Topics covered in this lecture

- Volume rendering
- Framebuffer objects and rendering to texture
- Screen-space (post-processing) effects
- Shadows
- Advanced lighting
Volume rendering
Volume rendering applications

- Medical visualization
- Scientific visualization
- Computer games and visual effects
  - Clouds
  - Fire
  - Smoke
  - Volumetric fog
Volume images

- Represented as regular 3D grids with scalar or vector values
- Can be acquired with, e.g., a CT scanner or be generated procedurally
- Elements in volume images are called voxels
- A volume image can be uploaded to a 3D texture in OpenGL for rendering

A computed tomography (CT) volume image of a foot. The intensity values of the voxels represent different tissue types (bone, soft tissue, skin, etc)
Digital images

2D image

3D (volume) image

Pixel

Voxel
GPU-accelerated ray-casting
GPU-accelerated ray-casting

• Basic idea:

1. Render the front face of the volume image's bounding box to a 2D RGB texture to obtain ray starting points

2. Render the back face of the bounding box to a 2D RGB texture to obtain ray end points

3. Render a fullscreen quad and (in the ray-casting fragment shader) subtract the back-face texture from the front-face texture to obtain ray direction

4. Given the ray starting points and direction vectors, cast (still in the fragment shader) a ray from each fragment into the volume image

5. Let the ray gather color and transparency information from the voxels it passes through
Front-face texture (starting points)

Ray-start positions* displayed as RGB colors

*Specified as 3D texture coordinates (s,t,p) ranging from 0.0 to 1.0
Back-face texture (end points)

Ray-end positions* displayed as RGB colors

*Specified as 3D texture coordinates (s,t,p) ranging from 0.0 to 1.0
Ray-casting

- For each fragment (i.e., each pixel in the viewport), cast a ray from the starting point (front face) and sample the volume along the ray direction at even intervals until the ray reaches the end point (back face)
Rendering modes

• Through the use of different rendering modes, ray-casting can be used to create a variety of volume visualizations

• Examples of rendering modes:
  - Maximum intensity projection (MIP)
  - Front-to-back alpha blending
  - Iso-surface rendering
Maximum intensity projection (MIP)

- Basic idea: extract the maximum intensity value along each ray to create an X-ray-like projection of the data
Front-to-back alpha blending

- Define a **transfer function** that maps voxel intensity to color and opacity
- Create a semi-transparent projection of the volume by accumulating opacity values along each ray while composing (blending) colors
Front-to-back alpha blending
Demo
Transfer function example

- Maps voxel intensity (x-axis) to color (x-axis) and opacity (y-axis)
- Bone tissue (which has high intensity) will become white/yellowish and less transparent
- Soft tissue (which has low intensity) will become red/orange and more transparent
Iso-surface rendering

• Basic idea:

1. Define an **iso-value** (intensity threshold) corresponding to the surface that you want to render (e.g., bone)

2. For each casted ray, extract the position of the **first** sample that has an intensity value greater than or equal to the defined iso-value

3. Compute the volume image gradients/normals at the extracted surface positions and use them for shading
Extracted iso-surface (without shading)
Color-coded gradients/(normals derived from the volume image
Extracted iso-surface (with Blinn-Phong shading)
Demo
Sampling artifacts

- The sampling rate (or step length) affects the visual quality and the frame rate of the volume rendering.
- A low sampling rate enables a high frame rate, but will introduce ugly wood-grain artifacts. Example:
Wood-grain artifacts
How to remove or suppress wood-grain artifacts

- Increase the sampling rate (expensive!)
- Alternative: Hide the artifacts by perturbing the sampling positions with a tiled noise texture (stochastic jittering)
Stochastic jittering

Without jittering

With jittering
Volume rendering references

- **Acceleration Techniques for GPU-based Volume Rendering** (describes the basic ray-casting technique)
- **Real-Time Volume Graphics Tutorial** (refer to this for more details)
- **Advanced Illumination Techniques for GPU-based Volume Raycasting** (mainly advanced topics, but the Introduction and Basics section is worth reading)
Framebuffers

- A framebuffer is a collection of buffers that can be used as targets for the rendering
- OpenGL has two kinds of framebuffers:
  - the Default Framebuffer
  - user-created framebuffers called Framebuffer Objects (FBOs)
- See http://www.opengl.org/wiki/Framebuffer for more details
Framebuffer objects (FBOs)

- Normally, we render our scene to the default framebuffer, which output the result on the screen
- However, in some cases we don't want to output the rendered scene directly to the screen. One example is when we apply post-processing effects
- Framebuffer objects (FBOs) allow us to create user-defined framebuffers
- With them, we can render to non-default framebuffer locations (2D textures or render buffers) without disturbing the main screen
- Very useful, but also complicated to use...
Framebuffer objects (FBOs), cont.

