Lecture 2:
The Reyes Rendering Algorithm

Image Synthesis
Stanford CS348b, Spring 2014
“Don’t make pictures that require apologies”
– Alvy Ray Smith
Reyes: Context and Goals

  - ~4000x slower than my laptop CPU
  - ~1000x less RAM than my laptop
    - Slightly less RAM than CPU’s L3 cache

- Goal: be able to render full-length feature films
  - Geometric complexity: millions of objects
  - Shading complexity: complex and varied surface models
  - Efficiency: if you render a movie on one computer in a year, you have 3 minutes per frame
  - Image quality: visually indistinguishable from live-action
Reyes 2009: Up, Pixar
Why Study Reyes?

- Reyes was a mainstay in film production for many years
  - Now, rapidly being displaced by physically-based ray tracing. (More on this later).
- Great example of system design that fulfilled a difficult set of requirements given various hard constraints
  - Many nice algorithms and good ways of thinking about high-quality rendering
- Explicitly designed for effective vector parallelism
- It may influence future real-time rendering pipelines
Key Ideas in Reyes

- Curved surfaces, not polygons, to describe the scene
- All geometric primitives eventually become micropolygons
  - Micropolygons are pixel-sized (can’t be seen individually)
- Shading calculations are performed at micropolygon vertices
  - Assume shading is expensive; shade a few times per pixel
- High quality rasterization of micropolygons to generate final image

- Key system design point: from the start, face up to the need to efficiently render 10s of millions of micropolygons to make a high-quality image
The Reyes Pipeline

Geometric primitives

- Split
- Bound
- Cull?
- Can Dice?
- Dice
- Shade
- Rasterize

Grids of micropolygons

Micropolygons
Reyes Primitive Interface

class Primitive {
public:
    virtual Bounds WorldBound() = 0;
    virtual bool NeedsSplit() = 0;
    virtual std::vector<Primitive *> Split();
    virtual Grid *Dice();
};

Must implement at least one of Split() or Dice().
Goals of Splitting

- Enables a wide variety of geometric primitives
  - A primitive either has to be able to dice itself, or split into something that (eventually) can
  - Parametric patches, subdivision surfaces, implicit surfaces, curves, ...
- Refines primitives into small sections
  - Enables fine-grained culling (frustum, occlusion, ...)
  - Minimizes over-dicing
Split or Dice?

- Must split if geometric primitive can’t dice itself
- Don’t want grids with too many micropolygons
  - Likely that not all will be visible
- Don’t want grids that have too few mps
  - Want enough to amortize per-grid overhead, vectorize well
- Don’t want too much variation in projected size of mps in grid on the screen
Splitting

Sphere

8 bicubic patches

Dice into micropolygons!

Hair Physics Simulation

1,000,000 hair strands

Groups of 1,000 strands

Bicubic curve for 1 strand

Dice into micropolygons!
Bicubic Bézier Patch Overview / Review

- Widely-used modeling primitive with Reyes
- Bicubic Bézier patch is defined by 16 control points $c_{i,j} \in \mathbb{R}^3$
- Parametric definition: $f(u,v) \rightarrow (x,y,z)$ for $u,v \in [0,1]^2$ is a 3rd degree polynomial in $u$ & $v$
Bicubic Béziers Patches in Reyes

- Bounding: the control points $c_{i,j}$ give a convex hull around the patch
  - An axis-aligned bounding box that encloses them therefore bounds the patch

- Computing dicing rates, splitting: following slides..

- Dicing: Efficient evaluation with bicubic basis functions or de Casteljau’s algorithm

\[
 c_{0-1} = (1-t) c_0 + t c_1 \\
 c_{01-12} = (1-t) c_{01} + t c_{1-2} \\
 c_{12-23} = (1-t) c_{12} + t c_{2-3} \\
 c_0 = (1-t) c_0 + t c_1 \\
 c_1 = (1-t) c_1 + t c_2 \\
 c_2 = (1-t) c_2 + t c_3 \\
 c_3 = (1-t) c_3 + t c_1
\]
Dicing Rates for Bicubic Bézier Patches

- Want # of micropolygons in the u and v directions $n_u$ and $n_v$, such that no mp edge is longer than a pixel's width.

