# Chapter five OpenGL Part I

- Introduction
- Languages and Libraires
- OpenGL Requirements
- Summery



# Introduction

- Graphics programming has a reputation for being among the most challenging computer science topics to learn.
- These days, graphics programming is shader based—that is, some of the program is written in a standard language such as C++ or Java for running on the CPU and some is written in a special-purpose shader language for running directly on the graphics card (GPU).

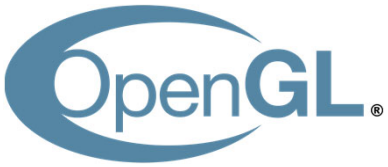
## Introduction-cont.

- Shader programming has a steep learning curve, so that even drawing something simple requires a convoluted set of steps to pass graphics data down a “pipeline.”
- Modern graphics cards are able to process this data in parallel, and so the graphics programmer must understand the parallel architecture of the GPU, even when drawing simple shapes.



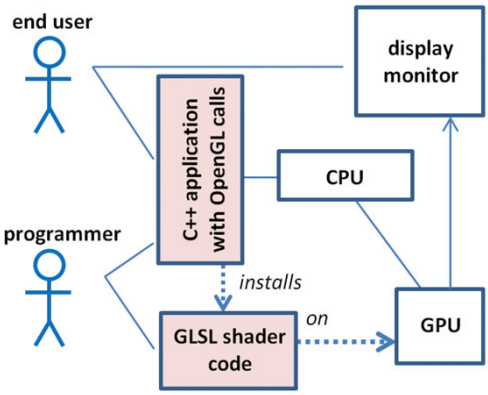

## Languages and Libraires

- Modern graphics programming is done using a graphics library. That is, the programmer writes code which invokes functions in a predefined library (or set of libraries) that provide support for lower-level graphical operations.
- There are many graphics libraries in use today, but the most common library for platform independent graphics programming is called OpenGL (Open Graphics Library).




# Languages and Libraires-cont.

- OpenGL (Open Graphics Library) is a cross-language, cross-platform application programming interface (API) for rendering 2D and 3D vector graphics.
- The API is typically used to interact with a graphics processing unit (GPU), to achieve hardware-accelerated rendering.
- There are many graphics libraries in use today, but the most common library for platform independent graphics programming is called OpenGL (Open Graphics Library).




92

# OpenGL Requirements

- Using OpenGL with C++ requires configuring several libraries. We will need languages and libraries for the following functions:
  - C++ development environment
  - OpenGL / GLSL
  - Window management
  - Extension library
  - Math library
  - Texture management

93

# C++ development environment

- C++ is a general-purpose programming language that first appeared in the mid-1980s.
- Its design, and the fact that it is generally compiled to native machine code, make it an excellent choice for systems that require high performance, such as 3D graphics computing.
- Another advantage of C++ is that the OpenGL call library is C based. Many C++ development environments are available.
- We recommend using Microsoft Visual Studio if using a PC, and Xcode if using a Macintosh



