Comp 410/510

Computer Graphics
Spring 2017

Overview - Graphics Pipeline
Recall: Basic Graphics System

Image is formed in **Frame Buffer** via the process **Rasterization**
In computer graphics, we form images which are generally two dimensional using a process analogous to how images are formed by physical imaging systems.

- Cameras, Microscopes, Telescopes, Human visual system
Elements of Image Formation

- Objects
- Viewer
- Light source(s)
- Attributes (that govern how light interacts with materials in the scene)
Light

- *Light* is the part of the electromagnetic spectrum that causes a reaction in our visual systems.
- Generally, these are wavelengths in the range of about 350-750 nm (nanometers).
- Long wavelengths appear as reds and short wavelengths as blues.
Human Eye as a Spherical Camera

- ~100M sensors in retina
- Rods sense only intensity
- 3 types of cones sense color
- Fovea has tightly packed sensors, more cones
- Periphery has more rods
- Focal length is about 20mm
- Pupil/iris controls light entry
Luminance and Color Images

• Luminance Image
  - Monochromatic
  - Values are gray levels
  - Analogous to working with black and white film or television

• Color Image
  - Has perceptual attributes of hue, saturation, and brightness
  - Do we have to consider every frequency in visible spectrum? Not necessarily.
  - Can use three primaries (red, green and blue) to approximate any color we can perceive.
Color Formation (or Synthesis)

- Form a color by adding amounts of three primaries
  - CRTs, projection systems, positive film

- Primaries are Red (R), Green (G), Blue (B)
# Color Systems

<table>
<thead>
<tr>
<th>RGB</th>
<th>CMY</th>
<th>HSI</th>
</tr>
</thead>
<tbody>
<tr>
<td>RED</td>
<td>(0, 255, 255)</td>
<td>(1.0, 1.0, 255)</td>
</tr>
<tr>
<td>YELLOW</td>
<td>(100, 100, 50)</td>
<td>(1.05, 0.5, 100)</td>
</tr>
<tr>
<td>GREEN</td>
<td>(0.255, 0.255)</td>
<td>(2.09, 1.0, 255)</td>
</tr>
<tr>
<td>BLUE</td>
<td>(255, 0, 0)</td>
<td>(4.19, 1.0, 255)</td>
</tr>
<tr>
<td>WHITE</td>
<td>(0, 0, 0)</td>
<td>(-1.0, 0.0, 255)</td>
</tr>
<tr>
<td>GREY</td>
<td>(192, 192, 192)</td>
<td>(-1.0, 0.0, 192)</td>
</tr>
</tbody>
</table>

- Convenient to scale values in the range 0 to 1 in algorithms.
- HSI values are computed from RGB values using Alg.
- $H \in [0, 2\pi], S \in [0, 1.0]$ and $I \in [0, 255]$.
- Equal proportions of RGB yield grey.
- Equal proportions of R and G yield yellow.

- R, G, B values normalized to (0, 1) interval
- Humans perceive gray for triples on the diagonal
- “Pure colors” on corners
HSI (or HSV) Color System

- Separates out intensity $I$ from the coding
- Two values ($H$ & $S$) encode chromaticity
- Convenient for designing colors; used in computer graphics and vision algorithms
- Hue $H$ refers to the perceived color (“purple”)
- Saturation $S$ models the purity of the color, that is, its dilution by white light (“light purple”)
- $I=(R+G+B)/3$: Conversion to gray-level
- Computation of $H$ and $S$ is a bit more complicated.

Darker

More intensity

Less and less pure (or less saturated)

Pure red

Brighter

Hue remains constant
Synthetic Camera Model

Equivalent alternatives:
Pinhole Camera

Use trigonometry to find projection of point at \((x, y, z)\)

\[
x_p = -\frac{x}{z/d} \quad y_p = -\frac{y}{z/d} \quad z_p = d
\]

These are equations of simple perspective projection
Application Programming Interface (API)

- Separation of objects, viewer, light sources
- Leads to simple software API
  - Specify objects, lights, camera, attributes
  - Let implementation determine the image
- Leads to fast hardware implementation
- But how is the API implemented?
How to Model Illumination?

- Some objects are blocked from light
- Light can reflect from object to object
- Some objects may be translucent
Local Illumination

- Computes color or shade independently for each object and for each light source
- Does not take into account interactions between objects
- Not very realistic, but fast
Global Illumination - Ray tracing

- For each image pixel, follow rays of light from center of projection until they are absorbed by objects, or go off to infinity, or reach a light source
  - Can handle global effects (though not exactly)
    - Multiple reflections
    - Translucent and reflecting objects
    - Shadows
  - Slower
  - Need whole data at once
Global (Ray Tracing) vs Local Illumination

The ambient lighting in the upper-right image is approximated by a constant value. This is typical of most scanline algorithms. The middle and lower-left images were rendered with a ray tracing global illumination algorithm.

