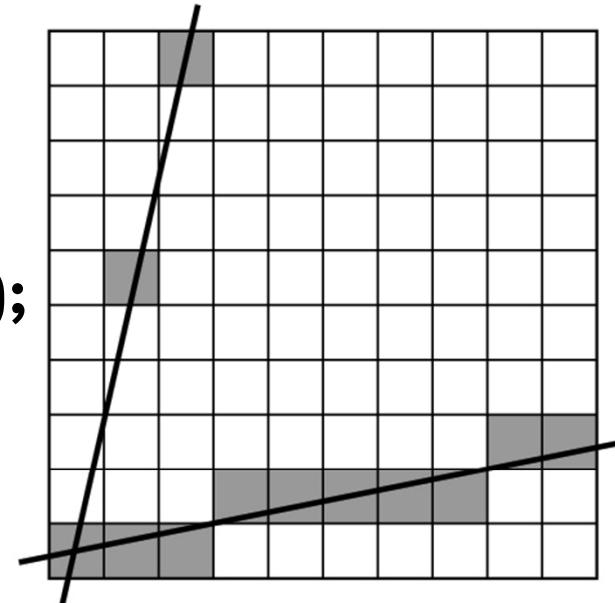
Rasterization

Digital Differential Analyzer (DDA)

- Assume the endpoint values of the line are x_1 and x_2 respectively then

```
write_pixel (int x, int y, int value)
    for (i = x1; i <= x2; x++) {
        y += m;
        write_pixel (i, round(y), color);
    }
```

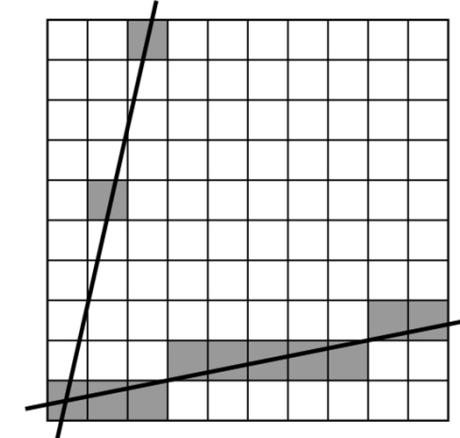
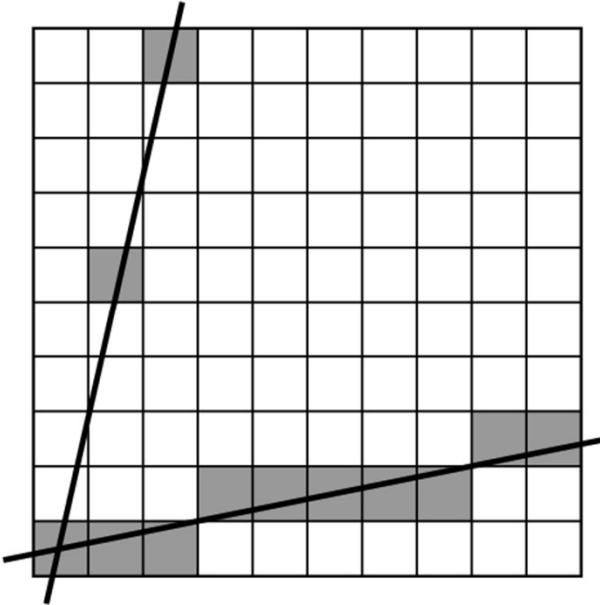
- Problems:**
 - Requires floating point addition
 - Missing pixels with steep slopes:
slope restriction needed



Rasterization

Digital Differential Analyzer (DDA)

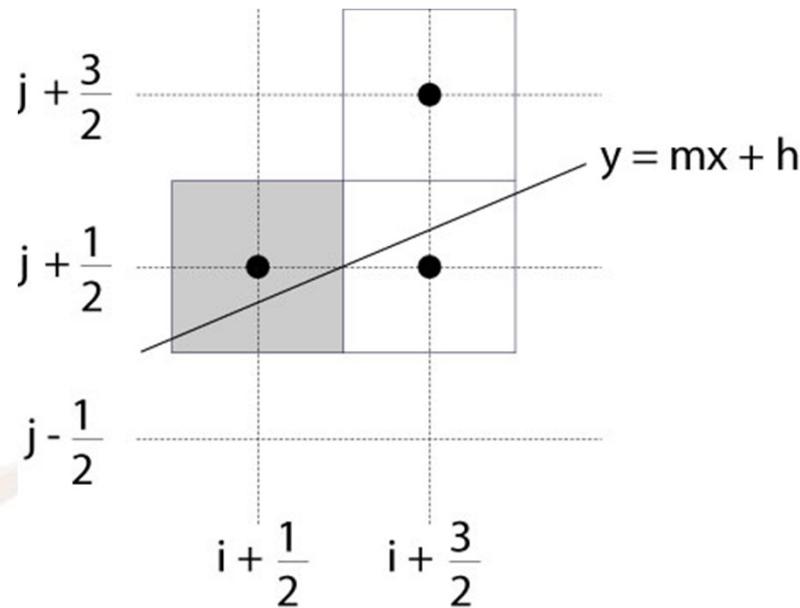
- Assume $0 \leq m \leq 1$
- Exploit symmetry
- Distinguish special cases
- But still requires floating point additions!



Rasterization

Bresenham's Algorithm I

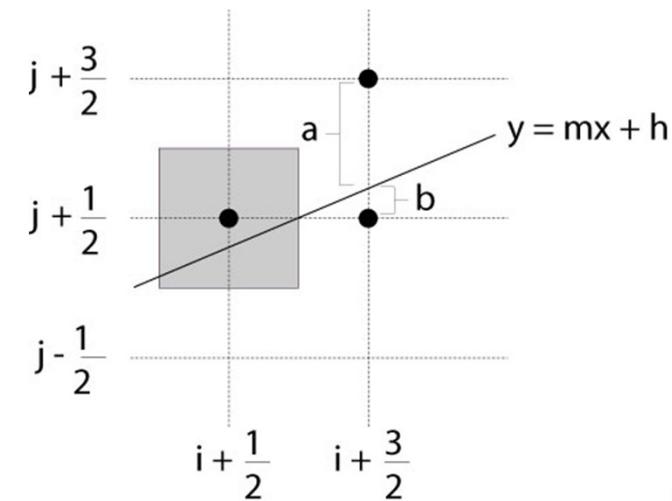
- Eliminate floating point addition from DDA
- Assume again $0 \leq m \leq 1$
- Assume pixel centers are halfway between integers



Rasterization

Bresenham's Algorithm II

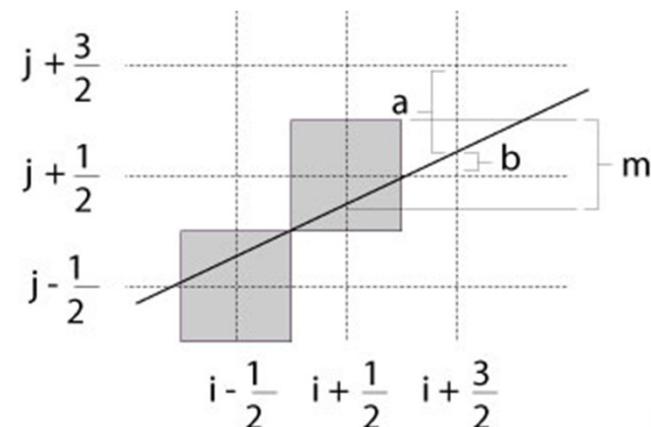
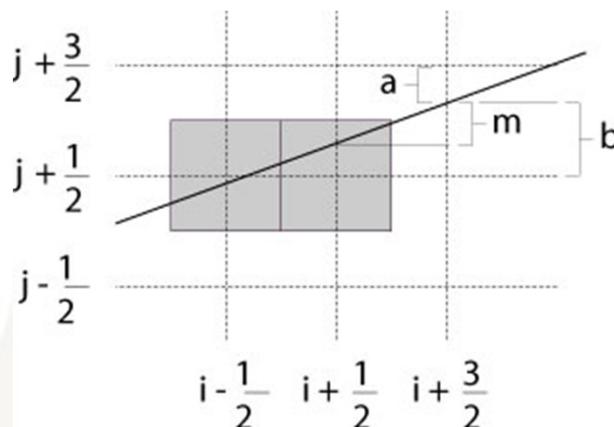
- **Decision variable $a - b$**
 - if $a - b \geq 0$ choose lower pixel
 - if $a - b < 0$ choose higher pixel
- **Goal: avoid explicit computation of $a - b$**
- **Step 1: re-scale**
$$d = (x_2 - x_1)(a - b) = \Delta x(a - b)$$
- **d is always integer**



Rasterization

Bresenham's Algorithm III

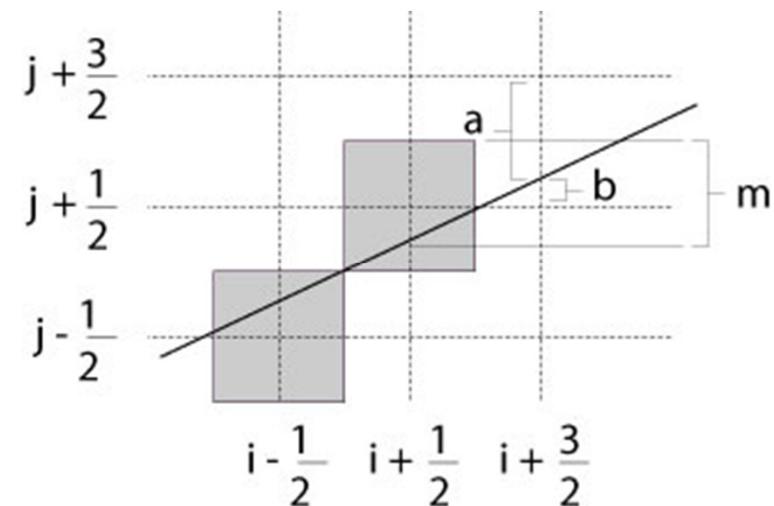
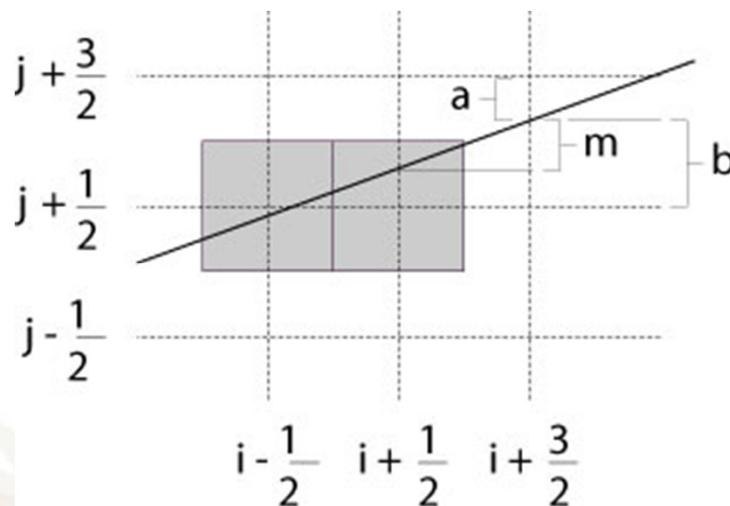
- Compute d at step $k + 1$ from d at step k !
- **Case:** j did not change ($d_k \geq 0$)
 - a decreases by m , b increases by m
 - $(a - b)$ decreases by $2m = 2(\Delta y / \Delta x)$
 - $\Delta x(a - b)$ decreases by $2\Delta y$



Rasterization

Bresenham's Algorithm IV

- Case: j did change ($d_k < 0$)
 - a decreases by $m-1$, b increases by $m-1$
 - $(a - b)$ decreases by $2m-2 = 2(\Delta y / \Delta x - 1)$
 - $\Delta x(a - b)$ decreases by $2(\Delta y - \Delta x)$



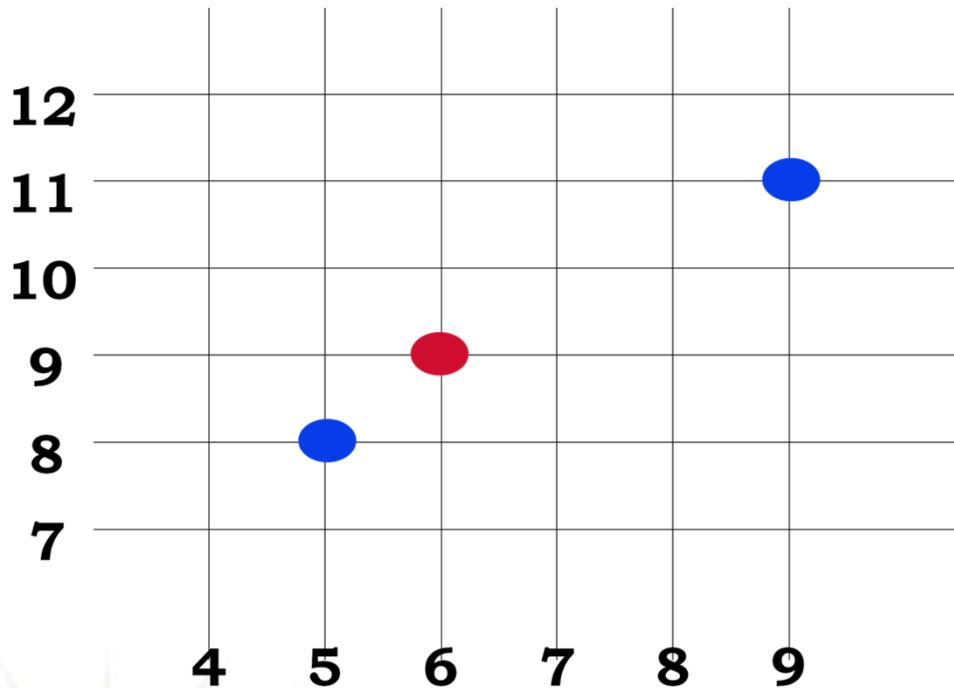
Rasterization

Bresenham's Algorithm V

- So $d_{k+1} = d_k - 2\Delta y$ if $d_k \geq 0$ (next pixel is E)
- And $d_{k+1} = d_k - 2(\Delta y - \Delta x)$ if $d_k < 0$ (next pixel is NE)
- Final (**efficient**) implementation:
- ```
void draw_line(int x1, int y1, int x2, int y2) {
 int x, y = y1;
 int dx = 2*(x2-x1), dy = 2*(y2-y1);
 int dydx = dy - dx, D = (dy-dx)/2;
 for (x = x1; x <= x2; x++) {
 write_pixel(x, y, color);
 if (D>0) D -= dy;
 else { y++; D -= dydx; }
 }
}
```

# Rasterization

## Bresenham's Algorithm: An example

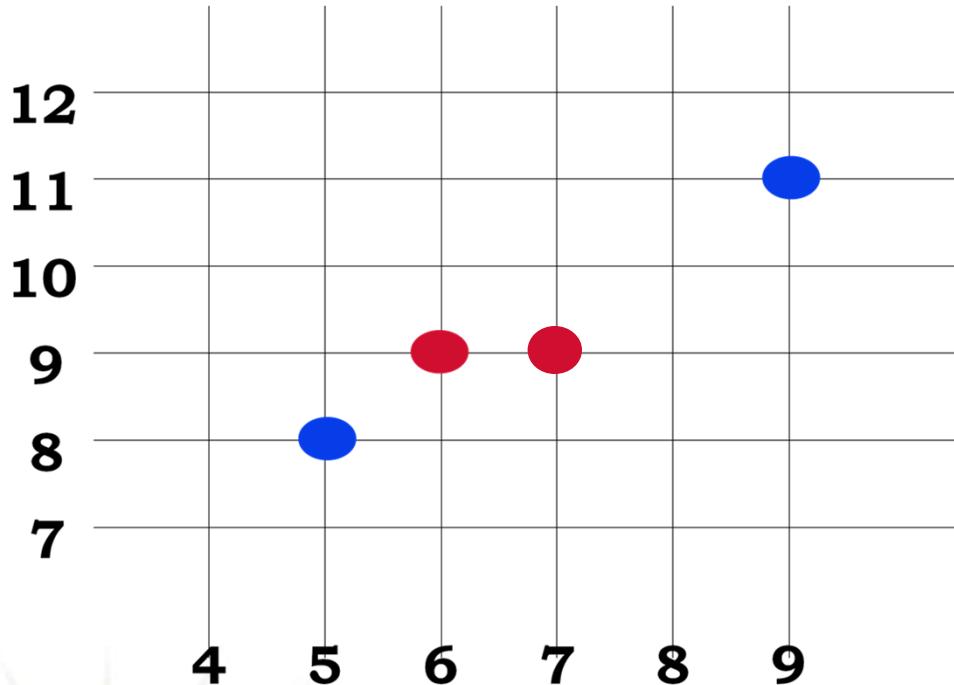


Line from (5,8) to (9,11)  
 $dx = 2*(9-5)$ ;  $dy = 2*(11-8)$ ;  
 $dy/dx = -2$ ;  $D = -1$ ;

So next pixel is NE

# Rasterization

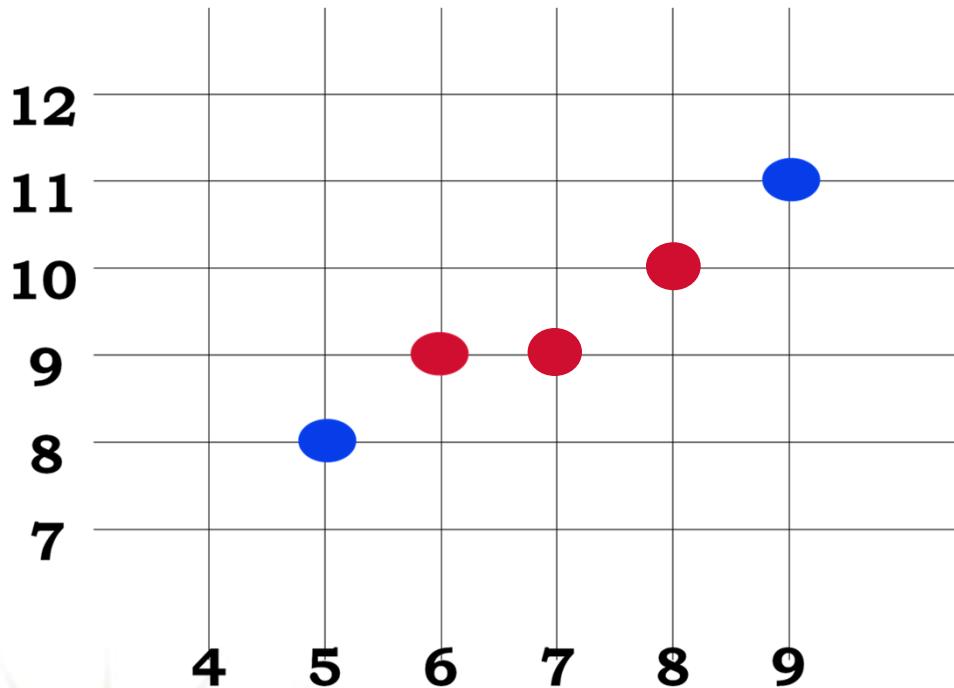
## Bresenham's Algorithm: An example



$D = -1 - (-2) = 1$   
So next pixel is  
E

# Rasterization

## Bresenham's Algorithm: An example



$D = 1 - (4) = -3$   
So next pixel is  
NE

# Rasterization

## Bresenham's Algorithm VI

- Need different cases to handle  $m > 1$
- Highly efficient
- Easy to implement in hardware and software
- Widely used
- Has been extended to do antialiasing
- Has been extended to rasterize circle and ellipse

# Rasterization

## Scan Conversion of Polygons

- **Multiple tasks:**
  - Filling polygon (inside/outside)
  - Pixel shading (color interpolation)
  - Blending (accumulation, not just writing)
  - Depth values (z-buffer, hidden-surface removal)
  - Texture coordinate interpolation (texture mapping)
- **Hardware efficiency is critical**
- Many algorithms for filling (inside/outside)
- **Much fewer that handle all tasks well**

# Rasterization

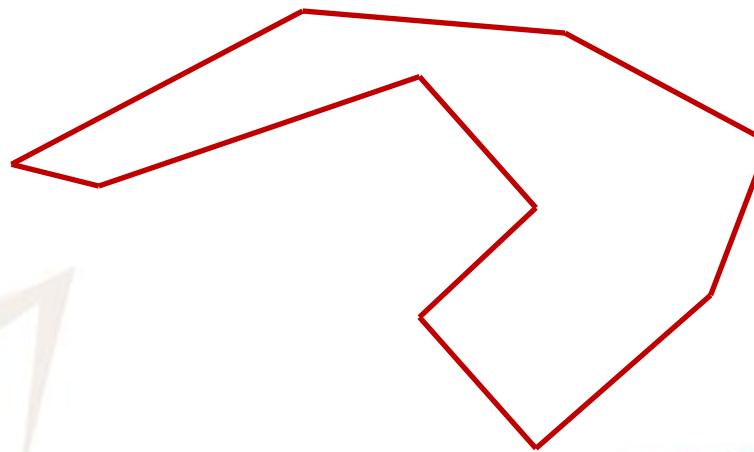
## Polygon filling

- Simplest method to fill a polygonal area is to test every pixel in the raster to see if it lies inside the polygon.
- There are *two methods* to make an inside check
  - even-odd test
  - non-zero winding number test
- Works for concave and convex polygons

# Rasterization

## Inside/Outside Testing for Polygons

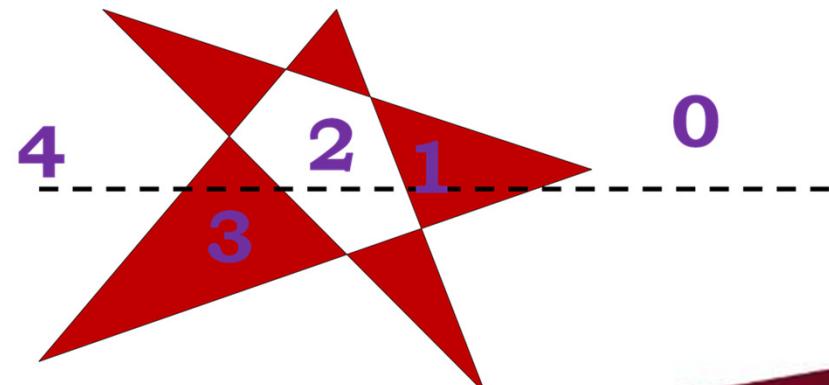
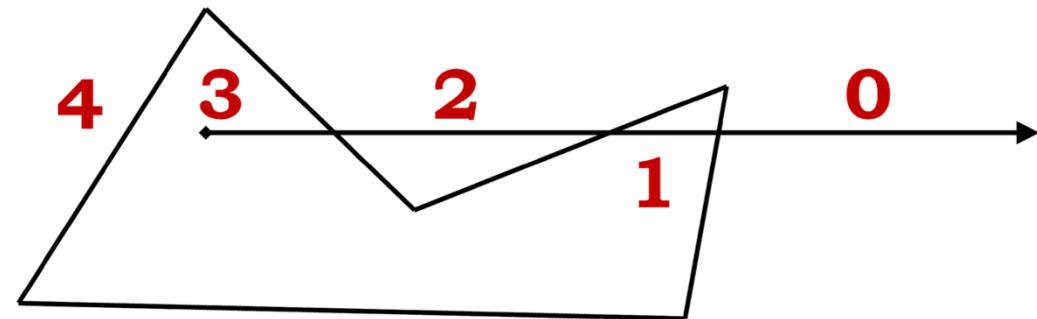
- **Testing for inside/outside of a polygon**
- **Two definitions of inside :**
  - Even-Odd parity
  - Non-zero Winding number



# Rasterization

## Even-Odd Parity Test

- **Even-Odd parity**
- **Even no. crossings :**  
**Outside polygon**
- **Odd no. crossings :**  
**Inside polygon**
- **Doesn't work for self-intersecting polygons**

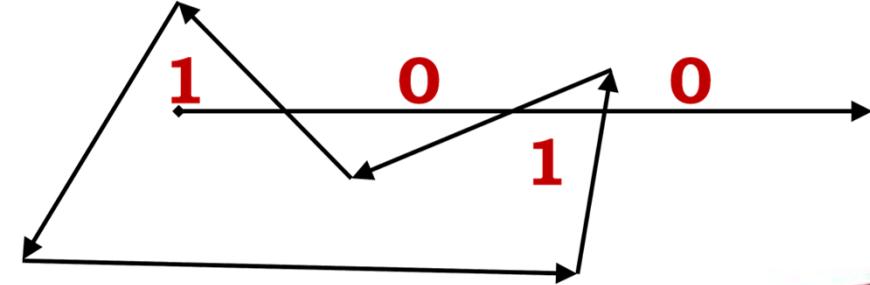
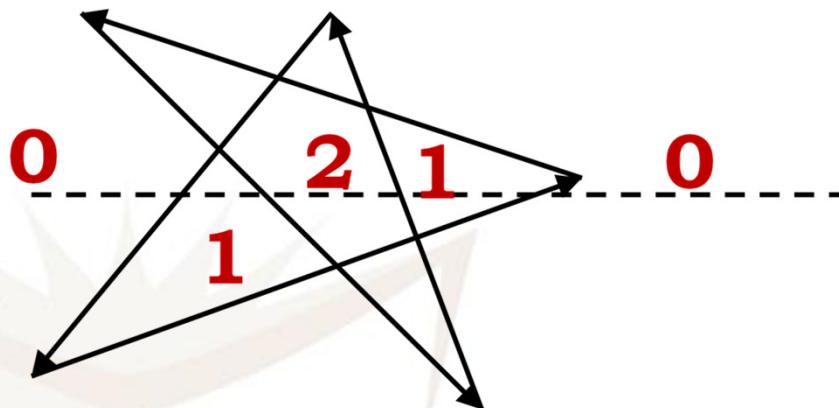


# Rasterization

## Non-zero Winding Number Test

### Non-zero Winding Number

- Use direction of line as well as crossing
- Set a value,  $e = 0$
- For right-left crossings,  $e++$ , for left-right  $e--$
- For inside,  $e \neq 0$



# Rasterization

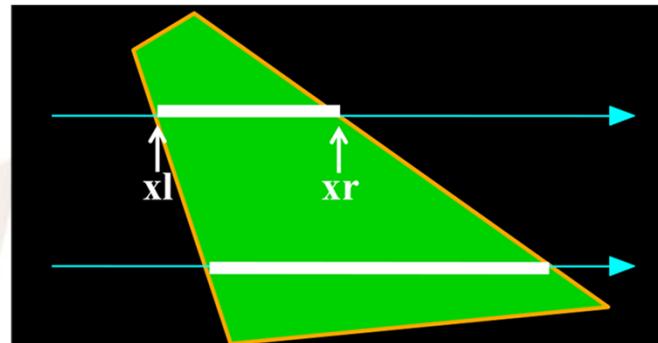
## Scan Line Methods

- Makes use of the *coherence properties*
  - **Spatial coherence:** Except at the boundary edges, adjacent pixels are likely to have the same characteristics
  - **Span coherence:** Pixels in a scan line will be set to same values for solid shaded primitives
  - **Scan line coherence:** Pixels in the adjacent scan lines are likely to have the same characteristics
- Uses *intersections between area boundaries and scan lines* to identify pixels that are inside the area

# Rasterization

## Filling Convex Polygons

- Find top and bottom vertices
- List edges along left and right sides
- For each scan line from bottom to top
  - Find left and right endpoints of span,  $x_l$ , and  $x_r$
  - Fill pixels between  $x_l$ , and  $x_r$
  - Can use Bresenham's algorithm to update  $x_l$ , and  $x_r$



# Rasterization

## Scan Line Method

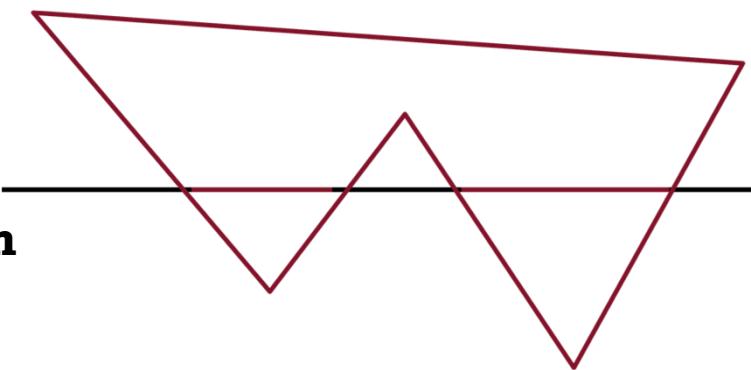
- A polygon fits easily into scan-line algorithms
- Uses intersections between area boundaries and scan lines to identify pixels that are inside the area
- Proceeding from left to right the intersections are paired and intervening pixels are set to the specified intensity

### Algorithm

Find the intersections of the scan line with all the edges in the polygon

Sort the intersections by increasing X-coordinates

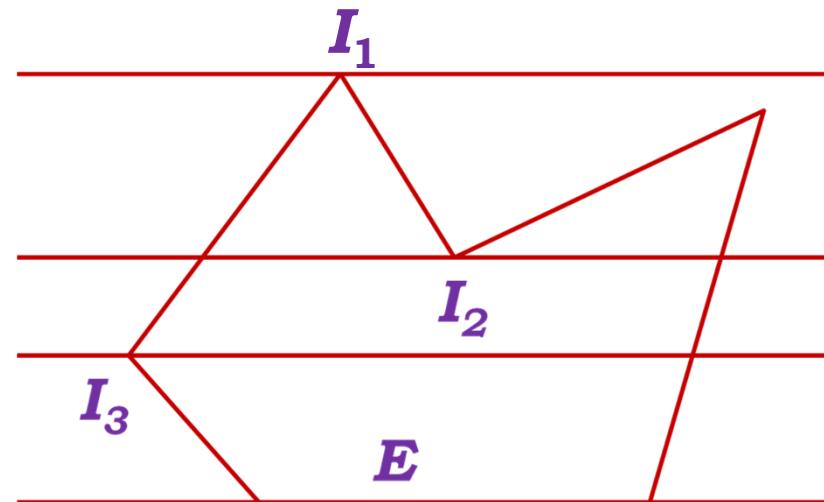
Fill the pixels between pair of intersections



# Rasterization

## Special cases for Scan Line Method

- Overall topology should be considered for intersection at the vertices
- Intersections like  $I_1$  and  $I_2$  should be considered as *two intersections*
- Intersections like  $I_3$  should be considered as *one intersection*
- Horizontal edges like  $E$  need not be considered



# Rasterization

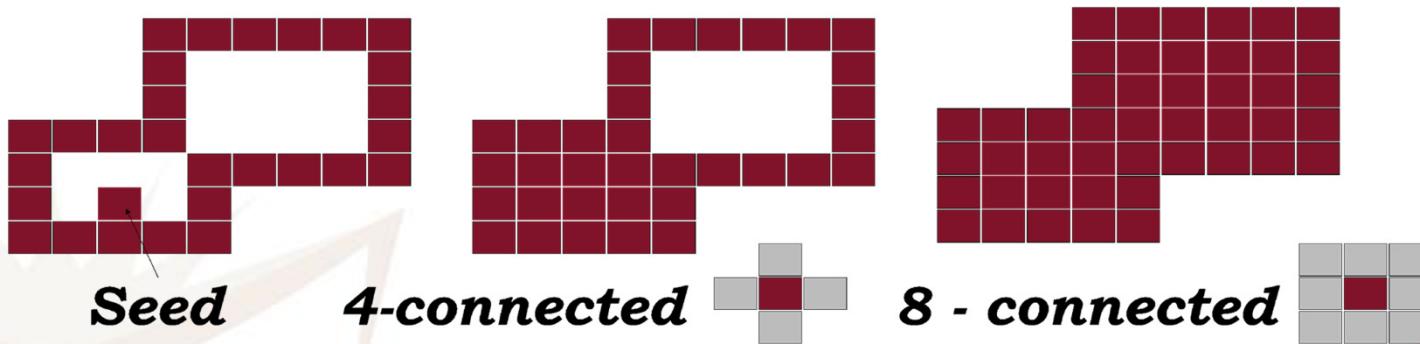
## Advantages of Scan Line method

- The algorithm is **efficient**
- *Each pixel is visited only once*
- Shading algorithms could be easily integrated with this method to obtain shaded area

# Rasterization

## Flood Fill Algorithms

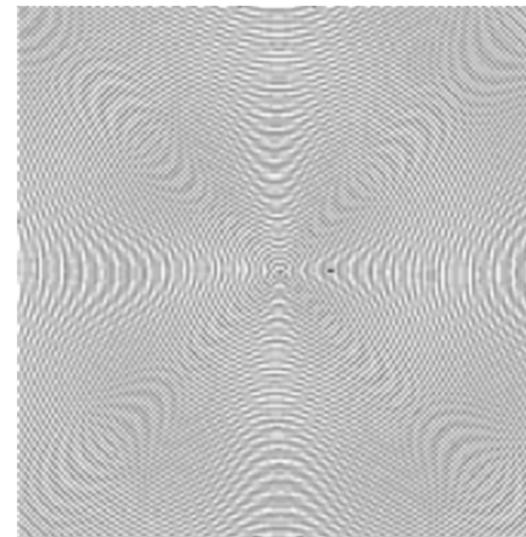
- Draw outline of polygon
- Pick color seed
- Color surrounding pixels and recurse
- Must be able to test boundary and duplication
- More appropriate for *drawing than rendering*
- Useful in interactive paint packages



# Rasterization

## Antialiasing

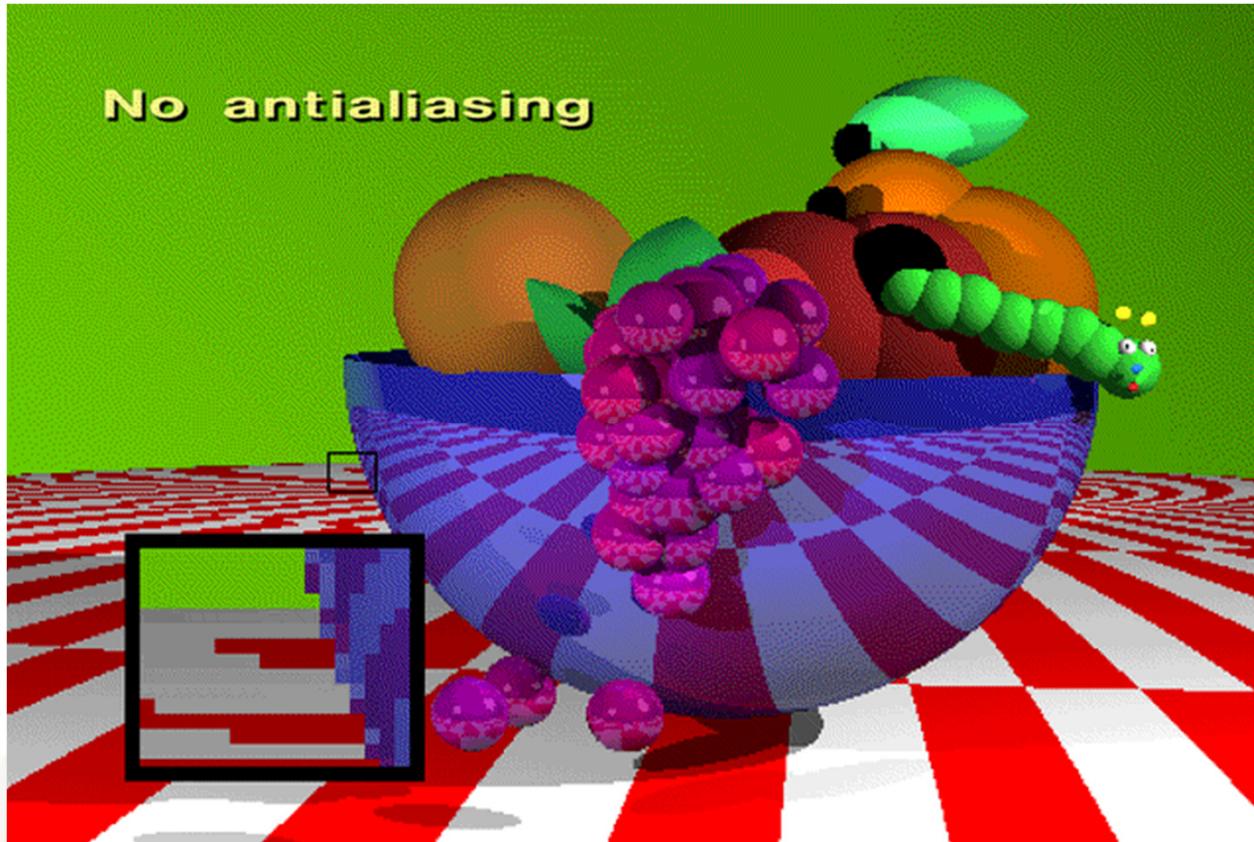
- Artifacts created during scan conversion
- **Inevitable** (going from continuous to discrete)
- Aliasing (name from digital signal processing): we sample a continuous image at grid points
- **Effect**
  - jagged edges
  - moire patterns



