University of British Columbia
CPSC 314 Computer Graphics
Jan-Apr 2010

Tamara Munzner

Modern Hardware II, Curves

Week 12, Wed Apr 7

http://www.ugrad.cs.ubc.ca/~cs314/Vjan2010

News

- Extra TA office hours in lab 005 for P4/H4
  - Wed 4/7 2-4, 5-7 (Shailen)
  - Thu 4/8 3-5 (Kai)
  - Fri 4/9 11-12, 2-4 (Garrett)
  - Mon 4/12 11-1, 3-5 (Garrett)
  - Tue 4/13 3:30-5 (Kai)
  - Wed 4/14 2-4, 5-7 (Shailen)
  - Thu 4/15 3-5 (Kai)
  - Fri 4/16 11-4 (Garrett)

News

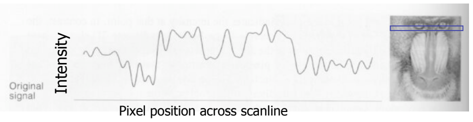
- please remember to fill out teaching evaluation surveys at CoursEval site
  https://eval.olt.ubc.ca/science

Review: Aliasing

- incorrect appearance of high frequencies as low frequencies
- to avoid: antialiasing
  - supersample
    - sample at higher frequency
  - low pass filtering
    - remove high frequency function parts
    - aka prefiltering, band-limiting

Review: Image As Signal

- 1D slice of raster image
  - discrete sampling of 1D spatial signal
- theorem
  - any signal can be represented as an (infinite) sum of sine waves at different frequencies



Examples from Foley, van Dam, Feiner, and Hughes

Review: Sampling Theorem and Nyquist Rate

- Shannon Sampling Theorem
  - continuous signal can be completely recovered from its samples iff sampling rate greater than twice maximum frequency present in signal
- sample past Nyquist Rate to avoid aliasing
  - twice the highest frequency component in the image's spectrum

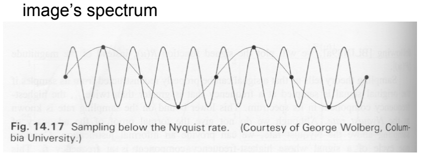
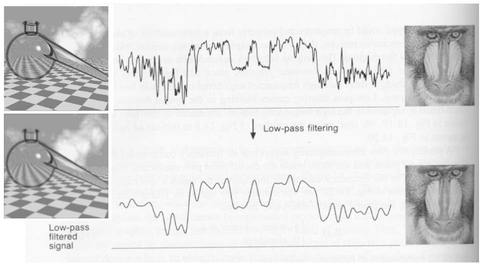


Fig. 14.17 Sampling below the Nyquist rate. (Courtesy of George Wolberg, Columbia University.)

Review: Low-Pass Filtering



Review: Rendering Pipeline

- so far rendering pipeline as a specific set of stages with **fixed functionality**
- modern graphics hardware more flexible
  - programmable "vertex shaders" replace several geometry processing stages
  - programmable "fragment/pixel shaders" replace texture mapping stage
  - hardware with these features now called Graphics Processing Unit (GPU)
- program shading hardware with assembly language analog, or high level shading language

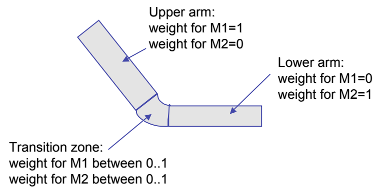
---

Review: Vertex Shaders

- replace model/view transformation, lighting, perspective projection
- a little assembly-style program is executed on every individual vertex independently
- it sees:
  - vertex attributes that change per vertex:
    - position, color, texture coordinates…
  - registers that are constant for all vertices (changes are expensive):
    - matrices, light position and color, …
  - temporary registers
  - output registers for position, color, tex coords…

Review: Skinning Vertex Shader

- arm example:
  - M1: matrix for upper arm
  - M2: matrix for lower arm



Review: Fragment Shaders

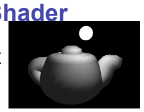
- fragment shaders operate on fragments in place of texturing hardware
  - after rasterization
  - before any fragment tests or blending
- input: fragment, with screen position, depth, color, and set of texture coordinates
- access to textures, some constant data, registers
- compute RGBA values for fragment, and depth
  - can also kill a fragment (throw it away)

