More GPU Stuff
and
Curves and Splines
Outline

• Announcements
• Project 2
• GLSL
• Hermite Splines
• Catmull-Rom Splines
• Bezier Curves
• Higher Continuity: Natural and B-Splines
• Drawing Splines
Announcements

• Don’t lock the door of the lab!

• Don’t paste solutions to the BBoards.

• **Final Exam:** Tues 5/12 from 8:30AM - 11:30AM, room TBA.
Outline

• Announcements

• Project 2

• GLSL

• Hermite Splines

• Catmull-Rom Splines

• Bezier Curves

• Higher Continuity: Natural and B-Splines

• Drawing Splines
Project 2

Get started early!!!
Outline

• Announcements
• Project 2

• GLSL
• Hermite Splines
• Catmull-Rom Splines
• Bezier Curves
• Higher Continuity: Natural and B-Splines
• Drawing Splines
OpenGL Pipeline
Simplified OpenGL Pipeline

Vertex Data

- Vertex Operation
- Rasterization (Interpolation)

Framebuffer
More Detail

Vertex Operation

Rasterization (Interpolation)

Fragment Operation

Framebuffer

Positions
Normals
Colors

Eye Positions
Eye Normals
Colors

Position
Normal
Color

(Interpolated)

Framebuffer
Programmability

Vertex Operation
- Positions
- Normals
- Colors

Rasterization (Interpolation)
- Eye Positions
- Eye Normals
- Colors
- Position
- Normal
- Color

Fragment Operation (Interpolated)

Framebuffer

Programmable

Programmable

Programmable

Framebuffer
Variable Types

• **const** - The declaration is of a compile time constant

• **attribute** - Global variables that may change per vertex, that are passed from the OpenGL application to vertex shaders. This qualifier can only be used in vertex shaders. For the shader this is a read-only variable.

• **varying** - used for interpolated data between a vertex shader and a fragment shader. Available for writing in the vertex shader, and read-only in a fragment shader.

• **uniform** - Global variables that may change per primitive (may not be set inside glBegin,/glEnd), that are passed from the OpenGL application to the shaders. This qualifier can be used in both vertex and fragment shaders. For the shaders this is a read-only variable.
Attribute Variables

- Positions
- Normals
- Colors

- Vertex Operation
  - Programmable
  - Eye Positions
  - Eye Normals
  - Colors

- Rasterization (Interpolation)
  - Position
  - Normal
  - Color
  - (Interpolated)

- Fragment Operation
  - Programmable

- Framebuffer
Varying Variables

- Positions
- Normals
- Colors
- Attributes

Vertex Operation

- Programmable
- Eye Positions
- Eye Normals
- Colors
- Varying

Rasterization (Interpolation)

- Position
- Normal
- Color
- Varying

Fragment Operation

- Programmable

Framebuffer
Uniform Variables

- Positions
- Normals
- Colors
- Attributes

- Eye Positions
- Eye Normals
- Colors
- Varying

- Rasterization (Interpolation)
- Position
- Normal
- Color
- Varying
  (Interpolated)

- Uniform

- Programmable

- Fragment Operation

- Programmable

- Framebuffer
Why So Fast?

- **Vertex Operation**
  - Programmable
  - ~4 Vertex Shaders

- **Fragment Operation**
  - Programmable
  - ~16 Fragment Shaders

- **Rasterization (Interpolation)**
  - Uniform
  - Position
  - Normal
  - Color
  - Varying
  - (Interpolated)

- **Attributes**
  - Positions
  - Normals
  - Colors
  - Varying

- **Framebuffer**
Types and Functions

OpenGL Shading Language (GLSL)
Quick Reference Guide

Describes GLSL version 1.10, as included in OpenGL v2.0, and specified by 'The OpenGL Shading Language', version 1.10.59. Section and page numbers refer to that version of the spec.