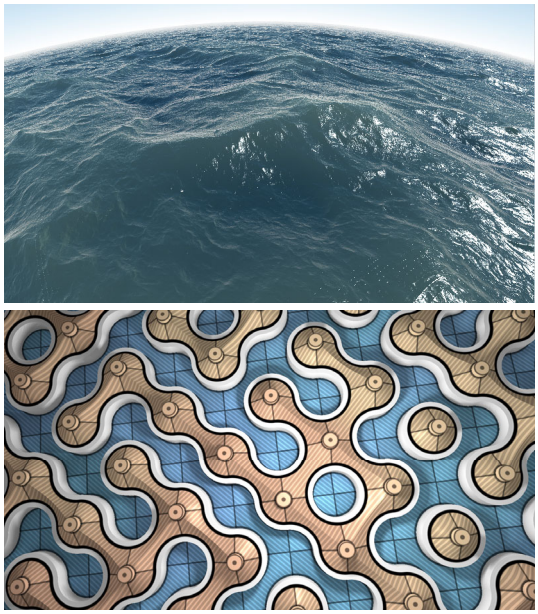
# OpenGL / GLSL

- Version 1.0 of OpenGL appeared in 1992 as an “open” alternative to vendor specific Application Programming Interfaces (APIs) for computer graphics.
- Its specification and development was managed and controlled by the OpenGL Architecture Review Board (ARB), a then newly formed group of industry participants.
- In 2006 the ARB transferred control of the OpenGL specification to the Khronos Group, a nonprofit consortium which manages not only the OpenGL specification but a wide variety of other open industry standards.

KHRONOS® GROUP

# OpenGL / GLSL-cont.

- In 2004, version 2.0 introduced the OpenGL Shading Language (GLSL), allowing “shader programs” to be installed and run directly in graphics pipeline stages.
- In 2009, version 3.1 removed a large number of features that had been deprecated, to enforce the use of shader programming as opposed to earlier approaches (referred to as “immediate mode”). Among the more recent features, version 4.0 (in 2010) added a tessellation stage to the programmable pipeline.



# Window management

- OpenGL doesn’t actually draw to a computer screen. Rather, it renders to a frame buffer, and it is the job of the individual machine to then draw the contents of the frame buffer onto a window on the screen.
- There are various libraries that support doing this. One option is to use the windowing capabilities provided by the operating system, such as the **Microsoft Windows API**. This is generally impractical and requires a lot of low-level coding.
- **GLUT** is a historically popular option; however, it is deprecated. A modernized extension is freeglut. Other related options are **CPW**, **GLOW**, and **GLUI**. But we will use the latest **GLFW**.

# Extension Library

- OpenGL is organized around a set of base functions and an extension mechanism used to support new functionality as technologies advance.
- Modern versions of OpenGL, such as those found in version 4+, require identifying the extensions available on the GPU.
- There are commands built into core OpenGL for doing this, but they involve several rather convoluted lines of code that would need to be performed for each modern command used—and in this book we use such commands constantly.
- Therefore, it has become standard practice to use an extension library to take care of these details, and to make modern OpenGL commands available to the programmer directly. Examples are **Glee**, **GLLoader**, **GLEW**, and more recently **GL3W** and **GLAD**.



# Math Library

- 3D graphics programming makes heavy use of vector and matrix algebra. For this reason, use of OpenGL is greatly facilitated by accompanying it with a function library or class package to support common mathematical tasks.
- Arguably the most popular, and the one used in this book, is OpenGL Mathematics, usually called **GLM**.
- **GLM** provides classes and basic math functions related to graphics concepts, such as vector, matrix, and quaternion. It also contains a variety of utility classes for creating and using common 3D graphics structures, such as perspective and look-at matrices.


## Texture Management

- Using image files to add “texture” to the objects in a graphics scenes, means that we will frequently need to load such image files into our C++/OpenGL code.
- It is possible to code a texture image loader from scratch; however, given the wide variety of image file formats, it is generally preferable to use a texture loading library.
- Some examples are **FreeImage**, **DevIL**, OpenGL Image (**GLI**), and **Glraw**. Probably the most commonly used OpenGL image loading library is Simple OpenGL Image Loader (**SOIL**), although it has become somewhat outdated.



## Summery

- OpenGL is cross-platform API
- GLSL is the shading language for OpenGL
- Khronos Group, a nonprofit consortium which the OpenGL
- **GLFW** is an Open Source, multi-platform library for OpenGL, OpenGL ES and Vulkan development on the desktop.
- The OpenGL Extension Wrangler Library (**GLEW**) is a cross-platform open-source C/C++ extension loading library
- OpenGL Mathematics (**GLM**) is a header only C++ mathematics library for graphics software based on the OpenGL Shading Language (**GLSL**) specifications.