Moire pattern from sandlotscience.com

# Rasterization

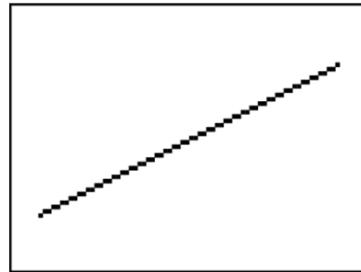
## More Aliasing



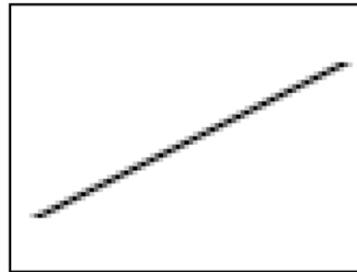
# Rasterization

## Antialiasing for Line Segments

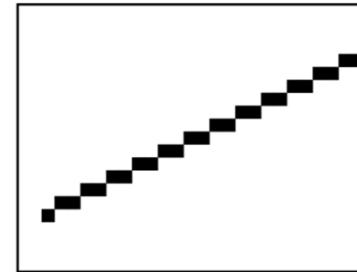
- Use area averaging at boundary



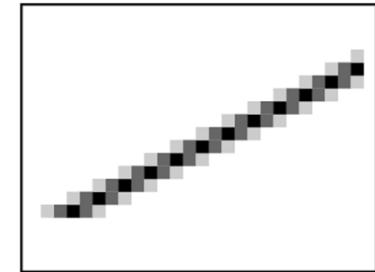
(a)



(b)

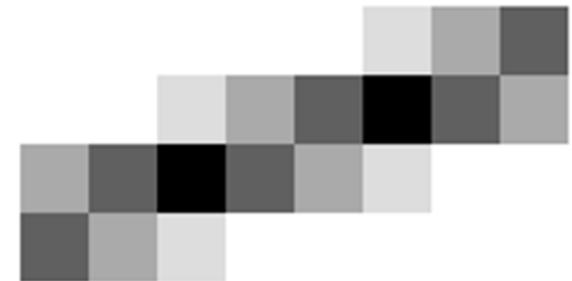


(c)



(d)

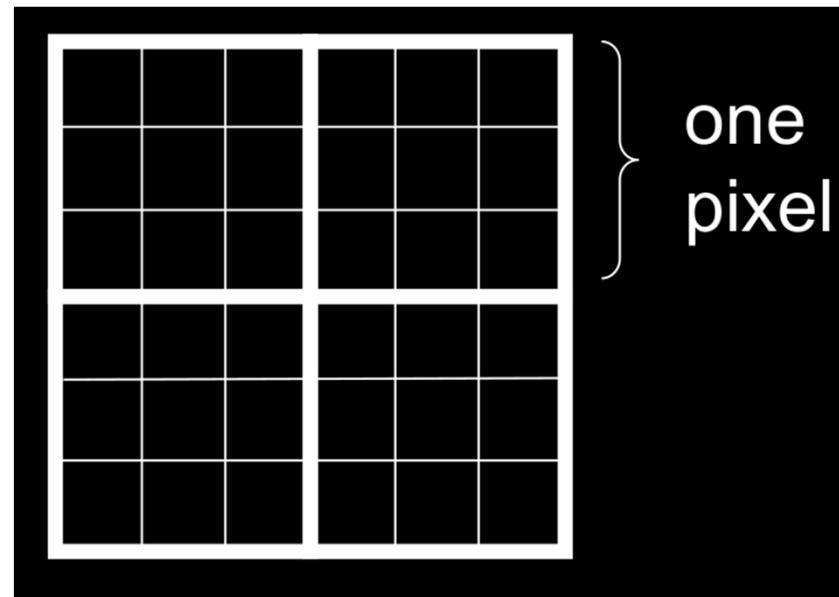
- (c) is aliased, magnified
- (d) is antialiased, magnified
- (In Bresenham's algorithm set grey level color to pixels adjacent to the chosen pixel)



# Rasterization

## Antialiasing by Supersampling

- Mostly for off-line rendering (e.g. ray tracing)
- Render, say, 3x3 grid of mini pixels
- Average results using a filter
- Can be done adaptively
- Stop if colors are similar
- Subdivide at discontinuities



# Rasterization

## Supersampling Example

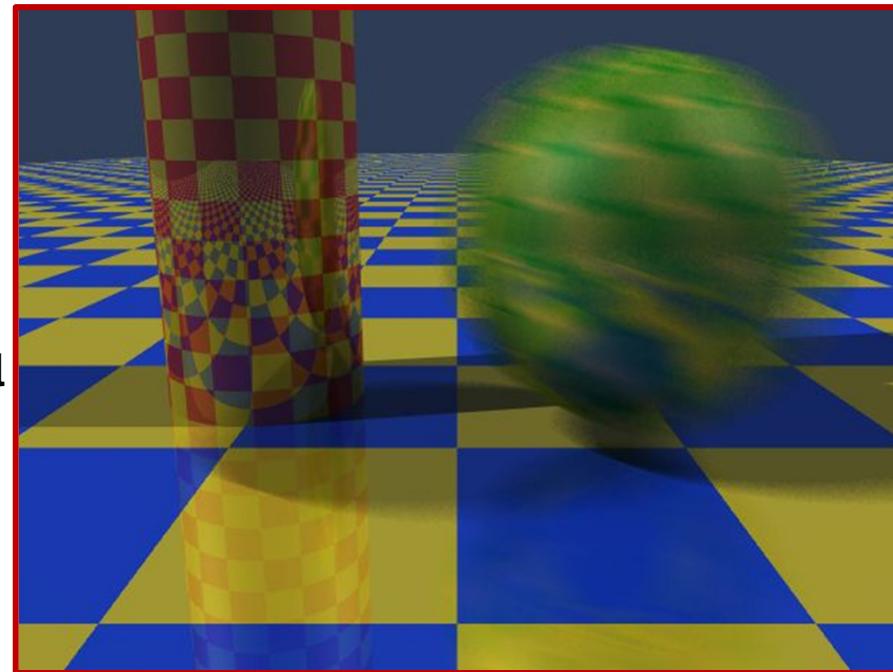


- Other improvements
  - Stochastic sampling: avoid sample position repetitions
  - Stratified sampling (jittering): perturb a regular grid of samples

# Rasterization

## Temporal Aliasing

- Sampling rate is frame rate (30 Hz for video)
- Example: spokes of wagon wheel in movies
- Solution: supersample in time and average
  - Fast moving objects are blurred
  - Happens automatically with real hardware (photo and video cameras)
  - Exposure time is important (shutter speed)
- Effect is called motion blur



# Rasterization

## Wagon Wheel Effect



# Rasterization

## Motion Blur Example

- Achieved by stochastic sampling in time





**COMP 371**

# **Computer Graphics**

**Session 11**

**PROGRAMMABLE SHADERS**

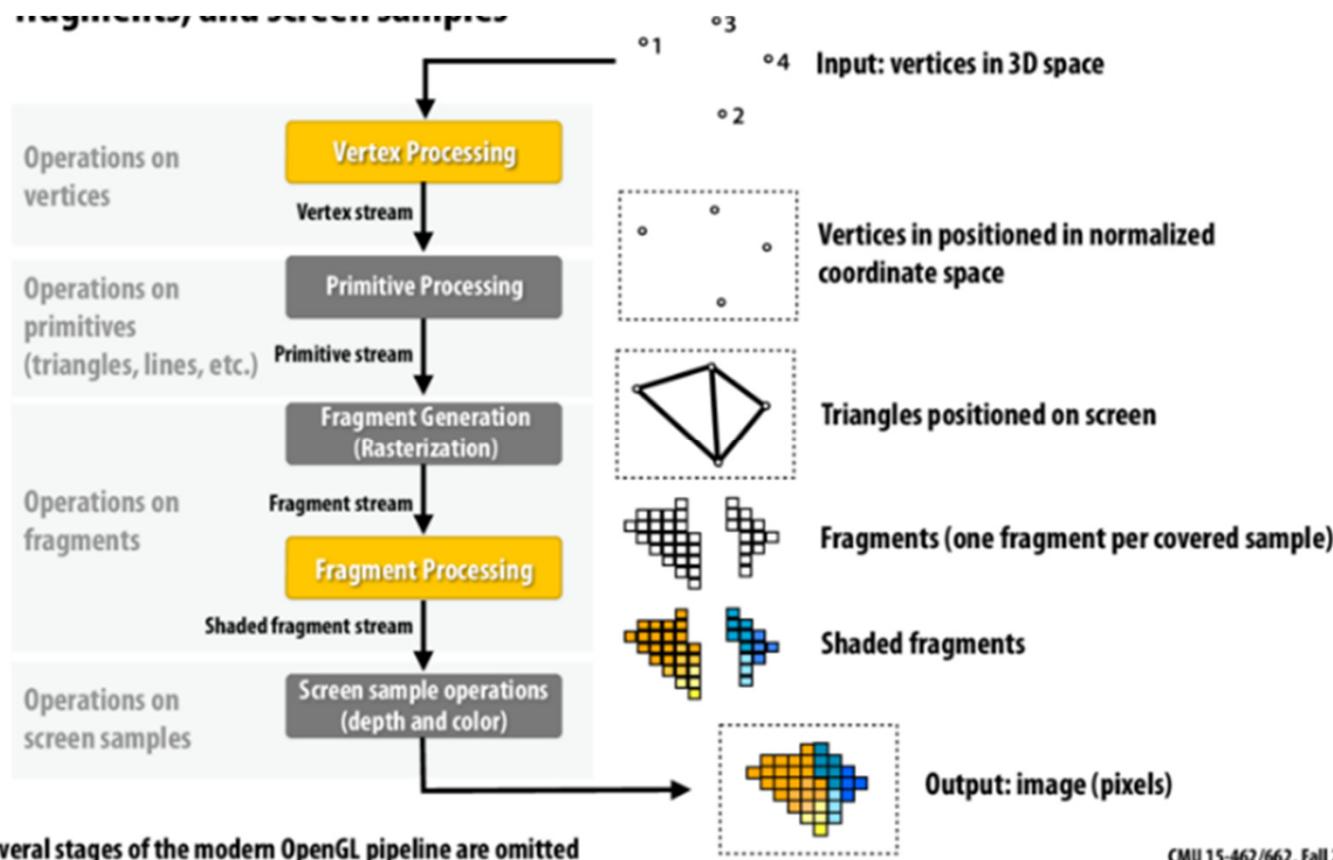


# Lecture Overview

## □ GPU Programming

# Programmable Shaders

## GPU Pipeline

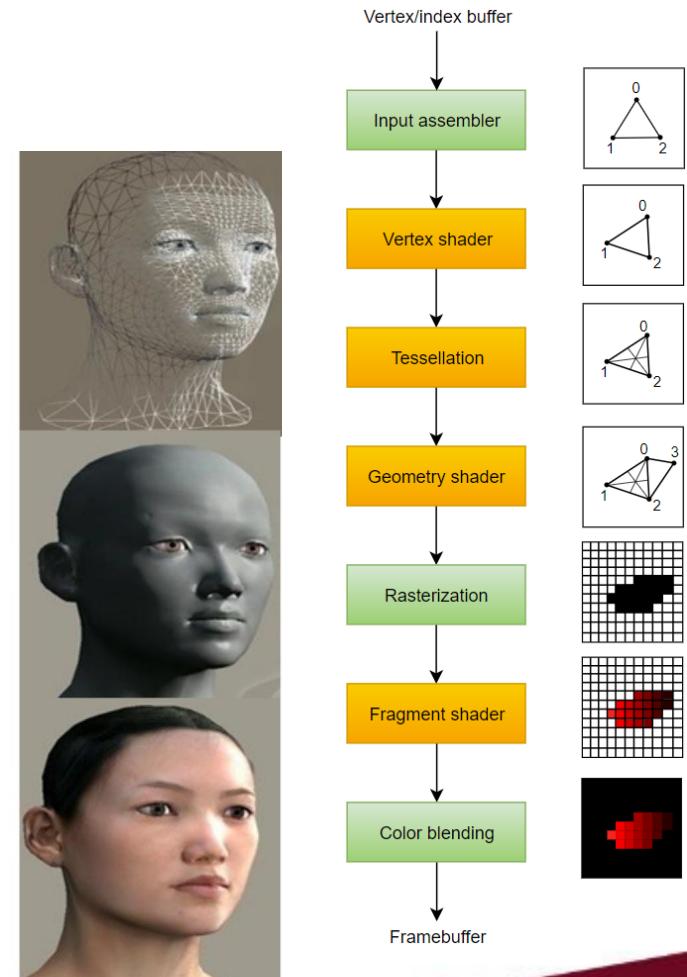
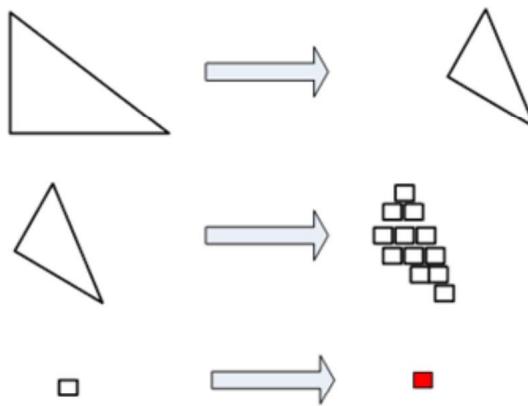
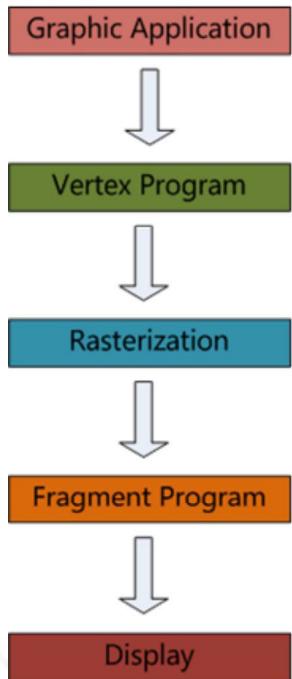


CMU 15-462/662, Fall 2015

Courtesy CMU

# Programmable Shaders

## GPU Pipeline



Courtesy: Vulkan-Tutorial

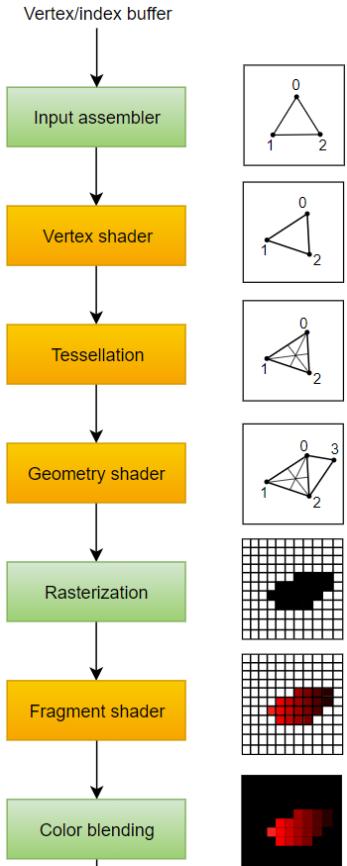
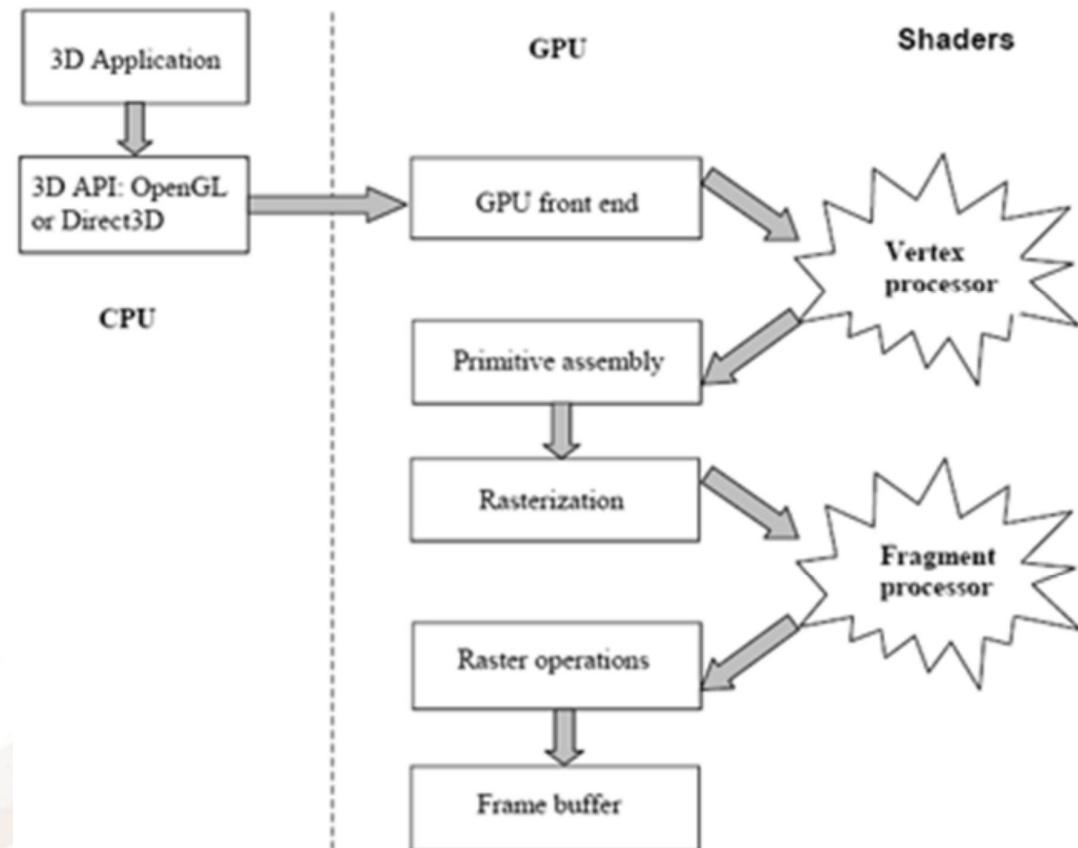
# Programmable Shaders

## Programmable Hardware

- Most modern graphics cards allow parts of the GPU pipeline to be modified by a developer
  - Increased functionality with hardware support
  - Complex graphics algorithms with fast implementation
  - General Computing
- Until recently, the programmable parts were
  - Vertex Processor
  - Fragment (Pixel) Processor
- OpenGL 4 also allows the programming of *Geometric processors and Compute Processors for General Purpose GPU programming*
- All these programs are called **Shaders**

# Programmable Shaders

## Programmable GPU Pipeline



Courtesy: Vulkan-Tutorial

# Programmable Shaders

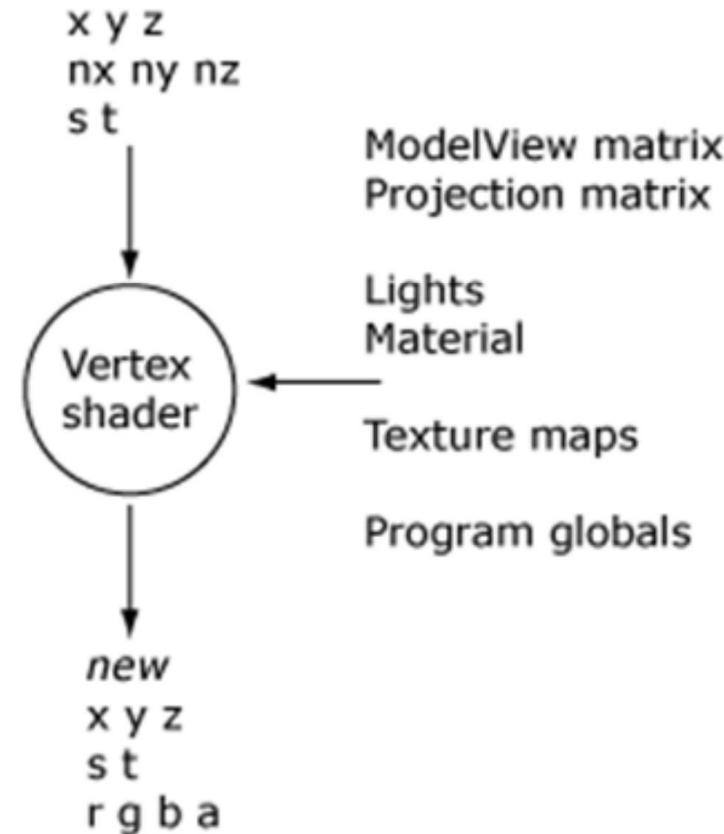
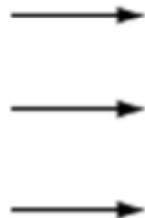
## Vertex Processor Responsibilities

- **Vertex transformation**
- **Normal transformation and normalization**
- **Texture Coordinates generation and transformation**
- **Lighting (per vertex)**
- **Color & Material (per vertex)**
- **A Vertex shader replaces the entire functionality of the fixed vertex processor!**
- **When writing a shader, one must replace the *entire functionality***
- **Vertex Shaders operate simultaneously on all vertices (in parallel)**
- **No vertex information can affect another vertex**

# Programmable Shaders

## Vertex Shader

Load shader source code  
Compile into machine code  
Bind vertex shader  
...  
**Set Shader Constants**  
...  
**Send Geometry (Vertices)**



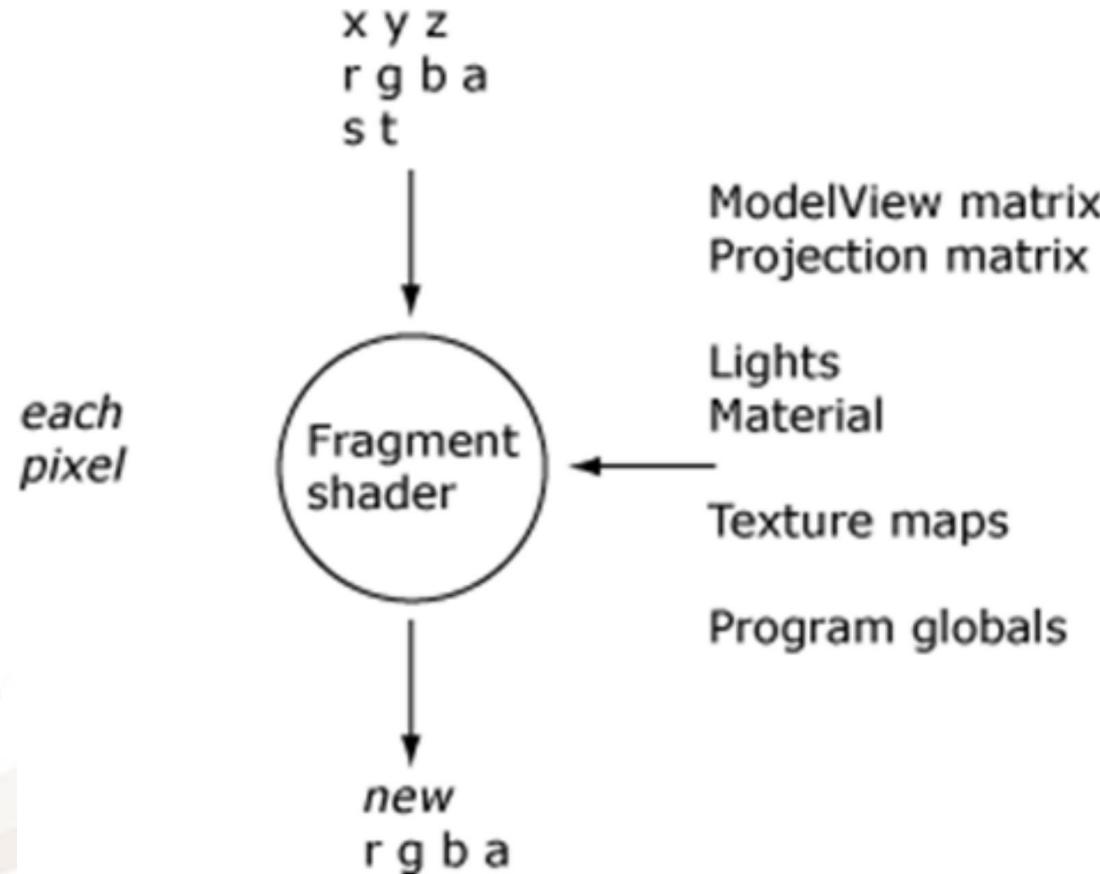
# Programmable Shaders

## Fragment Processor

- Operates on the fragments AFTER Interpolation and Rasterization of vertex data
- Allows the following programming:
- Operation on interpolated vertex values (**Phong** - in the next lecture)
- Texture access & application
- Fog
- Color sum, *etc.*
- All fragment shaders work in parallel
- Again, no access to neighboring fragments
- Does not replace back-end operations (alpha blending, and such).

# Programmable Shaders

## Fragment Shader



# Programmable Shaders

## OpenGL in Application Setup

- Create a Framebuffer, Depth Buffer, all the buffers you need, with the resolution you need
  - Customize with `glfwWindowHint`
  - Create with **GLFW**: `glfwCreateWindow`
- Set OpenGL Rendering States
  - Example
  - `glClearColor(0.0f, 0.0f, 0.0f, 0.0f);`
  - `glEnable(GL_DEPTH_TEST);`
  - `glDepthFunc(GL_LESS);`

# Programmable Shaders

## Create Vertex Array/Buffer Objects

- Create a Vertex Array Object which will manage your Vertex Buffer Objects  
`glGenVertexArrays(1, &vao);`
- Vertex Buffer Objects contains geometry information in Model Space. Draw calls will later use the VBO data
- Create an Array containing all of your geometry data (Position, Normals, Colors, UV Coordinates, etc.)  
`float vertexBuffer[ ] = {1.0f, 2.0f, ....};`
- Create a Vertex Buffer Object

`glGenBuffers(1, &vbo);`

`glBindBuffer(GL_ARRAY_BUFFER, vbo);`

`glBufferData(GL_ARRAY_BUFFER, sizeof(vertexBuffer),  
vertexBuffer, GL_STATIC_DRAW);`

# Programmable Shaders

## Load Shaders

- You can have multiple shader pairs (Vertex/Fragment) for different type of geometry
- To create Shader Objects, you must load, compile, and link your shaders (normally in the program initialization)
- `glCreateShader`
- `glCompileShader`
- `glAttachShader`
- `glLinkProgram`

# Programmable Shaders

## Frame Initialization

- When OpenGL Setup is done, you're ready to draw!
- Before drawing, make sure your scene is updated for the next frame. This means, the scene goes forward by a time step (typically 1/60 second)
- Animations, Physics, Camera, etc...
- Clear the rendering context to start drawing a new frame from scratch (frame buffer, depth buffer, etc)
- `glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);`

# Programmable Shaders

## Drawing

- Bind Shader

```
glUseProgram(shaderID);
```

- Set Shader Constants (eg: VAR matrix in Shader)

```
GLuint loc = glGetUniformLocation(shaderID, "VAR");
glUniformMatrix4fv(loc, 1, GL_FALSE, &varMatrix);
```

- Bind Vertex Array Object

```
glBindVertexArray(vao);
```

# Programmable Shaders

## Drawing

- Set Shader Input Data (VBO)

- `glEnableVertexAttribArray(0); // 1st input in vertex shader  
glBindBuffer(GL_ARRAY_BUFFER, vbo);`
  - `glVertexAttribPointer(0, // attribute  
3, // size  
GL_FLOAT, // type  
GL_FALSE, // normalized?  
sizeof(Vertex), // stride  
(void*)0 ); // offset`

- Draw Call

```
glDrawArrays(GL_TRIANGLE_STRIP, 0, numVertices);
```

- Cleanup `glDisableVertexAttribArray(0);`

# Programmable Shaders

## Finishing a Frame Swap Buffers

- Modern renderers typically use **double-buffering**
- While front-buffer is sent to video controller
- Draw in the back-buffer
- When done drawing, **swap the buffers**
- With GLFW this is achieved with the command:

**glfwSwapBuffers(...)**

# Programmable Shaders

## Shading Languages

- The first approaches used an assembler language of the cards (with significantly different instruction sets on both shaders)
- Both OpenGL and DirectX also provide a different API
  - load shaders
  - pass user-defined data
  - delete
- High level languages
  - Cg (NVidia), GLSL (OpenGL), HLSL (DirectX)

# Programmable Shaders

## GLSL

- A **high-level procedural language** (similar to C++).
- As of OpenGL 2.0, it is a standard, with a simple **gl\_XYZ API from OpenGL applications**
  - Former implementations of GLSL API to OpenGL used *OpenGL extensions*
- The same language, with small differences, is used for all types of shaders
- Natively supports “**graphical needs**” (such as containing types like vectors and matrices).

# Programmable Shaders

## Can I use GLSL

- Almost every current high-end GPU supports it
- How to check
  - Run `glewinfo.exe` [on windows]
  - It will tell you which features you can use

# Programmable Shaders

## Language Syntax

- **Similar to C/C++**
- **Data Types**
  - **vector, matrix, texture**
- **Control Flow**
- **same as C.**
  - e.g.: **for, if, while**

# Programmable Shaders

## Language Syntax

- **Built-in functions for**
  - **math**
  - **interpolation (mix, step, smoothstep)**
  - **geometric**
  - **matrix**
  - **vector relational**
  - **texture**
  - **shadow**
  - **vertex, fragment**
  - **noise**

# Programmable Shaders

## Communication

- OpenGL → Shaders
  - Uniform variables
- Application → Vertex Shader → Fragment shader
  - Input variables
  - Output variables

# Programmable Shaders

## Uniform Variables

- **uniform datatype dataname**
- **suitable for values that remain constant along a primitive, frame, or even the whole scene**
- **can be read (but not written) in both vertex and fragment shaders**

# Programmable Shaders

## Uniform Variables

```
uniform float specIntensity; // in shader

// in application
...
GLint loc1;
float specIntensity = 0.98;

// gets the location of the variable.
loc1 = glGetUniformLocation(p, "specIntensity");

// sets the value.
glUniform1f (loc1,specIntensity);
...
```

# Programmable Shaders

## Typical Uniform Variables

- **Some Matrices**
  - `uniform mat4 ModelViewMatrix;`
  - `uniform mat3 NormalMatrix;`
  - `uniform mat4 TextureMatrix[n];`
- **Lighting, depth, material**
  - defined as some structures
  - `uniform MaterialParameters FrontMaterial;`
  - `uniform LightSourceParameters LightSource[MaxLights];`
- **And many variables**
  - it refers *OpenGL state variables*

# Programmable Shaders

## Input Variables

- **in datatype dataname**
- *Read only variables*
- In **Vertex Shader**
  - Typically Vertex Data (position, normal, uv, color)
- In **Fragment Shader**
  - *Output variables* from Vertex Shader become input to Fragment Shader
  - The *variable names must match*

# Programmable Shaders

## Input Variable Example for Vertex Shader

```
in vec3 position; // in shader

// in application
...
//gets the location of the variable.
positionID = glGetUniformLocation (p,"position");
...

glEnableVertexAttribArray(positionID);
glBindBuffer(GL_ARRAY_BUFFER, vbo);
 glVertexAttribPointer(...);
```

**or:**

```
...
glBindAttribLocation(p, 0, "position");
.... glEnableVertexAttribArray(0);
....
```

# Programmable Shaders

## Output Variables

- **out datatype dataname**
- **In Vertex Shader**
  - Output variables are inputs in Fragment Shader
  - Values will be interpolated in Fragment Shader
- **Declared in both vertex and fragment shader**
  - *type and name should be matched*

# Programmable Shaders

```
//Vertex Shader Program
```

```
#version 330
```

```
uniform mat4 view_matrix;
uniform mat4 model_matrix, proj_matrix;
```

```
in vec3 in_Position, in_Color;
out vec3 out_Color;
```

```
void main () {
mat4 MVP = proj_matrix * view_matrix * model_matrix;
gl_Position = MVP * vec4 (in_Position, 1.0);
out_Color = in_Color;
}
```

```
//Fragment Shader Program
```

```
#version 330
```

```
in vec3 out_Color;
out vec4 frag_Color;
```

```
void main () {
frag_Color = vec4(out_Color, 1.0);
}
```

After loading the shaders:

```
view_matrix_id = glGetUniformLocation(ProgramID, "view_matrix");
model_matrix_id = glGetUniformLocation(ProgramID, "model_matrix");
proj_matrix_id = glGetUniformLocation(ProgramID, "proj_matrix");
glBindAttribLocation(ProgramID, 0, "in_Position");
glBindAttribLocation(ProgramID, 1, "in_Color");
```

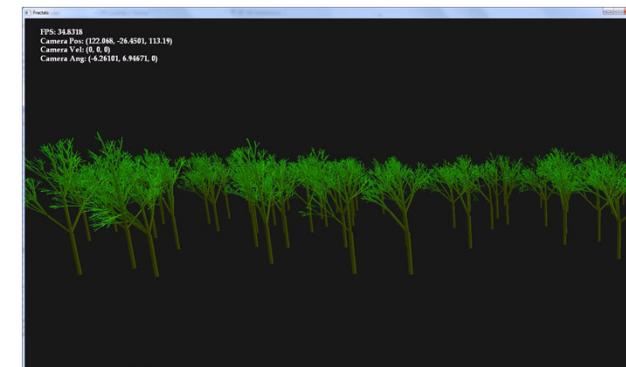
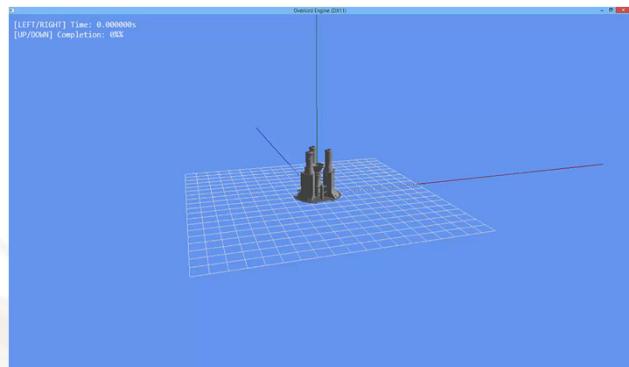
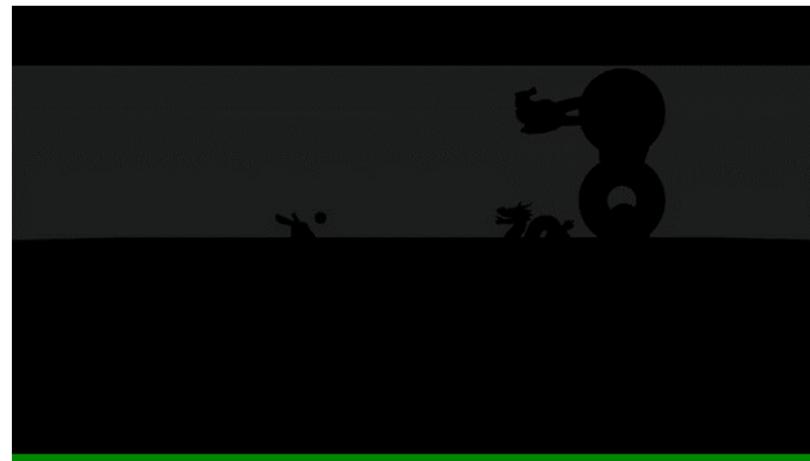
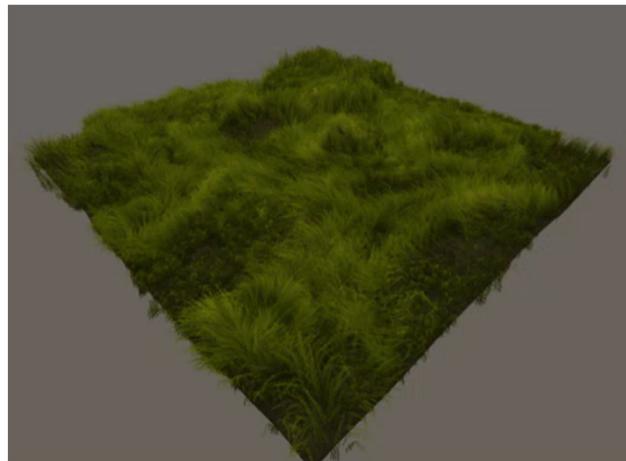
At every drawing iteration:

```
glUseProgram(shader_program);

glUniformMatrix4fv(proj_matrix_id, 1, GL_FALSE, glm::value_ptr(proj_matrix));
glUniformMatrix4fv(view_matrix_id, 1, GL_FALSE, glm::value_ptr(view_matrix));
glUniformMatrix4fv(model_matrix_id, 1, GL_FALSE, glm::value_ptr(model_matrix));
```