The middle image was rendered with no ambient light calculations. The lower-left image was rendered with several levels of diffuse re-reflection to give a better approximation of the ambient light in this scene.
Ray Tracing Example
Another Approach for Global Illumination: Radiosity

- Simulates the propagation of light starting at the light sources.
- Stores illumination values on the surfaces of the objects, as the light is propagated starting at the light sources.
- Computes interactions between lights and objects more accurately.
- Assumes that light striking a surface is reflected in all directions.
- Radiosity calculation requires the solution of a large set of equations involving all the surfaces.
- Models well diffuse surfaces but not specular surfaces.
- Very slow.

Light striking a surface is reflected in all directions, following the Lambertian reflection model. This diffuse reflection of light leads to color bleeding, as light striking a surface carries that surface's color into the environment.
Radiosity vs Ray tracing

Which one is which?
Why not always use global illumination?

• Seems more physically-based, so can be used to design a graphics system
• Possible for simple objects such as polygons and quadrics with simple point light sources
• But slow and not well-suited for interactive applications
• Good for creating realistic movies
• Ray tracing with latest GPUs is now close to real time
Pipeline architecture

• Practical approach implemented by most API’s such as OpenGL, Java3D, DirectX.
• Considers only local illumination
• Process objects one at a time in the order they are generated by the application

• All steps can be implemented in hardware on the graphics card
• Converts object representations from world coordinate system to camera and then to screen coordinates
• Every change of coordinates is equivalent to a matrix transformation
• Used to transform objects, e.g., rotate, translate and scale
• Vertex processor also computes vertex colors
Projection

- Must carry out the process that combines the 3D viewer with 3D objects to produce the 2D image
  - **Perspective projection:** all projectors meet at the center of projection
  - **Parallel projection:** projectors are parallel, center of projection is replaced by a direction of projection
Vertices must be collected into geometric primitives before clipping and rasterization can take place, such as
- Line segments
- Polygons
- Curves and surfaces
Clipping

- Just as a real camera cannot “see” the whole world, the virtual camera can only see part of the world space.
  - Objects (primitives) that are not within this volume are clipped out of the scene.
Rasterization

- If an object is visible in the image, the corresponding pixels in the frame buffer must be assigned colors.
- Rasterizer produces a set of fragments for each object.
- Fragments are “potential pixels” with a location in the frame buffer, color, and depth attributes.
- Vertex attributes are interpolated over objects by the rasterizer.
Fragment Processing

• Fragments are processed to determine the color of the corresponding pixel in the frame buffer.
• Colors can be determined by texture mapping or interpolation of vertex colors.
• Fragments may be blocked by other fragments closer to the camera.
  - Hidden-surface removal.
The Programming Interface

• Programmer sees the graphics system through a software interface: the Application Programming Interface (API)
API Contents

• Functions that specify what we need to form an image
  - Objects
  - Viewer
  - Light Source(s)
  - Materials

• Other information
  - Input from devices such as mouse and keyboard
Object Specification

• Most APIs support a limited set of primitives including
  - Points (0D object)
  - Line segments (1D objects)
  - Polygons (2D objects)
  - Some curves and surfaces
    • Quadrics
    • Parametric polynomials
• All are defined through locations in space or vertices
Example (old style)

```c
// type of object
// Alternatives: GL_POINTS, GL_LINE_STRIP

glBegin(GL_POLYGON)
    glVertex3f(0.0, 0.0, 0.0);
    glVertex3f(0.0, 1.0, 0.0);
    glVertex3f(0.0, 0.0, 1.0);
    glVertex3f(0.0, 0.0, 1.0);
glEnd();
```

end of object definition
Example (GPU based)

• Put geometric data in an array

```cpp
vec3 points[3];
points[0] = vec3(0.0, 0.0, 0.0);
points[1] = vec3(0.0, 1.0, 0.0);
points[2] = vec3(0.0, 0.0, 1.0);
```

• Send array to GPU

• Tell GPU to render as triangle
Camera Specification

- Six degrees of freedom
  - Position of center of projection
  - Orientation
- Lens (focal length)
- Film plane
  - Size
Lights and Materials

- **Types of lights**
  - Point sources vs distributed sources
  - Spot lights
  - Near and far sources
  - Color properties

- **Material properties**
  - Absorption: color properties
  - Scattering
    - Diffuse
    - Specular