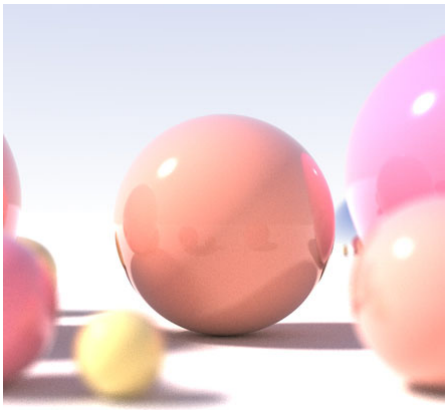
# Chapter 6 Ray Tracing

- Introduction
- History
- Another Definition
- The Basic Ray-Tracing Algorithm
- Perspective
- Computing Viewing Rays
- Parallel vs Perspective
- Ray - Object Intersection



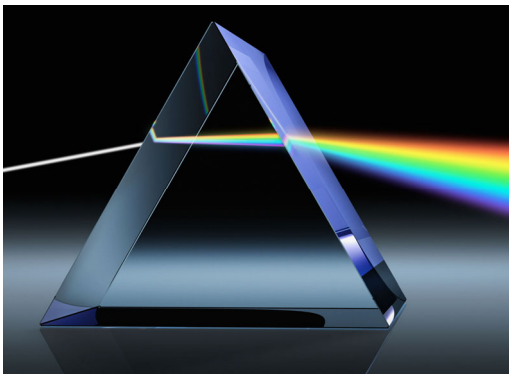
# Introduction

- In 3D computer graphics, ray tracing is a rendering technique for generating an image by tracing the path of light as pixels in an image plane and simulating the effects of its encounters with virtual objects.
- The technique is capable of producing a high degree of visual realism, more so than typical scanline rendering methods but at a greater computational cost.



# Introduction-cont.

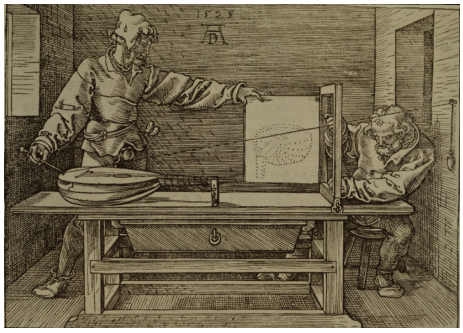
- Ray tracing is capable of simulating a variety of optical effects, such as reflection and refraction, scattering, and dispersion phenomena (such as chromatic aberration).
- It can also be used to trace the path of sound waves in a similar fashion to light waves, making it a viable option for more immersive sound design in video games by rendering realistic reverberation and echoes
- In fact, any physical wave or particle phenomenon with approximately linear motion can be simulated with these techniques.





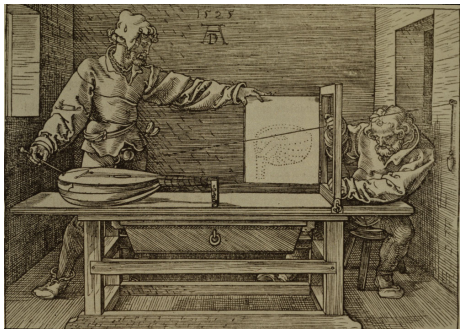
# History

- The idea of ray tracing comes from as early as the 16th century when it was described by Albrecht Dürer, who is credited for its invention.
- He described an apparatus called a Dürer's door using a thread attached to the end of a stylus that an assistant moves along the contours of the object to draw.
- The thread passes through the door's frame and then through a hook on the wall. The thread forms a ray and the hook acts as the center of projection and corresponds to the camera position in raytracing.