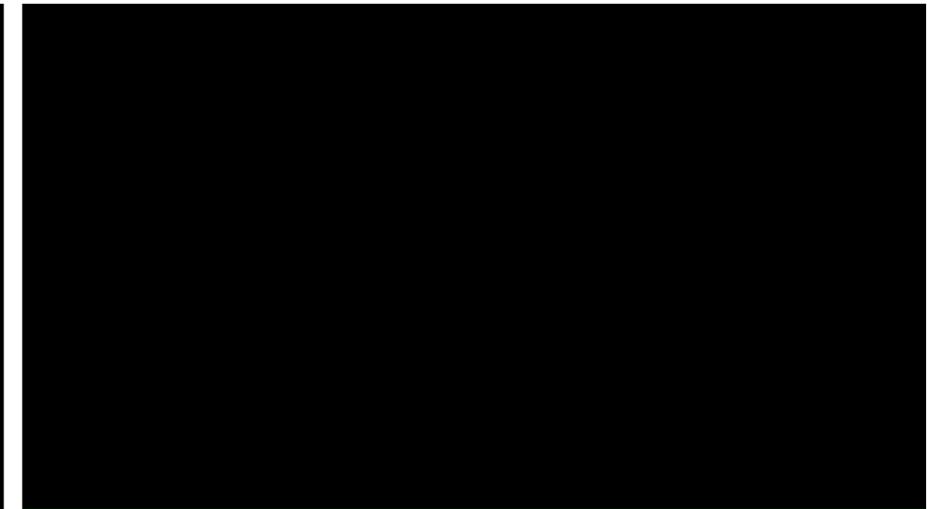
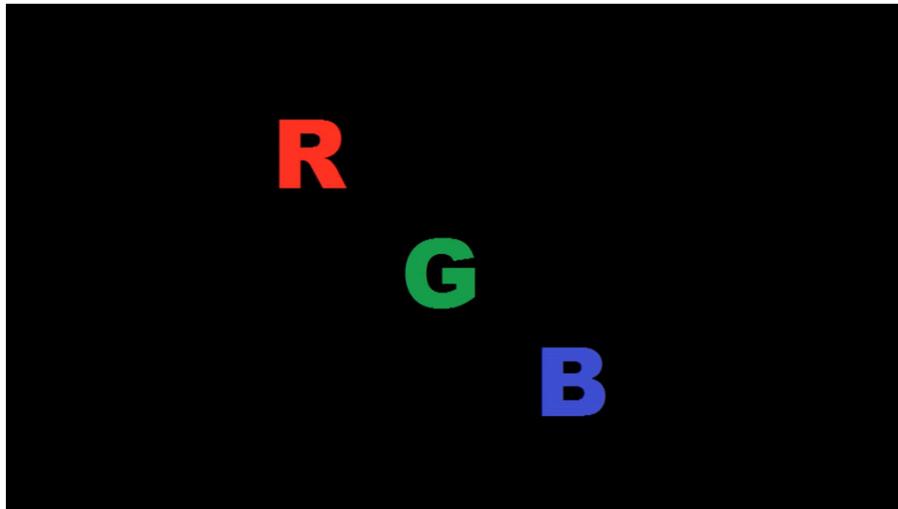
# Programmable Shaders

## Shader Examples



# Programmable Shaders

## Shader Examples





**COMP 371**

# **Computer Graphics**

**Session 12**

**LIGHTING AND SHADING**

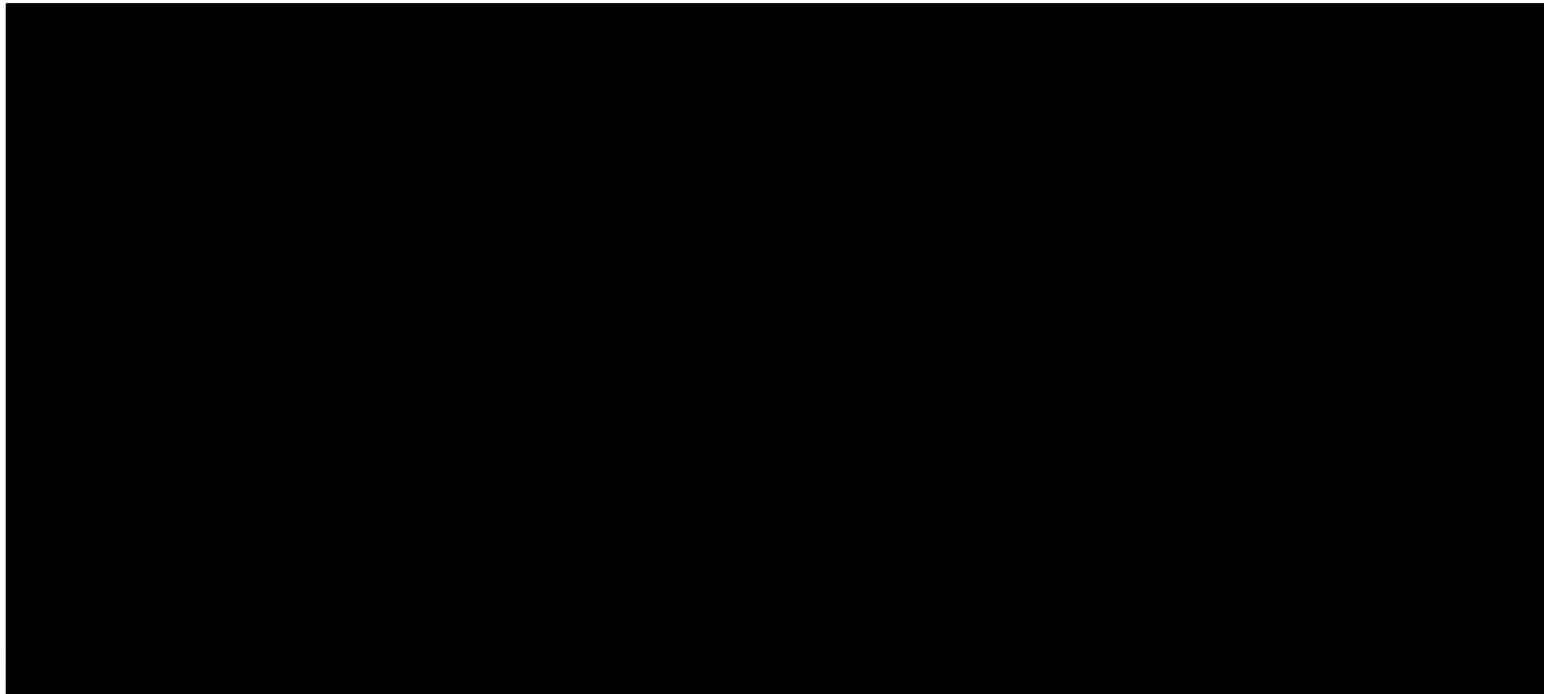


# Lecture Overview

- **Review of last class**
- **Global and Local Illumination**
- **Normal Vectors**
- **Light Sources**
- **Phong Illumination Model**

# Lighting and Shading

Without light ... we don't see our world!



# Lighting and Shading

**Without light ... we don't see our world!**



**Without shading, objects do not look three dimensional!**

# Lighting and Shading

**Without light ... we don't see our world!**



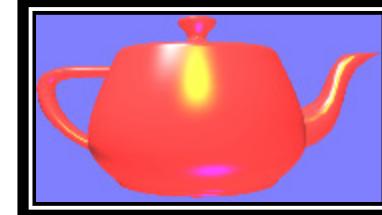
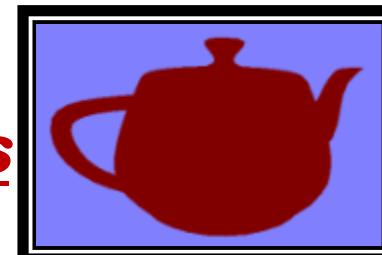
**Without shading, objects do not look  
three dimensional!**

Image: <https://blenderartists.org>

# Lighting and Shading

## Need for Illumination?

- Important for perception and understanding of 3D scenes
- Has *visual cues for humans*
- Provides information about
  - Positioning of light sources
  - Characteristics of light
  - Sources
  - Materials
  - Viewpoint



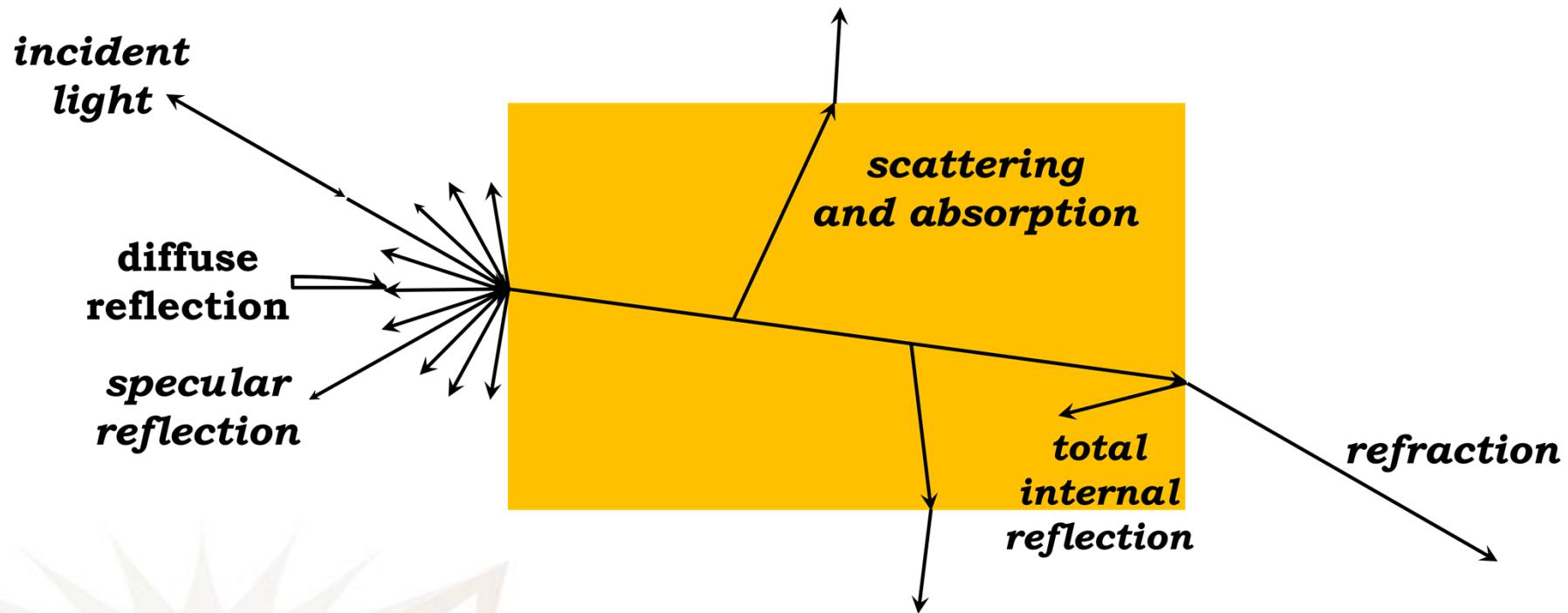
# Lighting and Shading

## Illumination

- **Illumination** is the complete description of all the light striking a particular point on a particular surface
- **Color** at a point on an object is decided by the properties of the light leaving that point
- Knowing the **illumination** and the **surface physics** at a point on a surface, we can *determine the properties of the light leaving that point*
- In order to generate realistic images we *need to understand how light interacts with the surface of objects*

# Lighting and Shading

## Interaction of light with a Solid



# Lighting and Shading

## Interaction of Light

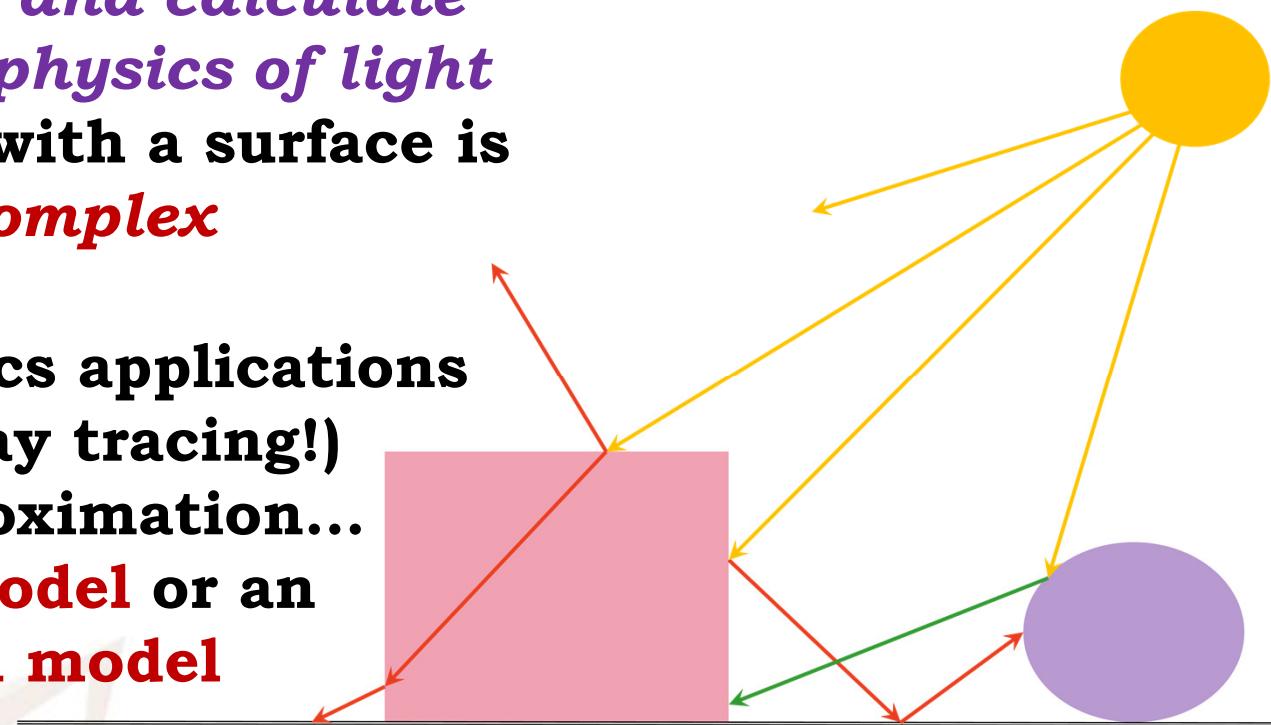
- There are two important illumination phenomena
  - interaction of light with the *boundaries between materials*
  - scattering and absorption of light as it *passes through the material*
- **Boundaries between materials** are surfaces which make up the environment
- Light striking a boundary is either **reflected** or **transmitted**
  - For **opaque** materials light is *absorbed on passing through the boundary*

# Lighting and Shading

## Light Interaction in a Scene

To simulate and calculate the precise physics of light interacting with a surface is **extremely complex**

Most graphics applications (including ray tracing!) use an approximation... a **lighting model** or an **illumination model**



# Lighting and Shading

## Illumination Models

- A surface point could be illuminated by
  - **local illumination**, light directly emitted by light sources
  - **global illumination**, light reflected from and transmitted through its own and other surfaces
- Illumination models express the *factors which determine the surface color* at a given point on a surface
- Illumination models compute the color at a point in terms of *both local and global illumination*

# Lighting and Shading

## Global Illumination

- Ray Tracing
- Radiosity
- Photon Mapping
- Follow light as it progresses through the scene
- Accurate, but expensive (*off-line*)



# Lighting and Shading

## Global Illumination

- Ray Tracing Example

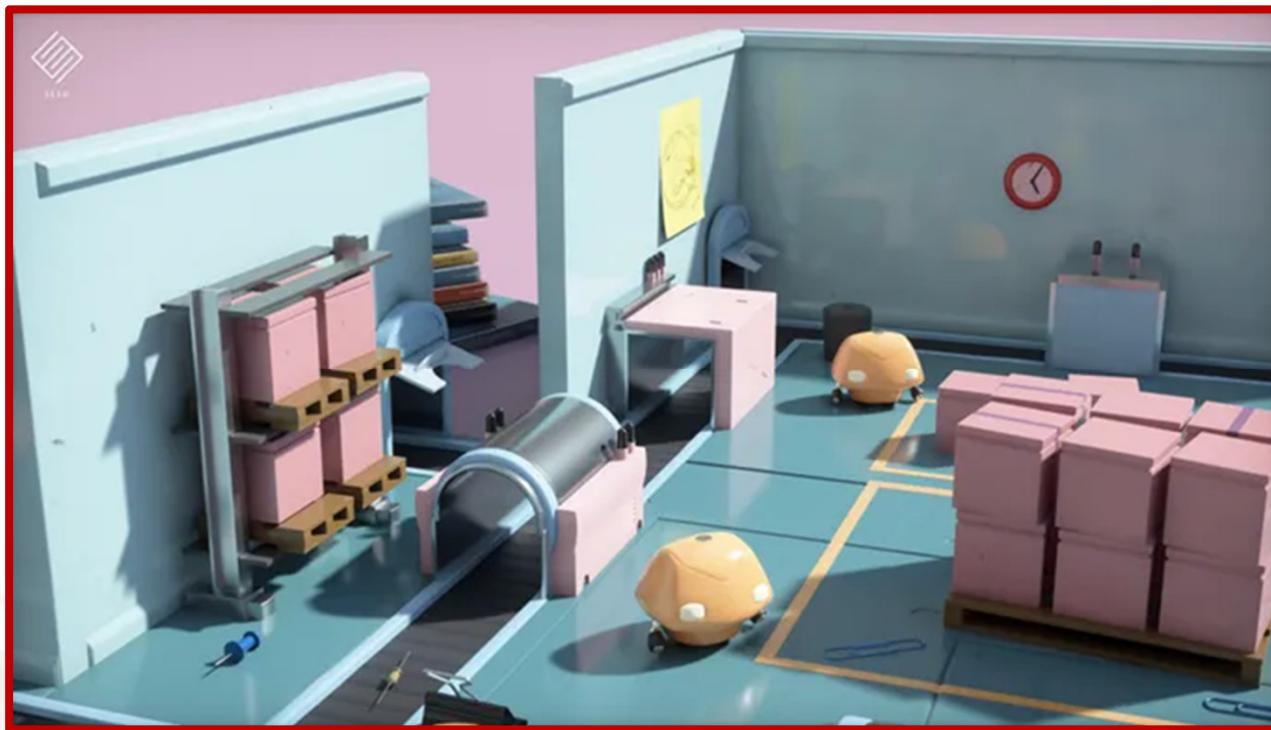


Image: <https://www.online-tech-tips.com/>

# Lighting and Shading

## Global Illumination

- Ray Tracing Example

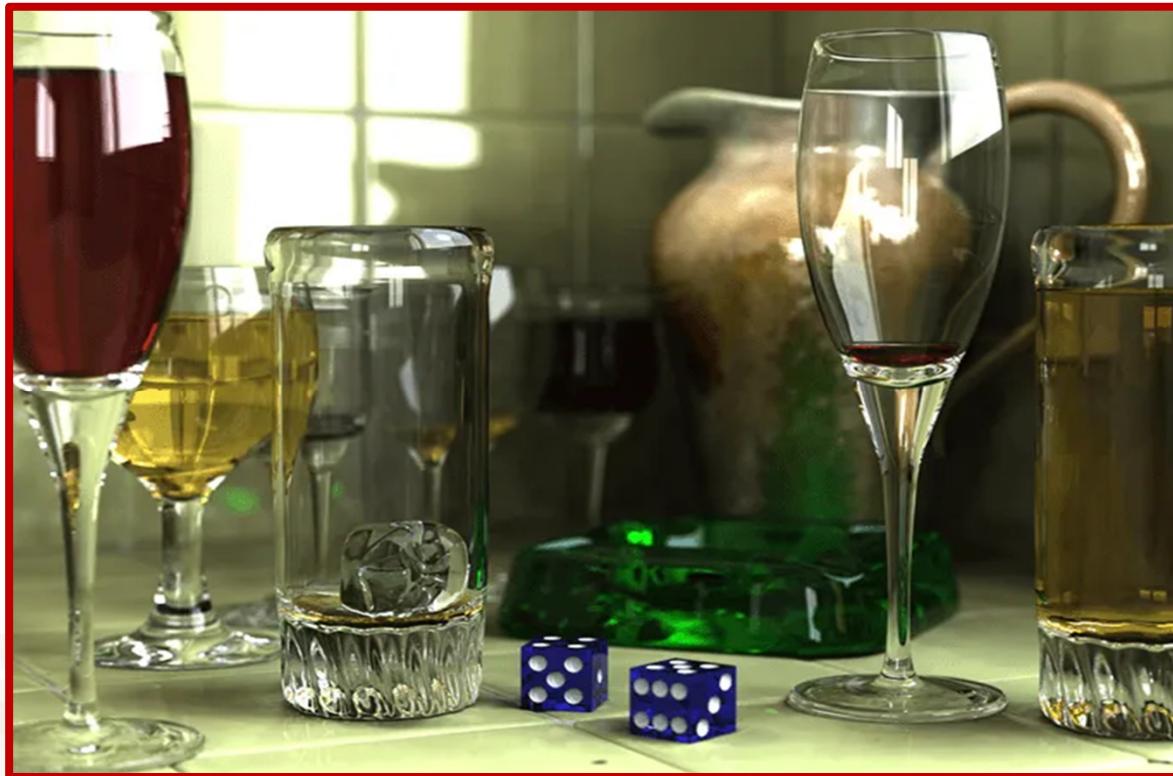
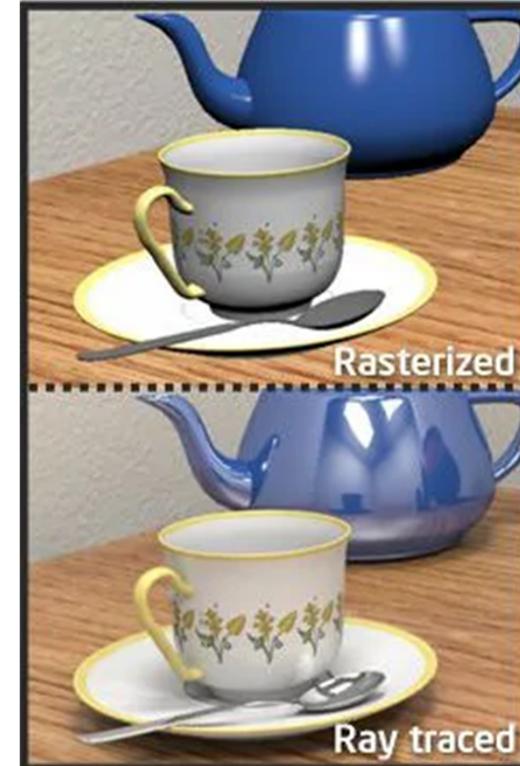


Image: <https://www.online-tech-tips.com/>



# Lighting and Shading

## Global Illumination

- Ray Tracing Example



video: [Microsoft DirectX Raytracing](#)

# Lighting and Shading

## Global Illumination

- Ray Tracing Example



video: [Beyond Turing](#)

# Lighting and Shading

## Global Illumination

- Radiosity Example



Image: [Battlefield3](#)

# Lighting and Shading

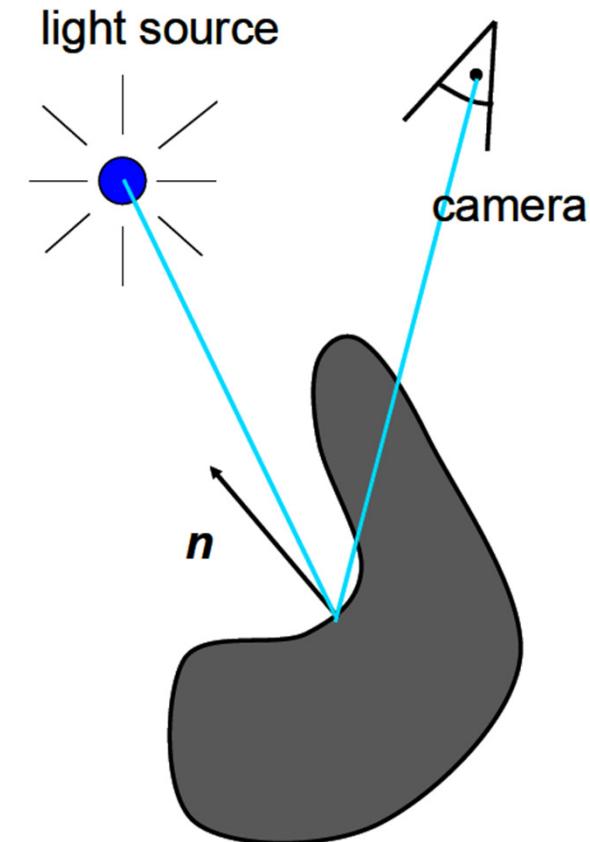
## Reflection Models

- Illumination models used for graphics initially were often **approximate models**
- Main goal was to model the interaction of light with matter in a way that appears **realistic** and is **fast**
- **Reflection Models** are the *simplest of illumination models*
- Reflection models assume
  - local illumination only, no global illumination
  - light is only reflected from the surface, *no transmission through the object*
  - there is no propagation media, surfaces are in vacuum

# Lighting and Shading

## Local Illumination

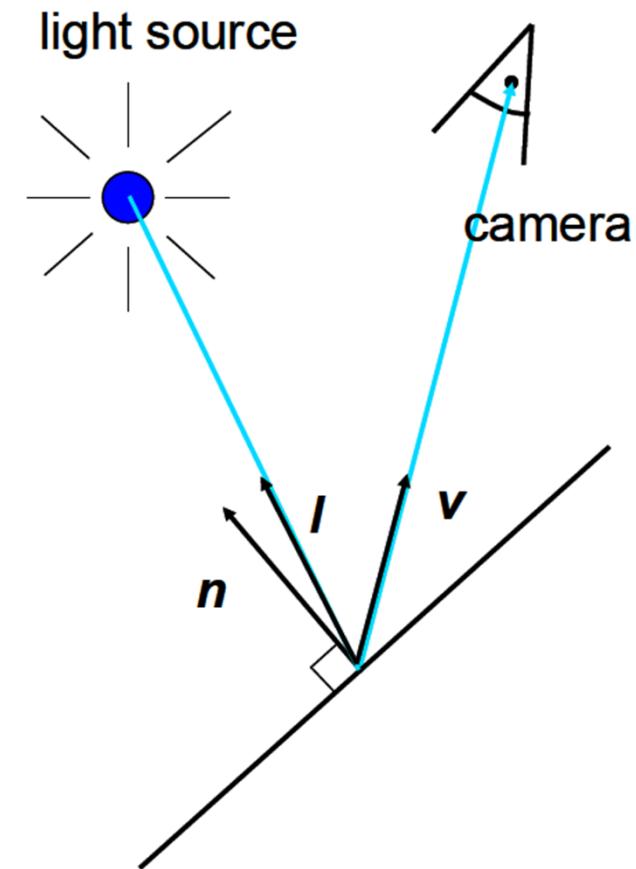
- **Approximate model**
- **Local interaction between light, surface, viewer**
- **Phong model (this lecture):**
  - fast, supported in OpenGL
- **GPU shaders**



# Lighting and Shading

## Local Illumination

- Color determined only based on surface normal, relative camera position, relative light position and reflection properties at the surface
- What effects does this ignore?



# Lighting and Shading

## Normal Vectors

- Must calculate and specify the normal vector
  - Even in OpenGL!
- Example: plane

# Lighting and Shading

## Method 1: Normal of a plane

- **Method 1:** given by  $f(p) = ax + by + cz + d = 0$
- Let  $p_0$  be a known point on the plane
- Let  $p$  be an arbitrary point on the plane
- **Recall:**  $u \cdot v = 0$  if and only if  $u$  is orthogonal to  $v$
- $n \cdot (p - p_0) = 0$
- Consequently  $n_0 = [a \ b \ c]^T$
- Normalize to  $n = n_0 / |n_0|$

# Lighting and Shading

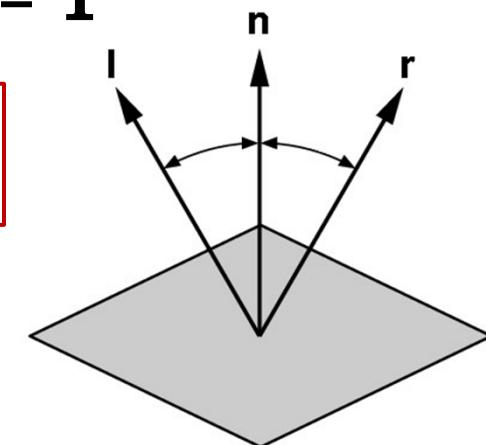
## Method 2: Normal of a plane

- **Method 2:** plane given by  $p_0, p_1, p_2$
- Points must not be collinear → cross product defined as 0
- **Recall:**  $u \times v$  orthogonal to  $u$  and  $v$
- $n_0 = (p_1 - p_0) \times (p_2 - p_0)$
- Order of cross product determines orientation
- Normalize to  $n = n_0 / |n_0|$

# Lighting and Shading

## Reflected Vector

- **Perfect Reflection:** angle of incidence equals angle of reflection
  - **Also:**  $l$ ,  $n$ , and  $r$  lie in the same plane  $\rightarrow$
  - Assume  $|l| = |n| = 1$ , guarantee  $|r| = 1$
  - $l \cdot n = \cos\theta = n \cdot r$
  - $r = al + bn$
- Can write  $r$  as a linear combination of  $l$  and  $n$
- (1)  $n \cdot r = al \cdot n + b = l \cdot n$
- (2)  $l = r \cdot r = a^2 + 2abl \cdot n + b^2$
- **Solution:**  $a = -l$  and  $b = 2(l - n) \rightarrow r = 2(l \cdot n)n - l$



# Lighting and Shading

## Light Sources and Material Properties

- Appearance depends on
  - Light sources, their locations and properties
  - Material (surface) properties
  - Viewer position

# Lighting and Shading

## Types of Light Sources

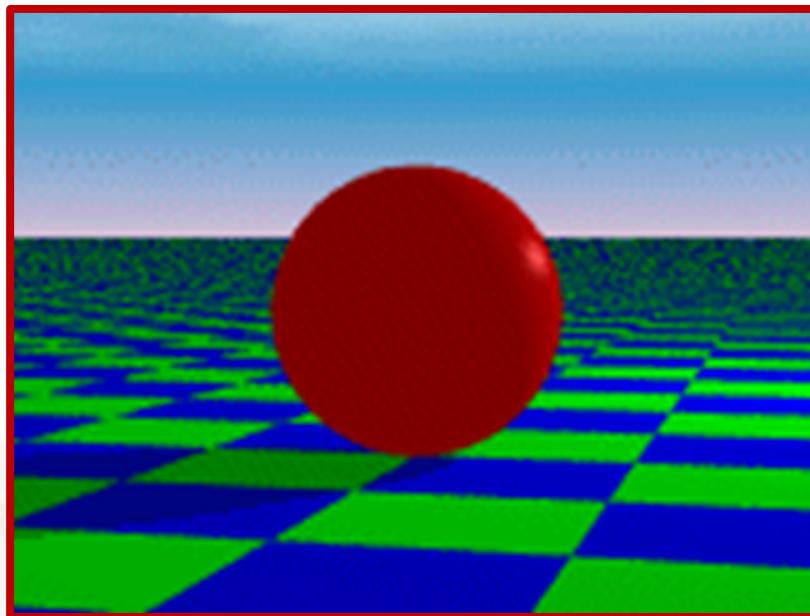
- **Ambient light:** no identifiable source or direction
- **Point source:** given only by point position
- **Directional light:** given only by direction
- **Spotlight:** point source + direction
  - Cut-off angle defines a cone of light
  - Attenuation function (brighter in center)



# Lighting and Shading

## Point Source

- Given by a point  $p_0$
- Light emitted equally in all directions
- Intensity decreases with square of distance



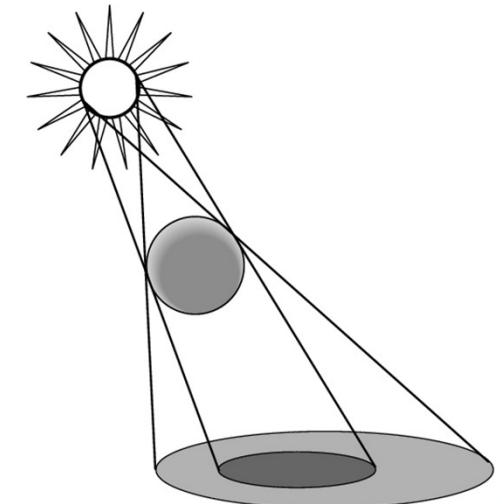
$$I \propto \frac{1}{|p - p_0|^2}$$

# Lighting and Shading

## Point Source (Limitations)

- Shading and shadows inaccurate
- Example: penumbra (partial “soft” shadow)
- Similar problems with highlights
- Compensate with attenuation
- Softens lighting
- Better with ray tracing
- Better with Radiosity

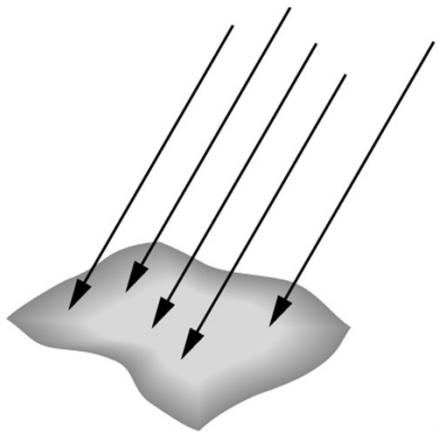
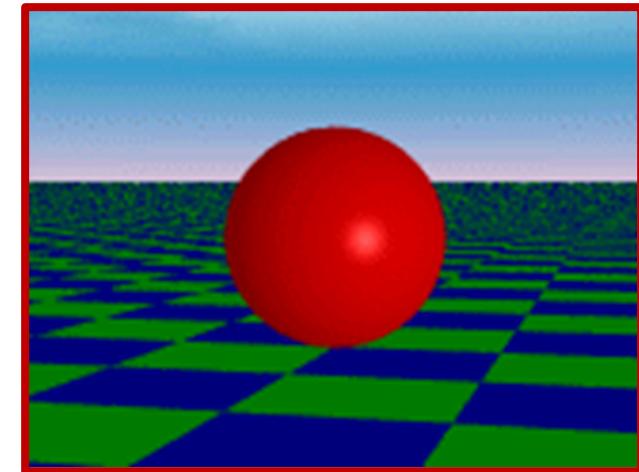
$$q = \text{distance } |p - p_0|$$
$$a, b, c \text{ constants}$$
$$\frac{1}{a + bq + cq^2}$$



# Lighting and Shading

## Directional Light Source

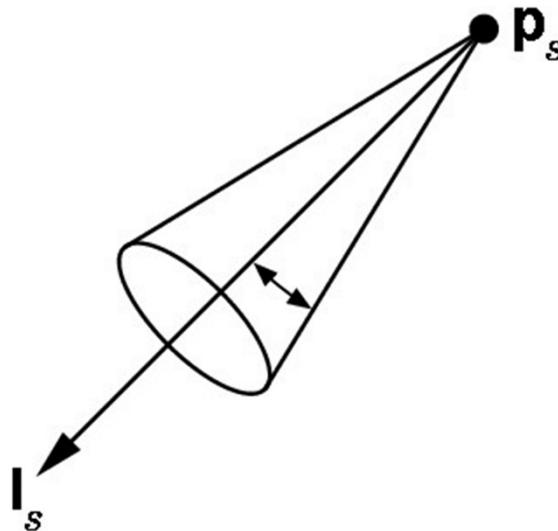
- Given by a **direction vector**
- Simplifies some calculations
- In OpenGL:
- Point source  $[x \ y \ z \ 1]^T$
- Directional source  $[x \ y \ z \ 0]^T$



# Lighting and Shading

## Spotlight

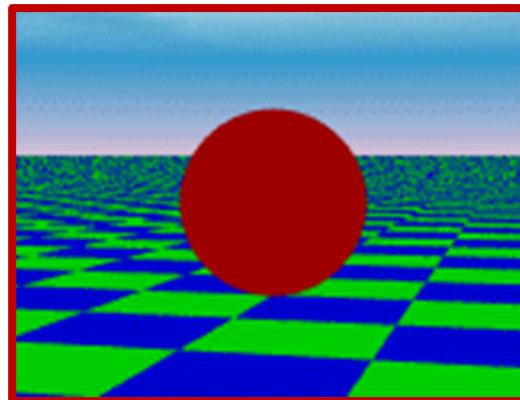
- **Most complex light source in OpenGL**
- Light still emanates from point
- Cut-off by cone determined by angle  $\theta$



# Lighting and Shading

## Global Ambient Light

- Independent of light source
- Lights entire scene
- Computationally inexpensive
- Not very interesting on its own
  - A cheap hack to make the scene brighter



# Lighting and Shading

## Phong Illumination Model

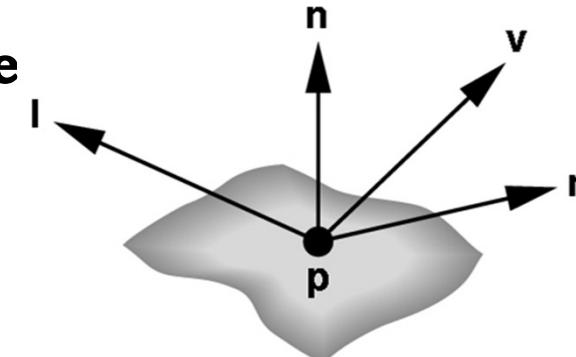
- Calculate color for arbitrary point on surface
- Compromise between realism and efficiency
- Local computation (*no inter-object visibility calculations*)
- Basic inputs are material properties and  $l$ ,  $n$ ,  $v$ :

