Subdivision and Project 2
Outline

- Subdivision
  - What is it?
  - What properties does it have / do we want?
  - What kinds of algorithms exist and what advantages do they have?
- Project 2
  - Texture mapping crash course
  - Loop subdivision algorithm
What is subdivision?

- Start with a given polygon mesh
- Apply refinement scheme to get an increasingly smooth surface by taking in the mesh and subdividing it to create new vertices and faces
- The limit of this subdivision is a smooth surface, though in practice we can’t apply it this many times
  - Caveat – provided we don’t define creases and boundaries
What properties might we want?

- Efficiency
  - use a small number of floating point operations
- Local definition
  - don’t look very far away from current point
- Simplicity
  - We probably do not want a ton of rules
- Continuity
  - What kind of properties can we prove about the resulting surface?
Efficiency & Local Definition

- Subdivision is efficient because only several neighboring points are used in the computation of new points.
- By contrast, rendering a surface defined by an implicit equation is expensive, requiring an algorithm such as marching cubes.
Quick terminology

- **Ordinary vertices**
  - For triangular meshes, vertices of valence 6 on the interior and valence 4 on the boundaries
  - For quadrilateral meshes, vertices of valence 4 on the interior and valence 3 on the boundaries

- **Extraordinary vertices**
  - All other valences

- **Odd and even vertices**
  - Odd vertices are those that are added on the current step of the subdivision
  - Even vertices are those that are inherited from the previous level
Subdivisions Schemes

In general, there is a fairly straightforward way to classify the subdivision schemes that exist:

- Type of refinement – face split or vertex split
- Type of generated mesh – triangular or quadrilateral
- Approximating vs. interpolating
- Smoothness of the limit surface for regular meshes
Subdivision Schemes

- Face split vs. Vertex split

Figure 4.1: Different refinement rules.
Subdivision Schemes

- Approximation vs. Interpolation
  - Interpolation – original points remain the same
  - Approximation – original points not the same
  - Face splitting can be either since the vertices of the coarser tiling are also vertices in the refined tiling
  - Approximating generally produces smoother surfaces
## Subdivision Schemes

<table>
<thead>
<tr>
<th>Face Split</th>
<th>Triangular meshes</th>
<th>Quadrilateral Meshes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Approximating</strong></td>
<td>Loop ($C^2$)</td>
<td>Catmull-Clark ($C^2$)</td>
</tr>
<tr>
<td><strong>Interpolating</strong></td>
<td>Modified Butterfly ($C^1$)</td>
<td>Kobbelt ($C^1$)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Vertex Split</th>
</tr>
</thead>
<tbody>
<tr>
<td>Doo-Sabin, Midedge ($C^1$)</td>
</tr>
<tr>
<td>Biquartic ($C^2$)</td>
</tr>
</tbody>
</table>
Loop Scheme

- Face splitting, approximating scheme for triangular meshes proposed by Charles Loop.
- $C^1$ continuity for all valences and $C^2$ continuity over regular meshes
- Can be applied to polygon meshes after triangulating the mesh
Figure 4.3: Loop subdivision: in the picture above, \( \beta \) can be chosen to be either \( \frac{1}{n}(5/8 - (\frac{3}{8} + \frac{1}{4} \cos \frac{2\pi}{n})^2) \) (original choice of Loop [16]), or, for \( n > 3 \), \( \beta = \frac{3}{8n} \) as proposed by Warren [33]. For \( n = 3 \), \( \beta = 3/16 \) can be used.
Computing Tangents

- **Interior**
  \[ t_1 = \sum_{i=0}^{k-1} \cos \frac{2\pi i}{k} p_{i,1} \]
  \[ t_2 = \sum_{i=0}^{k-1} \sin \frac{2\pi i}{k} p_{i,1}. \]

- **Boundary**
  \[ t_{along} = p_{0,1} - p_{k-1,1} \]

\[ t_{across} = p_{0,1} + p_{1,1} - 2p_0 \quad \text{for } k = 2 \]
\[ t_{across} = p_{2,1} - p_0 \quad \text{for } k = 3 \]
\[ t_{across} = \sin \theta (p_{0,1} + p_{k-1,1}) + (2 \cos \theta - 2) \sum_{i=1}^{k-2} \sin i\theta p_{i,1} \quad \text{for } k \geq 4 \]
\( \theta = \pi/(k-1) \)

- Computing the normal at that point is then just \( t_1 \times t_2 \).
First proposed by Dyn, Gregory and Levin, but was not $C^1$ continuous