Modern Hardware

- finish up nice slides by Gordon Wetzstein
  - lecture 23 from
  - http://www.ugrad.cs.ubc.ca/~cs314/Vjan2009/
    - slides, downloadable demos

Cg Example – Vertex Shader

- Vertex Shader: animated teapot



```
void main(  // input
            float4 position          : POSITION, // position in object coordinates
            float3 normal            : NORMAL,   // normal

            // user parameters
            uniform float4x4  objectMatrix,       // object coordinate system matrix
            uniform float4x4 objectMatrixIT,      // object coordinate system matrix inverse transpose
            uniform float4x4 modelViewMatrix,     // modelview matrix
            uniform float4x4 modelViewMatrixIT,   // modelview matrix inverse transpose
            uniform float4x4 projectionMatrix,    // projection matrix
            uniform float  deformation,           // deformation parameter
            uniform float3 lightPosition,         // light position
            uniform float3 lightAmbient,          // light ambient parameter
            uniform float3 lightDiffuse,          // light diffuse parameter
            uniform float3 lightSpecular,         // light specular parameter
            uniform float3 lightAttenuation,      // light attenuation parameter - constant, linear, quadratic
            uniform float3 materialEmission,      // material emission parameter
            uniform float3 materialAmbient,       // material ambient parameter
            uniform float3 materialDiffuse,       // material diffuse parameter
            uniform float3 materialSpecular,      // material specular parameter
            uniform float  materialShininess,     // material shininess parameter

            // output
            out float4 outPosition   : POSITION, // position in clip space
            out float4 outColor      : COLOR )   // out color
{
```

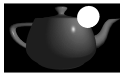
Cg Example – Vertex Shader

```
// transform position from object space to clip space
float4 positionObject = mul(objectMatrix, position);

// transform normal into world space
float4 normalObject = mul(objectMatrixIT, float4(normal,1));
float4 normalWorld  = mul(modelViewMatrixIT, normalObject);

// world position of light
float4 lightPositionWorld = \
            mul(modelViewMatrix, float4(lightPosition,1));

// assume viewer position is in origin
float4 viewerPositionWorld = float4(0.0, 0.0, 0.0, 1.0);

// apply deformation
positionObject.xyz    = positionObject.xyz + \
            deformation * normalize(normalObject.xyz);
float4 positionWorld = mul(modelViewMatrix, positionObject);
outPosition          = mul(projectionMatrix, positionWorld);

// two vectors
float3 P = positionWorld.xyz;
float3 N = normalize(normalWorld.xyz);

// compute the ambient term
float3 ambient = materialAmbient*lightAmbient;

// compute the diffuse term
float3 L = normalize(lightPositionWorld.xyz - P);
float   diffuseFactor = max(dot(N, L), 0);
float3 diffuse = materialDiffuse * lightDiffuse * diffuseFactor;
```

```
// compute the specular term
float3 V = normalize( viewerPositionWorld.xyz - \
                      positionWorld.xyz);
float3 H = normalize(L + V);
float specularFactor = \
        pow(max(dot(N, H), 0), materialShininess);
if (diffuseFactor <= 0) specularFactor = 0;
float3 specular = \
        materialSpecular * \
        lightSpecular * \
        specularFactor;

// attenuation factor
float distanceLightVertex = \
        length(P-lightPositionWorld.xyz);
float attenuationFactor = \
        1 / ( lightAttenuation.x + \
          distanceLightVertex*lightAttenuation.y + \
          distanceLightVertex*distanceLightVertex*\
          lightAttenuation.z );

// set output color
outColor.rgb =      materialEmission + \
                    ambient + \
                    attenuationFactor * \
                    ( diffuse + specular );
outColor.w = 1;
}
```

Cg Example – Phong Shading

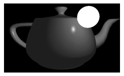
vertex shader



```
void main(  float4 position          : POSITION, // position in object coordinates
            float3 normal            : NORMAL,   // normal

            // user parameters
            ...

            // output
            out float4 outTexCoord0  : TEXCOORD0,  // world normal
            out float4 outTexCoord1  : TEXCOORD1,  // world position
            out float4 outTexCoord2  : TEXCOORD2,  // world light position
            out float4 outPosition   : POSITION)   // position in clip space
{
            // transform position from object space to clip space
            ...
            // transform normal into world space
            ...

            // set world normal as out texture coordinate0
            outTexCoord0 = normalWorld;
            // set world position as out texture coordinate1
            outTexCoord1 = positionWorld;
            // world position of light
            outTexCoord2 = mul(modelViewMatrix, float4(lightPosition,1));
}
```

