02562 Rendering - Introduction

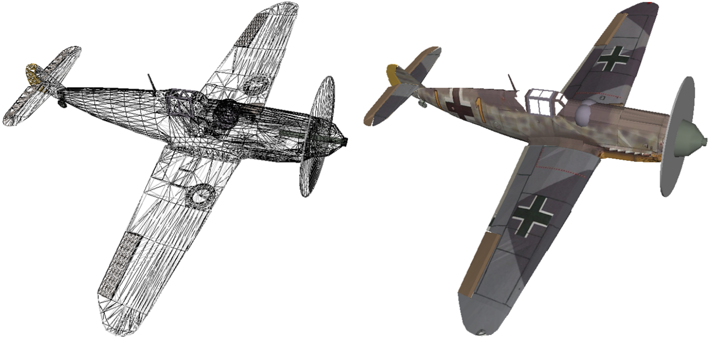
Triangle Meshes and Smooth Shading

Jeppe Revall Frisvad

October 2023

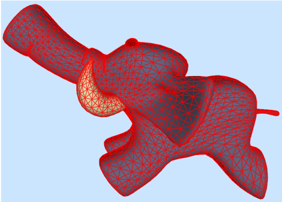
Scene geometry

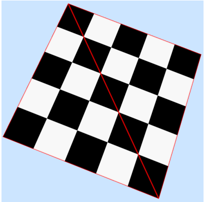
- Surface geometry is often modelled by a collection of triangles in which some triangles share edges (a triangle mesh).
- Triangles provide a discrete representation of an arbitrary surface.
  A quick example:



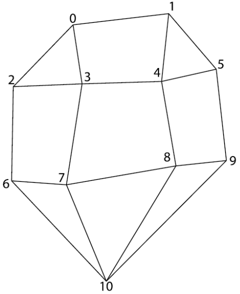
- Triangles are useful as they are defined by only three vertices.
  And ray-triangle intersection is simple.

Triangle meshes





- The *indexed face set* is a popular data representation of polygon meshes.
- Any polygon mesh can be converted to a triangle mesh.

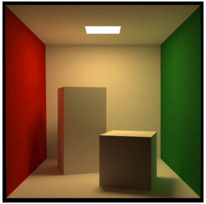


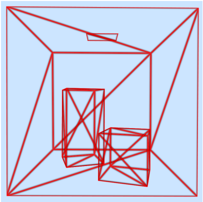
VERTICES
0: (-0.2, 1.5, 0)
1: (1.3, 1.7, 0)
2: (-1.1, 0.4, 0)
3: (0.0, 0.45, 1)
4: (1.1, 0.5, 1.2)
5: (2.1, 0.75, 0.2)
6: (-1.2, -1,0.01)
7: (-0.3, -1.2,2)
8: (1.3,-0.9, 3)
9: (2.0 -0.8,1.2)
10: (0.4, -2.1, -1.1)

FACES
0: 0,2,3
1: 0,3,4,1
2: 1,4,5
3: 2,6,7,3
4: 3,7,8,4
5: 4,8,9,5
6: 6,10,7
7: 7,10,8
8: 8,10,9

Figure from Bærentzen et al. *Guide to Computational Geometry Processing*, Springer, 2012.

The Cornell box





- The Cornell box is a convenient test scene for developing rendering algorithms.
  https://www.graphics.cornell.edu/online/box/data.html
- A library file `OBJParser.js` is available to help you load such a scene if provided in Wavefront OBJ format. Define a few global variables:

```
var obj_filename = 'data/CornellBoxWithBlocks.obj';
var g_objDoc = null;      // Info parsed from OBJ file
var g_drawingInfo = null; // Info for drawing the 3D model with WebGL
```

  and we can then read the OBJ file from disk by calling

```
async function main()
{
  readOBJFile(obj_filename, 1, true); // file name, scale, ccw vertices

  ⋮
}
```

- Ray-trimesh intersection is the same as ray-triangle intersection, except that you must now retrieve the vertices from an indexed face set.

Code for asynchronous file loading

Request

```
// Read a file
function readOBJFile(fileName, scale, reverse)
{
  var request = new XMLHttpRequest();
  request.onreadystatechange = function() {
    if (request.readyState === 4 && request.status !== 404)
      onReadOBJFile(request.responseText, fileName, scale, reverse);
  }
  request.open('GET', fileName, true); // Create a request to get file
  request.send(); // Send the request
}
```

Parse

```
// OBJ file has been read
function onReadOBJFile(fileString, fileName, scale, reverse)
{
  var objDoc = new OBJDoc(fileName); // Create a OBJDoc object
  var result = objDoc.parse(fileString, scale, reverse);
  if (!result) {
    g_objDoc = null; g_drawingInfo = null;
    console.log("OBJ file parsing error.");
    return;
  }
  g_objDoc = objDoc;
}
```