# History-cont.

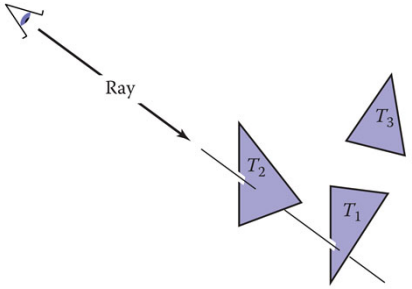
- The idea of ray tracing comes from as early as the 16th century when it was described by Albrecht Dürer, who is credited for its invention.
- He described an apparatus called a Dürer's door using a thread attached to the end of a stylus that an assistant moves along the contours of the object to draw.
- The thread passes through the door's frame and then through a hook on the wall. The thread forms a ray and the hook acts as the center of projection and corresponds to the camera position in raytracing.





# Another Definition

- Ray tracing is an image-order algorithm for making renderings of 3D scenes, and we'll consider it first because it's possible to get a ray tracer working without developing any of the mathematical machinery that's used for object-order rendering.



# The Basic Ray-Tracing Algorithm

- A basic ray tracer therefore has three parts:
  1. ray generation, which computes the origin and direction of each pixel's viewing ray based on the camera geometry;
  2. ray intersection, which finds the closest object intersecting the viewing ray;
  3. shading, which computes the pixel color based on the results of ray intersection.



# The Basic Ray-Tracing Algorithm-cont.

- for each pixel do compute viewing ray
  - find first object hit by ray and its surface normal n
  - set pixel color to value computed from hit point, light, and n

# Perspective

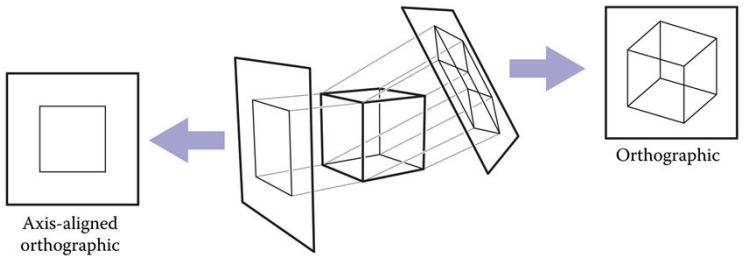
- The problem of representing a 3D object or scene with a 2D drawing or painting was studied by artists hundreds of years before computers. Photographs also represent 3D scenes with 2D images.
- While there are many unconventional ways to make images, from cubist painting to fisheye lenses (Figure A) to peripheral cameras, the standard approach for both art and photography, as well as computer graphics, is linear perspective, in which 3D objects are projected onto an image plane in such a way that straight lines in the scene become straight lines in the image.

(Figure A)



# Perspective-cont.

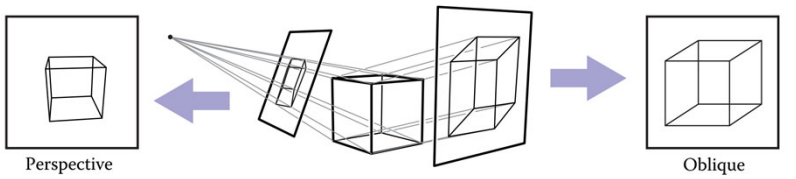
- The simplest type of projection is parallel projection, in which 3D points are mapped to 2D by moving them along a projection direction until they hit the image plane (Figures B)




(Figure B)

# Perspective-cont.

- The advantages of parallel projection are also its limitations. In our everyday experience (and even more so in photographs) objects look smaller as they get farther away, and as a result parallel lines receding into the distance do not appear parallel. This is because eyes and cameras don't collect light from a single viewing direction; they collect light that passes through a particular viewpoint.