Cg Example – Phong Shading

fragment shader

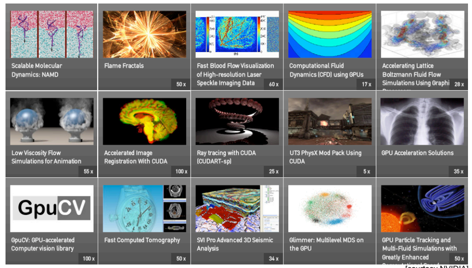


```
void main(  float4      normal        : TEXCOORD0,     // normal
            float4      position      : TEXCOORD1,     // position
            float4      lightPosition : TEXCOORD2,     // light position
            out float4  outColor      : COLOR )
{
            // compute the ambient term
            ...

            // compute the diffuse term
            ...

            // compute the specular term
            ...

            // attenuation factor
            ...

            // set output color
            outColor.rgb = materialEmission + ambient + attenuationFactor * (diffuse + specular);
}
```

GPGPU

- general purpose computation on the GPU
- in the past: access via shading languages and rendering pipeline
- now: access via cuda interface in C environment

GPGPU Applications



[courtesy NVIDIA]

Curves

Reading

- FCG Chap 15 Curves
  - Ch 13 2nd edition

Parametric Curves

- parametric form for a line:

$$x = x_0 t + (1-t)x_1$$
$$y = y_0 t + (1-t)y_1$$
$$z = z_0 t + (1-t)z_1$$

- x, y and z are each given by an equation that involves:
  - parameter t
  - some user specified control points, x_0 and x_1
- this is an example of a parametric curve

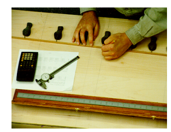
Splines

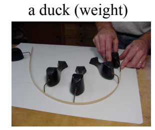
- a *spline* is a parametric curve defined by *control points*
  - term “spline” dates from engineering drawing, where a spline was a piece of flexible wood used to draw smooth curves
  - control points are *adjusted by the user* to control shape of curve

Splines - History

- draftsman used ‘ducks’ and strips of wood (splines) to draw curves
- wood splines have second-order continuity, pass through the control points

a duck (weight)





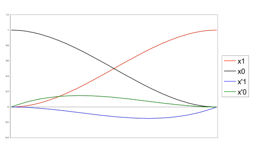
ducks trace out curve

Hermite Spline

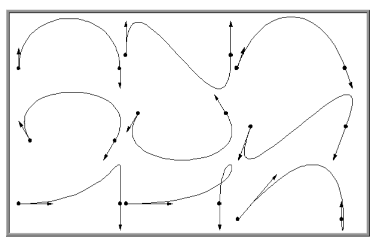
- *hermite spline* is curve for which user provides:
  - endpoints of curve
  - parametric derivatives of curve at endpoints
    - parametric derivatives are *dx/dt, dy/dt, dz/dt*
  - more derivatives would be required for higher order curves

Basis Functions

- a point on a Hermite curve is obtained by multiplying each control point by some function and summing
- functions are called *basis functions*

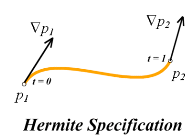


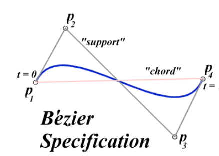
Sample Hermite Curves



Bézier Curves

- similar to Hermite, but more intuitive definition of endpoint derivatives
- four control points, two of which are knots





Bézier Curves

- derivative values of Bezier curve at knots dependent on adjacent points

$$\nabla p_1 = 3(p_2 - p_1)$$
$$\nabla p_4 = 3(p_4 - p_3)$$

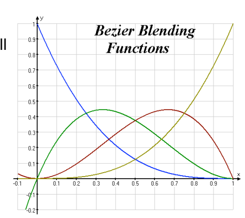
Bézier Blending Functions

- look at blending functions
- family of polynomials called order-3 Bernstein polynomials
  - C(3, k) t^k $(1-t)^{3-k}$; 0<= k <= 3
  - all positive in interval [0,1]
  - sum is equal to 1

$$p(t) = \begin{bmatrix} (1-t)^3 \\ 3t(1-t)^2 \\ 3t^2(1-t) \\ t^3 \end{bmatrix}^T \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{bmatrix}$$