Send to GPU

```
// OBJ File has been read completely
function onReadComplete(device, pipeline, buffers)
{
  // Get access to loaded data
  g_drawingInfo = g_objDoc.getDrawingInfo();

  // Set up buffers
  ⋮

  // Create and return bind group
  ⋮
}
```

Waiting for the data

- A new animate function:

```
var bindGroup;
function animate() {
  if (!g_drawingInfo && g_objDoc && g_objDoc.isMTLComplete()) {
    // OBJ and all MTLs are available
    bindGroup = onReadComplete(device, pipeline, buffers);
  }
  if (!g_drawingInfo) {
    requestAnimationFrame(animate);
    return;
  }
  render(device, context, pipeline, bindGroup);
}
animate();
```

- Setting up storage buffers with the data
  - WGSL

```
@group(0) @binding(i₁) var<storage> vPositions: array<vec3f>;
@group(0) @binding(i₂) var<storage> meshFaces: array<vec3u>;
```

  - JavaScript

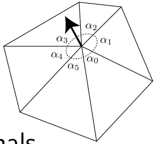
```
const positionsBuffer = device.createBuffer({
  size: g_drawingInfo.vertices.byteLength,
  usage: GPUBufferUsage.COPY_DST | GPUBufferUsage.STORAGE
});
device.queue.writeBuffer(positionsBuffer, 0, g_drawingInfo.vertices);

const indicesBuffer = device.createBuffer({
  size: g_drawingInfo.indices.byteLength,
  usage: GPUBufferUsage.COPY_DST | GPUBufferUsage.STORAGE
});
device.queue.writeBuffer(indicesBuffer, 0, g_drawingInfo.indices);
```

Flat shading or smooth shading

- Triangles are flat. Their *geometric* normals lead to flat shading.
- How do we make it smooth? Interpolated per-vertex lighting?
- What is the normal in a vertex?

  The angle-weighted pseudo-normal is a good choice.

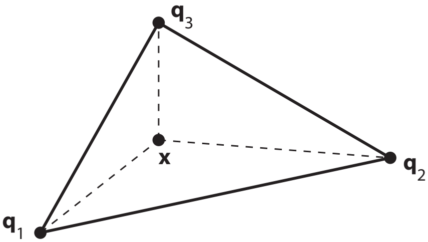
  [Bærentzen et al. 2012, Sec. 8.1].



$$\vec{n} = \frac{\sum \alpha_i \vec{n}_i}{\| \sum \alpha_i \vec{n}_i \|}$$

- Another indexed face set is created for the vertex normals.
- Interpolation of the vertex normals across each triangle leads to smooth shading.

Linear interpolation across triangles



- A point $\boldsymbol{x}$ in a triangle is given by a weighted average of the triangle vertices $(\boldsymbol{q}_1, \boldsymbol{q}_2, \boldsymbol{q}_3)$:

$$\boldsymbol{x} = \alpha \boldsymbol{q}_1 + \beta \boldsymbol{q}_2 + \gamma \boldsymbol{q}_3 \quad , \quad \alpha + \beta + \gamma = 1 \ .$$

- The weights (α, β, γ) are the *barycentric coordinates*.
- The point is in the triangle if $\alpha, \beta, \gamma \in [0, 1]$. That is,

$$\beta \geq 0 \quad \text{and} \quad \gamma \geq 0 \quad \text{and} \quad \beta + \gamma \leq 1 \ .$$

- Replace the triangle vertices $(\boldsymbol{q}_1, \boldsymbol{q}_2, \boldsymbol{q}_3)$ by vertex normals and *normalize* to get the interpolated normal.
- Replace the triangle vertices $(\boldsymbol{q}_1, \boldsymbol{q}_2, \boldsymbol{q}_3)$ by vertex texture coordinates to get (u, v)-coordinates for texturing.

Shading a triangle mesh

- The energy travelling along a ray is measured in radiance $L = \frac{\mathrm{d}^2\Phi}{\cos\theta\,\mathrm{d}A\,\mathrm{d}\omega}$ (radiant flux per projected area per solid angle).
- The outgoing radiance L_o at a surface point $\boldsymbol{x}$ in the direction $\vec{\omega}_o$ is the sum of emitted radiance L_e and reflected radiance L_r:

$$L_o(\boldsymbol{x}, \vec{\omega}_o) = L_e(\boldsymbol{x}, \vec{\omega}_o) + L_r(\boldsymbol{x}, \vec{\omega}_o) \ .$$

- Reflected radiance is computed using a BRDF f_r and an estimate of incident radiance L_i at the surface point.
- Algorithm:
  - Find all visible points in a scene (ray casting).
  - At every diffuse point in the scene (with $f_r = \rho_d/\pi$), gather the direct contribution from all light sources that are not occluded (shader and light source sampler).
  - If the light source has an areal extension (defined by a triangle mesh, for example), use the center of the source (average of vertex positions) to test visibility.