DATA TYPES (4.1 p16)
float, vec2, vec3, vec4
int,ivec2, ivec3, ivec4
bool, bvec2, bvec3, bvec4
mat2, mat3, mat4
void
sampler1D, sampler2D, sampler3D
sampler1DShadow, sampler2DShadow

DATA TYPE QUALIFIERS (4.3 p22)
global-variable declarations:
uniform Input to Vertex and Fragment shader from OpenGL or
application (READ-ONLY)
attribute Input per-vertex to Vertex shader from OpenGL or
application (READ-ONLY)
varying output from Vertex shader (READ/WRITE), interpolated, then
input to Fragment shader (READ-ONLY)
const compile-time constant (READ-ONLY)

function parameters:
in value initialized on entry, not copied on return (default)
out copied out on return, but not initialized
inout value initialized on entry, and copied out on return

VECTOR COMPONENTS (5.5 p 30)
component names may not be mixed across sets
x, y, z, w
r, g, b, a
t, l, p, q

PREPROCESSOR (3.3 p9)
t
define _LINE_
define _FILE_
define __VERSION_

GLSL version declaration and extensions protocol:
<version>
default is "version 110" (3.3 p11)
dextension [name | all] [require | enable | warn | disable]
default is "dextension all : disable" (3.3.11)

BUILT-IN FUNCTIONS

Key:
vec - vec2 | vec3 | vec4
mat - mat2 | mat3 | mat4
ivec - ivec2 | ivec3 | ivec4
bvec - bvec2 | bvec3 | bvec4
gtype = float | vec2 | vec3 | vec4

Angle and Trigonometry Functions (8.1 p51)
gtype sin(gtype)
gtype cos(gtype)
gtype tan(gtype)
gtype asin(gtype)
gtype acos(gtype)
gtype atan(gtype)
gtype atani(gtype, gtype)
gtype atan2(gtype, gtype)
gtype radans(gtype)
gtype degrees(gtype)

Exponential Functions (8.2 p52)
gtype exp(gtype)
gtype exp10(gtype)
gtype log(gtype)
gtype log10(gtype)
gtype log2(gtype)
gtype log102(gtype)
gtype inverse2(gtype)

Common Functions (8.3 p52)
gtype abs(gtype)
gtype ceil(gtype)
gtype clamp(gtype, gtype, gtype, gtype)
gtype clampf(gtype, gtype, float, float)
gtype floor(gtype)
gtype floorf(gtype)
gtype fract(gtype)
gtype max(gtype, gtype, gtype)
gtype maxf(gtype, gtype, float)
gtype min(gtype, gtype, gtype)
gtype minf(gtype, gtype, float)
gtype mod(gtype, gtype, gtype)
gtype modf(gtype, gtype, float)
gtype smoothstep(gtype, gtype, gtype, gtype)
gtype smoothstepf(float, float, gtype)
gtype step(gtype, gtype)

Geometric Functions (8.4 p54)
vec4 transpose(vec4, Vertex ONLY)
vec3 cross(vec3, vec3)
float distance(gtype, gtype)
float dot(gtype, gtype)
float faceforward(gtype, gtype, gtype, float)
float length(gtype)
float normalize(gtype)
gtype reflect(gtype, gtype, gtype)
gtype refract(gtype, gtype, float, gtype)

Fragmnet Processing Functions (8.8 p58)
float fdivision(gtype)
float fwidth(gtype)

Matrix Functions (8.5 p55)
mat matrixCompunexpected(mat, mat, mat)

Vector Relational Functions (8.6 p55)
bool all(bvec)
bool any(bvec)
bvec equal(bvec, vec)
bvec equalf(hvec, hvec)
bvec greaterThan(bvec, vec)
bvec greaterThanEqual(bvec, vec)
bvec lessThan(bvec, vec)
bvec lessThanEqual(bvec, vec)
bvec not(bvec)
bvec notEqual(bvec, vec)
bvec notEqualf(bvec, hvec)

Texture Lookup Functions (8.7 p56)
Texture Lookup Functions with LOD (8.7 p56)
Vertex ONLY: ensure GL_MAX_VERTEX_TEXTURE_IMAGE_UNITS > 0
texture1DLOD(sampler1D, vec3, float, float)
texture1DLOD(sampler1D, vec2, float, lod)
texture1DLOD(sampler1D, vec4, float, lod)
texture2DLOD(sampler2D, vec2, vec3, float)
texture2DLOD(sampler2D, vec4, float, lod)
texture3DLOD(sampler3D, vec3, float, lod)
texture3DLOD(sampler3D, vec4, float, lod)
textureCubeLOD(samplerCube, float, lod)
textureCubeLOD(samplerCube, vec3, float, lod)
textureCubeLOD(samplerCube, vec4, float, lod)
textureCubeLOD(samplerCube, vec3, float)
textureCubeLOD(samplerCube, vec4, float)
textureCubeLOD(samplerCube, vec3, float)
textureCubeLOD(samplerCube, vec4, float)
textureCubeLOD(samplerCube, vec3, float)
textureCubeLOD(samplerCube, vec4, float)
textureCubeLOD(samplerCube, vec3, float)
textureCubeLOD(samplerCube, vec4, float)