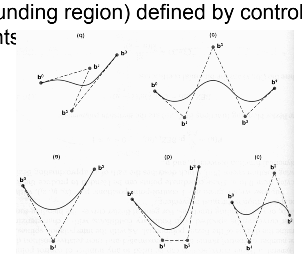
Bézier Blending Functions

- every point on curve is linear combination of control points
- weights of combination are all positive
- sum of weights is 1
- therefore, curve is a convex combination of the control points



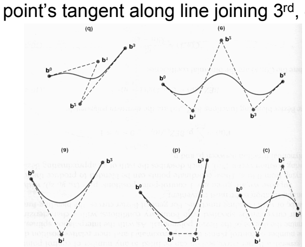
Bézier Curves

- curve will always remain within convex hull (bounding region) defined by control points



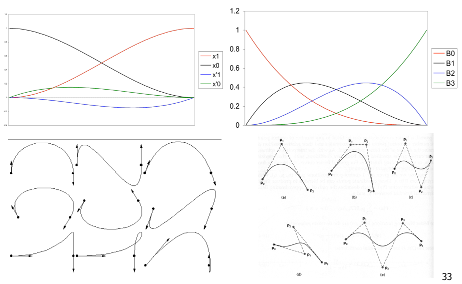
Bézier Curves

- interpolate between first, last control points
- 1st point’s tangent along line joining 1st, 2nd pts
- 4th point’s tangent along line joining 3rd, 4th pts



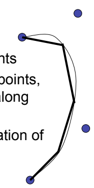
Comparing Hermite and Bézier

Hermite | Bézier



Rendering Bezier Curves: Simple

- evaluate curve at fixed set of parameter values, join points with straight lines
- advantage: very simple
- disadvantages:
  - expensive to evaluate the curve at many points
  - no easy way of knowing how fine to sample points, and maybe sampling rate must be different along curve
  - no easy way to adapt: hard to measure deviation of line segment from exact curve

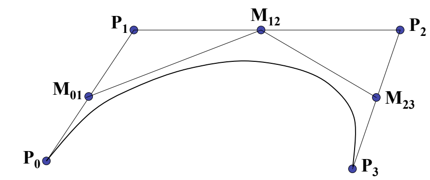


Rendering Beziers: Subdivision

- a cubic Bezier curve can be broken into two shorter cubic Bezier curves that exactly cover original curve
- suggests a rendering algorithm:
  - keep breaking curve into sub-curves
  - stop when control points of each sub-curve are nearly collinear
  - draw the control polygon: polygon formed by control points

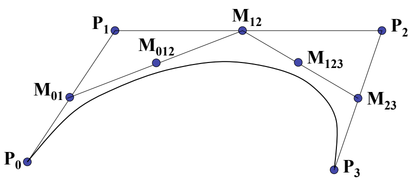
Sub-Dividing Bezier Curves

- step 1: find the midpoints of the lines joining the original control vertices. call them M_{01}, M_{12}, M_{23}



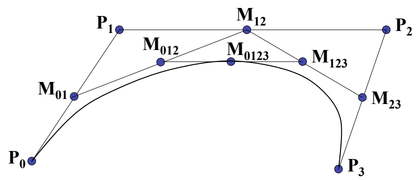
Sub-Dividing Bezier Curves

- step 2: find the midpoints of the lines joining M_{01}, M_{12} and M_{12}, M_{23}. call them M_{012}, M_{123}



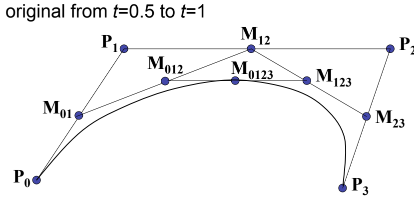
Sub-Dividing Bezier Curves

- step 3: find the midpoint of the line joining M_{012}, M_{123}. call it M_{0123}



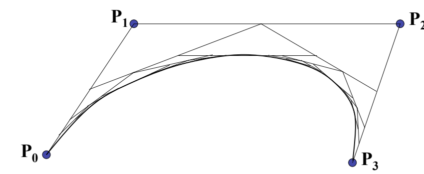
Sub-Dividing Bezier Curves

- curve P_0, M_{01}, M_{012}, M_{0123} exactly follows original from t=0 to t=0.5
- curve M_{0123} , M_{123} , M_{23}, P_3 exactly follows original from t=0.5 to t=1