- A FBO can have one or several 2D textures or render buffers as attachments
- These textures/buffers are attached to different attachment points depending on their content:
  - `GL_COLOR_ATTACHMENT i`
  - `GL_DEPTH_ATTACHMENT`
  - `GL_STENCIL_ATTACHMENT`
  - `GL_DEPTH_STENCIL_ATTACHMENT`

See [http://www.opengl.org/wiki/Framebuffer_Object](http://www.opengl.org/wiki/Framebuffer_Object) for more details
Screen-space (post-processing) effects

• Basic idea for single-pass post-processing:
  1. Render your scene to a 2D texture instead of to the default framebuffer (by attaching the texture to a FBO, which you then bind and clear before rendering the scene)
  2. Unbind the FBO and draw a fullscreen quad (i.e., two triangles) to drive a fragment shader that applies the post-processing effect on the 2D texture

• More advanced post-processing effects may require multiple passes and multiple textures/FBOs
Gaussian blur

Original image

Smoothed image
Image filtering (convolution)

Input image $f(x)$

Filter mask $h$

<table>
<thead>
<tr>
<th>$h(-1, -1)$</th>
<th>$h(0, -1)$</th>
<th>$h(1, -1)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$h(-1, 0)$</td>
<td>$h(0, 0)$</td>
<td>$h(1, 0)$</td>
</tr>
<tr>
<td>$h(-1, 1)$</td>
<td>$h(0, 1)$</td>
<td>$h(1, 1)$</td>
</tr>
</tbody>
</table>
Gaussian blur filter

- A Gaussian filter is separable, so we can perform the blurring in two passes (first along the x-axis and then along the y-axis) to speed up the filtering.
Bloom/glow

See GPU Gems, chapter 21
Toon shading

Depth-of-field

See http.developer.nvidia.com/GPUGems/gpugems_ch23.html

Image courtesy: http://www.realtimerendering.com
Motion blur

Image courtesy: http://www.realtimerendering.com
Shadows

- Important depth cue: helps us judge depth and distances between objects in a 3D scene
- Common techniques: projective shadows, shadow mapping, shadow volumes
- New variations of these techniques pops up every year
- High-quality shadows are difficult to implement...
- but even a simple and ugly shadow is usually better than no shadow at all
Without shadows
With projective shadow
Projective shadows

• Basic idea:
  1. construct a matrix $P$ that projects (in the light source direction) the shadowing objects onto the plane that should receive the shadows
  2. render the objects a second time (with $P$ applied) with a dark color and no illumination to create the shadows

• Simple to implement
• Works for objects that cast shadows on planar surfaces
• No self-shadowing

Image courtesy: http://www.realtimerendering.com
Shadow mapping

• Basic idea:

1. Render the scene depth values (from the light's perspective) into a texture to generate a **shadow map**

2. Render the scene again (color and depth values) from the viewer's perspective and compare the location of each fragment with the shadow map to determine if the point is in shadow

Image courtesy: http://www.realtimerendering.com
Shadow mapping

- Works for arbitrary objects and scenes (not only planar surfaces)
- Objects can shadow themselves
- Stairstepping artifacts can be a problem, even for high resolution shadow maps (see next slide)
- Soft shadows can (not trivial) be implemented with algorithms such as percentage-closer filtering (PCF)
- Implementing basic shadow mapping (without PCF) is a suitable course project
Stair-stepping artifacts

The projection of the shadow map stretches the shadow map texels, giving the shadow a blocky appearance

Image courtesy: http://www.realtimerendering.com
Ambient occlusion

- Simulates self-shadowing and shadowing of ambient light
- Surface points that are occluded becomes darker

No shadows  With ambient occlusion  Full global illumination

Image courtesy: http://www.realtimerendering.com
Screen-space ambient occlusion

- Render scene depth and normals to textures and compute dynamic ambient occlusion in a post-processing pass.
- Most techniques use a rotating filter mask with uniformly distributed points to sample the depth and normal textures.
- A subsequent blurring pass is required to remove noise.

Image courtesy: http://www.realtimerendering.com
Using pre-filtered cube maps for more realistic ambient lighting

Glossy surface

Rough surface
Using pre-filtered cube maps for more realistic ambient lighting

Standard flat ambient lighting  Ambient lighting from pre-filtered cube maps
Physically-based rendering (PBR)

[Image source: http://www.marmoset.co]
PBR textures

Image source: http://www.marmoset.co
PBR materials

Material values chart

- canvas
- rubber
- plaster
- brick
- rock
- dirt
- coal
- rust
- leaves
- satin
- wood
- concrete
- plastic (rough)
- mud
- painted metal
- ceramic
- satin
- brushed metal
- rough steel
- gold
- chrome

a = albedo (sRGB)  m = microsurface (linear)  r = reflectivity (sRGB)

Image source: http://www.marmoset.co
A few words about GPGPU

- **General-Purpose computing on Graphics Processing Units**
- Use the GPU as a Single Program Multiple Data (SPMD/SIMD) parallel processor

**Application areas:**
- Medical imaging
- Image processing and analysis
- Physics simulation
- Molecular dynamics
- Finance modeling
- Signal processing
- Cryptoanalysis
GPGPU programming languages

- **Compute Unified Device Architecture (CUDA)**
  - Created by NVIDIA
  - Proprietary framework
  - Excellent performance and mature tools/libraries, but only supported on NVIDIA GPUs

- **Open Computing Language (OpenCL)**
  - Open standard
  - Managed by the Khronos Group
  - Supported by all GPU vendors (Intel, AMD, and NVIDIA) :-)
  - Enables parallel programming of heterogeneous systems (GPUs, CPUs, and APUs)