A modified scheme was later proposed that produced $C^1$ continuous meshes for arbitrary surfaces

Interpolating scheme applied to triangular meshes
For a regular vertices, imagine arranging the control points into a vector
\[ p = [p_0, p_{0,1}, p_{1,1}, \ldots, p_{5,1}, p_{0,2}, p_{1,2}, \ldots, p_{5,3}] \]
of length 19, then the tangents are given as follows

\[ l_1 = \left[ 0, 16, 8, -8, -16, -8, 8, -4, 0, 4, 4, 0, -4, 1, \frac{1}{2}, -\frac{1}{2}, -1, -\frac{1}{2}, \frac{1}{2} \right] \]
\[ l_2 = \sqrt{3} \left[ 0, 0, 8, 8, 0, -8, -8, -\frac{4}{3}, -\frac{8}{3}, -\frac{4}{3}, \frac{4}{3}, \frac{8}{3}, 0, \frac{1}{2}, \frac{1}{2}, 0, -\frac{1}{2}, -\frac{1}{2} \right] \]  

Otherwise, the same tangent rules as the Loop scheme are applied.
## Modified Butterfly Scheme

- Boundary rules are much more complicated in the butterfly scheme because the stencil is much bigger.
- We can break them into groups based on the two points on the edge where the point is being added.

<table>
<thead>
<tr>
<th>Head</th>
<th>Tail</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>regular interior</td>
<td>regular interior</td>
<td>standard rule</td>
</tr>
<tr>
<td>regular interior</td>
<td>regular crease</td>
<td>regular interior-crease</td>
</tr>
<tr>
<td>regular crease</td>
<td>regular crease</td>
<td>regular crease-crease 1 or 2</td>
</tr>
<tr>
<td>extraordinary interior</td>
<td>extraordinary interior</td>
<td>average two extraordinary rules</td>
</tr>
<tr>
<td>extraordinary interior</td>
<td>extraordinary crease</td>
<td>same</td>
</tr>
<tr>
<td>extraordinary crease</td>
<td>extraordinary crease</td>
<td>same</td>
</tr>
<tr>
<td>regular interior</td>
<td>extraordinary interior</td>
<td>interior extraordinary</td>
</tr>
<tr>
<td>regular crease</td>
<td>extraordinary crease</td>
<td>crease extraordinary</td>
</tr>
<tr>
<td>extraordinary interior</td>
<td>regular crease</td>
<td>interior extraordinary</td>
</tr>
<tr>
<td>regular crease</td>
<td>extraordinary crease</td>
<td>crease extraordinary</td>
</tr>
</tbody>
</table>
Modified Butterfly Scheme

Figure 4.6: Regular Modified Butterfly boundary/crease rules.

\[ c_0 = 1 - \left( \frac{1}{k-1} \right) \sin \theta_k \sin i \theta_k / (1 - \cos \theta_k) \]

\[ c_{i0} = c_{ik} = \frac{1}{4} \cos i \theta_k - \frac{1}{4(k-1)} \sin 2 \theta_k \sin 2 i \theta_k / (\cos \theta_k - \cos 2 \theta_k) \]

\[ c_{ij} = \left( \frac{1}{k} \right) \left( \sin i \theta_k \sin j \theta_k + \frac{1}{2} \sin 2 i \theta_k \sin 2 j \theta_k \right) \]

Figure 4.7: Modified Butterfly rules for neighbors of a crease/boundary extraordinary vertex.
**Catmull-Clark Scheme**

- Face splitting, approximating scheme on quadrilaterals
- Produces surfaces that are $C^2$ everywhere except extraordinary vertices where they are $C^1$

---

**Figure 4.8:** Catmull-Clark subdivision. Catmull and Clark [4] suggest the following coefficients for rules at extraordinary vertices: $\beta = \frac{1}{12}$ and $\gamma = \frac{1}{12}$
Kobbelt Scheme

- Face splitting, interpolating scheme on quadrilateral meshes
- C1 continuous for all valences

![Figure 4.11: Kobbelt subdivision.](image-url)
Other Subdivision Schemes

- Doo-Sabin, Midedge ($C^1$)
- Biquartic ($C^2$)
- Theses are vertex splitting algorithms.
Figure 4.20: Different subdivision schemes produce similar results for smooth meshes.

Initial mesh  Loop  Catmull-Clark  Catmull-Clark, after triangulation
You will have 2 tasks in project 2.

- Texture map a mesh given the texture and the texture coordinates
  - Implement the loop subdivision algorithm
- All initial positions, normals, texture coordinates and whether or not this particular mesh needs to be texture are given to you.
What is a texture?

