Tessellation

Steve Rotenberg
CSE168: Rendering Algorithms
UCSD, Spring 2014
Tessellation

- *Tessellation* refers to the tiling of a surface (or higher dimensional shape) with no overlaps or gaps.
- Typically, in computer graphics, it refers more specifically to the triangulation of surfaces.
Parametric Surfaces
Parametric Surface

- A *parametric surface* (or *explicit surface*) is a surface that is explicitly defined mathematically as a function of two parameters:

\[ \mathbf{x} = \mathbf{f}(s, t) \]

- For some domain of \( s \) & \( t \) (typically from 0...1)
- Parametric surfaces have a rectangular topology
- Popular surfaces of this type include:
  - Bezier surfaces
  - B-splines
  - NURBS (non-uniform rational B-splines)
Beziers Surfaces

- Bezier surfaces are a straightforward extension to Bezier curves.
- Instead of the curve being parameterized by a single variable $t$, we use two variables, $s$ and $t$.
- By definition, we choose to have $s$ and $t$ range from 0 to 1 and we say that an $s$-tangent crossed with a $t$-tangent will represent the normal for the front of the surface.
Control Mesh

- Consider a *bicubic* Bezier surface (bicubic means that it is a cubic function in both the $s$ and $t$ parameters)
- A cubic curve has 4 control points, and a bicubic surface has a grid of 4x4 control points, $p_0$ through $p_{15}$
Surface Evaluation

- The bicubic surface can be thought of as 4 curves along the $s$ parameter (or alternately as 4 curves along the $t$ parameter).
- To compute the location of the surface for some $(s, t)$ pair, we can first solve each of the 4 $s$-curves for the specified value of $s$.
- Those 4 points now make up a new curve which we evaluate at $t$.
- Alternately, if we first solve the 4 $t$-curves and to create a new curve which we then evaluate at $s$, we will get the exact same answer.
- This gives a pretty straightforward way to implement smooth surfaces with little more than what is needed to implement curves.

\[(0.2, 0.6)\]
Normals

• To compute the normal of the surface at some location \((s,t)\), we compute the two tangents at that location and then take their cross product.

• Usually, it is normalized as well.

\[
\mathbf{n}^* = \frac{\partial \mathbf{x}}{\partial s} \times \frac{\partial \mathbf{x}}{\partial t}
\]

\[
\mathbf{n} = \frac{\mathbf{n}^*}{|\mathbf{n}^*|}
\]
Convex Hull Property

• If we take all of the control points for a Bezier curve and construct a convex polygon around them, we have the *convex hull* of the curve
• An important property of Bezier curves is that every point on the curve itself will be somewhere within the convex hull of the control points
• Surfaces have this same property
Tessellation
Uniform Tessellation

- The most straightforward way to tessellate a parametric surface is *uniform tessellation*
- With this method, we simply choose some resolution in $s$ and $t$ and uniformly divide up the surface like a grid
- This method is very efficient to compute, as the cost of evaluating points on the surface is only slightly more costly than evaluating points on a curve
- However, as the generated mesh is uniform, it may have more triangles than it needs in flatter areas and fewer than it needs in highly curved areas
Adaptive Tessellation

• Very often, the goal of a tessellation is to provide the fewest triangles necessary to accurately represent the original surface.
• For a curved surface, this means that we want more triangles in areas where the curvature is high, and fewer triangles in areas where the curvature is low.
• We may also want more triangles in areas that are closer to the camera, and fewer farther away.
• *Adaptive tessellation* schemes are designed to address these requirements.
Recursive Tessellation

- Most adaptive tessellation approaches are recursive and use a top-down process
- We will first look at a recursive tessellation of a single curve, and then extend it to surfaces
- We’ll assume that the curve is defined over the [0…1] domain in \( t \)
- We start by examining the entire curve in that domain and consider what would happen if we simply replaced it with a line segment
- We then measure how well the line segment represents the surface, and if we feel it is good enough, then we are done
- Otherwise, the curve is subdivided into 2 curves and we repeat the process recursively on each half
Recursive Tessellation

• How do we determine if a line segment is a good enough approximation to the curve?
• There are many variations on this process, but a simple and common approach is to compare the midpoint of the line segment to the midpoint of the curve
• If they are within some specified distance, then we say it’s good enough
• Otherwise, we split
Curve Tessellation
Curve Tessellation
Curve Tessellation
Curve Tessellation
Curve::Tessellate(float t0, float t1, Vector3 &p0, Vector3 &p1) {
    float t05 = 0.5*(t0+t1);  // mid-parameter
    Vector3 c05 = Evaluate(t05);  // evaluate curve at t05
    Vector3 p05 = 0.5*(p0+p1);  // midpoint of segment
    if(c05.Distance(p05) > TOLERANCE) {
        Tessellate(t0,t05,p0,p05);
        Tessellate(t05,t1,p05,p1);
    }
    else DrawSegment(p0,p1);
}
Subdivision Rules

• The previous example was based on comparing the distance of the midpoint of a curve to the midpoint of the approximating segment.
• This process is generally pretty good and widely used, however there are a few potential problems and other issues.
• For one thing, we could have a situation where the midpoint of the curve happens to hit the midpoint of the segment by chance, even though the two hardly match.
• There are some practical ways to handle this.
Subdivision Rules