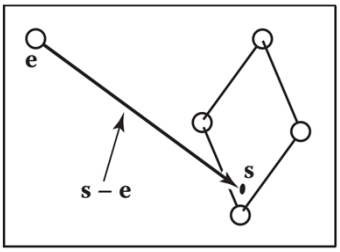

(Figure B)



# Computing Viewing Rays

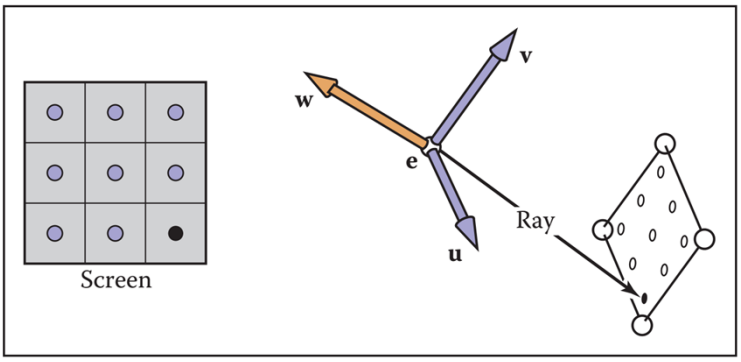
- In order to generate rays, we first need a mathematical representation for a ray.
- A **ray** is really just an **origin** point and a propagation **direction**; a 3D parametric line is ideal for this. the 3D parametric line from the eye **e** to a point **s** on the image plane is given by

$$\mathbf{p}(t) = \mathbf{e} + t(\mathbf{s} - \mathbf{e})$$



# Computing Viewing Rays-cont.

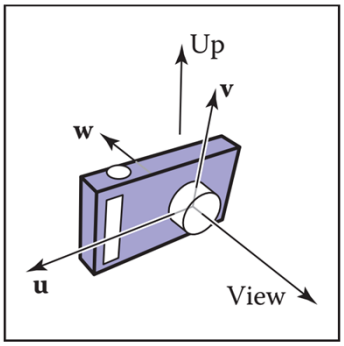
- The sample points on the screen are mapped to a similar array on the 3D window. A viewing ray is sent to each of these locations.





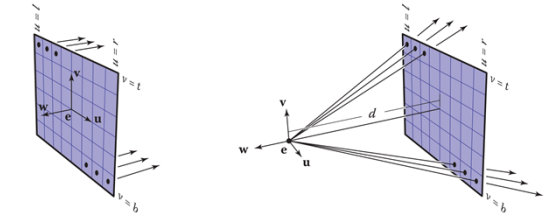

# Computing Viewing Rays-cont.

- The vectors of the camera frame, together with the view direction and up direction. The **w** vector is opposite the view direction, and the **v** vector is coplanar with **w** and the **up vector**.



# Parallel vs Perspective

- Ray generation using the camera frame.
  - **Left**: In an orthographic view, the rays start at the pixels' locations on the image plane, and all share the same direction, which is equal to the view direction.
  - **Right**: In a perspective view, the rays start at the viewpoint, and each ray's direction is defined by the line through the viewpoint, e, and the pixel's location on the image plane.



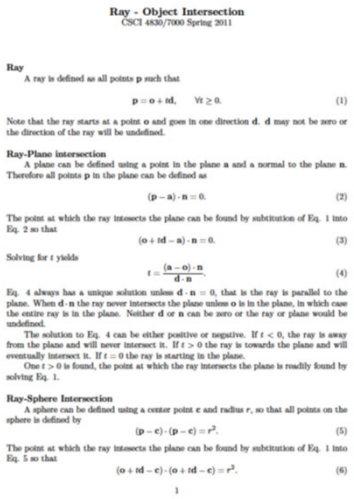

**Parallel projection**
same direction, different origins

**Perspective projection**
same origin, different directions



# Ray - Object Intersection

Ray - Object Intersection. CSCI 4830/7000 Spring 2011



**Ray - Object Intersection**
CSCI 4830/7000 Spring 2011

**Ray**
A ray is defined as all points $\mathbf{p}$ such that

$$\mathbf{p} = \mathbf{o} + t\mathbf{d}, \quad \forall t \geq 0. \tag{1}$$

Note that the ray starts at a point $\mathbf{o}$ and goes in one direction $\mathbf{d}$. $\mathbf{d}$ may not be zero or the direction of the ray will be undefined.