$l$  = unit vector to light source

$n$  = surface normal

$v$  = unit vector to viewer

$r$  = reflection of  $l$  at  $p$   
(determined by  $l$  and  $n$ )



# Lighting and Shading

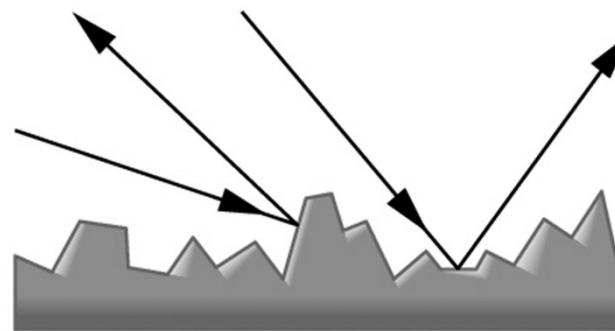
## Phong Illumination Overview

- 1) Add contributions from each light source
- 2) *Clamp the final result to [0, 1]*
- 3) Calculate each color channel (R,G,B) separately
- 4) Light source contributions decomposed into
  - **Diffuse** reflection
  - **Ambient** reflection
  - **Specular** reflection
  - (Based on ambient, diffuse, and specular lighting and material properties)

# Lighting and Shading

## Diffuse Reflection

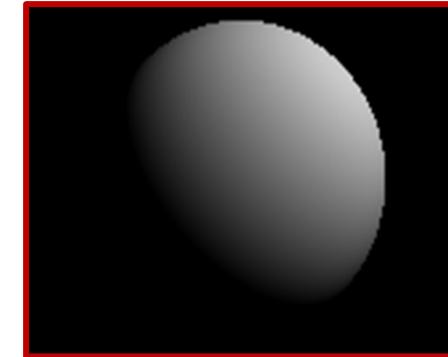
- Diffuse reflector scatters light equally to all directions
- Called Lambertian surface
- Diffuse reflection coefficient  $k_d$ ,  $0 \leq k_d \leq 1$
- Only the angle of incoming light is important
- Models non-shiny dull matt surfaces like chalk



# Lighting and Shading

## Diffuse Light Intensity Depends on Angle of Incoming Light

- Recall
- $l$  = unit vector to light;  $n$  = unit surface normal;  
 $\theta$  = angle to normal
- $\cos \theta = l \cdot n$
- $I_d = k_d L_d (l \cdot n)$
- With attenuation:  $I_d = \frac{k_d L_d}{a + bq + cq^2} (l \cdot n)$
- $q$  = distance to light source,
- $L_d$  = diffuse component of light



With the diffuse reflection the part of the surface not illuminated by the light is dark.  
Solution (hack): **Ambient lighting**

# Lighting and Shading

## Ambient Reflection

- $I_a = k_a L_a$
- Intensity of ambient light is *uniform at every point*
- Ambient reflection coefficient  $k_a$ ,  $0 \leq k_a \leq 1$
- May be different for every surface and  $r, g, b$
- Determines *reflected fraction of ambient light*
- $L_a$  = ambient component of light source (can be set to different value for each light source)
- **Note:** Defining  $L_a, k_a$  in this fashion is not physically meaningful

# Lighting and Shading

## Effect of ambient and diffuse reflections

- By adding the ambient term to the diffuse reflection the illumination equation is



$$k_a = 0.2, \quad k_d = 0.4$$



$$k_a = 0.2, \quad k_d = 0.6$$

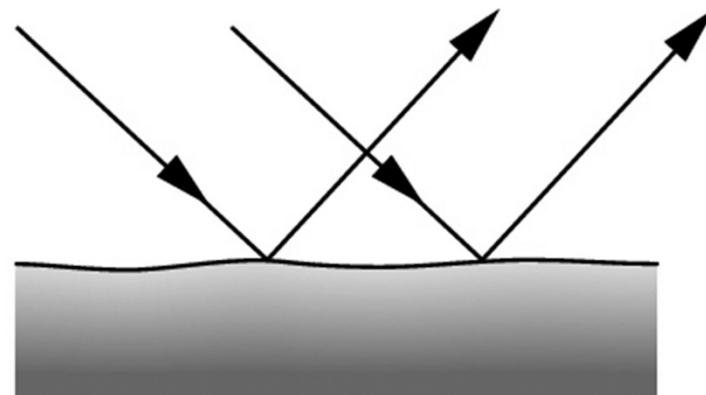


$$k_a = 0.2, \quad k_d = 0.8$$

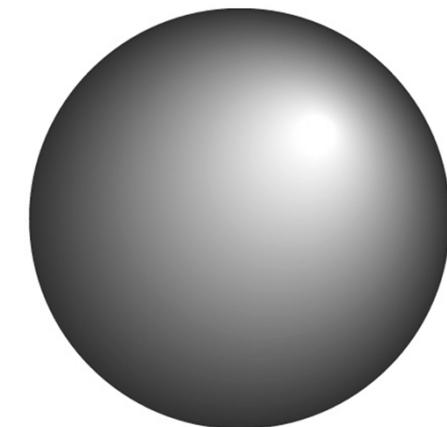
# Lighting and Shading

## Specular Reflection

- Specular reflection coefficient  $k_s$ ,  $0 \leq k_s \leq 1$
- Shiny surfaces have high specular coefficient
- Used to model specular highlights



Specular Reflection



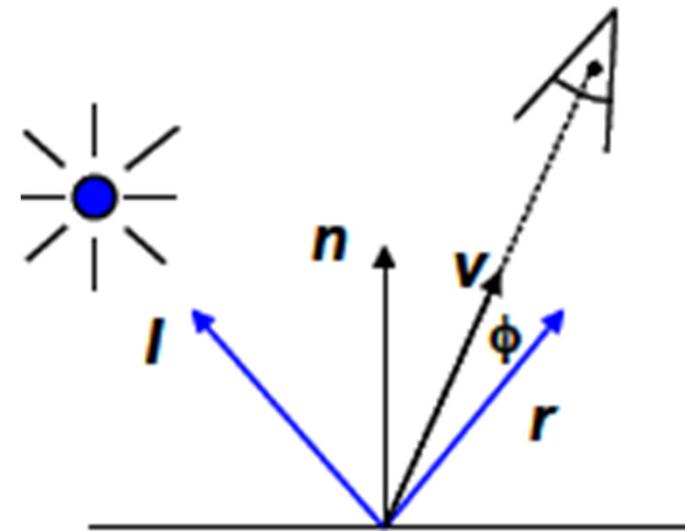
Specular Highlights

# Lighting and Shading

## Specular Reflection

- Recall

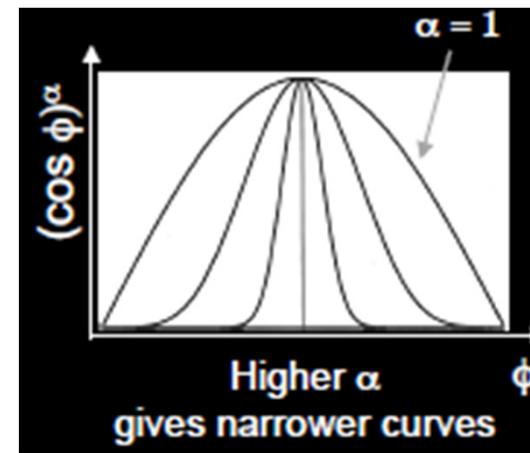
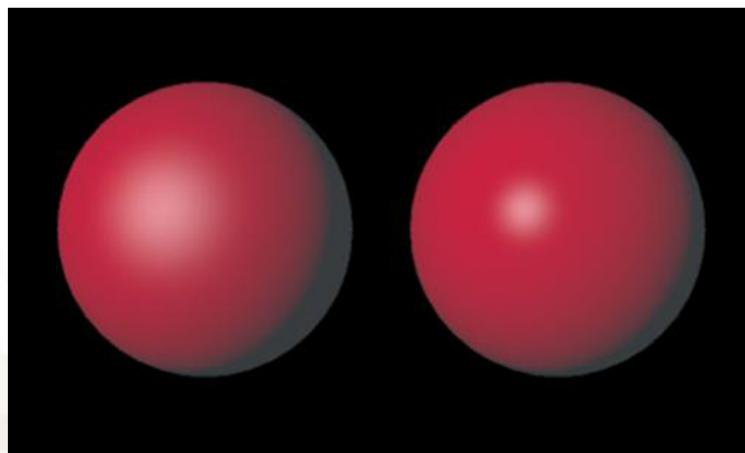
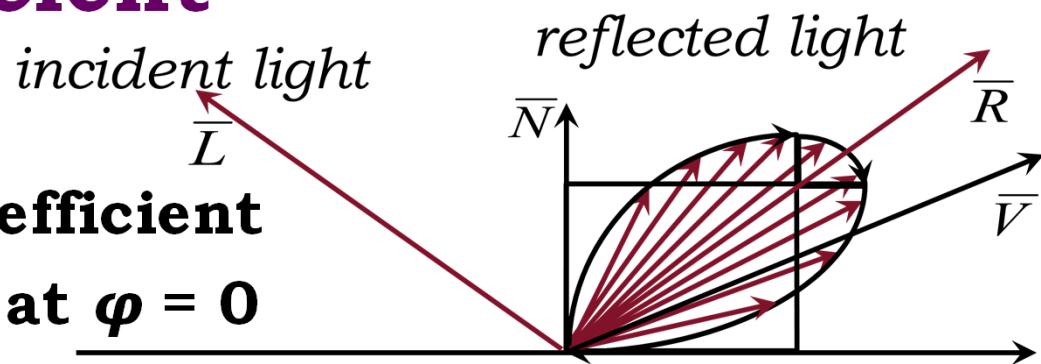
- $v$  = unit vector to camera
- $r$  = unit reflected vector
- $\varphi$  = angle between  $v$  and  $r$
- $\cos \varphi = v \cdot r$
- $I_s = k_s L_s (\cos \varphi)^a$
- $L_s$  is **specular component** of light
- $a$  is *shininess coefficient*
- Can add distance term as well



# Lighting and Shading

## Shininess Coefficient

- $I_s = k_s L_s (\cos \varphi)^\alpha$
- **$\alpha$  is the shininess coefficient**
- maximum reflection at  $\varphi = 0$
- $\alpha$  controls the drop off away from  $R$



Low  $\alpha$

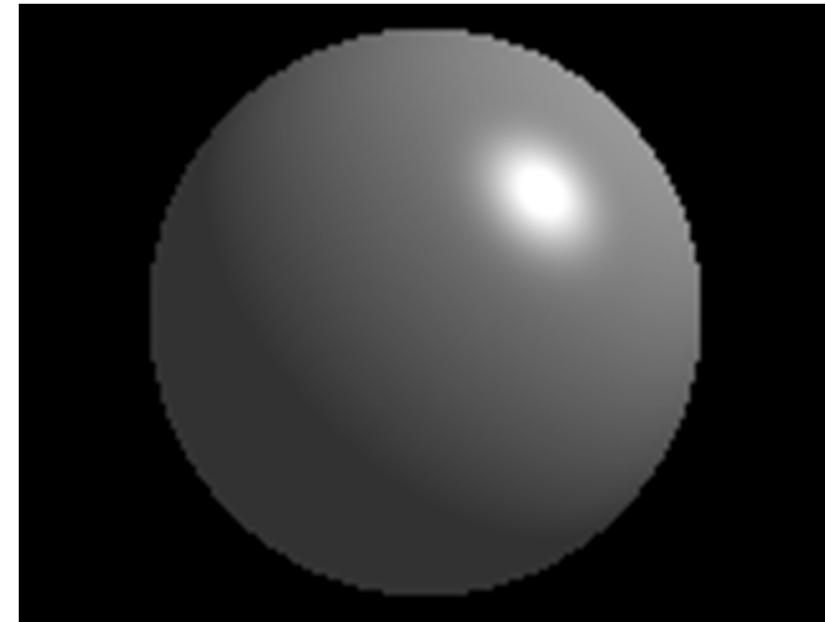
High  $\alpha$

# Lighting and Shading

Effect of adding specular reflection



Diffuse Illumination Model



Phong Illumination Model

# Lighting and Shading

## Summary of Phong Model

- Light components for each color:
  - Ambient ( $L_a$ ), diffuse ( $L_d$ ), specular ( $L_s$ )
- Material coefficients for each color:
  - Ambient ( $k_a$ ), diffuse ( $k_d$ ), specular ( $k_s$ )
- Distance  $q$  for surface point from light source
  - $\mathbf{l}$  = unit vector to light
  - $\mathbf{n}$  = surface normal
  - $\mathbf{r} = \mathbf{l}$  reflected about  $\mathbf{n}$
  - $\mathbf{v}$  = vector to viewer

$$I = \frac{1}{a + bq + cq^2} (k_d L_d (\mathbf{l} \cdot \mathbf{n}) + k_s L_s (\mathbf{r} \cdot \mathbf{v})^\alpha) + k_a L_a$$

# Lighting and Shading

## Material Coefficients - Examples

| Material        | GL_AMBIENT | GL_DIFFUSE | GL_SPECULAR | GL_SHININESS |
|-----------------|------------|------------|-------------|--------------|
| Brass           | 0.329412   | 0.780392   | 0.992157    | 27.8974      |
|                 | 0.223529   | 0.568627   | 0.941176    |              |
|                 | 0.027451   | 0.113725   | 0.807843    |              |
|                 | 1.0        | 1.0        | 1.0         |              |
| Bronze          | 0.2125     | 0.714      | 0.393548    | 25.6         |
|                 | 0.1275     | 0.4284     | 0.271906    |              |
|                 | 0.054      | 0.18144    | 0.166721    |              |
|                 | 1.0        | 1.0        | 1.0         |              |
| Polished Bronze | 0.25       | 0.4        | 0.774597    | 76.8         |
|                 | 0.148      | 0.2368     | 0.458561    |              |
|                 | 0.06475    | 0.1036     | 0.200621    |              |
|                 | 1.0        | 1.0        | 1.0         |              |
| Chrome          | 0.25       | 0.4        | 0.774597    | 76.8         |
|                 | 0.25       | 0.4        | 0.774597    |              |
|                 | 0.25       | 0.4        | 0.774597    |              |
|                 | 1.0        | 1.0        | 1.0         |              |
| Copper          | 0.19125    | 0.7038     | 0.256777    | 12.8         |
|                 | 0.0735     | 0.27048    | 0.137622    |              |
|                 | 0.0225     | 0.0828     | 0.086014    |              |
|                 | 1.0        | 1.0        | 1.0         |              |

©: Advanced Graphics Programming Using OpenGL by  
Tom McReynolds and David Blythe

# Lighting and Shading

## Material Coefficients - Examples

|                 |          |          |           |         |
|-----------------|----------|----------|-----------|---------|
| Polished Copper | 0.2295   | 0.5508   | 0.580594  | 51.2    |
|                 | 0.08825  | 0.2118   | 0.223257  |         |
|                 | 0.0275   | 0.066    | 0.0695701 |         |
|                 | 1.0      | 1.0      | 1.0       |         |
|                 |          |          |           |         |
| Gold            | 0.24725  | 0.75164  | 0.628281  | 51.2    |
|                 | 0.1995   | 0.60648  | 0.555802  |         |
|                 | 0.0745   | 0.22648  | 0.366065  |         |
|                 | 1.0      | 1.0      | 1.0       |         |
|                 |          |          |           |         |
| Polished Gold   | 0.24725  | 0.34615  | 0.797357  | 83.2    |
|                 | 0.2245   | 0.3143   | 0.723991  |         |
|                 | 0.0645   | 0.0903   | 0.208006  |         |
|                 | 1.0      | 1.0      | 1.0       |         |
|                 |          |          |           |         |
| Pewter          | 0.105882 | 0.427451 | 0.333333  | 9.84615 |
|                 | 0.058824 | 0.470588 | 0.333333  |         |
|                 | 0.113725 | 0.541176 | 0.521569  |         |
|                 | 1.0      | 1.0      | 1.0       |         |
|                 |          |          |           |         |

# Lighting and Shading

## Material Coefficients - Examples

|                 |                                      |                                       |                                          |      |
|-----------------|--------------------------------------|---------------------------------------|------------------------------------------|------|
| Silver          | 0.19225<br>0.19225<br>0.19225<br>1.0 | 0.50754<br>0.50754<br>0.50754<br>1.0  | 0.508273<br>0.508273<br>0.508273<br>1.0  | 51.2 |
| Polished Silver | 0.23125<br>0.23125<br>0.23125<br>1.0 | 0.2775<br>0.2775<br>0.2775<br>1.0     | 0.773911<br>0.773911<br>0.773911<br>1.0  | 89.6 |
| Emerald         | 0.0215<br>0.1745<br>0.0215<br>0.55   | 0.07568<br>0.61424<br>0.07568<br>0.55 | 0.633<br>0.727811<br>0.633<br>0.55       | 76.8 |
| Jade            | 0.135<br>0.2225<br>0.1575<br>0.95    | 0.54<br>0.89<br>0.63<br>0.95          | 0.316228<br>0.316228<br>0.316228<br>0.95 | 12.8 |
| Obsidian        | 0.05375<br>0.05<br>0.06625<br>0.82   | 0.18275<br>0.17<br>0.22525<br>0.82    | 0.332741<br>0.328634<br>0.346435<br>0.82 | 38.4 |

# Lighting and Shading

## Material Coefficients - Examples

|               |                                      |                                       |                                           |        |
|---------------|--------------------------------------|---------------------------------------|-------------------------------------------|--------|
| Pearl         | 0.25<br>0.20725<br>0.20725<br>0.922  | 1.0<br>0.829<br>0.829<br>0.922        | 0.296648<br>0.296648<br>0.296648<br>0.922 | 11.264 |
| Ruby          | 0.1745<br>0.01175<br>0.01175<br>0.55 | 0.61424<br>0.04136<br>0.04136<br>0.55 | 0.727811<br>0.626959<br>0.626959<br>0.55  | 76.8   |
| Turquoise     | 0.1<br>0.18725<br>0.1745<br>0.8      | 0.396<br>0.74151<br>0.69102<br>0.8    | 0.297254<br>0.30829<br>0.306678<br>0.8    | 12.8   |
| Black Plastic | 0.0<br>0.0<br>0.0<br>1.0             | 0.01<br>0.01<br>0.01<br>1.0           | 0.50<br>0.50<br>0.50<br>1.0               | 32     |
| Black Rubber  | 0.02<br>0.02<br>0.02<br>1.0          | 0.01<br>0.01<br>0.01<br>1.0           | 0.4<br>0.4<br>0.4<br>1.0                  | 10     |

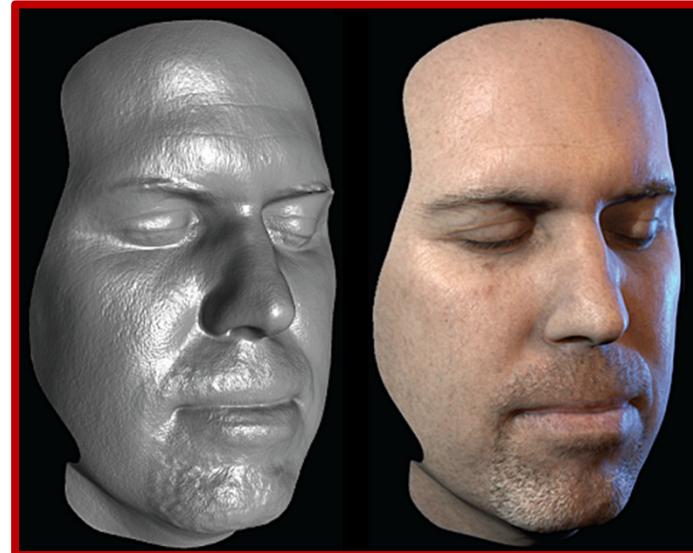
©: Advanced Graphics Programming Using OpenGL by  
Tom McReynolds and David Blythe



# Lighting and Shading

## BRDF

- Bidirectional Reflection Distribution Function
- Must measure for real materials
- Isotropic Vs. Anisotropic
- Mathematically complex
- Programmable pixel shading



Lighting properties of a human face were captured and face re-rendered;  
Institute for Creative Technologies

# Lighting and Shading

## Shading

- **Shading** is the process of *determining the colors of all the pixels covered by a surface using an illumination model*
- Simplest method is to
  - determine surface visible at each pixel
  - compute normal of the surface
  - evaluate light intensity and color using an illumination model
- This is **quite expensive**. The shading methods could be made efficient by customizing for specific surface representation

# Lighting and Shading

## Shading Models

- Shading Models give *a technique to determine the colors of all the pixels covered by a surface using appropriate illumination model*
- Triangle meshes are commonly used for representing complex surfaces
- The geometric information is available *only at the vertices of a triangle*
- Interpolative shading models could be used to *increase the efficiency* substantially

# Lighting and Shading

## Constant (Flat) Shading

- It is the simplest of the shading models
- Also called as **faceted shading** or **flat shading**
- *One polygon receives only one intensity value*
- Illumination model is applied *only once for each polygon*
- Makes the following assumptions
  - light source is at infinity, so  $\bar{N} \bullet \bar{L}$  is constant across a polygon face
  - viewer is at infinity, so  $\bar{R} \bullet \bar{V}$  is constant across the polygon face
  - polygon represents the actual surface being modeled



# Lighting and Shading

## Gouraud Shading Model

- “**Fixed Pipeline**” OpenGL implements Gouraud Shading
- Phong Lighting is computed *per-vertex*
- The result is then *interpolated on the whole Triangle*
- Specular highlights are typically blurry, a *high tessellation is required for good lighting*
- It can be completely implemented in a **Vertex Shader**

# Lighting and Shading

## Gouraud Shading Model

- $I_1$ ,  $I_2$  and  $I_3$  are computed using Phong Illumination.
- Intensity  $I_p$  at a point is calculated as:

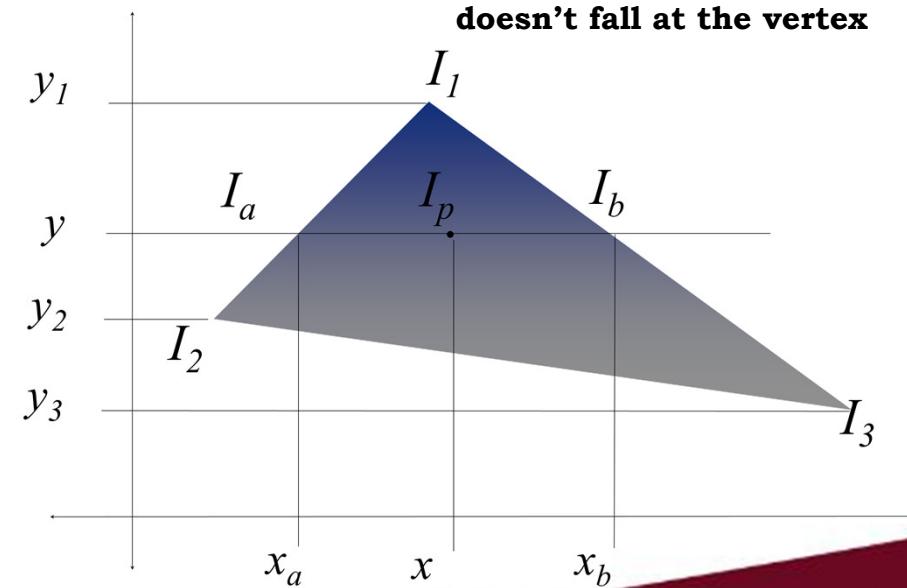
$$I_a = I_1 + (I_2 - I_1) \frac{y - y_1}{y_2 - y_1}$$

$$I_b = I_1 + (I_3 - I_1) \frac{y - y_1}{y_3 - y_1}$$

$$I_p = I_a + (I_b - I_a) \frac{x - x_a}{x_b - x_a}$$

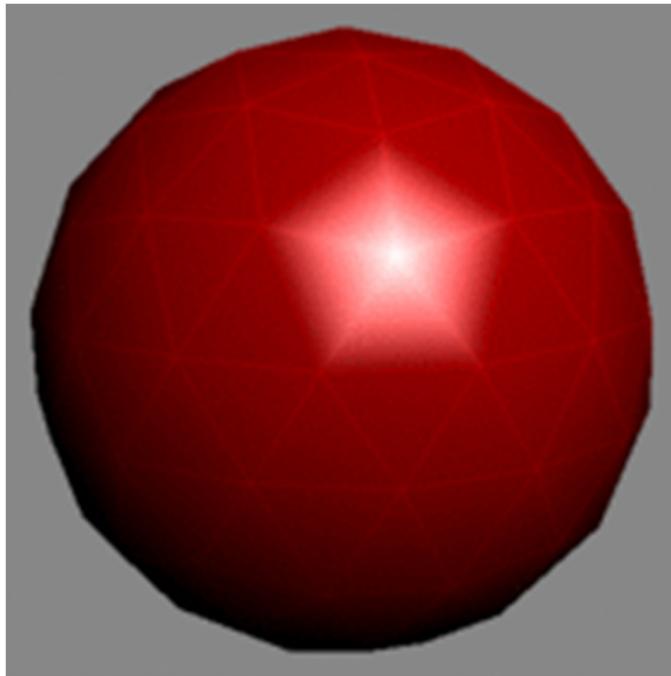


Might miss specular highlights, if the highlight doesn't fall at the vertex

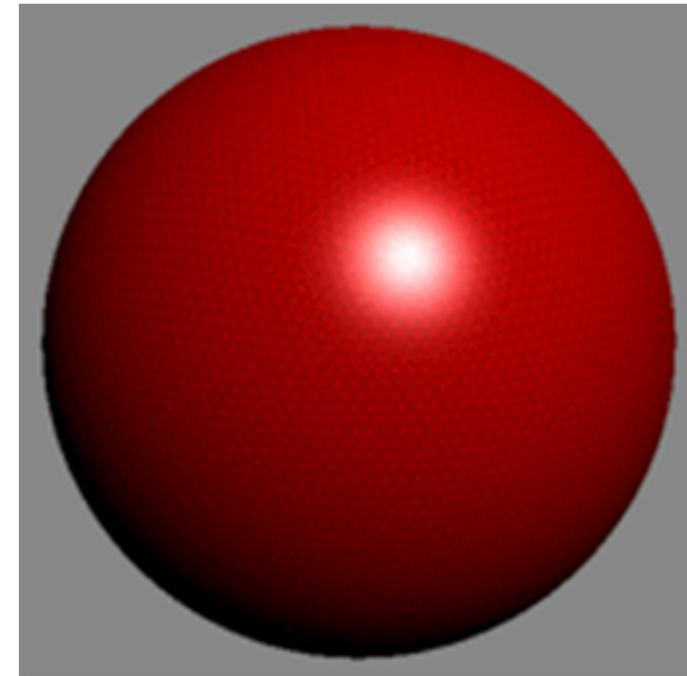


# Lighting and Shading

## Gouraud Shading Model



**Low Poly Count**



**High Poly Count**

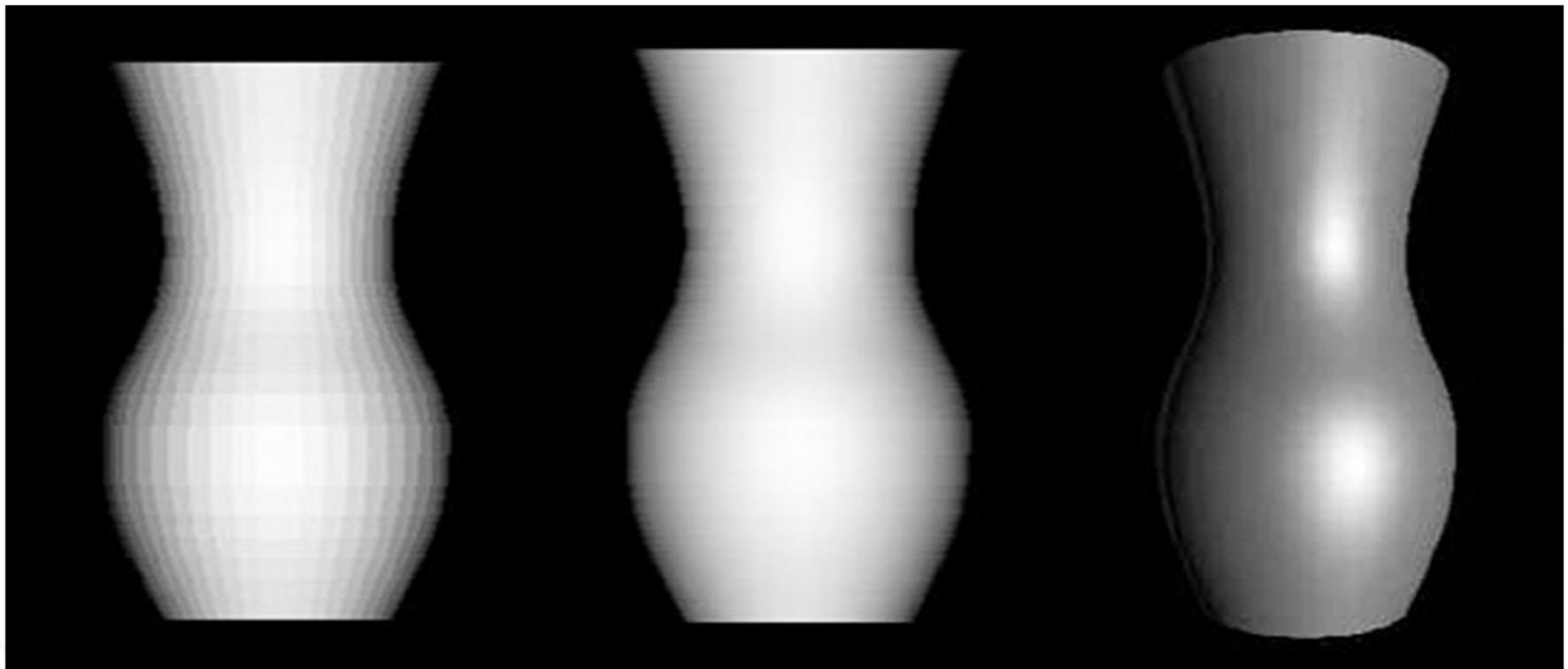
# Lighting and Shading

## Phong Shading

- Is typically implemented per Fragment (pixel)
- It provides “sharp” *specular highlights even with low polygon count* since lighting is calculated for each pixel individually
- It is *implemented on both the Vertex as well as Fragment Shader*
- Vertex Shader Transforms the Lighting Data to the View Space
- The Fragment Shader calculates the *Phong Shading Equation (Ambient + Diffuse + Specular)*

# Lighting and Shading

## Side by Side Comparison



**Constant Shading**

**Gouraud Shading**

**Phong Shading**

# Lighting and Shading

## Phong Model Implementation with GLSL

- Light Properties and Material Properties are implemented as *Shader Uniform Variables*
- Be careful from which Coordinate System the Light Positions are sent to the Shaders
- Typically, lighting is **computed in View Space**
  - Normals must be converted
  - Eye, Light vectors must be converted

# Lighting and Shading

## Phong Shading in GLSL

### Vertex Shader

- Transform vertex positions to Clip Space
- Transform lighting data to View Space

### Fragment Shader

- Calculate and add ambient, diffuse and specular components

# Lighting and Shading

## Phong Shading in GLSL

### Vertex Shader

- **Uniforms**
  - Light position
  - Transformation Matrices
- **Input**
  - Vertex Data (pos, normal, color)
- **Output**
  - Normal (N)
  - Light Vector (L)
  - Eye Vector (V)
  - Vertex Color

### Fragment Shader

- **Uniforms**
  - Material Coefficients ( $k_a$ ,  $k_d$ ,  $k_s$ ,  $n$ )
  - Light Coefficients (Color, Attenuation constant)
- **Inputs in View Space**
  - Normal (N)
  - Light Vector (L)
  - Eye Vector (V)
- **Output**
  - Fragment Color

# Lighting and Shading

## Phong Shading in GLSL Multiple Light Sources

- Use arrays for light parameters in your Shaders to support multiple light sources



**COMP 371**

# **Computer Graphics**

**Session 13**  
**RAY TRACING**



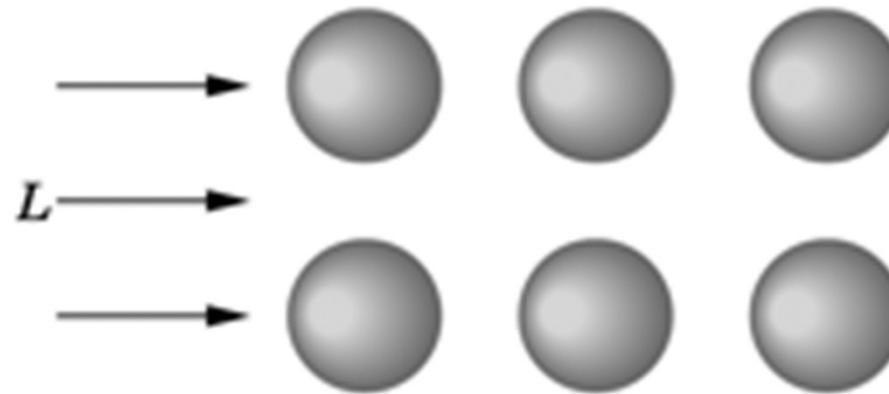
# Lecture Overview

- Review of last class
- Ray Tracing

# Ray Tracing

## Local Illumination

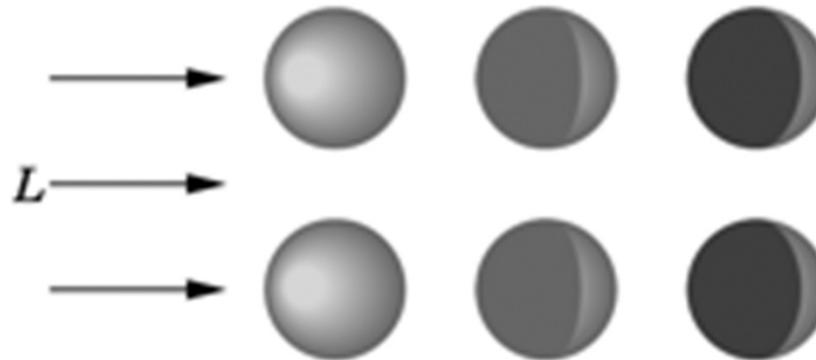
- Object illuminations are **independent**
- No light scattering between objects
- No real shadows, reflection, transmission
- OpenGL pipeline uses this



# Ray Tracing

## Global Illumination

- Ray tracing (highlights, reflection, transmission)
- Radiosity (surface inter-reflections)
- Photon mapping
- Pre-computed Radiance Transfer (PRT)



# Ray Tracing

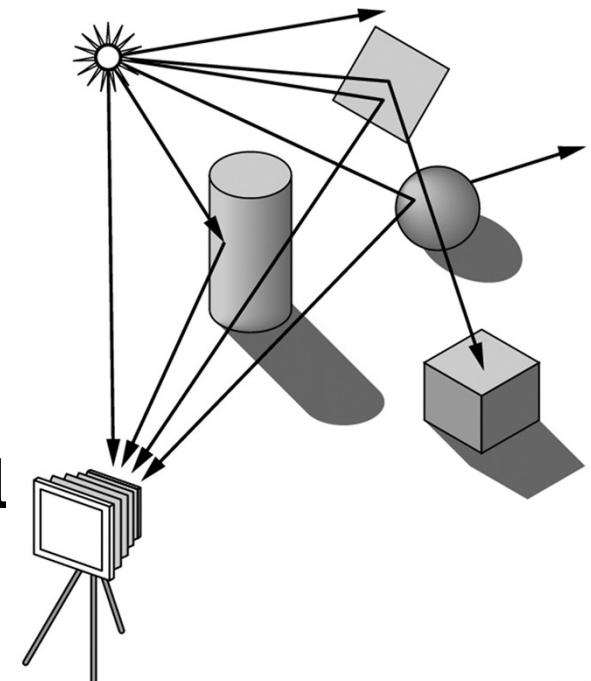
## Image/Object Space:

- Object Space
  - Graphics pipeline: **for each object**, render
    - Efficient pipeline architecture, real-time
    - Difficulty: object interactions (shadows, reflections, etc.)
- Image Space
  - Ray tracing: **for each pixel**, determine color
    - Pixel-level parallelism
    - Difficulty: very intensive computation, usually off-line