• In addition to the midpoint distance rule, we can add other rules that would trigger a segment to subdivide
• Some possibilities include:
  – Compare distances of other points (0.25 and 0.75 for example)
  – Compare the angle of the tangents at the start and end points
  – Mathematically evaluate the exact curvature at the midpoint (assuming the curve type allows this)
  – Compare the ratio of the midpoint distance to the length of the segment
• All these rules (and others) can be easily combined, since they are just binary decisions and the curve can be subdivided if any one of the rules triggers
Pixel-Based Error

• One nice advantage of the midpoint distance approach is that it can be divided by the distance to the camera to provide a pixel based error measurement

• This provides a scheme for tessellating curves based on the error in pixels rather than the error in 3D distance

• This will cause curves closer to the camera to automatically adapt more than curves far away
Surface Tessellation

- Most adaptive tessellation approaches are recursive and use a top-down process
- We’ll assume that the surface is defined over the [0...1] domain in s and t
- We start by examining the entire surface in that domain and consider what would happen if we simply replaced it with two triangles
- We then measure how well the two triangles represent the surface, and if we feel it is good enough, then we are done
- Otherwise, the surface is subdivided (usually into 2 or 4 sub-surfaces) and we repeat the process recursively on each sub-surface
Surface Tessellation

• As with curves, we can use a rule essentially based on midpoint distances
• One nice approach is to test the midpoints of the 4 edges of the surface as well as the midpoint of the entire surface
• This provides a scheme for splitting the surface only in $s$, only in $t$, or in both $s$ and $t$, depending on what is required
Mixed Uniform/Adaptive Tessellation

• Some practical renderers use a mixed tessellation scheme
• First, the original surface patch is adaptively subdivided into several subpatches, each approximately the same size (say around 10 pixels on a side)
• Then, each of the subpatches (which is just a rectangular \(s,t\) range within the larger 0,1 rectangle) is uniformly tessellated to some resolution (say 10 x 10)
• The result is that the curved surface is tessellated into triangles roughly the size of a single pixel
• The bulk of the cost of the algorithm is in the uniform tessellation, which can be implemented in a very efficient way
Ray Tracing Curved Surfaces
Ray Tracing Curved Surfaces

• There are several strategies for including curved surfaces into a ray tracer:
  – Pre-tessellation: before starting to render, we tessellate the surface into a bunch of triangles and add them to a spatial data structure
  – Lazy evaluation: tessellate and build a spatial data structure on the fly, as rays are tested against the surface
  – Direct ray intersection: don’t tessellate at all. Every ray is intersected with the surface through a more direct means
Pre-Tessellation

- The pre-tessellation approach is the most common in practice due to its simplicity and high performance.
- Curved surfaces can be pre-tessellated either based on 3D spatial properties and/or properties measured in pixel space.
- One catch however, is with instanced objects.
- We can’t really use pixel space metrics easily if we are using multiple copies of the geometry, each at different distances to the camera.
Lazy Evaluation

• For the lazy evaluation approach, we start by only computing the bounding box of
  the entire surface
• When the bounding box is intersected by a ray, we then recursively subdivide the
  surface, computing bounding boxes of the sub-surfaces as we go
• We continue subdividing the boxes that the ray intersects until we generate
  triangles
• These are then kept in a box-tree-like data structure so they don’t need to be
  recomputed for the next ray that comes by
• This approach does essentially the same work as recursive subdivision combined
  with box-tree generation, but will have some performance overhead for the
  additional bookkeeping and complexity
• This approach might be useful if we didn’t expect rays to hit all of the surfaces,
  which might happen for simple renders, but wouldn’t be effective in a path tracer
  that bounces rays all over the place
• This approach can also be useful for extremely large scenes that don’t fit into
  memory. If combined with a system for deleting sections of the box-tree that
  haven’t been intersected for a while (or moving them to a slower, larger memory
  pool), we can create a system that fits within some fixed memory limit
Direct Intersection

- We can intersect rays directly with the surface if we want.
- However, this is guaranteed to be much slower than pre-tessellated triangles, so it is only used in experimental cases or situations where memory is very limited.
- Some surfaces can actually be mathematically intersected by solving the roots of a large polynomial.
- Other surfaces are too complex and require a recursive subdivision, similar to the lazy evaluation, but without retention of the data structure for later use.
Displacement Mapping
Displacement Mapping

• To add additional geometric detail to a tessellated surface, we can use *displacement mapping*

• We can either store a displacement map as an image or use a procedural displacement function (or some combination)

• Typically, a displacement map is just a height map, representing the displacement along the normal of the surface

• It could however, also include horizontal displacement in the s/t tangent directions

• As we subdivide the surface, we evaluate the curved surface and then displace the position based on the map and the normal (and possibly tangents)

• Once we’ve displaced our tessellated triangle mesh, we will need to recompute accurate normals, as they will change based on the displacements
Displacement Mapping
Displacement Mapping

• The midpoint distance rule we discussed earlier is far less effective with displacement mapping
• The displacement map might have some localized area of high displacement that would be missed if we just looked at the midpoint
• Therefore, with displacement mapping, one typically has to subdivide the entire surface down to sub-pixel sized triangles to be certain to capture all the detail that might be in the map
• Alternately, one could do a lot of pre-processing on the map itself to identify areas of high displacement ahead of time, and drive the subdivision based on that (a min/max mipmap works well for this)