Computer Graphics

CSE167: Computer Graphics
Instructor: Steve Rotenberg
UCSD, Fall 2005
CSE167

- Computer Graphics
- Instructor: Steve Rotenberg (steve@graphics.ucsd.edu)
- TAs:
  - Wojciech Jarosz (wjarosz@ucsd.edu)
  - Alex Kozlowski (kman@graphics.ucsd.edu)
- Lecture: Center Hall 105 (TTh 11:00am-12:20pm)
- Section: TBD
- Office: CS 4112, (TTh 9:45-10:45am)
- Web page:
  - http://graphics.ucsd.edu/courses/cse167_f05/index.html
Prerequisites

- Linear algebra
- Familiarity with:
  - Vectors (dot products, cross products…)
  - Matrices
  - C++ or Java
  - Object oriented programming
Reading

- 3D Computer Graphics: A Mathematical Introduction with OpenGL (Buss)
- Required pages are listed on the class web page
Programming Projects

- NOTE: Details of this may change. Check the web page for updates
- Project 1: Due Thursday, Oct. 6, 11:00am
  - Make a program that draws some simple 3D objects
- Project 2: Due Thursday, Oct. 20, 11:00am
  - Implement a simple hierarchical object, like a hand. It should include basic lighting as well.
- Project 3: Due Thursday, Nov. 3, 11:00am
  - Make a program that renders a textured scene
- Project 4: Due Thursday, Nov. 17, 11:00am
  - Display a curved surface with texture and lighting
- Project 5: Due Thursday, Dec. 1, 11:00am
  - Choose one of the following
    - Water fountain (particle system)
    - Ray tracer
    - Procedural tree
    - Choose your own project (but talk to me first)
Grading

- 10% Project 1
- 10% Project 2
- 10% Project 3
- 10% Project 4
- 15% Project 5
- 20% Midterm
- 25% Final
Tests

- **Midterm**
  - Thursday, Oct 27, 11:00am-12:20pm
  - Center Hall 105

- **Final**
  - Tuesday, Dec 6, 11:30am – 2:30pm
  - Location: TBD
Undergraduate Computer Graphics at UCSD

- CSE 166: Image Processing
- CSE 167: Computer Graphics
- CSE 168: Rendering Algorithms
- CSE 169: Computer Animation
- Math 155B: Mathematics for Computer Graphics
Course Outline

1. Introduction
2. Matrices, 3D transforms
3. Viewing, perspective
4. Triangle rendering
5. Scene graphs, clipping
6. Lighting
7. Texture mapping
8. Aliasing, sampling
9. Advanced texturing
10. Shadows

1. Midterm
2. Color
3. Scene management
4. Curves & surfaces
5. Graphics hardware
6. Ray tracing
7. Global illumination
8. Procedural modeling
9. Animation
10. Review
Computer Graphics
Applications

- Movie, TV special effects
- Video games
- Scientific visualization
- Medical visualization
- Industrial design
- Simulation
- Communication
- Etc.
Image Processing

- Some computer graphics operations involve manipulating 2D images (bitmaps)
- *Image processing* applies directly to the pixel grid and includes operations such as color correction, scaling, blurring, sharpening, etc.
- Common example include digital photo processing and digital ‘painting’ programs (Adobe Photoshop…)
Image Synthesis

- *Image synthesis* or *image generation* refers more to the construction of images from scratch, rather than processing of existing images.

- Synthesis of a 2D image from a 3D scene description is more commonly called *rendering*.
Photoreal Rendering

- *Photoreal rendering* refers to rendering a 3D scene in a realistic way.
- Modern photoreal rendering algorithms are essentially a physically based simulation of light propagation and scattering throughout a 3D environment.
- In a sense, this means that there is a ‘correct’ image that should be generated, given an input data set. This allows the subject of photoreal rendering to have a strong theoretical basis (namely, the science of optics).
- Most modern photoreal rendering algorithms are based on the classic *ray tracing* algorithm, that traces the path of individual light rays starting from the eye and working backwards to the light sources.
Non-Photoreal Rendering

- Non-photoreal rendering (NPR) refers to rendering images in other ways…
- Sometimes, this is done to achieve aesthetic goals such as artificial water colors, pencil sketches, paint brushstrokes…
- Other times, the goal is to maximize the communication of visual information, as in scientific and medical visualization.
Computer Vision

- Computer vision is sometimes considered as a separate discipline from computer graphics, although they share many things in common.
- A central goal in computer vision is to take a set of 2D images (usually from a video or set of photos) and infer from that a 3D description of what is being viewed.
- This is a very different process than rendering, and is more of a form of artificial intelligence.
Computer Vision
Animation