Noise Functions (8.9 p60)
float noise1D(gtype)
float noise2D(gtype)
float noise3D(gtype)
float noise4D(gtype)
Type Properties

• Matrix / vector / integer / floating point types.

• *Strict* casting requirements.

• Const / attribute / varying / uniform variables.
Control Loops in GLSL

```glsl
if (condition) {
    statements;
} else {
    statements;
}
```

```
for (int i=0 ; i<10 ; i++) {
    statements;
}
```

```
while (condition) {
    statements;
}
```

_Beware:_ If Statement Implementation
### Builtins

**VERTEX SHADER VARIABLES**

<table>
<thead>
<tr>
<th>Special Output Variables (7.1.42)</th>
<th>access=RW</th>
<th>vec4 gl_Position; enable GL_VERTEX_PROGRAM_POINT_SIZE</th>
</tr>
</thead>
<tbody>
<tr>
<td>vec3 gl_ClipVertex;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Attribute Inputs (7.3.44) | access=RO | attribute vec3 gl_Vertex; attribute vec3 gl_Normal; attribute vec4 gl_Color; attribute vec4 gl_SecondaryColor; attribute vec2 gl_MultiTexCoord0; attribute vec2 gl_MultiTexCoord1; attribute vec2 gl_MultiTexCoord2; attribute vec2 gl_MultiTexCoord3; attribute vec2 gl_MultiTexCoord4; attribute vec2 gl_MultiTexCoord5; attribute vec2 gl_MultiTexCoord6; attribute vec2 gl_MultiTexCoord7; attribute float gl_FogCoord; |

<table>
<thead>
<tr>
<th>Varying Outputs (7.4.48)</th>
<th>access=RW</th>
<th>varying vec4 gl_FragColor; varynig vec4 gl_FrontColor; varying vec4 gl_BackColor; enable GL_VERTEX_PROGRAM_TWO_SIDE</th>
</tr>
</thead>
<tbody>
<tr>
<td>varying vec4 gl_FragCoord;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>varying float gl_FogCoord;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**FRAGMENT SHADER VARIABLES**

<table>
<thead>
<tr>
<th>Special Output Variables (7.2.34)</th>
<th>access=RW</th>
<th>vec4 gl_FragColor; vec4 gl_FragData[gl_MaxDrawBuffers]; float gl_FragDepth;</th>
</tr>
</thead>
<tbody>
<tr>
<td>float gl_FragCoord;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Varying Inputs (7.6.48)</th>
<th>access=RO</th>
<th>varying vec4 gl_FragColor; varynig vec4 gl_SecondaryColor; varying vec4 gl_TexCoord[ ]; MAP(gl_MaxTextureCoords)</th>
</tr>
</thead>
<tbody>
<tr>
<td>varying float gl_FogFragCoord;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Special Input Variables (7.2.43) | access=RO | bool gl_FrontFacing; pixel coordinates |

**BUILT-IN UNIFORMS (7.5.45) access=RO**

- \texttt{uniform mat4 gl_ModelViewMatrix;}
- \texttt{uniform mat4 gl_ModelViewProjectionMatrix;}
- \texttt{uniform mat4 gl_ProjectionMatrix;}
- \texttt{uniform mat4 gl_TextureMatrix[gl_MaxTextureCoords];}
- \texttt{uniform mat4 gl_TextureMatrixInverse[gl_MaxTextureCoords];}
- \texttt{uniform mat4 gl_TextureMatrixTranspose[gl_MaxTextureCoords];}
- \texttt{uniform mat4 gl_TextureMatrixInverseTranspose[gl_MaxTextureCoords];}