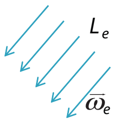
The rendering equation

- Surface scattering is defined in terms of
  - Radiance:
    $$L = \frac{\mathrm{d}^2\Phi}{\cos\theta \, \mathrm{d}A \, \mathrm{d}\omega} \, .$$
  - Irradiance:
    $$E = \frac{\mathrm{d}\Phi}{\mathrm{d}A} \quad , \quad \mathrm{d}E = L_i \cos\theta \, \mathrm{d}\omega \, .$$
  - Bidirectional Reflectance Distribution Function (BRDF):
    $$f_r(\boldsymbol{x}, \vec{\omega}_i, \vec{\omega}_o) = \frac{\mathrm{d}L_r(\boldsymbol{x}, \vec{\omega}_o)}{\mathrm{d}E(\boldsymbol{x}, \vec{\omega}_i)} \, .$$
- The rendering equation then emerges from $L_o = L_e + L_r$:
  $$L_o(\boldsymbol{x}, \vec{\omega}_o) = L_e(\boldsymbol{x}, \vec{\omega}_o) + \int_{2\pi} f_r(\boldsymbol{x}, \vec{\omega}_i, \vec{\omega}_o) L_i(\boldsymbol{x}, \vec{\omega}_i) \cos\theta_i \, \mathrm{d}\omega_i \, .$$
- This is an integral equation. Integral equations are recursive in nature.

Direct illumination due to different light sources

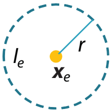
- A directional light emits a constant radiance L_e in one particular direction $\vec{\omega}_e = -\vec{\omega}_i$

$$L_r = \int_{2\pi} f_r L_i \cos\theta_i \, \mathrm{d}\omega_i = f_r \, V L_e \left(-\vec{\omega}_e \cdot \vec{n}\right).$$



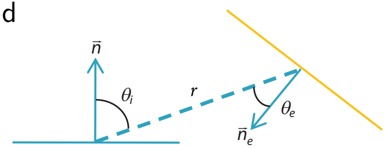
- A point light emits a constant intensity I_e in all directions from one particular point $\boldsymbol{x}_e$

$$L_r = f_r \frac{V}{r^2} \left(\vec{\omega}_i \cdot \vec{n}\right) I_e \,, \qquad \begin{aligned} r &= \|\boldsymbol{x}_e - \boldsymbol{x}\| \\ \vec{\omega}_i &= (\boldsymbol{x}_e - \boldsymbol{x})/r \end{aligned} \,.$$



- An area light emits a cosine weighted radiance distribution from each differential element of area $\mathrm{d}A$. We have $\mathrm{d}\omega_i = \frac{\cos\theta_e}{r^2}\mathrm{d}A_e$ and

$$L_r = \int f_r \, V L_e \, \cos\theta_i \, \frac{\cos\theta_e}{r^2} \mathrm{d}A_e \,.$$



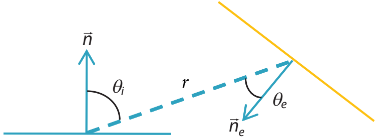
$$\cos\theta_e = -\vec{\omega}_i \cdot \vec{n}_e$$

V is visibility.

Approximation for a distant area light

- An area light emits a cosine weighted radiance distribution from each area element

$$L_r = \int f_r \, V L_e \, \cos\theta_i \, \frac{\cos\theta_e}{r^2} \mathrm{d}A_e \, .$$



$$\cos\theta_e = -\vec{\omega}_i \cdot \vec{n}_e$$

  V is visibility.

- Assuming the area light is distant, we can approximate its lighting of the scene using just one sample. Suppose we place it at the center of the bounding box $\boldsymbol{x}_e$. Then

$$L_r = f_r \, \frac{V}{r^2} \, (\vec{\omega}_i \cdot \vec{n}) \, \underbrace{\sum_{\triangle=1}^{N} (-\vec{\omega}_i \cdot \vec{n}_{e\triangle}) L_{e\triangle} \, A_{e\triangle}}_{I_e} \, ,$$

  where N is the number of triangles in the area light and $\triangle$ is a triangle index.

- This is like a point light, but with a different intensity.

Exercises

- Upload scene data to the GPU using storage buffers.
- Implement ray-trimesh intersection including interpolation of vertex normals.
- Set up buffers for rendering multi-material objects.
- Shade triangle meshes using a directional light
  and an arealight defined by a triangle mesh.