# Ray Tracing

## One idea: Forward Ray Tracing

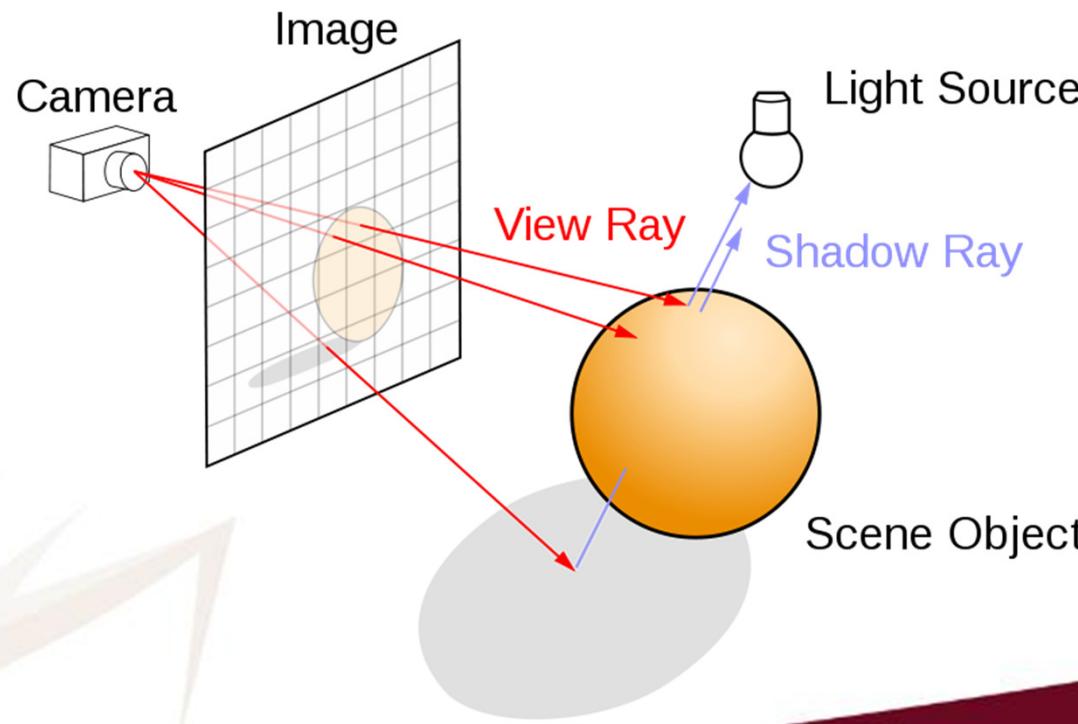
- Shoot (many) light rays from each light source
- Rays bounce off the objects
- Simulates paths of photons
- Problem: many rays will miss camera and not contribute to image
- This algorithm is not practical



# Ray Tracing

## Backward Ray Tracing

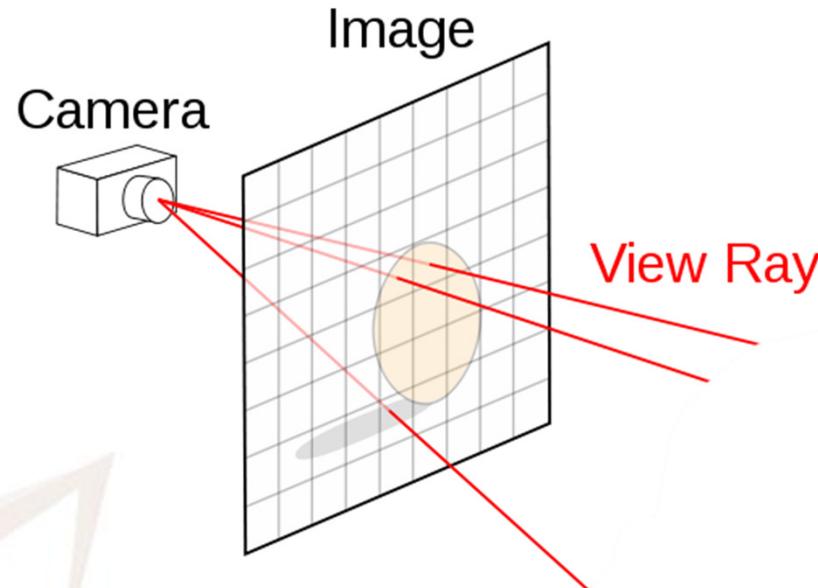
- Shoot one ray from camera through each pixel in image plane



# Ray Tracing

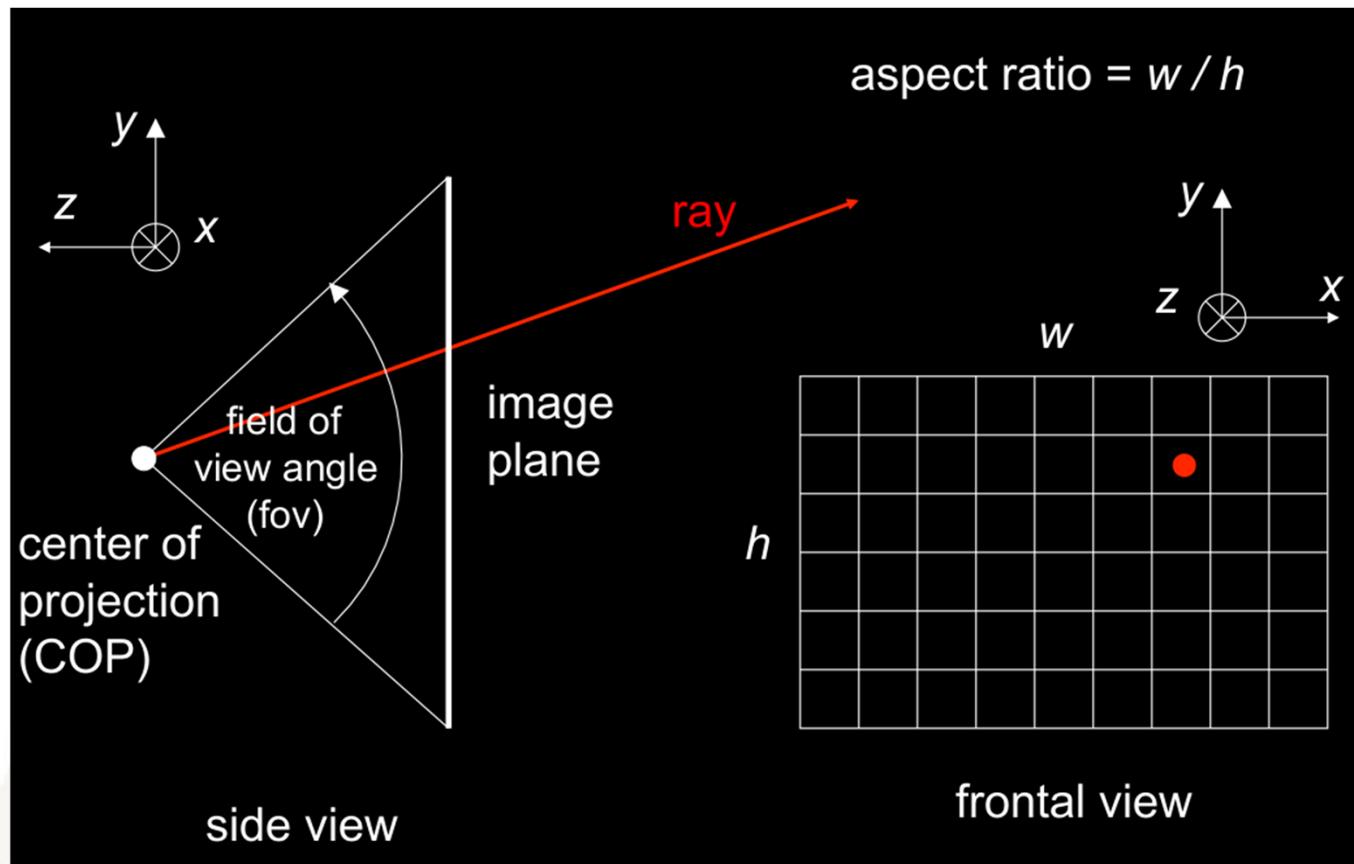
## Generating Rays

- Camera is at  $(0,0,0)$  and points in the negative z-direction
- Must determine coordinates of image corners in 3D



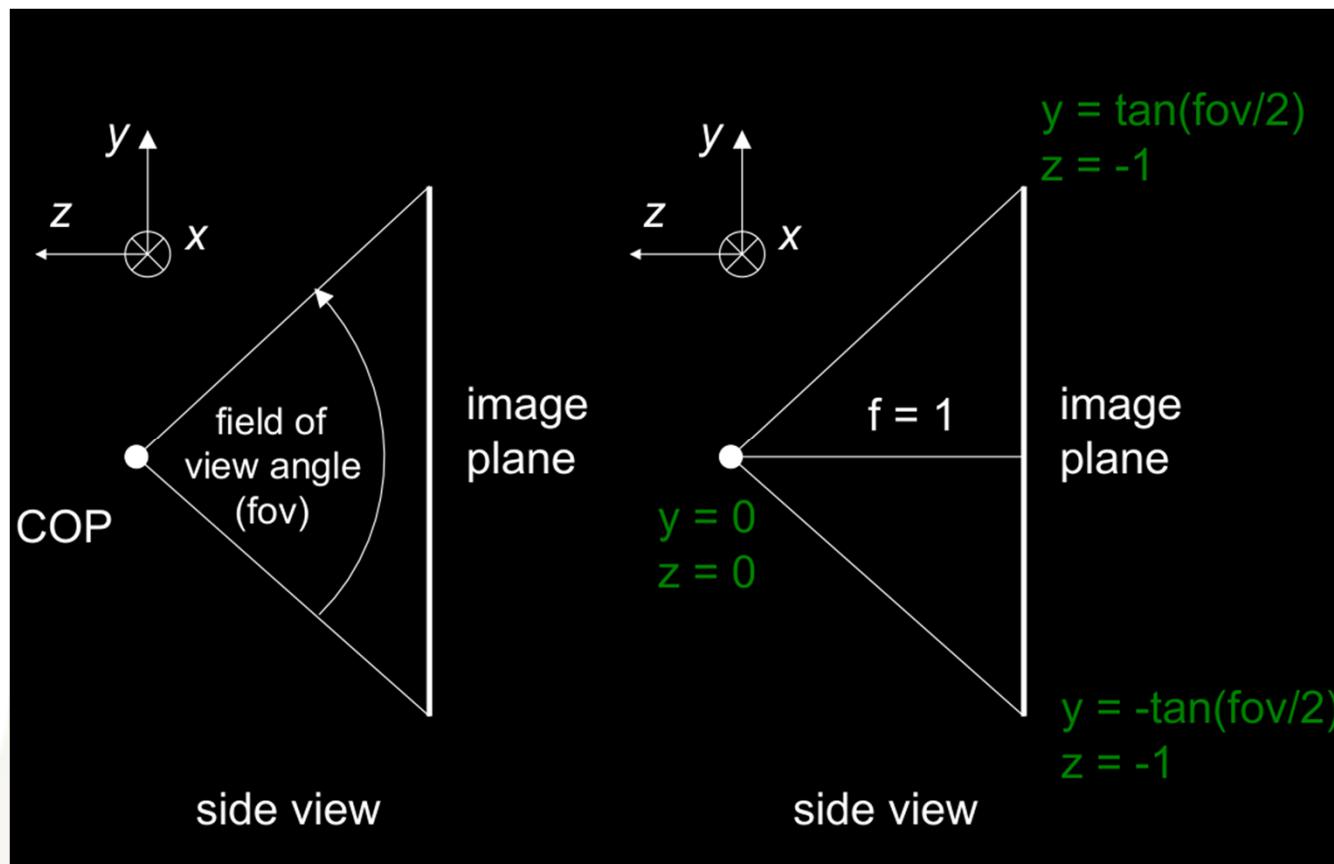
# Ray Tracing

## Generating Rays



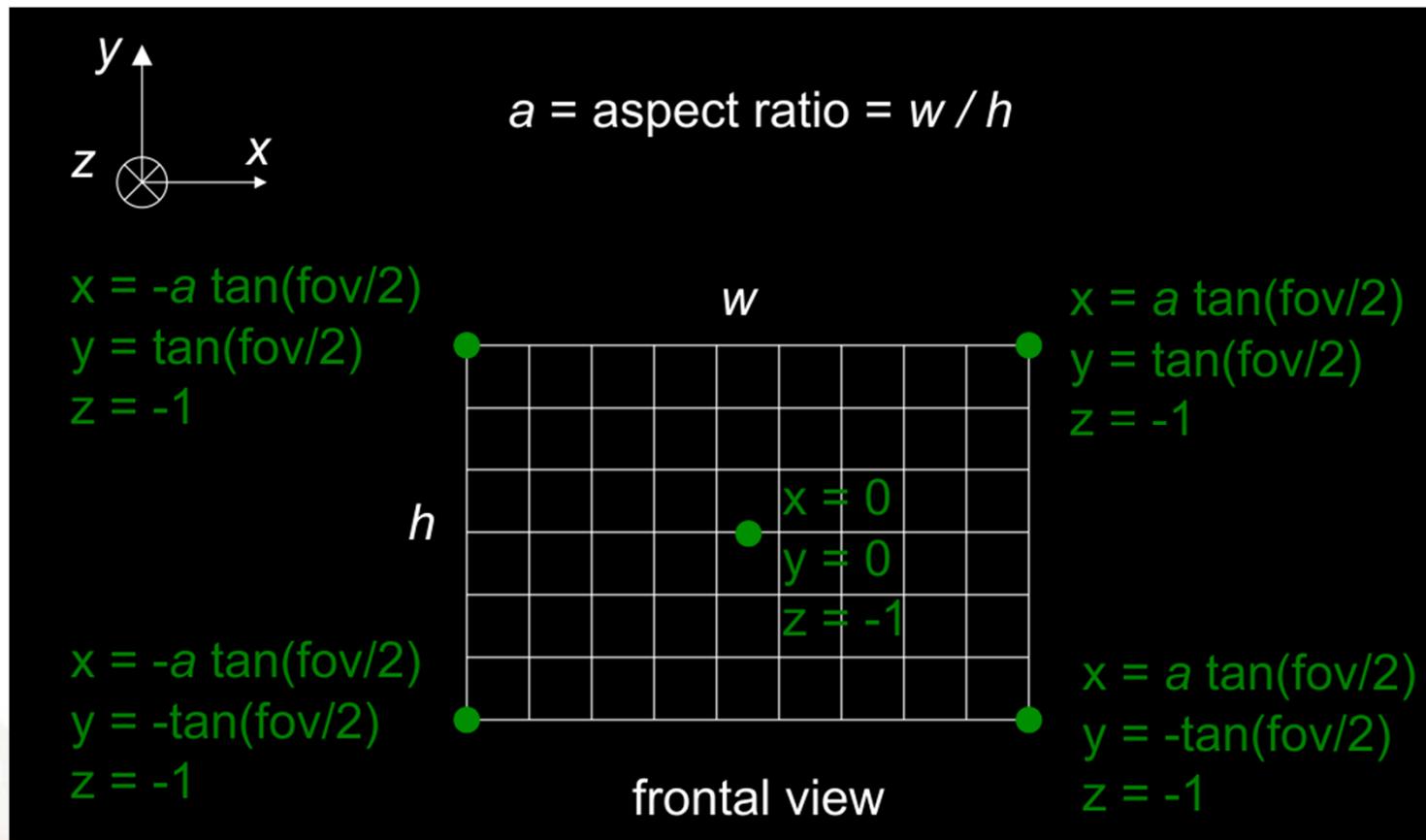
# Ray Tracing

## Generating Rays



# Ray Tracing

## Generating Rays

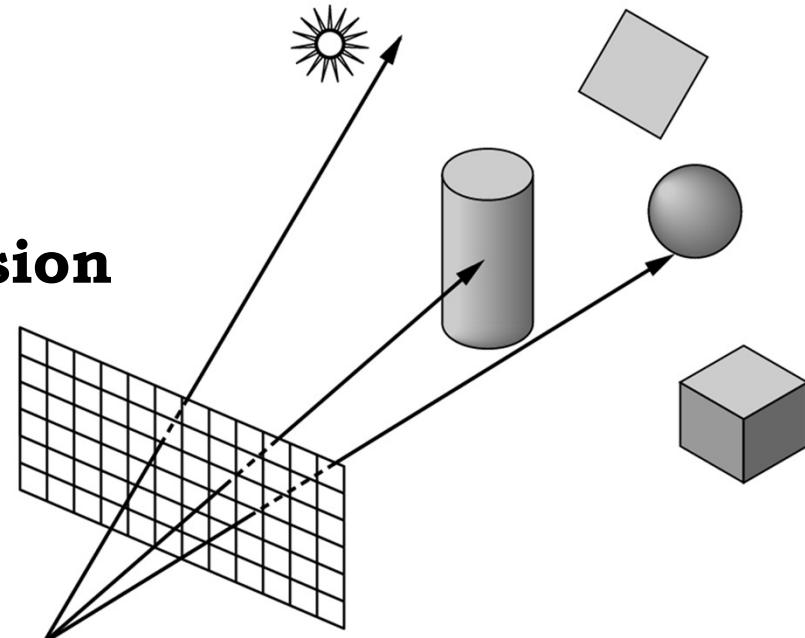


# Ray Tracing

## Determining Pixel Color

- Phong model (local as before)
- Shadow (Light) rays
- Specular reflection
- Specular transmission

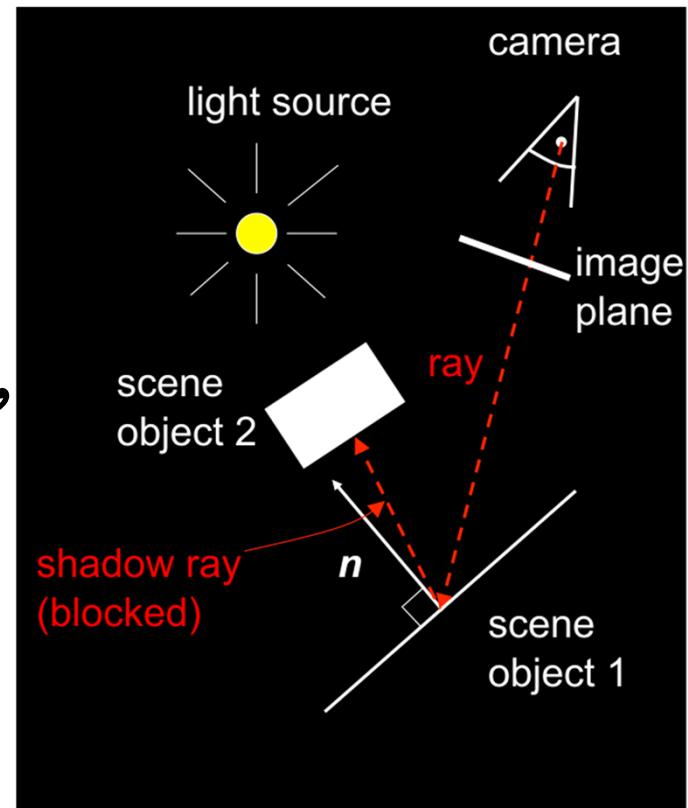
**Steps (3) and (4) require recursion**



# Ray Tracing

## Shadow (Light) Rays

- Determine if light “really” hits surface point
- Cast shadow ray from surface point to each light
- If shadow ray hits opaque object, no contribution from that light
- This is essentially improved diffuse reflection

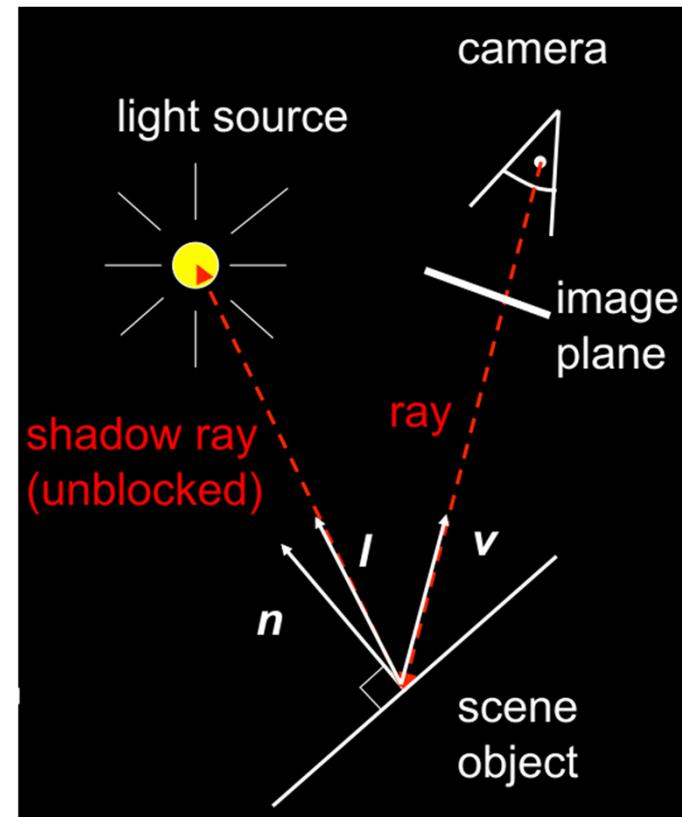


# Ray Tracing

## Phong Model

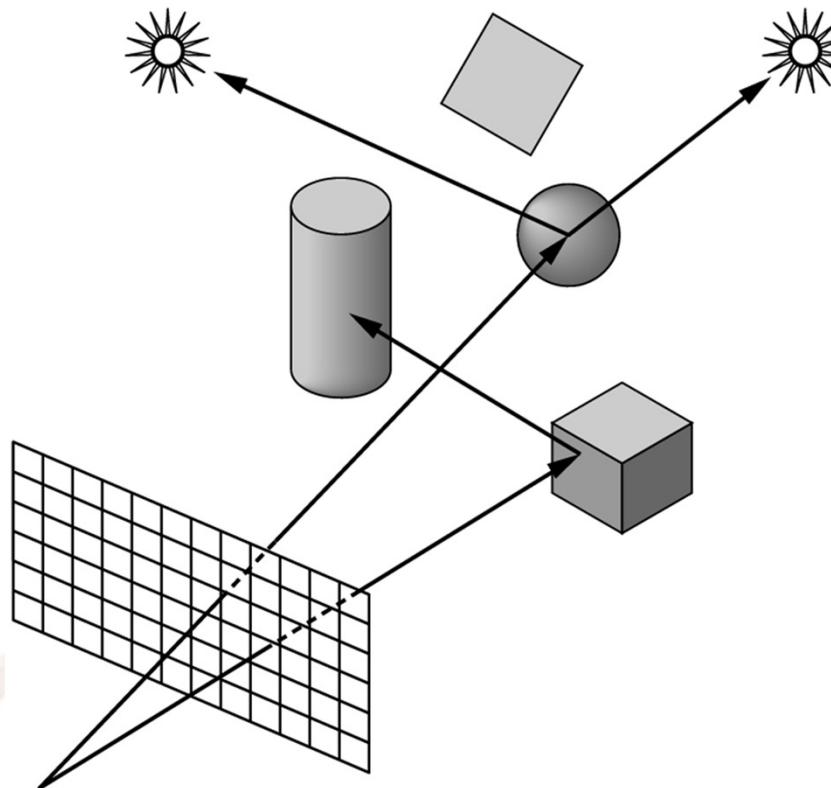
- If shadow ray can reach to the light, apply a standard Phong model

$$I = L \left( k_d (l \cdot n) + k_s (r \cdot v)^\alpha \right)$$



# Ray Tracing

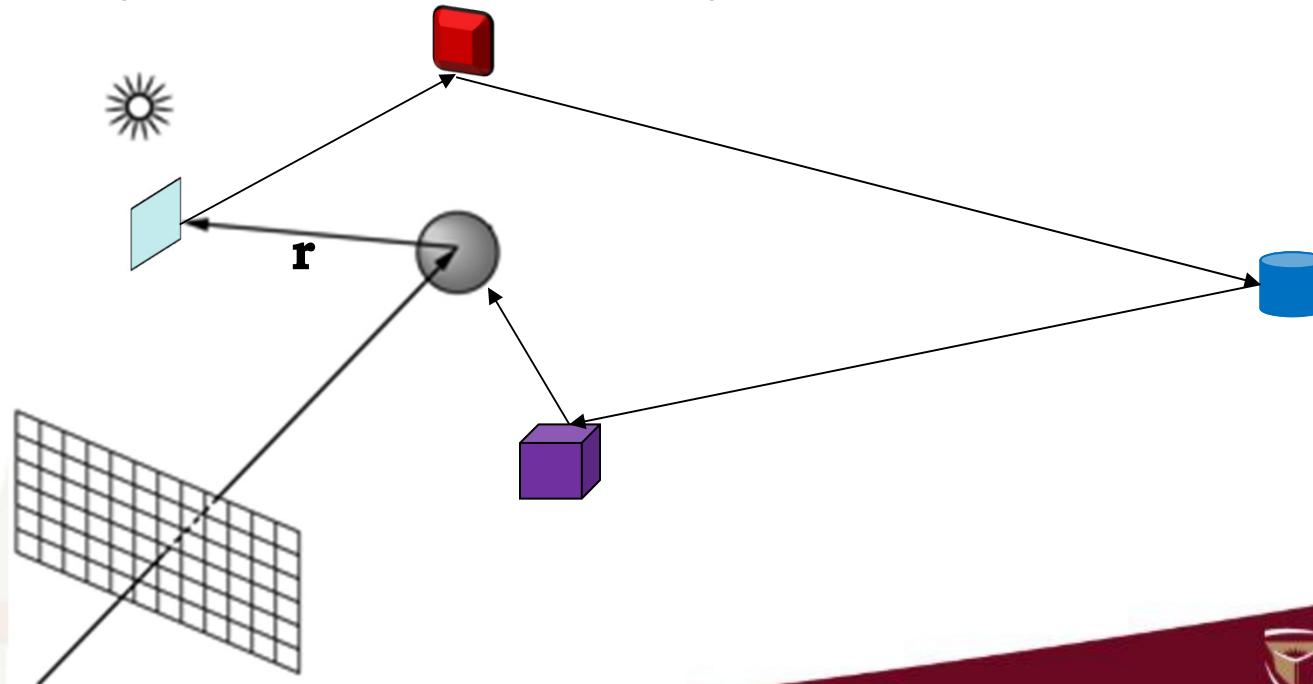
Where is Phong model applied in this example? Which shadow rays are blocked?



# Ray Tracing

## Reflection Rays

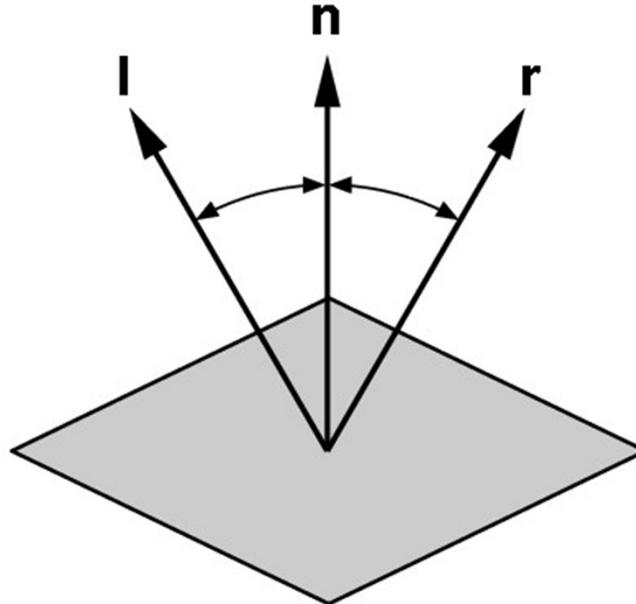
- For specular component of illumination
- Compute reflection ray (recall: backward)
- Call ray tracer recursively to determine color



# Ray Tracing

## Angle of Reflection

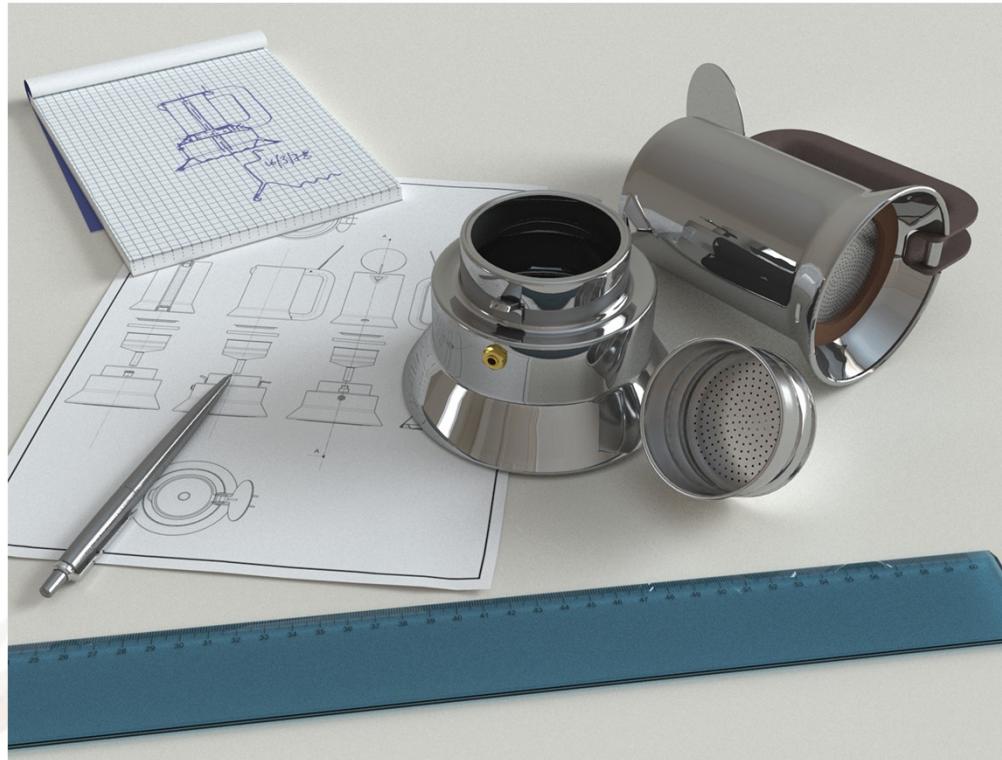
- Recall: incoming angle = outgoing angle
- $r = 2(l \cdot n) n-l$
- Compute only for surfaces that are reflective



# Ray Tracing

## Reflections Example

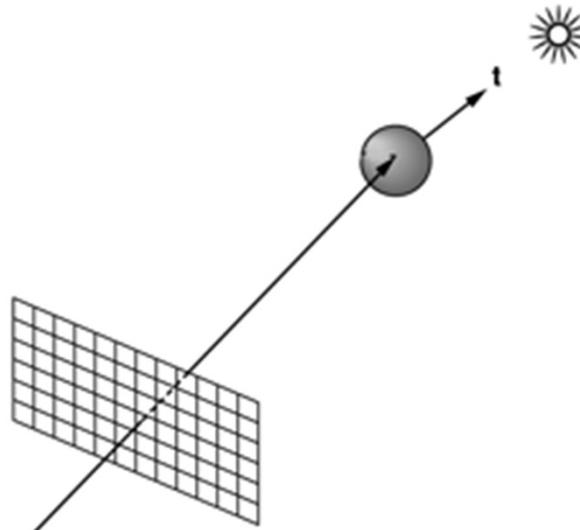
[www.yafaray.org](http://www.yafaray.org)



# Ray Tracing

## Transmission Rays

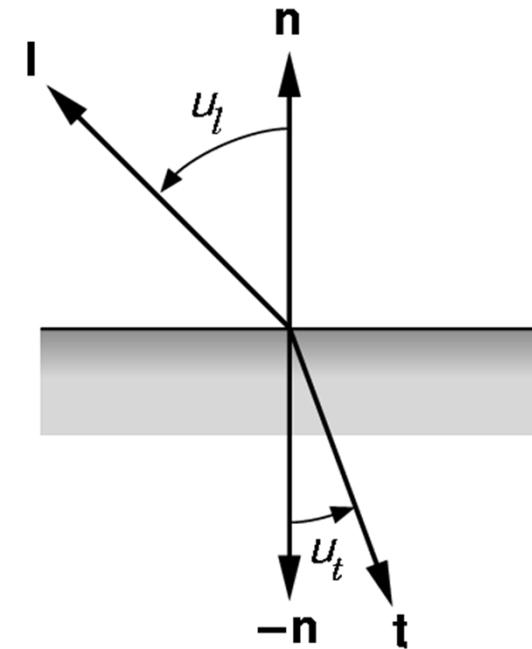
- Calculate light transmitted through surfaces
- Example: water, glass
- Compute transmission ray
- Call ray tracer recursively to determine color



# Ray Tracing

## Transmitted Light

- Index of refraction is speed of light, relative to speed of light in vacuum
  - Vacuum: 1.0 (per definition)
  - Air: 1.000277 (approx. to 1.0)
  - Water: 1.33
  - Glass: 1.49
- Compute  $t$  using Snell's law
  - $\eta_1$  = index for upper material
  - $\eta_2$  = index for lower material

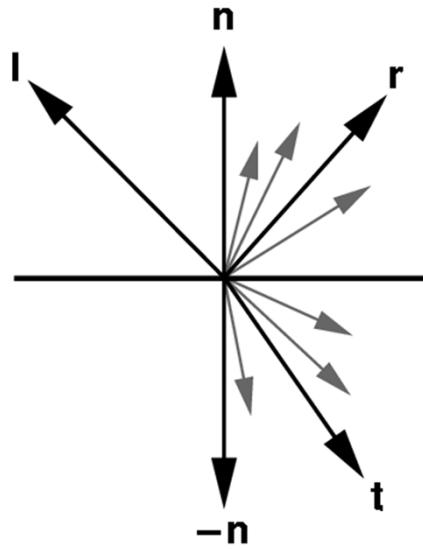


$$\frac{\sin(u_l)}{\sin(u_t)} = \frac{\eta_t}{\eta_l} = \eta$$

# Ray Tracing

## Translucency

- Most real objects are **not transparent** but blur the background image
- Scatter light on other side of surface
- Use stochastic sampling (called distribution ray tracing)



# Ray Tracing

## Transmission + Translucency Example

[www.povray.org](http://www.povray.org)



# Ray Tracing

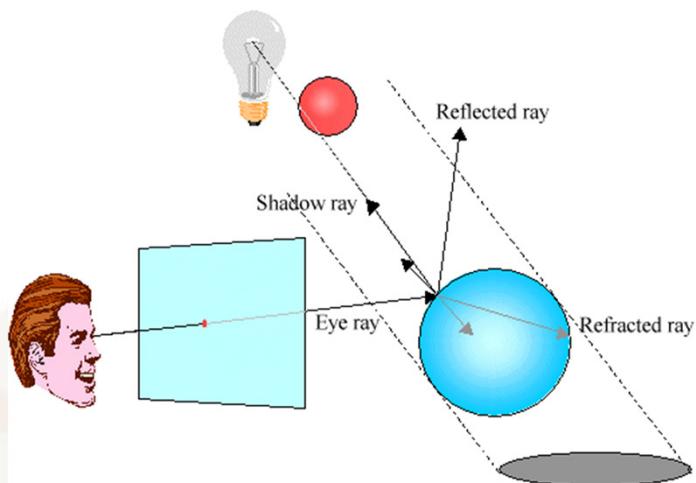
## The Ray Casting Algorithm

- **Simple case of ray tracing**
  - For each pixel  $(x, y)$  fire a ray from COP through  $(x, y)$
  - For each ray & object, calculate closest intersection
  - For closest intersection point  $p$ 
    - Calculate surface normal
    - For each light source, fire shadow ray
    - For each unblocked shadow ray, evaluate local Phong model for that light, and add the result to pixel color
- **Critical Operations**
  - Ray-surface intersections
  - Illumination calculation

# Ray Tracing

## Recursive Ray Tracing

- Also calculate specular component
  - Reflect ray from eye on specular surface
  - Transmit ray from eye through transparent surface
- Determine color of incoming ray by recursion
- Trace to fixed depth
- Cut off if contribution below threshold



# Ray Tracing

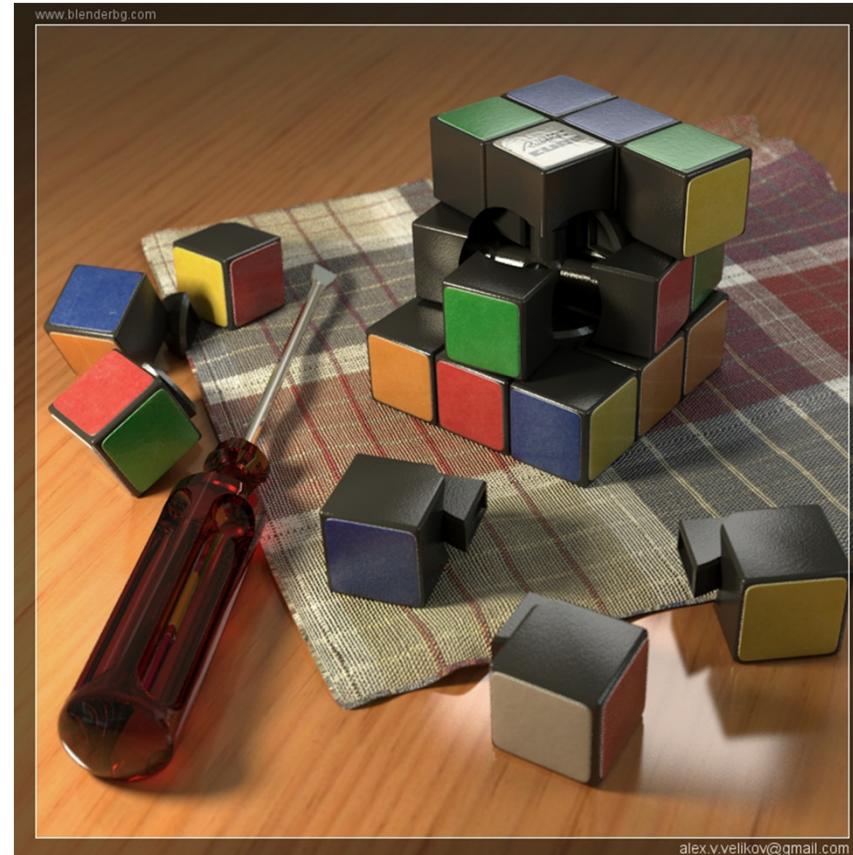
## Ray Tracing Assessment

- Global illumination method
- Image-based
- Pros
  - Relatively accurate shadows, reflections, refractions
- Cons
  - Slow (per pixel parallelism, not pipeline parallelism)
  - Aliasing
  - Inter-object diffuse reflections require many bounces