**BUILT-IN CONSTANTS (7.4.44)**

- \texttt{const int gl_MaxVerticesUniformComponents;}
- \texttt{const int gl_MaxFragmentUniformComponents;}
- \texttt{const int gl_MaxVertexAttributes;}
- \texttt{const int gl_MaxVaryingFloats;}
- \texttt{const int gl_MaxFloatBuffers;}
- \texttt{const int gl_MaxTextureUnits;}
- \texttt{const int gl_MaxTextureImageUnits;}
- \texttt{const int gl_MaxCombinedTextureImageUnits;}
- \texttt{const int gl_MaxLights;}
- \texttt{const int gl_MaxClipPlanes;}

细读范围

**OpenSceneGraph Preset Uniforms**

as of OSG 1.0

| int |  |  |  |
|-----|  |  |  |
| \texttt{osg::FrameNumber} | \texttt{float} | \texttt{osg::DeltaFrameTime} | \texttt{mat4 osg::ViewMatrix;}
| \texttt{mat4 osg::ViewMatrixInverse;}

Note: 1. Corrects a typo in the OpenGL 2.0 specification.
Happy GPU Coding!
Outline

- Announcements
- Project 2
- GLSL
  - Hermite Splines
  - Catmull-Rom Splines
  - Bezier Curves
  - Higher Continuity: Natural and B-Splines
- Drawing Splines
Modeling Complex Shapes

• We want to build models of very complicated objects
• An equation for a sphere is possible, but how about an equation for a telephone, or a face?
• Complexity is achieved using simple pieces
  – polygons, parametric curves and surfaces, or implicit curves and surfaces
  – This lecture: parametric curves
What Do We Need From Curves in Computer Graphics?

• Local control of shape (so that easy to build and modify)
• Stability
• Smoothness and continuity
• Ability to evaluate derivatives
• Ease of rendering

Demo
Curve Usage Demo
Curve Representations

- **Explicit:** \( y = f(x) \)
  
  \( y = mx + b \)
  
  - Easy to generate points
  - Must be a function: big limitation—vertical lines?
Curve Representations

• Explicit: \( y = f(x) \)
  \[
  y = mx + b
  \]
  – Easy to generate points
  – Must be a function: big limitation—vertical lines?

• Implicit: \( f(x,y) = 0 \)
  \[
  x^2 + y^2 - r^2 = 0
  \]
  +Easy to test if on the curve
  –Hard to generate points
Curve Representations

• Explicit: \( y = f(x) \)
  
  \[ y = mx + b \]
  
  + Easy to generate points
  - Must be a function: big limitation—vertical lines?

• Implicit: \( f(x,y) = 0 \)
  
  \[ x^2 + y^2 - r^2 = 0 \]
  
  + Easy to test if on the curve
  - Hard to generate points

• Parametric: \((x,y) = (f(u), g(u))\)
  
  \( (x, y) = (\cos u, \sin u) \)
  
  + Easy to generate points
Parameterization of a Curve

- *Parameterization* of a curve: how a change in $u$ moves you along a given curve in $xyz$ space.
Polynomial Interpolation

- An \( n \)-th degree polynomial fits a curve to \( n+1 \) points
  - called Lagrange Interpolation
  - result is a curve that is too wiggly, change to any control point affects entire curve (nonlocal) – \textit{this method is poor}

- We usually want the curve to be as smooth as possible
  - minimize the wiggles
  - high-degree polynomials are bad
Linear Interpolation
Spline Interpolation
Spine Interpolation Demo
Splines: Piecewise Polynomials

• A spline is a *piecewise polynomial* - many low degree polynomials are used to interpolate (pass through) the control points.

• *Cubic piecewise* polynomials are the most common:
  – piecewise definition gives local control
Piecwise Polynomials

- Spline: lots of little polynomials pieced together
- Want to make sure they fit together nicely

\[C_0 \text{ continuity} \quad \text{C}_0 \& C_1 \text{ continuity} \quad \text{C}_0 \& C_1 \& C_2 \text{ continuity}\]

- Continuous in position
- Continuous in position and tangent vector
- Continuous in position, tangent, and curvature
Splines

• Types of splines:
  – Hermite Splines
  – Catmull-Rom Splines
  – Bezier Splines
  – Natural Cubic Splines
  – B-Splines
  – NURBS
Hermite Curves