- Strategy: find upper bounds on the length of screen-space derivatives of the surface in u and v: $d_{u\text{max}}$, $d_{v\text{max}}$

- Then for pixel-sized micropolygons, $n_u = \lceil d_{u\text{max}} \rceil$, $n_v = \lceil d_{v\text{max}} \rceil$
Dicing Bézier Patches

- We computed dicing rates \( n_u \) and \( n_v \) as part of the split or dice decision

- To dice a patch that spans \((u_0,v_0) \rightarrow (u_1,v_1)\)

```c
float3 P[MAXP], dPdu[MAXP], dPdv[MAXP];
int o = 0;
for (int iv = 0; iv < nv+1; ++iv) {
    float v = lerp(v0, v1, float(iv)/float(nv));
    for (int iu = 0; iu < nu+1; ++iu, ++o) {
        float u = lerp(u0, u1, float(iu)/float(nu));
        evalBezier(u, v, &P[o], &dPdu[o], &dPdv[o]);
    }
}
```

- If we use de Castilejau, the partial derivatives \( dP/du \) and \( dP/dv \) are more or less free
Bounding Bicubic Bézier Patch Derivatives

- First, project control points \( c_{i,j} \) to the image plane \( c'_{i,j} \).
- Now, consider iso-curves in each parametric direction.
- Key properties of bicubic Bézier surfaces:
  - The length of \( dP/du \) is never greater than it is along the four \( u \) iso-hulls.
  - The derivative along an iso-hull is bounded by \( 3x \) the maximum of the differences of adjacent control points.
Uniform Tessellation is Insufficient
Patch Splitting: Lane-Carpenter

- If dicing the patch would generate too many micropolygons, split it.
- Pick either the u or v direction, split in half parametrically.
- Can easily compute control points for the new patches based on the old ones and the split direction.
Adaptive Dicing From Patch Splitting

- **Uniform Dicing Rate**
- **Splitting, Then Dicing**

- **Over-diced**
- **Well-diced**
- **Under-diced**
One Problem: Patch Cracks
Crack-Free Uniform Tessellation
DiagSplit

Use non-isoparametric “splits” to generate sub-patches with matching edge tessellations.
Interesting Implications of Splitting

- Encapsulates adaptivity (keep dice simple, regular, and fast)

- Divide and conquer:
  - Micropolygon generation order exhibits high spatial locality
  - Provides temporal stability

- Splitting implicitly creates a hierarchy of grids
  - Very useful for frustum/depth culling at largest possible granularity
  - Use bounding box to cull primitives prior to dicing (or prior to unnecessary split)

- Splitting enables clipless rasterization (see Reyes paper)
Why grids?

- **Execution coherence**
  - All vertices on grid shaded with same shader
  - Permits SIMD implementation

- **Locality**
  - Grid is contiguous region of surface: shading points together increases texture locality

- **Compact representation**
  - For regular (tensor product) grid, topology is implicit
  - Quad micropolygon grid: each interior vertex shared by four micropolygons

- **Connectivity leveraged to compute derivatives in shaders**
  - Can compute higher order derivatives

- **Preserve hierarchy**
  - Allows per-grid operations, in addition to per micropolygon or per-vertex
  - Useful for culling, etc.
Displacement Mapping

- Input: set of connected grid points with associated geometric information, additional shading parameters
- Output: may change position of each grid point
- Allows addition of fine detail to surface geometry
  - How this is done generally expressed with procedural shaders (more on this shortly)
Displacement Mapping
(Gratuitous)
Displacement Mapping
(More Realistic)
Surface Shading Grids

- Input: grid after displacement mapping
- Output: color and opacity at each point

Rasterizer will break the grid into micropolygons and rasterize them to generate the image
Programmable Shading

- Approach largely born in Reyes: user writes short programs (“shaders”) that run at the heart of the renderer

- Two motivations:
  - Allows user substantial control over the visual result of rendering, without modifying the renderer
  - Data-parallel execution model allows high-performance vectorized execution

```c
surface metal(float Ka = 1; float Ks = 1;
             float roughness = 0.1;)
{
    normal Nf = faceforward(normalize(N), I);
    vector V = -normalize(I);
    Oi = Os;
    Ci = Os * Cs * (Ka * ambient() +
                   Ks * specular(Nf, V, roughness));
}
```

Simple RenderMan Shader
Texture Coherence

How much memory do we need to use this texture in a scene?

~600x200 texture map
120,000 texels
It Depends…

(Apologies to Alvy)
It Depends…
The Principle of Texture Thrift

- Given a prefiltred texture map, the number of unique texels needed to render is proportional to the number of pixels the texture maps covers on the screen.

- Total texels required proportional to:
  - Image resolution
  - # Textures / object
  - Depth complexity
Texture Caching

- Store image on-disk in tiles (e.g. 32x32)
- Load tiles on-demand
- Maintain fairly small in memory cache
  - MBs vs GBs
- Principle of texture thrift + Reyes grid coherence make this work
Evaluating Reyes..

- Geometric locality is great; linear processing of primitives
  - But, need the entire framebuffer in memory...
  - Solution: bucketing / tiling

- # of points shaded proportional to # pixels times depth complexity
  - Occlusion culling has turned out to be more important than originally expected

- Image resolutions have grown slowly/barely since 1980s
  - Memory, disk, compute have grown massively
  - IMHO this is what has made ray tracing a better fit today
Acknowledgements

- Slides 17 and 22–25 were provided by Kayvon Fatahalian.
- Images on slides 4, 5, 29, © Pixar