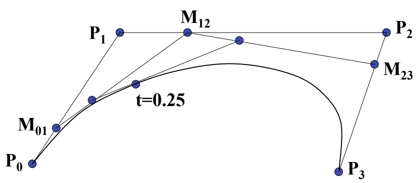
Sub-Dividing Bezier Curves

- continue process to create smooth curve



de Casteljau's Algorithm

- can find the point on a Bezier curve for any parameter value t with similar algorithm
  - for t=0.25, instead of taking midpoints take points 0.25 of the way

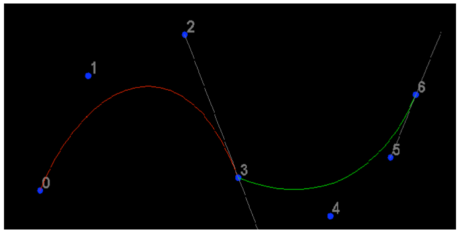


demo: www.saltire.com/applets/advanced_geometry/spline/spline.htm

Longer Curves

- a single cubic Bezier or Hermite curve can only capture a small class of curves
  - at most 2 inflection points
- one solution is to raise the degree
  - allows more control, at the expense of more control points and higher degree polynomials
  - control is not local, one control point influences entire curve
- better solution is to join pieces of cubic curve together into piecewise cubic curves
  - total curve can be broken into pieces, each of which is cubic
  - local control: each control point only influences a limited part of the curve
  - interaction and design is much easier

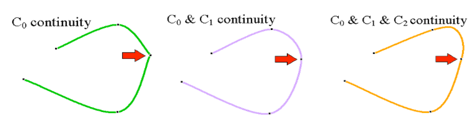
Piecewise Bezier: Continuity Problems



demo: www.cs.princeton.edu/~min/cs426/jar/bezier.html

Continuity

- when two curves joined, typically want some degree of continuity across knot boundary
  - C0, "C-zero", point-wise continuous, curves share same point where they join
  - C1, "C-one", continuous derivatives
  - C2, "C-two", continuous second derivatives



Geometric Continuity

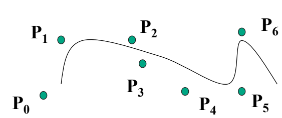
- derivative continuity is important for animation
  - if object moves along curve with constant parametric speed, should be no sudden jump at knots
- for other applications, *tangent continuity* suffices
  - requires that the tangents point in the same direction
  - referred to as G^1 *geometric continuity*
  - curves could be made C^1 with a re-parameterization
  - geometric version of C^2 is G^2, based on curves having the same radius of curvature across the knot

Achieving Continuity

- Hermite curves
  - user specifies derivatives, so C^1 by sharing points and derivatives across knot
- Bezier curves
  - they interpolate endpoints, so C^0 by sharing control pts
  - introduce additional constraints to get C^1
    - parametric derivative is a constant multiple of vector joining first/last 2 control points
    - so C^1 achieved by setting $P_{0,3}=P_{1,0}=J$, and making $P_{0,2}$ and J and $P_{1,1}$ collinear, with $J-P_{0,2}=P_{1,1}-J$
    - C^2 comes from further constraints on $P_{0,1}$ and $P_{1,2}$
  - leads to...

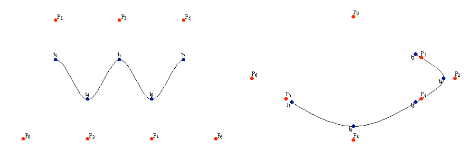
B-Spline Curve

- start with a sequence of control points
- select four from middle of sequence (p_{i-2}, p_{i-1}, p_i, p_{i+1})
  - Bezier and Hermite goes between p_{i-2} and p_{i+1}
  - B-Spline doesn't interpolate (touch) any of them but approximates the going through p_{i-1} and p_i



B-Spline

- by far the most popular spline used
- C_0, C_1, and C_2 continuous



demo: www.siggraph.org/education/materials/HyperGraph/modeling/splines/demoprog/curve.html

B-Spline

- locality of points

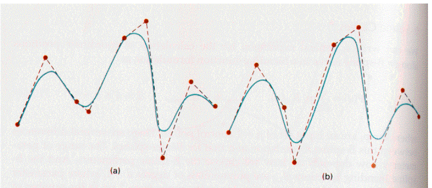


Figure 10-41
Local modification of a B-spline curve. Changing one of the control points in (a) produces curve (b), which is modified only in the neighborhood of the altered control point.