- Cubic Hermite Splines

That is, we want a way to specify the end points and the slope at the end points!
Splines

chalkboard
The Cubic Hermite Spline Equation

- Using some algebra, we obtain:

\[ p(u) = \begin{bmatrix} u^3 & u^2 & u \end{bmatrix} \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ \nabla p_1 \\ \nabla p_2 \end{bmatrix} \]

- This form typical for splines
  - basis matrix and meaning of control matrix change with the spline type
The Cubic Hermite Spline Equation

- Using some algebra, we obtain:

\[ p(u) = \begin{bmatrix} u^3 & u^2 & u \end{bmatrix} \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ \nabla p_1 \\ \nabla p_2 \end{bmatrix} \]

point that gets drawn

basis

control matrix
(what the user gets to pick)

\[ p(u) = \begin{bmatrix} 2u^3 - 3u^2 + 1 \\ -2u^3 + 3u^2 \\ u^3 - 2u^2 + u \\ u^3 - u^2 \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ \nabla p_1 \\ \nabla p_2 \end{bmatrix} \]

4 Basis Functions
Four Basis Functions for Hermite splines

\[ p(u) = \begin{bmatrix} 2u^3 - 3u^2 + 1 \\ -2u^3 + 3u^2 \\ u^3 - 2u^2 + u \\ u^3 - u^2 \end{bmatrix}^T \begin{bmatrix} p_1 \\ p_2 \\ \nabla p_1 \\ \nabla p_2 \end{bmatrix} \]

4 Basis Functions

Every cubic Hermite spline is a linear combination (blend) of these 4 functions
Piecing together Hermite Curves

- It's easy to make a multi-segment Hermite spline
  - each piece is specified by a cubic Hermite curve
  - just specify the position and tangent at each “joint”
  - the pieces fit together with matched positions and first derivatives
  - gives C1 continuity

- The points that the curve has to pass through are called *knots* or *knot points*
Outline

• Announcements
• Project 2
• GLSL
• Hermite Splines
  • Catmull-Rom Splines
• Bezier Curves
• Higher Continuity: Natural and B-Splines
• Drawing Splines
Problem with Hermite Splines?

- Must explicitly specify derivatives at each endpoint!
- How can we solve this?
Catmull-Rom Splines

• With Hermite splines, the designer must specify all the tangent vectors
• Catmull-Rom: an interpolating cubic spline with built-in $C^1$ continuity.

\[
\text{tangent at } p_i = s(p_{i+1} - p_{i-1})
\]
Catmull-Rom Splines

- With Hermite splines, the designer must arrange for consecutive tangents to be collinear, to get $C^1$ continuity. This gets tedious.
- Catmull-Rom: an interpolating cubic spline with built-in $C^1$ continuity.
Catmull-Rom Spline Matrix

\[ p(u) = \begin{bmatrix} u^3 & u^2 & u \end{bmatrix} \begin{bmatrix} -s & 2-s & s-2 & s \\ 2s & s-3 & 3-2s & -s \\ -s & 0 & s & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{bmatrix} \]

spline coefficients  CR basis  control vector

- Derived similarly to Hermite
- Parameter \( s \) is typically set to \( s=1/2 \).
Catmull-Rom Spline Matrix

\[
\begin{bmatrix}
  x \\
  y \\
  z
\end{bmatrix} =
\begin{bmatrix}
  u^3 & u^2 & u & 1
\end{bmatrix}
\begin{bmatrix}
  -s & 2 - s & s - 2 & s \\
  2s & s - 3 & 3 - 2s & -s \\
  -s & 0 & s & 0 \\
  0 & 1 & 0 & 0
\end{bmatrix}
\begin{bmatrix}
  x_1 & y_1 & z_1 \\
  x_2 & y_2 & z_2 \\
  x_3 & y_3 & z_3 \\
  x_4 & y_4 & z_4
\end{bmatrix}
\]

spline coefficients  CR basis  control vector
Catmull-Rom Splines

- With Hermite splines, the designer must specify all the tangent vectors
- Catmull-Rom: an interpolating cubic spline with \textit{built-in} $C^1$ continuity.
Catmull-Rom Spline Matrix

\[ p(u) = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \begin{bmatrix} -s & 2-s & s-2 & s \\ 2s & s-3 & 3-2s & -s \\ -s & 0 & s & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{bmatrix} \]