# Ray Tracing

## Ray Tracing Example I

[www.yafaray.org](http://www.yafaray.org)



alex.v.velikov@gmail.com

# Ray Tracing

## Ray Tracing Example II

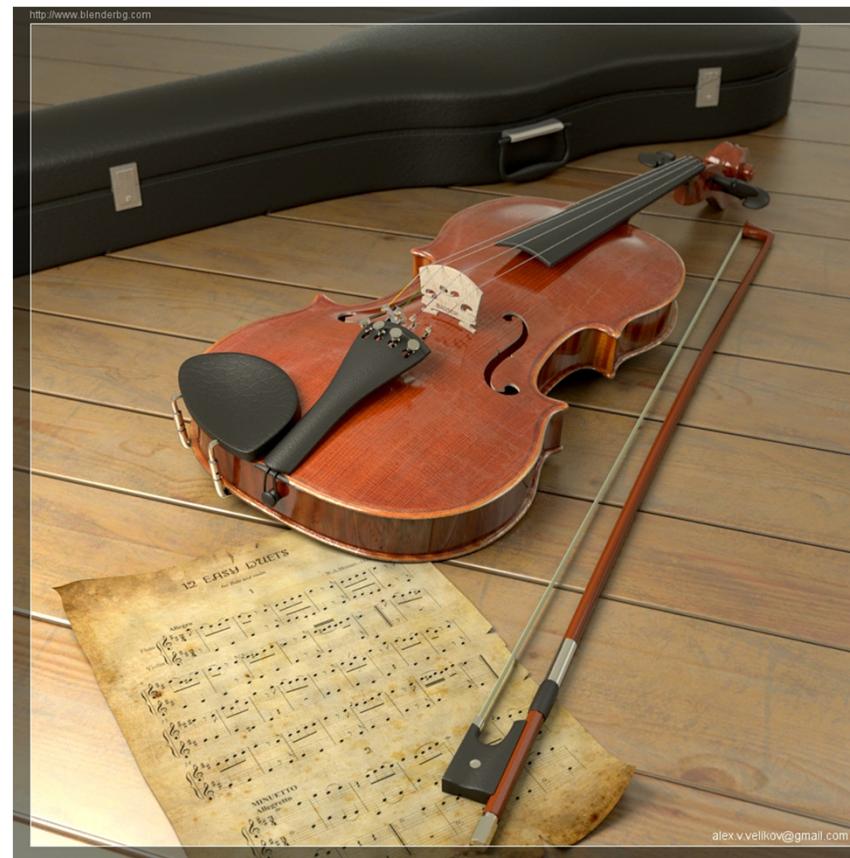
[www.povray.org](http://www.povray.org)



# Ray Tracing

## Ray Tracing Example III

[www.yafaray.org](http://www.yafaray.org)



# Ray Tracing

## Ray Tracing Example IV

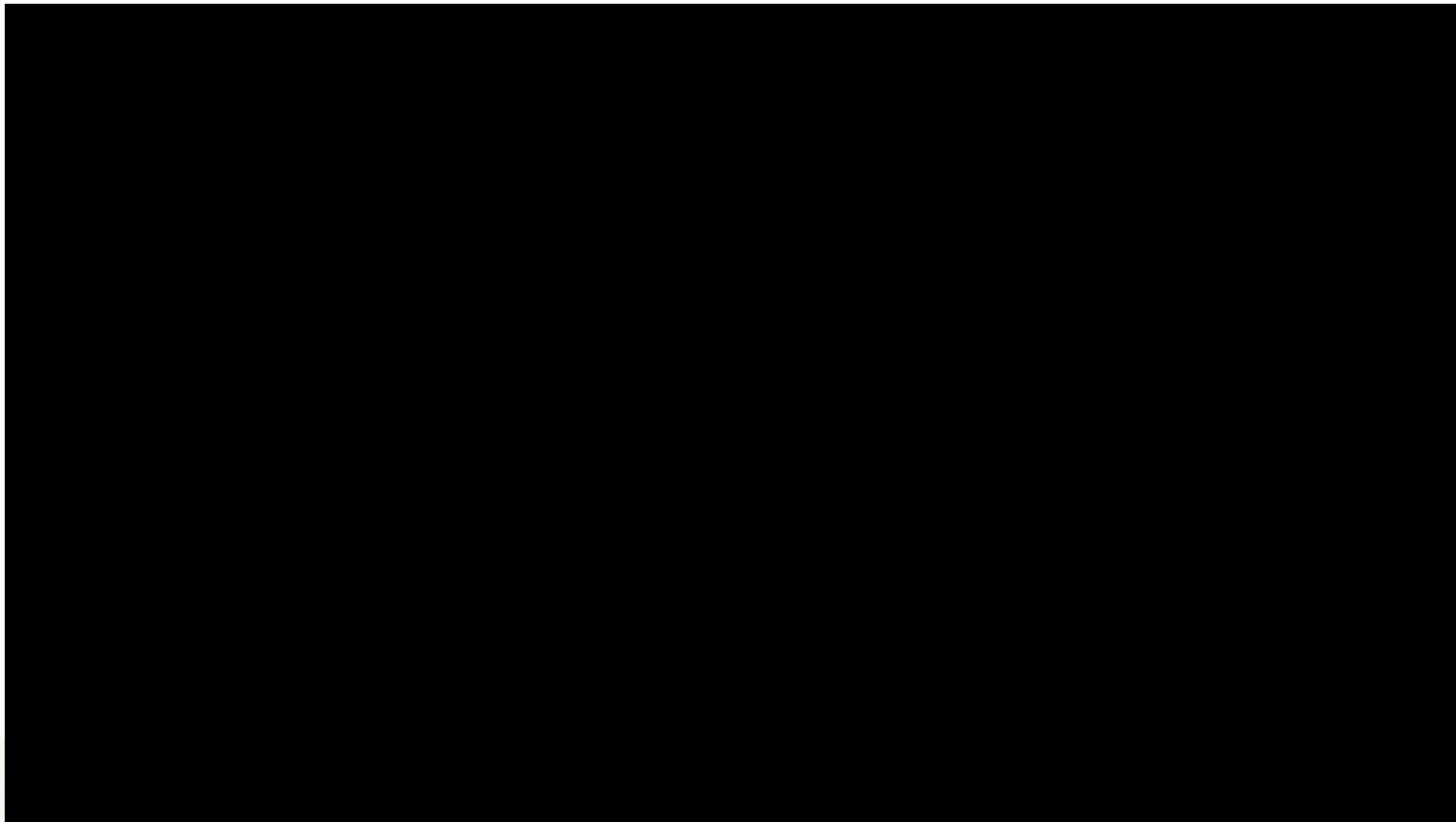
[www.povray.org](http://www.povray.org)



Copyright 2000 Gilles Tran

# Ray Tracing

## NVIDIA RTX Ray Tracing Tech Demo



<https://www.youtube.com/watch?v=pNmhJx8yPLk>

# Ray Tracing

## Review

- **Ray Casting/Tracing**
- **Shadow Rays and Local Phong Model**
- **Reflection**
- **Transmission**

# Next Lecture(s)

## Geometric Queries



**COMP 371**

# **Computer Graphics**

**Session 14**

**RAY TRACING/GEOMETRIC  
QUERIES**



# Lecture Overview

- Review of last class
- Ray Tracing/Geometric Queries

# Geometric Queries

## Ray-Surface Intersection

- Necessary in ray tracing
- General implicit surfaces
- General parametric surfaces
- Optimized solutions for special surfaces
  - Spheres
  - Planes
  - Polygons
  - Quadrics

# Geometric Queries

## Intersection of Rays and Parametric Surfaces

- Ray in parametric form
- Origin  $p_0 = [x_0 \ y_0 \ z_0]^T$
- Direction  $d = [x_d \ y_d \ z_d]^T$
- Assume  $d$  is normalized ( $x_d^2 + y_d^2 + z_d^2 = 1$ )
- Ray  $p(t) = p_0 + dt$  for  $t > 0$
- Surface in parametric form
  - Point  $q = g(u, v)$ , possible bounds on  $u, v$
  - Solve  $p_0 + dt = g(u, v)$
  - Three equations and three unknowns ( $t, u, v$ )

# Geometric Queries

## Intersection of Rays and Implicit Surfaces

- Ray in parametric form
- Origin  $p_0 = [x_0 \ y_0 \ z_0]^T$
- Direction  $d = [x_d \ y_d \ z_d]^T$
- Assume  $d$  is normalized ( $x_d^2 + y_d^2 + z_d^2 = 1$ )
- Ray  $p(t) = p_0 + dt$  for  $t > 0$
- Implicit surface
  - Given by  $f(q) = 0$
  - Consists of all points  $q$  such that  $f(q) = 0$
  - Substitute ray equation for  $q$ :  $f(p_0 + dt) = 0$
  - Solve for  $t$  (univariate root finding)
  - Closed form (if possible), otherwise numerical approximation

# Geometric Queries

## Ray-Sphere Intersection I

- Common and easy case

- Define sphere by

- Center  $c = [x_c \ y_c \ z_c]^T$

- Radius  $r$

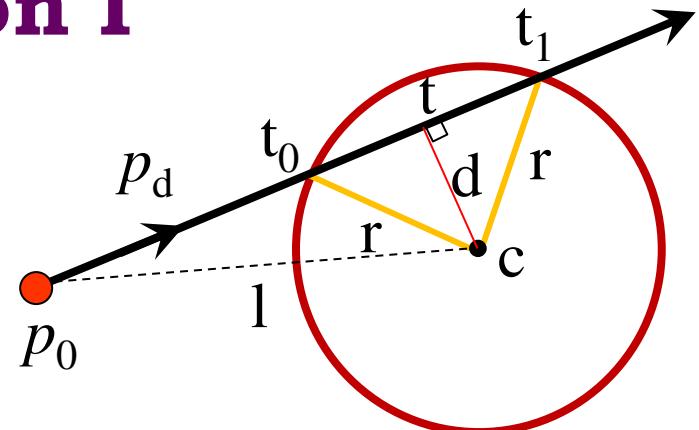
- Surface  $f(q) = (x-x_c)^2 + (y-y_c)^2 + (z-z_c)^2 - r^2 = 0$

- Plug in ray equations for  $x, y, z$ :

$$x = x_0 + x_d t, \quad y = y_0 + y_d t, \quad z = z_0 + z_d t$$

- And we obtain a scalar equation for  $t$ :

$$(x_0 + x_d t - x_c)^2 + (y_0 + y_d t - y_c)^2 + (z_0 + z_d t - z_c)^2 = r^2$$



# Geometric Queries

## Ray-Sphere Intersection II

- Simplify to  $at^2 + bt + c = 0$

- where

$$a = x_d^2 + y_d^2 + z_d^2 = 1 \quad \text{since } |d| = 1$$

$$b = 2(x_d(x_0 - x_c) + y_d(y_0 - y_c) + z_d(z_0 - z_c))$$

$$c = (x_0 - x_c)^2 + (y_0 - y_c)^2 + (z_0 - z_c)^2 - r^2$$

- Solve to obtain  $t_0$  and  $t_1$

$$t_{0,1} = \frac{-b \pm \sqrt{b^2 - 4c}}{2}$$

Check if  $t_0, t_1 > 0$  (ray)  
Return  $\min(t_0, t_1)$

# Geometric Queries

## Ray-Sphere Intersection III

For lighting, calculate unit normal

$$n = \frac{1}{r} [(x_i - x_c) \quad (y_i - y_c) \quad (z_i - z_c)]^T$$

Negate if ray originates inside the sphere

Note possible problems with round off errors

# Geometric Queries

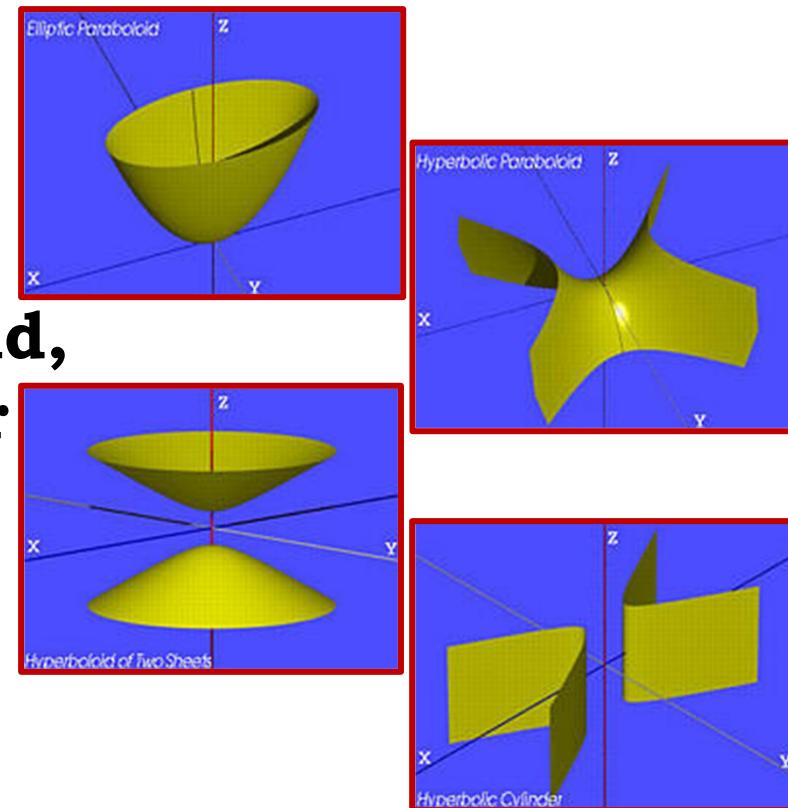
## Simple Optimizations

- Factor common sub-expressions
- Compute only what is necessary
  - Calculate  $b^2 - 4c$ , abort if negative
  - Compute normal only for closest intersection
  - Other similar optimizations

# Geometric Queries

## Ray-Quadric Intersection

- Quadric  $f(p) = f(x, y, z) = 0$ , where  $f$  is polynomial of order 2
- Sphere, ellipsoid, paraboloid, hyperboloid, cone, cylinder
- Closed form solution as for sphere
- Important case for modelling in ray tracing



# Geometric Queries

## Ray-Polygon Intersection I

- Assume planar polygon in 3D
  - Intersect ray with plane containing polygon
  - Check if intersection point is inside polygon
- Plane
  - Implicit form:  $ax + by + cz + d = 0$
  - Unit normal:  $n = [a \ b \ c]^T$  with  $a^2 + b^2 + c^2 = 1$
- Substitute  $a(x_0 + x_dt) + b(y_0 + y_dt) + c(z_0 + z_dt) + d = 0$
- Solve

$$t = \frac{-(ax_0 + by_0 + cz_0 + d)}{ax_d + by_d + cz_d}$$

# Geometric Queries

## Ray-Polygon Intersection II

- Substitute  $t$  to obtain intersection point in plane
- Rewrite using dot product

$$t = \frac{-(ax_0 + by_0 + cz_0 + d)}{ax_d + by_d + cz_d} = \frac{-(n \cdot p_0 + d)}{n \cdot d}$$

- If  $n \cdot d = 0$ , no intersection (ray parallel to plane)
- if  $t \leq 0$ , the intersection is behind ray origin

# Geometric Queries

## Test if point inside polygon

- Use even-odd rule or winding rule
- Easier if polygon is in 2D (project from 3D to 2D)
- Easier for triangles (tessellate polygons)

# Geometric Queries

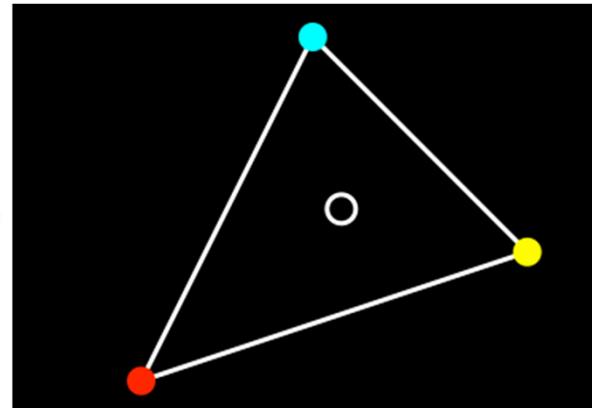
## Point-in-triangle testing

- Critical for polygonal models
- Project the triangle, and point of plane intersection, onto one of the planes  $x=0$ ,  $y = 0$  or  $z = 0$  (pick a plane not perpendicular to triangle - such a choice always exists)
- Then, do the 2D test in the plane, by computing barycentric coordinates (follows next)

# Geometric Queries

## Interpolated Shading for Ray Tracing

- Assume we know normal information at vertices
- How do we compute normal of interior point?
- Need linear interpolation between 3 points
- Barycentric coordinates
- Yields same answer as scan conversion



# Geometric Queries

## Barycentric Coordinates in 1D

- Linear interpolation
- $p(t) = (1-t)p_1 + tp_2, 0 \leq t \leq 1$
- $p(t) = ap_1 + bp_2$  where  $a + b = 1$
- $p$  is between  $p_1$  and  $p_2$  iff  $0 \leq a, b \leq 1$
- Geometric intuition
- Weigh each vertex by ratio distances from ends

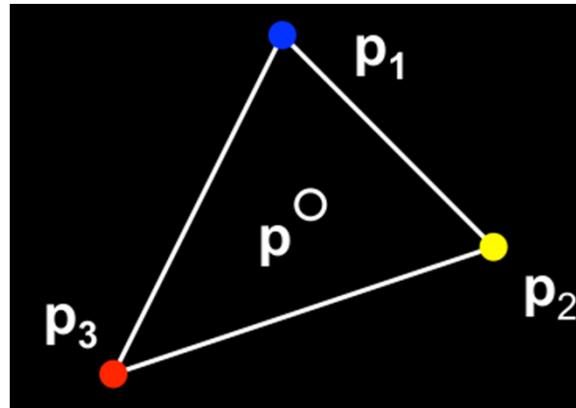


- $a, b$  are called barycentric coordinates

# Geometric Queries

## Barycentric Coordinates in 2D

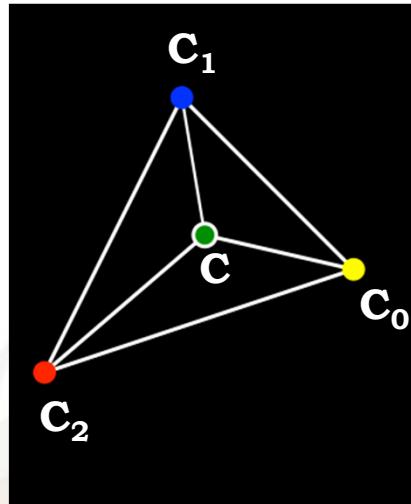
- Now, we have 3 points instead of 2
- Define 3 barycentric coordinates  $a, b, c$
- $p = ap_1 + bp_2 + cp_3$
- $p$  inside triangle iff  $0 \leq a, b, c \leq 1, a + b + c = 1$
- How do we calculate  $a, b, c$  given  $p$ ?



# Geometric Queries

## Barycentric Coordinates for Triangle

- Coordinates are ratios of triangle areas
- Areas in these formulas should be signed, depending on clockwise (-) or anti-clockwise orientation (+) of the triangle! Very important for point-in-triangle test



$$\alpha = \frac{\text{Area}(\mathbf{CC}_1\mathbf{C}_2)}{\text{Area}(\mathbf{C}_0\mathbf{C}_1\mathbf{C}_2)}$$

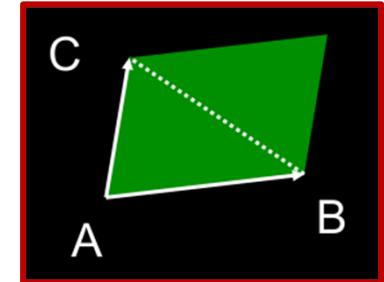
$$\beta = \frac{\text{Area}(\mathbf{C}_0\mathbf{C}\mathbf{C}_2)}{\text{Area}(\mathbf{C}_0\mathbf{C}_1\mathbf{C}_2)}$$

$$\gamma = \frac{\text{Area}(\mathbf{C}_0\mathbf{C}_1\mathbf{C})}{\text{Area}(\mathbf{C}_0\mathbf{C}_1\mathbf{C}_2)} = 1 - \alpha - \beta$$

# Geometric Queries

## Computing Triangle Area in 3D

- Use cross product
- Parallelogram formula
- $\text{Area(ABC)} = \frac{1}{2} | (\mathbf{B}-\mathbf{A}) \times (\mathbf{C}-\mathbf{A}) |$
- How to get correct sign for barycentric coordinates?
  - tricky, but possible:
    - compare directions of vectors  $(\mathbf{B}-\mathbf{A}) \times (\mathbf{C}-\mathbf{A})$ , for triangles  $\mathbf{CC}_1\mathbf{C}_2$  vs  $\mathbf{C}_0\mathbf{C}_1\mathbf{C}_2$ , etc. [either 0 (sign+) or 180 deg (sign-) angle]
    - easier alternative: project to 2D, use 2D formula
    - projection to 2D preserves barycentric coordinates



# Geometric Queries

## Computing Triangle Area in 2D

- Suppose we project the triangle to XY plane
- Area (xy-projection(ABC)) =  $(\frac{1}{2})((b_x - a_x)(c_y - a_y) - (c_x - a_x)(b_y - a_y))$
- This formula gives correct sign (important for barycentric coordinates)

# Geometric Queries

## Review

- Ray Tracing/Geometric Queries

## Next Lecture(s)

## Spatial Data Structures



**COMP 371**

# **Computer Graphics**

**Session 15**

**RAY TRACING ACCELERATION**



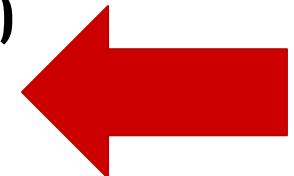
# Lecture Overview

- **Review of last class**
- **Spatial Data Structures**
  - **Hierarchical Bounding Volumes**
  - **Regular Grids**
  - **Octrees**
  - **BSP Trees**

# Ray Tracing

## Ray Tracing Acceleration

- **Faster Intersections**
  - Faster ray-object intersections
    - Object bounding volume
    - Efficient computation of intersections
- **Fewer Ray-Object Intersections**
  - Hierarchical bounding volumes (boxes, spheres)
  - Spatial data structures
  - Directional techniques
- **Fewer Rays**
  - Adaptive tree-depth control
  - Stochastic sampling
- **Generalized Rays (beams, cones)**



# Spatial Data Structures

## Advantages

- Data structures to store geometric information
- Speed up various applications like:
  - Collision detection
  - Location queries
  - Simulations
  - Rendering
- Spatial data structures for Ray Tracing
  - Object-centric data structures (bounding volumes)
  - Space subdivision (grids, Octrees, BSP trees)
  - Speed-up of 10x, 100x, or more

# Spatial Data Structures

## Bounding Volumes

- Wrap complex objects in simple ones
- Does ray intersect bounding volume?
  - No: *does not intersect* enclosed objects
  - Yes: calculate *intersection with closest objects*
- Common types:



Sphere



Axis-aligned  
Bounding  
Box (AABB)



Oriented  
Bounding  
Box (OBB)



6-dop



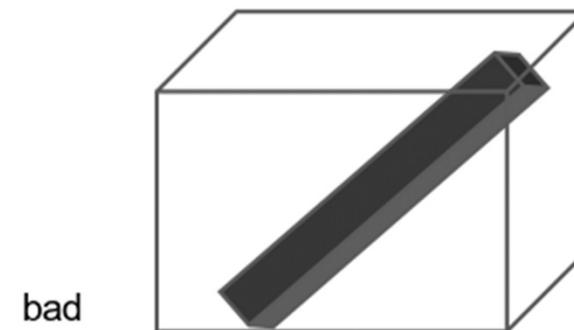
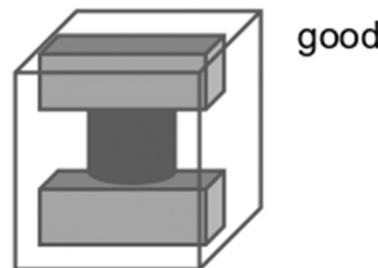
Convex Hull



# Spatial Data Structures

## Selection of Bounding Volumes

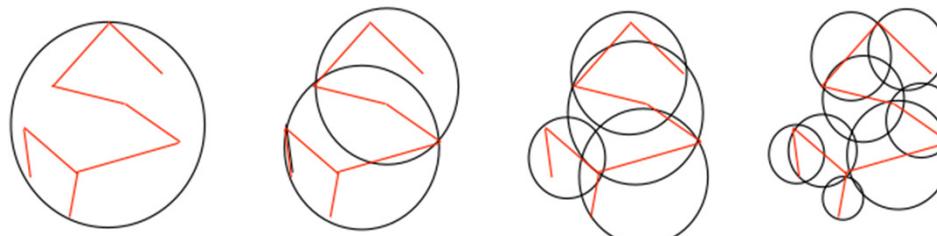
- Effectiveness depends on:
  - *Probability that ray hits bounding volume, but not enclosed objects (tight fit is better)*
  - Cost of calculating intersections with bounding volume versus enclosed objects
- Use Heuristics



# Spatial Data Structures

## Hierarchical Bounding Volumes

- With simple bounding volumes, ray casting still *requires  $O(n)$  intersection tests for each ray*
- Idea: *use tree data structure*
  - Larger bounding volumes contain smaller ones, etc.
  - Sometimes naturally available (e.g. human figure)
  - Sometimes difficult to compute
- Often *cuts complexity to  $O(\log(n))$  for each ray*



# Spatial Data Structures

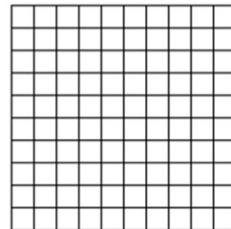
## Ray Intersection Algorithm

- Recursively descend down the tree
- If ray misses bounding volume, no intersection
- If ray intersects bounding volume, *recurse with the enclosed volumes and objects*
- Maintain near and far bounds to prune further
- Overall effectiveness depends on *model and constructed hierarchy*

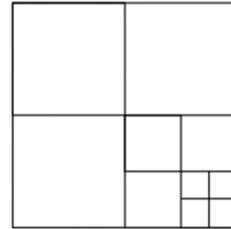
# Spatial Data Structures

## Spatial Subdivision

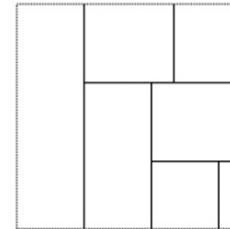
- Bounding volumes enclose objects, recursively *i.e. object-centric*
- Alternatively, divide space (*as opposed to objects*)
  - For each segment of space, keep a list of intersecting surfaces or objects
  - Basic techniques:



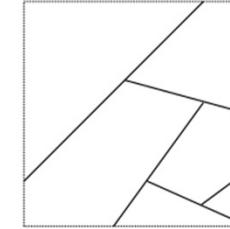
Uniform  
Spatial Sub



Quadtree/Octree



kd-tree

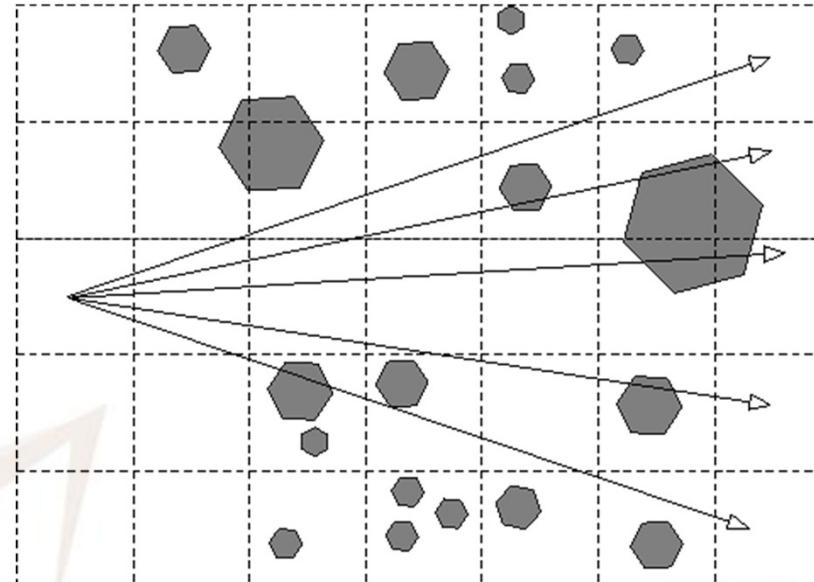


BSP-tree

# Spatial Data Structures

## Grids

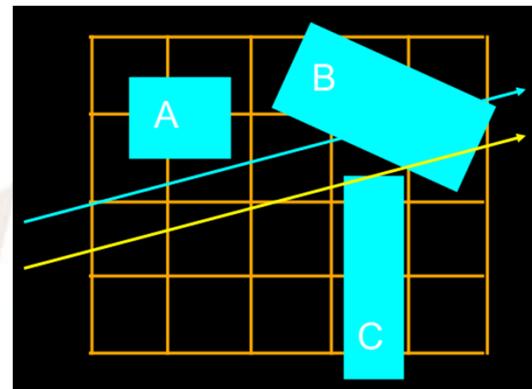
- 3D array of cells (**voxels**) that tile space
  - Each cell keeps *a list of all intersecting surfaces*
  - Intersection algorithm steps from cell to cell



# Spatial Data Structures

## Caching Intersection Points

- An object *can span multiple cells*
- One cell can be occupied by multiple objects
- For A need to test intersection only once
- For B need to *cache intersection* and **check next cell for any closer intersection** with other objects
  - If not, C could be missed (yellow ray)



# Spatial Data Structures

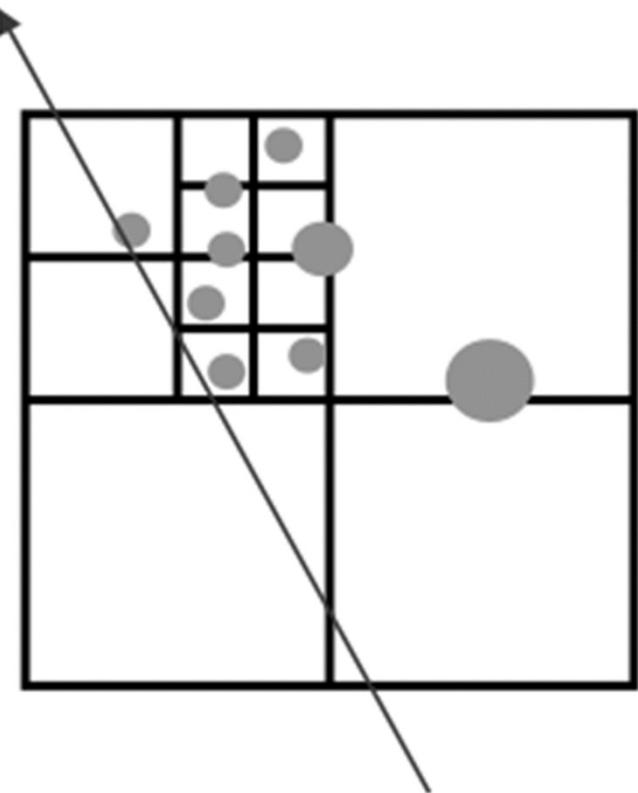
## Assessment of Grids

- Poor choice when world is non-homogeneous
- Grid resolution:
  - Too small: too many surfaces per cell
  - Too large: too many empty cells to traverse
  - Can use algorithms like 3D DDA (Bresenham) for *efficient traversal*
- Non-uniform spatial subdivision more flexible
  - Can adjust to objects that are present

# Spatial Data Structures

## Quadtrees

- Generalization of binary trees in 1D
  - Node (cell) is *a square*
  - Recursively split into 4 equal sub-squares
  - Stop subdivision based on number of objects
- Ray intersection has to *traverse quadtree*
- More difficult to step to next cell



# Spatial Data Structures

## Octrees

- Generalization of quadtree in 3D
- Each cell may be *split into 8 equal sub-cells*
- Internal nodes store pointers to children
- Leaf nodes store list of surfaces
- Adapts well to non-homogeneous scenes

# Spatial Data Structures

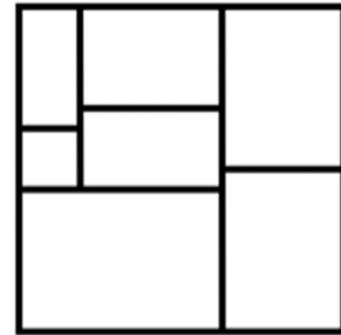
## Assessment for Ray Tracing

- **Grids**
  - Easy to implement
  - Require a lot of memory
  - Poor results for non-homogeneous scenes
- **Octrees**
  - Better on most scenes (*more adaptive*)
  - Alternative: nested grids
- **Spatial subdivision expensive for animations**
- **Hierarchical bounding volumes**
  - Natural for hierarchical objects
  - Better for *dynamic scenes*

# Spatial Data Structures

## Other Spatial Subdivision Techniques

- Relax rules for Quadtrees and Octrees
- **k-dimensional tree (k-d tree)**
  - Split at *arbitrary interior point*
  - Split *one dimension at a time*
- **Binary space partitioning tree (BSP tree)**
  - In 2 dimensions, *split with any line*
  - In k dims., split with k-1 dimensional hyperplane
  - Particularly *useful for painter's algorithm*
  - Can also be used for ray tracing



# Spatial Data Structures

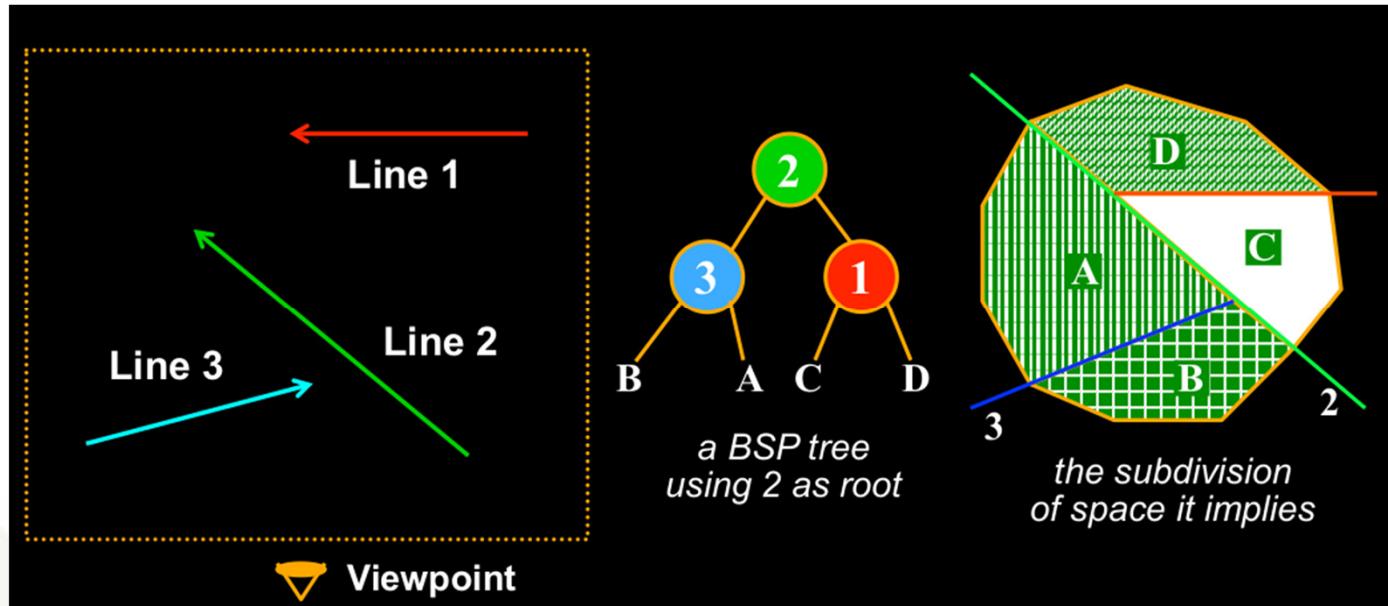
## BSP Trees

- Split space with any line (2D) or plane (3D)
- Applications
  - Painter's algorithm for hidden surface removal
  - Ray casting
- Inherent spatial ordering given viewpoint
  - Left subtree: in front, right subtree: behind
- Problem: finding good space partitions
  - Proper ordering for any viewpoint
  - How to balance the tree

# Spatial Data Structures

## Building a BSP Tree

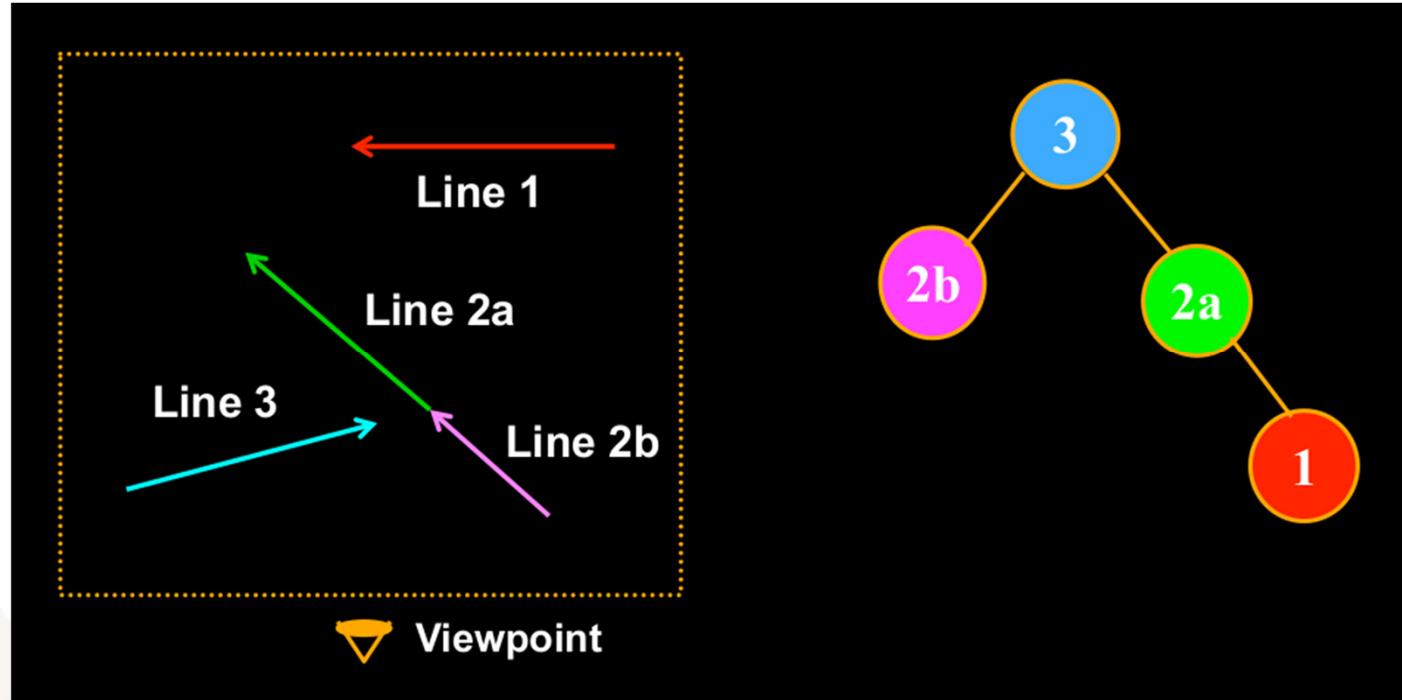
- Use *hidden surface removal* as intuition
- Using line 1 or line 2 as root is easy



# Spatial Data Structures

## Splitting of Surfaces

- Using line 3 as root requires splitting



# Spatial Data Structures

## Building a Good Tree

- **Naive partitioning of  $n$  polygons yields  $\underline{O(n^3)}$  polygons (in 3D)**
- **Algorithms with  $\underline{O(n^2)}$  exist**
  - Try all, use polygon with fewest splits
  - Do not need to split exactly along polygon planes
- **Should balance tree**
  - More splits allow easier balancing
  - Rebalancing?