**Ray-Plane intersection**
A plane can be defined using a point in the plane $\mathbf{a}$ and a normal to the plane $\mathbf{n}$. Therefore all points $\mathbf{p}$ in the plane can be defined as

$$(\mathbf{p} - \mathbf{a}) \cdot \mathbf{n} = 0. \tag{2}$$

The point at which the ray intesects the plane can be found by subtitution of Eq. 1 into Eq. 2 so that

$$(\mathbf{o} + t\mathbf{d} - \mathbf{a}) \cdot \mathbf{n} = 0. \tag{3}$$

Solving for $t$ yields

$$t = \frac{(\mathbf{a} - \mathbf{o}) \cdot \mathbf{n}}{\mathbf{d} \cdot \mathbf{n}}. \tag{4}$$

Eq. 4 always has a unique solution unless $\mathbf{d} \cdot \mathbf{n} = 0$, that is the ray is parallel to the plane. When $\mathbf{d} \cdot \mathbf{n}$ the ray never intersects the plane unless $\mathbf{o}$ is in the plane, in which case the entire ray is in the plane. Neither $\mathbf{d}$ or $\mathbf{n}$ can be zero or the ray or plane would be undefined.

The solution to Eq. 4 can be either positive or negative. If $t < 0$, the ray is away from the plane and will never intersect it. If $t > 0$ the ray is towards the plane and will eventually intersect it. If $t = 0$ the ray is starting in the plane.

One $t > 0$ is found, the point at which the ray intersects the plane is readily found by solving Eq. 1.

**Ray-Sphere Intersection**
A sphere can be defined using a center point $\mathbf{c}$ and radius $r$, so that all points on the sphere is defined by

$$(\mathbf{p} - \mathbf{c}) \cdot (\mathbf{p} - \mathbf{c}) = r^2. \tag{5}$$

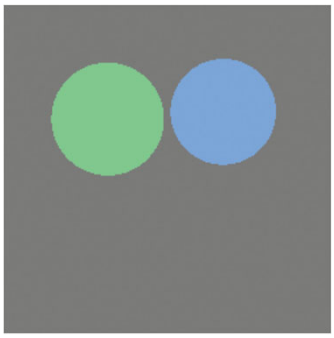
The point at which the ray intesects the plane can be found by subtitution of Eq. 1 into Eq. 5 so that

$$(\mathbf{o} + t\mathbf{d} - \mathbf{c}) \cdot (\mathbf{o} + t\mathbf{d} - \mathbf{c}) = r^2. \tag{6}$$

1

# Ray Tracing in Action

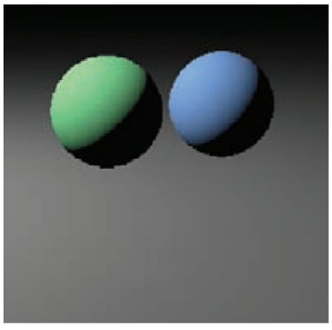
- A simple scene rendered with only ray generation and surface intersection, but no shading; each pixel is just set to a fixed color depending on which object it hit.





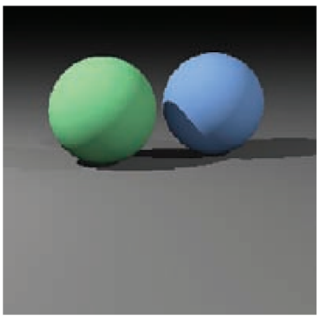
# Ray Tracing in Action-cont.

- A simple scene rendered with diffuse shading from a single light source.



## Ray Tracing in Action-cont.

- A simple scene rendered with diffuse shading and shadows from three light sources.





## Ray Tracing in Action-cont.

- A simple scene rendered with diffuse shading (blue sphere), Blinn-Phong shading (green sphere), and shadows from three light sources.