- Derived similarly to Hermite
- Parameter \( s \) is typically set to \( s=1/2 \).
Cubic Curves in 3D

- Three cubic polynomials, one for each coordinate
  - \( x(u) = a_x u^3 + b_x u^2 + c_x u + d_x \)
  - \( y(u) = a_y u^3 + b_y u^2 + c_y u + d_y \)
  - \( z(u) = a_z u^3 + b_z u^2 + c_z u + d_z \)

- In matrix notation

\[
\begin{bmatrix}
  x(u) \\
y(u) \\
z(u)
\end{bmatrix} =
\begin{bmatrix}
u^3 & u^2 & u & 1
\end{bmatrix}
\begin{bmatrix}
a_x & a_y & a_z \\
b_x & b_y & b_z \\
c_x & c_y & c_z \\
d_x & d_y & d_z
\end{bmatrix}
\]
Catmull-Rom Spline Matrix in 3D

\[
\begin{bmatrix}
  x(u) & y(u) & z(u)
\end{bmatrix} =
\begin{bmatrix}
  u^3 & u^2 & u & 1
\end{bmatrix}
\begin{bmatrix}
  -s & 2 - s & s - 2 & s \\
  2s & s - 3 & 3 - 2s & -s \\
  -s & 0 & s & 0 \\
  0 & 1 & 0 & 0
\end{bmatrix}
\begin{bmatrix}
  x_1 & y_1 & z_1 \\
  x_2 & y_2 & z_2 \\
  x_3 & y_3 & z_3 \\
  x_4 & y_4 & z_4
\end{bmatrix}
\]

CR basis control vector
Outline

• Announcements
• Project 2
• GLSL
• Hermite Splines
• Catmull-Rom Splines
• Bezier Curves
• Higher Continuity: Natural and B-Splines
• Drawing Splines
Problem with Catmull-Rom Splines?

- *No* control of derivatives at endpoints!
- How can we solve this?
  - We want something intuitive.
Beziers Curves*

- Another variant of the same game
- Instead of endpoints and tangents, four control points
  - points P0 and P3 are on the curve: P(u=0) = P0, P(u=1) = P3
  - points P1 and P2 are off the curve
  - P'(u=0) = 3(P1-P0), P'(u=1) = 3(P3 - P2)
- Convex Hull property
  - curve contained within convex hull of control points
- Gives more control knobs (series of points) than Hermite
- Scale factor (3) is chosen to make “velocity” approximately constant
Bezier Spline Example
The Bezier Spline Matrix*

\[
\begin{bmatrix}
x & y & z \\
u^3 & u^2 & u & 1
\end{bmatrix}
\begin{bmatrix}
-1 & 3 & -3 & 1 \\
3 & -6 & 3 & 0 \\
-3 & 3 & 0 & 0 \\
1 & 0 & 0 & 0
\end{bmatrix}
\begin{bmatrix}
x_1 & y_1 & z_1 \\
x_2 & y_2 & z_2 \\
x_3 & y_3 & z_3 \\
x_4 & y_4 & z_4
\end{bmatrix}
\]

Bezier basis

Bezier control vector
Bezifier Blending Functions

Also known as the order 4, degree 3 Bernstein polynomials
Nonnegative, sum to 1
The entire curve lies inside the polyhedron bounded by the control points
Outline

- Announcements
- Project 2
- GLSL
- Hermite Splines
- Catmull-Rom Splines
- Bezier Curves
- Higher Continuity: Natural and B-Splines
- Drawing Splines
Piecewise Polynomials

- Spline: lots of little polynomials pieced together
- Want to make sure they fit together nicely

$C_0$ continuity

Continuous in position

$C_0$ & $C_1$ continuity

Continuous in position and tangent vector

$C_0$ & $C_1$ & $C_2$ continuity

Continuous in position, tangent, and curvature
Splines with More Continuity?

• How could we get $C^2$ continuity at control points?