# Spatial Data Structures

## Painter's Algorithm with BSP Trees

- **Building the tree**
  - May need to split some polygons
  - Slow, but done only once
- **Traverse back-to-front or front-to-back**
  - Order is *viewer-direction dependent*
  - What is front and what is back of each line changes
  - Determine order on the fly

# Spatial Data Structures

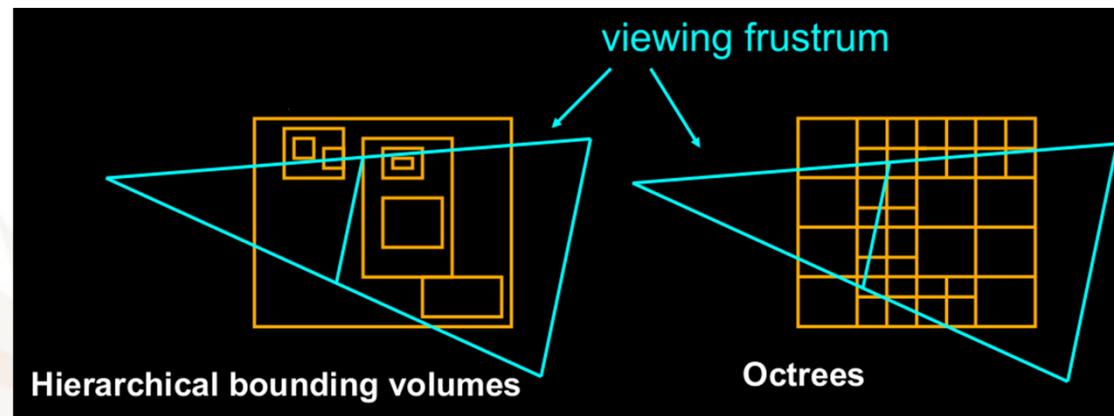
## Details of Painter's Algorithm

- Each plane has form  $Ax + By + Cz + D$
- Plug in coordinates and determine
  - Positive: front side
  - Zero: on plane
  - Negative: back side
- Back-to-front: inorder traversal, farther child first
- Front-to-back: inorder traversal, near child first
- Do *backface culling with same sign test*
- Clip against visible portion of space (portals)

# Spatial Data Structures

## Clipping with Spatial Data Structures

- Accelerate clipping
  - Goal *accept or reject whole sets of objects*
  - Can use any spatial data structures
- Scene should be mostly fixed
  - Terrain fly-through
  - Gaming



# Spatial Data Structures

## Data Structures Demos

- **BSP Tree construction**

<http://symbolcraft.com/graphics/bsp/index.html>

- **KD Tree construction**

<http://donar.umiacs.umd.edu/quadtree/points/kdtree.html>

# Spatial Data Structures

## Real-Time and Interactive Ray Tracing

- **Interactive ray tracing via space subdivision**

<http://www.cs.utah.edu/~reinhard/egwr/>

- **State of the art in interactive ray tracing**

<http://www.cs.utah.edu/~shirley/irt/>

# Review

## Spatial Data Structures

# Next Lecture

## Key-frame Animation



**COMP 371**

# **Computer Graphics**

**Session 16**  
**TEXTURE MAPPING**



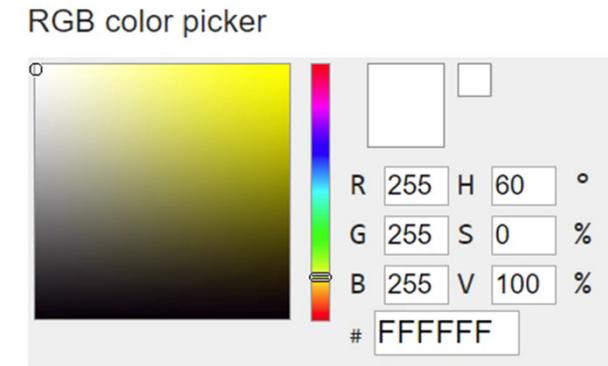
# Lecture Overview

- Review of last class
- Color
  - Blending and Fog effects
- Texture Mapping and Decals

# Texture Mapping

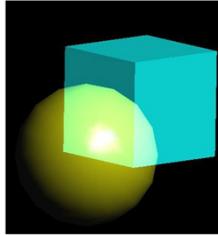
## Color in OpenGL

- **`vec3 col(1.0, 0.0, 0.0)`**
  - this presents a **red** color
  - use an **RGB** model, and value is **float** in C
- **Four-color (RGBA) system**
  - A is called alpha channel, stored in the frame buffer as are the RGB values
  - The alpha value will be treated by OpenGL as an opacity or transparency
  - **`vec4 col(1.0, 0.0, 0.0, 0.5)`**
    - this presents a partially transparent **red** color



# Texture Mapping

## Transparency



- When drawing an object (green window) atop another (gray interior), what should be the effect? We should see both the window and the interior with appropriate colors
- `vec4(1, 1, 1, 0.75)`: how does the 4<sup>th</sup> parameter, **the alpha value**, work?
- The alpha value represents the opacity of the object
  - Low alpha value: transparent or translucent objects
  - High alpha value: opaque objects
- It is used for blending, i.e., combining the new color values (**source**) with the existing ones (**destination**), making **translucent** scenes
- Must enable blending:

```
glBlendFunc(sFactor,dFactor);
	glEnable(GL_BLEND);
```

# Texture Mapping

## Color Blending in Frame Buffer

- OpenGL blends color of an object currently in frame buffer (destination color) with that of a second object (source color)
- There are several choices for the source blending factor ( $S_r$ ,  $S_g$ ,  $S_b$ ,  $S_a$ ) and the destination blending factor ( $D_r$ ,  $D_g$ ,  $D_b$ ,  $D_a$ ): **GL\_ONE** (default for source blending factor such that  $S_i = 1$ ), **GL\_ZERO** (default for destination blending factor such that  $D_i = 0$ ), **GL\_SRC\_ALPHA**, **GL\_DST\_ALPHA**, **GL\_ONE\_MINUS\_SRC\_ALPHA**, **GL\_SRC\_COLOR**, etc.
- New, blended color is:  
$$(S_r R_s + D_r R_d, S_g G_s + D_g G_d, S_b B_s + D_b B_d, S_a A_s + D_a A_d)$$
- Default replaces previous color in frame buffer with new one
- A good choice for blending is:

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

# Texture Mapping

## Transparent Objects

### 1. Where to set the transparency? (Opacity)

- Per vertex (vertex color), or
- Per model (shader uniform variable)

### 2. Drawing Order

- Disable blending
- Draw all opaque models
- Enable blending
- Draw transparent objects sorted from back to front



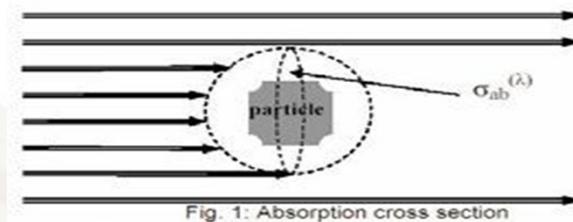
# Texture Mapping

## Light Extinction

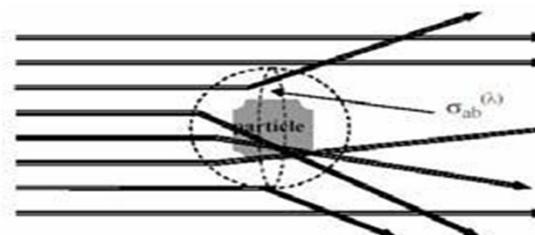
- Probability of photon absorption or out-scattering is proportional to distance light ray travels through a medium
- Monochromatic extinction: Fog makes more distant objects dimmer
- Different absorption probabilities for different wavelengths → Perceived color of source can change



(COURTESY: PARMOUNT PICTURES)



absorption



Out-scattering

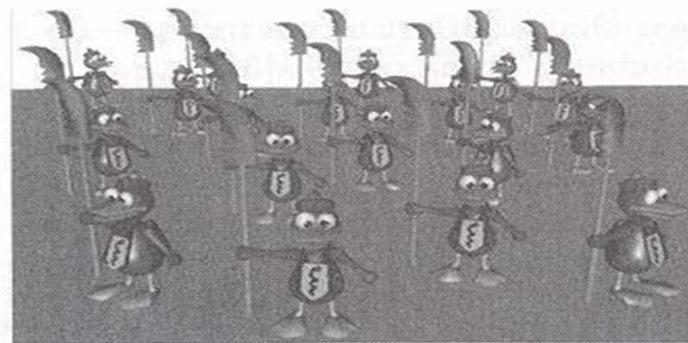
# Texture Mapping

## Distance fading and fogging

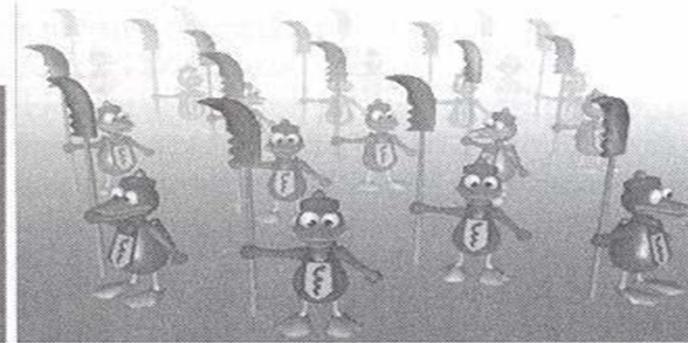
- Fogging can be used to recreate more natural scenes, by having distant objects merge into the color of the ‘background’

$$C = fC_i + (1 - f)C_{fog}$$

- $C_i$  is the incoming color;  $C_{fog}$  is the fog color



Without Fog



With  
(white  
colored)  
Fog

# Texture Mapping

## OpenGL Fog

- Attenuation: Fog “transparency”  $f$  (in [0, 1] range) as function of depth
  - Linear:  $f = (\text{end} - z)/(\text{end} - \text{start})$
  - Exponential:  $f = e^{-(\text{density} \cdot z)}$



Gray, exponential fog (density = 0.35, start/end coincident with first/last sphere centers)

# Texture Mapping

## Fogging functions

- $f$  can be generated by selecting one of three functions of distance (of the vertex from the camera position):

### Fog function parameters

| inverse exponential                 | inverse exponential Squared           | linear                                                 |
|-------------------------------------|---------------------------------------|--------------------------------------------------------|
| $f = e^{-(\text{density} \cdot z)}$ | $f = e^{-(\text{density} \cdot z)^2}$ | $f = \frac{\text{end} - z}{\text{end} - \text{start}}$ |

$$\text{FragmentColor} = f * \text{FogColor} + (1 - f) * \text{Color}$$

# Texture Mapping

## Fogging functions



# Texture Mapping

## Fog

- Fog is generally implemented in the Fragment shader
- Nice tutorial here:  
<http://www.mbssoftworks.sk/index.php?page=tutorials&series=1&tutorial=15>

# Texture Mapping

## Visual Details

- Add Visual Detail to Surfaces of 3D Objects



Polygonal Model

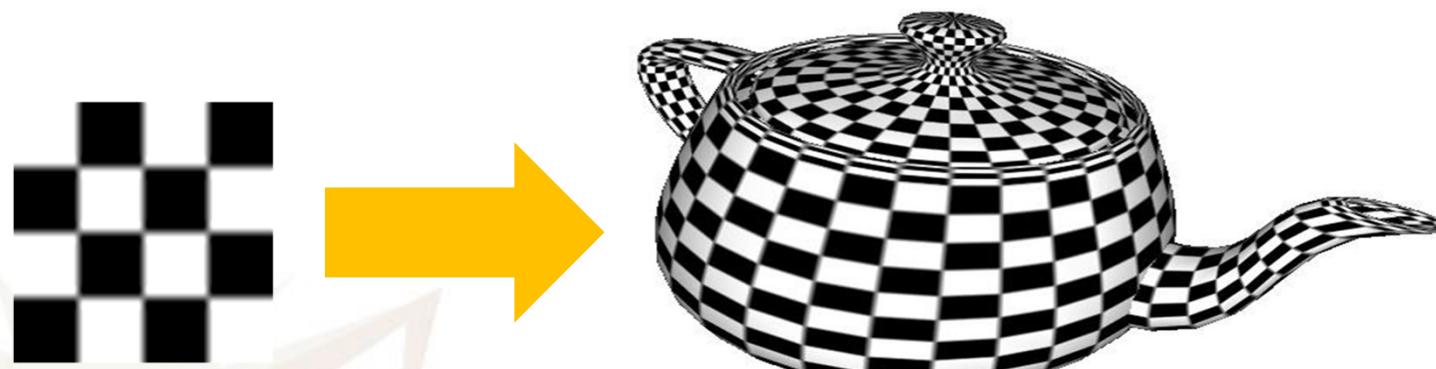


With Surface Texture

# Texture Mapping

## What is Texture Mapping?

- Allows you to attach an image to an object (*i.e.*, polygon)
- Supports all object transformations
- Texture is a rectangular array of data (colors)
- The individual values in a texture are called **texels**

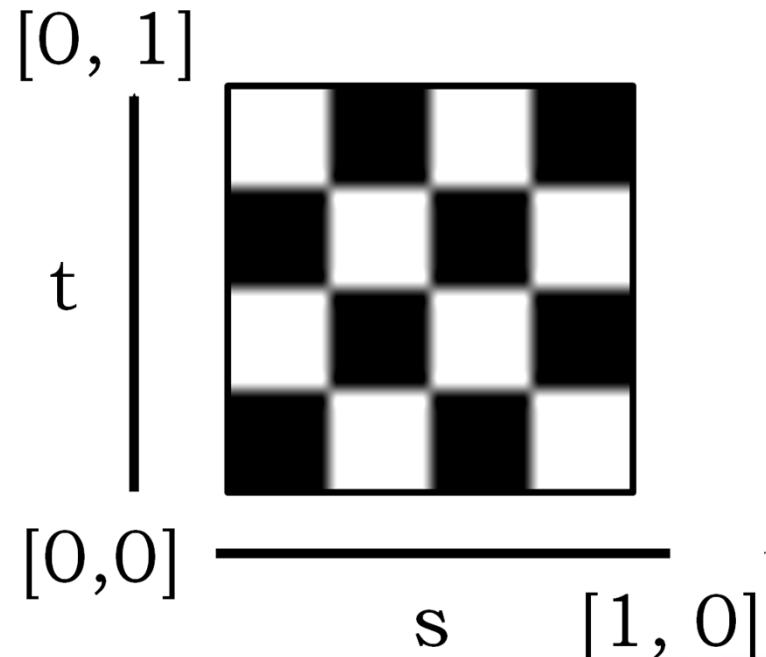


# Texture Mapping

## Texture Coordinates

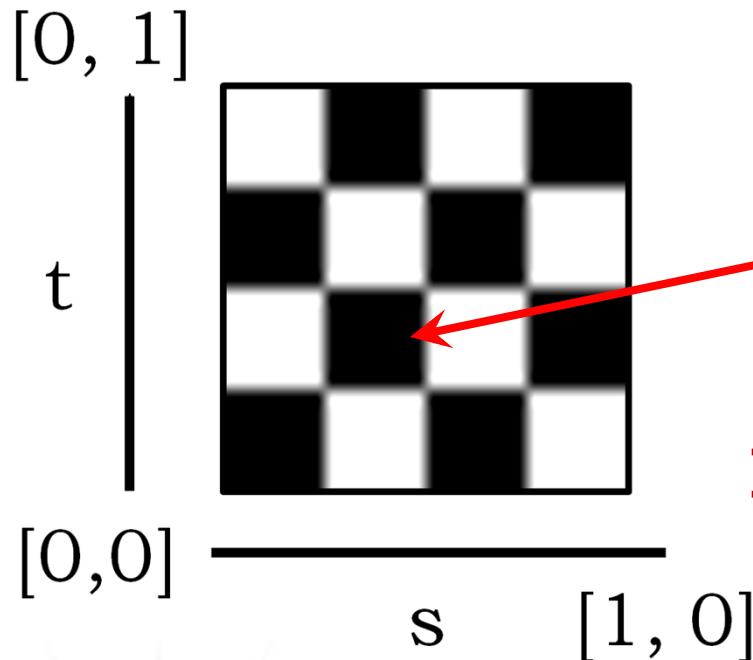
- A texture is usually addressed parametrically by values between zero and one
- The “addresses” of points on the texture consist of two numbers ( $s, t$ )
- A vertex can be associated with a point on the texture by giving it one of these texture coordinates
- Texture coordinates are part of the vertices like colors and normals

...although it is really just an array of pixels



# Texture Mapping

## Pixels & Texture Coordinates



For example:

a 32 x 32 pixel RGB image

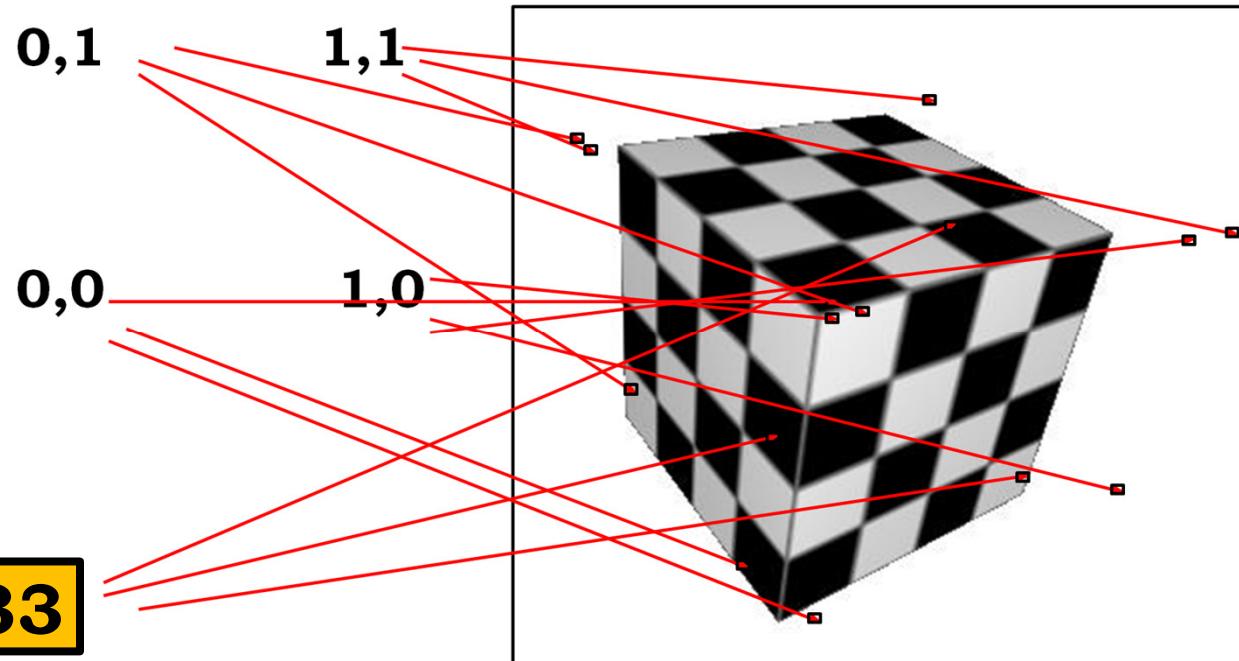
Glubyte **mychecker[32][32][3];**

# Texture Mapping

## Texture Coordinates to Polygons

**Texture  
Coordinates**

**s, t = .33, .33**

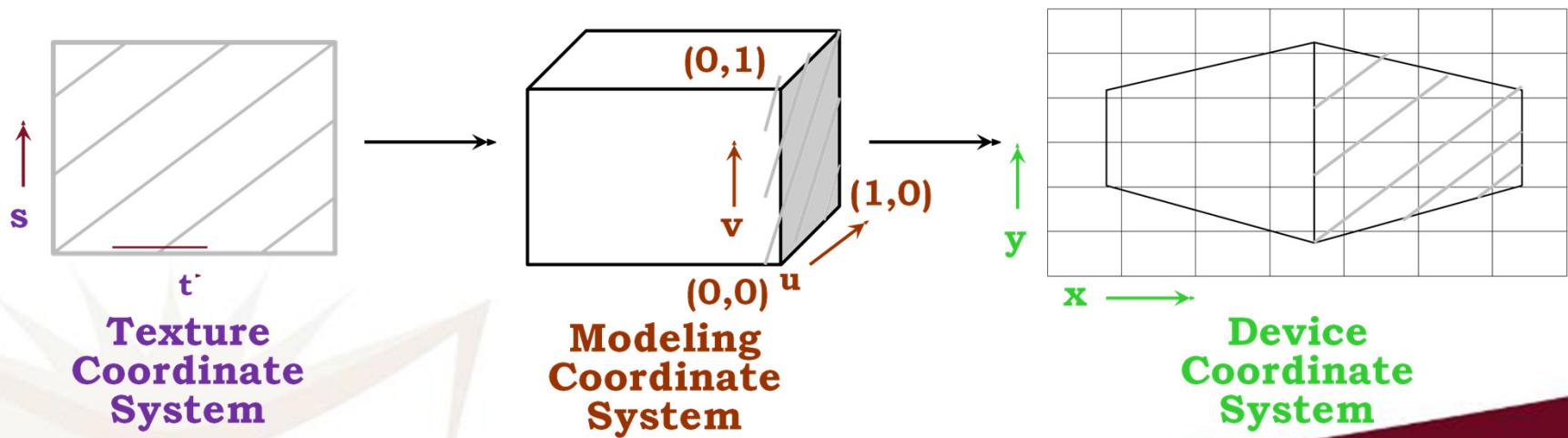


- Each vertex of each polygon is assigned a texture coordinate
- OpenGL finds the appropriate texture coordinate for points “between” vertices during Rasterization
  - same idea as smooth shading!

# Texture Mapping

## Steps

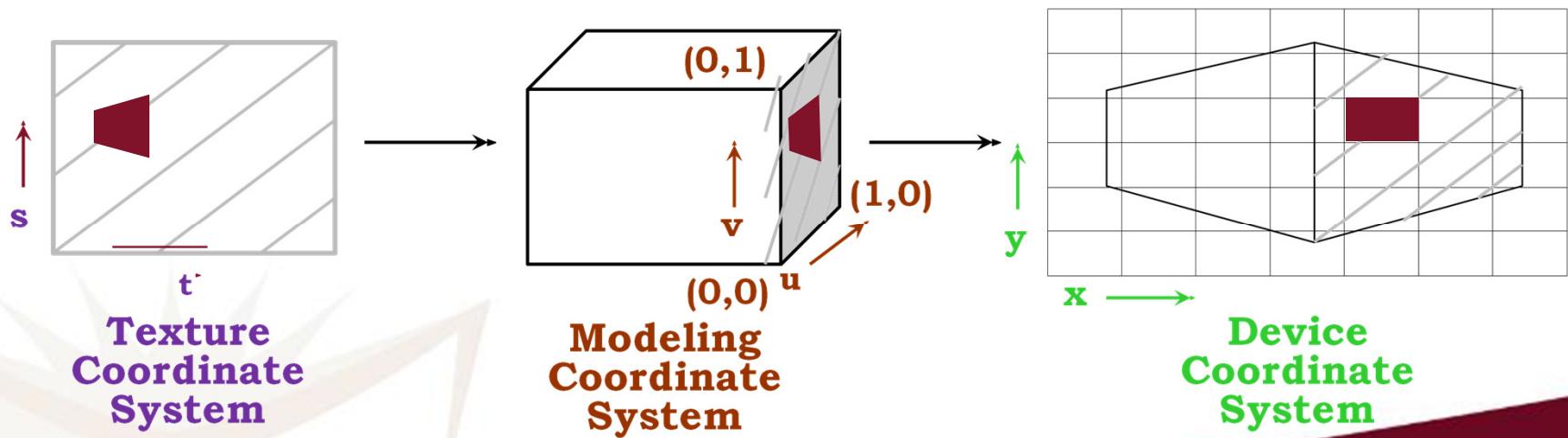
- Define/Load texture
- Specify mapping from **texture to surface**
- Lookup texture values during scan conversion
- But we really want to go the **other way**



# Texture Mapping

## Steps

- When Scan Converting, Map from ...
  - Device coordinate system ( $x, y$ ) to
  - Modeling coordinate system ( $u, v$ ) to
  - Texture image ( $t, s$ )



# Texture Mapping

## Texture Functions (Boundaries)

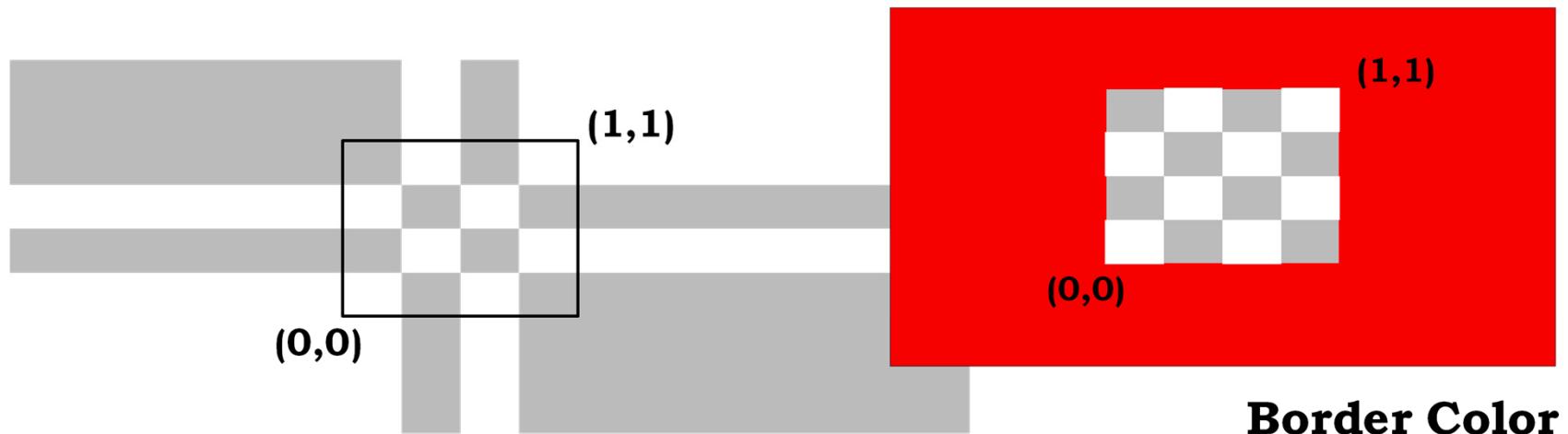
- **Textures are assumed to be in a (0,0) to (1,1) domain in texture space**

You can control what happens if a point maps to a texture coordinate outside of the domain

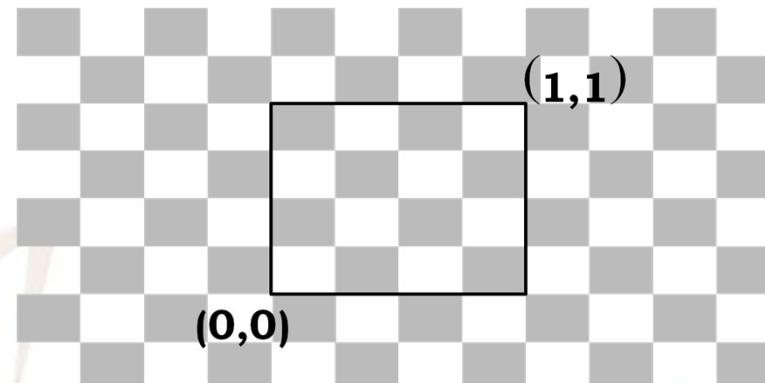
- **Repeat: Assume the texture is tiled**
  - `glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT)`
- **Clamp: Clamp to Edge: the texture coordinates are truncated to valid values, and then used**
  - `glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP)`
- **Can specify a special border color:**
  - `glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, R,G, B,A)`
- **A nice tutorial available at**  
<https://open.gl/textures>

# Texture Mapping

## Boundaries



**Clamped**

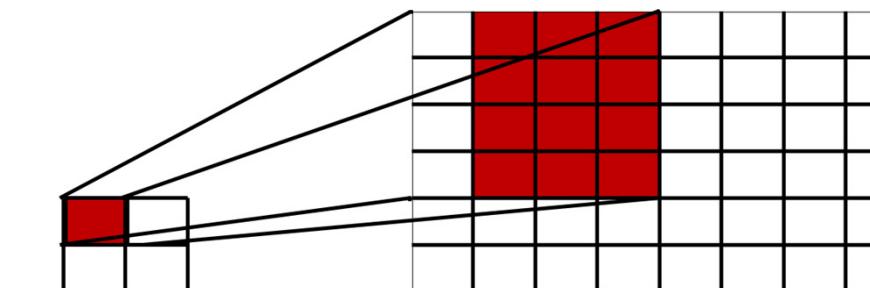


**Tiled**

# Texture Mapping

## Magnification and Minification

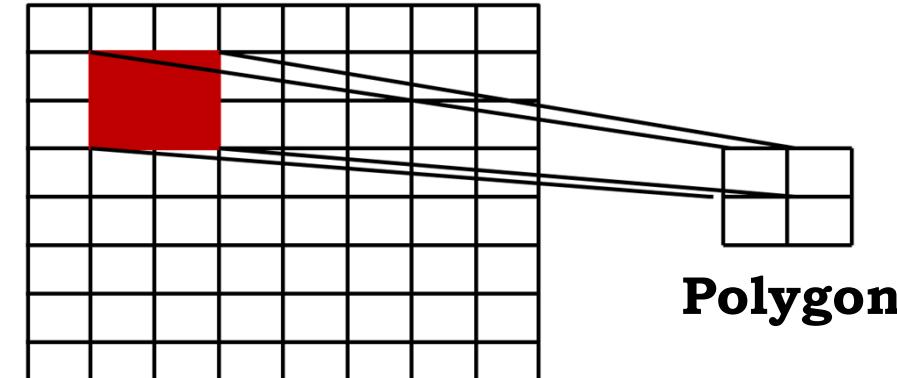
- More than one texel can cover a pixel (**minification**) or more than one pixel can cover a texel (**magnification**)
- Can use point sampling (nearest texel) or linear filtering ( 2 x 2 filter) to obtain texture values