- An animation is just a sequence of individual images.
- Basically, the subject of computer animation focuses on how things change over time. Usually, this refers to motion, but can also refer to other properties changing over time.
- Physical simulation is a very powerful tool in computer animation and can be used to generate believable animations of rigid objects, deformable objects, gasses, liquids, fracture, particle effects, and even explosions and fire.
- Computer animation also includes a large number of techniques specifically developed to manipulate virtual characters.
Physics Simulation
Character Animation
Modeling

- *Modeling* refers to the techniques involved with creating, scanning, editing, and manipulating 3D geometric data.
- Modeling is often done by a human user with an interactive editing program.
- More complex objects, such as trees, can be constructed with automatic **procedural modeling** algorithms.
- 3D models are also often acquired from real world objects using laser scanning or computer vision techniques.
- Modeling also includes the use of curved surfaces and other higher order primitives, which are often converted into triangles using various **tessellation** algorithms.
- Another important area of modeling includes mesh reconstruction for surface simplification.
- Modeling makes heavy use of **computational geometry**.
Modeling
Computer Graphics

- Rendering
  - Photoreal
  - NPR
- Animation
  - Physics
  - Character animation
- Modeling
  - Computational geometry
  - Procedural modeling
  - Data acquisition
Historical Milestones

1960’s:
- Early theoretical development, mainly limited to research and military
- 1962: Sketchpad (Ivan Sutherland)

1970’s:
- ‘Traditional’ graphics pipeline developed
- Driven by money from military simulation and automotive design industries

1980’s:
- Many important core algorithms developed
- Visual quality improved driven by demands from entertainment (movie) industry
- 1985: Rendering Equation (James Kajiya)

1990’s:
- Advanced algorithms developed as graphics theory matured
- Broader focus on animation, data acquisition, NPR, physics…
- 1995: Photon Mapping (Henrik Jensen)

2000’s:
- Photoreal rendering evolves to the point of being able to render convincing images of arbitrarily complex scenes on consumer hardware
- Merging of computer graphics and computer vision
- Cheap graphics hardware with vast capabilities, driven largely by video game industry
Modern graphics displays are *raster* based

This just means that they display a grid of pixels, where each pixel color can be set independently.

Individual pixels are usually formed from smaller red, green, and blue subpixels. If you look very closely at a TV screen or computer monitor, you will notice the pattern of subpixels.

Older style *vector* displays didn’t display a grid of pixels, but instead drew lines directly with an electron beam.

Raster graphics are also sometimes called *bitmapped graphics*.
Display Technology

- Vector display (cathode ray tube)
- Raster displays
  - CRT (cathode ray tube)
  - LCD (liquid crystal display)
  - TFT (thin film transistor)
  - OLED (organic light emitting diode)
  - Light valve
  - Plasma
  - HDR (high dynamic range: TFT / white LED hybrid)
- Film
- Print
Resolution & Frame Rates

- **Video:**
  - NTSC: 720 x 480 @ 30 Hz (interlaced)
  - PAL: 720 x 576 @ 25 Hz (interlaced)

- **HDTV:**
  - 720p: 1280 x 720 @ 60 Hz
  - 1080i: 1920 x 1080 @ 30 Hz (interlaced)
  - 1080p: 1920 x 1080 @ 60 Hz

- **Film:**
  - 35mm: ~2000 x ~1500 @ 24 Hz
  - 70mm: ~4000 x ~2000 @ 24 Hz
  - IMAX: ~5000 x ~4000 @ 24-48 Hz

- **Note:** Hz (Hertz) = frames per second (fps)
- **Note:** Video standards with an i (such as 1080i) are *interlaced*, while standards with a p (1080p) are *progressive* scan.
Interlacing

- Older video formats (NTSC, PAL) and some HD formats (1080i) use a technique called *interlacing*.
- With this technique, the image is actually displayed twice, once showing the odd *scanlines*, and once showing the even scanlines (slightly offset).
- This is a trick for achieving higher vertical resolution at the expense of frame rate (cuts effective frame rate in half).
- The two different displayed images are called *fields*.
- NTSC video, for example, is 720 x 480 at 30 *frames* per second, but is really 720 x 240 at 60 *fields* per second.
- Interlacing is an important issue to consider when working with video, especially in animation as in TV effects and video games.
- Computer monitors are generally not interlaced.
The framebuffer refers to the memory dedicated to storing the image.

It would generally be a 2D array of pixels, where each pixel stores a color (Note: pixel = picture element).

Color is typically stored as a 24 bit RGB value. This offers 8 bits (256 levels) for red, green, and blue, for a total of 16,777,216 different colors.

Very often, additional data is stored per pixel such as depth (z), or other info.