• Possible answers:
  – Use higher degree polynomials
    degree 4 = quartic, degree 5 = quintic, … but these get computationally expensive, and sometimes wiggly
  – Give up local control natural cubic splines
    A change to any control point affects the entire curve
  – Give up interpolation cubic B-splines
    Curve goes near, but not through, the control points
## Comparison of Basic Cubic Splines

<table>
<thead>
<tr>
<th>Type</th>
<th>Local Control</th>
<th>Continuity</th>
<th>Interpolation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hermite</td>
<td>YES</td>
<td>C1</td>
<td>YES</td>
</tr>
<tr>
<td>Bezier</td>
<td>YES</td>
<td>C1</td>
<td>YES</td>
</tr>
<tr>
<td>Catmull-Rom</td>
<td>YES</td>
<td>C1</td>
<td>YES</td>
</tr>
<tr>
<td>Natural</td>
<td>NO</td>
<td>C2</td>
<td>YES</td>
</tr>
<tr>
<td>B-Splines</td>
<td>YES</td>
<td>C2</td>
<td>NO</td>
</tr>
</tbody>
</table>

• **Summary**
  – Can’t get C2, interpolation and local control with cubics
Natural Cubic Splines*

• If you want 2nd derivatives at joints to match up, the resulting curves are called natural cubic splines.

• It’s a simple computation to solve for the cubics' coefficients. (See Numerical Recipes in C book for code.)

• Finding all the right weights is a global calculation (solve tridiagonal linear system)
B-Splines*

- Give up interpolation
  - the curve passes *near* the control points
  - best generated with interactive placement (because it’s hard to guess where the curve will go)
- Curve obeys the convex hull property
- C2 continuity and local control are good compensation for loss of interpolation
B-Spline Basis*

- We always need 3 more control points than spline pieces

\[ M_{BS} = \frac{1}{6} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix} \]

\[ G_{BSi} = \begin{bmatrix} P_{i-3} \\ P_{i-2} \\ P_{i-1} \\ P_i \end{bmatrix} \]
Outline

- Announcements
- Project 2
- GLSL
- Hermite Splines
- Catmull-Rom Splines
- Bezier Curves
- Higher Continuity: Natural and B-Splines
- Drawing Splines
How to Draw Spline Curves

- Basis matrix eqn. allows same code to draw any spline type
- **Method 1: brute force**
  - Calculate the coefficients
  - For each cubic segment, vary $u$ from 0 to 1 (fixed step size)
  - Plug in $u$ value, matrix multiply to compute position on curve
  - Draw line segment from last position to current position

\[
\begin{bmatrix}
  x \\
  y \\
  z 
\end{bmatrix} =
\begin{bmatrix}
  u^3 & u^2 & u & 1 \\
\end{bmatrix}
\begin{bmatrix}
  -s & 2-s & s-2 & s \\
  2s & s-3 & 3-2s & -s \\
  -s & 0 & s & 0 \\
  0 & 1 & 0 & 0 \\
\end{bmatrix}
\begin{bmatrix}
  x_1 & y_1 & z_1 \\
  x_2 & y_2 & z_2 \\
  x_3 & y_3 & z_3 \\
  x_4 & y_4 & z_4 \\
\end{bmatrix}
\]

CR basis
control vector
How to Draw Spline Curves

• What’s wrong with this approach?
  – Draws in even steps of u
  – Even steps of u ≠ even steps of x
  – Line length will vary over the curve
  – Want to bound line length
    » too long: curve looks jagged
    » too short: curve is slow to draw
• **Method 2: recursive subdivision** - vary step size to draw short lines

\[
\text{Subdivide}(u_0, u_1, \text{maxlinelength})
\]
\[
\begin{align*}
\text{umid} &= (u_0 + u_1)/2 \\
x_0 &= P(u_0) \\
x_1 &= P(u_1) \\
\text{if } |x_1 - x_0| > \text{maxlinelength} \\
\quad &\text{Subdivide}(u_0, \text{umid}, \text{maxlinelength}) \\
\quad &\text{Subdivide}(\text{umid}, u_1, \text{maxlinelength}) \\
\text{else } &\text{drawline}(x_0, x_1)
\end{align*}
\]

• **Variant on Method 2** - subdivide based on curvature
  
  – replace condition in “if” statement with straightness criterion
  
  – draws fewer lines in flatter regions of the curve
In Summary...

• Summary:
  – piecewise cubic is generally sufficient
  – define conditions on the curves and their continuity

• Things to know:
  – basic curve properties (what are the conditions, controls, and properties for each spline type)
  – generic matrix formula for uniform cubic splines $x(u) = uB\!G$
  – given definition derive a basis matrix