Texture

Polygon

Magnification



Polygon

Texture  
Minification

# Texture Mapping

## Filter Modes

- Modes determined by
  - `glTexParameteri(target, type, mode)`

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
 GL_NEAREST);
```

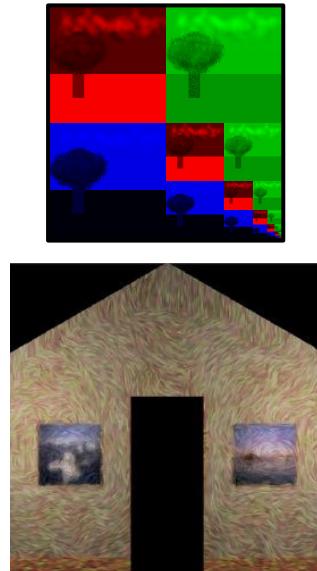
```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
 GL_LINEAR);
```

Note that linear filtering requires a border of an extra texel for filtering at edges (border = 1)

# Texture Mapping

## Mip Maps

- **Keep Textures Prefiltered at Multiple Resolutions**
  - For each pixel, linearly interpolate between two closest levels (e.g., trilinear filtering)
  - Fast, easy for hardware



# Texture Mapping

## Controlling Different Parameters

- The “pixels” in the texture map may be interpreted as many different things. For example:
  - As colors in RGB or RGBA format
  - As grayscale intensity
  - As alpha values only
- The data can be applied to the polygon in many different ways:
  - Replace: Replace the polygon color with the texture color
  - Modulate: Multiply the polygon color with the texture color or intensity
  - Similar to compositing: Composite texture with base color using an operator

# Texture Mapping

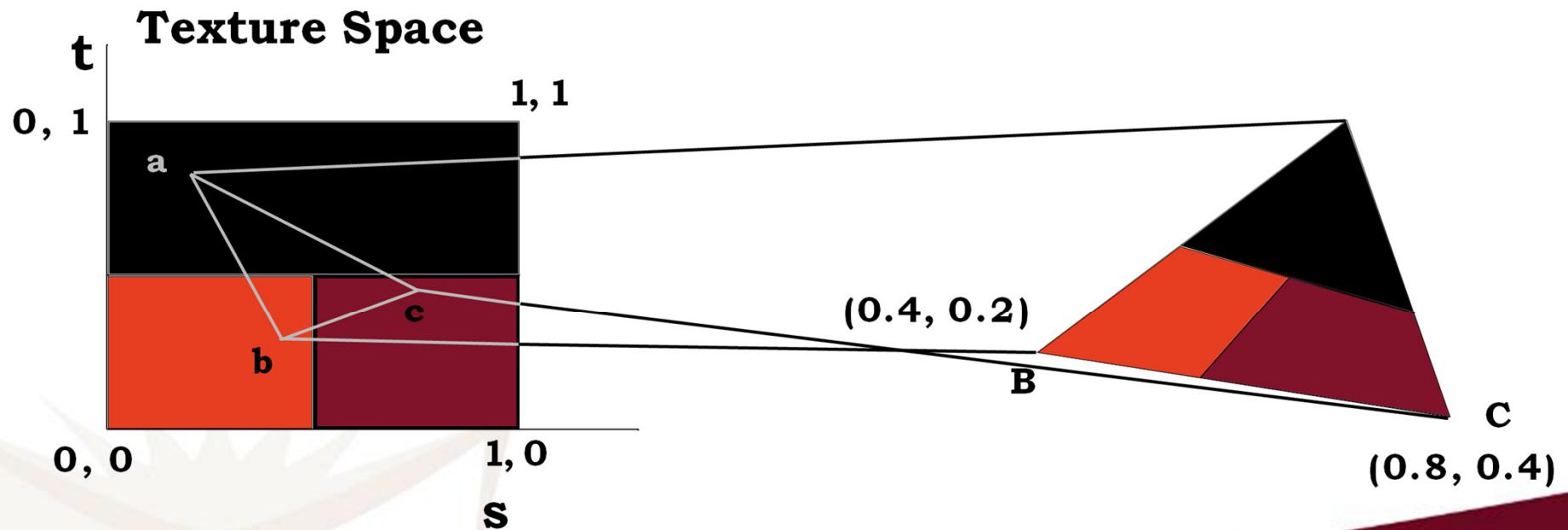
## Example: Texture and Lighting

- **Texture mapping happens in the Fragment shader**
- **Say you want to have an object textured and have the texture appear to be diffusely lit**
- **Problem:** Since texture is applied with lighting, so how do you calculate the texture's brightness?
- **Solution:**
  - Make the polygon white and light it normally
  - Then, modulate the texture color with the fragment color
  - If the texture contains transparency, multiply your texture color with your opacity value (alpha)

# Texture Mapping

## Mapping a Texture

- Based on parametric texture coordinates specified at each vertex

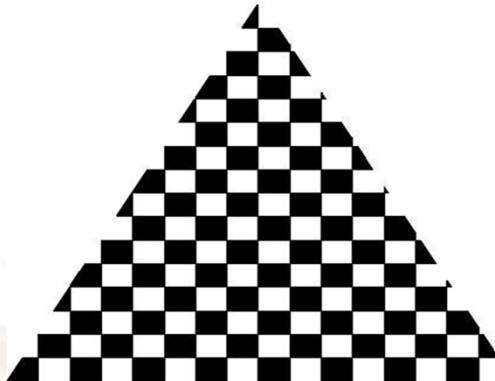


# Texture Mapping

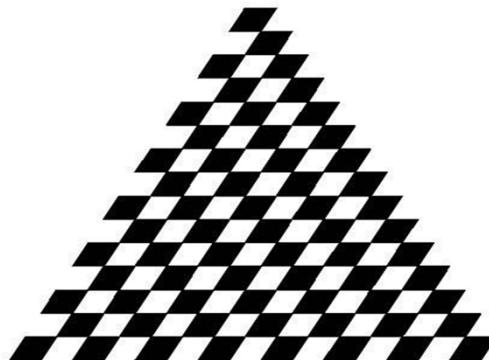
## Interpolation

- OpenGL uses interpolation to find proper texels from specified texture coordinates
- There can be **distortions**

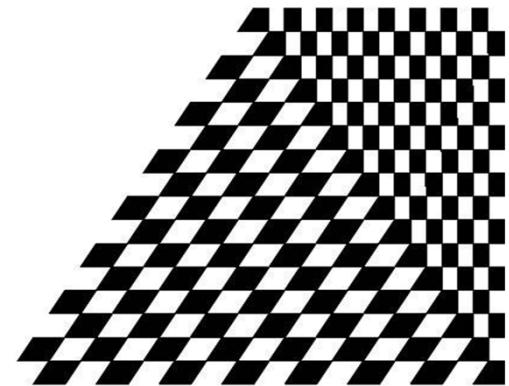
Good selection  
of tex coordinates



Poor selection  
of tex coordinates



texture stretched  
over trapezoid  
showing effects of  
bilinear interpolation



# Texture Mapping

## OpenGL implementation: Initialization

- **Load Textures (Initialization)**
  - Procedural (Generate Arrays with color values)
  - Import Files (BMP, PNG, JPG, etc. ...)
    - Convert to RGB/RGBA format
    - CImg (OpenGL templated class to load images)
- **Create Texture Objects** `GLuint textureID;`  
`glGenTextures(1, &textureID);`  
`glBindTexture(GL_TEXTURE_2D, textureID);`  
`glTexImage2D(textureID,...);`  
    // Set filtering attributes  
`glTexParameteri(..., GL_TEXTURE_MAG_FILTER, ...);`  
`glTexParameteri(..., GL_TEXTURE_MIN_FILTER, ...);`

# Texture Mapping

## OpenGL implementation: Shaders

- Create shaders supporting texturing
  - Vertex shader: UV coordinates inputs  
`in vec2 UV;`  
`out vec2 frag_UV;`
  - Fragment shader: UV + Texture Sampler Inputs:  
`uniform sampler2D textureSampler;`  
`in vec2 frag_UV;`
    - Texturing code  
`col = texture( textureSampler, frag_UV ).rgb + ...;`

Set the variable using:

```
tex_loc = glGetUniformLocation(program_id, "textureSampler");
glUniform1i(tex_loc, textureID);
```

# Texture Mapping

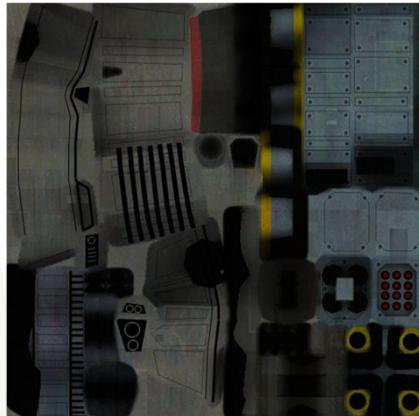
## OpenGL implementation: Application

- Add UV in your vertex data and send them as inputs to your vertex shader
- Draw Textured Models
  - Set Shader to a Textured shader
  - Set Shader Constants
  - Draw Geometry
- Good tutorial here:  
<http://www.opengl-tutorial.org/beginners-tutorials/tutorial-5-a-textured-cube/>

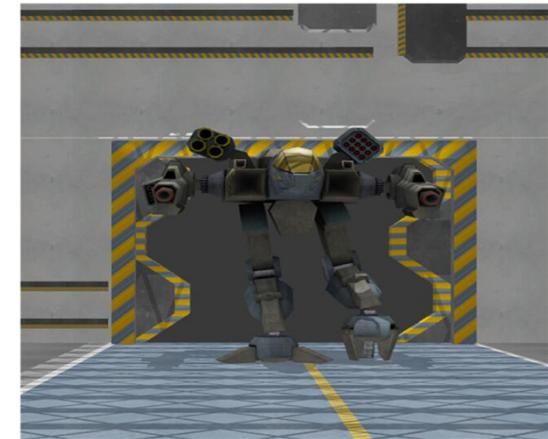
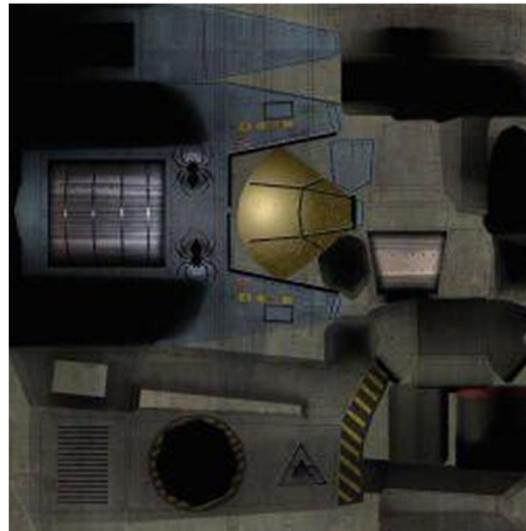
# Texture Mapping

## More Texture Stuff

- **Texture must be in fast memory - it is accessed for every pixel drawn**
  - If you exceed it, performance will degrade horribly
  - Skilled artists can pack textures for different objects into one image

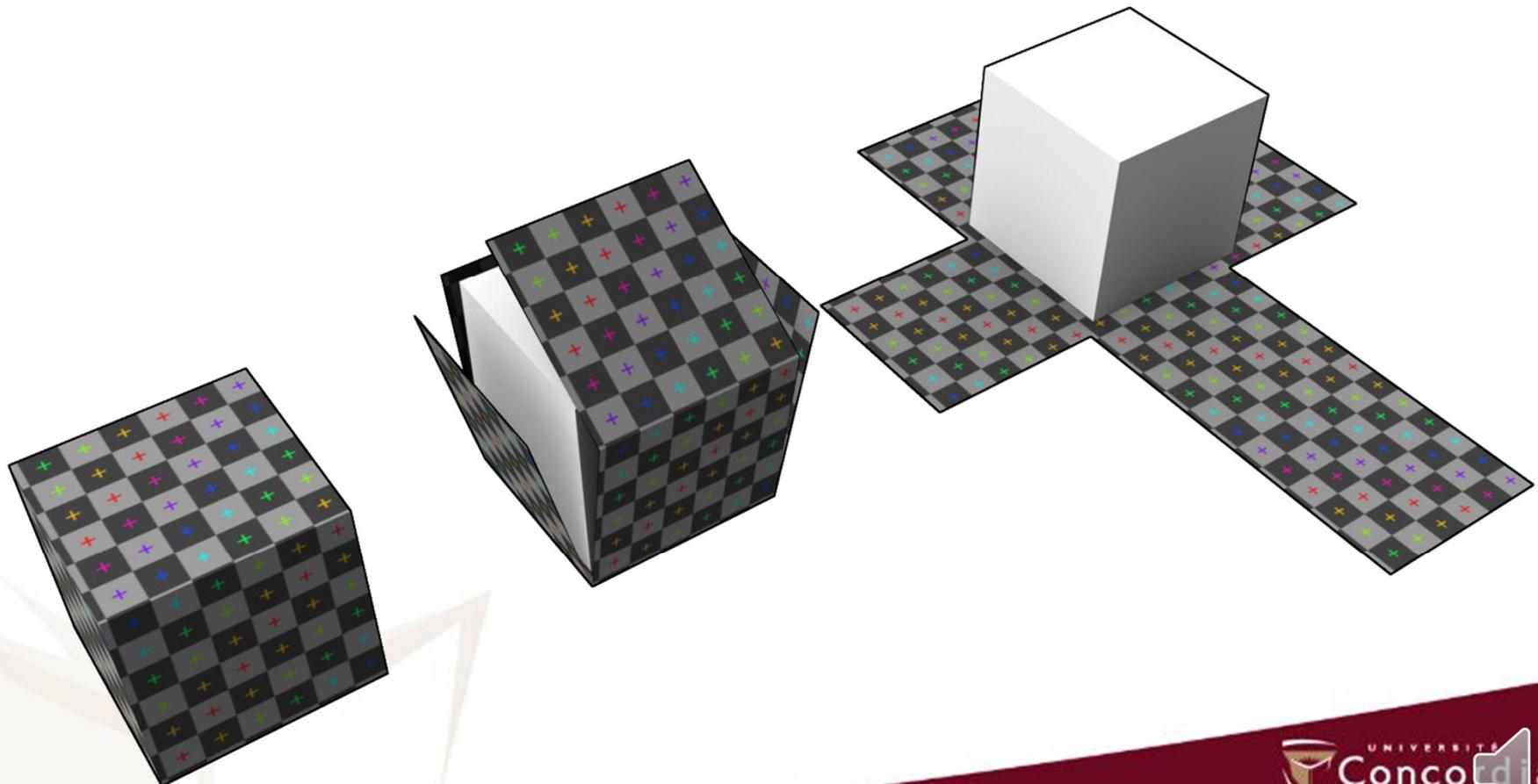


Specialized  
textures



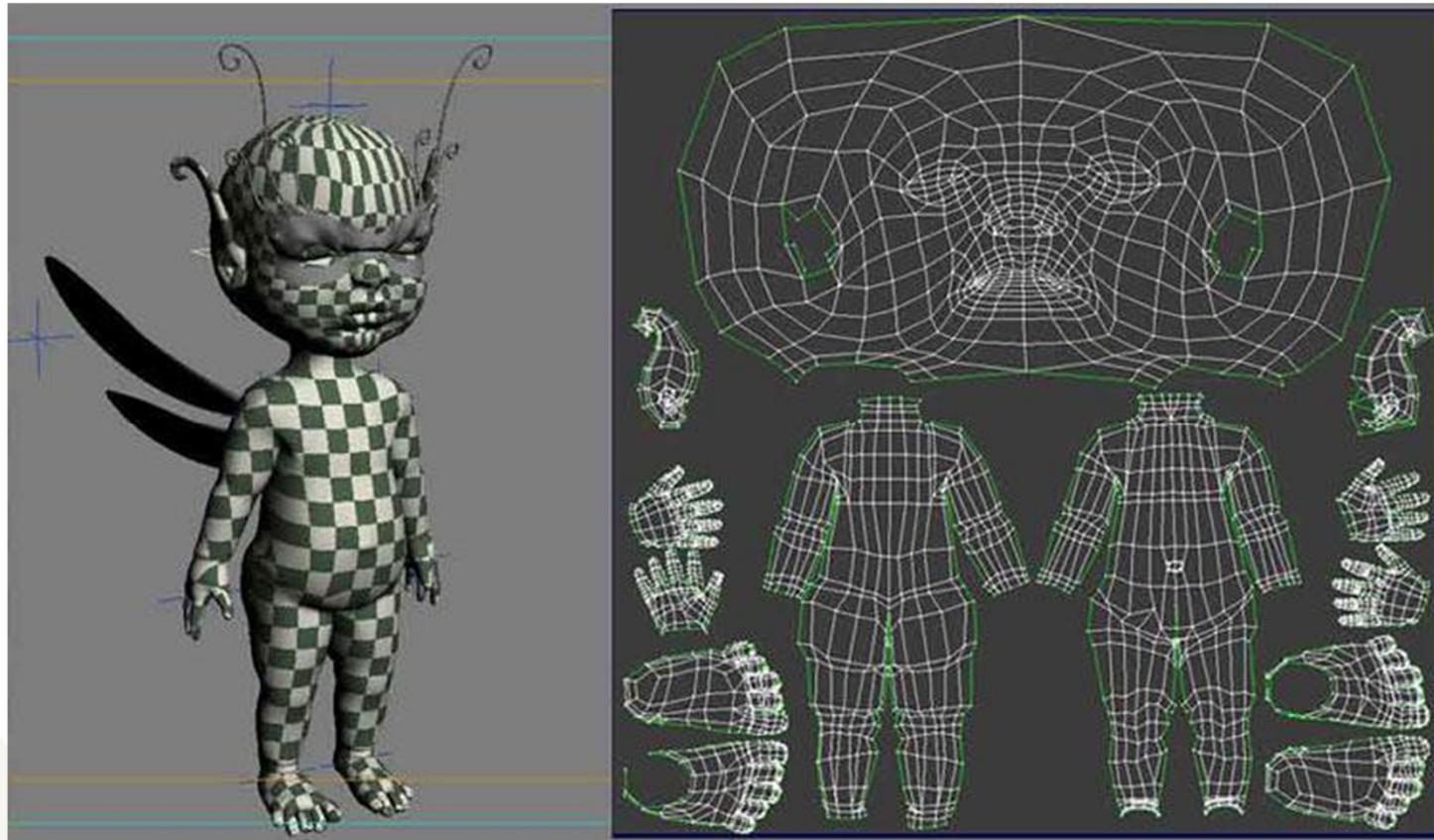
# Texture Mapping

## More Texture Stuff



# Texture Mapping

## More Texture Stuff



# Texture Mapping

## More Texture Stuff



© www.3D.sk

# Texture Mapping

## More Texture Stuff

- There are “image processing” operations that can be applied to the pixels coming out of the texture
- Sometimes you want to apply multiple textures to the same point:
- Multitexturing is now available with most new hardware
- There are 1D and 3D textures
  - Mapping works essentially the same
  - 3D textures are very memory intensive, and how they are used is very application dependent
  - 1D saves memory if the texture is inherently 1D, like stripes

# Texture Mapping

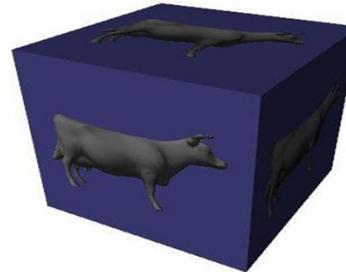
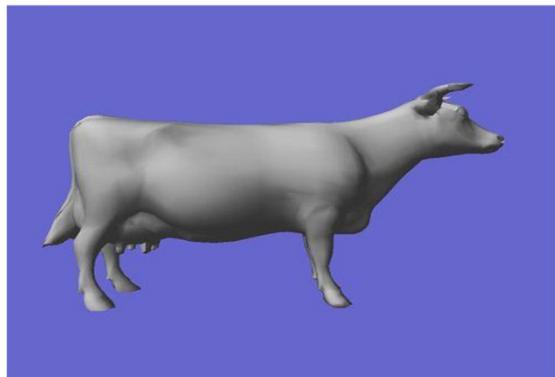
## Render To Texture

- Instead of rendering a scene on the Frame buffer, render the scene on a Texture that can be reused later
- You may need a Z-buffer for rendering into the texture
- Steps:
  - Create Texture in which we render later
  - Render Scene in Texture
  - Bind the generated texture while drawing on the Framebuffer
- Reference

<http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-14-render-to-texture/>

# Texture Mapping

## Render To Texture



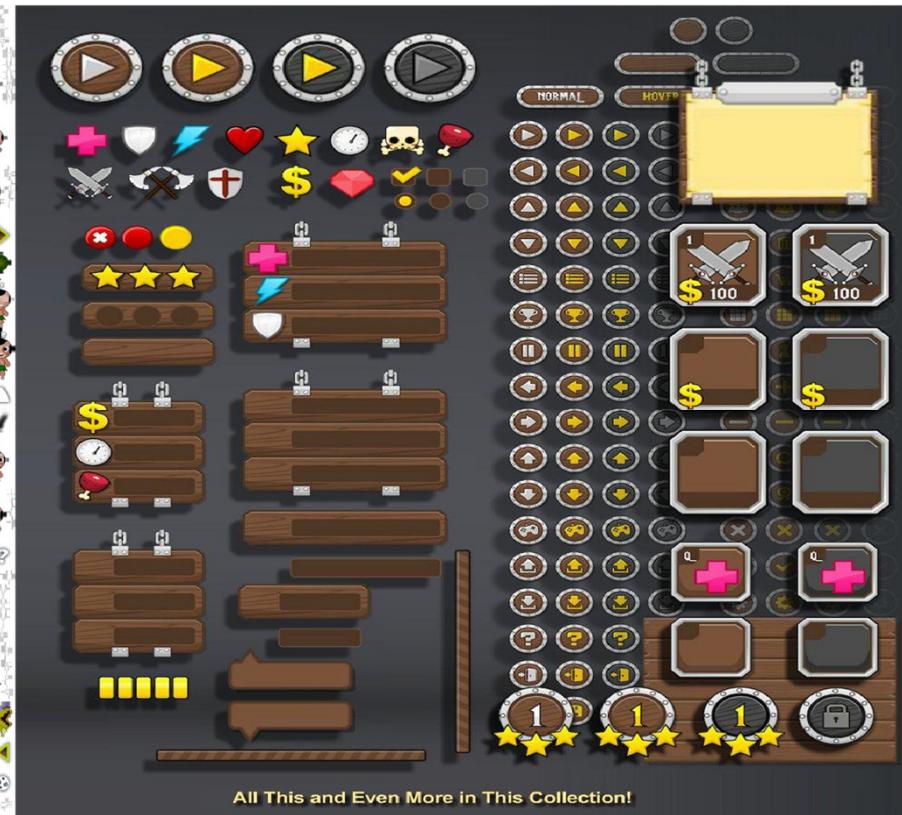
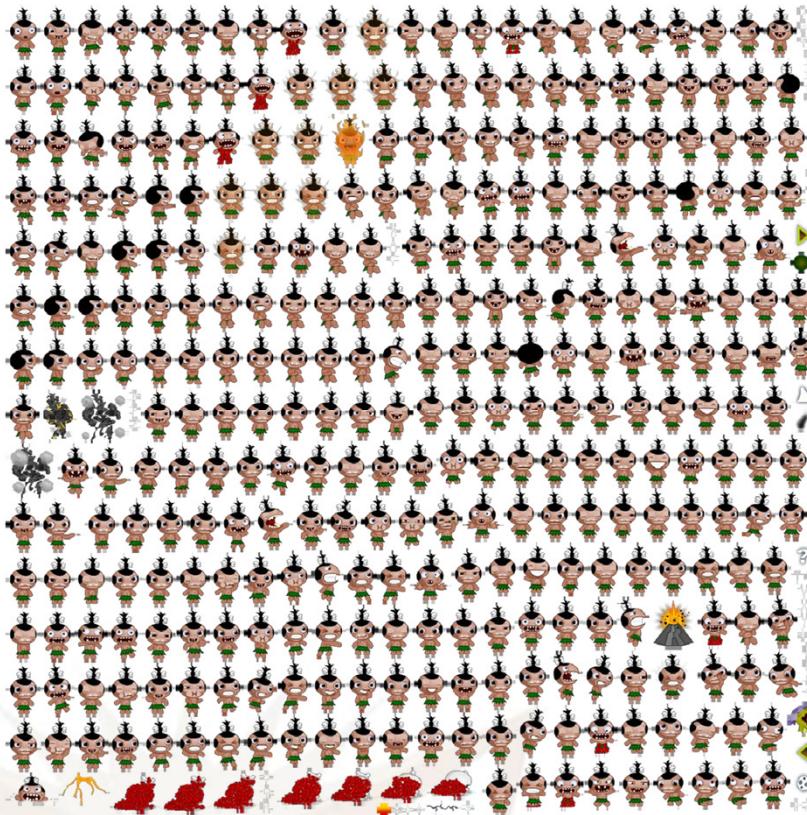
# Texture Mapping

## Texture Atlas

- **Changing the current texture in the Graphics Pipeline is expensive**
- **It is recommended to minimize these context switches which would affect the frame rate**
- **Using texture atlases is a smart way to make many textures available within a single texture**

# Texture Mapping

## Texture Atlas

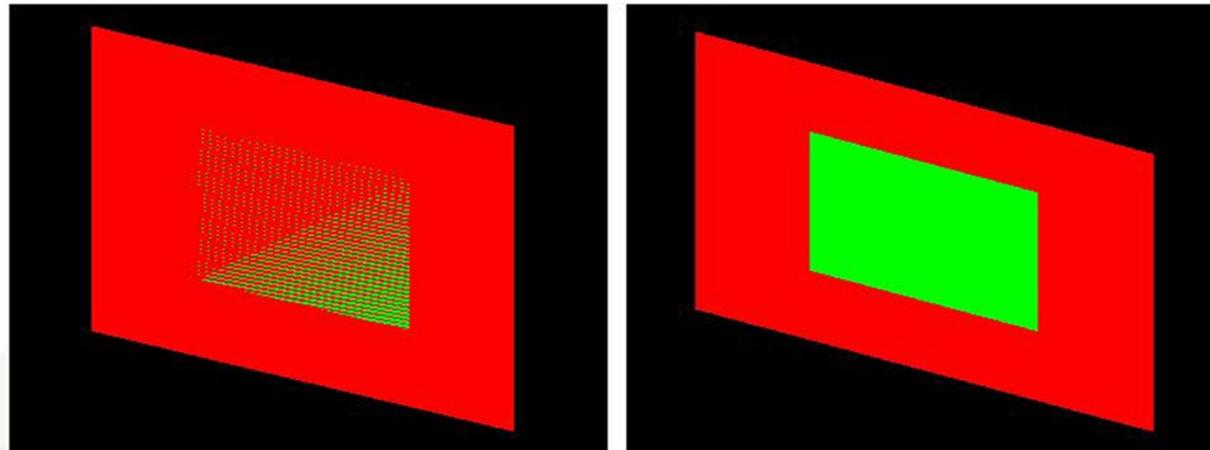


All This and Even More in This Collection!

# Texture Mapping

## Rendering Decals

- Decals are polygons rendered slightly above the geometry
- Since Z-Buffer precision degrades with distance, a constant offset leads to Z-Fighting



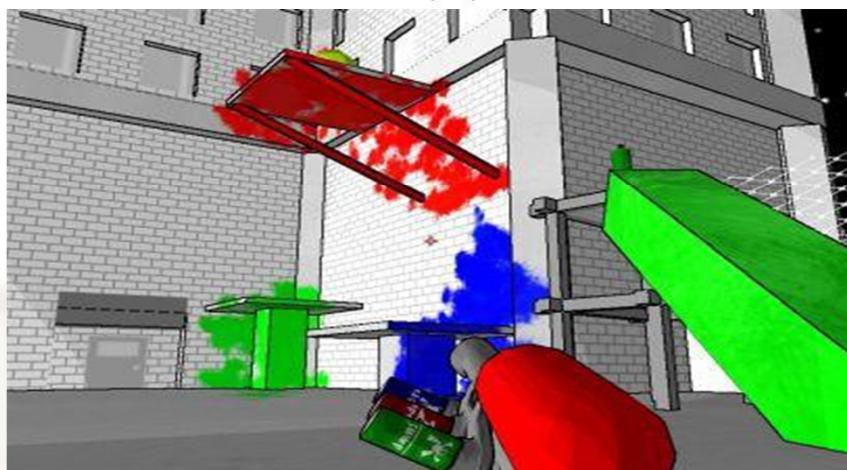
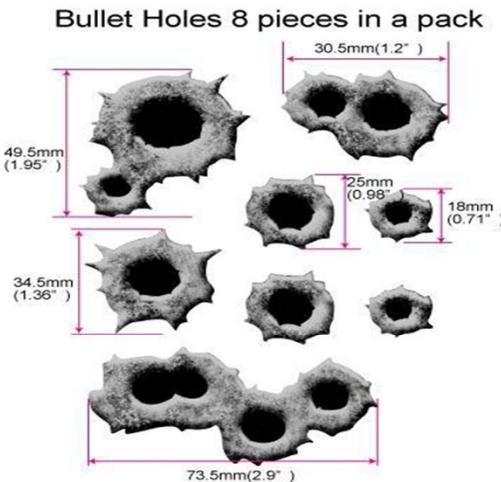
# Texture Mapping

## Rendering Decals with Depth Bias

- To properly render decals, the offset needs to increase with distance to prevent Z-fighting
- The best solution is that the offset value be implemented within the projection matrix
- Another solution is to use **glPolygonOffset** functionality, which is platform dependent
  - Slow/Inconsistent between platforms

# Texture Mapping

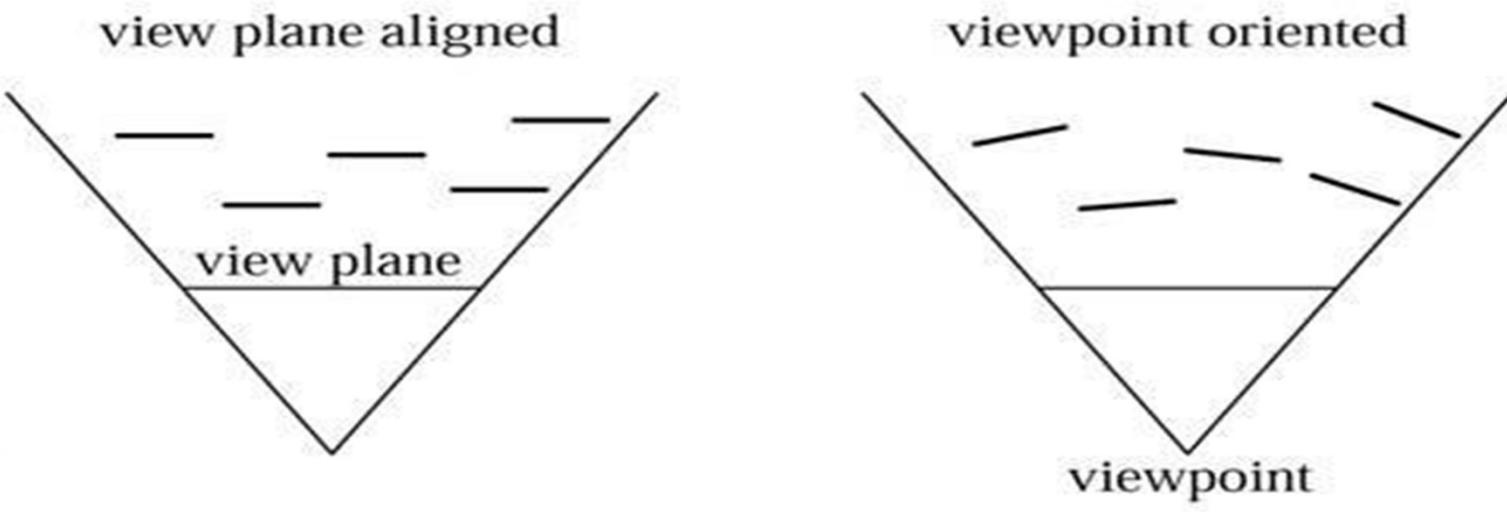
## Decals Examples



# Texture Mapping

## Billboards

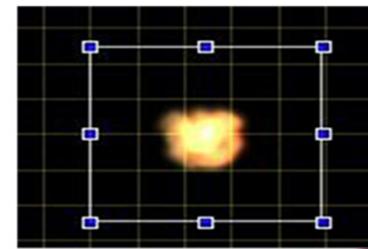
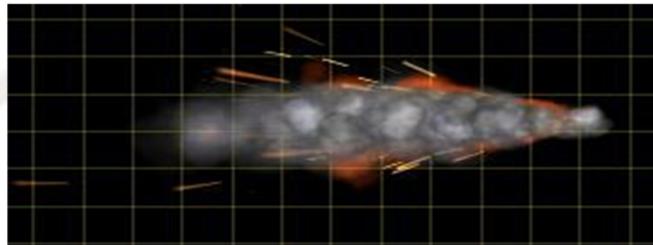
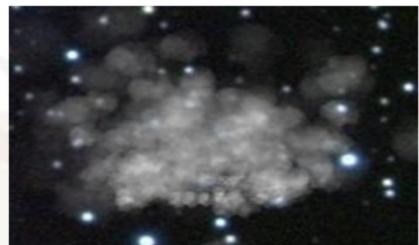
- Billboards are geometry that automatically align with the Camera
- Useful for Particle Systems, UI elements, etc.



# Texture Mapping

## Particle Effects

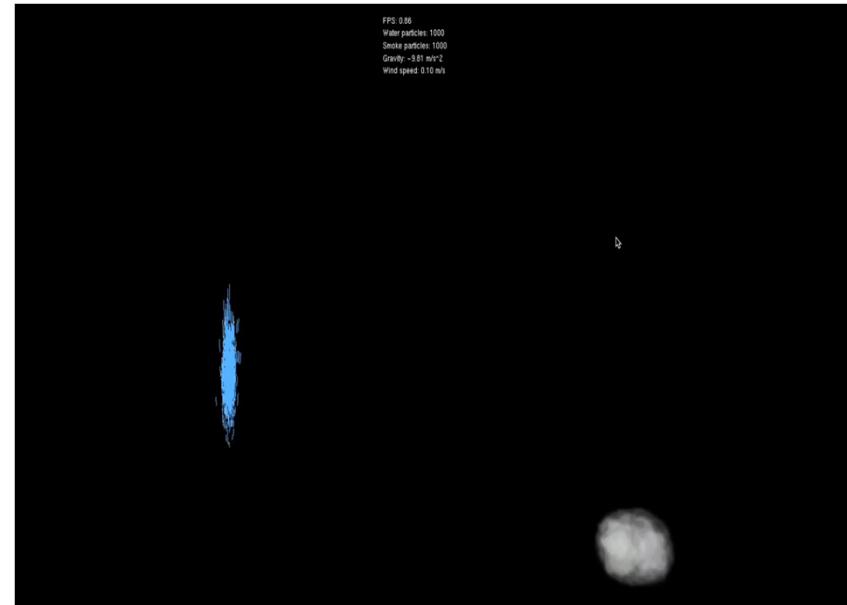
- The term particle system refers to a computer graphics technique to *simulate certain fuzzy phenomena*, which are otherwise very hard to reproduce with conventional rendering techniques. Examples of such phenomena which are commonly replicated using particle systems include fire, explosions, smoke, flowing water, sparks, falling leaves, clouds, fog, snow, dust, meteor tails, hair, fur, grass, or abstract visual effects like glowing trails, magic spells, etc.



# Texture Mapping

## Particle Systems

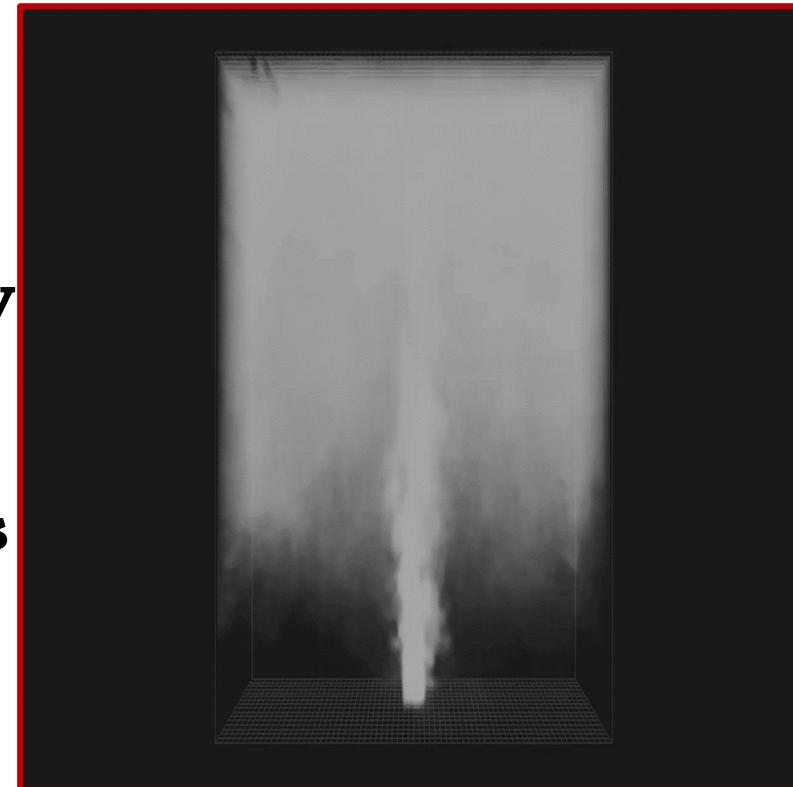
- **A particle has:**
  - A position in the world
  - Rules for how it moves over time
  - Rules for how it is drawn
- **A particle system:**
  - Controls when particles are created and destroyed
  - Makes sure that all the particles are updated



# Texture Mapping

## Smoke Particle System

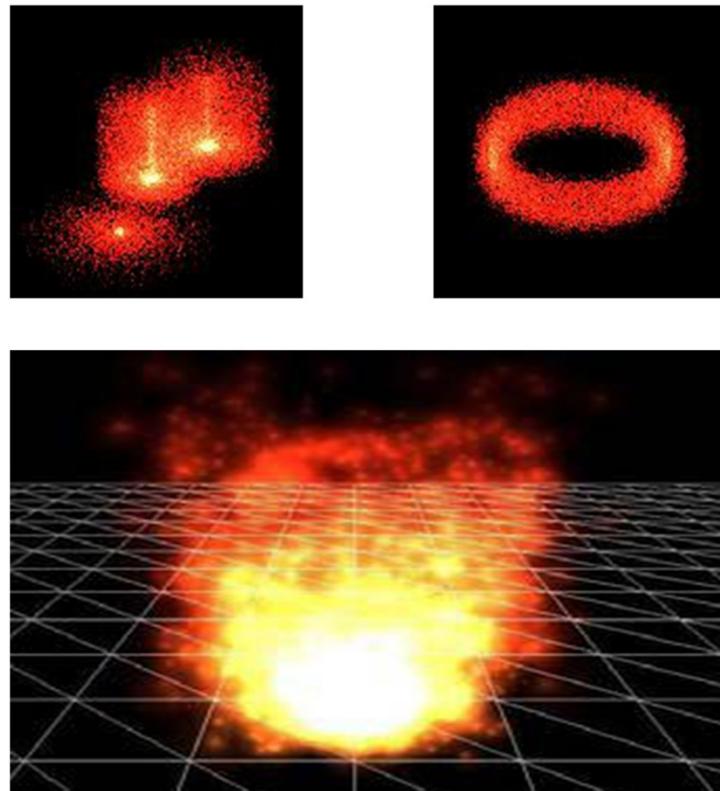
- Constantly create particles
- Particles move upwards, with turbulence added
  - Ken Perlin: Another academy award
- Render the particles as partially transparent circles that fade over time
- Particle systems are the standard way of doing smoke and water spray



# Texture Mapping

## Particle Systems: Steps

- 1. Create new particles and initialize their states**
  - Position, velocity, etc.
  
- 2. Delete “expired” particles**
  
- 3. Update particle states based on physical forces**
  
- 4. Render particles**



# Texture Mapping

## Particle Creation and Deletion

### ■ Creation: Where?

- Fixed source - *e.g.*, fountain, top of waterfall
- Low density areas (to ensure adequate coverage)
- On object surface
- Along user-designated curve

### ■ Deletion: Why?

- Fixed lifetime
- Left bounding box
- Local density over threshold (*e.g.*, have enough already for realistic effect)

# Texture Mapping

## Particle Update

- Given particle state at time  $t$  consisting of position, velocity, etc., how do we compute new values at time  $x(t + \Delta t)$  ?
- Typically, we don't have an explicit parametric function  $x(t)$  that we can just evaluate for any  $t$ . e.g., a spline curve
- Rather, we have a set of forces and an initial value for the particle state
- We have to simulate the action of the forces on the particle to “see what happens”!