- A texture is just a bitmap image
- Our image is a 2D array: 
  texture[height][width][4]
- Pixels of the texture are called *texels*
- Texel coordinates are in 2D, in the range $[0,1]$
  - OpenGL uses $(s, t)$ as the coordinate parameters.
  - Commonly referred to as $(u, v)$ coordinates by most graphics programs.
In order to map a 2D image to a piece of geometry, we consider two functions:

- A mapping function which takes 3D points to \((u, v)\) coordinates.
  - \(f(x, y, z)\) returns \((u, v)\)

- A sampling/lookup function which takes \((u, v)\) coordinates and returns a color.
  - \(g(u, v)\) returns \((r, g, b, a)\)
The basic idea is that for some polygon (which may have arbitrary shape and size), we manually assign each of its vertices \((u, v)\) coordinates in the range from \([0, 1]\).

We then use these \((u, v)\) coordinates as rough indices into our texture array.

- These don’t necessarily hit into the array so some sort of interpolation is generally used.
OpenGL Texture Mapping

- **Initialization**
  - Enable GL texture mapping
  - Specify texture
  - Read image from file into array in memory or generate image using the program (procedural generation)
  - Specify any parameters
  - Define and activate the texture

- **Draw**
  - Draw objects and assign texture coordinates to vertices
OpenGL Texture Mapping

- **Color blending**
  - How to determine the color of the final pixel?
    - GL_REPLACE – use texture color to replace object color
    - GL_BLEND – linear combination of texture and object color
    - GL_MODULATE – multiply texture and object color
  - Example:
    - `glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);`

- **Texture Coordinates outside [0,1]** ➔ Two choices:
  - Repeat pattern (GL_REPEAT)
  - Clamp to maximum/minimum value (GL_CLAMP)
  - Example:
    - `glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP)`
    - `glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP)`

![repeat](repeat.png) ![clamp](clamp.png)
OpenGL Texture Mapping

// somewhere else...
Gluinttexture_id;

void init(){
    // acquire load our texture into an array
    // the function we use this semester is in imageio.hpp
    char* pointer; // TODO: give me some values!

    // enable textures
    glEnable(GL_TEXTURE_2D);
    glGenTextures(1, &texture_id);
    glBindTexture(GL_TEXTURE_2D, texture_id);

    // sample: specify texture parameters
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);

    // set the active texture
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, 256, 256, 0, GL_RGBA,
                 GL_UNSIGNED_BYTE, pointer);
}

OpenGL Texture Mapping

- Use GLTexCoord2f(s,t) to specify texture coordinates
- Example:

```c
    glEnable(GL_TEXTURE_2D)
    glBegin(GL_QUADS);
    glTexCoord2f(0.0,0.0); glVertex3f(0.0,0.0,0.0);
    glTexCoord2f(0.0,1.0); glVertex3f(2.0,10.0,0.0);
    glTexCoord2f(1.0,0.0); glVertex3f(10.0,0.0,0.0);
    glTexCoord2f(1.0,1.0); glVertex3f(12.0,10.0,0.0);
    glEnd();
    glDisable(GL_TEXTURE_2D)
```

- State machine: Texture coordinates remain valid until you change them or exit texture mode via glDisable (GL_TEXTURE_2D)
Subdivision

- We provide you with the initial positions, normals and texture coordinates in this lab.
- Your job is to implement the loop subdivision algorithm and output a subdivided mesh.
- You can use the same algorithm for the position, normals and the texture coordinates.
Figure 4.3: Loop subdivision: in the picture above, $\beta$ can be chosen to be either $\frac{1}{n}(5/8 - (\frac{3}{8} + \frac{1}{4} \cos \frac{2\pi}{n})^2)$ (original choice of Loop [16]), or, for $n > 3$, $\beta = \frac{3}{8n}$ as proposed by Warren [33]. For $n = 3$, $\beta = 3/16$ can be used.
Subdivision

- Essentially requires 2 passes
  - First pass, handle creating odd vertices
  - Second pass, move even vertices
- Suggested path
  - Implement the interior cases first
    - This will allow you to test this on closed meshes before moving on to the ones with boundaries
  - Implement the boundary cases
Subdivision

- What is it?
- What properties does it have / do we want?
- What kinds of algorithms exist and what advantages do they have?

Project 2

- Texture mapping crash course
- Loop subdivision algorithm
Image sources

http://www.mrl.nyu.edu/~dzorin/sigoocourse/