A framebuffer can just be a block of main memory, but many graphics systems have dedicated framebuffer memory with a direct connection to video scan-out hardware and other special features.
Primitives

- Complex scenes are usually built up from simpler objects
- Objects are built from individual primitives
- The most common and general purpose 3D primitive is the triangle
- Points and lines are also useful primitives
- More complex shapes such as n-sided polygons, spheres, curves, curved surfaces, and fractals can be considered as primitives as well, but often, these are just automatically tessellated into triangles in a pre-rendering phase, and so aren’t true primitives
3D Models

- A basic 3D model might consist of a simple array of triangles
- Each triangle stores 3 vertices
- Each vertex contains an xyz position, and possibly some other information (color, normal...)
3D Models

class Vector3 {
    float x, y, z;
};

class Vertex {
    Vector3 Position;
};

class Triangle {
    Vertex Vert[3];
};

class Model {
    int NumTris;
    Triangle *Tri;
};
In the *traditional graphics pipeline*, each primitive is processed through the following steps:

- Transformation
- Lighting
- Clipping
- Scan conversion
- Pixel processing
Transformation

- The transformation process refers to the linear transformation from 3D space to a 2D viewing space.
- Ultimately, each vertex position must be transformed from its defining object space to the device coordinates (pixel space).
- This often involves a combination of rotations, translations, scales, and perspective transformations.
- We will learn how to represent all of this stuff with 4x4 homogeneous matrices.
Lighting

- Lighting operations are applied to each vertex to compute its color.
- In more advanced rendering, lighting operations are computed per pixel, rather than per vertex.
- A variety of light types can be defined such as point lights, directional lights, spot lights, etc.
- More advanced lighting operations can account for shadows, reflections, translucency, and a wide variety of optical effects.
Clipping

- Some triangles will be completely visible on the screen, while others may be completely out of view.
- Some may intersect the side of the screen and require special handling.
- The camera’s viewable space forms a volume called the view volume. Triangles that intersect the boundary of the view volume must be clipped.
- The related process of culling refers to the determination of which primitives are completely invisible.
- The output of the clipping/culling process is a set of visible triangles that lie within the dimensions of the display device.
Scan Conversion

- The scan conversion (or rasterization) process takes 2D triangles as input and outputs the exact pixels covered by the triangle.
- Per-vertex data, such as color, is \textit{interpolated} across the triangle, so each pixel may have a unique color.
The output of the scan conversion process is a bunch of individual xy pixels, plus additional data per pixel such as interpolated depth (z), color, or other information.

The pixel processing stage includes the operations that take place per pixel to compute the final color that gets rendered into the framebuffer.

Usually, the zbuffer technique is used to make sure that a pixel is rendered only if it is not blocked by an existing surface.

Other processing, such as texturing and transparency operations happen per pixel.

In some systems, the entire lighting process is computed per pixel, instead of per vertex.
Scene Rendering

- With the traditional zbuffered graphics pipeline, triangles can be rendered in any order without affecting the final image.
- Often, complex effects such as transparency, actually do depend on the rendering order, and so may require additional care.
- Still, it makes a nice basic approach, and it’s the approach taken by OpenGL and built into many modern hardware graphics boards.
- There are more advanced rendering algorithms (scan line, ray tracing, etc.) that don’t render triangles one at a time, and require the entire scene to be processed. We will learn about these later in the quarter as well.
OpenGL

- OpenGL is an interactive rendering standard designed for ease of use and portability across a wide range of systems.
- It is generally limited to things that can be done at interactive rates and does not include advanced rendering features such as ray tracing.
- It is based on the traditional graphics pipeline where triangles are processed one at a time.
- We will learn about OpenGL in this class, but the bulk of the class will focus on the underlying theory and isn’t dependent on any particular interface. Most of what we learn can be applied to Java3D and Direct3D, as well as other interfaces.
OpenGL Example

Here’s an example that renders a colored triangle (Note that this doesn’t include any ‘setup’ code)

```c
glBegin(GL_TRIANGLES);
   glColor3f(1.0, 0.0, 0.0); // red
   glVertex3f(-4.0, -2.0, 0.0);
   glColor3f(0.0, 1.0, 0.0); // green
   glVertex3f(4.0, -2.0, 0.0);
   glColor3f(0.0, 0.0, 1.0); // blue
   glVertex3f(0.0, 5.0, 0.0);
   glEnd();
```

Please read chapter 1 for an intro on rendering points, lines, triangles, strips, and fans in OpenGL.
OpenGL Program Organization

- Here’s a rough example of how a typical GL program might be organized:

```c
main() {
    // Initialization code:
    // - Open window
    // - Load & define texture maps, etc.
    while(not finished) {
        // Frame set up code:
        // - Clear screen
        // - Specify camera position & properties
        // - Specify lighting information
        for(each object) {
            // Object set up code
            // - Specify object position & orientation (matrix)
            for(each material) {
                // Specify material properties
                // - Lighting properties
                // - Texture properties
                for(each primitive) {
                    // Render primitive (glBegin, glVertex3f…)
                }
            }
        }
        // Frame completion code (glSwapbuffers()…)
    }
    // Shutdown code
}
```